FireSim Documentation

Release t

Sagar Karandikar, David Biancolin, Abraham Gonzalez, Howard Mao, Donggyu Kim, Alon Amid, Berkeley Architecture Research

Sep 23, 2024

GETTING STARTED:

1 FireSim Basics **3**
1.1 Common FireSim usage models . . . 3
1.1.1 1. Single-Node Simulations Using One or More On-Premises FPGAs . . . 3
1.1.2 2. Single-Node Simulations Using Cloud FPGAs . . . 4
1.1.3 3. Datacenter/Cluster Simulations on On-Premises or Cloud FPGAs . . . 4
1.2 Other Use Cases . . . 4
1.2.1 Non-Chipyard-based target simulation . . . 4
1.3 Choose your platform to get started . . . 4

2 Background/Terminology **7**

3 AWS EC2 F1 System Setup **9**
3.1 First-time AWS User Setup . . . 9
3.1.1 Creating an AWS Account . . . 9
3.1.2 Requesting Limit Increases . . . 9
3.2 Configuring Required Infrastructure in Your AWS Account . . . 10
3.2.1 Select a region . . . 10
3.2.2 Key Setup . . . 10
3.2.3 Double Check your EC2 Instance Limits . . . 10
3.2.4 Start a t2.nano instance to run the remaining configuration commands . . . 11
3.2.5 Run scripts from the t2.nano . . . 11
3.2.6 Terminate the t2.nano . . . 12
3.2.7 Subscribe to the AWS FPGA Developer AMI . . . 12
3.3 Setting up your Manager Instance . . . 12
3.3.1 Launching a “Manager Instance” . . . 12

4 Local FPGA System Setup **23**
4.1 `sudo` Setup . . . 23
4.2 Non-`sudo` Setup . . . 26

5 AWS EC2 F1 Getting Started Guide **29**
5.1 Setting up the FireSim Repo . . . 29
5.2 Completing Setup Using the Manager . . . 30
5.3 Running FireSim Simulations . . . 30
5.3.1 Running a Single Node Simulation . . . 31
5.3.2 Running a Cluster Simulation . . . 40
5.4 Building Your Own Hardware Designs (FireSim Amazon FPGA Images) . . . 51
5.4.1 Amazon S3 Setup . . . 51
5.4.2 Build Recipes . . . 52
5.4.3 Build Farm Instance Types . . . 52

5.4.4 Running a Build . . . 52

6 Xilinx Alveo U200 XDMA-based Getting Started Guide **55**
6.1 FPGA Setup . . . 55
6.2 FireSim Repo Setup . . . 56
6.2.1 Setting up the FireSim Repo . . . 56
6.2.2 Initializing FireSim Config Files . . . 57
6.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup . . . 57
6.3 Running a Single Node Simulation . . . 58
6.3.1 Building target software . . . 58
6.4 Setting up the manager configuration . . . 59
6.5 Building and Deploying simulation infrastructure to the Run Farm Machines . . . 59
6.6 Running the simulation . . . 60
6.7 Building Your Own Hardware Designs . . . 63
6.8 System Setup . . . 63
6.8.1 1. Install Vivado for Builds . . . 64
6.8.2 2. Verify Build Farm Machine environment . . . 64
6.9 Configuring a Build in the Manager . . . 64
6.10 Running the Build . . . 65

7 Xilinx Alveo U250 XDMA-based Getting Started Guide **67**
7.1 FPGA Setup . . . 67
7.2 FireSim Repo Setup . . . 68
7.2.1 Setting up the FireSim Repo . . . 68
7.2.2 Initializing FireSim Config Files . . . 69
7.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup . . . 69
7.3 Running a Single Node Simulation . . . 70
7.3.1 Building target software . . . 70
7.4 Setting up the manager configuration . . . 71
7.5 Building and Deploying simulation infrastructure to the Run Farm Machines . . . 71
7.6 Running the simulation . . . 72
7.7 Building Your Own Hardware Designs . . . 75
7.8 System Setup . . . 75
7.8.1 1. Install Vivado for Builds . . . 76
7.8.2 2. Verify Build Farm Machine environment . . . 76
7.9 Configuring a Build in the Manager . . . 76
7.10 Running the Build . . . 77

8 Xilinx Alveo U280 XDMA-based Getting Started Guide **79**
8.1 FPGA Setup . . . 79
8.2 FireSim Repo Setup . . . 80
8.2.1 Setting up the FireSim Repo . . . 80
8.2.2 Initializing FireSim Config Files . . . 81
8.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup . . . 81
8.3 Running a Single Node Simulation . . . 82
8.3.1 Building target software . . . 82
8.4 Setting up the manager configuration . . . 83
8.5 Building and Deploying simulation infrastructure to the Run Farm Machines . . . 83
8.6 Running the simulation . . . 84
8.7 Building Your Own Hardware Designs . . . 87
8.8 System Setup . . . 87
8.8.1 1. Install Vivado for Builds . . . 88
8.8.2 2. Verify Build Farm Machine environment . . . 88
8.9 Configuring a Build in the Manager . . . 88

8.10 Running the Build 89

9 **Xilinx VCU118 XDMA-based Getting Started Guide** **91**
9.1 FPGA Setup 91
9.2 FireSim Repo Setup 92
9.2.1 Setting up the FireSim Repo 92
9.2.2 Initializing FireSim Config Files 93
9.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup 93
9.3 Running a Single Node Simulation 94
9.3.1 Building target software 94
9.4 Setting up the manager configuration 95
9.5 Building and Deploying simulation infrastructure to the Run Farm Machines 96
9.6 Running the simulation 96
9.7 Building Your Own Hardware Designs 100
9.8 System Setup 100
9.8.1 1. Install Vivado for Builds 100
9.8.2 2. Verify Build Farm Machine environment 100
9.9 Configuring a Build in the Manager 101
9.10 Running the Build 102

10 **RHS Research Nitefury II XDMA-based Getting Started Guide** **103**
10.1 FPGA Setup 103
10.2 FireSim Repo Setup 104
10.2.1 Setting up the FireSim Repo 104
10.2.2 Initializing FireSim Config Files 105
10.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup 105
10.3 Running a Single Node Simulation 106
10.3.1 Building target software 106
10.4 Setting up the manager configuration 107
10.5 Building and Deploying simulation infrastructure to the Run Farm Machines 108
10.6 Running the simulation 108
10.7 Building Your Own Hardware Designs 112
10.8 System Setup 112
10.8.1 1. Install Vivado for Builds 112
10.8.2 2. Verify Build Farm Machine environment 112
10.9 Configuring a Build in the Manager 113
10.10 Running the Build 114

11 **(Experimental) Xilinx Alveo U250 Vitis-based Getting Started Guide** **115**
11.1 Initial Setup/Installation 116
11.1.1 FPGA and Tool Setup 116
11.1.2 Setting up your On-Premises Machine 117
11.2 Running a Single Node Simulation 118
11.2.1 Building target software 118
11.3 Setting up the manager configuration 118
11.4 Building and Deploying simulation infrastructure to the Run Farm Machines 122
11.5 Running the simulation 123
11.6 Building Your Own Hardware Designs 126
11.7 Configuring a Build in the Manager 126
11.8 Running the Build 127

12 **Manager Usage (the `firesim` command)** **129**
12.1 Overview 129
12.1.1 "Inputs" to the Manager 129
12.1.2 Logging 129

12.2 Manager Command Line Arguments . . . 130
12.2.1 --runtimeconfigfile FILENAME . . . 132
12.2.2 --buildconfigfile FILENAME . . . 132
12.2.3 --buildrecipesconfigfile FILENAME . . . 132
12.2.4 --hwdbconfigfile FILENAME . . . 132
12.2.5 --overrideconfigdata SECTION PARAMETER VALUE . . . 132
12.2.6 --launchtime TIMESTAMP . . . 132
12.2.7 TASK . . . 132
12.3 Manager Tasks . . . 132
12.3.1 firesim managerinit . . . 133
12.3.2 firesim buildbitstream . . . 133
12.3.3 firesim builddriver . . . 135
12.3.4 firesim tar2afi . . . 135
12.3.5 firesim shareagfi . . . 136
12.3.6 firesim launchrunfarm . . . 136
12.3.7 firesim terminaterunfarm . . . 137
12.3.8 firesim infrasetup . . . 138
12.3.9 firesim boot . . . 138
12.3.10 firesim kill . . . 138
12.3.11 firesim runworkload . . . 138
12.3.12 firesim runcheck . . . 139
12.3.13 firesim enumeratefpgas . . . 139
12.4 Manager URI Paths . . . 139
12.4.1 URI Support . . . 140
12.5 Manager Configuration Files . . . 140
12.5.1 config_runtime.yaml . . . 140
12.5.2 config_build.yaml . . . 146
12.5.3 config_build_recipes.yaml . . . 148
12.5.4 config_hwdb.yaml . . . 154
12.5.5 Run Farm Recipes (run-farm-recipes/*) . . . 156
12.5.6 Build Farm Recipes (build-farm-recipes/*) . . . 164
12.5.7 Bit Builder Recipes (bit-builder-recipes/*) . . . 168
12.6 Manager Environment Variables . . . 171
12.6.1 FIRESIM_RUNFARM_PREFIX . . . 171
12.6.2 FIRESIM_BUILDFARM_PREFIX . . . 171
12.7 Manager Network Topology Definitions (user_topology.py) . . . 171
12.7.1 user_topology.py contents: . . . 172
12.8 AGFI Metadata/Tagging . . . 185

13 Workloads **187**
13.1 FireMarshal . . . 187
13.2 [DEPRECATED] Defining Custom Workloads . . . 187
13.2.1 Uniform Workload JSON . . . 188
13.2.2 Non-uniform Workload JSON (explicit job per simulated node) . . . 189

14 Targets **191**
14.1 Restrictions on Target RTL . . . 191
14.1.1 Including Verilog IP . . . 191
14.1.2 Multiple Clock Domains . . . 192
14.2 Target-Side FPGA Constraints . . . 193
14.2.1 RAM Inference Hints . . . 193
14.3 Provided Target Designs . . . 194
14.3.1 Target Generator Organization . . . 194
14.3.2 Specifying A Target Instance . . . 194

14.4 Rocket Chip Generator-based SoCs (firesim project) . . . 196
14.4.1 Rocket-based SoCs . . . 196
14.4.2 BOOM-based SoCs . . . 196
14.4.3 Generating A Different FASED Memory-Timing Model Instance . . . 196
14.5 Midas Examples (midasexamples project) . . . 197
14.5.1 Examples . . . 197
14.6 FASED Tests (fasedtests project) . . . 197
14.6.1 Examples . . . 197

15 Debugging in Software **199**
15.1 Debugging & Testing with Metasimulation . . . 199
15.1.1 Supported Host Simulators . . . 199
15.1.2 Running Metasimulations using the FireSim Manager . . . 200
15.1.3 Understanding a Metasimulation Waveform . . . 201
15.1.4 Scala Tests . . . 203
15.1.5 Running Metasimulations through Make . . . 203
15.1.6 Metasimulation vs. Target simulation performance . . . 204

16 Debugging and Profiling on the FPGA **207**
16.1 Capturing RISC-V Instruction Traces with TracerV . . . 207
16.1.1 Building a Design with TracerV . . . 207
16.1.2 Enabling Tracing at Runtime . . . 207
16.1.3 Selecting a Trace Output Format . . . 208
16.1.4 Setting a TracerV Trigger . . . 208
16.1.5 Interpreting the Trace Result . . . 210
16.1.6 Caveats . . . 211
16.2 Assertion Synthesis: Catching RTL Assertions on the FPGA . . . 211
16.2.1 Enabling Assertion Synthesis . . . 211
16.2.2 Runtime Behavior . . . 211
16.2.3 Related Publications . . . 212
16.3 Printf Synthesis: Capturing RTL printf Calls when Running on the FPGA . . . 212
16.3.1 Enabling Printf Synthesis . . . 212
16.3.2 Runtime Arguments . . . 213
16.3.3 Related Publications . . . 213
16.4 AutoILA: Simple Integrated Logic Analyzer (ILA) Insertion . . . 213
16.4.1 Enabling AutoILA . . . 214
16.4.2 Annotating Signals . . . 214
16.4.3 Setting a ILA Depth . . . 214
16.4.4 Using the ILA at Runtime . . . 214
16.5 AutoCounter: Profiling with Out-of-Band Performance Counter Collection . . . 215
16.5.1 Chisel Interface . . . 215
16.5.2 Enabling AutoCounter in Golden Gate . . . 216
16.5.3 Rocket Chip Cover Functions . . . 216
16.5.4 AutoCounter Runtime Parameters . . . 217
16.5.5 AutoCounter CSV Output Format . . . 217
16.5.6 Using TracerV Trigger with AutoCounter . . . 218
16.5.7 AutoCounter using Synthesizable Printfs . . . 218
16.5.8 Reset & Timing Considerations . . . 218
16.6 TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation . . . 219
16.6.1 What are Flame Graphs? . . . 219
16.6.2 Prerequisites . . . 219
16.6.3 Enabling Flame Graph generation in `config_runtime.yaml` . . . 219
16.6.4 Producing DWARF information to supply to the TracerV driver . . . 220
16.6.5 Modifying your workload description . . . 220

16.6.6 Running a simulation . . . 221
16.6.7 Caveats . . . 221
16.7 Spike Co-simulation with BOOM designs . . . 221
16.7.1 Building a Design with Cospike . . . 222
16.7.2 Troubleshooting Cospike Simulations with Meta-Simulations . . . 222
16.8 Debugging a Hanging Simulator . . . 222
16.8.1 Case 1: Target hang. . . . 222
16.8.2 Case 2: Simulator hang due to FPGA-side token starvation. . . . 223
16.8.3 Case 3: Simulator hang due to driver-side deadlock. . . . 223
16.8.4 Simulator Heartbeat PlusArgs . . . 223
16.9 PlusArg Synthesis: Runtime Modification of RTL . . . 223
16.9.1 Enabling PlusArg Synthesis . . . 223
16.9.2 Runtime Arguments . . . 224

17 Non-Source Dependency Management **225**
17.1 Updating a Package Version . . . 225
17.2 Multiple Environments . . . 227
17.3 Adding a New Dependency . . . 227
17.4 Building From Source . . . 228
17.5 Running Conda with sudo . . . 228
17.6 Running things from your Conda environment with sudo . . . 228
17.7 Additional Resources . . . 229

18 Supernode - Multiple Simulated SoCs Per FPGA **231**
18.1 Introduction . . . 231
18.2 Building Supernode Designs . . . 231
18.3 Running Supernode Simulations . . . 232
18.4 Work in Progress! . . . 233

19 FireAxe - Partitioning onto Multiple FPGAs **235**
19.1 FireAxe Overview . . . 235
19.2 Partition Modes . . . 235
19.2.1 Exact-Mode . . . 235
19.2.2 Fast-Mode . . . 236
19.2.3 NoC-Partition-Mode . . . 236
19.3 Supported Platforms . . . 236
19.3.1 EC2 F1 . . . 236
19.3.2 Local FPGAs w/ QSFP Cables . . . 236
19.4 Running Fast Mode Simulations . . . 236
19.4.1 1. Building Partitioned Sims: Setting up FireAxe Target configs . . . 237
19.4.2 2. Building Partitioned Sims: *config_build_recipes.yaml* . . . 238
19.4.3 3. Running Partitioned Simulations: *user_topology.py* . . . 238
19.4.4 4. Running Partitioned Simulations: config_runtime.yaml . . . 240
19.5 Running Exact Mode Simulations . . . 241
19.5.1 1. Building Partitioned Sims: Setting up FireAxe Target configs . . . 241
19.5.2 2. Building Partitioned Sims: *config_build_recipes.yaml* . . . 242
19.5.3 3. Running Partitioned Simulations: *user_topology.py* . . . 242
19.5.4 4. Running Partitioned Simulations: *config_runtime.yaml* . . . 243
19.6 Running NoC Partition Mode Simulations . . . 243
19.6.1 1. Building Partitioned Sims: Setting up FireAxe Target configs . . . 243
19.6.2 2. Building Partitioned Sims: *config_build_recipes.yaml* . . . 245
19.6.3 3. Running Partitioned Simulations: *user_topology.py* . . . 245
19.6.4 4. Running Partitioned Simulations: *config_runtime.yaml* . . . 246
19.7 Miscellaneous Tips . . . 246

19.7.1 Running FireAxe Metasims . . . 246

20 Miscellaneous Tips **247**
20.1 Add the `fsimcluster` column to your AWS management console . . . 247
20.2 FPGA Dev AMI Remote Desktop Setup . . . 247
20.3 Experimental Support for SSHing into simulated nodes and accessing the internet from within simulations . . . 247
20.4 Navigating the FireSim Codebase . . . 249
20.5 Using FireSim CI . . . 249
20.6 How to view AWS build logs when AGFI build fails . . . 249

21 Adding support for a new FPGA **251**
21.1 Adding a new FireSim `platform` . . . 251
21.2 Adding other collateral (Scala, C++, Make, etc) . . . 252
21.3 Manager build modifications . . . 253
21.4 Manager run modifications . . . 253

22 Using FireSim without Chipyard **255**
22.1 Simple Counter Example Project . . . 255
22.1.1 Top-Level Harness . . . 255
22.1.2 C++ Driver Top . . . 258
22.1.3 Make fragments . . . 259
22.1.4 Running meta-simulations and more . . . 259
22.2 Chipyard Example . . . 260
22.2.1 Top-Level Harness . . . 260
22.2.2 C++ Driver Top . . . 260
22.2.3 Make fragments . . . 260

23 FireSim Asked Questions **261**
23.1 I just bumped the FireSim repository to a newer commit and simulations aren't running. What is going on? . . . 261
23.2 Is there a good way to keep track of what AGFI corresponds to what FireSim commit? . . . 261
23.3 Help, My Simulation Hangs! . . . 262
23.4 Should My Simulator Produce Different Results Across Runs? . . . 262
23.5 Is there a way to compress workload results when copying back to the manager instance? . . . 262

24 Overview & Philosophy **263**
24.1 Golden Gate vs FPGA Prototyping . . . 263
24.2 Why Use Golden Gate & FireSim . . . 263
24.3 Why Not Golden Gate . . . 264
24.4 How is Host-Decoupling Implemented? . . . 264

25 Target Abstraction & Host Decoupling **265**
25.1 The Target as a Dataflow Graph . . . 265
25.2 Model Implementations . . . 265
25.3 Expressing the Target Graph . . . 266
25.4 Latency-Insensitive Bounded Dataflow Networks . . . 266

26 Target-to-Host Bridges **267**
26.1 Terminology . . . 267
26.2 Target Side . . . 268
26.2.1 Type Parameters: . . . 268
26.2.2 Abstract Members: . . . 268
26.3 What Happens Next? . . . 268
26.4 Host Side . . . 269

26.5 Compile-Time (Parameterization) vs Runtime Configuration . . . 269
26.6 Target-Side vs Host-Side Parameterization . . . 269

27 Bridge Deep Dive **271**
27.1 UART Bridge (Host-MMIO) . . . 271
27.1.1 Target Side . . . 272
27.1.2 Host-Side BridgeModule . . . 273
27.1.3 Host-Side Driver . . . 275
27.1.4 Build-System Modifications . . . 277

28 Simulation Triggers **281**
28.1 Quick-Start Guide . . . 281
28.1.1 Level-Sensitive Trigger Source . . . 281
28.1.2 Distributed, Edge-Sensitive Trigger Source . . . 281
28.2 Chisel API . . . 282
28.2.1 Trigger Sources . . . 282
28.2.2 Trigger Sinks . . . 282
28.3 Trigger Timing . . . 283
28.4 Limitations & Pitfalls . . . 284

29 Optimizing FPGA Resource Utilization **285**
29.1 Multi-Ported Memory Optimization . . . 285
29.2 Multi-Threading of Repeated Instances . . . 286

30 Output Files **287**
30.1 Core Files . . . 287
30.2 FPGA Build Files . . . 287
30.3 Metasimulation Files . . . 288

31 Compiler & Driver Development **289**
31.1 Integration Tests . . . 289
31.1.1 Key Files & Locations . . . 290
31.1.2 Defining a New Test . . . 290
31.2 Synthesizable Unit Tests . . . 290
31.2.1 Key Files & Locations . . . 291
31.2.2 Defining a New Test . . . 291
31.3 Scala Unit Testing . . . 291
31.3.1 Key Files & Locations . . . 291
31.3.2 Defining A New Test . . . 292
31.4 C/C++ guidelines . . . 292
31.5 Scala guidelines . . . 292

32 Complete FPGA Metasimulation **293**
32.1 Usage . . . 293

33 Visual Studio Code Integration **295**
33.1 General Setup . . . 295
33.2 Workspace Locations . . . 295
33.3 Scala Development . . . 295
33.3.1 How To Use (Remote Manager) . . . 296
33.3.2 Limitations . . . 296
33.3.3 Other Notes . . . 296

34 Managing the Conda Lock File **297**
34.1 Updating Conda Requirements . . . 297

34.2 Caveats of the Conda Lock File and CI 297

35 Manager Development **299**

36 External Tutorial Setup **301**

37 Indices and tables **305**

New to FireSim? Jump to the *FireSim Basics* page for more info.

CHAPTER

ONE

FIRESIM BASICS

FireSim is an open-source FPGA-accelerated full-system hardware simulation platform that makes it easy to validate, profile, and debug RTL hardware implementations at 10s to 100s of MHz. FireSim simplifies co-simulating ASIC RTL with cycle-accurate hardware and software models for other system components (e.g. I/Os). FireSim can productively scale from individual SoC simulations hosted on on-prem FPGAs (e.g., a single Xilinx Alveo board attached to a desktop) to massive datacenter-scale simulations harnessing hundreds of cloud FPGAs (e.g., on Amazon EC2 F1).

FireSim users across academia and industry (at 20+ institutions) have published over 40 papers using FireSim in many areas, including computer architecture, systems, networking, security, scientific computing, circuits, design automation, and more (see the Publications page on the FireSim website to learn more). FireSim has also been used in the development of commercially-available silicon. FireSim was originally developed in the Electrical Engineering and Computer Sciences Department at the University of California, Berkeley, but now has industrial and academic contributors from all over the world.

This documentation will walk you through getting started with using FireSim on your platform and also serves as a reference for more advanced FireSim features. For higher-level technical discussion about FireSim, see the FireSim website.

1.1 Common FireSim usage models

Below are three common usage models for FireSim when paired with Chipyard (which provides the SoC design to run with FireSim). The first two are the most common, while the third model is primarily for those interested in warehouse-scale computer research. The getting started guides on this documentation site will cover all three models.

1.1.1 1. Single-Node Simulations Using One or More On-Premises FPGAs

In this usage model, FireSim allows for simulation of targets consisting of individual SoC designs (e.g., those produced by Chipyard) at 150+ MHz running on on-premises FPGAs, such as those attached to your local desktop, laptop, or cluster. Just like on the cloud, the FireSim manager can automatically distribute and manage jobs on one or more on-premises FPGAs, including running complex workloads like SPECInt2017 with full reference inputs.

1.1.2 2. Single-Node Simulations Using Cloud FPGAs

This usage model is similar to the previous on-premises case, but instead deploys simulations on FPGAs attached to cloud instances, rather than requiring users to obtain and set-up on-premises FPGAs. This allows for dynamically scaling the number of FPGAs in-use to match workload requirements. For example, on AWS EC2 F1, it is just as cost effective to run the 10 workloads in SPECInt2017 in parallel on 10 cloud FPGAs vs. running them serially on one cloud FPGA.

Note

All automation in FireSim works in both the on-premises and cloud usage models, which enables a **hybrid usage model** where early development happens on one (or a small cluster of) on-premises FPGA(s), while bursting to a large number of cloud FPGAs when a high degree of parallelism is necessary.

1.1.3 3. Datacenter/Cluster Simulations on On-Premises or Cloud FPGAs

In this mode, FireSim also models a cycle-accurate network with parameterizeable bandwidth, link latency, and configurable topology to accurately model current and future datacenter-scale systems. For example, FireSim has been used to simulate 1024 quad-core RISC-V Rocket Chip-based nodes, interconnected by a 200 Gbps, 2us Ethernet network. To learn more about this use case, see our ISCA 2018 paper.

1.2 Other Use Cases

If you have other use-cases that we haven't covered, feel free to contact us!

1.2.1 Non-Chipyard-based target simulation

While most users use FireSim with Chipyard, users can also use FireSim seperately from Chipyard. For now the best example on how to do this is to follow along with how Chipyard is setup with FireSim. Soon we will release a guide on how to use FireSim by itself! Stay tuned!

1.3 Choose your platform to get started

FireSim supports many types of FPGAs and FPGA platforms! Click one of the following links to work through the getting started guide for your particular platform.

- *AWS EC2 F1 Getting Started Guide*
  - Status: All FireSim Features Supported.
- *Xilinx Alveo U200 XDMA-based Getting Started Guide*
  - Status: All FireSim Features Supported.
- *Xilinx Alveo U250 XDMA-based Getting Started Guide*
  - Status: All FireSim Features Supported.
- *Xilinx Alveo U280 XDMA-based Getting Started Guide*
  - Status: All FireSim Features Supported.

- *Xilinx VCU118 XDMA-based Getting Started Guide*
  - Status: All FireSim Features Supported.
- *RHS Research Nitefury II XDMA-based Getting Started Guide*
  - Status: All FireSim Features Supported.
- *(Experimental) Xilinx Alveo U250 Vitis-based Getting Started Guide*
  - Status: DMA-based Bridges Not Supported. The Vitis-based U250 flow is **not recommended** unless you have specific constraints that require using Vitis. Notably, the Vitis-based flow does not support DMA-based FireSim bridges (e.g., TracerV, Synthesizable Printfs, etc.), while the XDMA-based flows support all FireSim features, as shown above. If you're unsure, use the XDMA-based U250 flow instead: *Xilinx Alveo U250 XDMA-based Getting Started Guide*.

CHAPTER TWO

BACKGROUND/TERMINOLOGY

Before we jump into setting up FireSim, it is important to clarify several terms that we will use throughout the rest of this documentation.

First, to disambiguate between the hardware being simulated and the computers doing the simulating, we define:

Target
: The design and environment being simulated. Commonly, a group of one or more RISC-V SoCs with or without a network between them.

Host
: The computers/FPGAs executing the FireSim simulation – the **Run Farm** below.

We frequently prefix words with these terms. For example, software can run on the simulated RISC-V system (*target*-software) or on a host x86 machine (*host*-software).

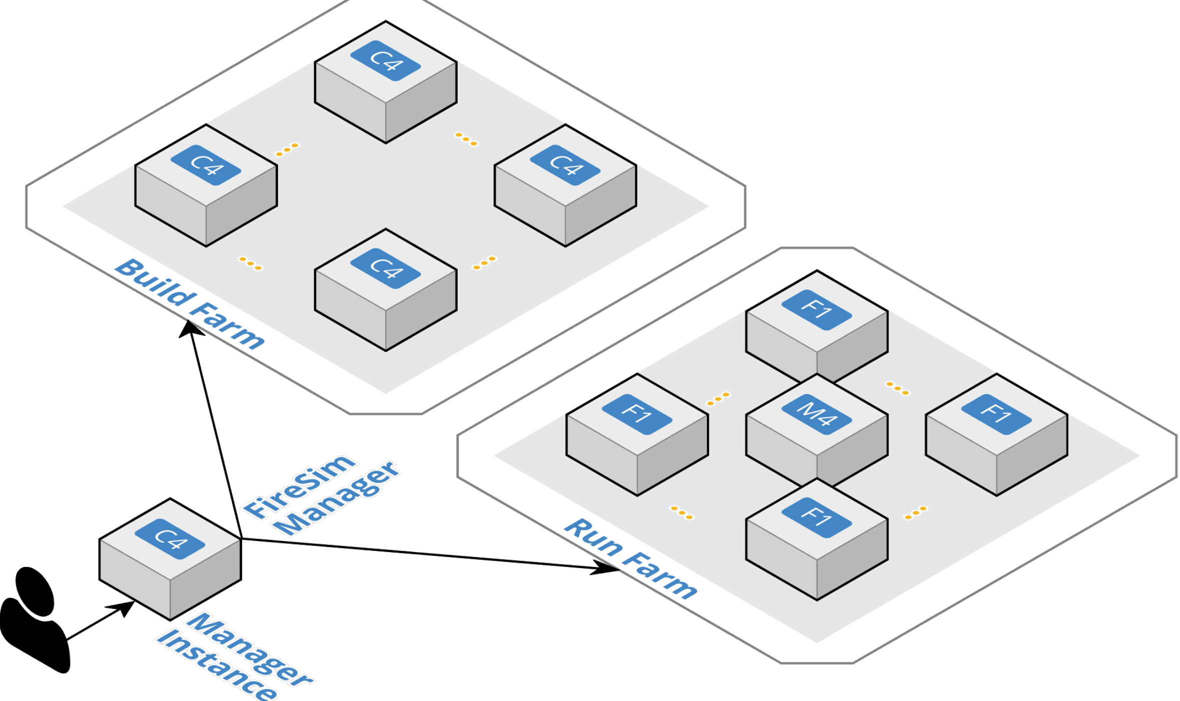


Fig. 1: FireSim Infrastructure Diagram

FireSim Manager (`firesim`)
This program (available on your path as `firesim` once we source necessary scripts) automates the work required to launch FPGA builds and run simulations. Most users will only have to interact with the manager most of the time. If you're familiar with tools like Vagrant or Docker, the `firesim` command is just like the `vagrant` and `docker` commands, but for FPGA simulators instead of VMs/containers.

Machines used to build and run FireSim simulations are broadly classified into three groups:

Manager Machine/Instance
This is the main host machine (e.g., a "vanilla" AWS EC2 instance without an FPGA attached or your local desktop/server) that you will "do work" on. This is where you'll clone your copy of FireSim and use the FireSim Manager to deploy builds/simulations from.

Build Farm Machine/Instances
These are a collection of cloud instances or local machines ("build farm instances/machines") that are used by the FireSim manager to run FPGA bitstream builds. The manager will automatically ship all sources necessary to run builds to these instances/machines and will run the Verilog to FPGA bitstream build process on them.

Run Farm Machine/Instances
These are a collection of cloud instances or local machines ("run farm instances/machines") with FPGAs attached that the manager manages and deploys simulations onto. You can use multiple Run Farms in parallel to run multiple separate simulations in parallel.

Please refer to the starter tutorials on specifics on how to manage these machines.

One final piece of terminology will also be referenced throughout these docs:

Golden Gate
The FIRRTL compiler in FireSim that converts target RTL into a decoupled simulator. Formerly named MIDAS.

CHAPTER

THREE

AWS EC2 F1 SYSTEM SETUP

This section will guide you through initial setup of your AWS account to support FireSim, as well as cloning/installing FireSim on your manager instance.

3.1 First-time AWS User Setup

If you've never used AWS before and don't have an account, follow the instructions below to get started.

3.1.1 Creating an AWS Account

First, you'll need an AWS account. Create one by going to aws.amazon.com and clicking "Sign Up." You'll want to create a personal account. You will have to give it a credit card number.

3.1.2 Requesting Limit Increases

AWS limits access to particular instance types for new/infrequently used accounts to protect their infrastructure. You can learn more about how these limits/quotas work here.

You should make sure that your account has the ability to launch a sufficient number of instances to follow this guide by looking at the "Service Quotas" page in the AWS Console, which you can access here. Be sure that the correct region is selected once you open this page.

The values listed on this page represent the maximum number vCPUs of any of these instances that you can run at once, which will limit the size of simulations (e.g., number of parallel FPGAs) that you can run. If you need to increase your limits, follow the instructions below.

To complete this guide, you need to have the following limits:

- `Running On-Demand F instances`: 64 vCPUs.
  - This is sufficient for 8 parallel FPGAs. Each 8 vCPUs = one FPGA.
- `Running On-Demand Standard (A, C, D, H, I, M, R, T, Z) instances`: 24 vCPUs.
  - This is sufficient for one `c5.4xlarge` manager instance and one `z1d.2xlarge` build farm instance.

If you have insufficient limits, request a limit increase by following these steps: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-resource-limits.html#request-increase

In your request, enter the vCPU limits for the two instance classes shown above. This process sometimes has a human in the loop, so you should submit it ASAP. At this point, you should wait for the response to this request.

Hit Next below to continue.

3.2 Configuring Required Infrastructure in Your AWS Account

Once we have an AWS Account setup, we need to perform some advance setup of resources on AWS. You will need to follow these steps even if you already had an AWS account as these are FireSim-specific.

3.2.1 Select a region

Head to the EC2 Management Console. In the top right corner, ensure that the correct region is selected. You should select one of: `us-east-1` (N. Virginia), `us-west-2` (Oregon), `ap-southeast-2` (Sydney), `eu-central-1` (Frankfurt), `eu-west-1` (Ireland), `eu-west-2` (London), since F1 instances are only available in those regions. For the most current list of regions supporting F1 instance, see Amazon EC2 instance types by Region.

Once you select a region, it's useful to bookmark the link to the EC2 console, so that you're always sent to the console for the correct region.

3.2.2 Key Setup

In order to enable automation, you will need to create a key named `firesim`, which we will use to launch all instances (Manager Instance, Build Farm, Run Farm).

To do so, click "Key Pairs" under "Network & Security" in the left-sidebar. Follow the prompts, name the key `firesim`, and save the private key locally as `firesim.pem`. You can use this key to access all instances from your local machine. We will copy this file to our manager instance later, so that the manager can also use it.

3.2.3 Double Check your EC2 Instance Limits

AWS limits access to particular instance types for new/infrequently used accounts to protect their infrastructure. You can learn more about how these limits/quotas work here.

You should make sure that your account has the ability to launch a sufficient number of instances to follow this guide by looking at the "Service Quotas" page in the AWS Console, which you can access here. Be sure that the correct region is selected once you open this page.

The values listed on this page represent the maximum number vCPUs of any of these instances that you can run at once, which will limit the size of simulations (e.g., number of parallel FPGAs) that you can run. If you need to increase your limits, follow the instructions below.

To complete this guide, you need to have the following limits:

- `Running On-Demand F instances`: 64 vCPUs.
  - This is sufficient for 8 parallel FPGAs. Each 8 vCPUs = one FPGA.
- `Running On-Demand Standard (A, C, D, H, I, M, R, T, Z) instances`: 24 vCPUs.
  - This is sufficient for one `c5.4xlarge` manager instance and one `z1d.2xlarge` build farm instance.

If you have insufficient limits, follow the instructions on the *Requesting Limit Increases* page.

3.2.4 Start a t2.nano instance to run the remaining configuration commands

To avoid having to deal with the messy process of installing packages on your local machine, we will spin up a very cheap `t2.nano` instance to run a series of one-time aws configuration commands to setup our AWS account for FireSim. At the end of these instructions, we'll terminate the `t2.nano` instance. If you happen to already have `boto3` and the AWS CLI installed on your local machine, you can do this locally.

Launch a `t2.nano` by following these instructions:

1. Go to the EC2 Management Console and click "Launch Instance"
2. In "Application and OS Images (Amazon Machine Image)", use "Amazon Linux", which should be the default.
3. In "Instance type", select `t2.nano`.
4. In "Key pair (login)", choose the `firesim` key pair we created previously.
5. Click "Launch Instance" in the right-hand sidebar (we don't need to change any other settings)
6. Click on the instance ID and note the instance's public IP address.

3.2.5 Run scripts from the t2.nano

SSH into the `t2.nano` like so:

```
ssh -i firesim.pem ec2-user@INSTANCE_PUBLIC_IP
```

Which should present you with something like:

```
   ,     #_
   ~\_  ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
[ec2-user@ip-172-31-85-76 ~]$
```

On this machine, run the following:

```
aws configure
[follow prompts]
```

Within the prompt, you should specify the same region that you chose above (one of `us-east-1`, `us-west-2`, `eu-west-1`) and set the default output format to `json`. You will need to generate an AWS access key in the "Security Credentials" menu of your AWS settings (as instructed in https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html#Using_CreateAccessKey). You should keep the AWS access key information in a safe place, so that you can refer to it again when setting up the manager instance. You can learn more about the `aws configure` command on the following page: https://docs.aws.amazon.com/cli/latest/reference/configure/index.html

Again on the `t2.nano` instance, do the following:

```
sudo yum install -y python3-pip
sudo python3 -m pip install boto3
```

(continues on next page)

(continued from previous page)

```
sudo python3 -m pip install --upgrade awscli
wget https://raw.githubusercontent.com/firesim/firesim/t/deploy/awstools/aws_setup.py
chmod +x aws_setup.py
./aws_setup.py
```

The final command should print the following:

```
Creating VPC for FireSim...
Success!
Creating a subnet in the VPC for each availability zone...
Success!
Creating a security group for FireSim...
Success!
```

This will have created a VPC named `firesim` and a security group named `firesim` in your account.

3.2.6 Terminate the t2.nano

At this point, we are finished with the general account configuration. You should terminate the t2.nano instance you created, since we do not need it anymore (and it shouldn't contain any important data).

3.2.7 Subscribe to the AWS FPGA Developer AMI

Go to the AWS Marketplace page for the FPGA Developer AMI. Click the button to subscribe to the FPGA Dev AMI (it should be free) and follow the prompts to accept the EULA (but do not launch any instances).

Now, hit next to continue on to setting up our Manager Instance.

3.3 Setting up your Manager Instance

3.3.1 Launching a "Manager Instance"

Warning

These instructions refer to fields in EC2's new launch instance wizard. Refer to version 1.13.4 of the documentation for references to the old wizard, being wary that specifics, such as the AMI ID selection, may be out of date.

Now, we need to launch a "Manager Instance" that acts as a "head" node that we will `ssh` or `mosh` into to work from. Since we will deploy the heavy lifting to separate `z1d.2xlarge` and `f1` instances later, the Manager Instance can be a relatively cheap instance. In this guide, however, we will use a `c5.4xlarge`, running the AWS FPGA Developer AMI. (Be sure to subscribe to the AMI if you have not done so. See *Subscribe to the AWS FPGA Developer AMI*. Note that it might take a few minutes after subscribing to the AMI to be able to launch instances using it.)

Head to the EC2 Management Console. In the top right corner, ensure that the correct region is selected.

To launch a manager instance, follow these steps:

1. From the main page of the EC2 Management Console, click *Launch Instance* button and click *Launch Instance* in the dropdown that appears. We use an on-demand instance here, so that your data is preserved when you stop/start the instance, and your data is not lost when pricing spikes on the spot market.

2. In the *Name* field, give the instance a recognizable name, for example `firesim-manager-1`. This is purely for your own convenience and can also be left blank.
3. In the *Application and OS Images* search box, search for `FPGA Developer AMI - 1.12.2-40257ab5-6688-4c95-97d1-e251a40fd1fc` and select the AMI that appears under the **Community AMIs** tab (there should be only one).
   - If you find that there are no results for this search, you can try incrementing the last part of the **version number** (`Z` in `X.Y.Z`) in the search string, e.g., `1.12.2 -> 1.12.3`. Other parts of the search string should be unchanged.
   - **Do not** use *FPGA Developer AMI* from the *AWS Marketplace AMIs* tab, as you will likely get an incorrect version of the AMI.
4. In the *Instance Type* drop-down, select the instance type of your choosing. A good choice is a `c5.4xlarge` (16 cores, 32 GiB DRAM) or a `z1d.2xlarge` (8 cores, 64 GiB DRAM).
5. In the *Key pair (login)* drop-down, select the `firesim` key pair we set up earlier.
6. In the *Network settings* drop-down click *edit* and modify the following settings:
   1. Under *VPC - required*, select the `firesim` VPC. Any subnet within the `firesim` VPC is fine.
   2. Under *Firewall (security groups)*, click *Select existing security group* and in the *Common security groups* dropdown that appears, select the `firesim` security group that was automatically created for you earlier. Do **NOT** select the `for-farms-only-firesim` security group that might also be in the list (it is also fine if this group does not appear in your list).
7. In the *Configure storage* section, increase the size of the root volume to at least 300GB. The default of 120GB can quickly become too small as you accumulate large Vivado reports/outputs, large waveforms, XSim outputs, and large root filesystems for simulations. You should remove the small (5-8GB) secondary volume that is added by default.
8. In the *Advanced details* drop-down, change the following:
   1. Under *Termination protection*, select Enable. This adds a layer of protection to prevent your manager instance from being terminated by accident. You will need to disable this setting before being able to terminate the instance using usual methods.
   2. Under *User data*, paste the following into the provided textbox:

```
#!/bin/bash

MACHINE_LAUNCH_DIR=/tmp
export HOME="${HOME:-/root}"

CONDA_INSTALL_PREFIX=/opt/conda
CONDA_INSTALLER_VERSION=23.1.0-1
CONDA_INSTALLER="https://github.com/conda-forge/miniforge/releases/download/$
↪{CONDA_INSTALLER_VERSION}/Miniforge3-${CONDA_INSTALLER_VERSION}-Linux-x86_64.
↪sh"
CONDA_CMD="conda" # some installers install mamba or micromamba
CONDA_ENV_NAME="firesim"
CONDA_SHELL_TYPE=bash

DRY_RUN_OPTION=""
DRY_RUN_ECHO=()
REINSTALL_CONDA=0
USE_LIBMAMBA_SOLVER=0
```

(continues on next page)

(continued from previous page)

```
usage()
{
    echo "Usage: $0 [options]"
    echo
    echo "Options:"
    echo "[--help]                  List this help"
    echo "[--prefix <prefix>]       Install prefix for conda. Defaults to
↪$CONDA_INSTALL_PREFIX."
    echo "                          If <prefix>/bin/conda already exists, it␣
↪will be used and install is skipped."
    echo "[--env <name>]            Name of environment to create for conda.␣
↪Defaults to '$CONDA_ENV_NAME'."
    echo "[--dry-run]               Pass-through to all conda commands and only␣
↪print other commands."
    echo "                          NOTE: --dry-run will still install conda to␣
↪--prefix"
    echo "[--reinstall-conda]       Repairs a broken base environment by␣
↪reinstalling."
    echo "                          NOTE: will only reinstall conda and exit␣
↪without modifying the --env"
    echo "[--shell]                 Run initialization for a specific shell.␣
↪Defaults to $CONDA_SHELL_TYPE."
    echo "[--use-libmamba-solver]   Use experimental libmamba solver for conda."
    echo
    echo "Examples:"
    echo "  % $0"
    echo "     Install into default system-wide prefix (using sudo if needed)␣
↪and add install to system-wide /etc/profile.d"
    echo "  % $0 --prefix ~/conda --env my_custom_env"
    echo "     Install into $HOME/conda and add install to $CONDA_SHELL_TYPE␣
↪init files (i.e. ~/.*rc)"
    echo "  % $0 --prefix \${CONDA_EXE%/bin/conda} --env my_custom_env"
    echo "     Create my_custom_env in existing conda install"
    echo "     NOTES:"
    echo "       * CONDA_EXE is set in your environment when you activate a␣
↪conda env"
    echo "       * my_custom_env will not be activated by default at login see /
↪etc/profile.d/conda.sh & $CONDA_SHELL_TYPE init files (i.e. ~/.*rc)"
}


while [ $# -gt 0 ]; do
    case "$1" in
        --help)
            usage
            exit 1
            ;;
        --prefix)
            shift
            CONDA_INSTALL_PREFIX="$1"
            shift
```

(continues on next page)

(continued from previous page)

```
            ;;
        --env)
            shift
            CONDA_ENV_NAME="$1"
            shift
            if [[ "$CONDA_ENV_NAME" == "base" ]]; then
                echo "::ERROR:: best practice is to install into a named␣
↪environment, not base. Aborting."
                exit 1
            fi
            ;;
        --dry-run)
            shift
            DRY_RUN_OPTION="--dry-run"
            DRY_RUN_ECHO=(echo "Would Run:")
            ;;
        --reinstall-conda)
            shift
            REINSTALL_CONDA=1
            ;;
        --shell)
            shift
            CONDA_SHELL_TYPE="$1"
            shift
            ;;
        --use-libmamba-solver)
            shift
            USE_LIBMAMBA_SOLVER=1
            ;;
        *)
            echo "Invalid Argument: $1"
            usage
            exit 1
            ;;
    esac
done

if [[ $REINSTALL_CONDA -eq 1 && -n "$DRY_RUN_OPTION" ]]; then
    echo "::ERROR:: --dry-run and --reinstall-conda are mutually exclusive. ␣
↪Pick one or the other."
fi

set -ex
set -o pipefail

{

    # uname options are not portable so do what https://www.gnu.org/software/
↪coreutils/faq/coreutils-faq.html#uname-is-system-specific
    # suggests and iteratively probe the system type
    if ! type uname >&/dev/null; then
        echo "::ERROR:: need 'uname' command available to determine if we␣
```

(continues on next page)

(continued from previous page)

```
↪support this sytem"
        exit 1
    fi

    if [[ "$(uname)" != "Linux" ]]; then
        echo "::ERROR:: $0 only supports 'Linux' not '$(uname)'"
        exit 1
    fi

    if [[ "$(uname -mo)" != "x86_64 GNU/Linux" ]]; then
        echo "::ERROR:: $0 only supports 'x86_64 GNU/Linux' not '$(uname -io)'"
        exit 1
    fi

    if [[ ! -r /etc/os-release ]]; then
        echo "::ERROR:: $0 depends on /etc/os-release for distro-specific setup␣
↪and it doesn't exist here"
        exit 1
    fi

    OS_FLAVOR=$(grep '^ID=' /etc/os-release | awk -F= '{print $2}' | tr -d '"')

    echo "machine launch script started" > "$MACHINE_LAUNCH_DIR/machine-
↪launchstatus"
    chmod ugo+r "$MACHINE_LAUNCH_DIR/machine-launchstatus"

    # platform-specific setup (pre-conda install)
    case "$OS_FLAVOR" in
        ubuntu)
            ;;
        centos)
            ;;
        amzn)
            ;;
        debian)
            ;;
        *)
            echo "::ERROR:: Unknown OS flavor '$OS_FLAVOR'. Unable to do␣
↪platform-specific setup."
            exit 1
            ;;
    esac

    # everything else is platform-agnostic and could easily be expanded to␣
↪Windows and/or OSX

    SUDO=""
    prefix_parent=$(dirname "$CONDA_INSTALL_PREFIX")
    if [[ ! -e "$prefix_parent" ]]; then
        mkdir -p "$prefix_parent" || SUDO=sudo
    elif [[ ! -w "$prefix_parent" ]]; then
        SUDO=sudo
```

(continues on next page)

(continued from previous page)

```
    fi

    if [[ -n "$SUDO" ]]; then
        echo "::INFO:: using 'sudo' to install conda"
        # ensure files are read-execute for everyone
        umask 022
    fi

    if [[ -n "$SUDO"  || "$(id -u)" == 0 ]]; then
        INSTALL_TYPE=system
    else
        INSTALL_TYPE=user
    fi

    # to enable use of sudo and avoid modifying 'secure_path' in /etc/sudoers,␣
↪we specify the full path to conda
    CONDA_EXE="${CONDA_INSTALL_PREFIX}/bin/$CONDA_CMD"

    if [[ -x "$CONDA_EXE" && $REINSTALL_CONDA -eq 0 ]]; then
        echo "::INFO:: '$CONDA_EXE' already exists, skipping conda install"
    else
        wget -O install_conda.sh "$CONDA_INSTALLER"  || curl -fsSLo install_
↪conda.sh "$CONDA_INSTALLER"
        if [[ $REINSTALL_CONDA -eq 1 ]]; then
            conda_install_extra="-u"
            echo "::INFO:: RE-installing conda to '$CONDA_INSTALL_PREFIX'"
        else
            conda_install_extra=""
            echo "::INFO:: installing conda to '$CONDA_INSTALL_PREFIX'"
        fi
        # -b for non-interactive install
        $SUDO bash ./install_conda.sh -b -p "$CONDA_INSTALL_PREFIX" $conda_
↪install_extra
        rm ./install_conda.sh

        # get most up-to-date conda version
        "${DRY_RUN_ECHO[@]}" $SUDO "$CONDA_EXE" update $DRY_RUN_OPTION -y -n␣
↪base -c conda-forge conda

        # see https://conda-forge.org/docs/user/tipsandtricks.html#multiple-
↪channels
        # for more information on strict channel_priority
        "${DRY_RUN_ECHO[@]}" $SUDO "$CONDA_EXE" config --system --set channel_
↪priority flexible
        # by default, don't mess with people's PS1, I personally find it␣
↪annoying
        "${DRY_RUN_ECHO[@]}" $SUDO "$CONDA_EXE" config --system --set changeps1␣
↪false
        # don't automatically activate the 'base' environment when initializing␣
↪shells
        "${DRY_RUN_ECHO[@]}" $SUDO "$CONDA_EXE" config --system --set auto_
↪activate_base false
```

(continues on next page)

(continued from previous page)

```
        # automatically use the ucb-bar channel for specific packages https://
↪anaconda.org/ucb-bar/repo
        "${DRY_RUN_ECHO[@]}" $SUDO "$CONDA_EXE" config --system --add channels␣
↪ucb-bar

        # conda-build is a special case and must always be installed into the␣
↪base environment
        $SUDO "$CONDA_EXE" install $DRY_RUN_OPTION -y -n base conda-build

        if [[ $USE_LIBMAMBA_SOLVER -eq 1 ]]; then
            # conda-libmamba-solver is a special case and must always be␣
↪installed into the base environment
            # see https://www.anaconda.com/blog/a-faster-conda-for-a-growing-
↪community
            $SUDO "$CONDA_EXE" install $DRY_RUN_OPTION -y -n base conda-
↪libmamba-solver
            # Use the fast solver by default
            "${DRY_RUN_ECHO[@]}" $SUDO "$CONDA_EXE" config --system --set␣
↪solver libmamba
        fi

        conda_init_extra_args=()
        if [[ "$INSTALL_TYPE" == system ]]; then
            # if we're installing into a root-owned directory using sudo, or we
↪'re already root
            # initialize conda in the system-wide rcfiles
            conda_init_extra_args=(--no-user --system)
        fi
        # run conda-init and look at its output to insert 'conda activate
↪$CONDA_ENV_NAME' into the
        # block that conda-init will update if ever conda is installed to a␣
↪different prefix and
        # this is rerun.
        $SUDO "${CONDA_EXE}" init $DRY_RUN_OPTION "${conda_init_extra_args[@]}"
↪$CONDA_SHELL_TYPE 2>&1 | \
            tee >(grep '^modified' | grep -v "$CONDA_INSTALL_PREFIX" | awk '
↪{print $NF}' | \
            "${DRY_RUN_ECHO[@]}" $SUDO xargs -r sed -i -e "/<<< conda␣
↪initialize <<</iconda activate $CONDA_ENV_NAME")

        if [[ $REINSTALL_CONDA -eq 1 ]]; then
            echo "::INFO:: Done reinstalling conda. Exiting"
            exit 0
        fi
    fi

    # https://conda-forge.org/feedstock-outputs/
    #   filterable list of all conda-forge packages
    # https://conda-forge.org/#contribute
    #   instructions on adding a recipe
    # https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/pkg-
↪specs.html#package-match-specifications
```

(continues on next page)

(continued from previous page)

```
    #   documentation on package_spec syntax for constraining versions
    CONDA_PACKAGE_SPECS=()

    # minimal specs to allow cloning of firesim repo and access to the manager
    CONDA_PACKAGE_SPECS+=(
        bash-completion \
        ca-certificates \
        mosh \
        vim \
        git \
        screen \
        argcomplete \
        expect \
        "python>=3.8" \
        boto3 \
        pytz \
        mypy-boto3-s3 \
        mypy_boto3_ec2 \
        "azure-mgmt-resource>=18" \
        azure-identity \
        azure-mgmt-compute \
        azure-mgmt-network \
        fsspec \
        "s3fs==0.4.2" \
        "cryptography<41" \
    )

    if [[ "$CONDA_ENV_NAME" == "base" ]]; then
        # NOTE: arg parsing disallows installing to base but this logic is␣
↪correct if we ever change
        CONDA_SUBCOMMAND=install
        CONDA_ENV_BIN="${CONDA_INSTALL_PREFIX}/bin"
    else
        CONDA_ENV_BIN="${CONDA_INSTALL_PREFIX}/envs/${CONDA_ENV_NAME}/bin"
        if [[ -d "${CONDA_INSTALL_PREFIX}/envs/${CONDA_ENV_NAME}" ]]; then
            # 'create' clobbers the existing environment and doesn't leave a␣
↪revision entry in
            # `conda list --revisions`, so use install instead
            CONDA_SUBCOMMAND=install
        else
            CONDA_SUBCOMMAND=create
        fi
    fi

    # to enable use of sudo and avoid modifying 'secure_path' in /etc/sudoers,␣
↪we specify the full path to conda
    $SUDO "${CONDA_EXE}" "$CONDA_SUBCOMMAND" $DRY_RUN_OPTION -n "$CONDA_ENV_NAME
↪" -c conda-forge -y "${CONDA_PACKAGE_SPECS[@]}"

    # to enable use of sudo and avoid modifying 'secure_path' in /etc/sudoers,␣
↪we specify the full path to pip
    CONDA_PIP_EXE="${CONDA_ENV_BIN}/pip"
```

(continues on next page)

(continued from previous page)

```
    # Install python packages using pip that are not available from conda
    #
    # Installing things with pip is possible.  However, to get
    # the most complete solution to all dependencies, you should
    # prefer creating the environment with a single invocation of
    # conda
    PIP_PKGS=( \
        "fab-classic>=1.19.2" \
        azure-mgmt-resourcegraph \
    )
    if [[ -n "$PIP_PKGS[*]" ]]; then
        "${DRY_RUN_ECHO[@]}" $SUDO "${CONDA_PIP_EXE}" install "${PIP_PKGS[@]}"
    fi

    argcomplete_extra_args=()
    if [[ "$INSTALL_TYPE" == system ]]; then
        BASH_COMPLETION_COMPAT_DIR="${CONDA_ENV_BIN}/../etc/bash_completion.d"
        "${DRY_RUN_ECHO[@]}" $SUDO mkdir -p "${BASH_COMPLETION_COMPAT_DIR}"
        argcomplete_extra_args=( --dest "${BASH_COMPLETION_COMPAT_DIR}" )

    else
        # if we aren't installing into a system directory, then initialize␣
↪argcomplete
        # with --user so that it goes into the home directory
        argcomplete_extra_args=( --user )
    fi
    set +o pipefail
    "${DRY_RUN_ECHO[@]}" yes | $SUDO "${CONDA_ENV_BIN}/activate-global-python-
↪argcomplete" "${argcomplete_extra_args[@]}"
    set -o pipefail

    # emergency fix for buildroot open files limit issue:
    if [[ "$INSTALL_TYPE" == system ]]; then
        "${DRY_RUN_ECHO[@]}" echo "* hard nofile 16384" | $SUDO tee --append /
↪etc/security/limits.conf
    else
        "${DRY_RUN_ECHO[@]}" echo "::WARN:: Unable to set open files limit␣
↪without sudo."
    fi

    # final platform-specific setup
    case "$OS_FLAVOR" in
        ubuntu)
            ;;
        centos)
            ;;
        amzn)
            echo "::INFO:: using 'sudo' to install NICE DCV"
            wget https://raw.githubusercontent.com/aws-samples/amazon-ec2-nice-
↪dcv-samples/5439e401d3aaf394588f1029e0ec7904d8cacc8f/scripts/AmazonLinux2-
↪user-data.sh
```

(continues on next page)

(continued from previous page)

```
            chmod +x AmazonLinux2-user-data.sh
            sudo ./AmazonLinux2-user-data.sh
            echo "firesim" | sudo passwd ec2-user --stdin # default password is
↪'firesim'
            ;;
        debian)
            ;;
        *)
            echo "::ERROR:: Unknown OS flavor '$OS_FLAVOR'. Unable to do␣
↪platform-specific setup."
            exit 1
            ;;
    esac

} 2>&1 | tee "$MACHINE_LAUNCH_DIR/machine-launchstatus.log"
chmod ugo+r "$MACHINE_LAUNCH_DIR/machine-launchstatus.log"

echo "machine launch script completed" >> "$MACHINE_LAUNCH_DIR/machine-
↪launchstatus"
```

When your instance boots, this will install a compatible set of all the dependencies needed to run FireSim on your instance using Conda.

9. Double check your configuration. The most common misconfigurations that may require repeating this process include:

   1. Not selecting the `firesim` vpc.
   2. Not selecting the `firesim` security group.
   3. Not selecting the `firesim` key pair.
   4. Selecting the wrong AMI.

10. Click the orange *Launch Instance* button.

> **Warning**
>
> Recently, some AWS users been having issues with the launch process (after you click `Launch Instance`) getting stuck trying to "Subscribe" to the AMI even when the account is already subscribed. We have been able to bypass this issue by going to the FPGA Developer AMI page on AWS Marketplace, clicking subscribe (even if already subscribed), then clicking "Continue to Configuration", then verify the correct AMI version and region are selected and click "Continue to Launch". Finally, change the dropdown that says "Launch from Website" to "Launch through EC2" and click "Launch". At this point, you will be brought back to the usual launch instance page, but the AMI will be pre-selected and you will be able to successfully launch at the end, after updating the rest of the options as noted above.

Access your instance

We **HIGHLY** recommend using mosh instead of `ssh` or using `ssh` with a screen/tmux session running on your manager instance to ensure that long-running jobs are not killed by a bad network connection to your manager instance. On this instance, the `mosh` server is installed as part of the setup script we pasted before, so we need to first ssh into the instance and make sure the setup is complete.

In either case, `ssh` into your instance (e.g. `ssh -i firesim.pem centos@YOUR_INSTANCE_IP`) and wait until the `/tmp/machine-launchstatus` file contains all the following text:

```
$ cat /tmp/machine-launchstatus
machine launch script started
machine launch script completed
```

You can also view the live output of the installation process by running `tail -f /tmp/machine-launchstatus.log`.

Once `machine launch script completed` appears in `/tmp/machine-launchstatus`, exit and re-`ssh` into the system. If you want to use `mosh`, `mosh` back into the system.

Key Setup, Part 2

Now that our manager instance is started, copy the private key that you downloaded from AWS earlier (`firesim.pem`) to `~/firesim.pem` on your manager instance. This step is required to give the manager access to the instances it launches for you.

CHAPTER

FOUR

LOCAL FPGA SYSTEM SETUP

The below sections outline what you need to install to run FireSim on each machine type in a FireSim cluster. **Note that the below three machine types can all map to a single machine in your setup**; in this case, you should follow all the installation instructions on your single machine, without duplication (i.e., don't re-run a step on the same machine if it is required on multiple machine types).

Warning

We highly recommend using Ubuntu 20.04 LTS as the host operating system for all machine types in an on-premises setup, as this is the OS recommended by Xilinx.

The following steps are separated into steps that require `sudo` and steps that do not. After initial setup with `sudo`, FireSim doesn't need `sudo` access. In many cases with a shared machine, `sudo`-based setup is already completed and thus users should continue onto the Non-`sudo`-based setup.

4.1 sudo Setup

1. Install/enable FireSim scripts to new `firesim` Linux group

Note

These scripts are used by the FireSim manager and other FireSim tooling (i.e. FireMarshal) to avoid needing `sudo` access.

Machines: Manager Machine, Run Farm Machines, Build Farm Machines.

First, let's clone a temporary version of FireSim with the scripts within it:

```
cd ~/     # or any scratch directory
mkdir firesim-script-installs
cd firesim-script-installs
git clone https://github.com/firesim/firesim
cd firesim
# checkout latest official firesim release
# note: this may not be the latest release if the documentation version != "stable"
git checkout t
```

Next, copy the required scripts to `/usr/local/bin`:

```
sudo cp deploy/sudo-scripts/* /usr/local/bin
sudo cp platforms/xilinx_alveo_u250/scripts/* /usr/local/bin
```

Now we can delete the temporary clone:

```
rm -rf ~/firesim-script-installs    # or the temp. dir. created previously
```

Next, lets change the permissions of the scripts and them to a new `firesim` Linux group.

```
sudo addgroup firesim
sudo chmod 755 /usr/local/bin/firesim*
sudo chgrp firesim /usr/local/bin/firesim*
```

Next, lets allow the `firesim` Linux group to run the pre-installed commands. Enter/create the following file with *sudo*:

```
sudo visudo /etc/sudoers.d/firesim
```

Then add the following lines:

```
%firesim ALL=(ALL) NOPASSWD: /usr/local/bin/firesim-*
```

Then change the permissions of the file:

```
sudo chmod 400 /etc/sudoers.d/firesim
```

This allows only users in the `firesim` group to execute the scripts.

2. Add your user to the `firesim` group

Machines: Manager Machine, Run Farm Machines, Build Farm Machines.

Next, add all user who want to use FireSim to the `firesim` group that you created. Make sure to replace `YOUR_USER_NAME` with the user to run simulations with:

```
sudo usermod -a -G firesim YOUR_USER_NAME
```

Finally, verify that the user can access the FireSim installed scripts by running:

```
sudo -l
```

The output should look similar to this:

```
User YOUR_USER_NAME may run the following commands on MACHINE_NAME:
    (ALL) NOPASSWD: /usr/local/bin/firesim-*
```

3. Install Vivado Lab and Cable Drivers

Machines: Run Farm Machines.

Go to the Xilinx Downloads Website and download `Vivado 2023.1: Lab Edition - Linux`.

Extract the downloaded `.tar.gz` file, then:

```
cd [EXTRACTED VIVADO LAB DIRECTORY]
sudo ./installLibs.sh
sudo ./xsetup --batch Install --agree XilinxEULA,3rdPartyEULA --edition "Vivado Lab␣
↪Edition (Standalone)"
```

This will have installed Vivado Lab to /tools/Xilinx/Vivado_Lab/2023.1/.

For ease of use, add the following to the end of your ~/.bashrc:

```
source /tools/Xilinx/Vivado_Lab/2023.1/settings64.sh
```

Then, open a new terminal or source your ~/.bashrc.

Next, install the cable drivers like so:

```
cd /tools/Xilinx/Vivado_Lab/2023.1/data/xicom/cable_drivers/lin64/install_script/install_
→drivers/
sudo ./install_drivers
```

4. Install the Xilinx XDMA and XVSEC drivers

Machines: Run Farm Machines.

Warning

These commands will need to be re-run everytime the kernel is updated (normally whenever the machine is rebooted).

First, run the following to clone the XDMA kernel module source:

```
cd ~/   # or any directory you would like to work from
git clone https://github.com/Xilinx/dma_ip_drivers
cd dma_ip_drivers
git checkout 0e8d321
cd XDMA/linux-kernel/xdma
```

Note

If using the RHS Research Nitefury II board, do the following: The directory you are now in contains the XDMA kernel module. For the Nitefury to work, we will need to make one modification to the driver. Find the line containing #define XDMA_ENGINE_XFER_MAX_DESC. Change the value on this line from 0x800 to 16. Then, build and install the driver:

```
sudo make install
```

Now, test that the module can be inserted:

```
sudo insmod $(find /lib/modules/$(uname -r) -name "xdma.ko") poll_mode=1
lsmod | grep -i xdma
```

The second command above should have produced output indicating that the XDMA driver is loaded.

Next, we will do the same for the XVSEC driver, which is pulled from a separate repository due to kernel version incompatibility:

```
cd ~/   # or any directory you would like to work from
git clone https://github.com/paulmnt/dma_ip_drivers dma_ip_drivers_xvsec
cd dma_ip_drivers_xvsec
```

(continues on next page)

(continued from previous page)

```
git checkout 302856a
cd XVSEC/linux-kernel/

make clean all
sudo make install
```

Now, test that the module can be inserted:

```
sudo modprobe xvsec
lsmod | grep -i xvsec
```

The second command above should have produced output indicating that the XVSEC driver is loaded.

Also, make sure you get output for the following (usually, `/usr/local/sbin/xvsecctl`):

```
which xvsecctl
```

5. Install sshd

Machines: Manager Machine, Run Farm Machines, and Build Farm Machines

On Ubuntu, install `openssh-server` like so:

```
sudo apt install openssh-server
```

7. Check Hard File Limit

Machine: Manager Machine

Check the output of the following command:

```
ulimit -Hn
```

If the result is greater than or equal to 16384, you can continue. Otherwise, run:

```
echo "* hard nofile 16384" | sudo tee --append /etc/security/limits.conf
```

Then, reboot your machine.

8. Install your FPGA

The starter tutorials will guide you through specific installation instructions for each FPGA.

4.2 Non-sudo Setup

1. Fix default `.bashrc`

Machines: Manager Machine, Run Farm Machines, Build Farm Machines.

We need various parts of the `~/.bashrc` file to execute even in non-interactive mode. To do so, edit your `~/.bashrc` file so that the following section is removed:

```
# If not running interactively, don't do anything
case $- in
     *i*) ;;
       *) return;;
esac
```

2. Set up SSH Keys

Machines: Manager Machine.

On the manager machine, generate a keypair that you will use to ssh from the manager machine into the manager machine (ssh localhost), run farm machines, and build farm machines:

```
cd ~/.ssh
ssh-keygen -t ed25519 -C "firesim.pem" -f firesim.pem
[create passphrase]
```

Next, add this key to the `authorized_keys` file on the manager machine:

```
cd ~/.ssh
cat firesim.pem.pub >> authorized_keys
chmod 0600 authorized_keys
```

You should also copy this public key into the `~/.ssh/authorized_keys` files on all of your Run Farm and Build Farm Machines.

Returning to the Manager Machine, let's set up an `ssh-agent`:

```
cd ~/.ssh
ssh-agent -s > AGENT_VARS
source AGENT_VARS
ssh-add firesim.pem
```

If you reboot your machine (or otherwise kill the `ssh-agent`), you will need to re-run the above four commands before using FireSim. If you open a new terminal (and `ssh-agent` is already running), you can simply run `source ~/.ssh/AGENT_VARS`.

Finally, confirm that you can now `ssh localhost` and ssh into your Run Farm and Build Farm Machines without being prompted for a passphrase.

3. Verify Run Farm Machine environment

Machines: Manager Machine and Run Farm Machines

Finally, let's ensure that the Xilinx Vivado Lab tools are properly sourced in your shell setup (i.e. `.bashrc`) so that any shell on your Run Farm Machines can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned.

You can check this by running the following on the Manager Machine, replacing `RUN_FARM_IP` with `localhost` if your Run Farm machine and Manager machine are the same machine, or replacing it with the Run Farm machine's IP address if they are different machines.

```
ssh RUN_FARM_IP printenv
```

Ensure that the output of the command shows that the Xilinx Vivado Lab tools are present in the printed environment variables (i.e., `PATH`).

If you have multiple Run Farm machines, you should repeat this process for each Run Farm machine, replacing `RUN_FARM_IP` with a different Run Farm Machine's IP address.

Congratulations! We've now set up your machine/cluster to run simulations.

CHAPTER

FIVE

AWS EC2 F1 GETTING STARTED GUIDE

The getting started guides that follow this page will guide you through the complete flow for getting an example Chipyard-based SoC FireSim simulation up and running using AWS EC2 F1. At the end of this guide, you'll have a simulation that simulates a single quad-core Rocket Chip-based node with a 4 MB last level cache, 16 GB DDR3, and no NIC. After this, you can continue to a guide that shows you how to simulate a globally-cycle-accurate cluster-scale FireSim simulation. The final guide will show you how to build your own FPGA images with customized hardware. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Make sure you have run/done the steps listed in *AWS EC2 F1 System Setup* before running this guide.

Here's a high-level outline of what we'll be doing in our AWS EC2 F1 getting started guides:

1. **Setting up the FireSim repo**: Cloning the repository needed for this guide.
2. **Single-node simulation guide**: This guide walks you through the process of running one simulation on a Run Farm consisting of a single `f1.2xlarge`, using Chipyard's pre-built public AGFIs.
3. **Cluster simulation guide**: This guide walks you through the process of running an 8-node cluster simulation on a Run Farm consisting of one `f1.16xlarge`, using Chipyard's pre-built public AGFIs and switch models.
4. **Building your own hardware designs guide (Chisel to FPGA Image)**: This guide walks you through the full process of taking Rocket Chip RTL and any custom RTL plugged into Rocket Chip and producing a FireSim AGFI to plug into your simulations. This automatically runs Chisel elaboration, FAME-1 Transformation, and the Vivado FPGA flow.

Generally speaking, you only need to follow step 4 if you're modifying Chisel RTL or changing non-runtime configurable hardware parameters.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

5.1 Setting up the FireSim Repo

Lets fetch FireSim's sources with Chipyard. Chipyard provides all the necessary target designs (e.g. RISC-V SoCs) and software (e.g. Linux) used for the rest of this guide.

Note

This guide was built using Chipyard version dbc082e2206f787c3aba12b9b171e1704e15b707. It is recommended to use the most up-to-date version of Chipyard with this tutorial.

Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
./build-setup.sh
```

This will have initialized submodules and installed the RISC-V tools and other dependencies.

Next, run:

```
cd sims/firesim
source sourceme-manager.sh
```

This will have initialized the AWS shell, added the RISC-V tools to your path, and started an `ssh-agent` that supplies `~/firesim.pem` automatically when you use `ssh` to access other nodes. Sourcing this the first time will take some time – however each time after that should be instantaneous. Also, if your `firesim.pem` key requires a passphrase, you will be asked for it here and `ssh-agent` should cache it.

Every time you login to your manager instance to use FireSim, you should `cd` into your firesim directory and source this file again.

5.2 Completing Setup Using the Manager

The FireSim manager contains a command that will interactively guide you through the rest of the FireSim setup process. To run it, do the following:

```
firesim managerinit --platform f1
```

This will first prompt you to setup AWS credentials on the instance, which allows the manager to automatically manage build/simulation nodes. You can use the same AWS access key you created when running setup commands on the `t2.nano` instance earlier (in *Run scripts from the t2.nano*). When prompted, you should specify the same region that you've been selecting thus far (one of `us-east-1`, `us-west-2`, `ap-southeast-2`, `eu-central-1`, `eu-west-1` or `eu-west-2`) and set the default output format to `json`.

Next, it will prompt you for an email address, which is used to send email notifications upon FPGA build completion and optionally for workload completion. You can leave this blank if you do not wish to receive any notifications, but this is not recommended. Next, it will create initial configuration files, which we will edit in later sections.

Now you're ready to launch FireSim simulations! Hit Next to learn how to run single-node simulations.

5.3 Running FireSim Simulations

These guides will walk you through running two kinds of simulations:

- First, we will simulate a single-node, non-networked target, using a pre-generated hardware image.
- Then, we will simulate an eight-node, networked cluster target, also using a pre-generated hardware image.

Hit next to get started!

5.3.1 Running a Single Node Simulation

Now that we've completed the setup of our manager instance, it's time to run a simulation! In this section, we will simulate **1 target node**, for which we will need a single `f1.2xlarge` (1 FPGA) instance.

Make sure you are `ssh` or `mosh`'d into your manager instance and have sourced `sourceme-manager.sh` before running any of these commands.

Building target software

In these instructions, we'll assume that you want to boot Linux on your simulated node. To do so, we'll need to build our FireSim-compatible RISC-V Linux distro. For this guide, we will use a simple buildroot-based distribution. You can do this like so:

```
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

This process will take about 10 to 15 minutes on a `c5.4xlarge` instance. Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the nodes we will simulate.
- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for each the nodes we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or to copy around for deploying.

Setting up the manager configuration

All runtime configuration options for the manager are set in a file called `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

If you open up this file, you will see the following default config (assuming you have not modified it):

```
# RUNTIME configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

run_farm:
  base_recipe: run-farm-recipes/aws_ec2.yaml
  recipe_arg_overrides:
    # tag to apply to run farm hosts
    run_farm_tag: mainrunfarm
    # enable expanding run farm by run_farm_hosts given
    always_expand_run_farm: true
    # minutes to retry attempting to request instances
    launch_instances_timeout_minutes: 60
    # run farm host market to use (ondemand, spot)
    run_instance_market: ondemand
    # if using spot instances, determine the interrupt behavior (terminate, stop,␣
↪hibernate)
```

(continues on next page)

(continued from previous page)

```
    spot_interruption_behavior: terminate
    # if using spot instances, determine the max price
    spot_max_price: ondemand
    # default location of the simulation directory on the run farm host
    default_simulation_dir: /home/centos

    # run farm hosts to spawn: a mapping from a spec below (which is an EC2
    # instance type) to the number of instances of the given type that you
    # want in your runfarm.
    run_farm_hosts_to_use:
      - f1.16xlarge: 0
      - f1.4xlarge: 0
      - f1.2xlarge: 1
      - m4.16xlarge: 0
      - z1d.3xlarge: 0
      - z1d.6xlarge: 0
      - z1d.12xlarge: 0

metasimulation:
  metasimulation_enabled: false
  # vcs or verilator. use vcs-debug or verilator-debug for waveform generation
  metasimulation_host_simulator: verilator
  # plusargs passed to the simulator for all metasimulations
  metasimulation_only_plusargs: "+fesvr-step-size=128 +max-cycles=100000000"
  # plusargs passed to the simulator ONLY FOR vcs metasimulations
  metasimulation_only_vcs_plusargs: "+vcs+initreg+0 +vcs+initmem+0"

target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1

    # This references a section from config_hwdb.yaml for fpga-accelerated simulation
    # or from config_build_recipes.yaml for metasimulation
    # In homogeneous configurations, use this to set the hardware config deployed
    # for all simulators
    default_hw_config: midasexamples_gcd

    # Advanced: Specify any extra plusargs you would like to provide when
    # booting the simulator (in both FPGA-sim and metasim modes). This is
    # a string, with the contents formatted as if you were passing the plusargs
    # at command line, e.g. "+a=1 +b=2"
    plusarg_passthrough: ""

tracing:
    enable: no

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
```

(continues on next page)

(continued from previous page)

```
    # unwinding -> Flame Graph)
    output_format: 0

    # Trigger selector.
    # 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
    # instruction trigger
    selector: 1
    start: 0
    end: -1

autocounter:
    read_rate: 0

workload:
    workload_name: null.json
    terminate_on_completion: no
    suffix_tag: null

host_debug:
    # When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
    # begin (takes 2-5 minutes).
    # In general, this is not required to produce deterministic simulations on
    # target machines running linux. Enable if you observe simulation non-determinism.
    zero_out_dram: no
    # If disable_synth_asserts: no, simulation will print assertion message and
    # terminate simulation if synthesized assertion fires.
    # If disable_synth_asserts: yes, simulation ignores assertion firing and
    # continues simulation.
    disable_synth_asserts: no

# DOCREF START: Synthesized Prints
synth_print:
    # Start and end cycles for outputting synthesized prints.
    # They are given in terms of the base clock and will be converted
    # for each clock domain.
    start: 0
    end: -1
    # When enabled (=yes), prefix print output with the target cycle at which the print␣
↪was triggered
    cycle_prefix: yes
# DOCREF END: Synthesized Prints
```

We won't have to modify any of the defaults for this single-node simulation guide, but let's walk through several of the key parts of the file.

First, let's see how the correct numbers and types of instances are specified to the manager:

- You'll notice first that in the `run_farm` mapping, the manager is configured to launch a Run Farm named `mainrunfarm` (given by the `run_farm_tag`). The tag specified here allows the manager to differentiate amongst many parallel run farms (each running some workload on some target design) that you may be operating. In this case, the default is fine since we're only running a single run farm.
- Notice that under `run_farm_hosts_to_use`, the only non-zero value is for `f1.2xlarge`, which should be set to 1. This is exactly what we'll need for this guide.

- You'll see other parameters in the `run_farm` mapping, like `run_instance_market`, `spot_interruption_behavior`, and `spot_max_price`. If you're an experienced AWS user, you can see what these do by looking at the *Manager Configuration Files* section. Otherwise, don't change them.

Next, let's look at how the target design is specified to the manager. This is located in the `target_config` section of `firesim/deploy/config_runtime.yaml`, shown below:

```
target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1

    # This references a section from config_hwdb.yaml for fpga-accelerated simulation
    # or from config_build_recipes.yaml for metasimulation
    # In homogeneous configurations, use this to set the hardware config deployed
    # for all simulators
    default_hw_config: midasexamples_gcd

    # Advanced: Specify any extra plusargs you would like to provide when
    # booting the simulator (in both FPGA-sim and metasim modes). This is
    # a string, with the contents formatted as if you were passing the plusargs
    # at command line, e.g. "+a=1 +b=2"
    plusarg_passthrough: ""
```

Here are some highlights of this section:

- `topology` is set to `no_net_config`, indicating that we do not want a network.
- `no_net_num_nodes` is set to 1, indicating that we only want to simulate one node.
- `default_hw_config` is `midasexamples_gcd`. This references a bitstream/build-recipe used to run a simulation specified in `${FS_DIR}/deploy/config_hwdb.yaml`.

Let's modify the `default_hw_config` to instead point to a Chipyard SoC publically available AWS FPGA image. Change the following:

```
default_hw_config: firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3
```

This references a pre-built, publically-available AWS FPGA Image that is specified in ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml. This pre-built image models a Quad-core Rocket Chip with 4 MB of L2 cache and 16 GB of DDR3, and no network interface card. **Future steps will require us to point to this Chipyard HWDB YAML file so that the FireSim manager can obtain the hardware configuration.**

> **Attention**
>
> **[Advanced users] Simulating BOOM instead of Rocket Chip**: If you would like to simulate a single-core BOOM as a target, set `default_hw_config` to `firesim_boom_singlecore_no_nic_l2_llc4mb_ddr3`.

Finally, let's take a look at the `workload` section, which defines the target software that we'd like to run on the simulated target design. By default, it should look like this:

```
workload:
    workload_name: null.json
    terminate_on_completion: no
    suffix_tag: null
```

Let's modify the `null.json` workload name to point to a workload definition that will boot Linux. Change the following:

```
workload_name: br-base-uniform.json
```

This tells the FireSim manager to run the specified buildroot-based Linux (`br-base-uniform.json`) on our simulated system. The `terminate_on_completion` feature is an advanced feature that you can learn more about in the *Manager Configuration Files* section.

Launching a Simulation!

Now that we've told the manager everything it needs to know in order to run our single-node simulation, let's actually launch an instance and run it!

Starting the Run Farm

First, we will tell the manager to launch our Run Farm, as we specified above. When you do this, you will start getting charged for the running EC2 instances (in addition to your manager). As mentioned earlier we need to point to Chipyard's HWDB file that holds the reference to `firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3`.

To do launch your run farm, run:

```
firesim launchrunfarm -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

You should expect output like the following:

```
FireSim Manager. Docs: http://docs.fires.im
Running: launchrunfarm

Waiting for instance boots: f1.16xlarges
Waiting for instance boots: f1.4xlarges
Waiting for instance boots: m4.16xlarges
Waiting for instance boots: f1.2xlarges
i-0d6c29ac507139163 booted!
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-19-43-launchrunfarm-B4Q2ROAK0JN9EDE4.
↪log
```

The output will rapidly progress to `Waiting for instance boots:  f1.2xlarges` and then take a minute or two while your `f1.2xlarge` instance launches. Once the launches complete, you should see the instance id printed and the instance will also be visible in your AWS EC2 Management console. The manager will tag the instances launched with this operation with the value you specified above as the `run_farm_tag` parameter from the `config_runtime.yaml` file, which we left set as `mainrunfarm`. This value allows the manager to tell multiple Run Farms apart – i.e., you can have multiple independent Run Farms running different workloads/hardware configurations in parallel. This is detailed in the *Manager Configuration Files* and the *firesim launchrunfarm* sections – you do not need to be familiar with it here.

Setting up the simulation infrastructure

The manager will also take care of building and deploying all software components necessary to run your simulation. The manager will also handle programming FPGAs. To tell the manager to set up our simulation infrastructure, let's run:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: http://docs.fires.im
Running: infrasetup

Building FPGA software driver for FireSim-FireSimQuadRocketConfig-BaseF1Config
[172.30.2.174] Executing task 'instance_liveness'
[172.30.2.174] Checking if host instance is up...
[172.30.2.174] Executing task 'infrasetup_node_wrapper'
[172.30.2.174] Copying FPGA simulation infrastructure for slot: 0.
[172.30.2.174] Installing AWS FPGA SDK on remote nodes.
[172.30.2.174] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.174] Copying AWS FPGA XDMA driver to remote node.
[172.30.2.174] Loading XDMA Driver Kernel Module.
[172.30.2.174] Clearing FPGA Slot 0.
[172.30.2.174] Flashing FPGA Slot: 0 with agfi: agfi-0eaa90f6bb893c0f7.
[172.30.2.174] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.174] Loading XDMA Driver Kernel Module.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-32-02-infrasetup-9DJJCX29PF4GAIVL.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `firesim/deploy/logs/`.

At this point, the `f1.2xlarge` instance in our Run Farm has all the infrastructure necessary to run a simulation.

So, let's launch our simulation!

Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: http://docs.fires.im
Running: runworkload

Creating the directory: /home/centos/firesim-new/deploy/results-workload/2018-05-19--00-
↪38-52-br-base/
[172.30.2.174] Executing task 'instance_liveness'
```

(continues on next page)

(continued from previous page)

```
[172.30.2.174] Checking if host instance is up...
[172.30.2.174] Executing task 'boot_simulation_wrapper'
[172.30.2.174] Starting FPGA simulation for slot: 0.
[172.30.2.174] Executing task 'monitor_jobs_wrapper'
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.
↪log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have shut down. So, let's let it run and open another ssh connection to the manager instance. From there, `cd` into your firesim directory again and `source sourceme-manager.sh` again to get our ssh key set up. To access our simulated system, ssh into the IP address being printed by the status page, **from your manager instance**. In our case, from the above output, we see that our simulated system is running on the instance with IP `172.30.2.174`. So, run:

```
[RUN THIS ON YOUR MANAGER INSTANCE!]
ssh 172.30.2.174
```

This will log you into the instance running the simulation. Then, to attach to the console of the simulated system, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
```

(continues on next page)

(continued from previous page)

```
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's poweroff the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.
↪log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.
↪log
```

If you take a look at the workload output directory given in the manager output (in this case, `/home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/results-workload/
↪2018-05-19--00-38-52-br-base$ ls -la */*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back to our manager after we run a simulation, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

For now, let's wrap-up our guide by terminating the `f1.2xlarge` instance that we launched. To do so, run:

```
firesim terminaterunfarm -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r $
↪{CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

Which should present you with the following:

```
FireSim Manager. Docs: http://docs.fires.im
Running: terminaterunfarm
```

(continues on next page)

(continued from previous page)

```
IMPORTANT!: This will terminate the following instances:
f1.16xlarges
[]
f1.4xlarges
[]
m4.16xlarges
[]
f1.2xlarges
['i-0d6c29ac507139163']
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
```

You must type yes then hit enter here to have your instances terminated. Once you do so, you will see:

```
[ truncated output from above ]
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
yes
Instances terminated. Please confirm in your AWS Management Console.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--00-51-54-terminaterunfarm-
↪T9ZAED3LJUQQ3K0N.log
```

At this point, you should always confirm in your AWS management console that the instance is in the shutting-down or terminated states. You are ultimately responsible for ensuring that your instances are terminated appropriately.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left, or you can continue on with the cluster simulation guide.

5.3.2 Running a Cluster Simulation

Now, let's move on to simulating a cluster of eight nodes, interconnected by a network with one 8-port Top-of-Rack (ToR) switch and 200 Gbps, 2s links. This will require one `f1.16xlarge` (8 FPGA) instance.

Make sure you are `ssh` or `mosh`'d into your manager instance and have sourced `sourceme-manager.sh` before running any of these commands.

Building target software

If you already built target software during the single-node getting started guide, you can skip to the next part (Setting up the manager configuration). If you haven't followed the single-node getting started guide, continue with this section.

In these instructions, we'll assume that you want to boot the buildroot-based Linux distribution on each of the nodes in your simulated cluster. To do so, we'll need to build our FireSim-compatible RISC-V Linux distro. You can do this like so:

```
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

This process will take about 10 to 15 minutes on a `c5.4xlarge` instance. Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the nodes we will simulate.
- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for each the nodes we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or to copy around for deploying.

Setting up the manager configuration

All runtime configuration options for the manager are set in a file called `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

If you open up this file, you will see the following default config (assuming you have not modified it):

```
# RUNTIME configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

run_farm:
  base_recipe: run-farm-recipes/aws_ec2.yaml
  recipe_arg_overrides:
    # tag to apply to run farm hosts
    run_farm_tag: mainrunfarm
    # enable expanding run farm by run_farm_hosts given
    always_expand_run_farm: true
    # minutes to retry attempting to request instances
    launch_instances_timeout_minutes: 60
    # run farm host market to use (ondemand, spot)
    run_instance_market: ondemand
    # if using spot instances, determine the interrupt behavior (terminate, stop,␣
↪hibernate)
    spot_interruption_behavior: terminate
    # if using spot instances, determine the max price
    spot_max_price: ondemand
    # default location of the simulation directory on the run farm host
    default_simulation_dir: /home/centos

    # run farm hosts to spawn: a mapping from a spec below (which is an EC2
    # instance type) to the number of instances of the given type that you
    # want in your runfarm.
    run_farm_hosts_to_use:
      - f1.16xlarge: 0
      - f1.4xlarge: 0
      - f1.2xlarge: 1
      - m4.16xlarge: 0
      - z1d.3xlarge: 0
      - z1d.6xlarge: 0
      - z1d.12xlarge: 0

metasimulation:
  metasimulation_enabled: false
```

(continues on next page)

(continued from previous page)

```
  # vcs or verilator. use vcs-debug or verilator-debug for waveform generation
  metasimulation_host_simulator: verilator
  # plusargs passed to the simulator for all metasimulations
  metasimulation_only_plusargs: "+fesvr-step-size=128 +max-cycles=100000000"
  # plusargs passed to the simulator ONLY FOR vcs metasimulations
  metasimulation_only_vcs_plusargs: "+vcs+initreg+0 +vcs+initmem+0"

target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1

    # This references a section from config_hwdb.yaml for fpga-accelerated simulation
    # or from config_build_recipes.yaml for metasimulation
    # In homogeneous configurations, use this to set the hardware config deployed
    # for all simulators
    default_hw_config: midasexamples_gcd

    # Advanced: Specify any extra plusargs you would like to provide when
    # booting the simulator (in both FPGA-sim and metasim modes). This is
    # a string, with the contents formatted as if you were passing the plusargs
    # at command line, e.g. "+a=1 +b=2"
    plusarg_passthrough: ""

tracing:
    enable: no

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
    # unwinding -> Flame Graph)
    output_format: 0

    # Trigger selector.
    # 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
    # instruction trigger
    selector: 1
    start: 0
    end: -1

autocounter:
    read_rate: 0

workload:
    workload_name: null.json
    terminate_on_completion: no
    suffix_tag: null

host_debug:
    # When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
```

(continues on next page)

(continued from previous page)

```
    # begin (takes 2-5 minutes).
    # In general, this is not required to produce deterministic simulations on
    # target machines running linux. Enable if you observe simulation non-determinism.
    zero_out_dram: no
    # If disable_synth_asserts: no, simulation will print assertion message and
    # terminate simulation if synthesized assertion fires.
    # If disable_synth_asserts: yes, simulation ignores assertion firing and
    # continues simulation.
    disable_synth_asserts: no

# DOCREF START: Synthesized Prints
synth_print:
    # Start and end cycles for outputting synthesized prints.
    # They are given in terms of the base clock and will be converted
    # for each clock domain.
    start: 0
    end: -1
    # When enabled (=yes), prefix print output with the target cycle at which the print␣
↪was triggered
    cycle_prefix: yes
# DOCREF END: Synthesized Prints
```

For the 8-node cluster simulation, the defaults in this file are close to what we want but require slight modification. Let's outline the important parameters we need to change:

- `f1.16xlarges:`: Change this parameter to 1. This tells the manager that we want to launch one `f1.16xlarge` when we call the `launchrunfarm` command.
- `f1.2xlarges:`: Change this parameter to `0`. This tells the manager to not launch any `f1.2xlarge` machines when we call the `launchrunfarm` command.
- `topology:`: Change this parameter to `example_8config`. This tells the manager to use the topology named `example_8config` which is defined in `deploy/runtools/user_topology.py`. This topology simulates an 8-node cluster with one ToR switch.
- `default_hw_config:` Change this parameter to `firesim_rocket_quadcore_nic_l2_llc4mb_ddr3`. This tells the manager that we want to simulate a quad-core Rocket Chip configuration with 512 KB of L2, 4 MB of L3 (LLC), 16 GB of DDR3, and a NIC, for each of the simulated nodes in the topology.

Attention

[Advanced users] Simulating BOOM instead of Rocket Chip: If you would like to simulate a single-core BOOM as a target, set `default_hw_config` to `firesim_boom_singlecore_nic_l2_llc4mb_ddr3`.

There are also some parameters that we won't need to change, but are worth highlighting:

- `link_latency: 6405`: This models a network with 6405 cycles of link latency. Since we are modeling processors running at 3.2 Ghz, 1 cycle = 1/3.2 ns, so 6405 cycles is roughly 2 microseconds.
- `switching_latency: 10`: This models switches with a minimum port-to-port latency of 10 cycles.
- `net_bandwidth: 200`: This sets the bandwidth of the NICs to 200 Gbit/s. Currently you can set any integer value less than this without making hardware modifications.

You'll see other parameters here, like `run_instance_market`, `spot_interruption_behavior`, and `spot_max_price`. If you're an experienced AWS user, you can see what these do by looking at the *Manager*

Configuration Files section. Otherwise, don't change them.

As in the single-node getting started guide, we will leave the `workload:` mapping unchanged here, since we want to run the default buildroot-based Linux on our simulated system. The `terminate_on_completion` feature is an advanced feature that you can learn more about in the *Manager Configuration Files* section.

Launching a Simulation!

Now that we've told the manager everything it needs to know in order to run our single-node simulation, let's actually launch an instance and run it!

Starting the Run Farm

First, we will tell the manager to launch our Run Farm, as we specified above. When you do this, you will start getting charged for the running EC2 instances (in addition to your manager).

To do launch your run farm, run:

```
firesim launchrunfarm -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

You should expect output like the following:

```
FireSim Manager. Docs: http://docs.fires.im
Running: launchrunfarm

Waiting for instance boots: f1.16xlarges
i-09e5491cce4d5f92d booted!
Waiting for instance boots: f1.4xlarges
Waiting for instance boots: m4.16xlarges
Waiting for instance boots: f1.2xlarges
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-05-53-launchrunfarm-ZGVP753DSU1Y9Q6R.
↪log
```

The output will rapidly progress to `Waiting for instance boots: f1.16xlarges` and then take a minute or two while your `f1.16xlarge` instance launches. Once the launches complete, you should see the instance id printed and the instance will also be visible in your AWS EC2 Management console. The manager will tag the instances launched with this operation with the value you specified above as the `run_farm_tag` parameter from the `config_runtime.yaml` file, which we left set as `mainrunfarm`. This value allows the manager to tell multiple Run Farms apart – i.e., you can have multiple independent Run Farms running different workloads/hardware configurations in parallel. This is detailed in the *Manager Configuration Files* and the *firesim launchrunfarm* sections – you do not need to be familiar with it here.

Setting up the simulation infrastructure

The manager will also take care of building and deploying all software components necessary to run your simulation (including switches for the networked case). The manager will also handle programming FPGAs. To tell the manager to set up our simulation infrastructure, let's run:

```
firesim infrasetup -a ${CY_DIR}sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}/
↪sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: http://docs.fires.im
Running: infrasetup

Building FPGA software driver for FireSim-FireSimQuadRocketConfig-BaseF1Config
Building switch model binary for switch switch0
[172.30.2.178] Executing task 'instance_liveness'
[172.30.2.178] Checking if host instance is up...
[172.30.2.178] Executing task 'infrasetup_node_wrapper'
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 0.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 1.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 2.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 3.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 4.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 5.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 6.
[172.30.2.178] Copying FPGA simulation infrastructure for slot: 7.
[172.30.2.178] Installing AWS FPGA SDK on remote nodes.
[172.30.2.178] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.178] Copying AWS FPGA XDMA driver to remote node.
[172.30.2.178] Loading XDMA Driver Kernel Module.
[172.30.2.178] Clearing FPGA Slot 0.
[172.30.2.178] Clearing FPGA Slot 1.
[172.30.2.178] Clearing FPGA Slot 2.
[172.30.2.178] Clearing FPGA Slot 3.
[172.30.2.178] Clearing FPGA Slot 4.
[172.30.2.178] Clearing FPGA Slot 5.
[172.30.2.178] Clearing FPGA Slot 6.
[172.30.2.178] Clearing FPGA Slot 7.
[172.30.2.178] Flashing FPGA Slot: 0 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 1 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 2 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 3 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 4 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 5 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 6 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Flashing FPGA Slot: 7 with agfi: agfi-09e85ffabe3543903.
[172.30.2.178] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[172.30.2.178] Loading XDMA Driver Kernel Module.
[172.30.2.178] Copying switch simulation infrastructure for switch slot: 0.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-07-33-infrasetup-2Z7EBCBIF2TSI66Q.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo (in particular, `f1.16xlarges`

usually take a couple of minutes to start, so don't be alarmed if you're stuck at `Checking if host instance is up...`) . The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `firesim/deploy/logs/`.

At this point, the `f1.16xlarge` instance in our Run Farm has all the infrastructure necessary to run everything in our simulation.

So, let's launch our simulation!

Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up the 8-port switch simulation and then starts 8 Rocket Chip FPGA Simulations, then prints out the live status of the simulated nodes and switch every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: http://docs.fires.im
Running: runworkload

Creating the directory: /home/centos/firesim-new/deploy/results-workload/2018-05-19--06-
↪28-43-br-base/
[172.30.2.178] Executing task 'instance_liveness'
[172.30.2.178] Checking if host instance is up...
[172.30.2.178] Executing task 'boot_switch_wrapper'
[172.30.2.178] Starting switch simulation for switch slot: 0.
[172.30.2.178] Executing task 'boot_simulation_wrapper'
[172.30.2.178] Starting FPGA simulation for slot: 0.
[172.30.2.178] Starting FPGA simulation for slot: 1.
[172.30.2.178] Starting FPGA simulation for slot: 2.
[172.30.2.178] Starting FPGA simulation for slot: 3.
[172.30.2.178] Starting FPGA simulation for slot: 4.
[172.30.2.178] Starting FPGA simulation for slot: 5.
[172.30.2.178] Starting FPGA simulation for slot: 6.
[172.30.2.178] Starting FPGA simulation for slot: 7.
[172.30.2.178] Executing task 'monitor_jobs_wrapper'
```

If you don't look quickly, you might miss it, because it will be replaced with a live status page once simulations are kicked-off:

```
FireSim Simulation Status @ 2018-05-19 06:28:56.087472
--------------------------------------------------------------------------------
This workload's output is located in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--06-28-43-br-base/
This run's log is located in:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-28-43-runworkload-ZHZEJED9MDWNSCV7.
↪log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.178 | Terminated: False
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.178 | Switch name: switch0 | Switch running: True
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.178 | Job: br-base1 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base0 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base3 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base2 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base5 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base4 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base7 | Sim running: True
Hostname/IP:   172.30.2.178 | Job: br-base6 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
8/8 simulations are still running.
--------------------------------------------------------------------------------
```

In cycle-accurate networked mode, this will exit when any ONE of the simulated nodes shuts down. So, let's let it run and open another ssh connection to the manager instance. From there, `cd` into your firesim directory again and `source sourceme-manager.sh` again to get our ssh key setup. To access our simulated system, ssh into the IP address being printed by the status page, **from your manager instance**. In our case, from the above output, we see that our simulated system is running on the instance with IP `172.30.2.178`. So, run:

```
[RUN THIS ON YOUR MANAGER INSTANCE!]
ssh 172.30.2.178
```

This will log you into the instance running the simulation. On this machine, run `screen -ls` to get the list of all running simulation components. Attaching to the screens `fsim0` to `fsim7` will let you attach to the consoles of any of the 8 simulated nodes. You'll also notice an additional screen for the switch, however by default there is no interesting output printed here for performance reasons.

For example, if we want to enter commands into node zero, we can attach to its console like so:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated node and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] Registered IceNet NIC 00:12:6d:00:00:02
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
```

(continues on next page)

(continued from previous page)

```
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: OK
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

If you also ran the single-node no-nic simulation you'll notice a difference in this boot output – here, Linux sees the NIC and its assigned MAC address and automatically brings up the `eth0` interface at boot.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this getting started guide, let's poweroff the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[    3.748000] reboot: Power down
Power off
time elapsed: 360.5 s, simulation speed = 37.82 MHz
*** PASSED *** after 13634406804 cycles
Runs 13634406804 cycles
[PASS] FireSim Test
SEED: 1526711978
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Terminated: False
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Switch name: switch0 | Switch running: True
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Instance IP:   172.30.2.178 | Job: br-base1 | Sim running: True
Instance IP:   172.30.2.178 | Job: br-base0 | Sim running: False
Instance IP:   172.30.2.178 | Job: br-base3 | Sim running: True
Instance IP:   172.30.2.178 | Job: br-base2 | Sim running: True
Instance IP:   172.30.2.178 | Job: br-base5 | Sim running: True
Instance IP:   172.30.2.178 | Job: br-base4 | Sim running: True
Instance IP:   172.30.2.178 | Job: br-base7 | Sim running: True
Instance IP:   172.30.2.178 | Job: br-base6 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
7/8 simulations are still running.
--------------------------------------------------------------------------------
Teardown required, manually tearing down...
[172.30.2.178] Executing task 'kill_switch_wrapper'
[172.30.2.178] Killing switch simulation for switchslot: 0.
[172.30.2.178] Executing task 'kill_simulation_wrapper'
[172.30.2.178] Killing FPGA simulation for slot: 0.
[172.30.2.178] Killing FPGA simulation for slot: 1.
[172.30.2.178] Killing FPGA simulation for slot: 2.
[172.30.2.178] Killing FPGA simulation for slot: 3.
[172.30.2.178] Killing FPGA simulation for slot: 4.
[172.30.2.178] Killing FPGA simulation for slot: 5.
[172.30.2.178] Killing FPGA simulation for slot: 6.
[172.30.2.178] Killing FPGA simulation for slot: 7.
[172.30.2.178] Executing task 'screens'
Confirming exit...
[172.30.2.178] Executing task 'monitor_jobs_wrapper'
[172.30.2.178] Slot 0 completed! copying results.
[172.30.2.178] Slot 1 completed! copying results.
[172.30.2.178] Slot 2 completed! copying results.
[172.30.2.178] Slot 3 completed! copying results.
[172.30.2.178] Slot 4 completed! copying results.
[172.30.2.178] Slot 5 completed! copying results.
[172.30.2.178] Slot 6 completed! copying results.
[172.30.2.178] Slot 7 completed! copying results.
[172.30.2.178] Killing switch simulation for switchslot: 0.
FireSim Simulation Exited Successfully. See results in:
/home/centos/firesim-new/deploy/results-workload/2018-05-19--06-39-35-br-base/
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-39-35-runworkload-4CDB78E3A4IA9IYQ.
↪log
```

In the cluster case, you'll notice that shutting down ONE simulator causes the whole simulation to be torn down – this is because we currently do not implement any kind of "disconnect" mechanism to remove one node from a globally-

cycle-accurate simulation.

If you take a look at the workload output directory given in the manager output (in this case, /home/centos/firesim-new/deploy/results-workload/2018-05-19--06-39-35-br-base/), you'll see the following:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/results-workload/
↪2018-05-19--06-39-35-br-base$ ls -la */*
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base0/os-release
-rw-rw-r-- 1 centos centos 7476 May 19 06:45 br-base0/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base1/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base1/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base1/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base2/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base2/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base2/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base3/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base3/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base3/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base4/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base4/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base4/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base5/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base5/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base5/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base6/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base6/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base6/uartlog
-rw-rw-r-- 1 centos centos  797 May 19 06:45 br-base7/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 06:45 br-base7/os-release
-rw-rw-r-- 1 centos centos 8157 May 19 06:45 br-base7/uartlog
-rw-rw-r-- 1 centos centos  153 May 19 06:45 switch0/switchlog
```

What are these files? They are specified to the manager in a configuration file (deploy/workloads/br-base-uniform.json) as files that we want automatically copied back to our manager after we run a simulation, which is useful for running benchmarks automatically. Note that there is a directory for each simulated node and each simulated switch in the cluster. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

For now, let's wrap-up our guide by terminating the f1.16xlarge instance that we launched. To do so, run:

```
firesim terminaterunfarm -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r $
↪{CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

Which should present you with the following:

```
FireSim Manager. Docs: http://docs.fires.im
Running: terminaterunfarm

IMPORTANT!: This will terminate the following instances:
f1.16xlarges
['i-09e5491cce4d5f92d']
f1.4xlarges
[]
m4.16xlarges
```

(continues on next page)

(continued from previous page)

```
[]
f1.2xlarges
[]
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
```

You must type yes then hit enter here to have your instances terminated. Once you do so, you will see:

```
[ truncated output from above ]
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
yes
Instances terminated. Please confirm in your AWS Management Console.
The full log of this run is:
/home/centos/firesim-new/deploy/logs/2018-05-19--06-50-37-terminaterunfarm-
↪3VF0Z2KCAKKDY0ZU.log
```

At this point, you should always confirm in your AWS management console that the instance is in the shutting-down or terminated states. You are ultimately responsible for ensuring that your instances are terminated appropriately.

Congratulations on running a cluster FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left. Or, hit next to continue to a guide that shows you how to build your own custom FPGA images.

5.4 Building Your Own Hardware Designs (FireSim Amazon FPGA Images)

This section will guide you through building an Amazon FPGA Image (AFI) image for a FireSim simulation.

5.4.1 Amazon S3 Setup

During the build process, the build system will need to upload a tar file to Amazon S3 in order to complete the build process using Amazon's backend scripts (which convert the Vivado-generated tar into an AFI). The manager will create this bucket for you automatically.

Bucket names must be globally unique, so the default bucket name used by the manager will be `firesim-(YOUR_AWS_USERNAME)-(REGION)`. If the bucket name that the manager tries to use is inaccessible to you (because someone else has taken the same name), the manager will notice and complain when you tell it to build an AFI.

In the unlikely event that you need to change the bucket name from the aforementioned default, you can edit the `s3_bucket_name` value in deploy/bit-builder-recipes/f1.yaml and set `append_userid_region` to `false`.

5.4.2 Build Recipes

In the deploy/config_build.yaml file, you will notice that the builds_to_run section currently contains several lines, which indicates to the build system that you want to run all of the listed builds in parallel, with the parameters for each listed in the relevant section of the deploy/config_build_recipes.yaml file. In deploy/config_build_recipes.yaml, you can set parameters of the simulated system.

To start out, let's build a simple Chipyard target design, firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3, which is the same design we used a pre-built version of to run simulations in the earlier single-node simulation guide. This is a design that has four cores, no nic, and uses the 4MB LLC + DDR3 memory model.

The build recipe of firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3 is located in Chipyard at ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. **We will later change how the FireSim manager is called to reference this recipe.**

To do so, delete (or comment out) all of the other build recipe names listed in the builds_to_run section of deploy/config_build.yaml, besides the one we want. So, you should end up with something like this (a line beginning with a # is a comment):

```
builds_to_run:
    # this section references builds defined in config_build_recipes.yaml
    # if you add a build here, it will be built when you run buildbitstream
    - firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3
```

5.4.3 Build Farm Instance Types

FireSim will run Vivado for each build on its own z1d.2xlarge instance. You can change the instance type used by modifying the instance_type value in deploy/build-farm-recipes/aws_ec2.yaml. From our experimentation, there are diminishing returns using anything larger than a z1d.2xlarge. If you do wish to use a different build instance type, keep in mind that Vivado will consume in excess of 32 GiB of DRAM for large designs.

5.4.4 Running a Build

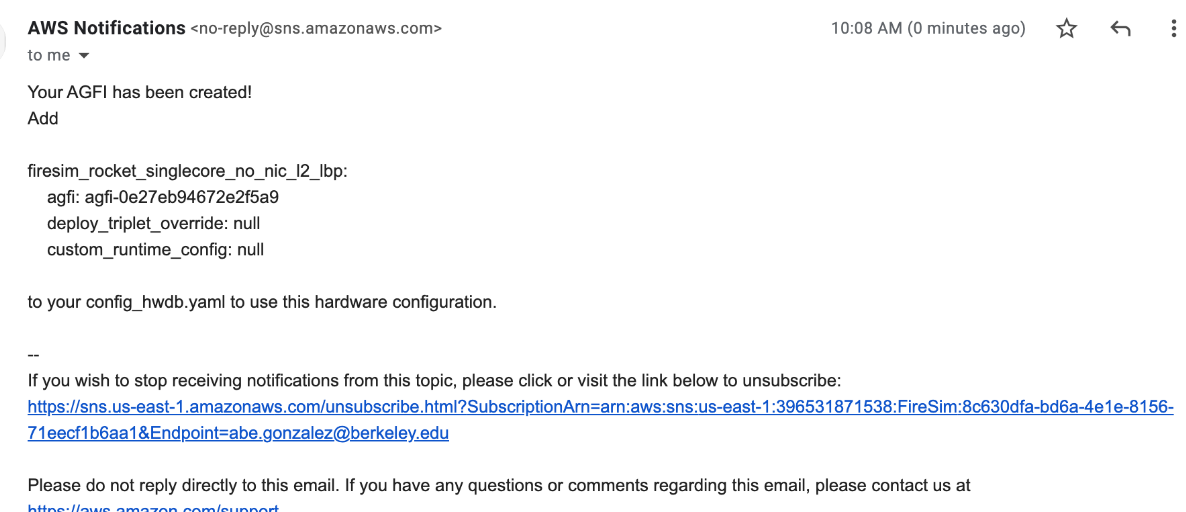
Now, we can run a build like so (ensuring that we point to the build recipe given by Chipyard):

```
firesim buildbitstream -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing an AFI/AGFI that runs on the FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in deploy/results-build/, named after your build parameter settings, that contains AGFI information (the AGFI_INFO file) and all of the outputs of the Vivado build process (in the cl_firesim subdirectory). Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems). If you provided the manager with your email address, you will also receive an email upon build completion, that should look something like this:

In addition to being included in the email, the manager will also print the entry that can be added to config_hwdb.yaml so that the generated AGFI can be used to run simulations. Note that on AWS, you will **not** have access to a physical bitstream file. The final bitstream is stored in a backend managed by AWS and the only piece of information we need to program the bitstream onto AWS F1 FPGAs is the value of the agfi: key in the config_hwdb.yaml entry.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically! To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.



AWS Notifications <no-reply@sns.amazonaws.com>
to me
10:08 AM (0 minutes ago)

Your AGFI has been created!
Add

firesim_rocket_singlecore_no_nic_l2_lbp:
agfi: agfi-0e27eb94672e2f5a9
deploy_triplet_override: null
custom_runtime_config: null

to your config_hwdb.yaml to use this hardware configuration.

--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.us-east-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-east-1:396531871538:FireSim:8c630dfa-bd6a-4e1e-8156-71eecf1b6aa1&Endpoint=abe.gonzalez@berkeley.edu

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

Fig. 1: Build Completion Email

CHAPTER SIX

XILINX ALVEO U200 XDMA-BASED GETTING STARTED GUIDE

The getting started guides that follow this page will walk you through the complete (XDMA-based) flow for getting an example FireSim simulation up and running using an on-premises Xilinx Alveo U200 FPGA, from scratch.

Make sure you have run/done the steps listed in *Local FPGA System Setup* before running this guide.

First, we'll set up your environment, then run a simulation of a single RISC-V Rocket-based SoC booting Linux, using a pre-built bitstream. Next, we'll show you how to build your own FPGA bitstreams for a custom hardware design. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

Here's a high-level outline of what we'll be doing in this guide:

6.1 FPGA Setup

The following installation steps are FPGA-specific and should be run on all **run farm machines** that install an FPGA. You might need `sudo` access to setup the FPGA.

1. Poweroff your machine.
2. Insert your Xilinx Alveo U200 FPGA into an open PCIe slot in the machine.
3. Attach any additional power cables between the FPGA and the host machine. For the U200, this is usually PCIe power coming directly from the system's PSU.
4. Attach the USB cable between the FPGA and the host machine for JTAG.
5. Boot the machine.
6. Obtain an existing bitstream tar file for your FPGA by opening the `bitstream_tar` URL listed under `alveo_u200_firesim_rocket_singlecore_no_nic` in the following file: `${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml`.
7. Download/extract the `.tar.gz` file to a known location. Inside, you will find three files; the one we are currently interested in will be called `firesim.mcs`. Note the full path of this `firesim.mcs` file for the next step.
8. Open Vivado Lab and click "Open Hardware Manager". Then click "Open Target" and "Auto connect".
9. Right-click on your FPGA and click "Add Configuration Memory Device". For a Xilinx Alveo U200, choose `mt25qu01g-spi-x1_x2_x4` as the Configuration Memory Part. Click "OK" when prompted to program the configuration memory device.

10. For Configuration file, choose the `firesim.mcs` file from step 7.
11. Uncheck "Verify" and click OK.
12. Right-click on your FPGA and click "Boot from Configuration Memory Device".
13. When programming the configuration memory device is completed, power off your machine fully (i.e., the FPGA should completely lose power).)
14. Cold-boot the machine. A cold boot is required for the FPGA to be successfully re-programmed from its flash.
15. Once the machine has booted, run the following to ensure that your FPGA is set up properly:

```
lspci -vvv -d 10ee:903f
```

If successful, this should show an entry with Xilinx as the manufacturer and two memory regions. There should be one entry for each FPGA you've added to the Run Farm Machine.

Note

Remember to keep the USB cable for JTAG connected at all times when running FireSim simulations (it is used to program the FPGA).

6.2 FireSim Repo Setup

Next, we'll clone FireSim through Chipyard on your Manager Machine and run a few final setup steps using scripts in the repo.

6.2.1 Setting up the FireSim Repo

Machine: From this point forward, run everything on your Manager Machine, unless otherwise instructed.

We're finally ready to fetch FireSim's sources through Chipyard. Chipyard provides all the necessary target designs (e.g. RISC-V SoCs) and software (e.g. Linux) used for the rest of this guide.

Note

This guide was built using Chipyard version dbc082e2206f787c3aba12b9b171e1704e15b707. It is recommended to use the most up-to-date version of Chipyard with this tutorial.

This should be done on your Manager Machine. Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
./build-setup.sh
```

Once `build-setup.sh` completes, run:

```
cd sims/firesim
source sourceme-manager.sh --skip-ssh-setup
```

This will perform various environment setup steps, such as adding the RISC-V tools to your path. Sourcing this the first time will take some time – however each subsequent sourcing should be instantaneous.

Warning

Every time you want to use FireSim, you should `cd` into your FireSim directory and source `sourceme-manager.sh` again with the arguments shown above.

6.2.2 Initializing FireSim Config Files

The FireSim manager contains a command that will automatically provide a fresh set of configuration files for a given platform.

To run it, do the following:

```
firesim managerinit --platform xilinx_alveo_u200
```

This will produce several initial configuration files, which we will edit in the next section.

6.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup

As our final setup step, we will edit FireSim's configuration files so that the manager understands our Run Farm machine setup and the set of FPGAs attached to each Run Farm machine.

Inside the cloned FireSim repo, open up the `deploy/config_runtime.yaml` file and set the following keys to the indicated values:

- `default_simulation_dir` should point to a temporary simulation directory of your choice on your Run Farm Machines. This is the directory that simulations will run out of.
- `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system.

Here are two examples of how this could be configured:

Example 1: Your Run Farm has a single machine with one FPGA attached and this machine is also your Manager Machine:

```
...
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec
...
```

Example 2: You have two Run Farm Machines (separate from your Manager Machine). The Run Farm Machines are accessible from your manager machine with the hostnames `firesim-runner1.berkeley.edu` and `firesim-runner2.berkeley.edu`, each with eight FPGAs attached.

```
...
    run_farm_hosts_to_use:
        - firesim-runner1.berkeley.edu: eight_fpgas_spec
```

(continues on next page)

(continued from previous page)

```
        - firesim-runner2.berkeley.edu: eight_fpgas_spec
...
```

- `default_hw_config` should be `alveo_u200_firesim_rocket_singlecore_no_nic`

Then, run the following command so that FireSim can generate a mapping from the FPGA ID used for JTAG programming to the PCIe ID used to run simulations. If you ever change the physical layout of the machine (e.g., which PCIe slot the FPGAs are attached to), you will need to re-run this command.

```
firesim enumeratefpgas
```

This will generate a database file in `/opt/firesim-db.json` on each Run Farm Machine that has this mapping.

Now you're ready to run your first FireSim simulation! Hit Next to continue with the guide.

6.3 Running a Single Node Simulation

Now that we've completed all the basic setup steps, it's time to run a simulation! In this section, we will simulate a single target node, for which we will use a single Xilinx Alveo U200.

Make sure you have sourced `sourceme-manager.sh --skip-ssh-setup` **before running any of these commands.**

6.3.1 Building target software

In this guide, we'll boot Linux on our simulated node. To do so, we'll need to build our RISC-V SoC-compatible Linux distro. For this guide, we will use a simple buildroot-based distribution. We can build the Linux distribution like so:

```
# assumes you already cd'd into your firesim repo
# and sourced sourceme-manager.sh
#
# then:
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the RISC-V SoC we will simulate.
- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for the RISC-V SoC we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or directly as a basic, interactive Linux distribution.

6.4 Setting up the manager configuration

All runtime configuration options for the manager are located in `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

Based on the changes we made earlier, this file will already have everything set correctly to run a simulation.

Below we'll highlight a few of these lines to explain what is happening:

- At the top, you'll notice the `run_farm` mapping, which describes and specifies the machines to run simulations on.
  - By default, we'll be using a `base_recipe` of `run-farm-recipes/externally_provisioned.yaml`, which means that our Run Farm machines are pre-configured, and do not require the manager to dynamically launch/terminate them (e.g., as we would do for cloud instances).
  - The `default_platform` has automatically been set for our FPGA board, to `XilinxAlveoU200InstanceDeployManager`.
  - The `default_simulation_dir` is the directory on the Run Farm Machines where simulations will run out of. The default is likely fine, but you can change it to any directory you have access to on the Run Farm machines.
  - `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system. We configured this already in the previous step.
- The `target_config` section describes the system that we'd like to simulate.
  - `topology: no_net_config` indicates that we're running simulations with no network between them.
  - `no_net_num_nodes: 1` indicates that we'll be a simulation of a single SoC
  - The `default_hw_config` will be set to a pre-built design. **Change this to** alveo_u200_firesim_rocket_singlecore_no_nic **to simulate a single RISC-V Rocket core SoC**.
- The `workload` section describes the workload that we'd like to run on our simulated systems. In this case, we need to change it to boot Linux on all SoCs in the simulation. **Change the line to the following**: `workload_name: br-base-uniform.json`.

6.5 Building and Deploying simulation infrastructure to the Run Farm Machines

The manager automates the process of building and deploying all components necessary to run your simulation on the Run Farm, including programming FPGAs. To tell the manager to setup all of our simulation infrastructure, run the following:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: https://docs.fires.im
Running: infrasetup

Building FPGA software driver.
...
[localhost] Checking if host instance is up...
[localhost] Copying FPGA simulation infrastructure for slot: 0.
[localhost] Clearing all FPGA Slots.
The full log of this run is:
.../firesim/deploy/logs/2023-03-06--01-22-46-infrasetup-35ZP4WUOX8KUYBF3.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `${FS_DIR}/deploy/logs/`.

At this point, our single Run Farm machine has all the infrastructure necessary to run a simulation, so let's launch our simulation!

6.6 Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: https://docs.fires.im
Running: runworkload

Creating the directory: .../firesim/deploy/results-workload/2023-03-06--01-25-34-br-base/
[localhost] Checking if host instance is up...
[localhost] Starting FPGA simulation for slot: 0.
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have powered off. So, let's let it run and open another terminal on the manager machine. From there, `cd` into your FireSim directory again and `source sourceme-manager.sh --skip-ssh-setup`.

Next, let's `ssh` into the Run Farm machine. If your Run Farm and Manager Machines are the same, replace `RUN_FARM_IP_OR_HOSTNAME` with `localhost`, otherwise replace it with your Run Farm Machine's IP or hostname.

```
source ~/.ssh/AGENT_VARS
ssh RUN_FARM_IP_OR_HOSTNAME
```

Next, we can directly attach to the console of the simulated system using `screen`, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
```

(continues on next page)

(continued from previous page)

```
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's power off the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
$ ls -la firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/*/*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back from the Run Farm Machine into the `results-workload` directory on our manager machine, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left.

Click Next if you'd like to continue on to building your own bitstreams.

6.7 Building Your Own Hardware Designs

This section will guide you through building a Xilinx Alveo U200 FPGA bitstream to run FireSim simulations.

6.8 System Setup

Here, we'll do some final one-time setup for your Build Farm Machines so that we can build bitstreams for FireSim simulations automatically.

These steps assume that you have already followed the earlier setup steps required to run simulations.

As noted earlier, it is highly recommended that you use Ubuntu 20.04 LTS as the host operating system for all machine types in an on-premises setup, as this is the OS recommended by Xilinx.

Also recall that we make a distinction between the Manager Machine, the Build Farm Machine(s), and the Run Farm Machine(s). In a simple setup, these can all be a single machine, in which case you should run the Build Farm Machine setup steps below on your single machine.

6.8.1 1. Install Vivado for Builds

Machines: Build Farm Machines.

Running builds for Xilinx Alveo U200 in FireSim requires Vivado 2021.1. Other versions are unlikely to work out-of-the-box.

On each Build Farm machine, do the following:

1. Install Vivado 2021.1 from the Xilinx Downloads Website. By default, Vivado will be installed to `/tools/Xilinx/Vivado/2021.1`. We recommend keeping this default. If you change it to something else, you will need to adjust the path in the rest of the setup steps.
2. Add the following to `~/.bashrc` so that `vivado` is available when `ssh`-ing into the machine:

```
source /tools/Xilinx/Vivado/2021.1/settings64.sh
```

3. Download the `au200` board support package directory from https://github.com/Xilinx/open-nic-shell/tree/main/board_files/Xilinx and place the directory in `/tools/Xilinx/Vivado/2021.1/data/xhub/boards/XilinxBoardStore/boards/Xilinx/`.

If you have multiple Build Farm Machines, you should repeat this process for each.

6.8.2 2. Verify Build Farm Machine environment

Machines: Manager Machine and Run Farm Machines

Finally, let's ensure that Vivado 2021.1 is properly sourced in your shell setup (i.e. `.bashrc`) so that any shell on your Build Farm Machines can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned.

You can check this by running the following on the Manager Machine, replacing `BUILD_FARM_IP` with `localhost` if your Build Farm machine and Manager machine are the same machine, or replacing it with the Build Farm machine's IP address if they are different machines.

```
ssh BUILD_FARM_IP printenv
```

Ensure that the output of the command shows that the Vivado 2021.1 tools are present in the printed environment variables (i.e., `PATH` and `XILINX_VIVADO`).

If you have multiple Build Farm machines, you should repeat this process for each Build Farm machine, replacing `BUILD_FARM_IP` with a different Build Farm Machine's IP address.

6.9 Configuring a Build in the Manager

In the `deploy/config_build.yaml` file, you will notice that the `builds_to_run` section currently contains several lines, which indicates to the build system that you want to run all of these "build recipes" in parallel, with the parameters for each "build recipe" listed in the relevant section of the `deploy/config_build_recipes.yaml` file (or in the case of building a Chipyard target SoC in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml).

In this guide, we'll build the default FireSim design for the Xilinx Alveo U200, which is specified by the `alveo_u200_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. This was the same configuration used to build the pre-built bitstream that you used to run simulations in the guide to running a simulation.

Looking at the `alveo_u200_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml, there are a few notable items:

- TARGET_CONFIG specifies that this configuration is a simple singlecore RISC-V Rocket with a single DRAM channel.
- TARGET_PROJECT_MAKEFRAG specifies to the FireSim Make system how to build the TARGET_CONFIG.
- bit_builder_recipe points to bit-builder-recipes/xilinx_alveo_u200.yaml, which is found in the deploy directory and tells the FireSim build system how to build bitstreams for this FPGA.

Having looked at this entry, let's now set up the build in deploy/config_build.yaml. First, we'll set up the build_farm mapping, which specifies the Build Farm Machines that are available to build FPGA bitstreams.

- base_recipe will map to build-farm-recipes/externally_provisioned.yaml. This indicates to the FireSim manager that the machines used to run builds are existing machines that have been set up by the user, instead of cloud instances that are automatically provisioned.
- default_build_dir is the directory in which builds will run out of on your Build Farm Machines. Change the default null to a path where you would like temporary build data to be stored on your Build Farm Machines.
- build_farm_hosts is a section that contains a list of IP addresses or hostnames of machines in your Build Farm. By default, localhost is specified. If you are using a separate Build Farm Machine, you should replace this with the IP address or hostname of the Build Farm Machine on which you would like to run the build.

Having configured our Build Farm, let's specify the design we'd like to build. To do this, edit the builds_to_run section in deploy/config_build.yaml so that it looks like the following:

```
builds_to_run:
    - alveo_u200_firesim_rocket_singlecore_no_nic
```

In essence, you should delete or comment out all the other items in the builds_to_run section besides alveo_u200_firesim_rocket_singlecore_no_nic.

6.10 Running the Build

Now, we can run a build like so (while also pointing to Chipyard's recipes provided):

```
firesim buildbitstream -r -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.
↪yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing a bitstream that runs on the Xilinx Alveo U200 FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in deploy/results-build/, named after your build parameter settings, that contains all of the outputs of the Xilinx Vivado build process. Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems).

The manager will also print an entry that can be added to config_hwdb.yaml so that the bitstream can be used to run simulations. This entry will contain a bitstream_tar key whose value is the path to the final generated bitstream file. You can share generated bitstreams with others by sharing the file listed in bitstream_tar and the config_hwdb.yaml entry for it.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically!

This is the end of the Getting Started Guide. To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

CHAPTER

SEVEN

XILINX ALVEO U250 XDMA-BASED GETTING STARTED GUIDE

The getting started guides that follow this page will walk you through the complete (XDMA-based) flow for getting an example FireSim simulation up and running using an on-premises Xilinx Alveo U250 FPGA, from scratch.

Make sure you have run/done the steps listed in *Local FPGA System Setup* before running this guide.

First, we'll set up your environment, then run a simulation of a single RISC-V Rocket-based SoC booting Linux, using a pre-built bitstream. Next, we'll show you how to build your own FPGA bitstreams for a custom hardware design. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

Here's a high-level outline of what we'll be doing in this guide:

7.1 FPGA Setup

The following installation steps are FPGA-specific and should be run on all **run farm machines** that install an FPGA. You might need `sudo` access to setup the FPGA.

1. Poweroff your machine.
2. Insert your Xilinx Alveo U250 FPGA into an open PCIe slot in the machine.
3. Attach any additional power cables between the FPGA and the host machine. For the U250, this is usually PCIe power coming directly from the system's PSU.
4. Attach the USB cable between the FPGA and the host machine for JTAG.
5. Boot the machine.
6. Obtain an existing bitstream tar file for your FPGA by opening the `bitstream_tar` URL listed under `alveo_u250_firesim_rocket_singlecore_no_nic` in the following file: `${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml`.
7. Download/extract the `.tar.gz` file to a known location. Inside, you will find three files; the one we are currently interested in will be called `firesim.mcs`. Note the full path of this `firesim.mcs` file for the next step.
8. Open Vivado Lab and click "Open Hardware Manager". Then click "Open Target" and "Auto connect".
9. Right-click on your FPGA and click "Add Configuration Memory Device". For a Xilinx Alveo U250, choose `mt25qu01g-spi-x1_x2_x4` as the Configuration Memory Part. Click "OK" when prompted to program the configuration memory device.

10. For Configuration file, choose the `firesim.mcs` file from step 7.
11. Uncheck "Verify" and click OK.
12. Right-click on your FPGA and click "Boot from Configuration Memory Device".
13. When programming the configuration memory device is completed, power off your machine fully (i.e., the FPGA should completely lose power).)
14. Cold-boot the machine. A cold boot is required for the FPGA to be successfully re-programmed from its flash.
15. Once the machine has booted, run the following to ensure that your FPGA is set up properly:

```
lspci -vvv -d 10ee:903f
```

If successful, this should show an entry with Xilinx as the manufacturer and two memory regions. There should be one entry for each FPGA you've added to the Run Farm Machine.

Note

Remember to keep the USB cable for JTAG connected at all times when running FireSim simulations (it is used to program the FPGA).

7.2 FireSim Repo Setup

Next, we'll clone FireSim through Chipyard on your Manager Machine and run a few final setup steps using scripts in the repo.

7.2.1 Setting up the FireSim Repo

Machine: From this point forward, run everything on your Manager Machine, unless otherwise instructed.

We're finally ready to fetch FireSim's sources through Chipyard. Chipyard provides all the necessary target designs (e.g. RISC-V SoCs) and software (e.g. Linux) used for the rest of this guide.

Note

This guide was built using Chipyard version dbc082e2206f787c3aba12b9b171e1704e15b707. It is recommended to use the most up-to-date version of Chipyard with this tutorial.

This should be done on your Manager Machine. Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
./build-setup.sh
```

Once `build-setup.sh` completes, run:

```
cd sims/firesim
source sourceme-manager.sh --skip-ssh-setup
```

This will perform various environment setup steps, such as adding the RISC-V tools to your path. Sourcing this the first time will take some time – however each subsequent sourcing should be instantaneous.

Warning

Every time you want to use FireSim, you should `cd` into your FireSim directory and source `sourceme-manager.sh` again with the arguments shown above.

7.2.2 Initializing FireSim Config Files

The FireSim manager contains a command that will automatically provide a fresh set of configuration files for a given platform.

To run it, do the following:

```
firesim managerinit --platform xilinx_alveo_u250
```

This will produce several initial configuration files, which we will edit in the next section.

7.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup

As our final setup step, we will edit FireSim's configuration files so that the manager understands our Run Farm machine setup and the set of FPGAs attached to each Run Farm machine.

Inside the cloned FireSim repo, open up the `deploy/config_runtime.yaml` file and set the following keys to the indicated values:

- `default_simulation_dir` should point to a temporary simulation directory of your choice on your Run Farm Machines. This is the directory that simulations will run out of.
- `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system.

Here are two examples of how this could be configured:

Example 1: Your Run Farm has a single machine with one FPGA attached and this machine is also your Manager Machine:

```
...
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec
...
```

Example 2: You have two Run Farm Machines (separate from your Manager Machine). The Run Farm Machines are accessible from your manager machine with the hostnames `firesim-runner1.berkeley.edu` and `firesim-runner2.berkeley.edu`, each with eight FPGAs attached.

```
...
    run_farm_hosts_to_use:
        - firesim-runner1.berkeley.edu: eight_fpgas_spec
```

(continues on next page)

(continued from previous page)

```
        - firesim-runner2.berkeley.edu: eight_fpgas_spec
...
```

- `default_hw_config` should be `alveo_u250_firesim_rocket_singlecore_no_nic`

Then, run the following command so that FireSim can generate a mapping from the FPGA ID used for JTAG programming to the PCIe ID used to run simulations. If you ever change the physical layout of the machine (e.g., which PCIe slot the FPGAs are attached to), you will need to re-run this command.

```
firesim enumeratefpgas
```

This will generate a database file in `/opt/firesim-db.json` on each Run Farm Machine that has this mapping.

Now you're ready to run your first FireSim simulation! Hit Next to continue with the guide.

7.3 Running a Single Node Simulation

Now that we've completed all the basic setup steps, it's time to run a simulation! In this section, we will simulate a single target node, for which we will use a single Xilinx Alveo U250.

Make sure you have sourced `sourceme-manager.sh --skip-ssh-setup` **before running any of these commands.**

7.3.1 Building target software

In this guide, we'll boot Linux on our simulated node. To do so, we'll need to build our RISC-V SoC-compatible Linux distro. For this guide, we will use a simple buildroot-based distribution. We can build the Linux distribution like so:

```
# assumes you already cd'd into your firesim repo
# and sourced sourceme-manager.sh
#
# then:
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the RISC-V SoC we will simulate.
- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for the RISC-V SoC we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or directly as a basic, interactive Linux distribution.

7.4 Setting up the manager configuration

All runtime configuration options for the manager are located in `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

Based on the changes we made earlier, this file will already have everything set correctly to run a simulation.

Below we'll highlight a few of these lines to explain what is happening:

- At the top, you'll notice the `run_farm` mapping, which describes and specifies the machines to run simulations on.
  - By default, we'll be using a `base_recipe` of `run-farm-recipes/externally_provisioned.yaml`, which means that our Run Farm machines are pre-configured, and do not require the manager to dynamically launch/terminate them (e.g., as we would do for cloud instances).
  - The `default_platform` has automatically been set for our FPGA board, to `XilinxAlveoU250InstanceDeployManager`.
  - The `default_simulation_dir` is the directory on the Run Farm Machines where simulations will run out of. The default is likely fine, but you can change it to any directory you have access to on the Run Farm machines.
  - `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system. We configured this already in the previous step.
- The `target_config` section describes the system that we'd like to simulate.
  - `topology: no_net_config` indicates that we're running simulations with no network between them.
  - `no_net_num_nodes: 1` indicates that we'll be a simulation of a single SoC
  - The `default_hw_config` will be set to a pre-built design. **Change this to** alveo_u250_firesim_rocket_singlecore_no_nic **to simulate a single RISC-V Rocket core SoC**.
- The `workload` section describes the workload that we'd like to run on our simulated systems. In this case, we need to change it to boot Linux on all SoCs in the simulation. **Change the line to the following**: `workload_name: br-base-uniform.json`.

7.5 Building and Deploying simulation infrastructure to the Run Farm Machines

The manager automates the process of building and deploying all components necessary to run your simulation on the Run Farm, including programming FPGAs. To tell the manager to setup all of our simulation infrastructure, run the following:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: https://docs.fires.im
Running: infrasetup

Building FPGA software driver.
...
[localhost] Checking if host instance is up...
[localhost] Copying FPGA simulation infrastructure for slot: 0.
[localhost] Clearing all FPGA Slots.
The full log of this run is:
.../firesim/deploy/logs/2023-03-06--01-22-46-infrasetup-35ZP4WUOX8KUYBF3.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `${FS_DIR}/deploy/logs/`.

At this point, our single Run Farm machine has all the infrastructure necessary to run a simulation, so let's launch our simulation!

7.6 Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: https://docs.fires.im
Running: runworkload

Creating the directory: .../firesim/deploy/results-workload/2023-03-06--01-25-34-br-base/
[localhost] Checking if host instance is up...
[localhost] Starting FPGA simulation for slot: 0.
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have powered off. So, let's let it run and open another terminal on the manager machine. From there, `cd` into your FireSim directory again and `source sourceme-manager.sh --skip-ssh-setup`.

Next, let's `ssh` into the Run Farm machine. If your Run Farm and Manager Machines are the same, replace `RUN_FARM_IP_OR_HOSTNAME` with `localhost`, otherwise replace it with your Run Farm Machine's IP or hostname.

```
source ~/.ssh/AGENT_VARS
ssh RUN_FARM_IP_OR_HOSTNAME
```

Next, we can directly attach to the console of the simulated system using `screen`, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
```

(continues on next page)

(continued from previous page)

```
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's power off the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
$ ls -la firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/*/*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back from the Run Farm Machine into the `results-workload` directory on our manager machine, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left.

Click Next if you'd like to continue on to building your own bitstreams.

7.7 Building Your Own Hardware Designs

This section will guide you through building a Xilinx Alveo U250 FPGA bitstream to run FireSim simulations.

7.8 System Setup

Here, we'll do some final one-time setup for your Build Farm Machines so that we can build bitstreams for FireSim simulations automatically.

These steps assume that you have already followed the earlier setup steps required to run simulations.

As noted earlier, it is highly recommended that you use Ubuntu 20.04 LTS as the host operating system for all machine types in an on-premises setup, as this is the OS recommended by Xilinx.

Also recall that we make a distinction between the Manager Machine, the Build Farm Machine(s), and the Run Farm Machine(s). In a simple setup, these can all be a single machine, in which case you should run the Build Farm Machine setup steps below on your single machine.

7.8.1 1. Install Vivado for Builds

Machines: Build Farm Machines.

Running builds for Xilinx Alveo U250 in FireSim requires Vivado 2021.1. Other versions are unlikely to work out-of-the-box.

On each Build Farm machine, do the following:

1. Install Vivado 2021.1 from the Xilinx Downloads Website. By default, Vivado will be installed to `/tools/Xilinx/Vivado/2021.1`. We recommend keeping this default. If you change it to something else, you will need to adjust the path in the rest of the setup steps.
2. Add the following to `~/.bashrc` so that `vivado` is available when `ssh`-ing into the machine:

```
source /tools/Xilinx/Vivado/2021.1/settings64.sh
```

3. Download the `au250` board support package directory from https://github.com/Xilinx/open-nic-shell/tree/main/board_files/Xilinx and place the directory in `/tools/Xilinx/Vivado/2021.1/data/xhub/boards/XilinxBoardStore/boards/Xilinx/`.

If you have multiple Build Farm Machines, you should repeat this process for each.

7.8.2 2. Verify Build Farm Machine environment

Machines: Manager Machine and Run Farm Machines

Finally, let's ensure that Vivado 2021.1 is properly sourced in your shell setup (i.e. `.bashrc`) so that any shell on your Build Farm Machines can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned.

You can check this by running the following on the Manager Machine, replacing `BUILD_FARM_IP` with `localhost` if your Build Farm machine and Manager machine are the same machine, or replacing it with the Build Farm machine's IP address if they are different machines.

```
ssh BUILD_FARM_IP printenv
```

Ensure that the output of the command shows that the Vivado 2021.1 tools are present in the printed environment variables (i.e., `PATH` and `XILINX_VIVADO`).

If you have multiple Build Farm machines, you should repeat this process for each Build Farm machine, replacing `BUILD_FARM_IP` with a different Build Farm Machine's IP address.

7.9 Configuring a Build in the Manager

In the `deploy/config_build.yaml` file, you will notice that the `builds_to_run` section currently contains several lines, which indicates to the build system that you want to run all of these "build recipes" in parallel, with the parameters for each "build recipe" listed in the relevant section of the `deploy/config_build_recipes.yaml` file (or in the case of building a Chipyard target SoC in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml).

In this guide, we'll build the default FireSim design for the Xilinx Alveo U250, which is specified by the `alveo_u250_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. This was the same configuration used to build the pre-built bitstream that you used to run simulations in the guide to running a simulation.

Looking at the `alveo_u250_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml, there are a few notable items:

- TARGET_CONFIG specifies that this configuration is a simple singlecore RISC-V Rocket with a single DRAM channel.
- TARGET_PROJECT_MAKEFRAG specifies to the FireSim Make system how to build the TARGET_CONFIG.
- bit_builder_recipe points to bit-builder-recipes/xilinx_alveo_u250.yaml, which is found in the deploy directory and tells the FireSim build system how to build bitstreams for this FPGA.

Having looked at this entry, let's now set up the build in deploy/config_build.yaml. First, we'll set up the build_farm mapping, which specifies the Build Farm Machines that are available to build FPGA bitstreams.

- base_recipe will map to build-farm-recipes/externally_provisioned.yaml. This indicates to the FireSim manager that the machines used to run builds are existing machines that have been set up by the user, instead of cloud instances that are automatically provisioned.
- default_build_dir is the directory in which builds will run out of on your Build Farm Machines. Change the default null to a path where you would like temporary build data to be stored on your Build Farm Machines.
- build_farm_hosts is a section that contains a list of IP addresses or hostnames of machines in your Build Farm. By default, localhost is specified. If you are using a separate Build Farm Machine, you should replace this with the IP address or hostname of the Build Farm Machine on which you would like to run the build.

Having configured our Build Farm, let's specify the design we'd like to build. To do this, edit the builds_to_run section in deploy/config_build.yaml so that it looks like the following:

```
builds_to_run:
    - alveo_u250_firesim_rocket_singlecore_no_nic
```

In essence, you should delete or comment out all the other items in the builds_to_run section besides alveo_u250_firesim_rocket_singlecore_no_nic.

7.10 Running the Build

Now, we can run a build like so (while also pointing to Chipyard's recipes provided):

```
firesim buildbitstream -r -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.
↪yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing a bitstream that runs on the Xilinx Alveo U250 FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in deploy/results-build/, named after your build parameter settings, that contains all of the outputs of the Xilinx Vivado build process. Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems).

The manager will also print an entry that can be added to config_hwdb.yaml so that the bitstream can be used to run simulations. This entry will contain a bitstream_tar key whose value is the path to the final generated bitstream file. You can share generated bitstreams with others by sharing the file listed in bitstream_tar and the config_hwdb.yaml entry for it.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically!

This is the end of the Getting Started Guide. To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

CHAPTER

EIGHT

XILINX ALVEO U280 XDMA-BASED GETTING STARTED GUIDE

The getting started guides that follow this page will walk you through the complete (XDMA-based) flow for getting an example FireSim simulation up and running using an on-premises Xilinx Alveo U280 FPGA, from scratch.

Make sure you have run/done the steps listed in *Local FPGA System Setup* before running this guide.

First, we'll set up your environment, then run a simulation of a single RISC-V Rocket-based SoC booting Linux, using a pre-built bitstream. Next, we'll show you how to build your own FPGA bitstreams for a custom hardware design. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

Here's a high-level outline of what we'll be doing in this guide:

8.1 FPGA Setup

The following installation steps are FPGA-specific and should be run on all **run farm machines** that install an FPGA. You might need `sudo` access to setup the FPGA.

1. Poweroff your machine.
2. Insert your Xilinx Alveo U280 FPGA into an open PCIe slot in the machine.
3. Attach any additional power cables between the FPGA and the host machine. For the U280, this is usually PCIe power coming directly from the system's PSU.
4. Attach the USB cable between the FPGA and the host machine for JTAG.
5. Boot the machine.
6. Obtain an existing bitstream tar file for your FPGA by opening the `bitstream_tar` URL listed under `alveo_u280_firesim_rocket_singlecore_no_nic` in the following file: `${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml`.
7. Download/extract the `.tar.gz` file to a known location. Inside, you will find three files; the one we are currently interested in will be called `firesim.mcs`. Note the full path of this `firesim.mcs` file for the next step.
8. Open Vivado Lab and click "Open Hardware Manager". Then click "Open Target" and "Auto connect".
9. Right-click on your FPGA and click "Add Configuration Memory Device". For a Xilinx Alveo U280, choose `mt25qu01g-spi-x1_x2_x4` as the Configuration Memory Part. Click "OK" when prompted to program the configuration memory device.

10. For Configuration file, choose the `firesim.mcs` file from step 7.
11. Uncheck "Verify" and click OK.
12. Right-click on your FPGA and click "Boot from Configuration Memory Device".
13. When programming the configuration memory device is completed, power off your machine fully (i.e., the FPGA should completely lose power).)
14. Cold-boot the machine. A cold boot is required for the FPGA to be successfully re-programmed from its flash.
15. Once the machine has booted, run the following to ensure that your FPGA is set up properly:

```
lspci -vvv -d 10ee:903f
```

If successful, this should show an entry with Xilinx as the manufacturer and two memory regions. There should be one entry for each FPGA you've added to the Run Farm Machine.

Note

Remember to keep the USB cable for JTAG connected at all times when running FireSim simulations (it is used to program the FPGA).

8.2 FireSim Repo Setup

Next, we'll clone FireSim through Chipyard on your Manager Machine and run a few final setup steps using scripts in the repo.

8.2.1 Setting up the FireSim Repo

Machine: From this point forward, run everything on your Manager Machine, unless otherwise instructed.

We're finally ready to fetch FireSim's sources through Chipyard. Chipyard provides all the necessary target designs (e.g. RISC-V SoCs) and software (e.g. Linux) used for the rest of this guide.

Note

This guide was built using Chipyard version dbc082e2206f787c3aba12b9b171e1704e15b707. It is recommended to use the most up-to-date version of Chipyard with this tutorial.

This should be done on your Manager Machine. Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
./build-setup.sh
```

Once `build-setup.sh` completes, run:

```
cd sims/firesim
source sourceme-manager.sh --skip-ssh-setup
```

This will perform various environment setup steps, such as adding the RISC-V tools to your path. Sourcing this the first time will take some time – however each subsequent sourcing should be instantaneous.

Warning

Every time you want to use FireSim, you should `cd` into your FireSim directory and source `sourceme-manager.sh` again with the arguments shown above.

8.2.2 Initializing FireSim Config Files

The FireSim manager contains a command that will automatically provide a fresh set of configuration files for a given platform.

To run it, do the following:

```
firesim managerinit --platform xilinx_alveo_u280
```

This will produce several initial configuration files, which we will edit in the next section.

8.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup

As our final setup step, we will edit FireSim's configuration files so that the manager understands our Run Farm machine setup and the set of FPGAs attached to each Run Farm machine.

Inside the cloned FireSim repo, open up the `deploy/config_runtime.yaml` file and set the following keys to the indicated values:

- `default_simulation_dir` should point to a temporary simulation directory of your choice on your Run Farm Machines. This is the directory that simulations will run out of.
- `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system.

Here are two examples of how this could be configured:

Example 1: Your Run Farm has a single machine with one FPGA attached and this machine is also your Manager Machine:

```
...
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec
...
```

Example 2: You have two Run Farm Machines (separate from your Manager Machine). The Run Farm Machines are accessible from your manager machine with the hostnames `firesim-runner1.berkeley.edu` and `firesim-runner2.berkeley.edu`, each with eight FPGAs attached.

```
...
    run_farm_hosts_to_use:
        - firesim-runner1.berkeley.edu: eight_fpgas_spec
```

(continues on next page)

(continued from previous page)

```
        - firesim-runner2.berkeley.edu: eight_fpgas_spec
...
```

- default_hw_config should be alveo_u280_firesim_rocket_singlecore_no_nic

Then, run the following command so that FireSim can generate a mapping from the FPGA ID used for JTAG programming to the PCIe ID used to run simulations. If you ever change the physical layout of the machine (e.g., which PCIe slot the FPGAs are attached to), you will need to re-run this command.

```
firesim enumeratefpgas
```

This will generate a database file in /opt/firesim-db.json on each Run Farm Machine that has this mapping.

Now you're ready to run your first FireSim simulation! Hit Next to continue with the guide.

8.3 Running a Single Node Simulation

Now that we've completed all the basic setup steps, it's time to run a simulation! In this section, we will simulate a single target node, for which we will use a single Xilinx Alveo U280.

Make sure you have sourced sourceme-manager.sh --skip-ssh-setup **before running any of these commands.**

8.3.1 Building target software

In this guide, we'll boot Linux on our simulated node. To do so, we'll need to build our RISC-V SoC-compatible Linux distro. For this guide, we will use a simple buildroot-based distribution. We can build the Linux distribution like so:

```
# assumes you already cd'd into your firesim repo
# and sourced sourceme-manager.sh
#
# then:
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

Once this is completed, you'll have the following files:

- ${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin - a bootloader + Linux kernel image for the RISC-V SoC we will simulate.
- ${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img - a disk image for the RISC-V SoC we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or directly as a basic, interactive Linux distribution.

8.4 Setting up the manager configuration

All runtime configuration options for the manager are located in `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

Based on the changes we made earlier, this file will already have everything set correctly to run a simulation.

Below we'll highlight a few of these lines to explain what is happening:

- At the top, you'll notice the `run_farm` mapping, which describes and specifies the machines to run simulations on.
  - By default, we'll be using a `base_recipe` of `run-farm-recipes/externally_provisioned.yaml`, which means that our Run Farm machines are pre-configured, and do not require the manager to dynamically launch/terminate them (e.g., as we would do for cloud instances).
  - The `default_platform` has automatically been set for our FPGA board, to `XilinxAlveoU280InstanceDeployManager`.
  - The `default_simulation_dir` is the directory on the Run Farm Machines where simulations will run out of. The default is likely fine, but you can change it to any directory you have access to on the Run Farm machines.
  - `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system. We configured this already in the previous step.
- The `target_config` section describes the system that we'd like to simulate.
  - `topology: no_net_config` indicates that we're running simulations with no network between them.
  - `no_net_num_nodes: 1` indicates that we'll be a simulation of a single SoC
  - The `default_hw_config` will be set to a pre-built design. **Change this to** alveo_u280_firesim_rocket_singlecore_no_nic **to simulate a single RISC-V Rocket core SoC**.
- The `workload` section describes the workload that we'd like to run on our simulated systems. In this case, we need to change it to boot Linux on all SoCs in the simulation. **Change the line to the following**: `workload_name: br-base-uniform.json`.

8.5 Building and Deploying simulation infrastructure to the Run Farm Machines

The manager automates the process of building and deploying all components necessary to run your simulation on the Run Farm, including programming FPGAs. To tell the manager to setup all of our simulation infrastructure, run the following:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: https://docs.fires.im
Running: infrasetup

Building FPGA software driver.
...
[localhost] Checking if host instance is up...
[localhost] Copying FPGA simulation infrastructure for slot: 0.
[localhost] Clearing all FPGA Slots.
The full log of this run is:
.../firesim/deploy/logs/2023-03-06--01-22-46-infrasetup-35ZP4WUOX8KUYBF3.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `${FS_DIR}/deploy/logs/`.

At this point, our single Run Farm machine has all the infrastructure necessary to run a simulation, so let's launch our simulation!

8.6 Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: https://docs.fires.im
Running: runworkload

Creating the directory: .../firesim/deploy/results-workload/2023-03-06--01-25-34-br-base/
[localhost] Checking if host instance is up...
[localhost] Starting FPGA simulation for slot: 0.
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have powered off. So, let's let it run and open another terminal on the manager machine. From there, `cd` into your FireSim directory again and `source sourceme-manager.sh --skip-ssh-setup`.

Next, let's `ssh` into the Run Farm machine. If your Run Farm and Manager Machines are the same, replace `RUN_FARM_IP_OR_HOSTNAME` with `localhost`, otherwise replace it with your Run Farm Machine's IP or hostname.

```
source ~/.ssh/AGENT_VARS
ssh RUN_FARM_IP_OR_HOSTNAME
```

Next, we can directly attach to the console of the simulated system using `screen`, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
```

(continues on next page)

(continued from previous page)

```
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's power off the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
$ ls -la firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/*/*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back from the Run Farm Machine into the `results-workload` directory on our manager machine, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left.

Click Next if you'd like to continue on to building your own bitstreams.

8.7 Building Your Own Hardware Designs

This section will guide you through building a Xilinx Alveo U280 FPGA bitstream to run FireSim simulations.

8.8 System Setup

Here, we'll do some final one-time setup for your Build Farm Machines so that we can build bitstreams for FireSim simulations automatically.

These steps assume that you have already followed the earlier setup steps required to run simulations.

As noted earlier, it is highly recommended that you use Ubuntu 20.04 LTS as the host operating system for all machine types in an on-premises setup, as this is the OS recommended by Xilinx.

Also recall that we make a distinction between the Manager Machine, the Build Farm Machine(s), and the Run Farm Machine(s). In a simple setup, these can all be a single machine, in which case you should run the Build Farm Machine setup steps below on your single machine.

8.8.1 1. Install Vivado for Builds

Machines: Build Farm Machines.

Running builds for Xilinx Alveo U280 in FireSim requires Vivado 2021.1. Other versions are unlikely to work out-of-the-box.

On each Build Farm machine, do the following:

1. Install Vivado 2021.1 from the Xilinx Downloads Website. By default, Vivado will be installed to `/tools/Xilinx/Vivado/2021.1`. We recommend keeping this default. If you change it to something else, you will need to adjust the path in the rest of the setup steps.
2. Add the following to `~/.bashrc` so that `vivado` is available when `ssh`-ing into the machine:

```
source /tools/Xilinx/Vivado/2021.1/settings64.sh
```

3. Download the `au280` board support package directory from https://github.com/Xilinx/open-nic-shell/tree/main/board_files/Xilinx and place the directory in `/tools/Xilinx/Vivado/2021.1/data/xhub/boards/XilinxBoardStore/boards/Xilinx/`.

If you have multiple Build Farm Machines, you should repeat this process for each.

8.8.2 2. Verify Build Farm Machine environment

Machines: Manager Machine and Run Farm Machines

Finally, let's ensure that Vivado 2021.1 is properly sourced in your shell setup (i.e. `.bashrc`) so that any shell on your Build Farm Machines can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned.

You can check this by running the following on the Manager Machine, replacing `BUILD_FARM_IP` with `localhost` if your Build Farm machine and Manager machine are the same machine, or replacing it with the Build Farm machine's IP address if they are different machines.

```
ssh BUILD_FARM_IP printenv
```

Ensure that the output of the command shows that the Vivado 2021.1 tools are present in the printed environment variables (i.e., `PATH` and `XILINX_VIVADO`).

If you have multiple Build Farm machines, you should repeat this process for each Build Farm machine, replacing `BUILD_FARM_IP` with a different Build Farm Machine's IP address.

8.9 Configuring a Build in the Manager

In the `deploy/config_build.yaml` file, you will notice that the `builds_to_run` section currently contains several lines, which indicates to the build system that you want to run all of these "build recipes" in parallel, with the parameters for each "build recipe" listed in the relevant section of the `deploy/config_build_recipes.yaml` file (or in the case of building a Chipyard target SoC in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml).

In this guide, we'll build the default FireSim design for the Xilinx Alveo U280, which is specified by the `alveo_u280_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. This was the same configuration used to build the pre-built bitstream that you used to run simulations in the guide to running a simulation.

Looking at the `alveo_u280_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml, there are a few notable items:

- TARGET_CONFIG specifies that this configuration is a simple singlecore RISC-V Rocket with a single DRAM channel.
- TARGET_PROJECT_MAKEFRAG specifies to the FireSim Make system how to build the TARGET_CONFIG.
- bit_builder_recipe points to bit-builder-recipes/xilinx_alveo_u280.yaml, which is found in the deploy directory and tells the FireSim build system how to build bitstreams for this FPGA.

Having looked at this entry, let's now set up the build in deploy/config_build.yaml. First, we'll set up the build_farm mapping, which specifies the Build Farm Machines that are available to build FPGA bitstreams.

- base_recipe will map to build-farm-recipes/externally_provisioned.yaml. This indicates to the FireSim manager that the machines used to run builds are existing machines that have been set up by the user, instead of cloud instances that are automatically provisioned.
- default_build_dir is the directory in which builds will run out of on your Build Farm Machines. Change the default null to a path where you would like temporary build data to be stored on your Build Farm Machines.
- build_farm_hosts is a section that contains a list of IP addresses or hostnames of machines in your Build Farm. By default, localhost is specified. If you are using a separate Build Farm Machine, you should replace this with the IP address or hostname of the Build Farm Machine on which you would like to run the build.

Having configured our Build Farm, let's specify the design we'd like to build. To do this, edit the builds_to_run section in deploy/config_build.yaml so that it looks like the following:

```
builds_to_run:
    - alveo_u280_firesim_rocket_singlecore_no_nic
```

In essence, you should delete or comment out all the other items in the builds_to_run section besides alveo_u280_firesim_rocket_singlecore_no_nic.

8.10 Running the Build

Now, we can run a build like so (while also pointing to Chipyard's recipes provided):

```
firesim buildbitstream -r -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.
↪yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing a bitstream that runs on the Xilinx Alveo U280 FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in deploy/results-build/, named after your build parameter settings, that contains all of the outputs of the Xilinx Vivado build process. Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems).

The manager will also print an entry that can be added to config_hwdb.yaml so that the bitstream can be used to run simulations. This entry will contain a bitstream_tar key whose value is the path to the final generated bitstream file. You can share generated bitstreams with others by sharing the file listed in bitstream_tar and the config_hwdb.yaml entry for it.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically!

This is the end of the Getting Started Guide. To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

CHAPTER

NINE

XILINX VCU118 XDMA-BASED GETTING STARTED GUIDE

The getting started guides that follow this page will walk you through the complete (XDMA-based) flow for getting an example FireSim simulation up and running using an on-premises Xilinx VCU118 FPGA, from scratch.

Make sure you have run/done the steps listed in *Local FPGA System Setup* before running this guide.

First, we'll set up your environment, then run a simulation of a single RISC-V Rocket-based SoC booting Linux, using a pre-built bitstream. Next, we'll show you how to build your own FPGA bitstreams for a custom hardware design. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

Here's a high-level outline of what we'll be doing in this guide:

9.1 FPGA Setup

The following installation steps are FPGA-specific and should be run on all **run farm machines** that install an FPGA. You might need `sudo` access to setup the FPGA.

1. Poweroff your machine.
2. Insert your Xilinx VCU118 FPGA into an open PCIe slot in the machine. Also, ensure that the SW16 switches on the board are set to `0101` to enable QSPI flashing over JTAG (i.e., `position 1 = 0`, `position 2 = 1`, `position 3 = 0`, and `position 4 = 1`. Having the switch set to the side of the position label indicates 0.)
3. Attach any additional power cables between the FPGA and the host machine. For the VCU118, this is usually ATX 4-pin peripheral power (**NOT** PCIe power) from the system's PSU, attached to the FPGA via the "ATX Power Supply Adapter Cable" that comes with the VCU118.
4. Attach the USB cable between the FPGA and the host machine for JTAG.
5. Boot the machine.
6. Obtain an existing bitstream tar file for your FPGA by opening the `bitstream_tar` URL listed under `xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic` in the following file: `${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml`.
7. Download/extract the `.tar.gz` file to a known location. Inside, you will find four files; the ones we are currently interested in will be called `firesim.mcs` and `firesim_secondary.mcs`. Note the full path of the `firesim.mcs` and `firesim_secondary.mcs` files for the next step.

8. Open Vivado Lab and click "Open Hardware Manager". Then click "Open Target" and "Auto connect".
9. Right-click on your FPGA and click "Add Configuration Memory Device". For a Xilinx VCU118, choose `mt25qu01g-spi-x1_x2_x4_x8` as the Configuration Memory Part. Click "OK" when prompted to program the configuration memory device.
10. For Configuration file, choose the `firesim.mcs` file from step 7 and for Configuration file 2, choose the `firesim_secondary.mcs` file from step 7.
11. Uncheck "Verify" and click OK.
12. Right-click on your FPGA and click "Boot from Configuration Memory Device".
13. When programming the configuration memory device is completed, power off your machine fully (i.e., the FPGA should completely lose power). Then, set the SW16 switches on the board to `0001` to set the board to automatically program the FPGA from the QSPI at boot (i.e., `position 1 = 0`, `position 2 = 0`, `position 3 = 0`, and `position 4 = 1`. Having the switch set to the side of the position label indicates 0.))
14. Cold-boot the machine. A cold boot is required for the FPGA to be successfully re-programmed from its flash.
15. Once the machine has booted, run the following to ensure that your FPGA is set up properly:

```
lspci -vvv -d 10ee:903f
```

If successful, this should show an entry with Xilinx as the manufacturer and two memory regions. There should be one entry for each FPGA you've added to the Run Farm Machine.

Note

If necessary, you can remove the USB cable for JTAG (the FPGA is programmed through PCIe for FireSim simulations on the Xilinx VCU118). However, we still recommend leaving the cable attached, since it will allow you to re-flash the SPI in case there are issues.

9.2 FireSim Repo Setup

Next, we'll clone FireSim through Chipyard on your Manager Machine and run a few final setup steps using scripts in the repo.

9.2.1 Setting up the FireSim Repo

Machine: From this point forward, run everything on your Manager Machine, unless otherwise instructed.

We're finally ready to fetch FireSim's sources through Chipyard. Chipyard provides all the necessary target designs (e.g. RISC-V SoCs) and software (e.g. Linux) used for the rest of this guide.

Note

This guide was built using Chipyard version dbc082e2206f787c3aba12b9b171e1704e15b707. It is recommended to use the most up-to-date version of Chipyard with this tutorial.

This should be done on your Manager Machine. Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
./build-setup.sh
```

Once `build-setup.sh` completes, run:

```
cd sims/firesim
source sourceme-manager.sh --skip-ssh-setup
```

This will perform various environment setup steps, such as adding the RISC-V tools to your path. Sourcing this the first time will take some time – however each subsequent sourcing should be instantaneous.

> **Warning**
>
> **Every time you want to use FireSim, you should `cd` into your FireSim directory and source `sourceme-manager.sh` again with the arguments shown above.**

9.2.2 Initializing FireSim Config Files

The FireSim manager contains a command that will automatically provide a fresh set of configuration files for a given platform.

To run it, do the following:

```
firesim managerinit --platform xilinx_vcu118
```

This will produce several initial configuration files, which we will edit in the next section.

9.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup

As our final setup step, we will edit FireSim's configuration files so that the manager understands our Run Farm machine setup and the set of FPGAs attached to each Run Farm machine.

Inside the cloned FireSim repo, open up the `deploy/config_runtime.yaml` file and set the following keys to the indicated values:

- `default_simulation_dir` should point to a temporary simulation directory of your choice on your Run Farm Machines. This is the directory that simulations will run out of.
- `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system.

Here are two examples of how this could be configured:

Example 1: Your Run Farm has a single machine with one FPGA attached and this machine is also your Manager Machine:

```
...
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec
...
```

Example 2: You have two Run Farm Machines (separate from your Manager Machine). The Run Farm Machines are accessible from your manager machine with the hostnames `firesim-runner1.berkeley.edu` and `firesim-runner2.berkeley.edu`, each with eight FPGAs attached.

```
...
    run_farm_hosts_to_use:
        - firesim-runner1.berkeley.edu: eight_fpgas_spec
        - firesim-runner2.berkeley.edu: eight_fpgas_spec
...
```

- `default_hw_config` should be `xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic`

Then, run the following command so that FireSim can generate a mapping from the FPGA ID used for JTAG programming to the PCIe ID used to run simulations. If you ever change the physical layout of the machine (e.g., which PCIe slot the FPGAs are attached to), you will need to re-run this command.

```
firesim enumeratefpgas
```

This will generate a database file in `/opt/firesim-db.json` on each Run Farm Machine that has this mapping.

Now you're ready to run your first FireSim simulation! Hit Next to continue with the guide.

9.3 Running a Single Node Simulation

Now that we've completed all the basic setup steps, it's time to run a simulation! In this section, we will simulate a single target node, for which we will use a single Xilinx VCU118.

Make sure you have sourced `sourceme-manager.sh --skip-ssh-setup` **before running any of these commands.**

9.3.1 Building target software

In this guide, we'll boot Linux on our simulated node. To do so, we'll need to build our RISC-V SoC-compatible Linux distro. For this guide, we will use a simple buildroot-based distribution. We can build the Linux distribution like so:

```
# assumes you already cd'd into your firesim repo
# and sourced sourceme-manager.sh
#
# then:
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the RISC-V SoC we will simulate.

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for the RISC-V SoC we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or directly as a basic, interactive Linux distribution.

9.4 Setting up the manager configuration

All runtime configuration options for the manager are located in `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

Based on the changes we made earlier, this file will already have everything set correctly to run a simulation.

Below we'll highlight a few of these lines to explain what is happening:

- At the top, you'll notice the `run_farm` mapping, which describes and specifies the machines to run simulations on.
  - By default, we'll be using a `base_recipe` of `run-farm-recipes/externally_provisioned.yaml`, which means that our Run Farm machines are pre-configured, and do not require the manager to dynamically launch/terminate them (e.g., as we would do for cloud instances).
  - The `default_platform` has automatically been set for our FPGA board, to `XilinxVCU118InstanceDeployManager`.
  - The `default_simulation_dir` is the directory on the Run Farm Machines where simulations will run out of. The default is likely fine, but you can change it to any directory you have access to on the Run Farm machines.
  - `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system. We configured this already in the previous step.
- The `target_config` section describes the system that we'd like to simulate.
  - `topology: no_net_config` indicates that we're running simulations with no network between them.
  - `no_net_num_nodes: 1` indicates that we'll be a simulation of a single SoC
  - The `default_hw_config` will be set to a pre-built design. **Change this to `xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic` to simulate a single RISC-V Rocket core SoC**.
- The `workload` section describes the workload that we'd like to run on our simulated systems. In this case, we need to change it to boot Linux on all SoCs in the simulation. **Change the line to the following**: `workload_name: br-base-uniform.json`.

9.5 Building and Deploying simulation infrastructure to the Run Farm Machines

The manager automates the process of building and deploying all components necessary to run your simulation on the Run Farm, including programming FPGAs. To tell the manager to setup all of our simulation infrastructure, run the following:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: https://docs.fires.im
Running: infrasetup

Building FPGA software driver.
...
[localhost] Checking if host instance is up...
[localhost] Copying FPGA simulation infrastructure for slot: 0.
[localhost] Clearing all FPGA Slots.
The full log of this run is:
.../firesim/deploy/logs/2023-03-06--01-22-46-infrasetup-35ZP4WUOX8KUYBF3.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `${FS_DIR}/deploy/logs/`.

At this point, our single Run Farm machine has all the infrastructure necessary to run a simulation, so let's launch our simulation!

9.6 Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: https://docs.fires.im
Running: runworkload

Creating the directory: .../firesim/deploy/results-workload/2023-03-06--01-25-34-br-base/
[localhost] Checking if host instance is up...
[localhost] Starting FPGA simulation for slot: 0.
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
```

(continues on next page)

(continued from previous page)

```
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have powered off. So, let's let it run and open another terminal on the manager machine. From there, `cd` into your FireSim directory again and `source sourceme-manager.sh --skip-ssh-setup`.

Next, let's `ssh` into the Run Farm machine. If your Run Farm and Manager Machines are the same, replace `RUN_FARM_IP_OR_HOSTNAME` with `localhost`, otherwise replace it with your Run Farm Machine's IP or hostname.

```
source ~/.ssh/AGENT_VARS
ssh RUN_FARM_IP_OR_HOSTNAME
```

Next, we can directly attach to the console of the simulated system using `screen`, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
```

(continues on next page)

(continued from previous page)

```
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's power off the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
$ ls -la firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/*/*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back from the Run Farm Machine into the `results-workload` directory on our manager machine, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left.

Click Next if you'd like to continue on to building your own bitstreams.

9.7 Building Your Own Hardware Designs

This section will guide you through building a Xilinx VCU118 FPGA bitstream to run FireSim simulations.

9.8 System Setup

Here, we'll do some final one-time setup for your Build Farm Machines so that we can build bitstreams for FireSim simulations automatically.

These steps assume that you have already followed the earlier setup steps required to run simulations.

As noted earlier, it is highly recommended that you use Ubuntu 20.04 LTS as the host operating system for all machine types in an on-premises setup, as this is the OS recommended by Xilinx.

Also recall that we make a distinction between the Manager Machine, the Build Farm Machine(s), and the Run Farm Machine(s). In a simple setup, these can all be a single machine, in which case you should run the Build Farm Machine setup steps below on your single machine.

9.8.1 1. Install Vivado for Builds

Machines: Build Farm Machines.

Running builds for Xilinx VCU118 in FireSim requires Vivado 2023.1. Other versions are unlikely to work out-of-the-box.

On each Build Farm machine, do the following:

1. Install Vivado 2023.1 from the Xilinx Downloads Website. By default, Vivado will be installed to `/tools/Xilinx/Vivado/2023.1`. We recommend keeping this default. If you change it to something else, you will need to adjust the path in the rest of the setup steps.
2. Add the following to `~/.bashrc` so that `vivado` is available when `ssh`-ing into the machine:

```
source /tools/Xilinx/Vivado/2023.1/settings64.sh
```

3. No special board support package is required for the VCU118. Move on to the next step.

If you have multiple Build Farm Machines, you should repeat this process for each.

9.8.2 2. Verify Build Farm Machine environment

Machines: Manager Machine and Run Farm Machines

Finally, let's ensure that Vivado 2023.1 is properly sourced in your shell setup (i.e. `.bashrc`) so that any shell on your Build Farm Machines can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned.

You can check this by running the following on the Manager Machine, replacing `BUILD_FARM_IP` with `localhost` if your Build Farm machine and Manager machine are the same machine, or replacing it with the Build Farm machine's IP address if they are different machines.

```
ssh BUILD_FARM_IP printenv
```

Ensure that the output of the command shows that the Vivado 2023.1 tools are present in the printed environment variables (i.e., PATH and XILINX_VIVADO).

If you have multiple Build Farm machines, you should repeat this process for each Build Farm machine, replacing BUILD_FARM_IP with a different Build Farm Machine's IP address.

9.9 Configuring a Build in the Manager

In the deploy/config_build.yaml file, you will notice that the builds_to_run section currently contains several lines, which indicates to the build system that you want to run all of these "build recipes" in parallel, with the parameters for each "build recipe" listed in the relevant section of the deploy/config_build_recipes.yaml file (or in the case of building a Chipyard target SoC in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml).

In this guide, we'll build the default FireSim design for the Xilinx VCU118, which is specified by the xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. This was the same configuration used to build the pre-built bitstream that you used to run simulations in the guide to running a simulation.

Looking at the xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml, there are a few notable items:

- TARGET_CONFIG specifies that this configuration is a simple singlecore RISC-V Rocket with a single DRAM channel.
- TARGET_PROJECT_MAKEFRAG specifies to the FireSim Make system how to build the TARGET_CONFIG.
- bit_builder_recipe points to bit-builder-recipes/xilinx_vcu118.yaml, which is found in the deploy directory and tells the FireSim build system how to build bitstreams for this FPGA.

Having looked at this entry, let's now set up the build in deploy/config_build.yaml. First, we'll set up the build_farm mapping, which specifies the Build Farm Machines that are available to build FPGA bitstreams.

- base_recipe will map to build-farm-recipes/externally_provisioned.yaml. This indicates to the FireSim manager that the machines used to run builds are existing machines that have been set up by the user, instead of cloud instances that are automatically provisioned.
- default_build_dir is the directory in which builds will run out of on your Build Farm Machines. Change the default null to a path where you would like temporary build data to be stored on your Build Farm Machines.
- build_farm_hosts is a section that contains a list of IP addresses or hostnames of machines in your Build Farm. By default, localhost is specified. If you are using a separate Build Farm Machine, you should replace this with the IP address or hostname of the Build Farm Machine on which you would like to run the build.

Having configured our Build Farm, let's specify the design we'd like to build. To do this, edit the builds_to_run section in deploy/config_build.yaml so that it looks like the following:

```
builds_to_run:
    - xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic
```

In essence, you should delete or comment out all the other items in the builds_to_run section besides xilinx_vcu118_firesim_rocket_singlecore_4GB_no_nic.

9.10 Running the Build

Now, we can run a build like so (while also pointing to Chipyard's recipes provided):

```
firesim buildbitstream -r -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.
↪yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing a bitstream that runs on the Xilinx VCU118 FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in `deploy/results-build/`, named after your build parameter settings, that contains all of the outputs of the Xilinx Vivado build process. Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems).

The manager will also print an entry that can be added to `config_hwdb.yaml` so that the bitstream can be used to run simulations. This entry will contain a `bitstream_tar` key whose value is the path to the final generated bitstream file. You can share generated bitstreams with others by sharing the file listed in `bitstream_tar` and the `config_hwdb.yaml` entry for it.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically!

This is the end of the Getting Started Guide. To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

CHAPTER

TEN

RHS RESEARCH NITEFURY II XDMA-BASED GETTING STARTED GUIDE

The getting started guides that follow this page will walk you through the complete (XDMA-based) flow for getting an example FireSim simulation up and running using an on-premises RHS Research Nitefury II FPGA, from scratch.

Make sure you have run/done the steps listed in *Local FPGA System Setup* before running this guide.

First, we'll set up your environment, then run a simulation of a single RISC-V Rocket-based SoC booting Linux, using a pre-built bitstream. Next, we'll show you how to build your own FPGA bitstreams for a custom hardware design. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

Here's a high-level outline of what we'll be doing in this guide:

10.1 FPGA Setup

The following installation steps are FPGA-specific and should be run on all **run farm machines** that install an FPGA. You might need `sudo` access to setup the FPGA.

1. Poweroff your machine.
2. Insert your RHS Research Nitefury II FPGA into either an open M.2. slot on your machine or into an M.2. to Thunderbolt enclosure (then attach the enclosure to your system via a Thunderbolt cable). We have successfully used this enclosure: https://www.amazon.com/ORICO-Enclosure-Compatible-Thunderbolt-Type-C-M2V01/dp/B08R9DMFFT. Before permanently installing your Nitefury into your M.2. slot or enclosure, ensure that you have attached the ribbon cable that will be used for JTAG to the underside of the board (see step 4 below).
3. Attach any additional power cables between the FPGA and the host machine. This step is not required for the Nitefury, since all power is delivered via M.2. or Thunderbolt.
4. Attach the USB cable between the FPGA and the host machine for JTAG. For the Nitefury, this requires attaching the 14-pin JTAG adapter included with the board to the board using the included ribbon cable, then attaching a USB to JTAG adapter such as the Digilent HS2: https://digilent.com/shop/jtag-hs2-programming-cable/.
5. Boot the machine.
6. Obtain an existing bitstream tar file for your FPGA by opening the `bitstream_tar` URL listed under `nitefury_firesim_rocket_singlecore_no_nic` in the following file: `${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml`.

7. Download/extract the `.tar.gz` file to a known location. Inside, you will find three files; the one we are currently interested in will be called `firesim.mcs`. Note the full path of this `firesim.mcs` file for the next step.
8. Open Vivado Lab and click "Open Hardware Manager". Then click "Open Target" and "Auto connect".
9. Right-click on your FPGA and click "Add Configuration Memory Device". For a RHS Research Nitefury II, choose `s25fl256xxxxxx0-spi-x1_x2_x4` as the Configuration Memory Part. Click "OK" when prompted to program the configuration memory device.
10. For Configuration file, choose the `firesim.mcs` file from step 7.
11. Uncheck "Verify" and click OK.
12. Right-click on your FPGA and click "Boot from Configuration Memory Device".
13. When programming the configuration memory device is completed, power off your machine fully (i.e., the FPGA should completely lose power).)
14. Cold-boot the machine. A cold boot is required for the FPGA to be successfully re-programmed from its flash.
15. Once the machine has booted, run the following to ensure that your FPGA is set up properly:

```
lspci -vvv -d 10ee:903f
```

If successful, this should show an entry with Xilinx as the manufacturer and two memory regions. There should be one entry for each FPGA you've added to the Run Farm Machine.

Note

Remember to keep the USB cable for JTAG connected at all times when running FireSim simulations (it is used to program the FPGA).

10.2 FireSim Repo Setup

Next, we'll clone FireSim through Chipyard on your Manager Machine and run a few final setup steps using scripts in the repo.

10.2.1 Setting up the FireSim Repo

Machine: From this point forward, run everything on your Manager Machine, unless otherwise instructed.

We're finally ready to fetch FireSim's sources through Chipyard. Chipyard provides all the necessary target designs (e.g. RISC-V SoCs) and software (e.g. Linux) used for the rest of this guide.

Note

This guide was built using Chipyard version dbc082e2206f787c3aba12b9b171e1704e15b707. It is recommended to use the most up-to-date version of Chipyard with this tutorial.

This should be done on your Manager Machine. Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
./build-setup.sh
```

Once `build-setup.sh` completes, run:

```
cd sims/firesim
source sourceme-manager.sh --skip-ssh-setup
```

This will perform various environment setup steps, such as adding the RISC-V tools to your path. Sourcing this the first time will take some time – however each subsequent sourcing should be instantaneous.

Warning

Every time you want to use FireSim, you should `cd` into your FireSim directory and source `sourceme-manager.sh` again with the arguments shown above.

10.2.2 Initializing FireSim Config Files

The FireSim manager contains a command that will automatically provide a fresh set of configuration files for a given platform.

To run it, do the following:

```
firesim managerinit --platform rhsresearch_nitefury_ii
```

This will produce several initial configuration files, which we will edit in the next section.

10.2.3 Configuring the FireSim manager to understand your Run Farm Machine setup

As our final setup step, we will edit FireSim's configuration files so that the manager understands our Run Farm machine setup and the set of FPGAs attached to each Run Farm machine.

Inside the cloned FireSim repo, open up the `deploy/config_runtime.yaml` file and set the following keys to the indicated values:

- `default_simulation_dir` should point to a temporary simulation directory of your choice on your Run Farm Machines. This is the directory that simulations will run out of.
- `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system.

Here are two examples of how this could be configured:

Example 1: Your Run Farm has a single machine with one FPGA attached and this machine is also your Manager Machine:

```
...
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec
...
```

Example 2: You have two Run Farm Machines (separate from your Manager Machine). The Run Farm Machines are accessible from your manager machine with the hostnames `firesim-runner1.berkeley.edu` and `firesim-runner2.berkeley.edu`, each with eight FPGAs attached.

```
...
    run_farm_hosts_to_use:
        - firesim-runner1.berkeley.edu: eight_fpgas_spec
        - firesim-runner2.berkeley.edu: eight_fpgas_spec
...
```

- `default_hw_config` should be `nitefury_firesim_rocket_singlecore_no_nic`

Then, run the following command so that FireSim can generate a mapping from the FPGA ID used for JTAG programming to the PCIe ID used to run simulations. If you ever change the physical layout of the machine (e.g., which PCIe slot the FPGAs are attached to), you will need to re-run this command.

```
firesim enumeratefpgas
```

This will generate a database file in `/opt/firesim-db.json` on each Run Farm Machine that has this mapping.

Now you're ready to run your first FireSim simulation! Hit Next to continue with the guide.

10.3 Running a Single Node Simulation

Now that we've completed all the basic setup steps, it's time to run a simulation! In this section, we will simulate a single target node, for which we will use a single RHS Research Nitefury II.

Make sure you have sourced `sourceme-manager.sh --skip-ssh-setup` **before running any of these commands.**

10.3.1 Building target software

In this guide, we'll boot Linux on our simulated node. To do so, we'll need to build our RISC-V SoC-compatible Linux distro. For this guide, we will use a simple buildroot-based distribution. We can build the Linux distribution like so:

```
# assumes you already cd'd into your firesim repo
# and sourced sourceme-manager.sh
#
# then:
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the RISC-V SoC we will simulate.

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for the RISC-V SoC we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or directly as a basic, interactive Linux distribution.

10.4 Setting up the manager configuration

All runtime configuration options for the manager are located in `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

Based on the changes we made earlier, this file will already have everything set correctly to run a simulation.

Below we'll highlight a few of these lines to explain what is happening:

- At the top, you'll notice the `run_farm` mapping, which describes and specifies the machines to run simulations on.
  - By default, we'll be using a `base_recipe` of `run-farm-recipes/externally_provisioned.yaml`, which means that our Run Farm machines are pre-configured, and do not require the manager to dynamically launch/terminate them (e.g., as we would do for cloud instances).
  - The `default_platform` has automatically been set for our FPGA board, to `RHSResearchNitefuryIIInstanceDeployManager`.
  - The `default_simulation_dir` is the directory on the Run Farm Machines where simulations will run out of. The default is likely fine, but you can change it to any directory you have access to on the Run Farm machines.
  - `run_farm_hosts_to_use` should be a list of `- IP-address: machine_spec` pairs, one pair for each of your Run Farm Machines. `IP-address` should be the IP address or hostname of the system (that the Manager Machine can use to ssh into the Run Farm Machine) and the `machine_spec` should be a value from `run_farm_host_specs` in deploy/run-farm-recipes/externally_provisioned.yaml. Each spec describes the number of FPGAs attached to a system and other properties about the system. We configured this already in the previous step.
- The `target_config` section describes the system that we'd like to simulate.
  - `topology: no_net_config` indicates that we're running simulations with no network between them.
  - `no_net_num_nodes: 1` indicates that we'll be a simulation of a single SoC
  - The `default_hw_config` will be set to a pre-built design. **Change this to `nitefury_firesim_rocket_singlecore_no_nic` to simulate a single RISC-V Rocket core SoC**.
- The `workload` section describes the workload that we'd like to run on our simulated systems. In this case, we need to change it to boot Linux on all SoCs in the simulation. **Change the line to the following**: `workload_name: br-base-uniform.json`.

10.5 Building and Deploying simulation infrastructure to the Run Farm Machines

The manager automates the process of building and deploying all components necessary to run your simulation on the Run Farm, including programming FPGAs. To tell the manager to setup all of our simulation infrastructure, run the following:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: https://docs.fires.im
Running: infrasetup

Building FPGA software driver.
...
[localhost] Checking if host instance is up...
[localhost] Copying FPGA simulation infrastructure for slot: 0.
[localhost] Clearing all FPGA Slots.
The full log of this run is:
.../firesim/deploy/logs/2023-03-06--01-22-46-infrasetup-35ZP4WUOX8KUYBF3.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `${FS_DIR}/deploy/logs/`.

At this point, our single Run Farm machine has all the infrastructure necessary to run a simulation, so let's launch our simulation!

10.6 Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: https://docs.fires.im
Running: runworkload

Creating the directory: .../firesim/deploy/results-workload/2023-03-06--01-25-34-br-base/
[localhost] Checking if host instance is up...
[localhost] Starting FPGA simulation for slot: 0.
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
```

(continues on next page)

(continued from previous page)

```
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have powered off. So, let's let it run and open another terminal on the manager machine. From there, `cd` into your FireSim directory again and `source sourceme-manager.sh --skip-ssh-setup`.

Next, let's `ssh` into the Run Farm machine. If your Run Farm and Manager Machines are the same, replace `RUN_FARM_IP_OR_HOSTNAME` with `localhost`, otherwise replace it with your Run Farm Machine's IP or hostname.

```
source ~/.ssh/AGENT_VARS
ssh RUN_FARM_IP_OR_HOSTNAME
```

Next, we can directly attach to the console of the simulated system using `screen`, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
```

(continues on next page)

(continued from previous page)

```
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's power off the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
$ ls -la firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/*/*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back from the Run Farm Machine into the `results-workload` directory on our manager machine, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left.

Click Next if you'd like to continue on to building your own bitstreams.

10.7 Building Your Own Hardware Designs

This section will guide you through building a RHS Research Nitefury II FPGA bitstream to run FireSim simulations.

10.8 System Setup

Here, we'll do some final one-time setup for your Build Farm Machines so that we can build bitstreams for FireSim simulations automatically.

These steps assume that you have already followed the earlier setup steps required to run simulations.

As noted earlier, it is highly recommended that you use Ubuntu 20.04 LTS as the host operating system for all machine types in an on-premises setup, as this is the OS recommended by Xilinx.

Also recall that we make a distinction between the Manager Machine, the Build Farm Machine(s), and the Run Farm Machine(s). In a simple setup, these can all be a single machine, in which case you should run the Build Farm Machine setup steps below on your single machine.

10.8.1 1. Install Vivado for Builds

Machines: Build Farm Machines.

Running builds for RHS Research Nitefury II in FireSim requires Vivado 2022.1. Other versions are unlikely to work out-of-the-box.

On each Build Farm machine, do the following:

1. Install Vivado 2022.1 from the Xilinx Downloads Website. By default, Vivado will be installed to `/tools/Xilinx/Vivado/2022.1`. We recommend keeping this default. If you change it to something else, you will need to adjust the path in the rest of the setup steps.
2. Add the following to `~/.bashrc` so that `vivado` is available when `ssh`-ing into the machine:

```
source /tools/Xilinx/Vivado/2022.1/settings64.sh
```

3. No special board support package is required for the Nitefury II. Move on to the next step.

If you have multiple Build Farm Machines, you should repeat this process for each.

10.8.2 2. Verify Build Farm Machine environment

Machines: Manager Machine and Run Farm Machines

Finally, let's ensure that Vivado 2022.1 is properly sourced in your shell setup (i.e. `.bashrc`) so that any shell on your Build Farm Machines can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned.

You can check this by running the following on the Manager Machine, replacing `BUILD_FARM_IP` with `localhost` if your Build Farm machine and Manager machine are the same machine, or replacing it with the Build Farm machine's IP address if they are different machines.

```
ssh BUILD_FARM_IP printenv
```

Ensure that the output of the command shows that the Vivado 2022.1 tools are present in the printed environment variables (i.e., PATH and XILINX_VIVADO).

If you have multiple Build Farm machines, you should repeat this process for each Build Farm machine, replacing BUILD_FARM_IP with a different Build Farm Machine's IP address.

10.9 Configuring a Build in the Manager

In the deploy/config_build.yaml file, you will notice that the builds_to_run section currently contains several lines, which indicates to the build system that you want to run all of these "build recipes" in parallel, with the parameters for each "build recipe" listed in the relevant section of the deploy/config_build_recipes.yaml file (or in the case of building a Chipyard target SoC in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml).

In this guide, we'll build the default FireSim design for the RHS Research Nitefury II, which is specified by the nitefury_firesim_rocket_singlecore_no_nic section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. This was the same configuration used to build the pre-built bitstream that you used to run simulations in the guide to running a simulation.

Looking at the nitefury_firesim_rocket_singlecore_no_nic section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml, there are a few notable items:

- TARGET_CONFIG specifies that this configuration is a simple singlecore RISC-V Rocket with a single DRAM channel.
- TARGET_PROJECT_MAKEFRAG specifies to the FireSim Make system how to build the TARGET_CONFIG.
- bit_builder_recipe points to bit-builder-recipes/rhsresearch_nitefury_ii.yaml, which is found in the deploy directory and tells the FireSim build system how to build bitstreams for this FPGA.

Having looked at this entry, let's now set up the build in deploy/config_build.yaml. First, we'll set up the build_farm mapping, which specifies the Build Farm Machines that are available to build FPGA bitstreams.

- base_recipe will map to build-farm-recipes/externally_provisioned.yaml. This indicates to the FireSim manager that the machines used to run builds are existing machines that have been set up by the user, instead of cloud instances that are automatically provisioned.
- default_build_dir is the directory in which builds will run out of on your Build Farm Machines. Change the default null to a path where you would like temporary build data to be stored on your Build Farm Machines.
- build_farm_hosts is a section that contains a list of IP addresses or hostnames of machines in your Build Farm. By default, localhost is specified. If you are using a separate Build Farm Machine, you should replace this with the IP address or hostname of the Build Farm Machine on which you would like to run the build.

Having configured our Build Farm, let's specify the design we'd like to build. To do this, edit the builds_to_run section in deploy/config_build.yaml so that it looks like the following:

```
builds_to_run:
    - nitefury_firesim_rocket_singlecore_no_nic
```

In essence, you should delete or comment out all the other items in the builds_to_run section besides nitefury_firesim_rocket_singlecore_no_nic.

10.10 Running the Build

Now, we can run a build like so (while also pointing to Chipyard's recipes provided):

```
firesim buildbitstream -r -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.
↪yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing a bitstream that runs on the RHS Research Nitefury II FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in `deploy/results-build/`, named after your build parameter settings, that contains all of the outputs of the Xilinx Vivado build process. Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems).

The manager will also print an entry that can be added to `config_hwdb.yaml` so that the bitstream can be used to run simulations. This entry will contain a `bitstream_tar` key whose value is the path to the final generated bitstream file. You can share generated bitstreams with others by sharing the file listed in `bitstream_tar` and the `config_hwdb.yaml` entry for it.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically!

This is the end of the Getting Started Guide. To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

Warning

We highly recommend using the XDMA-based U250 flow instead of this Vitis-based flow. You can find the XDMA-based flow here: *Xilinx Alveo U250 XDMA-based Getting Started Guide*. The Vitis-based flow does not support DMA-based FireSim bridges (e.g., TracerV, Synthesizable Printfs, etc.), while the XDMA-based flows support all FireSim features. If you're unsure, use the XDMA-based U250 flow instead: *Xilinx Alveo U250 XDMA-based Getting Started Guide*

CHAPTER

ELEVEN

(EXPERIMENTAL) XILINX ALVEO U250 VITIS-BASED GETTING STARTED GUIDE

The getting started guides that follow this page will walk you through the complete (Vitis-based) flow for getting an example FireSim simulation up and running using an on-premises Xilinx Alveo U250 FPGA, from scratch.

Make sure you have run/done the steps listed in *Local FPGA System Setup* before running this guide.

First, we'll set up your environment, then run a simulation of a single RISC-V Rocket-based SoC booting Linux, using a pre-built bitstream. Next, we'll show you how to build your own FPGA bitstreams for a custom hardware design. After you complete these guides, you can look at the "Advanced Docs" in the sidebar to the left.

Note

This section uses ${CY_DIR} and ${FS_DIR} to refer to the Chipyard and FireSim directories. These are set when sourcing the Chipyard and FireSim environments.

Here's a high-level outline of what we'll be doing in this guide:

1. **FPGA Setup**: Installing the FPGA board and relevant software.
2. **On-Premises Machine Setup**
   1. Setting up a "Manager Machine" from which you will coordinate building and deploying simulations locally.
3. **Single-node simulation guide**: This guide walks you through the process of running a simulation locally on a single Xilinx Alveo U250, using a pre-built, public bitstream.
4. **Building your own hardware designs guide (Chisel to FPGA Image)**: This guide walks you through the full process of taking Rocket Chip RTL and any custom RTL plugged into Rocket Chip and producing a FireSim bitstream to plug into your simulations. This automatically runs Chisel elaboration, FAME-1 Transformation, and the Xilinx Vitis FPGA flow.

Generally speaking, you only need to follow Step 4 if you're modifying Chisel RTL or changing non-runtime configurable hardware parameters.

11.1 Initial Setup/Installation

Warning

We highly recommend using the XDMA-based U250 flow instead of this Vitis-based flow. You can find the XDMA-based flow here: *Xilinx Alveo U250 XDMA-based Getting Started Guide*. The Vitis-based flow does not support DMA-based FireSim bridges (e.g., TracerV, Synthesizable Printfs, etc.), while the XDMA-based flows support all FireSim features. If you're unsure, use the XDMA-based U250 flow instead: *Xilinx Alveo U250 XDMA-based Getting Started Guide*

11.1.1 FPGA and Tool Setup

Requirements and Installations

We require a base machine that is able to support a Xilinx Vitis-enabled U250 and running Xilinx Vitis. For the purposes of this guide, we assume you are running with a Xilinx Vitis-enabled U250. Please refer to the minimum system requirements given in the following link: https://docs.xilinx.com/r/en-US/ug1301-getting-started-guide-alveo-accelerator-cards/Minimum-System-Requirements. sudo access is not needed for the machine except for when the Xilinx Vitis-enabled U250 and corresponding software is installed.

Next, install the Xilinx Vitis-enabled U250 as indicated: https://docs.xilinx.com/r/en-US/ug1301-getting-started-guide-alveo-accelerator-cards/Card-Installation-Procedures

We require the following programs/packages installed from the Xilinx website in addition to a physical Xilinx Vitis-enabled U250 installation:

- Xilinx Vitis 2022.1
  - Installation link: https://www.xilinx.com/products/design-tools/vitis/vitis-whats-new.html#20221
- Xilinx XRT and Xilinx Vitis-enabled U250 board package (corresponding with Vitis 2022.1)
  - Ensure you complete the "Installing the Deployment Software" and "Card Bring-Up and Validation" sections in the following link: https://docs.xilinx.com/r/en-US/ug1301-getting-started-guide-alveo-accelerator-cards/Installing-the-Deployment-Software

Setup Validation

After installing the Xilinx Vitis-enabled U250 using the Xilinx instructions and installing the specific versions of Vitis/XRT, let's verify that the Xilinx Vitis-enabled U250 can be used for emulations. Ensure that you can run the following XRT commands without errors:

```
xbutil examine # obtain the BDF associated with your installed Xilinx Vitis-enabled U250
xbutil validate --device <CARD_BDF_INSTALLED> --verbose
```

The `xbutil validate` command runs simple tests to ensure that the FPGA can be properly flashed with a bitstream by using XRT.

Warning

Anytime the host computer is rebooted you may need to re-run parts of the setup process (i.e. re-flash the shell). Before continuing to FireSim simulations after a host computer reboot, ensure that the previously mentioned `xbutil` command is successful.

Now you're ready to continue with other FireSim setup!

11.1.2 Setting up your On-Premises Machine

This guide will walk you through setting up a single node cluster (i.e. running FPGA bitstream builds and simulations on a single machine) for FireSim use. This single machine will serve as the "Manager Machine" that acts as a "head" node that all work will be completed on.

Finally, ensure that the Xilinx XRT/Vitis tools are sourced in your shell setup (i.e. `.bashrc` and or `.bash_profile`) so that any shell can use the corresponding programs. The environment variables should be visible to any non-interactive shells that are spawned. You can check this by ensuring that the output of the following command shows that the Xilinx XRT/Vitis tools are present in the environment variables (i.e. "XILINX_XRT"):

```
ssh localhost printenv
```

Setting up the FireSim Repo

We're finally ready to fetch FireSim's sources. Run:

```
git clone https://github.com/ucb-bar/chipyard
cd chipyard
# ideally use the main chipyard release instead of this
git checkout dbc082e2206f787c3aba12b9b171e1704e15b707
```

Next run:

```
./build-setup.sh
```

This will have initialized submodules and installed the RISC-V tools and other dependencies.

Next, run:

```
cd sims/firesim
 source sourceme-manager.sh --skip-ssh-setup
```

This will perform various environment setup steps, such as adding the RISC-V tools to your path. Sourcing this the first time will take some time – however each time after that should be instantaneous.

Every time you want to use FireSim, you should `cd` **into your FireSim directory and source this file again with the argument given.**

Completing Setup Using the Manager

The FireSim manager contains a command that will finish the rest of the FireSim setup process. To run it, do the following:

```
firesim managerinit --platform vitis
```

It will create initial configuration files, which we will edit in later sections.

Hit Next to continue with the guide.

Warning

We highly recommend using the XDMA-based U250 flow instead of this Vitis-based flow. You can find the XDMA-based flow here: *Xilinx Alveo U250 XDMA-based Getting Started Guide*. The Vitis-based flow does not support DMA-based FireSim bridges (e.g., TracerV, Synthesizable Printfs, etc.), while the XDMA-based flows support all FireSim features. If you're unsure, use the XDMA-based U250 flow instead: *Xilinx Alveo U250 XDMA-based Getting Started Guide*

11.2 Running a Single Node Simulation

Now that we've completed all the basic setup steps, it's time to run a simulation! In this section, we will simulate a single target node, for which we will use a single Xilinx Vitis-enabled U250.

Make sure you have sourced `sourceme-manager.sh --skip-ssh-setup` **before running any of these commands.**

11.2.1 Building target software

In this guide, we'll boot Linux on our simulated node. To do so, we'll need to build our RISC-V SoC-compatible Linux distro. For this guide, we will use a simple buildroot-based distribution. We can build the Linux distribution like so:

```
# assumes you already cd'd into your firesim repo
# and sourced sourceme-manager.sh
#
# then:
cd ${CY_DIR}/software/firemarshal
./marshal -v build br-base.json
./marshal -v install br-base.json
```

Once this is completed, you'll have the following files:

- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base-bin` - a bootloader + Linux kernel image for the RISC-V SoC we will simulate.
- `${CY_DIR}/software/firemarshal/images/firechip/br-base/br-base.img` - a disk image for the RISC-V SoC we will simulate

These files will be used to form base images to either build more complicated workloads (see the *[DEPRECATED] Defining Custom Workloads* section) or directly as a basic, interactive Linux distribution.

11.3 Setting up the manager configuration

All runtime configuration options for the manager are set in a file called `${FS_DIR}/deploy/config_runtime.yaml`. In this guide, we will explain only the parts of this file necessary for our purposes. You can find full descriptions of all of the parameters in the *Manager Configuration Files* section.

If you open up this file, you will see the following default config (assuming you have not modified it)(note the `default_platform` value might be different):

```
# Run-time configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

run_farm:
  base_recipe: run-farm-recipes/externally_provisioned.yaml
  recipe_arg_overrides:
    # REQUIRED: default platform used for run farm hosts. this is a class specifying
    # how to run simulations on a run farm host.
    default_platform: XilinxAlveoU250InstanceDeployManager

    # REQUIRED: default directory where simulations are run out of on the run farm hosts
    default_simulation_dir: /home/buildbot/FIRESIM_RUNS_DIR

    # REQUIRED: default fpga db file that enumerates what fpgas are available on the␣
↪machine (used by XilinxU* Deploy Managers)
    default_fpga_db: /opt/firesim-db.json

    # REQUIRED: List of unique hostnames/IP addresses, each with their
    # corresponding specification that describes the properties of the host.
    #
    # Ex:
    # run_farm_hosts_to_use:
    #     # use localhost which is described by "four_fpgas_spec" below.
    #     - localhost: four_fpgas_spec
    #     # supply IP address, which points to a machine that is described
    #     # by "four_fpgas_spec" below.
    #     - "111.111.1.111": four_fpgas_spec
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec

metasimulation:
  metasimulation_enabled: false
  # vcs or verilator. use vcs-debug or verilator-debug for waveform generation
  metasimulation_host_simulator: verilator
  # plusargs passed to the simulator for all metasimulations
  metasimulation_only_plusargs: "+fesvr-step-size=128 +max-cycles=100000000"
  # plusargs passed to the simulator ONLY FOR vcs metasimulations
  metasimulation_only_vcs_plusargs: "+vcs+initreg+0 +vcs+initmem+0"

# DOCREF START: target_config area
target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1

    # This references a section from config_hwdb.yaml for fpga-accelerated simulation
    # or from config_build_recipes.yaml for metasimulation
    # In homogeneous configurations, use this to set the hardware config deployed
    # for all simulators
```

(continues on next page)

(continued from previous page)

```
    default_hw_config: midasexamples_gcd

    # Advanced: Specify any extra plusargs you would like to provide when
    # booting the simulator (in both FPGA-sim and metasim modes). This is
    # a string, with the contents formatted as if you were passing the plusargs
    # at command line, e.g. "+a=1 +b=2"
    plusarg_passthrough: ""
# DOCREF END: target_config area

tracing:
    enable: no

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
    # unwinding -> Flame Graph)
    output_format: 0

    # Trigger selector.
    # 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
    # instruction trigger
    selector: 1
    start: 0
    end: -1

autocounter:
    read_rate: 0

workload:
    workload_name: null.json
    terminate_on_completion: no
    suffix_tag: null

host_debug:
    # When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
    # begin (takes 2-5 minutes).
    # In general, this is not required to produce deterministic simulations on
    # target machines running linux. Enable if you observe simulation non-determinism.
    zero_out_dram: no
    # If disable_synth_asserts: no, simulation will print assertion message and
    # terminate simulation if synthesized assertion fires.
    # If disable_synth_asserts: yes, simulation ignores assertion firing and
    # continues simulation.
    disable_synth_asserts: no

# DOCREF START: Synthesized Prints
synth_print:
    # Start and end cycles for outputting synthesized prints.
    # They are given in terms of the base clock and will be converted
    # for each clock domain.
    start: 0
    end: -1
    # When enabled (=yes), prefix print output with the target cycle at which the print␣
```

(continues on next page)

(continued from previous page)

```
↪was triggered
    cycle_prefix: yes
# DOCREF END: Synthesized Prints
```

We'll need to modify a couple of these lines.

First, let's tell the manager to use the single Xilinx Vitis-enabled U250 FPGA. You'll notice that in the `run_farm` mapping which describes and specifies the machines to run simulations on. First notice that the `base_recipe` maps to `run-farm-recipes/externally_provisioned.yaml`. This indicates to the FireSim manager that the machines allocated to run simulations will be provided by the user through IP addresses instead of automatically launched and allocated (e.g. launching instances on-demand in AWS). Let's modify the `default_platform` to be `VitisInstanceDeployManager` so that we can launch simulations using Xilinx XRT/Vitis. Next, modify the `default_simulation_dir` to a directory that you want to store temporary simulation collateral to. When running simulations, this directory is used to store any temporary files that the simulator creates (e.g. a uartlog emitted by a Linux simulation). Next, lets modify the `run_farm_hosts_to_use` mapping. This maps IP addresses (i.e. `localhost`) to a description/specification of the simulation machine. In this case, we have only one Xilinx Vitis-enabled U250 FPGA so we will change the description of `localhost` to `one_fpga_spec`.

Now, let's verify that the `target_config` mapping will model the correct target design. By default, it is set to model a single-node with no network. It should look like the following:

```
target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1

    # This references a section from config_hwdb.yaml for fpga-accelerated simulation
    # or from config_build_recipes.yaml for metasimulation
    # In homogeneous configurations, use this to set the hardware config deployed
    # for all simulators
    default_hw_config: midasexamples_gcd

    # Advanced: Specify any extra plusargs you would like to provide when
    # booting the simulator (in both FPGA-sim and metasim modes). This is
    # a string, with the contents formatted as if you were passing the plusargs
    # at command line, e.g. "+a=1 +b=2"
    plusarg_passthrough: ""
```

Note `topology` is set to `no_net_config`, indicating that we do not want a network. Then, `no_net_num_nodes` is set to 1, indicating that we only want to simulate one node. Lastly, the `default_hw_config` is `firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3`. Let's modify the `default_hw_config` (the target design) to "vitis_firesim_rocket_singlecore_no_nic". This new hardware configuration does not have a NIC and is pre-built for the Xilinx Vitis-enabled U250 FPGA. This hardware configuration models a Single-core Rocket Chip SoC and **no** network interface card.

We will leave the `workload` mapping unchanged here, since we do want to run the buildroot-based Linux on our simulated system. The `terminate_on_completion` feature is an advanced feature that you can learn more about in the *Manager Configuration Files* section.

As a final sanity check, in the mappings we changed, the `config_runtime.yaml` file should now look like this (with `PATH_TO_SIMULATION_AREA` replaced with your simulation collateral temporary directory):

```yaml
run_farm:
  base_recipe: run-farm-recipes/externally_provisioned.yaml
  recipe_arg_overrides:
    default_platform: VitisInstanceDeployManager
    default_simulation_dir: <PATH_TO_SIMULATION_AREA>
    run_farm_hosts_to_use:
        - localhost: one_fpga_spec

target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1
    default_hw_config: vitis_firesim_rocket_singlecore_no_nic
    plusarg_passthrough: ""

workload:
    workload_name: br-base-uniform.json
    terminate_on_completion: no
    suffix_tag: null
```

11.4 Building and Deploying simulation infrastructure to the Run Farm Machines

The manager automates the process of building and deploying all components necessary to run your simulation on the Run Farm, including programming FPGAs. To tell the manager to setup all of our simulation infrastructure, run the following:

```
firesim infrasetup -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_DIR}
↪/sims/firesim-staging/sample_config_build_recipes.yaml
```

For a complete run, you should expect output like the following:

```
FireSim Manager. Docs: https://docs.fires.im
Running: infrasetup

Building FPGA software driver.
...
[localhost] Checking if host instance is up...
[localhost] Copying FPGA simulation infrastructure for slot: 0.
[localhost] Clearing all FPGA Slots.
The full log of this run is:
.../firesim/deploy/logs/2023-03-06--01-22-46-infrasetup-35ZP4WUOX8KUYBF3.log
```

Many of these tasks will take several minutes, especially on a clean copy of the repo. The console output here contains the "user-friendly" version of the output. If you want to see detailed progress as it happens, `tail -f` the latest logfile in `${FS_DIR}/deploy/logs/`.

At this point, our single Run Farm machine has all the infrastructure necessary to run a simulation, so let's launch our simulation!

11.5 Running the simulation

Finally, let's run our simulation! To do so, run:

```
firesim runworkload -a ${CY_DIR}/sims/firesim-staging/sample_config_hwdb.yaml -r ${CY_
↪DIR}/sims/firesim-staging/sample_config_build_recipes.yaml
```

This command boots up a simulation and prints out the live status of the simulated nodes every 10s. When you do this, you will initially see output like:

```
FireSim Manager. Docs: https://docs.fires.im
Running: runworkload

Creating the directory: .../firesim/deploy/results-workload/2023-03-06--01-25-34-br-base/
[localhost] Checking if host instance is up...
[localhost] Starting FPGA simulation for slot: 0.
```

If you don't look quickly, you might miss it, since it will get replaced with a live status page:

```
FireSim Simulation Status @ 2018-05-19 00:38:56.062737
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   localhost | Job: br-base0 | Sim running: True
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
--------------------------------------------------------------------------------
```

This will only exit once all of the simulated nodes have powered off. So, let's let it run and open another terminal on the manager machine. From there, `cd` into your FireSim directory again and `source sourceme-manager.sh --skip-ssh-setup`.

Next, let's `ssh` into the Run Farm machine. If your Run Farm and Manager Machines are the same, replace `RUN_FARM_IP_OR_HOSTNAME` with `localhost`, otherwise replace it with your Run Farm Machine's IP or hostname.

```
source ~/.ssh/AGENT_VARS
ssh RUN_FARM_IP_OR_HOSTNAME
```

Next, we can directly attach to the console of the simulated system using `screen`, run:

```
screen -r fsim0
```

Voila! You should now see Linux booting on the simulated system and then be prompted with a Linux login prompt, like so:

```
[truncated Linux boot output]
[    0.020000] VFS: Mounted root (ext2 filesystem) on device 254:0.
[    0.020000] devtmpfs: mounted
[    0.020000] Freeing unused kernel memory: 140K
[    0.020000] This architecture does not have kernel memory protection.
mount: mounting sysfs on /sys failed: No such device
Starting logging: OK
Starting mdev...
mdev: /sys/dev: No such file or directory
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... done.
Starting network: ip: SIOCGIFFLAGS: No such device
ip: can't find device 'eth0'
FAIL
Starting dropbear sshd: OK

Welcome to Buildroot
buildroot login:
```

You can ignore the messages about the network – that is expected because we are simulating a design without a NIC.

Now, you can login to the system! The username is `root` and there is no password. At this point, you should be presented with a regular console, where you can type commands into the simulation and run programs. For example:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
#
```

At this point, you can run workloads as you'd like. To finish off this guide, let's power off the simulated system and see what the manager does. To do so, in the console of the simulated system, run `poweroff -f`:

```
Welcome to Buildroot
buildroot login: root
Password:
# uname -a
Linux buildroot 4.15.0-rc6-31580-g9c3074b5c2cd #1 SMP Thu May 17 22:28:35 UTC 2018␣
↪riscv64 GNU/Linux
# poweroff -f
```

You should see output like the following from the simulation console:

```
# poweroff -f
[   12.456000] reboot: Power down
Power off
time elapsed: 468.8 s, simulation speed = 88.50 MHz
```

(continues on next page)

(continued from previous page)

```
*** PASSED *** after 41492621244 cycles
Runs 41492621244 cycles
[PASS] FireSim Test
SEED: 1526690334
Script done, file is uartlog

[screen is terminating]
```

You'll also notice that the manager polling loop exited! You'll see output like this from the manager:

```
FireSim Simulation Status @ 2018-05-19 00:46:50.075885
--------------------------------------------------------------------------------
This workload's output is located in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
This run's log is located in:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
This status will update every 10s.
--------------------------------------------------------------------------------
Instances
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Terminated: False
--------------------------------------------------------------------------------
Simulated Switches
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Simulated Nodes/Jobs
--------------------------------------------------------------------------------
Hostname/IP:   172.30.2.174 | Job: br-base0 | Sim running: False
--------------------------------------------------------------------------------
Summary
--------------------------------------------------------------------------------
1/1 instances are still running.
0/1 simulations are still running.
--------------------------------------------------------------------------------
FireSim Simulation Exited Successfully. See results in:
.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/
The full log of this run is:
.../firesim/deploy/logs/2018-05-19--00-38-52-runworkload-JS5IGTV166X169DZ.log
```

If you take a look at the workload output directory given in the manager output (in this case, `.../firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/`), you'll see the following:

```
$ ls -la firesim/deploy/results-workload/2018-05-19--00-38-52-br-base/*/*
-rw-rw-r-- 1 centos centos  797 May 19 00:46 br-base0/memory_stats.csv
-rw-rw-r-- 1 centos centos  125 May 19 00:46 br-base0/os-release
-rw-rw-r-- 1 centos centos 7316 May 19 00:46 br-base0/uartlog
```

What are these files? They are specified to the manager in a configuration file (`deploy/workloads/br-base-uniform.json`) as files that we want automatically copied back from the Run Farm Machine into the `results-workload` directory on our manager machine, which is useful for running benchmarks automatically. The *[DEPRECATED] Defining Custom Workloads* section describes this process in detail.

Congratulations on running your first FireSim simulation! At this point, you can check-out some of the advanced features of FireSim in the sidebar to the left.

Click Next if you'd like to continue on to building your own bitstreams.

> **Warning**
>
> In some cases, simulation may fail because you might need to update the Xilinx Vitis-enabled U250 DRAM offset that is currently hard coded in both the FireSim Xilinx XRT/Vitis driver code and platform shim. To verify this, run `xclbinutil --info --input <YOUR_XCL_BIN>`, obtain the `bank0 MEM_DDR4` offset. If it differs from the hardcoded `0x40000000` given in driver code (`u250_dram_expected_offset` variable in sim/midas/src/main/cc/simif_vitis.cc) and platform shim (`araddr/awaddr` offset in sim/midas/src/main/scala/midas/platform/VitisShim.scala) replace both areas with the new offset given by `xclbinutil` and regenerate the bitstream.

> **Warning**
>
> **We highly recommend using the XDMA-based U250 flow instead of this Vitis-based flow. You can find the XDMA-based flow here:** *Xilinx Alveo U250 XDMA-based Getting Started Guide*. The Vitis-based flow does not support DMA-based FireSim bridges (e.g., TracerV, Synthesizable Printfs, etc.), while the XDMA-based flows support all FireSim features. If you're unsure, use the XDMA-based U250 flow instead: *Xilinx Alveo U250 XDMA-based Getting Started Guide*

11.6 Building Your Own Hardware Designs

This section will guide you through building a Xilinx Vitis-enabled U250 FPGA bitstream to run FireSim simulations.

11.7 Configuring a Build in the Manager

In the `deploy/config_build.yaml` file, you will notice that the `builds_to_run` section currently contains several lines, which indicates to the build system that you want to run all of these "build recipes" in parallel, with the parameters for each "build recipe" listed in the relevant section of the `deploy/config_build_recipes.yaml` file (or in the case of building a Chipyard target SoC in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml).

In this guide, we'll build the default FireSim design for the Xilinx Vitis-enabled U250, which is specified by the `vitis_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml. This was the same configuration used to build the pre-built bitstream that you used to run simulations in the guide to running a simulation.

Looking at the `vitis_firesim_rocket_singlecore_no_nic` section in ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.yaml, there are a few notable items:

- `TARGET_CONFIG` specifies that this configuration is a simple singlecore RISC-V Rocket with a single DRAM channel.
- `TARGET_PROJECT_MAKEFRAG` specifies to the FireSim Make system how to build the `TARGET_CONFIG`.
- `bit_builder_recipe` points to `bit-builder-recipes/vitis.yaml`, which is found in the deploy directory and tells the FireSim build system how to build bitstreams for this FPGA.

Having looked at this entry, let's now set up the build in `deploy/config_build.yaml`. First, we'll set up the `build_farm` mapping, which specifies the Build Farm Machines that are available to build FPGA bitstreams.

- `base_recipe` will map to `build-farm-recipes/externally_provisioned.yaml`. This indicates to the FireSim manager that the machines used to run builds are existing machines that have been set up by the user, instead of cloud instances that are automatically provisioned.
- `default_build_dir` is the directory in which builds will run out of on your Build Farm Machines. Change the default `null` to a path where you would like temporary build data to be stored on your Build Farm Machines.
- `build_farm_hosts` is a section that contains a list of IP addresses or hostnames of machines in your Build Farm. By default, `localhost` is specified. If you are using a separate Build Farm Machine, you should replace this with the IP address or hostname of the Build Farm Machine on which you would like to run the build.

Having configured our Build Farm, let's specify the design we'd like to build. To do this, edit the `builds_to_run` section in `deploy/config_build.yaml` so that it looks like the following:

```
builds_to_run:
    - vitis_firesim_rocket_singlecore_no_nic
```

In essence, you should delete or comment out all the other items in the `builds_to_run` section besides `vitis_firesim_rocket_singlecore_no_nic`.

11.8 Running the Build

Now, we can run a build like so (while also pointing to Chipyard's recipes provided):

```
firesim buildbitstream -r -r ${CY_DIR}/sims/firesim-staging/sample_config_build_recipes.
→yaml
```

This will run through the entire build process, taking the Chisel (or Verilog) RTL and producing a bitstream that runs on the Xilinx Vitis-enabled U250 FPGA. This whole process will usually take a few hours. When the build completes, you will see a directory in `deploy/results-build/`, named after your build parameter settings, that contains all of the outputs of the Xilinx Vitis build process. Additionally, the manager will print out a path to a log file that describes everything that happened, in-detail, during this run (this is a good file to send us if you encounter problems).

The manager will also print an entry that can be added to `config_hwdb.yaml` so that the bitstream can be used to run simulations. This entry will contain a `bitstream_tar` key whose value is the path to the final generated bitstream file. You can share generated bitstreams with others by sharing the file listed in `bitstream_tar` and the `config_hwdb.yaml` entry for it.

Now that you know how to generate your own FPGA image, you can modify the target-design to add your own features, then build a FireSim-compatible FPGA image automatically!

This is the end of the Getting Started Guide. To learn more advanced FireSim features, you can choose a link under the "Advanced Docs" section to the left.

CHAPTER

TWELVE

MANAGER USAGE (THE `FIRESIM` COMMAND)

12.1 Overview

When you source `sourceme-manager.sh` in your copy of the FireSim repo, you get access to a new command, `firesim`, which is the FireSim simulation manager. If you've used tools like Vagrant or Docker, the `firesim` program is to FireSim what `vagrant` and `docker` are to Vagrant and Docker respectively. In essence, `firesim` lets us manage the entire lifecycle of FPGA simulations, just like `vagrant` and `docker` do for VMs and containers respectively.

12.1.1 "Inputs" to the Manager

The manager gets configuration information from several places:

- Command Line Arguments, consisting of:
  - Paths to configuration files to use
  - A task to run
  - Arguments to the task
- Configuration Files
- Environment Variables
- Topology definitions for networked simulations (`user_topology.py`)

The following sections detail these inputs. Hit Next to continue.

12.1.2 Logging

The manager produces detailed logs when you run any command, which is useful to share with the FireSim developers for debugging purposes in case you encounter issues. The logs contain more detailed output than the manager sends to stdout/stderr during normal operation, so it's also useful if you want to take a peek at the detailed commands manager is running to facilitate builds and simulations. Logs are stored in `firesim/deploy/logs/`.

12.2 Manager Command Line Arguments

The manager provides built-in help output for the command line arguments it supports if you run `firesim --help`

```
usage: firesim [-h] [-c RUNTIMECONFIGFILE] [-b BUILDCONFIGFILE]
               [-r BUILDRECIPESCONFIGFILE] [-a HWDBCONFIGFILE]
               [-x OVERRIDECONFIGDATA] [-f TERMINATESOMEF116]
               [-g TERMINATESOMEF12] [-i TERMINATESOMEF14]
               [-m TERMINATESOMEM416] [--terminatesome TERMINATESOME] [-q]
               [-t LAUNCHTIME]
               [--platform {f1,rhsresearch_nitefury_ii,vitis,xilinx_alveo_u200,xilinx_
↪alveo_u250,xilinx_alveo_u280,xilinx_vcu118}]
               {managerinit,infrasetup,boot,kill,runworkload,buildbitstream,builddriver,
↪enumeratefpgas,tar2afi,runcheck,launchrunfarm,terminaterunfarm,shareagfi}

FireSim Simulation Manager.

positional arguments:
  {managerinit,infrasetup,boot,kill,runworkload,buildbitstream,builddriver,
↪enumeratefpgas,tar2afi,runcheck,launchrunfarm,terminaterunfarm,shareagfi}
                        Management task to run.

options:
  -h, --help            show this help message and exit
  -c RUNTIMECONFIGFILE, --runtimeconfigfile RUNTIMECONFIGFILE
                        Optional custom runtime/workload config file. Defaults
                        to config_runtime.yaml.
  -b BUILDCONFIGFILE, --buildconfigfile BUILDCONFIGFILE
                        Optional custom build config file. Defaults to
                        config_build.yaml.
  -r BUILDRECIPESCONFIGFILE, --buildrecipesconfigfile BUILDRECIPESCONFIGFILE
                        Optional custom build recipe config file. Defaults to
                        config_build_recipes.yaml.
  -a HWDBCONFIGFILE, --hwdbconfigfile HWDBCONFIGFILE
                        Optional custom HW database config file. Defaults to
                        config_hwdb.yaml.
  -x OVERRIDECONFIGDATA, --overrideconfigdata OVERRIDECONFIGDATA
                        Override a single value from one of the the RUNTIME
                        e.g.: --overrideconfigdata "target-config link-latency
                        6405".
  -f TERMINATESOMEF116, --terminatesomef116 TERMINATESOMEF116
                        DEPRECATED. Use --terminatesome=f1.16xlarge:count
                        instead. Will be removed in the next major version of
                        FireSim (1.15.X). Old help message: Only used by
                        terminaterunfarm. Terminates this many of the
                        previously launched f1.16xlarges.
  -g TERMINATESOMEF12, --terminatesomef12 TERMINATESOMEF12
                        DEPRECATED. Use --terminatesome=f1.2xlarge:count
                        instead. Will be removed in the next major version of
                        FireSim (1.15.X). Old help message: Only used by
                        terminaterunfarm. Terminates this many of the
                        previously launched f1.2xlarges.
  -i TERMINATESOMEF14, --terminatesomef14 TERMINATESOMEF14
```

(continues on next page)

(continued from previous page)

```
                        DEPRECATED. Use --terminatesome=f1.4xlarge:count
                        instead. Will be removed in the next major version of
                        FireSim (1.15.X). Old help message: Only used by
                        terminaterunfarm. Terminates this many of the
                        previously launched f1.4xlarges.
  -m TERMINATESOMEM416, --terminatesomem416 TERMINATESOMEM416
                        DEPRECATED. Use --terminatesome=m4.16xlarge:count
                        instead. Will be removed in the next major version of
                        FireSim (1.15.X). Old help message: Only used by
                        terminaterunfarm. Terminates this many of the
                        previously launched m4.16xlarges.
  --terminatesome TERMINATESOME
                        Only used by terminaterunfarm. Used to specify a
                        restriction on how many instances to terminate. E.g.,
                        --terminatesome=f1.2xlarge:2 will terminate only 2 of
                        the f1.2xlarge instances in the runfarm, regardless of
                        what other instances are in the runfarm. This argument
                        can be specified multiple times to terminate
                        additional instance types/counts. Behavior when
                        specifying the same instance type multiple times is
                        undefined. This replaces the old
                        --terminatesome{f116,f12,f14,m416} arguments. Behavior
                        when specifying these old-style terminatesome flags
                        and this new style flag at the same time is also
                        undefined.
  -q, --forceterminate  For terminaterunfarm and buildbitstream, force
                        termination without prompting user for confirmation.
                        Defaults to False
  -t LAUNCHTIME, --launchtime LAUNCHTIME
                        Give the "Y-m-d--H-M-S" prefix of results-build
                        directory. Useful for tar2afi when finishing a partial
                        buildbitstream
  --platform {f1,rhsresearch_nitefury_ii,vitis,xilinx_alveo_u200,xilinx_alveo_u250,
↪xilinx_alveo_u280,xilinx_vcu118}
                        Required argument for "managerinit" to specify which
                        platform you will be using
```

On this page, we will go through some of these options – others are more complicated, so we will give them their own section on the following pages.

12.2.1 `--runtimeconfigfile FILENAME`

This lets you specify a custom **runtime** config file. By default, `config_runtime.yaml` is used. See *config_runtime.yaml* for what this config file does.

12.2.2 `--buildconfigfile FILENAME`

This lets you specify a custom **build** config file. By default, `config_build.yaml` is used. See *config_build.yaml* for what this config file does.

12.2.3 `--buildrecipesconfigfile FILENAME`

This lets you specify a custom build **recipes** config file. By default, `config_build_recipes.yaml` is used. See *config_build_recipes.yaml* for what this config file does.

12.2.4 `--hwdbconfigfile FILENAME`

This lets you specify a custom **hardware database** config file. By default, `config_hwdb.yaml` is used. See *config_hwdb.yaml* for what this config file does.

12.2.5 `--overrideconfigdata SECTION PARAMETER VALUE`

This lets you override a single value from the **runtime** config file. For example, if you want to use a link latency of 3003 cycles for a particular run (and your `config_runtime.yaml` file specifies differently), you can pass `--overrideconfigdata target_config link_latency 6405` to the manager. This can be used with any task that uses the runtime config.

12.2.6 `--launchtime TIMESTAMP`

Specifies the "Y-m-d–H-M-S" timestamp to be used as the prefix in `results-build` directories. Useful when wanting to run `tar2afi` after an aborted `buildbitstream` was manually fixed.

12.2.7 `TASK`

This is the only required/positional command line argument to the manager. It tells the manager what it should be doing. See the next section for a list of tasks and what they do. Some tasks also take other command line arguments, which are specified with those tasks.

12.3 Manager Tasks

This page outlines all of the tasks that the FireSim manager supports.

12.3.1 `firesim managerinit`

This is a setup command that does the following:

- Backup existing config files if they exist (`config_runtime.yaml`, `config_build.yaml`, `config_build_recipes.yaml`, and `config_hwdb.yaml`).
- Replace the default config files (`config_runtime.yaml`, `config_build.yaml`, `config_build_recipes.yaml`, and `config_hwdb.yaml`) with clean example versions.

Then, do platform-specific init steps for the given `--platform`.

`f1`

All other platforms

- Run `aws configure`, prompt for credentials
- Prompt the user for email address and subscribe them to notifications for their own builds.
- Setup the `config_runtime.yaml` and `config_build.yaml` files with AWS run/build farm arguments.

This includes platforms such as: `xilinx_alveo_u200`, `xilinx_alveo_u250`, `xilinx_alveo_u280`, `xilinx_vcu118`, and `rhsresearch_nitefury_ii`.

- Setup the `config_runtime.yaml` and `config_build.yaml` files with externally provisioned run/build farm arguments.

You can re-run this whenever you want to get clean configuration files.

Note

For `f1`, you can just hit Enter when prompted for `aws configure` credentials and your email address, and both will keep your previously specified values.

If you run this command by accident and didn't mean to overwrite your configuration files, you'll find backed-up versions in `firesim/deploy/sample-backup-configs/backup*`.

12.3.2 `firesim buildbitstream`

This command builds a FireSim bitstream using a **Build Farm** from the Chisel RTL for the configurations that you specify. The process of defining configurations to build is explained in the documentation for *config_build.yaml* and *config_build_recipes.yaml*.

For each config, the build process entails:

F1

XDMA-based On-Prem.

Vitis-based On-Prem.

1. [Locally] Run the elaboration process for your hardware configuration
2. [Locally] FAME-1 transform the design with MIDAS
3. [Locally] Attach simulation models (I/O widgets, memory model, etc.)
4. [Locally] Emit Verilog to run through the FPGA Flow
5. Use a build farm configuration to launch/use build hosts for each configuration you want to build

6. [Local/Remote] Prep build hosts, copy generated Verilog for hardware configuration to build instance
7. [Local or Remote] Run Vivado Synthesis and P&R for the configuration
8. [Local/Remote] Copy back all output generated by Vivado including the final tar file
9. [Local/AWS Infra] Submit the tar file to the AWS backend for conversion to an AFI
10. [Local] Wait for the AFI to become available, then notify the user of completion by email

1. [Locally] Run the elaboration process for your hardware configuration
2. [Locally] FAME-1 transform the design with MIDAS
3. [Locally] Attach simulation models (I/O widgets, memory model, etc.)
4. [Locally] Emit Verilog to run through the FPGA Flow
5. Use a build farm configuration to launch/use build hosts for each configuration you want to build
6. [Local/Remote] Prep build hosts, copy generated Verilog for hardware configuration to build instance
7. [Local or Remote] Run Vivado Synthesis and P&R for the configuration
8. [Local/Remote] Copy back all output generated by Vivado (including `bit` bitstream)

1. [Locally] Run the elaboration process for your hardware configuration
2. [Locally] FAME-1 transform the design with MIDAS
3. [Locally] Attach simulation models (I/O widgets, memory model, etc.)
4. [Locally] Emit Verilog to run through the FPGA Flow
5. Use a build farm configuration to launch/use build hosts for each configuration you want to build
6. [Local/Remote] Prep build hosts, copy generated Verilog for hardware configuration to build instance
7. [Local or Remote] Run Vitis Synthesis and P&R for the configuration
8. [Local/Remote] Copy back all output generated by Vitis (including the `bitstream_tar` containing the `xclbin` bitstream)

This process happens in parallel for all of the builds you specify. The command will exit when all builds are completed (but you will get notified as INDIVIDUAL builds complete if on F1) and indicate whether all builds passed or a build failed by the exit code.

Note

It is highly recommended that you either run this command in a `screen` or use `mosh` to access the manager instance. Builds will not finish if the manager is killed due to ssh disconnection from the manager instance.

When you run a build for a particular configuration, a directory named `LAUNCHTIME-CONFIG_TRIPLET-BUILD_NAME` is created in `firesim/deploy/results-build/`. This directory will contain:

F1

XDMA-based On-Prem.

Vitis-based On-Prem.

- `AGFI_INFO`: Describes the state of the AFI being built, while the manager is running. Upon build completion, this contains the AGFI/AFI that was produced, along with its metadata.

- `cl_firesim:`: This directory is essentially the Vivado project that built the FPGA image, in the state it was in when the Vivado build process completed. This contains reports, stdout from the build, and the final tar file produced by Vivado. This also contains a copy of the generated verilog (`FireSim-generated.sv`) used to produce this build.

The Vivado project collateral that built the FPGA image, in the state it was in when the Vivado build process completed. This contains reports, `stdout` from the build, and the final `bitstream_tar` bitstream/metadata file produced by Vivado. This also contains a copy of the generated verilog (`FireSim-generated.sv`) used to produce this build.

The Vitis project collateral that built the FPGA image, in the state it was in when the Vitis build process completed. This contains reports, `stdout` from the build, and the final `bitstream_tar` produced from the Vitis-generated `xclbin` bitstream. This also contains a copy of the generated verilog (`FireSim-generated.sv`) used to produce this build.

If this command is cancelled by a SIGINT, it will prompt for confirmation that you want to terminate the build instances. If you respond in the affirmative, it will move forward with the termination. If you do not want to have to confirm the termination (e.g. you are using this command in a script), you can give the command the `--forceterminate` command line argument. For example, the following will terminate all build instances in the build farm without prompting for confirmation if a SIGINT is received:

```
firesim buildbitstream --forceterminate
```

12.3.3 `firesim builddriver`

For FPGA-based simulations (when `metasimulation_enabled` is `false` in `config_runtime.yaml`), this command will build the host-side simulation driver, also without requiring any simulation hosts to be launched or reachable. For complicated designs, running this before running `firesim launchrunfarm` can reduce the time spent leaving FPGA hosts idling while waiting for driver build.

For metasimulations (when `metasimulation_enabled` is `true` in `config_runtime.yaml`), this command will build the entire software simulator without requiring any simulation hosts to be launched or reachable. This is useful for example if you are using FireSim metasimulations as your primary simulation tool while developing target RTL, since it allows you to run the Chisel build flow and iterate on your design without launching/setting up extra machines to run simulations.

12.3.4 `firesim tar2afi`

Note

Can only be used for the F1 platform.

This command can be used to run only steps 9 & 10 from an aborted `firesim buildbitstream` for F1 that has been manually corrected. `firesim tar2afi` assumes that you have a `firesim/deploy/results-build/LAUNCHTIME-CONFIG_TRIPLET-BUILD_NAME/cl_firesim` directory tree that can be submitted to the AWS backend for conversion to an AFI.

When using this command, you need to also provide the `--launchtime LAUNCHTIME` cmdline argument, specifying an already existing LAUNCHTIME.

This command will run for the configurations specified in *config_build.yaml* and *config_build_recipes.yaml* as with *firesim buildbitstream*. It is likely that you may want to comment out build recipe names that successfully completed the *firesim buildbitstream* process before running this command.

12.3.5 firesim shareagfi

Note

Can only be used for the F1 platform.

This command allows you to share AGFIs that you have already built (that are listed in *config_hwdb.yaml*) with other users. It will take the named hardware configurations that you list in the `agfis_to_share` section of `config_build.yaml`, grab the respective AGFIs for each from `config_hwdb.yaml`, and share them across all F1 regions with the users listed in the `share_with_accounts` section of `config_build.yaml`. You can also specify `public: public` in `share_with_accounts` to make the AGFIs public.

You must own the AGFIs in order to do this – this will NOT let you share AGFIs that someone else owns and gave you access to.

12.3.6 firesim launchrunfarm

Note

Can only be used for the F1 platform.

This command launches a **Run Farm** on AWS EC2 on which you run simulations. Run farms consist of a set of **run farm instances** that can be spawned on AWS EC2. The `run_farm` mapping in `config_runtime.yaml` determines the run farm used and its configuration (see *config_runtime.yaml*). The `base_recipe` key/value pair specifies the default set of arguments to use for a particular run farm type. To change the run farm type, a new `base_recipe` file must be provided from `deploy/run-farm-recipes`. You are able to override the arguments given by a `base_recipe` by adding keys/values to the `recipe_arg_overrides` mapping. These keys/values must match the same mapping structure as the `args` mapping. Overridden arguments override recursively such that all key/values present in the override args replace the default arguments given by the `base_recipe`. In the case of sequences, a overridden sequence completely replaces the corresponding sequence in the default args.

An AWS EC2 run farm consists of AWS instances like `f1.16xlarge`, `f1.4xlarge`, `f1.2xlarge`, and `m4.16xlarge` instances. Before you run the command, you define the number of each that you want in the `recipe_arg_overrides` section of `config_runtime.yaml` or in the `base_recipe` itself.

A launched run farm is tagged with a `run_farm_tag`, which is used to disambiguate multiple parallel run farms; that is, you can have many run farms running, each running a different experiment at the same time, each with its own unique `run_farm_tag`. One convenient feature to add to your AWS management panel is the column for `fsimcluster`, which contains the `run_farm_tag` value. You can see how to do that in the *Add the fsimcluster column to your AWS management console* section.

The other options in the `run_farm` section, `run_instance_market`, `spot_interruption_behavior`, and `spot_max_price` define *how* instances in the run farm are launched. See the documentation for `config_runtime.yaml` for more details on other arguments (see *config_runtime.yaml*).

ERRATA: One current requirement is that you must define a target config in the `target_config` section of `config_runtime.yaml` that does not require more resources than the run farm you are trying to launch. Thus, you should also setup your `target_config` parameters before trying to launch the corresponding run farm. This requirement will be removed in the future.

Once you setup your configuration and call `firesim launchrunfarm`, the command will launch the run farm. If all succeeds, you will see the command print out instance IDs for the correct number/types of instances (you do not need to pay attention to these or record them). If an error occurs, it will be printed to console.

Warning

On AWS EC2, once you run this command, your run farm will continue to run until you call `firesim terminaterunfarm`. This means you will be charged for the running instances in your run farm until you call `terminaterunfarm`. You are responsible for ensuring that instances are only running when you want them to be by checking the AWS EC2 Management Panel.

12.3.7 `firesim terminaterunfarm`

Note

Can only be used for the F1 platform.

This command terminates some or all of the instances in the Run Farm defined in your `config_runtime.yaml` file by the `run_farm` `base_recipe`, depending on the command line arguments you supply.

By default, running `firesim terminaterunfarm` will terminate ALL instances with the specified `run_farm_tag`. When you run this command, it will prompt for confirmation that you want to terminate the listed instances. If you respond in the affirmative, it will move forward with the termination.

If you do not want to have to confirm the termination (e.g. you are using this command in a script), you can give the command the `--forceterminate` command line argument. For example, the following will TERMINATE ALL INSTANCES IN THE RUN FARM WITHOUT PROMPTING FOR CONFIRMATION:

```
firesim terminaterunfarm --forceterminate
```

The `--terminatesome=INSTANCE_TYPE:COUNT` flag additionally allows you to terminate only some (`COUNT`) of the instances of a particular type (`INSTANCE_TYPE`) in a particular Run Farm.

Here are some examples:

```
[ start with 2 f1.16xlarges, 2 f1.2xlarges, 2 m4.16xlarges ]

firesim terminaterunfarm --terminatesome=f1.16xlarge:1 --forceterminate

[ now, we have: 1 f1.16xlarges, 2 f1.2xlarges, 2 m4.16xlarges ]
```

```
[ start with 2 f1.16xlarges, 2 f1.2xlarges, 2 m4.16xlarges ]

firesim terminaterunfarm --terminatesome=f1.16xlarge:1 --terminatesome=f1.2xlarge:2 --
↪forceterminate

[ now, we have: 1 f1.16xlarges, 0 f1.2xlarges, 2 m4.16xlarges ]
```

Warning

On AWS EC2, once you call `launchrunfarm`, you will be charged for running instances in your Run Farm until you call `terminaterunfarm`. You are responsible for ensuring that instances are only running when you want them to be by checking the AWS EC2 Management Panel.

12.3.8 `firesim infrasetup`

Once you have launched a Run Farm and setup all of your configuration options, the `infrasetup` command will build all components necessary to run the simulation and deploy those components to the machines in the Run Farm. Here is a rough outline of what the command does:

- Constructs the internal representation of your simulation. This is a tree of components in the simulation (simulated server blades, switches)
- For each type of server blade, rebuild the software simulation driver by querying the bitstream metadata to get the build-quadruplet or using its override
- For each type of switch in the simulation, generate the switch model binary
- For each host instance in the Run Farm, collect information about all the resources necessary to run a simulation on that host instance, then copy files and flash FPGAs with the required bitstream.

Details about setting up your simulation configuration can be found in *config_runtime.yaml*.

Once you run a simulation, you should re-run `firesim infrasetup` before starting another one, even if it is the same exact simulation on the same Run Farm.

You can see detailed output from an example run of `infrasetup` in the *Running a Single Node Simulation* and *Running a Cluster Simulation* Getting Started Guides.

12.3.9 `firesim boot`

Once you have run `firesim infrasetup`, this command will actually start simulations. It begins by launching all switches (if they exist in your simulation config), then launches all server blade simulations. This simply launches simulations and then exits – it does not perform any monitoring.

This command is useful if you want to launch a simulation, then plan to interact with the simulation by-hand (i.e. by directly interacting with the console).

12.3.10 `firesim kill`

Given a simulation configuration and simulations running on a Run Farm, this command force-terminates all components of the simulation. Importantly, this does not allow any outstanding changes to the filesystem in the simulated systems to be committed to the disk image.

12.3.11 `firesim runworkload`

This command is the standard tool that lets you launch simulations, monitor the progress of workloads running on them, and collect results automatically when the workloads complete. To call this command, you must have first called `firesim infrasetup` to setup all required simulation infrastructure on the remote nodes.

This command will first create a directory in `firesim/deploy/results-workload/` named as `LAUNCH_TIME-WORKLOADNAME`, where results will be completed as simulations complete. This command will then automatically call `firesim boot` to start simulations. Then, it polls all the instances in the Run Farm every 10 seconds to determine the state of the simulated system. If it notices that a simulation has shutdown (i.e. the simulation disappears from the output of `screen -ls`), it will automatically copy back all results from the simulation, as defined in the workload configuration (see the *[DEPRECATED] Defining Custom Workloads* section).

For non-networked simulations, it will wait for ALL simulations to complete (copying back results as each workload completes), then exit.

For globally-cycle-accurate networked simulations, the global simulation will stop when any single node powers off. Thus, for these simulations, `runworkload` will copy back results from all nodes and force them to terminate by calling `kill` when ANY SINGLE ONE of them shuts down cleanly.

A simulation shuts down cleanly when the workload running on the simulator calls `poweroff`.

12.3.12 `firesim runcheck`

This command is provided to let you debug configuration options without launching instances. In addition to the output produced at command line/in the log, you will find a pdf diagram of the topology you specify, annotated with information about the workloads, hardware configurations, and abstract host mappings for each simulation (and optionally, switch) in your design. These diagrams are located in `firesim/deploy/generated-topology-diagrams/`, named after your topology.

Here is an example of such a diagram (click to expand/zoom, it will likely be illegible without expanding):

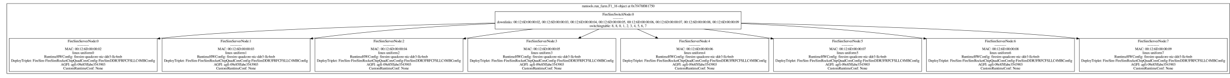


Fig. 1: Example diagram for an 8-node cluster with one ToR switch

12.3.13 `firesim enumeratefpgas`

Note

Can only be used for XDMA-based On-Premises platforms.

This command should be run once for each on-premises Run Farm you plan to use that contains XDMA-based FPGAs. When run, the command will generate a file (`/opt/firesim-db.json`) on each Run Farm Machine in the run farm that contains a mapping from the FPGA ID used for JTAG programming to the PCIe ID used to run simulations for each FPGA attached to the machine.

If you ever change the physical layout of a Run Farm Machine in your Run Farm (e.g., which PCIe slot the FPGAs are attached to), you will need to re-run this command.

12.4 Manager URI Paths

Some keys specified in `config_hwdb.yaml` may be specified as a `URI`

12.4.1 URI Support

A Uniform Resource Identifier (URI) which specifies a protocol supported either directly by the fsspec library or by one of the many third party extension libraries which build on fsspec.

Please note that while use use the `fsspec` library to handle many different URI protocols, many of them require additional dependencies that FireSim itself does not require you to install. `fsspec` will throw an exception telling you to install missing packages if you use one of the many URI protocols we do not test.

Likewise, individual URI protocols will have their own requirements for specifying credentials. Documentation supplying credentials is provided by the individual protocol implementation. For example:

- adlfs for Azure Data-Lake Gen1 and Gen2
- gcfs for Google Cloud Services
- s3fs for AWS S3

For SSH, add any required keys to your ssh-agent.

Please note that while some protocol backendss provide authentication via their own configuration files or environment variables (e.g. AWS credentials stored in `~/.aws`, created by `aws configure`), one can additionally configure `fsspec` with additional default keyword arguments per backend protocol by using one of the fsspec configuration methods.

12.5 Manager Configuration Files

This page contains a centralized reference for all of the configuration options in `config_runtime.yaml`, `config_build.yaml`, `config_build_farm.yaml`, `config_build_recipes.yaml`, and `config_hwdb.yaml`. It also contains references for all build and run farm recipes (in deploy/build-farm-recipes and deploy/run-farm-recipes).

12.5.1 config_runtime.yaml

Here is a sample of this configuration file:

```
# Run-time configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

run_farm:
  # managerinit replace start
  base_recipe: run-farm-recipes/aws_ec2.yaml
  # Uncomment and add args to override defaults.
  # Arg structure should be identical to the args given
  # in the base_recipe.
  #recipe_arg_overrides:
  #  <ARG>: <OVERRIDE>
  # managerinit replace end

metasimulation:
  metasimulation_enabled: false
  # vcs or verilator. use vcs-debug or verilator-debug for waveform generation
  metasimulation_host_simulator: verilator
  # plusargs passed to the simulator for all metasimulations
  metasimulation_only_plusargs: "+fesvr-step-size=128 +max-cycles=100000000"
```

(continues on next page)

(continued from previous page)

```
  # plusargs passed to the simulator ONLY FOR vcs metasimulations
  metasimulation_only_vcs_plusargs: "+vcs+initreg+0 +vcs+initmem+0"

# DOCREF START: target_config area
target_config:
    topology: no_net_config
    no_net_num_nodes: 1
    link_latency: 6405
    switching_latency: 10
    net_bandwidth: 200
    profile_interval: -1

    # This references a section from config_hwdb.yaml for fpga-accelerated simulation
    # or from config_build_recipes.yaml for metasimulation
    # In homogeneous configurations, use this to set the hardware config deployed
    # for all simulators
    default_hw_config: midasexamples_gcd

    # Advanced: Specify any extra plusargs you would like to provide when
    # booting the simulator (in both FPGA-sim and metasim modes). This is
    # a string, with the contents formatted as if you were passing the plusargs
    # at command line, e.g. "+a=1 +b=2"
    plusarg_passthrough: ""
# DOCREF END: target_config area

tracing:
    enable: no

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
    # unwinding -> Flame Graph)
    output_format: 0

    # Trigger selector.
    # 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
    # instruction trigger
    selector: 1
    start: 0
    end: -1

autocounter:
    read_rate: 0

workload:
    workload_name: null.json
    terminate_on_completion: no
    suffix_tag: null

host_debug:
    # When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
    # begin (takes 2-5 minutes).
    # In general, this is not required to produce deterministic simulations on
```

(continues on next page)

(continued from previous page)

```
    # target machines running linux. Enable if you observe simulation non-determinism.
    zero_out_dram: no
    # If disable_synth_asserts: no, simulation will print assertion message and
    # terminate simulation if synthesized assertion fires.
    # If disable_synth_asserts: yes, simulation ignores assertion firing and
    # continues simulation.
    disable_synth_asserts: no

# DOCREF START: Synthesized Prints
synth_print:
    # Start and end cycles for outputting synthesized prints.
    # They are given in terms of the base clock and will be converted
    # for each clock domain.
    start: 0
    end: -1
    # When enabled (=yes), prefix print output with the target cycle at which the print␣
↪was triggered
    cycle_prefix: yes
# DOCREF END: Synthesized Prints
```

Below, we outline each mapping in detail.

run_farm

The run_farm mapping specifies the characteristics of your FireSim run farm so that the manager can automatically launch them, run workloads on them, and terminate them.

base_recipe

The base_recipe key/value pair specifies the default set of arguments to use for a particular run farm type. To change the run farm type, a new base_recipe file must be provided from deploy/run-farm-recipes. You are able to override the arguments given by a base_recipe by adding keys/values to the recipe_arg_overrides mapping.

recipe_arg_overrides

This optional mapping of keys/values allows you to override the default arguments provided by the base_recipe. This mapping must match the same mapping structure as the args mapping within the base_recipe file given. Overridden arguments override recursively such that all key/values present in the override args replace the default arguments given by the base_recipe. In the case of sequences, a overridden sequence completely replaces the corresponding sequence in the default args. Additionally, it is not possible to change the default run farm type through these overrides. This must be done by changing the default base_recipe.

See *Run Farm Recipes (run-farm-recipes/*)* for more details on the potential run farm recipes that can be used.

metasimulation

The metasimulation options below allow you to run metasimulations instead of FPGA simulations when doing launchrunfarm, infrasetup, and runworkload. See *Debugging & Testing with Metasimulation* for more details.

metasimulation_enabled

This is a boolean to enable running metasimulations in-place of FPGA-accelerated simulations. The number of metasimulations that are run on a specific Run Farm host is determined by the num_metasims argument in each run farm recipe (see *Run Farm Recipes (run-farm-recipes/*)*).

metasimulation_host_simulator

This key/value pair chooses which RTL simulator should be used for metasimulation. Options include verilator and vcs if waveforms are unneeded and *-debug versions if a waveform is needed.

metasimulation_only_plusargs

This key/value pair is a string that passes plusargs (arguments with a + in front) to the metasimulations.

metasimulation_only_vcs_plusargs

This key/value pair is a string that passes plusargs (arguments with a + in front) to metasimulations using vcs or vcs-debug.

target_config

The target_config options below allow you to specify the high-level configuration of the target you are simulating. You can change these parameters after launching a Run Farm (assuming you have the correct number of instances), but in many cases you will need to re-run the infrasetup command to make sure the correct simulation infrastructure is available on your instances.

topology

This field dictates the network topology of the simulated system. Some examples:

no_net_config: This runs N (see no_net_num_nodes below) independent simulations, without a network simulation. You can currently only use this option if you build one of the NoNIC hardware configs of FireSim.

example_8config: This requires a single f1.16xlarge, which will simulate 1 ToR switch attached to 8 simulated servers.

example_16config: This requires two f1.16xlarge instances and one m4.16xlarge instance, which will simulate 2 ToR switches, each attached to 8 simulated servers, with the two ToR switches connected by a root switch.

example_64config: This requires eight f1.16xlarge instances and one m4.16xlarge instance, which will simulate 8 ToR switches, each attached to 8 simulated servers (for a total of 64 nodes), with the eight ToR switches connected by a root switch.

Additional configurations are available in deploy/runtools/user_topology.py and more can be added there. See the *Manager Network Topology Definitions (user_topology.py)* section for more info.

no_net_num_nodes

This determines the number of simulated nodes when you are using `topology: no_net_config`.

link_latency

In a networked simulation, this allows you to specify the link latency of the simulated network in CYCLES. For example, 6405 cycles is roughly 2 microseconds at 3.2 GHz. A current limitation is that this value (in cycles) must be a multiple of 7. Furthermore, you must not exceed the buffer size specified in the NIC's simulation widget.

switching_latency

In a networked simulation, this specifies the minimum port-to-port switching latency of the switch models, in CYCLES.

net_bandwidth

In a networked simulation, this specifies the maximum output bandwidth that a NIC is allowed to produce as an integer in Gbit/s. Currently, this must be a number between 1 and 200, allowing you to model NICs between 1 and 200 Gbit/s.

profile_interval

The simulation driver periodically samples performance counters in FASED timing model instances and dumps the result to a file on the host. `profile_interval` defines the number of target cycles between samples; setting this field to -1 disables polling.

default_hw_config

This sets the server configuration launched by default in the above topologies. Heterogeneous configurations can be achieved by manually specifying different names within the topology itself, but all the `example_Nconfig` configurations are homogeneous and use this value for all nodes.

You should set this to one of the hardware configurations you have defined already in `config_hwdb.yaml`. You should set this to the NAME (mapping title) of the hardware configuration from `config_hwdb.yaml`, NOT the actual AGFI or `bitstream_tar` itself (NOT something like `agfi-XYZ...`).

tracing

This section manages TracerV-based tracing at simulation runtime. For more details, see the *Capturing RISC-V Instruction Traces with TracerV* page for more details.

enable

This turns tracing on, when set to yes and off when set to no. See the *Enabling Tracing at Runtime*.

output_format

This sets the output format for TracerV tracing. See the *Selecting a Trace Output Format* section.

selector, start, and end

These configure triggering for TracerV. See the *Setting a TracerV Trigger* section.

autocounter

This section configures AutoCounter. See the *AutoCounter: Profiling with Out-of-Band Performance Counter Collection* page for more details.

read_rate

This sets the rate at which AutoCounters are read. See the *AutoCounter Runtime Parameters* section for more details.

workload

This section defines the software that will run on the simulated system.

workload_name

This selects a workload to run across the set of simulated nodes. A workload consists of a series of jobs that need to be run on simulated nodes (one job per node).

Workload definitions are located in firesim/deploy/workloads/*.json.

Some sample workloads:

br-base-uniform.json: This runs the default FireSim Linux distro on as many nodes as you specify when setting the target_config parameters.

Others can be found in the aforementioned directory. For a description of the JSON format, see *[DEPRECATED] Defining Custom Workloads*.

terminate_on_completion

Set this to no if you want your Run Farm to keep running once the workload has completed. Set this to yes if you want your Run Farm to be TERMINATED after the workload has completed and results have been copied off.

suffix_tag

This allows you to append a string to a workload's output directory name, useful for differentiating between successive runs of the same workload, without renaming the entire workload. For example, specifying suffix_tag: test-v1 with a workload named super-application will result in a workload results directory named results-workload/DATE--TIME-super-application-test-v1/.

host_debug

zero_out_dram

Set this to yes to zero-out FPGA-attached DRAM before simulation begins. This process takes 2-5 minutes. In general, this is not required to produce deterministic simulations on target machines running linux, but should be enabled if you observe simulation non-determinism.

disable_synth_asserts

Set this to yes to make the simulation ignore synthesized assertions when they fire. Otherwise, simulation will print the assertion message and terminate when an assertion fires.

12.5.2 config_build.yaml

Here is a sample of this configuration file:

```
# Build-time build design / AGFI configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

# this refers to build farms defined in config_build_farm.yaml
build_farm:
  # managerinit replace start
  base_recipe: build-farm-recipes/aws_ec2.yaml
  # Uncomment and add args to override defaults.
  # Arg structure should be identical to the args given
  # in the base_recipe.
  #recipe_arg_overrides:
  #  <ARG>: <OVERRIDE>
  # managerinit replace end

builds_to_run:
    # this section references builds defined in config_build_recipes.yaml
    # if you add a build here, it will be built when you run buildbitstream
    - midasexamples_gcd
```

(continues on next page)

(continued from previous page)

```
agfis_to_share:
    - midasexamples_gcd

share_with_accounts:
    # To share with a specific user:
    somebodysname: 123456789012
    # To share publicly:
    # public: public
```

Below, we outline each mapping in detail.

build_farm

In this section, you specify the specific build farm configuration that you wish to use to build FPGA bitstreams.

base_recipe

The `base_recipe` key/value pair specifies the default set of arguments to use for a particular build farm type. To change the build farm type, a new `base_recipe` file must be provided from `deploy/build-farm-recipes`. You are able to override the arguments given by a `base_recipe` by adding keys/values to the `recipe_arg_overrides` mapping.

See *Build Farm Recipes (build-farm-recipes/*)* for more details on the potential build farm recipes that can be used.

recipe_arg_overrides

This optional mapping of keys/values allows you to override the default arguments provided by the `base_recipe`. This mapping must match the same mapping structure as the `args` mapping within the `base_recipe` file given. Overridden arguments override recursively such that all key/values present in the override args replace the default arguments given by the `base_recipe`. In the case of sequences, a overridden sequence completely replaces the corresponding sequence in the default args. Additionally, it is not possible to change the default build farm type through these overrides. This must be done by changing the default `base_recipe`.

builds_to_run

In this section, you can list as many build entries as you want to run for a particular call to the `buildbitstream` command (see *config_build_recipes.yaml* below for how to define a build entry). For example, if we want to run the builds named `awesome_firesim_config` and `quad_core_awesome_firesim_config`, we would write:

```
builds_to_run:
    - awesome_firesim_config
    - quad_core_awesome_firesim_config
```

agfis_to_share

Note

This is only used in the AWS EC2 case.

This is used by the shareagfi command to share the specified agfis with the users specified in the next (share_with_accounts) section. In this section, you should specify the section title (i.e. the name you made up) for a hardware configuration in config_hwdb.yaml. For example, to share the hardware config:

```
firesim_rocket_quadcore_nic_l2_llc4mb_ddr3:
    # this is a comment that describes my favorite configuration!
    agfi: agfi-0a6449b5894e96e53
    deploy_quintuplet_override: null
    custom_runtime_config: null
```

you would use:

```
agfis_to_share:
    - firesim_rocket_quadcore_nic_l2_llc4mb_ddr3
```

share_with_accounts

Note

This is only used in the AWS EC2 case.

A list of AWS account IDs that you want to share the AGFIs listed in agfis_to_share with when calling the manager's shareagfi command. You should specify names in the form usersname: AWSACCTID. The left-hand-side is just for human readability, only the actual account IDs listed here matter. If you specify public: public here, the AGFIs are shared publicly, regardless of any other entires that are present.

12.5.3 config_build_recipes.yaml

Here is a sample of this configuration file:

```
# Build-time build recipe configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

# this file contains sections that describe hardware designs that /can/ be built.
# edit config_build.yaml to actually "turn on" a config to be built when you run
# buildbitstream

###########
# Schema:
###########
# <NAME>:
#    PLATFORM: <string>
```

(continues on next page)

(continued from previous page)

```
#    TARGET_PROJECT: <string>
#    TARGET_PROJECT_MAKEFRAG: <string | null>
#    DESIGN: <string>
#    TARGET_CONFIG: <string>
#    PLATFORM_CONFIG: <string>
#    deploy_quintuplet: <string | null>
#    platform_config_args:
#       fpga_frequency: <int>
#       build_strategy: <string>
#    post_build_hook: <string | null>
#    metasim_customruntimeconfig: <string | null>
#    bit_builder_recipe: <string>
#    # OPTIONAL: overrides for bit builder recipe
#    # Arg structure should be identical to the args given
#    # in the base_recipe.
#    #bit_builder_arg_overrides:
#    #  <ARG>: <OVERRIDE>

# MIDAS Examples -- BUILD SUPPORT ONLY; Can't launch driver correctly on run farm
midasexamples_gcd:
    PLATFORM: f1
    TARGET_PROJECT: midasexamples
    TARGET_PROJECT_MAKEFRAG: null
    DESIGN: GCD
    TARGET_CONFIG: NoConfig
    PLATFORM_CONFIG: DefaultF1Config
    deploy_quintuplet: null
    platform_config_args:
        fpga_frequency: 75
        build_strategy: TIMING
    post_build_hook: null
    metasim_customruntimeconfig: null
    bit_builder_recipe: bit-builder-recipes/f1.yaml

#########################################################################################
↪#######
# FireAxe Examples
#########################################################################################
↪#######

# DOC include start: F1 Rocket Partition Build Recipe
#########################################################################################
↪#######
# Fast-mode : pull out a RocketTile out from your SoC
#########################################################################################
↪#######
# f1_rocket_split_soc_fast:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: RocketTileF1PCIMBase
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
```

(continues on next page)

(continued from previous page)

```
#     ...
#
# f1_rocket_split_tile_fast:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: RocketTileF1PCIMPartition0
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
# DOC include end: F1 Rocket Partition Build Recipe

# DOC include start: F1 Exact Rocket Partition Build Recipe
#########################################################################################
↪#######
# Exact-mode : pull out a RocketTile out from your SoC
#########################################################################################
↪#######
# f1_firesim_rocket_soc_exact:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: ExactMode_RocketTileF1PCIMBase
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
#
# f1_firesim_rocket_tile_exact:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: ExactMode_RocketTileF1PCIMPartition0
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
# DOC include end: F1 Exact Rocket Partition Build Recipe

##############################################################################
# Splitting the design onto 3 FPGAs
##############################################################################
# xilinx_u250_firesim_dual_rocket_split_base:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: WithDefaultFireSimBridges_WithFireSimConfigTweaks_chipyard.
↪DualRocketConfig
#     PLATFORM_CONFIG: DualRocketTileQSFPBase
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
#
# xilinx_u250_firesim_dual_rocket_split_0:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: WithDefaultFireSimBridges_WithFireSimConfigTweaks_chipyard.
↪DualRocketConfig
#     PLATFORM_CONFIG: DualRocketTileQSFP0
```

(continues on next page)

(continued from previous page)

```
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
#
# xilinx_u250_firesim_dual_rocket_split_1:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: WithDefaultFireSimBridges_WithFireSimConfigTweaks_chipyard.
↪DualRocketConfig
#     PLATFORM_CONFIG: DualRocketTileQSFP1
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...

# DOC include start: Xilinx U250 NoC Partition Build Recipe
#############################################################################
# Using the NoC-partition-mode to partition the design across 3 FPGAs
# connected as a ring.
#############################################################################
# xilinx_u250_quad_rocket_ring_base:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: FireSimQuadRocketSbusRingNoCConfig
#     PLATFORM_CONFIG: QuadTileRingNoCBase
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
#
# xilinx_u250_quad_rocket_ring_0:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: FireSimQuadRocketSbusRingNoCConfig
#     PLATFORM_CONFIG: QuadTileRingNoC0
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
#
# xilinx_u250_quad_rocket_ring_1:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: FireSimQuadRocketSbusRingNoCConfig
#     PLATFORM_CONFIG: QuadTileRingNoC1
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
# DOC include end: Xilinx U250 NoC Partition Build Recipe
```

Below, we outline each section and parameter in detail.

Build definition sections, e.g. `awesome_firesim_config`

In this file, you can specify as many build definition sections as you want, each with a header like `awesome_firesim_config` (i.e. a nice, short name you made up). Such a section must contain the following fields:

`DESIGN`

This specifies the basic target design that will be built. Unless you are defining a custom system, this should be set to `FireSim`. We describe this in greater detail in *Generating Different Targets*).

`TARGET_CONFIG`

This specifies the hardware configuration of the target being simulated. Some examples include `FireSimRocketConfig` and `FireSimQuadRocketConfig`. We describe this in greater detail in *Generating Different Targets*).

`TARGET_PROJECT_MAKEFRAG`

This specifies where to find the FireSim makefile hooks to configure FireSim as a library in a larger project. See how Chipyard integrates FireSim for how to use this. This path can be an absolute path or relative to the location of the recipe file itself.

`PLATFORM_CONFIG`

This specifies parameters to pass to the compiler (Golden Gate). Notably, PLATFORM_CONFIG can be used to enable debugging tools like assertion synthesis, and resource optimizations like instance multithreading. Critically, it also calls out the host-platform (e.g., F1) to compile against: this defines the widths of internal simulation interfaces and specifies resource limits (e.g., how much DRAM is available on the platform).

`platform_config_args`

These configure the bitstream build, and are host-platform-agnostic. Platform-specific arguments, like the Vitis platform ("DEVICE"), are captured as arguments to the bitbuilder.

`fpga_frequency`

Specifies the host FPGA frequency for a bitstream build.

build_strategy

Specifies a pre-canned set of strategies and directives to pass to the bitstream build. Note, these are implemented differently on different host platforms, but try to optimize for the same things. Strategies supported across both Vitis, Xilinx Alveo U200/U250/U280, and EC2 F1 include:

- TIMING: Optimize for improved fmax.
- AREA: Optimize for reduced resource utilization.

Names are derived AWS's strategy set.

TARGET_PROJECT *(Optional)*

This specifies the target project in which the target is defined (this is described in greater detail *here*). If TARGET_PROJECT is undefined the manager will default to firesim. Setting TARGET_PROJECT is required for building the MIDAS examples (TARGET_PROJECT: midasexamples) with the manager, or for building a user-provided target project.

PLATFORM *(Optional)*

This specifies the platform for which the target will be built for (this is described in greater detail *here*). If PLATFORM is undefined the manager will default to f1.

deploy_quintuplet

This allows you to override the deployquintuplet stored with the AGFI. Otherwise, the PLATFORM/TARGET_PROJECT/DESIGN/TARGET_CONFIG/PLATFORM_CONFIG you specify above will be used. See the AGFI Tagging section for more details. Most likely, you should leave this set to null. This is usually only used if you have proprietary RTL that you bake into an FPGA image, but don't want to share with users of the simulator.

post_build_hook

(Optional) Provide an a script to run on the results copied back from a _single_ build instance. Upon completion of each design's build, the manager invokes this script and passing the absolute path to that instance's build-results directory as it's first argument.

metasim_customruntimeconfig

This is an advanced feature - under normal conditions, you can use the default parameters generated automatically by the simulator by setting this field to null for metasimulations. If you want to customize runtime parameters for certain parts of the metasimulation (e.g. the DRAM model's runtime parameters), you can place a custom config file in sim/custom-runtime-configs/. Then, set this field to the relative name of the config. For example, sim/custom-runtime-configs/GREATCONFIG.conf becomes metasim_customruntimeconfig: GREATCONFIG.conf.

bit_builder_recipe

This specifies the bitstream type to generate for a particular recipe. This must point to a file in deploy/bit-builder-recipes/. See *Bit Builder Recipes (bit-builder-recipes/*)* for more details on bit builders and their arguments.

bit_builder_arg_overrides

This optional mapping of keys/values allows you to override the default arguments provided by the bit_builder_recipe. This mapping must match the same mapping structure as the args mapping within the bit_builder_recipe file given. Overridden arguments override recursively such that all key/values present in the override args replace the default arguments given by the bit_builder_recipe. In the case of sequences, a overridden sequence completely replaces the corresponding sequence in the default args. Additionally, it is not possible to change the default bit builder type through these overrides. This must be done by changing the default bit_builder_recipe.

12.5.4 config_hwdb.yaml

Here is a sample of this configuration file:

```
# Hardware config database for FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

# Hardware configs represent a combination of:
#   - an agfi/bitstream_tar with the bitstream for an fpga
#   - (optional) a deployquintuplet override
#   - (optional) a deploy_makefrag_override
#   - (optional) a custom_runtime_config

# The HWDBs (and their AGFI's/bitstream_tar's) provided below are public and available to␣
↪all users.

# DOCREF START: Example HWDB Entry
midasexamples_gcd:
    bitstream_tar: https://raw.githubusercontent.com/invalid/address.tar.gz
    deploy_quintuplet_override: null
    deploy_makefrag_override: null
    custom_runtime_config: null
# DOCREF END: Example HWDB Entry
```

This file tracks hardware configurations that you can deploy as simulated nodes in FireSim. Each such configuration contains a name for easy reference in higher-level configurations, defined in the section header, an handle to a bitstream (i.e. an AGFI or bitstream_tar path), which represents the FPGA image, a custom runtime config, if one is needed, and a deploy quintuplet override if one is necessary.

When you build a new bitstream, you should put it in this file so that it can be referenced from your other configuration files.

The following is an example section from this file - you can add as many of these as necessary:

```
midasexamples_gcd:
    bitstream_tar: https://raw.githubusercontent.com/invalid/address.tar.gz
```

(continues on next page)

(continued from previous page)

```
    deploy_quintuplet_override: null
    deploy_makefrag_override: null
    custom_runtime_config: null
```

Here are the components of these entries:

The name: firesim_boom_singlecore_nic_l2_llc4mb_ddr3

In this example, `firesim_boom_singlecore_nic_l2_llc4mb_ddr3` is the name that will be used to reference this hardware design in other configuration locations. The following items describe this hardware configuration:

agfi

This represents the AGFI (FPGA Image) used by this hardware configuration. Only used in AWS EC2 F1 FireSim configurations (a `bitstream_tar` key/value cannot exist with this key/value in the same recipe).

bitstream_tar

This is not shown in the example entry above, but would be used for an on-premises bitstream.

Indicates where the bitstream (FPGA Image) and metadata associated with it is located, may be one of:

- A Uniform Resource Identifier (URI), (see *Manager URI Paths* for details)
- A filesystem path available to the manager. Local paths are relative to the *deploy* folder.

deploy_quintuplet_override

This is an advanced feature - under normal conditions, you should leave this set to `null`, so that the manager uses the configuration quintuplet that is automatically stored with the bitstream metadata at build time. Advanced users can set this to a different value to build and use a different driver when deploying simulations. Since the driver depends on logic now hardwired into the FPGA bitstream, drivers cannot generally be changed without requiring FPGA recompilation.

deploy_makefrag_override

This is an advanced feature - under normal conditions, you should leave this set to `null`, so that the manager uses the makefrag path that was automatically stored with the bitstream metadata at build time. Advanced users can set this to a different value to override the makefile hooks used to build the driver.

custom_runtime_config

This is an advanced feature - under normal conditions, you can use the default parameters generated automatically by the simulator by setting this field to `null`. If you want to customize runtime parameters for certain parts of the simulation (e.g. the DRAM model's runtime parameters), you can place a custom config file in `sim/custom-runtime-configs/`. Then, set this field to the relative name of the config. For example, `sim/custom-runtime-configs/GREATCONFIG.conf` becomes `custom_runtime_config: GREATCONFIG.conf`.

driver_tar

The value for this key can be one of:

- A Uniform Resource Identifier (URI), (see *Manager URI Paths* for details)
- A filesystem path available to the manager. Local paths are relative to the *deploy* folder.

When this key is present, the FireSim FPGA-driver software will not be built from source. Instead, during *firesim infrasetup*, this file will be deployed and extracted into the *sim_slot_X* folder on the run farm instance. This file may be a *.tar*, *.tar.gz*, *.tar.bz2* or any other format that GNU tar (version 1.26) can automatically detect. The purpose of this feature is to enable advanced CI configurations where the driver build step is decoupled. For now this can only accept a path to a file on the manager's local filesystem. This key can also be a URI.

Add more hardware config sections, like NAME_GOES_HERE_2

You can add as many of these entries to `config_hwdb.yaml` as you want, following the format discussed above (i.e. you provide `agfi` or `bitstream_tar`, `deploy_quintuplet_override`, and `custom_runtime_config`).

12.5.5 Run Farm Recipes (run-farm-recipes/*)

Here is an example of this configuration file:

```
# AWS EC2 run farm hosts recipe.
# all fields are required but can be overridden in the `*_runtime.yaml`

run_farm_type: AWSEC2F1
args:
    # managerinit arg start
    # tag to apply to run farm hosts
    run_farm_tag: mainrunfarm
    # enable expanding run farm by run_farm_hosts given
    always_expand_run_farm: true
    # minutes to retry attempting to request instances
    launch_instances_timeout_minutes: 60
    # run farm host market to use (ondemand, spot)
    run_instance_market: ondemand
    # if using spot instances, determine the interrupt behavior (terminate, stop,␣
↪hibernate)
    spot_interruption_behavior: terminate
    # if using spot instances, determine the max price
    spot_max_price: ondemand
    # default location of the simulation directory on the run farm host
    default_simulation_dir: /home/centos

    # run farm hosts to spawn: a mapping from a spec below (which is an EC2
    # instance type) to the number of instances of the given type that you
    # want in your runfarm.
    run_farm_hosts_to_use:
      - f1.16xlarge: 0
      - f1.4xlarge: 0
      - f1.2xlarge: 1
      - m4.16xlarge: 0
```

(continues on next page)

(continued from previous page)

```
    - z1d.3xlarge: 0
    - z1d.6xlarge: 0
    - z1d.12xlarge: 0
  # managerinit arg end

  # REQUIRED: List of host "specifications", i.e. re-usable collections of
  # host parameters.
  #
  # On EC2, most users will never need to edit this section,
  # unless you want to add new host instance types.
  #
  # The "name" of a spec below (e.g. "f1.2xlarge" below) MUST be a valid EC2
  # instance type and is used to refer to the spec above.
  #
  # Besides required parameters shown below, each can have multiple OPTIONAL
  # arguments, called "override_*", corresponding to the "default_*" arguments
  # specified above. Each "override_*" argument overrides the corresponding
  # "default_*" argument in that run host spec.
  #
  # Optional params include:
  #       override_simulation_dir: /scratch/specific-build-host-build-dir
  #       override_platform: EC2InstanceDeployManager
  run_farm_host_specs:
    - f1.2xlarge: # REQUIRED: On EC2, the spec name MUST be an EC2 instance type.
        # REQUIRED: number of FPGAs on the machine
        num_fpgas: 1
        # REQUIRED: number of metasims this machine can host
        num_metasims: 0
        # REQUIRED: whether it is acceptable to use machines of this spec
        # to host ONLY switches (e.g. any attached FPGAs are "wasted")
        use_for_switch_only: false
    - f1.4xlarge:
        num_fpgas: 2
        num_metasims: 0
        use_for_switch_only: false
    - f1.16xlarge:
        num_fpgas: 8
        num_metasims: 0
        use_for_switch_only: false
    - m4.16xlarge:
        num_fpgas: 0
        num_metasims: 0
        use_for_switch_only: true
    - z1d.3xlarge:
        num_fpgas: 0
        num_metasims: 1
        use_for_switch_only: false
    - z1d.6xlarge:
        num_fpgas: 0
        num_metasims: 2
        use_for_switch_only: false
    - z1d.12xlarge:
```

(continues on next page)

(continued from previous page)

```
        num_fpgas: 0
        num_metasims: 8
        use_for_switch_only: false
```

run_farm_type

This key/value specifies a run farm class to use for launching, managing, and terminating run farm hosts used for simulations. By default, run farm classes can be found in deploy/runtools/run_farm.py. However, you can specify your own custom run farm classes by adding your python file to the PYTHONPATH. For example, to use the AWSEC2F1 run farm class, you would write `run_farm_type: AWSEC2F1`.

args

This section specifies all arguments needed for the specific `run_farm_type` used. For a list of arguments needed for a run farm class, users should refer to the `_parse_args` function in the run farm class given by `run_farm_type`.

aws_ec2.yaml run farm recipe

The run farm recipe shown above configures a FireSim run farm to use AWS EC2 instances. It contains several key/value pairs:

run_farm_tag

Use `run_farm_tag` to differentiate between different Run Farms in FireSim. Having multiple `config_runtime.yaml` files with different `run_farm_tag` values allows you to run many experiments at once from the same manager instance.

The instances launched by the `launchrunfarm` command will be tagged with this value. All later operations done by the manager rely on this tag, so you should not change it unless you are done with your current Run Farm.

Per AWS restrictions, this tag can be no longer than 255 characters.

always_expand_run_farm

When `true` (the default behavior when not given) the number of instances of each type (see `f1.16xlarges` etc. below) are launched every time you run `launchrunfarm`.

When `false`, `launchrunfarm` looks for already existing instances that match `run_farm_tag` and treat `f1.16xlarges` (and other 'instance-type' values below) as a total count.

For example, if you have `f1.2xlarges` set to 100 and the first time you run `launchrunfarm` you have `launch_instances_timeout_minutes` set to 0 (i.e. giveup after receiving a `ClientError` for each AvailabilityZone) and AWS is only able to provide you 75 `f1.2xlarges` because of capacity issues, `always_expand_runfarm` changes the behavior of `launchrunfarm` in subsequent attempts. `yes` means `launchrunfarm` will try to launch 100 `f1.2xlarges` again. `no` means that `launchrunfarm` will only try to launch an additional 25 `f1.2xlarges` because it will see that there are already 75 that have been launched with the same `run_farm_tag`.

launch_instances_timeout_minutes

Integer number of minutes that the launchrunfarm command will attempt to request new instances before giving up. This limit is used for each of the types of instances being requested. For example, if you set to 60, and you are requesting all four types of instances, launchrunfarm will try to launch each instance type for 60 minutes, possibly trying up to a total of four hours.

This limit starts to be applied from the first time launchrunfarm receives a ClientError response in all AvailabilityZones (AZs) for your region. In other words, if you request more instances than can possibly be requested in the given limit but AWS is able to satisfy all of the requests, the limit will not be enforced.

To experience the old (<= 1.12) behavior, set this limit to 0 and launchrunfarm will exit the first time it receives ClientError across all AZ's. The old behavior is also the default if launch_instances_timeout_minutes is not included.

run_instance_market

You can specify either spot or ondemand here, to use one of those markets on AWS.

spot_interruption_behavior

When run_instance_market: spot, this value determines what happens to an instance if it receives the interruption signal from AWS. You can specify either hibernate, stop, or terminate.

spot_max_price

When run_instance_market: spot, this value determines the max price you are willing to pay per instance, in dollars. You can also set it to ondemand to set your max to the on-demand price for the instance.

default_simulation_dir

This is the path on the run farm host that simulations will run out of.

run_farm_hosts_to_use

This is a sequence of unique specifications (given by run_farm_host_specs) to number of instances needed. Set these key/value pairs respectively based on the number and types of instances you need. While we could automate this setting, we choose not to, so that users are never surprised by how many instances they are running.

Note that these values are ONLY used to launch instances. After launch, the manager will query the AWS API to find the instances of each type that have the run_farm_tag set above assigned to them.

Also refer to always_expand_runfarm which determines whether launchrunfarm treats these counts as an incremental amount to be launched every time it is envoked or a total number of instances of that type and run_farm_tag that should be made to exist. Note, launchrunfarm will never terminate instances.

run_farm_host_specs

This is a sequence of specifications that describe a AWS EC2 instance and its properties. A sequence consists of the AWS EC2 instance name (i.e. `f1.2xlarge`) and number of FPGAs it supports (`num_fpgas`), number of metasims it could support (`num_metasims`), and if the instance should only host switch simulations (`use_for_switch_only`). Additionally, a specification can optionally add `override_simulation_dir` to override the `default_simulation_dir` for that specific run farm host. Similarly, a specification can optionally add `override_platform` to choose a different default deploy manager platform for that specific run farm host (for more details on this see the following section). By default, the deploy manager is setup for AWS EC2 simulations.

externally_provisioned.yaml run farm recipe

This run farm allows users to provide a list of pre-setup unmanaged run farm hosts (by hostname or IP address) that they can run simulations on. Note that this run farm type does not launch or terminate the run farm hosts. This functionality should be handled by the user. For example, users can use this run farm type to run simulations locally.

Here is an example of this configuration file:

```
# Unmanaged list of run farm hosts. Assumed that they are pre-setup to run simulations.
# all fields are required but can be overridden in the `*_runtime.yaml`

run_farm_type: ExternallyProvisioned
args:
    # managerinit arg start
    # REQUIRED: default platform used for run farm hosts. this is a class specifying
    # how to run simulations on a run farm host.
    default_platform: EC2InstanceDeployManager

    # REQUIRED: default directory where simulations are run out of on the run farm hosts
    default_simulation_dir: /home/centos

    # REQUIRED: default fpga db file that enumerates what fpgas are available on the␣
↪machine (used by XilinxU* Deploy Managers)
    default_fpga_db: /opt/firesim-db.json

    # REQUIRED: List of unique hostnames/IP addresses, each with their
    # corresponding specification that describes the properties of the host.
    #
    # Ex:
    # run_farm_hosts_to_use:
    #     # use localhost which is described by "four_fpgas_spec" below.
    #     - localhost: four_fpgas_spec
    #     # supply IP address, which points to a machine that is described
    #     # by "four_fpgas_spec" below.
    #     - "111.111.1.111": four_fpgas_spec
    run_farm_hosts_to_use:
        - localhost: one_fpgas_spec
    # managerinit arg end

    # REQUIRED: List of host "specifications", i.e. re-usable collections of
    # host parameters.
    #
    # The "name" of a spec (e.g. "four_fpgas_spec" below) is user-determined
```

(continues on next page)

(continued from previous page)

```
    # and is used to refer to the spec above.
    #
    # Besides required parameters shown below, each can have multiple OPTIONAL
    # arguments, called "override_*", corresponding to the "default_*" arguments
    # specified above. Each "override_*" argument overrides the corresponding
    # "default_*" argument in that run host spec.
    #
    # Optional params include:
    #       override_platform: EC2InstanceDeployManager
    #       override_simulation_dir: /scratch/specific-build-host-build-dir
    #       override_fpga_db: /opt/firesim-db-specific.json
    run_farm_host_specs:
        - four_fpgas_spec:
            # REQUIRED: number of FPGAs on the machine
            num_fpgas: 4
            # REQUIRED: number of metasims this machine can host
            num_metasims: 0
            # REQUIRED: whether it is acceptable to use machines of this spec
            # to host ONLY switches (e.g. any attached FPGAs are "wasted")
            use_for_switch_only: false

        - four_metasims_spec:
            num_fpgas: 0
            num_metasims: 4
            use_for_switch_only: false

        - switch_only_spec:
            num_fpgas: 0
            num_metasims: 0
            use_for_switch_only: true

        - one_fpga_spec:
            num_fpgas: 1
            num_metasims: 0
            use_for_switch_only: false

        - one_fpgas_spec:
            num_fpgas: 1
            num_metasims: 0
            use_for_switch_only: false

        - two_fpgas_spec:
            num_fpgas: 2
            num_metasims: 0
            use_for_switch_only: false

        - three_fpgas_spec:
            num_fpgas: 3
            num_metasims: 0
            use_for_switch_only: false

        - four_fpgas_spec:
```

(continues on next page)

(continued from previous page)

```
        num_fpgas: 4
        num_metasims: 0
        use_for_switch_only: false

    - five_fpgas_spec:
        num_fpgas: 5
        num_metasims: 0
        use_for_switch_only: false

    - six_fpgas_spec:
        num_fpgas: 6
        num_metasims: 0
        use_for_switch_only: false

    - seven_fpgas_spec:
        num_fpgas: 7
        num_metasims: 0
        use_for_switch_only: false

    - eight_fpgas_spec:
        num_fpgas: 8
        num_metasims: 0
        use_for_switch_only: false

    - nine_fpgas_spec:
        num_fpgas: 9
        num_metasims: 0
        use_for_switch_only: false

    - ten_fpgas_spec:
        num_fpgas: 10
        num_metasims: 0
        use_for_switch_only: false

    - eleven_fpgas_spec:
        num_fpgas: 11
        num_metasims: 0
        use_for_switch_only: false

    - twelve_fpgas_spec:
        num_fpgas: 12
        num_metasims: 0
        use_for_switch_only: false

    - thirteen_fpgas_spec:
        num_fpgas: 13
        num_metasims: 0
        use_for_switch_only: false

    - fourteen_fpgas_spec:
        num_fpgas: 14
        num_metasims: 0
```

(continues on next page)

(continued from previous page)

```
          use_for_switch_only: false

      - fifteen_fpgas_spec:
          num_fpgas: 15
          num_metasims: 0
          use_for_switch_only: false

      - sixteen_fpgas_spec:
          num_fpgas: 16
          num_metasims: 0
          use_for_switch_only: false
```

default_platform

This key/value specifies a default deploy platform (also known as a deploy manager) class to use for managing simulations across all run farm hosts. For example, this class manages how to flash FPGAs with bitstreams, how to copy back results, and how to check if a simulation is running. By default, deploy platform classes can be found in deploy/runtools/run_farm_deploy_managers.py. However, you can specify your own custom run farm classes by adding your python file to the `PYTHONPATH`. There are default deploy managers / platforms that correspond to AWS EC2 F1 FPGAs, Vitis FPGAs, Xilinx Alveo U200/U250/U280 FPGAs, Xilinx VCU118 FPGAs, and RHS Research Nitefury II FPGAs: `EC2InstanceDeployManager`, `VitisInstanceDeployManager`, `Xilinx{AlveoU200,AlveoU250,AlveoU280,VCU118}InstanceDeployManager`, and `RHSResearchNitefuryIIInstanceDeployManager` respectively. For example, to use the `EC2InstanceDeployManager` deploy platform class, you would write `default_platform: EC2InstanceDeployManager`.

default_simulation_dir

This is the default path on all run farm hosts that simulations will run out of.

run_farm_hosts_to_use

This is a sequence of unique hostnames/IP address to specifications (given by `run_farm_host_specs`). Set these key/value pairs respectively to map unmanaged run farm hosts to their description (given by a specification). For example, to run simulations locally, a user can write a sequence element with `- localhost: four_fpgas_spec` to indicate that `localhost` should be used and that it has a type of `four_fpgas_spec`.

run_farm_host_specs

This is a sequence of specifications that describe an unmanaged run farm host and its properties. A sequence consists of the specification name (i.e. `four_fpgas_spec`) and number of FPGAs it supports (`num_fpgas`), number of metasims it could support (`num_metasims`), and if the instance should only host switch simulations (`use_for_switch_only`). Additionally, a specification can optionally add `override_simulation_dir` to override the `default_simulation_dir` for that specific run farm host. Similarly, a specification can optionally add `override_platform` to choose a different `default_platform` for that specific run farm host.

12.5.6 Build Farm Recipes (build-farm-recipes/*)

Here is an example of this configuration file:

```
# Build-time build farm design configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.
# all fields are required but can be overridden in the `*_runtime.yaml`

###########
# Schema:
###########
#   # Class name of the build farm type.
#   # This can be determined from `deploy/buildtools/buildfarm.py`).
#   build_farm_type: <TYPE NAME>
#   args:
#       # Build farm arguments that are passed to the `BuildFarmHostDispatcher`
#       # object. Determined by looking at `parse_args` function of class.
#       <K/V pairs of args>

# Note: For large designs (ones that would fill a EC2 F1.2xlarge/Xilinx VU9P)
# Vivado uses in excess of 32 GiB. Keep this in mind when selecting a
# non-default instance type.
build_farm_type: AWSEC2
args:
    # managerinit arg start
    # tag to apply to build farm hosts
    build_farm_tag: mainbuildfarm
    # instance type to use per build
    instance_type: z1d.2xlarge
    # instance market to use per build (ondemand, spot)
    build_instance_market: ondemand
    # if using spot instances, determine the interrupt behavior (terminate, stop,␣
↪hibernate)
    spot_interruption_behavior: terminate
    # if using spot instances, determine the max price
    spot_max_price: ondemand
    # default location of build directory on build host
    default_build_dir: /home/centos/firesim-build
    # managerinit arg end
```

build_farm_type

This key/value specifies a build farm class to use for launching, managing, and terminating build farm hosts used for building bitstreams. By default, build farm classes can be found in deploy/buildtools/buildfarm.py. However, you can specify your own custom build farm classes by adding your python file to the PYTHONPATH. For example, to use the AWSEC2 build farm class, you would write `build_farm_type: AWSEC2`.

args

This section specifies all arguments needed for the specific `build_farm_type` used. For a list of arguments needed for a build farm class, users should refer to the `_parse_args` function in the build farm class given by `build_farm_type`.

aws_ec2.yaml build farm recipe

This build farm recipe configures a FireSim build farm to use AWS EC2 instances enabled with Vivado.

Here is an example of this configuration file:

```
# Build-time build farm design configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.
# all fields are required but can be overridden in the `*_runtime.yaml`

###########
# Schema:
###########
#   # Class name of the build farm type.
#   # This can be determined from `deploy/buildtools/buildfarm.py`).
#   build_farm_type: <TYPE NAME>
#   args:
#       # Build farm arguments that are passed to the `BuildFarmHostDispatcher`
#       # object. Determined by looking at `parse_args` function of class.
#       <K/V pairs of args>

# Note: For large designs (ones that would fill a EC2 F1.2xlarge/Xilinx VU9P)
# Vivado uses in excess of 32 GiB. Keep this in mind when selecting a
# non-default instance type.
build_farm_type: AWSEC2
args:
    # managerinit arg start
    # tag to apply to build farm hosts
    build_farm_tag: mainbuildfarm
    # instance type to use per build
    instance_type: z1d.2xlarge
    # instance market to use per build (ondemand, spot)
    build_instance_market: ondemand
    # if using spot instances, determine the interrupt behavior (terminate, stop,␣
↪hibernate)
    spot_interruption_behavior: terminate
    # if using spot instances, determine the max price
    spot_max_price: ondemand
    # default location of build directory on build host
```

(continues on next page)

(continued from previous page)

```
    default_build_dir: /home/centos/firesim-build
    # managerinit arg end
```

build_farm_tag

Use build_farm_tag to differentiate between different Build Farms used across **multiple FireSim repositories**. The instances launched by the buildbitstream command will be tagged with this value. Mainly for CI use.

Per AWS restrictions, this tag can be no longer than 255 characters.

instance_type

The AWS EC2 instance name to run a bitstream build on. Note that for large designs, Vivado uses an excess of 32 GiB so choose a non-default instance type wisely.

build_instance_market

You can specify either spot or ondemand here, to use one of those markets on AWS.

spot_interruption_behavior

When run_instance_market: spot, this value determines what happens to an instance if it receives the interruption signal from AWS. You can specify either hibernate, stop, or terminate.

spot_max_price

When build_instance_market: spot, this value determines the max price you are willing to pay per instance, in dollars. You can also set it to ondemand to set your max to the on-demand price for the instance.

default_build_dir

This is the path on the build farm host that bitstream builds will run out of.

externally_provisioned.yaml build farm recipe

This build farm recipe allows users to provide an list of pre-setup unmanaged build farm hosts (by hostname or IP address) that they can run bitstream builds on. Note that this build farm type does not launch or terminate the build farm hosts. This functionality should be handled by the user. For example, users can use this build farm type to run bitstream builds locally.

Here is an example of this configuration file:

```
# Build-time build farm design configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

```

(continues on next page)

(continued from previous page)

```
###########
# Schema:
###########
#   # Class name of the build farm type.
#   # This can be determined from `deploy/buildtools/buildfarm.py`).
#   build_farm_type: <TYPE NAME>
#   args:
#       # Build farm arguments that are passed to the `BuildFarmHostDispatcher`
#       # object. Determined by looking at `parse_args` function of class.
#       <K/V pairs of args>

# Unmanaged list of build hosts. Assumed that they are pre-setup to run builds.
build_farm_type: ExternallyProvisioned
args:
    # managerinit arg start
    # REQUIRED: (replace this) default location of build directory on build host.
    default_build_dir: null
    # REQUIRED: List of IP addresses (or "localhost"). Each can have an OPTIONAL
    # argument, called "override_build_dir", specifying to override the default
    # build directory.
    #
    # Ex:
    # build_farm_hosts:
    #     # use localhost and don't override the default build dir
    #     - localhost
    #     # use other IP address (don't override default build dir)
    #     - "111.111.1.111"
    #     # use other IP address (override default build dir for this build host)
    #     - "222.222.2.222":
    #         override_build_dir: /scratch/specific-build-host-build-dir
    build_farm_hosts:
        - localhost
    # managerinit arg end
```

`default_build_dir`

This is the default path on all the build farm hosts that bitstream builds will run out of.

build_farm_hosts

This is a sequence of unique hostnames/IP addresses that should be used as build farm hosts. Each build farm host (given by the unique hostname/IP address) can have an optional mapping that provides an `override_build_dir` that overrides the `default_build_dir` given just for that build farm host.

12.5.7 Bit Builder Recipes (bit-builder-recipes/*)

Here is an example of this configuration file:

```
# Build-time bitbuilder design configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

###########
# Schema:
###########
# # Class name of the bitbuilder type.
# # This can be determined from `deploy/buildtools/bitbuilder.py`).
# bitbuilder_type: <TYPE NAME>
# args:
#     # Bitbuilder arguments that are passed to the `BitBuilder`
#     # object. Determined by looking at `_parse_args` function of class.
#     <K/V pairs of args>

bit_builder_type: F1BitBuilder
args:
    # REQUIRED: name of s3 bucket
    s3_bucket_name: firesim
    # REQUIRED: append aws username and current region to s3_bucket_name?
    append_userid_region: true
```

bit_builder_type

This key/value specifies a bit builder class to use for building bitstreams. By default, bit builder classes can be found in deploy/buildtools/bitbuilder.py. However, you can specify your own custom bit builder classes by adding your python file to the `PYTHONPATH`. For example, to use the `F1BitBuilder` build farm class, you would write `bit_builder_type: F1BitBuilder`.

args

This section specifies all arguments needed for the specific `bit_builder_type` used. For a list of arguments needed for a bit builder class, users should refer to the `_parse_args` function in the bit builder class given by `bit_builder_type`.

f1.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an AWS EC2 F1 AGFI (FPGA bitstream).

Here is an example of this configuration file:

```
# Build-time bitbuilder design configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

###########
# Schema:
###########
# # Class name of the bitbuilder type.
# # This can be determined from `deploy/buildtools/bitbuilder.py`).
# bitbuilder_type: <TYPE NAME>
# args:
#     # Bitbuilder arguments that are passed to the `BitBuilder`
#     # object. Determined by looking at `_parse_args` function of class.
#     <K/V pairs of args>

bit_builder_type: F1BitBuilder
args:
    # REQUIRED: name of s3 bucket
    s3_bucket_name: firesim
    # REQUIRED: append aws username and current region to s3_bucket_name?
    append_userid_region: true
```

s3_bucket_name

This is used behind the scenes in the AGFI creation process. You will only ever need to access this bucket manually if there is a failure in AGFI creation in Amazon's backend.

Naming rules: this must be all lowercase and you should stick to letters and numbers ([a-z0-9]).

The first time you try to run a build, the FireSim manager will try to create the bucket you name here. If the name is unavailable, it will complain and you will need to change this name. Once you choose a working name, you should never need to change it.

In general, `firesim-yournamehere` is a good choice.

append_userid_region

When enabled, this appends the current users AWS user ID and region to the `s3_bucket_name`.

vitis.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an Vitis bitstream (FPGA bitstream called an xclbin, packaged into a bitstream_tar).

device

This specifies a Vitis platform to compile against, for example: xilinx_u250_gen3x16_xdma_3_1_202020_1 when targeting a Vitis-enabled Alveo U250 FPGA.

Here is an example of this configuration file:

```
# Build-time bitbuilder design configuration for the FireSim Simulation Manager
# See https://docs.fires.im/en/stable/Advanced-Usage/Manager/Manager-Configuration-Files.
↪html for documentation of all of these params.

###########
# Schema:
###########
# # Class name of the bitbuilder type.
# # This can be determined from `deploy/buildtools/bitbuilder.py`).
# bitbuilder_type: <TYPE NAME>
# args:
#     # Bitbuilder arguments that are passed to the `BitBuilder`
#     # object. Determined by looking at `_parse_args` function of class.
#     <K/V pairs of args>

bit_builder_type: VitisBitBuilder
args:
    # REQUIRED: vitis fpga platform
    device: xilinx_u250_gen3x16_xdma_4_1_202210_1
```

xilinx_alveo_u200.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an Xilinx Alveo U200 bitstream, packaged into a bitstream_tar.

xilinx_alveo_u250.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an Xilinx Alveo U250 bitstream, packaged into a bitstream_tar.

xilinx_alveo_u280.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an Xilinx Alveo U280 bitstream, packaged into a bitstream_tar.

xilinx_vcu118.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an Xilinx VCU118 bitstream, packaged into a bitstream_tar.

rhsresearch_nitefury_ii.yaml bit builder recipe

This bit builder recipe configures a build farm host to build an RHS Research Nitefury II bitstream, packaged into a bitstream_tar.

12.6 Manager Environment Variables

This page contains a centralized reference for the environment variables used by the manager.

12.6.1 FIRESIM_RUNFARM_PREFIX

This environment variable is used to prefix all Run Farm tags with some prefix in the AWS EC2 case. This is useful for separating run farms between multiple copies of FireSim.

12.6.2 FIRESIM_BUILDFARM_PREFIX

This environment variable is used to prefix all Build Farm tags with some prefix in the AWS EC2 case. This is mainly for CI use only.

12.7 Manager Network Topology Definitions (user_topology.py)

Custom network topologies are specified as Python snippets that construct a tree. You can see examples of these in deploy/runtools/user_topology.py, shown below. Better documentation of this API will be available once it stabilizes.

Fundamentally, you create a list of roots, which consists of switch or server nodes, then construct a tree by adding downlinks to these roots. Since links are bi-directional, adding a downlink from node A to node B implicitly adds an uplink from B to A.

You can add additional topology generation methods here, then use them in config_runtime.yaml.

12.7.1 user_topology.py contents:

```
""" Define your additional topologies here. The FireSimTopology class inherits
from UserToplogies and thus can instantiate your topology. """

from __future__ import annotations

from runtools.firesim_topology_elements import (
    FireSimPipeNode,
    FireSimSwitchNode,
    FireSimServerNode,
    FireSimSuperNodeServerNode,
    FireSimDummyServerNode,
    FireSimNode,
)
from runtools.simulation_data_classes import (
    PartitionConfig,
    PartitionMode,
    PartitionNode,
    FireAxeEdge,
    FireAxeNodeBridgePair,
)

from typing import (
    Optional,
    Union,
    Callable,
    Sequence,
    TYPE_CHECKING,
    cast,
    List,
    Any,
    Dict,
)

if TYPE_CHECKING:
    from runtools.firesim_topology_with_passes import FireSimTopologyWithPasses


class UserTopologies:
    """A class that just separates out user-defined/configurable topologies
    from the rest of the boilerplate in FireSimTopology()"""

    no_net_num_nodes: int
    custom_mapper: Optional[Union[Callable, str]]
    roots: Sequence[FireSimNode]

    def __init__(self, no_net_num_nodes: int) -> None:
        self.no_net_num_nodes = no_net_num_nodes
        self.custom_mapper = None
        self.roots = []

    def clos_m_n_r(self, m: int, n: int, r: int) -> None:
```

(continues on next page)

(continued from previous page)

```
        """DO NOT USE THIS DIRECTLY, USE ONE OF THE INSTANTIATIONS BELOW."""
        """ Clos topol where:
        m = number of root switches
        n = number of links to nodes on leaf switches
        r = number of leaf switches

        and each leaf switch has a link to each root switch.

        With the default mapping specified below, you will need:
        m switch nodes (on F1: m4.16xlarges).
        n fpga nodes (on F1: f1.16xlarges).

        TODO: improve this later to pack leaf switches with <= 4 downlinks onto
        one 16x.large.
        """

        rootswitches = [FireSimSwitchNode() for x in range(m)]
        self.roots = rootswitches
        leafswitches = [FireSimSwitchNode() for x in range(r)]
        servers = [[FireSimServerNode() for x in range(n)] for y in range(r)]
        for rswitch in rootswitches:
            rswitch.add_downlinks(leafswitches)

        for leafswitch, servergroup in zip(leafswitches, servers):
            leafswitch.add_downlinks(servergroup)

        def custom_mapper(fsim_topol_with_passes: FireSimTopologyWithPasses) -> None:
            for i, rswitch in enumerate(rootswitches):
                switch_inst_handle = (
                    fsim_topol_with_passes.run_farm.get_switch_only_host_handle()
                )
                switch_inst = fsim_topol_with_passes.run_farm.allocate_sim_host(
                    switch_inst_handle
                )
                switch_inst.add_switch(rswitch)

            for j, lswitch in enumerate(leafswitches):
                numsims = len(servers[j])
                inst_handle = (
                    fsim_topol_with_passes.run_farm.get_smallest_sim_host_handle(
                        num_sims=numsims
                    )
                )
                sim_inst = fsim_topol_with_passes.run_farm.allocate_sim_host(
                    inst_handle
                )
                sim_inst.add_switch(lswitch)
                for sim in servers[j]:
                    sim_inst.add_simulation(sim)

        self.custom_mapper = custom_mapper
```

(continues on next page)

(continued from previous page)

```
    def clos_2_8_2(self) -> None:
        """clos topol with:
        2 roots
        8 nodes/leaf
        2 leaves."""
        self.clos_m_n_r(2, 8, 2)

    def clos_8_8_16(self) -> None:
        """clos topol with:
        8 roots
        8 nodes/leaf
        16 leaves. = 128 nodes."""
        self.clos_m_n_r(8, 8, 16)

    def fat_tree_4ary(self) -> None:
        # 4-ary fat tree as described in
        # http://ccr.sigcomm.org/online/files/p63-alfares.pdf
        coreswitches = [FireSimSwitchNode() for x in range(4)]
        self.roots = coreswitches
        aggrswitches = [FireSimSwitchNode() for x in range(8)]
        edgeswitches = [FireSimSwitchNode() for x in range(8)]
        servers = [FireSimServerNode() for x in range(16)]
        for switchno in range(len(coreswitches)):
            core = coreswitches[switchno]
            base = 0 if switchno < 2 else 1
            dls = list(map(lambda x: aggrswitches[x], range(base, 8, 2)))
            core.add_downlinks(dls)
        for switchbaseno in range(0, len(aggrswitches), 2):
            switchno = switchbaseno + 0
            aggr = aggrswitches[switchno]
            aggr.add_downlinks([edgeswitches[switchno], edgeswitches[switchno + 1]])
            switchno = switchbaseno + 1
            aggr = aggrswitches[switchno]
            aggr.add_downlinks([edgeswitches[switchno - 1], edgeswitches[switchno]])
        for edgeno in range(len(edgeswitches)):
            edgeswitches[edgeno].add_downlinks(
                [servers[edgeno * 2], servers[edgeno * 2 + 1]]
            )

        def custom_mapper(fsim_topol_with_passes: FireSimTopologyWithPasses) -> None:
            """In a custom mapper, you have access to the firesim topology with passes,
            where you can access the run_farm nodes:

            Requires 2 fpga nodes w/ 8+ fpgas and 1 switch node

            To map, call add_switch or add_simulation on run farm instance
            objs in the aforementioned arrays.

            Because of the scope of this fn, you also have access to whatever
            stuff you created in the topology itself, which we expect will be
            useful for performing the mapping."""
```

(continues on next page)

(continued from previous page)

```
        # map the fat tree onto one switch host instance (for core switches)
        # and two 8-sim-slot (e.g. 8-fpga) instances
        # (e.g., two pods of aggr/edge/4sims per f1.16xlarge)

        switch_inst_handle = (
            fsim_topol_with_passes.run_farm.get_switch_only_host_handle()
        )
        switch_inst = fsim_topol_with_passes.run_farm.allocate_sim_host(
            switch_inst_handle
        )
        for core in coreswitches:
            switch_inst.add_switch(core)

        eight_sim_host_handle = (
            fsim_topol_with_passes.run_farm.get_smallest_sim_host_handle(num_sims=8)
        )
        sim_hosts = [
            fsim_topol_with_passes.run_farm.allocate_sim_host(eight_sim_host_handle)
            for _ in range(2)
        ]

        for aggrsw in aggrswitches[:4]:
            sim_hosts[0].add_switch(aggrsw)
        for aggrsw in aggrswitches[4:]:
            sim_hosts[1].add_switch(aggrsw)

        for edgesw in edgeswitches[:4]:
            sim_hosts[0].add_switch(edgesw)
        for edgesw in edgeswitches[4:]:
            sim_hosts[1].add_switch(edgesw)

        for sim in servers[:8]:
            sim_hosts[0].add_simulation(sim)
        for sim in servers[8:]:
            sim_hosts[1].add_simulation(sim)

    self.custom_mapper = custom_mapper

def example_multilink(self) -> None:
    self.roots = [FireSimSwitchNode()]
    midswitch = FireSimSwitchNode()
    lowerlayer = [midswitch for x in range(16)]
    self.roots[0].add_downlinks(lowerlayer)
    servers = [FireSimServerNode()]
    midswitch.add_downlinks(servers)

def example_multilink_32(self) -> None:
    self.roots = [FireSimSwitchNode()]
    midswitch = FireSimSwitchNode()
    lowerlayer = [midswitch for x in range(32)]
    self.roots[0].add_downlinks(lowerlayer)
    servers = [FireSimServerNode()]
```

(continues on next page)

(continued from previous page)

```
        midswitch.add_downlinks(servers)

    def example_multilink_64(self) -> None:
        self.roots = [FireSimSwitchNode()]
        midswitch = FireSimSwitchNode()
        lowerlayer = [midswitch for x in range(64)]
        self.roots[0].add_downlinks(lowerlayer)
        servers = [FireSimServerNode()]
        midswitch.add_downlinks(servers)

    def example_cross_links(self) -> None:
        self.roots = [FireSimSwitchNode() for x in range(2)]
        midswitches = [FireSimSwitchNode() for x in range(2)]
        self.roots[0].add_downlinks(midswitches)
        self.roots[1].add_downlinks(midswitches)
        servers = [FireSimServerNode() for x in range(2)]
        midswitches[0].add_downlinks([servers[0]])
        midswitches[1].add_downlinks([servers[1]])

    def small_hierarchy_8sims(self) -> None:
        self.custom_mapper = "mapping_use_one_8_slot_node"
        self.roots = [FireSimSwitchNode()]
        midlevel = [FireSimSwitchNode() for x in range(4)]
        servers = [[FireSimServerNode() for x in range(2)] for x in range(4)]
        self.roots[0].add_downlinks(midlevel)
        for swno in range(len(midlevel)):
            midlevel[swno].add_downlinks(servers[swno])

    def small_hierarchy_2sims(self) -> None:
        self.custom_mapper = "mapping_use_one_8_slot_node"
        self.roots = [FireSimSwitchNode()]
        midlevel = [FireSimSwitchNode() for x in range(1)]
        servers = [[FireSimServerNode() for x in range(2)] for x in range(1)]
        self.roots[0].add_downlinks(midlevel)
        for swno in range(len(midlevel)):
            midlevel[swno].add_downlinks(servers[swno])

    def example_1config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(1)]
        self.roots[0].add_downlinks(servers)

    def example_2config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(2)]
        self.roots[0].add_downlinks(servers)

    def example_4config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(4)]
        self.roots[0].add_downlinks(servers)
```

(continues on next page)

(continued from previous page)

```
    def example_8config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = [FireSimServerNode() for y in range(8)]
        self.roots[0].add_downlinks(servers)

    def example_16config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(2)]
        servers = [[FireSimServerNode() for y in range(8)] for x in range(2)]

        for root in self.roots:
            root.add_downlinks(level2switches)

        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def example_32config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(4)]
        servers = [[FireSimServerNode() for y in range(8)] for x in range(4)]

        for root in self.roots:
            root.add_downlinks(level2switches)

        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def example_64config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(8)]
        servers = [[FireSimServerNode() for y in range(8)] for x in range(8)]

        for root in self.roots:
            root.add_downlinks(level2switches)

        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def example_128config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(2)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in range(2)]
        servers = [
            [[FireSimServerNode() for y in range(8)] for x in range(8)]
            for x in range(2)
        ]

        self.roots[0].add_downlinks(level1switches)

        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
```

(continues on next page)

(continued from previous page)

```
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_downlinks(
                    servers[switchgroupno][switchno]
                )

    def example_256config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(4)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in range(4)]
        servers = [
            [[FireSimServerNode() for y in range(8)] for x in range(8)]
            for x in range(4)
        ]

        self.roots[0].add_downlinks(level1switches)

        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])

        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_downlinks(
                    servers[switchgroupno][switchno]
                )

    @staticmethod
    def supernode_flatten(arr: List[Any]) -> List[Any]:
        res: List[Any] = []
        for x in arr:
            res = res + x
        return res

    def supernode_example_6config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        self.roots[0].add_downlinks([FireSimSuperNodeServerNode()])
        self.roots[0].add_downlinks([FireSimDummyServerNode() for x in range(5)])

    def supernode_example_4config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        self.roots[0].add_downlinks([FireSimSuperNodeServerNode()])
        self.roots[0].add_downlinks([FireSimDummyServerNode() for x in range(3)])

    def supernode_example_8config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = UserTopologies.supernode_flatten(
            [
                [
                    FireSimSuperNodeServerNode(),
                    FireSimDummyServerNode(),
                    FireSimDummyServerNode(),
                    FireSimDummyServerNode(),
```

(continues on next page)

(continued from previous page)

```
                ]
                for y in range(2)
            ]
        )
        self.roots[0].add_downlinks(servers)

    def supernode_example_16config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = UserTopologies.supernode_flatten(
            [
                [
                    FireSimSuperNodeServerNode(),
                    FireSimDummyServerNode(),
                    FireSimDummyServerNode(),
                    FireSimDummyServerNode(),
                ]
                for y in range(4)
            ]
        )
        self.roots[0].add_downlinks(servers)

    def supernode_example_32config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        servers = UserTopologies.supernode_flatten(
            [
                [
                    FireSimSuperNodeServerNode(),
                    FireSimDummyServerNode(),
                    FireSimDummyServerNode(),
                    FireSimDummyServerNode(),
                ]
                for y in range(8)
            ]
        )
        self.roots[0].add_downlinks(servers)

    def supernode_example_64config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(2)]
        servers = [
            UserTopologies.supernode_flatten(
                [
                    [
                        FireSimSuperNodeServerNode(),
                        FireSimDummyServerNode(),
                        FireSimDummyServerNode(),
                        FireSimDummyServerNode(),
                    ]
                    for y in range(8)
                ]
            )
            for x in range(2)
```

(continues on next page)

(continued from previous page)

```
        ]
        for root in self.roots:
            root.add_downlinks(level2switches)
        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def supernode_example_128config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(4)]
        servers = [
            UserTopologies.supernode_flatten(
                [
                    [
                        FireSimSuperNodeServerNode(),
                        FireSimDummyServerNode(),
                        FireSimDummyServerNode(),
                        FireSimDummyServerNode(),
                    ]
                    for y in range(8)
                ]
            )
            for x in range(4)
        ]
        for root in self.roots:
            root.add_downlinks(level2switches)
        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def supernode_example_256config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level2switches = [FireSimSwitchNode() for x in range(8)]
        servers = [
            UserTopologies.supernode_flatten(
                [
                    [
                        FireSimSuperNodeServerNode(),
                        FireSimDummyServerNode(),
                        FireSimDummyServerNode(),
                        FireSimDummyServerNode(),
                    ]
                    for y in range(8)
                ]
            )
            for x in range(8)
        ]
        for root in self.roots:
            root.add_downlinks(level2switches)
        for l2switchNo in range(len(level2switches)):
            level2switches[l2switchNo].add_downlinks(servers[l2switchNo])

    def supernode_example_512config(self) -> None:
        self.roots = [FireSimSwitchNode()]
```

(continues on next page)

(continued from previous page)

```
        level1switches = [FireSimSwitchNode() for x in range(2)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in range(2)]
        servers = [
            [
                UserTopologies.supernode_flatten(
                    [
                        [
                            FireSimSuperNodeServerNode(),
                            FireSimDummyServerNode(),
                            FireSimDummyServerNode(),
                            FireSimDummyServerNode(),
                        ]
                        for y in range(8)
                    ]
                )
                for x in range(8)
            ]
            for x in range(2)
        ]
        self.roots[0].add_downlinks(level1switches)
        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_downlinks(
                    servers[switchgroupno][switchno]
                )

    def supernode_example_1024config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(4)]
        level2switches = [[FireSimSwitchNode() for x in range(8)] for x in range(4)]
        servers = [
            [
                UserTopologies.supernode_flatten(
                    [
                        [
                            FireSimSuperNodeServerNode(),
                            FireSimDummyServerNode(),
                            FireSimDummyServerNode(),
                            FireSimDummyServerNode(),
                        ]
                        for y in range(8)
                    ]
                )
                for x in range(8)
            ]
            for x in range(4)
        ]
        self.roots[0].add_downlinks(level1switches)
        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
```

(continues on next page)

(continued from previous page)

```
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_downlinks(
                    servers[switchgroupno][switchno]
                )

    def supernode_example_deep64config(self) -> None:
        self.roots = [FireSimSwitchNode()]
        level1switches = [FireSimSwitchNode() for x in range(2)]
        level2switches = [[FireSimSwitchNode() for x in range(1)] for x in range(2)]
        servers = [
            [
                UserTopologies.supernode_flatten(
                    [
                        [
                            FireSimSuperNodeServerNode(),
                            FireSimDummyServerNode(),
                            FireSimDummyServerNode(),
                            FireSimDummyServerNode(),
                        ]
                        for y in range(8)
                    ]
                )
                for x in range(1)
            ]
            for x in range(2)
        ]
        self.roots[0].add_downlinks(level1switches)
        for switchno in range(len(level1switches)):
            level1switches[switchno].add_downlinks(level2switches[switchno])
        for switchgroupno in range(len(level2switches)):
            for switchno in range(len(level2switches[switchgroupno])):
                level2switches[switchgroupno][switchno].add_downlinks(
                    servers[switchgroupno][switchno]
                )

    def dual_example_8config(self) -> None:
        """two separate 8-node clusters for experiments, e.g. memcached mutilate."""
        self.roots = [FireSimSwitchNode()] * 2
        servers = [FireSimServerNode() for y in range(8)]
        servers2 = [FireSimServerNode() for y in range(8)]
        self.roots[0].add_downlinks(servers)
        self.roots[1].add_downlinks(servers2)

    def triple_example_8config(self) -> None:
        """three separate 8-node clusters for experiments, e.g. memcached mutilate."""
        self.roots = [FireSimSwitchNode()] * 3
        servers = [FireSimServerNode() for y in range(8)]
        servers2 = [FireSimServerNode() for y in range(8)]
        servers3 = [FireSimServerNode() for y in range(8)]
        self.roots[0].add_downlinks(servers)
        self.roots[1].add_downlinks(servers2)
```

(continues on next page)

(continued from previous page)

```
        self.roots[2].add_downlinks(servers3)

    def no_net_config(self) -> None:
        self.roots = [FireSimServerNode() for x in range(self.no_net_num_nodes)]

    # Spins up all of the precompiled, unnetworked targets
    def all_no_net_targets_config(self) -> None:
        hwdb_entries = [
            "firesim_boom_singlecore_no_nic_l2_llc4mb_ddr3",
            "firesim_rocket_quadcore_no_nic_l2_llc4mb_ddr3",
        ]
        assert len(hwdb_entries) == self.no_net_num_nodes
        self.roots = [
            FireSimServerNode(hwdb_entries[x]) for x in range(self.no_net_num_nodes)
        ]

    ######################################################################

    # DOC include start: user_topology.py fireaxe_topology_config
    def fireaxe_topology_config(
        self,
        hwdb_entries: Dict[int, str],
        edges: List[FireAxeEdge],
        slotid_to_pidx: List[int],
        mode: PartitionMode,
    ) -> None:
        pidx_to_slotid = dict()
        for sid, pid in enumerate(slotid_to_pidx):
            pidx_to_slotid[pid] = sid

        # Create partition nodes
        pidx_to_partition_node: Dict[int, PartitionNode] = dict()
        for pidx, hwdb in hwdb_entries.items():
            node = PartitionNode(hwdb, pidx)
            pidx_to_partition_node[pidx] = node

        # Add edges to nodes
        for edge in edges:
            u_node = pidx_to_partition_node[edge.u.pidx]
            v_node = pidx_to_partition_node[edge.v.pidx]
            u_node.add_edge(edge.u.bidx, edge.v.bidx, v_node)
            v_node.add_edge(edge.v.bidx, edge.u.bidx, u_node)

        # Create PartitionConfigs and FireSimServerNode
        servers: Dict[int, FireSimServerNode] = dict()
        for pidx, node in pidx_to_partition_node.items():
            partition_cfg = PartitionConfig(node, pidx_to_slotid, mode)
            servers[pidx_to_slotid[node.pidx]] = FireSimServerNode(
                partition_cfg.get_hwdb(), partition_config=partition_cfg
            )

        # Sort the servers by their sim slot id
```

(continues on next page)

(continued from previous page)

```
        servers = dict(sorted(servers.items()))
        self.roots = list(servers.values())

    # DOC include end: user_topology.py fireaxe_topology_config

    # DOC include start: user_topology.py fireaxe_rocket_fastmode_config
    def fireaxe_rocket_fastmode_config(self) -> None:
        # DOC include start: fireaxe_fastmode_config hwdb_entries
        # hwdb_entries maps the partition index to the hwdb name
        hwdb_entries = {0: "f1_rocket_split_soc_fast", 1: "f1_rocket_split_tile_fast"}
        # DOC include end: fireaxe_fastmode_config hwdb_entries
        # DOC include start: fireaxe_fastmode_config slot_to_pidx
        # slotid_to_pidx maps the partition index to the FPGA slotid
        # For instance, `slotid_to_pidx = [2, 1, 0]` will map partition
        # index 2 to simulation slot 0, partition index 1 to simulation slot 1,
        # and partition index 0 to simulation slot 2.
        slotid_to_pidx = [0, 1]
        # DOC include end: fireaxe_fastmode_config slot_to_pidx
        # DOC include start: fireaxe_fastmode_config edges
        # The `FireAxeEdge` class is used to depict the connections between the␣
↪partitions.
        edges = [FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(1, 0))]
        # DOC include end: fireaxe_fastmode_config edges
        # DOC include start: fireaxe_fastmode_config mode
        # The PartitionMode enum contains the different partition modes that are␣
↪available
        mode = PartitionMode.FAST_MODE
        # DOC include end: fireaxe_fastmode_config mode
        # DOC include start: fireaxe_fastmode_config summing it all up
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)
        # DOC include end: fireaxe_fastmode_config summing it all up

    # DOC include end: user_topology.py fireaxe_rocket_fastmode_config

    # DOC include start: user_topology.py fireaxe_rocket_exactmode_config
    def fireaxe_rocket_exactmode_config(self) -> None:
        hwdb_entries = {
            0: "f1_firesim_rocket_tile_exact",
            1: "f1_firesim_rocket_soc_exact",
        }
        slotid_to_pidx = [0, 1]
        edges = [
            # DOC include start: fireaxe_rocket_exactmode_config edge 0
            FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(1, 0)),
            # DOC include end: fireaxe_rocket_exactmode_config edge 0
            # DOC include start: fireaxe_rocket_exactmode_config edge 1
            FireAxeEdge(FireAxeNodeBridgePair(0, 1), FireAxeNodeBridgePair(1, 1)),
            # DOC include end: fireaxe_rocket_exactmode_config edge 1
        ]
        # DOC include start: fireaxe_rocket_exactmode_config mode
        mode = PartitionMode.EXACT_MODE
        # DOC include end: fireaxe_rocket_exactmode_config mode
```

(continues on next page)

(continued from previous page)

```
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)

    # DOC include end: user_topology.py fireaxe_rocket_exactmode_config

    # DOC include start: user_topology.py fireaxe_ring_noc_config
    def fireaxe_ring_noc_config(self) -> None:
        hwdb_entries = {
            0: "xilinx_u250_quad_rocket_ring_0",
            1: "xilinx_u250_quad_rocket_ring_1",
            2: "xilinx_u250_quad_rocket_ring_base",
        }
        slotid_to_pidx = [0, 1, 2]
        # DOC include start: fireaxe_ring_noc_config edges
        edges = [
            FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(2, 1)),
            FireAxeEdge(FireAxeNodeBridgePair(2, 0), FireAxeNodeBridgePair(1, 1)),
            FireAxeEdge(FireAxeNodeBridgePair(1, 0), FireAxeNodeBridgePair(0, 1)),
        ]
        # DOC include end: fireaxe_ring_noc_config edges
        mode = PartitionMode.NOC_MODE
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)

    # DOC include end: user_topology.py fireaxe_ring_noc_config


#    ######Used only for tutorial purposes###################
#    def example_sha3hetero_2config(self):
#        self.roots= [FireSimSwitchNode()]
#        servers = [FireSimServerNode(server_hardware_config=
#                     "firesim_boom_singlecore_nic_l2_llc4mb_ddr3"),
#                   FireSimServerNode(server_hardware_config=
#                     "firesim_rocket_singlecore_sha3_nic_l2_llc4mb_ddr3")]
#        self.roots[0].add_downlinks(servers)
```

12.8 AGFI Metadata/Tagging

In the AWS EC2 case, when you build an AGFI in FireSim, the AGFI description stored by AWS is populated with metadata that helps the manager decide how to deploy a simulation. The important metadata is listed below, along with how each field is set and used:

- `firesim-buildquintuplet`: This always reflects the quintuplet combination used to BUILD the AGFI.
- `firesim-deployquintuplet`: This reflects the quintuplet combination that is used to DEPLOY the AGFI. By default, this is the same as `firesim-buildquintuplet`. In certain cases however, your users may not have access to a particular configuration, but a simpler configuration may be sufficient for building a compatible software driver (e.g. if you have proprietary RTL in your FPGA image that doesn't interface with the outside system). In this case, you can specify a custom deployquintuplet at build time. If you do not do so, the manager will automatically set this to be the same as `firesim-buildquintuplet`.
- `firesim-commit`: This is the commit hash of the version of FireSim used to build this AGFI. If the AGFI was created from a dirty copy of the FireSim repo, "-dirty" will be appended to the commit hash.

CHAPTER

THIRTEEN

WORKLOADS

This section describes workload definitions in FireSim.

13.1 FireMarshal

Workload generation in FireSim is handled by a tool called **FireMarshal** in Chipyard (located in ${CY_DIR}/software):

Workloads in FireMarshal consist of a series of **Jobs** that are assigned to logical nodes in the target system. If no jobs are specified, then the workload is considered `uniform` and only a single image will be produced for all nodes in the system. Workloads are described by a JSON file and a corresponding workload directory and can inherit their definitions from existing workloads. Typically, workload configurations are kept in `<firemarshal-dir>/workloads/` although you can use any directory you like. We provide a few basic workloads to start with including buildroot or Fedora-based linux distributions and bare-metal.

Once you define a workload, the `marshal` command will produce a corresponding boot-binary and rootfs for each job in the workload. This binary and rootfs can then be launched on qemu or spike (for functional simulation), or installed to firesim for running on real RTL.

For more information, see the official FireMarshal documentation, and its quickstart tutorial.

13.2 [DEPRECATED] Defining Custom Workloads

Danger

This version of the Defining Custom Workloads page is kept here to document some of the legacy workload configurations still present in `deploy/workloads/`. New workloads should NOT be generated using these instructions.

This page documents the `JSON` input format that FireSim uses to understand your software workloads that run on the target design. Most of the time, you should not be writing these files from scratch. Instead, use *FireMarshal* to build a workload (including Linux kernel images and root filesystems) and use `firemarshal`'s `install` command to generate an initial `.json` file for FireSim. Once you generate a base `.json` with FireMarshal, you can add some of the options noted on this page to control additional files used as inputs/outputs to/from simulations.

Workloads in FireSim consist of a series of **Jobs** that are assigned to be run on individual simulations. Currently, we require that a Workload defines either:

- A single type of job, that is run on as many simulations as specfied by the user. These workloads are usually suffixed with `-uniform`, which indicates that all nodes in the workload run the same job. An example of such a workload is deploy/workloads/br-base-uniform.json.

- Several different jobs, in which case there must be exactly as many jobs as there are running simulated nodes. An example of such a workload is deploy/workloads/br-base-non-uniform.json.

FireSim can take these workload definitions and deploy them using the manager.

In the following subsections, we will go through the two aforementioned example workload configurations, describing how these two functions use each part of the JSON file inline.

ERRATA: You will notice in the following JSON files the field "workloads" this should really be named "jobs" – we will fix this in a future release.

13.2.1 Uniform Workload JSON

deploy/workloads/br-base-uniform.json is an example of a "uniform" style workload, where each simulated node runs the same software configuration.

Let's take a look at this file:

```
{
  "benchmark_name": "br-base-uniform",
  "common_bootbinary": "../../../target-design/chipyard/software/firemarshal/images/
↪firechip/br-base/br-base-bin",
  "common_rootfs": "../../../target-design/chipyard/software/firemarshal/images/firechip/
↪br-base/br-base.img"
  "common_outputs": [
    "/etc/os-release"
  ],
  "common_simulation_outputs": [
    "uartlog",
    "memory_stats*.csv"
  ],
}
```

There is also a corresponding directory named after this workload/file: `deploy/workloads/br-base-uniform`. We will elaborate on this later.

Looking at the JSON file, you'll notice that this is a relatively simple workload definition.

In this "uniform" case, the manager will name simulations after the `benchmark_name` field, appending a number for each simulation using the workload (e.g. `br-base-uniform0`, `br-base-uniform1`, and so on). It is standard pratice to keep `benchmark_name`, the JSON filename, and the above directory name the same. In this case, we have set all of them to `br-base-uniform`.

Next, the `common_bootbinary` field represents the binary that the simulations in this workload are expected to boot from. The manager will copy this binary for each of the nodes in the simulation (each gets its own copy). The `common_bootbinary` path is relative to the workload's directory, in this case deploy/workloads/br-base-uniform.

Similarly, the `common_rootfs` field represents the disk image that the simulations in this workload are expected to boot from. The manager will copy this root filesystem image for each of the nodes in the simulation (each gets its own copy). The `common_rootfs` path is relative to the workload's directory, in this case deploy/workloads/br-base-uniform.

The `common_outputs` field is a list of outputs that the manager will copy out of the root filesystem image AFTER a simulation completes. You can add multiple paths here.

The `common_simulation_outputs` field is a list of outputs that the manager will copy off of the simulation host machine AFTER a simulation completes. In this example, when a workload running on a simulated cluster with `firesim runworkload` completes, the `uartlog` (an automatically generated file that contains the full console output of the simulated system) and `memory_stats.csv` files will be copied out of the simulation's base directory on the host

instance and placed in the job's output directory within the workload's output directory (see the *firesim runworkload* section). You can add multiple paths here.

ERRATA: "Uniform" style workloads currently do not support being automatically built – you can currently hack around this by building the rootfs as a single-node non-uniform workload, then deleting the `workloads` field of the JSON to make the manager treat it as a uniform workload. This will be fixed in a future release.

13.2.2 Non-uniform Workload JSON (explicit job per simulated node)

Now, we'll look at the `br-base-non-uniform` workload, which explicitly defines a job per simulated node.

```
{
  "benchmark_name": "br-base-non-uniform",
  "common_bootbinary": "../../../target-design/chipyard/software/firemarshal/images/
↪firechip/br-base/br-base-bin",
  "common_rootfs": "../../../target-design/chipyard/software/firemarshal/images/firechip/
↪br-base/br-base.img"
  "common_outputs": [
    "/etc/os-release"
  ],
  "common_simulation_outputs": [
    "uartlog",
    "memory_stats*.csv"
  ],
  "deliver_dir": "/",
  "common_args": [],
  "common_files": [
    "bin/echome.sh"
  ],
  "no_post_run_hook": "",
  "workloads": [
    {
      "name": "job0",
      "files": [],
      "command": "echome.sh && poweroff -f",
      "simulation_outputs": [],
      "outputs": []
    },
    {
      "name": "job1",
      "files": [],
      "command": "poweroff -f",
      "simulation_outputs": [],
      "outputs": []
    }
  ]
}
```

Additionally, let's take a look at the state of the required `br-base-non-uniform` directory:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/workloads/br-base-
↪non-uniform$ ls -la
...
drwxrwxr-x  3 centos centos         16 May 17 21:58 overlay
```

Let's look at the `overlay` subdirectory:

```
centos@ip-172-30-2-111.us-west-2.compute.internal:~/firesim-new/deploy/workloads/br-base-
↪non-uniform/overlay$ ls -la */*
-rwxrwxr-x 1 centos centos 249 May 17 21:58 bin/echome.sh
```

This is a file that's actually committed to the repo, that in theory would run the benchmark we want to run on one of our simulated systems. In this case, it's a simple echo.

Now, let's take a look at how we got here. First, let's review some of the new fields present in this JSON file:

- `common_files`: This is an array of files that will be included in ALL of the job rootfses when they're built. This is relative to a path that we'll pass to the script that generates rootfses.
- `workloads`: This time, you'll notice that we have this array, which is populated by objects that represent individual jobs. Each job has some additional fields:
  - `name`: In this case, jobs are each assigned a name manually. These names MUST BE UNIQUE within a particular workload.
  - `files`: Just like `common_files`, but specific to this job.
  - `command`: This is the command that will be run automatically immediately when the simulation running this job boots up. This is usually the command that starts the workload we want.
  - `simulation_outputs`: Just like `common_simulation_outputs`, but specific to this job.
  - `outputs`: Just like `common_outputs`, but specific to this job.

In this example, we specify one node that boots up and runs `echome.sh && poweroff -f` while the other just runs `poweroff -f`.

You can run works like this with the manager by setting `workload_name: br-base-non-uniform.json` in `config_runtime.yaml`. The manager will automatically look for the generated rootfses (based on workload and job names that it reads from the json) and distribute work appropriately.

Just like in the uniform case, it will copy back the results that we specify in the JSON file. We'll end up with a directory in `firesim/deploy/results-workload/` named after the workload name, with a subdirectory named after each job in the workload, which will contain the output files we want.

CHAPTER

FOURTEEN

TARGETS

FireSim generates SoC models by transforming RTL emitted by a Chisel generator, such as the Rocket SoC generator. Subject to conditions outlined in *Restrictions on Target RTL*, if it can be generated by Chisel, it can be simulated in FireSim.

14.1 Restrictions on Target RTL

Current limitations in Golden Gate place the following restrictions on the (FIR)RTL that can be transformed and thus used in FireSim:

1. The top-level module must have no inputs or outputs. Input stimulus and output capture must be implemented using target RTL or target-to-host Bridges.
2. All target clocks must be generated by a single `RationalClockBridge`.
3. Black boxes must be "clock-gateable" by replacing its input clock with a gated equivalent which will be used to stall simulation time in that module.
    a. As a consequence, target clock-gating cannot be implemented using black-box primitives, and must instead be modeled by adding clock-enables to all state elements of the gated clock domain (i.e., by adding an enable or feedback mux on registers to conditionally block updates, and by gating write-enables on memories).
4. Asynchronous reset must only be implemented using Rocket Chip's black-box async reset. These are replaced with synchronously reset registers using a FIRRTL transformation.

14.1.1 Including Verilog IP

FireSim now supports target designs that incorporate Verilog IP using the standard `BlackBox` interface from Chisel. For an example of how to add Verilog IP to a target system based on Rocket Chip, see the Incorporating Verilog Blocks section of the Chipyard documentation.

1. For the transform to work, the Chisel Blackbox that wraps the Verilog IP must have input clocks that can safely be clock-gated.
2. The compiler that produces the decoupled simulator ("FAME Transform") automatically recognizes such black-boxes inside the target design.
3. The compiler automatically gates each clock of the Verilog IP to ensure that it deterministically advances in lockstep with the rest of the simulator.
4. This allows any Verilog module, subject to the constraint above, to be instantiated anywhere in the target design using the standard Chisel Blackbox interface.

14.1.2 Multiple Clock Domains

FireSim can support simulating targets that have multiple clock domains. As stated above, all clocks must be generated using a single `RationalClockBridge`. For most users the default FireSim test harness in Chipyard will suffice, if you need to define a custom test harness instantiate the `RationalClockBridge` like so:

```
// Here we request three target clocks (the base clock is implicit). All
// clocks beyond the base clock are specified using the RationalClock case
// class which gives the clock domain's name, and its clock multiplier and
// divisor relative to the base clock.
val clockBridge = RationalClockBridge(RationalClock("HalfRate", 1, 2), RationalClock(
↪"ThirdRate", 1, 3))

// The clock bridge has a single output: a Vec[Clock] of the requested clocks
// in the order they were specified, which we are now free to use through our
// Chisel design.  While not necessary, here we unassign the Vec to give them
// more informative references in our Chisel.
val Seq(fullRate, halfRate, thirdRate) = clockBridge.io.clocks.toSeq
```

Further documentation can be found in the source (`sim/midas/src/main/scala/midas/widgets/ClockBridge.scala`).

The Base Clock

By convention, target time is specified in cycles of the *base clock*, which is defined to be the clock of the `RationalClockBridge` whose clock ratio (multiplier / divisor) is one. While we suggest making the base clock the fastest clock in your system, as in any microprocessor-based system it will likely correspond to your core clock frequency, this is not a constraint.

Limitations:

- The number of target clocks FireSim can simulate is bounded by the number of BUFGCE resources available on the host FPGA, as these are used to independently clock-gate each target clock.
- As its name suggests, the `RationalClockBridge` can only generate target clocks that are rationally related. Specifically, all requested frequencies must be expressable in the form:

  $$f_i = \frac{f_{lcm}}{k_i}$$

  Where,

  - f_i is the desired frequency of the ith clock
  - f_{lcm}, is the least-common multiple of all requested frequencies
  - k_i is a 16-bit unsigned integer

  An arbitrary frequency can be modeled using a sufficiently precise rational multiple. Golden Gate will raise a compile-time error if it cannot support a desired frequency.
- Each bridge module must reside entirely within a single clock domain. The Bridge's target interface must contain a single input clock, and all inputs and outputs of the bridge module must be latched and launched, respectively, by registers in the same clock domain.

14.2 Target-Side FPGA Constraints

FireSim provides utilities to generate Xilinx Design Constraints (XDC) from string snippets in target's Chisel. Golden Gate collects these annotations and emits separate xdc files for synthesis and implementation. See *FPGA Build Files* for a complete listing of output files used in FPGA compilation.

14.2.1 RAM Inference Hints

Vivado generally does a reasonable job inferring embedded memories from FireSim-generated RTL, though there are some cases in which it must be coaxed. For example:

- Due to insufficient BRAM resources, you may wish to use URAM for a memory that'd infer as BRAM.
- If Vivado can't find pipeline registers to absorb into a URAM or none exist in the target, you may get an warning like:

```
[Synth 8-6057] Memory: "<memory>" defined in module: "<module>" implemented as␣
↪Ultra-Ram
has no pipeline registers. It is recommended to use pipeline registers to achieve␣
↪high
performance.
```

Since Golden Gate modifies the module hierarchy extensively, it's highly desirable to annotate these memories in the Chisel source so that their hints may move with the memory instance. This is a more robust alternative to relying on wildcard / glob matches from a static XDC specification.

Chisel memories can be annotated *in situ* like so:

```
import midas.targetutils.xdc._
val mem = SyncReadMem(1 << addrBits, UInt(dataBits.W))
RAMStyleHint(mem, RAMStyles.ULTRA)
// Alternatively: RAMStyleHint(mem, RAMStyles.BLOCK)
```

Alternatively, you can "dot-in" (traverse public members of a Scala class hierarchy) to annotate a memory in a sub-module. Here's an example:

```
val modA = Module(new SyncReadMemModule(None))
val modB = Module(new SyncReadMemModule(None))
RAMStyleHint(modA.mem, RAMStyles.ULTRA)
RAMStyleHint(modB.mem, RAMStyles.BLOCK)
```

These annotations can be deployed anywhere: in the target, in bridge modules, and in internal FireSim RTL. The resulting constraints should appear the synthesis xdc file emitted by Golden Gate. For more information see the ScalaDoc for `RAMStyleHint` or read the source located at: sim/midas/targetutils/src/main/scala/midas/xdc/RAMStyleHint.scala.

14.3 Provided Target Designs

14.3.1 Target Generator Organization

FireSim provides multiple *projects*, each for a different type of target. Each project has its own chisel generator that invokes Golden Gate, its own driver sources, and a makefrag that plugs into the Make-based build system that resides in `sim/`. These projects are:

1. **firesim** (Default): rocket chip-based targets. These include targets with either BOOM or rocket pipelines, and should be your starting point if you're building an SoC with the Rocket Chip generator.
2. **midasexamples**: the Golden Gate example designs. Located at sim/src/main/scala/midasexamples, these are a set of simple chisel circuits like GCD, that demonstrate how to use Golden Gate. These are useful test cases for bringing up new Golden Gate features.
3. **bridges**: tests for firesim-lib bridges. These have more dependencies and involve more logic than *midasexamples*.
4. **fasedtests**: designs to do integration testing of FASED memory-system timing models.

Projects have the following directory structure:

```
sim/
├-Makefile # Top-level makefile for projects where FireSim is the top-level repo
├-Makefrag # Target-agnostic makefrag, with recipes to generate drivers and RTL␣
↪simulators
├-src/main/scala/{target-project}/
│                 └─Makefrag # Defines target-specific make variables and recipes.
├-src/main/cc/{target-project}/
│              ├─{driver-csrcs}.cc # The target's simulation driver, and sofware-model␣
↪sources
│              └─{driver-headers}.h
└-src/main/makefrag/{target-project}/
                     ├─Generator.scala # Contains the main class that generates target␣
↪RTL and calls Golden Gate
                     └─{other-scala-sources}.scala
```

14.3.2 Specifying A Target Instance

To generate a specific instance of a target, the build system leverages five Make variables:

1. `TARGET_PROJECT`: this points the Makefile (*sim/Makefile*) at the right target-specific Makefrag, which defines the generation and metasimulation software recipes. The makefrag for the default target project is defined at `sim/src/main/makefrag/firesim`.
2. `DESIGN`: the name of the top-level Chisel module to generate (a Scala class name). These are defined in FireChip Chipyard generator.
3. `TARGET_CONFIG`: specifies a `Config` instance that is consumed by the target design's generator. For the default firesim target project, predefined configs are described in in the FireChip Chipyard generator.
4. `PLATFORM_CONFIG`: specifies a `Config` instance that is consumed by Golden Gate and specifies compiler-level and host-land parameters, such as whether to enable assertion synthesis, or multi-ported RAM optimizations. Common platform configs are described in `firesim-lib/sim/src/main/scala/configs/CompilerConfigs.scala`).
5. `PLATFORM`: this points the Makefile (*sim/Makefile*) at the right FPGA platform to build for. This must correspond to a platform defined at platforms.

TARGET_CONFIG and PLATFORM_CONFIG are strings that are used to construct a Config instance (derives from RocketChip's parameterization system, Config, see the CDE repo). These strings are of the form "{. . . _}{<Class Name>_}<Class Name>". Only the final, base class name is compulsory: class names that are prepended with "_" are used to create a compound Config instance.

```
// Specify by setting TARGET_CONFIG=Base
class Base extends Config((site, here, up) => {...})
class Override1 extends Config((site, here, up) => {...})
class Override2 extends Config((site, here, up) => {...})
// Specify by setting TARGET_CONFIG=Compound
class Compound extends Config(new Override2 ++ new Override1 ++ new Base)
// OR by setting TARGET_CONFIG=Override2_Override1_Base
// Can specify undefined classes this way. ex: TARGET_CONFIG=Override2_Base
```

With this scheme, you don't need to define a Config class for every instance you wish to generate, making it very useful for sweeping over a parameterization space.

Note that the precedence of Configs decreases from left to right in a string. Appending a config to an existing one will only have an effect if it sets a field not already set in higher precendence Configs. For example, "BaseF1Config_SetFieldAtoX" is equivalent to "BaseF1Config_SetFieldAtoX_SetFieldAtoY".

How a particular Config resolves it's Field s can be unintuitive for complex compound Config s. One precise way to check a config is doing what you expect is to open the scala REPL, instantiate an instance of the desired Config, and inspect its fields.

```
$ make sbt # Launch into SBT's shell with extra FireSim arguments

sbt:firechip> console # Launch the REPL

scala> val inst = (new firesim.firesim.FireSimRocketChipConfig).toInstance # Make an␣
↪instance

inst: freechips.rocketchip.config.Config = FireSimRocketChipConfig

scala> import freechips.rocketchip.subsystem._ # Get some important Fields

import freechips.rocketchip.subsystem.RocketTilesKey

scala> inst(RocketTilesKey).size # Query number of cores

res2: Int = 1

scala> inst(RocketTilesKey).head.dcache.get.nWays # Query L1 D$ associativity

res3: Int = 4
```

14.4 Rocket Chip Generator-based SoCs (firesim project)

Using the Make variables listed above, we give examples of generating different targets using the default Rocket Chip-based target project.

14.4.1 Rocket-based SoCs

Three design classes use Rocket scalar in-order pipelines.

Single core, Rocket pipeline (default)

```
make TARGET_CONFIG=FireSimRocketConfig
```

Single-core, Rocket pipeline, with network interface

```
make TARGET_CONFIG=WithNIC_FireSimRocketChipConfig
```

Quad-core, Rocket pipeline

```
make TARGET_CONFIG=FireSimQuadRocketConfig
```

14.4.2 BOOM-based SoCs

The BOOM (Berkeley Out-of-Order Machine) superscalar out-of-order pipelines can also be used with the same design classes that the Rocket pipelines use. Only the TARGET_CONFIG needs to be changed, as shown below:

Single-core BOOM

```
make TARGET_CONFIG=FireSimLargeBoomConfig
```

Single-core BOOM, with network interface

```
make TARGET_CONFIG=WithNIC_FireSimBoomConfig
```

14.4.3 Generating A Different FASED Memory-Timing Model Instance

Golden Gate's memory-timing model generator, FASED, can elaborate a space of different DRAM model instances: we give some typical ones here. These targets use the Makefile-defined defaults of `DESIGN=FireSim PLATFORM_CONFIG=BaseF1Config`.

Quad-rank DDR3 first-ready, first-come first-served memory access scheduler

```
make TARGET_CONFIG=DDR3FRFCFS_FireSimRocketConfig
```

As above, but with a 4 MiB (maximum simulatable capacity) last-level-cache model

```
make TARGET_CONFIG=DDR3FRFCFSLLC4MB_FireSimRocketConfig
```

FASED *timing-model* configurations are passed to the FASED Bridges in your Target's FIRRTL, and so must be prepended to `TARGET_CONFIG`.

14.5 Midas Examples (midasexamples project)

This project can generate a handful of toy target-designs (set with the make variable `DESIGN`). Each of these designs has their own chisel source file and serves to demostrate the features of Golden Gate.

Some notable examples are:

1. `GCD`: the "Hello World!" of hardware.
2. `WireInterconnect`: demonstrates how combinational paths can be modeled with Golden Gate.
3. `PrintfModule`: demonstrates synthesizable printfs
4. `AssertModule`: demonstrates synthesizable assertions

To generate a target, set the make variable `TARGET_PROJECT=midasexamples`. so that the right project makefrag is sourced.

14.5.1 Examples

To generate the GCD midasexample:

```
make DESIGN=GCD TARGET_PROJECT=midasexamples
```

14.6 FASED Tests (fasedtests project)

This project generates target designs capable of driving considerably more bandwidth to an AXI4-memory slave than current FireSim targets. These are used to do integration and stress testing of FASED instances.

14.6.1 Examples

Generate a synthesizable AXI4Fuzzer (based off of Rocket Chip's TL fuzzer), driving a DDR3 FR-FCFS-based FASED instance.

```
make TARGET_PROJECT=fasedtests DESIGN=AXI4Fuzzer TARGET_CONFIG=FRFCFSConfig
```

As above, now configured to drive 10 million transactions through the instance.

```
make TARGET_PROJECT=fasedtests DESIGN=AXI4Fuzzer TARGET_CONFIG=NT10e7_FRFCFSConfig
```

CHAPTER

FIFTEEN

DEBUGGING IN SOFTWARE

This section describes methods of debugging the target design and the simulation in FireSim, *before running on the FPGA*.

15.1 Debugging & Testing with Metasimulation

When discussing RTL simulation in FireSim, we are generally referring to *metasimulation*: simulating the FireSim simulator's RTL, typically using VCS or Verilator. In contrast, we'll refer to simulation of the target's unmodified (by GoldenGate decoupling, host and target transforms) RTL as *target-level* simulation. Target-level simulation in Chipyard is described at length here.

Metasimulation is the most productive way to catch bugs before generating an AGFI, and a means for reproducing bugs seen on the FPGA. By default, metasimulation uses an abstract but fast model of the host: the FPGA's DRAM controllers are modeled with a single-cycle memory system, the PCI-E subsystem is not simulated, instead the driver presents DMA and MMIO traffic directly on the FPGATop interfaces. Since FireSim simulations are robust against timing differences across hosts, target behavior observed in an FPGA-hosted simulation should be exactly reproducible in a metasimulation.

As a final note, metasimulations are generally only slightly slower than target-level simulations. Example performance numbers can be found at *Metasimulation vs. Target simulation performance*.

15.1.1 Supported Host Simulators

Currently, the following host simulators are supported for metasimulation:

- Verilator
  - FOSS, automatically installed during FireSim setup.
  - Referred to throughout the codebase as `verilator`.
- Synopsys VCS
  - License required.
  - Referred to throughout the codebase as `vcs`.

Pull requests to add support for other simulators are welcome.

15.1.2 Running Metasimulations using the FireSim Manager

The FireSim manager supports running metasimulations using the standard `firesim {launchrunfarm, infrasetup, runworkload, terminaterunfarm}` flow that is also used for FPGA-accelerated simulations. Rather than using FPGAs, these metasimulations run within one of the aforementioned software simulators (*Supported Host Simulators*) on standard compute hosts (i.e. those without FPGAs). This allows users to write a single definition of a target (configured design and software workload), while seamlessly moving between software-only metasimulations and FPGA-accelerated simulations.

As an example, if you have the default `config_runtime.yaml` that is setup for FPGA-accelerated simulations (e.g. the one used for the 8-node networked simulation from the :ref:`cluster-sim` section), a few modifications to the configuration files can convert it to running a distributed metasimulation.

First, modify the existing `metasimulation` mapping in `config_runtime.yaml` to the following:

```
metasimulation:
    metasimulation_enabled: true
    # vcs or verilator. use vcs-debug or verilator-debug for waveform generation
    metasimulation_host_simulator: verilator
    # plusargs passed to the simulator for all metasimulations
    metasimulation_only_plusargs: "+fesvr-step-size=128 +max-cycles=100000000"
    # plusargs passed to the simulator ONLY FOR vcs metasimulations
    metasimulation_only_vcs_plusargs: "+vcs+initreg+0 +vcs+initmem+0"
```

This configures the manager to run Verilator-hosted metasimulations (without waveform generation) for the target specified in `config_runtime.yaml`. When in metasimulation mode, the `default_hw_config` that you specify in `target_config` references an entry in `config_build_recipes.yaml` instead of an entry in `config_hwdb.yaml`.

As is the case when the manager runs FPGA-accelerated simulations, the number of metasimulations that are run is determined by the parameters in the `target_config` section, e.g. `topology` and `no_net_num_nodes`. Many parallel metasimulations can then be run by writing a FireMarshal workload with a corresponding number of jobs.

In metasimulation mode, the run farm configuration must be able to support the required number of metasimulations (see *run_farm* for details). The `num_metasims` parameter on a run farm host specification defines how many metasimulations are allowed to run on a particular host. This corresponds with the `num_fpgas` parameter used in FPGA-accelerated simulation mode. However `num_metasims` does not correspond as tightly with any physical property of the host; it can be tuned depending on the complexity of your design and the compute/memory resources on a host.

For example, in the case of the AWS EC2 run farm (`aws_ec2.yaml`), we define three instance types (`z1d.{3, 6, 12}xlarge`) by default that loosely correspond with `f1.{2, 4, 16}xlarge` instances, but instead have no FPGAs and run only metasims (of course, the `f1.*` instances could run metasims, but this would be wasteful):

```
run_farm_hosts_to_use:
    - z1d.3xlarge: 0
    - z1d.6xlarge: 0
    - z1d.12xlarge: 1

run_farm_host_specs:
    - z1d.3xlarge:
        num_fpgas: 0
        num_metasims: 1
        use_for_switch_only: false
    - z1d.6xlarge:
        num_fpgas: 0
        num_metasims: 2
```

(continues on next page)

(continued from previous page)

```
        use_for_switch_only: false
    - z1d.12xlarge:
        num_fpgas: 0
        num_metasims: 8
        use_for_switch_only: false
```

In this case, the run farm will use a `z1d.12xlarge` instance to host 8 metasimulations.

To generate waveforms in a metasimulation, change `metasimulation_host_simulator` to a simulator ending in `-debug` (e.g. `verilator-debug`). When running with a simulator with waveform generation, make sure to add `waveform.vpd` to the `common_simulation_outputs` area of your workload JSON file, so that the waveform is copied back to your manager host when the simulation completes.

A last notable point is that unlike the normal FPGA simulation case, there are two output logs in metasimulations. There is the expected `uartlog` file that holds the `stdout` from the metasimulation (as in FPGA-based simulations). However, there will also be a `metasim_stderr.out` file that holds `stderr` coming out of the metasimulation, commonly populated by `printf` calls in the RTL, including those that are not marked for `printf` synthesis. If you want to copy `metasim_stderr.out` to your manager when a simulation completes, you must add it to the `common_simulation_outputs` of the workload JSON.

Other than the changes discussed in this section, manager behavior is identical between FPGA-based simulations and metasimulations. For example, simulation outputs are stored in `deploy/results-workload/` on your manager host, FireMarshal workload definitions are used to supply target software, etc. All standard manager functionality is supported in metasimulations, including running networked simulations and using existing FireSim debugging tools (i.e. AutoCounter, TracerV, etc).

Once the configuration changes discussed thus far in this section are made, the standard `firesim {launchrunfarm, infrasetup, runworkload, terminaterunfarm}` set of commands will run metasimulations.

If you are planning to use FireSim metasimulations as your primary simulation tool while developing a new target design, see the (optional) `firesim builddriver` command, which can build metasimulations through the manager without requiring run farm hosts to be launched or accessible. More about this command is found in the *firesim builddriver* section.

15.1.3 Understanding a Metasimulation Waveform

Module Hierarchy

To build out a simulator, Golden Gate adds multiple layers of module hierarchy to the target design and performs additional hierarchy mutations to implement bridges and resource optimizations. Metasimulation uses the `FPGATop` module as the top-level module, which excludes the platform shim layer (`F1Shim`, for EC2 F1). The original top-level of the input design is nested three levels below FPGATop:

Note that many other bridges (under `FPGATop`), channel implementations (under `SimWrapper`), and optimized models (under `FAMETop`) may be present, and vary from target to target. Under the `FAMETop` module instance you will find the original top-level module (`FireSimPDES_`, in this case), however it has now been host-decoupled using the default LI-BDN FAME transformation and is referred to as the *hub model*. It will have ready-valid I/O interfaces for all of the channels bound to it, and internally containing additional channel enqueue and clock firing logic to control the advance of simulated time. Additionally, modules for bridges and optimized models will no longer be found contained in this submodule hierarchy. Instead, I/O for those extracted modules will now be as channel interfaces.

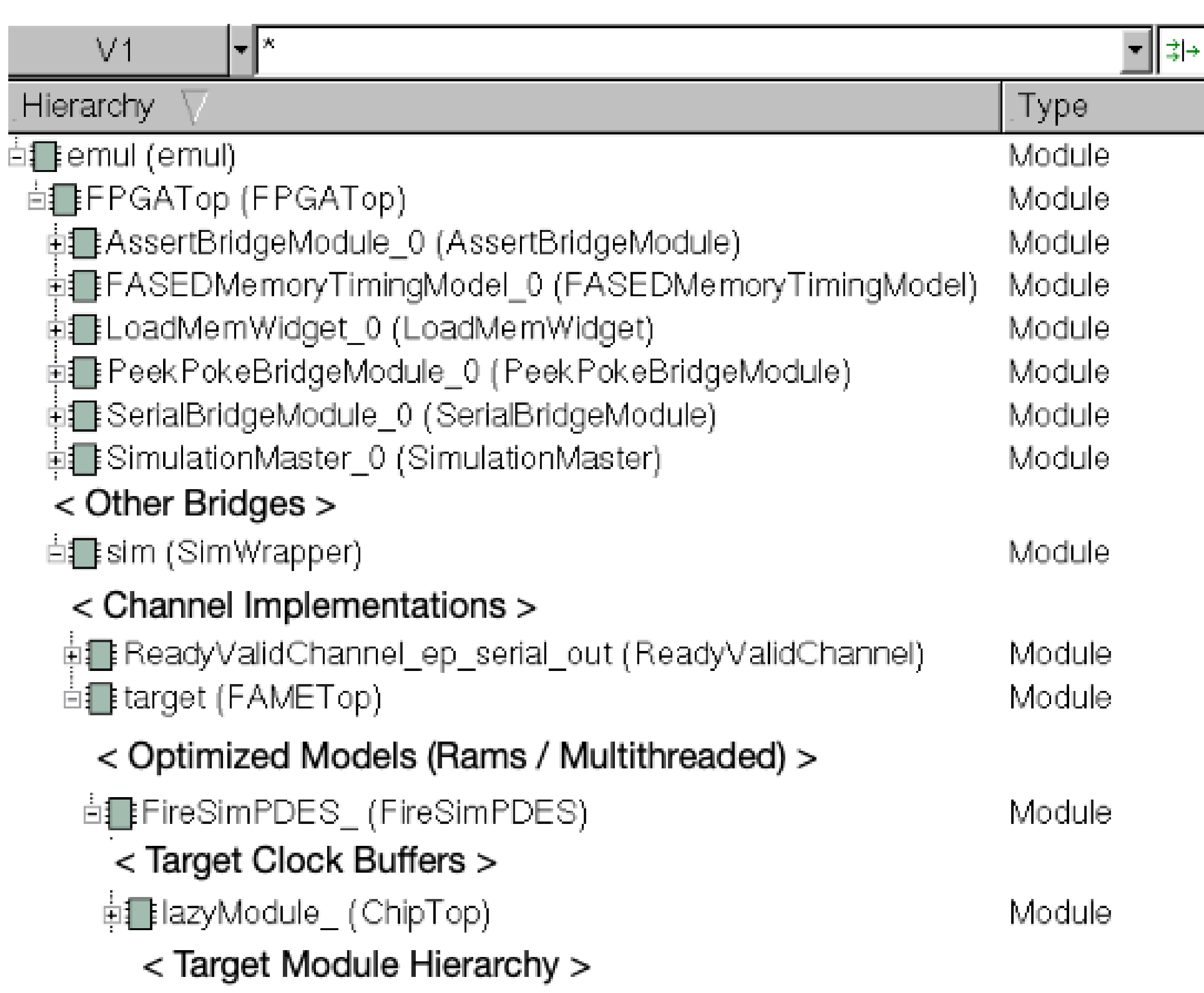


Fig. 1: The module hierarchy visible in a typical metasimulation.

Clock Edges and Event Timing

Since FireSim derives target clocks by clock gating a single host clock, and since bridges and optimized models may introduce stalls of their own, timing of target clock edges in a metasimulation will appear contorted relative to a conventional target-simulation. Specifically, the host-time between clock edges will not be proportional to target-time elapsed over that interval, and will vary in the presence of simulator stalls.

Finding The Source Of Simulation Stalls

In the best case, FireSim simulators will be able to launch new target clock pulses on every host clock cycle. In other words, for single-clock targets the simulation can run at FMR = 1. In the single clock case delays are introduced by bridges (like FASED memory timing models) and optimized models (like a multi-cycle Register File model). You can identify which bridges are responsible for additional delays between target clocks by filtering for `*sink_valid` and `*source_ready` on the hub model. When `<channel>_sink_valid` is deasserted, a bridge or model has not yet produced a token for the current timestep, stalling the hub. When `<channel>_source_ready` is deasserted, a bridge or model is back-pressuring the channel.

15.1.4 Scala Tests

To make it easier to do metasimulation-based regression testing, the ScalaTests wrap calls to Makefiles, and run a limited set of tests on a set of selected designs, including all of the MIDAS examples and a handful of Chipyard-based designs. This is described in greater detail in the *Developer documentation*.

15.1.5 Running Metasimulations through Make

> **Warning**
>
> This section is for advanced developers; most metasimulation users should use the manager-based metasimulation flow described above.

Metasimulations are run out of the `firesim/sim` directory. If you are running a metasim for Chipyard, ensure you properly add the `TARGET_PROJECT_MAKEFRAG` variable to point to Chipyard's makefrag. Generally this is set to `TARGET_PROJECT_MAKEFRAG=${CY_DIR}/generators/firechip/src/main/makefrag/firesim`.

```
[in firesim/sim]
make <verilator|vcs>
```

To compile a simulator with full-visibility waveforms, type:

```
make <verilator|vcs>-debug
```

As part of target-generation, Rocket Chip emits a make fragment with recipes for running suites of assembly tests. MIDAS puts this in `firesim/sim/generated-src/f1/<DESIGN>-<TARGET_CONFIG>-<PLATFORM_CONFIG>/firesim.d`. Make sure your `$RISCV` environment variable is set by sourcing `firesim/sourceme-manager.sh` or `firesim/env.sh`, and type:

```
make run-<asm|bmark>-tests EMUL=<vcs|verilator>
```

To run only a single test, the make target is the full path to the output. Specifically:

```
make EMUL=<vcs|verilator> $PWD/output/f1/<DESIGN>-<TARGET_CONFIG>-<PLATFORM_CONFIG>/
↪<RISCV-TEST-NAME>.<vpd|out>
```

A `.vpd` target will use (and, if required, build) a simulator with waveform dumping enabled, whereas a `.out` target will use the faster waveform-less simulator.

Additionally, you can run a unique binary in the following way:

```
make SIM_BINARY=<PATH_TO_BINARY> run-<vcs|verilator>
make SIM_BINARY=<PATH_TO_BINARY> run-<vcs|verilator>-debug
```

Examples

Run all RISCV-tools assembly and benchmark tests on a Verilated simulator.

```
[in firesim/sim]
make
make -j run-asm-tests
make -j run-bmark-tests
```

Run all RISCV-tools assembly and benchmark tests on a Verilated simulator with waveform dumping.

```
make verilator-debug
make -j run-asm-tests-debug
make -j run-bmark-tests-debug
```

Run `rv64ui-p-simple` (a single assembly test) on a Verilated simulator.

```
make
make $(pwd)/output/f1/FireSim-FireSimRocketConfig-BaseF1Config/rv64ui-p-simple.out
```

Run `rv64ui-p-simple` (a single assembly test) on a VCS simulator with waveform dumping.

```
make vcs-debug
make EMUL=vcs $(pwd)/output/f1/FireSim-FireSimRocketConfig-BaseF1Config/rv64ui-p-simple.
↪vpd
```

15.1.6 Metasimulation vs. Target simulation performance

Generally, metasimulations are only slightly slower than target-level simulations. This is illustrated in the chart below.

Type	Waves	VCS	Verilator -O1	Verilator -02	Verilator
Target	Off	4.8 kHz	3.9 kHz	6.6 kHz	N/A
Target	On	0.8 kHz	3.0 kHz	5.1 kHz	N/A
Meta	Off	3.8 kHz	2.4 kHz	4.5 kHz	5.3 KHz
Meta	On	2.9 kHz	1.5 kHz	2.7 kHz	3.4 KHz

Note that using more aggressive optimization levels when compiling the Verilated-design dramatically lengthens compile time:

Type	Waves	VCS	Verilator -O1	Verilator -02	Verilator
Meta	Off	35s	48s	3m32s	4m35s
Meta	On	35s	49s	5m27s	6m33s

Notes: Default configurations of a single-core, Rocket-based instance running `rv64ui-v-add`. Frequencies are given in target-Hz. Presently, the default compiler flags passed to Verilator and VCS differ from level to level. Hence, these numbers are only intended to give ball park simulation speeds, not provide a scientific comparison between simulators. VCS numbers collected on a local Berkeley machine, Verilator numbers collected on a `c4.4xlarge`. (metasimulation Verilator version: 4.002, target-level Verilator version: 3.904)

CHAPTER

SIXTEEN

DEBUGGING AND PROFILING ON THE FPGA

A common issue with FPGA-prototyping is the difficulty involved in trying to debug and profile systems once they are running on the FPGA. FireSim addresses these issues with a variety of tools for introspecting on designs *once you have a FireSim simulation running on an FPGA*. This section describes these features.

16.1 Capturing RISC-V Instruction Traces with TracerV

FireSim can provide a cycle-by-cycle trace of a target CPU's architectural state over the course of execution, including fields like instruction address, raw instruction bits, privilege level, exception/interrupt status and cause, and a valid signal. This can be useful for profiling or debugging. **TracerV** is the FireSim bridge that provides this functionality. This feature was introduced in our FirePerf paper at ASPLOS 2020 .

This section details how to capture these traces in cycle-by-cycle formats, usually for debugging purposes.

For profiling purposes, FireSim also supports automatically producing stack traces from this data and producing Flame Graphs, which is documented in the *TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation* section.

16.1.1 Building a Design with TracerV

In all FireChip designs, TracerV is included by default. Other targets can enable it by attaching a TracerV Bridge to the RISC-V trace port of each core they wish to trace (there should be one bridge per core). By default, only the cycle number, instruction address, and valid bit are collected.

16.1.2 Enabling Tracing at Runtime

To improve simulation preformance, FireSim does not collect and record data from the TracerV Bridge by default. To enable collection, modify the `enable` flag in the `tracing` section in your `config_runtime.yaml` file to `yes` instead of `no`:

```
tracing:
    enable: yes
```

Now when you run a workload, a trace output file will be placed in the `sim_slot_<slot #>` directory on the F1 instance under the name `TRACEFILE-C0`. The `C0` represents core 0 in the simulated SoC. If you have multiple cores, each will have its own file (ending in `C1`, `C2`, etc). To copy all TracerV trace files back to your manager, you can add `TRACEFILE*` to your `common_simulation_outputs` or `simulation_outputs` in your workload `.json` file. See the *[DEPRECATED] Defining Custom Workloads* section for more information about these options.

16.1.3 Selecting a Trace Output Format

FireSim supports three trace output formats, which can be set in your `config_runtime.yaml` file with the `output_format` option in the `tracing` section:

```
tracing:
    enable: yes

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
    # unwinding -> Flame Graph)
    output_format: 0
```

See the "Interpreting the Trace Result" section below for a description of these formats.

16.1.4 Setting a TracerV Trigger

Tracing the entirety of a long-running job like a Linux-based workload can generate a large trace and you may only care about the state within a certain timeframe. Therefore, FireSim allows you to specify a trigger condition for starting and stopping trace data collection.

By default, TracerV does not use a trigger, so data collection starts at cycle 0 and ends at the last cycle of the simulation. To change this, modify the following under the `tracing` section of your `config_runtime.yaml`. Use the `selector` field to choose the type of trigger (options are described below). The `start` and `end` fields are used to supply the start and end values for the trigger.

```
tracing
    enable: yes

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
    # unwinding -> Flame Graph)
    output_format: 0

    # Trigger selector.
    # 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
    # instruction trigger
    selector: 1
    start: 0
    end: -1
```

The four triggering methods available in FireSim are as follows:

No trigger

Records the trace for the entire simulation.

This is option `0` in the `.yaml` above.

The `start` and `end` fields are ignored.

Target cycle trigger

Trace recording begins when a specified start cycle is reached and ends when a specified end cycle is reached. Cycles are specified in base target-clock cycles (the zeroth output clock from the ClockBridge). For example, if the base clock drives the uncore, and the core clock frequency is 2x the uncore frequency, specifying start and end cycles of 100 and 200 result in instructions being collected between core-clock cycles 200 and 400.

This is option 1 in the `.yaml` above.

The `start` and `end` fields are interpreted as decimal integers.

Program Counter (PC) value trigger

Trace recording begins when a specified program counter value is reached and ends when a specified program counter value is reached.

This is option 2 in the `.yaml` above.

The `start` and `end` fields are interpreted as hexadecimal values.

Instruction value trigger

Trace recording begins when a specific instruction is seen in the instruction trace and ends when a specific instruction is seen in the instruction trace. This method is particularly valuable for setting the trigger from within the target software under evaluation, by inserting custom "NOP" instructions. Linux distributions included with FireSim include small trigger programs by default for this purpose; see the end of this subsection.

This is option 3 in the `.yaml` above.

The `start` and `end` fields are interpreted as hexadecimal values. For each, the field is a 64-bit value, with the upper 32-bits representing a mask and the lower 32-bits representing a comparison value. That is, the start or stop condition will be satisfied when the following evaluates to true:

```
((inst value) & (upper 32 bits)) == (lower 32 bits)
```

That is, setting `start: ffffffff00008013` will cause recording to start when the instruction value is exactly `00008013` (the `addi x0, x1, 0` instruction in RISC-V).

This form of triggering is useful when recording traces only when a particular application is running within Linux. To simplify the use of this triggering mechanism, workloads derived from `br-base.json` in FireMarshal automatically include the commands `firesim-start-trigger` and `firesim-end-trigger`, which issue a `addi x0, x1, 0` and `addi x0, x2, 0` instruction respectively. In your `config_runtime.yaml`, if you set the following trigger settings:

```
selector: 3
start: ffffffff00008013
end: ffffffff00010013
```

And then run the following at the bash prompt on the simulated system:

```
$ firesim-start-trigger && ./my-interesting-benchmark && firesim-end-trigger
```

The trace will contain primarily only traces for the duration of `my-interesting-benchmark`. Note that there will be a small amount of extra trace information from `firesim-start-trigger` and `firesim-end-trigger`, as well as the OS switching between these and `my-interesting-benchmark`.

Attention

While it is unlikely that a compiler will generate the aforementioned trigger instructions within normal application code, it is also a good idea to confirm that these instructions are not inadvertently present within the section of code you wish to profile. This will result in the trace recording inadvertently turning on and off in the middle of the workload.

On the flip-side, a developer can deliberately insert the aforementioned `addi` instructions into the code they wish to profile, to enable more fine-grained control.

16.1.5 Interpreting the Trace Result

Human readable output

This is `output_format: 0`.

The human readable trace output format looks like so:

```
# Clock Domain: baseClock, Relative Frequency: 1/1 of Base Clock
Cycle: 0000000000000079 I0: 0000000000010040
Cycle: 0000000000000105 I0: 000000000001004c
Cycle: 0000000000000123 I0: 0000000000010054
Cycle: 0000000000000135 I0: 0000000000010058
Cycle: 0000000000000271 I0: 000000000001005c
Cycle: 0000000000000307 I0: 0000000000010000
Cycle: 0000000000000327 I0: 0000000000010008
Cycle: 0000000000000337 I0: 0000000000010010
Cycle: 0000000000000337 I1: 0000000000010014
Cycle: 0000000000000337 I2: 0000000000010018
```

In this output, each line begins with the cycle (in decimal) in the core's clock domain that instruction was committed. For a given cycle, the instruction address (in hex) of each committed is prefixed `I<#>` according to their appearance in program order: `I0` is the oldest instruction committed, `I1` is the second oldest, and so forth. If no instructions were committed in a given cycle, that cycle will be skipped in the output file.

```
Cycle: 0000000000000337 I0: 0000000000010010
Cycle: 0000000000000337 I1: 0000000000010014
       |--------------|  ^        |--------|
              |          |            └ 40 bits of instruction address (hex)
              |          └ per-cycle commit-order
              └ 64-bit local-cycle count
```

Binary output

This is `output_format: 1`.

This simply writes the 512 bits received from the FPGA each cycle to the output file in binary. Each 512-bit chunk is stored little-endian. The lowermost 64 bits stores the cycle, the second 64-bits stores the address and valid bits of committed instruction 0 in little-endian, the next 64-bits stores the address and valid bits of committed instruction 1 in little-endian, and so on, up to a maximum of 7 instructions.

Flame Graph output

This is `output_format: 2`. See the *TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation* section.

16.1.6 Caveats

There are currently a few restrictions / manual tweaks that are required when using TracerV under certain conditions:

- TracerV by default outputs only instruction address and a valid bit and assumes that the combination of these fits within 64 bits. Changing this requires modifying `sim/firesim-lib/src/main/scala/bridges/TracerVBridge.scala`.
- The maximum IPC of the traced core cannot exceed 7.
- Please reach out on the FireSim mailing list if you need help addressing any of these restrictions: https://groups.google.com/forum/#!forum/firesim

16.2 Assertion Synthesis: Catching RTL Assertions on the FPGA

Golden Gate can synthesize assertions present in FIRRTL (implemented as `stop` statements) that would otherwise be lost in the FPGA synthesis flow. Rocket and BOOM include hundreds of such assertions which, when synthesized, can provide great insight into why the target may be failing.

16.2.1 Enabling Assertion Synthesis

To enable assertion synthesis prepend `WithSynthAsserts` config to your PLATFORM_CONFIG. During compilation, Golden Gate will print the number of assertions it's synthesized. In the generated header, you will find the definitions of all synthesized assertions. The `synthesized_assertions_t` bridge driver will be automatically instantiated.

16.2.2 Runtime Behavior

If an assertion is caught during simulation, the driver will print the assertion cause, the path to module instance in which it fired, a source locator, and the cycle on which the assertion fired. Simulation will then terminate.

An example of an assertion caught in a dual-core instance of BOOM is given below:

```
id: 1190, module: IssueSlot_4, path: FireSimNoNIC.tile_1.core.issue_units_0.slots_3]
Assertion failed
    at issue_slot.scala:214 assert (!slot_p1_poisoned)
    at cycle: 2142042185
```

Just as in a software-hosted RTL simulation using verilator or VCS, the reported cycle is the number of target cycles that have elapsed in the clock domain in which the assertion was instantiated (in Chisel specifically this is the implicit clock at the time you called `assert`). If you rerun a FireSim simulation with identical inputs, the same assertion should fire deterministically at the same cycle.

16.2.3 Related Publications

Assertion synthesis was first presented in our FPL2018 paper, DESSERT.

16.3 Printf Synthesis: Capturing RTL printf Calls when Running on the FPGA

Golden Gate can synthesize printfs present in Chisel/FIRRTL (implemented as `printf` statements) that would otherwise be lost in the FPGA synthesis flow. Rocket and BOOM have printfs of their commit logs and other useful transaction streams.

```
C0:        409 [1] pc=[008000004c] W[r10=0000000000000000][1] R[r 0=0000000000000000]␣
↪R[r20=0000000000000003] inst=[f1402573] csrr    a0, mhartid
C0:        410 [0] pc=[008000004c] W[r 0=0000000000000000][0] R[r 0=0000000000000000]␣
↪R[r20=0000000000000003] inst=[f1402573] csrr    a0, mhartid
C0:        411 [0] pc=[008000004c] W[r 0=0000000000000000][0] R[r 0=0000000000000000]␣
↪R[r20=0000000000000003] inst=[f1402573] csrr    a0, mhartid
C0:        412 [1] pc=[0080000050] W[r 0=0000000000000000][0] R[r10=0000000000000000]␣
↪R[r 0=0000000000000000] inst=[00051063] bnez    a0, pc + 0
C0:        413 [1] pc=[0080000054] W[r 5=0000000080000054][1] R[r 0=0000000000000000]␣
↪R[r 0=0000000000000000] inst=[00000297] auipc   t0, 0x0
C0:        414 [1] pc=[0080000058] W[r 5=0000000080000064][1] R[r 5=0000000080000054]␣
↪R[r16=0000000000000003] inst=[01028293] addi    t0, t0, 16
C0:        415 [1] pc=[008000005c] W[r 0=0000000000010000][1] R[r 5=0000000080000064]␣
↪R[r 5=0000000080000064] inst=[30529073] csrw    mtvec, t0
```

Synthesizing these printfs lets you capture the same logs on a running FireSim instance.

16.3.1 Enabling Printf Synthesis

To synthesize a printf, you need to annotate the specific printfs you'd like to capture in your Chisel source code like so:

```
midas.targetutils.SynthesizePrintf(printf("x%d p%d 0x%x\n", rf_waddr, rf_waddr, rf_
↪wdata))
```

Be judicious, as synthesizing many, frequently active printfs will slow down your simulator.

Once your printfs have been annotated, enable printf synthesis by prepending the `WithPrintfSynthesis` configuration mixin to your `PLATFORM_CONFIG` in `config_build_recipes.yaml`. For example, if your previous `PLATFORM_CONFIG` was `PLATFORM_CONFIG=BaseF1Config`, then change it to `PLATFORM_CONFIG=WithPrintfSynthesis_BaseF1Config`. Note, you must prepend the mixin. During compilation, Golden Gate will print the number of printfs it has synthesized. In the target's generated header (`FireSim-generated.const.h`), you'll find metadata for each of the printfs Golden Gate synthesized. This is passed as argument to the constructor of the `synthesized_prints_t` bridge driver, which will be automatically instantiated in FireSim driver.

16.3.2 Runtime Arguments

+print-file
Specifies the file name prefix. Generated files will be of the form <*print-file*><*N*>, with one output file generated per clock domain. The associated clock domain's name and frequency relative to the base clock is included in the header of the output file.

+print-start
Specifies the target-cycle in cycles of the base clock at which the printf trace should be captured in the simulator. Since capturing high-bandwidth printf traces will slow down simulation, this allows the user to reach the region-of-interest at full simulation speed.

+print-end
Specifies the target-cycle in cycles of the base clock at which to stop pulling the synthesized print trace from the simulator.

+print-binary
By default, a captured printf trace will be written to file formatted as it would be emitted by a software RTL simulator. Setting this dumps the raw binary coming off the FPGA instead, improving simulation rate.

+print-no-cycle-prefix
(Formatted output only) This removes the cycle prefix from each printf to save bandwidth in cases where the printf already includes a cycle field. In binary-output mode, since the target cycle is implicit in the token stream, this flag has no effect.

You can set some of these options by changing the fields in the "synthprint" section of your config_runtime.yaml.

```
synth_print:
    # Start and end cycles for outputting synthesized prints.
    # They are given in terms of the base clock and will be converted
    # for each clock domain.
    start: 0
    end: -1
    # When enabled (=yes), prefix print output with the target cycle at which the print␣
↪was triggered
    cycle_prefix: yes
```

The "start" field corresponds to "print-start", "end" to "print-end", and "cycleprefix" to "print-no-cycle-prefix".

16.3.3 Related Publications

Printf synthesis was first presented in our FPL2018 paper, DESSERT.

16.4 AutoILA: Simple Integrated Logic Analyzer (ILA) Insertion

Sometimes it takes too long to simulate FireSim on RTL simulators, and in some occasions we would also like to debug the simulation infrastructure itself. For these purposes, we can use the Xilinx Integrated Logic Analyzer resources on the FPGA.

ILAs allows real time sampling of pre-selected signals during FPGA runtime, and provided and interface for setting trigger and viewing samples waveforms from the FPGA. For more information about ILAs, please refer to the Xilinx guide on the topic.

The `midas.targetutils` package provides annotations for labeling signals directly in the Chisel source. These will be consumed by a downstream FIRRTL pass which wires out the annotated signals, and binds them to an appropriately sized ILA instance.

16.4.1 Enabling AutoILA

To enable AutoILA, mixin *WithAutoILA* must be prepended to the *PLATFORM_CONFIG*. Prior to version 1.13, this was done by default.

16.4.2 Annotating Signals

In order to annotate a signal, we must import the `midas.targetutils.FpgaDebug` annotator. FpgaDebug's apply method accepts a vararg of chisel3.Data. Invoke it as follows:

```
import midas.targetutils.FpgaDebug

class SomeModuleIO(implicit p: Parameters) extends SomeIO()(p){
   val out1 = Output(Bool())
   val in1 = Input(Bool())
   FpgaDebug(out1, in1)
}
```

You can annotate signals throughout FireSim, including in Golden Gate Rocket-Chip Chisel sources, with the only exception being the Chisel3 sources themselves (eg. in Chisel3.util.Queue).

Note: In case the module with the annotated signal is instantiated multiple times, all instatiations of the annotated signal will be wired to the ILA.

16.4.3 Setting a ILA Depth

The ILA depth parameter specifies the duration in cycles to capture annotated signals around a trigger. Increasing this parameter may ease debugging, but will also increase FPGA resource utilization. The default depth is 1024 cycles. The desired depth can be configured much like the desired HostFrequency by appending a mixin to the *PLATFORM_CONFIG*. See *Target-Side FPGA Constraints* for details on *PLATFORM_CONFIG*.

Below is an example *PLATFORM_CONFIG* that can be used in the *build_recipes* config file.

```
PLATFORM_CONFIG=ILADepth8192_BaseF1Config
```

16.4.4 Using the ILA at Runtime

Prerequisite: Make sure that ports 8443, 3121 and 10201 are enabled in the "firesim" AWS security group.

In order to use the ILA, we must enable the GUI interface on our manager instance. In the past, AWS had a custom `setup_gui.sh` script. However, this was recently deprecated due to compatibility issues with various packages. Therefore, AWS currently recommends using NICE DCV as a GUI client. You should download a DCV client, and then run the following commands on your FireSim manager instance:

```
sudo yum -y groupinstall "GNOME Desktop"
sudo yum -y install glx-utils
sudo rpm --import https://s3-eu-west-1.amazonaws.com/nice-dcv-publish/NICE-GPG-KEY
```

(continues on next page)

(continued from previous page)

```
wget https://d1uj6qtbmh3dt5.cloudfront.net/2019.0/Servers/nice-dcv-2019.0-7318-el7.tgz
tar xvf nice-dcv-2019.0-7318-el7.tgz
cd nice-dcv-2019.0-7318-el7
sudo yum -y install nice-dcv-server-2019.0.7318-1.el7.x86_64.rpm
sudo yum -y install nice-xdcv-2019.0.224-1.el7.x86_64.rpm
sudo systemctl enable dcvserver
sudo systemctl start dcvserver
sudo passwd centos
sudo systemctl stop firewalld
dcv create-session --type virtual --user centos centos
```

These commands will setup Linux desktop pre-requisites, install the NICE DCV server, ask you to setup the password to the `centos` user, disable firewalld, and finally create a DCV session. You can now connect to this session through the DCV client.

After access the GUI interface, open a terminal, and open `vivado`. Follow the instructions in the AWS-FPGA guide for connecting xilinx hardware manager on vivado (running on a remote machine) to the debug target .

where `<hostname or IP address>` is the internal IP of the simulation instance (not the manager instance. i.e. The IP starting with 192.168.X.X). The probes file can be found in the manager instance under the path `firesim/deploy/results-build/<build_identifier>/cl_firesim/build/checkpoints/<probes_file.ltx>`

Select the ILA with the description of *WRAPPER_INST/CL/CL_FIRESIM_DEBUG_WIRING_TRANSFORM*, and you may now use the ILA just as if it was on a local FPGA.

16.5 AutoCounter: Profiling with Out-of-Band Performance Counter Collection

FireSim can provide visibility into a simulated CPU's architectural and microarchitectural state over the course of execution through the use of counters. These are similar to performance counters provided by processor vendors, and more general counters provided by architectural simulators.

This functionality is provided by the AutoCounter feature (introduced in our FirePerf paper at ASPLOS 2020), and can be used for profiling and debugging. Since AutoCounter injects counters only in simulation (unlike target-level performance counters), these counters do not affect the behavior of the simulated machine, no matter how often they are sampled.

16.5.1 Chisel Interface

AutoCounter enables the addition of ad-hoc counters using the `PerfCounter` object in the *midas.targetutils* package. PerfCounters counters can be added in one of two modes:

1. *Accumulate*, using the standard `PerfCounter.apply` method. Here the annotated UInt (1 or more bits) is added to a 64b accumulation register: the target is treated as representing an N-bit UInt and will increment the counter by a value between [0, 2^n - 1] per cycle.
2. *Identity*, using the `PerfCounter.identity` method. Here the annotated UInt is sampled directly. This can be used to annotate a sample with values are not accumulator-like (e.g., a PC), and permits the user to define more complex instrumentation logic in the target itself.

We give examples of using PerfCounter below:

```
// A standard boolean event. Increments by 1 or 0 every local clock cycle.
midas.targetutils.PerfCounter(en_clock, "gate_clock", "Core clock gated")

// A multibit example. If the core can retire three isntructions per cycle,
// encode this as a two-bit unit. Extra-width is OK but the encoding to the UInt
// (e.g., doing a pop count), must be done by the user.
midas.targetutils.PerfCounter(insns_ret, "iret", "Instructions retired")

// An identity value. Note: the pc here must be <= 64b wide.
midas.targetutils.PerfCounter.identity(pc, "pc", "The value of the program counter at␣
↪the time of a sample")
```

See the PerfCounter Scala API docs for more detail about the Chisel-side interface.

16.5.2 Enabling AutoCounter in Golden Gate

By default, annotated events are not synthesized into AutoCounters. To enable AutoCounter when compiling a design, prepend the `WithAutoCounter` config to your `PLATFORM_CONFIG`. During compilation, Golden Gate will print the signals it is generating counters for.

16.5.3 Rocket Chip Cover Functions

The cover function is applied to various signals in the Rocket Chip generator repository to mark points of interest (i.e., interesting signals) in the RTL. Tools are free to provide their own implementation of this function to process these signals as they wish. In FireSim, these functions can be used as a hook for automatic generation of counters.

Since cover functions are embedded throughout the code of Rocket Chip (and possibly other code repositories), AutoCounter provides a filtering mechanism based on module granularity. As such, only cover functions that appear within selected modules will generate counters.

The filtered modules can be indicated using one of two methods:

1. An annotation attached to the module for which cover functions should be turned into AutoCounters. The annotation requires a `ModuleTarget` which can be pointed to any module in the design. Alternatively, the current module can be annotated as follows:

```
class SomeModule(implicit p: Parameters) extends Module
{
  chisel3.experimental.annotate(AutoCounterCoverModuleAnnotation(
      Module.currentModule.get.toTarget))
}
```

2. An input file with a list of module names. This input file is named `autocounter-covermodules.txt`, and includes a list of module names separated by new lines (no commas).

16.5.4 AutoCounter Runtime Parameters

AutoCounter currently takes a single runtime configurable parameter, defined under the `autocounter:` section in the `config_runtime.yaml` file. The `read_rate` parameter defines the rate at which the counters should be read, and is measured in target-cycles of the base target-clock (clock 0 produced by the ClockBridge). Hence, if the read_rate is defined to be 100 and the tile frequency is 2x the base clock (ex., which may drive the uncore), the simulator will read and print the values of the counters every 200 core-clock cycles. If the core-domain clock is the base clock, it would do so every 100 cycles. By default, the read_rate is set to 0 cycles, which disables AutoCounter.

```
autocounter:
    # read counters every 100 cycles
    read_rate: 100
```

Note

AutoCounter is designed as a coarse-grained observability mechanism, as sampling each counter requires two (blocking) MMIO reads (each read takes O(100) ns on EC2 F1). As a result sampling at intervals less than O(10000) cycles may adversely affect simulation performance for large numbers of counters. If you intend on reading counters at a finer granularity, consider using synthesizable printfs.

16.5.5 AutoCounter CSV Output Format

AutoCounter output files are CSVs generated in the working directory where the simulator was invoked (this applies to metasimulators too), with the default names `AUTOCOUNTERFILE<i>.csv`, one per clock domain. The CSV output format is depicted below, assuming a sampling period of `N` base clock cycles.

Table 1: AutoCounter CSV Format

version	*version number*				
Clock Domain Name	*name*	Base Multiplier	*M*	Base Divisor	*N*
label	local_cycle	label0	label1	…	labelN
description	local clock cycle	desc0	desc1	…	descN
type	Accumulate	type0	type1	…	typeN
event width	1	width0	width1	…	widthN
accumulator width	64	64	64	…	64
N	cycle @ time N	value0 @ tN	value1 @ tN	…	value @ tN
…	…	…	…	…	
kN	cycle @ time kN	value0 @ tkN	value1 @ tkN	…	valueN @ tkN

Column Notes:

1. Each column beyond the first two corresponds to a PerfCounter instance in the clock domain.
2. Column 0 past the header corresponds to the base clock cycle of the sample.
3. The local_cycle counter (column 1) is implemented as an always enabled single-bit event, and increments even when the target is under reset.

Row Notes:

1. Header row 0: autocounter csv format version, an integer.
2. Header row 1: clock domain information.
3. Header row 2: the label parameter provided to PerfCounter suffixed with the instance path.

4. Header row 3: the description parameter provided to PerfCounter. Quoted.
5. Header row 4: the width of the field annotated in the target.
6. Header row 5: the width of the accumulation register. Not configurable, but makes it clear when to expect rollover.
7. Header row 6: indicates the accumulation scheme. Can be "Identity" or "Accumulate".
8. Sample row 0: sampled values at the bitwidth of the accumulation register.
9. Sample row k: ditto above, k * N base cycles later

16.5.6 Using TracerV Trigger with AutoCounter

In order to collect AutoCounter results from only from a particular region of interest in the simulation, AutoCounter has been integrated with TracerV triggers. See the *Setting a TracerV Trigger* section for more information.

16.5.7 AutoCounter using Synthesizable Printfs

The AutoCounter transformation in Golden Gate includes an event-driven mode that uses Synthesizable Printfs (see *Printf Synthesis: Capturing RTL printf Calls when Running on the FPGA*) to export counter results *as they are updated* rather than sampling them periodically with a dedicated Bridge. This mode can be enabled by prepending the `WithAutoCounterCoverPrintf` config to your `PLATFORM_CONFIG` instead of `WithAutoCounterCover`. Based on the selected event mode the printfs will have the following runtime behavior:

- *Accumulate*: On a non-zero increment, the local cycle count and the new counter value are printed. This produces a series of prints with monotonically increasingly values.
- *Identity*: On a transition of the annotated target, the local cycle count and the new value are printed. Thus a target that transitions every cycle will produce printf traffic every cycle.

This mode may be useful for temporally fine-grained observation of counters. The counter values will be printed to the same output stream as other synthesizable printfs. This mode uses considerably more FPGA resources per counter, and may consume considerable amounts of DMA bandwidth (since it prints every cycle a counter increments), which may adversly affect simulation performance (increased FMR).

16.5.8 Reset & Timing Considerations

- Events and identity values provided while under local reset, or while the `GlobalResetCondition` asserted, are zero-ed out. Similarly, printfs that might otherwise be active under a reset are masked out.
- The sampling period in slower clock domains is currently calculated using a truncating division of the period in the base clock domain. Thus, when the base clock period can not be cleanly divided, samples in the slower clock domain will gradually fall out of phase with samples in the base clock domain. In all cases, the "local_cycle" column is most accurate measure of sample time.

16.6 TracerV + Flame Graphs: Profiling Software with Out-of-Band Flame Graph Generation

FireSim supports generating Flame Graphs out-of-band, to visualize the performance of software running on simulated processors. This feature was introduced in our FirePerf paper at ASPLOS 2020 .

Before proceeding, make sure you understand the *Capturing RISC-V Instruction Traces with TracerV* section.

16.6.1 What are Flame Graphs?

Fig. 1: Example Flame Graph (from http://www.brendangregg.com/FlameGraphs/)

Flame Graphs are a type of histogram that shows where software is spending its time, broken down by components of the stack trace (e.g., function calls). The x-axis represents the portion of total runtime spent in a part of the stack trace, while the y-axis represents the stack depth at that point in time. Entries in the flame graph are labeled with and sorted by function name (not time).

Given this visualization, time-consuming routines can easily be identified: they are leaves (top-most horizontal bars) of the stacks in the flame graph and consume a significant proportion of overall runtime, represented by the width of the horizontal bars.

Traditionally, data to produce Flame Graphs is collected using tools like `perf`, which sample stack traces on running systems in software. However, these tools are limited by the fact that they are ultimately running additional software on the system being profiled, which can change the behavior of the software that needs to be profiled. Furthermore, as sampling frequency is increased, this effect becomes worse.

In FireSim, we use the out-of-band trace collection provided by TracerV to collect these traces *cycle-exactly* and *without perturbing running software*. On the host-software side, TracerV unwinds the stack based on DWARF information about the running binary that you supply. This stack trace is then fed to the open-source FlameGraph stack trace visualizer to produce Flame Graphs.

16.6.2 Prerequisites

1. Make sure you understand the *Capturing RISC-V Instruction Traces with TracerV* section.
2. You must have a design that integrates the TracerV bridge. See the *Building a Design with TracerV* section.

16.6.3 Enabling Flame Graph generation in `config_runtime.yaml`

To enable Flame Graph generation for a simulation, you must set `enable: yes` and `output_format: 2` in the `tracing` section of your `config_runtime.yaml` file, for example:

```
tracing:
    enable: yes

    # Trace output formats. Only enabled if "enable" is set to "yes" above
    # 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (stack
    # unwinding -> Flame Graph)
    output_format: 2

    # Trigger selector.
```

(continues on next page)

(continued from previous page)

```
    # 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
    # instruction trigger
    selector: 1
    start: 0
    end: -1
```

The trigger selector settings can be set as described in the *Setting a TracerV Trigger* section. In particular, when profiling the OS only when a desired application is running (e.g., `iperf3` in our ASPLOS 2020 paper), instruction value triggering is extremely useful. See the *Instruction value trigger* section for more.

16.6.4 Producing DWARF information to supply to the TracerV driver

When running in FirePerf mode, the TracerV software driver expects a binary containing DWARF debugging information, which it will use to obtain labels for stack unwinding.

TracerV expects this file to be named exactly as your `bootbinary`, but suffixed with `-dwarf`. For example (and as we will see in the following section), if your `bootbinary` is named `br-base-bin`, TracerV will require you to provide a file named `br-base-bin-dwarf`.

If you are generating a Linux distribution with FireMarshal, this file containing debug information for the generated Linux kernel will automatically be provided (and named correctly) in the directory containing your images. For example, building the `br-base.json` workload will automatically produce `br-base-bin`, `br-base-bin-dwarf` (for TracerV flame graph generation), and `br-base.img`.

16.6.5 Modifying your workload description

Finally, we must make three modifications to the workload description to complete the flame graph flow. For general documentation on workload descriptions, see the *[DEPRECATED] Defining Custom Workloads* section.

1. We must add the file containing our DWARF information as one of the `simulation_inputs`, so that it is automatically copied to the remote F1 instance running the simulation.
2. We must modify `simulation_outputs` to copy back the generated trace file.
3. We must set the `post_run_hook` to `gen-all-flamegraphs-fireperf.sh` (which FireSim puts on your path by default), which will produce flame graphs from the trace files.

To concretize this, let us consider the default `br-base-uniform.json` workload, which does not support Flame Graph generation with modifications found here: `br-base-flamegraph.json`, which makes the aforementioned three changes:

```
{
  "benchmark_name": "br-base-flamegraph",
  "common_bootbinary": "../../../target-design/chipyard/software/firemarshal/images/
↪firechip/br-base/br-base-bin",
  "common_rootfs": "../../../target-design/chipyard/software/firemarshal/images/firechip/
↪br-base/br-base.img"
  "common_simulation_inputs": [
    "../../../target-design/chipyard/software/firemarshal/images/firechip/br-base/br-
↪base-bin-dwarf"
  ],
  "common_outputs": [
    "/etc/os-release"
```

(continues on next page)

(continued from previous page)

```
  ],
  "common_simulation_outputs": [
    "uartlog",
    "memory_stats*.csv",
    "TRACEFILE*"
  ],
  "post_run_hook": "gen-all-flamegraphs-fireperf.sh"
}
```

Note that we are adding `TRACEFILE*` to `common_simulation_outputs`, which will copy back all generated trace files to your workload results directory. The `gen-all-flamegraphs-fireperf.sh` script will automatically produce a flame graph for each generated trace.

Lastly, if you have created a new workload definition, make sure you update your `config_runtime.yaml` to use this new workload definition.

16.6.6 Running a simulation

At this point, you can follow the standard FireSim flow to run a workload. Once your workload completes, you will find trace files with stack traces (as opposed to instruction traces) and generated flame graph SVGs in your workload's output directory.

16.6.7 Caveats

The current stack trace construction code does not distinguish between different userspace programs, instead consolidating them into one entry. Expanded support for userspace programs will be available in a future release.

16.7 Spike Co-simulation with BOOM designs

Instead of using TracerV to provide a cycle-by-cycle trace of a target CPU's architectural state, you can use the Spike co-simulator to verify the functionality of a BOOM design.

Note

This work currently only works for single core BOOM designs.

Note

Cospike only supports non block-device simulations at this time.

16.7.1 Building a Design with Cospike

In all FireChip designs, TracerV is included by default. To enable Cospike, you just need to add the Cospike bridge (`WithCospikeBridge`) to your BOOM target design config (default configs. located in `$CHIPYARD/generators/firechip/src/main/scala/TargetConfigs.scala`). An example configuration with Cospike is shown below:

```
class FireSimLargeBoomConfig extends Config(
  new WithCospikeBridge ++ // add Cospike bridge to simulation
  new WithDefaultFireSimBridges ++
  new WithDefaultMemModel ++
  new WithFireSimConfigTweaks ++
  new chipyard.LargeBoomV3Config)
```

At this point, you should run the `firesim buildbitstream` command for the BOOM config wanted. At this point you are ready to run the simulation with Cospike by default enabled.

16.7.2 Troubleshooting Cospike Simulations with Meta-Simulations

If FPGA simulation fails with Cospike, you can use metasimulation to determine if your Cospike setup is correct. First refer to *Debugging & Testing with Metasimulation* for more information on metasimulation.

Note

Sometimes simulations in VCS will diverge unless a `+define+RANDOM=0` is added to the VCS flags in `sim/midas/src/main/cc/rtlsim/Makefrag-vcs`.

16.8 Debugging a Hanging Simulator

A common symptom of a failing simulation is that appears to hang. Debugging this is especially daunting in FireSim because it's not immediately obvious if it's a bug in the target, or somewhere in the host. To make it easier to identify the problem, the simulation driver includes a polling watchdog that tracks for simulation progress, and periodically updates an output file, `heartbeat.csv`, with a target cycle count and a timestamp. When debugging these issues, we always encourage the use of metasimulation to try reproducing the failure if possible. We outline three common cases in the section below.

16.8.1 Case 1: Target hang.

Symptoms: There is no output from the target (i.e., the uartlog might cease), but simulated time continues to advance (`heartbeat.csv` will be periodically updated). Simulator instrumentation (TracerV, printf) may continue to produce new output.

Causes: Typically, a bug in the target RTL. However, bridge bugs leading to erroneous token values will also produce this behavior.

Next steps: You can deploy the full suite of FireSim's debugging tools for failures of this nature, since assertion synthesis, printf synthesis, and other target-side features still function. Assume there is a bug in the target RTL and trace back the failure to a bridge if applicable.

16.8.2 Case 2: Simulator hang due to FPGA-side token starvation.

Symptoms: The driver's main loop spins freely, as no bridge gets new work to do. As a result, the polling interval quickly elapses and the simulation is torn down due to a lack of forward progress.

Causes: Generally, a bug in a bridge implementation (ex. the BridgeModule has accidentally dequeued a token without producing a new output token; the BridgeModule is waiting on a driver interaction that never occurs).

Next steps: These are the trickiest to solve. Try to identify the bridge that's responsible by removing unnecessary ones, using an AutoILA, and adding printfs to BridgeDriver sources. Target-side debugging utilities may be used to identify problematic target behavior, but tend not to be useful for identifying the root cause.

16.8.3 Case 3: Simulator hang due to driver-side deadlock.

Symptoms: The loss of all output, notably, `heartbeat.csv` ceases to be further updated.

Causes: Generally, a bridge driver bug. For example, the driver may be busy waiting on some output from the FPGA, but the FPGA-hosted part of the simulator has stalled waiting for tokens.

Next Steps: Identify the buggy driver using printfs or attaching to the running simulator using GDB.

16.8.4 Simulator Heartbeat PlusArgs

`+heartbeat-polling-interval=<int>`: Specifies the number of round trips through the simulator main loop before polling the FPGA's target cycle counter. Disable the heartbeat by setting this to -1.

16.9 PlusArg Synthesis: Runtime Modification of RTL

Golden Gate can synthesize Rocket Chip plusargs present in Chisel/FIRRTL that would otherwise be lost in the FPGA synthesis flow. These plusargs allow you to drive a wire to specific value at the start of simulation.

For example:

```
import freechips.rocketchip.util._

val my_wire = PlusArg("set_my_wire", 0, "Description")
```

Then you can change the value of the `my_wire` during runtime instead of having to re-synthesize the module with a different value.

16.9.1 Enabling PlusArg Synthesis

To synthesize a plusarg, you need to first use a Rocket Chip `plusarg_reader` module directly, like so:

```
import freechips.rocketchip.util._

// see the API for plusarg_reader in the Rocket Chip source code
// this adds a plusarg with the name 'set_my_wire', default of '0', and a width of '32'
val my_plusarg_module = Module(new plusarg_reader("set_my_wire=%d", 0, "Description",␣
↪32))
val my_wire = my_plusarg_module.io.out
```

Then you can annotate the specific plusargs you'd like to capture in your Chisel source code like so:

```
midas.targetutils.PlusArg(my_plusarg_module)
```

During compilation, Golden Gate will print the number of plusargs it has synthesized. In the target's generated header (`FireSim-generated.const.h`), you'll find metadata for each of the plusargs Golden Gate synthesized. This is passed as argument to a bridge driver, which will be automatically instantiated in FireSim driver.

16.9.2 Runtime Arguments

By default, the plusarg will default to the default value specified in the `plusarg_reader` module that was annotated. To change this value you can directly call the runtime argument of the same name with the new value to be given at simulation start.

For example:

+set_my_wire=50
Sets the value at the start of simulation to '50'

You can set this in the `target_config` `plusarg_passthrough` field of your `config_runtime.yaml`.

CHAPTER

SEVENTEEN

NON-SOURCE DEPENDENCY MANAGEMENT

In the AWS EC2 F1 setup, in *Setting up your Manager Instance*, we quickly copy-pasted the contents of `scripts/machine-launch-script.sh` into the EC2 Management Console and that script installed many dependencies that FireSim needs using Conda, a platform-agnostic package manager, specifically using packages from the conda-forge community (or in the case of *Setting up your Manager Instance*, we ran `scripts/machine-launch-script.sh`).

In many situations, you may not need to know anything about `conda`. By default, the `machine-launch-script.sh` installs `conda` into `/opt/conda` and all of the FireSim dependencies into a 'named environment' `firesim` at `/opt/conda/envs/firesim`. `machine-launch-setup.sh` also adds the required setup to the system-wide `/etc/profile.d/conda.sh` init script to add `/opt/conda/envs/firesim/bin` to everyone's path.

However, the script is also flexible. For example, if you do not have root access, you can specify an alternate install location with the `--prefix` option to `machine-launch-script.sh`. The only requirement is that you are able to write into the install location. See `machine-launch-script.sh --help` for more details.

> **Warning**
>
> To *run a simulation on a F1 FPGA* , FireSim currently requires that you are able to act as root via `sudo`.
>
> However, you can do many things without having root, like *Building Your Own Hardware Designs (FireSim Amazon FPGA Images)*, meta-simulation of a FireSim system using Verilator or even developing new features in FireSim.

17.1 Updating a Package Version

If you need a newer version of package, the most expedient method to see whether there is a newer version available on conda-forge is to run `conda update <package-name>`. If you are lucky, and the dependencies of the package you want to update are simple, you'll see output that looks something like this:

```
bash-4.2$ conda update moto
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /opt/conda

  added / updated specs:
    - moto


```

(continues on next page)

(continued from previous page)

```
The following NEW packages will be INSTALLED:

  graphql-core       conda-forge/noarch::graphql-core-3.2.0-pyhd8ed1ab_0

The following packages will be UPDATED:

  moto                                 2.2.19-pyhd8ed1ab_0 --> 3.1.0-pyhd8ed1ab_0

Proceed ([y]/n)?
```

The addition of `graphql-core` makes sense because the diff of moto's setup.py between 2.2.19 and 3.1.0 shows it was clearly added as a new dependence.

And this output tells us that latest version of `moto` available is 3.1.0. Now, you might be tempted to hit <<`Enter`>> and move forward with your life.

Attention

However, it is always a better idea to modify the version in `machine-launch-script.sh` so that: #. you remember to commit and share the new version requirement. #. you are providing a complete set of requirements for `conda` to solve. There is a subtle difference between installing everything you need in a single *conda install* vs incrementally installing one or two packages at a time because the version constraints *are not maintained between conda invocations*. (NOTE: certain packages like Python are implicitly pinned at environment creation and will only be updated if explicitly requested .)

So, modify `machine-launch-script.sh` with the updated version of `moto`, and run it. If you'd like to see what `machine-launch-script.sh` will do before actually making changes to your environment, feel free to give it the `--dry-run` option, look at the output and then run again without `--dry-run`.

In this case, when you are finished, you can run `conda list --revisions` and you should see output like the following

```
bash-4.2$ conda list --revisions
2022-03-15 19:21:10  (rev 0)
+_libgcc_mutex-0.1 (conda-forge/linux-64)
+_openmp_mutex-4.5 (conda-forge/linux-64)
+_sysroot_linux-64_curr_repodata_hack-3 (conda-forge/noarch)
+alsa-lib-1.2.3 (conda-forge/linux-64)
+appdirs-1.4.4 (conda-forge/noarch)
+argcomplete-1.12.3 (conda-forge/noarch)

 ...   many packages elided for this example ...

+xxhash-0.8.0 (conda-forge/linux-64)
+xz-5.2.5 (conda-forge/linux-64)
+yaml-0.2.5 (conda-forge/linux-64)
+zipp-3.7.0 (conda-forge/noarch)
+zlib-1.2.11 (conda-forge/linux-64)
+zstd-1.5.2 (conda-forge/linux-64)

2022-03-15 19:34:06  (rev 1)
     moto  {2.2.19 (conda-forge/noarch) -> 3.1.0 (conda-forge/noarch)}
```

This shows you that the first time `machine-launch-script.sh` was run, it created 'revision' 0 of the environment with many packages. After updating the version of `moto` and rerunning, 'revision' 1 was created by updating the version of `moto`. At any time, you can revert your Conda environment back to an older 'revision' using `conda install -revision <n>`

17.2 Multiple Environments

In the example above, we only wanted to update a single package and it was fairly straightforward – it only updated that package and installed a new dependency. However, what if we're making a larger change and we think we might need to have both sets of tools around for awhile?

In this case, make use of the `--env <name>` option of `machine-launch-script.sh`. By giving a descriptive name with that option, you will create another 'environment'. You can see a listing of available environments by running `conda env list` to get output similar to:

```
bash-4.2$   conda env list
# conda environments:
#
base                     /opt/conda
firesim                  /opt/conda/envs/firesim
doc_writing           *  /opt/conda/envs/doc_writing
```

In the output above, you can see that I had the 'base' environment that is created when you install `conda` as well as the `firesim` environment that `machine-launch-script.sh` creates by default. I also created a 'doc_writing' environment to show some of the examples pasted earlier.

You can also see that 'doc_writing' has an asterisk next to it, indicating that it is the currently 'activated' environment. To switch to a different environment, I could `conda activate <name>` e.g. `conda activate firesim`

By default, `machine-launch-script.sh` installs the requirements into 'firesim' and runs `conda init` to ensure that the 'firesim' environment is activated at login.

17.3 Adding a New Dependency

Look for what you need in this order:

1. The existing conda-forge packages list. Keep in mind that since `conda` spans several domains, the package name may not be exactly the same as a name from PyPI or one of the system package managers.
2. Adding a conda-forge recipe. If you do this, let the firesim@googlegroups.com mailing list know so that we can help get the addition merged.
3. PyPI (for Python packages). While it is possible to install packages with pip into a `conda` environment, there are caveats. In short, you're less likely to create a mess if you use only Conda to manage the requirements and dependencies in your environment.
4. System packages as a last resort. It's very difficult to have the same tools on different platforms when they are being built and shipped by different systems and organizations. That being said, in a pinch, you can find a section for platform-specific setup in `machine-launch-script.sh`.
5. As a *super* last resort, add code to `machine-launch-script.sh` or `build-setup.sh` that installs whatever you need and during your PR, we'll help you migrate to one of the other options above.

17.4 Building From Source

If you find that a package is missing an optional feature, consider looking up it's 'feedstock' (aka recipe) repo in The existing conda-forge packages list. and submitting an issue or PR to the 'feedstock' repo.

If you instead need to enable debugging or possibly actively hack on the source of a package:

1. Find the feedstock repo in the feedstock-list
2. Clone the feedstock repo and modify `recipe/build.sh` (or `recipe/meta.yaml` if there isn't a build script)
3. `python build-locally.py` to build using the conda-forge docker container If the build is successful, you will have an installable `conda` package in `build_artifacts/linux-64` that can be installed using `conda install -c ./build_artifacts <packagename>`. If the build is not successful, you can add the `--debug` switch to `python build-locally.py` and that will drop you into an interactive shell in the container. To find the build directory and activate the correct environment, just follow the instructions from the message that looks like:

```
################################################################################
Build and/or host environments created for debugging.  To enter a debugging␣
↪environment:

cd /Users/UserName/miniconda3/conda-bld/debug_1542385789430/work && source /Users/
↪UserName/miniconda3/conda-bld/debug_1542385789430/work/build_env_setup.sh

To run your build, you might want to start with running the conda_build.sh file.
################################################################################
```

If you are developing a Python package, it is usually easiest to install all dependencies using `conda` and then install your package in 'development mode' using `pip install -e <path to clone>` (and making sure that you are using `pip` from your environment).

17.5 Running Conda with sudo

`tl;dr;` run Conda like this when using `sudo`:

```
sudo -E $CONDA_EXE <remaining options to conda>
```

If you look closely at `machine-launch-script.sh`, you will notice that it always uses the full path to `$CONDA_EXE`. This is because `/etc/sudoers` typically doesn't bless our custom install prefix of `/opt/conda` in the `secure_path`.

You also probably want to include the `-E` option to `sudo` (or more specifically `--preserve-env=CONDA_DEFAULT_ENV`) so that the default choice for the environment to modify is preserved in the sudo environment.

17.6 Running things from your Conda environment with sudo

If you are running other commands using sudo (perhaps to run something under gdb), remember, the `secure_path` does not include the Conda environment by default and you will need to specify the full path to what you want to run, or in some cases, it is easiest to wrap what you want to run in a full login shell invocation like:

```
sudo /bin/bash -l -c "<command to run as root>"
```

The `-l` option to `bash` ensures that the **default** Conda environment is fully activated. In the rare case that you are using a non-default named environment, you will want to activate it before running your command:

```
sudo /bin/bash -l -c "conda activate <myenv> && <command to run as root>"
```

17.7 Additional Resources

- conda-forge
- Conda Documentation

CHAPTER

EIGHTEEN

SUPERNODE - MULTIPLE SIMULATED SOCS PER FPGA

Supernode allows users to run multiple simulated SoCs per-FPGA in order to improve FPGA resource utilization and reduce cost. For example, in the case of using FireSim to simulate a datacenter scale system, supernode mode allows realistic rack topology simulation (32 simulated nodes) using a single `f1.16xlarge` instance (8 FPGAs).

Below, we outline the build and runtime configuration changes needed to utilize supernode designs. Supernode is currently only enabled for RocketChip designs with NICs. More details about supernode can be found in the FireSim ISCA 2018 Paper.

18.1 Introduction

By default, supernode packs 4 identical designs into a single FPGA, and utilizes all 4 DDR channels available on each FPGA on AWS F1 instances. It currently does so by generating a wrapper top level target which encapsualtes the four simulated target nodes. The packed nodes are treated as 4 separate nodes, are assigned their own individual MAC addresses, and can perform any action a single node could: run different programs, interact with each other over the network, utilize different block device images, etc. In the networked case, 4 separate network links are presented to the switch-side.

18.2 Building Supernode Designs

Here, we outline some of the changes between supernode and regular simulations that are required to build supernode designs.

The Supernode target configuration wrapper can be found in Chipyard in `chipyard/generators/firechip/src/main/scala/TargetConfigs.scala`. An example wrapper configuration is:

```
class SupernodeFireSimRocketConfig extends Config(
   new WithNumNodes(4) ++
   new freechips.rocketchip.subsystem.WithExtMemSize((1 << 30) * 8L) ++ // 8 GB
   new FireSimRocketConfig)
```

In this example, `SupernodeFireSimRocketConfig` is the wrapper, while `FireSimRocketConfig` is the target node configuration. To simulate a different target configuration, we will generate a new supernode wrapper, with the new target configuration. For example, to simulate 4 quad-core nodes on one FPGA, you can use:

```
class SupernodeFireSimQuadRocketConfig extends Config(
   new WithNumNodes(4) ++
   new freechips.rocketchip.subsystem.WithExtMemSize((1 << 30) * 8L) ++ // 8 GB
   new FireSimQuadRocketConfig)
```

Next, when defining the build recipe, we must remember to use the supernode configuration: The `DESIGN` parameter should always be set to `FireSim`, while the `TARGET_CONFIG` parameter should be set to the wrapper configuration that was defined in `chipyard/generators/firechip/src/main/scala/TargetConfigs.scala`. The `PLATFORM_CONFIG` can be selected the same as in regular FireSim configurations. For example:

```
DESIGN: FireSim
TARGET_CONFIG: SupernodeFireSimQuadRocketConfig
PLATFORM_CONFIG: BaseF1Config
deploy_quintuplet: null
```

We currently provide a single pre-built AGFI for supernode of 4 quad-core RocketChips with DDR3 memory models. You can build your own AGFI, using the supplied samples in `config_build_recipes.yaml`. Importantly, in order to meet FPGA timing contraints, Supernode target may require lower host clock frequencies. Host clock frequencies can be configured as parts of the `platform_config_args` (this must be done using `PLATFORM_CONFIG` if not using F1) in `config_build_recipes.yaml`.

18.3 Running Supernode Simulations

Running FireSim in supernode mode follows the same process as in "regular" mode. Currently, the only difference is that the main simulation screen remains with the name `fsim0`, while the three other simulation screens can be accessed by attaching `screen` to `uartpty1`, `uartpty2`, `uartpty3` respectively. All simulation screens will generate uart logs (`uartlog1`, `uartlog2`, `uartlog3`). Notice that you must use `sudo` in order to attach to the uartpty or view the uart logs. The additional uart logs will not be copied back to the manager instance by default (as in a "regular" FireSim simulation). It is neccessary to specify the copying of the additional uartlogs (uartlog1, uartlog2, uartlog3) in the workload definition.

Supernode topologies utilize a `FireSimSuperNodeServerNode` class in order to represent one of the 4 simulated target nodes which also represents a single FPGA mapping, while using a `FireSimDummyServerNode` class which represent the other three simulated target nodes which do not represent an FPGA mapping. In supernode mode, topologies should always add nodes in pairs of 4, as one `FireSimSuperNodeServerNode` and three `FireSimDummyServerNode` s.

Various example Supernode topologies are provided, ranging from 4 simulated target nodes to 1024 simulated target nodes.

Below are a couple of useful examples as templates for writing custom Supernode topologies.

A sample Supernode topology of 4 simulated target nodes which can fit on a single `f1.2xlarge` is:

```python
def supernode_example_4config(self):
    self.roots = [FireSimSwitchNode()]
    servers = [FireSimSuperNodeServerNode()] + [
        FireSimDummyServerNode() for x in range(3)
    ]
    self.roots[0].add_downlinks(servers)
```

A sample Supernode topology of 32 simulated target nodes which can fit on a single `f1.16xlarge` is:

```python
def supernode_example_32config(self):
    self.roots = [FireSimSwitchNode()]
    servers = UserTopologies.supernode_flatten(
        [
            [
                FireSimSuperNodeServerNode(),
```

(continues on next page)

(continued from previous page)

```
                FireSimDummyServerNode(),
                FireSimDummyServerNode(),
                FireSimDummyServerNode(),
            ]
            for y in range(8)
        ]
    )
    self.roots[0].add_downlinks(servers)
```

Supernode `config_runtime.yaml` requires selecting a supernode agfi in conjunction with a defined supernode topology.

18.4 Work in Progress!

We are currently working on restructuring supernode to support a wider-variety of use cases (including non-networked cases, and increased packing of nodes). More documentation will follow. Not all FireSim features are currently available on Supernode. As a rule-of-thumb, target-related features have a higher likelihood of being supported "out-of-the-box", while features which involve external interfaces (such as TracerV) has a lesser likelihood of being supported "out-of-the-box"

CHAPTER

NINETEEN

FIREAXE - PARTITIONING ONTO MULTIPLE FPGAS

Although FPGA capacity has become large enough to simulate many large SoCs, there still are cases when a design does not fit on a single FPGA. When the design contains multiple duplicate modules, you should refer to the *Multithreading* section first. When there aren't enough duplicate modules you can use FireAxe to obtain higher simulation capacity. FireAxe is also compatible with *Multithreading* as well which enables scaling the size of the design even further.

19.1 FireAxe Overview

Since the initial release of FireSim, the proliferation of open hardware IP has enabled researchers to generate new SoC configurations that no longer fit in a single FPGA. FireAxe builds on top of FireSim to enable *partitioning of large designs onto multiple FPGAs*, overcoming the single-FPGA limitation for monolithic RTL designs.

19.2 Partition Modes

FireAxe provides users with three options (modes) to perform partitioning: exact-mode, fast-mode, and the NoC-partition-mode. These options are passed on to the compiler which will generate the correct circuitry to deal with the odd things that happen on the partition interface. A more detailed explanation about the circuitry and the partition modes are provided in the FireAxe paper. However, if you just want to use FireAxe, the below explanations are sufficient for you to get started.

19.2.1 Exact-Mode

In the exact-mode, users can choose the modules to partition out and place on separate FPGAs. The partitioned simulation will behave *exactly* the same as when running the target on a software RTL simulator. This is useful when the partition boundary is not latency insensitive (i.e. the interface is not ready-valid interface nor credit-based) and when the boundary contains combinational logic running through it.

An example of a module that can be partitioned out using the exact-mode is a RoCC accelerator since the ports to access the page-table-walker boundaries are combinationally dependent to each other.

19.2.2 Fast-Mode

When the partition boundary is latency insensitive (i.e., the interface is ready-valid or credit based), you can use the fast-mode to perform partitioned simulations. Similar to exact-mode, users can choose the modules to partition out. However, fast-mode provides higher simulation throughput by trading off simulation accuracy with performance. By injecting a single cycle of latency on the partition boundary, simulated design will run nearly 2x faster than the exact-mode.

Example modules that can be partitioned out are core tiles as the tile to bus connections are ready-valid and the interrupt signals are also latency insensitive. In practice, adding a cycle of latency on the partition boundary has nearly zero accuracy implications.

19.2.3 NoC-Partition-Mode

In the NoC partition mode, we exploit the fact that the NoC router boundaries are latency insensitive (credit based). Users simply has to specify the router nodes to be grouped together and the compiler will automatically group the modules to partition out with the selected routers. The NoC-partition-mode works only when partitioning tiles out.

19.3 Supported Platforms

Like all FireSim simulations, FireAxe can be run on both F1 and local FPGAs.

19.3.1 EC2 F1

To improve simulation performance on AWS EC2 F1 cloud FPGAs, we utilize their direct peer-to-peer inter-FPGA PCIe communication mechanism to reduce token exchange latency AWS PCIe Peer to Peer Guides. The f1.16xlarge and f1.4xlarge instances each contain multiple FPGAs (8 or 2 respectively) that can send and receive AXI4 transactions directly to/from one another without going through the host. This provides the simulator up to 1MHz of simulation throughput.

19.3.2 Local FPGAs w/ QSFP Cables

For on-premises FPGAs, we achieve an even lower link latency by utilizing cheap, off-the-shelf QSFP direct-attach-cables and integrating IP for the Aurora protocol into the FPGA shell. This exposes an AXI4-Stream interface to FireAxe. This ultra-low-latency interconnect enabled us to achieve a simulation throughput of 2MHz.

19.4 Running Fast Mode Simulations

In this section, we provide a step-by-step tutorial on how to partition a *RocketTile* out from the *SoC* and run fast-mode simulations using 2 FPGAs on EC2 F1. This assumes that you have completed the *AWS F1 Getting Started Guide* and that you are familiar with the FireSim workflow.

19.4.1 1. Building Partitioned Sims: Setting up FireAxe Target configs

To build bitstreams for partitioned simulations, we need specify which modules we want to partition out. We can do that in sim/midas/src/main/scala/configs/FireAxeTargetConfigs.scala.

```
//////////////////////////////////////////////////////////////////////////////
// - F1 partition a RocketTile out
// - Connect FPGAs using PCIe peer to peer communication scheme
//
// FPGA 0 (RocketTile) ----------- FPGA 1 (SoC subsystem)
//////////////////////////////////////////////////////////////////////////////
class RocketTileF1Config
    extends Config(
      new WithPCIM ++ // Use PCIM (PCIe peer to peer) communication scheme
        // WithPartitionGlobalInfo takes in a Seq of Seq
        // The inner Seq specifies the list of modules that should be grouped together␣
↪in a single partition.
        // The outer Seq specifies the list of partition groups.
        // Each partition group is mapped onto a separate FPGA.
        new WithPartitionGlobalInfo(
          Seq(
            Seq("RocketTile")
          )
        ) ++
        new BaseF1Config
    )

class RocketTileF1PCIMBase
    extends Config(
      new WithPartitionBase ++ // Base partition (SoC subsystem)
        new RocketTileF1Config
    )

class RocketTileF1PCIMPartition0
    extends Config(
      new WithPartitionIndex(0) ++ // Partition 0 containing the partitioned RocketTile
        new RocketTileF1Config
    )
```

The `WithPartitionGlobalInfo` fragment takes in an argument of type `Seq[Seq[String]]` where each `Seq[String]` indicates the group of modules to be extracted out to the same partition.

```
new WithPartitionGlobalInfo(Seq(
  Seq("A", "B"),
  Seq("C", "D"))
```

For instance, in the above example, modules “A”, “B” will be grouped and extracted out to one FPGA, “C”, “D” will be grouped and extracted out to another FPGA, and the base SoC will be placed on the third FPGA.

As in the above example, we can use `WithPartitionBase` and `WithPartitionIndex(idx)` to specify the partitions. `WithPartitionIndex(0)` for example, will choose `Seq("A", "B")` as the group of modules to partition out. How each partition group is mapped to the FPGA is specified in deploy/runtools/user_topology.py.

19.4.2 2. Building Partitioned Sims: *config_build_recipes.yaml*

Now, we can specify the `config_build_recipes.yaml`. We need to set the `TARGET_CONFIG` to the FireChip configuration that we want to build, and the `PLATFORM_CONFIG` to the partition configs that we defined in the above step.

```
#####################################################################################
↪#######
# Fast-mode : pull out a RocketTile out from your SoC
#####################################################################################
↪#######
# f1_rocket_split_soc_fast:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: RocketTileF1PCIMBase
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
#
# f1_rocket_split_tile_fast:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: RocketTileF1PCIMPartition0
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
```

You can now use the `firesim buildbitstream` command to build bitstreams of these partition configurations.

19.4.3 3. Running Partitioned Simulations: *user_topology.py*

Once the bitstreams are built and you copied the hwdb entries into `config_hwdb.yaml`, we need to setup `config_runtime.yaml` and deploy/runtools/user_topology.py to run FireAxe simulations.

```
    def fireaxe_rocket_fastmode_config(self) -> None:
        # DOC include start: fireaxe_fastmode_config hwdb_entries
        # hwdb_entries maps the partition index to the hwdb name
        hwdb_entries = {0: "f1_rocket_split_soc_fast", 1: "f1_rocket_split_tile_fast"}
        # DOC include end: fireaxe_fastmode_config hwdb_entries
        # DOC include start: fireaxe_fastmode_config slot_to_pidx
        # slotid_to_pidx maps the partition index to the FPGA slotid
        # For instance, `slotid_to_pidx = [2, 1, 0]` will map partition
        # index 2 to simulation slot 0, partition index 1 to simulation slot 1,
        # and partition index 0 to simulation slot 2.
        slotid_to_pidx = [0, 1]
        # DOC include end: fireaxe_fastmode_config slot_to_pidx
        # DOC include start: fireaxe_fastmode_config edges
        # The `FireAxeEdge` class is used to depict the connections between the␣
↪partitions.
        edges = [FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(1, 0))]
        # DOC include end: fireaxe_fastmode_config edges
        # DOC include start: fireaxe_fastmode_config mode
        # The PartitionMode enum contains the different partition modes that are␣
↪available
```

(continues on next page)

(continued from previous page)

```
        mode = PartitionMode.FAST_MODE
        # DOC include end: fireaxe_fastmode_config mode
        # DOC include start: fireaxe_fastmode_config summing it all up
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)
        # DOC include end: fireaxe_fastmode_config summing it all up
```

This is how the FireAxe topology looks like when partitioning a RocketTile out from the SoC. Lets look at each line individually.

The `hwdb_entries` is a dictionary mapping the partition index to the `hwdb` name.

```
        # hwdb_entries maps the partition index to the hwdb name
        hwdb_entries = {0: "f1_rocket_split_soc_fast", 1: "f1_rocket_split_tile_fast"}
```

The `slot_to_pidx` is a list mapping the partition index to the FPGA slotid. The list entries are the partition indices and the slotid is dictated by the position of the list. For instance, `slotid_to_pidx = [2, 1, 0]` will map partition index 2 to simulation slot 0, partition index 1 to simulation slot 1 and partition index 0 to simulation slot 2.

```
        # slotid_to_pidx maps the partition index to the FPGA slotid
        # For instance, `slotid_to_pidx = [2, 1, 0]` will map partition
        # index 2 to simulation slot 0, partition index 1 to simulation slot 1,
        # and partition index 0 to simulation slot 2.
        slotid_to_pidx = [0, 1]
```

We need to specify the SoC partition topology to run FireAxe simulations. The below figure depicts the current SoC partition topology. The `pidx` indicates the partition index : idx 0 corresponds to `f1_rocket_split_soc_fast` and idx 1 corresponds to `f1_rocket_split_tile_fast`. In FireAxe, each partition contains a set of bridges that are used to communicate with other partitions. The bridges attached to a partition all have a unique id starting from zero. In our example, partition 0 and partition 1 communicates through a single edge where the bridge id is both 0.

The `FireAxeEdge` class is used to depict the connections between the partitions. When connecting partition `X` to partition `Y`, we need to be aware of the bridge index that connects `X` and `Y` (`Xbidx`, `Ybidx`). The vertices of the edge can be described as a tuple of the partition index and bridge index: `Xpair = FireAxeNodeBridgedPair(X, Xbidx)` and `Ypair = FireAxeNodeBridgedPair(Y, Ybidx)`. Then, the edge can be described as `FireAxeEdge(Xpair, Ypair)`. So the edge in the above figure can be described as follows:

```
        # The `FireAxeEdge` class is used to depict the connections between the␣
↪partitions.
        edges = [FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(1, 0))]
```

Next, we need to specify the partitioning mode that we chose to build the bitstreams. In this example, we chose the `FAST_MODE`.

```
        # The PartitionMode enum contains the different partition modes that are␣
↪available
        mode = PartitionMode.FAST_MODE
```

At this point, all we need to do is call the `fireaxe_topology_config` with the above parameters.

```
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)
```

Users shouldn't need to worry about this code. All it is doing is generating the partition topology and creating a `PartitionConfig` instance that contains information about the topology and passing it to the `FireSimServerNode`.

```
    def fireaxe_topology_config(
        self,
        hwdb_entries: Dict[int, str],
        edges: List[FireAxeEdge],
        slotid_to_pidx: List[int],
        mode: PartitionMode,
    ) -> None:
        pidx_to_slotid = dict()
        for sid, pid in enumerate(slotid_to_pidx):
            pidx_to_slotid[pid] = sid

        # Create partition nodes
        pidx_to_partition_node: Dict[int, PartitionNode] = dict()
        for pidx, hwdb in hwdb_entries.items():
            node = PartitionNode(hwdb, pidx)
            pidx_to_partition_node[pidx] = node

        # Add edges to nodes
        for edge in edges:
            u_node = pidx_to_partition_node[edge.u.pidx]
            v_node = pidx_to_partition_node[edge.v.pidx]
            u_node.add_edge(edge.u.bidx, edge.v.bidx, v_node)
            v_node.add_edge(edge.v.bidx, edge.u.bidx, u_node)

        # Create PartitionConfigs and FireSimServerNode
        servers: Dict[int, FireSimServerNode] = dict()
        for pidx, node in pidx_to_partition_node.items():
            partition_cfg = PartitionConfig(node, pidx_to_slotid, mode)
            servers[pidx_to_slotid[node.pidx]] = FireSimServerNode(
                partition_cfg.get_hwdb(), partition_config=partition_cfg
            )

        # Sort the servers by their sim slot id
        servers = dict(sorted(servers.items()))
        self.roots = list(servers.values())
```

19.4.4 4. Running Partitioned Simulations: config_runtime.yaml

Now, we need to setup `config_runtime.yaml` to run FireAxe simulations. All we need to do is change the `config_runtime[target_config][topology]` to the topology that we defined in step 3.

```
target_config:
    topology: fireaxe_rocket_fastmode_config
```

At this point, you can run the `firesim runworkload` to kick off simulations.

19.5 Running Exact Mode Simulations

In this section, we provide a step-by-step tutorial on how to partition a *RocketTile* out from the *SoC* and run exact-mode simulations on EC1 F1. Similar steps can be applied to perform locally partitioned FPGA simulations. This assumes that you have read the *FireAxe running fast-mode simulations*.

19.5.1 1. Building Partitioned Sims: Setting up FireAxe Target configs

We will be reusing the FireAxe target configurations from *fast-mode*.

```
//////////////////////////////////////////////////////////////////////////////
// - F1 partition a RocketTile out
// - Connect FPGAs using PCIe peer to peer communication scheme
//
// FPGA 0 (RocketTile) ----------- FPGA 1 (SoC subsystem)
//////////////////////////////////////////////////////////////////////////////
class RocketTileF1Config
    extends Config(
      new WithPCIM ++ // Use PCIM (PCIe peer to peer) communication scheme
        // WithPartitionGlobalInfo takes in a Seq of Seq
        // The inner Seq specifies the list of modules that should be grouped together␣
↪in a single partition.
        // The outer Seq specifies the list of partition groups.
        // Each partition group is mapped onto a separate FPGA.
        new WithPartitionGlobalInfo(
          Seq(
            Seq("RocketTile")
          )
        ) ++
        new BaseF1Config
    )

class RocketTileF1PCIMBase
    extends Config(
      new WithPartitionBase ++ // Base partition (SoC subsystem)
        new RocketTileF1Config
    )

class RocketTileF1PCIMPartition0
    extends Config(
      new WithPartitionIndex(0) ++ // Partition 0 containing the partitioned RocketTile
        new RocketTileF1Config
    )
```

19.5.2 2. Building Partitioned Sims: *config_build_recipes.yaml*

We can specify the `config_build_recipes.yaml` at this point. One thing to note is that we added the `ExactMode_` in the `PLATFORM_CONFIG` field. This indicates to the FireAxe compiler to perform additional steps while partitioning so that the target behavior can be simulated in a cycle-exact manner.

```
##########################################################################################
↪#######
# Exact-mode : pull out a RocketTile out from your SoC
##########################################################################################
↪#######
# f1_firesim_rocket_soc_exact:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: ExactMode_RocketTileF1PCIMBase
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
#
# f1_firesim_rocket_tile_exact:
#     ...
#     PLATFORM: f1
#     TARGET_CONFIG: FireSimRocketConfig
#     PLATFORM_CONFIG: ExactMode_RocketTileF1PCIMPartition0
#     bit_builder_recipe: bit-builder-recipes/f1.yaml
#     ...
```

19.5.3 3. Running Partitioned Simulations: *user_topology.py*

Again, we have to specify the deploy/runtools/user_topology.py to run FireAxe simulations.

```
    def fireaxe_rocket_exactmode_config(self) -> None:
        hwdb_entries = {
            0: "f1_firesim_rocket_tile_exact",
            1: "f1_firesim_rocket_soc_exact",
        }
        slotid_to_pidx = [0, 1]
        edges = [
            # DOC include start: fireaxe_rocket_exactmode_config edge 0
            FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(1, 0)),
            # DOC include end: fireaxe_rocket_exactmode_config edge 0
            # DOC include start: fireaxe_rocket_exactmode_config edge 1
            FireAxeEdge(FireAxeNodeBridgePair(0, 1), FireAxeNodeBridgePair(1, 1)),
            # DOC include end: fireaxe_rocket_exactmode_config edge 1
        ]
        # DOC include start: fireaxe_rocket_exactmode_config mode
        mode = PartitionMode.EXACT_MODE
        # DOC include end: fireaxe_rocket_exactmode_config mode
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)
```

We should go over a couple of changes that are made compared to the fast-mode configuration.

First of all the FireAxe topology specified by `edges` has changed. This is because in the exact-mode, the compiler has to generate multiple communication channels (or edges) in between the partitions in order to model combinational logic correctly.

The partitioning topology now looks like this:

The upper edge connecting partition 0's bridge 0 to partition 1's bridge 0 can be described like this:

```
            FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(1, 0)),
```

The lower edge connecting partition 0's bridge 1 to partition 1's bridge 1 can be described like this:

```
            FireAxeEdge(FireAxeNodeBridgePair(0, 1), FireAxeNodeBridgePair(1, 1)),
```

We also changed the partition mode to `EXACT_MODE`:

```
        mode = PartitionMode.EXACT_MODE
```

19.5.4 4. Running Partitioned Simulations: *config_runtime.yaml*

Now we can update `config_runtime.yaml` to run FireAxe simulations.

```
target_config:
    topology: fireaxe_rocket_exactmode_config
```

19.6 Running NoC Partition Mode Simulations

In this section, we provide a step-by-step tutorial on how to partition a SoC with a ring NoC containing four tiles into 3 FPGAs connected as a ring. We will run this on the local Xilinx Alveo U250 FPGAs. Note that this feature is *highly experimental and has not been tested on EC2 F1*. This assumes that you have read the *FireAxe running fast-mode simulations*.

Some caveats of this mode:

- You have to *disable TracerV* by mixing in the `WithNoTraceIO` configuration fragment to the `TARGET_CONFIG`.
- Works only for mesh or ring based NoC topologies where each FPGA is connected to exactly 2 other FPGAs.

19.6.1 1. Building Partitioned Sims: Setting up FireAxe Target configs

Similarly to the *fast-mode*, we need to setup the FireAxe target configurations. One change to note is that we are now using the `WithQSFP` configuration fragment. With this information, FireAxe will generate an Aurora IP in the FPGA shell to exchange data between multiple FPGAs using the QSFP interconnects. Also, note that we added the `WithFireAxeNoCPart`. This tells the compiler to perform the NoC-partition pass by grouping router nodes and modules connected to those router nodes. The indices in `WithPartitionGlobalInfo` now indicates the router node indices.

```
//////////////////////////////////////////////////////////////////////////////
// - Xilinx U250 partition a ring NoC onto 3 FPGA connected as a ring
//
//        FPGA 0     ------------------ FPGA 1
```

(continues on next page)

(continued from previous page)

```
// - router 0 & tile 0                 - router 2 & tile 2
// - router 1 & tile 1                 - router 3 & tile 3
//                 \                      /
//                  \                    /
//                   \                  /
//                    \                /
//                          FPGA 2
//                   - router nodes 4 ~ 10
//                   - SoC subsystem
//////////////////////////////////////////////////////////////////////////////
class QuadTileRingNoCTopoQSFPXilinxAlveoConfig
    extends Config(
      new WithQSFP ++             // FireSim will instantiate Aurora IPs
        new WithFireAxeNoCPart ++ // Compiler will detect NoC router nodes to partition
        new WithPartitionGlobalInfo(
          Seq(
            Seq("0", "1"),
            Seq("2", "3"),
            // base group has to be put last
            (4 until 10).map(i => s"${i}"),
          )
        ) ++
        new BaseXilinxAlveoU250Config
    )

class QuadTileRingNoCU250Base
    extends Config(
      new WithPartitionBase ++
        new QuadTileRingNoCTopoQSFPXilinxAlveoConfig
    )

class QuadTileRingNoCU250Partition0
    extends Config(
      new WithPartitionIndex(0) ++
        new QuadTileRingNoCTopoQSFPXilinxAlveoConfig
    )

class QuadTileRingNoCU250Partition1
    extends Config(
      new WithPartitionIndex(1) ++
        new QuadTileRingNoCTopoQSFPXilinxAlveoConfig
    )
```

19.6.2 2. Building Partitioned Sims: *config_build_recipes.yaml*

We can specify the `config_build_recipes.yaml` at this point.

```
###############################################################################
# Using the NoC-partition-mode to partition the design across 3 FPGAs
# connected as a ring.
###############################################################################
# xilinx_u250_quad_rocket_ring_base:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: FireSimQuadRocketSbusRingNoCConfig
#     PLATFORM_CONFIG: QuadTileRingNoCBase
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
#
# xilinx_u250_quad_rocket_ring_0:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: FireSimQuadRocketSbusRingNoCConfig
#     PLATFORM_CONFIG: QuadTileRingNoC0
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
#
# xilinx_u250_quad_rocket_ring_1:
#     ...
#     PLATFORM: xilinx_alveo_u250
#     TARGET_CONFIG: FireSimQuadRocketSbusRingNoCConfig
#     PLATFORM_CONFIG: QuadTileRingNoC1
#     bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml
#     ...
```

19.6.3 3. Running Partitioned Simulations: *user_topology.py*

Once again, we need to specify the FireAxe topology to run simulations.

```
    def fireaxe_ring_noc_config(self) -> None:
        hwdb_entries = {
            0: "xilinx_u250_quad_rocket_ring_0",
            1: "xilinx_u250_quad_rocket_ring_1",
            2: "xilinx_u250_quad_rocket_ring_base",
        }
        slotid_to_pidx = [0, 1, 2]
        # DOC include start: fireaxe_ring_noc_config edges
        edges = [
            FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(2, 1)),
            FireAxeEdge(FireAxeNodeBridgePair(2, 0), FireAxeNodeBridgePair(1, 1)),
            FireAxeEdge(FireAxeNodeBridgePair(1, 0), FireAxeNodeBridgePair(0, 1)),
        ]
        # DOC include end: fireaxe_ring_noc_config edges
        mode = PartitionMode.NOC_MODE
        self.fireaxe_topology_config(hwdb_entries, edges, slotid_to_pidx, mode)
```

Lets look at how the FireAxe topology is set in this example. We can see that bridge 0 of partition `N` is always connected to bridge 1 of partition `(N + 1) % NFPGAs` and bridge 1 of partition `N` is always connected to bridge 0 of partition `(N + NFPGAs - 1) % NFPGAs`.

We can specify the above topology as below:

```
        edges = [
            FireAxeEdge(FireAxeNodeBridgePair(0, 0), FireAxeNodeBridgePair(2, 1)),
            FireAxeEdge(FireAxeNodeBridgePair(2, 0), FireAxeNodeBridgePair(1, 1)),
            FireAxeEdge(FireAxeNodeBridgePair(1, 0), FireAxeNodeBridgePair(0, 1)),
        ]
```

19.6.4 4. Running Partitioned Simulations: *config_runtime.yaml*

Now we can update `config_runtime.yaml` to run FireAxe simulations.

```
target_config:
    topology: fireaxe_rocket_ring_noc_config
```

19.7 Miscellaneous Tips

19.7.1 Running FireAxe Metasims

Users can run FireAxe metasims(*Debugging & Testing with Metasimulation*) by setting the `metasimulation` mapping in `config_runtime.yaml`. Metasimulations are only supported for partitioned on-premise FPGA simulations using QSFP ports.

CHAPTER

TWENTY

MISCELLANEOUS TIPS

20.1 Add the `fsimcluster` column to your AWS management console

Once you've deployed a simulation once with the manager, the AWS management console will allow you to add a custom column that will allow you to see at-a-glance which FireSim run farm an instance belongs to.

To do so, click the gear in the top right of the AWS management console. From there, you should see a checkbox for `fsimcluster`. Enable it to see the column.

20.2 FPGA Dev AMI Remote Desktop Setup

To Remote Desktop into your manager instance, you must do the following:

```
curl https://s3.amazonaws.com/aws-fpga-developer-ami/1.5.0/Scripts/setup_gui.sh -o /home/
↪centos/src/scripts/setup_gui.sh
sudo sed -i 's/enabled=0/enabled=1/g' /etc/yum.repos.d/CentOS-CR.repo
/home/centos/src/scripts/setup_gui.sh
# keep manager paramiko compatibility
sudo pip2 uninstall gssapi
```

See

https://forums.aws.amazon.com/message.jspa?messageID=848073#848073

and

https://forums.aws.amazon.com/ann.jspa?annID=5710

20.3 Experimental Support for SSHing into simulated nodes and accessing the internet from within simulations

This is assuming that you are simulating a 1-node networked cluster. These instructions will let you both ssh into the simulated node and access the outside internet from within the simulated node:

1. Set your config files to simulate a 1-node networked cluster (`example_1config`)
2. Run `firesim launchrunfarm && firesim infrasetup` and wait for them to complete
3. cd to `firesim/target-design/switch/`
4. Go into the newest directory that is prefixed with `switch0-`

5. Edit the `switchconfig.h` file so that it looks like this:

```
// THIS FILE IS MACHINE GENERATED. SEE deploy/buildtools/switchmodelconfig.py

#ifdef NUMCLIENTSCONFIG
#define NUMPORTS 2
#define NUMDOWNLINKS 2
#define NUMUPLINKS 0
#endif
#ifdef PORTSETUPCONFIG
ports[0] = new ShmemPort(0,
↪"0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪", false);
ports[1] = new SSHPort(1);

#endif

#ifdef MACPORTSCONFIG
uint16_t mac2port[3]  {1, 2, 0};
#endif
```

6. Run `make` then `cp switch switch0`
7. Run `scp switch0 YOUR_RUN_FARM_INSTANCE_IP:switch_slot_0/switch0`
8. On the RUN FARM INSTANCE, run:

```
sudo ip tuntap add mode tap dev tap0 user $USER
sudo ip link set tap0 up
sudo ip addr add 172.16.0.1/16 dev tap0
sudo ifconfig tap0 hw ether 8e:6b:35:04:00:00
sudo sysctl -w net.ipv6.conf.tap0.disable_ipv6=1
```

9. Run `firesim runworkload`. Confirm that the node has booted to the login prompt in the fsim0 screen.
10. To ssh into the simulated machine, you will need to first ssh onto the Run Farm instance, then ssh into the IP address of the simulated node (172.16.0.2), username `root`. You should also prefix with TERM=linux to get backspace to work correctly: So:

```
ssh YOUR_RUN_FARM_INSTANCE_IP
# from within the run farm instance:
TERM=linux ssh root@172.16.0.2
```

11. To also be able to access the internet from within the simulation, run the following on the RUN FARM INSTANCE:

```
sudo sysctl -w net.ipv4.ip_forward=1
export EXT_IF_TO_USE=$(ifconfig -a | sed 's/[ \t].*//;/^\(lo:\|\)$/d' | sed 's/[ \t].*//;
↪/^\(tap0:\|\)$/d' | sed 's/://g')
sudo iptables -A FORWARD -i $EXT_IF_TO_USE -o tap0 -m state --state RELATED,ESTABLISHED -
↪j ACCEPT
sudo iptables -A FORWARD -i tap0 -o $EXT_IF_TO_USE -j ACCEPT
sudo iptables -t nat -A POSTROUTING -o $EXT_IF_TO_USE -j MASQUERADE
```

12. Then run the following in the simulation:

```
route add default gw 172.16.0.1 eth0
echo "nameserver 8.8.8.8" >> /etc/resolv.conf
echo "nameserver 8.8.4.4" >> /etc/resolv.conf
```

At this point, you will be able to access the outside internet, e.g. `ping google.com` or `wget google.com`.

20.4 Navigating the FireSim Codebase

This is a large codebase with tons of dependencies, so navigating it can be difficult. By default, a `tags` file is generated when you run `./build-setup.sh` which aids in jumping around the codebase. This file is generated by Exuberant Ctags and many editors support using this file to jump around the codebase. You can also regenerate the `tags` file if you make code changes by running `./gen-tags.sh` in your FireSim repo.

For example, to use these tags to jump around the codebase in `vim`, add the following to your `.vimrc`:

```
set tags=tags;/
```

Then, you can move the cursor over something you want to jump to and hit `ctrl-]` to jump to the definition and `ctrl-t` to jump back out. E.g. in top-level configurations in FireSim, you can jump all the way down through the Rocket Chip codebase and even down to Chisel.

20.5 Using FireSim CI

For more information on how to deal with the FireSim CI and how to run FPGA simulations in the CI, refer to the the `CI_README.md` under the `.github/` directory.

20.6 How to view AWS build logs when AGFI build fails

There are times when we want to view Vivado logs when a bitstream build fails (especially when a bitstream build fails while the manager is printing out `pending`). The AWS AGFI creation backend stores these logs in storage servers called S3 buckets. The following steps will guide you on how to copy these logs from the bucket to your manager instance:

1. Go to your AWS console.
2. Select "View all services".
3. Under "Storage", select "S3".
4. On the left panel, select "Buckets".
5. Now select the bucket that you created.

See https://docs.aws.amazon.com/AmazonS3/latest/userguide/access-bucket-intro.html for the bucket naming style. The bucket name is defined in `firesim/deploy/bit-builder-recipes/f1.yaml`

6. Under "logs/agfi-<somenumber>", you will see "<date and time>_vivado.log". Select it and copy the S3 URI.
7. Now, go back to your manager instance and run `aws s3 cp <URI that you just copied> some_descriptive_name.log`.

Now you should be able to view the Vivado log using your favorite text editor.

CHAPTER

TWENTYONE

ADDING SUPPORT FOR A NEW FPGA

To use FireSim, an FPGA, at minimum, needs to provide an MMIO (i.e. 32b AXI4-Lite) interface that can interact with a C++ driver. This MMIO interface is used to coordinate the simulation. However, for all FireSim features, an FPGA needs to also provide a DMA (i.e. 512b AXI4) interface that can interact with a C++ driver and DRAM (also AXI4). While optional, DMA and DRAM provide extra features of FireSim such as TracerV and DRAM for the FASED DDR memory model.

In the case of Xilinx FPGAs, both the MMIO and DMA interface are provided by XDMA. XDMA allows a C++ driver to send data to/from the FPGA.

The following sections talk about the different changes needed to support a new FPGA using the Xilinx Alveo U250 as an example. When creating new folders and files, rename `xilinx_alveo_u250` to your specific FPGA name. We will use this name across various files as the "platform" name. For each of these sections, feel free to copy/modify them to your new FPGA.

Note

FPGAs in FireSim, when first developed, start with implementing/testing the AXI4-Lite interface before moving to add the DMA interface and DRAM. We highly recommend you to follow the same flow when adding an FPGA.

Warning

This documentation is new. Feel free to make any PRs or give any feedback to make this easier to read and understand. Additionally, if a new FPGA works, feel free to PR it to the FireSim mainline repo. Thanks!

21.1 Adding a new FireSim `platform`

FireSim first needs to understand how to build a bitstream for the FPGA wanted using a synthesis tool like Vivado. This code is held in platforms.

Let's take a look at platforms/xilinx_alveo_u250. At the top-level is platforms/xilinx_alveo_u250/build-bitstream.sh which invokes Vivado in a specific directory called `CL_DIR` (custom logic directory) to build the bitstream wanted. This serves as the interface for the FireSim manager to modify a bitstream build (i.e. change the frequency of a design, pass synthesis options from the YAML, etc).

platforms/xilinx_alveo_u250/cl_firesim holds all RTL, TCL, and more needed to build a bitstream for a specific FPGA. Typically, during `firesim buildbitstream`, FireSim's build system copies this folder to a new location on the manager machine with a unique name (i.e. the configuration quintuplet), adds the RTL generated from Golden Gate (i.e. `FireSim-generated.sv` specifying the top-level module), then copies it to the build farm machine/instance. Then

the manager copies the platforms/xilinx_alveo_u250/build-bitstream.sh to the build farm machine/instance and calls it with the path to the build farm machine/instance `CL_DIR`.

Within platforms/xilinx_alveo_u250/cl_firesim you'll see an RTL top-level file that instantiates the top-level FireSim module called `F1Shim` (in this case the name `F1Shim` is because this top-level module is virtually the same as the AWS equivalent) and connects both 32b/512b AXI4 buses from the FPGA top-level to the FireSim module. An MMIO-only (32b AXI4-Lite-only) design can just tieoff the DMA interface and DRAM to begin with.

The various scripts in the platforms/xilinx_alveo_u250 area are for flashing the FPGA, building the bitstream, etc.

Most likely you can copy this platform and modify it to your new FPGA if you are running an XDMA-enabled FPGA.

Note

Unlike the AWS version of the equivalent platform, the Xilinx Alveo U250 platform creates a full bitstream that flashes the entire FPGA. The scripts that reside in the Xilinx Alveo U250 platform also work around an issue related to this where you need PCIe to always be functioning (even when flashing the FPGA). The AWS equivalent doesn't need this extra scripting since it uses a shell to selectively program all parts that aren't associated with PCIe, resulting in the PCIe link always being up.

21.2 Adding other collateral (Scala, C++, Make, etc)

Next, you'll need to tell the FireSim build system (i.e. Make) how to build the top-level FireSim module copied into the `CL_DIR` mentioned above. Additionally, you'll also tell FireSim how to build a corresponding C++ driver for simulation.

First, you'll need to add new Scala configurations to tell Golden Gate there is a new FPGA. This can be done in sim/midas/src/main/scala/midas/Config.scala and sim/midas/src/main/scala/configs/CompilerConfigs.scala. These configs indicate what the FireSim top-level is (i.e. `F1Shim`), what its AXI4 parameters are for ports, what ports are available (of what type), what DRAM is available, etc.

Next, you'll need to provide a C++ interface that allows FireSim to read/write to the FPGA's MMIO (AXI4-Lite) and DMA (AXI4) port through XDMA. An example of doing this is sim/midas/src/main/cc/simif_xilinx_alveo_u250.cc. This file implements functions like `fpga_setup`, `read`, `write`, `cpu_managed_axi4_read`, and `cpu_managed_axi4_write` which correspond to setting up the XDMA interfaces, and MMIO and DMA read/writes.

Next, you'll need to add a hook to FireSim's make system to build the FPGA RTL and also build the C++ driver with the given `simif_*` file. This is done in sim/make/fpga.mk and sim/make/driver.mk. For most cases, you can copy/paste and follow along with the `xilinx_alveo_u250` example (if you are building a driver that only depends on Conda and doesn't depend on system C++ libraries).

At this point you should be able to build the RTL using something like `make -C sim PLATFORM=xilinx_alveo_u250 xilinx_alveo_u250` where you can replace `xilinx_alveo_u250` with your FPGA platform name. This should build both the C++ driver and the RTL associated with it that is copied for synthesis.

21.3 Manager build modifications

Next, you'll need to tell the FireSim manager a new platform exists to use it in `firesim buildbitstream`.

First, we need to add a "bit builder" class that gives the Python code necessary to build and synthesize the RTL on a build farm instance/machine and copy the results back into a FireSim HWDB entry. This code is located in deploy/buildtools/bitbuilder.py. In summary, this class should implement the `build_bitstream`, and `setup` methods from `BitBuilder`. In the Xilinx Alveo U250 case, the `build_bitstream` function builds a bitstream by doing the following in Python:

1. Creates a copy of the `platform` area previously described on the build farm machine/instance
2. Adds the RTL built with the `make` command from the prior section to that copied area (i.e. `CL_DIR`)
3. Runs the platforms/xilinx_alveo_u250/build-bitstream.sh script with the copied area.
4. Retrieves the bitstream built and compiles a `*.tar.gz` file with it. Uses that file in a HWDB entry.

Next, since this class can take arguments from FireSim's YAML, you'll need to add a YAML file for a new FPGA in deploy/bit-builder-recipes (even if it has no args).

Now you can build a bitstream using the FireSim manager by changing the build recipe arguments (i.e. `PLATFORM`, `PLATFORM_CONFIG`, `bit_builder_recipe`). For example, here is a `xilinx_alveo_u250` recipe:

```
custom_recipe:
    ...
    PLATFORM: xilinx_alveo_u250 # platform name
    PLATFORM_CONFIG: BaseXilinxAlveoU250Config # config added for platform
    bit_builder_recipe: bit-builder-recipes/xilinx_alveo_u250.yaml # extra yaml file␣
↪given earlier
    ...
```

21.4 Manager run modifications

Next, you'll need to tell the FireSim manager a new platform exists to use it run simulation commands like `firesim runworkload`.

First, for convenience, you'll need to indicate a new platform exists by adding it in deploy/firesim. This modifies the YAML files when running `firesim managerinit` to have the right values initially.

Finally, you'll need to add an "instance deploy manager" to tell the FireSim manager how to flash an FPGA, start a simulation, and more. This is seen in deploy/runtools/run_farm_deploy_managers.py. Typically, FPGAs need to implement the `infrasetup_instance` method of `InstanceDeployManager` telling a run farm machine how to flash an FPGA. These Python steps create a simulation directory on the run farm machine/instance, copy simulation collateral to it (including the bitstream), and flash the FPGA.

Now you should be able to run a simulation with the given bitstream by pointing to your specific instance deploy manager and bitstream that was built. For example, in the Xilinx Alveo U250 case, in the `config_runtime.yaml` you can modify the `default_platform` to be `XilinxAlveoU250InstanceDeployManager` and change the HWDB entry to be the recipe built for the new FPGA.

CHAPTER TWENTYTWO

USING FIRESIM WITHOUT CHIPYARD

FireSim is now standalone allowing (1) FireSim developers to test the repository without Chipyard and (2) allowing non-Chipyard top-level projects to integrate FireSim as a library. We will discuss option (2) in this section.

A non-Chipyard top-level serves as the target which FireSim will simulate. It must provide a few items:

- A Chisel top-level "harness" to connect FireSim bridges to drive things like the clock and reset.
- A C++ top-level simulation driver to indicate how to progress in the simulation.
- A series of Make fragments to configure the FireSim build system (otherwise called MIDAS or Golden Gate).

We name this set of sources a "project".

22.1 Simple Counter Example Project

By default, FireSim provides a simple example on how to add your own RTL can create a simulator with just a clock and reset. This serves as the starting point for users to add future bridges or more complicated designs. Throughout this section you will see the name `examples` at the end of paths, this is the FireSim "project" that we are using.

22.1.1 Top-Level Harness

sim/src/main/scala/examples/SimpleCounter.scala holds the simple top-level harness which wraps around a simple counter that increments to 1000.

Looking at the counter module, it outputs a `done` signal when the counter reaches 1000.

```
// Simple module that when started (i.e. after reset) counts to 1000 then signals 'done'
class SimpleCounter extends Module {
  val io = IO(new Bundle {
    val done = Output(Bool())
  })

  val cnt = RegInit(0.U(16.W))

  when (cnt === 1000.U) {
    io.done := true.B
  } .otherwise {
    io.done := false.B
    cnt := cnt + 1.U
  }
```

(continues on next page)

(continued from previous page)

```
  when (cnt % 100.U === 0.U && cnt =/= 1000.U) {
    printf("Counter reached %d\n", cnt)
  }
}
```

To simulate this module, we need to wrap it in test harness that will source/sink it's IOs and will also drive the reset and clock of the module. This is shown below:

```
// Simple example harness that runs a simulation. This harness does not terminate the␣
↪simulation,
// instead it is the job of the C++ top-level to terminate the simulation.
class SimpleCounterHarness(implicit val p: Parameters) extends RawModule {
// DOC include start: ClockResetWire
  val clock = Wire(Clock())
  val reset = Wire(Bool())
// DOC include end: ClockResetWire

  // Boilerplate code:
  // The peek-poke bridge must still be instantiated even though it's
  // functionally unused. This will be removed in a future PR.
  val dummy = WireInit(false.B)
  val peekPokeBridge = PeekPokeBridge(clock, dummy)

// DOC include start: Bridges
  // Drive with default clock provided by a bridge.
  clock := RationalClockBridge().io.clocks.head

  // Drive reset with a bridge.
  val resetBridge = Module(new ResetPulseBridge(ResetPulseBridgeParameters()))
  // In effect, the bridge counts the length of the reset in terms of this clock.
  resetBridge.io.clock := clock
  // Drive with pulsed reset for a default amount of time.
  reset := resetBridge.io.reset
// DOC include end: Bridges

  // Boilerplate code:
  // Ensures FireSim-synthesized assertions and instrumentation is disabled
  // while 'resetBridge.io.reset' is asserted.  This ensures assertions do not fire at
  // time zero in the event their local reset is delayed (typically because it
  // has been pipelined).
  GlobalResetCondition(resetBridge.io.reset)

// DOC include start: CL
  // Custom logic.
  withClockAndReset(clock, reset) {
    val simpleCounter = Module(new SimpleCounter)

    // Print once when counter 'done' signal asserted.
    val printDone = RegInit(false.B)
    when (simpleCounter.io.done && !printDone) {
      printDone := true.B
      printf("Counter has completed!\n")
```

(continues on next page)

(continued from previous page)

```
    }
  }
// DOC include end: CL
}
```

First, we create a top-level `clock` and `reset` wire that is used for the simple counter module. This is shown here:

```
  val clock = Wire(Clock())
  val reset = Wire(Bool())
```

Next, we connect those `clock` and `reset` wires to two corresponding bridges that can drive the `clock` and `reset` for the simulation. In this case, we use the `RationalClockBridge` and the `ResetPulseBridge` which run the simulation on one clock domain and reset the simulation. In more complex cases, these bridges can be used to drive multi-clock simulations or drive a reset pulse for a longer period of time. This is shown here:

```
  // Drive with default clock provided by a bridge.
  clock := RationalClockBridge().io.clocks.head

  // Drive reset with a bridge.
  val resetBridge = Module(new ResetPulseBridge(ResetPulseBridgeParameters()))
  // In effect, the bridge counts the length of the reset in terms of this clock.
  resetBridge.io.clock := clock
  // Drive with pulsed reset for a default amount of time.
  reset := resetBridge.io.reset
```

Finally, we need to instantiate our simple counter and wire up it's IOs. This is done here:

```
  // Custom logic.
  withClockAndReset(clock, reset) {
    val simpleCounter = Module(new SimpleCounter)

    // Print once when counter 'done' signal asserted.
    val printDone = RegInit(false.B)
    when (simpleCounter.io.done && !printDone) {
      printDone := true.B
      printf("Counter has completed!\n")
    }
  }
```

Since we are creating logic within a Chisel `RawModule` we need to indicate that the `SimpleCounter` and registers we are using have a default `clock` and `reset`. This is done with the `withClockAndReset` scope. Also note that this RTL just prints, we will use the C++ top-level to terminate the simulation by timing out in the next section.

22.1.2 C++ Driver Top

sim/src/main/cc/examples/simple_counter_top.cc defines the C++ top-level simulation driver called `simple_counter_top_t` for the simulation. It is in charge of adding any extra widgets/bridges, determining how to step the simulation, and terminating. Most of this file is boilerplate code (i.e. code that can be copied from the example), but two sections are highlighted here.

First, we need to define a core simulation loop. This loop is in charge of managing the simulation and indicating when bridges need to run their logic. This is shown here:

```
int simple_counter_top_t::simulation_run() {
  int exit_code = 0;
  // infinite loop until '+max-cycles' value is reached (within 'systematic_scheduler_t')
  while (!terminated && !finished_scheduled_tasks()) {
    // step forward maximum amount of allowable cycles
    peek_poke.step(get_largest_stepsize(), false);
    // while the simulation is running N cycles, run all simulation bridges
    while (!peek_poke.is_done() && !terminated) {
      for (auto *bridge : registry.get_all_bridges()) {
        // do bridge work
        bridge->tick();
        // if a bridge has finished then fully exit
        if (bridge->terminate()) {
          exit_code = bridge->exit_code();
          terminated = true;
          break;
        }
      }
    }
  }
  return exit_code;
}
```

You can see things like `peek_poke.step` being called to "step" forward in the simulation, `bridge->tick()` to run bridge logic and more. This loop is terminated after N cycles which is given adding `+max-cycles=N` to the simulator binary (this is defined in the `systematic_scheduler_t` class).

Second, we need to register the `simple_counter_top_t` class as the main simulation driver class in the default FireSim main function. This is done here:

```
// used in firesim's 'main' to instantiate the custom C++ class you want for a simulation.
// in this case our 'simple_counter_top_t'
std::unique_ptr<simulation_t>
create_simulation(simif_t &simif,
                  widget_registry_t &registry,
                  const std::vector<std::string> &args) {
  return std::make_unique<simple_counter_top_t>(simif, registry, args);
}
```

This code simply creates a unique pointer to the simulation class you want (in this case `simple_counter_top_t`) in a function that is called in FireSim's main function.

22.1.3 Make fragments

Next, you need to provide a series of Make fragments to configure the FireSim build system to generate the RTL to run with Golden Gate. This is located in sim/src/main/makefrag/examples.

This area consists of four Make fragments that indicate how to build/run/configure the project:

- sim/src/main/makefrag/examples/build.mk - Target-RTL generation
- sim/src/main/makefrag/examples/config.mk - Variable defaults
- sim/src/main/makefrag/examples/driver.mk - MIDAS/Golden-Gate sources
- sim/src/main/makefrag/examples/metasim.mk - Top-level meta-simulator targets

Starting with the sim/src/main/makefrag/examples/build.mk, this file specifies a rule to build the `FIRRTL_FILE` and `ANNO_FILE` files needed for downstream FireSim Make rules. This FIRRTL file needs to be a Chisel 3.6 (i.e. Scala FIRRTL Compiler) compatible FIRRTL file. In this case, we reuse the Chisel generator binary (i.e. `midas.chiselstage.Generator`) for RTL generation since all the Scala sources are defined in the FireSim top-level sim/build.sbt

Next is sim/src/main/makefrag/examples/config.mk. This file provides capitalized variables used throughout the FireSim Make system. This is set to sensible defaults but each of these variables can be overridden on the make command line (i.e. `make TARGET_CONFIG=... ...`). In this case, we point to our simulation top-level module `SimpleCounterHarness`.

Next is sim/src/main/makefrag/examples/driver.mk . This file provides the `DRIVER_H`, `DRIVER_CC`, `TARGET_CXX_FLAGS`, and `TARGET_LD_FLAGS` needed for the FireSim Make system to build a C++ driver for the simulation. In this case, we point to our `simple_counter_top.cc` file that we created earlier.

Finally, the sim/src/main/makefrag/examples/metasim.mk file provides Make targets for running metasimulations through the FireSim MakeFile. You can use this to add targets to run your target with any simulator, or whatever else. In this case, we define simulation targets for Verilator and VCS that indicate to the simulation that it should timeout after 10000 cycles.

22.1.4 Running meta-simulations and more

Once these Make fragments are added, you can then run the FireSim MakeFile or build FireSim recipes by invoking the Make system with `TARGET_PROJECT_MAKEFRAG` pointing to the Make fragments (i.e. `make TARGET_PROJECT_MAKEFRAG=<PATH/TO/MAKEFRAG/FOLDER> ...`). In this case, since the files reside within FireSim, the `TARGET_PROJECT_MAKEFRAG` will be properly set to `sim/src/main/makefrag/<TARGET_PROJECT>` so we just need to define the `TARGET_PROJECT` to be `examples`.

The following code runs a metasimulation with VCS for our example RTL test harness, C++ code, and make fragments.

```
cd $FS_DIR
source sourceme-manager.sh --skip-ssh-setup
make TARGET_PROJECT=examples run-vcs
```

22.2 Chipyard Example

For the remainder of this section we will use Chipyard as an example of how to integrate FireSim into a top-level project. In the future, we will provide a simplified example non-Chipyard top-level setup that users can reference.

22.2.1 Top-Level Harness

An example of a FireSim top-level harness is in ${CY_DIR}/generators/firechip/chip/src/main/scala/FireSim.scala. While there are alot of extra Chipyard specific features, the main focus should be on adding a `ResetPulseBridge` to drive the top-level reset of the system, and adding a `RationalClockBridge` to drive system clocks. Then the harness can choose to instantiate any other target-specific bridges (i.e. the FASED DRAM model or a UART bridge for example).

22.2.2 C++ Driver Top

Next, you need to provide a top-level C++ driver such as ${CY_DIR}/generators/firechip/chip/src/main/cc/firesim/firesim_top.cc. This indicates how the bridges should be run, and when.

22.2.3 Make fragments

Next, you need to provide a series of Make fragments to configure the FireSim build system to generate the RTL to run with Golden Gate. Chipyard's example is here: ${CY_DIR}/generators/firechip/chip/src/main/makefrag/firesim.

Starting with the `build.mk`, this file specifies a rule to build the `FIRRTL_FILE` and `ANNO_FILE` files needed for downstream FireSim Make rules. This FIRRTL file needs to be a Chisel 3.6 (i.e. Scala FIRRTL Compiler) compatible FIRRTL file. In Chipyard's case, the Chipyard MakeFile is invoked always (due to the dependency on a `.PHONY` target) to create the two files. However, Chipyard's MakeFile doesn't update the files if nothing has changed preventing downstream FireSim Make rules from re-running. Additionally, the file has a variable `TARGET_COPY_TO_MIDAS_SCALA_DIRS` this allows the `firesim_target_symlink_hook` target to symlink target-specific bridge directories into MIDAS to compile (in this case, `bridgeinterfaces` and `golgengateimplementations`). If you are using the default bridges (i.e. `ResetPulseBridge` and `RationalClockBridge`) then this variable and target shouldn't be needed.

Next is `config.mk`. This file provides capitalized variables used throughout the FireSim Make system. This also provides Make variables that can be visible from the other `*.mk` files in this directory (like `chipyard_dir`).

Next is `driver.mk`. This file provides the `DRIVER_H`, `DRIVER_CC`, `TARGET_CXX_FLAGS`, and `TARGET_LD_FLAGS` needed for the FireSim Make system to build a C++ driver for the simulation. When using default bridges, you should just need to add your `firesim_top.cc` file to the `DRIVER_CC`. Additionally, you might need to add to `TARGET_CXX_FLAGS` an include path to the generated directory (i.e. `-I$(GENERATED_DIR)`).

Finally, the `metasim.mk` file provides Make targets for running metasimulations through the FireSim MakeFile. You can use this to add targets to run your target with any simulator, or whatever else.

Once these Make fragments are added, you can then run the FireSim MakeFile or build FireSim recipes by invoking the Make system with `TARGET_PROJECT_MAKEFRAG` pointing to the Make fragments (like ${CY_DIR}/generators/firechip/chip/src/main/makefrag/firesim).

CHAPTER

TWENTYTHREE

FIRESIM ASKED QUESTIONS

23.1 I just bumped the FireSim repository to a newer commit and simulations aren't running. What is going on?

Anytime there is an AGFI bump, FireSim simulations will break/hang due to outdated AFGI. To get the new default AGFI's you must run the manager initialization again by doing the following:

```
cd firesim
source sourceme-manager.sh
firesim managerinit
```

23.2 Is there a good way to keep track of what AGFI corresponds to what FireSim commit?

When building an AGFI during `firesim buildbitstream`, FireSim keeps track of what FireSim repository commit was used to build the AGFI. To view a list of AGFI's that you have built and what you have access to, you can run the following command:

```
cd firesim
source sourceme-manager.sh
aws ec2 describe-fpga-images --fpga-image-ids # List all AGFI images
```

You can also view a specific AGFI image by giving the AGFI ID (found in `deploy/config_hwdb.yaml`) through the following command:

```
cd firesim
source sourceme-manager.sh
aws ec2 describe-fpga-images --filter Name=fpga-image-global-id,Values=agfi-<Your ID␣
↪Here> # List particular AGFI image
```

After querying an AGFI, you can find the commit hash of the FireSim repository used to build the AGFI within the "Description" field.

For more information, you can reference the AWS documentation at https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-fpga-images.html.

23.3 Help, My Simulation Hangs!

Oof. It can be difficult to pin this one down, read through *Debugging a Hanging Simulator* for some tips to get you started.

23.4 Should My Simulator Produce Different Results Across Runs?

No.

Unless you've intentionally introduced a side-channel (e.g., you're running an interactive simulation, or you've connected the NIC to the internet), this is likely a bug in one of your custom bridge implementations or in FireSim. In fact, for a given target-design, enabling printf synthesis, assertion synthesis, autocounter, or Auto ILA, should not change the simulated behavior of the machine.

23.5 Is there a way to compress workload results when copying back to the manager instance?

FireSim doesn't support compressing workload results before copying them back to the manager instance. Instead we recommend that you use a modern filesystem (like ZFS) to provide compression for you. For example, if you want to use ZFS to transparently compress data:

1. Attach a new volume to your EC2 instance (either at runtime or during launch). This is where data will be stored in a compressed format.
2. Make sure that the volume is attached (using something like `lsblk -f`). This new volume should not have a filesystem type and should be unmounted (volume name example: `nvme1n1`).
3. Install ZFS according to the ZFS documentation. Check `/etc/redhat-release` to verify the CentOS version of the manager instance.
4. Mount the volume and setup the ZFS filesystem with compression.

> **Warning**
>
> Creating the zpool will destroy all pre-existing data on that partition. Double-check that the device node is correct before running any commands.

```
# replace /dev/nvme1n1 with the proper device node
zpool create -o ashift=12 -O compression=on <POOL_NAME> /dev/nvme1n1
zpool list
zfs list
```

5. At this point, you can use `/<POOL_NAME>` as a normal directory to store data into where it will be compressed. To see the compression ratio, use `zfs get compressratio`.

CHAPTER

TWENTYFOUR

OVERVIEW & PHILOSOPHY

Underpinning FireSim is Golden Gate (MIDAS II), a FIRRTL-based compiler and C++ library, which is used to transform Chisel-generated RTL into a deterministic FPGA-accelerated simulator.

24.1 Golden Gate vs FPGA Prototyping

Key to understanding the design of Golden Gate, is understanding that Golden Gate-generated simulators are not FPGA prototypes. Unlike in a prototype, Golden Gate-generated simulators decouple the target-design clocks from all FPGA-host clocks (we say it is *host-decoupled*): one cycle in the target machine is simulated over a dynamically variable number FPGA clock cycles. In constrast, a conventional FPGA-prototype "emulates" the SoC by implementing the target directly in FPGA logic, with each FPGA-clock edge executing a clock edge of the SoC.

24.2 Why Use Golden Gate & FireSim

The host decoupling by Golden Gate-generated simulators enables:

1. **Deterministic simulation** Golden Gate creates a closed simulation environment such that bugs in the target can be reproduced despite timing-differences (eg. DRAM refresh, PCI-E transport latency) in the underlying host. The simulators for the same target can be generated for different host-FPGAs but will maintain the same target behavior.
2. **FPGA-host optimizations** Structures in ASIC RTL that map poorly to FPGA logic can be replaced with models that preserve the target RTL's behavior, but take more host cycles to save resources. eg. A 5R, 3W-ported register file with a dual-ported BRAM over 4 cycles.
3. **Distributed simulation & software co-simulation** Since models are decoupled from host time, it becomes much easier to host components of the simulator on multiple FPGAs, and on a host-CPU, while still preserving simulation determinism. This feature serves as the basis for building cycle-accurate scale-out systems with FireSim.
4. **FPGA-hosted, timing-faithful models of I/O devices** Most simple FPGA-prototypes use FPGA-attached DRAM to model the target's DRAM memory system. If the available memory system does not match that of the target, the target's simulated performance will be artificially fast or slow. Host-decoupling permits writing detailed timing models that provide host-independent, deterministic timing of the target's memory system, while still use FPGA-host resources like DRAM as a functional store.

24.3 Why Not Golden Gate

Ultimately, Golden Gate-generated simulators introduce overheads not present in an FPGA-prototype that *may* increase FPGA resource use, decrease fmax, and decrease overall simulation throughput[1] . Those looking to develop soft-cores or develop a complete FPGA-based platform with their own boards and I/O devices would be best served by implementing their design directly on an FPGA. For those looking to building a system around Rocket-Chip, we'd suggest looking at SiFive's Freedom platform to start.

24.4 How is Host-Decoupling Implemented?

Host-decoupling in Golden Gate-generated simulators is implemented by decomposing the target machine into a dataflow graph of latency-insensitive models. As a user of FireSim, understanding this dataflow abstraction is essential for debugging your system and for developing your own software models and bridges. We describe it in the next section.

[1] These overheads varying depending on the features implemented and optimizations applied. Certain optimizations, currently in development, may increase fmax or decrease resource utilization over the equivalent prototype.

CHAPTER

TWENTYFIVE

TARGET ABSTRACTION & HOST DECOUPLING

Golden Gate-generated simulators are deterministic, cycle-exact representations of the source RTL fed to the compiler. To achieve this, Golden Gate consumes input RTL (as FIRRTL) and transforms it into a latency-insensitive bounded dataflow network (LI-BDN) representation of the same RTL.

25.1 The Target as a Dataflow Graph

Dataflow graphs in Golden Gate consist of models, tokens, and channels:

1. Models – the nodes of the graph, these capture the behavior of the target machine by consuming and producing tokens.
2. Tokens – the messages of dataflow graph, these represent a hardware value as they would appear on a wire after they have converged for a given cycle.
3. Channels – the edges of the graph, these connect the output port of one model to the input of another.

In this system, time advances locally in each model. A model advances once cycle in simulated time when it consumes one token from each of its input ports and enqueues one token into each of its output ports. Models are *latency-insensitive*: they can tolerate variable input token latency as well as backpressure on output channels. Give a sequence of input tokens for each input port, a correctly implemented model will produce the same sequence of tokens on each of its outputs, regardless of when those input tokens arrive.

We give an example below of a dataflow graph representation of a 32-bit adder, simulating two cycles of execution.

25.2 Model Implementations

In Golden Gate, there are two dimensions of model implementation:

1) CPU- or FPGA-hosted: simply, where the model is going to execute. CPU-hosted models, being software, are more flexible and easy to debug but slow. Conversely, FPGA-hosted models are fast, but more difficult to write and debug.

2) Cycle-Exact or Abstract: cycle-exact models faithfully implement a chunk of the SoC's RTL~(this formalized later), where as abstract models are handwritten and trade fidelity for reduced complexity, better simulation performance, improved resource utilization, etc…

Hybrid, CPU-FPGA-hosted models are common. Here, a common pattern is write an RTL timing-model and a software functional model.

25.3 Expressing the Target Graph

The target graph is captured in the FIRRTL for your target. The bulk of the RTL for your system will be transformed by Golden Gate into one or more cycle-exact, FPGA-hosted models. You introduce abstract, FPGA-hosted models and CPU-hosted models into the graph by using Target-to-Host Bridges. During compilation, Golden Gate extracts the target-side of the bridge, and instantiates your custom RTL, called an BridgeModule, which together with a CPU-hosted Bridge Driver, gives you the means to model arbitrary target-behavior. We expand on this in the Bridge section.

25.4 Latency-Insensitive Bounded Dataflow Networks

In order for the resulting simulator to be a faithful representation of the target RTL, models must adhere to three properties. We refer the reader to the LI-BDN paper for the formal definitions of these properties. English language equivalents follow.

Partial Implementation: The model output token behavior matches the cycle-by-cyle output of the reference RTL, given the same input provided to both the reference RTL and the model (as a arbitrarily delayed token stream). Cycle exact models must implement PI, whereas abstract models do not.

The remaining two properties ensure the graph does not deadlock, and must be implemented by both cycle-exact and abstract models.

Self-Cleaning: A model that has enqueued N tokens into each of it's output ports *must* eventually dequeue N tokens from each of it's input ports.

No Extranenous Dependencies: If a given output channel of an LI-BDN simulation model has received a number of tokens no greater than any other channel, and if the model receives all input tokens required to compute the next output token for that channel, the model must eventually enqueue that output token, regardless of future external activity. Here, a model enqueueing an output token is synonymous with the corresponding output channel "receiving" the token.

CHAPTER

TWENTYSIX

TARGET-TO-HOST BRIDGES

A custom model in a FireSim Simulation, either CPU-hosted or FPGA-hosted, is deployed by using a *Target-to-Host Bridge*, or Bridge for short. Bridges provide the means to inject hardware and software models that produce and consume token streams.

Bridges enable:

1. **Deterministic, host-agnostic I/O models.** This is the most common use case. Here you instantiate bridges at the I/O boundary of your chip, to provide a simulation models of the environment your design is executing in. For an FPGA-hosted model, see FASED memory timing models.
2. **Verification against a software golden model.** Attach an bridge (anywhere in your target RTL) to an interface you'd like to monitor, (e.g., a processor trace port). In the host, you can pipe the token stream coming off this interface to a software model running on a CPU (e.g, a functional ISA simulator).
3. **Distributed simulation.** The original FireSim application. You can stitch together networks of simulated machines by instantiating bridges at your SoC boundary. Then write software models and bridge drivers that move tokens between each FPGA.
4. **Resource optimizations.** Resource-intensive components of the target can be replaced with models that use fewer FPGA resources or run entirely in software.

The use of Bridges in a FireSim simulation has many analogs to doing mixed-language (Verilog-C++) simulation of the same system in software. Where possible, we'll draw analogies.

26.1 Terminology

Bridges have a *target side*, consisting of a specially annotated Module, and *host side*, which consists of an FPGA-hosted *bridge module* (deriving from `BridgeModule`) and an optional CPU-hosted *bridge driver* (deriving from `bridge_driver_t`).

In a mixed-language software simulation, a verilog procedural interface (VPI) is analogous to the target side of a bridge, with the C++ backing that interface being the host side.

26.2 Target Side

In your target side, you will mix-in `firesim.lib.bridgeutils.Bridge` into a Chisel `BaseModule` (this can be a black or white-box Chisel module) and implement its abstract members. This trait indicates that the associated module will be replaced with a connection to the host-side of the bridge that sources and sinks token streams. During compilation, the target-side module will be extracted by Golden Gate and its interface will be driven by your bridge's host-side implementation.

This trait has one type parameter and few abstract members you'll need define for your Bridge. Since you must mix `Bridge` into a Chisel `BaseModule`, the IO you define for that module constitutes the target-side interface of your bridge.

26.2.1 Type Parameters:

1. **Host Interface Type** `HPType <: TokenizedRecord`: The Chisel type of your Bridge's host-land interface. This describes how the target interface has been divided into separate token channels. One example, `HostPortIO[T]`, divides a Chisel Bundle into a single bi-directional token stream and is sufficient for defining bridges that do not model combinational paths between token streams. We suggest starting with `HostPortIO[T]` when defining a Bridge for modeling IO devices, as it is the simplest to reasonable about and can run at FMR = 1. For other port types, see Bridge Host Interaces.

26.2.2 Abstract Members:

1. **Host Interface Mock** `bridgeIO: HPType`: Here you'll instantiate a mock instance of your host-side interface. **This does not add IO to your target-side module**. Instead used to emit annotations that tell Golden Gate how the target-land IO of the target-side module is being divided into channels.
2. **Bridge Module Constructor Arg** `constructorArg: Option[AnyRef]`: A optional Scala case class you'd like to pass to your host-land BridgeModule's constructor. This will be serialized into an annotation and consumed later by Golden Gate. If provided, your case class should capture all target-land configuration information you'll need in your Module's generator.
3. **BridgeModule Type** `moduleName: String`: The type of the host-land BridgeModule you want Golden Gate to connect in-place of your target-side module as a fully qualifed Scala class (package and name of class). Golden Gate will use its class name to invoke its constructor.

Finally at the bottom of your Bridge's class definition **you'll need to call generateAnnotations()**. This is necessary to have Golden Gate properly detect your bridge.

You can freely instantiate your Bridge anywhere in your Target RTL: at the I/O boundary of your chip or deep in its module hierarchy. Since all of the Golden Gate-specific metadata is captured in FIRRTL annotations, you can generate your target design and simulate it a target-level RTL simulation or even pass it off to ASIC CAD tools – Golden Gate's annotations will simply be unused.

26.3 What Happens Next?

If you pass your design to Golden Gate, it will find your target-side module, remove it, and wire its dangling target-interface to the top-level of the design. During host-decoupling transforms, Golden Gate aggregates fields of your bridge's target interface based on channel annotations emitted by the target-side of your bridge, and wraps them up into decoupled interfaces that match your host interface definition. Finally, once Golden Gate is done performing compiler transformations, it generates the bridge modules (by looking up their constructors and passing them their serialized constructor argument) and connects them to the tokenized interfaces presented by the now host-decoupled simulator.

26.4 Host Side

The host side of a bridge has two components:

1. A FPGA-hosted bridge module (`BridgeModule`).
2. An optional, CPU-hosted, bridge driver (`bridge_driver_t`).

In general, bridges have both: in FASED memory timing models, the BridgeModule contains a timing model that exposes timing parameters as memory-mapped registers that the driver configures at the start of simulation. In the Block Device model, the driver periodically polls queues in the bridge module checking for new functional requests to be served. In the NIC model, the driver moves tokens in bulk between the software switch model and the bridge module, which simply queues up tokens as they arrive.

Communication between a bridge module and driver is implemented with two types of transport:

1. **MMIO**: In the module, this is implemented over a 32-bit AXI4-lite bus. Reads and writes to this bus are made by drivers using `simif_t::read()` and `simif_t::write()`. Bridge modules register memory mapped registers using methods defined in `midas.widgets.Widget`, addresses for these registers are passed to the drivers in a generated C++ header.
2. **DMA**: In the module this is implemented with a wide (e.g., 512-bit) AXI4 bus, that is mastered by the CPU. Bridge drivers initiate bulk transactions by passing buffers to `simif_t::push()` and `simif_t::pull()` (DMA from the FPGA). DMA is typically used to stream tokens into and out of out of large FIFOs in the BridgeModule.

26.5 Compile-Time (Parameterization) vs Runtime Configuration

As when compiling a software RTL simulator, the simulated design is configured over two phases:

1. **Compile Time**, by parameterizing the target RTL and BridgeModule generators, and by enabling Golden Gate optimization and debug transformations. This changes the simulator's RTL and thus requires a FPGA-recompilation. This is equivalent to, but considerably slower than, invoking VCS to compile a new simulator.
2. **Runtime**, by specifying plus args (e.g., +latency=1) that are passed to the BridgeDrivers. This is equivalent to passing plus args to a software RTL simulator, and in many cases the plus args passed to an RTL simulator and a FireSim simulator can be the same.

26.6 Target-Side vs Host-Side Parameterization

Unlike in a software RTL simulation, FireSim simulations have an additional phase of RTL elaboration, during which bridge modules are generated (they are themselves Chisel generators).

The parameterization of your bridge module can be captured in two places.

1. **Target side.** here parameterization information is provided both as free parameters to the target's generator, and extracted from the context in which the bridge is instantiated. The latter might include things like widths of specific interfaces or bounds on the behavior the target might expose to the bridge (e.g., a maximum number of inflight requests). All of this information must be captured in a _single_ serializable constructor argument, generally a case class (see `Bridge.constructorArg`).
2. **Host side.** This is parameterization information captured in Golden Gate's `Parameters` object. This should be used to provide host-land implementation hints (that ideally don't change the simulated behavior of the system), or to provide arguments that cannot be serialized to the annotation file.

In general, if you can capture target-behavior-changing parameterization information from the target-side you should. This makes it easier to prevent divergence between a software RTL simulation and FireSim simulation of the same FIRRTL. It's also easier to configure multiple instances of the same type of bridge from the target side.

CHAPTER

TWENTYSEVEN

BRIDGE DEEP DIVE

In this section, we'll walkthrough a simple Target-to-Host bridge associated with Chipyard called the UARTBridge. It serves as an example of how to use FireSim as a library to create your own bridges. The UARTBridge uses host-MMIO to model a UART device.

Reading the *Target-to-Host Bridges* section is a prerequisite to reading these sections.

27.1 UART Bridge (Host-MMIO)

Source code for the UART Bridge lives in Chipyard in the ${CY_DIR}/generators/firechip area. Specifically, the following directories:

```
chipyard/generators/firechip/
    bridgeinterfaces/
        src/main/scala/
            UART.scala # Chisel IOs and Scala case classes shared between FireSim/
↪Chipyard
    goldengateimplementations/
        src/main/scala/
            UARTBridge.scala # BridgeModule definition
    bridgestubs/
        src/main/
            scala/uart/UARTBridge.scala # Target-Side Bridge
            cc/bridges/uart.cc          # Bridge Driver source
            cc/bridges/uart.h           # Bridge Driver header
    chip/
        src/main/
            cc/firesim/firesim_top.cc # Driver instantiation in the main simulation␣
↪driver
            makefrag/firesim/ # Target-specific build rules
                build.mk      # Definition of the Chisel elaboration step
                config.mk     # Target-specific configuration and path setup
                driver.mk     # Build rules for the driver
                metasim.mk    # Custom run commands for meta-simulation
```

27.1.1 Target Side

The first order of business when designing a new bridge is to implement its target side. In the case of UART we've defined a Chisel BlackBox[1] extending Bridge. We'll instantiate this BlackBox and connect it to UART IO in the top-level of our chip. We first define a class that captures the target-side interface of the Bridge (in ${CY_DIR}/generators/firechip/bridgeinterfaces/src/main/scala/UART.scala):

```
class UARTPortIO extends Bundle {
   val txd = Output(Bool())
   val rxd = Input(Bool())
 }

 class UARTBridgeTargetIO extends Bundle {
   val clock = Input(Clock())
   val uart = Flipped(new UARTPortIO)
   // Note this reset is optional and used only to reset target-state modeled
   // in the bridge. This reset is just like any other Bool included in your target
   // interface, simply appears as another Bool in the input token.
   val reset = Input(Bool())
 }
```

Here, we define a case class (in ${CY_DIR}/generators/firechip/bridgeinterfaces/src/main/scala/UART.scala) that carries additional metadata to the host-side BridgeModule. For UART, this is simply the clock-division required to produce the baudrate:

```
// Out bridge module constructor argument. This captures all of the extra
// metadata we'd like to pass to the host-side BridgeModule. Note, we need to
// use a single case class to do so, even if it is simply to wrap a primitive
// type, as is the case for the div Int.
case class UARTKey(div: Int)
```

Both these IOs and the case class needs be fully isolated from the target. This means that they should compile with FireSim's Chisel version (Chisel 3) and not include any target-specific classes, IOs, etc. Both the IOs and the case class is compiled in the target and is copied to FireSim to be compiled there as well.

Finally, we define the actual target-side module (specifically, a BlackBox)(in ${CY_DIR}/generators/firechip/bridgestubs/src/main/scala/uart/UARTBridge.scala):

```
class UARTBridge(initBaudRate: BigInt, freqMHz: Int)(implicit p: Parameters) extends␣
↪BlackBox
    with Bridge[HostPortIO[UARTBridgeTargetIO]] {
  // Module portion corresponding to this bridge
  val moduleName = "firechip.firesimonly.bridges.UARTBridgeModule"
  // Since we're extending BlackBox this is the port will connect to in our target's RTL
  val io = IO(new UARTBridgeTargetIO)
  // Implement the bridgeIO member of Bridge using HostPort. This indicates that
  // we want to divide io, into a bidirectional token stream with the input
  // token corresponding to all of the inputs of this BlackBox, and the output token␣
↪consisting of
  // all of the outputs from the BlackBox
  val bridgeIO = HostPort(io)
```

(continues on next page)

[1] You can also extend a non-BlackBox Chisel Module, but any Chisel source contained within will be removed by Golden Gate. You may wish to do this to enclose a synthesizable model of the Bridge for other simulation backends, or simply to wrap a larger chunk RTL you wish to model in the host-side of the Bridge.

(continued from previous page)

```
  // Do some intermediate work to compute our host-side BridgeModule's constructor␣
↪argument
  val div = (BigInt(freqMHz) * 1000000 / initBaudRate).toInt

  // And then implement the constructorArg member
  val constructorArg = Some(UARTKey(div))

  // Finally, and this is critical, emit the Bridge Annotations -- without
  // this, this BlackBox would appear like any other BlackBox to Golden Gate
  generateAnnotations()
}
```

To make it easier to instantiate our target-side module, we've also defined an optional companion object (in ${CY_DIR}/generators/firechip/bridgestubs/src/main/scala/uart/UARTBridge.scala):

```
object UARTBridge {
  def apply(clock: Clock, uart: sifive.blocks.devices.uart.UARTPortIO, reset: Bool,␣
↪freqMHz: Int)(implicit p: Parameters): UARTBridge = {
    val ep = Module(new UARTBridge(uart.c.initBaudRate, freqMHz))
    ep.io.uart.txd := uart.txd
    uart.rxd := ep.io.uart.rxd
    ep.io.clock := clock
    ep.io.reset := reset
    ep
  }
}
```

This target-side module doesn't compile with FireSim at all (except for the APIs given by the `firesim-lib` Scala project).

That completes the target-side definition.

27.1.2 Host-Side BridgeModule

The remainder of the file is dedicated to the host-side BridgeModule definition. Here we have to process tokens generated by the target, and expose a memory-mapped interface to the bridge driver.

Inspecting the top of the class (in ${CY_DIR}/generators/firechip/goldengateimplementations/src/main/scala/UARTBridge.scala):

```
// Our UARTBridgeModule definition, note:
// 1) it takes one parameter, key, of type UARTKey --> the same case class we captured␣
↪from the target-side
// 2) It accepts one implicit parameter of type Parameters
// 3) It extends BridgeModule passing the type of the HostInterface
//
// While the Scala type system will check if you parameterized BridgeModule
// correctly, the types of the constructor arugument (in this case UARTKey),
// don't match, you'll only find out later when Golden Gate attempts to generate your␣
↪module.
class UARTBridgeModule(key: UARTKey)(implicit p: Parameters) extends␣
↪BridgeModule[HostPortIO[UARTBridgeTargetIO]]()(p) {
```

(continues on next page)

(continued from previous page)

```
  lazy val module = new BridgeModuleImp(this) {
    val div = key.div
    // This creates the interfaces for all of the host-side transport
    // AXI4-lite for the simulation control bus, =
    // AXI4 for DMA
    val io = IO(new WidgetIO())

    // This creates the host-side interface of your TargetIO
    val hPort = IO(HostPort(new UARTBridgeTargetIO))

    // Generate some FIFOs to capture tokens...
    val txfifo = Module(new Queue(UInt(8.W), 128))
    val rxfifo = Module(new Queue(UInt(8.W), 128))

    val target = hPort.hBits.uart
    // In general, your BridgeModule will not need to do work every host-cycle. In␣
↪simple Bridges,
    // we can do everything in a single host-cycle -- fire captures all of the
    // conditions under which we can consume and input token and produce a new
    // output token
    val fire = hPort.toHost.hValid && // We have a valid input token: toHost ~= leaving␣
↪the transformed RTL
               hPort.fromHost.hReady && // We have space to enqueue a new output token
               txfifo.io.enq.ready      // We have space to capture new TX data
    val targetReset = fire & hPort.hBits.reset
    rxfifo.reset := reset.asBool || targetReset
    txfifo.reset := reset.asBool || targetReset

    hPort.toHost.hReady := fire
    hPort.fromHost.hValid := fire
```

Most of what follows is responsible for modeling the timing of the UART. As a bridge designer, you're free to take as many host-cycles as you need to process tokens. In simpler models, like this one, it's often easiest to write logic that operates in a single cycle but gate state-updates using a "fire" signal that is asserted when the required tokens are available.

Now, we'll skip to the end to see how to add registers to the simulator's memory map that can be accessed using MMIO from bridge driver.

```
// Exposed the head of the queue and the valid bit as a read-only registers
// with name "out_bits" and out_valid respectively
genROReg(txfifo.io.deq.bits, "out_bits")
genROReg(txfifo.io.deq.valid, "out_valid")

// Generate a writeable register, "out_ready", that when written to dequeues
// a single element in the tx_fifo. Pulsify derives the register back to false
// after pulseLength cycles to prevent multiple dequeues
Pulsify(genWORegInit(txfifo.io.deq.ready, "out_ready", false.B), pulseLength = 1)

// Generate regisers for the rx-side of the UART; this is eseentially the reverse of the␣
↪above
genWOReg(rxfifo.io.enq.bits, "in_bits")
Pulsify(genWORegInit(rxfifo.io.enq.valid, "in_valid", false.B), pulseLength = 1)
```

(continues on next page)

(continued from previous page)

```
genROReg(rxfifo.io.enq.ready, "in_ready")

// This method invocation is required to wire up all of the MMIO registers to
// the simulation control bus (AXI4-lite)
genCRFile()
```

This module is injected into the FireSim compiler and is never compiled by the target. Thus it has access to all APIs given by MIDAS.

27.1.3 Host-Side Driver

To complete our host-side definition, we need to define a CPU-hosted bridge driver. Bridge Drivers extend the `bridge_driver_t` interface, which declares 5 virtual methods a concrete bridge driver must implement:

```
/**
 * @brief Base class for Bridge Drivers
 *
 * Bridge Drivers are the CPU-hosted component of a Target-to-Host Bridge. A
 * Bridge Driver interacts with their accompanying FPGA-hosted BridgeModule
 * using MMIO (via read() and write() methods).
 */
class bridge_driver_t : public widget_t {
public:
  using widget_t::widget_t;

  ~bridge_driver_t() override = default;

  /**
   * Initialize the bridge state.
   *
   * This method can be overriden to initialise the hardware through MMIO.
   */
  virtual void init() {}

  /**
   * Final post-run cleanup.
   *
   * The analog of init(), this provides a final opportunity to interact with
   * the FPGA before destructors are called at the end of simulation. Useful
   * for doing end-of-simulation clean up that requires communication with
   * the hardware through MMIO or streams.
   */
  virtual void finish() {}

  /**
   * Does work that allows the bridge to advance in simulation time.
   *
   * Progresses for one or more cycles. The standard FireSim driver calls the
   * tick methods of all registered bridge drivers. Bridges whose BridgeModule
   * is free-running need not implement this method
   */
```

(continues on next page)

(continued from previous page)

```
  virtual void tick() {}

  /**
   * Returns true if the bridge requests explicit termination.
   */
  virtual bool terminate() { return false; }

  /**
   * If the bridge driver calls for termination, encode a cause here.
   *
   * 0 = PASS. All other codes are bridge-implementation defined
   */
  virtual int exit_code() { return 0; }

protected:
  void write(size_t addr, uint32_t data);

  uint32_t read(size_t addr);
};
```

The declaration of the UART bridge is inlined below from ${CY_DIR}/generators/firechip/bridgestubs/src/main/cc/bridges/uart.h:

```
#ifndef __UART_H
#define __UART_H

#include "bridges/serial_data.h"
#include "core/bridge_driver.h"

#include <cstdint>
#include <memory>
#include <optional>
#include <signal.h>
#include <string>
#include <vector>

/**
 * Structure carrying the addresses of all fixed MMIO ports.
 *
 * This structure is instantiated when all bridges are populated based on
 * the target configuration.
 */
struct UARTBRIDGEMODULE_struct {
  uint64_t out_bits;
  uint64_t out_valid;
  uint64_t out_ready;
  uint64_t in_bits;
  uint64_t in_valid;
  uint64_t in_ready;
};

/**
 * Base class for callbacks handling data coming in and out a UART stream.
```

(continues on next page)

(continued from previous page)

```
*/
class uart_handler {
public:
  virtual ~uart_handler() = default;

  virtual std::optional<char> get() = 0;
  virtual void put(char data) = 0;
};

class uart_t final : public bridge_driver_t {
public:
  /// The identifier for the bridge type used for casts.
  static char KIND;

  /// Creates a bridge which interacts with standard streams or PTY.
  uart_t(simif_t &simif,
         const UARTBRIDGEMODULE_struct &mmio_addrs,
         int uartno,
         const std::vector<std::string> &args);

  ~uart_t() override;

  void tick() override;

private:
  const UARTBRIDGEMODULE_struct mmio_addrs;
  std::unique_ptr<uart_handler> handler;

  serial_data_t<char> data;

  void send();
  void recv();
};

#endif // __UART_H
```

The bulk of the driver's work is done in its `tick()` method. Here, the driver polls the BridgeModule and then does some work. Note: the name, `tick` is vestigial: one invocation of tick() may do work corresponding to an arbitrary number of target cycles. It's critical that tick be non-blocking, as waiting for work from the BridgeModule may deadlock the simulator.

27.1.4 Build-System Modifications

The final consideration in adding your bridge concerns the build system. You should be able to host the Scala sources for your bridge with rest of your target RTL: SBT will make sure those classes are available on the runtime classpath. If you're hosting your bridge driver sources outside of the existing directories, you'll need to modify your target-project make fragments to include them. The default Chipyard/Rocket Chip-based one lives in ${CY_DIR}/generators/firechip/chip/src/main/makefrag/firesim.

Here the main order of business is to add header and source files to `DRIVER_H` and `DRIVER_CC` respectively in *driver.mk*, by modifying the lines below:

```
##########################
# Driver Sources & Flags #
##########################

ifeq (,$(wildcard $(RISCV)/lib/libriscv.so))
$(warning libriscv not found)
LRISCV=
else
LRISCV=-lriscv
endif

firechip_lib_dir = $(chipyard_dir)/generators/firechip/chip/src/main/cc
firechip_bridgestubs_lib_dir = $(chipyard_dir)/generators/firechip/bridgestubs/src/main/
↪cc
testchipip_csrc_dir = $(chipyard_dir)/generators/testchipip/src/main/resources/
↪testchipip/csrc

# DRIVER_H only used to update recipe pre-reqs (ok to track more files)

# fesvr and related srcs
DRIVER_H += \
                $(shell find $(testchipip_csrc_dir) -name "*.h") \
                $(shell find $(firechip_bridgestubs_lib_dir)/fesvr -name "*.h")
DRIVER_CC += \
                $(testchipip_csrc_dir)/cospike_impl.cc \
                $(testchipip_csrc_dir)/testchip_tsi.cc \
                $(testchipip_csrc_dir)/testchip_dtm.cc \
                $(testchipip_csrc_dir)/testchip_htif.cc \
                $(firechip_bridgestubs_lib_dir)/fesvr/firesim_tsi.cc \
                $(firechip_bridgestubs_lib_dir)/fesvr/firesim_dtm.cc \
                $(RISCV)/lib/libfesvr.a
# Disable missing override warning for testchipip.
TARGET_CXX_FLAGS += \
                -isystem $(testchipip_csrc_dir) \
                -isystem $(RISCV)/include \
                -Wno-inconsistent-missing-override
TARGET_LD_FLAGS += \
                -L$(RISCV)/lib \
                -Wl,-rpath,$(RISCV)/lib \
                $(LRISCV)

# top-level sources
DRIVER_CC += $(addprefix $(firechip_lib_dir)/firesim/, $(addsuffix .cc, firesim_top))
TARGET_CXX_FLAGS += -I$(firechip_bridgestubs_lib_dir)/bridge/test

# bridge sources
DRIVER_H += $(shell find $(firechip_bridgestubs_lib_dir) -name "*.h")
DRIVER_CC += \
                $(wildcard \
                        $(addprefix \
                                $(firechip_bridgestubs_lib_dir)/, \
                                $(addsuffix .cc,bridges/* bridges/tracerv/* bridges/
↪cospike/*) \
```

(continues on next page)

(continued from previous page)

```
                        ) \
                )
TARGET_CXX_FLAGS += \
                -I$(firechip_bridgestubs_lib_dir) \
                -I$(firechip_bridgestubs_lib_dir)/bridge \
                -I$(firechip_bridgestubs_lib_dir)/bridge/tracerv \
                -I$(firechip_bridgestubs_lib_dir)/bridge/cospike
TARGET_LD_FLAGS += \
        -l:libdwarf.so -l:libelf.so \
        -lz \

# other
TARGET_CXX_FLAGS += \
                -I$(GENERATED_DIR) \
                -g
```

Then the other .mk files in the directory handle copying sources from the target to FireSim, building RTL, and more. That's it! At this point you should be able to both test your bridge in software simulation using metasimulation, or deploy it to an FPGA.

CHAPTER

TWENTYEIGHT

SIMULATION TRIGGERS

It is often useful to globally coordinate debug and instrumentation features using specific target-events that may be distributed across the target design. For instance, you may wish to enable collection of synthesized prints and sampling of AutoCounters simulataenously when a specific instruction is committed on any core, or alternatively if the memory system sees a write to a particular memory address. Golden Gate's trigger system enables this by aggregating annotated `TriggerSources` distributed throughout the design using a centralized credit-based system which then drives a single-bit level-sensitive enable to all `TriggerSinks` distributed throughout the design. This enable signal is asserted while the design remains in the region-of-interest (ROI). Sources signal the start of the ROI by granting a credit and signal the end of the ROI by asserting a debit. Since there can be multiple sources, each of which might grant credits, the trigger is only disabled when the system has been debited as exactly as many times as it has been credited (it has a balance of 0).

28.1 Quick-Start Guide

28.1.1 Level-Sensitive Trigger Source

Assert the trigger while some boolean `enable` is true.

```
import midas.targetutils.TriggerSource
TriggerSource.levelSensitiveEnable(enable)
```

Caveats:

- The trigger visible at the sink is delayed. See *Trigger Timing*.
- Assumes this is the only source; the trigger is only cleared if no additional credit has been granted.

28.1.2 Distributed, Edge-Sensitive Trigger Source

Assert trigger enable when some boolean *start* undergoes a positive transition, and clear the trigger when a second signal *stop* undergoes a positive transition.

```
// Some arbitarily logic to drive the credit source and sink. Replace with your own!
val start = lfsr(1)
val stop  = ShiftRegister(lfsr(0), 5)

// Now annotate the signals.
import midas.targetutils.TriggerSource
TriggerSource.credit(start)
```

(continues on next page)

(continued from previous page)

```
TriggerSource.debit(stop)
// Note one could alternatively write: TriggerSource(start, stop)
```

Caveats:

- The trigger visible at the sink is delayed. See *Trigger Timing*.
- Assumes these are the only sources; the trigger is only cleared if no additional credit has been granted.

28.2 Chisel API

Trigger sources and sinks are Boolean signals, synchronous to a particular clock domain, that have been annotated as such. The `midas.targetutils` package provides chisel-facing utilities for annotating these signals in your design. We describe these utilities below, the source for which can be found in sim/midas/targetutils/src/main/scala/midas/annotations.scala.

28.2.1 Trigger Sources

In order to permit distributing trigger sources across the whole design, you must annotate distinct boolean signals as credits and debits using methods provided by the `TriggerSource` object. We provide an example below (the distributed example from the quick-start guide).

```
// Some arbitarily logic to drive the credit source and sink. Replace with your own!
val start = lfsr(1)
val stop  = ShiftRegister(lfsr(0), 5)

// Now annotate the signals.
import midas.targetutils.TriggerSource
TriggerSource.credit(start)
TriggerSource.debit(stop)
// Note one could alternatively write: TriggerSource(start, stop)
```

Using the methods above, credits and debits issued while the design is under reset are not counted (the reset used is implicit reset of the Chisel Module in which you invoked the method). If the module provides no implicit reset or if you wish to credit or debit the trigger system while the local module's implicit reset is asserted, use `TriggerSource.{creditEvenUnderReset, debitEvenUnderReset}` instead.

28.2.2 Trigger Sinks

Like sources, trigger sinks are boolean signals that have been annotated alongside their associated clock. These signals will be driven by a Boolean value created by the trigger system. If trigger sources exist in your design, the generated trigger will **override all assignments made in the chisel to the same signal**, otherwise, it will take on a default value provided by the user. We provide an example of annotating a sink using the the `TriggerSink` object below.

```
// Note: this can be any reference you wish to have driven by the trigger.
val sinkBool  = WireDefault(true.B)

import midas.targetutils.TriggerSink
// Drives true.B if no TriggerSource credits exist in the design.
```

(continues on next page)

(continued from previous page)

```
// Note: noSourceDefault defaults to true.B if unset, and can be omitted for brevity
TriggerSink(sinkBool, noSourceDefault = true.B)
```

Alternatively, if you wish to use a trigger sink as a predicate for a Chisel `when` block, you may use `TriggerSink.whenEnabled` instead

```
/** A simpler means for predicating stateful updates, printfs, and assertions. Sugar␣
↪for: val sinkEnable =
  * Wire(Bool()) TriggerSink(sinkEnable, false.B) when (sinkEnable) { <...> }
  */
TriggerSink.whenEnabled(noSourceDefault = false.B) {
  SynthesizePrintf(printf(s"${printfPrefix}CYCLE: %d\n", cycle))
}
```

28.3 Trigger Timing

Golden Gate implements the trigger system by generating a target circuit that synchronizes all credit and debits into the base clock domain using a *single* register stage, before doing a global accounting. If the total number of credits exceeds debits the trigger is asserted. This trigger is then synchronized in each sink domain using a single register stage before driving the annotated sink. The circuit that implements this functionality is depicted below:

Fig. 1: Trigger generation circuit. Not shown: a sub-circuit analagous to that which totalCredit' is replicated to count debits. Similarly, the sub-circuit feeding the add-reduction is generated for each clock domain that contains at least one source annotation.

Given the present implementation, an enabled trigger becomes visible in a sink domain no sooner than one base-clock edge and one local-clock edge have elapsed, in that order, after the credit was asserted. This is depicted in the waveform below.

Fig. 2: Trigger timing diagram.

Note that trigger sources and sinks that reside in the base clock domain still have the additional synchronization registers even though they are uneeded. Thus, a credit issued by a source in the base clock domain will be visible to a sink also in the base clock domain exactly 2 cycles after it was issued.

Bridges that use the default `HostPort` interface add an additional cycle of latency in the bridge's local domain since their token channels model a single register stage to improve simulation FMR. Thus, without using a different `HostPort` implementation, trigger sources generated by a Bridge and trigger sinks that feed into a Bridge will each see one additional bridge-local cycle of latency. In constrast, synthesized printfs and assertions, and AutoCounters all use wire channels (since they are unidirectional interfaces, the extra register stage is not required to improve FMR) and will see no additional sink latency.

28.4 Limitations & Pitfalls

- The system is limited to no more than one trigger signal. Presently, there is no means to generate unique triggers for distinct sets of sinks.
- Take care to never issue more debits than credits, as this may falsely enable the trigger under the current implementation.

CHAPTER

TWENTYNINE

OPTIMIZING FPGA RESOURCE UTILIZATION

One advantage of a host-decoupled simulator is the ability to spread expensive operations out over multiple FPGA cycles while maintaining perfect cycle accuracy. When employing this strategy, a simulator can rely on a resource-efficient implementation that takes multiple cycles to complete the underlying computation to determine the next state of the target design. In the abstract, this corresponds with the simulator having *less* parallelism in its host implementation than the target design. While this strategy is intrinsic to the design of the compilers that map RTL circuits to software simulators executing on sequential, general-purpose hardware, it is less prevalent in FPGA simulation. These *space-time* tradeoffs are mostly restricted to hand-written, architecture-specific academic simulators or to implementing highly specific host features like I/O cuts in a partitioned, multi-FPGA environment.

With the Golden Gate compiler, we provide a framework for automating these optimization, as discussed in the 2019 ICCAD paper on the design of Golden gate. Furthermore, current versions of FireSim include two optional optimizations that can radically reduce resource utilization (and therefore simulate much large SoCs). The first optimization reduces the overhead of memories that are extremely to implement via direct RTL translation on an FPGA host, including multi-ported register files, while the second applies to repeated instances of large blocks in the target design by *threading* the work associated with simulating multiple instances across a single underlying host implementation.

29.1 Multi-Ported Memory Optimization

ASIC multi-ported RAMs are a classic culprit of poor resource utilization in FPGA prototypes, as they cannot be trivially implemented with Block RAMs (BRAMs) and are instead decomposed into lookup tables (LUTs), multiplexers and registers. While using double-pumping, BRAM duplication, or FPGA-optimized microarchitectures can help, Golden Gate can automatically extract such memories and replace them with a decoupled model that simulates the RAM via serialized accesses to an underlying implementation that is amenable mapping to an efficiency Block RAM (BRAM). While this serialization comes at the cost of reduced emulation speed, the resulting simulator can fit larger SoCs onto existing FPGAs. Furthermore, the decoupling framework of Golden Gate ensures that the simulator will still produce bit-identical, cycle-accurate results.

While the details of this optimization are discussed at length in the ICCAD paper, it is relatively simple to deploy. First, the desired memories must be annotated via Chisel annotations to indicate that they should be optimized; for Rocket- and BOOM-based systems, these annotations are already provided for the cores' register files, which are the most FPGA-hostile memories in the designs. Next, with these annotations in place, enabling the optimization requires mixing in the `MCRams` class to the platform configuration, as shown in the following example build recipe:

```
firesim-boom-mem-opt:
    DESIGN: FireSim
    TARGET_CONFIG: WithNIC_DDR3FRFCFSLLC4MB_FireSimLargeBoomConfig
    PLATFORM_CONFIG: MCRams_BaseF1Config
    deploy_quintuplet: null
```

29.2 Multi-Threading of Repeated Instances

While optimizing FPGA-hostile memories can allow up to 50% higher core counts on the AWS-hosted VU9P FPGAs, significantly larger gains can be had by threading repeated instances in the target system. The *model multi-threading* optimization extracts these repeated instances and simulates each instance with a separate thread of execution on a shared underlying physical implementation.

As with the memory optimization, this requires the desired set of instances to be annotated in the target design. However, since the largest effective FPGA capacity increases for typical Rocket Chip targets are realized by threading the tiles that each contain a core complex, these instances are pre-annotated for both Rocket- and BOOM-based systems. To enable this tile multi-threading, it is necessary to mix in the `MTModels` class to the platform configuration, as shown in the following example build recipe:

```
firesim-threaded-cores-opt:
    DESIGN: FireSim
    TARGET_CONFIG: WithNIC_DDR3FRFCFSLLC4MB_FireSimQuadRocketConfig
    PLATFORM_CONFIG: MTModels_BaseF1Config
    deploy_quintuplet: null
```

This simulator configuration will rely on a single threaded model to simulate the four Rocket tiles. However, it will still produce bit- and cycle-identical results to any other platform configuration simulating the same target system.

In practice, the largest benefits will be realized by applying both the `MCRams` and `MCModels` optimizations to large, multi-core BOOM-based systems. While these simulator platforms will have reduced throughput relative to unoptimized FireSim simulators, very large SoCs that would otherwise never fit on a single FPGA can be simulated without the cost and performance drawbacks of partitioning.

```
firesim-optimized-big-soc:
    DESIGN: FireSim
    TARGET_CONFIG: MyMultiCoreBoomConfig
    PLATFORM_CONFIG: MTModels_MCRams_BaseF1Config
    deploy_quintuplet: null
```

CHAPTER

THIRTY

OUTPUT FILES

Golden Gate generates many output files, we describe them here. Note, the GG CML-argument `--output-filename-base=<BASE>` defines defines a common prefix for all output files.

30.1 Core Files

These are used in nearly all flows.

- **<BASE>.sv**: The verilog implementation of the simulator which will be synthesized onto the FPGA. The top-level is the Shim module specified in the `PLATFORM_CONFIG`.
- **<BASE>.const.h**: A target-specific header containing all necessary metadata to instantiate bridge drivers. This is linked into the simulator driver and meta-simulators (FPGA-level / MIDAS-level). Often referred to as "the header".

30.2 FPGA Build Files

These are additional files passed to the FPGA build directory.

- **<BASE>.defines.vh**: Verilog macro definitions for FPGA synthesis.
- **<BASE>.ila_insert_vivado.tcl**: Synthesizes an ILA for the design. See *AutoILA: Simple Integrated Logic Analyzer (ILA) Insertion* for more details about using ILAs in FireSim.
- **<BASE>.ila_insert_{inst, ports, wires}.v**: Instantiated in the FPGA project via `` `include `` directives to instantiate the generated ILA.
- **<BASE>.synthesis.xdc**: Xilinx design constraints for synthesis derived from collected XDCAnnotations.
- **<BASE>.implementation.xdc**: Xilinx design constraints for implementation derived from collected XDCAnnotations.

30.3 Metasimulation Files

These are additional sources used only for compiling metasimulators.

- **<BASE>.const.vh**: Verilog macros to define variable width fields.

CHAPTER

THIRTYONE

COMPILER & DRIVER DEVELOPMENT

31.1 Integration Tests

These are `ScalaTests` that call out to FireSim's Makefiles. These constitute the bulk of FireSim's tests for Target, Compiler, and Driver side features. Each of these tests proceeds as follows:

1. Elaborate a small Chisel target design that exercises a single feature (e.g., printf synthesis)
2. Compile the design with GoldenGate
3. Compile metasimulator using a target-specific driver and the Golden Gate-generated collateral
4. Run metasimulation with provided arguments (possibly multiple times)
5. Post-process metasimulation outputs in Scala

Single tests may be run directly out of sim/ as follows:

```
# Run all Chipyard-based tests (uses Rocket + BOOM)
make test

# Run all integration tests (very long running, not recommended)
make test TARGET_PROJECT=midasexamples

# Run a specific integration test (desired)
make testOnly TARGET_PROJECT=midasexamples SCALA_TEST=firesim.midasexamples.GCDF1Test

# note: you can disable certain subsets of tests by using a
# TEST_DISABLE_{VERILATOR,VCS,VIVADO}=1 environment variable
```

These tests may be run from the SBT console continuously, and SBT will rerun them on Scala changes (but not driver changes). Out of sim/:

```
# Launch the SBT console into the firesim subproject
# NB: omitting TARGET_PROJECT will put you in the FireChip subproject instead
make TARGET_PROJECT=midasexamples sbt

# Compile the Scala test sources (optional, to enable tab completion)
sbt:firesim> Test / compile

# Run a specific test once
sbt:firesim> testOnly firesim.midasexamples.GCDF1Test
```

(continues on next page)

(continued from previous page)

```
# Continuously rerun the test on Scala changes
sbt:firesim> ~testOnly firesim.midasexamples.GCDF1Test
```

31.1.1 Key Files & Locations

- **sim/firesim-lib/src/test/scala/TestSuiteCommon.scala**
  Base ScalaTest class for all tests that use FireSim's make build system
- **sim/src/test/scala/midasexamples/TutorialSuite.scala**
  Extension of TestSuiteCommon for most integration tests + concrete subclasses
- **sim/src/main/cc/midasexamples/**
  C++ sources for target-specific drivers
- **sim/src/main/cc/midasexamples/TestHarness.h**
  A common driver to extend for simple tests
- **sim/src/main/scala/midasexamples/**
  Where top-level Chisel modules (targets) are defined

31.1.2 Defining a New Test

1. Define a new target module (if applicable) under `sim/src/main/scala/midasexamples`.
2. Define a driver by extending `simif_t` or another child class under `src/main/cc/midasexamples`. Tests sequenced with the Peek Poke bridge may extend `simif_peek_poke_t`.
3. Create a test in `src/main/cc/midasexamples`. Register bridges and add override the `run` method.
4. Define a ScalaTest class for your design by extending `TutorialSuite`. Parameters will define define the tuple (`DESIGN`, `TARGET_CONFIG`, `PLATFORM_CONFIG`), and call out additional plusArgs to pass to the metasimulator. See the ScalaDoc for more info. Post-processing of metasimulator outputs (e.g., checking output file contents) can be implemented in the body of your test class.

31.2 Synthesizable Unit Tests

These are derived from Rocket-Chip's synthesizable unit test library and are used to test smaller, stand-alone Chisel modules.

Synthesizable unit tests may be run out of sim/ as follows:

```
# Run default tests without waves
$ make run-midas-unittests

# Run default suite with waves
$ make run-midas-unittests-debug

# Run default suite under Verilator
$ make run-midas-unittests  EMUL=verilator

# Run a different suite (registered under class name TimeOutCheck)
$ make run-midas-unittests  CONFIG=TimeOutCheck
```

Setting the make variable `CONFIG` to different scala class names will select between different sets of unittests. All synthesizable unittests registered under `WithAllUnitTests` class are run from ScalaTest and in CI.

31.2.1 Key Files & Locations

- **sim/midas/src/main/scala/midas/SynthUnitTests.scala**
  Synthesizable unit test modules are registered here.
- **sim/midas/src/main/cc/unittest/Makefrag**
  Make recipes for building and running the tests.
- **sim/firesim-lib/src/test/scala/TestSuiteCommon.scala**
  ScalaTest wrappers for running synthesizable unittests

31.2.2 Defining a New Test

1. Define a new Chisel module that extends `freechips.rocketchip.unittest.UnitTest`
2. Register your modules in a `Config` using the `UnitTests` key. See `SynthUnitTests.scala` for examples.

31.3 Scala Unit Testing

We also use ScalaTest to test individual transforms, classes, and target-side Chisel features (in `targetutils` package). These can be found in `<subproject>/src/test/scala` as is customary of Scala projects. ScalaTests in `targetUtils` generally ensure that target-side annotators behave correctly when deployed in a generator (they elaborate correctly or they give the desired error message.) ScalaTests in `midas` are mostly tailored to testing FIRRTL transforms, and have copied FIRRTL testing utilities into the source tree to make that process easier.

`targetUtils` scala tests can be run out of sim/ as follows:

```
# Pull open the SBT console in the firesim subproject
$ make TARGET_PROJECT=midasexamples sbt

# Switch to the targetutils package
sbt:firesim> project targetutils

# Run all scala tests under the ``targetutils`` subproject
sbt:midas-targetutils> test
```

Golden Gate (formerly midas) scala tests can be run by setting the scala project to `midas`, as in step 2 above.

31.3.1 Key Files & Locations

- **sim/midas/src/test/scala/midas**
  Location of GoldenGate and targetutils ScalaTests (targetutils is moved here to reduce Chisel 3.6 and 6+ APIs)

31.3.2 Defining A New Test

Extend the appropriate ScalaTest spec or base class, and place the file under the correct `src/test/scala` directory. They will be automatically enumerated by ScalaTest and will run in CI by default.

31.4 C/C++ guidelines

The C++ sources are formatted using `clang-format` and all submitted pull-requests must be formatted prior to being accepted and merged. The sources follow the coding style defined here. Additionally, `clang-tidy` is also run on CI to lint and validate C++ sources. The tool follows the guidelines and configuration of LLVM.

`git clang-format` can be used before committing to ensure that files are properly formatted. `make -C sim clang-tidy` can be used to run `clang-tidy`. *make -C sim clang-tidy-fix* automatically applies most fixes, but some errors and warnings require user intervention.

31.5 Scala guidelines

The Scala sources are formatted using both `Scalafmt` and `Scalafix`. All submitted pull-requests must be formatted prior to being accepted and merged. The configuration files are found here: Scalafmt config, Scalafix config. Run `make -C sim scala-lint-check` to check your code for compliance. Run `make -C sim scala-lint` to automatically apply fixes.

CHAPTER THIRTYTWO

COMPLETE FPGA METASIMULATION

Generally speaking, users will only ever need to use conventional metasimulation (formerly, MIDAS-level simulation). However, when bringing up a new FPGA platform, or making changes to an existing one, doing a complete pre-synthesis RTL simulation of the FPGA project (which we will refer to as FPGA-level metasimulation) may be required. This will simulate the entire RTL project passed to Vivado, and includes exact RTL models of the host memory controllers and PCI-E subsystem used on the FPGA. Note, since FPGA-level metasimulation should generally not be deployed by users, when we refer to metasimulation in absence of the FPGA-level qualifier we mean the faster form described in *Debugging & Testing with Metasimulation*

FPGA-level metasimulations run out of sim/, and consist of two components:

1. A FireSim-f1 driver that talks to a simulated DUT instead of the FPGA
2. The DUT, a simulator compiled with either XSIM or VCS, that receives commands from the aforementioned FireSim-f1 driver

32.1 Usage

To run a simulation you need to make both the DUT and driver targets by typing:

```
make xsim
make xsim-dut <VCS=1> & # Launch the DUT
make run-xsim SIM_BINARY=<PATH/TO/BINARY/FOR/TARGET/TO/RUN> # Launch the driver
```

When following this process, you should wait until `make xsim-dut` prints `opening driver to xsim` before running `make run-xsim` (getting these prints from `make xsim-dut` will take a while).

Once both processes are running, you should see:

```
opening driver to xsim
opening xsim to driver
```

This indicates that the DUT and driver are successfully communicating. Eventually, the DUT will print a commit trace from Rocket Chip. There will be a long pause (minutes, possibly an hour, depending on the size of the binary) after the first 100 instructions, as the program is being loaded into FPGA DRAM.

XSIM is used by default, and will work on EC2 instances with the FPGA developer AMI. If you have a license, setting `VCS=1` will use VCS to compile the DUT (4x faster than XSIM). Berkeley users running on the Millennium machines should be able to source scripts/setup-vcsmx-env.sh to setup their environment for VCS-based FPGA-level simulation.

The waveforms are dumped in the FPGA build directories (`firesim/platforms/f1/aws-fpga/hdk/cl/developer_designs/cl_<DESIGN>-<TARGET_CONFIG>-<PLATFORM_CONFIG>`).

For XSIM:

```
<BUILD_DIR>/verif/sim/vivado/test_firesim_c/tb.wdb
```

And for VCS:

```
<BUILD_DIR>/verif/sim/vcs/test_firesim_c/test_null.vpd
```

When finished, be sure to kill any lingering processes if you interrupted simulation prematurely.

CHAPTER

THIRTYTHREE

VISUAL STUDIO CODE INTEGRATION

VSCode is a powerful IDE that can be used to do code and documentation development across the FireSim repository. It supports a client-server protocol over SSH that enables you to run a local GUI client that interacts with a server running on your remote manager.

33.1 General Setup

1. Install VSCode. You can grab installers here.
2. Open VSCode and install the `Remote Developer Plugin`. See the marketplace page for a complete description of its features.

At this point, VSCode will read in your `.ssh/config`. Hosts you've listed there will be listed under the `Remote Explorer` in the left sidebar. You'll be able to connect to these hosts and create workspaces under FireSim clones you've created there. You may need to give explicit names to hosts that would otherwise be captured as part of a pattern match or glob in your ssh config.

33.2 Workspace Locations

Certain plugins assume the presence of certain files in particular locations, and often it is diserable to reduce the scope of files that VSCode will index. We recommend opening workspaces at the following locations:

- Scala and C++ development: `sim/`
- RST docs: `docs/`
- Manager (python): `deploy/`

You can always open a workspace at the root of FireSim – just be cognizant that certain language-specific plugins (e.g., may not be configured correclty).

33.3 Scala Development

Warning

Until Chipyard is bumped, you must add bloop to Chipyard's `plugins.sbt` for this to work correctly. See sim/project/plugins.sbt and copy the bloop installation into `target-design/chipyard/project/plugins.sbt`.

VSCode has rich support for Scala development, and the Metals plugin is really what makes the magic happen.

33.3.1 How To Use (Remote Manager)

1. If you haven't already, clone FireSim and run `build-setup.sh` on your manager.
2. Ensure your manager instance is listed as a host in your `.ssh/config`. For example:

```
Host ec2-manager
    User centos
    IdentityFile ~/.ssh/<your-firesim.pem>
    Hostname <IP ADDR>
```

3. In VSCode, using the `Remote Manager` on the left sidebar, connect to your manager instance.
4. Open a workspace in your FireSim clone under `sim/`.
5. First time per remote: install the Metals plugin on the *remote* machine.
6. Metals will prompt you with the following: "New SBT Workspace Detected, would you like to import the build?". Click *Import Build*.

At this point, metals should automatically attempt to import the SBT-defined build rooted at `sim/`. It will:

1. Call out to SBT to run `bloopInstall`
2. Spin up a bloop build server.
3. Compile all scala sources for the default SBT project in firesim.

Once this process is complete, autocompletion, jump to source, code lenses, and all that good stuff should work correctly.

33.3.2 Limitations

1. **No test task support for ScalaTests that use make.** Due to the way FireSim's ScalaTest calls out to make to invoke the generator and Golden Gate, Metals's bloop instance must initialized with `env.sh` sourced. This will be resolved in a future PR.

33.3.3 Other Notes

Reliance on SBT multi-project builds breaks the default metals integration. To hide this, we've put workspace-specific settings for metals in sim/.vscode/settings.json which should permit metals to run correctly out of `sim/`. This instructs metals that:

1. We've already installed bloop (by listing it as a plugin in FireSim and Chipyard).
2. It should use a different sbt launch command to run `bloopInstall`. This sources `env.sh` and uses the sbt-launcher provided by Chipyard.

CHAPTER

THIRTYFOUR

MANAGING THE CONDA LOCK FILE

The default Conda environment set by `build-setup.sh` uses the lock file ("*.conda-lock.yml") in `conda-reqs/*`. This file is derived from the Conda requirements files (`*.yaml`) also located at `conda-reqs/*`.

34.1 Updating Conda Requirements

If developers want to update the requirements files, they should also update the lock file accordingly. There are two different methods:

1. Running `build-setup.sh --unpinned-deps`. This will update the lock file in place so that it can be committed and will re-setup the FireSim repository.
2. Running scripts/generate-conda-lockfile.sh. This will update the lock file in place without setting up your directory.

34.2 Caveats of the Conda Lock File and CI

Unfortunately, so far as we know, there is no way to derive the Conda requirements files from the Conda lock file. Thus, there is no way to verify that a lock file satisfies a set of requirements given by a requirements file(s). It is recommended that anytime you update a requirements file, you update the lock file in the same PR. This check is what the `check-conda-lock-modified` CI job does. It doesn't check that the lock file and requirements files have the same packages and versions, it only checks that all files are modified in the PR.

CHAPTER

THIRTYFIVE

MANAGER DEVELOPMENT

CHAPTER

THIRTYSIX

EXTERNAL TUTORIAL SETUP

This section of the documentation is for external attendees of a in-person FireSim and Chipyard tutorial. Please follow along with the following steps to get setup if you already have an AWS EC2 account.

Note

These steps should take around 2hrs if you already have an AWS EC2 account.

1. Start following the FireSim documentation from initial-setup but ending at *Setting up the FireSim Repo* (make sure to **NOT** clone the FireSim repository)
2. Run the following commands:

```
#!/bin/bash

FIRESIM_MACHINE_LAUNCH_GH_URL="https://raw.githubusercontent.com/firesim/firesim/final-
↪tutorial-2022-isca/scripts/machine-launch-script.sh"

curl -fsSLo machine-launch-script.sh "$FIRESIM_MACHINE_LAUNCH_GH_URL"
chmod +x machine-launch-script.sh
./machine-launch-script.sh

source ~/.bashrc

export MAKEFLAGS=-j16

sudo yum install -y nano

mkdir -p ~/.vim/{ftdetect,indent,syntax} && for d in ftdetect indent syntax ; do wget -O␣
↪~/.vim/$d/scala.vim https://raw.githubusercontent.com/derekwyatt/vim-scala/master/$d/
↪scala.vim; done

echo "colorscheme ron" >> /home/centos/.vimrc


cd ~/

(
git clone https://github.com/ucb-bar/chipyard -b final-tutorial-2022-isca-morning␣
↪chipyard-morning
cd chipyard-morning
```

(continues on next page)

(continued from previous page)

```
./scripts/init-submodules-no-riscv-tools.sh

./scripts/build-toolchains.sh ec2fast
source env.sh

./scripts/firesim-setup.sh --fast
cd sims/firesim
source sourceme-manager.sh

cd ~/chipyard-morning/sims/verilator/
make
make clean

cd ~/chipyard-morning
chmod +x scripts/repo-clean.sh
./scripts/repo-clean.sh
git checkout scripts/repo-clean.sh

)

cd ~/

(
git clone https://github.com/ucb-bar/chipyard -b final-tutorial-2022-isca chipyard-
↪afternoon
cd chipyard-afternoon
./scripts/init-submodules-no-riscv-tools.sh

./scripts/build-toolchains.sh ec2fast
source env.sh

./scripts/firesim-setup.sh --fast
cd sims/firesim
source sourceme-manager.sh
cd sim
unset MAKEFLAGS
make f1
export MAKEFLAGS=-j16

cd ../target-design/chipyard/software/firemarshal
./init-submodules.sh
marshal -v build br-base.json
marshal -v install br-base.json

cd ~/chipyard-afternoon/generators/sha3/software/
git submodule update --init esp-isa-sim
git submodule update --init linux
./build-spike.sh
./build.sh

cd ~/chipyard-afternoon/generators/sha3/software/
marshal -v build marshal-configs/sha3-linux-jtr-test.yaml
```

(continues on next page)

(continued from previous page)

```
marshal -v build marshal-configs/sha3-linux-jtr-crack.yaml
marshal -v install marshal-configs/sha3*.yaml

cd ~/chipyard-afternoon/sims/firesim/sim/
unset MAKEFLAGS
make f1 DESIGN=FireSim TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.QuadRocketConfig PLATFORM_CONFIG=BaseF1Config
make f1 DESIGN=FireSim TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.LargeBoomV3Config PLATFORM_CONFIG=BaseF1Config
make f1 DESIGN=FireSim TARGET_CONFIG=WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.RocketConfig PLATFORM_CONFIG=BaseF1Config
make f1 DESIGN=FireSim TARGET_CONFIG=WithNIC_DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.Sha3RocketConfig PLATFORM_CONFIG=BaseF1Config
make f1 DESIGN=FireSim TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.Sha3RocketConfig PLATFORM_CONFIG=BaseF1Config
make f1 DESIGN=FireSim TARGET_CONFIG=DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_
↪WithFireSimHighPerfConfigTweaks_chipyard.Sha3RocketPrintfConfig PLATFORM_
↪CONFIG=WithPrintfSynthesis_BaseF1Config
export MAKEFLAGS=-j16

cd ~/chipyard-afternoon
chmod +x scripts/repo-clean.sh
./scripts/repo-clean.sh
git checkout scripts/repo-clean.sh

)
```

3. Next copy the following contents and replace your entire `~/.bashrc` file with this:

```
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
# Uncomment the following line if you don't like systemctl's auto-paging feature:
# export SYSTEMD_PAGER=
# User specific aliases and functions
cd /home/centos/chipyard-afternoon && source env.sh && cd sims/firesim && source␣
↪sourceme-manager.sh && cd /home/centos/
export FDIR=/home/centos/chipyard-afternoon/sims/firesim/
export CDIR=/home/centos/chipyard-afternoon/
```

4. All done! Now continue with the in-person tutorial.

CHAPTER

THIRTYSEVEN

INDICES AND TABLES

- genindex
- modindex
- search