EENG 284 – Digital Design

Lab 9 – Stored Program Computer

Part 2 – Datapath, RAM

# Objective

The objective of this lab is to automate the fetching and executing of instructions for the simpleDatapath 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.

**Today’s Computer**

In the last lab you were introduced to the ideas of instructions and how the bits of these instructions can control the elements in the datapath. One of the major short comings of this lab was that we had to provide the instructions to the datapath through the testbench signals. This is not a practical method of operation for an automatic computer.

So today, we will store a sequence of instructions in an external random-access memory illustrated in Figure 1. The datapath will read consecutive 16-bit words of the RAM and will interpret the retrieved words as instructions, and so execute them. We will use a testbench to assert the control word (cw in Figure 1) that will cause the datapath to fetch an instruction from RAM into the IR and execute them. You will automate this process next week using the finite state machine.

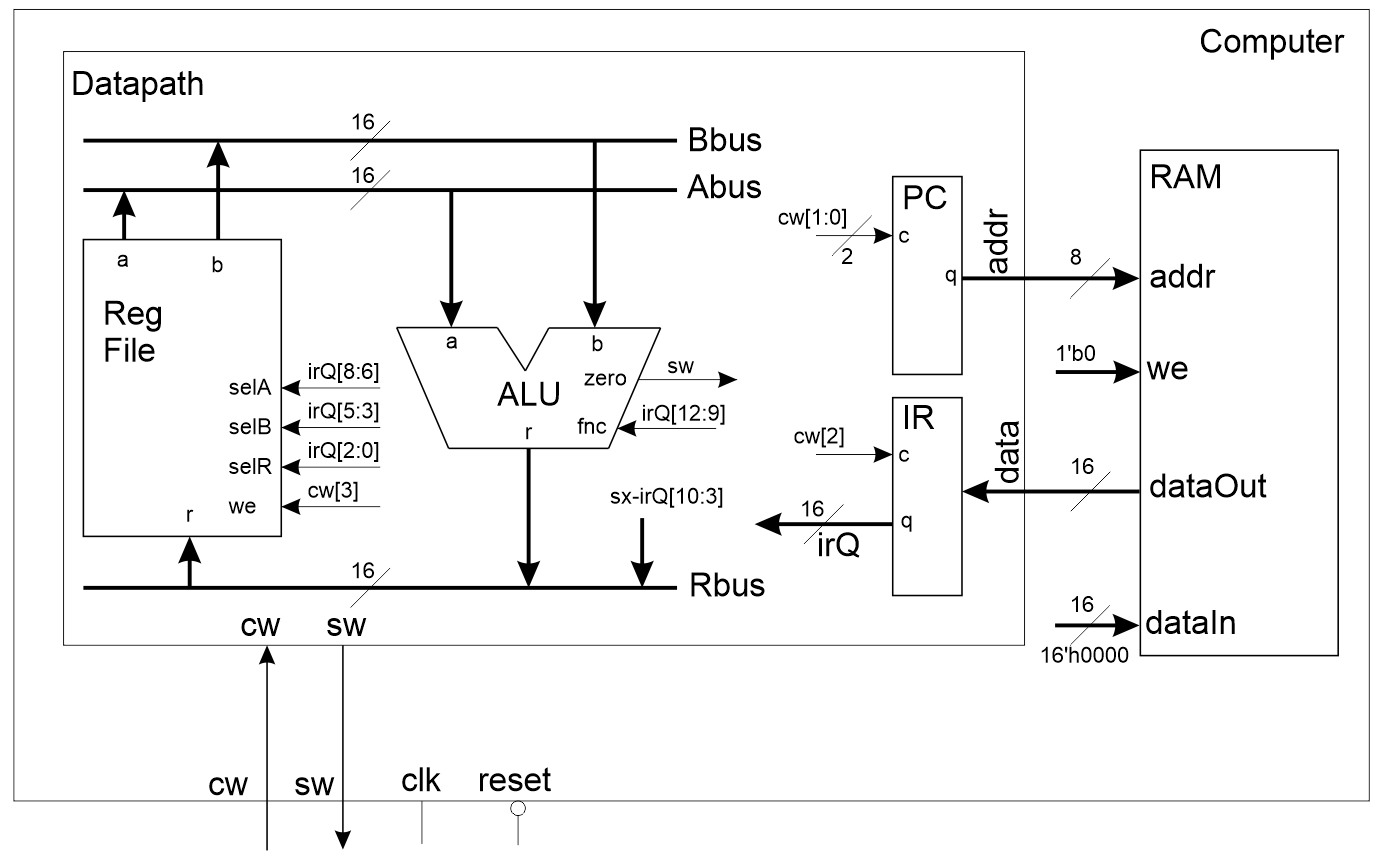


Figure 1: The datapath interfaced with a random access memory.

Let’s start by examining the RAM in Figure 1.

**Random Access Memory in Verilog**

Of all the hardware in digital design, in my opinion, none got stuck with a worse name than RAM. There is really nothing random about them except the choice of which memory location you want to access. Regardless, memories are central to the development of the stored program computer and are actually pretty easy to manage in Verilog.

The module declaration for the RAM you will use is shown in Listing 1. Note that it this RAM is different from the one we developed in class, but it still reads and writes data. Let’s talk through the operation, then you will examine a timing diagram to better understand how the RAM works in practice.

The RAM provided in Verilog is clocked, meaning that in order to perform a write operation, you need to provide a positive edge on the **clk** input. There is no reset input; the initial value of the RAM contents is set using the values in the file located at the path given as the argument of the $readmemh directive (in this case testRAM.lst). You will need to change the name of the .lst file for the computer testbench created at the end of the lab.The final argument to the $readmemh directive is the name of the Verilog object which holds the RAM contents (in this case ramArray).

Listing 1: Module declaration for the random-access memory.

module mainMemoryRAM(addr, clk, dataIn, we, dataOut);

input wire [7:0] addr;

input wire clk;

input wire [15:0] dataIn;

input wire we;

output reg [15:0] dataOut;

reg [15:0] ramArray [0:255]; // 256 16-bit words

initial

begin

$readmemh("C:/Users/chris/Dropbox/Mycourses/EENG284/lab/lab9/controlSimpleDatapath/testRAM.lst", ramArray);

end

Note when copying the file name and directory path from Windows Explorer, make sure to change the windows backslashes to forward slashes like those shown in Listing 1.

The 16-bit **datain** signal is written to the memory location given by the 8-bit **addr** when the **we** input is 1 and on a rising edge of **clk**. Our RAM will only have 256 words, mostly to make the simulation run quickly. The **dataOut** output is updated with the contents of the RAM pointed to by the address independent of the clk. There is no read enable signal.

An example testRAM.lst file is shown in Listing 2. The contents of the first row, 0000 are stored at memory location 0 and are represented in hexadecimal.

Listing 2: The contents of the testRAM.lst file are just the consecutive words you want stored in the RAM in hexadecimal.

0000

1111

2222

3333

AAAA

Each consecutive word is stored in a consecutive address. That means that AAAA is store at address 4 (remember to start counting from address 0). To better understand the behavior of the RAM, you will look at the output of a testbench that instantiates the mainMemoryRAM. Run this testbench by following these steps:

* Create a Verilog project called controlSimpleDatapath.
* Add:
  + mainMemoryRAM.v
  + mainMemoryRAM\_tb.v
  + ramTestBench.lst and
  + mainMemoryRAM\_tb.do
* Make mainMemoryRAM\_tb.v s the top-level entity
* Compile the files using “Start Analysis and Elaboration”
* Run the simulation “RTL Simulation”
* Use the do file to setup the waveforms.
  + Vsim> do ../../mainMemoryRAM\_tb.do
  + Vsim> run 200
* Save a copy of the waveform as a deliverable

Before proceeding, you need to know the difference between asynchronous or synchronous behavior because the RAM you just simulated exhibits both. An asynchronous behavior is one that occurs without respect to the clock signal. In other words, an asynchronous behavior is continuously performed; any time the inputs change the outputs respond immediately. A synchronous behavior is one that only is manifest itself when the clock signal rises.

Now answer the following questions using the information in the timing diagram formed from the testbench. Phrase your answer to each question in the form of a complete English sentence.

* Is dataOut an asynchronous or synchronous output?
* Is the RAM write operation asynchronous or synchronous?
* What is the initial value of the word stored at address 2? Represent your answer in hexadecimal.
* When is the value of the word stored at address 2 changed? (answer in nanoseconds).
* What is the value written to address 2? Represent your answer in hexadecimal.
* What value does the **we** signal need to be in order to write to the RAM?

**The Modified Datapath**

In order to integrate the RAM with the datapath you need to make changes to the datapath’s module interface and the internal organization of the datapath. Let’s look at the changes to the datapath module interface first. Using the signals that enter and leave the datapath shown in Figure 1 yields the module interface shown in Listing 3. There are four changes from the datapath you made in the previous lab; the cw, sw, address and data.

Listing : The modified datapath module interface.

module simpleDatapath(clk, reset, cw, sw, address, data);

input wire clk, reset;

input wire [3:0] cw;

output wire sw;

output wire [7:0] address;

input wire [15:0] data;

The address is generated by the datapath (from the PC) and tells the RAM which word of memory the datapath wants to read. This word of memory is an instruction that needs to be “fetched” into the datapath. The data lines hold the instruction stored at this address. Since we are not writing to the RAM in this assignment, the data lines are inputs to the datapath.

In addition to the address and data, the datapath has a control word (cw) input and status word (sw) output. The sw output is connected to the zero output from the ALU. The 4-bit control word provides control signals to the basic building blocks inside the datapath which is highlighted in Figure 2. To get a better understanding of where the control word bits are being sent and what they are doing, complete the table shown in Table 1.

* In the **Hardware** column, put the generic name of the hardware component being controlled.
* In the **Name** column, put the instance name of the hardware being controlled
* In the **Function** column put the function performed when the control bit equals the given value. You may have to open the Verilog file for the associated hardware to discover the functions performed by these bits.

Table 1: The control word bits for the datapath in Figure 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Control bits | Hardware | Name | Function | |
| cw[3] |  | Reg File | 0 |  |
| 1 |  |
| cw[2] |  |  | 0 |  |
| 1 | Load |
| cw[1:0] | Counter |  | 00 |  |
| 01 |  |
| 10 |  |
| 11 |  |

After you modify the datapath module interface, you will need to add the 8-bit program counter (PC) that holds the address of the currently executing instruction and the 16-bit instruction register (IR) that holds the currently executing instruction. If you call the output of the instruction register irQ, then you can keep all the control inputs to the register file, ALU and Rbus mux unchanged. The clk and reset inputs to both of these components are not shown

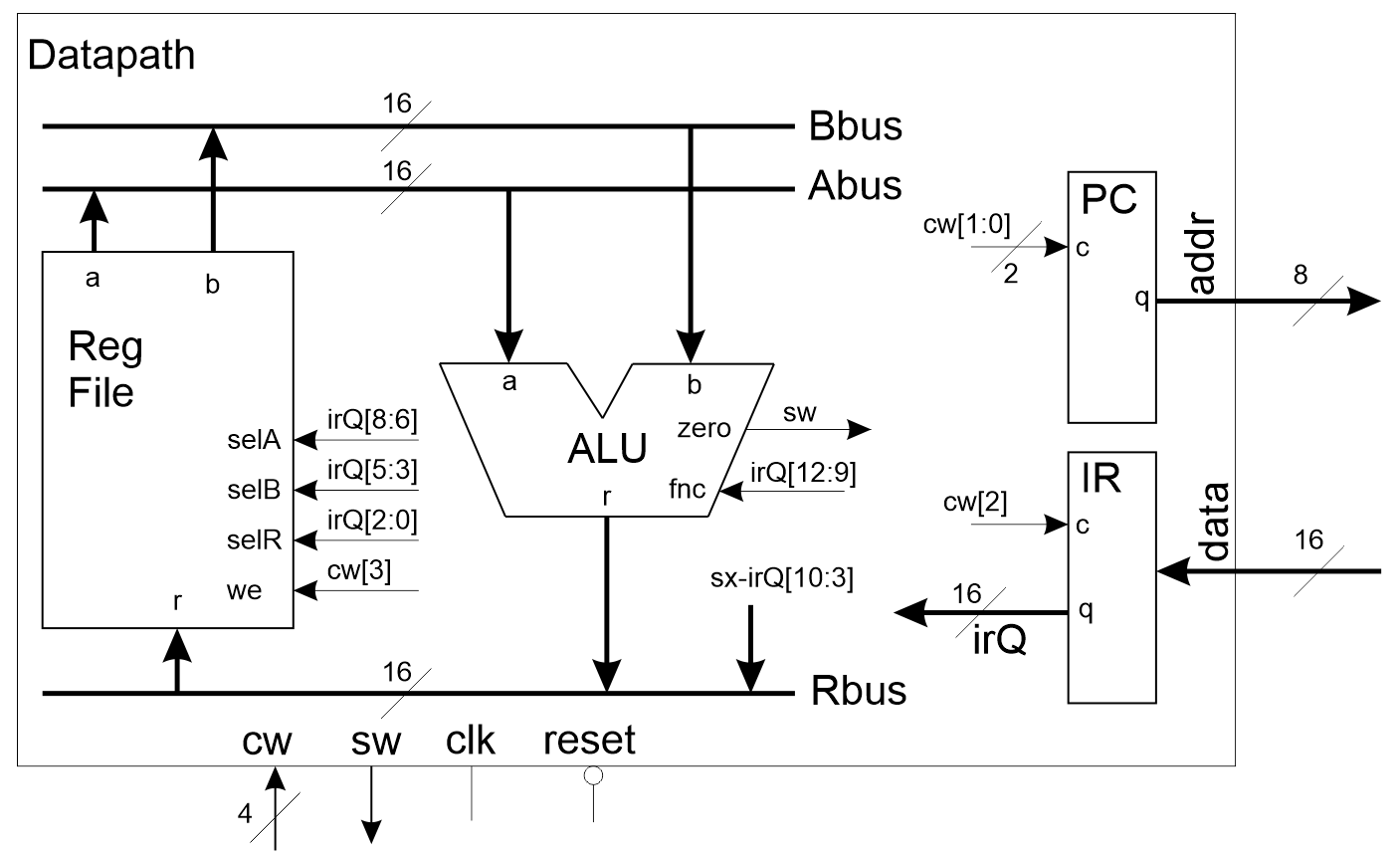


Figure : Modified datapath from the previous assignment.

**The Simple Computer**

Once you modify the datapath and have the RAM incorporated into your project, the final step in today’s lab is to create a top-level Verilog module that connects these two components together, the computer module. The architecture of the computer module is shown in Figure 1.

The module declaration for the computer is given in Listing 4. Please use this module description so that the testbench provided in Canvas works.

Listing 4: The module description for the simple computer.

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

input wire clk, reset;

input wire [3:0] cw;

output wire sw;

The body of the computer module consists of an instantiation of the datapath and RAM along with a pair of signal declarations. Hardwire the write enable and data input of the RAM to the values shown in Figure 1.

You will need to fill the RAM up with instructions, not the garbage values shown in Listing 2. You will use the instruction sequence created for the test program that you wrote last week. To do this create a new file called “ramLab09.lst” and fill it with the hexadecimal machine code instruction that you used to test the datapath in the previous assignment. The first 10 words are shown in Listing 5. Do not forget to update the $readmemh in the mainMemoryRAM module, otherwise you will be executing garbage.

Listing 5: The first 10 words of the memory that you used to test last week’s lab.

02A8

2400

2400

2400

2400

2400

2400

2400

2400

0331

Finally, you need to tell the datapath how to fetch and execute instructions by writing the control word table. To do this, you need to coordinate the functions the components controlled by the control word (and defined in Table 1) to perform a FETCH or EXECUTE operation.

To do this complete Table 2 by:

* Filling in the actions associated with each control bit for the hardware component listed. For example, when the control input to a register is 1, the register loads its data input as the stored value. The values will be same as those you write down in Table 1.
* The FETCH operation should load the contents of the RAM into the Instruction Register and hold all other values.
* The EXECUTE operation should tell the Register File to load, the Program counter to count up, and hold all other values.
* In the cw[3:0] put the concatenated 4-bit value of the 4 control bits (Reg File as most significant bit and Program Counter as the 2 least significant bits) written using the Verilog format for 4-bit values (4’bxxxx).

Table 2: The control word table for the datapath in Figure 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| State | Reg File | | Instruction Register | | Program Counter | | cw[3:0] |
|  | 0 |  | 0 |  | 00 |  |  |
| 01 |  |  |
| 1 |  | 1 | Load | 10 |  |  |
| 11 |  |  |
| FETCH |  | |  | |  | |  |
| EXECUTE |  | |  | |  | |  |

Use the values that you derrived in **cw[3:0]** column of Table 2 as the control word values in line 27 and 28 of the computer\_tb.v file provided to you on Canvas.

A waveform configuration file, computer\_tbWaveSetup.do, has been provided to you on Canvas. Use this waveform file to setup the waveform. You may need to modify the file because your datapath instance name may be different. You can get the name of the signals by manually adding the signal to the timing diagram and then looking in the console area for the command used to perform that action. Copy and paste the name of the signal into the corresponding name I used in the .do file. In case you want to do this manually, here are the signals and radix that I want in your final testbench waveform.

* clk
* reset
* address hex
* data hex
* irQ hex
* cw default yellow
* sw default yellow
* abus hex orange
* bbus hex orange
* rbus hex orange
* R0 hex red
* …
* R7 hex red

Run the simulation to 1.41ns to execute all the instruction in the memory. This is the same as executing “run 1410” from within the console area. The results of the testbench should be the same as those from the previous lab. Check that the contents of the registers match those produced by last week’s testbench.

**Deliverables**

**RAM**

* Complete testbench timing diagram for the RAM.
* Answers to the questions about RAM behavior.

**Modified Datapath**

* Complete Table 1.
* Verilog code for the body of the datapath.

**Computer**

* Verilog code for the body of the computer.
* Complete Table 2.
* Complete testbench simulation of the computer.