ECE 4750 Computer Architecture, Fall 2016
Lab 2: Pipelined Processor

School of Electrical and Computer Engineering
Cornell University

revision: 2023-09-17-00-26

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. **As with all lab assignments, 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 begin the lab, download the starter files from the course website and extract it into a directory titled `sim` with the following commands:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750

# download the file to this folder from Canvas

% tar -xvzf lab2.tar.gz
% cd sim
% make setup
```

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

```
% cd ${HOME}/ece4750/sim/lab2_proc
% make run-all
```

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:

- `ProcFLMultiCycle.v` – Functional Level Multi-Cycle processor

- `DropUnit.v` – Verilog RTL unit for dropping inst mem response on squash
- `tinyrv2_encoding.v` – Verilog RTL helper constants, functions for TinyRV2 ISA
- `ProcDpathComponents.v` – Verilog data-path components
- `ProcBaseDpath.v` – Verilog RTL stalling processor's datapath
- `ProcBaseCtrl.v` – Verilog RTL stalling processor's control unit
- `ProcBase.v` – Verilog RTL stalling processor
- `ProcAlt.v` – Verilog RTL bypassing processor

- `tinyrv2_encoding_assembler.py` – Tinyrv2 Assembler
- `asm/` – Example assembly files for testing and for microbenchmarks

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 (multicycle) 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 to exploit instruction-level parallelism, enabling improved performance at the cost of increased energy, area, and complexity over this lab.

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 the public course web page. 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 executign 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 FSM multi-cycle model of a TinyRV2 processor. You can find this model in `ProcFLMultiCycle.v`. 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

```
* 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     |
 +------------+---------+---------+------+---------+-------------+
```

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.

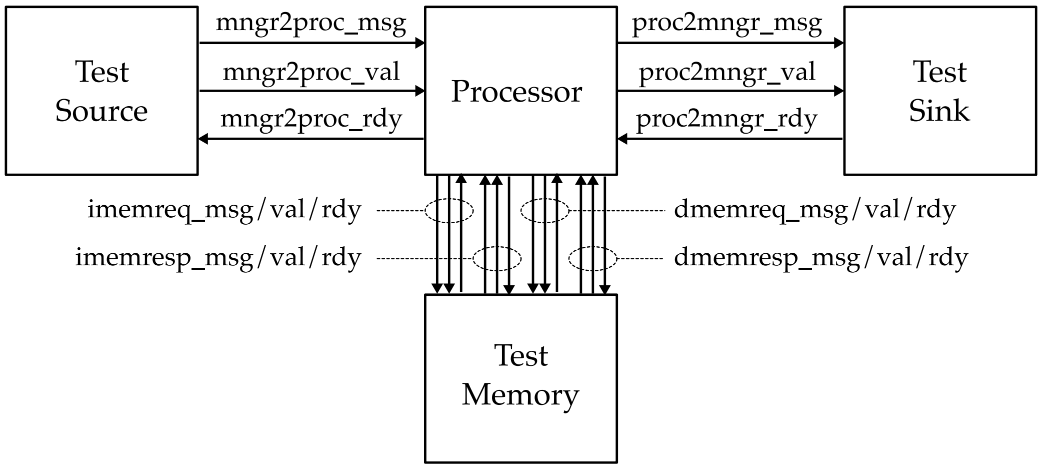

Figure 2: Processor System – The processor is integrated with a test source, test sink, and test memory for testing and evaluation.

multi-cycle, 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.

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

- `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.

Verilog `structs` are defined in `vc/mem-msgs.v` included within the lab release. 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

Mem Req	type	opaque	addr	len	data
Bits	76–74	73–66	65–34	33–32	31–0

Mem Resp	type	opaque	test	len	data
Bits	46–44	43–36	35–34	33–32	31–0

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 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 15. 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 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 may need to add and/or modify datapath components as you support more TinyRV2 instructions in your baseline, but especially in the alternate 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 `vclib`. We have also provided you the initial implementations of the immediate generator unit and the ALU (see `ProcDpathALU.v` and `ProcDpathImmGen.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.

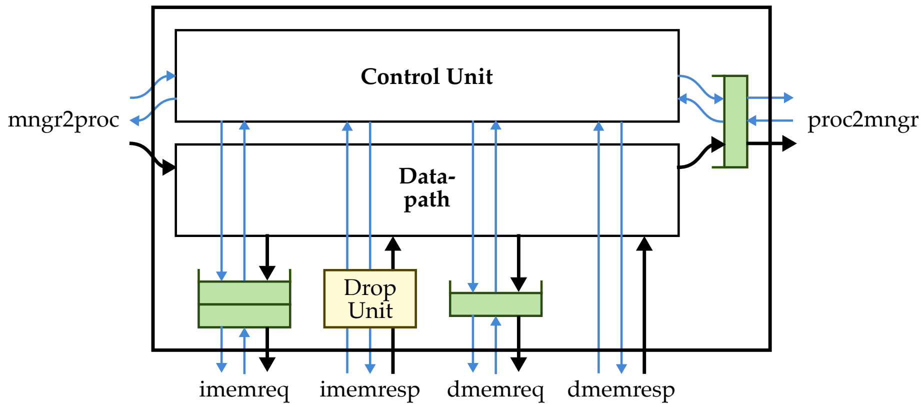

Figure 4: Processor Datapath and Control Composition – In addition to the datapath and control unit, the processor also includes bypass queues on output val/rdy interfaces and a drop unit for the input val/rdy instruction memory response interface.

If you look carefully at the datapath diagram in Figure 15, 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 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 `DropUnit.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

```
always_comb begin
  casez ( inst_D )
    //                         br      imm   rs1 op2    rs2 alu      dmm wbmux rf
    //                     val type    type   en muxsel  en fn       typ sel   wen csrr csrw
    `RV2ISA_INST_NOP   :cs( y, br_na,  imm_x, n, bm_x,   n, alu_x,   nr, wm_a, n,  n,   n    );
    `RV2ISA_INST_ADD   :cs( y, br_na,  imm_x, y, bm_rf,  y, alu_add, nr, wm_a, y,  n,   n    );
    `RV2ISA_INST_LW    :cs( y, br_na,  imm_i, y, bm_imm, n, alu_add, ld, wm_m, y,  n,   n    );
    `RV2ISA_INST_BNE   :cs( y, br_bne, imm_b, y, bm_rf,  y, alu_x,   nr, wm_a, n,  n,   n    );
    `RV2ISA_INST_CSRR  :cs( y, br_na,  imm_i, n, bm_csr, n, alu_cp1, nr, wm_a, y,  y,   n    );
    `RV2ISA_INST_CSRW  :cs( y, br_na,  imm_i, y, bm_rf,  n, alu_cp0, nr, wm_a, n,  n,   y    );

    `RV2ISA_INST_ADDI  :cs( y, br_na,  imm_i, y, bm_imm, n, alu_add, nr, wm_a, y,  n,   n    );

    default            :cs( n, br_x,   imm_x, n, bm_x,   n, alu_x,   nr, wm_x, n,  n,   n    );
  endcase
end
```

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

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 import 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:

- **imul.req.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!).
- **imul.req.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. You should always originate a stall in D if the multiplier is not ready regardless of what instruction is in the D stage (e.g., we do not want an `add` instruction to "slip by" a multiply instruction that is using the multiplier).
- **imul.resp.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.resp.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!).

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

```
# signed-less-than operation

logic slt;
assign slt = $signed(a) < $signed(b);
```

```
# signed-right-shift

logic [31:0] srs
assign srs = $signed(a) >>> b
```

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

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 val/rdy 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 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 `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 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. 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 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 one instruction in TinyRV2. 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 run these tests on the FL muticycle model (ISA emulator). Once these tests are working on the ISA emulator, you can move on to testing the baseline and alternative designs. As in the last lab, you will write individual unit testbenches for all your modules, which you can run as follows:

```
% cd ${HOME}/ece4750/sim/lab2_proc
% make utb_XXXX.v.sim [DESIGN=${DESIGN}$] [RUN_ARG=--trace] [COVERAGE=${COVERAGE}]
```

The following commands illustrate how to run just the tests for this lab, and how to run just the tests for the `add` instruction for each model.

```
% cd ${HOME}/ece4750/sim/lab2_proc
% make add.hex
% make add.hex.sim DESGIN=${DESIGN}$ [RUN_ARG=--trace][COVERAGE=${COVERAGE}]
% diff -y add.hex memdump.hex
```

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` instructions.

Our directed testing will be done using short assembly sequences represented as multi-line text files. Each assembly sequence usually starts with one or more instructions to initialize the registers. You can either use `csrr` instructions to receive input data from the test source, `lw` to load data from the data section, or obtain them through intermediate operands with instructions such as `lui` or `addi`.

Depending on your testing methodology, you should either end with `sw` instructions to store data back to memory, or with `csrw` instructions to send 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

```
.text
xor x1, x1, x1
xori x1, x1, 0x0200
addi x3, x1, 0x0004
xor x1, x1, x1
lui x1, 0x0002      #location of data section
sw x3,  label_a(x1) #label_a is 0x000 \
                    #only lower 12 bits is used
.data               #Data section
label_a:
.word 5000
```

Figure 7: Example Assembly Sequence for `addi` Instruction

```
#.text
00000200:B3C01000
00000204:93C00020
00000208:93814000
0000020C:B3C01000
00000210:B7200000
00000214:23A03000
00000218:F32000FC
#.data
#label_a is at 0x2000
00002000:88130000
```

Figure 8: Hexdump of Example Assembly Sequence `addi` Instruction

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.

4.1. Assembly Testing

To test the processor with assembly instructions, you will need to assemble your code:

```
% cd ${HOME}/ece4750/sim/lab2_proc
% make add.hex
```

After you have done so, you can have your processor execute the assembled instructions. Lastly, you may use the diff command to highlight the results:

```
% make add.hex.sim DESGIN=${DESIGN} [COVERAGE=${COVERAGE}]
% diff -y add.hex memdump.hex
```

The following instructions allow you to have the processor execute the assembled instructions and compare the memory dump against the ISA emulator:

```
% make add.hex.diff DESGIN=${DESIGN} [COVERAGE=${COVERAGE}]
```

Figure 10 shows a simple assembly program that is meant to illustrate the assembly syntax we will be using for testing. The assembly program is first assembled with an assembler, a Python script, which assembles the assembly instructions to Hexadecimal numbers on a 1:1 ratio. In order for each of the

```
#.text
00000200:B3C01000
00000204:93C00020
00000208:93814000
0000020C:B3C01000
00000210:B7200000
00000214:23A03000
00000218:F32000FC
#.data
00002000:88130000
```

Figure 9: Hexdump of Example Assembly Sequence `addi` Instruction

```
#.text
00000200:B3C01000
00000204:93C00020
00000208:93814000
0000020C:B3C01000
00000210:B7200000
00000214:23A03000
00000218:F32000FC
#.data
|00002000:04020000
```

Figure 10: Output Memdump of Example Assembly Sequence `addi` Instruction

instructions and hexdump to align with each other, we purpose leave out line 9 and 10, as sections declarations and labels do not have corresponding to any instruction, but is rather used as a tool to manipulate the address the of the hexdump, or a way to reference a specific location, respectively. For the purpose of this class the .text section starts at 0x200, while the .data section starts at 0x2000. The format of the hexdump is Addr:Data, but student need to be careful that the data is in little endian. In this example in Figure 10, line 2 zero the registers with `xor`, line 3 uses `xori` to initialize the value of the register, line 4 preforms the `addi` operation, line 5-6 set the data offset location to the register `x1` to prepare for storing the output, and line 7 stores x1 to data section. Line 8 is an instruction that we add to the end of the .text section to halt the processor (csrr mgn2proc). In additional to hexdump, the assembler also generate the .in and .out files for the sink and source testing.

4.2. Testing with Sink and Sources

Figure 11 shows a simple assembly program that is meant to illustrate the assembly syntax we will be using for testing with sources and sinks. 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. The top of the files shows two ways that the student can use to set the initial values in the registers. 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 11) and can also be used as the target for control flow instructions (e.g., `bne` instruction on line 18). 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 24, 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 39, 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 42–51). The final value of any modified value in dest array can be view by inspecting the hexdump.

Figure 12 shows example assembly sequence to test the `addi` instruction. We include plenty of `nop` instructions before the instruction under test to ensure there are no RAW hazards with reading the source register.

Figure 13 shows an example assembly sequence tests the `jal` instruction. Testing control flow instructions is particularly challenging since our test hexdump/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.

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. You will need to add random delay testing for each instruction you implement.

```
# Send value 0x00002000 from test source into processor
csrr x2, mngx2proc < 0x00002000
#lui x1, 0x00002
#ori x1, 0x000
csrr x4, mngx2proc < 0x00002010
#lui x4, 0x00002
#ori x4, 0x010
# Loop over four elements in array

addi  x1, x0, 4
loop:
lw    x3, 0(x2)
addi  x3, x3, 1
sw    x3, 0(x4)
addi  x2, x2, 4
addi  x4, x4, 4
addi  x1, x1, -1
bne   x1, x0, loop

# Read out the four results and send to test sink for verification

addi  x1, x0, 0x2010
lw    x2, 0(x1)
csrw  proc2mngr, x2 > 2

addi  x1, x0, 0x2014
lw    x2, 0(x1)
csrw  proc2mngr, x2 > 3

addi  x1, x0, 0x2018
lw    x2, 0(x1)
csrw  proc2mngr, x2 > 4

addi  x1, x0, 0x201c
lw    x2, 0(x1)
csrw  proc2mngr, x2 > 5

# Data section
.data

# src array
.word 0x00000001
.word 0x00000002
.word 0x00000003
.word 0x00000004

# dest array
.word 0x00000000
.word 0x00000000
.word 0x00000000
.word 0x00000000
```

Figure 11: Example Assembly Program Illustrating Acceptable Syntax

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 14 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 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,

```
  csrr  x1, mngr2proc < 1
  nop
  nop
  nop
  nop
  nop
  nop
  nop
  nop
  addi  x3, x1, 1
  nop
  nop
  nop
  nop
  nop
  csrw  proc2mngr, x3 > 2

```

Figure 12: Example Assembly Sequence for `addi` Instruction

```
  # Use x3 to track the control flow pattern
  addi  x3, x0, 0                    # 0x00000200

  jal   x1, label_a        # j -.     # 0x00000204
  addi  x3, x3, 0b000001   #    |     # 0x00000208
                           #    |     #
label_b:                   # <--+-.   #
  addi  x3, x3, 0b000010   #    | |   # 0x0000020c
  addi  x5, x1, 0          #    | |   # 0x00000210
  jal   x1, label_c        # j -+-+-. # 0x00000214
  addi  x1, x3, 0b000100   #    | | | # 0x00000218
                           #    | | | #
label_a:                   # <--' | | #
  addi  x3, x3, 0b001000   #      | | # 0x0000021c
  addi  x4, x1, 0          #      | | # 0x00000220
  jal   x1, label_b        # j ---' | # 0x00000224
  addi  x3, x3, 0b010000   #        | # 0x00000228
                           #        | #
label_c:                   # <------' #
  addi  x3, x3, 0b100000   #          # 0x0000022c
  addi  x6, x1, 0          #          # 0x00000230

  # Carefully determine which bits are expected
  # to be set if jump operates correctly.
  csrw  proc2mngr, x3 > 0b101010

  # Check the link addresses
  csrw  proc2mngr, x4 > 0x00000208
  csrw  proc2mngr, x5 > 0x00000228
  csrw  proc2mngr, x6 > 0x00000218

```

Figure 13: Example Assembly Sequence Generation Function for `jal` Instruction

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

   0: .         >         |                     |    |    |     > .
   1: #         > 00000200|                     |    |    |     >
   2: 00000001  > 00000204|csrr  x01, 0xfc0     |    |    |     >
   3: #         > 00000208|nop                  |csrr|    |     >
   4: #         > 0000020c|nop                  |nop |csrr|     >
   5: #         > 00000210|nop                  |nop |nop |csrr >
   6: #         > 00000214|nop                  |nop |nop |nop  >
   7: #         > 00000218|nop                  |nop |nop |nop  >
   8: 00000002  > 0000021c|csrr  x02, 0xfc0     |nop |nop |nop  >
   9: #         > #       |#                    |csrr|nop |nop  >
  10: #         > #       |#                    |    |csrr|nop  >
  11: #         > #       |#                    |    |    |csrr >
  12: #         > 00000220|add   x03, x01, x02  |    |    |     >
  13: #         > 00000224|nop                  |add |    |     >
  14: #         > 00000228|nop                  |nop |add |     >
  15: #         > 0000022c|nop                  |nop |nop |add  >
  16: #         > 00000230|nop                  |nop |nop |nop  >
  17: #         > 00000234|nop                  |nop |nop |nop  >
  18: #         > 00000238|csrw  x03, 0x7c0     |nop |nop |nop  >
  19: 0000000e  > 0000023c|csrr  x01, 0xfc0     |csrw|nop |nop  >
  20: #         > 00000240|nop                  |csrr|csrw|nop  >
  21: #         > 00000244|nop                  |nop |csrr|csrw > 00000003
```

Figure 14: 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 test source, test sink, and each of the five pipeline stages.

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. Debugging through line tracing alone will simply not be possible; students will almost certainly need to use gtkwave to view 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 a unit testbench utb_*v.

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 ${HOME}/ece4750/sim/build
% make add.hex
# Assemble the asm file to hexdump
% make ubench_vvadd_opt.hex.sim DESGIN=${DESIGN}$ [RUN_ARG=--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 DESIGN environmental variable. 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. You can use "diff -y" to check the output memory-dump 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.

In additional to assembly tests, you will be in charge of writing micro benchmarks to evaluate the performance of the processor. While some micro benchmarks will be provided in the discussion section, ultimately it is your job to create a set of instruction to show case your processor's performance.

Acknowledgments

Socrates Wong, Cecilio C. Tamarit, and José Martínez prepared this version of Lab 2 as part of the course ECE 4750 Computer Architecture at Cornell University. Significant parts were adopted from earlier versions by Shunning Jiang, Shuang Chen, Ian Thompson, Moyang Wang, Christopher Torng, Berkin Ilbeyi, Shreesha Srinath, Christopher Batten, and Ji Kim.

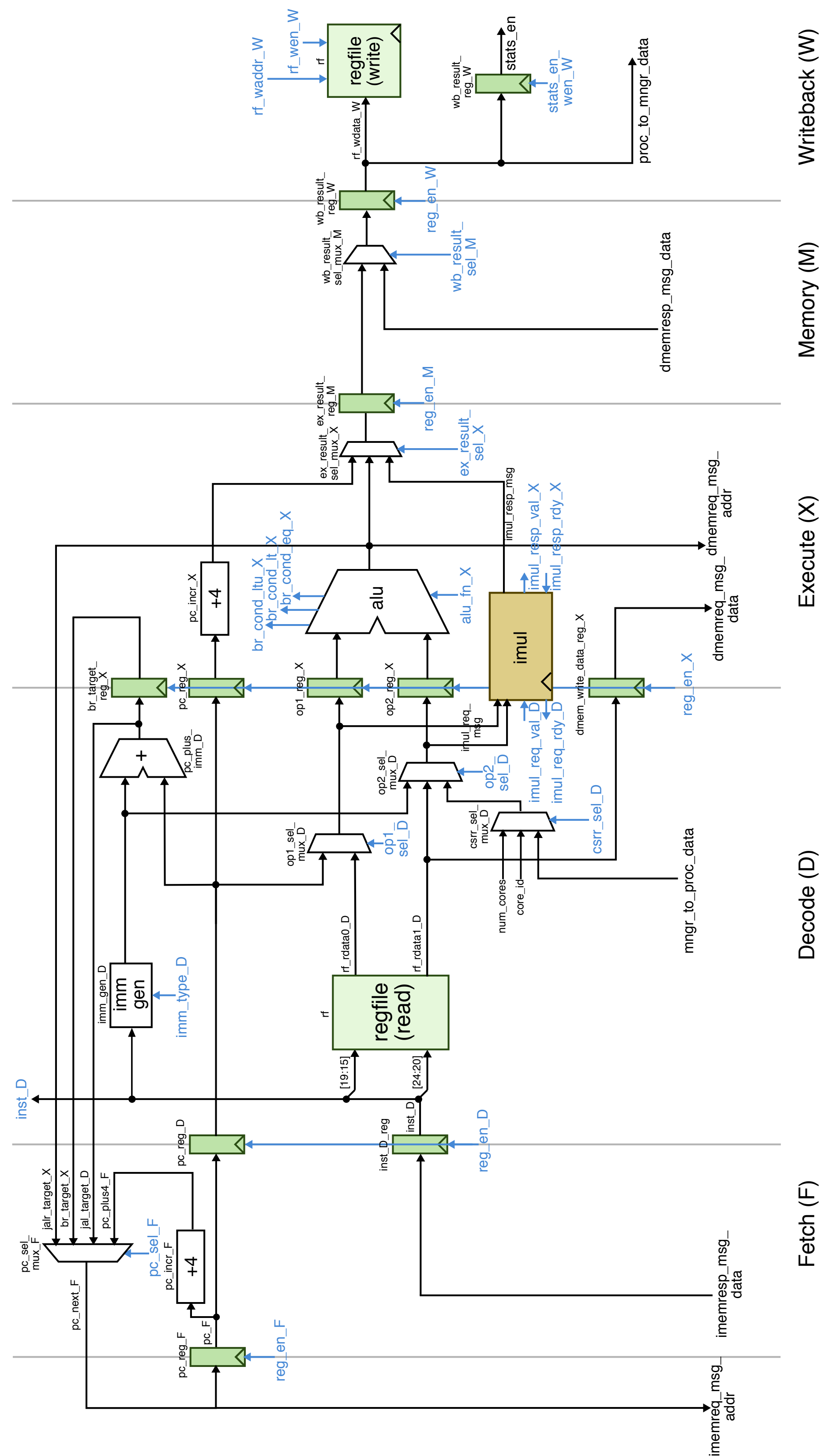

Figure 15: Baseline Design: Five-Stage Stalling Processor Datapath