ISCAS 2021
Daegu, KOREA, MAY 22-28
IEEE International Symposium on Circuits and Systems

Vertically Integrated Computing Labs Using Open-Source Hardware Generators and Cloud-Hosted FPGAs

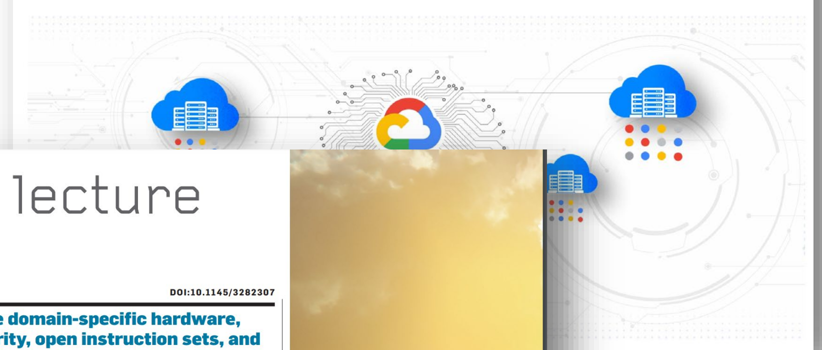
Alon Amid, Albert Ou,
Krste Asanovic, Yakun Sophia Shao, Borivoje Nikolic

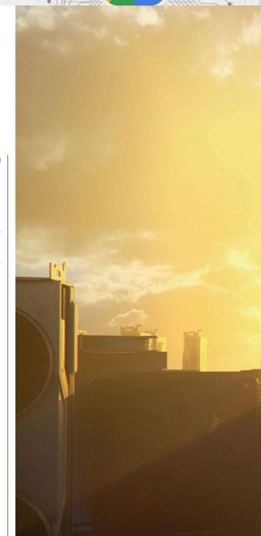
University of California, Berkeley

2021 IEEE International Symposium on Circuits and Systems
May 22-28, 2021 Virtual & Hybrid Conference

Vertically Integrated Computer Arch.

- The future of computers
  - Specialization and customization
  - HW/SW co-design
  - Vertical integration
- Computer architecture classes
  - Intersection between CS and EE
  - Opportunity to demonstrate and exercise vertical topics
  - HW/SW integration often discussed in lectures, but not in labs assignments and exercises
  - Hardware-centric vs. software centric classes





[1] https://cloud.google.com/blog/topics/systems/the-past-present-and-future-of-custom-compute-at-google
[2] https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/

HW-Centric vs. SW-Centric

$$\frac{time}{program} = \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle}$$

Software-Centric (instructions/program × cycles/instruction) Hardware-Centric (cycles/instruction × time/cycle)

Software-Centric

- No RTL prerequisites
- Assignment types:
  - Real-system Profiling
  - Binary instrumentation
  - Abstract Simulators (Champsim, Ramulator, custom, etc.)
- Examples: MIT 6.823 [1], Stanford EE282 [2], ETH Zurich Computer Architecture [3]

[1] http://csg.csail.mit.edu/6.823/labs.html
[2] https://web.stanford.edu/class/ee282/
[3] https://safari.ethz.ch/architecture/fall2020/

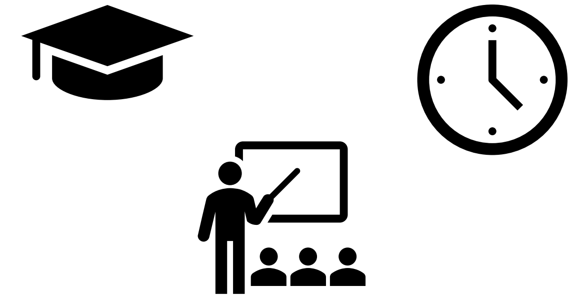
Hardware-Centric

- RTL-based implementation and evaluation
- Assignment types:
  - Extending a processor
  - Module implementations
  - RISC-V/MIPS Simple processor implementations
- Examples: Princeton ELE/COS 475 [4], Michigan EECS 470[5], Pohang University of Science and Technology

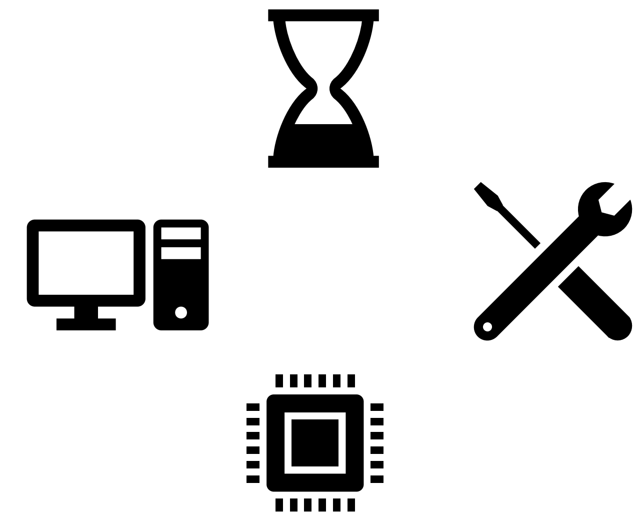
[4] http://eleclass.princeton.edu/classes/ele475/fall_2019
[5] https://www.eecs.umich.edu/courses/eecs470/

Can we do both?

- Course Design Aspects:
  - Time
  - Class scope and objectives
  - Prior student experience (pre-requisites)
  - Teaching staff expertise



- Tool/Infrastructure Aspects:
  - System support (compilers, OS, runtimes, processors, etc.)
  - Base implementation detail
  - Simulation time
  - Computing resources

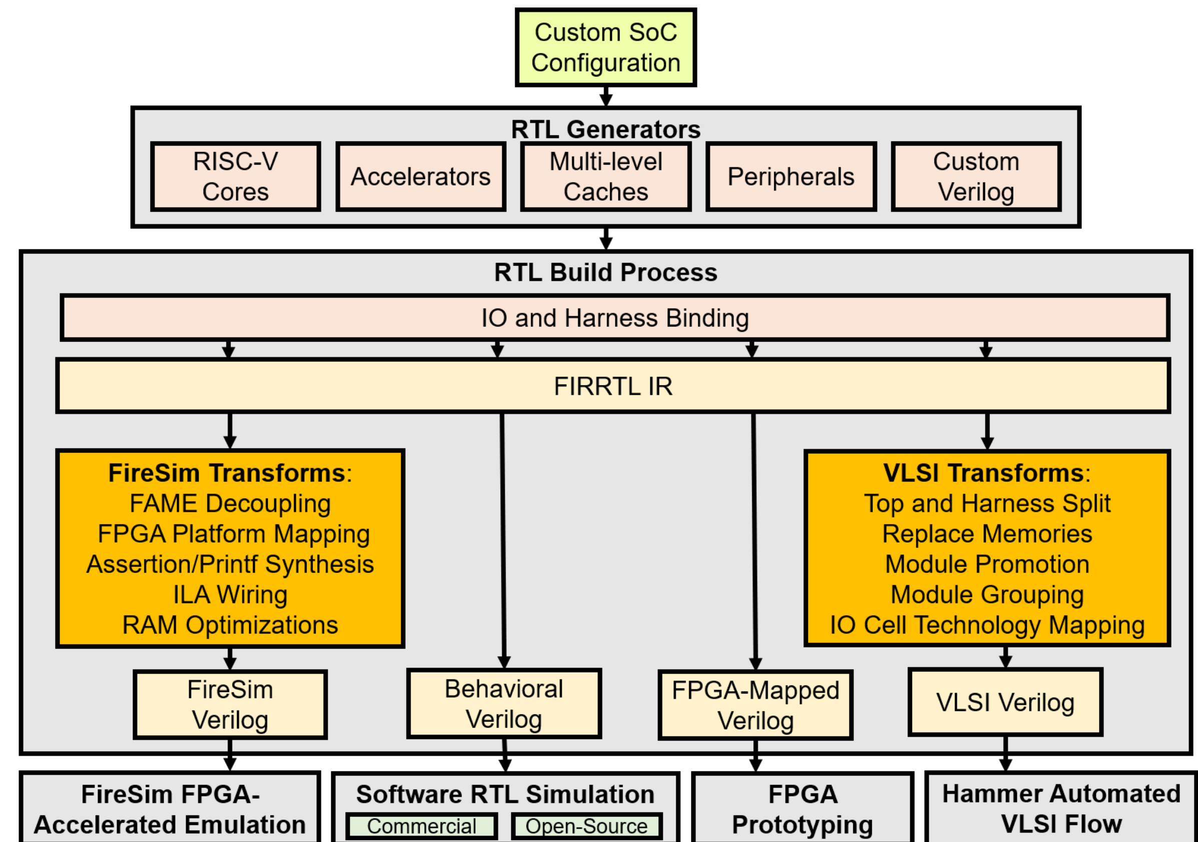


New/Re-emerging Technologies

- Free and open system-capable ISA specifications
  - RISC-V
  - Full-system software support (SPEC, Linux, etc.)
- Open-source software + hardware
  - Software: GCC, Linux
  - Hardware: Swerv, OpenPiton, Rocket, BOOM, PULP
- Hardware generators
  - Highly parameterized and modular implementations
  - High-level language abstractions
- Accessible FPGA acceleration
  - Cloud-hosted FPGAs: Amazon EC2 F1 Instances
  - FPGA-based emulation/simulation: FireSim

Open Source Hardware Generators

- Expertise required for ***design***, but ***usage*** through configurations is simple
- Cycle-accurate generated-RTL simulation
- Complete open-source processor and system implementation
- Rocket-chip generator and the Chipyard framework
  - Complete flows
  - Silicon proven
  - Industry-usage

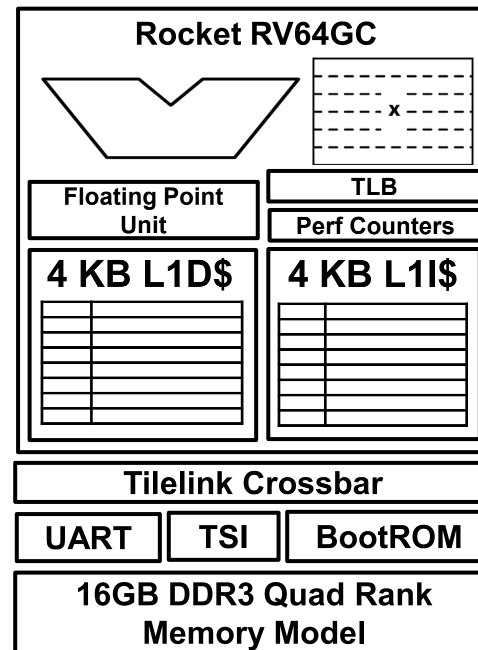


Hardware Generators in Class

- Students interact with configurations
  - Can add RTL if class calls for it
- Access to complete systems
  - Explore hardware modifications
  - Explore software modifications
- Detailed implementations can be used to train students on professional tools
  - Electronic design automation (EDA)
  - Compilers, OS kernels and runtimes
- Labs can be tailored to the scope of the class
  - Software labs
  - High-level labs
  - Low-level labs

Hardware Generators – Example 1

```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                  ++
  new WithBootROM                         ++
  new WithSerial                          ++
  new WithPerfCounters                    ++
  new WithDefaultMemModel                 ++
  new WithL1ICacheSets(64)                ++
  new WithL1ICacheWays(1)                 ++
  new WithL1DCacheSets(64)                ++
  new WithL1DCacheWays(1)                 ++
  new WithCacheBlockBytes(64)             ++
  new freechips.rocketchip.system.DefaultConfig)
```



CS 152 Laboratory Exercise 2

3.3 Matrix Transposition Case Study

The directed portion will lead you through a simple case study of a matrix transposition kernel with these objectives:

- Illustrate some basic cache optimization techniques
- Conduct a brief design-space exploration of cache configurations using the Rocket Chip parameterization system
- Familiarize you with the RTL simulation flow

We begin with a naive implementation of matrix transposition in `${LAB2ROOT}/lab/directed/transpose.c` that is derived directly from the mathematical definition. Take a moment to understand the source code. Note that both the 256×64 input matrix and 64×256 output matrix are stored in row-major order. The matrix elements are 64-bit integers.

Compile it into a bare-metal binary:

```
eecs$ TESTDIR=${LAB2ROOT}/lab/directed
eecs$ cd ${TESTDIR}
eecs$ make
```

Next, navigate to the Verilator directory and build the simulator. Notice that the `CONFIG` variable selects the top-level SoC design to generate – what exactly this means will be described in the next section. The Chisel design is elaborated into Verilog RTL, which is then compiled into a cycle-accurate simulator.

```
eecs$ SIMDIR=${LAB2ROOT}/sims/verilator
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketConfig -j4
```

! → This particular configuration contains a 4 KiB direct-mapped L1 data cache and a 4 KiB direct-mapped L1 instruction cache, both with 64-byte cache lines.

Next, run the naive matrix transposition kernel on the simulator:

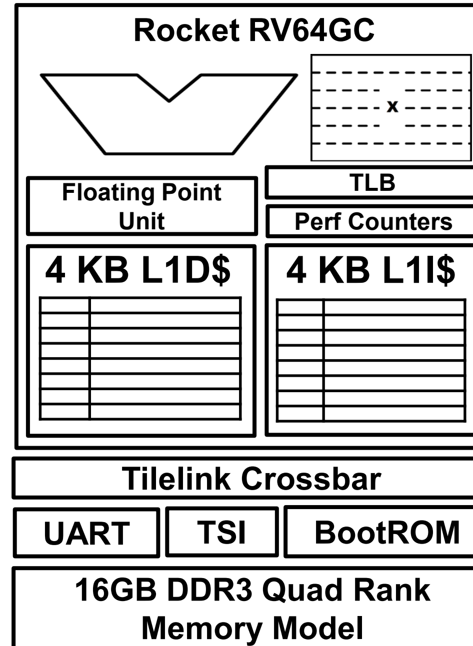
```
eecs$ make CONFIG=CS152RocketConfig run-binary-hex BINARY=${TESTDIR}/transpose.riscv
```

This will involve a few minutes of waiting, as the entire program takes approximately 2 million target cycles to execute.

[1] https://inst.eecs.berkeley.edu/~cs152/sp21/assignments/sp21/lab2.pdf

Hardware Generators – Example 1

```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                ++
  new WithBootROM                       ++
  new WithSerial                        ++
  new WithPerfCounters                  ++
  new WithDefaultMemModel               ++
  new WithL1ICacheSets(64)              ++
  new WithL1ICacheWays(1)               ++
  new WithL1DCacheSets(64)              ++
  new WithL1DCacheWays(1)               ++
  new WithCacheBlockBytes(64)           ++
  new freechips.rocketchip.system.DefaultConfig)
```



CS 152 Laboratory Exercise 2

3.3 Matrix Transposition Case Study

The directed portion will lead you through a simple case study of a matrix transposition kernel with these objectives:

- Illustrate some basic cache optimization techniques
- Conduct a brief design-space exploration of cache configurations using the Rocket Chip parameterization system
- Familiarize you with the RTL simulation flow

We begin with a naive implementation of matrix transposition in `${LAB2ROOT}/lab/directed/transpose.c` that is derived directly from the mathematical definition. Take a moment to understand the source code. Note that both the 256×64 input matrix and 64×256 output matrix are stored in row-major order. The matrix elements are 64-bit integers.

Compile it into a bare-metal binary:

```
eecs$ TESTDIR=${LAB2ROOT}/lab/directed
eecs$ cd ${TESTDIR}
eecs$ make
```

Next, navigate to the Verilator directory and build the simulator. Notice that the `CONFIG` variable selects the top-level SoC design to generate – what exactly this means will be described in the next section. The Chisel design is elaborated into Verilog RTL, which is then compiled into a cycle-accurate simulator.

```
eecs$ SIMDIR=${LAB2ROOT}/sims/verilator
eecs$ cd ${SIMDIR}
eecs$ make CONFIG=CS152RocketConfig -j4
```

! → This particular configuration contains a 4 KiB direct-mapped L1 data cache and a 4 KiB direct-mapped L1 instruction cache, both with 64-byte cache lines.

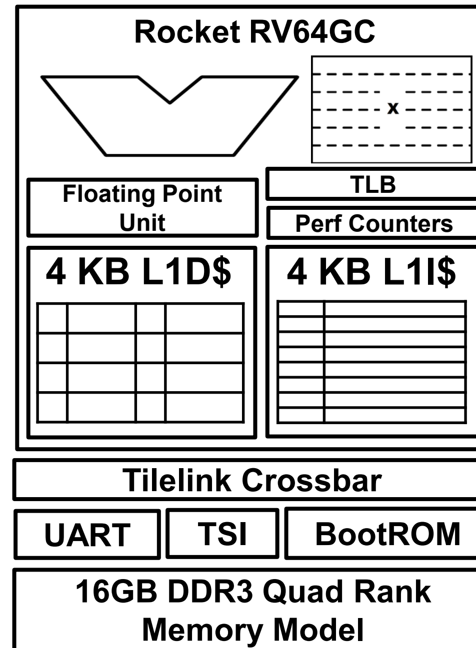
Next, run the naive matrix transposition kernel on the simulator:

```
eecs$ make CONFIG=CS152RocketConfig run-binary-hex BINARY=${TESTDIR}/transpose.riscv
```


This particular configuration contains a 4 KiB direct-mapped L1 data cache and a 4 KiB direct-mapped L1 instruction cache, both with 64-byte cache lines.

Hardware Generators – Example 1

```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                     ++
  new WithBootROM                            ++
  new WithSerial                             ++
  new WithPerfCounters                       ++
  new WithDefaultMemModel                    ++
  new WithL1ICacheSets(64)                   ++
  new WithL1ICacheWays(1)                    ++
  new WithL1DCacheSets(32)                   ++
  new WithL1DCacheWays(2)                    ++
  new WithCacheBlockBytes(64)                ++
  new freechips.rocketchip.system.DefaultConfig)
```



CS 152 Laboratory Exercise 2

bottom to top) in the chain, by reverse order of precedence. Thus, a Config appearing to the left of (or above) another Config overrides any parameters previously set by the latter. For more information on the Rocket Chip parameter system, read through the Chipyard documentation.[5]

In `CS152RocketConfig`, change the associativity of the L1 data cache to 2 by modifying `WithL1DWays` parameter, while also adjusting `WithL1DSets` to keep the overall cache size constant. Rebuild the simulator and re-run your blocked matrix transpose version. Repeat this with 4 and 8 ways.

(3.5.a) How does performance and miss rate change when associativity is increased?

(3.5.b) Explain why higher associativity is or is not beneficial for this particular kernel.

3.6 Multi-level Caches

We will continue our experimentation with the `CS152RocketL2Config` design, which is derived from `CS152RocketConfig` but adds a 64 KiB 8-way inclusive L2 cache. (Remember to change the L1 configuration back to a 4 KiB direct-mapped cache.)

Modify your transposition code to introduce another level of cache blocking for the L2. Simulate on `CS152RocketL2Config` and answer the following:

```
eecs$ make CONFIG=CS152RocketL2Config run-binary-hex BINARY=${TESTDIR}/transpose.riscv
```

(3.6.a) Does adding another level of cache blocking improve performance compared to your previous code from 3.5? Why or why not? (**Hint**: Consider whether there is any locality left for the L2 cache to exploit.)

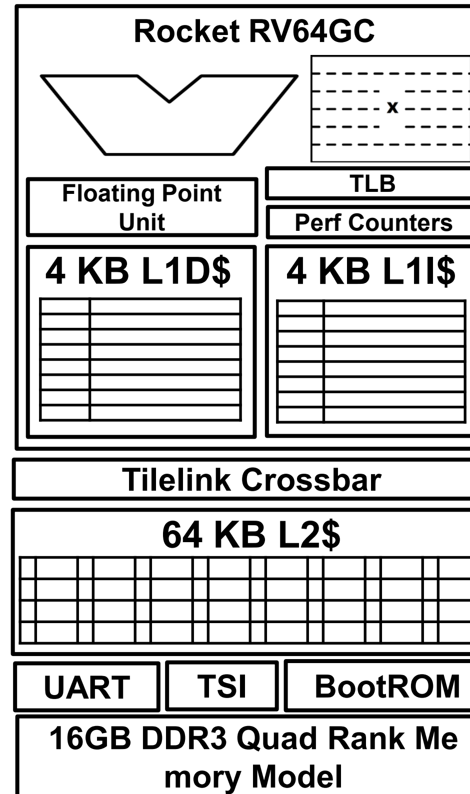
In `CS152RocketConfig`, change the associativity of the L1 data cache to 2 by modifying `WithL1DWays` parameter, while also adjusting `WithL1DSets` to keep the overall cache size constant. Rebuild the simulator and re-run your blocked matrix transpose version. Repeat this with 4 and 8 ways.

(3.5.a) How does performance and miss rate change when associativity is increased?

(3.5.b) Explain why higher associativity is or is not beneficial for this particular kernel.

Hardware Generators – Example 1

```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                          ++
  new WithBootROM                                 ++
  new WithSerial                                  ++
  new WithPerfCounters                            ++
  new WithDefaultMemModel                         ++
  new WithL1ICacheSets(64)                        ++
  new WithL1ICacheWays(1)                         ++
  new WithL1DCacheSets(64)                        ++
  new WithL1DCacheWays(1)                         ++
  new WithCacheBlockBytes(64)                     ++
  new WithInclusiveCache(nBanks = 1,
                         nWays = 8,
                         capacityKB = 64)         ++
  new freechips.rocketchip.system.DefaultConfig)
```



CS 152 Laboratory Exercise 2

bottom to top) in the chain, by reverse order of precedence. Thus, a Config appearing to the left of (or above) another Config overrides any parameters previously set by the latter. For more information on the Rocket Chip parameter system, read through the Chipyard documentation.[5]

In `CS152RocketConfig`, change the associativity of the L1 data cache to 2 by modifying `WithL1DWays` parameter, while also adjusting `WithL1DSets` to keep the overall cache size constant. Rebuild the simulator and re-run your blocked matrix transpose version. Repeat this with 4 and 8 ways.

(3.5.a) How does performance and miss rate change when associativity is increased?

(3.5.b) Explain why higher associativity is or is not beneficial for this particular kernel.

3.6 Multi-level Caches

We will continue our experimentation with the `CS152RocketL2Config` design, which is derived from `CS152RocketConfig` but adds a 64 KiB 8-way inclusive L2 cache. (Remember to change the L1 configuration back to a 4 KiB direct-mapped cache.)

Modify your transposition code to introduce another level of cache blocking for the L2. Simulate on `CS152RocketL2Config` and answer the following:

```
eecs$ make CONFIG=CS152RocketL2Config run-binary-hex BINARY=${TESTDIR}/transpose.riscv
```

(3.6.a) Does adding another level of cache blocking improve performance compared to your previous code from 3.5? Why or why not? (**Hint**: Consider whether there is any locality left for the L2 cache to exploit.)

3.6 Multi-level Caches

We will continue our experimentation with the `CS152RocketL2Config` design, which is derived from `CS152RocketConfig` but adds a 64 KiB 8-way inclusive L2 cache. (Remember to change the L1 configuration back to a 4 KiB direct-mapped cache.)

Hardware Generators – Example 2

- Use a small configuration of the same in-order core implementation
- Generate synthesizable Verilog for a complete processor core
- Exercise the use of VLSI tools to extract physical properties of the processor core – power and rail analysis
- Train students in physical design and VLSI tools + enable complete system PPA analysis

Power and Rail Analysis on a RocketChip

EE241B Lab 5
Written by Harrison Liew (2021)

Overview

In prior labs, you increasingly dove deeper into the details of VLSI design, all the way down to the custom design flow. In this lab, we will explore power consumption using different analysis methodologies for a small RocketChip configuration from Chipyard and also learn how to do and interpret IR drop analysis.

Getting Started

We will once again start with updating our environment. Pull the latest changes to the lab Chipyard repository and update your submodules (specifically, `hammer` and `hammer-cadence-plugins`).

Then, run the flow on the `TinyRocketConfig`. This will take a few hours through elaboration/S-RAM mapping (as seen in Lab 1), synthesis, and P&R. The binary from Lab 1 is included for the extra ungraded portion. Be mindful of compute resource utilization as well, since the tools will use bursts of 8 CPU cores. Continue reading through the lab while everything is running.

```
export CONFIG=TinyRocketConfig
export BINARY=$RISCV/riscv64-unknown-elf/share/riscv-tests/isa/rv64ui-p-simple
export INPUT_CONFS=lab5/ChipTop.yml
make par
```

Post-P&R Questions

You can come back to answering these questions after reading the next section and P&R finishes.

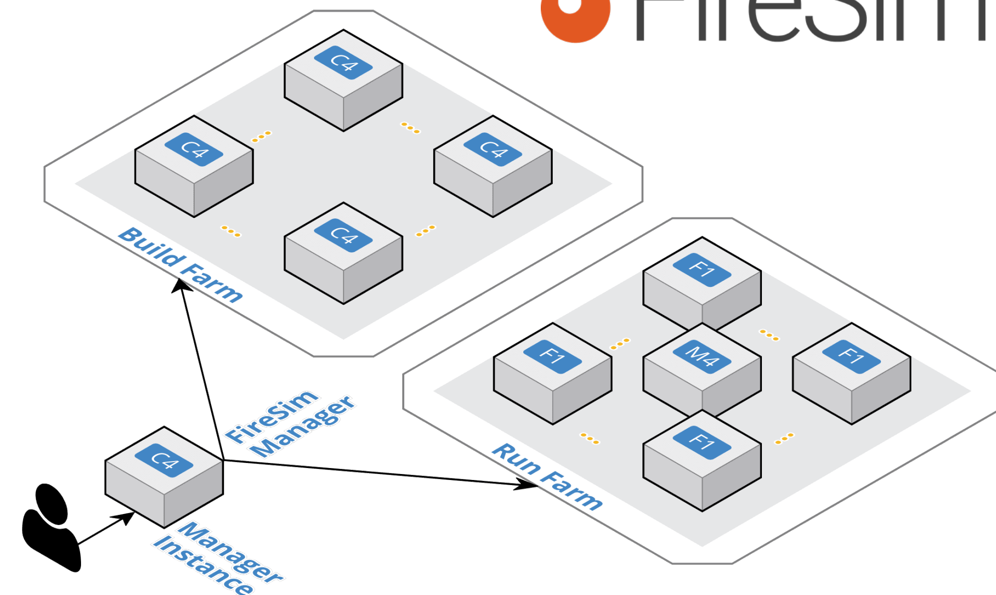
Q1: What is the clock period and uncertainty? Does it meet timing post-P&R? Are any of the SRAMs for the caches in the critical path? If not, where do they rank in the list of longest paths?

Q2: Open up the final Innovus database with the `open_chip` script as before. Turn off all the metal layers. Then, highlight all instances of `DECAP*` cells, and submit a screenshot in your lab report. You will refer back to this later. Note: you can do this through a command in the Innovus terminal like `select_obj [get_db insts -if {.base_cell.name == DECAP*}]`

[1] https://inst.eecs.berkeley.edu/~ee241b/sp21/assignments/lab5.pdf

Accessible FPGA Emulation

- Hardware-centric labs are limited by slow RTL simulation time
- FPGAs as an accelerated alternative
  - Requires significant boilerplate infrastructure work – pin assignments, IP generation, software host management and data movement.
  - Lab limited by the amount and type of FPGA board available.
- FPGA-accelerated Emulation vs. FPGA prototyping
  - Memory, IO and peripheral timing models
  - Observability and debug abilities
- FireSim
  - Automation
  - Accessibility through the public cloud



Full System FPGA Emulation - Example

- Hardware for machine learning class
  - HW/SW co-design
  - Long, compute-intensive workloads
- Lab Assignments:
  - Implement parts of an accelerator
  - Optimize software for the accelerator
- ResNet-50 Inference workloads:
  - Hours in software RTL-simulation
  - Minutes in FireSim
- Rapid development iterations
- Control over the complete HW/SW system

EE 290-2 Spring 2021 Lab 2: Systolic Arrays and Dataflows

Now that we have lowered the convolution operation into a GEMM operation, let us look at a common 3-level matrix multiplication loop for C = A*B:

```
for (int k = 0; k < DIM_K; k++) {
  for (int j = 0; j < DIM_J; j++) {
    for (int i = 0; i < DIM_I; i++) {
      C[i,j] += A[i,k] * B[k,j];
    }
  }
}
```

This particular dataflow is called a *weight-stationary* dataflow, since the weight (filters) elements remain constant, while the output data and feature input data flow during it's computation.
In your lab report, answer the following question:

1. Write down the loop ordering for an *output-stationary* dataflow, where the output matrix is the stationary matrix.

3 Your Assignment

The goals of this lab are to familiarize you with the concepts of dataflows in systolic array architectures, as well as the Chipyard and Gemmini tools. Therefore, in this lab, you will replace the existing Chisel implementation of the systolic array mesh in Gemmini with your own Verilog implementation. This will enable you to get hands-on experience with dataflow routing and processing elements implementations, as well as the components of the Gemmini accelerator and the Chipyard framework.
We have pr
/gemmini/Mes
accelerator. Y
/gemmini/src
will notice tha
You will need
understand wh
While the
an output-stat
dataflow. As a

EE 290-2 Spring 2021 Lab 3: Tiling and Optimization for Accelerators

1 Introduction

This lab will provide you with hands-on experience on the implications of mapping large matrix multiplication operations onto 2D systolic array accelerators, and give you experience using the Amazon Web Services (AWS) EC2 public cloud for FPGA-accelerated simulation.
As most neural network models do not fit in on-chip memory, loop blocking/tiling is an important tool in writing a performant neural network implementation. The additional degree of freedom afforded by the scratchpad in ML accelerators requires further planning of data re-use within the scratchpad. Furthermore, the size of the compute array adds an additional constraint on the tiling hierarchy.
There is a wide body of literature on loop blocking and scheduling for standard scalar processors. However, this body of literature is less extensive with regards to on-chip accelerators with dedicated memories that may be connected to the memory hierarchy in various forms.
Specifically, the DMA of the Gemmini accelerator is currently connected to the shared L2 cache of the scalar host processor. As such, it may see caching affects from the tiling scheme, as well as other parts of the program.
In this lab you will continue using the Chipyard and Gemmini platforms from the previous lab to further improve the performance of DNN execution using ML accelerators through software optimization. You will also be using the FireSim platform (within Chipyard) on the AWS EC2 public cloud.

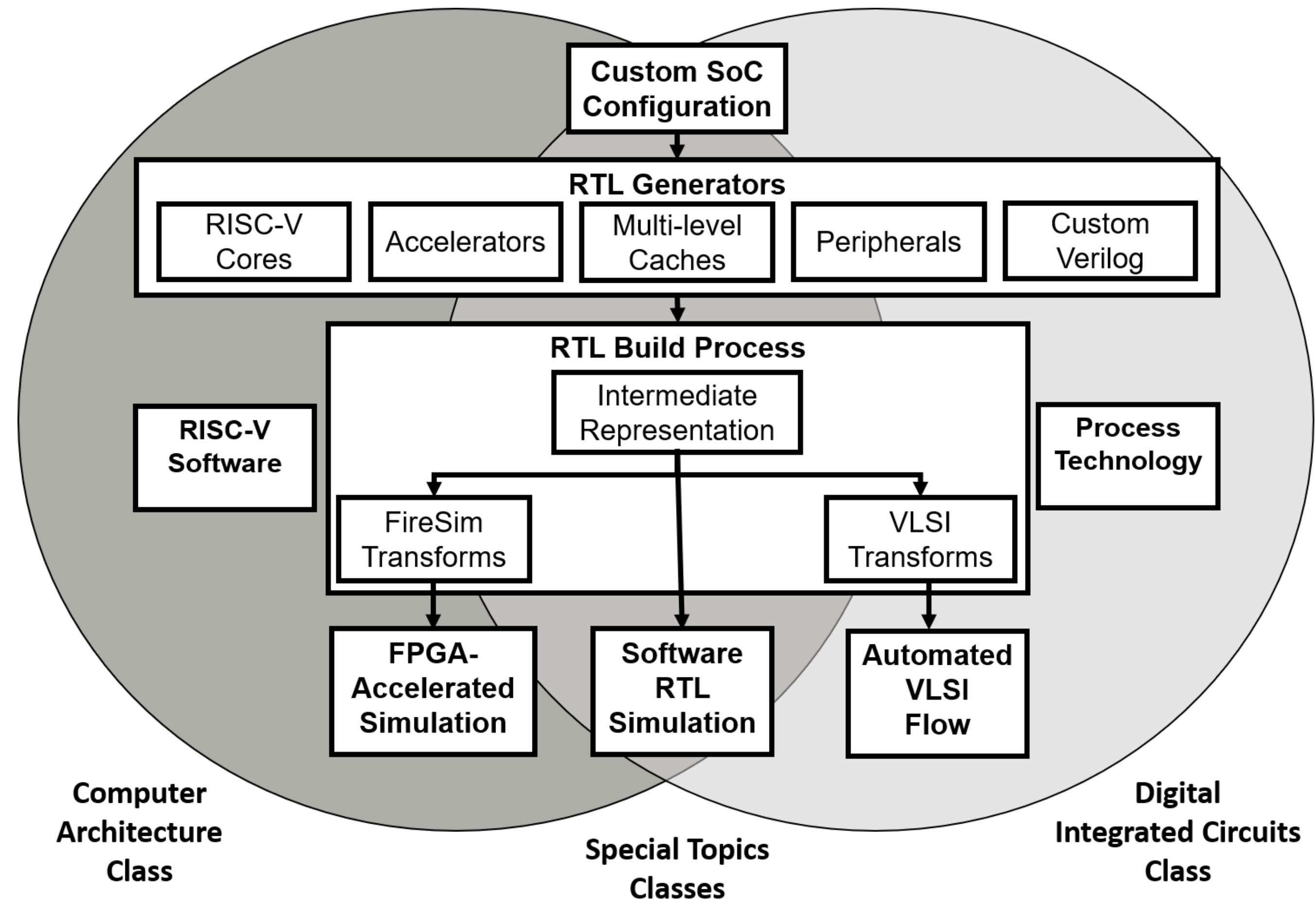
[1] https://inst.eecs.berkeley.edu//~ee290-2/sp21/

Cloud-Hosted FPGAs

- Remote Instruction
  - Class enrollment not limited by FPGA boards and lab space
  - No interruption to assignments during COVID lockdown
- Billing and class management questions
  - Usage cost vs. capital cost, and their relation to student effort
  - Account management and monitoring
  - Overspending

Multi-Class Curricula

- Classes
  - EE290 – Hardware for Machine Learning
  - CS152/252 - Computer Architecture
  - EE241B – Advanced Digital Integrated Circuits
  - EE194 - 28nm SoC for IoT
- Spring 2020: 10%-15% student overlap
- Reduced ramp-up time
- More comprehensive projects



Acknowledgments

- Graduate Student Instructors: David Biancolin, Albert Magyar, Abraham Gonzalez, Jerry Zhao, Daniel Grubb, Harrison Liew
- The Chipyard and FireSim development teams
- NSF CCRI Award 2016662
- ADPET Lab sponsors and affiliates
- The students of UC Berkeley classes CS152/252, EE290-2, EE241B, EE-194

Conclusion

- Practical experience in HW/SW co-design and full-system topics
- Open-source hardware generators
  - Broad spectrum of uses – from architecture to circuits and design automation
  - Accommodate differing levels of prerequisite knowledge
- Accessible FPGA-emulation
  - Full-system experiments: hardware + software
  - Deterministic modeling and characterization
  - Remote instruction
- Multi-class curricula