Lab 08: Designing Simplified Parts of a CPU

Assigned: *Wednesday, November 27th, 2019*
Due: *Wednesday, December 4th, 2019*
Points: *100*

- You may collaborate on this homework with AT MOST one person, an optional "homework buddy".
- MAY ONLY BE TURNED ON **GRADESCOPE as a PDF file**.
- There is NO MAKEUP for missed assignments.
- We are strict about enforcing the LATE POLICY for all assignments (see syllabus).

Step by Step Instructions

For this week, you will sharpen your skills drawing circuits, and start to build bigger components from smaller ones. To this end, you will design some core components of a simple processor this week, broken up into three tasks. While each task can be completed independently of each other, they progressively build in difficulty, so it is recommended to complete them in order. Since this is a long and complex lab, working with a partner is recommended (optional). The tasks are listed below:

- **Task 1**: Design a Small ALU
- **Task 2**: Design a Small Register File
- **Task 3**: Design a Small Instruction Decoder and Executor

IMPORTANT! READ THIS!
This is a complex assignment. Read it CAREFULLY and DON'T SKIP any parts! There are several details to keep track of.

Each task will end up with a drawing – make it clear and readable!

You will take all 3 drawings – each one on one page by itself – and scan them all into ONE PDF file to submit on Gradescope (call it lab08.pdf). Make your drawings neat or you will lose points!

Make sure that your submitted PDF has, on top of the front page, the following information:

1. **Your name**
2. **Your email address**
3. **Your Perm ID number**
4. **Your Lab Section time**
5. **Name of your homework partner (if this applies)**

Even if you work with a partner, each of you needs to turn in their OWN submission.

Task 1: Design a Small ALU (30 pts)

For this task, you will design a simple single-bit ALU. The ALU takes the following inputs:

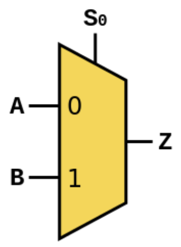
Input Name	Input Description
Operation	Which operation is to be performed, specified with a single bit. There are two possible operations: 1. XOR, specified when Operation = 0 2. NOR, specified when Operation = 1. Note that nor is equivalent to ~(A | B). Th at is, it is an OR operation whose result is negated.
A	The first operand, specified with a single bit
B	The second operand, specified with a single bit

The ALU returns the following single output:

Output Name	Output Description
ALUout	The result of whatever operator was chosen, applied to the given operands. For example, if XOR was chosen, then the result should be A ^ B.

To complete this task, you may use **only** the following components, in unlimited supply:

- AND, OR, XOR, and NOT gates, using the notation shown in lecture.
- 2-input multiplexers, which take a selector bit S0 and two single-bit input operands A and B, and return a single-bit output Z. They should be drawn using the symbol below (again, as used in lecture).



Task 2: Design a Small Register File (30 pts)

For this task, you will design a small register file. The register file contains two registers, each one bit long, which are named reg0 and reg1, respectively. The register file allows two registers to be read simultaneously, and also allows a register to be written to. The register file takes the following inputs:

Input Name	Input Description
R0	The first register to read, as a single bit. If 0, then reg0 should be read. If 1, then reg1 should be read.
R1	The second register to read, as a single bit. If 0, then reg0 should be read. If 1, then reg1 should be read.
WR	Short for "Write Register". Specifies which register to write to. If 0, then reg0 should be written to. If 1, then reg1 should be written to.
W	The data that should be written to the register specified by WR. This corresponds to a single bit.
WE	Short for "Write Enable". If 1, then we will write to a register. If 0, then we will **not** write to a register. Note that if WE = 0, then the inputs to WR and W are effectively ignored.

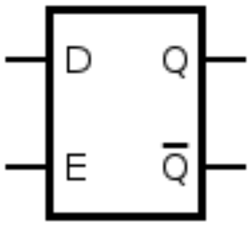
The register file returns the following two outputs:

Output Name	Output Description
O1	Value of the first register read. As described previously, this depends on which register was selected to be read, via R0.
O2	Value of the second register read. As described previously, this depends on which register was selected to be read, via R1.

To complete this task, you may use **only** the following components, in unlimited supply:

- AND, OR, XOR, and NOT gates, as well as 2-input multiplexers, using the notation shown in lecture. These are the same gates as what you used in **Task 1**.

- Gated D-latches, which take a single-bit input D and an "enable" bit E, which is used to indicate that the value of D should be saved within. They produce single-bit outputs Q and !Q, which represent the value stored in the latch along with its negation, respectively. They should be drawn using the symbol below (as used in lecture):



Task 3: Design a Small Instruction Decoder and Executor (40 pts)

This is the largest task of the three. For this task, you will design a small instruction decoder and executor, which is at the heart of a processor. This component will use the components you designed in **Tasks 1 and 2**, in an abstract manner. That is, instead of needing to rewrite your definitions from Tasks 1 and 2 for this, we'll provide you boxes that perform these operations. You will ONLY use 1 box from Task 1 and 1 box from Task 2. The processor instructions you will be working with are described below.

Instructions are uniformly **encoded with 5 bits**, and there are **4 instructions in total**: **xor**, **nor**, **load**, and **store**.

The first 2 bits of the instructions encode the ***opcode***, which is used to differentiate between instruction types. The next 3 bits encode which registers the instruction operates on, or a memory address, depending on the particular instruction. Registers are specified using **a single bit**, and memory addresses are specified with **2 bits**. A table of all the possible instructions is below.

Instruction Name	Opcode	First Operand	Second Operand	Third Operand	Description
xor	00	REG0: Register where the result should be placed, specified with a single bit	REG1: Register specifying where the first operand is, specified with a single bit	REG2: Register specifying where the second operand is, specified with a single bit	XORs the contents of REG1 and REG2 together, putting the results in REG0. This should involve the ALU. In more terse notation, this calculates: REG0 = REG1 ^ REG2
nor	01	REG0: Register where the result should be placed, specified with a single bit	REG1: Register specifying where the first operand is, specified with a single bit	REG2: Register specifying where the second operand is, specified with a single bit	NORs the contents of REG1 and REG2 together, putting the results in REG0. This should involve the ALU. In more terse notation, this calculates: REG0 = ~(REG1 | REG2)
load	10	REG: Register where the result should be placed, specified with a single bit	A: A memory address, specified with two bits	N/A	This loads in a single bit of memory located in address A, and loads its value into register REG. In our setup, each cell in memory only holds a single bit (as opposed to a byte), and we can address individual bits. For example, memory address 00 would refer to the first bit in memory, 01 would refer to the second bit in memory, and so on.

store	11	REG: Register containing the value to store into memory, specified with a single bit	A: A memory address, specified with two bits	N/A	This copies the value stored in REG to the memory slot located at address A. In our setup, each cell in memory only holds a single bit (as opposed to a byte), and we can address individual bits. For example, memory address 00 would refer to the first bit in memory, 01 would refer to the second bit in memory, and so on.

Instructions are encoded with the 2 opcode bits on the far left, with the remaining 3 bits following. For example, consider the following instruction:

```
01110
```

The leftmost 2 bits are 01 which refers to the opcode for nor. As such, this is a nor instruction. The next 3 bits are 110. Going from left to right, along with the definition of our nor instruction, this means that:

- REG0 = 1
- REG1 = 1
- REG2 = 0

Taking all this information together, this means to first get the values stored in registers reg1 and reg0 (because REG1 = 1 and REG2 = 0). These values are then NORed together, and then the result is stored in register reg1 (because REG0 = 1).

As another example, consider the following instruction:

```
11011
```

The leftmost 2 bits are 11, which refers to the opcode for store. As such, this is a store instruction. The next 3 bits are 011. Going from left to right, along with the definition of our store instruction, this means that:

- REG = 0
- A = 11

Taking all this information together, this means to first get the value stored in register reg0 (because REG = 0). This value should then be put into the single-bit memory slot stored at address 11 (because A = 11).

The instruction decoder / executor takes in a single instruction to work with, specified with five bits, as our instructions are five bits long. Each of these bits corresponds to a distinct input, each of which is described below:

Input Name	**Input Description**
OP0	Bit 0 of the opcode. Recall that bit 0 refers to the rightmost bit, so for opcode 01, this refers to the bit with value 1.
OP1	Bit 1 of the opcode. For example, for opcode 01, this refers to the bit with value 0.
B0	The first non-opcode bit of the instruction. For example, with instruction 00100, this refers to the bit with value 1.
B1	The second non-opcode bit of the instruction. For example, with instruction 00010, this refers to the bit with value 1.
B2	The third non-opcode bit of the instruction, which is the last bit of the instruction. For example, with instruction 00001, this refers to the bit with value 1.

Another view of the above information is provided in the table below, which shows where the different inputs bits in an instruction are located:

OP1	OP0	B0	B1	B2

Some more examples are shown below:

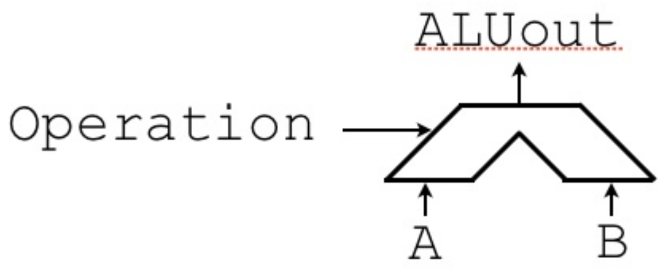
OP1	**OP0**	**B0**	**B1**	**B2**	**Human-readable Encoding**	**Description**
0	0	0	0	0	xor reg0, reg0, reg0	Compute the XOR of the contents of reg0 with the contents of reg0, storing the result in reg0
0	1	1	0	1	nor reg1, reg0, reg1	Compute the NOR of the contents of reg0 with the contents of reg1, storing the result in reg1
1	0	1	0	1	load reg1, 01	Copy the bit stored at address 01 (decimal 1) into register reg1
1	1	0	1	0	store reg0, 10	Store the contents of reg0 at address 10 (decimal 2)
1	1	1	1	1	store reg1, 11	Store the contents of reg1 at address 11 (decimal 3)

The instruction decoder / executor does not produce any outputs. This makes sense, as the effect of running a single instruction is reflected in changes made to the register file and to memory.

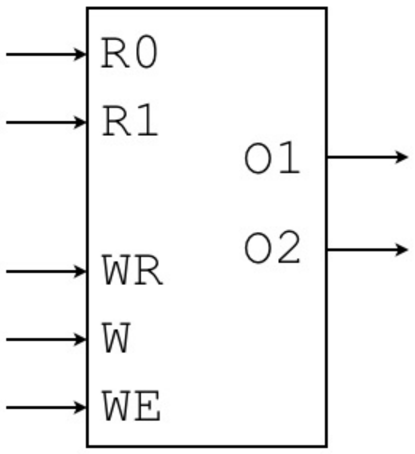
To complete this task, you may use **only** the following components, in unlimited supply:

- AND, OR, XOR, and NOT gates, as well as 2-input multiplexers, using the notation shown in lecture. These are the same gates as what you used in **Task 1**.

- An abstract component, representing the ALU you designed in **Task 1**. In should be drawn using the symbol below:

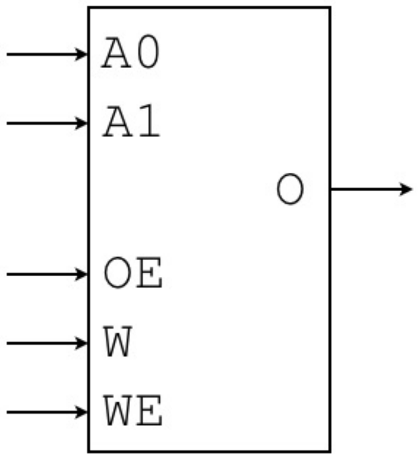


- An abstract component, representing the register file you design in **Task 2**. It should be drawn using the symbol below:



- An abstract component, representing the interface for the memory chip. The memory chip contains 4 bits of memory, which are individually addressed.
  This component has the following inputs:
  - A0: Bit 0 of the address. Recall that bit 0 refers to the rightmost bit, so for address 01, this refers to the bit with value 1.
  - A1: Bit 1 of the address. For example, with address 01, this refers to the bit with value 0.
  - OE: Short for "Output Enable". If 1, then the value at the address specified by A0 and A1 will be read, and sent to the output line O. If 0, then the memory will not be accessed, and the value sent to the output line is unspecified (could be either 0 or 1, in an unpredictable fashion).
  - W: The value to write to memory
  - WE: Short for "Write Enable". If 1, then the value sent into W will be written to memory at the address specified by A0 and A1. If 0, then no memory write occurs (the value sent to W will be ignored).

The output of the component is O, which refers to the value read from memory (or unspecified if OE = 0).
The memory component should be drawn using the symbol below:



Note that D latches are **not** a component you may introduce for this task. While these are needed to implement the memory file and the memory interface, these aren't needed at the higher level of abstraction we are working with in **Task 3**. You are forbidden to use them only to prevent confusing situations where D latches are added needlessly.

Implementation Hints for Task 3

Rather than implementing **Task 3** all at once, it is ***strongly recommended*** to take the following incremental approach:

1. Design a version that assumes the instruction is either xor or nor. This version would do something broken on load or store instructions.
2. Design a version that assumes the instruction is load. This version would do something broken on any other instruction.
3. Design a version that assumes the instruction is store. This version would do something broken on any other instruction.
4. Finally, stitch the above three solutions together, introducing additional gates and multiplexers to only trigger the correct portions of the circuit depending on whatever the whole opcode is. This should be fairly mechanical in nature.