Lab 10: Register File and RAM
CS/CoE 0447: Spring 2017

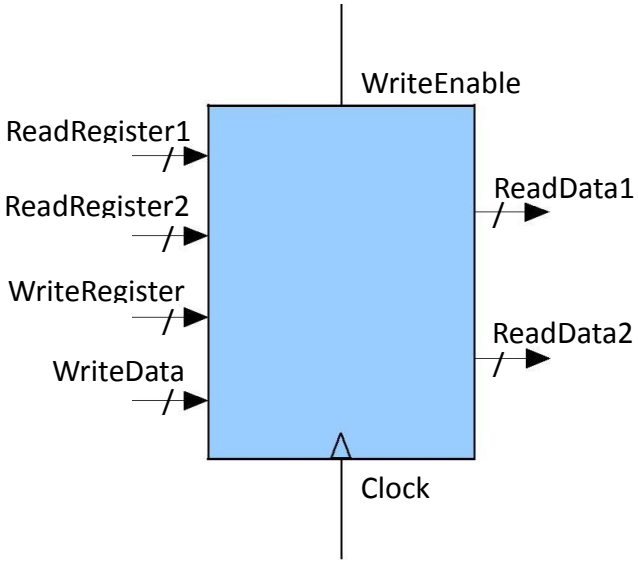
Each person must turn in their own copy of the lab. If you choose to work with a neighbor/partner, enter your partner's name into CourseWeb. Submission timestamps will be checked. Late submissions will not be accepted. **Follow instructions.**

In this lab, you will build two sample circuits using Logisim. (Logisim can be downloaded from http://www.cburch.com/logisim/download.html.) There are **two** parts to this lab. In the first part, you will design a Register File. The register file can be used (with appropriate modification) in your Project 2. In the second part, you will work with RAM.

Part 1: Register File

Logisim has a very detailed set of documentation. Go to the "Help" menu then choose **"Library Reference."** You will be able to read detailed information about each component within Logisim, including how to use the components (inputs and outputs) and the different ways of configuring the components. You are using several never-before-seen components in this lab. The technical library will greatly assist you in understanding them.

A register file is a subcircuit that lets the surrounding circuitry read from one or two registers, which are chosen dynamically. A register file also lets the surrounding circuitry optionally write to one register, which is also chosen dynamically. In this part of the lab, you will build a register file that has two read ports and one write port. That means that your register file circuit should let outside circuitry read two registers[1] at a time while also supporting writing to one register at a time. You should support four registers total. Each register must hold 16 bits of data. Use Logisim's built-in **register component**. **Save your register file circuit as lab10part1-<Your Pitt Mail ID>.circ (for example, lab10part1-xyz123.circ).**



Here is a picture of what your register file subcircuit, *symbolically*, looks like. The slash indicates that an input/output contains more than one bit of data.

[1] The registers read do not necessarily have to be different. For example, it is valid to read from $r0 and $r1, but it is also valid to read from $r2 and $r2 simultaneously.

- **ReadRegister1** specifies which register should be read from. That register's data should be presented on the ReadData1 output.
- **ReadRegister2** specifies which register should be read from. That register's data should be presented on the ReadData2 output.
- **WriteRegister** specifies which register should be written to.
- **WriteData** is the data that will be written to the register specified by WriteRegister. On a rising clock edge, the data will be written to the specified register.
- **WriteEnable,** if true, will cause the WriteData to be written to the register specified by WriteRegister; otherwise, WriteData will be ignored and no register will be written to.
- **Clock** conveys the clock signal.

Before starting, ask yourself the following questions:

1. How can we tell how many bits will be outputted on ReadData1? On ReadData2?
2. How can we tell how many bits specify ReadRegister1? ReadRegister2?
3. How can we tell how many bits specify WriteRegister?
4. How can we tell how many bits specify WriteData?
5. How can we tell how many bits specify WriteEnable?
6. Why is WriteEnable necessary?
7. Why does the circuit not have ReadEnable inputs?
8. Why does the register file have a clock input? Why doesn't it use its own clock instead of requiring a clock from outside circuitry?

To assist you, here are some tips about sizing different bit widths:

1. Since the registers are 16 bits wide, and since ReadData1 will output the entire contents of a register, ReadData1 will be 16 bits wide. By the same reasoning, ReadData2 will be 16 bits wide.
2. Since in this lab there are 4 registers, then 2 bits are needed to "name" each register. The register having the name 00 is register 0, 01 is register 1, 10 is register 2, and 11 is register 3. ReadRegister1 and ReadRegister2 are therefore each 2 bits wide.
3. WriteRegister describes which register of the four registers is being written to. Therefore, WriteRegister is 2 bits wide.
4. WriteData specifies the data that will be written into a register. Since the registers are 16 bits, WriteData will be 16 bits.
5. WriteEnable says whether or not a register write should actually be performed (yes or no). Therefore, WriteEnable is 1 bit wide.

Here are some general tips to help you implement the register file:

- The data that should be written to a register must be able to be 'given' to *any* of the 4 registers in the register file (i.e., your circuit must be able to write to any register).
- Registers can be made read-only by setting one of their pins appropriately. This can let you control which register, if any, is written to. Make sure that the contents of the registers that are not being written to preserve their original values.
- Given multiple inputs, a multiplexer lets you select from exactly one of them. This component

is under the plexers category. The number of inputs from which a multiplexer chooses can be configured by changing its "select bits" property; e.g., the multiplexer's "Select Bits" property equaling 2 lets the multiplexer choose from 4 channels. The multiplexer's "Data Bits" property controls how many bits are on each of its inputs; e.g., if the MUX were choosing one of eight 32-bit inputs, Select Bits would equal 3 and Data Bits would be 32.

- A **decoder** has multiple output wires, but only one of those output wires is true at any given time. It is easy to control which output is true. Decoders are found under the plexers category. A **demultiplexer** is like a decoder in that it sends its input to exactly one of many outputs (the other outputs will be set to 0).
- **Pins** (whether input or output pins) can be made to support multiple bits at a time by changing their Data Bits property. You can use the poke tool to test out different values for an input pin (e.g., to simulate outside circuitry trying to read to different registers).
- You can poke a register and give it a value. This will let you give each register a different value to make sure that the circuitry reads from the correct register (according to ReadRegister1 and ReadRegister2).

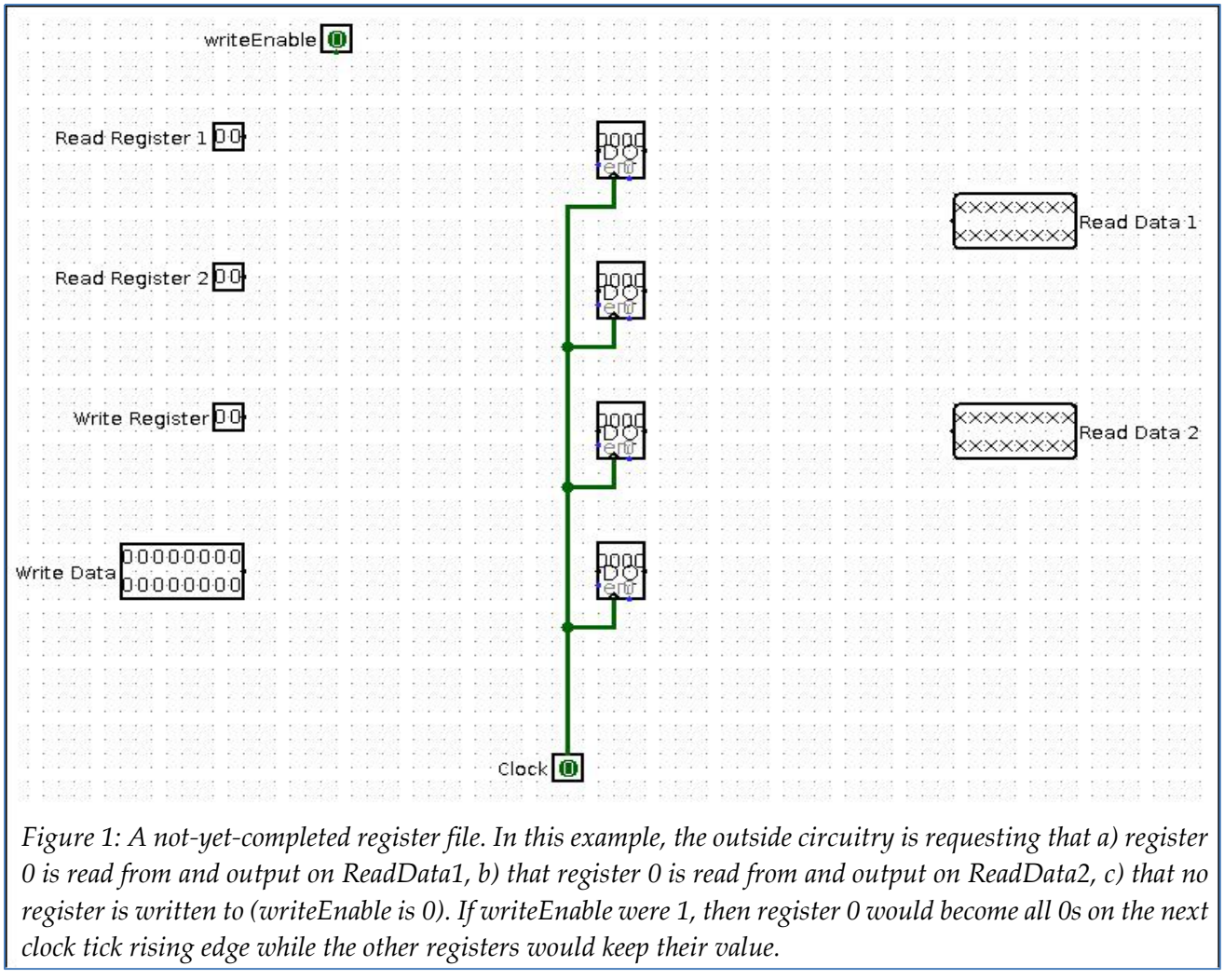


Figure 1: A not-yet-completed register file. In this example, the outside circuitry is requesting that a) register 0 is read from and output on ReadData1, b) that register 0 is read from and output on ReadData2, c) that no register is written to (writeEnable is 0). If writeEnable were 1, then register 0 would become all 0s on the next clock tick rising edge while the other registers would keep their value.

Part 2: Memory

Logisim provides a RAM component which can store up to 2^{24} values, each up to 32-bits wide. To store and load data from RAM, you have to specify an address, which is up to 24-bits wide. The RAM component provides either two separate ports for loads and stores or one single load/store port that is shared by reads and writes. For this problem, we will be using a RAM memory with 16 4-bit values and a single load/store port. You will have to configure the RAM component appropriately.

Consider the problem of setting every byte of the RAM to a specified value. For example, this might need to be done when a device is first powered on. We can set each byte of RAM to a predetermined value by having a counter generate every address of the memory and storing the predetermined value at each memory location. The component that looks like a triangle is called a Controlled Buffer. Its purpose is to put the value of its input at its output only when the controlling signal is high (1). When it is not high (0), the output is held floating, which means that another component can control the signal [2]. A controlled buffer lets a part of the circuit "disconnect" itself from part of the circuit.

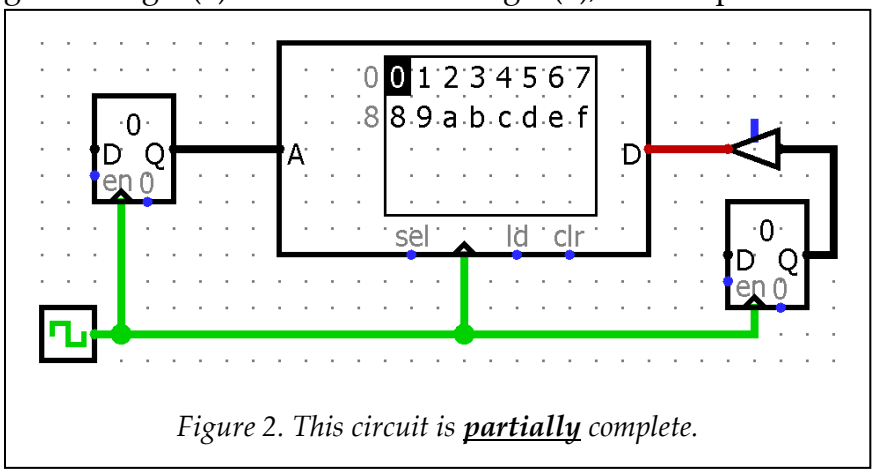


*Figure 2. This circuit is **partially** complete.*

Build a circuit in Logisim that writes the sequence of values 0x0 to 0xf to memory, as shown in Figure 2. The value 0x0 should first be written to the first memory location, then the value 0x1 to the second memory location, and so on. Your circuit should allow the user to reset the counters anytime (via a **button**). In addition, the circuit should stop writing values to memory after it has already written all memory locations exactly once. **The circuit shown in Figure 2 is not your final circuit**. You will have additional components and slightly changed components. **Save your final circuit file as lab10part2-<Your Pitt Mail ID>.circ (for example, lab10part2-xyz123.circ).**

Your task is not as easy as it initially sounds. Your circuit must detect when all values have been written to. It should then disable the RAM component and/or the controlled buffer, so that no more write 'commands' are sent to the RAM module and so that no more data is

[2] On more complicated circuits, multiple controlled buffers could work together so that, for example, one controlled buffer is always held high.

being placed on the data input of the RAM module. By no longer placing data on the RAM component, other circuits (which you do not have to build) could then use the RAM component.

Easily made mistakes for this part of the lab include these:

- Correctly detecting that the automated writes should stop, but still placing data into the RAM component and clocking it (e.g., having the circuit do the last write operation over and over again, forever)

- Writing to all spots in RAM except the last one, then stopping all future writes

- Writing to all spots in RAM, detecting that all spots have been written to, but then accidentally also rewriting to the very first spot in RAM one more time (or, writing to the last spot twice) before stopping all future writes

There are a several valid ways to have your circuit stop writing values. Pick a method or invent your own:

- Stop clocking the RAM component
- Force the RAM component to become read-only
- Use controlled buffers to disconnect both the counter's output (which determines the address to write) and the data to be written

Here are a few hints:

- RAM components must be set so that they accept writes or reads. Use the technical documentation to find out how to write to RAM
- Combinational logic cannot cause delays. If for some reason you need to introduce a delay, you will need to rely on sequential logic