ECE 4750 Computer Architecture, Fall 2024
Lab 2: Pipelined Processor

School of Electrical and Computer Engineering
Cornell University

revision: 2024-09-19-17-55

In this lab, you will design two pipelined processor microarchitectures for the TinyRV2 instruction set architecture. After implementing all TinyRV2 instructions, your processors will be capable of executing simple C programs that do not use system calls. The baseline design is a five-stage processor pipeline that uses stalling to resolve data hazards and the alternative design is a five-stage processor pipeline that uses bypassing to improve the processor performance. You are required to implement the baseline and alternative designs, verify the designs using an effective testing strategy, and perform an evaluation comparing the two implementations. **The milestone for this lab is to complete as many instructions as you can for the baseline design along with an initial set of directed tests. You should consult the course lab logistics document for more information about the expectations for all lab assignments and how they will be assessed.**

This lab is designed to give you experience with:

- instruction set architecture;
- basic pipelined processor microarchitecture;
- microarchitectural techniques for handling data and control hazards;
- interfacing processors and memories;
- abstraction levels including functional- and register-transfer-level modeling;
- design principles including modularity, hierarchy, and encapsulation;
- design patterns including message interfaces, control/datapath split, and pipelined control;
- agile design methodologies including incremental development and test-driven development.

This handout assumes that you have read and understand the course tutorials. To get started, login to an `ecelinux` server and source the setup script. Since you have already cloned your lab group's repository, use `git pull` to ensure you have any recent updates before running all of the tests.

```
% source setup-ece4750.sh
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase
```

You can run all of the tests in the lab like this:

```
% mkdir -p sim/build
% cd sim/build
% pytest ../lab2_proc
```

All of the tests for the provided functional-level model should pass, and the tests for a few instructions we have already implemented for you should pass on the baseline design. For this lab you will be working in the `lab2_proc` subproject which includes the following files:

- `tinyrv2_encoding.py`   – Classes/functions for the TinyRV2 ISA encoding (FL model)
- `tinyrv2_semantics.py`   – Classes/functions for the TinyRV2 ISA semantics (FL model)
- `ProcFL.py`   – FL processor

- `DropUnit.v`   – Verilog RTL unit for dropping inst mem response on squash
- `ProcDpathAlu.v`   – Verilog RTL for ALU used in dpath
- `ProcDpathAlu.py`   – Python wrapper for ALU
- `ProcDpathImmGen.v`   – Verilog RTL for immediate generation unit used in dpath
- `ProcDpathImmGen.py`   – Python wrapper for immediate generation unit

- `tinyrv2_encoding.v`   – Verilog RTL helper constants, functions for TinyRV2 ISA
- `ProcBaseDpath.v`   – Verilog RTL stalling processor's datapath
- `ProcBaseCtrl.v`   – Verilog RTL stalling processor's control unit
- `ProcBase.v`   – Verilog RTL stalling processor
- `ProcBase.py`   – Python wrapper for stalling processor
- `ProcAlt.v`   – Verilog RTL for fully bypassed processor
- `ProcAlt.py`   – Python wrapper for fully bypassed processor

- `proc-sim`   – Processor simulator for evaluation
- `__init__.py`   – Package setup

- `test/tinyrv2_encoding_test.py`   – Tests for inst encoding
- `test/ProcDpathAlu_test_test.py`   – Tests for alu
- `test/ProcDpathImmGen_test_test.py` – Tests for immediate generation unit
- `test/Proc<impl>_csr_test.py`   – Tests for csr related insts (`<impl>` = FL, Base, Alt)
- `test/Proc<impl>_rr_test.py`   – Tests for reg-to-reg insts (`<impl>` = FL, Base, Alt)
- `test/Proc<impl>_rimm_test.py`   – Tests for reg-to-imm insts (`<impl>` = FL, Base, Alt)
- `test/Proc<impl>_mem_test.py`   – Tests for memory insts (`<impl>` = FL, Base, Alt)
- `test/Proc<impl>_jump_test.py`   – Tests for jump insts (`<impl>` = FL, Base, Alt)
- `test/Proc<impl>_branch_test.py`   – Tests for branch insts (`<impl>` = FL, Base, Alt)
- `test/harness.py`   – Test harness
- `test/inst_<inst>.py`   – Test cases for each instruction
- `test/inst_utils.py`   – Utility functions and templates for assembly fragment
- `test/__init__.py`   – Package setup

- `ubmark/`   – Data and assembly code for microbenchmarks

This lab uses of your solution from the first lab. If you did not complete the first lab, you can ask to use the instructor's baseline solution, but in lab 4 this will result in a performance penalty since the baseline solution is not well optimized. Feel free to discuss this with the instructor to decide what is best for your group.

1. Introduction

Pipelining is a design pattern that enables overlapping the execution of multiple transactions. A pipelined microarchitecture is divided into stages with each stage performing specific tasks in a similar manner to car manufacturing in an assembly line. Compared to a single-cycle processor, pipelining reduces the cycle time (clock period) while still approximately achieving an average of one cycle per instruction (CPI). Compared to an FSM processor, pipelining reduces the CPI while approximately achieving a similar cycle time (clock period). However, pipelining introduces various hazards that complicate the control logic. In this lab, you will implement and evaluate two five-stage pipelined processor microarchitectures that avoid hazards in two different ways: (1) by stalling, and (2) by bypassing. Later in the course, you will see how modern processors combine pipelin-

ing with more sophisticated techniques to exploit instruction-level parallelism, enabling improved performance at the cost of increased energy, area, and complexity over this lab.

Figure 1: ADD Instruction from RISC-V ISA Manual – The RISC-V ISA manual specifies the assembly syntax, semantics, and encoding for every instruction in the RISC-V ISA.

```
* ADD

  - Summary    : Addition with 3 GPRs, no overflow exception
  - Assembly   : add rd, rs1, rs2
  - Semantics  : R[rd] = R[rs1] + R[rs2]
  - Format     : R-type

   31          25 24      20 19      15 14  12 11      7 6           0
  +------------+---------+---------+------+---------+-------------+
  | 0000000    | rs2     | rs1     | 000  | rd      | 0110011     |
  +------------+---------+---------+------+---------+-------------+
```

We will be using the RISC-V instruction set architecture (ISA) for this course and all the labs. More specifically we will be using the Tiny RISC-V ISA subset which is suitable for teaching. The Tiny RISC-V ISA was introduced in lecture, and both the full RISC-V ISA manual and the Tiny RISC-V ISA manual are available on Canvas. As an example, the specification from the Tiny RISC-V ISA manual for the add instruction is shown in Figure 1. You will be implementing the TinyRV2 subset which is sufficient for executing simple C programs. The list of instructions that constitute TinyRV2 are below.

- CSR   : csrr, csrw
- Reg-Reg : add, sub, mul, and, or, xor, slt, sltu, sra, srl, sll
- Reg-Imm : addi, ori, andi, xori, slti, sltiu, srai, srli, slli, lui, auipc
- Memory   : lw, sw
- Jump   : jal, jalr
- Branch   : bne, beq, blt, bltu, bge, bgeu

We have provided you a complete functional-level model of a TinyRV2 processor. You can find this model in `ProcFL.py`. The functional-level model executes one instruction at time "magically". It is not synthesizable and is purely meant to be used as a reference design. This kind of functional-level model is often called an "instruction-set-architecture emulator" (or ISA emulator) since it simulates just the ISA with no microarchitectural details.

Figure 2 shows a block-level diagram illustrating how the baseline and alternative designs are integrated with a test source, test sink, and test memory for testing and evaluation. The interfaces for the FL, baseline, and alternative designs are identical. We will load a program (and potentially some data) into the test memory before resetting the processor. Once the processor starts execution, we can send test data into the processor using the test source and the `csrr` instruction, and we can have the processor verify data using the test sink and the `csrw` instruction.

The processor interface makes extensive use of the latency insensitive stream interfaces that are implemented using the val/rdy microprotocol. There are six different stream interfaces

- `mngr2proc`   : stream from test source to processor
- `proc2mngr`   : stream from processor to test sink
- `imem_reqstream` : instruction memory request stream

- `imem_respstream`   : instruction memory response stream
- `dmem_reqstream`   : data memory request stream
- `dmem_respstream`   : data memory response stream

The processor interacts with the memory using memory messages. The message format for memory requests and responses are shown in Figure 3. Corresponding Verilog `structs` are defined in `vc/mem-msgs.v` included within the lab release. Memory requests use fields to encode the type (e.g., loads read, stores write), the address, the length of data in bytes, and the data. Memory responses use fields to encode the type (e.g., read, write), the length of data in bytes, and the data. The data field is fixed at 32-bits or four bytes. If the length field is one then only the least significant byte of the data field (i.e., bits 7–0) is valid. If the length field is two then only the least significant two bytes of the data field (i.e., bits 15–0) are valid. If the length field is zero then all four bytes are valid. Both memory requests and responses have an eight-bit opaque field, which is reserved for use by the requester. Memory systems must ensure that the exact same opaque field is included in the corresponding response. For now you should always set the opaque field to zeros. Memory response messages also include a test field that is for testing memory systems. For now you can ignore this field.

The processor sends a memory request message via a stream interface to the memory, and then the memory will send a response message back to the processor one or more cycles later. You can assume that the memory will always take at least one cycle (i.e., there will be one clock edge between when the request is sent and when the response is received), but you cannot assume how many cycles it
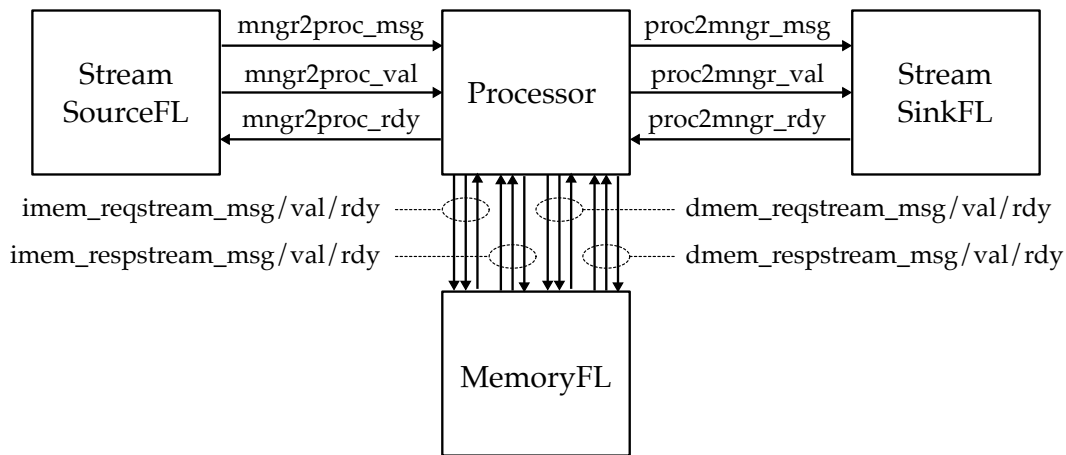
Figure 2: Processor System – The processor is integrated with a source, sink, and memory for testing and evaluation. Notice the use of the **extremely important** `val and rdy signals`.
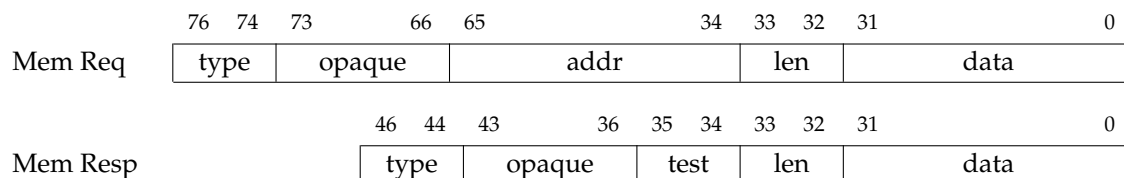
Figure 3: Memory Request/Response Message Formats – Memory request messages are sent from the processor to memory. Memory response messages are sent from memory back to the processor.

will take for the response to return. The response could return in one cycle or 100 cycles. You must also correctly deal with situations where the memory is not ready to accept a request. This means you must carefully handle the val/rdy signals to ensure correct operation. For example, your designs will need to wait if the manager or memory is not ready yet, and your designs will also need to wait if a message from the manager or memory has not arrived yet. Using latency insensitive stream interfaces will enable us to easily compose our processor designs with the memories and networks we design later in the course.

2. Baseline Design

The baseline design for this lab assignment is a five-stage stalling processor that supports the TinyRV2 ISA. As with the first lab, we will be decomposing the baseline design into two separate modules: the datapath which has paths for moving data through various arithmetic blocks, muxes, and registers; and the control unit which is in charge of managing the movement of data through the datapath. Unlike the first lab, the control unit will not use an FSM but will instead use pipelined control logic. Because the processor design is significantly more complicated than the previous designs we have worked on, we have decided to place the datapath module, control unit module, and the parent module that connects the datapath and control unit together in three different files.

Our pipelined processors have five stages: F – fetch instructions, increment PC; D – decode instructions, read register operands, handle jumps; X – arithmetic operations, address generation, branch comparison; M – access data memory; and W – write register file. The datapath for the baseline design is shown in Figure 13. The blue boxes and signals indicate the control and status signals between the control and datapath units. To help you get started, we have already implemented three primary instructions (add, lw, bne). We have also implemented the csrr (move from the test manager) and csrw (move to the test manager) instructions which are used for testing. Figure 14 illustrates the datapath that we provide to get you started.

Your datapath module should instantiate a child module for each of the blocks in the datapath diagram; in other words, you must use a structural design style in the datapath. You will need to add and/or modify datapath components as you support more TinyRV2 instructions in your baseline design. Although you are free to develop your own modules to use in the datapath, you can also use the ones provided for you in the vc directory. We have also provided you the initial implementations of the immediate generator unit (see ProcDpathImmGen.v) and the ALU (see ProcDpathAlu.v). You will need to add functionality to the each of these modules as you add more instructions to the baseline design. As you add and/or modify datapath components, you will also need to , add another row to the control signal table in the control unit and potentially more columns in the control signal table to handle new control signals.

If you look carefully at the datapath diagram in Figure 13, you will notice several important differences from the basic pipeline discussed in lecture. The TinyRV1 processor described in lecture assumed a combinational memory where the memory response would always be returned in the same cycle as the memory request. This simplified our discussion, but it prevents composing the processor with more sophisticated memory systems that may be busy and/or take multiple cycles. As mentioned above, our memory interface assumes that a response can be returned in one or more cycles after the request. This means we must send the request into the memory system *one cycle earlier* than we would with a purely combinational memory system. Notice that the address for a data request (due to a load/store instruction) is sent into the memory system *at the end of the X stage, not the beginning of the M stage*. This allows the read data to be returned at the end of the M stage. Similarly, the instruction address is sent into the memory system *before the F stage*. This allows the instruction to be returned at the end of the F stage.
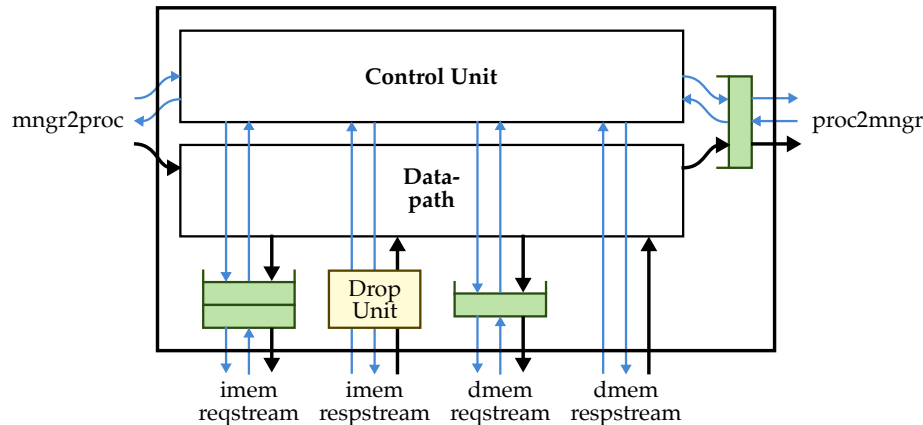
Figure 4: Processor Datapath and Control Composition – In addition to the datapath and control unit, the processor also includes bypass queues on output stream interfaces and a drop unit for the input stream instruction memory response interface.

Figure 4 shows how the datapath and control unit are composed in the top-level processor model. Note that we include several additional components in this composition. We include bypass queues on output stream interfaces. If a bypass queue is empty, then the message "bypasses" the queue and is immediately sent out the corresponding stream interface. If the stream interface is not ready, then we can buffer the message in the bypass queue. These queues simplify our processor implementation since they remove the requirement that a valid signal cannot depend on a ready signal. Note that the queue on the `imem_reqstream` interface actually requires two elements of buffering; this extra buffering ensures that we always have a place to put new instruction memory requests when we are redirecting the control flow at the front-end of the pipeline, even if the front-end of the pipeline is stalled. There is one more subtle but very important issue we must consider when using this kind of latency insensitive interface for our memory system. Once we send a memory request into the memory system we cannot "cancel" that request. This is not a problem with data memory requests since we never need to cancel such a request. The situation is more complicated for instruction memory requests. When we need to squash instructions at the beginning of the pipeline due to a control hazard, we also need to handle instruction memory requests that are currently in flight. Since we cannot actually cancel these instruction memory requests, we insert a special drop unit (see `DropUnitVRTL.v`) where the instruction memory response comes back into the processor. When we squash an instruction, we also tell the drop unit to remember to drop the next instruction that is returned from the memory system. Note that the baseline processor we provide you already correctly interacts with the memory system, so you should hopefully not have to worry too much about these subtle issues.

You will use the variable-latency integer multiplier that you worked so hard on in the first lab to implement the `mul` instruction. You can include your multiplier like this:

```
`include "lab1_imul/IntMulAlt.v"
```

Send the request to the multiplier in the D stage and wait for the response in the X stage. Integrating the multiplier unit into the processor can be difficult since you will need to carefully manage the val/rdy signals for requests to the multiplier and for responses from the multiplier. Here are some hints to get you started:

```
1  always_comb begin
2    casez ( inst_D )
3      //                        br      imm   rs1 op2    rs2 alu      dmm wbmux rf
4      //                        val type     type  en muxsel  en fn        typ sel   wen csrr csrw
5      `TINYRV2_INST_NOP  :cs( y, br_na,  imm_x, n, bm_x,   n, alu_x,   nr, wm_a, n,  n,   n   );
6      `TINYRV2_INST_ADD  :cs( y, br_na,  imm_x, y, bm_rf,  y, alu_add, nr, wm_a, y,  n,   n   );
7      `TINYRV2_INST_LW   :cs( y, br_na,  imm_i, y, bm_imm, n, alu_add, ld, wm_m, y,  n,   n   );
8      `TINYRV2_INST_BNE  :cs( y, br_bne, imm_b, y, bm_rf,  y, alu_x,   nr, wm_a, n,  n,   n   );
9      `TINYRV2_INST_CSRR :cs( y, br_na,  imm_i, n, bm_csr, n, alu_cp1, nr, wm_a, y,  y,   n   );
10     `TINYRV2_INST_CSRW :cs( y, br_na,  imm_i, y, bm_rf,  n, alu_cp0, nr, wm_a, n,  n,   y   );
11
12     `TINYRV2_INST_ADDI :cs( y, br_na,  imm_i, y, bm_imm, n, alu_add, nr, wm_a, y,  n,   n   );
13
14     default            :cs( n, br_x,   imm_x, n, bm_x,   n, alu_x,   nr, wm_x, n,  n,   n   );
15   endcase
16 end
```

Figure 5: Updated Control Signal Table for `addi` in Baseline Design

- **imul_istream_val**: This signal is sent from the D stage of the processor to the multiplier. You should factor the D stage's stall signal into the logic for setting the multiplier's request val signal, since if the D stage is stalling we do not want to send a request into the multiplier (otherwise we might end up sending the same request multiple times while we continue to stall!). You should also factor the D stage's squash signal into the logic for the same reason. If we have a branch in the X stage and it needs to squash a multiply instruction in the D stage, we do not want to send a request into the multiplier (otherwise we might not be able to squash the multiply transaction once it is in the multiplier!).

- **imul_istream_rdy**: This signal is sent from the multiplier back to the D stage of the processor. You should factor the multiplier's request rdy signal into the ostall logic for the D stage, since if the multiplier is not ready to accept a new request you must originate a stall.

- **imul_ostream_val**: This signal is sent from the multiplier back to the X stage of the processor. If a `mul` instruction is in the X stage, then you should factor the multiplier's response val into the ostall logic for the X stage. If the multiplier has not returned the response, we must wait for the multiplier to finish.

- **imul_ostream_rdy**: This signal is sent from the X stage of the processor to the multiplier. You should factor the X stage's stall signal into the logic for setting the multiplier's response rdy signal, since if the X stage is stalling we do not want to accept a response from the multiplier (we have no where to store that response since we are stalling!). There is no way to squash the X stage so we don't need to worry about that situation.

We strongly encourage you to use an incremental development design methodology. You should add one instruction at a time to your baseline processor, test that instruction, ensure it is working, and then move onto the next instruction. We recommend implementing the instructions in the following order: register-register arithmetic instructions, register-immediate instructions, memory instructions, jump instructions, branch instructions. We do not recommend waiting until the end to add the `mul` instruction. Since the `mul` instruction uses a stream interface, it is probably easier to integrate it into the pipeline after completing the other register-register arithmetic instructions.

To add a new instruction to the baseline design, first update Figure 14 with any changes you need to support the new instruction, update the code for the datapath, update the control signal table in the control unit, update the top-level module, and thoroughly test your instruction before moving onto the next instruction. For example, the `addi` instruction only requires a new row in the control signal table. Figure 5 shows what the Verilog control signal table would look like for the baseline processor

```
1  # signed-less-than operation          1  # signed-right-shift
2                                         2
3  logic slt;                            3  logic [31:0] srs
4  assign slt = $signed(a) < $signed(b); 4  assign srs = $signed(a) >>> b
```

Figure 6: Verilog Signed Less-Than and Right-Shift

after adding the `addi` instruction to the five instructions we provide. We need to specify that this is a valid instruction, it is not a branch, that the `rs1` field is valid, that the operand mux is set to select the immediate, the ALU function is set to add, it is not a data memory instruction, the write-back data comes from the ALU output, and the instruction writes the register file. Note that the write register address is always `rd` according to RISC-V ISA manual. To implement the `jal` instruction we would need to change both the datapath and the control unit. In the datapath, we would connect the sum of the generated immediate and PC to another input of the PC select mux, and as a consequence the `pc_sel_F` control signal would need to be wider than 1 bit. In the control unit, we need to add a column in the control signal table indicating if this instruction is `jal`. In the D stage there should be some logic to redirect the PC (`pc_sel_D`). For example, you should have a `pc_redirect_D` signal set to be high if the instruction is valid and it is a jump. In the F stage, you need to factor in both the branch (`pc_redirect_X`) and jump (`pc_redirect_D`) to decide `pc_sel_F`, which is the signal used to set the `pc_sel_mux_F` in the datapath. `jalr` is probably the most interesting instruction. As you can see from the baseline processor datapath diagram, you might ask why is there another PC+4 incrementer in X stage? (Hint: how is the `jalr_target` calculated and why?)

You will end up with around 13 or so different operations in your ALU. Most of these are pretty straight-forward. You can use standard arithmetic, shift, comparison, and logical operators, but all of these operators are agnostic to whether the inputs are signed or unsigned. For example, the addition operator (+) will work correctly regardless of whether or not the inputs are signed or unsigned (this is the beauty of two's complement!). However, some instructions will require ALU operations that are specifically designed to treat the inputs as signed values. More specifically, students will need to carefully consider the `slt` (register-register signed-less-than), `slti` (register-immediate signed-less-than), `sra` (shift right arithmetic), `blt` (branch signed-less-than), and `bge` (branch signed-greater-than-or-equal). Figure 6 shows how to implement signed-less-than and signed-right-shift on 32-bit input signals in Verilog. The `$signed` system task indicates that a value should be treated as a signed value. The `>>>` Verilog operator is specifically designed for signed-right-shift operations. Both `$signed` and `>>>` are synthesizable and allowed according to the course Verilog usage rules.

3. Alternative Design

The alternative design for this lab is a five-stage fully bypassed processor for the same TinyRV2 ISA. Once you get your baseline design working and passing all of your tests, you should copy your baseline processor design into `ProcAltDpath.v`, `ProcAltCtrl.v`, and `ProcAlt.v`, and then start working on the alternative design. Bypassing avoids data hazards by forwarding values from later pipeline stages to earlier stages. Your design should be fully-bypassed, i.e., it should be possible to forward values from the end of the X, M, and W stages to the instruction in D stage. To add bypassing to the processor, you will need to add bypass muxes to the datapath. Examine the datapath for the baseline design and determine where the muxes would need to be placed, as well as where the values would need to be bypassed from. We should emphasize that the goal is not just to pass the tests, but to pass the tests with a fully bypassed datapath. Check your line traces for your tests, and also judge your performance in your evaluation to make sure your design is working as you expect. Keep in mind that implementing bypassing does not remove the need to stall in some cases. Specifically,

load-use dependencies cannot be avoided by bypassing data; you will still need to stall in this case. We strongly encourage you to use an incremental development design methodology. Add bypass paths from one stage and test your design before starting to add the next set of bypass paths.

4. Testing Strategy

We provide you with one very basic test for each instruction in TinyRV2. We have also provide more comprehensive directed and random tests for add, lw, bne, csrr, csrw, and, andi, or, ori, xor, xori. Writing tests for this lab will be very challenging due to both the number of instructions and the number of cases we need to test for each instruction. As with the previous lab, you will want to initially write tests using the functional-level model (ISA emulator). Once these tests are working on the ISA emulator, you can move on to testing the baseline and alternative designs.

The following commands illustrate how to run all of tests for the entire project, how to run just the tests for this lab, and how to just the tests for a specific model, and how to run just the tests for add instruction for each each model.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ..
% pytest ../lab2_proc
% pytest ../lab2_proc/test/ProcFL_*_test.py
% pytest ../lab2_proc/test/ProcBase_*_test.py
% pytest ../lab2_proc/test/ProcAlt_*_test.py
% pytest ../lab2_proc/test/ProcFL_rr_test.py   -k test_add
% pytest ../lab2_proc/test/ProcBase_rr_test.py -k test_add
% pytest ../lab2_proc/test/ProcAlt_rr_test.py  -k test_add
```

Other useful flags (-dump-vcd, -tb=long, -tb=short, -tb=auto, -s, -x) are introduced and explained in the Verilog tutorial. Please familiarize yourself with them!

All of the tests should pass on the FL model, and as you add more tests and incrementally develop your designs you will slowly start passing more and more of the tests for your baseline and alternative designs. The baseline processor that we provide to get you started will pass all of the tests for the add, lw, csrr, and csrw instructions. It should pass the very first test, but will fail the remaining tests for the bne instruction for example. This is because we use the addi instruction in most of our control flow tests. Once you implement and test the addi instruction the remaining tests should start passing for the bne instruction.

Our directed testing will be done using short assembly sequences represented as multi-line Python strings. Each assembly sequence usually starts with one or more csrr instructions to receive input data from the test source, and ends with one or more csrw instructions to send output data to the test sink for verification. You will need to think critically about how to test each instruction. Pick one instruction, think through what it does, and trace its flow through the datapath diagram. Where can things go wrong? You can choose large or small values, force stalls or bypassing, or stress its interaction with other instruction classes. You will need many assembly sequences for each instruction to test basic operation, proper handling of hazards, various input values, and random delays on the test source, sink, and memory. Once you have thoroughly tested an instruction of one class (e.g., register-register instructions, branch instructions), you can usually leverage a very similar approach for other instructions in that class.

Each assembly sequence is generated by Python functions defined in the test subdirectory. You can use the assembly sequence generation functions that we provide in test/inst_add.py, test/inst_lw.py,

```
1   # Send value 0x00002000 from test source into processor
2   csrr x2, mngx2proc < 0x00002000
3   csrr x4, mngx2proc < 0x00002010
4
5   # Loop over four elements in array
6   addi  x1, x0, 4
7 loop:
8   lw    x3, 0(x2)
9   addi  x3, x3, 1
10  sw    x3, 0(x4)
11  addi  x2, x2, 4
12  addi  x4, x4, 4
13  addi  x1, x1, -1
14  bne   x1, x0, loop
15
16  # Read out the four results and send to test sink for verification
17
18  addi  x1, x0, 0x2010
19  lw    x2, 0(x1)
20  csrw  proc2mngr, x2 > 2
21
22  addi  x1, x0, 0x2014
23  lw    x2, 0(x1)
24  csrw  proc2mngr, x2 > 3
25
26  addi  x1, x0, 0x2018
27  lw    x2, 0(x1)
28  csrw  proc2mngr, x2 > 4
29
30  addi  x1, x0, 0x201c
31  lw    x2, 0(x1)
32  csrw  proc2mngr, x2 > 5
33
34  # Data section
35  .data
36
37  # src array
38  .word 0x00000001
39  .word 0x00000002
40  .word 0x00000003
41  .word 0x00000004
42
43  # dest array
44  .word 0x00000000
45  .word 0x00000000
46  .word 0x00000000
47  .word 0x00000000
```

Figure 7: Example Assembly Program Illustrating Acceptable Syntax

and `test/inst_bne.py` as examples. Note that these examples use helper functions defined in `test/inst_utils.py`. You are free to use these helper functions in your own assembly sequence generation functions. Developing these assembly sequences can be tedious, so we strongly encourage students to leverage the productivity of Python to create parameterized helper functions.

Figure 7 shows a simple assembly program that is meant to illustrate the assembly syntax we will be using for testing. Note that this program does *not* make a very good unit test since it uses too many instructions all at once. However, an assembly sequence like this might be a reasonable integration test once all instructions have been unit tested individually. Comments are denoted with the # character. All registers are denoted using xN where N is the register number. Immediate literals can be in either signed decimal (e.g., 16 or -16), hexidecimal (e.g., 0x10), or binary (e.g., 0b10000). Labels are

allowed (e.g., `loop:` on line 7) and can also be used as the target for control flow instructions (e.g., `bne` instruction on line 14). Note the special syntax for specifying the values that should be retrieved from a test source, or the values expected in a test sink. On line 1, we send the value `0x2000` from the test source into the processor where it is written to register `x2`. On line 20, we send the value in register `x2` out to the test sink, where the sink will expect to see the value 2. If the sink receives a value other than 2, then it will cause a test sink failure. Please keep in mind that the messages are added to the test source and sink *in static program order*. In other words, the messages are added to the test source and sink in the order they appear in the static assembly sequence regardless of any control flow. The very first instruction in an assembly sequence that we load into memory is always at address `0x200`. As illustrated on line 35, data is specified in a special `.data` section which is always located at address `0x2000`. Raw values can be initialized in the data section using `.word` (see lines 38–41).

Figure 8 shows example assembly sequence generation functions that test the `addi` instruction. The `gen_single_dest_dep_test` function is meant to just test that the processor correctly resolves RAW hazards for the destination register (i.e., that the consuming `csrw` instruction correctly stalls or by-passes the result of the instruction under test). We include plenty of `nop` instructions before the instruction under test to ensure there are no RAW hazards with reading the source register. The `gen_single_dest_dep_test` function is parameterized by the number of `nop`s to insert after the in-struction under test. The `gen_nops` helper function is included as part of `test/inst_utils.py`. The assembly sequence generation function is also parameterized by the input value, immediate value, and expected result. The `gen_dest_dep_test` uses the `gen_single_dest_dep_test` to generate a more complicated sequence of six tests. You can use the Python interpreter and print statements to verify that the generated assembly is as expected.

Figure 9 shows an example assembly sequence generation function that tests the `jal` instruction. Testing control flow instructions is particularly challenging since our test sink verifies values not control flow. We use the `addi` instruction to "track" the control flow; whenever we want to record that processor visited a certain point in our assembly sequence, we simply set a unique bit in a common register (`x3` in this case). Then at the end of the assembly sequence, we can send this common register to the test sink and verify that only the expected bits are set (i.e., that the processor only visited the expected points in our assembly sequence). There are 12 bits in the immediate field, but you should only use 11 bits to avoid issues with sign extension. This means you can track up to 11 control flow points in a single assembly sequence.

Once we have developed assembly sequence generation functions in `test`, we can then use these generation functions to create the actual unit tests for the various processor implementations. These unit tests are divided into six categories and six corresponding test scripts for each implementation (FL, baseline, alternative):

- `Proc<impl>_csr_test.py` – Tests for csr insts (`<impl>` = FL, Base, Alt)
- `Proc<impl>_rr_test.py` – Tests for reg-to-reg insts (`<impl>` = FL, Base, Alt)
- `Proc<impl>_rimm_test.py` – Tests for reg-to-imm insts (`<impl>` = FL, Base, Alt)
- `Proc<impl>_mem_test.py` – Tests for memory insts (`<impl>` = FL, Base, Alt)
- `Proc<impl>_jump_test.py` – Tests for jump insts (`<impl>` = FL, Base, Alt)
- `Proc<impl>_branch_test.py` – Tests for branch insts (`<impl>` = FL, Base, Alt)

```
1  def gen_single_dest_dep_test( num_nops,
2                                  src, imm, result ):
3    return """
4      csrr  x1, mngr2proc < {src}
5      nop
6      nop
7      nop
8      nop
9      nop
10     nop
11     nop
12     nop
13     addi  x3, x1, {imm}
14     {nops}
15     csrw  proc2mngr, x3 > {result}
16   """.format(
17     nops = gen_nops( num_nops ),
18     **locals()
19   )
20
21 def gen_dest_dep_test():
22   return [
23     gen_single_dest_dep_test( 5, 1, 1, 2 ),
24     gen_single_dest_dep_test( 4, 2, 1, 3 ),
25     gen_single_dest_dep_test( 3, 3, 1, 4 ),
26     gen_single_dest_dep_test( 2, 4, 1, 5 ),
27     gen_single_dest_dep_test( 1, 5, 1, 6 ),
28     gen_single_dest_dep_test( 0, 6, 1, 7 ),
29   ]
```

Figure 8: Example Assembly Sequence Generation Function for `addi` **Instruction**

```
1  def gen_multijump_test():
2    return """
3
4      # Use x3 to track the control flow pattern
5      addi  x3, x0, 0                # 0x00000200
6
7      jal   x1, label_a    # j -.    # 0x00000204
8      addi  x3, x3, 0b000001 #    |    # 0x00000208
9                             #    |    #
10   label_b:                 # <--+-.  #
11     addi  x3, x3, 0b000010 #    | |  # 0x0000020c
12     addi  x5, x1, 0        #    | |  # 0x00000210
13     jal   x1, label_c    # j -+-+-. # 0x00000214
14     addi  x1, x3, 0b000100 #    | | | # 0x00000218
15                           #    | | | #
16   label_a:                 # <--' | | #
17     addi  x3, x3, 0b001000 #      | | # 0x0000021c
18     addi  x4, x1, 0        #      | | # 0x00000220
19     jal   x1, label_b    # j ---' | # 0x00000224
20     addi  x3, x3, 0b010000 #      | # 0x00000228
21                           #      | #
22   label_c:                 # <------' #
23     addi  x3, x3, 0b100000 #        # 0x0000022c
24     addi  x6, x1, 0        #        # 0x00000230
25
26     # Carefully determine which bits are expected
27     # to be set if jump operates correctly.
28     csrw  proc2mngr, x3 > 0b101010
29
30     # Check the link addresses
31     csrw  proc2mngr, x4 > 0x00000208
32     csrw  proc2mngr, x5 > 0x00000228
33     csrw  proc2mngr, x6 > 0x00000218
34   """
```

Figure 9: Example Assembly Sequence Generation Function for `jal` **Instruction**

```
1  from test import inst_addi
2
3  @pytest.mark.usefixtures("cmdline_opts")
4  class Tests:
5
6    @classmethod
7    def setup_class( cls ):
8      cls.ProcType = ProcFL
9
10   @pytest.mark.parametrize( "name,test", [
11     asm_test( inst_addi.gen_basic_test      ),
12     asm_test( inst_addi.gen_dest_dep_test  ),
13   ])
14   def test_addi( s, name, test ):
15     run_test( s.ProcType, test,
16       cmdline_opts=s.__class__.cmdline_opts )
```

Figure 10: Example Test Function for `addi` **in** `ProcFL_rimm_test.py`

```
1  from test import inst_addi
2
3  @pytest.mark.usefixtures("cmdline_opts")
4  class Tests:
5
6    @classmethod
7    def setup_class( cls ):
8      cls.ProcType = ProcFL
9
10   def test_add_delays( s ):
11     run_test( s.ProcType, inst_add.gen_random_test,
12             delays=True,
13             cmdline_opts=s.__class__.cmdline_opts )
```

Figure 11: Example Test Function for `add` **with Random Delays in** `ProcFL_rr_test.py`

Each test script already has the basic test we provide for you. To add more tests you simply add more rows to the `pytest` parameterized function. You should always start by running your tests on the FL model to ensure that the test themselves are correct. So if we want to actually use the `gen_dest_dep_test` assembly sequence generation function on the FL model, we would modify the `ProcFL_rimm_test.py` test script as shown in Figure 10. We can run all of the tests for the `addi` instruction and then just the new test case like this:

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab2_proc/test/ProcFL_rimm_test.py -k test_addi
% pytest ../lab2_proc/test/ProcFL_rimm_test.py -k test_addi[dest_dep
```

Once we are sure our tests pass on the FL model, then we can try running those tests on the baseline and alternative designs. Note that we have set up the test scripts so that any test you use on the FL model will also automatically run on the baseline and alternative designs. This means you just need to focus on adding more tests to the `ProcFL_*_test.py` scripts.

In addition to testing the functionality of each instruction, we also want to make sure every instruction functions correctly when faced with delays on the stream source, sink, and memory. Figure 11 illustrates the random delay testing we provide for the `add` instruction in `ProcBaseRTL_rr_test.py`. You will need to add similar random delay testing for each instruction you implement.

You will almost certainly want to use line tracing to help you visualize instructions moving through the pipeline. We have provided most of the important line tracing code for you in the baseline design. Figure 12 illustrates a line trace from the baseline design for a assembly sequence generated to test the `add` instruction. Extra annotations are included to indicate what the columns mean. The first column shows when data is sent from the stream source into the processor, and the last column shows when data is sent from the processor to the stream sink. There are five columns that show the five pipeline stages with the PC shown in the F stage, the disassembled instruction in the D stage, and a short four-character instruction mnemonic in the X, M, and W stages. The # symbol means an instruction is stalling in that stage, and the ~ symbol means an instruction is being squashed in that stage. The instruction memory request/response stream and data memory request/response stream are also shown. Debugging through line tracing alone will simply not be possible; students will almost certainly need to use `surfer` to view VCD waveforms for debugging as well.

We cannot stress enough how important it is for students to take an incremental, test-driven design approach. Students should implement one and only one new instruction by modifying the datapath and control unit. Students should then implement the corresponding unit tests, verify that the tests are correct on the FL model, then verify that their baseline design passes the same test. Then, and only then, should students move onto the next instruction. As mentioned above, we recommend implementing the instructions in the following order: register-register arithmetic instructions, register-immediate instructions, memory instructions, jump instructions, branch instructions.

In addition to the assembly tests for each instruction, you must also add additional unit tests for any datapath components you add or modify. So when you add new operations to the ALU, you must add corresponding unit tests to `ProcDpathAlu_test.py`.

```
            fetch    decode              exe mem  wb
cyc from src  PC      instruction        inst inst inst   imem_reqstream        imem_respstream              dmem            to sink

 1r .        >        |                  |    |    |    |[ ]                >              [  ]|[ ] >. [  ] >
 2r .        >        |                  |    |    |    |[ ]                >              [  ]|[ ] >. [  ] >
 3: .        >        |                  |    |    |    |[ ]rd:00:00000200:0:>            [  ]|[ ] >. [  ] >
 4: #        > 00000200|                 |    |    |    |[ ]rd:00:00000204:0:>rd:00:0:0:fc0020f3[ *]|[ ] >. [  ] >
 5: 00000005 > 00000204|csrr   x01, mngr2proc |    |    |[ ]rd:00:00000208:0:>rd:00:0:0:fc002173[ *]|[ ] >. [  ] >
 6: 00000004 > 00000208|csrr   x02, mngr2proc |csrr|    |[ ]rd:00:0000020c:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
 7: .        > 0000020c|nop               |csrr|csrr|    |[ ]rd:00:00000210:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
 8: .        > 00000210|nop               |nop |csrr|csrr|[ ]rd:00:00000214:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
 9: .        > 00000214|nop               |nop |nop |csrr|[ ]rd:00:00000218:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
10: .        > 00000218|nop               |nop |nop |nop |[ ]rd:00:0000021c:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
11: .        > 0000021c|nop               |nop |nop |nop |[ ]rd:00:00000220:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
12: .        > 00000220|nop               |nop |nop |nop |[ ]rd:00:00000224:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
13: .        > 00000224|nop               |nop |nop |nop |[ ]rd:00:00000228:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
14: .        > 00000228|nop               |nop |nop |nop |[ ]rd:00:0000022c:0:>rd:00:0:0:002081b3[ *]|[ ] >. [  ] >
15: .        > 0000022c|add    x03, x01, x02 |nop |nop |nop |[ ]rd:00:00000230:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
16: .        > 00000230|nop               |add |nop |nop |[ ]rd:00:00000234:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
17: .        > 00000234|nop               |nop |add |nop |[ ]rd:00:00000238:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
18: .        > 00000238|nop               |nop |nop |add |[ ]rd:00:0000023c:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
19: .        > 0000023c|nop               |nop |nop |nop |[ ]rd:00:00000240:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
20: .        > 00000240|nop               |nop |nop |nop |[ ]rd:00:00000244:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
21: .        > 00000244|nop               |nop |nop |nop |[ ]rd:00:00000248:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
22: .        > 00000248|nop               |nop |nop |nop |[ ]rd:00:0000024c:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
23: .        > 0000024c|nop               |nop |nop |nop |[ ]rd:00:00000250:0:>rd:00:0:0:7c019073[ *]|[ ] >. [  ] >
24: .        > 00000250|csrw   proc2mngr, x03 |nop |nop |nop |[ ]rd:00:00000254:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
25: .        > 00000254|nop               |csrw|nop |nop |[ ]rd:00:00000258:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
26: .        > 00000258|nop               |nop |csrw|nop |[ ]rd:00:0000025c:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
27: .        > 0000025c|nop               |nop |nop |csrw|[ ]rd:00:00000260:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] > 00000009
28: .        > 00000260|nop               |nop |nop |nop |[ ]rd:00:00000264:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
29: .        > 00000264|nop               |nop |nop |nop |[ ]rd:00:00000268:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
30: .        > 00000268|nop               |nop |nop |nop |[ ]rd:00:0000026c:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
31: .        > 0000026c|nop               |nop |nop |nop |[ ]rd:00:00000270:0:>rd:00:0:0:00000013[ *]|[ ] >. [  ] >
32: .        > 00000270|nop               |nop |nop |nop |[ ]rd:00:00000274:0:>rd:00:0:0:fc0020f3[ *]|[ ] >. [  ] >
```

Figure 12: Line Trace for ADD Directed Test – The line trace clearly shows the instructions going down the pipeline. Each line corresponds to one cycle, and the columns correspond to the stream source, sink, memory system, and each of the five pipeline stages.

5. Evaluation

Once you have verified the functionality of the baseline and alternate design, you can use the provided simulator to evaluate your two designs. You can run the simulator like this:

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% ../lab2_proc/proc-sim --impl base --input vvadd-unopt --verify --stats
% ../lab2_proc/proc-sim --impl alt  --input vvadd-unopt --verify --stats --trace
```

The simulator will display the total number of cycles to execute the specified benchmark. It will also show you the instruction count and the CPI. You can choose the implementation you want to evaluate with the `--impl` command line option. You should study the line traces (with the `--trace` command line option) to understand the reason why each design performs as it does on the various benchmarks. The `--verify` command line option enables verification by checking the output array to see if the values are as expected. The benchmarks provide non-trivial and realistic sequences of instructions, so passing the verification is a good sanity check that your processor is working as expected. Having said this, the simulator is not meant for verifying your design; you should use

a systematic testing strategy to ensure your design is fully functional before attempting to use the simulator.

We have provided you with four different benchmarks and two versions for vvadd. These benchmarks are:

- vvadd-unopt   : Element-wise vector-vector add (unoptimized)
- vvadd-opt   : Element-wise vector-vector add (optimized)
- cmult   : Element-wise complex multiplication
- bsearch   : Binary search in a linear array of key/value pairs
- mfilt   : Masked convolution on a small image

For vvadd we provide both unoptimized and optimized versions. The optimized version unrolls the loop to minimize both data and control hazards. Each of these benchmarks are in their respective proc_ubmark_<ubmark>.py file and there is more information on what each algorithm does as well as a C code snippet, the assembly instructions, and the input and reference data used for verification. You should take a look at these to get a feeling on what each benchmark does.

6. Looking Towards the Multicore System in Lab 4

In lab 4, we will compose the pipelined processor and cache memory designed in labs 2–3 to create a baseline single-core system and we will add a ring network to create the alternative multicore system. You will be developing a serial and parallel sorting microbenchmark, and comparing the performance of this microbenchmark across the baseline and alternative designs. You will quickly find that the performance of your alternative multicore system is limited by the hit latency of the cache you will design in lab 3, but after optimizing the hit latency the next critical performance bottleneck will likely be the branch resolution latency of your pipelined processor. Our sorting microbenchmark will have one or more loops, and the pipelined processor designed in this lab will almost always mispredict the backwards branch used in loops. Later in the semester, we will learn about dynamic hardware branch prediction techniques. Adding a simple branch target buffer (BTB) in the F stage could have a significant impact on the performance of the sorting microbenchmark by effectively eliminating almost all squashes due to the backwards branch used in loops. There is no need to wait until lab 4. More advanced students should feel free to start optimizing their pipelined processor as part of the alternative design in this lab, or after this lab is submitted. You will need to read ahead to learn about BTBs. A simple, yet effective approach would be to include a four entry BTB in the F stage. Each entry would include a valid bit, the PC of the branch, and the target address for the branch when it is taken. In the F stage, your processor would need to search the BTB for the current PC. If there is a hit, then the F stage can use the corresponding target address in the BTB. If there is a miss, then the F stage can simply use PC+4. If a branch is taken, then in the X stage we would need to write the BTB with the corresponding PC of the branch and target address. This is obviously a very simple predictor. More complicated schemes are certainly possible.
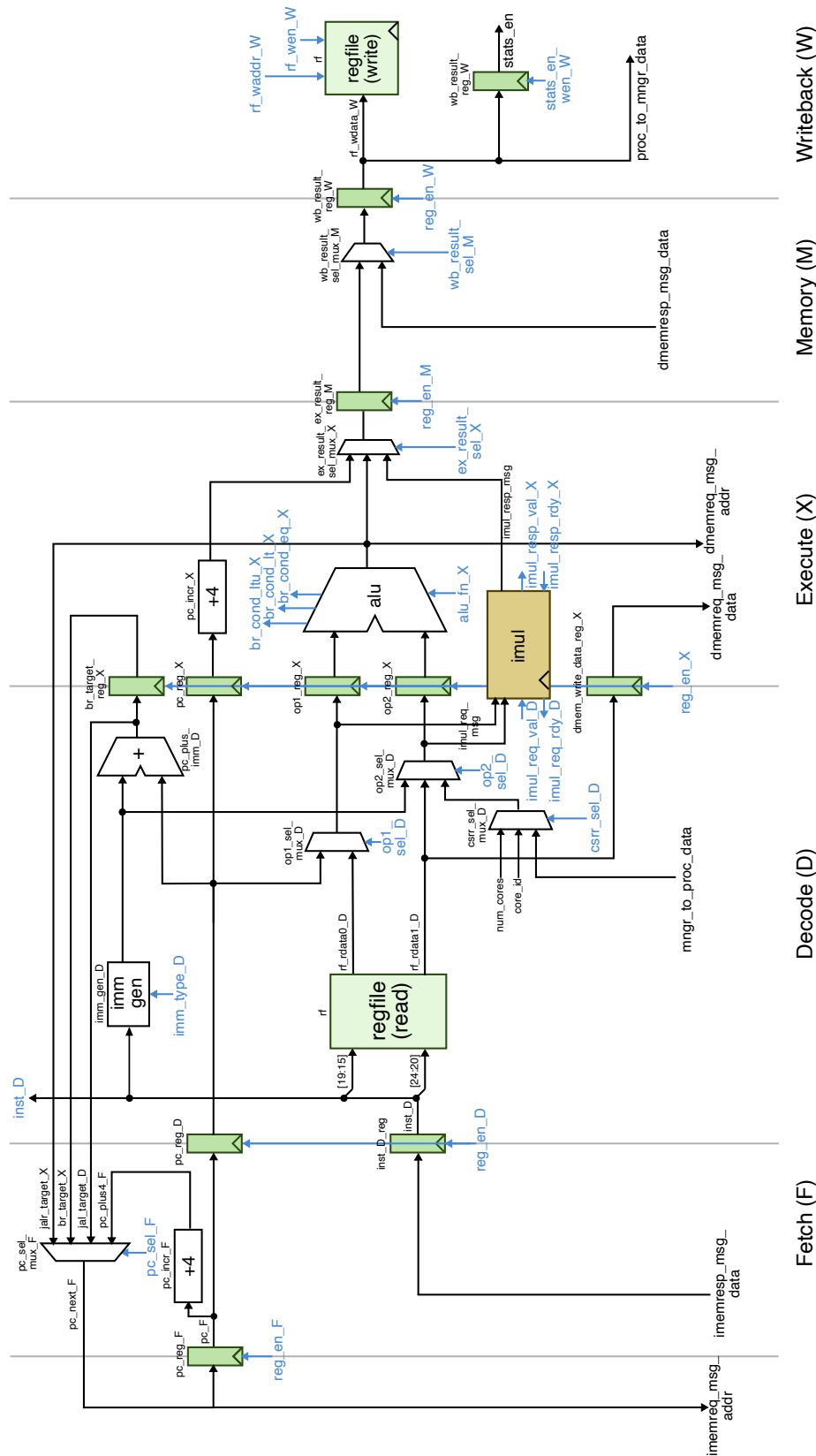
Acknowledgments

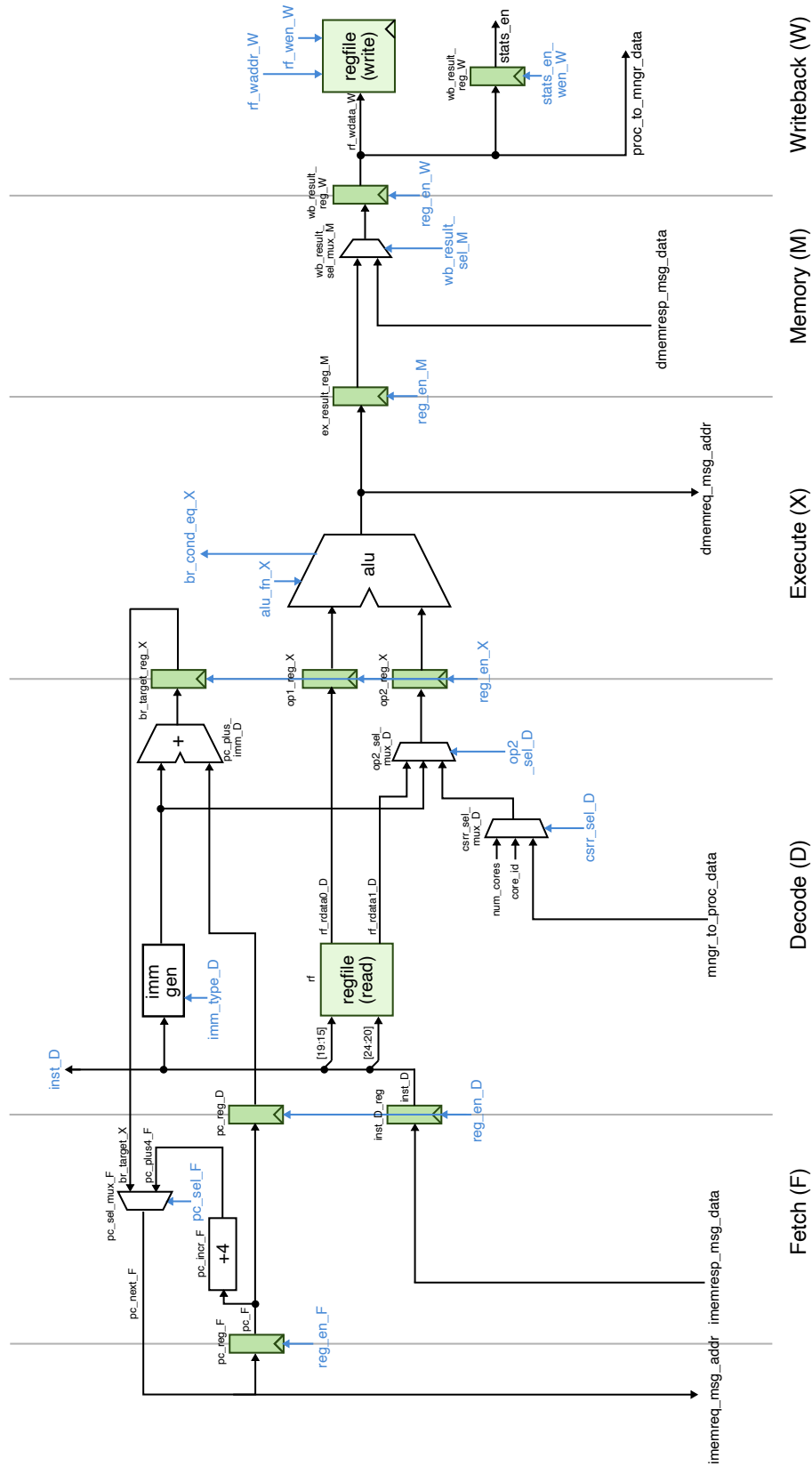**Figure 13:** Baseline Design: Five-Stage Stalling Processor Datapath

Figure 14: Initial Baseline Design Provided To Students