CSCI 5204/EE 5364 Programming Assignment 1: Microarchitecture Simulation and Testing Due Oct 5, 2025, 11:59pm

Assignment Goals

A goal of this assignment is to learn how to use microarchitecture simulators at both the register transfer level (RTL) and event-driven abstraction levels. A second goal is to learn how to use a standard testing framework, such as COCOTB. At the end of this assignment, you would have started developing intuition and skill for evaluating both performance and correctness of microarchitecture components. The programming assignment can be done in groups of up to three students. Please submit a clear and concise report on Canvas, with only one submission per team.

What You Will Do

In this assignment, you will carry out a performance study for in-order processors, out-of-order processors. You will also test an ALU using the COCOTB, a standard python-based hardware testing framework:

- **RTL Simulators:** Use BlueSim, Icarus Verilog (iverilog), or Verilator to simulate:
  - Piccolo (in-order 32-bit processor).
  - Flute (in-order 64-bit-capable processor).
  - Toooba (out-of-order 64-bit processor).
- **Event-driven Simulator:** Use GEM5 with its O3 CPU model for event-driven microarchitecture simulation.
- **Testing:** Test an ALU unit using the COCOTB framework. This framework allows for testing specific inputs and also can generate random inputs for a device under test (DUT).

CSELabs Setup

The following tools are used in this project:

- **Verilog Simulators:** Verilator (preferred) and/or Icarus Verilog (`iverilog`).
- **Bluespec Compiler (for Bluesim or regenerating RTL):** `bsc`.

The handout contains the binaries for `verilator` and `bluesim`, whereas, `iverilog` is installed by default in CSELabs machines.

NOTE: Using Scratch. We need to use the scratch directory due to the significant amount of disk space used by simulators. The scratch is cleaned up every few days or weeks. Please do not store any simulation result files in here long term. Any simulation results or logs should be stored in your home folder, whose contents are always backed up.

1 How to Build and Run Piccolo, Flute, and Toooba

Access a CSELabs machine (e.g., Keller 1-250). Make a working folder in the scratch directory. Then, download and decompress the provided handout. The terminal commands are as follows. Replace <X500> with your actual X500 ID.

```
bash
pip install gdown
mkdir -p /export/scratch/users/<X500>
cd /export/scratch/users/<X500>
gdown --id  1_h0-LZNxbU-v2paqDoEo7UVcR6Cz-YqC
tar xvzf handout_5204_Fall2025.tar.gz
export PATH=/export/scratch/users/<X500>/bin:$PATH
gdown --id 1B1oCjq7ijsJU7pRJFiNFL5Oxo_V1oyFj
tar xvzf patch_for_verilator.tar.gz
export VERILATOR_ROOT=/export/scratch/users/<X500>/verilator-install
chmod +x *.sh
```

1.1 Piccolo (RV32 in-order)

Clone and Patch

```
git clone https://github.com/bluespec/Piccolo.git
```

After cloning Piccolo, patch it using the provided script.

```
./patch_piccolo.sh
```

Option A: Verilator

```
# Use a provided Verilog build directory for RV32ACIMU:
cd Piccolo/builds/RV32ACIMU_Piccolo_verilator
make simulator           # builds the Verilator executable
make test                # runs a default RISC-V ISA test
make isa_tests           # runs the relevant ISA regression suite
```

Option B: Bluesim

```
cd Piccolo/builds/RV32ACIMU_Piccolo_bluesim
make compile simulator   # builds the Bluesim executable
make test
make isa_tests
```

Running a specific ISA test

```
# Replace TEST=... with any test present under Tests/isa/
make test TEST=rv32ui-p-add
```

Result

A `Logs` folder is created which contains the results of all the tests. It contains debug information about the execution of a test RISC-V binary. At the bottom, it shows the simulated cycles/second.

1.2 Flute (RV64GC in-order; RV32CI variants also provided)

Clone and Patch

```
git clone https://github.com/bluespec/Flute.git
```

After cloning Flute, patch it using the provided script.

```
./patch_flute.sh
```

Common provided builds

- RV64GC, Bluesim: `Flute/builds/Flute_RV64GC_MSU_WB_L1_L2_bluesim_tohost`
- RV64GC, Verilator: `Flute/builds/Flute_RV64GC_MSU_WB_L1_L2_verilator_tohost`
- RV32CI, Bluesim: `Flute/builds/Flute_RV32CI_MU_WT_L1_bluesim_tohost`
- RV32CI, Icarus: `Flute/builds/Flute_RV32CI_MU_WT_L1_iverilog_tohost`

Verilator build

```
cd Flute/builds/Flute_RV64GC_MSU_WB_L1_L2_verilator_tohost
make simulator
make test
make isa_tests
```

Bluesim build

```
cd Flute/builds/Flute_RV64GC_MSU_WB_L1_L2_bluesim_tohost
make compile simulator
make test
make isa_tests
```

Run a specific test

```
make test TEST=rv64ui-p-add
```

Result

A `Logs` folder is created which contains the results of all the tests. It contains debug information about the execution of a test RISC-V binary. At the bottom, it shows the simulated cycles/second.

1.3 Toooba (RV64GC out-of-order)

Clone and Patch

```
git clone https://github.com/bluespec/Toooba.git
cd Toooba
# Also initialize the BlueStuff submodule once:
git submodule update --init --recursive
cd ..
gdown --id 1dTKq82tK2E7YNTjIN2YLlJup2hZ7XTor
tar xvzf patch_toooba_time.tar.gz
```

After cloning Toooba and downloading the patch, patch it using the provided script.

```
./patch_toooba.sh
```

Option A: Verilator

```
cd Toooba/builds/RV64ACDFIMSU_Toooba_verilator
make simulator
make test
make isa_tests
```

Option B: Bluesim

```
# Pattern matches other repos:
cd Toooba/builds/RV64ACDFIMSU_Toooba_bluesim
make compile simulator
make test
```

Run a specific test

```
make test TEST=rv64ui-p-add
```

Result
A `Logs` folder is created which contains the results of all the tests. It contains debug information about the execution of a test RISC-V binary. At the bottom, it shows the simulated cycles/second.

2 How to Build and Run GEM5 O3 and PARSEC

To set up the files for your GEM5 simulations, run the following commands.

```
bash
cd /export/scratch/users/<X500>/parsec-tests
tar xvzf parsec_checkpoints_single_core.tar.gz
mkdir ~/.cache
ln -s /export/scratch/users/<X500>/parsec-tests/resources /home/<X500>/.cache/gem5
```

Once the above is executed, clone GEM5 and build GEM5 and simulate any desired PARSEC benchmark.

```
bash
cd /export/scratch/users/<X500>/parsec-tests
gdown --id  1DxShZ1JWICIAdX2mMMfrDlKS6W644fjb
chmod +x run_parsec.sh
git clone --branch v23.0.0.0 https://github.com/gem5/gem5.git
cd gem5
scons build/X86_MESI_Two_Level/gem5.opt -j8
cd ..
./run_parsec.sh
```

Result: The results of the PARSEC simulation are found in a folder corresponding to the simulated benchmark. For example, if you simulated `blackscholes`, the simulation results are in `blackscholes_simulation/m5out/stats.txt`. Search for the keyword `cpi` to get the cycles-per-instruction metric for your simulation. It is possible to modify `run_parsec.sh`'s `SIM_INSTS` and `BENCHMARK` variables using a text editor, to specify a particular instruction count and benchmark.

Modifying GEM5: You can modify GEM5's O3 parameters in the associated Python file. This includes parameters such as ROB size, issue width and LSQ size. Together, these experiments help to determine an appropriate sizing of the O3 core. The file you want to modify is `gem5/src/cpu/o3/BaseO3CPU.py`. The variables you want to modify are `LQEntries`, `SQEntries`, `issueWidth` and `numROBEntries`. The default values are 32, 32, 8 and 192. Please try values 16 and 64 for `LQEntries` and `SQEntries`, values 4 and 12 for `issueWidth` and 128/384 for `numROBEntries`.

Once you modify any of these entries, it is required to recompile the GEM5 processor. Then, you can run the simulation again to obtain new simulation results. Since the new results will overwrite previous simulation results, remember to backup those simulation results.

3 How to run the ALU tests

To run the ALU tests, create a virtual python environment, install COCOTB (a testing framework) and then run `iverilog`. The commands to use are as follows.

```
bash
cd /export/scratch/users/<X500>/HW-cocotb-alu
```

```
python3 -m venv .venv && source .venv/bin/activate
pip install --upgrade pip
pip install cocotb cocotb-coverage pytest
cd sim
make SIM=icarus
```

Tests will be carried out on the ALU (`rtl/alu.v`) using the COCOTB testbench specified in `sim/test_alu.py`. A test result will be generated, indicating either PASS or FAIL. If there is a FAIL message, it indicates that one of the test cases went wrong. If there is a PASS message it means that no problems were found from the provided test cases.

What You Submit

Your submission must be a PDF file, named `5204_Fall2025_ProgrammingAssignment1.pdf` that contains:

a. The **names, emails and X500s** of all the team members.

b. **Piccolo**, **Flute**, and **Toooba**: Present the *cycles-per-second* for each simulator (Verilator and Bluesim). Which simulator works faster and by how much? Do you have a hypothesis as to why?

c. **GEM5 O3 + PARSEC**: Plot the CPI observed when the core is configured for each size metric and hypothesize why the performance has changed in the observed manner. Also plot the `hostTickRate` metric. Compared to the RTL simulation, is the `hostTickRate` higher or lower?

d. **ALU Tests**: Report pass/fail results for the provided ALU implememtation; document any observed bugs with minimal logs. How will you fix the bug (if any)?

Submission Guidelines

- Include plots, tables, and explanations, to present significant amounts of data in an accessible way.
- Ensure all plots are clearly labeled with axes and units.
- Keep the report concise (less than 2000 words preferable).

Evaluation Criteria

- The experimental results presented in the report. (50 points)
- Explanation of the performance trends. (40 points)
- Clarity and organization of the submitted report. (10 points)