EENG 284 – Digital Design

Lab 8 – Stored Program Computer

Part 1 – ALU and Register File

# Objective

The objective of this lab is to design the computational core of a stored program computer. This work will form the basis for the next three labs.

**Theory – The Stored Program Computer**

The history of computing really picks up during the second world war when the U.S. defense department needed to create ballistic tables for artillery gunners. The result of work on this project was the Electronic Discrete Variable Automatic Computer (EDVAC). In many ways EDVAC operates in the same way as the central processing unit (CPU) in your computer. The genius of EDVAC’s creators Mauchly and Eckert was to store the program and data in the same memory. This architectural form was named after one of EDVAC’s contributors, the Von Neumann architecture[[1]](#footnote-1). You will be building a Von Neumann architecture computer. Let’s briefly explore how a Von Neuman architecture computer operates.

Along with binary data (think unsigned binary numbers), the memory of Von Neumann architecture computer holds instructions coded in binary. These instructions, called assembly language instructions, are very specific to each type of CPU. The CPU must fetch an assembly language instruction from the memory into the CPU. After fetching an instruction, the CPU executes the instruction. After executing the instruction, the CPU fetches the next instructions. This process goes on forever and is called the fetch-execute cycle and is illustrated in Figure 1.

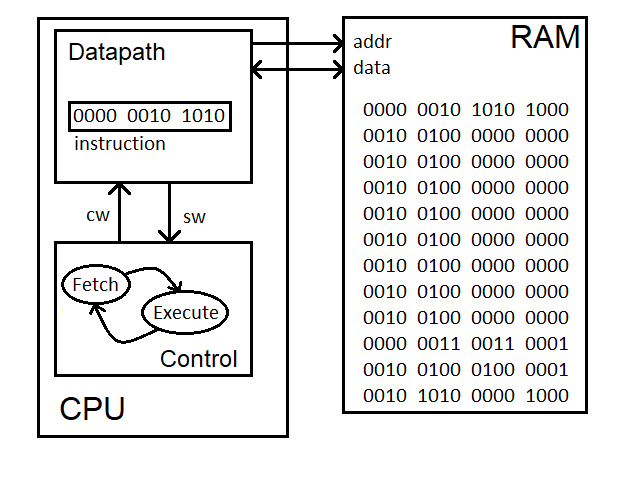


Figure : A CPU fetching and executing instructions from its external memory.

There are a lot of details that are missing from this explanation. Instead of trying to fill in all the pieces now, this week we will build the heart of the CPU, its register file and arithmetic and logic unit (ALU).

**The simplified datapath**

When I am designing a piece of electronics, I generally start with some core functionality I know I need the system to perform. I work out some of the details of this core, like what parts it will be build from and then I start spiraling outward from there, adding layers to support the operation of the core. It’s often a messy iterative, process and I seldom know what the final form will look like when I start.

The core of the computer, the register file, ALU and instruction register, are shown in Figure 2. The register file contains 8 registers, each register 16-bits wide. These registers hold the variables used in assembly language programs. If a program needs more than 8 variables, the memory is used to hold the extra. Two of the 8 registers in the register file are selected and sent to the A and B bus. The A and B bus are just a set 16-wires used to transmit data inside the CPU. The ALU takes in the 16-bit values from the A and B bus, performs an arithmetic or logic operation with them and sends the result to the R bus. The register file takes the value in from the R bus and writes it to one of the 8 registers. The instruction register (IR) holds the information about which register file registers are sent to the bus, which ALU operation to perform and which register file stored the ALU result.

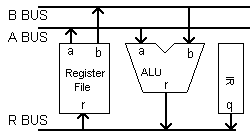


Figure : The organization of the simplified CPU datapath.

The main focus of this assignment is the design of the ALU and register file. The design of these two units is the focus of the next two sections. For the time being let’s focus on the datapath shown in Figure 2. Listing 1 shows the module declaration for the simplified datapath. Note that the IR register shown in Figure 2 is just an input port. In the next labs, this port will be replaced by an actual register, but more on that later. Beyond the register file and the ALU, the datapath will need a 16-bit 2:1 mux to determine which of the IR or ALU is put onto the R bus. On this point, not all the bits of the IR are sent to the 2:1 mux to be placed on the R bus. When the I-type instruction is discussed in a later section, you will learn more about what bits of the IR are sent to the R-bus and how to pad these bits to make them 16-bits wide.

Listing 1: Module declaration for the simplified datapath.

module simpleDatapath(clk, reset, irQ);

input wire clk, reset;

input wire [15:0] irQ;

Before you build the simplified datapath in Figure 2 you will need to first understand the operation of the ALU, register file, and the instructions. Let’s turn our attention first to the ALU.

**The Arithmetic Logic Unit (ALU)**

As its name implies, the arithmetic logic unit (ALU) performs arithmetic and logical operations on the pair of 16-bit data inputs(Abus and Bbus) and sends the result to the 16-bit output (Rbus). Our ALU is built strictly from combinational logic, meaning that the ALU is unclocked and continuously computing a result. The module description in Listing 2 contains an extra signal, zero, not shown in Figure 2. The zero output equals 1 when the result of an ALU operation equals 0.

Listing 2: The module description for the ALU.

module alu(Abus, Bbus, Rbus, fnc, zero);

input wire [15:0] Abus, Bbus;

output reg [15:0] Rbus;

input wire [3:0] fnc;

output wire zero;

The module description in Listing 2 reveals that the function input is 4-bits wide. The relationship between the 4-bit value of fnc and the operations performed by the ALU is given in Table 1. You will need to complete the **R Bus** column writing the Verilog operation in terms of Abus and Bbus. To do this, start by downloading the pdf version of the Verilog HDL Reference Manual by searching for it online. Look in the Operators chapter of the manual for the symbols used by Verilog for the operation listed in the **Operation** column. If an operation has a single operand (shift and unary negation), then perform the operation on the A bus only.

Table : The operations performed by the ALU.

|  |  |  |  |
| --- | --- | --- | --- |
| fnc | Operation | Mnemonic | R Bus |
| 4’b0000 | Addition | ADD | Rbus = Abus + Bbus; |
| 4’b0001 | Subtraction | SUB | Rbus = |
| 4’b0010 | Logical shift left one bit | SHL | Rbus = Abus << 1; |
| 4’b0011 | Logical shift right one bit | SHR | Rbus = |
| 4’b0100 | Bitwise AND | AND | Rbus = |
| 4’b0101 | Bitwise OR | OR | Rbus = |
| 4’b0110 | Bitwise XOR | XOR | Rbus = |
| 4’b0111 | Bitwise Not (unary negation) | NOT | Rbus = |
| 4’b1000 | Zero | MVZ | Rbus = |

Practice applying these operations using the values provided in Table 2. You can use the Windows calculator in programmer mode to quickly compute the values. Make sure to truncate any answers to 16-bits and write the results in hexadecimal with a “0x” in front of the value.

Table : Input and output to/from the ALU that will be used in the testbench.

|  |  |  |  |
| --- | --- | --- | --- |
| Abus | Bbus | fnc | Rbus |
| 0x55CC | 0xA3A3 | 4’b0000 |  |
| 4’b0001 |  |
| 4’b0010 | 0xAB98 |
| 4’b0011 |  |
| 4’b0100 |  |
| 4’b0101 |  |
| 4’b0110 |  |
| 4’b0111 |  |
| 4’b1000 |  |

Instead of building individual modules for each of the functions in Table 1, we will use the built-in Verilog functions. The body of the code is given in Listing 3 provides a start for what the code in the body of the ALU will look like. Use the operators you discovered in Table 1 to complete the remaining function and the code for the ALU.

Listing 3: The code to perform the ALU operations.

always @(\*)

case (fnc)

4'b0000: Rbus = Abus + Bbus;

4'b0001: Rbus =

4'b0010: Rbus = Abus << 1;

4'b0011: Rbus =

4'b0100: Rbus =

4'b0101: Rbus =

4'b0110: Rbus =

4'b0111: Rbus =

default: Rbus =

endcase

**<<Complete the missing lines of code here>>**

After completing the code Listing 3 you must remember to create an assign statement for the zero output. Do this by comparing R bus to 0. Existing code in the comparator.v file might provide some hints or you could look in the Verilog HDL Reference Manual for some guidance on how to accomplish this. This code for the zero logic goes outside the always statement.

After you build the ALU, you will check it using the testbench provided to you on Canvas. The values you computed by hand in Table 2 will be the values output by the ALU via the testbench.

**The Register File**

The register file consists of eight 16-bit registers that can be read or written to using the module interface described in Listing 4. The eight registers are identified by their 3-bit index, starting at 3’b000 and going to 3’b111. All 8 registers are reset to 0 when the **reset** input equals 0. The 16-bit output **outA** is the value held by register with index given by the 3-bit input **selA**. Similarly, **outB** is the value of register **selB**. The 16-bit input **inR** is written to the registrar with index given by **selR** when we (write enable) is equal to 1 and the clock rises.

Listing 4: The module declaration for the register file.

module regFile(clk, reset, we, selA, selB, selR, outA, outB, inR);

input wire clk, reset, we;

input wire [2:0] selA, selBb, selRb;

output wire [15:0] outA, outB;

input wire [15:0] inR;

The internal organization of the register file consists of 8 registers (provided in a previous lab), a pair of 16-bit 8x1 muxes, and a 3:8 decoder shown in Figure 3. The **clk**, **reset** and **inR** inputs are applied to all the registers. The **we** signal is sent to the control input of one of the 8 register depending on the value of the 3-bit signal selR. The 16-bit 8:1 muxes select one of the 8 register’s Q outputs to assert on the **outA** or **outB** outputs.

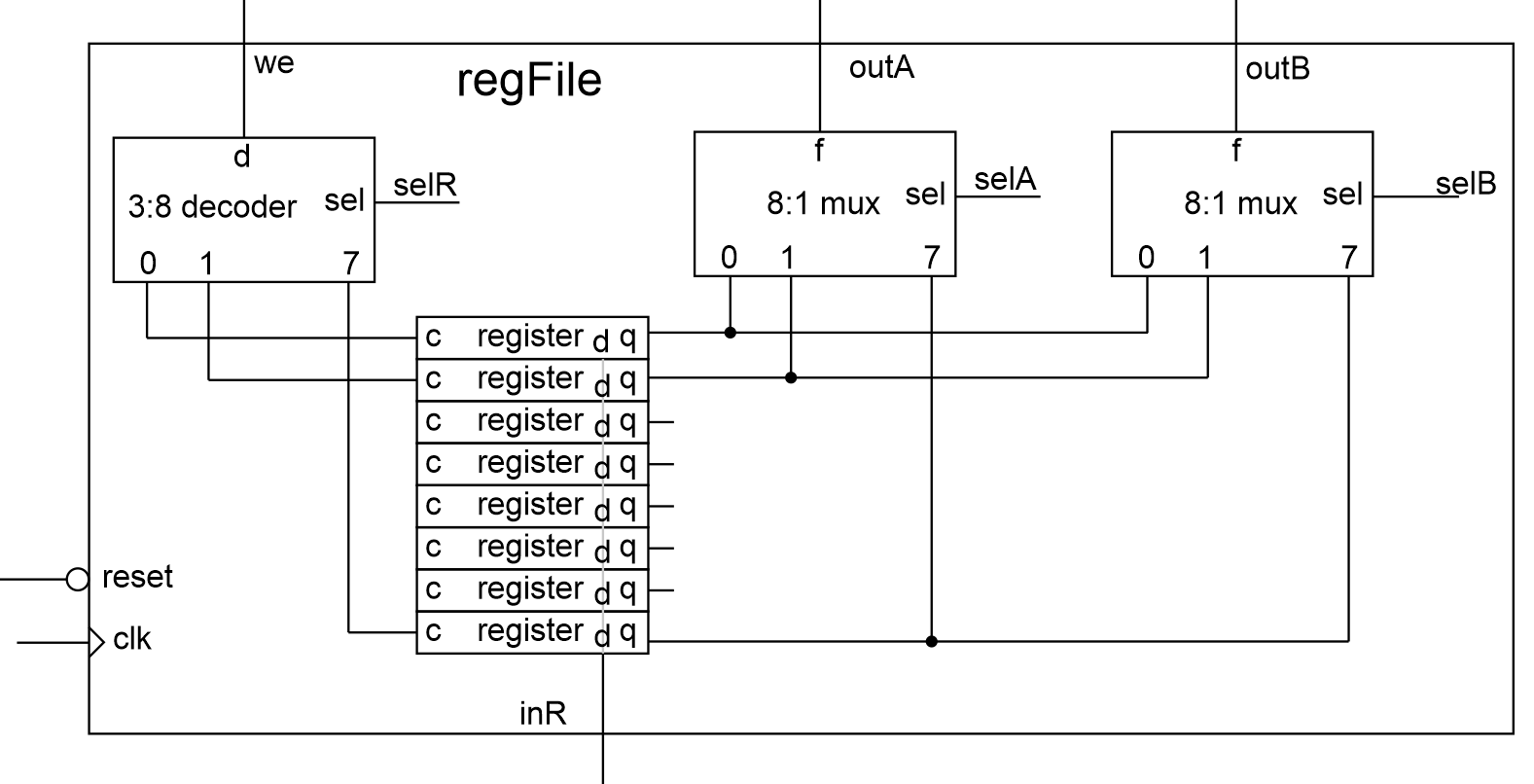


Figure : The organization of building blocks inside the register file.

To get a better understanding of how the register file operates, complete Table 3. Note that a row labeled Qi is the Q output of the ith register. Assume that the inputs, **selA**, **inR**, we are applied both before and after the clock edge. Write down the value of the outputs **outA** and **Qi** just after the clock edge.

Table : The input and output of the register file under different conditions.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| clk | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| reset | 0 | 1 | 1 | 1 | 1 | 1 |
| selA | 3’b000 | 3’b000 | 3’b001 | 3’b011 | 3’b101 | 3’b101 |
| outA (after clk) | 4’h0000 | 4’h0000 | 4’h1111 | 4’h3333 | 4’h5555 | 4’hAAAA |
| selR | 3’b000 | 3’b001 | 3’b011 | 3’b101 | 3’b101 | 3’b101 |
| inR | 4’h0000 | 4’h1111 | 4’h3333 | 4’h5555 | 4’hAAAA | 4’hAAAA |
| we | 1 | 1 | 1 | 1 | 0 | 1 |
| Q5 (after clk) | 4’h0000 |  |  | 4’h5555 |  |  |
| Q3 (after clk) | 4’h0000 |  |  |  | 4’h3333 |  |
| Q1 (after clk) | 4’h0000 |  |  |  |  |  |

After you build the register file, you will check it using the register file testbench provided to you on Canvas. The values you computed by hand in Table 3 are very similar to the values stored in the register file via the testbench.

**The Instruction Register**

You will include the instruction register (IR) as an actual register in next week’s lab. For this week, we will use the testbench to pass in values for the IR. The IR will contain all the necessary control information to run the register file and the ALU. For today’s assignment, we’ll limit ourselves to two different type of instructions in the instruction register, the R-type and I-type instructions. The “I” in “I-type” stands for immediate. An I-type instruction moves a signed 16-bit value into one of the registers in the register file. The “R” in “R-type” stands for result. An R-type instruction performs an ALU operation on two of the registers from the register file and stores the result in one of the registers in the register file. The 16-bits which make up an I or R-type instruction are arranged to the format shown in Table 4.

Table : The instruction format for the I-type and R-type 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 | | |

**I-type instruction.**

The most significant 3 bits of the I-type instruction are always 3’b000. Bits 12 and 11 are unused and generally will be assigned 0’s by default. Bits 10-3 of the I-type instruction are a signed 8-bit value that will be moved into the register with index specified by bits 2-0. The signed 8-bit value must be sign extended before it’s moved into the 16-bit file register by copying the most significant bit 8 times in front of the value. The least significant 3-bits are the index of the file register that has its value written.

When an assembly instruction is written out in 0’s and 1’s, it is said to be written out in machine language. The “machine” is the CPU. This binary form is the code that the CPU natively understands and processes. Unfortunately, this format is difficult for humans to understand, so we provide an human-readable interpretation of the machine code called assembly language. Assembly language is only a convenient re-interpretation of the machine code for humans. The assembly language interpretation of the format given in Table 4 is given by:

LDI #<8-bit signed value>, Rd

From left to right, let’s look at each of the parts of the assembly instruction to make sure that we understand what they mean. LDI stands for “load immediate”. The “#” symbol means that the thing that follows is a constant value found somewhere in the machine code for the instruction. The 8-bit signed value should be prefixed with “0b” if the immediate value is written out in binary and “0x” of the value is written out in hex. Finally, “Rd” is the destination register.

Let’s try our hand at converting between assembly and machine code while trying to better understand how the LDI instructions works by completing Table 5.

Table : The machine and assembly code for several load immediate instructions.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assembly code | Machine code | Bit 15-11 | | | | | Bits 10-3 | | | | | | | | Bits 2-0 | | |
| LDI #0x55, R0 | 0x02A8 |  |  |  |  |  | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |  |  |  |
| LDI #0x66, R1 | 0x0331 |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 | 0 | 1 |
| LDI #0x52, R2 | 0x028A | 0 | 0 | 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |
| LDI #0x01, R3 | 0x000B |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

In order to get the datapath to execute an I-type instruction correctly, you need to send bits 10-3 to the R bus and use bits 2-0 as the selR input to the register file. The immediate 8-bit value in bits 10-3 must be sign-extended into a 16-bit value before being sent to the register file. To do this, you could copy bit 10 of the IR 8 times or use a repetition multiplier (search through Verilog HDL Reference) to simplify this assign statement. Note, that for this lab, you should hardwire the write enable input of the register file to 1 because both the I-type and R-type instructions write to the register.

Now let’s turn our attention to the R-type instruction to figure out how to perform mathematical and logical operations.

**R-type instruction.**

The most significant 3 bits of the R-type instruction are always 3’b001. Bits 12 through 9 are a 4-bit fnc code that determines the operation performed by the ALU and is identical to that given in Table 1. Bits 8-6 are the index of the file register used as a source operand and sent to the ALU on the A bus. Bits 5-3 are the index of the file register used as source operand sent to the ALU on the B bus. Bits 2-0 are the index of the file register used as the destination of the operation performed by the ALU on the source operands.

Like the load immediate instruction, R-type instructions have both a machine and assembly language representation. For example, an ADD instruction using the format given in Table 4 is written as:

ADD Rs1, Rs2, Rd

This instruction is fairly straight forward to understand, it addresses the contents of the register file registers with index Rs1 and Rs2 and places the sum into the register file with index Rd. The first register reference, Rs1 in this case, is placed on the A bus, where-as the second register reference is place on the B-bus. This is important for operations where the order of operands is important.

Any of the register indices can overlap with another, even all three can be the same. This might be useful if you wanted to double the value of a register using the ADD operation. Each R-type instruction will have its own assembly language mnemonic (ADD in the example above) that corresponds to the operation performed and is given in the mnemonic column of Table 1.

Let’s try our hand at converting between assembly and machine code while trying to better understand how the LDI instructions works by completing Table 6.

Table : The machine and assembly code for several ALU instructions.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assembly code | Machine code | Bits 15-13  op-code | | | Bits 12-9  fnc | | | | Bits 8-6  selA | | | Bits 5-3  selB | | | Bits 2-0  selR | | |
| SHL R0, R0 | 0x2400 | 0 | 0 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| OR R0, R1, R0 | 0x2A08 |  |  |  | 0 | 1 | 0 | 1 |  |  |  |  |  |  |  |  |  |
| AND R0, R1, R6 | 0x280E |  |  |  |  |  |  |  | 0 | 0 | 0 |  |  |  |  |  |  |
| XOR R0, R1, R2 | 0x2C0A |  |  |  |  |  |  |  |  |  |  |  |  |  | 0 | 1 | 0 |

**Assembly language programs**

I’ll be the first to admit, programming in assembly language is a dying art. With generous memory sizes, fast processors and impressively good compilers, there is little need for assembly language programming. However, there are some cases, like ours, where we are forced to program in assembly language.

When I write assembly language, I “program” in my favorite language (C programming language), and convert from it to assembly. For today’s lab, you will not need to write a complete program, rather you will need to write enough code to replicate the set of tests in Table 2. To do this, you will use LDI statements to load registers 0 and 1 with 0x55CC and 0xA3A3 respectively. Then you will perform 9 ALU operations on registers 0 and 1. In order to accomplish the first goal, we would like to perform LDI #0x55CC, R0 and LDI #0xA3A3, R1. However, the LDI instruction moves a sign-extended 8-bit value into the destination register, so we cannot perform these operations as shown because each of them moves a 16-bit value into either R0 or R1, LDI can only move 8-bits. To explain how we will solve this problem, let’s focus on the sequence of steps needed to perform the LDI #0x55CC, R0 operation:

* LDI #0x55, R0 // Move 0x55 into R0
* SHL R0, R0 // Shift R0 left 1 bit
* SHL R0, R0 // Shift R0 left 1 bit. R0 is now left shifted 2-bits.
* SHL R0, R0 // Shift R0 left 1 bit. R0 is now left shifted 3-bits.
* …
* SHL R0, R0 // Shift R0 left 1 bit. R0 is now left shifted 8-bits.
* LDI #0xCC, R1 // Load lower 8-bits into R1
* OR R0, R1, R0 // Or the lower 8-bits into R0

For each of these assembly language instructions, you will need to write the machine code. I’ve provided a start with the simpleDatapath excel file posted online and shown in Figure 4. This file will help you write the program and testbench file used to test your simpleDatapath. To do this, start by writing out your assembly language commands in the are outlined in red. Then use the reference at the top of the file to fill in the bits for either the I-type or R-type instructions outlined in blue. I’ve tried to help myself by color coding the individual fields of the instructions in different shade of the same color. As you fill in the bits in the blue outlined area, the bits in the rightmost three columns will auto-populate. These represent the binary and hexadecimal machine language instructions. For you the right-most column will be most important, as this is the one that you will cut-and-paste into your simpleDatapath testbench file.

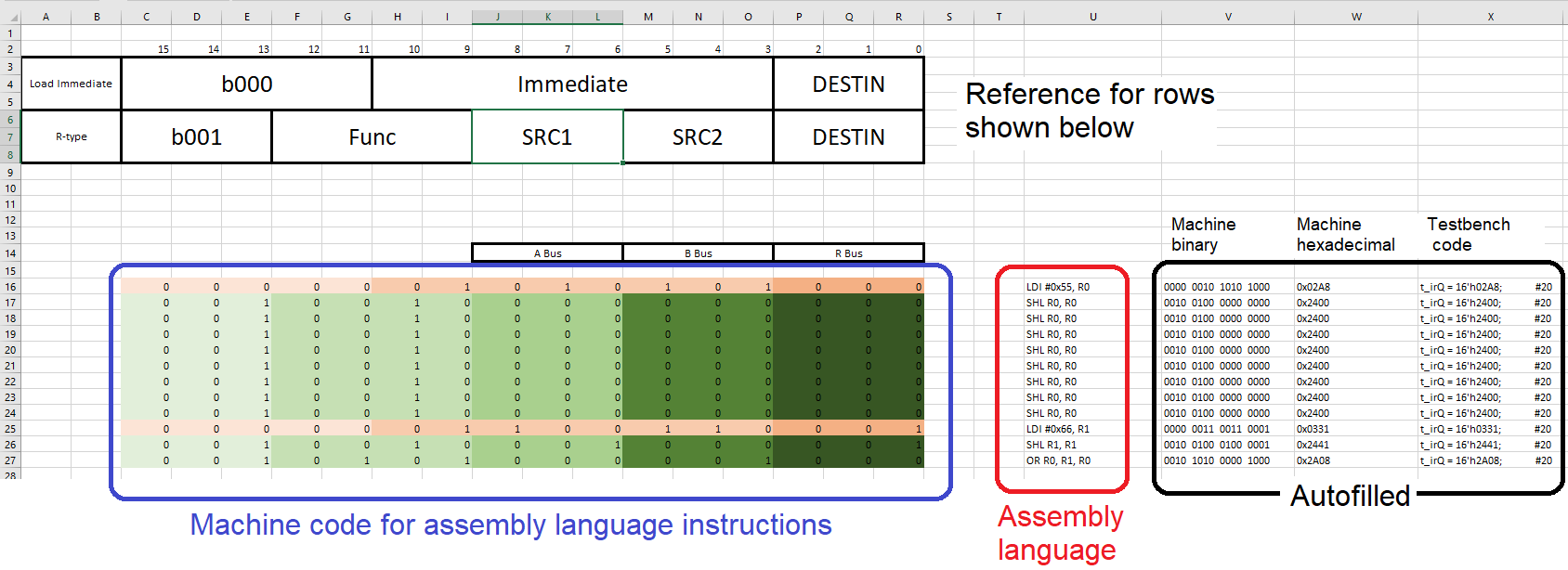


Figure : The Excel file used to help you write machine language programs used in your testbench.

Let’s try our hand at understanding how assembly instructions work by executing the Table 7 program segments that you will use in your testbench.

Table : An assembly language program fragment that is used in the final testbench.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assembly Instruction | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
| LDI #0x55, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| SHL R0, R0 |  |  |  |  |  |  |  |  |
| LDI #0x66, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| OR R0, R1, R0 | 0x55CC |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| LDI #0xA3, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| SHL R1, R1 |  |  |  |  |  |  |  |  |
| LDI #0x51, R2 |  |  |  |  |  |  |  |  |
| SHL R2, R2 |  |  |  |  |  |  |  |  |
| LDI #0x01, R3 |  |  |  |  |  |  |  |  |
| OR R2, R3, R2 |  |  |  |  |  |  |  |  |
| OR R1, R2, R1 |  | 0xA3A3 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| ADD R0, R1, R2 |  |  |  |  |  |  |  |  |
| SUB R0, R1, R3 |  |  |  |  |  |  |  |  |
| SHL R0, R4 |  |  |  |  |  |  |  |  |
| SHR R0, R5 |  |  |  |  |  |  |  |  |
| AND R0, R1, R6 |  |  |  |  |  |  |  |  |
| OR R0, R1, R7 |  |  |  |  |  |  |  |  |
| XOR R0, R1, R2 |  |  |  |  |  |  |  |  |
| NOT R0, R3 |  |  |  |  |  |  |  |  |
| MVZ R4 |  |  |  |  | 0x0000 |  |  |  |

**simpleDatapath**

We will end this where we started, with the datapath in Figure 2. You will need to create a Verilog file for the using the module declaration in Listing 1. The body of my Verilog code had 4 lines of code, three component instantiations and an assignment statement for the immediate. Once you have this complete, you will need to check simpleDatapath. The value you computed by hand in Table 7 are the values your datapath will compute using the provided datapath on Canvas.

**Deliverables**

**ALU**

* Complete Table 1.
* Complete Table 2.
* Verilog code for the ALU body.
* Complete testbench simulation
  + Abus radix hex trace color Green
  + Bbus radix hex trace color Green
  + Fnc radix binary trace color Magenta
  + Zero radix binary trace color Yellow
  + Rbus radix hex trace color Red

**Register File**

* Complete Table 3.
* Verilog code for the register file body.
* Complete testbench simulation.
  + clk default radix trace color Green
  + reset default radix trace color Green
  + selA radix unsigned trace color Magenta
  + outA radix hex trace color Magenta
  + selB radix unsigned trace color Turquoise
  + outB radix hex trace color Turquoise
  + selR radix unsigned trace color Gold
  + inR radix hex trace color Gold
  + we default radix trace color Gold
  + Q7 … Q0 radix hex trace color Red

**The Instruction Register**

* Complete Table 5.
* Complete Table 6.

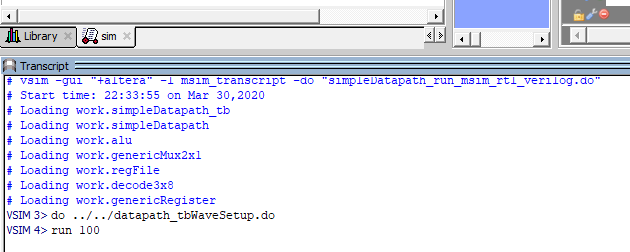
**Assembly Language Programs**

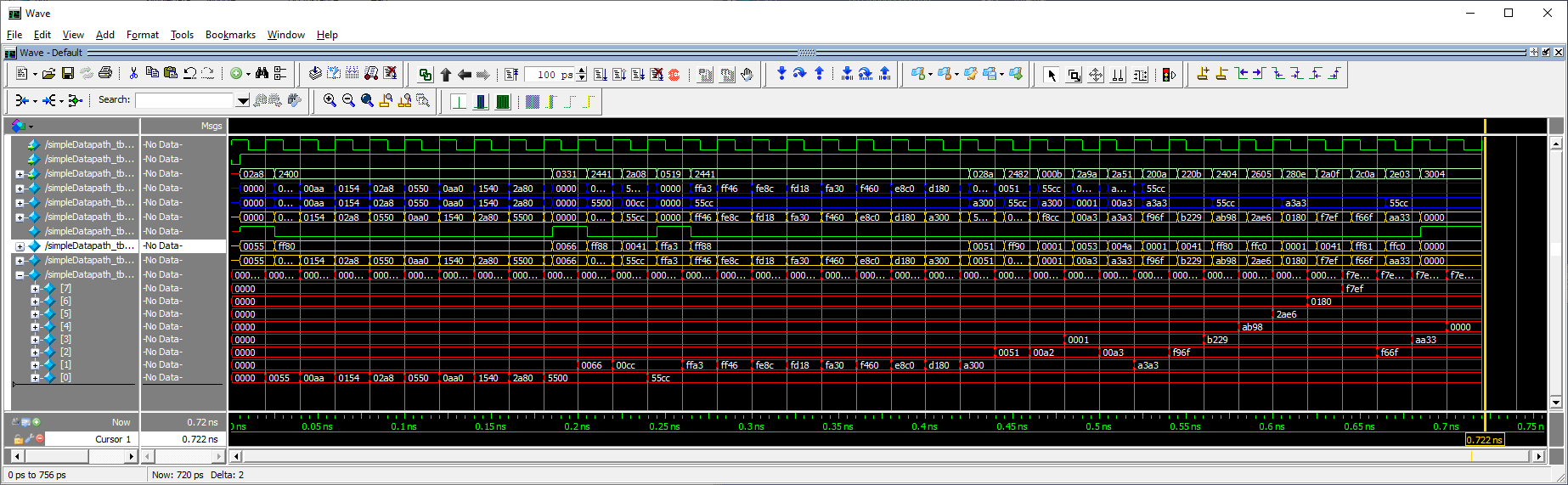
* Complete Table 7.

**simpleDatapath**

* Verilog code for the datapath body.
* Complete testbench simulation.
  + clk default radix trace color Green
  + reset default radix trace color Green
  + selA radix unsigned trace color Magenta
  + outA radix hex trace color Magenta
  + selB radix unsigned trace color Turquoise
  + outB radix hex trace color Turquoise
  + selR radix unsigned trace color Gold
  + inR radix hex trace color Gold
  + we default radix trace color Gold
  + Q7 … Q0 radix hex trace color Red

I find setting up the testbench waves to be a pain, especially if you make a mistake and need to run it again. In order to simplify the process of setting up the waveforms, you can use the provided DO file to setup the simpleDatapath simulation as follows:

* Download “datapath\_tbWaveSetup.do” into the project directory which contains simpleDatapath.v and simpleDatapath\_tb.v
* Open datapath\_tbWaveSetup.do using Notepad. The syntax is pretty straight forward and corresponds to the buttons pressed in ModelSim.
* Successfully compile your Verilog file with simpleDatapath\_tb as the top-level
* Launch Model Sim
* Open work and simulate simpleDatapath\_tb
* In the console area of ModelSim (shown in the image below) type:
  + VSIM 3> do ../../datapath\_tbWaveSetup.do
  + 
* You can type “run <time>” in this area (as shown) to simulate some amount of time. I found this VERY handy when debugging my datapath.



1. The other major architectural form, the Harvard architecture, stores the program in a physically different memory from the data. [↑](#footnote-ref-1)