Read-only memory (ROM) using combinational logic circuits

The truth tables are defined by "input variables" and "output variables", and we have been thinking of them as evaluating logical expressions. Another way to think of a combinational circuit is as a *Read Only Memory* (ROM). The inputs encode a memory address. The outputs encode the value stored at the address. We say such a memory is read-only because the gates of the circuit are fixed.

For example, suppose the memory address is specified by the two input variables A_1, A_0, and the contents at each address are specified by the bits $Y_2Y_1Y_0$. Here is a truth table:

A_1	A_0	Y_2	Y_1	Y_0
0	0	0	1	1
0	1	0	0	1
1	0	0	0	0
1	1	1	0	0

and here is the corresponding memory. (Note that the address is not stored in the memory!)

input (address)	output (contents of memory)
00	011
01	001
10	000
11	100

What is new about this concept? You now have to think of the input and output variables in a different way than you did before. Before, you thought of the input variables as having TRUE or FALSE values, and you did not associate any particular significance to their order. Thinking about the circuit as a ROM is quite different. You no longer think of each of the input variables as TRUE or FALSE. Rather, you think of the input variables as defining a binary number, namely an address. The order of the input variables now has a significance, since it defines an ordering on the address space. Similarly, the output variables are a string of bits whose order may or may not matter.

Arithmetic Circuits

Last class we discussed gates and circuits and how they could be used to compute the values of logical expressions, formed by NOT, AND, OR and other operators. Such circuits are called *combinational logic circuits* or *combinational digital circuits*. We will look at several more useful examples today.

How would we implement addition using these gates? Recall the algorithm from Lecture 1 for adding two n-bit numbers, A and B :

$$\begin{array}{cc} & C_{n-1}\ C_{n-2}\ \ldots\ C_2\ C_1\ C_0 \\ & A_{n-1}\ A_{n-2}\ \ldots\ A_2\ A_1\ A_0 \\ + & B_{n-1}\ B_{n-2}\ \ldots\ B_2\ B_1\ B_0 \\ & \hline \end{array}$$

$$S_{n-1}\ \ S_{n-2}\ \ \ldots\ \ S_2\ \ S_1\ S_0$$

The C_i are the carry bits, and S_i are the sum bits. Note that $C_0 = 0$ because there is no way to carry in. (It will be clear later why we are defining this bit.)

How are we interpreting these bits? We are doing "odometer arithmetic" here, so it doesn't matter whether we are thinking of the numbers as signed (twos complement) or unsigned. This does matter when we interpret the results, but it doesn't matter from the standpoint of the mechanisms for computing the sum.

The rightmost bit (or *least significant bit*) S_0 of the sum is simple to determine using logical expressions and combinational circuits, as is the carry out bit C_1.

A_0	B_0	S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Thus,

$$S_0 = A_0 \oplus B_0$$

$$C_1 = A_0 \cdot B_0$$

The circuit that implements these is called a *half adder*. It has two inputs and two outputs. You should be able to draw this circuit yourself. More generally, the values of the k^{th} sum and carry out bits are given as follows:

C_k	A_k	B_k	S_k	C_{k+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

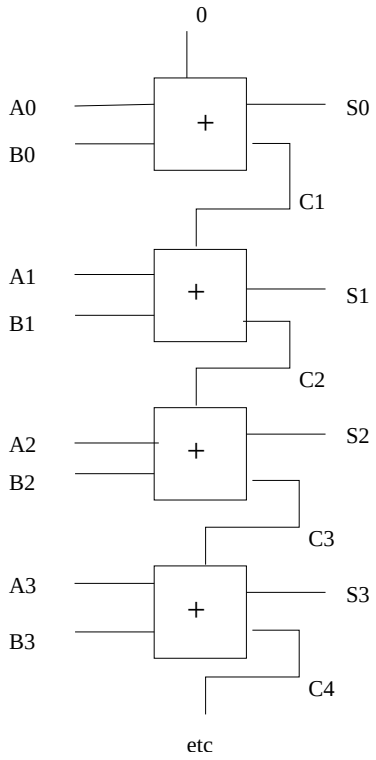
which we can represent using sum-of-products circuits as defined by these expressions:

$$S_k = \overline{A_k} \cdot B_k \cdot \overline{C_k} + A_k \cdot \overline{B_k} \cdot \overline{C_k} + \overline{A_k} \cdot \overline{B_k} \cdot C_k + A_k \cdot B_k \cdot C_k$$

$$C_{k+1} = A_k \cdot B_k \cdot \overline{C_k} + \overline{A_k} \cdot B_k \cdot C_k + A_k \cdot \overline{B_k} \cdot C_k + A_k \cdot B_k \cdot C_k$$

The box with the '+' below contains the combinational circuitry for implementing the truth table above. Note that we have used a full adder for the least significant bit (LSB) and set C_0 to 0.

This circuit is called a *ripple adder*. To compute S_{n-1} you need to allow the carries to ripple through from bits 0 up to bit $n-1$. This suggests that for large n (say 32 bits for A and B each), you would have to allow a long delay. Later this lecture, we will discuss how this delay can be avoided. For now, we turn to other useful combinational circuits.



Encoders

An encoder is a combinational logical circuit whose outputs specify which of a set of possible inputs or groups of inputs has occured. The term *encoder* is used somewhat loosely. (There is no generally agreed upon formal definition of an encoder that I am aware of.) Let's look at a few examples.

A common example is a set of m input wires, such that exactly one of them has the value 1. For example, consider a keyboard with 5 buttons and suppose that the mechanics of these buttons are such that one and only one of these buttons is pressed down. (You may have seen something like this on a children's toy.) Such an encoder might be constructed using the truth table:

A	B	C	D	E	Y_1	Y_2	Y_3
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	0	0	0	1	0	0
1	0	0	0	0	1	0	1

Notice that if none of the button is pressed, or if multiple buttons is pressed (because you push hard and break the mechanism), then the above truth table does not specify what the value of the output variables Y_i is. But this is not a problem, as long as the design is such that only one *can* be pressed (1) at any time.

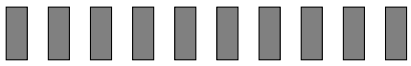
Suppose we allow for multiple buttons to be pressed and we want to explicitly indicate in the truth table what the output of the circuit will be for each possible input. One way to do it is shown below. Consider the following truth table and recall that X means "don't care". If no buttons are pressed, the output is 000. If two or more buttons are pressed, then the output is the same as if only the rightmost of these buttons was pressed.

A	B	C	D	E	Y_1	Y_2	Y_3	interpretation
0	0	0	0	0	0	0	0	none of the buttons is pressed
X	X	X	X	1	0	0	1	E is pressed (and maybe A,B,C,D)
X	X	X	1	0	0	1	0	D and not E (but maybe A,B,C)
X	X	1	0	0	0	1	1	C and neither E nor D (but maybe A,B)
X	1	0	0	0	1	0	0	etc
1	0	0	0	0	1	0	1	etc

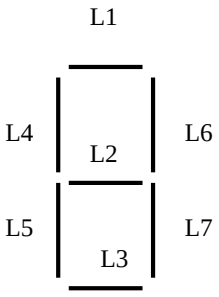
For example, written as a sum-of-products, we would have

$$\begin{aligned} Y_1 &= (A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} \cdot \overline{E}) + (B \cdot \overline{C} \cdot \overline{D} \cdot \overline{E}) \\ Y_2 &= (D \cdot \overline{E}) + (C \cdot \overline{D} \cdot \overline{E}) \\ Y_3 &= E + (C \cdot \overline{D} \cdot \overline{E}) + (A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D} \cdot \overline{E}) \end{aligned}$$

A second example of an encoder that you should be familiar with is an output device (an LCD) that displays a single digit from $\{0, 1, ..., 8, 9\}$. Each digit is defined by "turning on" a subset of the line segments $L_1, \cdots, L_7$. For example, when all seven of the line segments are on, the output digit represents "8", whereas when all lines except L2 are turned on, the output digit represents "0". In Exercises 2, you will write out the truth table for such an encoder.



ten buttons



Decoders

A decoder is the opposite of an encoder. It takes a code (a binary number) as input, and turns on one of several output wires (or some other event, i.e. subset of the output wires). For example, here is a truth table for a 1-to-2 decoder:

A	Y1	Y2
0	1	0
1	0	1

Here is the truth table for a "2-to-4 decoder":

A	B	Y1	Y2	Y3	Y4
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

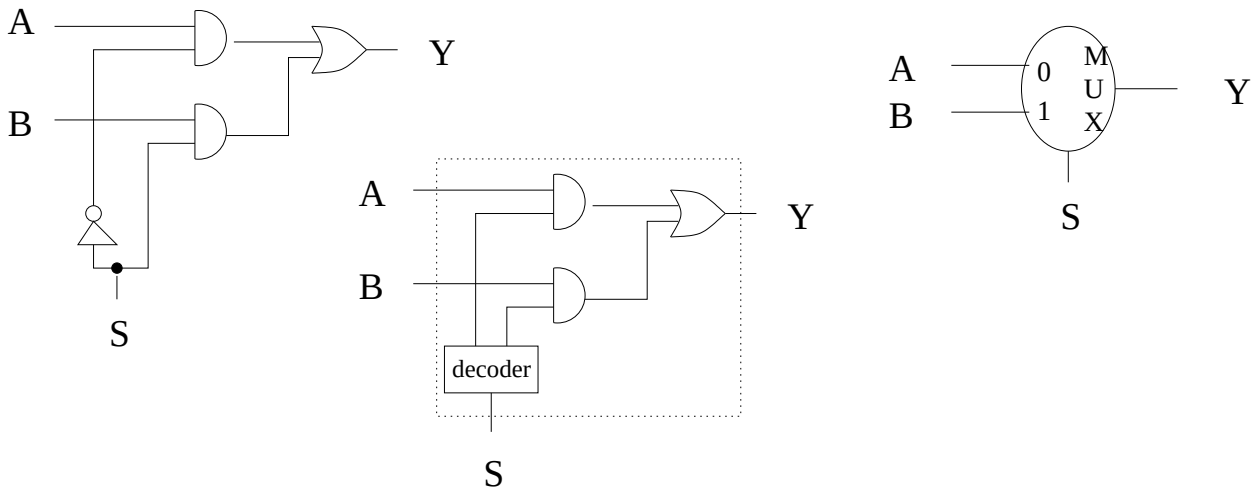
You should be able to draw the circuit for these two examples.

Multiplexor (or selector)

One common way that decoders are used is to select from a set of possible input variables. The circuit that does a selection is called a multiplexor. A multiplexor takes n inputs and chooses among them. A multiplexor is also called a *selector*.

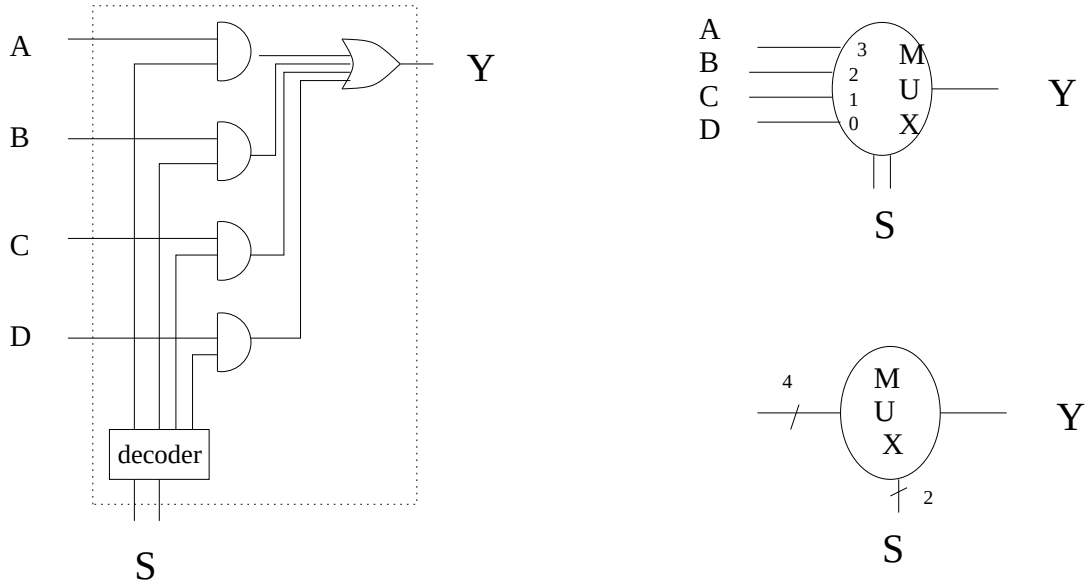
A multiplexor requires two types of inputs. The first carries the data that can go potentially through the circuit. The second carries a code saying which of the data goes through.

The simplest example of a multiplexor has two input wires that carry a bit of data – call these A and B – and a selector input wire – call it S – that says which of A and B to select. It is called a 1-bit multiplexor. (We saw this example at the end of last lecture.) The three circuits shown below show this "one bit multiplexor" at three levels of increasing abstraction, from left to right.



Suppose instead we had four different inputs (A, B, C, D) and we wanted to select from them. Then we would use a "two bit multiplexor" circuit as shown below on the left. The MUX symbol on the right will be used whenever we want to hide the underlying circuitry.

Sometime the inputs A, B, C, D are not single bits, but may have many bits, e.g. we might be selecting one of four integers. In this case we might use the same notation show in the two bit multiplexor figure, and if we wanted to explicitly show that the inputs had a certain number (say 16) of bits, we would label the wires as such, by putting a little slash in each wire and writing 16. We will see examples later in the course and the notation will be come natural to you.

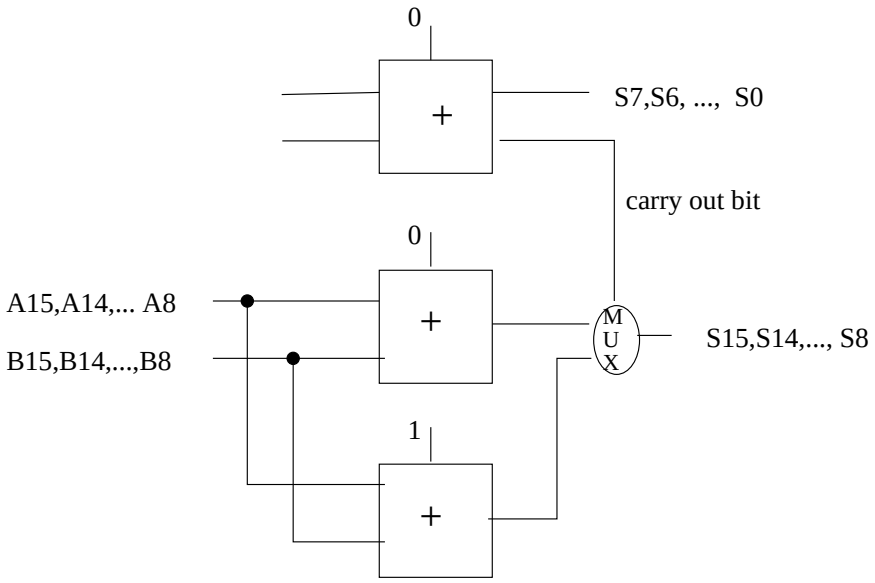


Fast adder

Let's look at a few examples of how multiplexors can be useful. Recall the ripple adder discussed at the beginning of the lecture. We can reduce the long delays for the carries to propagate, using the following trick. Assume n is an even number. Divide the bits into low order bits $(0, 1, \ldots, n/2-1)$ and high order bits $(n/2, \ldots, n-1)$. Build one circuit to compute the sum for the low bits. Build two circuits to compute the sum for the high order bits. Set the carry in bit for the LSB in the lower order bit circuit to 0. Set the carry in bits for the LSB in the two high order circuits to 0 and 1, respectively. In the figure below, the "+" box is a 16 bit adder, rather than a one bit adder. (I drew a slightly different looking figure in the lectures.)

The main idea here is that we don't need to wait for the worse case in which the carries are passed through the entire circuit (e.g. adding $A = 011111 \cdots 111$ and $B = 0000 \cdots 0001$). Instead, once the carries pass through the first $n/2$ full adders and the circuits stabilize, we know which of the upper order sums we want. We just select it. Thus we have almost cut the computation time in half. I write "almost" because we also have to do the selection, which takes time.

Note that we could repeat this idea, and speed up the computation of the three $n/2$ bit additions, by decomposing each of them into three additions of $n/4$ bits, and so on. There is a tradeoff here between time (fast) and space (having the replicate the upper order bits to handle the two cases of the carry out from low order bits).



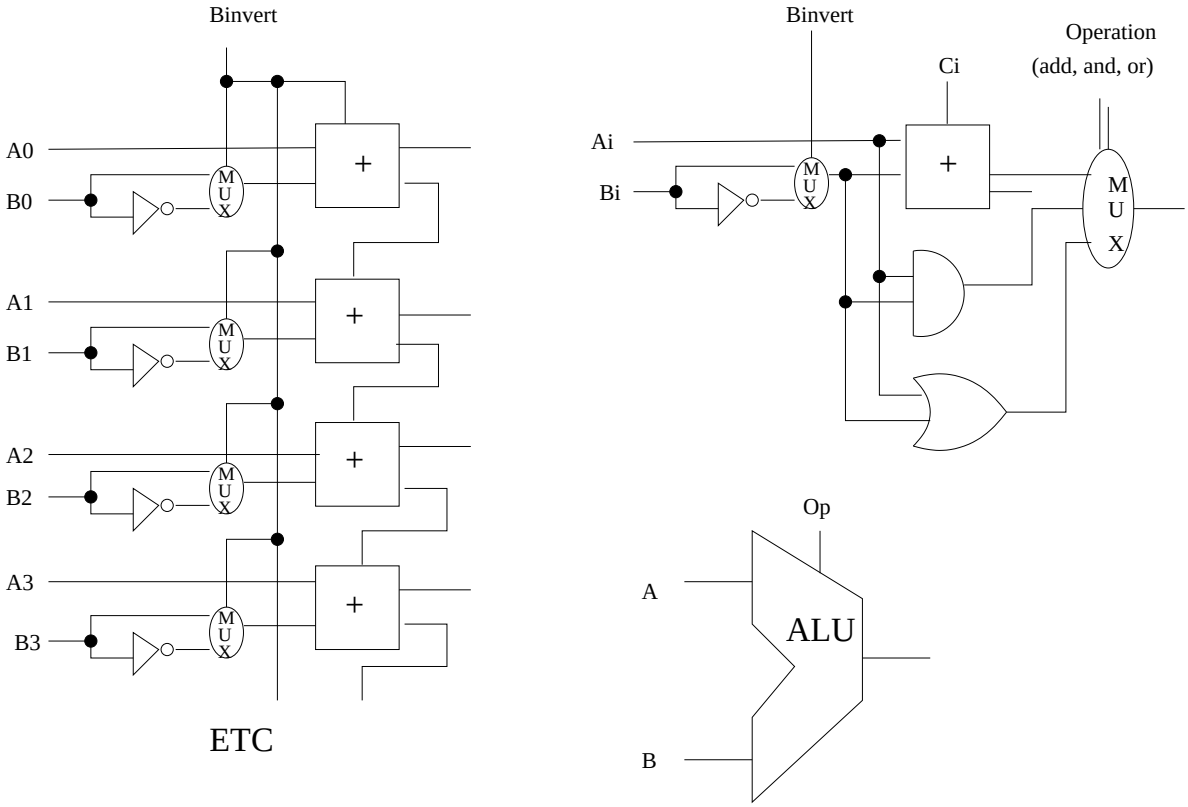
Subtraction

Suppose we wanted to build a circuit for performing subtraction.

$$\begin{array}{cc} & A_{n-1}\ A_{n-2}\ \ldots\ A_2\ A_1\ A_0 \\ - & B_{n-1}\ B_{n-2}\ \ldots\ B_2\ B_1\ B_0 \\ \hline \end{array}$$

To perform $A - B$, we compute $A + (-B)$. This requires us to negate B and then send the result through the adder.

To negate a number that is represented in twos complement, we invert each of the bits and then add 1 (lecture 1). To do this with a combinational circuit is easy. We can use circuit similar to the adder except that the B bits need to be inverted $\overline{B}$. To add one, we can set the carry in bit C_0 to 1. See below on the left. *Binvert* is the variable that specifies whether we are doing addition (0) or subtraction (1). Note that we can use this variable both to select the B versus $\overline{B}$, and for the C_0.



Arithmetic Logic Unit (ALU)

We can make the circuit even more powerful by also computing the operations AND and OR for each of the bits. In the figure above right, a dotted square indicates the adder part of the circuit for the ith bit. The entire local circuit computes four different operations, namely, addition and subtraction, AND, OR. We select from these four operations using a multiplexor. [**MODIFIED March 11:**] Notice the selector for the operation has only three possible values (add, and, or) rather than four (add, sub, and, or). The reason that the "sub" doesn't have its own value is that the subtraction operation is specifed by the *Binvert* signal, namely if the operation is subtraction then *Binvert* should also be set to 1, and otherwise *Binvert* should be set to 0.

Such a circuit is called an *arithmetic logic unit* or ALU. Every computer has (at least) one.