ECE 4750 Computer Architecture, Fall 2013

Lab 2: Pipelined Processor Microarchitecture

School of Electrical and Computer Engineering
Cornell University

revision: 2013-09-28-14-14

In this lab, you will design two pipelined processor microarchitectures for the PARC-tiny instruction set architecture. The baseline design is a 5-stage stalling processor pipeline which stalls the processor to handle data hazards and the alternative design is a 5-stage bypassing processor pipeline which adds bypass paths to the stalling processor to handle data hazards. You are given a stalling 5-stage pipelined processor template which only implements three instructions. We also provide you the datapath diagram and guidelines to implement the baseline design. You are required to implement the alternative design, create effective assembly tests to test both the processor implementations and evaluate the two processor pipeline microarchitectures using the provided benchmark suite. **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:

- implementing pipelined processor designs
- the squash and stall logic interaction to handle control and data hazards in stalling and bypassing processor designs
- effective design principles including abstraction, modularity, hierarchy, and encapsulation
- effective design methodologies including test-driven development and incremental development

This handout assumes that you have read and understand the course tutorials. To get started, you should access a course development machine and perform the following steps:

```
% cd ${HOME}
% source setup-ece4750.sh
% ece4750-lab-admin start ece4750-lab2-proc
% mkdir ${HOME}/ece4750/ece4750-lab2-proc/build
% cd ${HOME}/ece4750/ece4750-lab2-proc/build
% ../configure
% make
% make check
```

The course staff will update the lab harness to fix bugs and add more features. **These will be announced to the whole class and if you have already started on the lab then you or your lab partner must update the lab harness using the ece4750-lab-admin script. The details on this are included in the tutorial.**

The processor template that you were given should pass the assembly tests for three instructions (`addu`, `lw`, `bne`). **Currently, the both the bypassing and stalling processor has a stalling processor template implementation. Once your stalling processor implements all of the PARC-tiny instructions and passes all of the tests, you should copy this implementation as the bypassing processor and add the necessary changes to control and datapath to support bypassing for the alternative**

design. For this lab you will be working in the `plab2-proc` subproject which includes the following files:

- `plab2-proc/plab2-proc-PipelinedProcStall.v` – 5-stage stalling processor implementation
- `plab2-proc/plab2-proc-PipelinedProcStallDpath.v` – 5-stage stalling processor datapath
- `plab2-proc/plab2-proc-PipelinedProcStallCtrl.v` – 5-stage stalling processor control
- `plab2-proc/plab2-proc-PipelinedProcBypass.v` – 5-stage bypassing processor implementation
- `plab2-proc/plab2-proc-PipelinedProcBypassDpath.v` – 5-stage bypassing processor datapath
- `plab2-proc/plab2-proc-PipelinedProcBypassCtrl.v` – 5-stage bypassing processor control
- `plab2-proc/plab2-proc-dpath-components.v` – Datapath components shared across implementations such as ALU and register file
- `plab2-proc/plab2-proc-test-harness.v` – Test harness for the shared across implementations
- `plab2-proc/plab2-proc-PipelinedProcStall.t.v` – Stalling processor tests
- `plab2-proc/plab2-proc-PipelinedProcBypass.t.v` – Bypassing processor tests
- `plab2-proc/plab2-proc-sim-harness.v` – Simulator harness shared across implementations
- `plab2-proc/plab2-proc-sim-stall.v` – Simulator for stalling processor
- `plab2-proc/plab2-proc-sim-bypass.v` – Simulator for bypassing processor

You can use the `ece4750-lab-admin` script and the `git` command to add members to your lab group, use your own group repository to collaborate with your group members, and ultimately prepare your submission for uploading to CMS. **Remember to use the ece4750-lab-admin script to create the submission tarball. All the tests, simulation experiments and VCD dump generation must be run in the build directory!**

1. Introduction

Pipelining is a fundamental microarchitectural technique where multiple instructions overlap in exection. The processor microarchitecture is divided into stages with each stage performing specific tasks to process an instruction while handing off its results to the next stage in sequence. Pipelining allows us to achieve higher performance by lowering the CPI, without significantly affecting the cycle time. The tradeoff is that now we must be careful of the data and control hazards we discussed in class, which complicates the control logic. In this lab you will implement and evaluate two 5-stage PARC-tiny processor pipeline microarchitectures: stalling and bypassing. Later, in the course you will see how modern processors combine pipelining with techniques which exploit instruction level parallelism while trying to balance the design complexity, energy, area while achieving higher performance.

We have provided a template for the stalling PARC-tiny processor. This template has all the pipeline stages of the full processor, however it only implements three instructions: `addu`, `lw` and `bne`. The datapath for the processor template is as shown in Figure 1. This processor template has built-in logic to propagate stalls, squashes and valid signals throughout pipeline stages. In general, there are three classes of instructions in PARC-tiny: arithmetic operations, memory operations and control operations. Each of the instructions already implemented in the template belong to one of these classes.

In addition to the three instructions implemented, the template also implements two more instructions required for testing. These are `mfc0` and `mtc0` instructions, which allow the manager to send data to processor and receive data from processor respectively. We use the instructions to communicate with the manager, which is our testing or simulation harness. With the associated ports and

these instructions, we connect test sources and sinks with a val/rdy interface in order to be able to verify the functionality of the processor.

The documentation on the PARC-tiny instructions are available as part of the ISA document on the course webpage. Notice that PARC-tiny is a subset of the PARCv2 instruction set in that document. We will also upload an ISA document specifically for PARC-tiny in a few days.

The list of instructions that constitute PARC-tiny are below (note **boldfaced** instructions are the instructions that are already implemented as part of the template):

- Manager communication instructions for testing: **mfc0**, **mtc0**
- Register-Immediate Arithmetic Instructions: addiu, lui, ori, sra, sll
- Register-Register Arithmetic Instructions: **addu**, subu, and, or, slt
- Memory Instructions: **lw**, sw
- Jump Instructions: j, jal, jr
- Branch Instructions: **bne**, beq
- Multiply Instructions: mul

Before you start implementing the baseline and alternative designs, however, you should first familiarize yourself with the stalling processor template. All the source code for this lab can be found in the `plab2-proc` directory. As with our previous lab, we separate the processor into a control unit and a datapath. These modules are in separate files, `plab2-proc/plab2-proc-PipelinedProcStallCtrl.v` and `plab2-proc/plab2-proc-PipelinedProcStallDpath.v`, respectively. The `plab2-proc/plab2-proc-PipelinedProcStall.v` file wraps the control unit and datapath together and packs/unpacks memory messages. The assembly tests for the reference processor are available in `plab2-proc/plab2-proc-test-harness.v`. There are also models for the branch address calculation, ALU and Register File units instantiated in the datapath. The ALU unit we have provided has functionality required by the PARC-tiny ISA. To understand how the ALU and Register File units work take a look at the designs.

2. Baseline Design

The baseline design for this lab assignment is a 5-stage stalling processor which runs the PARC-tiny ISA. We will provide the datapath for the stalling processor and the highlight the key features of the control unit which will let you implement the processor design. As usual, we will decompose the processor design into control and datapath units. Since the processor is a more complex design than the iterative multiplier designs you implemented in the previous lab, we use the following file organization to implement the processor. This decomposition is an example of the *modular design principle* you have learnt about.

- `plab2-proc/plab2-proc-PipelinedProcStall.v` – 5-stage stalling processor implementation
- `plab2-proc/plab2-proc-PipelinedProcStallDpath.v` – 5-stage stalling processor datapath
- `plab2-proc/plab2-proc-PipelinedProcStallCtrl.v` – 5-stage stalling processor control

The pipelined 5-stage stalling processor is composed of the 5 stages we learned about in class: (F)etch, (D)ecode, E(X)ecute, (M)emory, and (W)riteback. The control unit uses the `PipeCtrl` model to handle the stall and squash logic. You will pipeline control signals and output them to the datapath at the stages they will be used. The datapath for the stalling processor is as given in Figure 2. The blue boxes and signals indicate the control and status signals between the control and datapth units. For each instruction you should determine what would be the values of the control signals when an instruction gets decoded in the Decode stage. For a given instruction, we obtain the control signals from the control module.

As earlier, you will implement the datapath of the processor structurally by instantiating a child model for each of the blocks in the datapath diagram. This recursive modular decomposition is an example of the *hierarchy design principle* which you should be familiar with. Note that for some of the functional blocks such as the jump and branch target calculations you could create a separate model. For such blocks, you should add your design to `plab2-proc-dpath-components.v` and write unit tests to ensure correct functionality. If you find a need to change the datapath diagram in anyway be sure to implement the new blocks you need, unit test them and write about your design decisions in your lab report. You can reuse the ALU, and Register File units we have provided.

To add a new instruction to the stalling PARCv1 processor you need to determine the datapath unit changes, add a new entry to the decode control signals model to set the control units, add new control/status signals between the control-datapath units and finally update the top-level processor wrapper unit which instantiates the control-datapath units to compose the processor.

As an example, let us consider the `j` instruction. This instruction requires us to modify the datapath. In particular, we need a new functional block, similar to `br_targ` to calculate the jump address from the given PC and immediate bits in the decode stage. We also need to forward this to the front of the pipeline, to the fetch stage. Finally, we also need to use a 3-input `pc_sel_mux_F`.

In order to do these, we can modify our datapath starting from the F stage in `plab2-proc-PipelinedProcStallDpath.v`. We need a new wire that will forward the jump address from D stage to F stage. Because we always have to declare the wires before we use them, we have to declare this wire in the F stage. We also need to grow the PC select multiplexer:

```
//------------------------------------------------------------------------
// F stage
//------------------------------------------------------------------------

// more wires and components here

wire [31:0] j_target_D;

vc_Mux3 #(32) pc_sel_mux_F
(
  .in0  (pc_plus4_F),
  .in1  (br_target_X),
  .in2  (j_target_D),
  .sel  (pc_sel_F),
  .out  (pc_next_F)
);

// more wires and components here
```

In the decode stage, assuming we have created the `plab2_proc_JTarget` module which calculates the jump target, here is the modification we need to make to the D stage:

```
//------------------------------------------------------------------------
// D stage
//------------------------------------------------------------------------

// more wires and components here
```

```
plab2_proc_JTarget j_target_calc_D
(
  .pc_plus4   (pc_plus4_D),
  .imm_target (inst_target_D),
  .j_target   (j_target_D)
);

// more wires and components here
```

The final thing we need to modify in the datapath is to increase the number of bits for `pc_sel_F`, because a 3-input multiplexer needs a 2-bit select. We need to make sure we grow the number of bits for each reference of `pc_sel_F`, in the datapath, control and in the top level.

In the control, in the F stage, we need to define a new PC Mux select parameter that corresponds to the input we picked for the jump target into the `pc_sel_mux_F`, which is 2. We need to introduce a new wire coming from the D stage which will tell us that we have a jump or not. So the F stage control modifications will look like:

```
//------------------------------------------------------------------------
// F stage
//------------------------------------------------------------------------

// more wires and components here

// PC Mux select

localparam pm_x     = 2'dx; // Don't care
localparam pm_p     = 2'd0; // Use pc+4
localparam pm_b     = 2'd1; // Use branch address
localparam pm_j     = 2'd2; // Use jump address (imm)

wire [1:0] j_pc_sel_D;
wire [1:0] br_pc_sel_D;

assign pc_sel_F = ( br_pc_sel_X ? br_pc_sel_X :
                  (  j_pc_sel_D ? j_pc_sel_D  :
                                  pm_p          ) );

// more wires and components here
```

In the D stage of the control, we need to add a new control field to distinguish jump instructions, add parameters to contain the options, and add a new entry to the decode table:

```
//------------------------------------------------------------------------
// D stage
//------------------------------------------------------------------------

// more wires and components here
```

```
// Jump type

localparam j_x = 2'dx; // Don't care
localparam j_n = 2'd0; // No jump
localparam j_j = 2'd1; // jump (imm)

// Instruction Decode

// ...
reg     j_type_D;
// ...

task cs
(
  // ...
  input   cs_j_type,
  // ...
);
begin
  // ...
  j_type_D = cs_j_type;
  // ...
end
entask

always @(*) begin
  casez ( inst_D ) begin
    // ...                              ...  j     ...
    //                                      type
    `PISA_INST_MSG_J       :cs( y,  j_j, br_none, am_x,   // ...
    // ...
  endcase
end
// more wires and components here
```

When you add a new field to the control decoder, make sure you add the no-jump flag (`j_n` in this case) for any other instruction that is not a jump.

Another modification you need to do is to set the `j_pc_sel_D`, which will be similar to the one for branches:

```
assign j_pc_sel_D = ( val_D && ( j_type_D == j_j ) ) ? pm_j : pm_p;
```

The final thing you need to do when you branch is to drive the squash signal in the D stage:

```
wire squash_j_D;

assign squash_j_D = ( j_pc_sel_D == pm_j );

assign squash_D = squash_j_D;
```

Add the necessary functional units and control unit functionality for each instruction leading to the required PARC-tiny implementation. Make sure that you add assembly tests for each new instruction you implement. We provide you with most of the assembly tests, and you need to implement the remaining.

In the your lab report, discuss how your processor handles the structural, data and control various *transaction delay latencies*. Summarize the baseline design and any changes you made to the provided datapath.

3. Alternative Design

More information about this later.

4. Testing Strategy

Once you run `make` in your build directory, you can run the unit tests for the entire project:

```
% make check
```

This will run all the unit tests for the entire project (including, vc, plab1 etc.) and it will give a summary on how many test suites have passed and how many have failed. You will see an output similar to this:

```
[ passed ]  P: 17 F:  0  vc
[ passed ]  P:  1 F:  0  pex-regincr
[ passed ]  P:  3 F:  0  pex-sorter
[ passed ]  P:  3 F:  0  pex-gcd
[ passed ]  P:  1 F:  0  pisa
[ passed ]  P:  5 F:  0  plab1-imul
[ FAILED ]  P:  0 F:  2  plab2-proc

Test Summary:
 - Number of test subprojects   :   7
 - Number of test suites passed :  30
 - Number of test suites failed :   2
```

This means all of the subprojects other than `plab2-proc` are passing the unit tests. `plab2-proc` is failing with two test suites. Now we check for `plab2-proc` specifically:

```
% make check-plab2-proc

Subproject: plab2-proc

 [ FAILED ]  P: 13 F: 11  plab2-proc-PipelinedProcBypass
 [ FAILED ]  P: 13 F: 11  plab2-proc-PipelinedProcStall

 Test Summary:
  - Number of test suites        :   2
  - Number of test cases passed  :  26
  - Number of test cases failed  :  22
```

Now we can see that both of our processor implementations are passing 13 of the test cases but failing for the other ones. We should focus on implementing instructions on the baseline stalling processor first, so we need to see which test cases are failing:

```
% ./plab2-proc-PipelinedProcStall +verbose=1

Test Suite: plab2-proc-PipelinedProcStall
 + Test Case 1: mngr interfacing
 + Test Case 2: addu
 + Test Case 3: addu src delay=4, sink delay=6
 + Test Case 4: addu mem delay=4
 + Test Case 5: addu src delay=4, sink delay=4, mem delay=4
 + Test Case 6: bne
 + Test Case 7: bne src delay=4, sink delay=6
 + Test Case 8: bne mem delay=4
 + Test Case 9: bne src delay=4, sink delay=4, mem delay=4
 + Test Case 10: lw
 + Test Case 11: lw src delay=4, sink delay=6
 + Test Case 12: lw mem delay=4
 + Test Case 13: lw src delay=4, sink delay=4, mem delay=4
 + Test Case 14: addiu
    [ FAILED ] msg, expected = 00000000, actual = 0000200c
    [ FAILED ] msg, expected = 00000009, actual = 0000200c
    [ FAILED ] msg, expected = 7fff7fff, actual = 0000200c
 + Test Case 15: lui
    [ FAILED ] msg, expected = 00000000, actual = 7fffffff
    [ FAILED ] msg, expected = faa00000, actual = 7fffffff
    [ FAILED ] msg, expected = 7afe0000, actual = 7fffffff
 + Test Case 16: sw
    [ FAILED ] msg, expected = cafecafe, actual = xxxxxxxx
    [ FAILED ] msg, expected = deadbeef, actual = xxxxxxxx
    [ FAILED ] msg, expected = 01230123, actual = xxxxxxxx
    [ FAILED ] msg, expected = 00abefac, actual = xxxxxxxx
 + Test Case 17: sw src delay=4, sink delay=6
    [ FAILED ] msg, expected = cafecafe, actual = xxxxxxxx
    [ FAILED ] msg, expected = deadbeef, actual = xxxxxxxx
    [ FAILED ] msg, expected = 01230123, actual = xxxxxxxx
    [ FAILED ] msg, expected = 00abefac, actual = xxxxxxxx

 ...
```

This tells us that we are passing the test cases for the three instructions that are given in the template, as well as manager interfacing and the versions of the tests with random delays in the source, sink and/or the memory. The first failing test tells us that `addiu` fails the tests. After this, as we look deeper and deeper into the particular failing test case, we can turn the line tracing on and we can turn off the other test cases with the following command:

```
% ./plab2-proc-PipelinedProcStall +verbose=1 +trace=1 +test-case=14

VCD info: dumpfile plab2-proc-PipelinedProcStall-test.vcd opened for
```

```
output.

 Test Suite: plab2-proc-PipelinedProcStall
  + Test Case 14: addiu
   0: .        >         |                      |    |    |     > .
   1: #        > 00001000|                      |    |    |     >
   2: 00000000 > 00001004|mfc0  r02, r00        |    |    |     >
   3: #        > 00001008|addiu r03, r02, 0x0000|mfc0|    |     >
   4: #        > 0000100c|mtc0  r03, r00        |addi|mfc0|     >
   5: 00000005 > 00001010|mfc0  r02, r00        |mtc0|addi|mfc0 >
   6: #        > 00001014|addiu r03, r02, 0x0004|mfc0|mtc0|addi >
   7: #        > 00001018|mtc0  r03, r00        |addi|mfc0|mtc0 > xxxxxxxx
     [ FAILED ] msg, expected = 00000000, actual = xxxxxxxx
   8: 7fffffff > 0000101c|mfc0  r02, r00        |mtc0|addi|mfc0 >
   9: .        > 00001020|addiu r03, r02, 0x8000|mfc0|mtc0|addi >
  10: .        > 00001024|mtc0  r03, r00        |addi|mfc0|mtc0 > xxxxxxxx
     [ FAILED ] msg, expected = 00000009, actual = xxxxxxxx
  11: .        > 00001028|nop                   |mtc0|addi|mfc0 >
  12: .        > 0000102c|nop                   |nop |mtc0|addi >
  13: .        > 00001030|nop                   |nop |nop |mtc0 > xxxxxxxx
     [ FAILED ] msg, expected = 7fff7fff, actual = xxxxxxxx
```

Line trace is very useful in debugging your design. In our testing framework, we have a test source and a test sink (that `mfc0` and `mtc0` instructions communicate with) which are shown at the leftmost and the rightmost columns respectively. We can see that the test source stalled for a cycle (shown with # character), and sent the value 0 on cycle 2. Each column in the processor line tracing represents a pipeline stage, for F, D, X, M, W stages respectively, separated with a | character. In the F stage, since we haven't decoded the instruction yet, we represent the instruction with the PC only. In decode, we have finally figured out what the instruction means, so we print the disassembly for the instruction at this state. The later pipeline stages use compacter 4-character opcodes to save space. So in this particular case, we can see that the sink is getting Xs from the processor. This is because we haven't implemented `addiu` yet, so the processor is not producing the correct result.

5. Evaluation

More information about this later.

6. Extensions

More information about this later.

Acknowledgments

This lab was created by Berkin Ilbeyi, Shreesha Srinath, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University. Derek Lockhart designed and implemented the PyMTL hardware modeling framework.

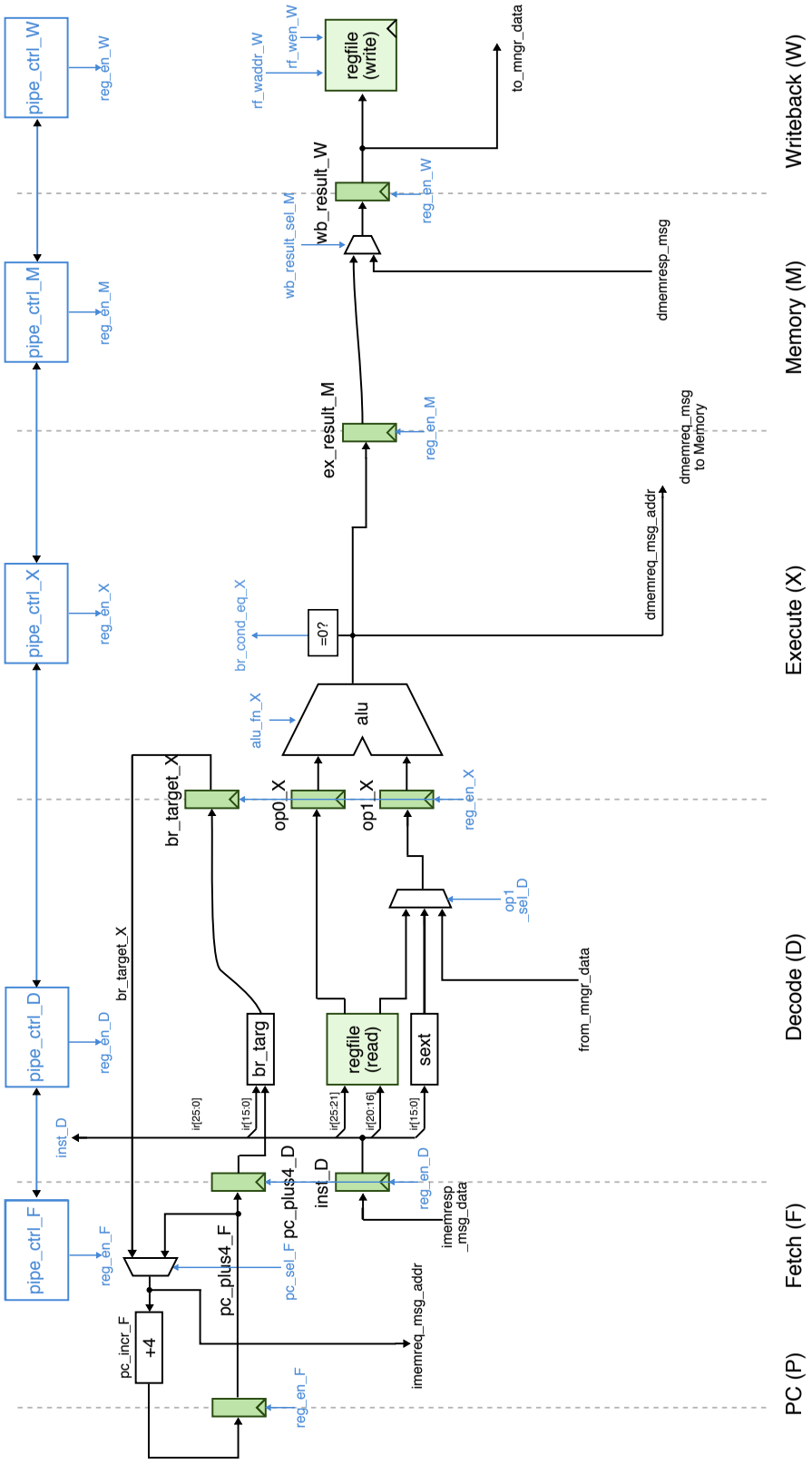

Figure 1: 5 stage stalling Processor Template Datapath

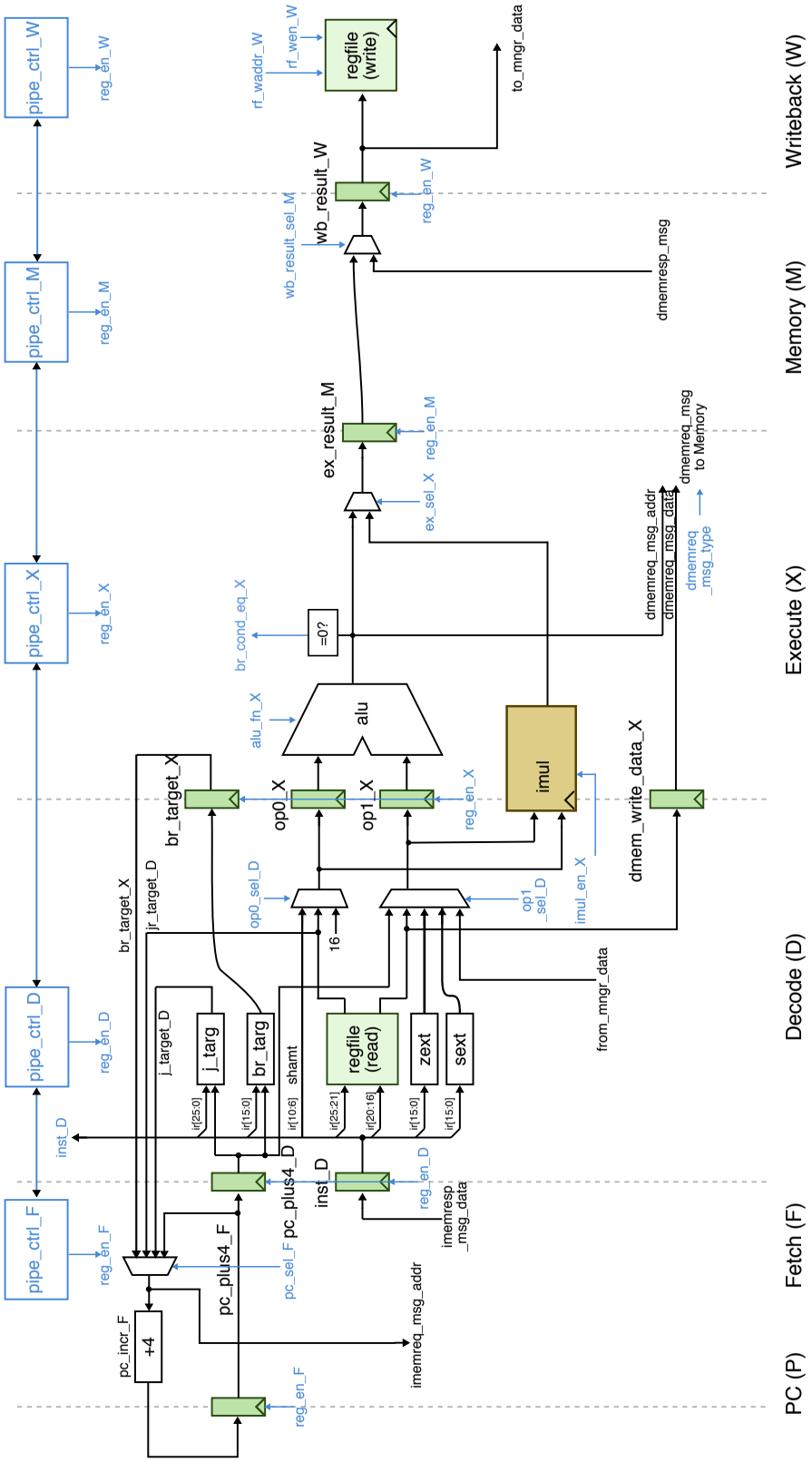

Figure 2: 5 stage stalling Processor Datapath