CIS 351 – Building a Simple Computer

At this point in the course, you know most of the logical parts you need to build a basic computer. Much of what a computer does is (a) store information in registers, (b) perform operations on the stored information, and often (c) store the result in a register. We'll build up this functionality a little bit at a time.

1. You already know how to build an adder and registers. Using these as black boxes, draw a circuit that adds the contents of two registers. (Don't overthink it!). You can leave the inputs to the registers loose for now. We will connect them later.

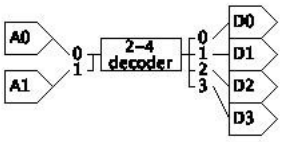
2. It's hard to do a lot of work with just two registers. Let's assume you now have 4 registers. You now need some way of choosing which two to pass to your adder for a particular operation. Draw a circuit that **accepts inputs to choose any pair** of registers and send them to the adder. How many bits will each input need to be?

3. We can now perform work on stored operands, but we currently are not putting our results anywhere. Next, we want to **store the result of the addition to a register**. For now, assume we always put the result in register 0. Modify your circuit to write the sum to register 0.

4. Why does writing to a register that we are reading from not cause us to get the wrong result?

There is a circuit element we will need that we have not yet discussed called a *decoder*. A decoder takes *n* bits of input and produces 2^n wires of output, each with one bit. If you think of the inputs as an *n* bit binary number, the decoder puts a logical 1 on the corresponding output wire while putting the rest at logical 0.

For example, the decoder shown below is a 2-4 decoder, meaning it has 2 bits of input and therefore 4 outputs. When the inputs are 00, the 0^{th} wire is active/high voltage/logical 1 while the rest are 0. When the inputs are 10 (which is binary 2), the output wire 2 is active while the rest are low. The truth table to the right shows the behavior of a 2-4 decoder for all possible inputs.



Truth Table

A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

You can use a decoder if you want to specify exactly one output wire to be active. For example, if you want the last wire (D3) active, you would choose input 11 (binary 3).

5. To check your understanding of decoders, draw a 3-input decoder and answer the
   following about it:

   How many outputs does it have?

   Which output is active if the input is 010?

   How would you set the inputs if you wanted output 5 to be active?

6. It would be very helpful if we could write to any register we wanted instead of just register 0. Our next goal is to modify the circuit from question (4) to allow writing to any register of our choice.

   Before we get there, we will start with an obviously bad idea. Modify your circuit so that it writes to *every* register.

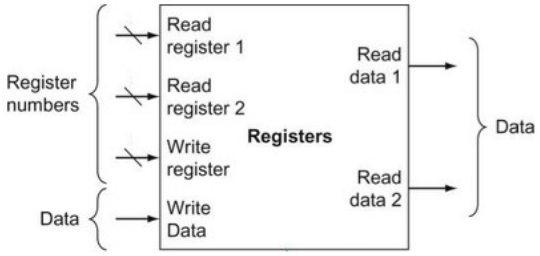
7. To allow a choice of which register to overwrite, we will need to accept another input specifying that register. When designing the circuit, two key things will be helpful to us – enabled registers and the decoders we just introduced.

   You have seen how to build *enabled* registers – registers that do not change their stored value when their 'enable' input is set to False, regardless of the clock. Assume all of your registers have an enable input built in.

   Modify your circuit to **write data to a specified register given a new input**.
   Hint: there is a reason we just introduced decoders.

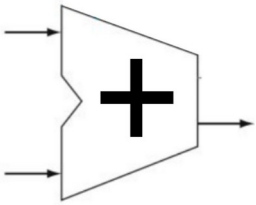
8. We now have something that looks a little like a basic calculator. It stores numbers, adds them, and stores the results. Before we add any more functionality, we'll use our idea of *abstraction* to make our circuit look a little simpler.

   We're going to encapsulate all the logic for choosing which registers to read and write inside one of our boxes and call it a *register file*. It will look something like this in diagram form:



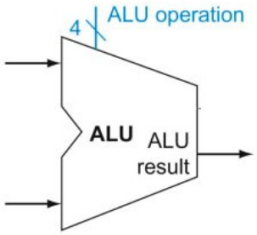
[1]

   With this change, the diagram should look much simpler. Connect the wires below to complete the circuit (you can add "input pins" to represent the inputs to the circuit).



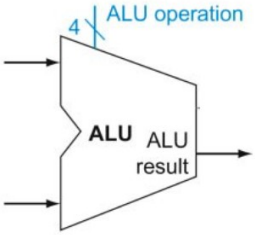
9. Being able to add is well and good, but you want to perform more operations than that. For example, you are currently building a subtractor and an SLT. In project 3, you will combine all of these and more into a single circuit called an *Arithmetic-Logic Unit (ALU)*. This circuit takes two inputs to operate on, like an adder does, and an additional control signal specifying what operation to perform (addition, subtraction, etc.) on the inputs. The

diagram form of an ALU, along with a subset of its operation codes, is shown below.



Some ALU operation codes
Add: 0000
Subtract: 0001

We can drop this into our previous circuit and connect it in the same way, just with one additional input for ALU control. With this, our "computer" is as complete as we will make it for today. Now, we will start programming it!



10. We "program" our simple computer by controlling its inputs. For example, if I wanted to add the contents of register 0 to register 1 and store the result in register 2, my inputs would be

ReadReg1	ReadReg2	WriteReg	Op
00	01	10	0000

If I wanted to add the contents of every register and put the result in register 0, I would need more than one command. Convince yourself that the following inputs, given in this order, would obtain the desired result.

ReadReg1	ReadReg2	WriteReg	Op
00	01	00	0000
00	10	00	0000
00	11	00	0000

Now, write your own program to implement the following (`RX` is register `X`):

```
R0 = R1 + (R2 – R3)
```

ReadReg1	ReadReg2	WriteReg	Op

(You may not need to use every row of the table).

11. If we are careful to keep inputs in the same order and size, we can write code more concisely without the table. Each binary string below represents a command. All we did was squeeze the columns of the table together into a single bit string. What does the resulting program do?

    Even though this notation is more concise, it is harder to read. You need to first break each binary string into pieces according to the table and translate them to human-readable operations.

    ```
    0001000000
    1011100000
    0010001000
    ```

12. As programmers, we don't want to use a table every time, but programming directly in binary would be the worst! Instead, we use mnemonics to write out our code. One example is:

    ```
    add $r2, $r0, $r1
    ```

    This instruction is adding register 0 to register 1 and storing the result in register 2. That's a lot more clear than the equivalent binary (`0001100000`), but it isn't much more complicated. As long as you know that the pattern is

    ```
    op_mnemonic writeReg, readReg1, readReg2
    ```

you can easily recover the binary using a table like problem (10). This is what *assembly language* is all about. You will learn to write code like this, a program called an *assembler* will make the simple changes to turn it into the equivalent binary, and then the computer can run it!

Write the same program you did in (10) to compute

```
R0 = R1 + (R2 – R3)
```

but, this time, use assembly language.

13. You could use a carry-select, ripple-carry, carry-lookahead, or any of a number of other adders in your ALU to change the efficiency of your circuit. These adders produce the same result, but do so at different speeds. How would the program we wrote change if we changed the adder used in our ALU?

14. If we wanted to add four more registers to our circuit (for eight total), we would put them in the register file and the circuit would look the same. Also, the assembly code we wrote wouldn't change in this case. However, this change *would* affect the binary commands. Why? How? Try writing out a few if you are unsure.

There may have been some questions that occur to you about this simple machine, such as:

- How do we get values into the registers in the first place?
- What if we need to store more information than can fit in the registers?
- I wouldn't want to put these binary commands into the circuit by hand one at a time – I want to write a program and have it run automatically! How do I do that?

We will add all of this functionality to the circuit over time.

[1] Register file and ALU images from "Computer Organization and Design" by Patterson and Hennessy.Retrieved from http://cs.middlesexcc.edu/~schatz/csc264/handouts/mips.datapath.html Minor modifications made.