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	$\overline{x}\overline{y}\overline{z}$	m_0	$x+y+z$	M_0
0	0	1	$\overline{x}\overline{y}z$	m_1	$x+y+\overline{z}$	M_1
0	1	0	$\overline{x}y\overline{z}$	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	$x\overline{y}\overline{z}$	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

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

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 $!(m_0 + m_2) = !m_1 !m_3 = M_1 M_3$ 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.

Clocked Synchronous Circuits

Moore Machine

A Moore machine is a circuit whose outputs depend solely on the currect state of the circuit. In a Moore Machine, outputs change synchronously with the clock, as does the state.

Mealy Machine

If our circuit depends on both the current state and the current input values, we call it a Mealy Machine. In a Mealy Machine, outputs change asynchonrously with the clock, but the state remains synchronous.

State Diagrams

We can pictorially describe a sequential circuit with a state diagram. This diagram illustrates the possible states of the circuit, the possible transitions between them, and the circuit output values.

Possible states are represented with **state bubbles** and transitions are represented with **directed edges** between them. Circuit output values are labelled either inside the bubbles or on the edges, depending on which type of machine we are describing.

In a Moore Machine, a circuit's outputs are a function of only the current circuit state, thus they can be labelled within the state bubbles. In a Mealy Machine, the labels must appear on the edges of the diagram, since the output is a function of both the current state and th current inputs.

A state diagram heavily resembles a mind map, with arrows (directed edges) pointing to each reachable state.

State Tables

State tables (also called **transition tables**) describe the behavious of a sequential circuit in a tabular format. It serves the same purpose as does a state diagram.

Current State	Next State		Output	
	$X = 0$	$X = 1$	$X = 0$	$X = 1$
00	00	01	0	0
01	10	01	1	1
10	00	11	0	0
11	10	01	0	1

Synchronous Circuit Analysis

Given a circuit containing combinational logic and FFs, analysis involves figuring out the state table/diagram associated with that circuit.

The basic procedure is the following:

1. Identify the FFs used to hold the current state information.
2. Identify the outputs of the circuit.
3. Find the logic equations for the circuit outputs and the FF inputs (these are the next state equations).
4. Derive a state table which describes the next state and circuit outputs.
5. Obtain a state diagram from a state table.

Note: we sometimes refer to FF input equations as the **excitation equations**.

We use the reverse procedure for **synchronous circuit design**:

1. Create a state diagram/table from the problem description (states may only have symbolic names at this point).
2. Reduce the number of required states to get a circuit with fewer FFs.
3. Assign unique binary values to represent each state.
4. Select FF types to store the state information.
5. Derive simplified output equations.
6. Derive the next-state/FF input equations.
7. Draw a schematic of the resulting circuit.

Note that using DFFs is the easiest choice in most cases, since the input equations are also the next state equations. In some cases, though, using TFFs or JKFFs will result in logic equations which are much simpler than if DFFs were used. JKFFs tend to be the simplest option in most cases, since they tend to have many don't cares in their K-Maps. Note that this changes our design procedure.

For DFFs:

1. Look at the state table and derive equations for the next state given the current state and the circuit inputs.
2. Connect the next state logic directly to the DFF inputs.

For non-DFFs:

1. Look at the state table to see how the current state changes to the next state for a gven state and inputs.

2. Examine the excitation table for the FF to determine how the FF input must be set to get the desired change from the current state to the next state.
3. Derive logic equations for the FF inputs given the current state and inputs and connect these equations directly to the FF inputs.

Important note: when not using DFFs, the logic applied to the FF inputs are not the next-state equations, but instead *generate the correct next state in conjunction with the FF behaviour*.

Excitation Tables

We can rewrite the behaviour of all FFs in terms of how the output changes given the current inputs to the FFs. Changes in FF output depending on FF inputs is done via an excitation table.

For example, we have the standard DFF excitation table given by

D	$Q(t)$	$Q(t+1)$
0	0	0
1	0	1
0	1	0
1	1	1

Output Encoding

It can sometimes be beneficial to manipulate the circuit such that the outputs of our FFs are directly equal to the output of the circuit. If this is our goal, we must combine multiple states with identical output and use extra bits to distinguish between them. This will virtually always cause us to use extra FFs, but eliminates the need for output equations.

One common type of encoding is **one-hot encoding**. We useone FF per state, and have only one FF with an active output at any given time. One-hot encoding often reduces logic required for output equations and next-state equations, but uses more FFs.

State Reduction

In generating state tables, we can find ourselves with more than the required number of states. Sometimes, these states can be funtionally equivalent to each other, and thus can be combines into a single state.

Two states are said to be equivalent if

- for each circuit input, the states give exactly the same outputs, and

- for each circuit input, the states give the same next state (or an equivalent next state).

There are two methods for state reduction: implication charts and merger diagrams, and paritioning.

Implication charts are half-grids in which we put the next-states for each possible state. By iterating through the chart, we find the definitely equivalent states and the definitely not equivalent states. Further iterations allow us to find the more hidden equivalent states, by crossing out the states which can not be equivalent due to differing implied decisions.

We then generate a **merger diagram** from the completed iplication chart. The merger chart is a pictoral representation of each state, with lines connecting the ones which should be merged.

To reduce a circuit with **partioning**, we

1. Group states according to the outputs the produce.
2. For each group, consider each input pattern: if, for any input pattern different states in a group result in a transition to a different group, those states are not equivalent. We divide these into separate groups.
3. We repeat the last step recursively until all the states in each group transition to the same other groups for any input pattern.
4. Once we've finished, any states in the same group are equivalent.

Algortihmic State Machines

An algorithmic state machine (ASM) is an alternative to a state diagram which can be helpful when our hardware is implementing an algorithm. It can be drawn as a flowchart.

An ASM is tied in closely with a hardware implementation, especially if we use DFF and one-hot encoding. In this way, we can go directly from an ASM chart to a circuit; each element in an ASM chart has a direct circuit translation.

An ASM chart consists of three types of elements: the **state box**, **decision box**, and **conditional output box**.

State Boxes

The state box is equivalent to the state bubble in a state diagram. It can have both a name and a binary encoding. Moore machine outputs can be shown inside of the box, while the state name and binary code are drawn at the state box's input.

When we are using DFFs and one-hot encoding, state boxes become DFFs.

Note that we only label non-zero outputs.

Decision Boxes

The decision box represents a choice that depends on one or more inputs to the circuit. By custom, the left-most output is false and the right-most is true.

When we are using DFFs and one-hot encoding, decision boxes become AND and NOT gates: the entry is ANDed with the input and its complement to determine the decision and its complement.

Conditional Output Boxes

The conditional output box allows for specifying outputs that depend on both the current state and the current inputs. In other words, it contains Mealy outputs.

When we are using DFFs and one-hot encoding, we simply split the circuit at that location and output it without interrupting the rest of our circuit.

If the ASM has conditional output boxes, it describes a Mealy Machine. If not, it describes a Moore Machine.

Design with ASMs

The design of a circuit from an ASM chart is similar to generating one from a state diagram. The difference is in how we generate the state table. When generating a state table from an ASM chart, we must

1. Determine the number of states from the the number of state boxes.
2. Perform state assignment for the state boxes.
3. Use the conditional output boxes, state boxes, and decision boxes to determine output values.
4. Use the decision boxes to determine the next state from the current state.

Asynchronous Sequential Circuits

Recall that asynchronous sequential circuits are a type of circuits without clocks, but with memory. Asynchronous sequential circuits resemble combinatorial circuits with feedback paths.

A circuit is operating in **fundamental mode** if we assume/force the following restrictions on how the inputs can change: only one input may change at a time, and the input changes only after the circuit is stable. A circuit is **stable** if, for a given set of inputs, the circuit eventually reaches a state such that the excitation and secondary variables are equal and unchanging.

Asynchronous Circuit Analysis

After identifying an asynchronous circuit, we obtain **transition tables** or **flow tables**. We can also use state diagrams to describe asynchronous circuits.

Transition tables show the excitation variables and outputs as functions of inputs and secondary variables. Flow tables are simply transition tables with th ebinary numbers replaced by symbols. A flow table with only one stable state per row is called a **primitive flow table**. Note that we tend to circle the stable states for both transition and flow tables.

Our procedure for the generation of either type of table is given by

1. Identify feedback paths.
2. Label excitation variables at outout and secondary variables at input.
3. Derive logic expressions for the excitation variables in terms of circuit inputs and secondary variables. Do the same for circuit outputs.
4. Create a transition table and flow table.
5. Circle stable states (i.e. states where the excitation variablesa are equal to the secondary variables).

Note that the method of analysis is similar when latches are present

1. Label each latch output and feedback path.
2. Derive logic equations for the latch inputs.
3. Check if $SR = 0$ for NOR latches and $\overline{S} \cdot \overline{R} = 0$ for NAND latches. If not satisfied, the circuit may not function correctly.
4. Create logic equations for latch outputs using the behaviour of a latch ($Y = S + \overline{R}y$ for NOR latches and $Y = \overline{S} + Ry$ for NAND latches).
5. Construct a transition table using the logic equations for the latch outputs and circle stable states.
6. Obtain a flow table, if desired.

Asynchronous Circuit Design

Given a verbal problem description

1. Obtain a primitive flow table.
2. Reduce the flow table to get a smaller table with fewer states.
3. Perform state assignment to obtain a transition tables. Make sure to avoid **race conditions** (explained later).
4. Obtain next-state and output equations. Make sure to avoid hazards and glitches.

5. Draw circuit with or without latches.

State Minimization in Asynchronous Circuits

There are lots of opportunities for state minimization in asynchronous flow tables since there tend to be many don't care outputs for unstable states as well as don't care next-state information if we assume fundamental mode operation.

With don't cares, equivalency is replaced with **compatibility**. Two state are compatibile if, for every possible input combination, they produce the same outputs where specified and the have compatible next states where specified.

In essence, don't cares match with anything.

We follow the same procedure as with synchronous circuits (i.e. implication tables and merger diagrams), and upon creating the merger diagram may find ourselves with many interconnected states.

We now look for large **cliques** (parts of the graph in which every node is connected to every other node) and merge them. Note that we need to ensure the implied compatibilities are still true given our selected cliques, and may have to chose sub-optimal ones.

Race Conditions

A race condition occurs in an asynchronous circuit when two or more state cariables change in response to a change int he value of a circuit input. Unequal circuit delays may imply that the two or more state variables may not change simultaneously - this can cause problems.

Assume two state variables change. If the circuit reaches the same final, stable state regardless of the order in which the state variables change, then the race is **non-critical**. If the circuit reaches a different final, stable state depending on the order in which the state variables change, then the race is **critical**.

We need to avoid critical races for predictability and to ensure our circuit does the intended function.

Races are a consequence of how states are assigned when designing a circuit. In other words, races exist in transition tables, but not in flow tables. As such, we can pre-emptively prevent races by performing state assignment in such a way that transitions from one stable state to another stable state only require one state variable to change at a time.

A useful method for doing so is to try to create a n-dimensional cube from a flow table (by introducing unstable, intermediary states) and follow along the paths it creates. Alternatively, we can use a one-hot encoding scheme to avoid the problem.

Hazards

A hazard is a momentary unwanted switching transient at a logic function's output. Hazards occur due to unequal propogation delays along different paths in a combinatorial circuit. There are two types of hazards: **static** and **dynamic**.

For asynchronous circuits in particular, hazards can cause problems in addition to other issues like races and non-fundamental mode operation; momentary false logic function values in an asynchronous circuit can cause a transition to an incorrect stable state.

Static-0 Hazards

Static-0 hazards occur when the output is zero and should remain at zero, but temporarily switches to one due to a change in input.

Static-1 Hazard

Static-1 hazards occur when the output is one and should remain are one, but temporarily switches to zero due to a change in input.

Dynamic Hazards

A dynamic hazard occurs when an input changes and an output should change from one to zero or vice-versa, but temporarily flips between values.

Fixing Hazards

When circuits are implemented as two-level POSs or SOPs, we can detect and remove hazards by inspecting the K-Map and *adding redundant terms*. This increases our circuit complexity, but removes the hazards.

For two-level circuits, if we remove all static-1 hazards using the K-Map, we are guarantedd that there will be no static-0 or dynamic hazards.

Note that two-level circuits are easy to deal with (i.e. hazards can be easily removed), but the situation is much more difficult with multi-level circuits in which there are multiple paths from an input to an output.

We can either fix multi-level circuits by reducing it two a two layer one, or by using SR or $\overline{S} \cdot \overline{R}$ latches. An SR latch can tolerate momentary zeroes appearing at its inputs, and vice versa, so we can use an SR latch to fix static-1 hazards (and vice versa).

Output Assignment

FLow and transition tables might have unspecified entries for circuit outputs due to either fundamental mode assumptino or unstable states. Depending on the output equations we derive, we might end up having **glitches** at our circuit outputs. Obviously, glitches are bad; they could get fed into another circuit causing problems and they waste power.

To avoid glitches, we need to be careful when minimizing flow tables: given a don't care, if both stable states produce the same output, change the don't care to that output. If not, it can remain a don't care.

CMOS

One popular technology for implementing logic gates (via transistors) is **Complementary Metal Oxide Semiconductor** technology. CMOS uses two types of **Metal Oxide Semiconductor Field Effect Ransistors (MOSFETs)** to implement logic gates: n-channel and p-channel transistors.

Logic levels are represented with voltages: the ground and highest voltage states for 0 and 1, respectively.

NMOS Transistors

A simplified NMOS transistor has 3 terminals: the gate, source, and drain. The source is at a lower voltage and the drain is at a higher voltage.

When a high voltage is applied to the ground (with respect to the source), and this voltage difference is above some threshold, the switch closes and the two are connected such that current flows from the drain to the source. This equalizes the two voltages to that of the source voltage. When the voltage difference is less than that threshold, the switch opens and no current flows between the two.

PMOS Transistors

A simpified PMOS transistor has the same three components as an NMOS transistor, but the relative voltages between the source and drain are switched.

When a *low* voltage is applied to the ground)with respect to the source), and this voltage difference is above some threshold, the switch closes, the drain and source are connected, and their voltages both equalize to the source voltage. Again, a voltage difference less than the threshold causes the switch to open.

CMOS Structure

CMOS combines the two MOSFETs in a structure which consists of a **Pull-Up Network (PUN)** and a **Pull-Down Network (PDN)** to implement logic functions. Note that PUNs and PDNs are duals of each other.

A connection between the high voltage source gives a high output, and a connection to the ground gives a low output. We implement ANDs and ORs with transistors in series and parallel, respectively.

CMOS Inverter

Assume some input is connected to an NMOS and a PMOS. A high input will open the PMOS, and a low input will open the NMOS. To create an inverter, then, we connect the high voltage to the output through the NMOS, and vice-versa.

CMOS NAND and NOR

We can implement NANDs and NORs by translating out knowledge of circuit design to CMOS systems. In CMOS, these two gates are virtual opposites, as each is created by having two of one type of gate in series and two of the other tpe in parallel.

Transistion Gates

Transition gates are gates which connect the input to the output, given a certain high second input. They are implemented with an NMOS and a PMOS of opposite input, and are drawn as a diamond enclosed in four triangles.

ROM

Read-Only Memory is a device which allows for the permanent storage of information. The device has k input lines and n output lines. They are also refered to as address and data lines. We can store $2^k n$ bits of information inside the device. The address lines specify a memory location and the data outputs at any given time represent the value stored at that memory location.

We can implement multi-input and -output logic functions inside of ROM, since we can treat the address lines and data as the inputs and output of a logic function.

We draw ROMs as block diagrams, but by examining the contents of the block, we can see how it could be used to implement multiple logic functions.

The inputs of a ROM are fed into a decoder ($k \to 2^k$), which gives 2^k outputs as rows. One column for each function is created, and the proper outputs are created at the intersection of these function columns and the decoded rows. We use a read buffer at the base of each column (another input to the ROM), and give the output of ach function as a unique output.

Implementation of ROM

There are several methods for implementing ROMs

- **Programmable Read-Only Memory** (PROM) contains fuses giving high logic values to all bits. Programming a PROM involves blowing fuses to give some bits a zero. Obviously, this is permanent.
- **Electrically Programmable Read-Only Memory** (EPROM) can be erased by exposure to UV light, but otherwise acts the same as PROM.
- **Electrically Erasable Programmable Read-Only Memory** (EEPROM) can be erased electrically, but is otherwise the same as PROM.

Each of these devices has an extra pin (or pins) where the programming information (or bit stream) is applied to do the programming.

RAM

Read-Only Memory is a storage device which can both read and write information. It is the same as a ROM, with the addition of having a "write" bit and n data inputs.

Programmable Logic Devices

There are many difference programmable logic devices available. The idea behind these devices is that they are "generic", in that they can be programmed to implement a wide variety of types of digital circuits.

PLA

A **Programmable Logic Array** is capable of implementing SOP functions. It consists of input buffers and inverters followed by a programmable AND plane and a programmable OR plane. It can implement m logic functions of n variables. The limit is the number of product terms which can be generated inside the device.

PAL

The **Programmable Array Logic** is similar to the PLA, but has a fixed OR plane. As such, it is less flexible than a PLA, but requires less programming. Often, the outputs are fed back internally and can be used to create product terms and create multi-level logic functions.

SPLD

A **Simple Programmable Logic Device** can be created as follows: We implement a sequential circuit by taking a PLA or PAL and adding some FFs to the output of the OR plane such that it feeds through a FF to the output, and is then (optionally) fed back into the PLA or PAL. This sequential circuit is referred to as a **MacroCell**, and we create a SPLD by having several MacroCells together in the same IC.

CPLD

PLAs, PALs, and SPLDs typically contain small numbers of outputs (i.e. 2^4) with many inputs (i.e. 2^5) and many product terms. As such, these are only useful when each equation has a wide fanin.

A **Complex Programmable Logic Device** can be used to implement large and complicated circuits, and is created by connecting many SPLDs with a **Programmable Routing Fabric**.

Types of PLDs

In addition to implementing the above devices with PRAM, EPRAM, and EEPRAM, it is also possible to have SRAM-based PLDs. In SRAM devices, the programming data is lost when the power is turned off, and so it is necessary to reprogram the device each time the system is powered on.

We often see a configuration EPROM beside an SROM-based PLD on a circuit board: this is a two chip solution wherein we hold the program on the EPROM and apply it to the PLD upon power up.

FPGA

A **Field Programmable Gate Array** is another type of programmable device capable of handling large circuits. It is different from a CPLD in that logic is impleneted using a **Lookup Table** (LUT), which consists of a large number of small memory spaces. FPGAs are typically SRAM-based.