EENG 284 – Digital Design

Lab 10 – Stored Program Computer

Part 3 – Control, Datapath, RAM

# Objective

The objective of this lab is to fully automate the fetching and executing of instructions for the datapath + memory you designed last week. This will enable you to store sequences of R and I type instructions in the memory and have them executed in order. In addition, we will be introducing a pair of branch instructions that will allow you to control the flow of execution.

**Today’s Computer**

In the last lab you added a random access memory to your datapath so that you could store R-type and I-type instructions in the memory and have the datapath execute them in order. This is shown in the left-side of Figure 1. One of the major short comings of this lab was that we had to manually fetch and execute these instructions using using a control word input that was set through the testbench signals. This is not a practical method of operation for large scale computers.

So today, you are going to design a control unit to assert the control word inputs to the datapath so that the datapath can fetch and execute instructions. This together with the RAM forms the computer shown in Figure 1. In addition, you are going to add a pair of new instructions to your computer that will enable it to perform conditional branch instructions. This gets us one step closer to having a general-purpose computer, a goal that we will fulfill with next week’s lab.

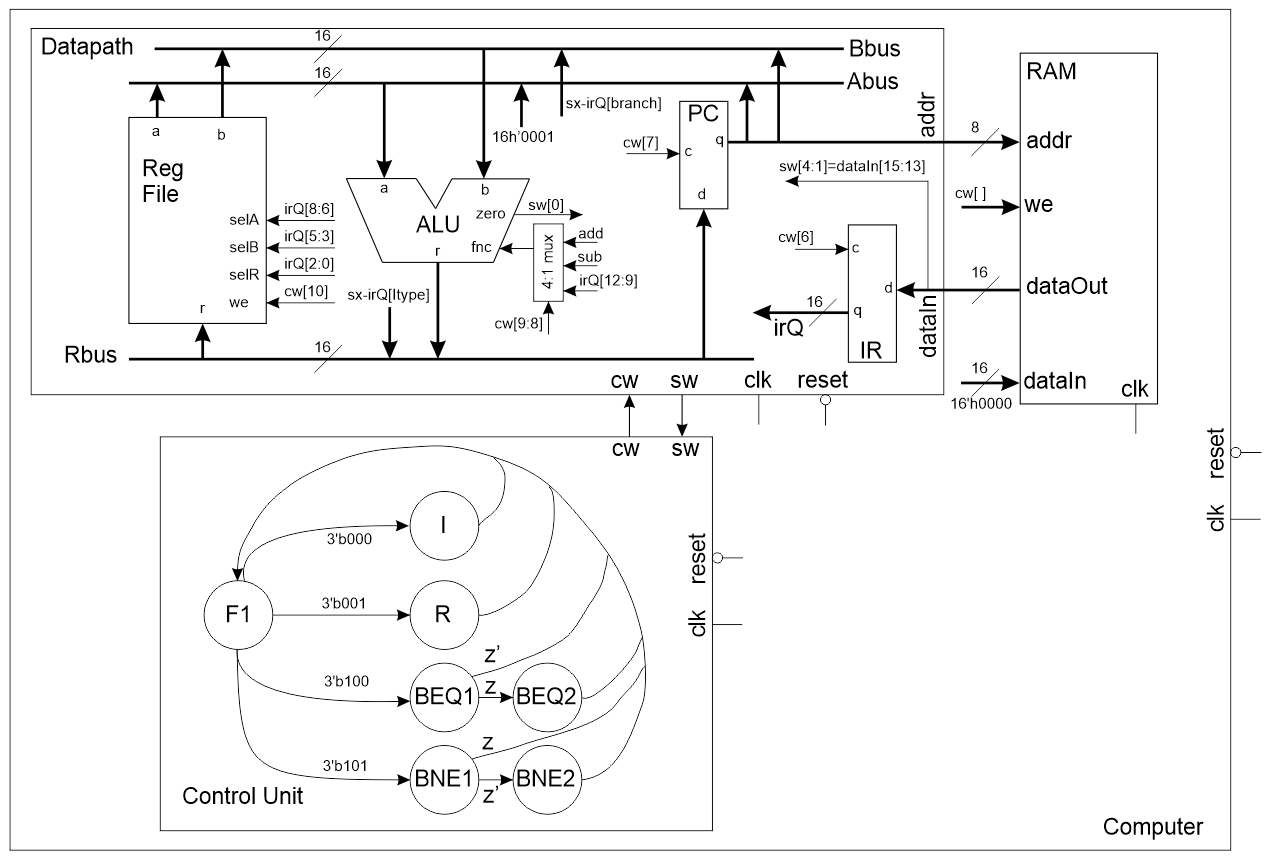


Figure 1: Our computer consists of three major subsystems, a datapath, a control unit and a random-access memory.

Let’s start our work on this lab by looking at the two new instructions that you are going to incorporate into your computer, branch if equal and branch if not equal.

**The Instruction Set**

From the previous two labs you should be familiar with the 16-bit I or R-type instruction shown in the top two rows of Table 1. The last two rows of this table introduce the binary format of the two new instructions, branch if equal (BEQ) and branch if not equal (BNE).

Table 1: The instruction format for the I-type, R-type and branch instructions.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| I | 0 | 0 | 0 | x | x | Immediate | | | | | | | | Dest | | |
| R | 0 | 0 | 1 | fnc | | | | selA | | | selB | | | selR | | |
| BEQ | 1 | 0 | 0 | Offset MS | | | | sel A | | | sel B | | | Offset – LS | | |
| BNE | 1 | 0 | 1 | Offset – MS | | | | sel A | | | sel B | | | Offset – LS | | |

**BEQ and BNE instructions**

A conditional operation is a program statement that allows you to perform some operation conditional on some criteria. For example, you might check if x does not equal 0 before performing y/x. You might describe this as:

if (x != 0) z = y/x;

Anytime that you perform a conditional operation in a program, you are using a branch instruction. The branch if equal (BEQ) instruction:

* Adds the 2’s complement offset to the PC if the values of the two registers is equal.
* Increments the PC if the if the values of the two registers is not equal.

The process of adding an offset to the program counter is referred to as jump because the computer is jumping over one or more instructions to execute an instruction that is not necessarily immediately adjacent to the BEQ instruction. Note, that the offset is 2’s complement meaning that the value added to the PC can be positive or negative. Thus, the BEQ instruction can jump forward or backwards in a program. The bits of the BEQ instruction deserve some explanation before we mote on to explore how the BEQ instruction works in a program.

* The register values being compared are in the register at index specified by the 3-bit binary code contained at bits [8:6] and [5:3] of the BEQ instruction.
* The 2’s complement offset added to the PC is formed by appending bits [12:9] and [2:0] of the instruction. This slight inconvenience significantly simplifies the design of the datapath.

Now, let’s look at how you can use the BEQ instruction to program the equivalent of a for-loop using the code snippet in Listing 1.

Listing 1: A for loop code in C (left) and the corresponding assembly code (right).

C code line assembly code

R0 = 0; 1 LDI #0x00, R0

2 LDI #0x00, R1

3 LDI #0x01, R6

4 LDI #0x08, R7

for (R1=0; R1<8; R1++) { 5 loop: BEQ R1, R7, +3

R0 = R0 + R1; 6 ADD R0, R1, R0

7 ADD R1, R6, R1

} 8 BEQ R6, R6, -4

9 next:

On the left side of this listing is a for-loop, written in C, that uses variable R1 as a loop counting variable that starts at 0 and increments until it is greater than or equal to 8. The body of the for-loop adds R1 to a running total maintained to R0.

The assembly code corresponding to the for-loop is shown in the right column of Listing 1. Two notes are in order. First, I’ve added line numbers to make the discussion easier to follow. Second, the assembly code shown in Listing 1 contains two labels, “loop” and “next”. You should place labels on lines of code that are the target of a branch instruction. The target of a branch instruction is the line of code that the branch jumps to when the branch is taken. We will examine the (seemingly incorrect) offset values of the branch instructions on line 5 and 8 after we look at all the other code.

* Line 1 Initialize the variable R0 to 0,
* Line 2 Initialize the loop counting variable R1 to 0,
* Line 3 Store the value 1 in R6. R6 will increment the loop counting variable on line 7,
* Line 4 Store the value 8 in R7. R7 will be compared to the loop counting variable in line 5,
* Line 5 From the C code, the body of the for-loop is executed as long as R1 is less than 8. Since R1 is being incremented up from 0, we could restate this condition as “Stop executing the body of the for-loop when R1 equals to 8”. The body of the for-loop is contained on lines 6,7,8 of the assembly program. Hence our program will jump over these lines of code when R1 is equal to 8,
* Line 6 This is the body of the for-loop, putting R0 + R1 into R0,
* Line 7 Incrementing the loop counting variable R1,
* Line 8 Equivalent to “GOTO loop”. Since R6 is always equal to itself, this branch is always taken. And yes, I could have used any register in place of R6,
* Line 9 The instruction for this line of code is not specified because it is whatever operation is performed after the for-loop is done executing.

Let’s talk about the offsets for the BEQ instructions in Listing 1, we’ll focus on the BEQ instruction on line 8. For reasons that will be made clear later, the branch offset is the difference between the target of the branch instruction and branch instruction plus 1. In other words:

Offset = target\_address – (branch\_instruction\_address + 1)

In order to make sense of this equation, let’s calculate the offset of the BEQ instruction on line 8.

* branch\_instruction\_address = 8
* target\_address = 5
* offset = 5 – (8+1) = 5 – 9 = -4

You should be able to verify that the offset of the BEQ instruction on line 5 is 9 – (5+1) = +3.

The branch if not equal (BNE) instruction:

* Adds the 2’s complement offset to the PC if the values of the two registers are not equal.
* Increments the PC if the if the values of the two registers is equal.

In other words, the BNE instruction jumps to the target if the value of the two registers are not the same. With the exception of when the branch is taken, the BNE operates the same as the BEQ instruction.

**Today’s Program**

You are going to the program given in shaded region of the left column of to test the operation of the datapath, control and computer for today’s lab. I choose this program because it is short, includes every instruction type, and tests BEQ and BNE when they are taken and not taken. Complete by filling in the operation code (Op Code) column in hexadecimal for each instruction. For convienience, I’ve include the instruction format for each instruction at the top of

Table 2: The program to test the datapath, control and computer for today’s lab.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 15 | 14 | 13 | 12 | | 11 | | 10 | 9 | | 8 | 7 | 6 | | 5 | 4 | 3 | 2 | 1 | 0 | Op Code |
| I | 0 | 0 | 0 | x | | x | | Immediate | | | | | | | | | | Dest | | |  | |
| R | 0 | 0 | 1 | fnc | | | | | | selA | | | | selB | | | | selR | | |  | |
| BEQ | 1 | 0 | 0 | Offset – MS | | | | | | sel A | | | | sel B | | | | Offset – LS | | |  | |
| BNE | 1 | 0 | 1 | Offset – MS | | | | | | sel A | | | | sel B | | | | Offset – LS | | |  | |
| LDI #0x01, R0 | 0 | 0 | 0 |  |  | |  | | | | | | | | | | |  | | |  | |
| BNE R5, R5, -2 | 1 | 0 | 1 | 1111 | | | | | | 101 | | | | 101 | | | | 110 | | |  | |
| ADD R0, R1, R1 |  |  |  |  | | | | | |  | | | |  | | | |  | | | 0x2009 | |
| BEQ R0, R1, -2 |  |  |  |  | | | | | |  | | | |  | | | |  | | |  | |
| BNE R0, R1, -3 |  |  |  |  | | | | | |  | | | |  | | | |  | | |  | |

In order to better understand the behavior of the instructions and the output from the test program, execute the test program by filling in. I’ve taken the liberty of listing the instructions out in the order they are executed. Since there are branch instructions, many of the instructions in the program will execute multiple times. In the right most column, write “Not taken” or “Taken” to indicate if the branch on that line is taken or not taken.

Table 3: Determine the values of the registers in the unshaded cells. Indicate if the branch is taken or not taken in the rightmost column. Do not fill in the shaded cells.

|  |  |  |  |
| --- | --- | --- | --- |
|  | R0 | R1 | Branch taken or not taken? |
| LDI #0x01, R0 |  |  |  |
| BNE R5, R5, -2 |  |  |  |
| ADD R0, R1, R1 |  |  |  |
| BEQ R0, R1, -2 |  |  |  |
| ADD R0, R1, R1 |  |  |  |
| BEQ R0, R1, -2 |  |  |  |
| BNE R0, R1, -3 |  |  | Taken |
| ADD R0, R1, R1 |  |  |  |
| BEQ R0, R1, -2 |  |  |  |
| BNE R0, R1, -3 |  |  |  |
| ADD R0, R1, R1 |  | 4 |  |
| BEQ R0, R1, -2 |  |  |  |
| BNE R0, R1, -3 |  |  |  |
| ADD R0, R1, R1 |  |  |  |

Before you leave this section, add the code you generated in Table 2 to the ramLab10.lst file posted on Canvas and turn it in as the solution to this problem. After you finish this, let’s now turn our attention to building the hardware to fetch and execute this program. We will start by examining how the control unit that will be responsible for fetching and executing the instructions store in the RAM.

**The Control Unit**

The state diagram for our control unit is shown in Figure 2. The status word input consists of 4-bit; the upper 3 are the instruction opcode (upper 3 bits of the instruction) and the least significant bit is the zero output from the ALU. The instruction opcode bits, sw[3:1] are listed by their 3-bit codes out of the F1 state. The zero bit, sw[0], goes by the alias “z” inside the state diagram. Z’ means that the transition is taken when sw[0] = 0.

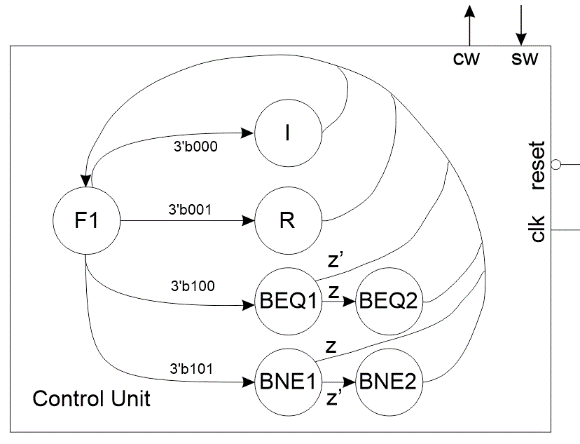


Figure 2: The FSM used to fetch and execute instructions.

The behavior of the control unit will become more clear as you dig into the operation of the datapath, but a short overview will prepare you understand the changes being made to the datapath.

F1: Allows the output of the RAM (an instruction) to be latched into the instruction register. Simultaneously, increments the PC by sending the PC and the value 16’h0001 to the ALU and having the ALU perform an add. Stores the result on the Rbus back into the PC.

R: Sends RS1 and RS2 to the ALU over the A and B bus. Use the IR bits to perform the correct ALU operation. Stores the value on the R bus to the RD file register.

I Store the sign extended immediate to the RD file register.

BEQ1, BNE1: Subtract RS1 and RS2 setting the zero bit if the two are equal.

BEQ1, BEQ2: Using the ALU, adds PC and sign extended IR. Stores the result on the Rbus back into the PC.

We start our construction of the Verilog code for the control unit with its module declaration given in Listing 2. For the time being, you do not need to be concerned with the control word, cw, output, – we’ll work on that in the datapath section. Just make sure that it’s 11-bits wide.

Listing 2: The module description for the traffic light FSM shown in Figure 2.

module controlUnit(clk, reset, cw, sw);

input wire clk, reset;

output reg [10:0] cw;

input wire [3:0] sw;

The development of the code for the control unit will follow the steps outlined in the traffic light controller handout given in class. I would advise that you use that code as the starting point for your Verilog code as we will touch on all the sections here.

**State definitions**

Use a dense coding to assign each state in Figure 2 a unique binary code (the choice of which code is assigned to each state is immaterial) using the localparam statement. Append all your state names with “\_STATE” to help you tell what sort of thing you are looking at.

localparam FETCH\_STATE = 3'b000;

While you are defining things, I found my code much more readiable when I defined the 3-bot opcodes as well as the control words using the localparam statement as shown below. Likewise, I made up a set of control word alias shown below. Even though you do not know what these values should be, give each state a different, arbitrary, code. You’ll fix these when you design the datapath.

localparam RTYPE\_OPCODE = 3'b001;

localparam FETCH\_CW = 11'b00000000001;

**Define the reset state**

In all our testbenches, we will briefly hold the reset line low when our simulation first starts. This reset should put your control unit in the **F1** state.

**Next state logic**

The next state logic captures the logic describing the outgoing arcs from each state. Each state will have a case statement where you describe the next state in terms of the input associated with each arc. When you use the if/then structure you **must always** include a final else statement.

FETCH\_STATE:

begin

case(instrOpCode)

RTYPE\_OPCODE: nextstate = RTYPE\_STATE;

<other stuff goes here>

default: nextstate = FETCH\_STATE;

endcase

end

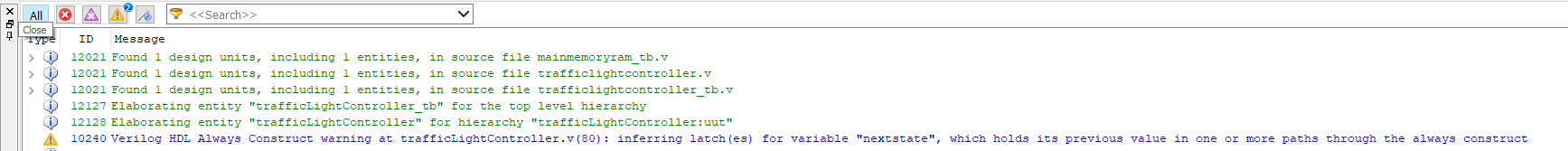
**Output logic**

This section of logic is pretty straight forward, have a row for each state in the always/case statement. The case for each state assigns the control word output the value of the control word associated with that state.

FETCH\_STATE: cw = FETCH\_CW;

**Warning**

If you get the following warning, you may have missed a then statement in your next state logic or a default in your case statement.



**Testbench**

As our systems get more complex, it is imperative that you run and carefully check your testbench results. Use the control\_tbWaveSetup.do file provided on Canvas. Deviations from the behavior shown in Figure 3 (I intentionally shaded out the control word because it’s not important now) will result in the computer not operating correctly.

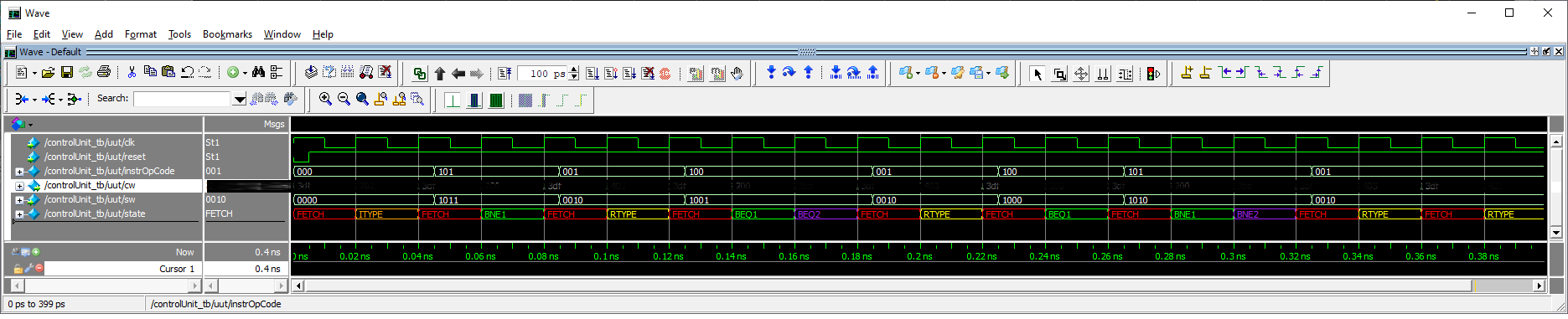


Figure 3: Testbench behavior for the control unit.

**The Datapath**

The datapath that you will create for this week’s labs, shown in, is a variation of the one used in last week’s lab. Before going much further, note that there are two labels in that deserve some explanation; sx-irQ[branch] and sx-irQ[Itype].

* sx-irQ[branch] = 16-bit sign-extend value of IR bits [12:9] appended with IR bits [2:0]
* sx-irQ[Itype] = 16-bit sign-extend value of IR bits [10:3]

There are two significant changes incorporated into the datapath shown in Figure 2 to accommodate the introduction of branch instructions; the PC and the ALU function mux. The PC will change its value in one of two ways; either it will be incremented to fetch the next instruction in memory or the PC will have the 2’s complement offset in the instruction register added to it.

Since the PC will be doing more than just counting (as it did in last week’s lab), you will change it to a register in this week’s lab. The ALU will perform the addition of the PC with either 16’h0001 or sx-irQ[branch] (see the Abus and Rbus in), putting the result on the Rbus. The PC will then load this value from the Rbus. Each of the three busses shown in Figure 2 is the output of a 16-bit 4:1 mux. These muxes are not explicitly shown, but can be inferred by the multiplicity of arrows going to each of the busses. The mux for each bus will take, as input, the signals on the arrows pointing to the horizontal bus line and output a 16-bit signal with that busses name.

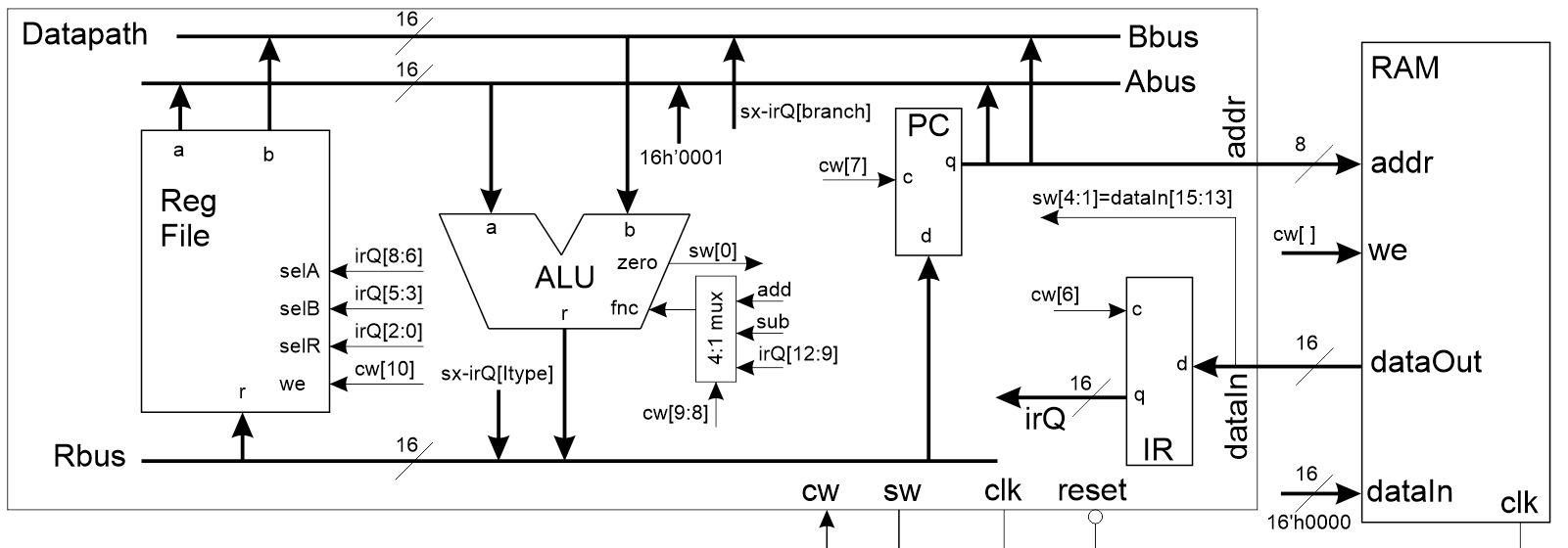


Figure 4: The datapath for this week's lab is able to execute I-type, R-type and branch instructions.

The second major change is the introduction of a mux in front of the function input of the ALU. This mux selects one of three function, add (to add 1 to the PC), subtract (to check the equality of two of the file registers) or irQ[12:9] (to perform an R-type instruction). These and other modifications that are outlined in the list below:

* Change the PC into a register.
* Add a 4-bit 4:1 mux in front of the ALU function input
* Add 16-bit 4:1 mux for the Abus
  + Select register file or 16’h0001 or PC
* Add 16-bit 4:1 mux for the Bbus
  + Select register file or sx-irQ[branch] or PC
* Add 16-bit 4:1 mux for the R bus
  + Select ALU or sx-irQ[Itype] or something TBD
* Create a sign extended branch offset

To better understand these changes, how to implement them, and how they are used, let’s examine the control word shown in Table 4 for the datapath shown in Figure 2.

The top row of the control word table is the bit index of the signal. Devices that require 1-bit of control have a single index and are control with that bit of the control word. For example, the control input of the instruction register is cw[6]. Below the IR in this column are the words “Hold” and “Load”. Hold refers to the register holding its value and happens when the control input is equal to the value in the left-most column labeled “Code”. In this case, the hold happens when the control inputs equal 0; not 00 because this is a 1-bit control input. This makes sense, because registers hold their value when the control input is 0 and load when the control input is 1.

Devices with two bits of control are given two indices and their control word is formed from the sub-vector of these two indices. For example, the A bus mux select input is controlled with cw[5:4]. Looking below the word “ABUS” in this column are the words “RFa”, “0x01”, and “PC”. Each of these labels refers to the value that is placed on the output of the mux when the control signal equals the value in the left-most column labeled “Code”. For example, the ABUS will get the value of register file port A (RFa) when the control input equals 00; not 0 because this is a 2-bit control input. Some of the input combinations are not used, you can leave these parameter positions empty in your Verilog or put in some filler value – the choice is not important.

Control word indices 9:8 deserve special mention. These two bits control the 4-bit 4:1 mux in front of the ALU function input. The terms “Sub” and “Add” in this column refers to the 4-bit control codes which cause the ALU to subtract (4’b0001) and add (4’b0000).

Table 4: The control word for the datapath shown in Figure 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 9,8 | 7 | 6 | 5,4 | 3,2 | 1,0 |
| Code | RF WE | ALU | PC | IR | ABUS | BBUS | RBUS |
| 00 or 0 | Hold | IR[12:9] | Hold | Hold | RFa | RFb | MBR |
| 01 |  |  |  |  | 16’h0001 |  |  |
| 10 |  | Sub |  |  |  | IR[BR] | IR[IMM] |
| 11 or 1 | Write | Add | Load | Load | PC | PC | ALU |

To be complete, the following list outlines the function associated with each control bit.

Bit 10 register file write enable

Bit 9,8 Mux select for 4:1 mux that selects the ALU function source

Bit 7 Control input for the PC register

Bit 6 Control input for the IR register

Bit 5,4 Mux select for the 4:1 mux that selects the A bus source

Bit 3,2 Mux select for the 4:1 mux that selects the B bus source

Bit 51,0 Mux select for the 4:1 mux that selects the R bus source

One significant change in this datapath is that the control signal for each basic building block is brought out to the control word. In previous labs, we had fixed some of these control signals to the instruction register. We can no longer do this because the branch instructions require added functionality that requires the busses and data inputs to have several different sources.

Fetch1: Enable the IR to latch the RAM output, Increment the PC

ALU: Rs1 operation Rs2 stored in Rd

Imm Store SX immediate into register file

Branch: Subtract two registers so that the ALU zero bit is set/clear

Branch: Add PC + IR offset and store in PC

Table 5: Complete the control word for each state.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 9,8 | 7 | 6 | 5,4 | 3,2 | 1,0 |
| Code | RF WE | ALU | PC | IR | ABUS | BBUS | RBUS |
| 00 or 0 | Hold | IR[12:9] | Hold | Hold | RFa | RFb | MBR |
| 01 |  |  |  |  | 16’h0001 |  |  |
| 10 |  | Sub |  |  |  | IR[BR] | IR[IMM] |
| 11 or 1 | Write | Add | Load | Load | PC | PC | ALU |
|  |  |  |  |  |  |  |  |
| F1 |  |  |  |  | 01 |  |  |
| R |  |  |  |  |  | 00 |  |
| I | 1 |  |  |  |  |  |  |
| BEQ1  BNE1 |  | 10 |  |  |  |  |  |
| BEQ2  BNE2 |  |  |  |  | 11 |  |  |

You need to incorporate these control word vectors into the control unit Verilog file (that you created) and the datapath\_tb.v file (I provided to you on Canvas).

**Testbench**

As our systems get more complex, it is imperative that you run and carefully check your testbench results. Use the datapath\_tbWaveSetup.do file provided on Canvas. You will need to alter the names of the registers in your register file as I probably implemented my register file differently that you. Please check your results against mine (except for the control word which I shaded out) and correct errors. Deviations from the behavior shown in Figure 5 will result in the computer not operating correctly. Make sure that you have changed the ramLab10.lst file from an earlier section.

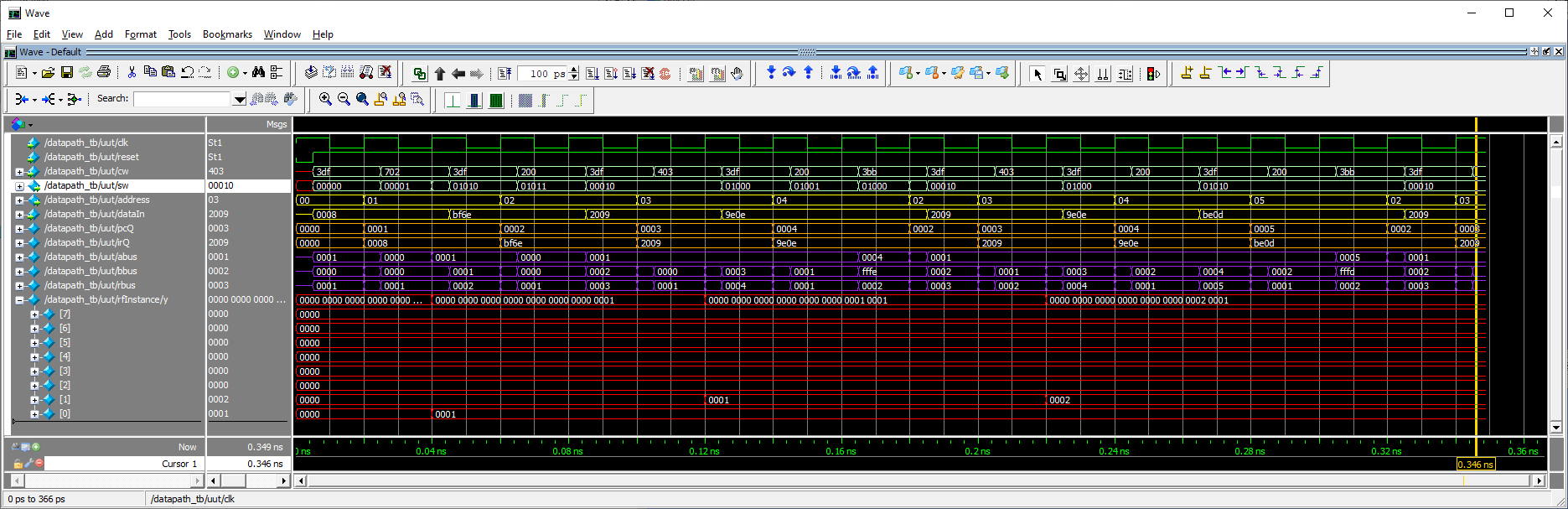
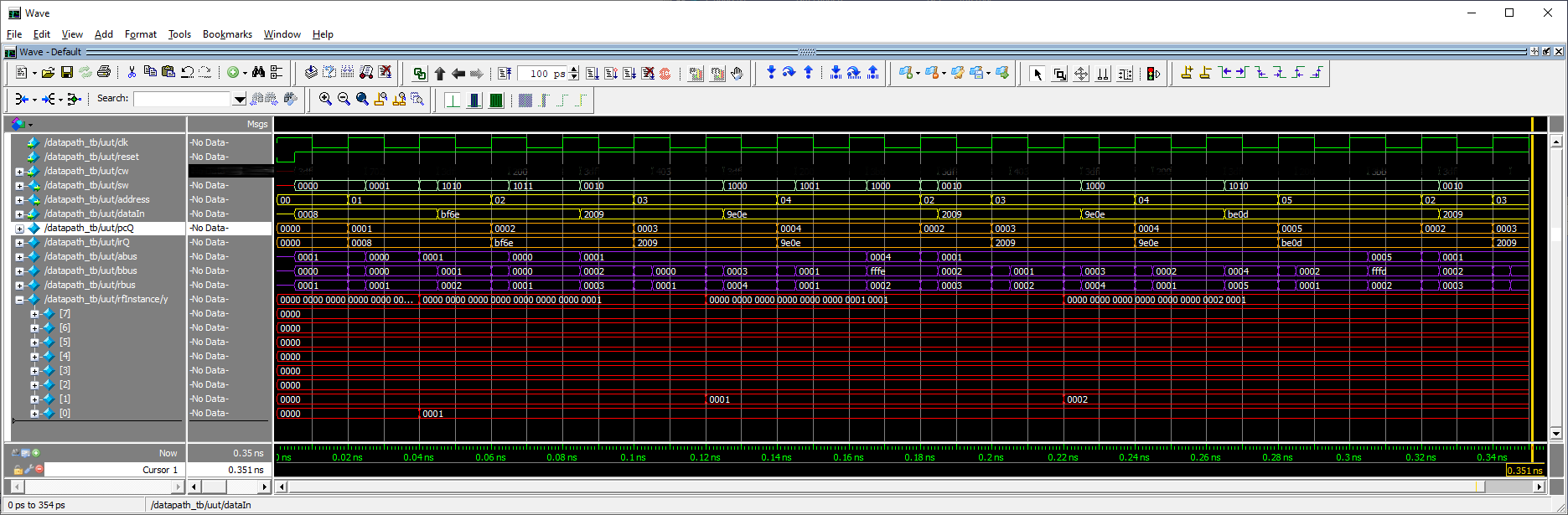


Figure 5: Testbench for the datapath.

****

**The Computer**

The computer module contains three lines of Verilog code, one each to instantiate the datapath, control unit and RAM. Listing 3 shows the module declaration and internal signs for the computer. It’s remarkably simple, just send in a clock and reset signal and the datapath and control will execute the instructions stored in the RAM.

Listing 3: The module declaration of the computer only has a clock and reset signal.

module computer(clk, reset);

input wire clk, reset;

wire [10:0] cw;

wire [15:0] data;

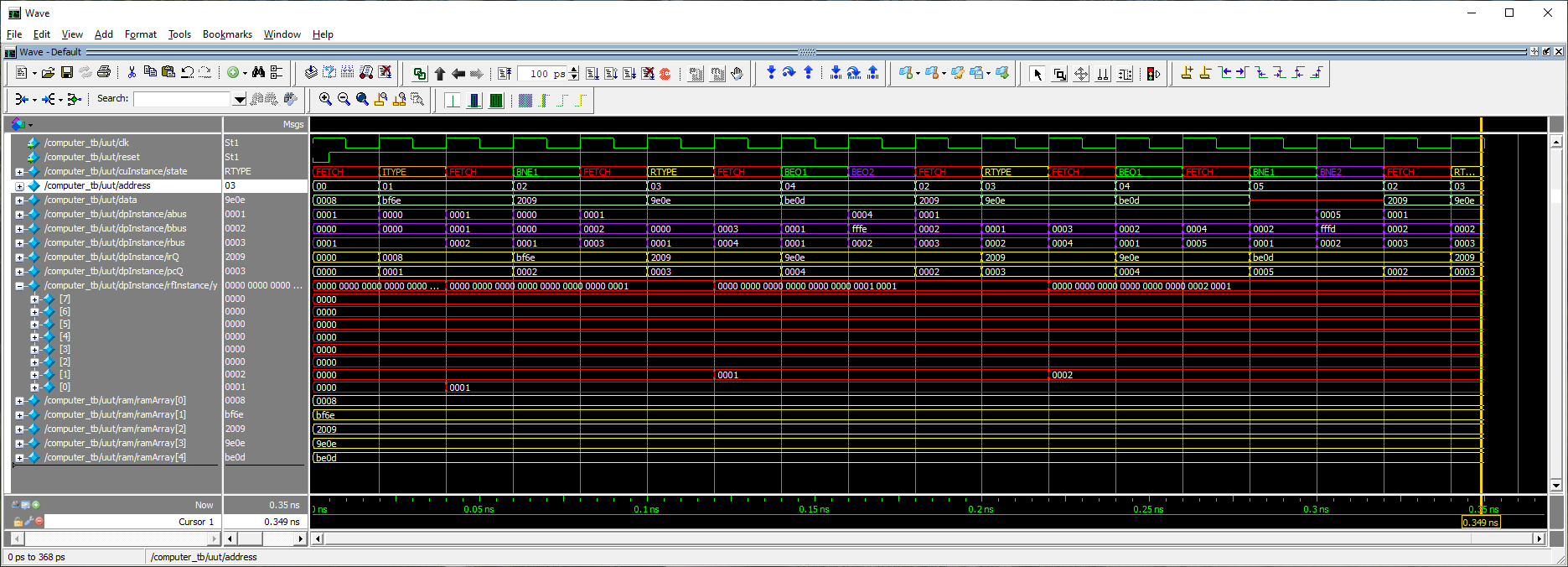
wire [7:0] address;

wire [3:0] sw;

You could easily and quickly modify the top-level module from the previous assignment to finish this module.

**Testbench**

The testbench for the computer is what is actually running the program you assembled in Table 2. Pretty cool that we are using a computer running ModelSim to simulate a computer running a program. Use the computer\_tbWaveSetup.do file provided on Canvas. You will need to alter the names of the registers in your register file as I probably implemented my register file differently that you. Please check your results against mine (except for the control word which I shaded out) and correct errors. I’ve provided two testbench outputs in Figure 6, one detailed (run for 350 time units) and one long for a prolonged duration (run for 3000 time units).



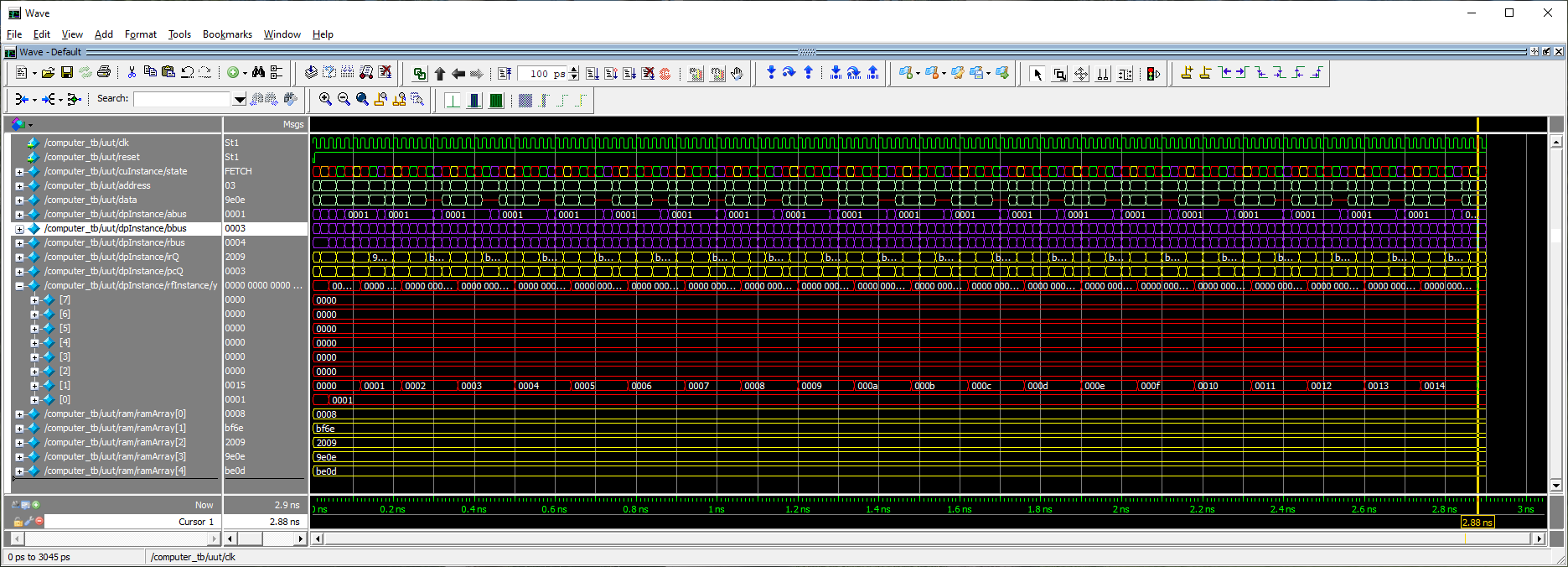


Figure 6: The simulated output of our computer running the program from Table 2. Top, run for 350 time units, bottom run for 3,000 time units.

**Deliverables**

**Today’s Program**

* Complete Table 2
* Complete Table 3

**The Control unit**

* Verilog code for the body of the datapath
* Complete testbench simulation of the control unit

**The Datapath**

* Complete Table 5
* Verilog code for the body of the datapath
* Complete testbench simulation of the datapath

**Computer**

* Complete testbench simulation.