ENEL260 CA4: Combinational Logic

Ciaran Moore

"No ducks waltz. No officers ever decline to waltz. All my poultry are ducks. Therefore, . . . " – Lewis Carroll

A computer performs arithmetic and logical operations under the guidance of a stored program. Those operations are implemented in a component of the CPU called an Arithmetic Logic Unit, which is made up of *combinational logic* circuits.

Combinational logic circuits are made up of logic gates, and *do not have feedback connections*[1] within the design. Several kinds of combinational logic circuits are important building blocks in computers, and in particular within the ALU (arithmetic logic unit). These circuits include:

[1] Without feedback connections, the circuits don't retain *state*. So the output of a combinational logic circuit is some *combination* of the inputs–hence the name.

- Decoders
- Adders
- Multiplexers
- Shifters

The ALU itself is a multipurpose combinational logic circuit, with control signals[2] that select the arithmetic or logical operation it should perform. In a microprocessor with an N-bit wordlength, the ALU needs to be able to accept two N-bit *operands* (from general purpose registers) and produce an N-bit result (back into a register). Additionally, the ALU should be able to accept a carry-in and produce a carry-out for some operations. The ALU often also provides a means of checking the nature of the result: for example if the ALU is used to compare two operands for equality, a logical TRUE is generated if the operands are equal and a logical FALSE if they are not.

[2] Typically in the form of a binary "operation code" (or op-code) that indicates which operation to perform.

Designing Combinational Logic

The basic steps in designing a combinational logic circuit are:

1. Write out a *truth table* or *Boolean function* that defines the desired function of the logic circuit.
2. If starting from a truth table, convert[3] the truth table to a Boolean function.
3. Create a logic gate circuit that implements the Boolean function by placing a gate for each Boolean operator and connecting up inputs to outputs through the gates.

[3] If a simple expression isn't obvious, you can always use a "sum-of-products" expression and then simplify.

In some cases, where we're designing more complex logical functions, we may already have building blocks like adders or multiplexers to work with.

Sum-Of-Products

A *minterm* is an AND (or "product") term[4] that contains all of the input variables to a Boolean function, with each variable appearing *exactly once* (in either positive or inverted form).

[4] That is, a component of an expression.

Example: For the 3-input Boolean function $f(a, b, c)$:

- abc is a minterm
- $a\bar{b}\bar{c}$ is a minterm (each variable appears once, some are inverted)
- ab is *not* a minterm (c is missing)
- b is *not* a minterm (a and c are missing)
- $(a + b)$ is *not* a minterm (it's not a product)

A minterm produces a 1 for just one possible combination of input values. So we can think of a minterm as corresponding to a single row in a truth table (the combination of input values that make the minterm 1). To convert a truth table into a Boolean function, we can simply construct the function by ORing together ("summing") the minterms that correspond to the rows of the truth table for which the output should be 1. This is known as the *sum-of-products* form[5] of the function.

[5] Or "sum-of-minterms" forms, or minterm canonical form.

Example: For the truth table

a	b	c	z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

we can find $z = f(a, b, c)$ in sum-of-products form by observing that the rows where $z = 1$ are:

- $a = 0, b = 0, c = 0$ which corresponds to minterm $\bar{a}\bar{b}\bar{c}$
- $a = 0, b = 1, c = 0$ which corresponds to minterm $\bar{a}b\bar{c}$
- $a = 0, b = 1, c = 1$ which corresponds to minterm $\bar{a}bc$

So $z = f(a, b, c) = \bar{a}\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}bc$.

By applying some Boolean algebra, we can simplify this to $f(a, b, c) = \bar{a}(b + \bar{c})$.

Decoders and Encoders

DECODING IS A WAY to convert a number that is encoded as a bit pattern (such as an address, or an op-code) into a signal that can enable a single component (for example a memory location, or an operation in the ALU). Decoders are a common building block in computing systems. They are typically named based on the number of bits in the code and the number of output lines. Thus a decoder that takes a 2-bit code and has 4 output lines (one line for each value representable by the code) is a 2-to-4 decoder (sometimes 2:4 decoder).

Example: To design a decoder, we can follow the steps outlined above for designing a generic combinational logic circuit. Let's say we're trying to design a 2-to-4 decoder that accepts a 2-bit address (identifying one of 4 possible memory locations) and outputs a 1 on a single one of the 4 output lines (activating the appropriate memory location for input or output).

1. The truth table that captures our desired function is:

A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

   Only one of the D_i outputs is a 1 for a given input code A_1A_0.

2. From the truth table, the Boolean expressions for each output are[6]:

$$\begin{aligned} D_0 &= \overline{A}_0\overline{A}_1 \\ D_1 &= A_0\overline{A}_1 \\ D_2 &= \overline{A}_0A_1 \\ D_3 &= A_0A_1 \end{aligned}$$

3. Fig. 1 shows a logic gate implementation of the 2-to-4 decoder, derived from the Boolean expressions. The decoder takes a number encoded as a 2-bit pattern (thus encoding 4 possible values) and decodes it into a signal on one of 4 output lines (each line corresponding to one of the possible encoded values).

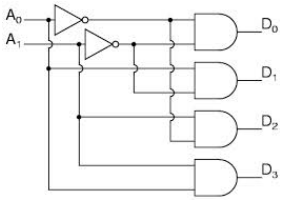


Figure 1: Circuit for a 2-to-4 decoder

[6] Note that each of these is a minterm

The inverse of decoding is *encoding*, which is (typically) changing a bit pattern that contains only a single 1 into a binary number, e.g. encoding the binary pattern 00001000 (8-bit) $\longrightarrow$ 011 (3-bit). An example application for an encoder is to encode which single key on a keyboard has been pushed.

To handle situations where more than one input has a 1 you can use what's known as a *priority encoder*, which outputs a code for the "highest priority" input that is 1.

$I_3I_2I_1I_0$	O_1O_0
0001	00
0010	01
0100	10
1000	11

Figure 2: Truth table for a 4-to-2 encoder

Multiplexers

THE *multiplexer* IS effectively a switching circuit that allows one of several inputs to be selected as the output[7]. In its simplest form, Fig. 3, a single control line S controls whether the value of input A or the value of input B appears at the output O.

[7] It might, for example, be used in an ALU to select between outputting the result of an arithmetic operation or a logical operation.

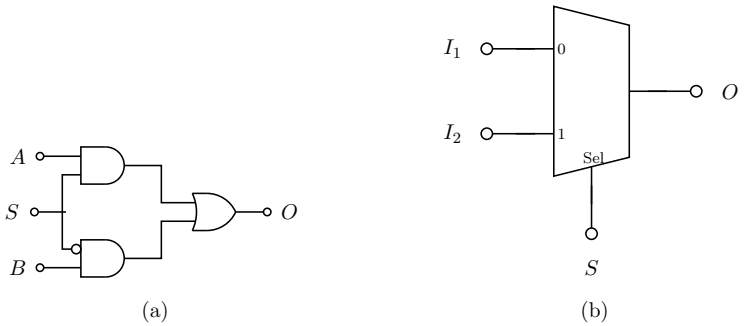


Figure 3: A 2:1 multiplexer with 2 inputs, 1 control line: (a) logical implementation, (b) common circuit symbol.

Example: Here's the truth table for a 2:1 multiplexer:

Inputs			Output O
S	A	B	
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Although this is readable[8], it does somewhat obscure the functionality. By viewing the control signal as a selector between possible outputs we can rewrite the truth table in a form that makes the multiplexer function more obvious:

[8] And we can extract minterms from it easily: $O = \overline{S}A\overline{B} + \overline{S}AB + SA\overline{B} + SAB$

	S	0	1
Inputs		Output O	
A	B		
0	0	0	0
0	1	1	0
1	0	0	1
1	1	1	1

When $S = 0$, the output is B. When $S = 1$, the output is A. By inspecting the table we can see that $O = \overline{S}B + SA$.

An N:1 multiplexer with $N > 2$ needs a greater number of control lines; in general P control lines allows N to be up to 2^P in size.

Multiplexing is very commonly used in computing and telephony. It allows an efficient use of resources. The opposite of multiplexing is *demultiplexing*, i.e. using one or more control lines to connect a single input to one out of two or more outputs. Thus a 1:8 demultiplexer has 1 input, 8 outputs and 3 control lines.

Shifting

A set of 2:1 multiplexers can be combined to form a *barrel shifter*. Fig. 4 shows how that can be done. In the figure the three inputs $shift(i)$, $i = 0, 1$ and 2, control the size of the shift (0 to 7). Control input $shift(2)$ corresponds to the MSB for the control binary sequence. For each of the 2:1 multiplexers, control input = 0 means that the upper data input is directed to the output, while control input = 1 means that the lower is.

To see how the shifter works, sketch the path for data input $inp(0)$ with $shift = 001$ and also for $shift = 111$.

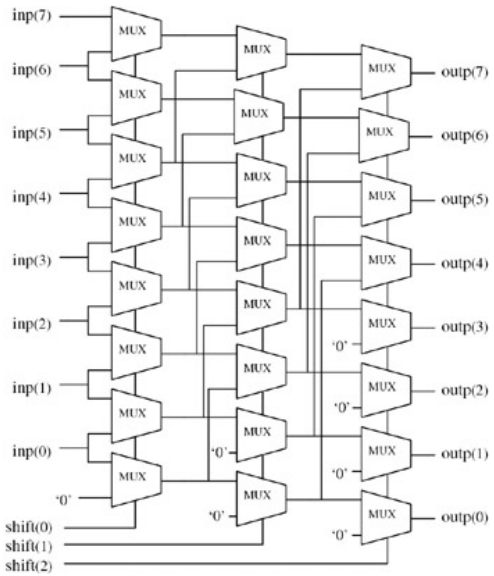


Figure 4: An 8-bit barrel shifter.

Adder Circuits

THE ADDER CIRCUIT is one of the primary building blocks of the ALU. We've already looked at the combinational circuit for a 1-bit adder (Fig. 5 shows the 1-bit adder logic circuit from lecture

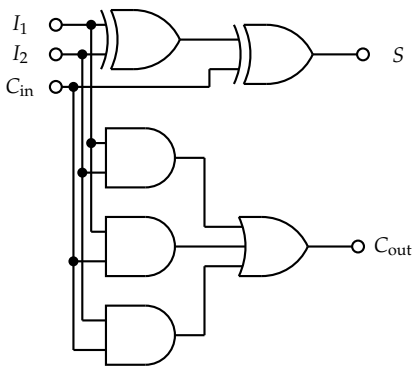
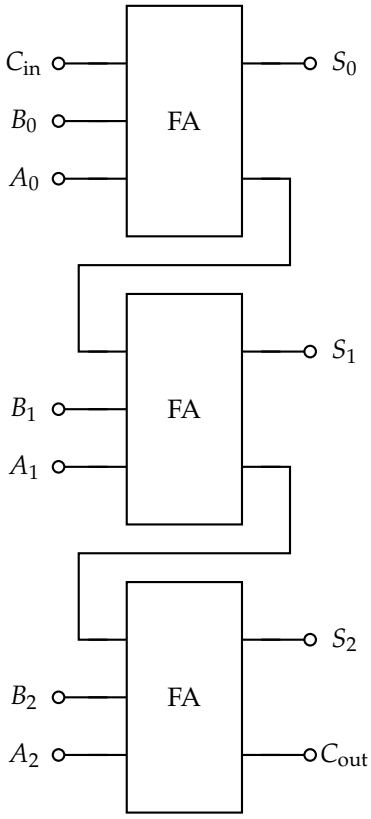


Figure 5: A 1-bit full-adder (FA)

CA1). If we use a sequence of bits to encode integer numbers, or real numbers to a *fixed precision*, then we can add these numbers by performing 1-bit additions on the components of the encoded number. Fig. 6 shows one possible way of implementing an N-bit adder, by connecting together a set of 1-bit full-adders.



(b)

Figure 6: An N-bit ripple-carry adder for $N = 3$

The N-bit adder in Fig. 6 is called a *ripple-carry adder*, and is perhaps the simplest adder to design. Its limitations are in the cascading of carries from the LSB position to the MSB position. This effectively sets the overall speed performance[9] of the adder, as the long chain of logic gates has a substantial *propagation delay*. There are many other adder circuit designs that are intended to improve the performance, and developing better adder designs is still an active area of research.

[9] Logic gates can't change their outputs instantly. There is always some finite–but usually small–time between a change in an input and the corresponding change in the output. This is called propagation delay.

Example: Consider adding the two integers 2 and 3, with each represented in 2's complement, with a full-adder circuit exactly like Fig. 6, but with $N = 4$. We can find the overall sum by performing a 1-bit addition on each bit:

	0	0	1	0	= 2
+	0	0	1	1	= 3
	0	1	0	1	= 5

Looking Deeper: Python Model We can build a simple Python model of the ripple-carry adder using our existing full-adder models:

```
def ripple_carry_4bit_adder(A, B, cin):
    """
    Takes a pair of 4-tuples (representing 4-bit numbers)
    and a carry
    value as inputs.
    Returns a 5-tuple containing
    1. the binary sum of the inputs (including a 'carry'
    input)
    2. a 'carry' value that is non-zero if the sum overflows.
    For all tuples, the first element is the MSB.
    """
    # Decompose input tuples
    A3, A2, A1, A0 = A
    B3, B2, B1, B0 = B
    # Perform addition
    s0, c = full_adder(A0, B0, cin)
    s1, c = full_adder(A1, B1, c)
    s2, c = full_adder(A2, B2, c)
    s3, cout = full_adder(A3, B3, c)
    return (s3, s2, s1, s0, cout)
```

You can try out arithmetic using this multi-bit adder:

```
>>> ripple_carry_4bit_adder((0,0,1,0), (0,0,1,1), 0)
(0, 1, 0, 1, 0)
```

What about subtraction? We *could* design a special logic circuit for subtraction, but it turns out to be simpler to change the sign of the subtrahend[10] and then do an addition, i.e. $p - q = p + (-q)$.

[10] The number being subtracted (https://en.wikipedia.org/wiki/Subtraction#Notation_and_terminology). To find the 2's complement representation of a negative number, take the bitwise logical complement of the corresponding positive number and add 1.

Example: Consider the calculation 3 - 4. To turn 4 into -4, we need to form the 2's complement of 4, which can be achieved by taking its complement and adding 1.

	0	1	0	0	= +4
	1	0	1	1	1's complement
+	0	0	0	1	add 1
	1	1	0	0	= -4

And now to complete the calculation

	0	0	1	1	
+	1	1	0	0	
	1	1	1	1	= -1

Given a control signal, *Sub*?, that is equal to 1 when a subtraction is required, we can build a truth table for forming a single bit of the adder input (complementing the bit in the case of a subtraction):

Inputs		Output
Sub?	*In*	*Out*
0	0	0
0	1	1
1	0	1
1	1	0

You should recognize this as the XOR of the *Sub*? control signal and the *In* input. If the *Sub*? signal is also connected to the "Carry In" input of the adder circuit[11] we end up with a way to perform either addition or subtraction depending on the value of *Sub*?.

[11] Effectively adding 1 to the input when *Sub*? = 1.

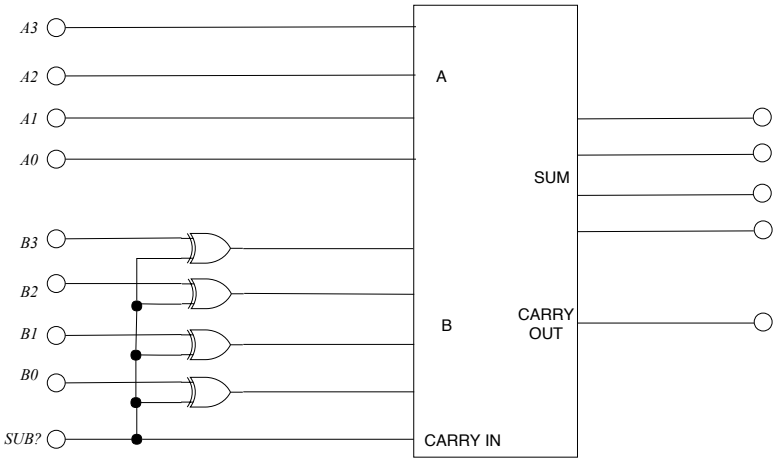


Figure 7: Adder with subtraction control signal

Op-Codes and Logical Operations

THE ALU MUST PROVIDE for performing several different logical operations, with the operation to be performed selectable using a control signal. The following example illustrates how combinational logic can be used to make a single circuit that can be controlled to perform various logical operations.

Example: The truth table in Fig. 8 specifies how a 2-bit control code s_1 s_0 controls whether the operation performed on inputs a and b is AND, OR, or XOR.

	s_1	0	1	1
	s_0	1	0	1
Inputs		Output O		
a	b	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Figure 8: Truth table for a 1-bit selectable logical operation.

From Fig. 8, the output O is given by:

$$\begin{aligned} O &= \overline{s_1} s_0 ab + s_1 \overline{s_0}(a + b) + s_1 s_0 (a \oplus b) \\ &= \overline{s_1} s_0 ab + s_1 \overline{s_0} a + s_1 \overline{s_0} b + s_1 s_0 \overline{a} b + s_1 s_0 a \overline{b} \end{aligned}$$

This equation can be implemented using a set of AND and OR gates, along with some inverters.
