CPS311 Lecture: Introduction to Combinatorial Logic

last revised June 10, 2021

Objectives:

1. Ability to read simple logic diagrams
2. Ability to design simple combinatorial circuits

Materials:

1. Samples of logic technologies
2. Projectable of logic axioms
3. Circuit Sandbox for demos - including prebuilt circuits - Inverter, Two Input Gates, 4 Input NAND, A xor BC (five parts),
4. Setup with scope, etc. to demonstrate physical properties of gates (needed for final session devoted to this topic)

I. Introduction

A. Today, we begin discussing the digital components that provide that are used to create the implementation of a computer system. We will consider these in two groups:

1. Combinatorial logic: components whose output at any time is just a function of its input at that time: gates.

2. Sequential logic: components that have some sort of internal state, so that their output at any given time is a function of the history of inputs they have received: flip-flops and memory.

B. From physics we learn that all the complexity of the universe is built up from combinations of certain fundamental building blocks: electrons, protons, neutrons (or quarks if you will.) In like manner, all the complexity of digital computers is built up from fundamental building blocks like GATES.

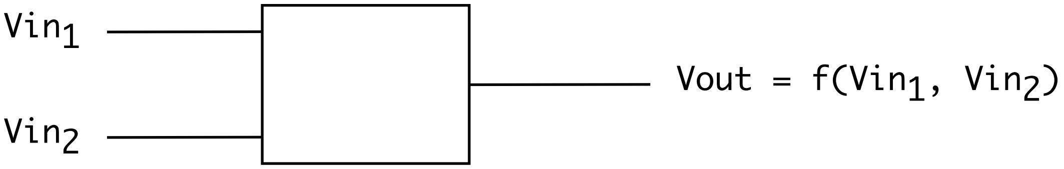
1. A gate is an electronic circuit having one or more inputs and one output, each of which can be in one of two states, commonly called 0 and 1. Further, the output at any time is some function of the inputs.

2. Gates can be realized in many different ways, but we will say little about this. We should note that the generations of computer hardware are largely distinguished by the technology used to realize gates: relays (0); vacuum tubes (1); transistors (2); integrated circuits (3); very large scale integration (4). With all pre-IC technologies, a gate would consist of a number of discrete components wired together on a circuit board (cf sample transistor board); with IC technologies, one or more gates are completely contained on a silicon chip. With today’s technology, we may have one chip containing millions of gates.

C. For our purposes, we will think of a gate as a “black box” having one or more inputs plus a single output.

1. The voltage at the output of the gate is some function of the input voltages.

2. Example: a two-input gate



3. Gates are designed so that they respond to two discrete voltages - one of which is associated with the value 0 (false) and the other with the value 1 (true). With the kinds of gates we will be be using in lab (TTL and CMOS), the value 0 (as input or output) will normally be represented by a voltage level close to 0. The value 1 will normally be represented by a voltage level of about 5 volts. (In more recent computers, the there has a been a trend to use a lower power supply voltage - and hence lower value for 1 - to conserve power - e.g. 3.3 to as low as 1.2 Volts).

4. In actual practice, any gate technology will accept a RANGE of voltages for each logic value. For example, with TTL gates of the kind we will be using in lab, the following rules hold:

a) Any voltage between 0 and 0.8 Volts is regarded as low, and is interpreted as a zero.

b) Any voltage between 2 and 5 volts is regarded as high, and is interpreted as a one.

c) Any voltage between 0.8 and 2 volts, or if an input is not connected at all, constitutes an undefined input. The output of any gate which has any undefined input can be 0, 1, or undefined itself. (This may be represented by the symbol "X")

d) Any voltage less than 0 or greater than 5 volts is illegal, and may physically destroy the gate.

e) This turns out to be a fundamental distinction between digital and analog devices. Because any value in a certain range is interpreted as a particular discrete value (0 or 1), digital systems have a high tolerance for electrical “noise”. On the other hand, when digital systems fail, they tend to fail catastrophically. (Contrast the signal on an old VCR tape versus a DVD disk; consider also the effect of a scratch on a DVD.)

5. The output of a gate whose inputs are all defined will normally always lie in either the defined 0 or 1 range.

a) However if the outputs of two or more gates are connected, with at least one trying to output a 0 and the other a 1, then the output is itself is undefined.

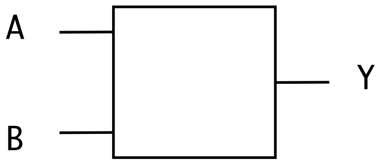
b) The exception to this statement is that there is a type of gate called a tri-state gate whose output can be in one of 3 states: 0, 1, or floating (denoted by the symbol "Z"). If the outputs of two or more tri-state gates are connected, and only one is actually output a 0 or 1 (the rest are floating), then the value of the output signal will be the one specified by the non-floating gate. (We will see examples of this later in the course.)

6. Actually, the convention we have been describing is one of two possible options, known as positive logic. It is also possible to adopt the

convention of letting a low voltage (between 0 and 0.8 volts) represent a 1, and a high voltage (between 2 and 5 volts) represent a 0. This yields negative logic - but we won't discuss this further now - however, we will run across examples where this is used in later labs.

7. To keep our discussion simple, we normally describe the inputs and outputs of a gate as being 0 or 1 (or sometimes L = "low" or H = "high", rather than worrying about the specific voltages.

D. We will represent the behavior of a gate by a table of combinations or truth table which shows the output of the gate for each possible combination of inputs. For example, a 2 input AND gate has truth table:

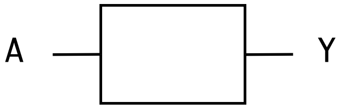


Note: we are using a nondescript box. Later we will see that there is a special shape for certain types of gate.

Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

1. Note that for a gate with 1 input there are 4 possible truth tables, representing:

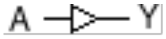
   - a gate whose output is always 0, regardless of input
   - a gate whose output is always 1, regardless of input
   - a gate whose output is the same as its input
   - a gate whose output is the opposite of its input:



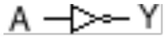
Input A	Output Y
0	1
1	0

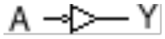
Of these, only the last two are of any interest at all, and he last one is most useful. (The first two are sometimes called “stuck at 0” and “stuck at 1”, and typically arise from a gate being defective for some reason.)

a) The gate whose output is the same as its input is often used in situations where it is necessary to amplify a signal to drive a number of gates. Such a device is called a BUFFER:

symbol: 

b) The gate whose output is the opposite of its input is called an INVERTER. Two symbols can be used for this function:



or


Note use of the "bubble" to symbolize inversion of the logic sense. In the first case, the symbolism is that the output of the gate is the opposite of that from a simple buffer. In the second case, the symbolism is that the output of the gate is the same as that of a simple buffer whose input has been inverted. Obviously, the two symbols describe two ways of looking at the SAME BEHAVIOR, and so can be used interchangeably to describe the same physical circuit.

DEMONSTRATE (File Inverter)

2. For a gate with 2 inputs, there would be 16 possible truth tables. (For each of the 4 combinations of the 2 inputs, we could choose one of two outputs. Thus, there are 2^4 possible truth tables. Again, only a limited number are of interest - we defer discussion until later.)

3. In general, for a gate with n inputs, how many truth tables are possible?

   ASK

   There are 2^{2^n} possible truth tables - e.g. for 3 inputs there are 2^{2^3} $= 2^8 = 256$ possible truth tables.

II. **Boolean Algebra**

A. To describe the behavior of networks of gates, we use a notational system called boolean algebra (or switching algebra). In this system, we have only two values, 0 and 1, and three primitive operations:

1. Inversion or complementation, written $\bar{A}$ or A'. (Note - in these notes I will use the symbol A' because it's easier to type! The book consistently uses $\bar{A}$ - pronounced "A-bar"):

A	A'
0	1
1	0

2. Addition or Logical or, written A + B or $A \cup B$ (I'll use + in these notes, which is also the notation the book uses)

A	B	A + B	
0	0	0	
0	1	1	
1	0	1	
1	1	1	Note: 1 + 1 = 1!

3. Multiplication or Logical and, written A • B or A * B or A ∩ B or AB (I'll use • in these notes. The book doesn't use any special symbol - just writes something like AB):

A	B	A • B
0	0	0
0	1	0
1	0	0
1	1	1

B. As in any algebraic system, more complex operations can be built up out of the primitive operations, with operator precedence rules or parentheses for grouping - e.g:

A' + A•B = (A') + (A•B) - truth table:

A	A	A'	AB	A'+A•B
0	0	1	0	1
0	1	1	0	1
1	0	0	0	0
1	1	0	1	1

C. As in other algebras, there are certain properties that govern the behavior of more complicated expressions, such as associativity, commutativity, and distributivity. However, whereas these are axioms for most algebraic systems, for boolean algebra they are theorems since any one of them can be proved from the definitions above by exhaustion (perfect induction). Here are some of the key properties:

(Note: there are several others that appear in the book but are omitted here)
PROJECT STEPPING THROUGH

∀ A, B, C:

Identity:	A • 1 = 1 • A = A A + 0 = 0 + A = A
Null Element:	A • 0 = 0 • A = 0 A + 1 = 1 + A = 1

Idempotence:	$A + A = A$ $A \bullet A = A$
Involution:	$(A')' = A$
Complements:	$A \bullet A' = 0$ $A + A' = 1$
Commutativity:	$A + B = B + A$ $A \bullet B = B \bullet A$
Associativity:	$A + (B + C) = (A + B) + C$ $A \bullet (B \bullet C) = (A \bullet B) \bullet C$
Distributivity:	$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$ $A + (B \bullet C) = (A + B) \bullet (A + C)$!!
Demorgan's Theorems:	$(A + B)' = A' \bullet B'$ $(A \bullet B)' = A' + B'$

D.As an illustration of proof by perfect induction, we prove the first form of Demorgan's theorem:

A	B	A+B	(A+B)'	A'	B'	A'•B'
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

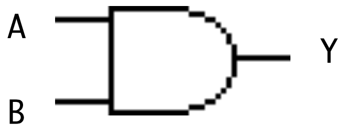
Notice that, for all possible values of A and B, the (A+B)' and A'•B' columns are the same - implying that (A+B)' = A'•B' for all A and B

III.**Realization of Boolean Functions - Basic Types of Gates**

A. Crucial to the design of computer systems is the ability to realize circuits composed of gates whose output corresponds to some boolean function of its inputs. To do this, we usually use certain types of gates as building blocks, with more complex functions realized by combinations of gates in which the output of one gate becomes an input to another.

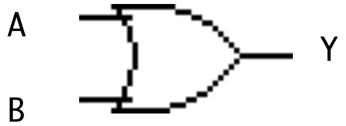
B. While we could theoretically implement any truth table as a hardware primitive, most designs are built around a limited number of basic functions implemented in hardware. We have met one already, the inverter. Two-input functions of interest include:

1. 2-input AND:



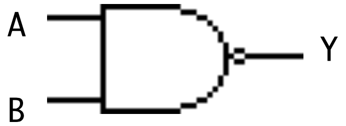
$Y = A \bullet B$

2. 2-input OR:



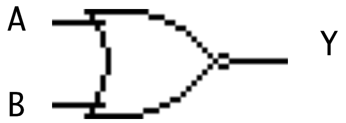
$Y = A + B$

3. 2-input NAND:



$Y = (A \bullet B)'$

4. 2-input NOR:



$Y = (A + B)'$

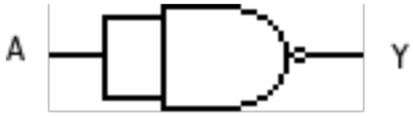
DEMONSTRATE each of the above using Circuit Sandbox. (File Two-Input Gates)

5. The latter two are of particular interest, for two reasons:

   a) Most of the transistor circuits used in realizing gates have the effect of inverting the signal passing through them. Thus, NAND and NOR can generally be realized more simply than AND or OR.

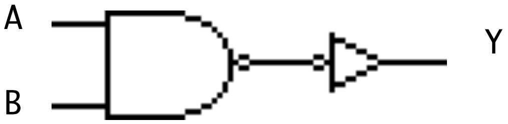
   b) NAND and NOR are logically complete. Given either one, it is possible to realize any boolean function. Examples

(1) Invert realized using NAND:



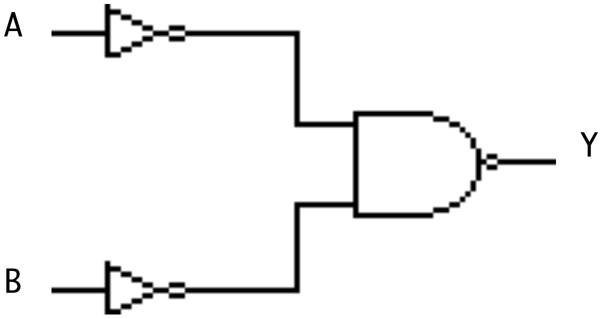
Y = (A • A)' = A'

(2)AND realized using NAND:



Y = ((A • B)')' = A • B

(3)OR realized using NAND:

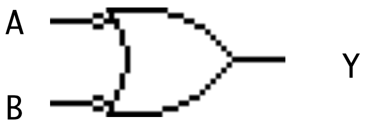


Y = (A' • B')' = A + B
(by DeMorgan's theorem)

DEMONSTRATE EACH OF THE ABOVE (Create on screen)

6. For any particular gate technology, usually one of the latter two will be more easily realized. In the case of the TTL technology we will be using in lab, this is NAND. For consistency, we will work most of our examples for the rest of this unit using NAND.

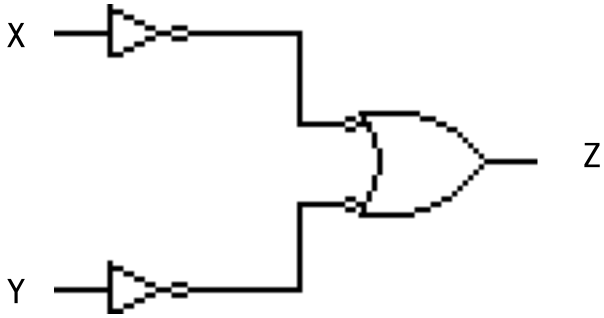
7. Note that DeMorgan's theorem and the above construction suggests an alternate symbol for NAND:



Y = A' + B' = (A • B)'

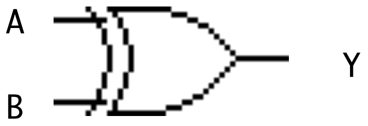
This symbol says that the output of a NAND gate is the same as that of an or gate presented with the inverse of the inputs presented to the NAND. (The first symbol said that the output of a NAND gate is the inverse of that of an AND gate presented with the same inputs.)

Note that both symbols describe exactly the same gate - a NAND gate may be drawn either way. By convention, the symbol that is chosen is the one that most clearly represents the intended use of the gate - e.g. the above diagram for an OR implemented with NANDS is more conventionally drawn as follows:



C. Two other functions of two inputs are of some interest, though less than the previous four

1. 2-input XOR:



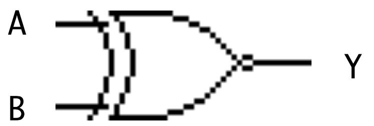
$Y = A \oplus B$

Truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

DEMONSTRATE (using file Two Input Gates)

2. 2-input XNOR (Coincidence)



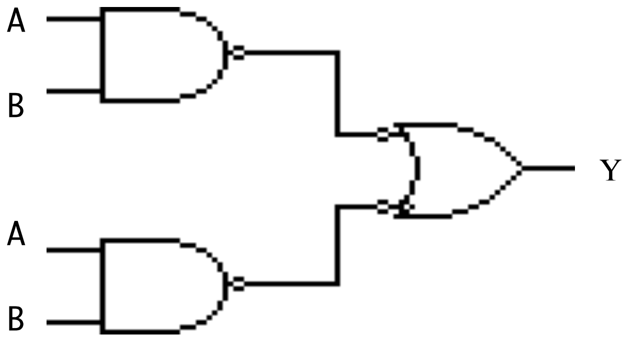
$Y = (A \oplus B)'$

Truth table:

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

Note that A XNOR B is true iff A = B. (XNOR and XOR are complements)

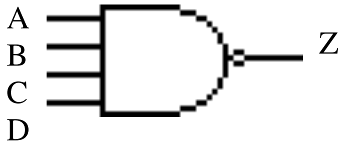
3. Both can be realized using other types of gates. For example, the following is a realization of XOR using NAND - assuming that both the inputs and their complements are available. (If not, two inverters would also be needed - however, for reasons we will see later, it is often that case that both the uncomplemented and complemented form of each input are readily available).



$$Z = (A'\bullet B)'' + (A\bullet B')'' = A'\bullet B + A\bullet B' = A \approx B$$

DEMONSTRATE (Create on screen)

D. We often find gates with more than two inputs, but they are usually extensions of the types of 2-input gates we have been looking at. For example, 4-input NAND:



Truth Table:

A	B	C	D	Z
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

1. DEMONSTRATE using Circuit Sandbox (File 4 Input NAND)
2. Observe that, interestingly, this circuit outputs a zero just when its four inputs are all 1. The fact that transistor circuits naturally inject a level of inversion into their computation - making NAND easier to implement than AND - leads to the use of negative logic (true = 0, false = 1) in some situations. In this case, we could think of our gate as a sort of “and” gate whose inputs use positive logic and whose output uses negative logic !

IV. Realizing Any Boolean Function Using A Network of Gates

A. Observe that, since we have gates that are directly equivalent to the primitives of boolean algebra, we can convert any boolean expression into a network of gates that realizes it.

Example: $Y = A \oplus (B \bullet C)$

DEMONSTRATE (File A xor BC)

B. Actually, we can realize any boolean function using just AND and OR gates. Observe that any boolean function can be written in sum-of-products form by using its truth table.

Example: the above

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

1. Find each row in the truth table where the output is 1. This will contribute one term to the sum. Thus, this function can be represented as a sum of four products:

2. Each product will contain each of the inputs either plain or complemented. An input will appear plain if the row in question contains a value of 1 for that input, and complemented if it contains a 0. Thus, the fourth row in the truth table (the first containing a 1 output) gives rise to the term A'BC.

   A term that is the product of the true or complemented form of all the inputs is called a minterm.

3. A function equivalent to our original function is therefore

   Y = A'BC + AB'C' + AB'C + ABC'

   a) Note that each term is 1 for exactly one combination of inputs and 0 for all others.

   b) Each term therefore gives us one of the 1's of the function.

   c) This form of expression is called <u>sum-of-products</u> form.

   d) This can then be realized by using a two-level AND-OR network:

      DEMONSTRATE - Second part of File A xor BC

      This realization is called a two-level AND-OR realization, since

      (1)The input must pass through two gate levels before arriving at the output. This is of some significance, since all gates have a certain finite propagation delay. Note that, apart from building a special-purpose gate, two logic levels is the best one can do for most functions.

      (2)It is called an AND-OR realization because the first level consists of AND gates, and the second level is a single OR gate.

C. A boolean function in sum of products form can also be realized by using just one type of gate: the NAND gate.

1. Realize each term by using a NAND gate with as many inputs as there are inputs to the overall function. Each gate input goes either to one of the function inputs or its complement as discussed above. The NAND gate's output is 0 just when the term itself is 1. (Negative logic)

2. Sum all of the terms using a NAND gate (represented - for clarity - in its alternate form) with as many inputs as there are terms.

DEMONSTRATE - NAND version of the above (Third part of file)

Note that the only difference between this and the previous realization is adding "bubbles" to turn the ANDs and the ORs into NANDs.

3. We call this circuit a two-level NAND-NAND realization.

D. While the method outlined above gives a realization that cannot generally be improved on in terms of number of levels, it is not optimal in terms of number of gates.

1. Observe that our sum of products form can be simplified by the properties of boolean algebra as follows:

$$\begin{aligned} A'BC+AB'C'+AB'C+ABC' &= A'BC + A(B'C' + B'C + BC') \\ &= A'BC + A(B'C' + B'C' + B'C + BC') \\ &= A'BC + A(B'C' + B'C + B'C' + BC') \\ &= A'BC + A(B'(C'+C) + C'(B'+B)) \\ &= A'BC + A(B'\cdot 1 + C'\cdot 1) \\ &= A'BC + A(B' + C') \\ &= A'BC + AB' + AC' \end{aligned}$$

2. This yields a simplified two-level NAND-NAND realization

   DEMONSTRATE - Fourth part of file

3. This simplification would result in a considerable cost savings in construction.

   a) The original circuit needed:

   4 3-input NAND gates
   1 4-input NAND gate

   b) The new circuit requires:

   2 3-input gates
   2 2-input gates

This involves both fewer gates and simpler gates, and so would be easier to build from SSI chips (as we will do in lab) or on a VLSI integrated circuit.

E. For functions of four variables or less, this simplification can be done simply by the use of a Karnaugh map.

1. Example, for the above:

AB/C	0	1
00	0	0
01	0	1
11	1	0
10	1	1

(Note order of rows!)

2. Example for four variables:

Y = A'BC'D' + A'B'C' + A'C'D + AB'CD + AB'C' + AB'D'

AB/CD	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	0	0	0	0
10	1	1	1	1

Simplified form: A'C' + AB'

3. In using a Karnaugh map, we cover all of the 1's by using maximal subcubes of one of the following forms:

"8's" completely covering two adjacent rows or two adjacent columns
"4's" completely covering one row or one column
squares of 4 - including ones that wrap around top and bottom or left and right, and ones that involve the four corners

"2's" that cover two adjacent squares horizontally or vertically - including ones that wrap around top and bottom or left and right

4. The method works because the values are ordered in such a way that any two horizontally or vertically adjacent entries differ in exactly one input, whose value is irrelevant if both squares contain a 1.

5. In setting up functions to be minimized using a Karnaugh map, it is often helpful to use a different notation for the function. Basically, we represent each 1 of the function by a decimal number that is the equivalent of the binary number representing the values of the inputs.

   a) Example: Y = A'B'CD + AB'C'D can be written

      $Y(A,B,C,D) = \sum\ (3, 9)$

      because A'B'CD corresponds to an input pattern of 0011 = decimal 3 and AB'C'D corresponds to an input pattern of 1001 = decimal 9

   b) In doing this, one must be sure to include all the variables in each term.

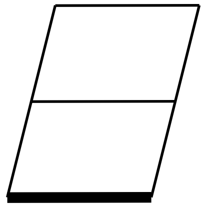
      Example: Y = A'CD + ABC

      must be written as Y = A'B'CD + A'BCD + ABCD' + ABCD

      which yields $\sum$ (3, 7, 14, 15)

F. Often, in designing a logic circuit, it happens that certain input patterns are known to not be possible. Thus, the output produced for these patterns is irrelevant. We refer to this as a don't care condition.

1. Example: a 7-segment decoder translates the binary representation of a decimal digit into 7 outputs corresponding to the 7 segments of an LED display. Consider just the output for the bottom bar:



This needs to be on when the input is 0, 2, 3, 5, 6, or 8, and off when the input is 1, 4, 7, or 9. We don't care what its state is for inputs 10-15, since these don't correspond to any decimal digit and thus presumably cannot occur. Thus, our function for the bottom bar is

$$Y(A_3,A_2,A_1,A_0) = \sum\ (0, 2, 3, 5, 6, 8) + d(10, 11, 12, 13, 14, 15)$$

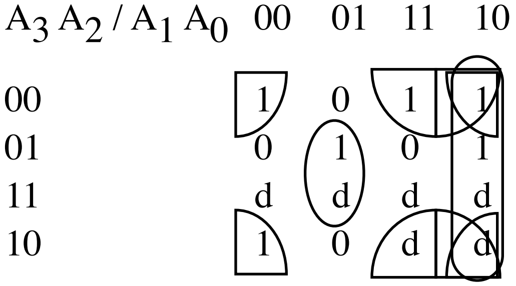
(where d stands for “dont care”)

2. In constructing the map, we put 1's for the 1's of the function, d's for the don't cares, and 0's for everything else.

Example:

$A_3\,A_2$ / $A_1\,A_0$	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	d	d	d	d
10	1	0	d	d

3. In constructing a covering, we must cover ALL the one's, but only cover d's which help us to construct bigger sub-cubes. For example, in the above it would be convenient to cover four of the d's, while leaving two uncovered. This means that the circuit will output 1 for inputs 10, 11, 13, and 14; but 0 for inputs 12 and 15. But since those input patterns do not represent legal decimal digits, we really don’t care what the circuit would do in those cases!



$A_3\,A_2$ / $A_1\,A_0$	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	d	d	d	d
10	1	0	d	d

4. (If time) do another segment as an example

G. When working with NOR logic, it is convenient to use a mirror-image technique:

1. Working directly with the truth table, we can represent function as a product of sums, derived by looking at the zeroes of truth table. Each input appears in ordinary form if the row contains a zero; complemented form if it contains a 1. Example: our earlier example Y = A ⊕ (B • C). We look for the zeroes of the truth table:

A B C	Y
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	0

This yields a product of sums form in which each factor is zero for just one row of the truth table; of course, the overall function is zero just when any factor is zero, and one when all the factors are one:

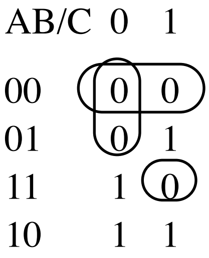
Y = (A + B + C)(A + B + C')(A + B' + C)(A' + B' + C')

2. We can realize with a product of sums with one NOR gate for each factor plus one NOR (drawn in dual form) to multiply all factors.

DEMONSTRATE: Fifth part of A xor BC file

3. We can also use Karnaugh-map minimization, we covering the ZEROES, and put the variable in the expression in straight form if it is a zero and complemented if it is a one.

Example: map for the above:



AB/C	0	1
00	0	0
01	0	1
11	1	0
10	1	1

Simplified expression is (A + C)(A + B)(A' + B' + C')

V. **The Physical Characteristics of Gates (Omit/Compress if Necessary)**

A. In our discussion thus far, we have treated gates as “black boxes” - i.e. idealized devices whose inputs and outputs are one of two discrete values, and whose output at any time is purely a function of its inputs at that time. In reality, of course, gates are physical devices, and their actual behavior differs a bit from the ideal behavior we have discussed. We want to look briefly at a few of the ways that actual physical characteristic impact on functionality.

B. Noise Immunity

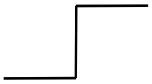
1. We have talked as if gate inputs and outputs are simply 0 and 1. In reality, the input to a gate is a voltage, with any voltage in a certain range being treated as 0, and any voltage in a different range being treated as 1, and any other voltage being prohibited.

   Example: TTL gates of the sort we are using in lab:

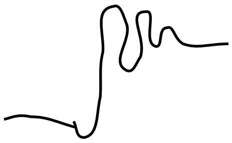
   0 .. 0.8 volts = 0
   2.0 .. 5.0 volts = 1

2. In reality, there is always a certain amount of electrical “noise” in the circuits of which a gate is a part.

a) Some of this comes from the gates themselves. Ideally, when a gate switches from - say - 0 to 1, its output should look like this:

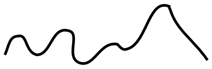


In reality, its output typically looks more like this:



This behavior (known as “ringing”) is a consequence of the physical transitions occurring inside the transistors from which the gate is constructed.

b) Moreover, electrical noise is often induced by other nearby gates. When a nearby gate changes state, a gate that is not changing may still exhibit some fluctuation in its output.



c) Also, other nearby devices - especially those that spark (e.g. motors), radio signals, etc. can also result in fluctuations in the actual voltage on a wire connecting two gates.

3. As long as the electrical noise does not cause the output of a gate to move outside the range that is reliably recognized as 0 or 1 (as the case may be), it should have no effect on other gates. The safety margin is known as the noise immunity of the gate.

Example: TTL gates are typically designed so that their “zero” output is guaranteed to be no greater than 0.4 volt. Since another gate will recognize any voltage less than 0.8 as a zero, there is a margin for error of 0.4 volts. As long as electrical noise does not cause the apparent output to rise more than this, the circuit will behave reliably.

Noise becomes more of a problem as supply voltage drops.

4. However, if there is too much electrical noise present (due to inattentiveness to certain issues in design), the circuit will not be reliable. This is why digital systems typically incorporate various kinds of metal shielding to keep out ambient signals, and why circuit boards typically incorporate devices known as a capacitors at the power pins of chips to minimize interaction between chips through the power supply.

5. Digital systems designed for use in environments where there is a large amount of electrical noise (e.g. systems exposed to nuclear radiation, systems that control electrical devices such as motors) require special "hardening" to prevent problems. Absent this, a system that performed reliably in design may fail in surprising ways in the field.

C. Fanout

1. We have designed networks of gates in which the output of one gate may be connected to the input(s) of some number of other gates. However, the number of gates one gate may drive is not unlimited.

2. The term fanout refers to the maximum number of other gates of the same type one output is designed to drive. For example, the TTL gates we are using in lab are typically have a fanout of 10 - one output of a particular gate may be connected to a maximum of 10 inputs of other gates.

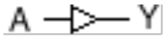
3. Exceeding the design fanout can result in a reduction in the noise margin; exceeding it sufficiently far can produce unreliable behavior. (I.e. a gate may be outputting a zero, but other gates may see the output as undefined and may interpret it as one instead.)

4. What does one do if a single output must drive more inputs than the fanout allows?

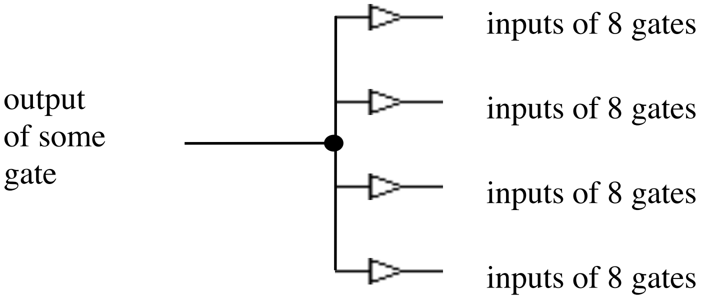
   For example, what if we are using gates designed for a fanout of 10 in a circuit where one output has to drive 32 inputs?

ASK

5. This is a case where we need to make use of a buffer - a gate whose output is identical to its input:



What we do is have the gate in question drive some number of buffers, each of whose output drives (up to) the maximum number of inputs.



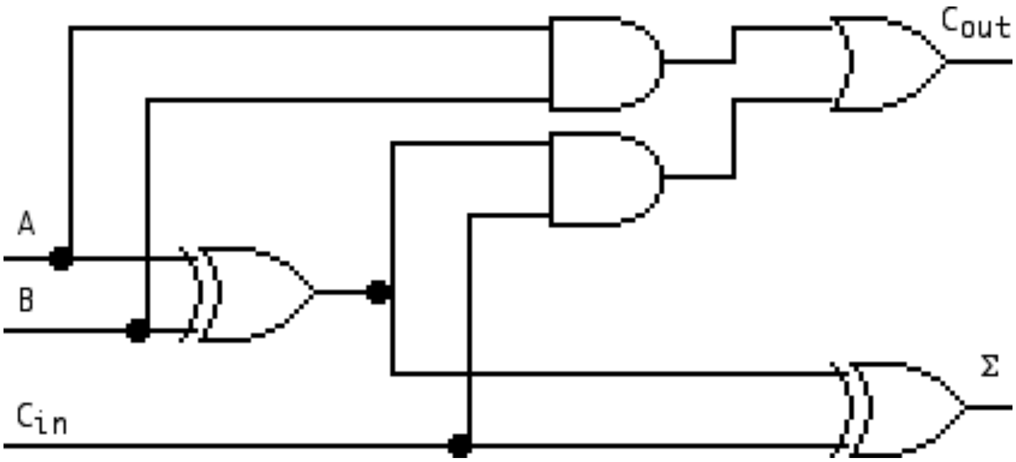
D. Propagation Delay

1. In our ideal description of gates, we assumed that the output at any point in time reflects the inputs at that same time. In reality, as physical devices, all gates exhibit a finite propagation delay, such that when the input changes, the corresponding change at the output occurs some amount of time later.

   Example: the typical propagation delay for TTL gates of the sort we are using in lab is about 10 ns.

2. Of course, when gates are interconnected in such a way that the output of one gate is connected to the input of another, the overall propagation delay for a network is the <u>sum</u> of the propagation delays down the <u>longest</u> (in terms of time) path from input to output in the network.

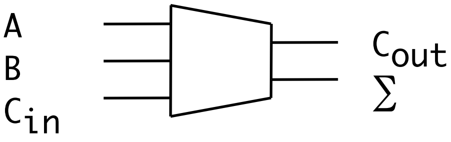
Example: consider a circuit known as a full adder, that is used in doing arithmetic addition of two binary numbers:



PROJECT

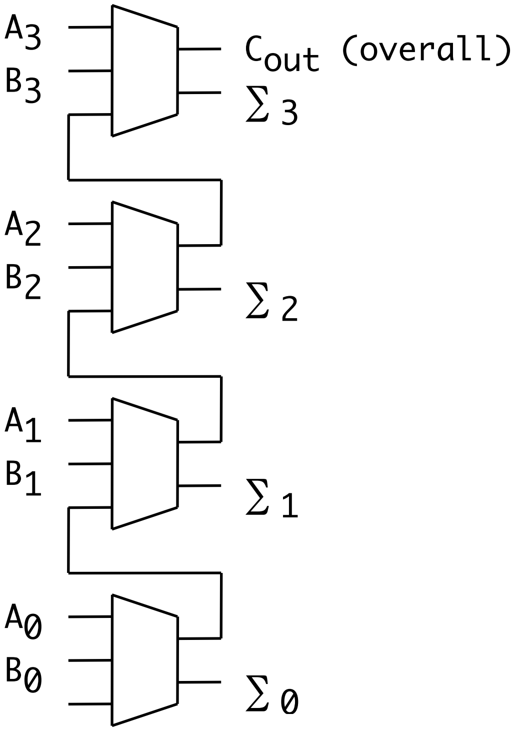
a) The full adder handles one bit of calculating a sum, and has inputs for one bit of each of the numbers being added plus a carry from the next least significant bit.

b) This is one of the building blocks we will talk about soon, and has a special symbol:



PROJECT

c) N full adders can be interconnected to add 2 N-bit numbers to produce an N-bit sum and a carry out, as follows (for simplicity doing only 4 bits in this case)



PROJECT

d) Suppose that the propagation delays for the and and or gates are 0.1 ns each, while that for the xors is 0.2 ns. (XOR requires a more complex circuit). Then the delay from A or B to Cout is max(0.1 + 0.1 ns, 0.2 + 0.1 + 0.1 ns) = max (0.2, 0.4 ns) = 0.4 ns.

For an N bit adder constructed from full adders, the maximum propagation delay from the least significant bits in to the final carry out is N * 0.4 ns. (This can be significantly reduced by using a strategy we won't discuss.)

3. This delay is very important; it is, in fact, one of the key factors in determining how fast a digital system can run.

Example: Suppose a CPU is designed to run at 2 GHz. Then the worst-case propagation delay down any path must be less than $1 / (2 \text{ GHz}) = 1 / (2 \times 10^9 \text{ sec}^{-1}) = 0.5 \times 10^{-9} \text{ sec} = 0.5$ ns. If all gates in the system have the same propagation delay, and the the longest path in the system involves 10 gates, then an individual gate's propagation delay must be less that 0.05 ns.

4. As you know, CPU clock speeds have increased dramatically in your life time (by a factor of better than 1000 : 1). This has been made possible, in part, by a dramatic decrease in propagation delay due to technological improvements. (There have also been reductions due to techniques that have shortened the longest path)

   Example: One could not build a 2 GHz CPU using TTL gates of the sort we are using in lab, since the typical propagation delay for a gate is about 10 ns - about 200 times as great as what is needed for such a clock rate!

   a) One key factor has been using processes that reduce the feature size - the size of the individual components on the chip.

      (1)Propagation delay is much less if a gate is designed only to drive other gates on the same chip, as opposed to sending a signal to another chip. (The issue is not so much the physical distance, but rather the strength of the signal needed to ensure sufficient noise immunity.)

      (2)Physically smaller circuits can change state more quickly.

   b) Another factor has been the use of lower voltages, which reduces the “swing” between low and high, and, in particular, the time needed to charge/discharge capacitance on the chip. Of course, this comes at the cost of a reduction in noise immunity, which calls for other compensating measures to minimize extraneous signals.

5. While clock speed increases arising from propagation delay decreases have been dramatic in the past few decades, there is strong evidence that we have“hit a wall”.

   Cf an Dr. Dobb's Journal entitled ”The Free Lunch is Over” (March 2005). The essential thesis is that techniques for increasing clock speeds that have worked in the past are beginning to hit fundamental limits, and that future increases in speed will largely come through various forms of concurrency, rather than improvements to raw speed. While this is now an old article, this observation about clock speed has held true for well over a decade.

Today, one manifestation of this is the growing use of multicore processors, GPU's, etc. . We will spend quite a bit of time on this later in the course.