ECE 3700 Lab 5

Design of a 4-bit Computer

November 4, 2024

Contents

Chapter 1

Introduction

Congratulations! You are now familiar with both combinational and sequential logic design and are therefore experts in writing Verilog. You may not yet feel confident in your abilities, but you now know enough about digital design to build quite complex circuits. This lab will demonstrate that for you by walking you through the simple task of building an entire (sort of) CPU. After completing this lab, you will have some insight into the design of microprocessors and microcontrollers.

1.1 Introduction to the Problem

You may have heard that most digital computers are based on the Von Neumann Architecture which is, to put simply, an organization where memory and processor are separated from each other. This separation allows us to build general purpose computers that can execute more than one program instead of having to build a new circuit every time we want to do a new operation.

Regardless of which architecture they use, all computers consist of the three following logical parts (observe figure 1.1):

1. Memory: The storage space for instructions to be processed by the computer.

2. Datapath: The circuits necessary to perform any operation the computer is capable of.

3. Control: The circuitry needed to configure the data path so that it executes specific instructions.

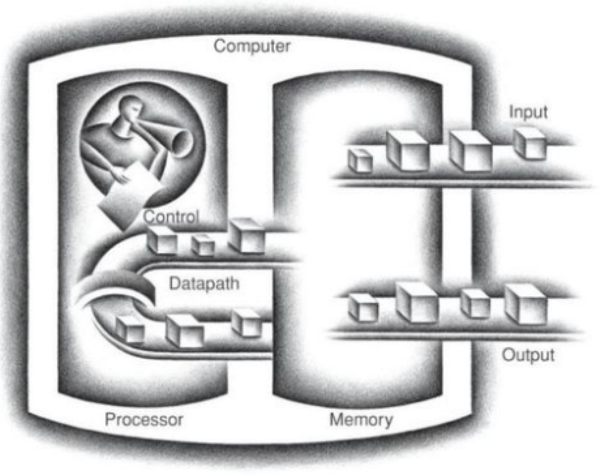

Figure 1.1: The most simple abstraction of a computer.
(Figure was pulled from the Patterson and Hennessy textbook.)

To get a better understanding of each part, observe Figure 1.2 which shows a processor for the MIPS assembly language. Do not try to understand the entire picture as it is quite complicated. Just read and consider the following points one at a time:

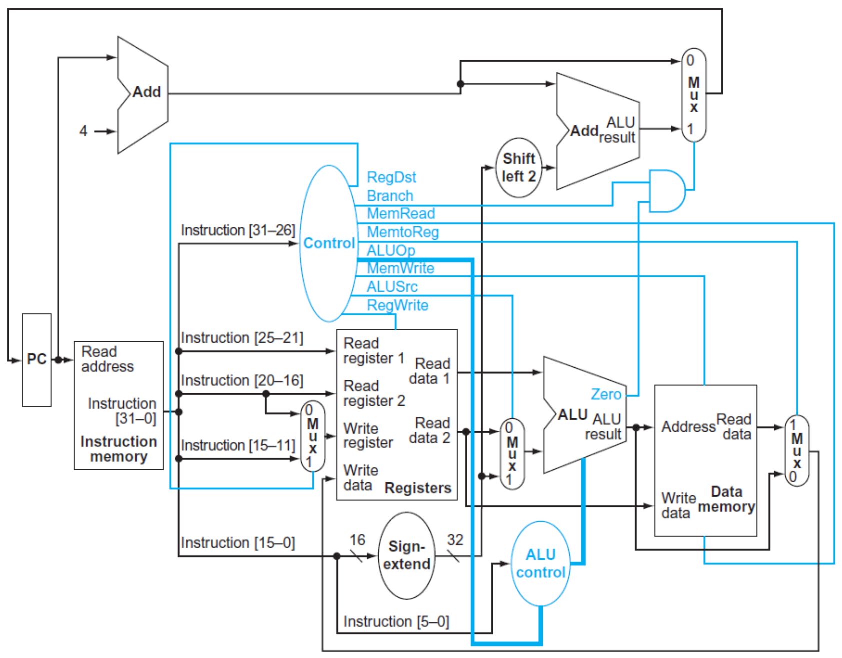

Figure 1.2: A more practical example of a computer.
(Figure was pulled from the Patterson and Hennessy textbook.)

- Everything colored in blue is circuitry for the Control portion of the computer.
- Everything colored in black is the circuitry for the Datapath.

- From our perspective as Verilog coders, the Datapath acts as the 'top level module'. Memory, Control, as well as a variety of other circuit blocks, like ALU and multiplexors, are instantiated within the datapath.
- At any given moment of time, one instruction is fed from the output of the memory unit (labeled "instruction memory" in the figure) into the datapath. A portion of this instruction is fed into the Control unit (anything that's blue is part of the control unit).
- The control unit outputs multiple signals that act as multiplexor selectors, enable/disable signals, or other control such as ALU operation selection.

To summarize, the picture in figure 1.2 is showing that an instruction is read from the memory unit, decoded by the controller, then executed in the datapath. This is essentially all a computer does but repeated multiple times through the use of a program counter that fetches a new instruction after the current one has been executed.

There are many more details that would take multiple semesters to cover. If you take ECE3710, you will understand figure 1.2 better after designing your very own computer just as complicated as the one in figure 1.2. In ECE5780, you will learn that technically, a microcontroller runs through a Fetch, decode, execute loop while a microprocessor runs through a fetch, decode, load_operands, execute, write_result loop. These details are not important to us now. This lab is simply meant to expose you to the design process.

For the purposes of this lab, we will provide you with a definition for the datapath and ask you to design the control unit for it. We will not ask you to design memory either. Instead, your control will be a Moore-style FSM that runs through ten states and at each state it will perform an instruction. That's pretty much the same as having a program/memory with only ten instructions in it because

> **any program can be thought of as a Moore-FSM where each instruction is just one of the states.**

Chapter 2

Your Assignment

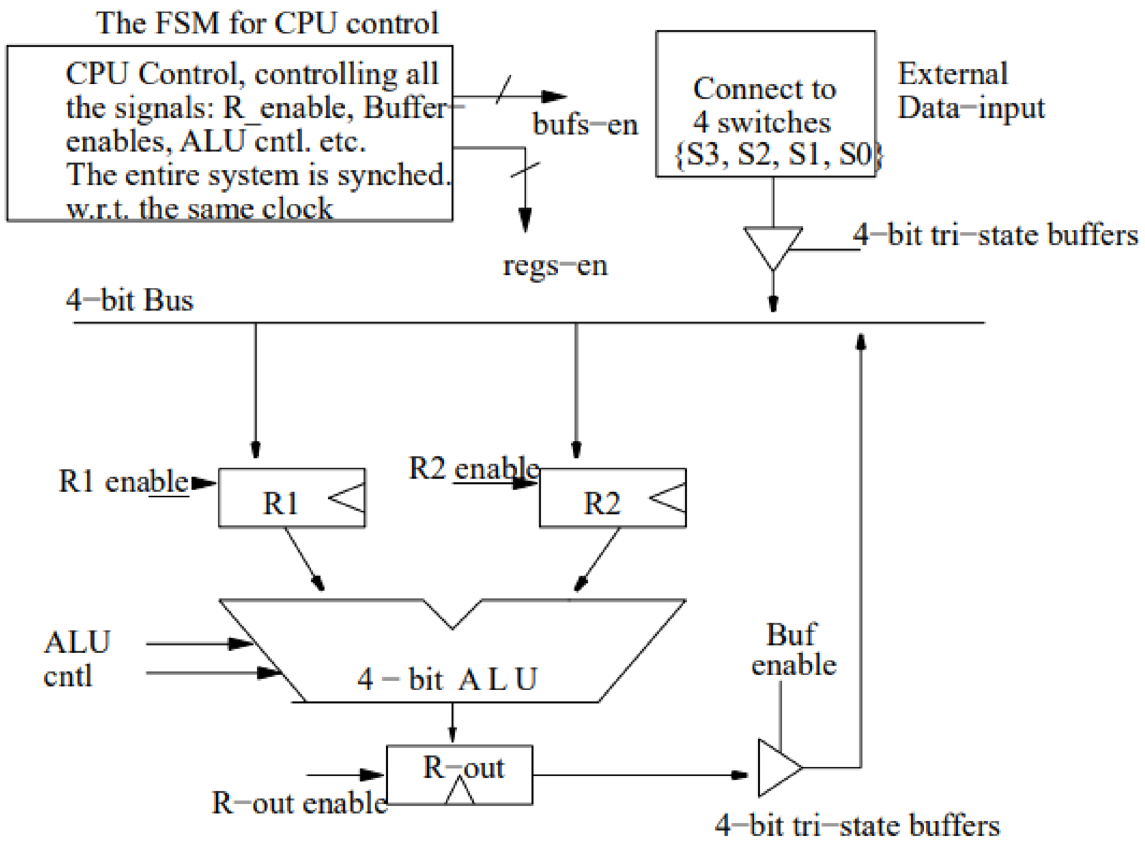

Figure 2.1: The datapath for 3700 Lab 5's 4 bit computer.

We have provided you a block diargram of the datapath (figure 2.1) to use as the 'top-level' module in your Verilog project.

2.1 Your Datapath will have the following:

- An Arithmetic and Logic Unit (ALU) that takes 2 4-bit numbers as inputs and produces an output (4-bit for logical ops, 5 bits for ADD, but you may choose to ignore the carry bit). The operations that the ALU performs are: ADD, AND, OR, and NOT (of one of the inputs).
- There are three 4-bit registers R_1,R_2, and R_{out}.
- There is a 4-bit BUS that enables us to move data. In figure 2.1, that data may come from the R_{out} register or 4 switches on your FPGA.
- Tri-state buffers are used to prevent multiple drivers on the same wire (the 4-bit 'main' BUS). If you prefer not to use multiple tri-state buffers, then you can use a single Multiplexer at the bus to receive all control signals needed to 'select' a single value to drive on the BUS.

Note: In a general purpose computer, the bus is connected to a memory that holds a program. The CPU control then has the responsibility to fetch/decode/execute the instructions - usually done by enabling/disabling the buffers and registers, just as required above. The FPGA board does have an SDRAM where you can store a large program. However this requires an interface via a memory controller and the subsequent design and synthesis would require a lot more time and effort than a 2-week lab project. It would, in fact, turn out to be a mini-3710 project. So, I'm asking you to just "hard-code" a program in an FSM, and implement the required CPU control. This way, you will get a fair idea of why CPU control is a state machine and how it works!

Note: The entire system is synchronized with a common clock. The block-diagram is just a reference, **you may (or may not) be required to add/delete a few signals**.

Just make sure to implement: i) a global reset as an external combinational input connected to a push-button or switch on the FPGA board; ii) connect a 7-segment display to R_{out} to view the result; and iii) 4 slide-switches to read external data as input.

2.2 Your FSM will interface with the datapath to meet the following specifications:

- At Reset, it goes to State-0 (S_0). Here, it tri-states all bufs, and resets all registers to 4'b0000.
- When reset is de-activated, the machine transitions to a next state (from S_i to S_{i+1}) at every positive-edge of the clock.
- In State-1 (S_1), load a four-bit integer value from your FPGA switches into R_2.
- In state S_2, load a four-bit integer value of 3 (4'b0011) into R_2. This is akin to the load immediate instruction.

- In S_3, ADD $R_1 + R_2$ and store the result into R_{out}. For this, you will have to give a signal to the ALU that 'tells' it to perform the ADD operation on the data available at its inputs, and you have to enable R_{out} so it will store the result at the next posedge clk. Instruction: (Rout ← R1 + R2).
- In S_4, transfer the data from R_{out} to R_2: Mov R2 ← Rout
- In S_5: $R_{out} \leftarrow R_1$ OR R_2 (bit-wise OR).
- In S_6: Mov $R_1 \leftarrow R_{out}$
- In S_7: $R_{out} \leftarrow NOT(R_1)$ (bit-wise complement).
- In S_8: Mov $R_1 \leftarrow R_{out}$
- In S_9: $R_{out} \leftarrow R_1 \oplus R_2$ (bit-wise XOR)
- When you get to S9, you display the result on the 7-segment display and stay in this state until the reset is pressed. In which case re-start the process from S_0.

For your convenience, I put the instructions for each state into a table you can easily reference.

Table T-1: State/Instructions

Present State	instruction to execute
s0	reset all registers
s1	$R_1 \leftarrow$ FPGA switch input
s2	$R_2 \leftarrow 3$
s3	$R_{out} \leftarrow \mathrm{R}_1 + R_2$
s4	$R_2 \leftarrow \mathrm{R}_{out}$
s5	$R_{out} \leftarrow \mathrm{R}_1$ OR R_2
s6	$R_1 \leftarrow \mathrm{R}_{out}$
s7	$R_{out} \leftarrow \neg \mathrm{R}_1$
s8	$R_1 \leftarrow \mathrm{R}_{out}$
s9	$R_{out} \leftarrow \mathrm{R}_1$ XOR R_2

Chapter 3

Advice

- USE MODULAR DESIGN! Break the problem into as many pieces as you can that can be individually tested. It'll make your life easier.

  For example:

  1. A 4-bit register.
  2. A 4-bit ALU, with 2 4-bit data inputs, a 4-bit output, and as many control signals as it takes to specify each operation you're performing ($log_2(4) = 2$).
  3. Tri-state buffers that enable you to write multiple values to the BUS. Be aware however that you cannot have multiple drivers to the same wire. This is the challenge that tri-state buffers overcome. You could also use a mux as mentioned previously.
  4. A HEX-to-7-segment-display module, connected to R_{out}.
  5. A module for your FSM. Think about what the inputs of this FSM are going to be. The outputs of this FSM should be all the required buf-control or MUX-control signals, and the reg-enable signals.
  6. Your top-level hierarchy would then interconnect all these modules via wires/tri-state buffers.

- Regarding your Register module, there are a few ways to make a register. For example:

  i Asynchronous (Event based)

  ii Synchronous (Respond to the clock)

  iii Level Sensitive

  iv Edge-triggered

  Each attribute has consequences on how the component behaves. When it comes to CPU design, registers are typically Synchronous Clock-Edge-Triggered with Asynchronous Reset and Enable signals.

  If you don't know what that means, I recommend asking someone who does to show you what an always block looks like for each attribute listed.

- For your convenience, I calculated what the final result should be for each possible combination of the slide switch input (you're welcome). Here are the results:

Table T-2: Input/Output mappings

slide switch input	hex result
0000	F
0001	E
0010	D
0011	E
0100	F
0101	A
0110	9
0111	A
1000	F
1001	E
1010	D
1011	E
1100	F
1101	2
1110	1
1111	2

- This lab is highly susceptible to bugs. Putting work into designing and understanding the problem before you start coding will make your life easier. Be aware that you can debug testbenches (and any Verilog file that uses the "initial" keyword) in Modelsim/Questa. The process is the same as we followed when simulating waveforms except, instead of adding a wave after you begin your simulation, you will right-click your Verilog file and click "edit". Then you can set breakpoints, and step through the code and observe how data changes. There is also a "watch window" you can open to keep track of data from multiple modules at the same time.

- There are some instructions that don't need access the 4-bit BUS. What value should the BUS hold in this case? Would it be okay to not drive any value on the BUS? If there are no drivers on a wire, what value does the wire hold? My advice is to assume that if there are no drivers to a wire, then the wire will hold random values that can cause problems if propagated throughout the circuit. Therefore, a wire's value should be known, even if the wire is not used. This is just my advice. Not a requirement.