ECE 206/COS 306 Lab 6: PUnC – A Microprocessor

Checkpoint 1 Due Date: 11/29/2023, 11:59 pm
Checkpoint 2 Due Date: 12/6/2023, 11:59 pm
Final Demo and Report Due Date: Dean's Date (12/15/2023), 5 pm

Checkpoint 1: 5 points
Checkpoint 2: 5 points
Demo: 60 points
Report: 30 points

We are using "Group Submission" feature of Gradescope and so only one report per group needs to be submitted. Please do not forget to write your name and your partner's name on all reports.

Introduction

This lab is a large-scale project intended to bring together all of your knowledge on RTL design, datapath construction, and modular testing. This project is significantly more involved than previous labs, so it's best to get started early. Note that there are weekly due dates for portions of the lab, although the only demonstration is for the final product.

In this lab, you will design and build the Princeton University Computer (PUnC), a 16-bit processor with a simple but versatile instruction set. This processor is Turing complete and is a full-fledged stored program computer, so you will be able to compile code and run it on your design!

As always, you will test your circuit thoroughly via simulation.

To begin the lab, download and unzip *punc.zip* from Canvas, and follow the instructions in this document. All of the source code for this lab, includ-

ing testbenches, is stored in the */punc* folder. We have also provided the Verilog codes for a Six-instruction Programmable Processor (*SIPP*), which is explained in chapter 8 of your textbook. It may be helpful to start by looking at how *SIPP* is implemented in Verilog. You may download the corresponding files (*SIPP.zip*) from Canvas.

Introduction to PUnC

PUnC is a 16-bit stored-program computer. In PUnC, all data and instructions are aligned 16-bit words, and both programs and data reside in the same memory unit. PUnC has several required hardware modules, which are described in detail in the following section.

Hardware Modules

Memory

PUnC memory addresses are 16 bits long, and each unique address points to a full 16-bit data word. For example, the address `0x0000` points to the first 16-bit entry in memory, while the address `0x0001` points to the next 16-bit entry in memory.

Ideally, PUnC's memory would contain 2^{16} 16-bit entries, but simulation with that many entries takes a long time. For efficiency, our version only implements the first 128 entries, leaving the rest disconnected. If you find that you need more memory elements, please ask a lab TA for help expanding the memory.

Memory.v contains an implementation of PUnC's memory module. It has already been wired into your datapath for use in simulation.

Here's an overview of the Memory module's ports:

Port	I/O	Width	Use
`clk`	I	1	Clock
`rst`	I	1	Synchronous Reset
`r_addr_0`	I	16	Async Read Address
`r_addr_1`	I	16	External Debug Read Address
`w_addr`	I	16	Synchronous Write Address
`w_data`	I	16	Synchronous Write Data
`w_en`	I	1	Synchronous Write Enable
`r_data_0`	O	16	Async Read Data
`r_data_1`	O	16	External Debug Read Data

This memory has asynchronous reads and synchronous writes. As soon as an address appears on one of the read address ports, the corresponding data will appear on the matching read data port. On the other hand, the data on `w_data` will be written to memory on positive clock edges and only whenever `w_en` is high. Resetting the memory will restore whatever program and data was initially present in the memory.

The second read port on the memory is used to allow external inspection of memory. It is already wired up for you, and it will be useful for debugging in simulations.

Register File

PUnC has eight general-purpose 16-bit registers, addressed from `0x0` to `0x7`. These registers are used as scratch space for arithmetic operations and are used in nearly every instruction. None of the registers have a special meaning, but register 7 (`0x7`) is automatically used to save the program counter during Jump to Subroutine calls and is used to load a new program counter during the Return instruction.

RegisterFile.v contains an implementation of PUnC's register file. It has already been wired into your datapath for use in simulation.

Here's an overview of the register file's ports:

Port	I/O	Width	Use
`clk`	I	1	Clock
`rst`	I	1	Synchronous Reset
`r_addr_0`	I	3	Async Read Address
`r_addr_1`	I	3	Async Read Address
`r_addr_2`	I	3	External Debug Read Address
`w_addr`	I	3	Synchronous Write Address
`w_data`	I	16	Synchronous Write Data
`w_en`	I	1	Synchronous Write Enable
`r_data_0`	O	16	Async Read Data
`r_data_1`	O	16	Async Read Data
`r_data_2`	O	16	External Debug Read Data

Much like the memory, this register file has asynchronous reads and synchronous writes. Resetting the register file sets all registers to zero. The first two read ports on the register file are for you to use when constructing your processor – the third is used for external inspection of the register file. It is already wired up for you, and it will be useful for debugging in simulations.

Condition Codes

PUnC also has three 1-bit condition code registers: N (Negative), Z (Zero), and P (Positive). These condition codes are set by arithmetic and load operations based on the value of the data being saved to the register file. PUnC treats that data as a two's complement number, and sets N to 1 if the number is negative, Z to 1 if the number is exactly zero, and P to 1 if the number is positive. All codes that are not set to 1 are set to zero, so only one of N, Z, and P can be 1 at any given time.

Resetting the processor should set all condition code registers to zero.

Program Counter

In PUnC, all programs are stored in memory as a series of 16-bit binary instructions. To keep track of which instruction we should currently be executing, PUnC has a 16-bit program counter (PC) register that is used to address memory.

In ordinary operation, the PC increments by 1 immediately after fetching and decoding an instruction, so the program counter actually points to the *next* instruction to be fetched while the current instruction executes. The program counter can also be modified by control flow instructions, which will be described in detail later.

Note that the program counter's value is tied to a signal (`pc_debug_data`) that goes all the way to the top of the PUnC hierarchy in the given source code. This signal enables external inspection of the program counter, so keep it in place for simulation. Resetting the processor should set the PC to zero.

Instruction Register

Finally, while not architecturally required, it will be very useful to have one 16-bit instruction register (IR) that stores the currently-executing instruction. Saving fetched instructions in the IR will make it much easier to set control signals during execution.

Other

There may be other hardware modules you need in your datapath. Feel free to include any as you see fit.

Ports

PUnC has only seven external ports. They are listed in the table below:

Port	I/O	Width	Use
`clk`	I	1	Clock
`rst`	I	1	Synchronous Reset
`mem_debug_addr`	I	16	Memory Inspection Address
`rf_debug_addr`	I	16	RegFile Inspection Address
`mem_debug_data`	O	16	Memory Inspection Data
`rf_debug_data`	O	16	RegFile Inspection Address
`pc_debug_data`	O	16	Current Program Counter

The memory and register file inspection addresses can be set to view the contents of those modules, and the PC value is always available for viewing on the `pc_debug_data` line.

The LC3 Instruction Set

PUnC uses the LC3 instruction set, an educational instruction-set architecture (ISA) that was created by Professors Yale N. Patt and Sanjay J. Patel. The assignment files contain *LC3ISA.pdf*, a full specification of the LC3 architecture.

For this project, we will only be implementing a subset of the LC3 ISA. Specifically, you will implement the instructions shown in Figure 6.1. Each instruction is 16 bits long and begins with an opcode, which is a 4-bit code that indicates what type of instruction it is. Some instructions share an opcode – specifically, there are two types of `ADD`, two types of `AND`, and two types of `JSR` (and `RET` is just a special version of the `JMP` instruction). In the event that two instructions share an opcode, some other bit is present that distinguishes the two instructions – for example, the two `ADD` instructions are differentiated by the bit in position 5.

In addition to the opcode, there are several other pieces of information in each instruction. Here's a list of each named field and what it means:

- **DR** – The 3-bit address of the destination register. The value generated by the instruction (whether through an arithmetic operation or through a load from memory) is written to `regfile[DR]`.
- **SR/SR1/SR2/BaseR** – The 3-bit address of a source register. Source registers are used as operands for arithmetic operations, targets for branching instructions, or sources for stores to memory.
- **imm5/offset6/PCoffset9/PCoffset11** – Immediate values. These fields are bit sequences that are stored as part of the instruction and

denote a constant value. They are used in arithmetic operations, such as calculating memory addresses for the PC or for loads and stores. Before they are used, they are sign-extended to 16 bits. This means that the highest-order bit in the immediate is copied to pad the immediate to 16 bits. For example, an immediate of `5'b10001` would be extended to `16'b1111111111110001`, while an immediate of `5'b00011` would become `16'b0000000000000011`.

All of the instructions listed are described in detail in *LC3ISA.pdf* with the exception of `HALT`. `HALT` should simply stop the processor – the program counter will be left pointing to the instruction after `HALT`, and the only way to get the processor to continue executing is to reset it.

Refer to *LC3ISA.pdf* (pages 524-540) as you implement your PUnC design. This document can be found on Canvas. There are a couple of key things to be aware of as you read.

First, all arithmetic operations that involve the PC are performed with the *incremented* PC. For example, when calculating the target address for the `BR` instruction, we add the sign-extended offset to the PC. This PC should be pointing to the instruction directly *after* the `BR` instruction during the computation. This should actually require almost no work to implement, as the PC should be incremented immediately after decoding but before executing an instruction.

Secondly, all instructions that are marked with a "+" sign in Figure 6.1 set the condition code registers based on whatever value is being stored into `DR`. No other instructions should modify the condition codes.

Finally, do not worry about any topics covered in *LC3ISA.pdf* that do not involve the instructions shown in Figure 6.1. We're just implementing a simple subset of LC3's functionality, so don't worry about interrupts or LC3's other available instructions.

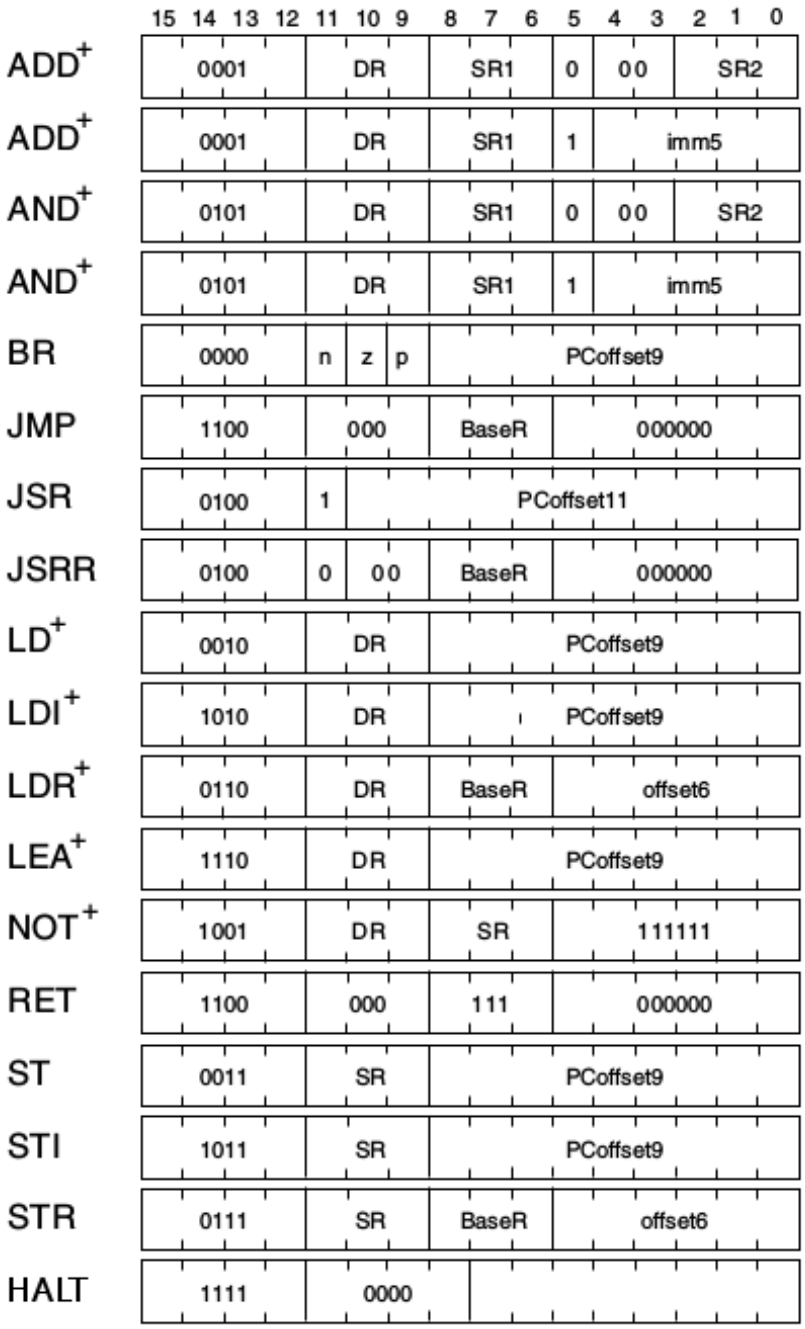

Figure 6.1: LC3 Instruction Set

Instruction Stages

PUnC works in a continuous cycle of three main stages, each of which is described in detail below. PUnC is an unpipelined processor, which means that only one instruction is in flight at any time – an instruction must fully complete its execution before the next instruction can be fetched. Here are the three main phases in the life of an instruction:

1. **Fetch** – An instruction is loaded from memory into the IR (`IR <= Mem[PC]`). This phase takes one cycle.

2. **Decode** – The instruction is now in the IR. In this stage, we simply increment the PC. On a pipelined processor, we would also prepare the control signals for executing the instruction, but since PUnC is unpipelined, we'll actually do that in the execute phase. This phase takes one cycle.

3. **Execute** – The control unit sets control signals to manipulate the datapath into executing the decoded instruction. This could involve reading from memory, executing an arithmetic operation, or even modifying the PC. This phase may take as many cycles as you need, although most instructions can be executed in a single cycle. Some instructions, such as `LDI`, may require at least two cycles.

As PUnC processes the program stored in memory, it will cycle through these three stages for every executed instruction. The Fetch-Decode-Execute paradigm will serve as an essential starting point for your PUnC design.

Week 1: Datapath Design

The first step in tackling PUnC is creating a datapath. This datapath should contain all of the hardware modules listed in the description of PUnC, including:

- A Memory Unit
- A Register File
- A Program Counter
- Condition Codes
- An Instruction Register
- Any other items you need, like ALU's, multiplexers, and extra registers.

Designing Your Datapath

Sitting down to design your datapath may seem like a daunting task, but it can be broken down into a simple series of tasks:

1. Draw your memory unit, register file, program counter, Condition Codes, and instruction register. Attach write-enable signals to all the registers, but leave the other connections blank.

2. Pick an unconnected port on one of your datapath modules (remember that the PC, condition codes, and IR all have an implicit input port that holds a new value to be loaded). Then, iterate through all of the LC3 instructions (in addition to the universal fetch and decode stages) and note the possible values that port might need to take on (for input ports) or the possible places the port's value might be used (for output ports). For example, the register file's `w_data` port can take on values from memory's `r_data_0` port, from the PC, or from the result of an arithmetic operation.

3. Once you've compiled a list of the possible inputs to each port and the places the outputs must be routed to, add hardware modules as needed to enable the requisite behavior. For example, I would add a multiplexer that selects between the three possible input values to the register file's `w_data` port.

4. If you find that you need additional local registers, add them.

5. Once you've iterated through all ports and all instructions and are sure you have the elements required to support all possible operations, your datapath is ready to go!

Drawing Your Datapath

Draw your datapath as you see fit, but please follow these guidelines:

1. Feel free to use high-level modules like adders, comparators, multi-function registers, sign-extenders, and the like. We do **not** want a gate-level description of all of these components – just make assumptions about the control signals each module needs and mark your modules so that it's clear what they do. When in doubt, keep it simple!

2. Clearly label the bit-width of every wire in your design.

3. Clearly indicate the inputs and the outputs of your datapath, and provide names for each. Signal names should be simple but descriptive enough to explain what they do. For example, a control signal that resets a counting register to zero could be labelled with `count_rst`.

Once you have a datapath design that you believe is complete, save a copy of it. If you drew it by hand, take a picture or scan it in.

Control Signal Table

As a final step, you will set up a table that has a column for every possible phase in your processor – specifically, one column for the fetch phase, one column for the decode phase, and a column for every possible execute phase. You should have an execute phase column for every possible instruction type. If an instruction's execute phase takes multiple cycles, add an execute column for each cycle.

The rows of this table should be the datapath control signals you specified in your diagram. For each phase, fill in the slots for the control signals that matter in that phase. Leaving a cell blank for a phase indicates that the corresponding control signal is zero (for write-enable signals) or that we don't care what its value is for that phase.

For example, here's part of an example output table:

CTRL SIG	Fetch	Decode	ADDR	...
`ir_w_en`	1			...
`status_w_en`			1	...
...	...	...	...	...

Feel free to use named values for multiplexer select signals instead of simple numerical values. For example, if I'm designing a control signal for a multiplexer that selects between the possible inputs to the register file's `w_data` port, then I might name the possible values of that signal `MEM_DATA`, `PC_DATA`, or `ALU_DATA` (instead of just 0, 1, or 2). These names for the multiplexer select signal paint a clearer picture of exactly what each value does to the datapath, and you can incorporate them using define macros in your code for readability.

Demonstration

Demonstration for this checkpoint is not required. However, we encourage you to visit one of the lab help sessions and show your design. If the TA notices a glaring error with your design, they will point it out to you. But otherwise, individual feedback or comments will not be provided as it gets very hard to dive into everyone's design.

Writeup and Submission

Save your control signal table in an Excel file. Make sure to enable the option "All Borders" in Excel. Then save your file as a PDF.

Also, save your datapath design as a single PDF file – if you hand-drew it, take a picture of it and save it as a PDF using a document editor of your choice.

Finally, merge the two PDF files you have created (your datapath and the control signal table) and name it *ECE206_PUnC_c1_netid.pdf*. Feel free to use any software of your choice to merge the two PDF files together. There are many free online tools available, e.g., `https://combinepdf.com/`.

Congratulations! You've finished work for week one. It's a good idea to start the next part early, as it may take longer to complete!

Week 2: Datapath and Controller Implementation

Now that you have a design hammered out for your datapath and the control signals necessary for each instruction, it's time to implement the datapath and controller for PUnC.

Datapath Implementation

Let's begin with the datapath – open up *PUnCDatapath.v*, which has been started already for you. You'll notice that the register file and memory have already been instantiated. Feel free to connect wires to their ports as needed, but don't modify the signals that are already connected to their debug ports.

Several datapath module ports have already been declared as well. Do not modify these, as they'll be needed for debugging later. However, you may (and should) add your own ports to the datapath module declaration.

Finally, note that the PC and IR are already instantiated for your convenience. If you modify their names, be sure to ensure that the PC remains connected to the `pc_debug_data` port via an `assign` statement.

Write your full datapath using synthesizable Verilog. If you wish to name the possible values for multiplexer select signals, use define statements and place them in *Defines.v* so that they can be used in the controller as well. For example, I might have this set of definitions for the multiplexer select that controls which value is written to the register file:

```
1 `define RF_W_DATA_SEL_ALU    2'b00
2 `define RF_W_DATA_SEL_PC     2'b01
3 `define RF_W_DATA_SEL_MEM    2'b10
```

Controller Implementation

Once your datapath is completed, you should implement your controller.

In this case, your finite state machine should be very simple, proceeding from Fetch to Decode to a variable number of Execute stages before cycling back to the Fetch stage. The complex part is setting the datapath control signals correctly for each phase based on the instruction being executed. Your signal table from week 1 should be helpful in this regard.

The code for the controller should be written in *PUnCControl.v*. Once again, some code has been supplied for you to help you get started. Feel free to modify it as you wish, adding ports to produce the control signals needed by the datapath and to pull in the data predicates generated by the datapath.

Connection and Testing

Once both your datapath and controller have been completed, it's time to test them out.

Open *PUnC.v*, the top-level module for PUnC. This module simply serves to tie the datapath and controller together, and the provided file already has instantiated them.

Add your ports to the datapath and controller instances, wiring them together. Do not modify the port signals provided by default in *PUnC.v* – they will be used for debugging.

Once you've connected the datapath and controller, it's time to test out your design. We've included a full-fledged suite that tests individual instructions one-by-one. Additionally, the suite includes a program that uses a variety of instructions to calculate the greatest common denominator of two values (see next week's notes for more information on this test). Our advice is to get the instructions working in the order that they appear in the testing output, as there are some dependencies between test cases. For example, almost all of the test cases depend on the ADDI instruction working, as it is used to load values into the register file. Finally, note that your HALT instruction must work for the testbench to function. The tests run a simple program to completion, and if the program doesn't complete by halting, then neither will the simulation.

Compile this testbench with the following command:

```
1 iverilog -g2005 -Wall -Wno-timescale -DSIM=1 -o PUnCTest PUnC.t.v
```

Fix any compiler errors or warnings, and then execute the suite with the following command:

```
1 vvp PUnCTest
```

Ensure that there are no failures. You'll see some warnings that look like this when you compile the datapath tests:

```
1 VCD warning: array word PUnCTest.punc.dpath.mem.mem[0] will conflict with an
      escaped identifier.
```

Just ignore those type of warnings throughout the rest of the lab – they're caused by dumping out array contents (in this case, your memory and register file contents) to a VCD waveform file.

If you need waveforms for debugging, they will be dumped out as *PUnCTest.vcd*. You can open them with this command:

```
1 gtkwave PUnCTATest.vcd
```

A Note About the Testbench

During your debugging, it may be useful to understand what the testbench is doing and how it works. A test case looks roughly like this:

```
1 `START_TEST("addi");
2 `WAIT_PC_FREEZE;
3 `ASSERT_REG_EQ(0, 16'd3);
```

The first macro, `START_TEST`, takes as an argument the name of a memory image file from the */images* folder. For example, the provided code will use */images/addi.vmh*. These memory image files are simply a list of 128 sixteen-bit values that are specified in hexadecimal. They may contain comments, and each value must be on its own line. The `START_TEST` macro loads these files into memory, with the first value in the image file corresponding to the first value in memory (`mem[0]`) and so on. Then, it resets the processor.

Next, the `WAIT_PC_FREEZE` macro waits for the PC to remain unchanged for at least 10 clock cycles. Once this happens, we know that the processor has hit a `HALT` instruction. Every memory image file included in our test cases ends with a `HALT` instruction to terminate the program.

Finally, once the program runs to completion, we examine memory and the register file to ensure that the program ran successfully. The `ASSERT_REG_EQ` and `ASSERT_MEM_EQ` macros take two arguments – an address and a value – and ensure that the corresponding memory element or register holds the given value at the provided address.

All of our memory images are commented with assembly code, which is the low-level language that maps one-to-one to binary machine instructions. The *LC3ISA.pdf* document provides assembly-code examples for each instruction – read through these to understand what each assembly instruction means. Then, you can use the assembly comments to figure out what each program is doing if you find yourself mysteriously failing a test case.

Some helpful Advice

- Verilog inequalities $<, >$ are unsigned, so be careful when setting the condition codes for negative numbers.
- There are many ways to perform sign extension, but the replication and concatenation operators may be helpful. See page 7 of the Verilog tutorial for concatenation and this webpage (`https://www.nandland.com/verilog/examples/example-replication-operator.html`) for replication syntax.

- Keep in mind that the execution stage does not have to be finished in one state - you can use as many states as you need for the execution stage. For PUNC, you may actually need to use more than one state for a single execution stage.
- The negative numbers are stored using two's complement format in memory but you must make sure to sign extend them if they are immediate values. Also for the n,z,p flags, you need to either manually test bits according to two's complement or by using the $\$signed()$ verilog function.
- Do not use any of the ports marked "External Debug" in the tables on pages 3 and 4 of the lab handout.
- In the six-instruction example, the IR is sent to the controller. However, that is not necessary for PUnC. You can parse the IR entirely in the datapath (extracting bits for register addresses, offsets, etc.). If you prefer to send the IR to the controller, that's fine too, but it won't be necessary. That being said, the controller will need some bits from the instruction, namely the opcode and a few other bits that distinguish between two instructions that have the same opcode.
- "Bitwise Logical AND": This should be the bitwise AND (&), not the logical AND (&&).
- When debugging, you might get infinite loops sometimes where vvp never terminates. If this happens, you can press ctrl-c, type the word "finish", and press enter. This will save the vcd file so you can run gtkwave and see what went wrong.
- A frequent issue occurs when you put MUXes in an always block with if-statements. Remembering that always blocks run sequentially, if a MUX which depends on the output of another MUX is ordered improperly you can run into execution issues that are pesky to debug. Therefore, it is recommended to use continuous assignment and ternary operators for MUXes as seen in the following pseudo-code.

```
1 // Defines.v
2 `define MUX_SELECT_ZERO  2'b00
3 `define MUX_SELECT_ONE   2'b01
4 `define MUX_SELECT_TWO   2'b10
5 `define MUX_SELECT_THREE 2'b11
6
7 // PUnCDatapath.v
```

```
8  assign output = (input == `MUX_SELECT_THREE) ? out3 :
9          (input == `MUX_SELECT_TWO)   ? out2 :
10         (input == `MUX_SELECT_TWO)   ? out2 :
11       /* (input == `MUX_SELECT_ZERO) */ out0;
```

Demonstration

Demonstration for this checkpoint is not required. However, we encourage you to visit one of the lab help sessions and show your design. If the TA notices a glaring error with your design, they will point it out to you. But otherwise, individual feedback or comments will not be provided as it gets very hard to dive into everyone's design.

Write-up and Submission

For this week's writeup, you'll simply be submitting an updated version of your datapath and signal table. Similar to the procedure for Week 1, merge your new PDF files for your signal table and the datapath and name it *ECE206_PUnC_c2_netid.pdf*, with *netid* replaced with your Princeton NetID. Finally, submit this on Gradescope.

Congratulations! You've finished the work for week 2. It's a good idea to start the next part early.

Week 3: Creating Your Own Program

Now that your code passes our basic functional tests, you will create your own test for PUnC. For this part, you will create your own LC3 program that uses at least ten of the provided instructions to perform some sort of interesting computation.

Our Example Program

For example, our provided program (saved in */images/gcd.vmh*) computes the greatest common denominator of two integers.

In C, the code for this program would look something like this:

```c
int main(void) {
  int a = 15;
  int b = 13;
  int gcd = 0;

  while (a != b) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  gcd = a;
}
```

Once the program runs to completion, the GCD of A and B will be stored in both `a` and `b`, as well as in `gcd`.

We can translate this to LC3 assembly fairly easily, which produces this program:

```
// INITIAL BLOCK:
/*0:*/  LD R0, #23          // Load A into R0 from mem[24]
/*1:*/  LD R1, #23          // Load B into R1 from mem[25]
/*2:*/  NOT R2, R1          // Load -B into R2
/*3:*/  ADD R2, R2, #1
/*4:*/  NOT R3, R0          // Load -A into R3
/*5:*/  ADD R3, R3, #1
/*6:*/  ADD R4, R0, R2      // Load A - B into R4
/*7:*/  BRz #13             // If A - B == 0, goto A_EQUAL_TO_B
/*8:*/  BRn #6              // IF A - B < 0,  goto A_LESS_THAN_B

// A_GREATER_THAN_B:
/*9: */ ADD R0, R0, R2      // Load A - B into R0 (A = A - B)
```

```
/*10:*/ NOT R3, R0          // Load -A into R3
/*11:*/ ADD R3, R3, #1
/*12:*/ ADD R4, R0, R2      // Load A - B into R4
/*13:*/ BRp #-5             // If A - B > 0, goto A_GREATER_THAN_B
/*14:*/ BRz #6              // If A - B == 0, goto A_EQUAL_TO_B

// A_LESS_THAN_B:
/*15:*/ ADD R1, R1, R3      // Store B - A into R1 (B = B - A)
/*16:*/ NOT R2, R1          // Store -B into R2
/*17:*/ ADD R2, R2, #1
/*18:*/ ADD R5, R1, R3      // Store B - A into R5
/*19:*/ BRn #-11            // If B - A < 0, goto A_GREATER_THAN_B
/*20:*/ BRp #-6             // If B - A > 0, goto A_LESS_THAN_B

// A_EQUAL_TO_B:
/*21:*/ ST R0, #1           // Store A into mem[23] (GCD)
/*22:*/ HALT                // HALT -- GCD of A and B will be in R0 and R1

// DATA:
/*23:*/ 0000                // GCD
/*24:*/ 000C                // A
/*25:*/ 000F                // B
```

Note that the numbers at the beginning of each line are the memory addresses for each instruction and are just there for your convenience. They are **not** part of the assembly code, which is why they're commented out.

The initial numbers that we wish to take the GCD of are stored in memory at addresses 24 and 25 respectively. In this case, `A` is 12 and `B` is 15, and their GCD is 3. This program generally keeps the current value of `A` in `R0`, `B` in `R1`, `-A` in `R2`, and `-B` in `R3`. Once the `HALT` instruction has been hit, the GCD of `A` and `B` is stored in `R0`, in `R1`, and in memory at address 23.

Note that data can be written in assembly simply as a 16-bit hexadecimal number instead of an instruction. Hard-coding initial values in memory is a useful way to make a program easy to edit and test for different datasets.

This assembly code produces the following memory image file:

```
2017   // LD R0, #23
2217   // LD R1, #23
947F   // NOT R2, R1
14A1   // ADD R2, R2, #1
963F   // NOT R3, R0
16E1   // ADD R3, R3, #1
1802   // ADD R4, R0, R2
040D   // BRz #13
0806   // BRn #6
1002   // ADD R0, R0, R2
963F   // NOT R3, R0
```

```
16E1   // ADD R3, R3, #1
1802   // ADD R4, R0, R2
03FB   // BRp #-5
0406   // BRz #6
1243   // ADD R1, R1, R3
947F   // NOT R2, R1
14A1   // ADD R2, R2, #1
1A43   // ADD R5, R1, R3
09F5   // BRn #-11
03FA   // BRp #-6
3001   // ST R0, #1
F000   // HALT
0000   // 0000
000C   // 000C
000F   // 000F
0000   // Zeros from here on
//.....
```

This file is padded with zeros until memory address 127 is reached. The memory image for the GCD program has been saved as *images/gcd.vmh*, and it was included in the TA testbench from last week's section.

Using the Assembler

Write your own assembly program using ECE 206's custom LC3 assembler, which can be found on Canvas. In the left-hand window, simply write your LC3 assembly code (without any comments). Once you're done, click on the "To Mem Image" button to produce a commented memory image file. Type a filename into the "Output Filename" box and click "Download Result" to download the memory image file. For consistency, save the file with a *.vmh* extension.

Once you're sure your program works as you wish, download your assembly code using the "Download ASM" button. For consistency, save your file with an *.asm* extension.

Remember that **your program must use ten unique instructions** from the provided set, and it must produce some sort of meaningful result. It doesn't need to be anything fancy, but don't just place random instructions!

Running Your Program

Move your *.vmh* file into */images*, and then open up `PUnC.t.v`. Find the end of the provided tests (the last one is our GCD program), and add your own using the provided macros. These macros are described in detail in last week's

notes on the TA testbench, but specifically, you can run your test by inserting the following sequence of commands:

```
`START_TEST("mytest");
`WAIT_PC_FREEZE;
```

This code will load your program into memory and then run it to completion. Ensure that your program produces the correct results by checking the end state of memory and the register file. Use the `ASSERT_REG_EQ` and `ASSERT_MEM_EQ` macros at your convenience to do this.

Once your program works correctly, save your assembly code – you'll be required to turn in a commented copy of your program's assembly code as part of your final write-up.

PUnC Online Debugger

As with the other labs we have built a signal viewer website to check that your PUnC implementation is correct: `https://verilog-puncviewer.vercel.app/`.

The viewer allows you to see the value of all register and memory addresses at once. The viewer should automatically set the signal values as needed after your final VCD file is uploaded, but if it cannot find the correct signal you will have to select it with the searchable drop down menus as before.

Demonstration

For your demonstration this week, visit one of the lab sessions to demo your work to a graduate TA.

Write-Up and Submission

For your final write-up, create a PDF file that includes your revised datapath drawing, an annotated, commented copy of your program's assembly code, and an explanation of what your assembly program does. Additionally, please leave a bit of feedback regarding this lab. How much time did it take, how difficult was it, did you get stuck anywhere? There are no wrong answers here – we'll be using the feedback to adjust this lab for the future.

In addition, save your revised Excel spreadsheet that contains your control signal table and prepare it for final submission. Make sure you have enabled the option "Border All" in Excel. Then save your Excel spreadsheet as a PDF file.

Finally, similar to the procedure for Week 1 and Week 2, merge your PDF files togother and name it *ECE206_PUnC_netid.pdf*, with *netid* replaced with your Princeton NetID. Finally, submit this on Gradescope.