ECE 124 - Digital Circuits and Systems

Kevin Carruthers

Winter 2013

Boolean Logic

We use the binary system for virtually all computations, and as such rely heavily on boolean logic. Since a value represented in binary contains only 1s and 0s, we can easily relate this to boolean algebra, which has only *true*s and *false*s.

Truth Tables

Truth tables are a method to define binary / logic functions. A truth table lists the values of a function for all possible input combinations.

There are three main operators: AND, OR, and NOT.

Table 1: AND (denoted by multiplication)

x	y	f
0	0	0
0	1	0
1	0	0
1	1	1

Table 2: OR (denoted by addition)

x	y	f
0	0	0
0	1	1
1	0	1
1	1	1

We can write logic functions as equations, using these denotations.

Table 3: NOT (denoted by exclamation marks, negations, overlines, and primes)

x	f
0	1
1	0

For example $f = !x!y + xy = \overline{x} \cdot \overline{y} + x \cdot y$ is

x	y	f
0	0	1
0	1	0
1	0	0
1	1	1

Note the following transformation from the truth table to the equation form

$$\begin{aligned} f &= !(x + y - xy) \\ &= !(x + y) - \, !xy \\ &= !(x + y) + xy \\ &= !x!y + xy \end{aligned}$$

Thus we can derive $!(x + y) =!x!y$ which is often represented as

$$\overline{x + y} = \overline{x} \cdot \overline{y}$$

Boolean Algebra

Boolean algebra is an algebraic structure defined by a set of elements (0 and 1) and operators (+, *, and !) which satisfies the following postulates:

1. Closure - The system is closed with respect to its elements for each operator.
2. Identity - The operators must have identity elements $\big(x + 0 = x, 1x = x\big)$
3. Comutivity - The operators must perform the same function regardless of element order $\big(x + y = y + x, xy = yx\big)$
4. Distributivity - The elements must distribute within operators $\big(x(y+z) = xy+xz, x+yz = (x + y)(x + z)\big)$
5. Opposite - The elements must have opposites within their scope $\big(x + \overline{x} = 1, x\overline{x} = 0\big)$
6. Uniqueness - There are at least two non-equal elements. $\big(0 \neq 1\big)$

There are also a collection of useful theorems which can be used to simplify equations:

- $x + x = x, \ xx = x$

- $x + 1 = 1,\ 0x = 0$
- Involution: $\overline{\overline{x}} = x$
- Associativity: $x + (y + z) = (x + y) + z,\ x(yz) = (xy)z$
- DeMorgan: $\overline{x + y} = \overline{x} \cdot \overline{y},\ \overline{xy} = \overline{x} + \overline{y}$
- Absorption: $x + xy = x,\ x(x + y) = x$
- Cancellation: $x(x + y) = x + xy = x(1 + y) = x$
- Reduction: $x + \overline{x}y = x + y$

Function Manipulation

We can use theorems and postulates to simplify and manipulate functions. This can give us smaller, cheaper, faster circuits. Aside: for any binary variable x, you always have a positive literal x and a negative literal $\overline{x}$.

Example:

$$\begin{aligned} f &= !c!ba + c!ba + !cba + cba + !cb!a \\ &= !c!ba + c!ba + !cba + !cba + cba + !cb!a \\ &= (!c + c)!ba + (!c + c)ba + !cb(a + !a) \\ &= !ba + ba + !cb \\ &= (!b + b)a + !cb \\ &= a + !cb \end{aligned}$$

Physical Cost of Circuits

When we are attempting to determine the most reduced form of a circuit, it is useful to assign a cost to each of the inputs. We use the following system: inverter gates at the input are free, but any other non-negation gate costs one plus the number of inputs they have. Negation gates later in the circuit cost one.

Minterms and Maxterms

Every n variable truth table has 2^n rows. For each row, we can write its minterm (an AND which evaluates to 1 when the associated input appears, otherwise 0) and maxterm (an OR which evaulates to 0 when the associated input appears, otherwise 1).

x	y	z	Minterm	Name	Maxterm	Name
0	0	0	$\overline{xyz}$	m_0	$x+y+z$	M_0
0	0	1	$\overline{xy}z$	m_1	$x+y+\overline{z}$	M_1
0	1	0	$x\overline{y}z$	m_2	$x+\overline{y}+z$	M_2
0	1	1	$\overline{x}yz$	m_3	$x+\overline{y}+\overline{z}$	M_3
1	0	0	$x\overline{yz}$	m_4	$\overline{x}+y+z$	M_4
1	0	1	$x\overline{y}z$	m_5	$\overline{x}+y+\overline{z}$	M_5
1	1	0	$xy\overline{z}$	m_6	$\overline{x}+\overline{y}+z$	M_6
1	1	1	xyz	m_7	$\overline{x}+\overline{y}+\overline{z}$	M_7

Canonical Sum-of-Products (SOP)

A way to write down a logic expression from a truth table using minterms. It has a number of products equal to the number of 1s in the table. Thus $f(x)$ gives

Table 4: f(x)

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$f(x) = \overline{xy}z + x\overline{yz} + xyz$$

Canonical Product-of-Sums (POS)

A way to write down a logic expression from a truth table using maxterms. It has a number of sums equal to the number of zeros in the table. Thus $f(x)$ gives

$$f(x) = (x+y+z)(x+\overline{y}+z)(x+\overline{y}+\overline{z})(\overline{x}+y+\overline{z})(\overline{x}+\overline{y}+z)$$

To **convert** between the two, we have $!(m_0 + m_2) = !m_1!m_3 = M_1M_3$ or, more generally,

$$\overline{m_n} = M_n$$

Standard SOP

The canonical SOP is fine, but is not minimal. Any AND of literals is called a **product term** (ie. a minterm is a product term, but not all product terms are minterms).

For example, for f we have

$$\begin{aligned} f &= m_3 + m_5 + m_6 + m_7 \\ &= \overline{x}yz + x\overline{y}z + xy\overline{z} + xyz \\ &= yz + xz + xy \end{aligned}$$

These are **two-level** circuits: if you ignore inversions, you have two levels of logic (a single set of each AND and OR gates, for each possible circuit).

Standard POS

Similarly, canonical POS are not minimal. Any OR of literals is called a **sum term** (ie. a maxterm is a sum term, but not all sum terms are maxterms).

For example

$$\begin{aligned} f &= (x + y + z)(x + \overline{y} + z)(x + \overline{y} + \overline{z})(\overline{x} + y + \overline{z})(\overline{x} + \overline{y} + z) \\ &= (x + z)(\overline{y} + z)(x + \overline{y})(\overline{x} + y + \overline{z}) \end{aligned}$$

Other Logic Gates

For actual implementations, some other types of logic gates are useful.

NAND

Performs the AND then NOT function.

Table 5: NAND, $f = \overline{xy}$

x	y	f
0	0	1
0	1	1
1	0	1
1	1	0

NOR

Performs the OR then NOT function.

Note that NAND and NOR gates are **universal** (they can implement any function). On CMOS, NAND and NOR gates use less transistors (four, instead of the normal six); this makes them extremely useful in creating low-cost circuits.

Table 6: NOR, $f = \overline{x + y}$

x	y	f
0	0	1
0	1	0
1	0	0
1	1	0

XOR/NXOR

The exclusive OR is an OR gate which does not result in true when the input can be ANDed to produce a true. In essence, it is an OR minus an AND.

Table 7: XOR, $f = x \oplus y$

x	y	f
0	0	0
0	1	1
1	0	1
1	1	0

We also have NXOR, which is simply an XOR which is then negated.

Table 8: NXOR, $f = \overline{x \oplus y}$

x	y	f
0	0	1
0	1	0
1	0	0
1	1	1

For more than two inputs, an XOR function will be true if the number of true inputs is odd.

Buffers

Denoted as a triangle, a buffer does nothing to the function. It amplifies the signal, which is especially useful in long wires.

Karnough Maps

A **K-Map** is a different graphical representation of a logic function equivalent to a truth table (i.e. it holds the same value). It's not tabular, but is drawn much like a matrix. Note that it only works for up to five inputs, beyond that it becomes ungainly and useless.

It is useful because we it allows us to do logic optimization / simplification graphically.

For a two-input function

x	y	f
0	0	1
0	1	1
1	0	0
1	1	1

we have

	0	1
0	1	0
1	1	1

By "drawing" rectangles on this chart (over top of the true values), we can find an equation for the function. Note that these rectangles can "wrap around" the chart or overlap each other, and that we can also go backwards to generate the table based on an equation (especially simplified equations).

Each rectangle gives an ANDed function, and the rectangles can then be ORed. Longer rectangles cover more possibilities. All rectangles must have dimensions equal to 2^n for any integer n.

We can also draw rectangles on the false responses. This will give us a product of sums, instead of a sum of products. This can be easier that placing a rectangle around the true results in some cases.

In essence, the best option is to pick whichever one gives you fewer, larger rectangles. Though you will be able to simplify your answer either way, this will give you the simplest possible starting equation (note that some equations may still require some basic simplifications to be made the most simple possible).

Note that we can also do this in three or more dimensions with

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

and get

	00	01	11	10
0	0	0	1	0
1	1	0	1	1

Note that the xy values are not strictly increasing. This is because the difference in variables between any row or column must have no more than one changing variable.

Four-variable maps have dimensions four by four. Five-variable ones can be represented as two four-variable maps.

"Don't Cares"

Sometimes the value of f doesn't matter, and we can exploit this face when minimizing: these cases are "don't cares". You can chose to set these values as either 0 or 1, whatever makes the logic simpler. For K-Maps, we simply include these in any rectangles which could be "improved" by their inclusion.

General Circuitry

Multiple Output Functions

When dealing with two-output functions, ie circuits with two different functions, optimizing each one seperately is not always the best option. If we can design a circuit with shared circuitry, this will often decrease our total cost. For example, if we have $f = a+b+c+d+e$ and $g = a+b+c+d+f$, it is much better to share $a+b+c+d$ between the two circuits.

Factoring

We may sometimes factor a circuit into smaller components to reduce the total cost. For example, if we can write $f(x) = g(h(x))$, sometimes it is cheaper to implement the latter. If the factored circuit has any inputs shared between its subfunctions, we call it a **disjoint decomposition**.

Combinational Circuits

A combinational circuit is one which consists of logic gates with outputs determined entirely by the present value of the inputs. They can be any number of levels, with any number of inputs or outputs. These are the types of circuits we've discussed so far and include any circuit without storage elements.

Arithmetic Circuits

Arithmetic circuit are a useful type of combinational circuit which perform some arithmetic operation of binary numbers (see: binary number representations, binary arithmetic, "two's complement").

Half-Adders

A binary half-adder is a circuit which takes two bits, adds them, and produces a sum and a carryout (i.e. $0 + 0 = 0, 0,\ 0 + 1 = 1, 0,\ 1 + 1 = 0, 1$).

The sum in a half-adder is given by an XOR, and the carryout is given by an AND.

Full-Adders

A binary full-adder is similar to a half-adder, with the addition of accepting a carryin value. This allows us to add n-bit numbers.

We can either represent a full-adder as two half-adders with ORed carryouts, or as follows: the sum is given by a three-input XOR, and the carryout is given by each set of two inputs (i.e. there are three) ANDed together then ORed.

Ripple Adders

By linking together n 1-bit full-adders, we can build an n-bit adder. However since gates do not change values instantly, these circuits will have some delay. By tracing the slowest path, we can find the minimum amount of time we need to wait for the output to be correct.

By combining multiple levels of a ripple adder, we can increase the performance of the circuit in exchange for increasing its cost. In this case, we call these circuits **carry look-ahead** circuits, and will often see things such as a 16-bit ripple adder composed of four 4-bit CLAs.

Subtraction

We can make subtracting circuits through clever manipulation of binary: since subtraction is performed by taking the 2s complement of the subtrahend and performing addition, we can see that we can feed our subtrahend into a full-adder (after each digit has been XORed and thus transformed into a twos complement).

Note another benefit to this approach: by attaching our XOR to a switch, we can create on circuit which performs both addition and subtraction at almost no overhead.

Multiplication

Since multiplication is simply a combination of addition and ANDs, we can create an array multiplier with only AND gates and 1-bit FAs.

Overflow Detection

When two n-bit numbers are added/subtracted/etc and the sum requires $n+1$ bits, we say an overflow has occurred.

When adding unsigned numbers, an overflow is detected if there is a carryout from the most significant bit (MSB). For signed numbers, we can detect overflow by examining the carryin and carryout of the MSB. If they are different, there must be an overflow.

Magnitude Comparisons

Equality

The ith digit of A and B are equal if $A_i = B_i$. We introduce $e_i = A_iB_i + \overline{A_i} \cdot \overline{B_i} = \overline{A_i \oplus B_i}$ and show that A and B are equal if $e_ne_{n-1}...e_0$.

Inequality

We can use our equality comparison to determine inequality, as well. Consider: for any n-bit numbers, if we compare the MSBs and find them to have an inequality, we have the overall inequality. If they are equal, we simply go to the next lower bit and repeat this process.

We can write this algorithm as follows: $A > B$ if $(a_n\overline{b_n}) + e_n(a_{n-1}\overline{a_{n-1}}) + (e_ne_{n-1})(a_{n-2}\overline{n_{n-2}}) + ...(e_ne_{n-1}...e_1)(a_0\overline{b_0})$. For $A < B$, we simply NOT the B bits instead of the A bits.

Alternatively, we can implement a reduced circuit as follows: $(A < B) = \overline{(A = B) + (A > B)}$.

Decoders

When we have n bits, we can possibly represent 2^n distinct patterns. Figuring out which pattern is represented in n-bits is called **decoding**. We can consider a decoder to recognize input patterns and output a 1 corresponding to the minterm seen at the output. We represent m-to-n decoders as $m = 2^n$.

Often, decoders also have **enable signals**: when it is true, the decoder behaves normally, otherwise all outputs are zero.

For example (ignoring the enable bit): a decoder with three inputs will have eight outputs. From top to bottom, we have 000, 001, 010, ... 110, 111. Thus decoders can be used to immensely simplify the representation of a circuit.

Sometimes, decoders are built to be the opposite of the above example. In this case, the decoder has been built to be **active low**, instead of what common sense dictates to be normal (i.e. **active high**).

Decoder Trees

We can implement larger decoders with smaller ones: i.e. we can create a 4-to-16 decoder using five 2-to-4-decoders. This is done by connecting the outputs of one decoder to the enables of the next "layer" of decoders. Note that the MSBs should be in the earliest layers.

Encoders

Encoders are backwards decoders, and provide n outputs given 2^n inputs (plus any enable switches).

This approach has multiple problems: if multiple inputs are active, the output is undefined. Additionally, it produces a zero'd output both when no input is active and when the first bit is active.

As such, we use **priority encoders**, which are encoders which give priority to higher numbered (later) inputs. They also have validity outputs to indicate when not all inputs are zero.

Multiplexers

Multiplexers are a combinational circuit block which has data inputs, select inputs, and a single data output. Data is passed from one of the inputs through to the output bassed on the setting of the select lines.

For a 2-input multiplexer with inputs a, b, select s, and output f, we have $f = \overline{s}a + sb$

s	f
0	a
1	b

We can build larger multiplexers from many smaller ones by using the same select on each row of multiplexers, and slowly whittling down our inputs.

Multiplexers can be used to greatly reduce our circuits: for example, take $f = \overline{a}c + a\overline{b}c + ab\overline{c}$. If we use a and b as our select, then our four inputs simply need to be $c, c, c, \overline{c}$.

This should make it obvious that we can implement any function with MUX. This is called a **Shannon Decomposition**.

Demultiplexers

A demultplexer switches a single data input into one of several output lines. It is simply a decoder in which the meanings of the inputs has been changed.

Tri-State Buffers

Buffers are circuit elements which do "nothing" to your signal. A tri-state buffer has a single input, output, and select *oe*. When the select is enabled, the output equals the input. When the select is disabled, the output is disconnected from the input.

This can allow us to drive multiple signals down a singal wire (at different times). In other words: we can implement multiplexers with tri-state buffers.

Consider a 4-to-1 MUX made with tri-state buffers and a decoder as follows: the two inputs to a 2-to-4 decoder are the selects and each of the four signals enters a buffer. With the decoder, we select which of the signals goes through their buffer, and come out with a single signal.

Busses

Busses are collections of multiple wires. They can be represented either as parallel lines or a single bold wire with a slash and number count of individual wires. There can never by more than one source driving each wire, so a 8-bit bus through a circuit implies the existance of 8 of that element.

Sequential Circuits

We can include **storage elements** which act like memory to store a system state into a combinational circuit to get a sequential circuit. Outputs are a function of both the current circuit inputs and the system state (i.e. what happened in the circuit before).

There are two main types of sequential circuits: **synchronous sequential circuits**, which derive their behavior from the knowledge of signal values at discrete times, and **asynchronous sequential circuits**, which derive their behavior from the values of signals at any instant in time, in the order in which input signals change.

Clock Signals

Clock signals are particularly important to understand for designing synchronous circuits (which we also refer to as **clocked circuits**, when they involve a clock).

Clock signals are periodic signals (i.e. they have both a frequency and a period). We can use clocks to control when things happen because their transitions (from off to on or on to off) occur at discrete instances in time. We refer to the 0 to 1 transition as the rising edge of the clock, and the other as the falling edge.

Storage Elements

Latches

Latches are **level sensitive** storage elements. Level sensitive elements are ones which operate when a control signal is at either 0 or 1, but *not* at its rising or falling edges. Latches are not necessarily useful for clocked circuits, but they are useful for synchronous ones.

There are several types of latches.

The **SR latch** can be built with either NANDs or NORs. It has two signals S and R and two outputs Q and $\overline{Q}$. In the NOR implementation, we have two NORs. Each of them has a single input from our given inputs, as well as the output of the other NOR gate. In this way, we can see that the cross-coupled gates introduce a **combinational loop**. The NAND implementation is the same, only with NANDs instead of NORs.

For the NOR implementation, an input of $(S, R) = (1, 0)$ will set $(Q, \overline{Q}) = (1, 0)$ and $(S, R) = (0, 1)$ will set $(Q, \overline{Q}) = (0, 1)$. The interesting aspect of this circuit is when we set $(S, R) = (0, 0)$. In this case, the outputs *do not change* from whatever they were immediately beforehand. If S had been enabled and R wasn't, this would produce Q. If the opposite signals were enabled, the output would remain NOT Q. Setting $(S, R) = (1, 1)$ gives $(Q, \overline{Q}) = (0, 0)$ which is (obviously) undesireable.

The NAND implementation simply gives the opposite outputs, and uses $(1, 1)$ as its hold state, and $(0, 0)$ as its undesireable one.

Often, we create **gated latches** by adding some extra NAND gates in front of a latch. In this way, we can use an enable signal by NANDing it with each element. Note that this will negate the outputs, so a gated NAND SR gate will produce outputs like a non-gated NOR SR gate with an enable signal. In this case, the enable signal acts as a permanent hold.

The **D latch** is an SR latch where instead of having two signals we simply have D and $\overline{D}$. We thus avoid the undesireable state, and can maintain our hold state if we gate it.

Note the following potential issue with latches: they are level sensitive, and thus do not allow for precise control (i.e. the output of a latch can change at any time while the clock is active). This creates an interval in time over which an aspect of the circuit could change rather than a select instant for the same. It would be better to allow the output to change only on the rising or falling edge of a clock. This brings us to flip-flops.

Flip-Flops

Flip-flops are edge-triggered storage mechanisms. Consider a D-latches with output piped into a second D-latch. Assume both have a clock attached to their enable bit, but it has been NOTed for one of them. Only one can change at a time, so the second latch will always have a stable input source. Note that these are referred to as **DFF**s.

Flip-flops can have asynchronous signals which force the output Q to a known value. One which forces $Q = 1$ is called a **set** or **preset**. One which forces $Q = 0$ is called a **clear** or **reset**. A signal which prevents the clock from causing changes is called a clock enable.

Besides DFFs, which give D as an output when Q changes, we have **TFF**s, which toggle the output when D (referred to as T in this case) is one and hold when it is zero, and **JKFF**s, which are a combination of both (i.e. for (J, K) we have the following four potential outcomes: no change, reset, set, complement). We can build both TFFs and JKFFs from BFFs and basic circuit elements.

To analyze the timing of a flip-flop, it is important to not that it takes time for gates to change their output values (there are propogation delays due to resistance, capacitance, et cetera). We need to ensure the steup time (TSU), hold time (TH), and clock-to-output (TCO) are all functional.

The TSU is the amount of time that the data inputs need to be held stable prior to the active clock edge, the TH is the amount of thime the inputs need to be held stable after the active clock edge, and the CO is the amount of time it takes for the output to become stable after the active clock edge.

Registers

A single FF stores one bit, so a group of n FFs is called a n-bit register and stores n bits. The clock is shared amongst all FFs, as is the clear. When the clear is enabled, all outputs are forced to zero. Otherwise, the FFs behave as expected.

Parallel Loads

By adding some logic to a register, we can create a **load** input. When the load is enabled, the data inputs reach the input of the FF, to be loaded when the next clock edge arrives. When the load is disabled, the outout of each FF is fed back into its input and it holds its current value until the load is re-enabled.

Shift Registers

Shift registers act like multiple FFs connected in series. At each arrival of the clock's active edge, the data shifts one FF forward. With enough manipulation, we can create **universal shift registers** which will shift data forward or backward, hold data, or act as a parallel load.

Counters

A counter is a register whose outputs go through a predetermined sequence of states up the arrival of the active edge of a clock or other signal. Note that counters may or may not be synchronous.

Ripple counters consist of a series of FFs where the output of one FF is used as the clock for the next FF. The lack of a common clock signal for each FF makes this counter asynchronous.

Ripple counters can be made to count in binary up to a certain number, then repeat. SUch elements are called **binary ripple counters**, and one with n bits can count from 0 to $2^n - 1$. We have up and down counters, which (obviously) count either upwards or downwards.

Note that it can take a lot of time for the higher order bits of a ripple counter to change (since FFs have CTOs).

We can also design synchronous counters by using the same clock signal for all of our FFs.

It may sometimes be useful to set an initial value for our counter. We do this be using a **load** input.

Since we sometimes may not want to count all the way to $2^n - 1$, we may sometimes wish to implement **modulo counters** (example: zero through ten, then repeat). We could either design this with synchronous principles, or be a bit trickier: we can use additional circuitry to detect our maximum count number, and use a parallel load to restart the counting.

Another useful type of counter is the **ring counter**. It has multiple outputs, only one of which is active at any given time. These can be useful to generate timing signals that indicate an order of operation in another circuit block. We can design a ring counter by creating a shift register, and taking the output from between each register. Alternatively, we could connect a decoder to the output of a simpler counter.

A variation of this is the **switch-tail ring counter**. By feeding the complement of the final output back into the first FF instead of the original output, we create a counter which counts up and then down, over and over.

By adding additional detection circuitry to this counter, we can create a **Johnson counter**,