Building a processor from scratch

In this assignment we are going build a computer processor from the ground up, starting with transistors, and ending with a small but powerful processor. The purpose of this assignment is to make you more familiar with the notion of logical circuits and the boolean functions which they implement. Starting with very simple logic gates, we will work our way up to a complete processor. At each level of the processor design, we will use the circuits that we have designed at the lower levels as black boxes, ignoring their implementation details. Last but not least, *understanding the internal workings of a simple processor is a key piece of becoming a computer scientist.*

To aid in the clarity of your solution as well as the clarity of your thinking, *you should design your circuits modularly.* Once you have shown how to build a piece of circuitry (such as an adder) it is never again necessary to draw the details of its implementation: simply use an appropriately labelled box to represent it. Similarly, you may wish to bundle parallel bunches of wires together into a single appropriately labelled "n-bit wide" wire. With these techniques, you should be able to describe very complicated circuits with very simple and elegant circuit diagrams.

We encourage you to go through the **Guide to Drawing Circuits** (available on the course website) before you begin answering the questions. This assignment requires substantial drawing, you still must submit your solution electronically in PDF format as usual. Circuit diagrams are well-supported in both Microsoft Word and the X Windows program `xfig`.

The following facts cannot be stressed too much:

> **Start Early!** This assignment requires **lots** of drawing, which is quite time consuming. Come to office hours if you need; your TAs are ready to help.

> **Think modularly!** Remember, modularity is your friend.

Warmup: Building an Adder

Problem 1 (3 Points):

Give a complete schematic for a 4-bit adder. Using any combination of AND, OR, NOT, XOR, and NAND gates, draw a circuit that implements 4-bit grade school addition, which should be able to do calculations like $0110 + 0100 = 1010$. You do not need to deal with overflow (for example, $0110 + 1100 = 0010$ is OK).

Multiplexers and Demultiplexers

Two types of circuits will come in very handy as we design our processor: *multiplexers* and *demultiplexers*. Intuitively, a multiplexer (mux) takes a binary number n and selects the nth one of its inputs to forward on. A demultiplexer (demux) takes a number n and sends its input out on the nth output line.
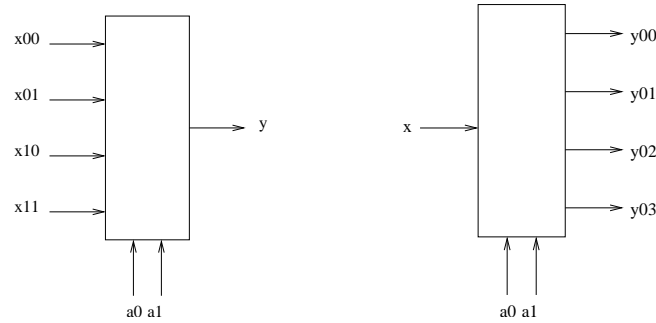

Figure 1: At left, a four-way one-bit mux. At right, a four-way one-bit demux.

The schematic for a 4-way 1-bit mux is shown in Figure 1 (Left).
It has four data inputs, $x_{00}, x_{01}, x_{10}, x_{11}$, two control inputs a_0, a_1, and one output, y. The value of y is equal to the input specified by the binary number $a_0 a_1$, i.e., $y = x_{a_0 a_1}$. As an example, if $x_{00} = 1, x_{01} = 1, x_{10} = 0, x_{11} = 1$ and $a_0 a_1 = 10$, then $y = x_{10} = 0$. A mux is also called a *selector* because the control lines select from among the possible inputs.

The schematic for a 4-way 1-bit demux is shown in Figure 1 (Right). This circuit has a 1-bit data input x, a 2-bit control input a_0, a_1, and 4 1-bit outputs, $y_{00}, y_{01}, y_{10}, y_{11}$.

A demux works as follows. The value of x is transferred to the output line specified by a_0 and a_1, i.e., $y_{a_0 a_1} = x$. The value on all of the other outputs is 0. As an example, if $x = 1$, and $a_0 a_1 = 01$, then $y_{01} = 1$ and $y_{00} = y_{10} = y_{11} = 0$. As another example, if $x = 0$, then independent of the value of $a_0 a_1$, $y_{00} = y_{01} = y_{10} = y_{11} = 0$.

Problem 2a (10 points):

Using the gates that you've designed so far, design a 4-way 1-bit multiplexer.

Problem 2b (10 points):

Using the gates that you've designed so far, design a 4-way 1-bit demultiplexer.

Problem 2c (4 points):

A k-way b-bit mux is one in which each of the k-inputs is a group of b bits rather than a single bit, and the output is a group of b-bits. There are also $\log_2 k$ control bits (let's assume k is a power of 2), and these control bits are used to select one of the k groups of b bits rather than a single bit. A k-way b-bit demultiplexer can be defined in a similar fashion. *Explain* how to construct a k-way b-bit multiplexer using b k-way 1-bit multiplexers (you do not actually need to draw a circuit diagram).

Clocked circuits

The circuits that we have constructed so far use what is called *combinational logic*. What this means is that whenever any of the inputs to a circuit change, that change will trickle through the gates of the circuit all the way to the outputs of the circuit without being held up. This is good for things like adders where we want to get the answer out the other end as quickly as possible, but for other things it can be useful to be able to control when circuits change their values. In the next part of the assignment, we will look at circuits where the flow of data can be regulated through the use of *clocks*.

high

low
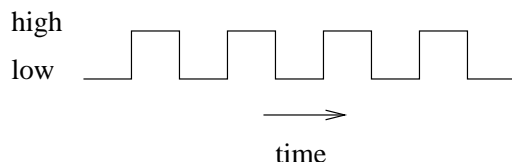
time

A clock is a periodic oscillating signal, as above. The high parts indicate a value of 1, and the low parts a value of 0. These days clocks are very fast, and can oscillate at speeds more than 2 GHz (2 billion times per second).

Here's how the clocked NOT gate works. When *clock* goes high, the value at x is negated and (if necessary) carried over to z. Even if x changes, the value at z will not change until *clock* again pulses.

The schematic below indicates a clocked not gate.

clocks

x   z
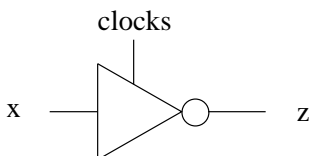
Notice that a clock signal is not attached to the normal input of a gate. It should *only* be attached to the clock input of the gate as in the diagram above. We can similarly construct clocked versions of the other logic gates.

We view the two clock transition as a *pulse* or *strobe*. The input x can change anytime before the clock pulse arrives without affecting the value of the output z. When the pulse occurs, the value at x is sampled, and shortly thereafter, z takes on its new value (the opposite of the value that x had before the pulse occurred, because this is a NOT gate). Notice that a clocked NOT gate behaves differently from an unclocked NOT gate. When the input to an unclocked NOT gate changes, the output will change soon afterwards, whether or not a pulse occurs.

You might be wondering why we have introduced these clocks. The purpose of the clocks is to break the computation performed by the circuit into discrete points in time. You can think of the circuit as being in a stable state between clock pulses, i.e., the output of every gate holds steady, as does the input to every gate. The value of the output of every clocked gate, however, is determined by the inputs to the gate *before* the most recent clock pulse. When another pulse occurs, the output of each gate takes on a new value based on the input values that were present before the pulse.

Sequential Circuits and Feedback

As you have seen before, *combinational* circuits simply compute the value of their output based on their input value at that instant. Such circuits have no "memory". The output is computed based on the current inputs, and anything that happened to the circuit in the *past* has no effect. If the input signals are the same at two different times, the outputs at those times will be the same.

We will now design circuits with memory. These are called *sequential* circuits. The output of these circuits depends on the current as well as past inputs. Memory circuits contain *feedback loops*. A feedback loop occurs when the output from a gate is connected back to an input of the same gate—possibly through one or more other gates. Such a loop allows previous inputs to affect current outputs, which is exactly what we want in a memory circuit.

As an example, consider the following sequential circuit :
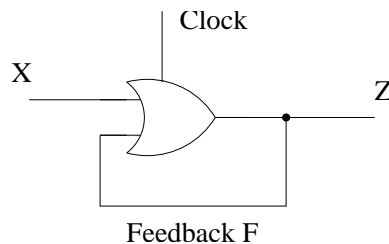

Feedback F

Suppose that the feedback wire is initially set to 0. What would the output be in terms of the input? To see this, let us compute the output value at specific time instants, specifically the time instants at which the clock goes high.

If the value of the output at instant t is termed as z_t, then, the value on the feedback wire signifies the output value at the previous instant, which is z_{t-1}. Hence, the output of this gate is given by $z_t = x_t$ OR z_{t-1}, where x_t is the input at time t (that is, just before the clock goes high).

As long as the feedback wire is set to 0, we get $z_1 = x_1$ (refer to the truth table for OR gate). However, when the feedback wire becomes 1, the output also becomes 1, and will stay 1, irrespective of what the input value is. Hence, when the input wire becomes 1, the output stays at 1 for all times after that. It is not hard to see that the input is acting like a *turn-on* signal. The circuit has memorized that at some time instant in the past, the value of the input was 1.

Problem 3 (5 points):

Now consider the following circuit which is a minor extension of the above.
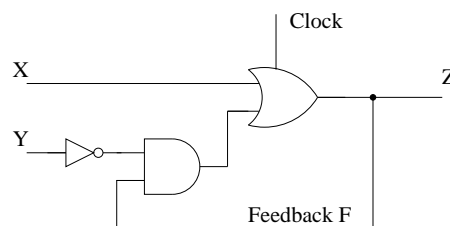
Having seen the above analysis, you can now perform a similar analysis for this circuit. What role does the second input play? Briefly describe the behavior of this circuit.

Registers

In modern processors, values that are to be used repeatedly in a calculation are stored in structures called *registers*. A schematic for a 4-bit register is shown below.
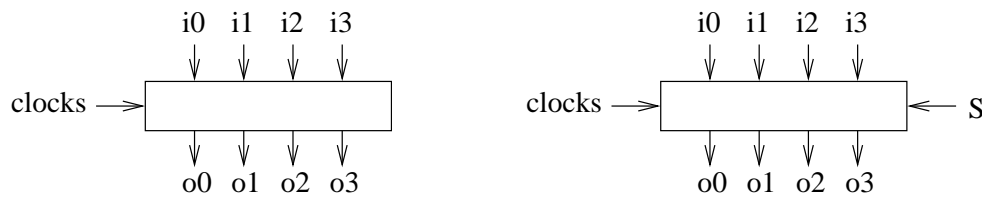

Figure 2: At left, a 4-bit buffer for problem 5b. At right, a 4-bit register for problem 5c.

Problem 4a (1 point):

Using clocked NOT gates, design a clocked YES gate, also known as a *buffer*. The output value should be what the input value was at the last clock pulse.

Problem 4b (3 points):

Using a combination of clocked and unclocked gates, show how to design a 4-bit buffer. The buffer should accept a 4-bit value on its inputs i_0, i_1, i_2, i_3 before a clock pulse, and it should put that value on its outputs o_0, o_1, o_2, o_3 after the pulse.

Problem 4c (8 points):

Notice that the buffer from the previous problem only delays values for one clock cycle. A better design would allow you to store values in the buffer as long as you wanted. This is called a register. Create a 4-bit register by extending the buffer circuit from above so that it accepts an extra input S (for Set). If S is 1, then the new circuit should behave exactly as above when the clock pulses. However, if S is 0 when the clock pulses then the register should ignore its inputs, keeping the same outputs as before the clock pulse.

Processor Design

We are now ready to complete our processor design. The processor will have the following components.

1. 8 32-bit registers, $R_0, R_1, \ldots, R_7$ for holding data

2. 256 16-bit registers $P_0, P_1, \ldots, P_{255}$ for holding a program

3. a 32-bit adder

4. an 8-bit register PC that will serve as a program counter

It will turn out to be convenient to hard-wire register R_0 to the value 0, and register R_1 to 1.

The processor will be very simple. It will have only five types of instructions: add, negate, load immediate, and jump if zero. (This is a Really Reduced Instruction Set Computer.) A program consists of a sequence of instructions stored in the 256 program registers. Each of these program registers is a 16-bit register. The 16 bits specify the type of instruction and its operands. The first two bits of the instruction specify the instruction type. The remaining bits are operands to the instruction. The formats of the instructions are shown below.

The add (*add*) instruction adds the contents of the registers specified in the R_a and R_b fields. Each of these fields consists of three bits, which can be thought of as the *index* of the register R_a or R_b. For example, if the R_a field contains 010, then, since $a = 010 = 2$, the index value in the R_a field specifies R_2. The result of the addition is placed into R_c. You do not need to deal with overflow. This instruction, as well as the negate and load immediate instructions, also increments the program counter PC by 1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Rc = Ra+Rb; PC = PC+1;	0	0	Ra			Rb			Rc			0	0	0	0	0

The negate (*neg*) instruction replaces R_a with $-R_a$, using the two's complement representation. *If you are unfamiliar with two's complement representation, see the notes at the end of this assignment.* Again, you do not need to deal with overflow.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ra = −Ra; PC = PC+1;	0	1	Ra			0	0	0	0	0	0	0	0	0	0	0

The load immediate instruction (*loi*) places the 8-bit value d in the low order 8-bits of register R_a, and sets the remaining 24 high-order bits of R_a to 0.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ra = d; PC = PC+1;	1	0	Ra			0	0	0	d							

The jump if zero (*jz*) instruction changes the program counter PC to the value specified by d if the value in register R_a is 0, and otherwise it increments the program counter by 1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
if (Ra == 0) PC = d; else PC = PC+1;	1	1	Ra			0	0	0	d							

The processor will be clocked. The value of PC when a clock pulse occurs specifies which of the 256 registers $P_0, \ldots, P_{255}$ holds the instruction that should be executed. The operation specified by the instruction is performed, which may include both an operation on data held in the registers, as well as a change in the program counter. The initial state of PC can be assumed to be 0.

Building Operators

The action of our processor will be built around 4 operator units (one for each instruction). Each operator unit will have different inputs, but they will all have *the same four outputs*:

1. A 1-bit *jump* flag J that indicates whether or not the program counter should jump.

2. A 32-bit output value V that is the value that the module wishes to have saved in some register.

3. A 3-bit index value I that indicates which register to save V into.

4. A 1-bit *enable* flag E, which indicates whether or not V should be saved into register I, or if V can just be discarded.

Some of these outputs will be somewhat redundant, for instance the *add* operator unit should always set J to 0.

Problem 5a (6 points):

Give a complete schematic for the addition operator unit. It should have three inputs, a 32-bit value (assumed to have *already been loaded* from R_a), a 32-bit value from R_b, and an 8-bit wire with the rest of the instruction. The addition operator unit should have the outputs discussed above.

Problem 5b (10 points):

Give a complete schematic for each of the other operator units (neg, jz, and loi). Your outputs must be (J, V, I, E) as given above. Design your inputs to closely resemble those of the addition operator unit as much as possible. Even if it involves ignoring some input, this will simplify later work.

Problem 6 (30 points):

Draw a diagram of the entire processor. You may use any of the circuits that we have designed so far, including gates, adders, multiplexers, demultiplexers, and registers. The adders, multiplexers, demultiplexers, and registers can be of any size and number. You may also design and use your own circuits if you so wish. Finally you may use the constant values 0 and 1 as the inputs to any of your circuits, if that is convenient.

Here is some advice. Make your design as simple as possible, and don't worry too much about trying to minimize the number or the size of the circuits in your diagram.

Strive for modularity by building simple units and putting them together.

Some possibilities for modules (not necessarily required) include:

- a *register file unit* that encapsulates the functionality of $R_0 \ldots R_7$, selecting registers for reading/writing/etc.

- an *instruction file unit* that stores the program

- a *program counter unit*

Depending on your design, you may desire other modules.

For sake of simplicity, you can assume that registers $P_0, \ldots, P_{255}$ already contain the instructions to be run, and their contents do not need to be changed. They should be treated as *read-only* registers (with Set always set to 0). Also, don't actually draw 256 registers—just indicate where they go, and how they are connected to other circuits.

Problem 7 (10 points):

Write a program for this processor to perform a multiplication $x \cdot y$. Assume that x is stored in register R_2, y is in R_3, and the final answer is to be placed in register R_4. Don't worry about overflows. Since we do not have an end or halt instruction, you should end your program by jumping to instruction number 251. (Hint: Don't write a recursive program!)

You do not need to write your program out in binary (in fact, please don't!) You should write out the sequence of instructions giving their names and their operands, with a label next to each instruction indicating which instruction register it occupies. For example, the following program puts 12 in register 2 and then repeatedly adds -1 to it until it reaches 0, whereupon it exits.

$$
\begin{array}{ll}
0 & loi\ 1 \rightarrow R_3 \\
1 & neg\ R_3 \\
2 & loi\ 12 \rightarrow R_2 \\
3 & add\ R_2, R_3 \rightarrow R_2 \\
4 & jzr\ R_2 \rightarrow 251 \\
5 & jzr\ R_0 \rightarrow 3
\end{array}
$$

Remarks on Two's Complement Representation:

Two's complement is a method to encode signed integers. The leading (or leftmost) bit is called the *sign bit*. If the sign bit is 0, then the integer represented is non-negative. For example, the string 00100 represents the integer 4.

If the sign bit is 1, the integer represented is negative. To obtain the absolute value of a negative integer represented in two's complement, we first invert all the bits, and then add one to the result. For example, the string 10011 represents a negative integer (since the leading bit is 1). To see what the magnitude of this integer is, we first invert all the bits and get 01100, to which we add one to get 01101. Hence, the string 10011 represents the integer -13.

We can use the same procedure for converting a negative integer into its two's complement representation. To represent the integer -13, we start with 01101, the binary representation of 13. We first invert all the bits to get 10010, to which we add one to obtain 10011, the two's complement representation of -13.

In order to test your understanding of two's complement, ask yourself the following questions: (a) what is the largest (positive) number that can be represented by n bits in two's complement? (b) What is the smallest (negative) number that can be represented by n bits in two's complement?