ECE 4750 Computer Architecture, Fall 2015
Lab 2: Pipelined Processor

School of Electrical and Computer Engineering
Cornell University

revision: 2015-09-26-22-56

In this lab, you will design two pipelined processor microarchitectures for the PARCv2 instruction set architecture. After implementing all PARCv2 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. **As with all lab assignments, the majority of your grade will be determined by the lab report. You should consult the course lab assignment assessment rubric 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 and the lab assessment rubric. To get started, you should access the ECE computing resources and you have should have used the `ece4750-lab-admin` script to create or join a GitHub group. If you have not do so already, source the setup script and clone your lab group's remote repository from GitHub:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:cornell-ece4750/lab-groupXX
```

where `XX` is your group number. **You should never fork your lab group's remote repository! If you need to work in isolation then use a branch within your lab group's remote repository.** If you have already cloned your lab group's remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests.You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% py.test ../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:

- `ProcFL.py` – FL processor
- `elf.py` – Classes/functions for reading/writing ELF binary files
- `elf_test.v` – Unit tests for above
- `SparseMemoryImage.py` – Class representing memory image for loading test memory
- `SparseMemoryImage_test.py` – Unit tests for above
- `parc_encoding.py` – Classes/functions for the PARC ISA encoding used in FL model
- `parc_encoding_test.v` – Unit tests for above
- `parc_semantics.v` – Classes/functions for the PARC ISA semantics used in FL model

- `DropUnitPRTL.py` – PyMTL RTL unit for dropping inst mem response on squash
- `DropUnitVRTL.v` – Verilog RTL unit for dropping inst mem response on squash
- `PARCInstPRTL.py` – PyMTL RTL helper constants, functions for PARC ISA
- `PARCInstPRTL.v` – Verilog RTL helper constants, functions for PARC ISA
- `ProcDpathComponentsPRTL.py` – PyMTL data-path components
- `ProcDpathComponentsVRTL.v` – Verilog data-path components

- `ProcBasePRTL.py` – PyMTL RTL stalling processor
- `ProcBaseCtrlPRTL.py` – PyMTL RTL stalling processor's control unit
- `ProcBaseDpathPRTL.py` – PyMTL RTL stalling processor's datapath
- `ProcBaseVRTL.v` – Verilog RTL stalling processor
- `ProcBaseVRTL.py` – PyMTL wrapper around Verilog RTL
- `ProcBaseCtrlVRTL.v` – Verilog RTL stalling processor's control unit
- `ProcBaseDpathVRTL.v` – Verilog RTL stalling processor's datapath
- `ProcBaseRTL.py` – Wrapper to choose which RTL language

- `ProcAltPRTL.py` – PyMTL RTL bypassing processor
- `ProcAltVRTL.v` – Verilog RTL bypassing processor
- `ProcAltVRTL.py` – PyMTL wrapper around Verilog RTL
- `ProcAltRTL.py` – Wrapper RTL to choose which RTL language

- `Proc<impl>_mngr_test.py` – Unit tests for manager insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_rimm_test.py` – Unit tests for reg-to-imm insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_rr_test.py` – Unit tests for reg-to-reg insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_mem_test.py` – Unit tests for memory insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_jump_test.py` – Unit tests for jump insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_branch_test.py` – Unit tests for branch insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)

- `tests/` – Directory with helper models/functions for unit testing

- `proc_ubmark_<ubmark>_data` – Data for microbenchmark
- `proc_ubmark_<ubmark>` – Assembly code for microbenchmark
- `proc-sim` – Processor simulator for evaluation

- `__init__.py` – Package setup

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

We will be using the PARC instruction set architecture (ISA) which is closely related to the MIPS32 ISA. The PARC ISA was introduced in lecture, and the PARC ISA manual is available on the public course web page. As an example, the specification from the PARC ISA manual for the `addu` instruction is shown in Figure 1. You will be implementing the PARCv2 subset, which should be sufficient to execute simple C programs. The list of instructions that constitute PARCv2 are below.

- Manager : `mfc0, mtc0`
- Reg-Reg : `addu, subu, mul, and, or, xor, nor, slt, sltu, srav, srlv, sllv`
- Reg-Imm : `addiu, lui, ori, andi, xori, slti, sltiu, sra, srl, sll`
- Memory : `lw, sw`
- Jump : `j, jal, jr`
- Branch : `bne, beq, bgtz, bltz, bgez, blez`

We have provided you a complete functional-level model of a PARCv2 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 simulator" (or ISA simulator) 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

```
* addu

 - Summary   : Signed addition with 3 GPRs, no overflow exception
 - Assembly  : addu r_dst, r_src0, r_src1
 - Semantics : R[r_dst] = R[r_src0] + R[r_src1]
 - Format    : R-Type

  31    26 25   21 20   16 15   11 10    6 5      0
 +--------+-------+-------+-------+-------+--------+
 |   op   |  rs   |  rt   |  rd   |  sa   |  cmd   |
 | 000000 | src0  | src1  | dst   | 00000 | 100001 |
 +--------+-------+-------+-------+-------+--------+
```

Figure 1: ADDU Instruction from PARC ISA Manual – The PARC ISA manual specifies the assembly syntax, semantics, and encoding for every instruction in the PARC ISA.

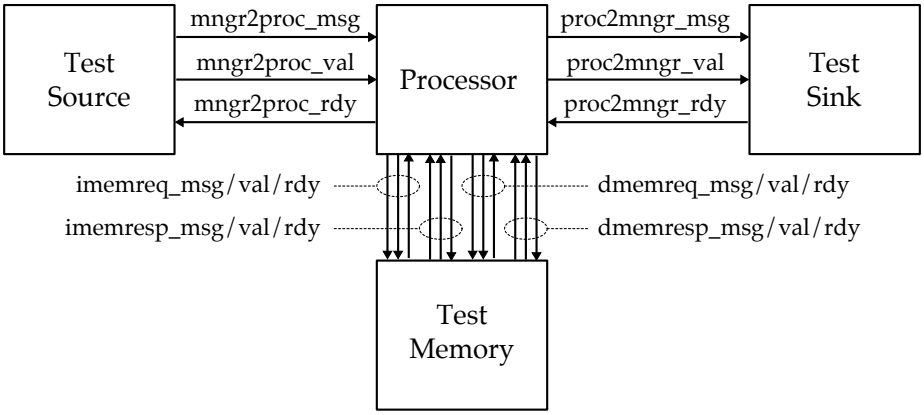


Figure 2: Processor System – The processor is integrated with a test source, test sink, and test memory for testing and evaluation.

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 `mfc0` instruction, and we can have the processor verify data using the test sink and the `mtc0` instruction.

We make extensive use of the latency insensitive val/rdy microprotocol in the processor interface. There are six different val/rdy channels.

- `mngr2proc` : from test source to processor
- `proc2mngr` : from processor to test sink
- `imemreq` : instruction memory request
- `imemresp` : instruction memory response
- `dmemreq` : data memory request
- `dmemresp` : data memory response

The processor interacts with the memory using memory messages. The message format for memory requests and responses are shown in Figure 3. Corresponding PyMTL `BitStructs` are defined in `pclib` here:

- `https://github.com/cornell-brg/pymtl/blob/ece4750/pclib/ifcs/MemMsg.py`

There are equivalent helper macros for Verilog located in `vc/mem-msgs.v`. Memory requests use fields to encode the type (e.g., read, 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 across a val/rdy 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

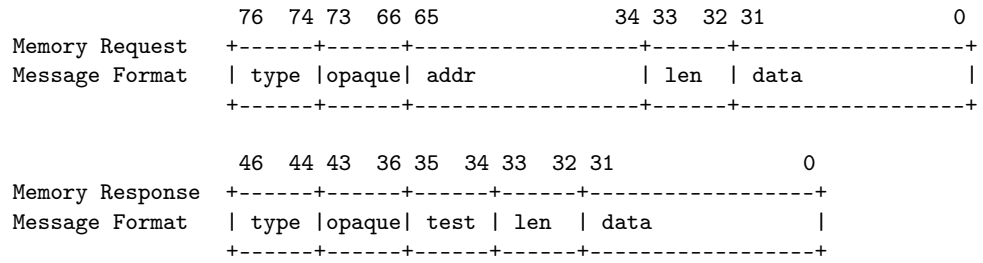


Figure 3: Memory Request/Response Message Formats – Memory request messages are sent from the processor to the memory, while memory response messages are sent from the memory back to the processor.

when the request is sent and when the response is received), but you cannot assume how many cycles it 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 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 PARCv2 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 14. 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 (`addu`, `lw`, `bne`). We have also implemented the `mtc0` (move to the test manager) and `mfc0` (move from the test manager) instructions which are used for testing. Figure 15 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 PARCv2 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 library. We have also provided you initial implementations of the branch and jump target logic, the register file, and the ALU (see `ProcDpathComponentsPRTL.py` or `ProcDpathComponentsVRTL.v`). You will need to add functionality to the ALU as you add more instructions to the baseline design. As you add and/or modify datapath components, you will also

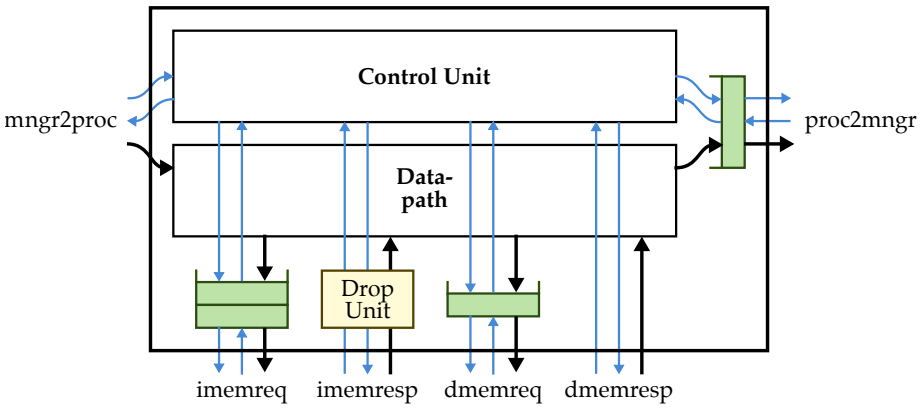


Figure 4: Processor Datapath and Control Composition – In addition to the datapath and control unit, the processor also includes bypass queues on output val/rdy interfaces and a drop unit for the input val/rdy instruction memory response interface.

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 14, you will notice several important differences from the basic pipeline discussed in lecture. The PARCv1 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 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 val/rdy interfaces. If a bypass queue is empty, then the message "bypasses" the queue and is immediately sent out the corresponding val/rdy interface. If the val/rdy 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 `imemreq` 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 `DropUnitPRTL.py` or `DropUnitVRTL.v`) where the instruction memory response comes back

```
always_comb begin
  casez ( inst_D )
    //                          br       rs op1      rt alu      dmm wbmux rf      thst fhst
    //                    val type     en muxsel   en fn       typ sel   wen wa  val  rdy
    `PISA_INST_NOP     :cs( y,  br_none, n, bm_x,    n, alu_x,   nr, wm_a, n,  rx, n,   n   );
    `PISA_INST_ADDIU   :cs( y,  br_none, y, bm_si,   n, alu_add, nr, wm_a, y,  rt, n,   n   );
    `PISA_INST_ADDU    :cs( y,  br_none, y, bm_rdat, y, alu_add, nr, wm_a, y,  rd, n,   n   );
    `PISA_INST_BNE     :cs( y,  br_bne,  y, bm_rdat, y, alu_x,   nr, wm_a, n,  rx, n,   n   );
    `PISA_INST_LW      :cs( y,  br_none, y, bm_si,   n, alu_add, ld, wm_m, y,  rt, n,   n   );
    `PISA_INST_MFC0    :cs( y,  br_none, n, bm_fhst, n, alu_cp1, nr, wm_a, y,  rt, n,   y   );
    `PISA_INST_MTC0    :cs( y,  br_none, n, bm_rdat, y, alu_cp1, nr, wm_a, n,  rx, y,   n   );
    default            :cs( n,  br_x,    n, bm_x,    n, alu_x,   nr, wm_x, n,  rx, n,   n   );

  endcase
end
```

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

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. If you are using PyMTL for this lab, you can import your multiplier like this:

```
from lab1_imul import IntMulAltRTL
```

This should work regardless of whether you used PyMTL or Verilog for the first lab. If you are using Verilog for this lab, and also used Verilog for the first lab, you can import your multiplier like this:

```
`include "lab1_imul/IntMulAltVRTL.v"
```

If you used PyMTL for the first lab and would like to use Verilog for this lab, please speak with the instructors on how to proceed. While we strongly encourage you to use your own implementation, if you do not feel confident in the functionality of your integer multiplier, the course staff can make our solution available to you. 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. Send the request to the multiplier in the D stage and wait for the response in the X stage. Factor the multiplier's request rdy signal into your stall logic for the D stage and the multiplier's response val signal into your stall logic for the X stage.

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. To add a new instruction to the baseline design, first update Figure 15 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 `addiu` 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 after adding the `addiu` instruction to the five instructions we provide. We need to specify that this is a valid instruction, it is not a branch, that the `rs` field is valid, that operand mux is set to select the sign-extended immediate, the ALU

```
1 # signed-less-than operation
2
3 logic slt;
4 assign slt = $signed(a) < $signed(b);
```

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

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

```
1 # signed-less-than operation
2
3 s.slt = Wire(1)
4
5 @s.combinational
6 def block():
7   tmp = sext( a, 33 ) - sext( b, 33 )
8   s.slt.value = tmp[32]
```

```
1 # signed-right-shift
2
3 s.srs = Wire(32)
4
5 @s.combinational
6 def block():
7   tmp = sext( a, 64 ) >> b
8   s.srs.value = tmp[0:32]
```

Figure 7: PyMTL Signed Less-Than and Right-Shift

function is set to add, it is not a data memory instruction, the write-back data comes from the ALU output, the instruction writes the register file, and the write address is `rt`. To implement the `j` instruction we would need to change both the datapath and the control unit. In the datapath, we would instantiate and connect the jump-target calculation unit provided in `ProcDpathComponentsPRTL.py` (or `ProcDpathComponentsVRTL.v`). We will need to add another input to the PC select mux, and as a consequence the `pc_sel_F` control signal would need to be wider. In the control unit, we need to add a column in the control signal table indicating if this instruction is a jump. 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. The D stage also needs to send the `pc_sel_D` signal to F. 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.

You will end up with around 12 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), and `sra` (shift right arithmetic). 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. Figure 7 shows how to implement signed-less-than and signed-right-shift on 32-bit input signals in PyMTL. Since PyMTL does not have `$signed` and >>>, we must be a bit more clever. For signed-less-than, we simply subtract the two inputs and check if the result is less than zero using the sign bit. We need to sign-extend the inputs by one bit to avoid overflow or underflow. For signed-right-shift, we simply sign-extend the input before using the standard right shift operator. Students are strongly encouraged to experiment with small code snippets until they feel comfortable with these signed operations.

3. Alternative Design

The alternative design for this lab is a five-stage bypassing processor for the same PARCv2 ISA. Once you get your baseline design working and passing all of your tests, you should copy your baseline processor design into `ProcAltDpathPRTL.py`, `ProcAltCtrlPRTL.py`, and `ProcAltPRTL.py` (or `ProcAltDpathVRTL.v`, `ProcAltCtrlVRTL.v`, and `ProcAltVRTL.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 PARCv2. We have also provide more comprehensive directed and random tests for `addu`, `lw`, `bne`, `mfc0`, `mtc0`. 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 simulator). Once these tests are working on the ISA simulator, 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 `addu` instruction for each each model.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% py.test ..
% py.test ../lab2_proc
% py.test ../lab2_proc/ProcFL_*_test.py
% py.test ../lab2_proc/ProcBaseRTL_*_test.py
% py.test ../lab2_proc/ProcAltRTL_*_test.py
% py.test ../lab2_proc/ProcFL_rr_test.py      -k test_addu
% py.test ../lab2_proc/ProcBaseRTL_rr_test.py -k test_addu
% py.test ../lab2_proc/ProcAltRTL_rr_test.py  -k test_addu
```

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 `addu`, `lw`, `mfc0`, and `mtc0` instructions. It should pass the very first test, but will fail the remaining tests for the `bne` instruction. This is because we use the `addiu` instruction in most of our control flow tests. Once you implement and test the `addiu` 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 `mfc0` instructions to receive input data from the test source, and ends with one or more `mtc0` 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 `test` subdirectory. You can use the assembly sequence generation functions that we provide in `test/inst_addu.py`, `test/inst_lw.py`, 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 8 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 `rN` 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 `r2`. On line 20, we send the value in register `r2` 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 `0x1000`. 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 9 shows example assembly sequence generation functions that tests the `addiu` instruction. The `gen_single_dest_byp_test` function is meant to just test that the processor correctly resolves RAW hazards for the destination register (i.e., that the consuming `mtc0` instruction correctly stalls or bypasses 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_byp_test` function is parameterized by the number of `nops` to insert after the instruction 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_byp_test` uses the `gen_single_dest_byp_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.

```
1  # Send value 0x00002000 from test source into processor
2  mfc0 r2, mngr2proc < 0x00002000
3  mfc0 r4, mngr2proc < 0x00002010
4
5  # Loop over four elements in array
6  addiu r1, r0, 4
7 loop:
8  lw    r3, 0(r2)
9  addiu r3, r3, 1
10 sw    r3, 0(r4)
11 addiu r2, r2, 4
12 addiu r4, r4, 4
13 addiu r1, r1, -1
14 bne   r1, r0, loop
15
16 # Read out the four results and send to test sink for verification
17
18 addiu r1, r0, 0x2010
19 sw    r2, 0(r1)
20 mtc0  r2, proc2mngr > 2
21
22 addiu r1, r0, 0x2014
23 sw    r2, 0(r1)
24 mtc0  r2, proc2mngr > 3
25
26 addiu r1, r0, 0x2018
27 sw    r2, 0(r1)
28 mtc0  r2, proc2mngr > 4
29
30 addiu r1, r0, 0x201c
31 sw    r2, 0(r1)
32 mtc0  r2, proc2mngr > 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 8: Example Assembly Program Illustrating Acceptable Syntax

Figure 10 shows an example assembly sequence generation function that tests the `j` instruction. Testing control flow instructions is particularly challenging since our test sink verifies values not control flow. We use the `addiu` 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 (`r3` 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 16 bits in the immediate field, but you should only use 15 bits to avoid issues with sign extension. This means you can track up to 15 control flow points in a single assembly sequence.

```
def gen_single_dest_byp_test( num_nops,
                              src, imm, result ):
  return """
    mfc0  r1, mngr2proc < {src}
    nop
    nop
    nop
    nop
    nop
    nop
    nop
    addiu r3, r1, {imm}
    {nops}
    mtc0  r3, proc2mngr > {result}
  """.format(
    nops = gen_nops( num_nops ),
    **locals()
  )

def gen_dest_byp_test():
  return [
    gen_single_dest_byp_test( 5, 1, 1, 2 ),
    gen_single_dest_byp_test( 4, 2, 1, 3 ),
    gen_single_dest_byp_test( 3, 3, 1, 4 ),
    gen_single_dest_byp_test( 2, 4, 1, 5 ),
    gen_single_dest_byp_test( 1, 5, 1, 6 ),
    gen_single_dest_byp_test( 0, 6, 1, 7 ),
  ]
```

Figure 9: Example Assembly Sequence Generation Function for `addiu` Instruction

```
def gen_multijump_test():
  return """

    # Use r3 to track the control flow pattern
    addiu r3, r0, 0

    j     label_a           # j -.
    addiu r3, r3, 0b000001  #    |
                            #    |
  label_b:                  # <--+-.
    addiu r3, r3, 0b000010  #    | |
    j     label_c           # j -+-+-.
    addiu r3, r3, 0b000100  #    | | |
                            #    | | |
  label_a:                  # <--' | |
    addiu r3, r3, 0b001000  #      | |
    j     label_b           # j ---' |
    addiu r3, r3, 0b010000  #        |
                            #        |
  label_c:                  # <------'
    addiu r3, r3, 0b100000  #

    # Carefully determine which bits are expected
    # to be set if jump operates correctly.
    mtc0  r3, proc2mngr > 0b101010
  """
```

Figure 10: Example Assembly Sequence Generation Function for `j` Instruction

```
from test import inst_addiu

@pytest.mark.parametrize( "name,test", [
  asm_test( inst_addiu.gen_basic_test     ),
  asm_test( inst_addiu.gen_dest_byp_test  ),
])
def test_addiu( name, test ):
  run_test( ProcFL, test )
```

Figure 11: Example Test Function for `addiu` in `ProcFL_rimm_test.py`

```
def test_addu_rand_delays( dump_vcd ):
  run_test( ProcSimpleRTL,
            inst_addu.gen_random_test,
            dump_vcd,
            src_delay      = 3,
            sink_delay     = 5,
            mem_stall_prob = 0.5,
            mem_latency    = 3 )
```

Figure 12: Example Test Function for `addu` with Random Delays in `ProcBaseRTL_rr_test.py`

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>_mngr_test.py` – Unit tests for manager insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_rimm_test.py` – Unit tests for reg-to-imm insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_rr_test.py` – Unit tests for reg-to-reg insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_mem_test.py` – Unit tests for memory insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_jump_test.py` – Unit tests for jump insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)
- `Proc<impl>_branch_test.py` – Unit tests for branch insts (`<impl>` = `FL`, `BaseRTL`, `AltRTL`)

Each test script already has the basic test we provide for you. To add more tests you simply add more rows to the `py.test` parameterized function. You should always start by running your tests

```
                fetch    decode                exe  mem  wb
cycle from src  PC       instruction           inst inst inst   to sink

   0: .        >         |                     |    |    |     > .
   1: #        > 00001000|                     |    |    |     >
   2: 00000001 > 00001004|mfc0  r01, r01       |    |    |     >
   3: #        > 00001008|nop                  |mfc0|    |     >
   4: #        > 0000100c|nop                  |nop |mfc0|     >
   5: #        > 00001010|nop                  |nop |nop |mfc0 >
   6: #        > 00001014|nop                  |nop |nop |nop  >
   7: #        > 00001018|nop                  |nop |nop |nop  >
   8: 00000002 > 0000101c|mfc0  r02, r01       |nop |nop |nop  >
   9: #        > #       |#                    |mfc0|nop |nop  >
  10: #        > #       |#                    |    |mfc0|nop  >
  11: #        > #       |#                    |    |    |mfc0 >
  12: #        > 00001020|addu  r03, r01, r02  |    |    |     >
  13: #        > 00001024|nop                  |addu|    |     >
  14: #        > 00001028|nop                  |nop |addu|     >
  15: #        > 0000102c|nop                  |nop |nop |addu >
  16: #        > 00001030|nop                  |nop |nop |nop  >
  17: #        > 00001034|nop                  |nop |nop |nop  >
  18: #        > 00001038|mtc0  r03, r02       |nop |nop |nop  >
  19: 0000000e > 0000103c|mfc0  r01, r01       |mtc0|nop |nop  >
  20: #        > 00001040|nop                  |mfc0|mtc0|nop  >
  21: #        > 00001044|nop                  |nop |mfc0|mtc0 > 00000003
```

Figure 13: Line Trace for ADDU 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 test source, test sink, and each of the five pipeline stages.

on the FL model to ensure that the test themselves are correct. So if we want to actually use the `gen_dest_byp_test` assembly sequence generation function on the FL model, we would modify the `ProcFL_rimm_test.py` test script as shown in Figure 11. We can run all of the tests for the `addiu` instruction and then just the new test case like this:

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

Once we are sure our tests pass on the FL model, then we can add a similar line to `ProcBaseRTL_rimm_test.py` to test the baseline design. Finally, we can and a similar line to `ProcAltRTL_rimm_test.py` to test the alternative design. We can use a similar process to run the `gen_dest_byp_test` assembly sequence generation function on the FL, baseline, and alternative models.

In addition to testing the functionality of each instruction, we also want to make sure every instruction functions correctly when faced with random delays on the test source, sink, and memory. Figure 12 illustrates the random delay testing we provide for the `addu` 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 13 illustrates a line trace from the baseline design for a assembly sequence generated to test the `addu` instruction. Extra annotations to indicate what the columns mean. The first column shows when data is sent from the test source into the processor, and the last column shows when data is sent from the processor to the test sink. The middle five columns 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.

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 `ProcDpathComponentsRTL_test.py`.

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)
- `cmplx-mult` : Element-wise complex multiplication
- `bin-search` : Binary search in a linear array of key/value pairs
- `masked-filter` : Masked convolution on a small image

For `vvadd` we provide both an unoptimized and optimized version. The optimized version unrolls the loop to minimize both data and control hazards. Each of these benchmarks are in the respective `proc_ubmark_<bmark>.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. PARCv3 Extensions

Each lab assignment includes additional extensions that would be required to transform the alternative design into a subsystem suitable for use in a full PARCv3 multicore. Students are free to work on these extensions, although they must implement them in a separate `lab2_proc_ext` subdirectory. Do not implement any design extensions in the main lab subdirectory! It is important that any work on design extensions not cause the tests for your baseline and alternative designs to fail. Design extensions will not be used to award extra credit, nor will they be factored into the grading of labs 1–4 in any way. However, design extensions can be factored into the grading for the baseline and alternative design section of the final lab assignment.

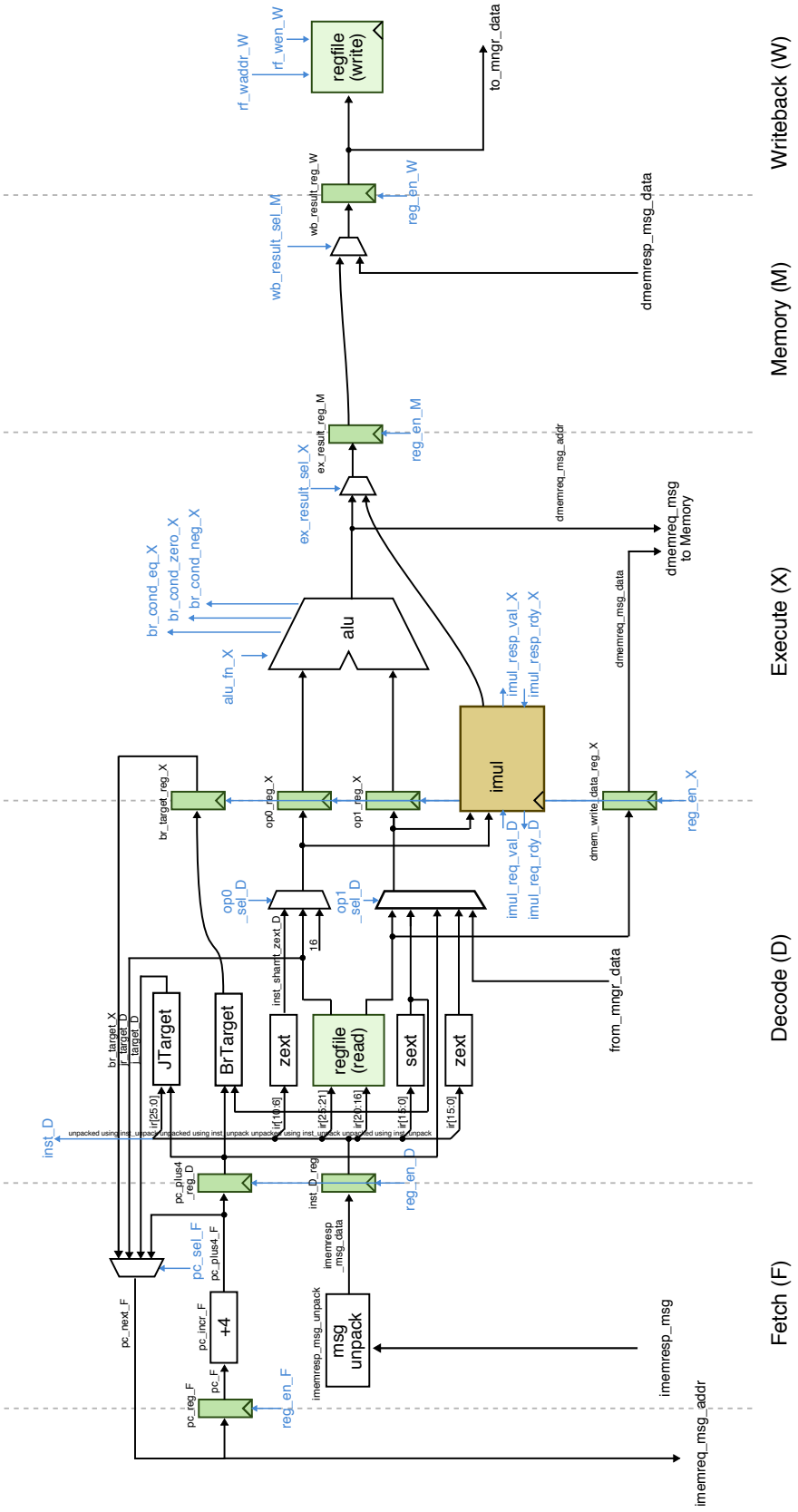
In order to evolve the PARCv2 alternative design in this lab into a full PARCv3 processor, we would need to implement the following additional instructions. The encoding and semantics for all of these instructions are in the PARC ISA manual.

- Additional basic instructions from PARCv3 including: jump-and-link register (`jalr`), division and remainder instructions (`div, divu, rem, remu`), sub-word loads and stores (`lb, lbu, lh, lhu, sb, sh`), and conditional moves (`movn, movz`).
- Atomic memory and concurrency operations `amo.add, amo.and, amo.or, sync`. These instructions will require modifications to the test memory as well, so the students would probably need to copy the `TestMemory` from `pclib` and modify it to support new memory request types for atomic memory operations.
- Floating-point instructions. There are quite a few floating-point instructions, and an RTL implementation of a floating-point unit can be challenging. Floating-point multiplication is actually much simpler than floating-point addition. Students might want to consider designing, testing, and evaluating a floating point unit in isolation with val/rdy message-base interfaces, and then integrating this floating point unit into the processor pipeline similar to the approach used with the multiplier.
- Support for the exceptions using the `syscall` and `eret` instructions. Testing exceptions can be challenging. Note that we cannot support running a real operating system on PARCv3, but after implementing support for `syscall` we can run a bare-metal "proxy kernel" which proxies system calls to a host machine. This enables interacting with files and the console from programs running on a PARCv3 processor, but the actual I/O is really happening on the host machine.

Advanced students interested in adding support for new instructions are strongly encouraged to discuss their ideas with the instructors. When adding new instructions, students would need to modify the `parc_encoding.py`, `parc_semantics.py`, and the FL model to be able to first write unit tests before actually implementing any new instructions in the processor. We might recommend students consider implementing subword loads/stores and the division/remainder instructions. This would enable students to execute C programs that use strings or the divide and modulus operators. Conditional moves are also interesting to implement, and might enable students to do a comparison of branches versus conditional moves in the final lab. Implementing just the floating-point addition instruction (`add.s`) might also be a fun an engaging way to learn about floating-point arithmetic. If we look forward to which instructions might enable the most benefit in the final lab assignment, we might recommend that students implement the atomic memory operations. This would enable more complicated parallelization strategies in the final lab assignment. Implementing atomic memory operations in the processor is not too difficult, but it does require a new memory message type, and will require changes to the memory lab as well.

Acknowledgments

This lab was created by Moyang Wang, Christopher Torng, Berkin Ilbeyi, Shreesha Srinath, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.



17

Figure 14: Baseline Design: Five-Stage Stalling Processor Datapath

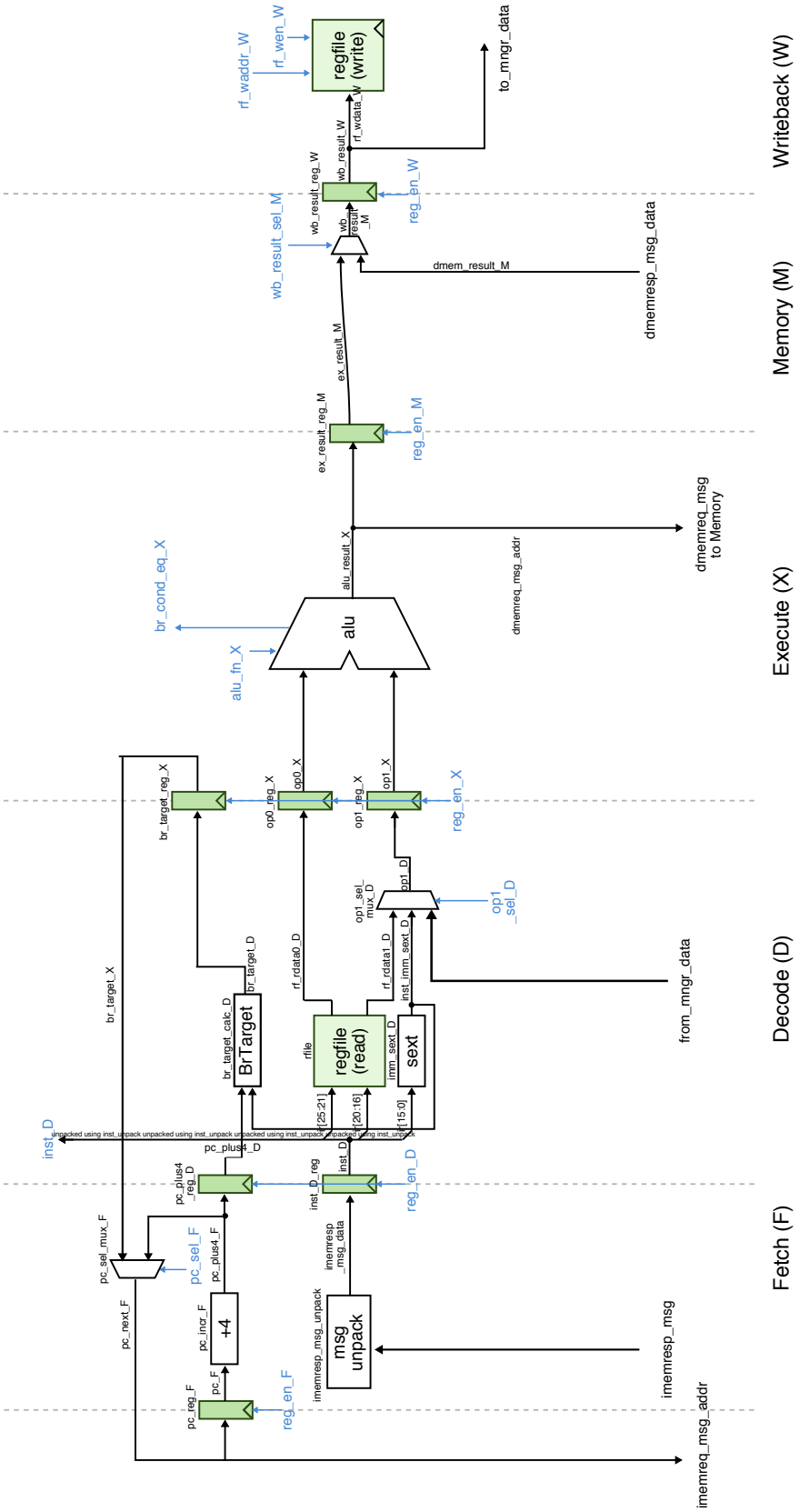


Figure 15: Initial Baseline Design Provided To Students