10 Powerful Ideas	Bruce Maggs
Assignment 6	Due: **Tuesday** March 3, 1998

1 Building a processor from scratch

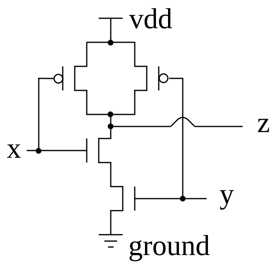
In this assignment we are going build a computer processor from the ground up, starting with transistors, and ending with a small but powerful processor. The purpose of this assignment is to make you more familiar with the notion of logical circuits, which we will revisit later in the semester. It also illustrates the principle of modularity. Starting with very simple transistor-level circuits, 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.

2 Gates

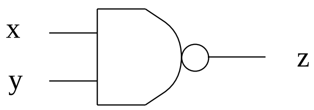
Our basic building block will be a two-input CMOS NAND gate. The truth table for a two-input NAND gate is shown below. The output z is 1 if and only if at least one of the two inputs x and y is 0. (Which is the same as saying that the two outputs are not both 1.)

x	y	$z = \overline{x \cdot y}$
0	0	1
0	1	1
1	0	1
1	1	0

A NAND gate can be implemented in CMOS using four transistors, as shown below. You don't have to understand how this works, but if you're curious, the basic idea is this. The horizontal line at the very top of the figure represents a high voltage value (vdd), say 5 volts, while the three horizontal lines at the bottom of the figure represent the ground voltage, 0 volts. The high voltage value represents a binary 1, while the low voltage value represents 0. The inputs x and y are each connected to the gates of two transistors. A transistor is basically a door. Each of the bottom two transistors in this figure is "open" when the voltage on its gate is high, and "closed" when it is low. Each of the top transistors is open when the voltage on its gate is low, and closed when it is high. When either of x or y is 0, the path from the ground to the output z is closed, but at least one path from vdd to z is open, so z takes on the high voltage. When both x and y are one, there is no path from vdd to z, but there is a path from the ground to z, so z takes on the low voltage.



A NAND gate is drawn schematically as shown below.

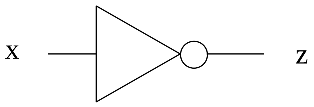


Problem 1 (10 points):

Using a NAND gate as your basic building block, show how to construct circuits that perform the following functions: NOT, AND, OR, and XOR. You should not draw any transistors – begin with only the NAND gate schematic. You may design the circuits in any order that you want, and once you have designed a circuit, you may use it in the construction of other circuits (using the appropriate schematic). The truth tables for these functions, along their schematics are shown below.

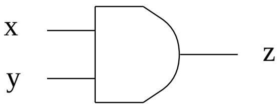
1. NOT gate

x	$z = \overline{x}$
0	1
0	1
1	0
1	0



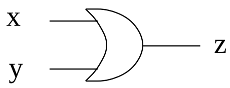
2. AND gate

x	y	$z = x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1



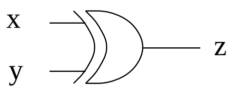
3. OR gate

x	y	$z = x + y$
0	0	0
0	1	1
1	0	1
1	1	1



4. XOR gate

x	y	$z = x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



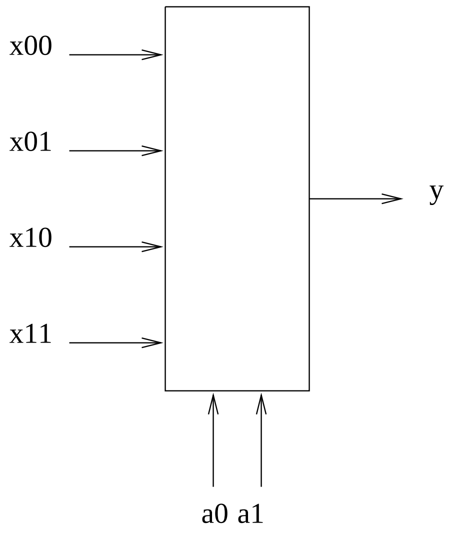
3 Building an adder

Problem 2 (10 points): Now that you have designed a collection of useful gates, you can use them to construct more sophisticated circuits. Using your collection of gates, draw a schematic for a 4-bit ripple-carry adder, as described in Lecture 8. Show all of the gates that you use.

4 Multiplexors and demultiplexors

Two types of circuits will come in very handy as we design our processor: *multiplexors* and *demultiplexors*.

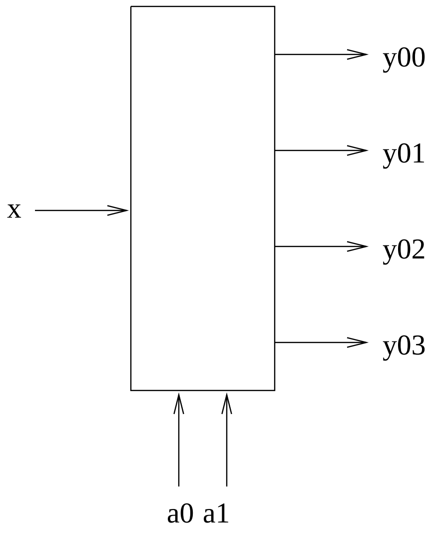
The schematic for a 4-way 1-bit multiplexor is shown below.



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 $a_0 a_1$, 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 multiplexor is also called a *selector* because the control lines select from among the possible inputs.

Problem 3 (10 points): Using the gates that you've designed so far, design a 4-bit multiplexor.

The function of a demultiplexor is opposite that of a multiplexor. The schematic for a 4-way 1-bit demultiplexor is shown below. 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 demultiplexor 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 $a_0 a_1$, $y_{00} = y_{01} = y_{10} = y_{11} = 0$.

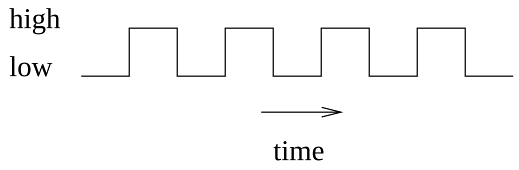
Problem 4 (10 points): Using the gates that you've designed so far, design a 4-bit demultiplexor.

Problem 5 (10 points): A k-way b-bit multiplexor 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 demultiplexor can be defined in a similar fashion. Explain how to construct a k-way b-bit multiplexor using b k-way 1-bit multiplexors.

5 Clocked circuits

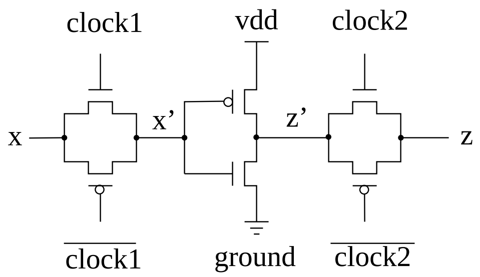
The types of circuits that we have constructed so far are called *combinational logic*. What this means is that whenever any of the inputs to a circuit change, that change will trickle through the gates of the circuit all the way to the outputs of the circuit without being held up. In the next part of the assignment, we'll look at circuits where the flow of data can be regulated by through the use of *clocks*.

A clock is a periodic oscillating signal, as shown below.

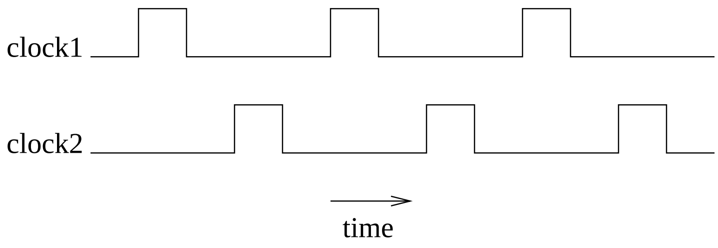


These days clocks are very fast, and can oscillate at speeds up to 1 Ghz (1 billion times per second)!

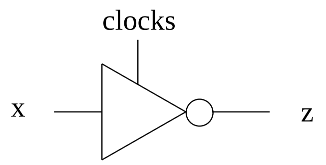
A clock is used in a processor to regulate the order in which calculations are performed. The circuit shown below is a one-bit clocked NOT gate.



The clocked NOT gate actually uses two clocks, whose signals are interleaved as shown below. (Real processors typically have even more than two clock signals.)



Here's how the clocked NOT gate works. (If you find transistors frightening, don't worry – you don't really have to understand this circuit diagram, to use a clocked NOT gate in the rest of this assignment.) When clock1 goes high, the value at x is carried over to x'. This causes the output z' to change (if necessary) to the new value, $\overline{x'}$. When clock2 goes high, the new value of z' is carried over to z. You can simply draw a schematic like the one below to indicate that the NOT gate is clocked.

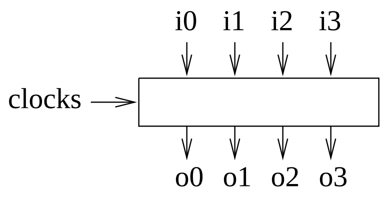


This clocked NOT gate has the following functionality. We will think of a clock *pulse* as consisting first of a low-high-low transition by clock1 and then a low-high-low transition of clock2. The input x arrives before a clock pulse. There is no change in z yet. Then the pulse occurs, after which z takes on its new value, the opposite of the value that x had before the pulse occurred. 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.

6 Registers

In a processor like the Pentium, 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.



Problem 6 (10 points): Using a combination of clocked and unclocked NOT gates, show how to design a 4-bit register. The register 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.

7 Putting it all together

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 turns 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 four 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 registers holds 16 bits. The 16 bits specify the type of instruction and its operands. The first two bits of each instruction specify the type. The formats of the instructions are shown below.

The 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 010 is the binary representation of 2, the value in the R_a specifies R_2. The result of the addition is placed into R_c. 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 instruction replaces R_a with $-R_a$.

	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 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 instruction changes the program counter PC to the value specified by d if $R_a = 0$, 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 operations specified by the instruction is performed, which may include both an operation on data held in the registers, as well as a chance in the program counter.

Problem 7 (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, multiplexors, demultiplexors, and registers. The adders, multiplexors, demultiplexors, and registers can be of any size and number. You may also 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. You should be able to design the processor using at most two or three adders. The program counter PC should serve as the control inputs to a multiplexor, as should each of the fields R_a and R_b of the current instruction. The registers $R_0, \ldots, R_7$ should be the data inputs to one or more multiplexors, as should $P_0, \ldots, P_{255}$. Also, don't actually draw 256 registers – just indicate where they go, and how they are connected to other circuits.

Problem 8 (10 points): Write a program for this processor to compute F_n, the nth Fibonnaci number. Assume that n is stored in register R_2, and that the final answer is to be placed in register R_3. Don't worry about overflows. Hint: Don't write a recursive program!