ELE/COS 475/575 Computer Architecture, Fall 2018
Lab 3: Superscalar RISCV Processor
(adapted from ECE 4750 at Cornell University)
Due Date: Monday, November 12, 2018 at 1:30pm

In this lab, your task is to create a two-wide superscalar in-order processor. We will provide you with a working datapath of such processor, and you will need to add the processor control logic, instruction steering logic, and a scoreboard to enable your processor to properly execute two instructions per cycle.

1. Getting Started

First, download and unpack the lab tarball from Blackboard, or get it from **adroit**:

```
cd ele475
tar -xzf /home/ee475/dropbox/ele475-lab3.tar.gz
cd ele475-lab3
export LAB3_ROOT=$PWD
```

If you are working on adroit, also make sure to source the correct environment for the lab:

```
source /home/ee475/env.sh
```

There will be nine directories in the lab root. The purpose of each directory is outlined below:

- *build*: Build directory for Verilog source code
- *imuldiv*: Mul/Div unit inherited from lab 2
- *riscvstall*: Pipelined **RISC-V** processor with stalling source code
- *riscvbyp*: Pipelined **RISC-V** processor with bypassing source code
- *riscvlong*: Pipelined **RISC-V** processor integrated with pipelined muldiv unit and bypass logic
- *riscvdualfetch*: Superscalar **RISC-V** processor source code with dual-fetch, but single instruction issue
- *riscvssc* (created by you from *riscvdualfetch*): Superscalar **RISC-V** processor source code
- *tests*: Assembly test build system
    - *riscv*: RISC-V assembly tests
    - *scripts*: miscellaneous scripts for build system
- *ubmark*: benchmarks for evaluation
- *vc*: Additional Verilog components

Most directories are the same as in the previous lab. The new directories are *riscvdualfetch* and *riscvssc*. The *riscvdualfetch* directory begins with the datapath for a two-wide superscalar, but without the scoreboard, steering logic, and control logic.

We will begin as usual by compiling the reference processors and running the **RISC-V** assembly tests:

```
cd $LAB3_ROOT/tests
mkdir build && cd build
../configure --host=riscv32-unknown-elf
make
../convert
cd $LAB3_ROOT/build
make*
make check
make check-asm-riscvstall
make check-asm-rand-riscvstall
make check-asm-riscvbyp
make check-asm-rand-riscvbyp
```

The methodology is the same as in the previous lab. Refer to the lab 2 handout for details about the test suite.

***Notice**: when you try to run make for the first time, you will get errors because the control unit included in *riscvdualfetch* is not complete, and some of the macros are not defined. You first need to make some modification before being able to compile it and run tests. However, you can perform above checks for other designs.

2. 2-Wide Superscalar, In-order Processor Description

For this lab, we provide you with a **7-stage, two-wide superscalar, in-order** processor without the control logic, scoreboard logic, or instruction steering logic. Each stage of the new superscalar processor should be able to handle up to two instructions at a time. The execute stages will consist of two heterogeneous functional units. The first pipeline **(A)** is capable of executing all ALU operations, memory operations, control flow operations, and multiply/divide operations. The first **(A)** pipeline has a four stage multiply/divide unit included. The second pipeline **(B)** is only capable of executing simple ALU operations (no multiply/branch/jump/divide/memory instructions).

Presented below is a high-level description of the superscalar in-order processor pipeline. Following that, we provide the instructions for each of the two parts of this lab.

Fetch stage:

The fetch stage should fetch two instructions per cycle when it is not stalled. We have provided for you a triple-ported memory unit that replaces the dual-ported unit from the previous lab. This will ensure that up to three memory accesses (two instructions, one data) can be performed on each cycle without the memory unit causing a structural hazard. The number of ports you will use will change for different parts of the lab.

We will not worry about instruction alignment issues in this lab. It is safe to assume that all accesses are aligned properly, and that the memory unit will properly handle all requests. Keep in mind, however, that real systems would need to check for properly aligned instructions, and they would have trouble dealing with unaligned instructions, as we discussed in class.

Decode/Issue stage:

In lecture, we described the **I4** and **I2OI** processors architecture as having a separate decode and issue stage. However, for the sake of continuity from the previous lab, **we will combine the decode and issue stages from the lecture into a single Decode/Issue stage for the purposes of this lab.** This is meant to make your job easier by not requiring you to rewrite all of the logic from the previous lab in order to incorporate a new stage. However, just remember that in a real processor, the stages would likely be separated in order to keep the cycle time reasonable.

Register File:

In order to prevent the register file from frequently becoming a structural hazard, we have provided for you a six-ported register file. This will allow for four reads (two for each execute pipeline) and two writes per cycle. The provided register file will not detect data hazards by itself, however. Writing to the same register has undefined behavior. It is the job of your stalling and bypassing logic to ensure that register values do not get overwritten incorrectly.

Steering Logic:

The decode logic itself will be similar as in the previous lab. However, since our two execute pipelines are functionally heterogeneous, you will need to add **steering logic** that issues instructions to the two pipelines correctly. Specifically, only simple-ALU instructions can be issued to the second **(B)** pipeline. If two instructions that cannot be paired given the constraints of the pipeline are decoded at the same time, the second instruction will have to stall in the decode stage for one cycle in order to be sent down the correct pipeline.

Steering Logic			
Instruction 0 Type	**Instruction 1 Type**	**Instruction 0 Dest.**	**Instruction 1 Dest.**
ALU	ALU	A	B
ALU	Non-ALU	B	A
Non-ALU	ALU	A	B
Non-ALU	Non-ALU	A	Stall, then A

The steering logic should be able to decode two instructions and issue two instructions on every cycle if there are no structural hazards. If the second fetched instruction is dependent on the first instruction, it cannot issue simultaneously with the first instruction. **Also, to simplify the design, if two fetched instructions write to the same destination register, the second instruction will stall while the first one issues.** This avoids two writes occurring to the same register on the register file in the same cycle.

Scoreboard:

You will also need to create a scoreboard that should be read in the decode/issue stage. This will be needed to ensure that all of the input registers for instructions at the decode/issue stage are ready to be read and where to bypass (pipeline and stage) a particular data value from. Because all instructions write the register file at the end of the pipeline, the scoreboard is not needed to detect writeback hazards. A scoreboard should be used to determine bypassing and data stall information because it is a smaller structure than the stall logic used in lab 2.

	Scoreboard						
Dest. Reg.	**Pending (Y/N)**	**Functional Unit (Pipeline A/B)**	**4**	**3**	**2**	**1**	**0**
R0	1	0	0	1	0	0	0
R1	0	1	0	0	0	0	0
…	…		.	.	.	.	.

Execute stage:

As mentioned earlier, the execute stage should be broken into two parallel pipelines. The first **(A)** is a fully-functional pipeline identical to the *riscvlong* execute stages. The second **(B)** is a pipeline which only implements simple ALU operations. There will be no *muldiv* and no memory unit in the **B** pipeline.

Memory (X1) stage:

Stage **X0** is followed by **X1**, which will be the memory stage as usual. As mentioned earlier, we have provided you with a triple-ported memory unit for this lab.

X2/X3 stage:

Stage **X2** and **X3** are added only for timing purpose. Though there is no *muldiv* unit in the **B** pipeline, the two pipelines both have stage **X2** and **X3**, because we are still implementing an in-order processor.

Writeback stage:

The writeback stage will remain the same as in the previous lab. As mentioned earlier, we have provided register file with two write ports so that the number of write ports will not in itself be a structural hazard.

3. Part 1: Two-Wide Superscalar Processor with Dual Fetch, but Single Issue

In *riscvdualfetch* we have provided a datapath with most of the *riscvlong* datapath logic duplicated. For **Part 1**, we have removed most of the logic from the control unit of the processor. **Your task in Part 1 is to implement a processor which can fetch two instructions per cycle and then steer instructions down the pipeline one at a time.** Your processor pipeline will alternate between the two instructions which have been fetched. To get started with this task, we suggest getting basic instructions flowing down the pipeline. We have provided a core control file which provides most of the control logic duplicated with important portions missing. For **Part 1**, the goal is to have instructions fetched two at a time, but every other cycle stall the front end of the pipeline. Alternate between steering the first instruction down pipeline **A** and steering the second instruction steer down pipeline **A**. Note that at this point, you will not have to detect instruction steering problems due to the heterogeneity of the pipelines or interdependence of the two fetched instructions. However, you will be doing that in **Part 2**. Be sure to incrementally test this pipeline. For this dual-fetch, single issue pipeline, remember that branches and jumps will have to kill instructions, including the second fetched instruction. For **Part 1**, you can elect to use the same bypassing and stalling logic from previous labs, or you can use a scoreboard to detect data dependent stalls. In **Part 2**, you will have to build a scoreboard. Modify the control logic to pass all tests executing instructions with the same execution bandwidth as *riscvlong* from Lab 2. One point that you should be aware of is that you may want to inject an *invalid instruction* into the first instruction after it issues to prevent problems of executing a single instruction twice. Note that the *nop* instruction in RISC-V is essentially an *addi* instruction, so be careful if you use *nop* as an *invalid* instruction.

Some changes to the processor that you should be aware of:

- The *pc_plus4_*hl* signals have been replaced with *pc_plus8_*hl* equivalents to signify the fact that we are fetching two instructions per cycle
- The provided register file now has six ports.
- Similarly, the memory unit has been updated with an extra read port.
- The *br_targ*, *j_targ*, and *jr* signals should be de-multiplexed from the two decoded instructions as each is issued.

- If the first of two simultaneously decoded instructions turns out to be a taken branch or jump, the second instruction (which should be stalling in the decode stage) will need to be killed (i.e. just dropped).
- The pipeline diagrams used for debugging execution are drawn by code near the end of *riscvdualfetch-sim.v* and *riscvdualfetch-randdelay-sim.v*. We have corrected this code to display all 7 stages of both pipelines in parallel. However, if you choose signal names which do not match exactly, then you will see a compilation error. In that case, either change the names of your signals to match the ones in the debugger, or change the names of the signals being printed within the debugger.

4. Part 2: Two-Wide Superscalar Processor

For this section, the primary change will be the ability to issue and execute two simultaneous instructions. This will necessitate the addition of full instruction steering logic and a scoreboard. You will need to implement the steering logic in the decode/issue stage to make sure that instructions are only issued to valid pipelines. **The steering logic must maintain RAW dependencies between two simultaneously fetched instructions.** The control logic should also stall the second fetched instruction if two simultaneously fetched instructions write to the same register.

Begin by creating a new *riscvssc* folder and copying over your new code from the *riscvdualfetch* folder, renaming as needed. Replace any instances of *riscvdualfetch* in the code with references to *riscvssc* instead. The following sequence of bash commands will accomplish this for you:

```
mkdir $LAB3_ROOT/riscvssc
cd $LAB3_ROOT/riscvssc
cp ../riscvdualfetch/* .
for filename in *; do mv $filename ${filename/riscvdualfetch/riscvssc};
done
sed -i “s/riscvdualfetch/riscvssc/g” *
```

Before going any further, rebuild your project to make sure that everything is copied correctly.

```
cd $LAB3_ROOT/build
make
```

First, create the steering logic. Make sure to add the correct stalling logic so that your processor properly handles two non-ALU instructions decoded simultaneously. Run all of the tests on your processor as is, making sure that it executes correctly.

To simplify the design, if two instructions are writing to the same register within a pair of instructions, simply stall the second instruction. This is to avoid the problem that two instructions that are to be concurrently issued may need to write to the same register in the register file. This also simplifies bypass calculations.

Lastly, at the bottom of the **-CoreCtrl.v* files, there is a piece of (non-synthesizable!) code which calculates the number of instructions that have been executed in the pipeline. However, this code does not account for the superscalar width of the pipeline. Modify the condition of the instruction count incrementer `num_inst = num_inst + 1;` so that it measures the program instruction count correctly.

For the report, create at least one new assembly test that demonstrates that the processor correctly detects a WAW hazard in simultaneously decoded instructions but still continues to issue and execute correctly. Make sure to explain exactly how this is demonstrated by your new unit test.

Also, for the report, create at least one new assembly test that demonstrates that the processor correctly handles two non-ALU instructions that are decoded simultaneously. Make sure to explain exactly how this is demonstrated by your new unit test.

Once your processor passes all of the existing and custom-made tests and micro-benchmarks, congratulations!

5. Testing Methodology

We will provide for you the **RISC-V** assembly tests and microbenchmarks from the previous lab. **You will need to create multiple assembly tests to verify the correctness of your processor as described in the previous sections.** For instructions on how to do this, refer to the handout for the previous lab.

6. Evaluation

For this lab, you will be comparing the performance of the *riscvdualfetch* and *riscvssc* processors to the *riscvlong* processor. Report the cycle count, as well as the IPC of each benchmark for all of the above microarchitectures. Analyze the performance differences between the implementations in the report and discuss in which situations each of the modifications is helpful or harmful and in which situations it will have limited effect. What parts of the Iron Law of Processor Performance are affected?

7. Lab Report

In addition to the source code for the lab, you will need to submit a lab report which includes the following sections:

- **Abstract**: introductory paragraph summarizing the lab
- **Design**: describe your implementation, justifications for design decisions (if any), deviations from the prescribed datapath, discussion of any extensions
- **Testing Methodology**: describe how you tested the modules and your overall testing strategy–justify the effectiveness of your assembly test(s) in testing the functionality of the processor
- **Evaluation**: report your simulation results and cycle counts comparing *riscvdualfetch*, *riscvssc*, and *riscvlong*

- **Discussion**: comparison and analysis of benchmark results, discussion of tradeoffs of using dual fetch and dual issue.
- **Figures**: updated datapath of the new architecture

The report can be a maximum of 4 pages, you will be penalized if you go over. Figures count against this limit.

8. Deliverables

For submission, please turn in a .tar.gz file of your working directory without changing the original directory structure. All source files should be located in `$LAB3_ROOT/riscvdualfetch` and `$LAB3_ROOT/riscvssc`. Please be sure to delete any generated waveforms or compiled code by running `make clean` in each of the build directories. Also, delete the build directories of the *tests* and *ubmark* directories.

```
cd $LAB3_ROOT/build
make clean
cd $LAB3_ROOT/tests
rm -rf build
cd $LAB3_ROOT/ubmark
rm -rf build
```

You can execute the following commands to make a tarball of your completed lab, assuming that you did not change the name of the lab root directory.

```
cd $LAB3_ROOT
cd ..
tar -cvzf {netid}-lab3.tar.gz ele475-lab3
```

Below is a list of files we will need to submit to meet the expectations of this lab:

- *riscvdualfetch* source code
- *riscvssc* source code
- Custom assembly tests
- Datapath diagram of heterogeneous dual-issue superscalar **I4** processor
- Diagram showing the scoreboard that you ended up implementing
- Lab report

9. Tips

- Use incremental development–never code everything at once and hope it works!
- Use the unit testing framework!
- Always draw the hardware before you start coding!
- Clearly define the interaction between the control logic and the *datapath*!

- Feel free to remove any signal or logic from the control logic and declare your own. The provided control unit is just a skeleton for you to fill in.
- Start early – you will not be able to finish if you leave this assignment to the last minute!
- If you can't get everything working, be sure to thoroughly explain how far you did get in the lab report in addition to the debugging strategy you used!