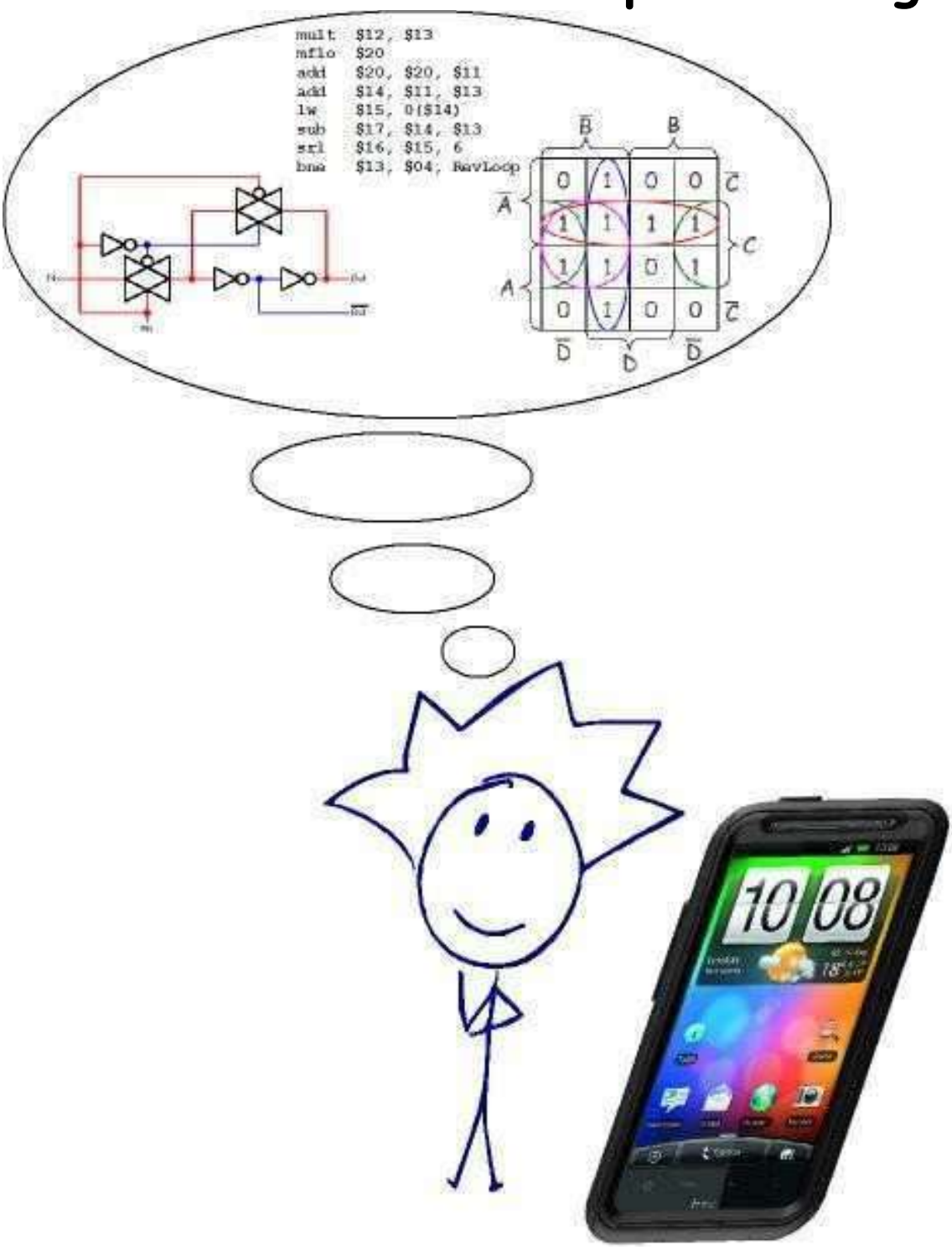
Designing Computer Systems

Introduction to Computer Design



Introduction to Computer Design

Our world is full of computer-based systems: smartphones, laptops, tablets. And new, useful and attractive products are appearing regularly. How are they designed? What are they made of? Today's high tech products contain dozens of subsystems, each composed of many components. Some are specialized, like a color display or a wireless transceiver. But the *computing* in a computing system is general and programmable. Interestingly, while computing provides the system "smarts", it is built using two very simple components: switches and wire.

So how does one create a smart product out of dumb components? First, these components are extremely versatile. Wire-connected switches can be used to build functional elements (e.g., perform arithmetic operations), memory elements (e.g., store values and program instructions), and control elements (e.g., sequence program instructions in a execution datapath). These elements can for a programmable computer that can exhibit a very smart behavior.

Second, technology offers the ability to integrate millions or even billions of wire-connected switches in an extremely compact integrated circuit (IC). This large number of switches supports the construction of a powerful programmable computer and sophisticated real-world interfaces. The low cost of these devices allows them to be incorporated in a wide range of products.

Designing systems with millions of components is a challenge. A successful strategy is a *design hierarchy*. Use several simple elements to produce a more complex component. Then design the next level using these more complex components. The design hierarchy below spans from switches and wire to assembly programming.

<table>
<tr><td colspan="3">Assembly Language</td></tr>
<tr><td colspan="3">Instruction Set</td></tr>
<tr><td>Memory</td><td>Datapath</td><td>Controller</td></tr>
<tr><td>Storage</td><td>Functional Units</td><td>State Machines</td></tr>
<tr><td colspan="3">Building Blocks</td></tr>
<tr><td colspan="3">Gates</td></tr>
<tr><td colspan="3">Switches and Wire</td></tr>
</table>

While there are other ways to describe the organization of programmable computers, this hierarchy illustrates the design path presented in the following chapters. Some chapters address specific levels of the hierarchy (designing with switches). Others focus on design techniques (simplification) and mathematics (Boolean algebra). All chapters emphasize design; how do you build a computer system.

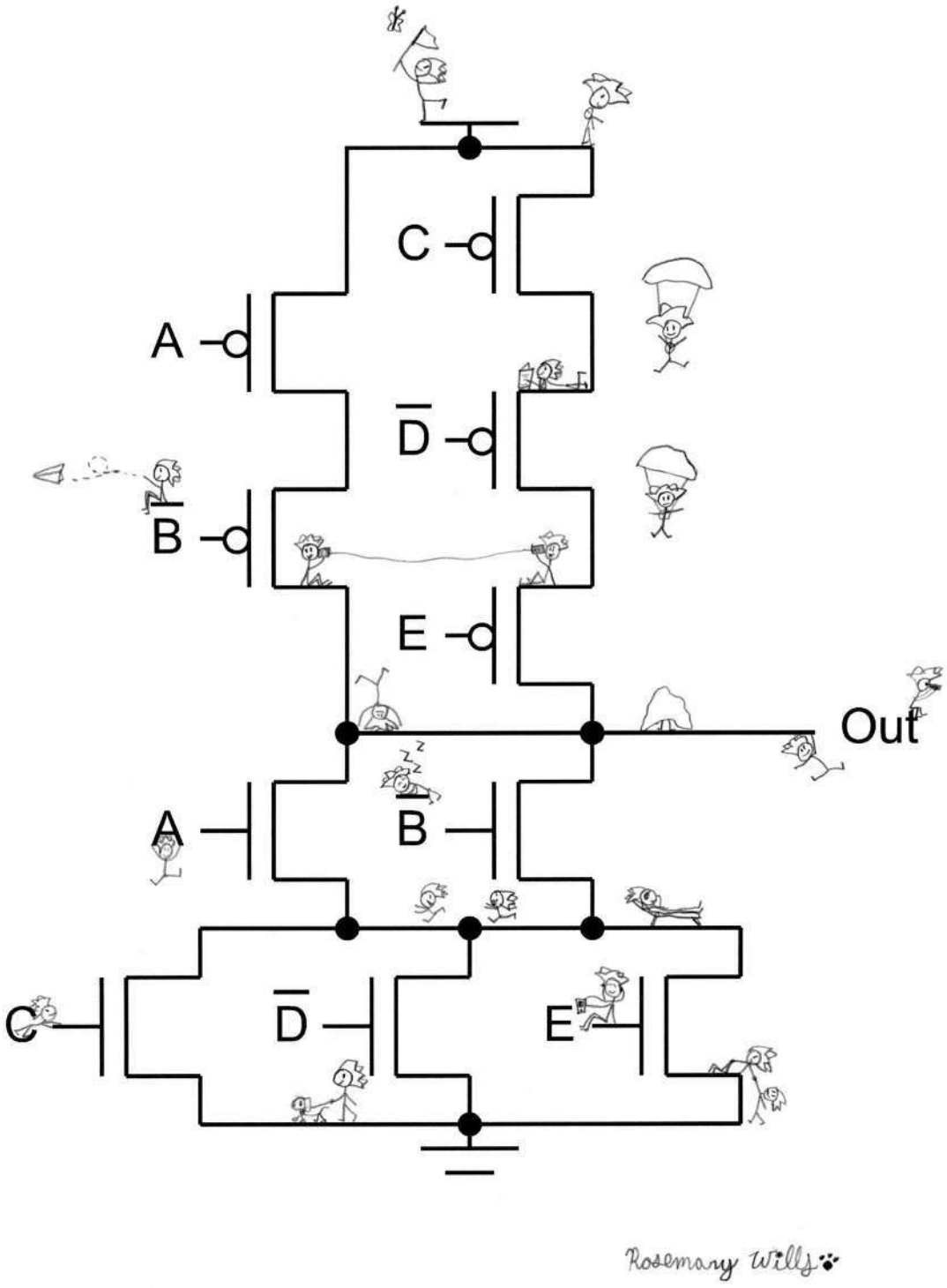
Chapters

1. Introduction to Computer Design
2. Switches and Wire
3. Boolean Algebra
4. Gate Design
5. Simplification
6. Building Blocks
7. Number Systems
8. Arithmetic
9. Latches and Registers
10. Counters
11. State Machines
12. Memory
13. Datapath
14. Controller and Instruction Set
15. Assembly Programming

CompuCanvas: There is a free, open-source tool that can help you try out many of the ideas and techniques describe in these chapters. It is written in the Python programming language (available at www.python.org); so it runs on almost everything. Find out more about it at the CompuCanvas website www.compucanvas.org.

Designing Computer Systems

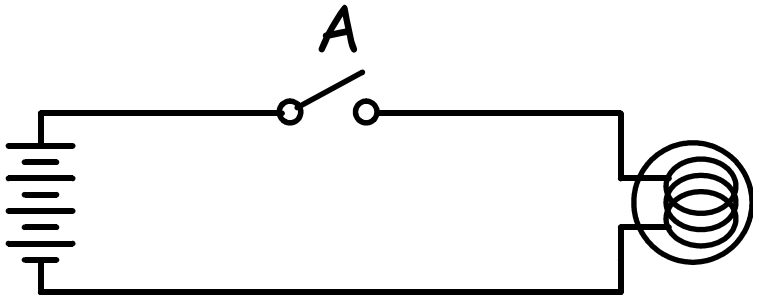
Switches and Wire



Designing Computer Systems

Switches and Wire

Despite their apparent complexity, digital computers are built from simple elements, namely *switches* and *wire*. To see how switches and wire can perform computations, consider the circuit below. The battery on the left is connected to the bulb on the right through the switch labeled **A**.

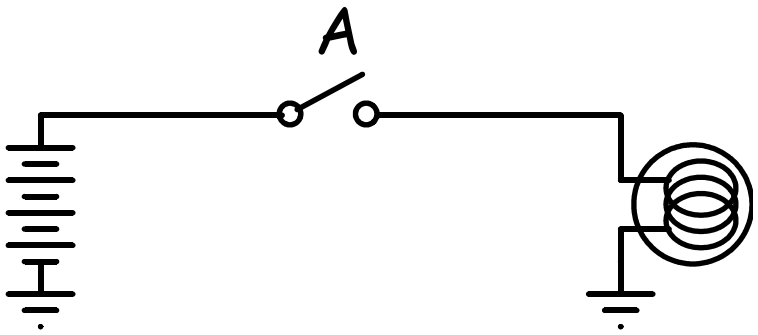


The battery will light the bulb if there is a complete path for current to flow from one side of the battery to the other. If the switch is open, no current can flow so the light is off. If the switch is closed, current flows and the light is on. The behavior of this simple circuit can be expressed using a table.

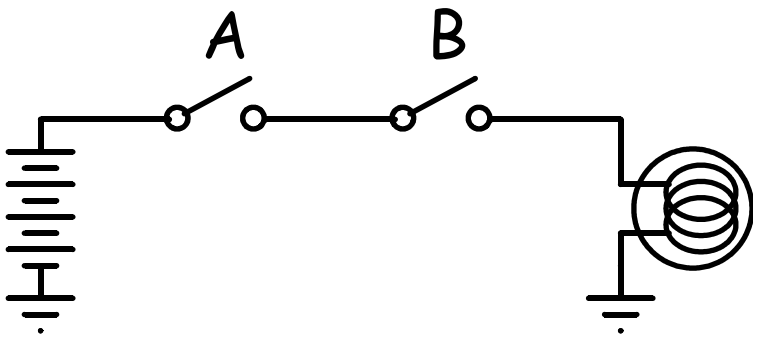
switch	light
open	off
closed	on

This type of table has been given the lofty name *Truth Table*. A more meaningful name would be *behavior table* since it describes the behavior of the circuit. Truth tables list all possible inputs to a system on the left and resulting outputs on the right. A truth table specifies how a system should behave. It does not specify how it should be implemented; this can be done in many ways.

Sometimes an icon is used to show connected nodes without drawing a wire. In the circuit below, the triangular symbols below the battery and bulb represent *ground*. We can imagine that all points attached to ground icons are connected together. So this circuit behaves identically to the circuit above.



Here's a system with two switches in series.



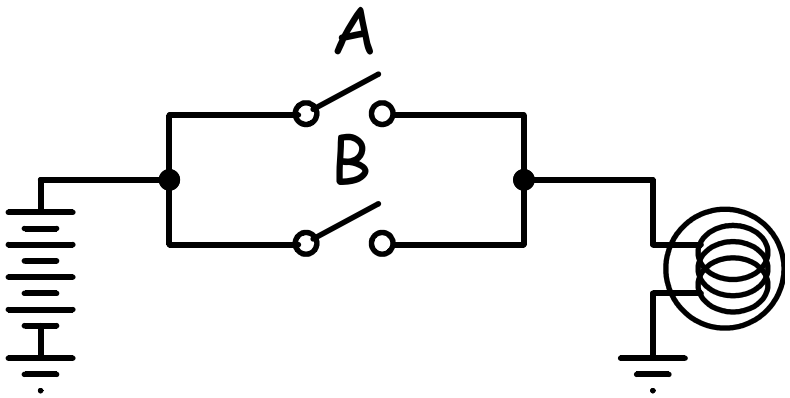
Because each switch can be in one of two states (open or closed) and there are two switches, the truth table has four rows. It's not so important how we list the input combinations so long as all cases are included exactly once.

switch A	switch B	light
open	open	off
closed	open	off
open	closed	off
closed	closed	on

In this circuit, the light is on when switch A is closed *AND* switch B is closed. This illustrates an important point; **series switches produce AND behavior**. Using words like open/closed and on/off to describe system behavior is verbose. We can assign the value **0** to an open switch and **1** to a closed switch. Further we can assign the value **0** to an off (dark) bulb and **1** to an on (lit) bulb. Sometimes we'll refer to 1 as *true* and 0 as *false*. Now the truth table becomes more compact.

A	B	Out
0	0	0
1	0	0
0	1	0
1	1	1

The next system has two switches in parallel.

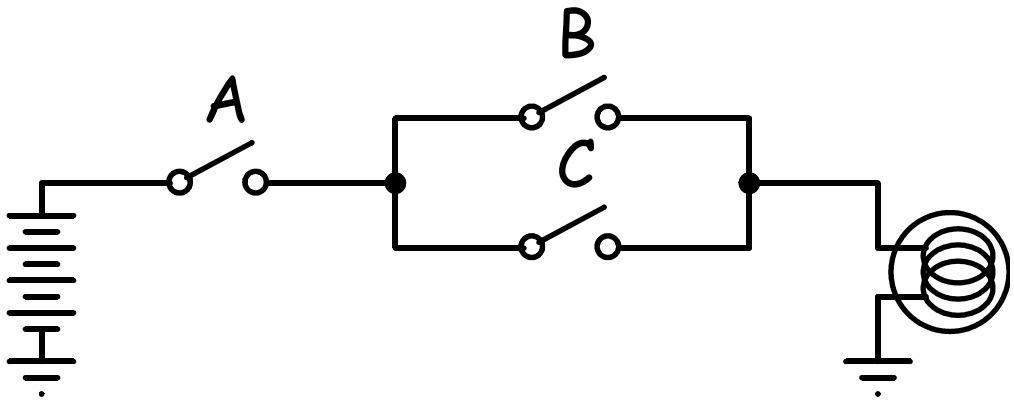


Here the output is true if switch A is closed OR switch B is closed. This illustrates another important point; **parallel switches produce OR behavior**. Here's the truth table.

A	B	Out
0	0	0
1	0	1
0	1	1
1	1	1

We might call this an "OR circuit" (in contrast to the previous "AND circuit") because its output is true when either A is true OR B is true. Defining a system's behavior by when its output is true is call *positive logic*. This contrasts with *negative logic* where a system's behavior is defined when its output is false. We have to pick one convention; positive logic seems more intuitive.

The last circuit is more complex.

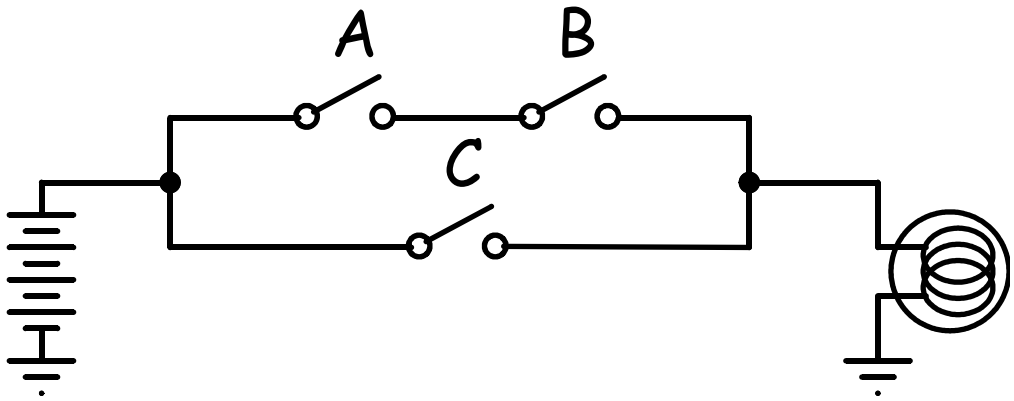


The truth table has eight rows to capture all input combinations of the switches. In general, if a system has *N* binary inputs (i.e., each input can be in one of two states), there are 2^N entries in the truth table. This circuit behavior is accurately, if not clearly, described in the truth table.

A	B	C	Out
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	1

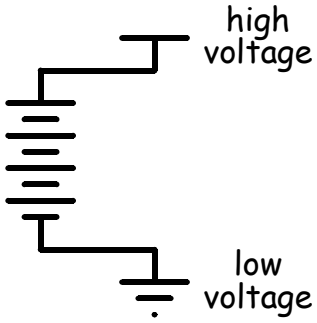
A more concise description of the behavior is derived from the series and parallel arrangement of the switches. By starting at the outer-most connections, switch A is in **series** with a second switch combination, switch B in **parallel** with switch C. Therefore, the output is true when **A AND (B OR C)**.

The parentheses are significant since AND has higher precedence than OR, just as multiplication has higher precedence than addition. The arithmetic expression $A \cdot (B + C)$ differs from $(A \cdot B) + C$ that results if the parentheses are removed. Similarly, A AND (B OR C) differs from (A AND B) OR C. The second expression would be implemented as the following, non-equivalent circuit.



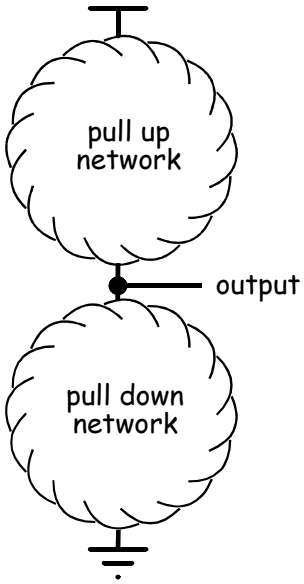
Designing systems with these switches has one major limitation. The input to the switch is a mechanical activator (your finger). But the output of the system is optical (light). This prevents composing systems since the output of one system cannot control the input of another. To remedy this problem, we'll consider a different kind of switch.

A voltage controlled switch fits our need since (A) most systems have a handy voltage source from a power supply or batteries, (B) switches can easily connect to either a high or low voltage to produce an output, and (C) controlling switches with a voltage does not implicitly dissipate a lot of energy. The sources of high (1) and low (0) voltages are show below.

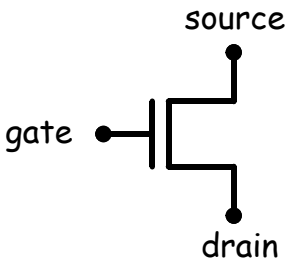


A battery is connected to the ground symbol to represent the low voltage source. The other side of the battery is connected to a new symbol that resembles a "T". It is used to represent the high voltage source. These high and low supply symbols are used throughout a design to provide necessary voltages needed to activate voltage controlled switches. But only one battery is needed to generate them!

In order to build composable circuits (i.e., where the output of one circuit can control the input of another), voltage controlled switches must connect an output to either the high or low supply, as shown below. We'll use networks of voltage controlled switches to provide correct output values for each combination of inputs.

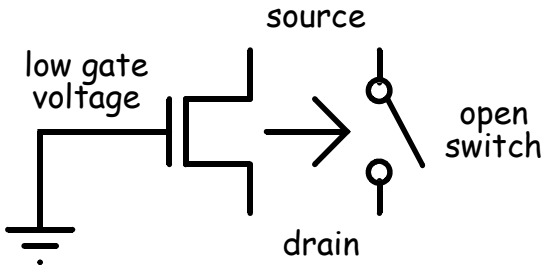


N-Type Switch: The voltage controlled switch is called an **N-type** switch. It has three places for wires to connect called terminals. The source and drain terminals represent the ends of a switch that are connected when the switch is closed. The gate terminal is the voltage controlled input that opens or closes the switch.

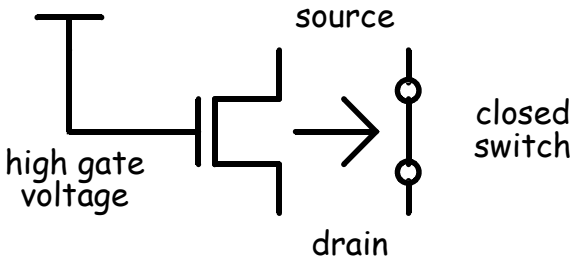


gate	switch
low (0)	open
high (1)	closed

If the voltage connected to the gate is low, the switch is open. If the voltage on the gate is high, the switch is closed. N-type switches are called an **active high**.

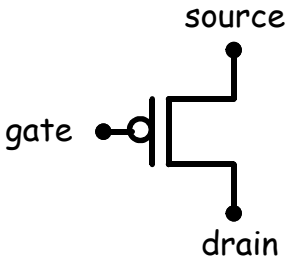


A low gate voltage opens the switch



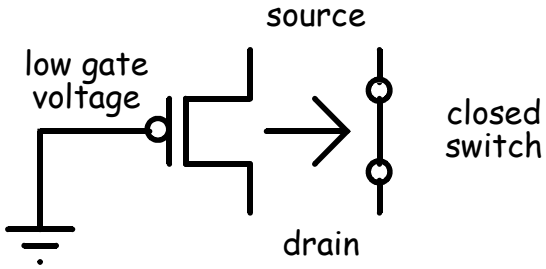
A high gate voltage closes the switch

P-Type Switch: As one might expect in a binary world, there is also a **P-type** switch (note the bubble drawn on the gate terminal). It behaves just like an N-type switch, except that the source-drain switch closes when the gate voltage is low.

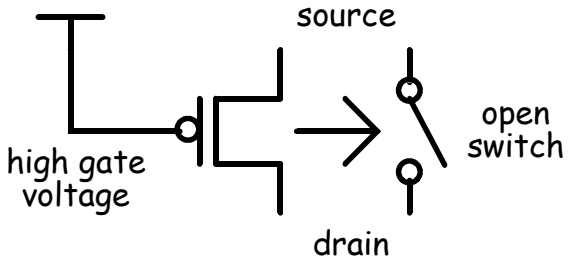


gate	switch
low (0)	closes
high (1)	open

Since the switch closes when the gate voltage is low, a P-type switch is called **active low**.



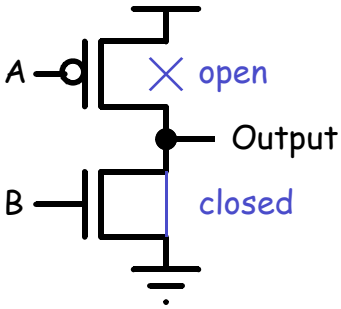
A low gate voltage closes the switch



A high gate voltage opens the switch

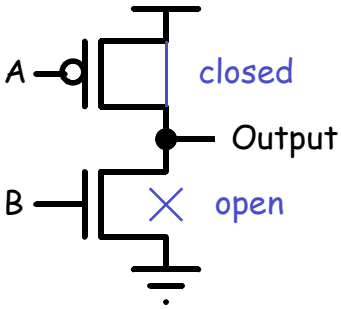
Designing Logic: Building logical circuits with voltage controlled switches begins like the battery and light designs. Only now a complimentary circuit must be

created to connect the output to the high voltage sometimes and the low voltage other times. Errors in the design process can lead to unfortunate consequences. Consider this simple two input circuit.



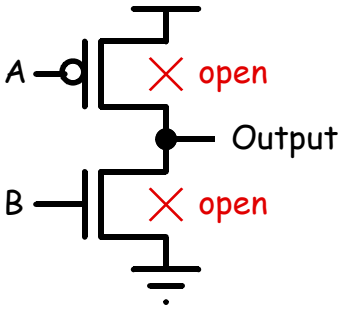
A	B	Out
0	0	
1	0	
0	1	
1	1	0

When A and B are both high, the N-type switch is closed connecting the output to the low voltage. Since the P-type switch is open, it does not participate.



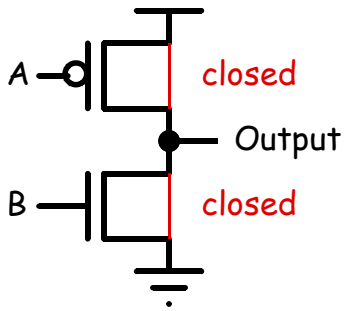
A	B	Out
0	0	1
1	0	
0	1	
1	1	0

When A and B are both low, the P-type switch is closed connecting the output to the high voltage. Since the N-type switch is open, it does not participate.



A	B	Out
0	0	1
1	0	float
0	1	
1	1	0

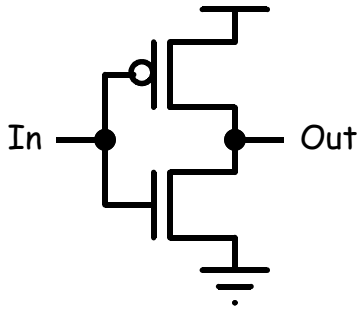
When A is high and B is low, both the N-type and P-type switch are open. The output is not connected to the high voltage or the low voltage. This undefined state, often called a *floating node*, does not provide a valid output for controlling other switches. For this reason, this condition should be avoided.



A	B	Out
0	0	1
1	0	float
0	1	short
1	1	0

When A is low and B is high, both the N-type and P-type switch are closed. This condition is more serious than a floating output in that the high voltage is connected to the low voltage. This is called a *short*. It is particularly bad since, in addition to having an undefined output value, a lot of current flows, generating heat that can damage the switches. Shorts should be eliminated in all designs!

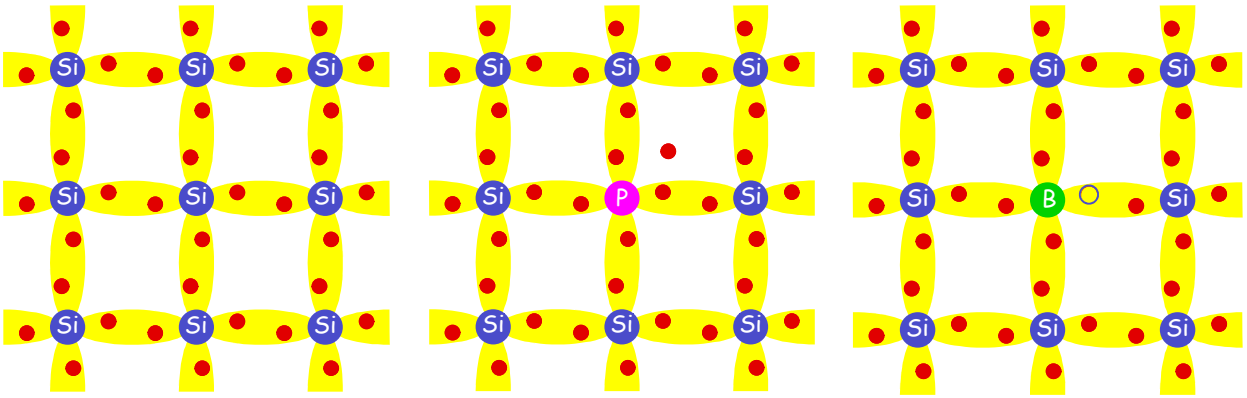
NOT Gate Implementation: A NOT gate can be implemented using a variation of this circuit. The two undesirable states (float and short) occur when A and B have different values. If the two inputs are connected together as a single input (In), unwanted output conditions are eliminated. This circuit implements a NOT gate.



In	Out
0	1
1	0

Note that this circuit is controlled by a high or low voltage, and it produces a high or low voltage output. These switches can be used to implement complex expressions of AND and OR functions. But first we need to understand a few things about switch technology.

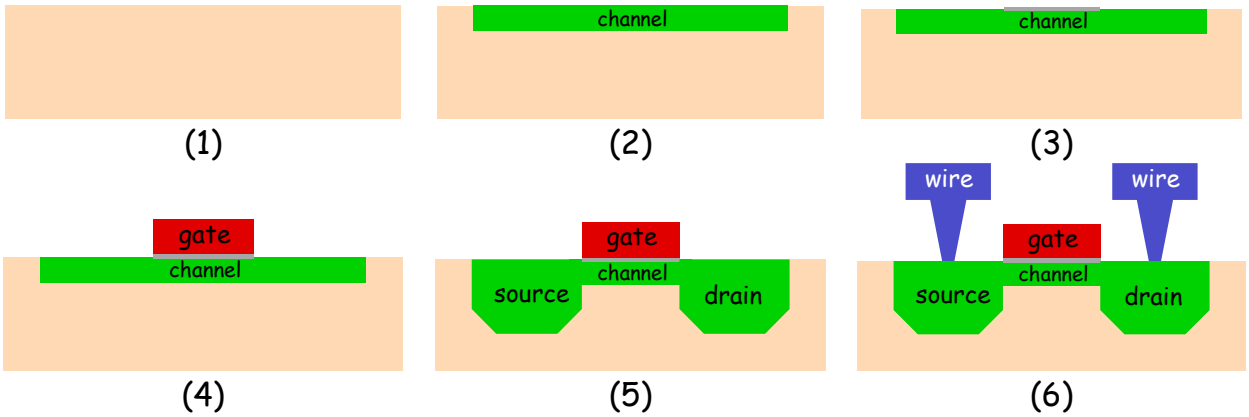
See MOS Switches: These voltage-controlled switches are built out of silicon (Si). This is a surprise since pure silicon and silicon dioxide (silica) are insulators. In fact, most high voltage insulators are ceramics made from silicon. Si atoms have four valance (outer-shell) electrons. When arranged in a crystal lattice, all of these electrons are bound so charge carriers are not available for conduction (A). In order to change things, atoms of other elements are embedded in the lattice through ion implantation or diffusion. These elements, called dopants, have either one more or one less valance electron. Phosphorus has an extra electron (five) (B); Boron has one less (three) (C).



(A) (B) (C)

When these elements find themselves in the lattice, their extra electron contributes a negative charge carrier (an electron) while their lacking electron contributes a positive charge carrier (a hole). Introducing these charge carries produces an interesting *semiconductor* material that can be controlled by an electrical field.

Through clever processing of a silicon wafer, doped regions can form the semiconducting channels of a switch. The basic structure is shown below in cross-section. First a wafer of pure silicon is cut and polished to a smooth surface (1). Then dopants are selectively implanted to form a channel (2). A small isolative layer is grown above the channel (3). Then a layer of conductive polysilicon is selectively formed over the channel to act as the gate (4). Additional dopant implantation deepens the source and drain regions (5). Then wires (typically aluminum or copper) are formed to contact and connect source, drain, and gate terminals of the formed devices (6). The completed device is a Metal Oxide Semiconductor Field Effect Transistor (MOSFET).

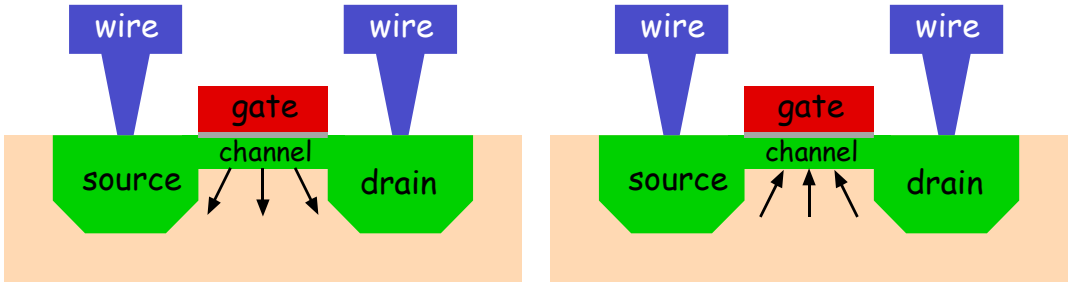


(1) (2) (3) (4) (5) (6)

This selective processing is accomplished using photolithography. In this process, a light sensitive material is used to pattern the layered structures on the silicon surface. Because this fabrication process is performed on the entire wafer surface, a couple hundred chips, each containing hundreds of millions of transistors

can be fabricated at the same time. This dramatically reduces the cost for each chip.

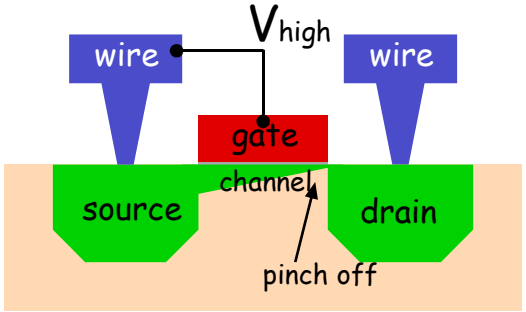
This semiconductor device can operate as a voltage controlled switch. When a voltage is placed on the gate terminal, relative to the silicon around the device (know as the substrate), a vertical electrical field is generated. This field can push charge carries out of the channel, opening the switch (A). Or the field can attract charge carries into the channel, closing the switch (B).



(A) (B)

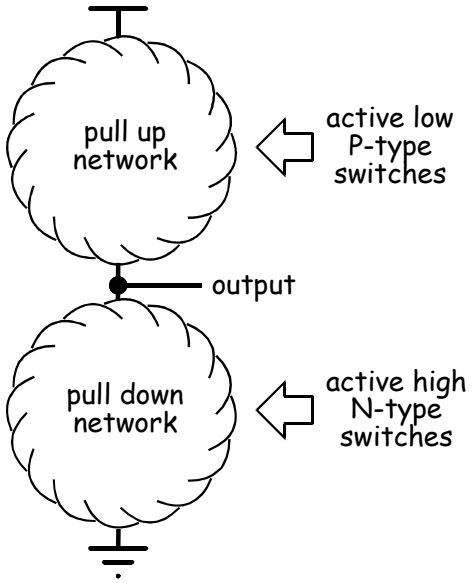
These semiconducting switches come in two types. An N-type Field Effect Transistor (NFET) *closes* (conducts) when a high voltage is placed on its gate. A P-type Field Effect Transistor (PFET) *opens* (isolates) when a high voltage is placed on its gate. The two switch types differ only in the dopant used and some early preparation of the surrounding silicon so the substrate can be properly biased to create the field. When these two types are used to implement a Boolean expression, the resulting design is called Complementary Metal Oxide Semiconductor or CMOS.

These switches have a critical limitation. When the voltage applied to the gate to close the switch is also present at the source or drain, the generated horizontal field will deplete the charge carriers at the opposite side of the channel, *pinching off* channel conduction. As the source-drain voltage approaches a technology specific *threshold voltage*, the opposite terminal will no longer be pulled towards the supply voltage. Here's an N-type switch connected to the high voltage.



This means N-type switches cannot be used to pull the output high. Nor can a P-type switch be used to pull the output low. Our switch design strategy, using

networks of voltage controlled switches to produce an output voltage must incorporate this technology limitation.



Time to Design: We're now ready to implement Boolean expressions using N and P type switches. We need to use ideas we've covered so far:

- Series switches produce the AND function.
- Parallel switches produce the OR function.
- P-type switches are active low and can pull high.
- N-type switches are active high and can pull low.
- An output should always be pulled either high or low.

Plus we must add a small but significant relationship called DeMorgan's Theorem that states:

- X AND Y = NOT(NOT X OR NOT Y)
- X OR Y = NOT(NOT X AND NOT Y)

Boolean expressions quickly become awkward when written this way. We can use symbols from arithmetic · and + to represent AND and OR functions. We can also use bars to represent the NOT operation. So DeMorgan's Theorem becomes:

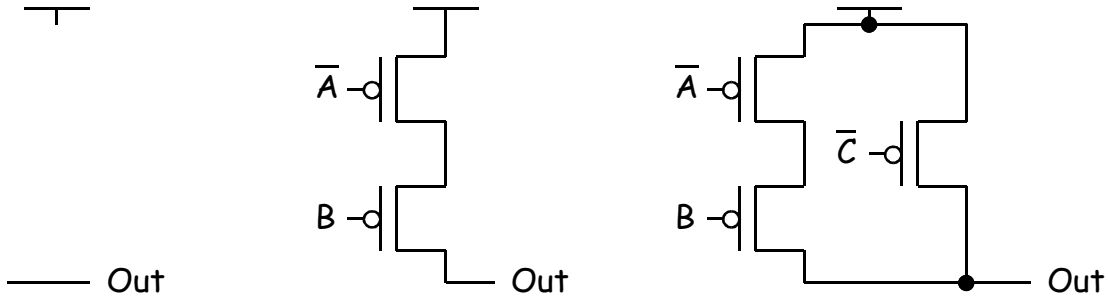
- $X \cdot Y = \overline{\overline{X} + \overline{Y}}$
- $X + Y = \overline{\overline{X} \cdot \overline{Y}}$

This means an AND and OR operations can be exchanged by complementing their inputs and output. This will come in handy in switch design.

Suppose we want to implement a Boolean expression composed of AND and OR operations applied to binary inputs (in their true and complemented form). Here's an example.

$$Out = A \cdot \overline{B} + C$$

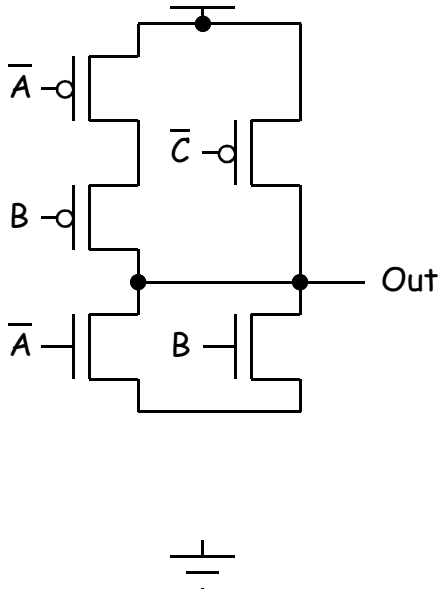
We'll need to design a pull high network of P-type switches, and a pull low network of N-type switches (1). The output should be high when A is high AND B is low OR when C is high. The first part of this OR expression should connect the output to the high supply when A is high AND B is low. This is accomplished by a series combination of switches. But with active low P-type switches, we must complement the inputs (2). So when A is high, $\overline{A}$ is low closing the active low P-type switch. If B is also low ($\overline{B}$ is high), then this low input will close the other P-type switch, completing the connection between the output and the high supply. The second part of the OR pulls the output high if C is high. OR is implemented by a parallel connection of switches. Again, C must be complemented so that when C is high, the active low P-type switch will close pulling the output high (3).



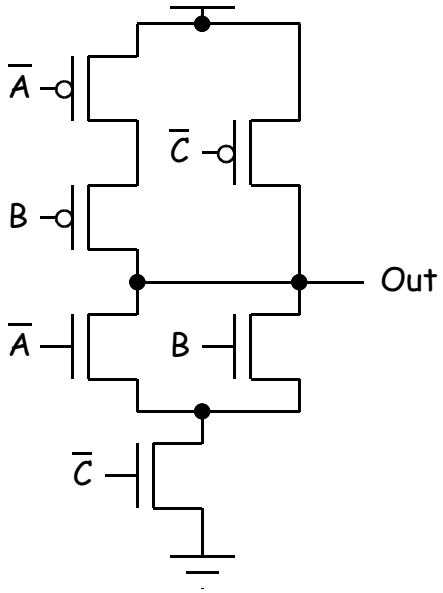
(1) (2) (3)

Now we must design a pull low network that connects the output to the low supply whenever it is not being pulled high. DeMorgan's Theorem shows that ANDs and ORs can be swapped if inputs and outputs are complemented. Using N-type rather than P-type switches complements all inputs. Pulling low rather than high complements the output. So we can exchange AND and OR by exchanging series and parallel switch arrangements. We must begin with the outermost operation, in this case the OR. In the first part of the OR, A and are in series in the pull up network. So they are in parallel in the pull down network (4). Since $A \cdot \overline{B}$ are in

parallel with C on in the pull up network, they will be in series in the pull down network (5).

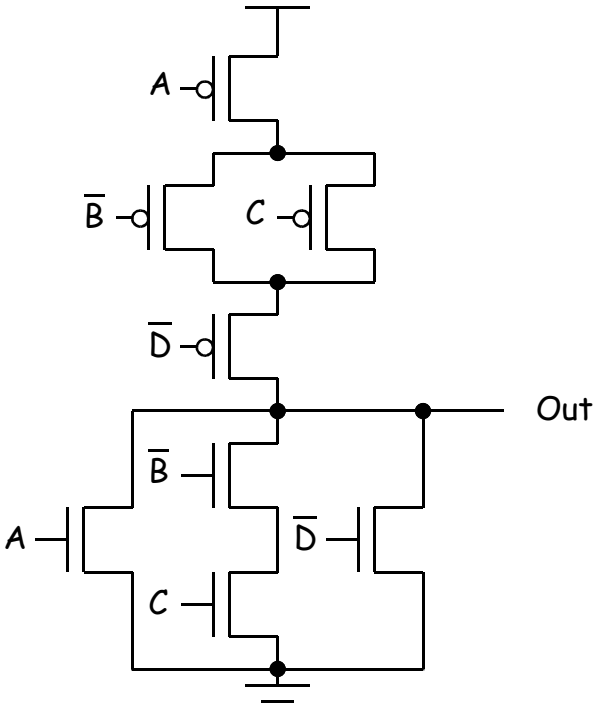


(4)

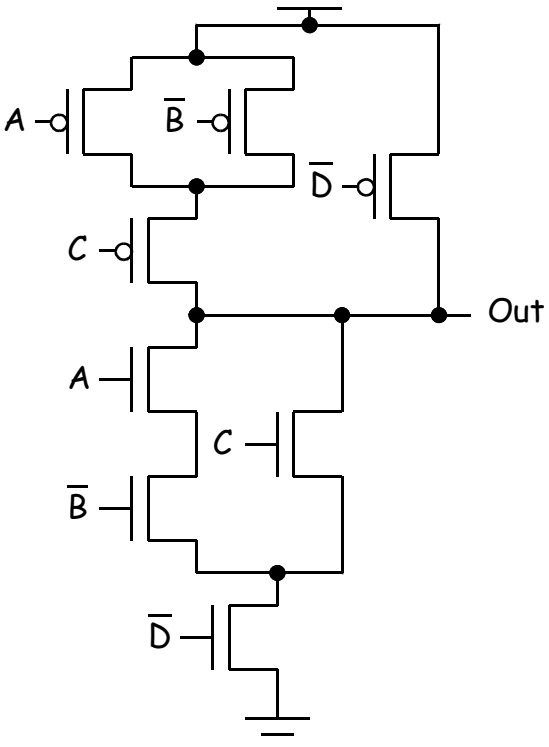


(5)

Here are a few more examples:



$\overline{A} \cdot (B + \overline{C}) \cdot D$



$(\overline{A} + B) \cdot \overline{C} + D$

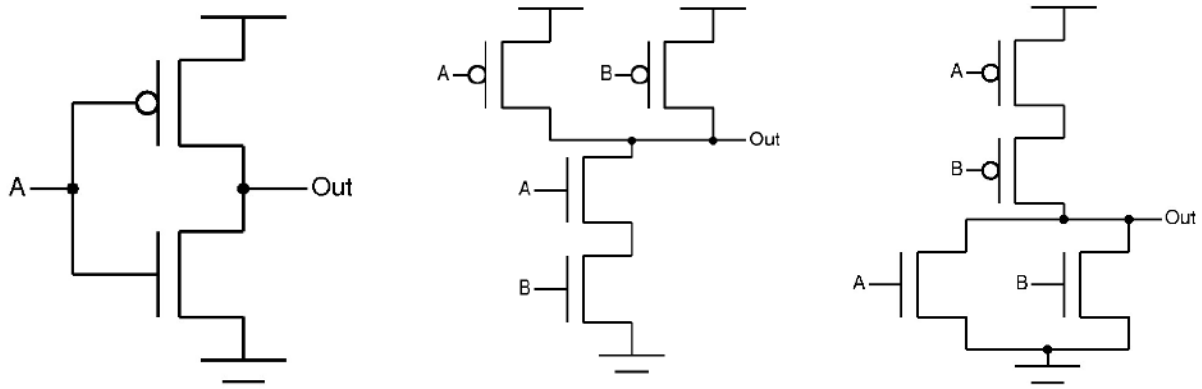
Not that parallel switches in the pull up network are in series in the pull down network, and vise versa. Care must be exercised to work on the operations from

the outside in. That is, the last evaluated operation in the expression is the outermost combination of switching circuits.

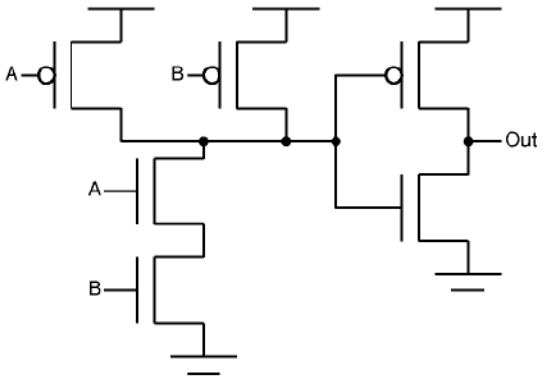
Building Abstractions: Switch design allows direct implementation of a behavior described in Boolean expression. It is often yields the fastest (in terms of delay) and most efficient (in terms of switches required) solution. But sometimes a little convenience is worth a slightly higher cost. People don't prepare all their food from scratch even though it would be more healthy and less expensive. Engineers don't write programs in the machine language of computers, even though the executable file would be smaller and it would run faster. And when we are designing a digital system with a couple hundred million transistors, we may prefer not to implement all functions with switches.

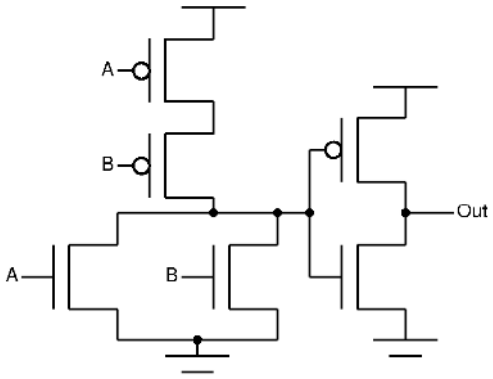
So we do what all engineers do. We create larger, more complex functional abstractions and then design with them. An automobile is a complex system. Fortunately automobile designers combine already understood subsystems for power, steering, braking, etc. and then adapt as necessary. Computer designers do the same thing, but with different building blocks.

Basic Gates: Since we already express our designs using logical functions, a natural choice for new, more complex abstractions would be logical gates. Here are the basic gates used in digital design: NOT, NAND, NOR, AND, and OR. Consider a gate with **i** inputs. inverting gates (gates that begin with "N") require **2i** switches for each input. Non-inverting gates (AND and OR) require **2i+2** switches.



NOT NAND NOR





AND OR

In the chapter on Gate Design, we'll see how designing with gates compares to the switch design. But first we will visit the mathematics of digital design in Boolean Algebra.

Summary: Switch design is at the heart of nearly every computer technology we use today. Here are the key points.

- In contrast with human experience, computation is largely performed on binary values (zeros and ones).
- The computing world is built with digital switches.
- These switches are voltage controlled, and are assembled in networks to produce high or low voltage outputs.
- Series switches implement the AND function; Parallel switches implement OR.
- NFETs are active high switches, and are preferred for pulling an output low.
- PFETs are active low switches, and are preferred for pulling an output high.
- Outputs that are not connected to the high or low voltage are floating (undefined), and this is not good.
- Outputs are are connected to both the high and the low voltages result in a short, and this is bad.
- Switches are actually MOSFETs, made from silicon with other dopant elements that create free charge carriers.
- High integration of MOSFETs on a single chip provides many connected switches for digital computation, at a low cost.
- Boolean expressions can be efficiently implemented using MOSFETs.

Boolean Algebra

← complement output →

inputs →

←complement

AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1

NAND

A	B	$\overline{A \cdot B}$
0	0	1
1	0	1
0	1	1
1	1	0

NOR

A	B	$\overline{A+B}$
0	0	1
1	0	0
0	1	0
1	1	0

OR

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

© Rosemary Wills

Designing Computer Systems

Boolean Algebra

Programmable computers can exhibit amazing complexity and generality. And they do it all with simple operations on binary data. This is surprising since our world is full of quantitative computation. How can a computer complete complex tasks with simple skills?

A Little Logic: Computers use logic to solve problems. Computation is built from combinations of three logical operations: AND, OR, and NOT. Lucky for us, these operations have intuitive meanings.

AND	In order to get a good grade in ECE 2030, a student should come to class **AND** take good notes **AND** work study problems.
OR	Today's computers run Microsoft Windows 7 **OR** Mac OS X **OR** Linux.
NOT	Campus food is **NOT** a good value.

Surprisingly, these three functions underlie every operation performed by today's computers. To achieve usefulness and generality, we must be able to express them precisely and compactly. From an early age, we have used arithmetic expressions to represent equations with multi-valued variables and values.

$$Cost = X \cdot \$2.00 + Y \cdot \$1.50$$

In the world of logic, all variables have one of two values: *true* or *false*. But expressions can be written in the otherwise familiar form of an arithmetic expression. We'll use the "+" operator to represent OR and the "·" operator to represent AND. The following is a simple example of a Boolean expression:

$Out = A \cdot B + C$ Out is true if A AND B are true OR C is true

Just like in arithmetic expressions, operation precedence determines the order of evaluation. AND has higher precedence than OR just as multiplication has higher precedence than addition. Parentheses can be used to specify precise operation evaluation order if precedence is not right. Note that the expression below closely resembles the previous example. But it has a different behavior (e.g., consider each when A is false and C is true.)

$Out = A \cdot (B + C)$ Out is true if A is true AND (B OR C is true)

Is NOT enough?: NOT (also known as complement) is represented by a bar over a variable or expression. So $\overline{A}$ is the opposite of A (i.e., if A is true, $\overline{A}$ is false and vise versa). When a bar extends over an expression, (e.g., $\overline{A+B}$) the result of the

expression is complemented. When a bar extends over a subexpression, it implies that the subexpression is evaluated first and then complemented. It's like parentheses around the subexpression.

Many years ago in the 1800s, the mathematics of these binary variables and logical functions was described by a man named George Boole and a few of his colleagues. Now we call this mathematics *Boolean Algebra*.

Operation Behavior: These logical functions have intuitive behaviors. An AND expression is true if *all* of its variables are true. An OR expression is true if *any* of its variables are true. A NOT expression is true if its single variable is *false*.

Sometimes a table is used to specify the behavior of a Boolean expression. The table lists all possible input combinations of the right side and the resulting outputs on the left side. This behavior specification is called a truth table. Because "true" and "false" are hard to right compactly, we'll use 1 and 0 to represent these values. Here is a summary of AND, OR, and NOT behaviors using true tables.

A	B	A · B
0	0	0
1	0	0
0	1	0
1	1	1

A	B	A + B
0	0	0
1	0	1
0	1	1
1	1	1

A	$\overline{A}$
0	1
1	0

Truth tables can have more than two inputs; just so long as all combinations of inputs values are included. If a combination was left out, then the behavior would not be fully specified. If there are i inputs, then there are 2^i combinations. It is also possible to have multiple outputs in a table, so long as all results are functions of the same inputs. Here are several Boolean expressions with three variables:

A	B	C	A · B · C	A + B + C	A · B + C	A · (B + C)
0	0	0	0	0	0	0
1	0	0	0	1	0	0
0	1	0	0	1	0	0
1	1	0	0	1	1	1
0	0	1	0	1	1	0
1	0	1	0	1	1	1
0	1	1	0	1	1	0
1	1	1	1	1	1	1

Three variable AND and OR functions have expected behaviors. The AND output is true if all of the inputs are true. The OR output is true if any of the inputs are true. In the third expression, AND is higher precedence than OR. So the output is

true if either A AND B are true OR C is true. In the last expression, A must be true AND either B OR C (or both B and C) must be true for the output to be true.

There are a few basic properties of Boolean algebra that make it both familiar and convenient (plus a few new, not-so-familiar properties).

property	AND	OR
identity	$A \cdot 1 = A$	$A + 0 = A$
commutativity	$A \cdot B = B \cdot A$	$A + B = B + A$
associativity	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
distributivity	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
absorption	$A \cdot (A + B) = A$	$A + (A \cdot B) = A$

The identity, commutative, and associative properties are intuitive. Distributivity of AND over OR makes sense. OR over AND is new (don't try this with arithmetic addition over multiplication; it doesn't work!). Absorption is a new property of Boolean algebra. It comes in handy for simplifying expressions.

Generally, working with Boolean expressions is a lot like working with arithmetic expressions, with a few notable differences.

And that's NOT all: The complement (NOT) function adds an interesting dimension to the math. Where quantitative expressions have a rich range and domain for inputs and outputs, binary expression are decidedly limited. Any operation in a Boolean expression can have its inputs and/or its output complemented. But results will still be either true or false.

In fact most Boolean expression design extends the set of logical functions with NOTed AND (NAND) and NOTed OR (NOR). These functions are computed by complementing the result of the core operation.

AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1

NAND

A	B	$\overline{A \cdot B}$
0	0	1
1	0	1
0	1	1
1	1	0

OR

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

NOR

A	B	$\overline{A+B}$
0	0	1
1	0	0
0	1	0
1	1	0

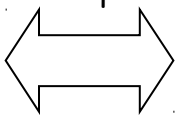
Here's where limited variable values and a small collection of basic operations leads to one of the most significant relationships in computation … DeMorgan's Theorem!

Sometimes the most amazing concepts are easy to see, when you look in the right way. In the table above, it's clear that NAND is just AND with its output complemented. All the zeros become ones and the one becomes zero. It's also clear that OR resembles NAND but for it being upside down. If all inputs to OR are complemented, the table flips and it matches NAND.

Complementing the inputs or the output of a NAND reverses this transformation. If inputs or an output is complemented twice, the function returns to its original behavior, leaving it unchanged. This supports reversible transformations between NAND and its left and right neighbors.

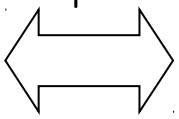
AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1



NAND

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



OR

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

Note that the transformations to obtain the NAND function can be employed for any of the four logical functions. To determine the necessary neighbor functions, consider cutting out the **four** function table above and wrapping it into a cylinder where AND and NOR are now neighbors. Or better still, let's draw the four functions in a two dimensional table, shown below. This is DeMorgan's square and it shows how any logical function can be transformed into any other logical function using NOT gates.

← complement output →

inputs →

AND

A	B	A·B
0	0	0
1	0	0
0	1	0
1	1	1

NAND

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

←complement

NOR

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

OR

A	B	A+B
0	0	0
1	0	1
0	1	1
1	1	1

You can start with a logical function, and by complementing all its inputs and/or its output, you can arrive at any other logical function. This has a profound effect on digital system design. Let's hear it for DeMorgan!

This principle can be applied to Boolean expressions as well. If you want to transform an OR into an AND, just complement all the OR inputs and its output. Let's try this process on a few expressions.

original expression	$A \cdot B$	$A \cdot (B + C)$	$A \cdot \overline{B} \cdot C$
AND becomes OR	$A + B$	$A + (B + C)$	$A + \overline{B} + C$
complement inputs	$\overline{A} + \overline{B}$	$\overline{A} + \overline{(B + C)}$	$\overline{A} + \overline{\overline{B}} + \overline{C}$
complement output	$\overline{\overline{A} + \overline{B}}$	$\overline{\overline{A} + \overline{(B + C)}}$	$\overline{\overline{A} + \overline{\overline{B}} + \overline{C}}$
equivalent expression	$\overline{\overline{A} + \overline{B}}$	$\overline{\overline{A} + \overline{(B + C)}}$	$\overline{\overline{A} + \overline{\overline{B}} + \overline{C}}$

In the first example, an AND function is turned into an OR function by complementing the inputs and the output. The second example has the same change, but one of the inputs to the AND is a subexpression. Note that when inputs are complemented, this subexpression receives a bar, but is otherwise unchanged. Just like this first input A, the subexpression is the input to the original AND function. The third example has a three input AND, so all three inputs must be complemented. Note also that the second input is already complemented. When it is complemented again, it has double bars. But when any variable or subexpression is complemented twice, the bars cancel out.

This DeMorgan transformation allows transformation of an OR to an AND using the same steps. It can be applied to the last evaluated function, the first evaluated function, or anything in between. It can even be applied to an entire expression (or subexpression) all at once … although some care must be exercised.

original expression	$A + (\overline{B} \cdot C)$	$\overline{(\overline{A} + B) \cdot (\overline{C} + D)}$	$\overline{A \cdot B + C} + \overline{D}$
swap AND and OR	$A \cdot (\overline{B} + C)$	$\overline{(\overline{A} \cdot B) + (\overline{C} \cdot D)}$	$(\overline{A + B}) \cdot \overline{C} \cdot \overline{D}$
complement inputs	$\overline{A} \cdot (\overline{\overline{B}} + \overline{C})$	$\overline{\overline{\overline{A}} \cdot \overline{B} + \overline{\overline{C}} \cdot \overline{D}}$	$(\overline{\overline{A} + \overline{B}}) \cdot \overline{\overline{C}} \cdot \overline{\overline{D}}$
complement output	$\overline{\overline{A} \cdot (B + \overline{C})}$	$\overline{\overline{A} \cdot \overline{B} + C \cdot \overline{D}}$	$\overline{(\overline{\overline{A} + \overline{B}}) \cdot \overline{\overline{C}} \cdot D}$
equivalent expression	$\overline{\overline{A} \cdot (B + \overline{C})}$	$A \cdot \overline{B} + C \cdot \overline{D}$	$\overline{(\overline{\overline{A} + \overline{B}}) \cdot \overline{\overline{C}} \cdot D}$

In the first example, both AND and OR functions are swapped. Then all inputs and the output are complemented. One might ask why no bars are added on subexpressions (e.g., over $(\overline{B} \cdot C)$). The reason is that each subexpression is both an output for one function and an input for another. Since both are complemented,

the two bars cancel out. Only the input variables (e.g., A, B, and C) and the last function to be executed (the outermost function) will be complemented.

Note also that the function evaluation order is invariant throughout this process. In the first example, $\overline{B}$ is first ANDed with C. Then the result is ORed with A. After the transformation is complete, this is still the order. Often parentheses must be added to preserve this order since AND and OR have different precedence. Sometime parentheses can be dropped (like in the second example) since the new function precedence implies the correct (original) evaluation order.

In the second example, an initial bar over the outermost function (AND) is canceled when the entire expression is complemented. Note also that the bars over inputs are reversed. In the third example, a bar over the earlier OR function ($\overline{A \bullet B+C}$) remains unchanged through the transformation.

Eliminating Big Bars: Often implementation of Boolean expressions requires transforming them to a required form. For example, switch implementation needs a Boolean expression with complements (bars) only over the input variables (literals). If an expression has complements over larger subexpressions (big bars), DeMorgan's theorem must be applied to eliminate them. Here's an example.

	$Out=\overline{\overline{A+\overline{B}+\overline{C}\cdot D}}$	expression with many big bars
1	$\overline{\overline{A+\overline{B}}\cdot\overline{\overline{C}\cdot D}}$	replace final AND with OR, and
2	$Out=\overline{\overline{\overline{A+\overline{B}}}\cdot\overline{\overline{\overline{C}\cdot D}}}$	complement inputs and output
3	$Out=(A+\overline{B})\cdot\overline{\overline{\overline{C}\cdot D}}$	remove double bars
4	$(A+\overline{B})\cdot\overline{\overline{C}+D}$	replace first AND with OR, and
5	$Out=(A+\overline{B})\cdot\overline{\overline{\overline{C}+\overline{D}}}$	complement inputs and output
6	$Out=(A+\overline{B})\cdot(C+\overline{D})$	remove double bars

When eliminating big bars, one should start with the outermost complemented function. In this case, the OR in the center of the expression comes first. In step 1, it is replaced by an AND. The function's inputs and outputs are then complemented. Then double bars are removed. Note that parentheses must be added to maintain the same evaluation order. These first steps remove the big bars from the initial expression; but a new big bar is created over $\overline{C}$+D. So in step 4, this OR is replaced by an AND. Then its inputs and outputs are complemented. Again parentheses must be added to preserve the original evaluation order. The final expression (step 6) has an equivalent expression without big bars.

DeMorgan's Theorem allows us to transform a Boolean expression into many equivalent expressions. But which one is right? That depends on the situation. If we are designing an implementation with switches, eliminating big bars is an important step in the process. For gate design, we might want to use logical operations that better match the implementation technology. Regardless of implementation, we might just want to use a form of the expression that most clearly expresses (to a fellow engineer) the function we require.

In most cases, we can choose the equivalent expression that fits our needs. But how can we evaluate expressions for equivalence?

Standard Forms: There are two standard forms that offer a canonical representation of the expression. Let's explore these forms starting with a function's behavior in a truth table.

A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	1
0	1	1	0
1	1	1	1

To correctly express this function, we must show where its output is true (1) and where its output is false (0). We can accomplish this in two ways. Let's start with the "easy" one, expressing when the output is true. There are four cases.

A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

Consider the first case, when A is true and B is false and C is false. We can create an expression to cover the case: $A \cdot \overline{B} \cdot \overline{C}$. If this were the *only* case where the output is true, this would accurately describe the function. It is an AND expression that contains all the inputs in their true (e.g., A) or complemented (e.g., $\overline{B}$) form. This is called a *minterm*. But there are three other cases. The output is true when $A \cdot \overline{B} \cdot \overline{C}$ is

true or when its the second case $A \cdot B \cdot \overline{C}$ or the third case $\overline{A} \cdot B \cdot C$ or the fourth case $A \cdot B \cdot C$. This behavior forms an OR expression.

$$\text{Out} = A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot C$$

Since this is an OR function applied to AND expressions, it's called a *Sum Of Products* (SOP). All inputs are included in each product term (minterms). So this becomes a canonical expression for the function's behavior: *a sum of products using minterms*. Everyone starting with this behavior will arrive at the identical Boolean expression.

If one works from bottom to top in the truth table, a different order of inputs can be derived.

$$\text{Out} = C \cdot B \cdot A + C \cdot B \cdot \overline{A} + \overline{C} \cdot B \cdot A + \overline{C} \cdot \overline{B} \cdot A$$

This is an *identical* expression (but for commutative ambiguity). It has the same logical operations applied to the same forms of the inputs.

You might notice that when B and C are true, the output is true, independent of A. The resulting expression becomes: Out = $A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + B \cdot C$. This is simpler, but not canonical since it is not composed of minterms.

There's another way to express this function behavior that is rooted in the binary world.

Popeye Logic: In the 1980 movie "Popeye", the title character is in denial about his father being the oppressive "Commodore" in their town, Sweet Haven ("My Papa ain't the Commodore!"). This denial is present when he asks directions to the Commodore's location ("Where ain't he?"). In our multivalued world, this is not so easy. While "north" is unambiguous, "not north" could be any direction except north. But in binary, things are different. We can state when something is true. Or we can use "Popeye Logic" and state when it is NOT false. Let's try Popeye logic on this behavior.

A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

Suppose the only false value was the second one, when A is false, B is true, and C is false. If all the other outputs were true, we could express the function by stating when its *not* this case ("when it ain't $\overline{A}\cdot B\cdot\overline{C}$ "). As in the real world, this is more than one thing; it is all the truth table entries except for $\overline{A}\cdot B\cdot\overline{C}$. But binary makes expressing all the cases easier. In the expression $\overline{A}\cdot B\cdot\overline{C}$, A is false. So whenever A is true, the output is true. Or whenever B is false, the output is true. Or whenever C is true, the output is true. In fact, the function behavior is true whenever A is true *or* B is false *or* C is true ($A+\overline{B}+C$). This expression does not describe when $\overline{A}\cdot B\cdot\overline{C}$ is true, rather it covers all other cases, when "$\overline{A}\cdot B\cdot\overline{C}$ ain't true". The term $A+\overline{B}+C$ is an OR function of all inputs in their true or compliment form. This is a *maxterm*. But this only works if the second case was the only case when the output is false. What about when more than one case is false?

In this example, the output is false when $\overline{A}\cdot\overline{B}\cdot\overline{C}$ or $\overline{A}\cdot B\cdot\overline{C}$ or $\overline{A}\cdot\overline{B}\cdot C$ or $A\cdot\overline{B}\cdot C$. So showing when the output is true requires expressing when it is not any of these cases. It is not $\overline{A}\cdot\overline{B}\cdot\overline{C}$ when A is true or B is true or C is true ($A+B+C$). It is not $\overline{A}\cdot B\cdot\overline{C}$ when A is true or B is false or C is true ($A+\overline{B}+C$). It is not $\overline{A}\cdot\overline{B}\cdot C$ when A is true or B is true or C is false ($A+B+\overline{C}$). It is not $A\cdot\overline{B}\cdot C$ when A is false or B is true or C is false ($\overline{A}+B+\overline{C}$). But since the function output is only true when it is none of these cases, $A+B+C$, $A+\overline{B}+C$, $A+B+\overline{C}$, and $\overline{A}+B+\overline{C}$ must all be true for the function output to be true. So we can express the function:

$$\text{Out} = (A+B+C)\cdot(A+\overline{B}+C)\cdot(A+B+\overline{C})\cdot(\overline{A}+B+\overline{C})$$

Since this is an AND expression of OR terms, it is called a *Product of Sums* (POS). Using maxterms makes this canonical, but different from the sum of products using minterms. There is no direct way to transform a SOP using minterms expression into a POS expression using maxterms or vice versa. Standard forms provide a good way to clearly express a behavior.

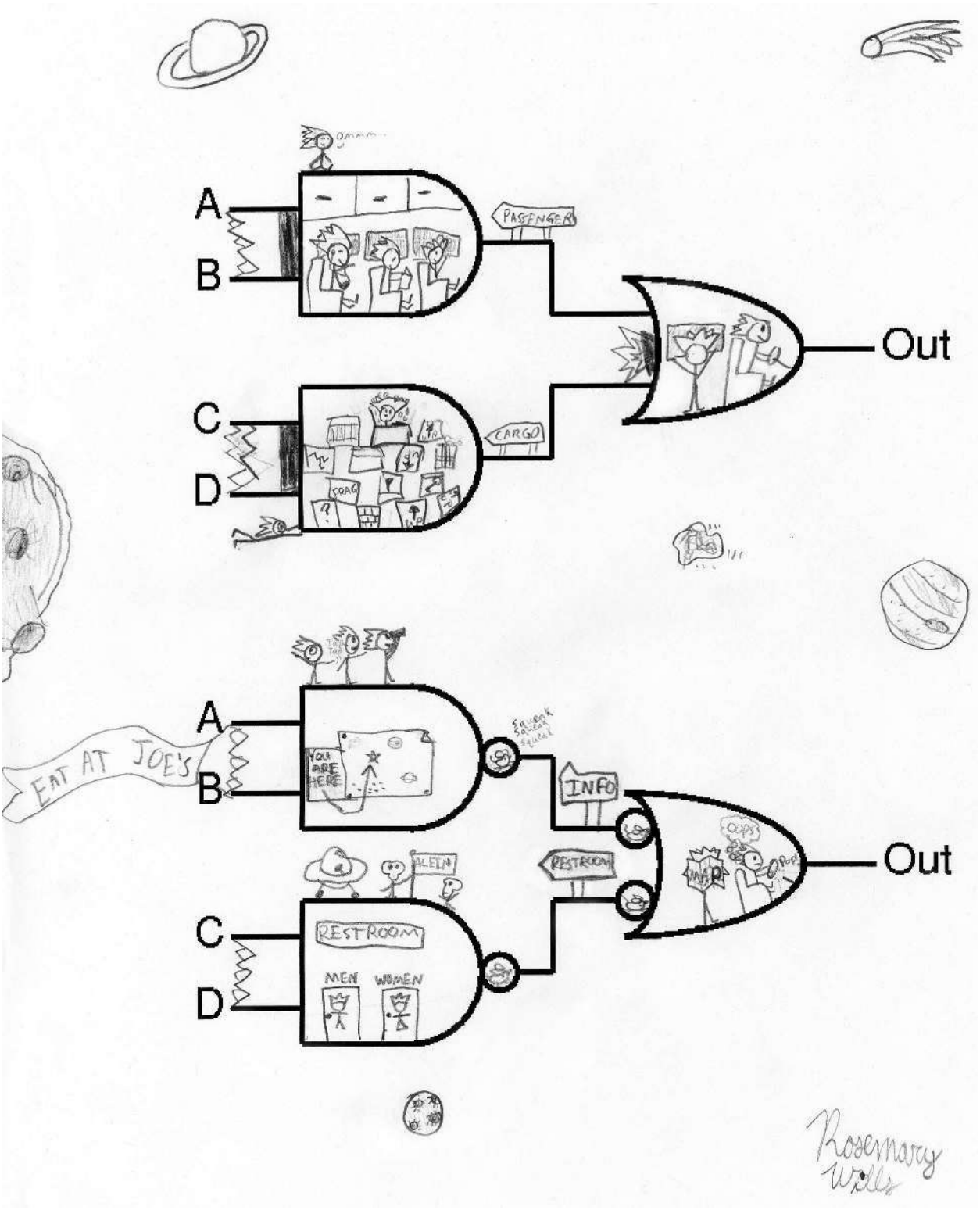
Summary: Boolean algebra is the mathematics of digital computers. Here are the key points:

- Variables have one of two values (0 or 1).
- Functions include AND, OR, and NOT.
- A Boolean expression containing these functions can be used to specify a more complex behavior. Truth tables can also define this behavior.
- Boolean algebra exhibits many familiar and useful properties (plus some new ones).

- DeMorgan's square shows how any logical operation can be transformed into any other logical function by complementing the inputs and/or output.
- DeMorgan's Theorem allows Boolean expressions to be transformed into an equivalent expression that employs different logical functions.
- Standard forms provide a canonical expression in SOP and POS forms.

Designing Computer Systems

Gate Design



Designing Computer Systems

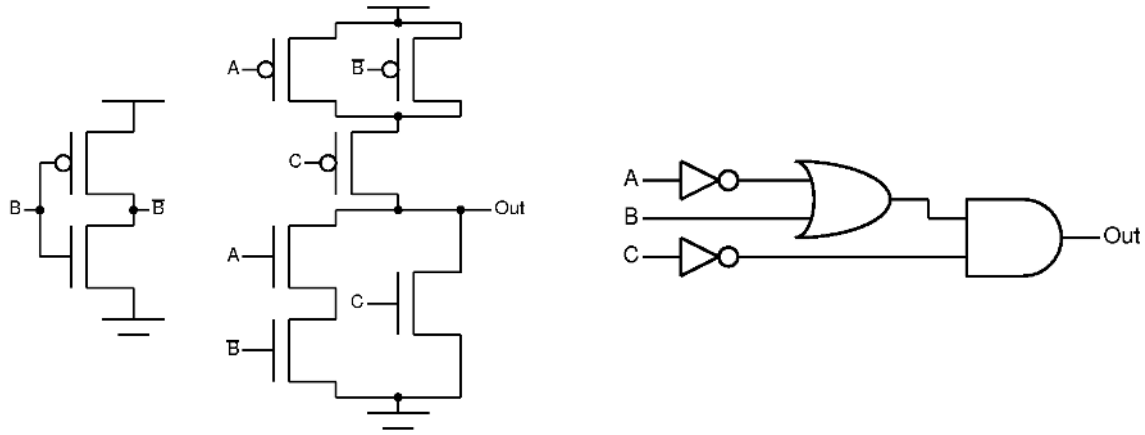
Gate Design

Logical functions that are specified in Boolean algebra, can be implemented with switches and wire. The resulting designs are often the fastest and most efficient implementations possible. But the time and effort required for design is often greater. And switch design requires the manipulating the desired expression so that only input variables are complemented (no big bars). Often after the design process, the desired expression is lost. Is there a way to implement a Boolean expression quickly, without distorting the expression?

Yes!

We can simplify the design process by using more powerful components. We'll work with *gates*, building blocks that match the logical operations in our expression. Wires still connect outputs to inputs. Data still is digital. In fact, we use switches to implement these new gate abstractions.

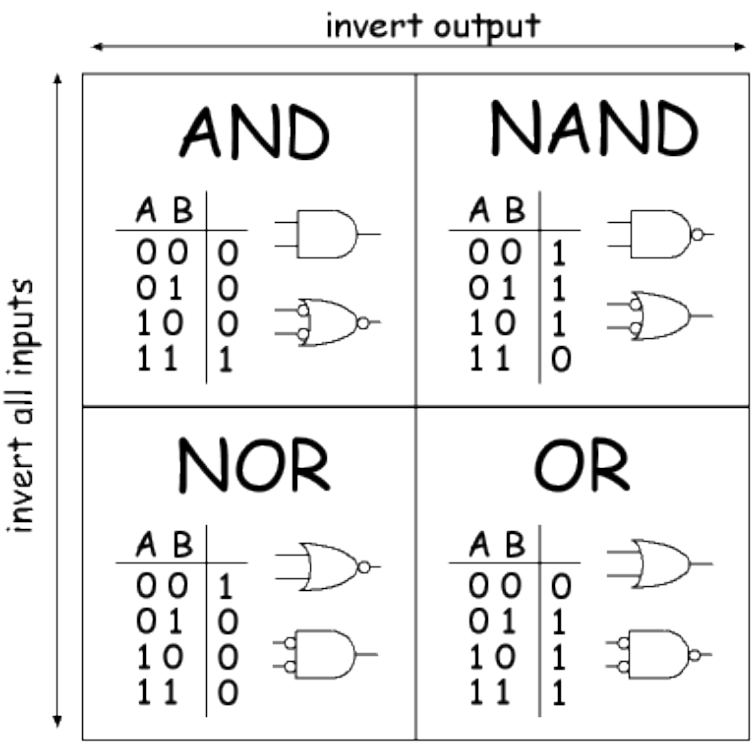
Suppose we want to implement the expression Out = $(\overline{A} + B) \cdot \overline{C}$. Using switches, details of the implementation technology (e.g., P-type switches are active low and pull high) are visible and affect the design. Using gates, technology details are hidden and the desired expression is easily discerned. Unfortunately this gate design is twice as slow and uses twice as many switches. Convenience has a cost!



Of course, gate design can be improved if the choice of implementation components is not tied to the desired expression. For CMOS technology, NAND and NOR gates require fewer switches than AND and OR. So in this example, the OR and AND gates can be replaced by NOR gates. Unfortunately, this requires DeMorgan transformations of the desired expression. This distorts the

expression, increases design time, and increases the possibility for errors. Why can't we leave the expression alone?

We can. DeMorgan's square suggests that all gate types have two equivalent representations. One is built on an AND body. The other employs an OR body.



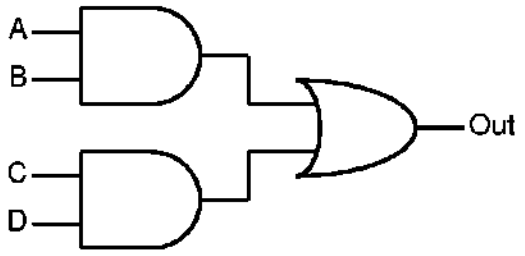
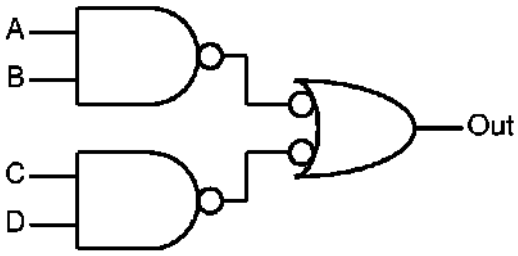
At first this duplicity may seem a complication. But it can be productively used to separate *specification* from *implementation*. Here's how.

When a desired expression is derived, AND and OR functions provide the relationship between binary variables. The choice of gate types can improve the implementation efficiency and performance. But it should not distort the meaning of the desired expression.

Since each gate function can be drawn with either an AND or OR body, a desired logical function can be realized using any gate type by simply adding a bubble to the inputs and/or output. Unfortunately, a bubble also changes the behavior by inverting the signal. But bubble pairs (bubbles at both ends of a wire) cancel out and the behavior is unchanged.

So we can draw a gate design using the logical functions in the desired expression. Then we can then add bubble pairs to define the implementation gate type without changing the gate body (i.e., distort the expression being captured).

Here's an example: Out = $A \cdot B + C \cdot D$

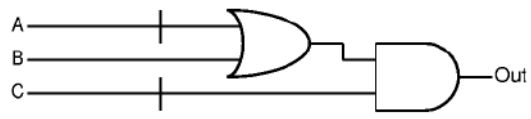
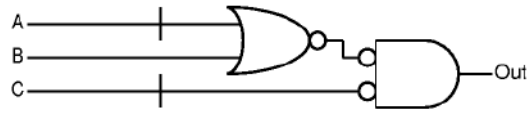
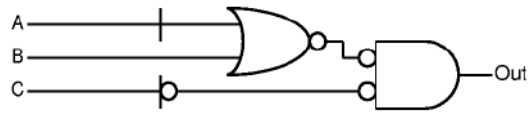
	
using AND and OR	using NAND

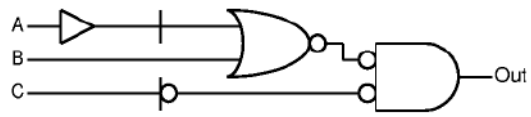
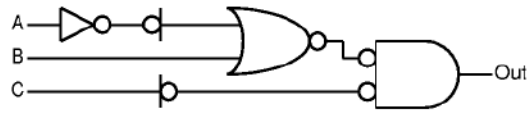
If we draw the circuit directly using AND and OR gates, the expression is clear. But the implementation cost is high (18 switches). If we preserve the gate bodies, but add bubble pairs, the behavior is unchanged. But the implementation cost is lowered (12 switches).

A bar over an input or subexpression indicates that an inversion is required. This bar is part of the desired expression and should be preserved along with gate bodies. But the implementation must include, in some way, the required inversion of the signal. Again, bubble pairs can help.

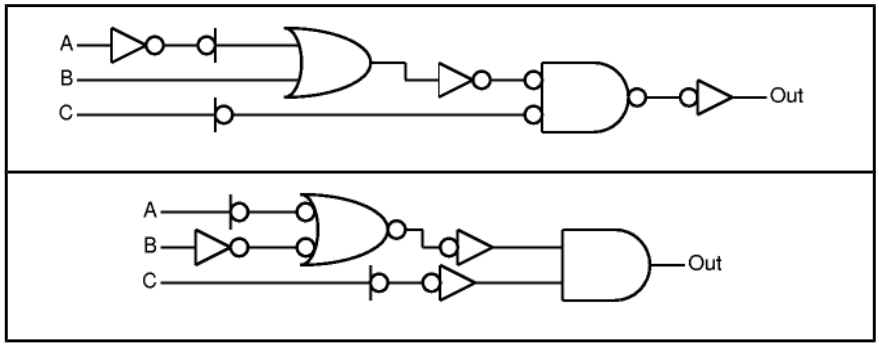
Let's add bars to our gate design, not as active devices, but as a notational reminder that a real signal inversion is needed. Then during implementation, we'll place exactly one bubble on the bar. Bubble pairs are added to change the implementation without changing the behavior. If one bubble in a bubble pair does not actually cause a real inversion (because it is a notation), the signal will be inverted (by the other bubble).

Consider the expression Out = $(\overline{A} + B) \cdot \overline{C}$.

	First let's draw the expression as a circuit. Bars are added where they appear in the expression. This is *not* an implementation.
	Now suppose we want to use NOR gates for the implementation. This is partially accomplished by adding a bubble pair between the gates. But the bubble on the lower input of the AND gate is unmatched.
	We can put the matching bubble on the bar, indicating that we really do want the inversion. The bubbled bar does not actually do anything; it's just notation. The bubble on the AND input does the required inversion.

	We also need a buhhle on the A input. But we can't add the matching bubble to the OR gate body without changing its implementation. Instead let's add a buffer on the A input.
	Now we can add a bubble pair between the buffer and bar. The implementation gate type is NOR, all bubbles are matched, and all bars have exactly one bubble. This implementation is complete.

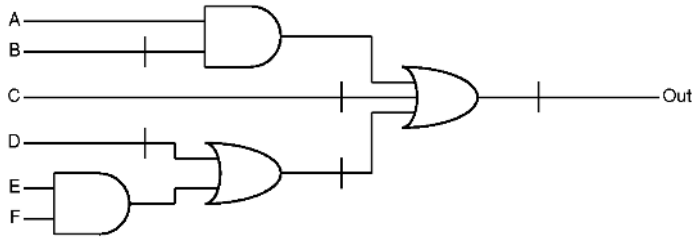
The gate implementation of this example requires ten switches. That's two more than the switch design. But it is six less that the original gate implementation. Note that by ignoring bubble pairs and buffers, we still see the desired expression, graphically displayed. Specification and implementation are now decoupled.



We can also implement the design using OR or AND gates. DeMorgan's equivalence allows any gate body to be implemented in any multi-input gate type. In CMOS technology, OR and AND implementations requires more switches (18 for this design).

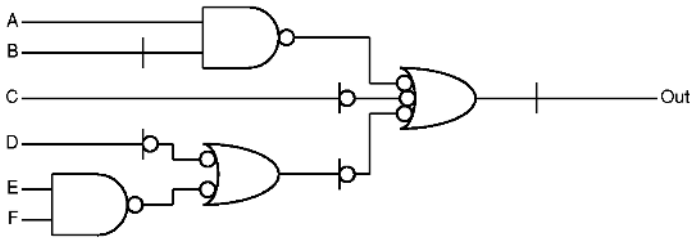
Here's another example: $Out = \overline{A \cdot \overline{B} + \overline{C} + \overline{\overline{D} + E \cdot F}}$

We start with the expression as a graph using gates and bars. It captures the function. But its not an implementation.

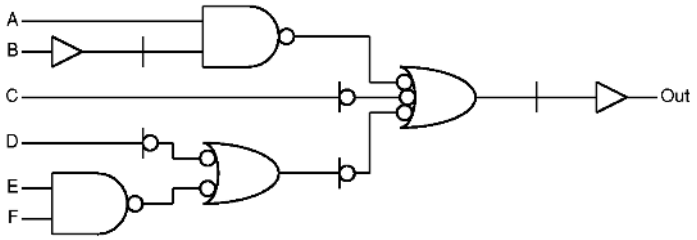


Now we select a good implementation gate. One doesn't always need to use one gate type for a design. The technology may favor an implementation approach. In CMOS, inverting gates (NAND and NOR) use fewer switches than non-inverting

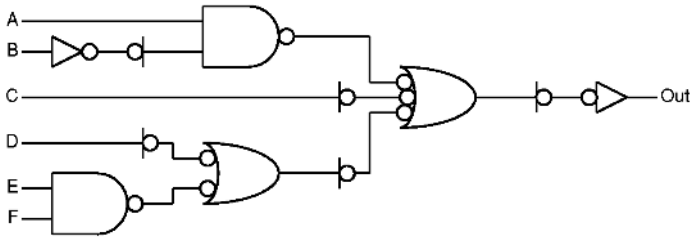
gates (AND and OR). In this case, we use NAND gates. Bubble pairs are added to gate bodies to transform the implementation.



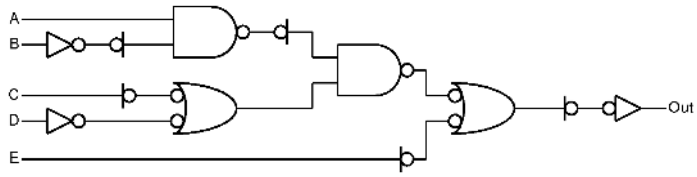
Buffers are added where bubbles pairs are still needed for bars.



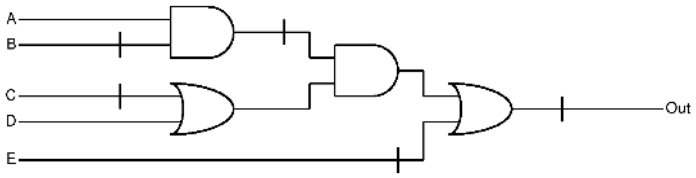
Finally bubbles pairs are added to complete the implementation.



Desired Expression: This gate design technique is called *mixed logic*. Its name is derived from the fact the implementation combines positive (active true) and negative (active false) logic. A key advantage is the ability to preserve the desired expression (i.e., the expression the designer specified) in an implementation. For example, the circuit below is built with NAND gates.

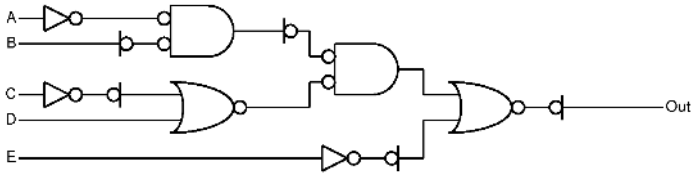


To see the desired expression, ignore the bubbles and buffers and read the expression from the gate bodies and bars.



This expression is $Out = \overline{\overline{A \cdot \overline{B}} \cdot (\overline{C} + D) + \overline{E}}$

If we wish to reimplement it, say using NOR gates, we just move around bubble pairs, adding and removing buffer bodies as needed.

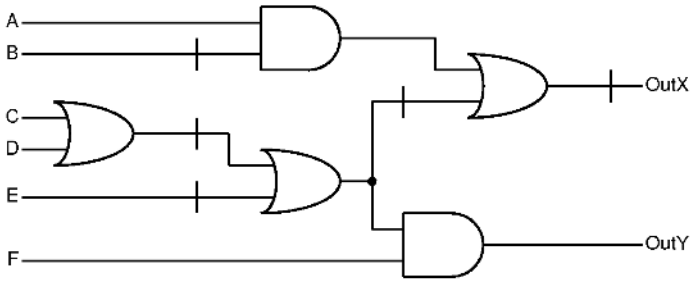


Note that the desired expression has not changed.

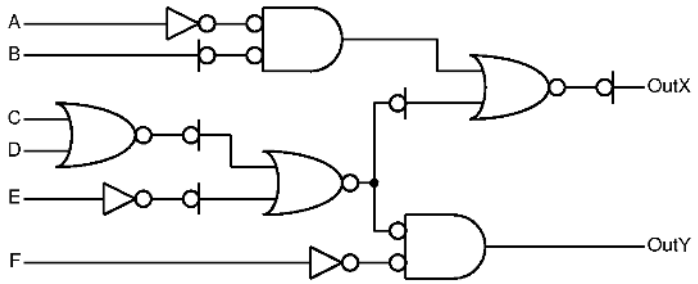
Common Subexpressions: Often in design, a logical expression is required for multiple outputs. It would be wasteful to build multiple copies. We can just use a computed value in multiple places. This is called *fanout* since a single gate output fans out to multiple gate inputs. Consider these two equations.

$$OutX = \overline{A \cdot \overline{B} + \overline{\overline{C + D} + \overline{E}}} \qquad OutY = (\overline{C + D} + \overline{E}) \cdot F$$

Both expressions require the subexpression $\overline{C + D} + \overline{E}$ so it can used in creating both outputs.



During implementation, here using NOR gates, special attention is needed for fan out connections. In order to ignore a bubble on an output, there must be a bubble on each input that uses it. The bubble pair on the output of the subexpression becomes a *bubble trio*.



Propagation Delay

When considering the speed of circuits, one must look at underlying technology – here, switches and wire. The two parameters that dominate delay are *resistance* and *capacitance*.

Resistance is an abundance of charge carriers. It is proportional to the availability of charge carriers brought in by the electronic field on the gate. It is proportional to the charge carrier mobility. Metals are a charge carrier gas. They have clouds of electrons that are easy to acquire. A semiconductor has more bound charge carriers that are harder to acquire and more difficult to move around with a field, leading to higher resistance to pull a node to a high voltage or to a low voltage.

The charge it takes to reach a high or low voltage is proportional to a node's *capacitance C*. Capacitance forms automatically when two insulated conductors are near one another separated by a dielectric material. The higher the dielectric constant, the higher capacitance. Dislike charges attract to form an electric field when the insulated conductive dislike charges appear on the conducting surfaces (for example, the polysilicon gate oxide on the switch).

The bigger the switch, the more charge carriers are needed to charge the switch voltage to the On level, which is a product of the switch resistance R and the gate capacitance C. RC is proportional to the propagation delay through the switch.

CompuCanvas models delay as unit delay, which assumes a fixed constant delay through each gate.

Energy

Energy is proportional to the product of induced voltage on a node and channel conductance, which is the inverse of the resistance through a conducting channel of a turned on switch. This resistance is proportional to the major charge carrier mobility. Conductors have an electron cloud of free electrons that can be easily

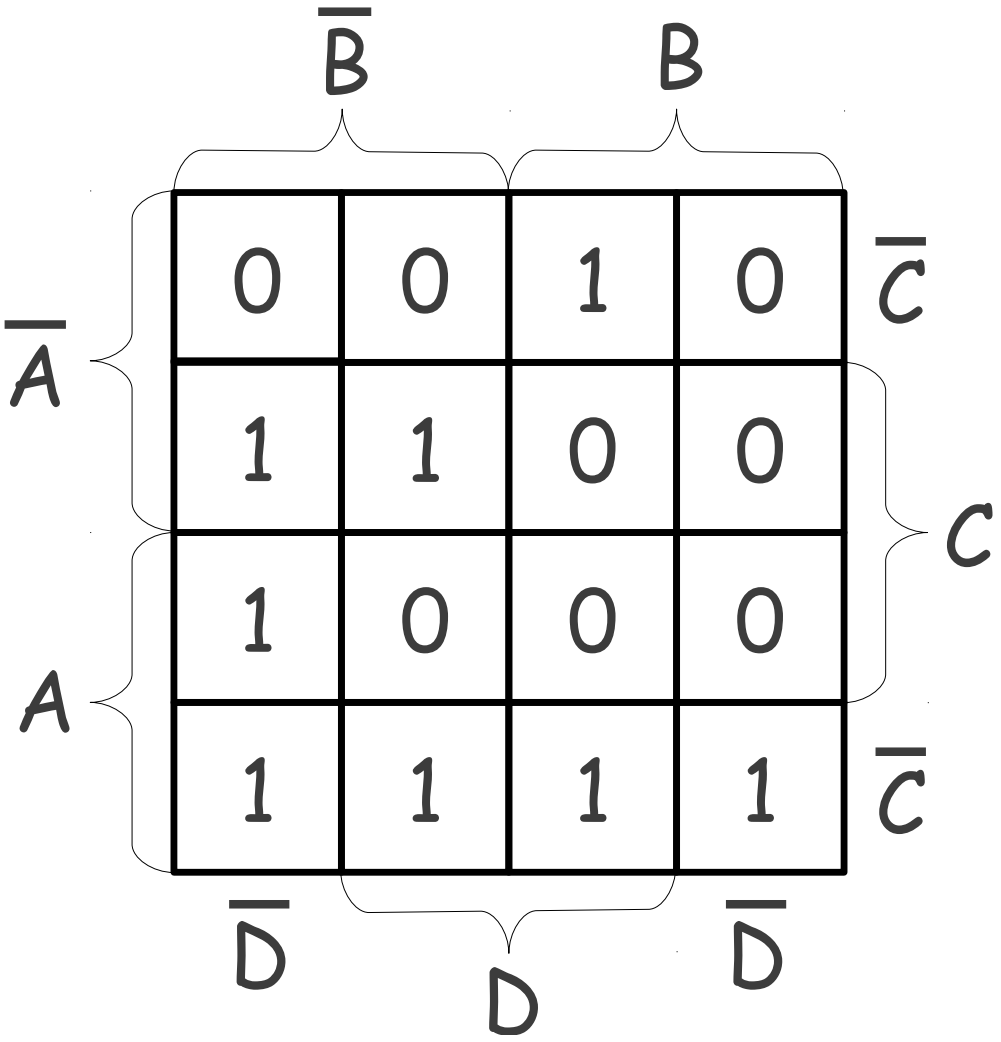
moved by a field. Doped silicon has limited charge carrier mobility that limits conductance and energy.

Summary: Gate design of Boolean expression is a fast and clean alternative to switch design.

- Gate design is easier to understand than switches and is independent of implementation technology.
- Gate implementations often require more switches than direct switch implementations, but designs can still be optimized.
- DeMorgan's gate equivalence allows specification and implementation to be separated using mixed logic design.
- Mixed logic design also preserves the designer's desired expression.

Designing Computer Systems

Simplification



Designing Computer Systems

Simplification

Using DeMorgan's theorem, any binary expression can be transformed into a multitude of equivalent forms that represent the same behavior. So why should we pick one over another? One form provides a canonical representation. Another provides a clear representation of the desired function. As engineers, we want more than functionality. We crave performance and efficiency!

So what improves an expression? … fewer logical functions. Every logical function requires computational resources that add delay and/or cost energy, switches, design time, and dollars. If we can capture the desired behavior with fewer logical operations, we are building the Ferrari of computation; or maybe the Prius?

Simplify Your Life: There are many techniques to simplify Boolean expressions. Expression reductions (e.g., $X + \overline{X} \cdot Y \rightarrow X + Y$) is good. But it's not obvious what to reduce first, and it's hard to know when you're finished. An intuitive method can be seen in a truth table … sometimes. Here's an example. Consider the expression:

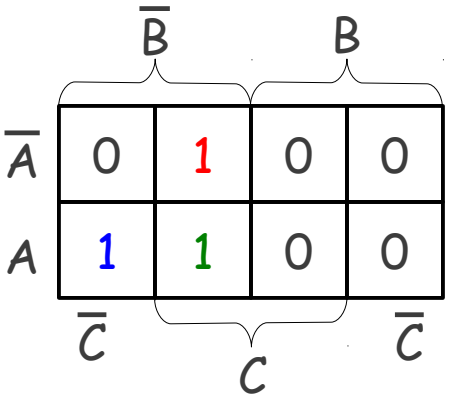
$$Out = A \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$$

In truth table form, one might notice that the red and green terms suggest that when B is false (zero) and C is true (one), Out is true (one) no matter what state A is in. The blue and green terms express a similar simplification. If A is true and B is false, Out is true independent of C. A simplified expression $Out = \overline{B} \cdot C + A \cdot \overline{B}$ expresses the same behavior with three dyadic (two input) logical operations versus eight for the original expression.

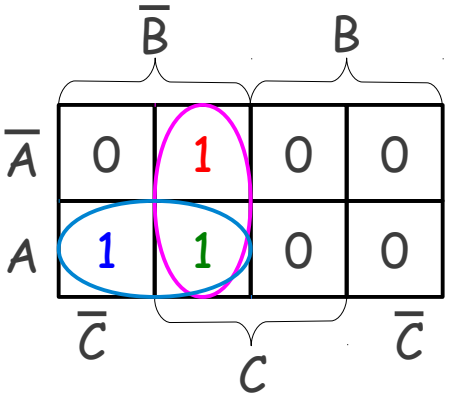
A	B	C	Out
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	0
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	0

This simplification is intuitive. In all cases when a subexpression (e.g., $\overline{B}\cdot C$) is true, the output is true, then including extra qualifying terms is unnecessary. Unfortunately, truth tables lack uniform adjacency of these simplifying groupings.

For a small number of variables, a Karnaugh Map (K-map) displays the same behavior information in a different way. A K-map are composed of a two-dimensional map displaying the output for every combination of input values. But these combinations are arranged so that horizontal or vertical movement results in exactly one variable changing. Here's the K-map for the function being considered.



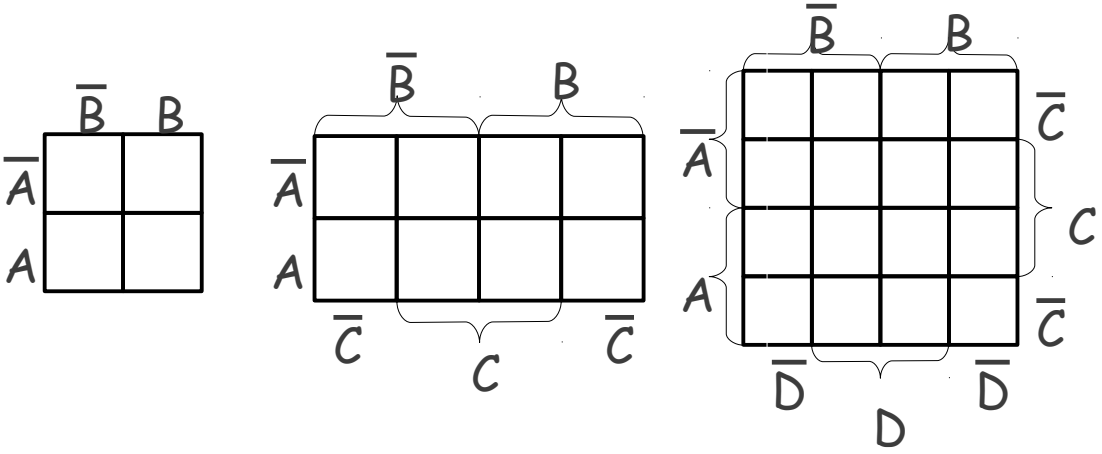
In this map, the top row includes all input combinations where A is false. The second row includes all combinations where A is true. The left two columns include input combinations where B is false. The right two columns cover when B is true. The outermost columns include input combinations where C is false. The middle two columns include cases where C is true.



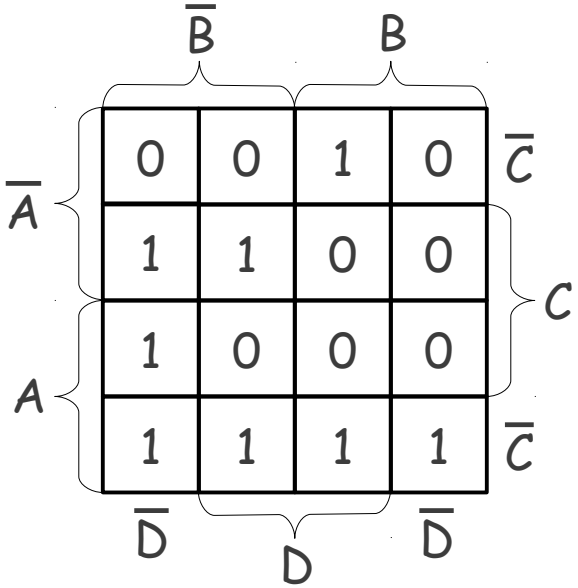
In this arrangement, adjacent ones (true outputs) suggests an opportunity for simplification. The red and green ones can be grouped into a single term covering all combinations where B is zero and C is one ($\overline{B}\cdot C$). The adjacent blue and green ones are grouped to cover where A is one and B is zero ($A\cdot\overline{B}$). Since a simplified expression must cover all cases when the output is one, these terms can be

combined to express the function's behavior: Out = $\overline{B}\cdot C$ + $A\cdot\overline{B}$, a simplified sum of products expression.

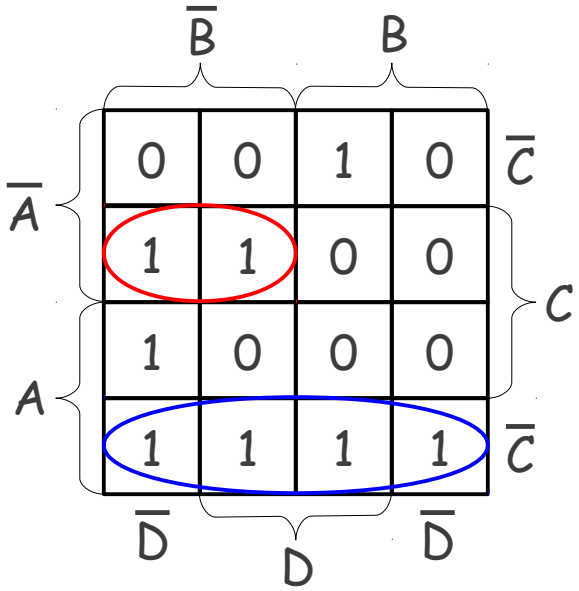
Two-dimensional K-maps accommodate two-, three-, and four-variable expressions. Larger K-maps (five- and six-variable) are possible in three dimensions. But they are error prone and better simplification techniques exist.



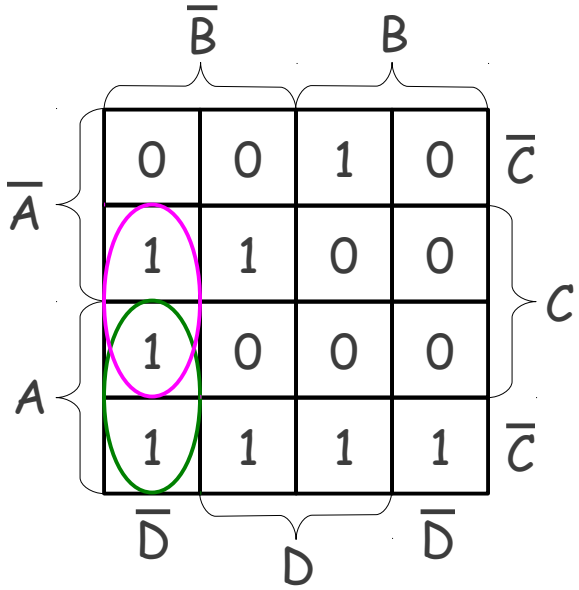
Just like a truth table, the K-map describes a function's behavior by giving the output for every combination of the inputs. But adjacency in a K-map also indicates opportunities for expression simplification. Here's a four-variable K-map.



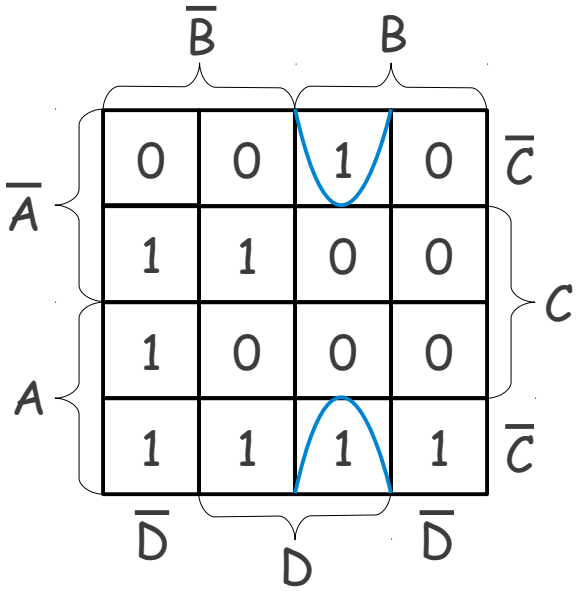
The behavior represented by this K-map could be represented as a truth table. Adjacent ones are opportunities for simplification. The size of groupings are given as (width x height). So a (2x1) grouping is two squares wide and one square high.



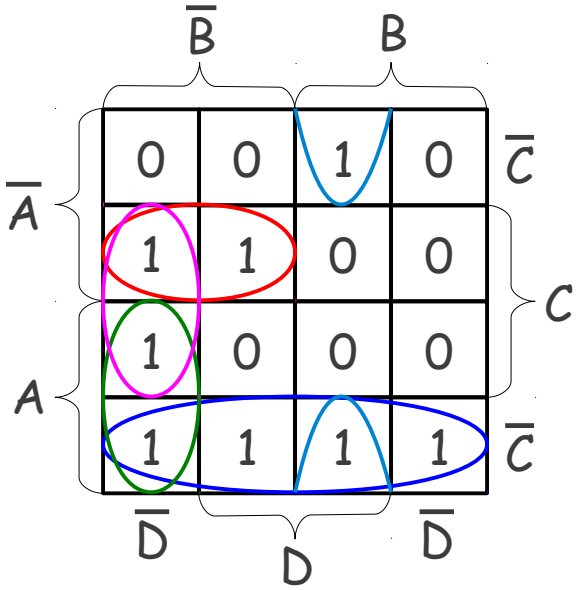
The second row grouping (2x1) represents all cases where A is false, B is false, and C is true ($\overline{A}\cdot\overline{B}\cdot C$). The bottom row (4x1) is all cases where A is true and C is false ($A\cdot\overline{C}$). Larger groupings lead to smaller terms. But the grouping has to be describable as *all cases where variables have a certain value*.



The first column contains three adjacent ones. In this candidate grouping, B and D are zero. But it is not *all cases* where B and D are zero. A (1x3) grouping is not a describable grouping. Instead these adjacent ones are cover by two overlapping (1x2) groupings: $C\cdot\overline{D}$ and $A\cdot\overline{D}$. Overlapping groups are okay, so long as one grouping is not subsumed by another grouping. Groupings of ones always have power of two dimensions (1, 2, 4).



Adjacency extends at the K-map edges. The one in the third column of the top row can be grouped with the corresponding one in the bottom row. This grouping represents all cases where B is true, C is false, and D is true ($B \cdot \overline{C} \cdot D$). Here are all legal groupings in this K-map.



Note that while groupings overlap, no grouping falls completely within another. The grouped terms: $\overline{A} \cdot \overline{B} \cdot C$, $A \cdot \overline{C}$, $C \cdot \overline{D}$, $A \cdot \overline{D}$, and $B \cdot \overline{C} \cdot D$, represent all candidate terms in the simplified expression. But are all terms necessary?

The objective is to correctly define the behavior by expressing all cases when the output is one. This means selecting groupings that cover all true outputs in the K-map. Sometimes this requires all groupings. Sometimes not. In this K-map, three groupings, $\overline{A} \cdot \overline{B} \cdot C$, $A \cdot \overline{C}$, and $B \cdot \overline{C} \cdot D$, include a true not covered by any other grouping.

This makes them *essential* for the simplified expression. If they are not included, the behavior is not accurately defined. But in the example, these essential groupings don't cover all the ones in the K-map. Additional groupings are needed.

Two groupings $C\cdot\overline{D}$ and $A\cdot\overline{D}$, are not essential (*non-essential*) but cover the missing one in the behavior ($A\cdot B\cdot C\cdot\overline{D}$). Since they have the same number of variables, and the same cost to implement, either will provide an equivalently simplified expression.

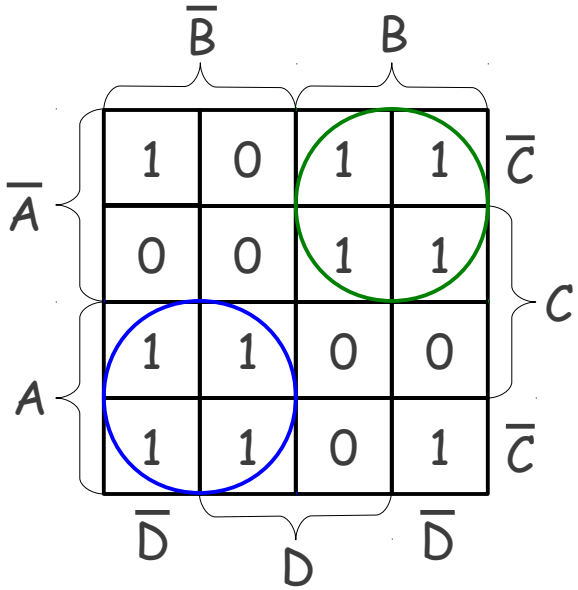
Out = $\overline{A}\cdot\overline{B}\cdot C + A\cdot\overline{C} + C\cdot\overline{D} + B\cdot\overline{C}\cdot D$ Out = $\overline{A}\cdot\overline{B}\cdot C + A\cdot\overline{C} + A\cdot\overline{D} + B\cdot\overline{C}\cdot D$

These two simplified expressions are significantly less expensive to implement than the canonical sum of products expression.

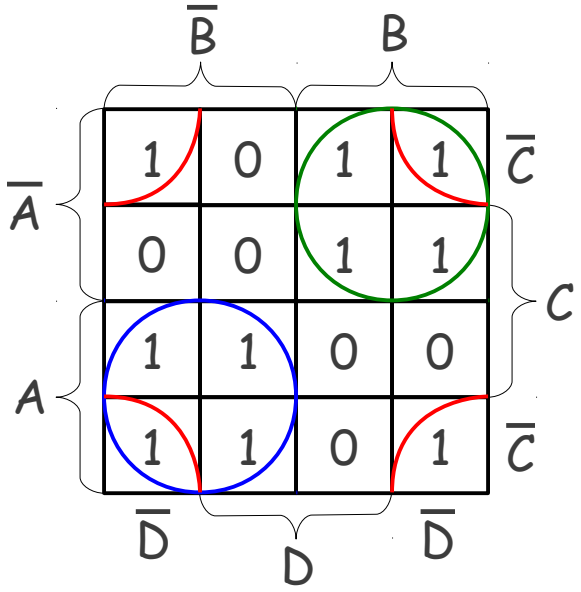
Out = $\overline{A}\cdot B\cdot\overline{C}\cdot D + \overline{A}\cdot\overline{B}\cdot C\cdot\overline{D} + \overline{A}\cdot\overline{B}\cdot C\cdot D + A\cdot\overline{B}\cdot C\cdot\overline{D} + A\cdot\overline{B}\cdot\overline{C}\cdot\overline{D} + A\cdot\overline{B}\cdot\overline{C}\cdot D + A\cdot B\cdot\overline{C}\cdot D + A\cdot B\cdot\overline{C}\cdot\overline{D}$

Parlance of the Trade: Due to its arithmetical origin, these groupings are named *Prime Implicants*. PIs for short. Essential prime implicants are always included in the simplified expression because they exclusively contain one of the grouped elements. Non-essential PIs may or may not be needed, depending on whether essential PIs cover the selected outputs. Formally, this simplification process is defined as *minimally spanning the selected outputs*.

But aren't the selected outputs always true? Not always. More on this later. Here's another example.



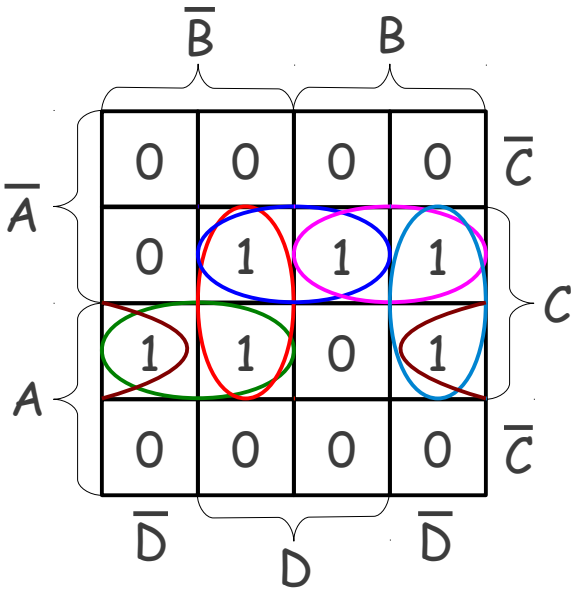
Two (2x2) PIs stand out: all cases where A is false and B is true ($\overline{A}\cdot B$), and all cases where A is true and B is false ($A\cdot\overline{B}$). But how to group the remaining true outputs?



The final (2x2) PI groups the four corners, covering all cases where C and D are zero ($\overline{C}\cdot\overline{D}$). The edges of a K-map connected; there's just no good way to draw it on a two-dimensional plane. If the vertical edges are joined, the map becomes a cylinder. If the ends of the cylinder are joined, it becomes a donut (torus). In two-dimensions, one must look for connections on the edges of K-maps.

Since all PIs are essential and necessary to span true outputs in the behavior, the simplified sum or products expression is Out = $\overline{A}\cdot B + A\cdot\overline{B} + \overline{C}\cdot\overline{D}$

Here's another example.



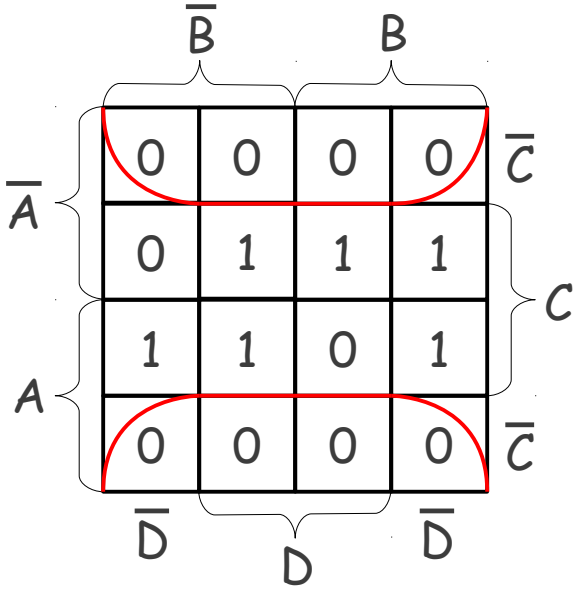
This example contains six overlapping PIs: $A\cdot\overline{B}\cdot C$, $\overline{B}\cdot C\cdot D$, $\overline{A}\cdot C\cdot D$, $\overline{A}\cdot B\cdot C$, $B\cdot C\cdot\overline{D}$, and $A\cdot C\cdot\overline{D}$. What makes them interesting is that *none are essential*. Sometimes there is an urge to ignore non-essential PIs when simplifying a K-map. This example

demonstrates the need to record all legal PIs before considering minimal spanning. There are two, equally simplified sum of products expressions for this behavior.

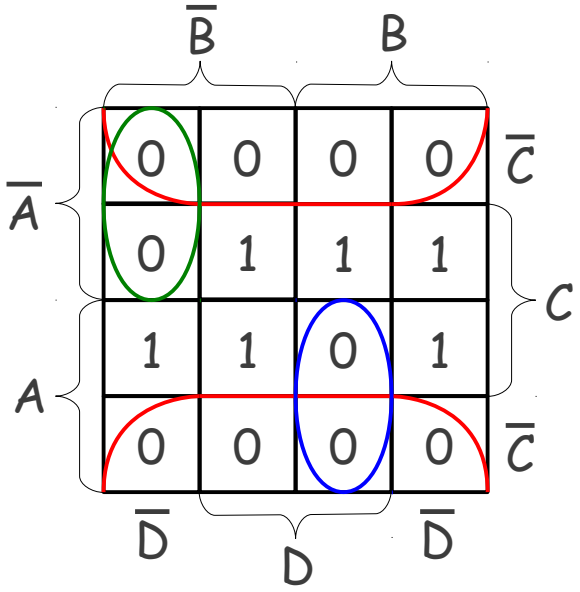
Out = $A\cdot\overline{B}\cdot C + \overline{A}\cdot C\cdot D + B\cdot C\cdot\overline{D}$ Out = $\overline{B}\cdot C\cdot D + \overline{A}\cdot B\cdot C + A\cdot C\cdot\overline{D}$

Incomplete listing of PIs might not expose both of these expressions due to false appeasement of essentialness. It might even yield a less simplified result.

SoP versus PoS: Boolean Algebra explains the duality where a function's behavior can be defined by stating where the output is true (sum of products) or by stating where the function is not false (product of sums). This applies in K-maps. In the first case (SoP), product terms define a group of true outputs (a PI). A spanning set of these groupings is ORed together to form the simplified expression. In the second case (PoS), groupings represent where the output is not true, but false. Since the simplified expression must still represent where the behavior is true, a grouping (PI) must express *states not in the grouping*. Here's the same example, targeting a simplified product of sums expression.



The largest grouping of zeros (false) covers the cases where C is false. As in the SoP process, a (4x2) grouping is drawn. But labeling this group $\overline{C}$ is not helpful, since the goal is expressing when the behavior is true. This grouping include many of the false outputs and none of the true outputs. So the PI should represent when it is *not in this grouping*, namely C. Being outside the red PI C is not enough; additional Pis are required to guarantee a true output.



Just as ones where grouped in SoP, here zeros are grouped. A (1x2) grouping of false outputs represents when A is true and B is true and D is true. But the PI for this grouping represents cases *not in this grouping*. That occurs when A is false OR B is false OR D is false ($\overline{A}+\overline{B}+\overline{D}$). It is ORed because the output is not in this grouping if any of the variables are false. Again, this doesn't mean the output is true, its just not in this grouping of zeros.

Another (1x2) grouping of false outputs occurs when A is false and B is false and D is false. This PI is expressed as A is true or B is true or D is true (A+B+D). These are all cases not in the green grouping.

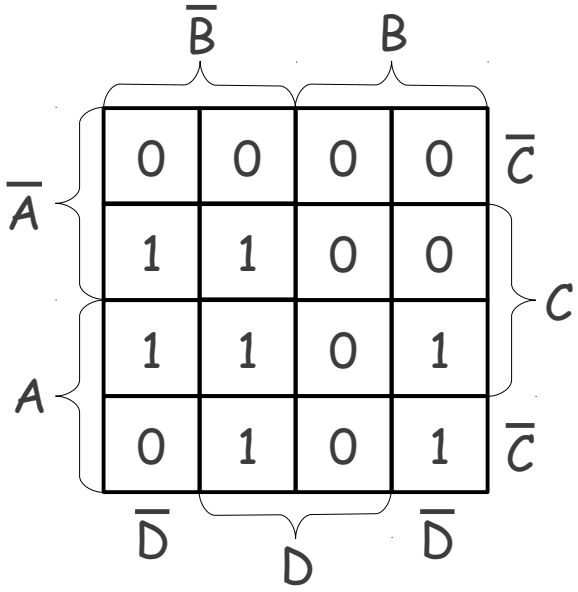
So how does the simplified expression show when the behavior is true? By showing when it it is not false! Each of the PIs represents cases that are not in one of the groups of zeros. If the PIs span all false outputs, and all terms are true, the output must be true. In this example, all PIs are essential (i.e., they contain a false output not included in any other PI). So the simplified product of sums expression is:

$$\text{Out} = C \cdot (\overline{A}+\overline{B}+\overline{D}) \cdot (A+B+D)$$

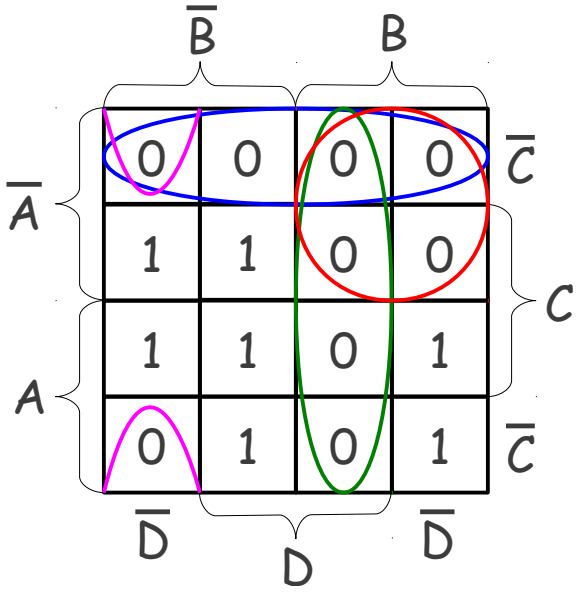
Note that this simplified PoS expression has no obvious relationship to the equivalent simplified SoP expressions:

$$\text{Out} = A\cdot\overline{B}\cdot C + \overline{A}\cdot C\cdot D + B\cdot C\cdot\overline{D} = \overline{B}\cdot C\cdot D + \overline{A}\cdot B\cdot C + A\cdot C\cdot\overline{D}$$

This PoS example has unusual symmetries in its PIs. Here's a different example.



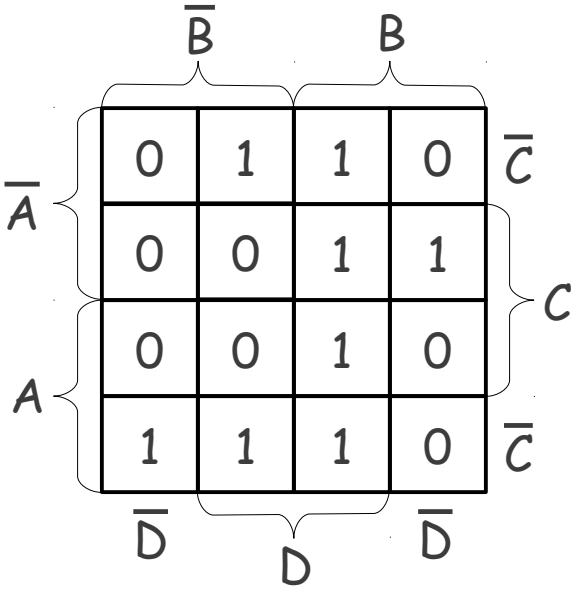
The groupings of false outputs mirror the SoP technique for true outputs. Only here the PI is labeled for cases outside its group. The (4x1) in the top row contains cases where A and C are false. The PI is labeled (A+C). The (1x4) in the third column includes cases where B and D are true. The PI is labeled ($\overline{B}$+$\overline{D}$). The upper right quadrant (2x2) is selected when A is false and B is true. The PI is (A+$\overline{B}$). The (1x2) in the first column is cases where B, C and D are all false. The PI is (B+C+D).



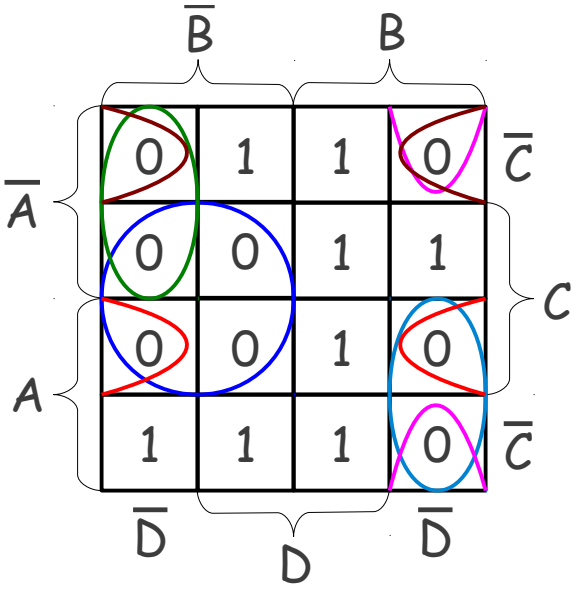
All PIs are essential. So the simplified PoS expression is:

$$\text{Out} = (A+C) \cdot \overline{B}+\overline{D} \cdot (B+C+D)$$

Here's another example.



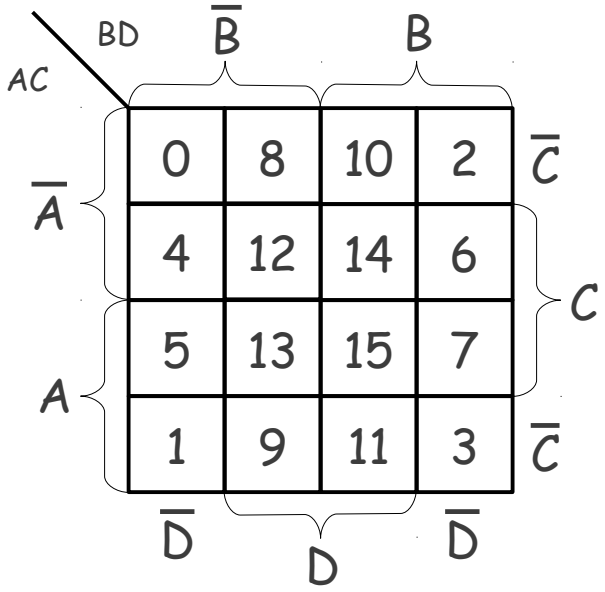
The PIs include the following: $(B+\overline{C})$, $(A+B+D)$, $(A+C+D)$, $(\overline{B}+C+D)$, $(\overline{A}+\overline{B}+D)$, $(\overline{A}+\overline{C}+D)$.



Only $(B+\overline{C})$ is essential. No other PI exclusively contains a false output. But this simplified expression is not ambiguous. The minimal span of zeros yields:

$$\text{Out} = (B+\overline{C}) \cdot (A+C+D) \cdot (\overline{A}+\overline{B}+D)$$

Six of One; A Half Dozen of Another: Some folks find it advantageous to place the input variables in the upper left corner of a K-map and assign truth table row numbers to each square. While truth table/K-map correspondences are never row/column sequential, the numbering can have some semi-sequential ordering. In our examples, it looks like this:

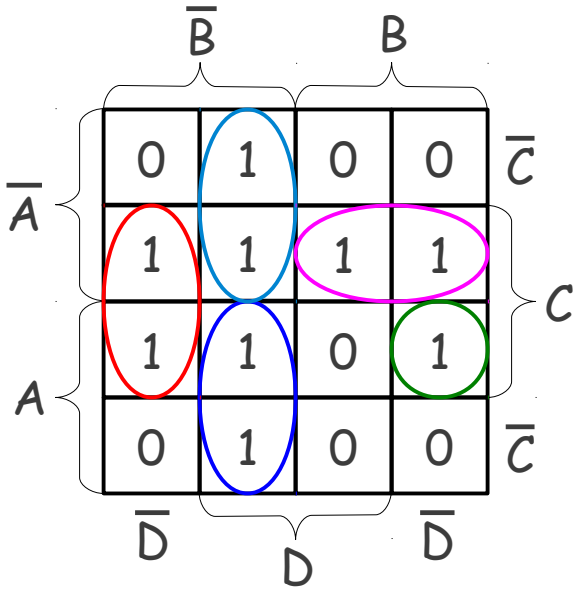


So what difference does it make? In terms of simplification, none. Reordering the input variables scrambles the K-map cells. But any proper ordering, where vertical and horizontal movements change exactly one variable, yields the same PIs and the same simplified expression.

Simplifying Boolean Expression: This ordering helps simplify Boolean expressions. Suppose a behavior, defined as a Boolean expression, is to be simplified.

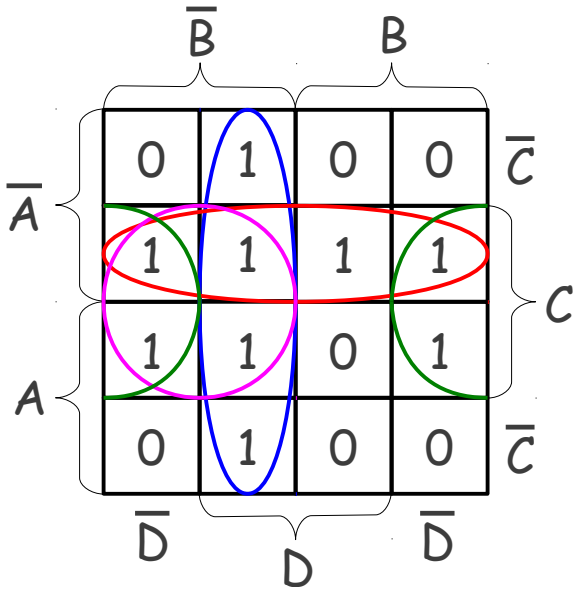
$$Out = A\cdot\overline{B}\cdot D + A\cdot B\cdot C\cdot\overline{D} + \overline{B}\cdot C\cdot\overline{D} + \overline{A}\cdot B\cdot C + \overline{A}\cdot\overline{B}\cdot D$$

In this SoP expression, each product term represents a grouping of true outputs. The ungrouped cases represent false outputs. Here's the mapping of each term.



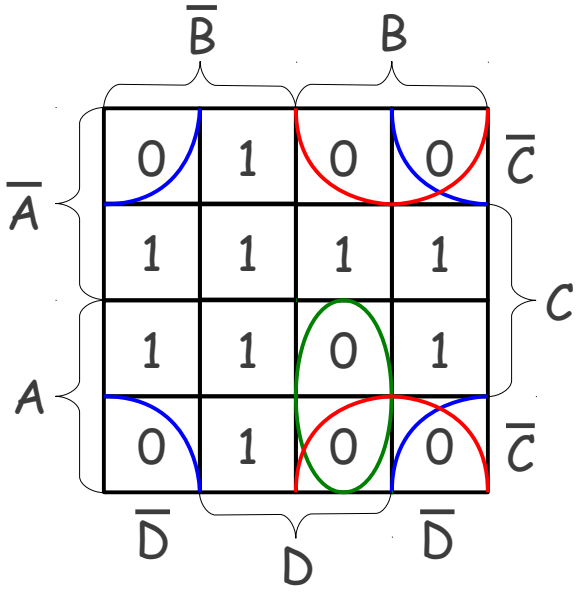
Once the expression's behavior is defined, it can be simplified as either a SoP or PoS expression. In SoP simplification, there are four PIs ($\overline{B}\cdot D$, $\overline{A}\cdot C$, $C\cdot\overline{D}$, $\overline{B}\cdot C$), three of which are essential and span the true outputs.

$$Out = \overline{B}\cdot D + \overline{A}\cdot C + C\cdot\overline{D}$$



In PoS, false outputs are grouped. Here three PIs are identified. All are essential.

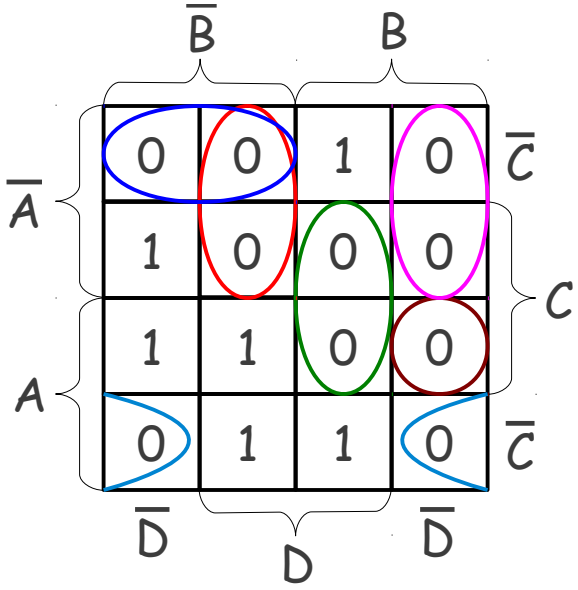
$$Out = (C+D)\cdot(\overline{B}+C)\cdot(\overline{A}+\overline{B}+\overline{D})$$



Both simplified expressions require fewer dyadic logical operations than the original expression (five for SoP, six for PoS versus 15 for original).

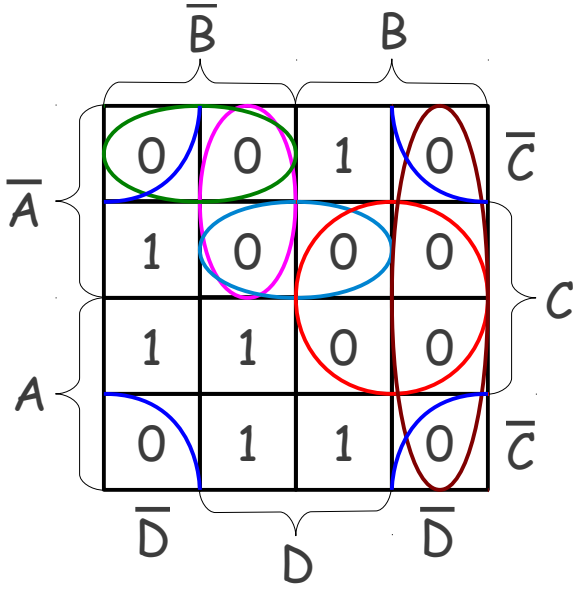
Here's a PoS expression to be simplified. Each term represents potentially true outputs *outside* a grouping of known false values. So zeros can be added for each term. Finally, the ungrouped cases are assigned a true value (one).

$$\text{Out} = (A+B+C) \cdot (A+B+\overline{D}) \cdot (\overline{B}+\overline{C}+\overline{D}) \cdot (A+\overline{B}+D) \cdot (\overline{A}+C+D) \cdot (\overline{A}+\overline{B}+\overline{C}+D)$$



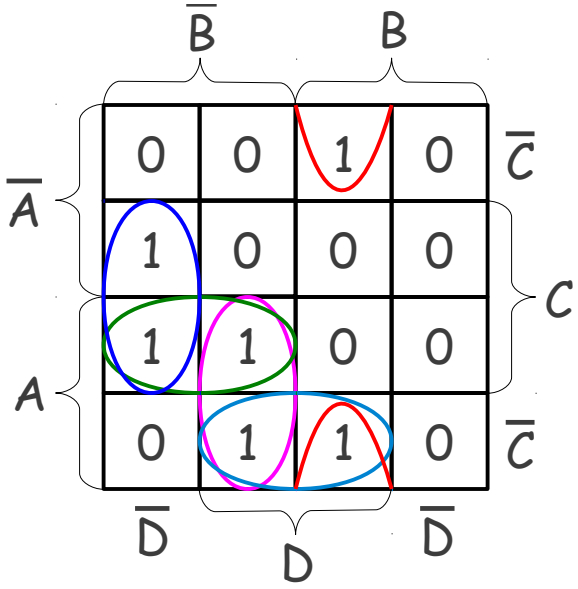
The PoS simplification produces six PIs: $(C+D)$, $(\overline{B}+\overline{C})$, $(A+B+C)$, $(A+B+\overline{D})$, $(A+\overline{C}+\overline{D})$, $(\overline{B}+D)$, two of which are essential: $(C+D)$, $(\overline{B}+\overline{C})$. One non-essential is require to span false outputs: $(A+B+\overline{D})$.

$$\text{Out} = (C+D) \cdot (\overline{B}+\overline{C}) \cdot (A+B+\overline{D})$$



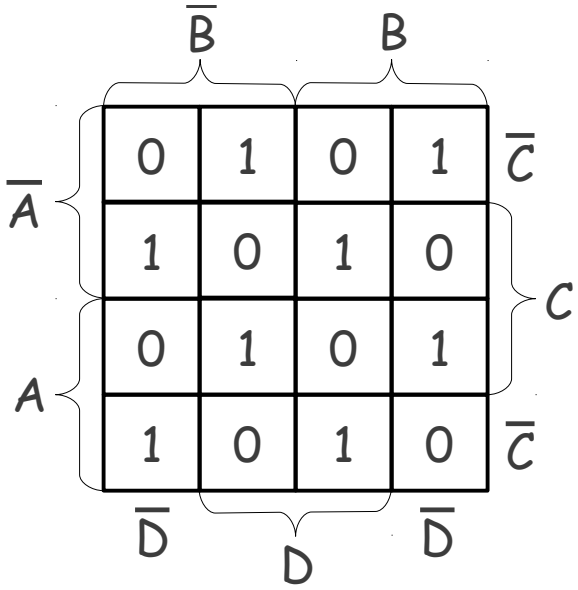
This behavior also can be simplified to a SoP expression. It produces five PIs: $\overline{B}\cdot C\cdot\overline{D}$, $A\cdot\overline{B}\cdot C$, $A\cdot\overline{B}\cdot D$, $A\cdot\overline{C}\cdot D$, $B\cdot\overline{C}\cdot D$, two are essential: $\overline{B}\cdot C\cdot\overline{D}$, $B\cdot\overline{C}\cdot D$. One non-essential PI is required to cover true outputs: $A\cdot\overline{B}\cdot D$.

$$\text{Out} = \overline{B}\cdot C\cdot\overline{D} + B\cdot\overline{C}\cdot D + A\cdot\overline{B}\cdot D$$



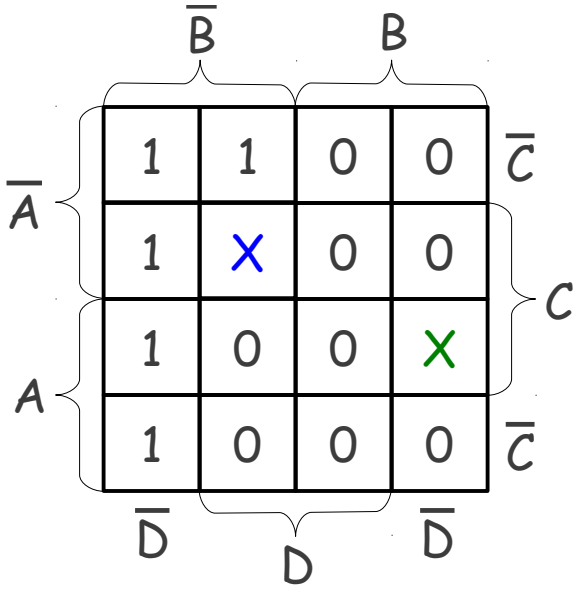
Again, simplification reduced implementation cost from 18 logical operations (original) to six (PoS) and eight (SoP). It's clear from these examples that there is no direct translation of a product term to a sum term, or vice versa.

Simplification Nightmare: Is there an *unsimplifiable behavior*? Yes. Here's odd parity (XOR).

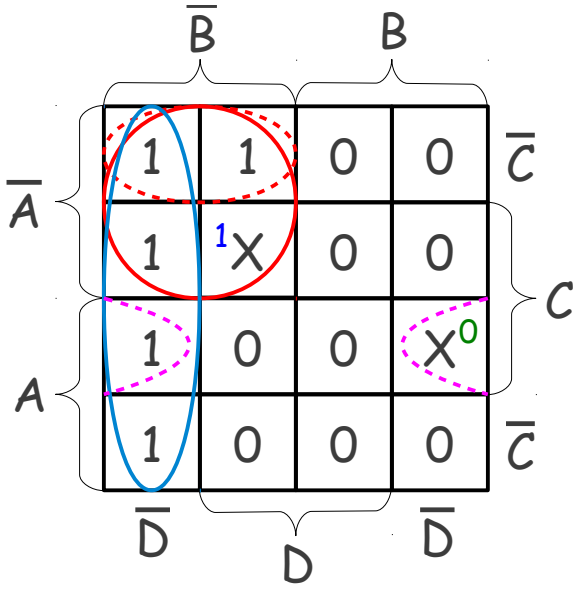


Odd and even parity (XOR and XNOR) toggle their outputs with every horizontal or vertical movement. Exactly one variable is changing. So the number of ones also toggles between odd and even. For this reason, there are no adjacent true or false outputs. SoP terms are always *minterms* and PoS terms are always *maxterms*. Parity is a useful function. But it is relatively costly.

What If You Don't Care?: Sometimes a function's behavior is unimportant for certain combinations of the inputs. Maybe it cannot occur. Or when a combination of the inputs occurs, the output is not used. This is recorded in the truth table as an "X" for the output. When an "X" occurs in a K-map, it can be defined as either a zero or one to improve the simplification process. Here's an example.



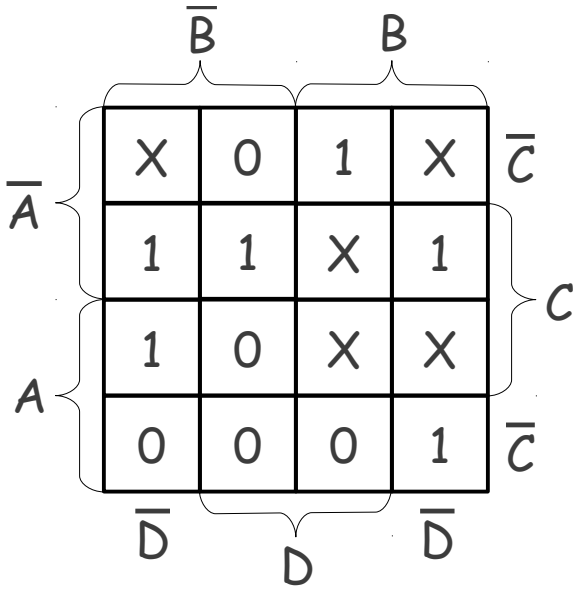
For two combinations of inputs: $\overline{A}\cdot\overline{B}\cdot C\cdot D$ and $A\cdot B\cdot C\cdot\overline{D}$, the output is unspecified and listed as don't care "X". This underspecification of the behavior permits the don't cares to be specified to reduce implementation cost.



To simplify this behavior to a SoP expression, the don't cares must be specified. For the first case, the choice of a true output doesn't add additional PIs. Rather it increases a PI from a (2x1) $\overline{A}\cdot\overline{B}\cdot\overline{C}$ to a (2x2) $\overline{A}\cdot\overline{B}$. This eliminates a logical operation from the simplified expression. In the second case, a false output is selected to eliminate the need for an additional PI: $A\cdot C\cdot\overline{D}$, to cover it. The simplified expression is found:

$$\text{Out} = \overline{A}\cdot\overline{B} + \overline{B}\cdot\overline{D}$$

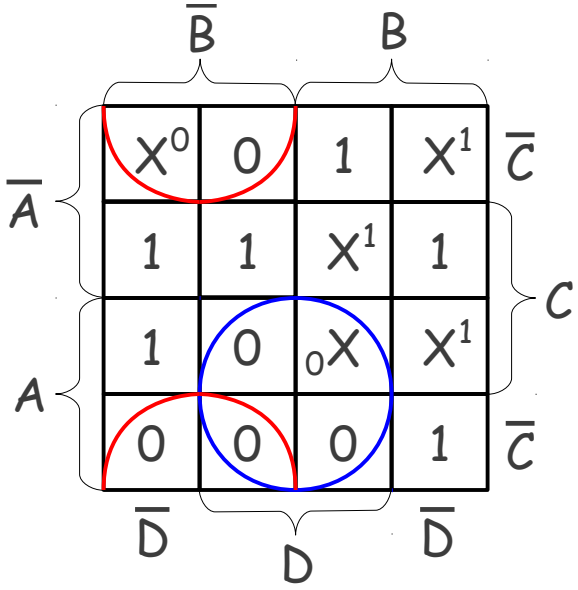
Sometimes specifying don't cares can take some thought. Here's another example.



Suppose a simplified PoS expression is required. Don't cares are specified to maximize the size of false output PIs while minimizing the number of PIs.

Here, the don't cares a specifies false to create two (2x2) PIs: (B+C), ($\overline{A}$+$\overline{D}$). The remaining don't cares are specified as true and don't contribute to groupings. The simplified expressing is:

Out = (B+C) · ($\overline{A}$+$\overline{D}$)



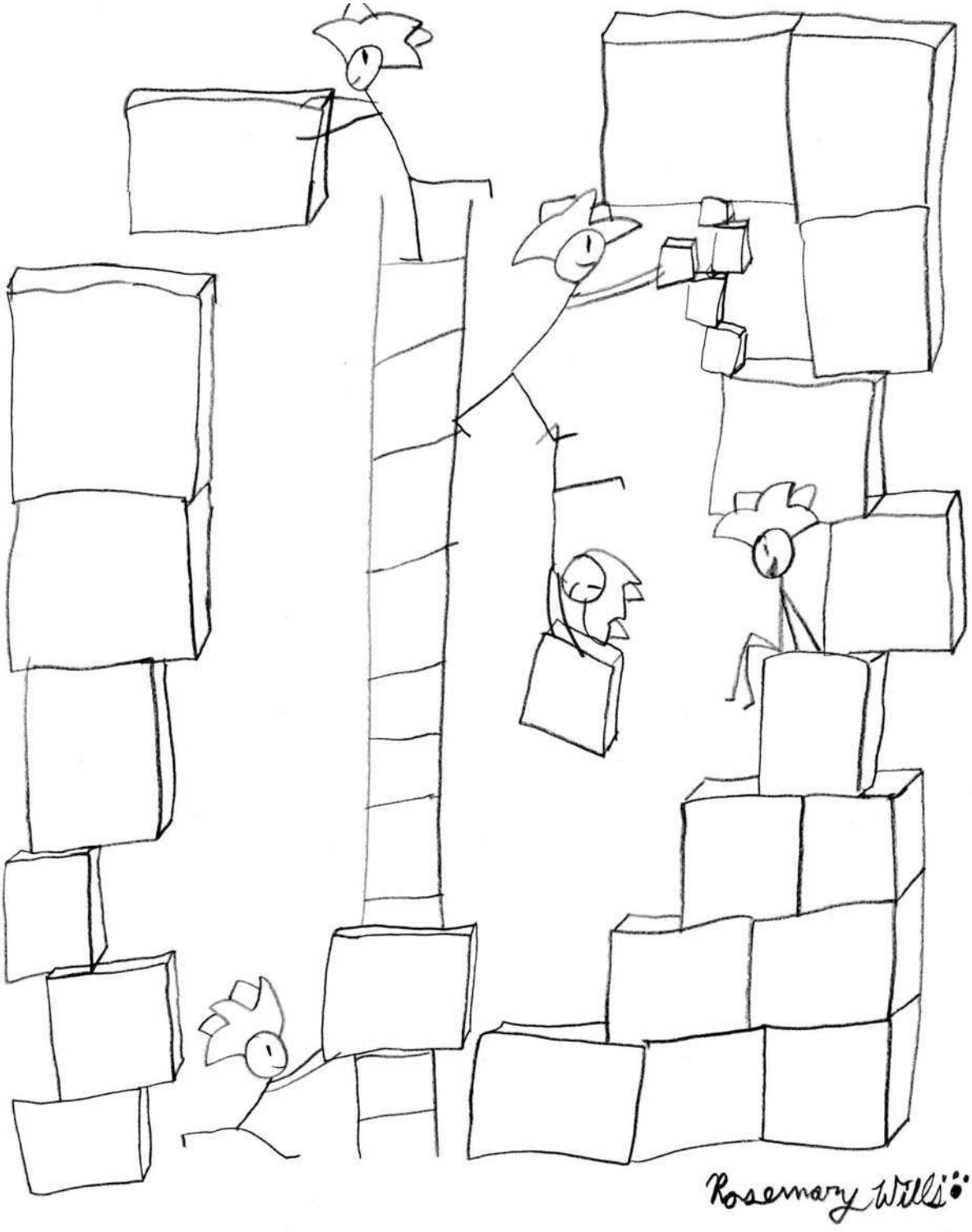
More than Four Variables?: How does handle more complex behaviors with over four variables. Other simplification techniques, like the Quine-McCluskey algorithm, are not limited in the number of variables. They are less intuitive to humans. But they are more amenable to computers and can be programmed into design tools like Espresso.

Summary: This chapter addresses methods for Boolean expression simplification using Karnaugh maps.

- Term reduction is possible when all combinations of a subexpression have a true (or false) output.
- A Karnaugh map specifies a behavior where vertical and horizontal movement changes exactly one variable.
- For a simplified sum of products expressions, true outputs are grouped. For product of sums expression, false outputs are grouped.
- A prime implicant is a group of adjacent true or false outputs that are of power of two (1, 2, 4) dimensions, and not enclosed in a larger grouping.
- A product PI lists where the output is true. A sum PI lists where the output is not in a specified grouping of false outputs.
- Boolean expressions and behaviors with don't cares can be simplified.

Designing Computer Systems

Building Blocks



Designing Computer Systems

Building Blocks

A logic gate employs many switches to achieve a more complex behavior. Now we'll use gates to build an even more specialized, more powerful set of building blocks.

Encoders / Decoders: A binary digit or *bit* is the fundamental representational element in digital computers. But a bit by itself is limited to two states: 0 and 1. Fortunately, many bits can be grouped to form more interesting strings. This can be done in different ways. For example, three bits in a car might indicate whether (A) the door is open (B) the headlights are on, and (C) the seatbelt is fastened. These conditions are independent and can occur in any combination. So each bit in the string has a simple coding:

A	door	B	headlights	C	seatbelt
0	open	0	off	0	unfastened
1	closed	1	on	1	fastened

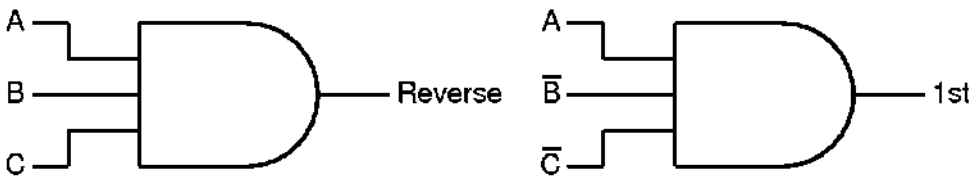
In the transmission, three bits might represent its operation state:

C	B	A	state
0	0	0	neutral
0	0	1	1st gear
0	1	0	2nd gear
0	1	1	3rd gear
1	0	0	4th gear
1	0	1	5th gear
1	1	0	fault
1	1	1	reverse

In this case, only one state can exist at a time. So rather than having eight separate bits to represent the transmission's state, three bits are used to *encode* one of the eight possible states. This requires additional decoding when this information is used. In order to illuminate the "reverse" indicator on the dash board (and turn on the backup lights), all three bits are required to generate the control signal. A different set of values for A, B, and C indicates a different state. The Boolean expression for two of these states are:

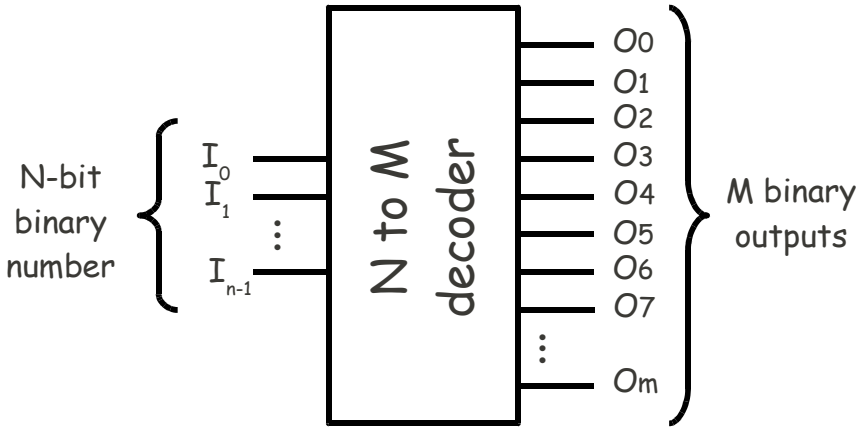
$$\text{Reverse} = A \cdot B \cdot C \qquad\qquad \text{1st gear} = A \cdot \overline{B} \cdot \overline{C}$$

The logic to decode these conditions from the three bits is shown below.



Encoding things in this ways reduces the number of bits to be stored and communicated (a good thing). But it requires logic to *decode* a condition from that signal. Let's explore a more common type of decoder: a multi-bit binary decoder.

N-to-M Binary Decoder: In many systems, it is useful to encode one of several states as a multi-bit binary number. In the transmission example, we used a three number value to represent on of eight conditions. More generally we can use N bits to represent 2^N unique states. Although we can use logic to decode each state independently, we can envision a generic decoder that takes an N bit binary number as input, and produces M separate outputs. Here's a N to M decoder.



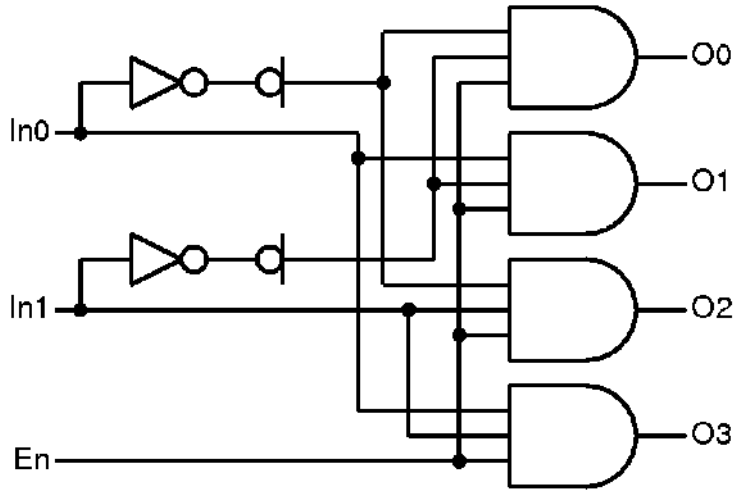
The value of the input, 0 to (2^N – 1) causes the corresponding output to be asserted (set true), while the other outputs remain false. If the input is "101", O5 is high while all other outputs are low. Because there are times when the input data may not be valid, an enable input controls when the decoding process takes place. When this is low, the input binary number is ignored and all outputs are low.

Here are the behaviors of several binary decoders: 1 to 2, 2 to 4, and 3 to 8. Note that I_2, I_1, I_0 form a one, two, or three bit binary number whereas O0 – O7 are just outputs labeled with a number. The input binary number determines which output is asserted.

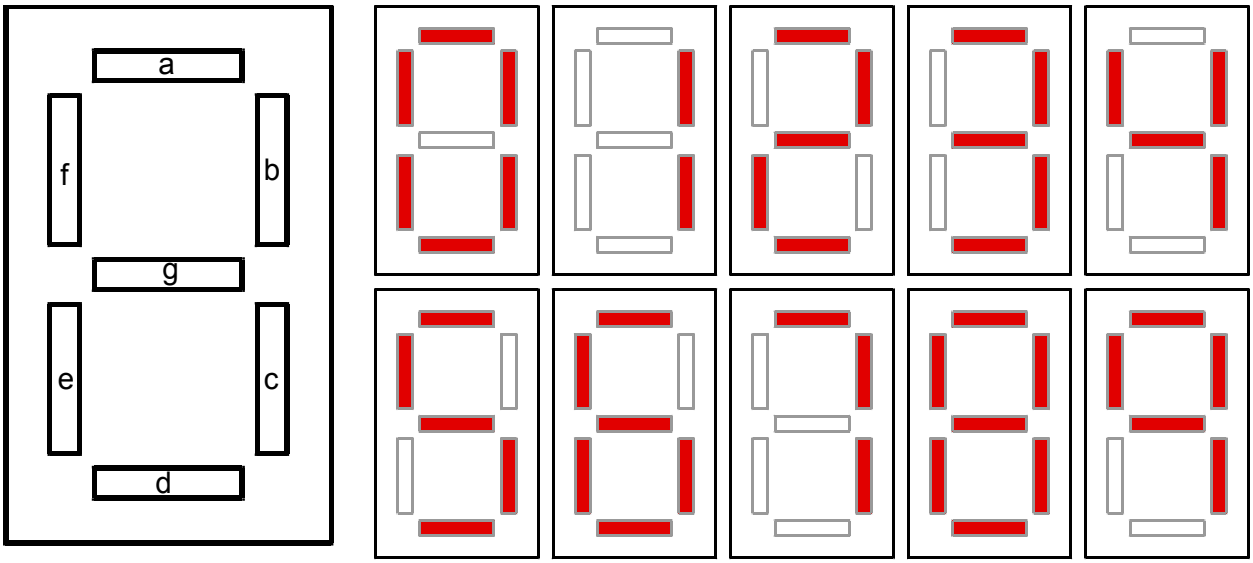
I_2	I_1	I_0	En	$O0$	$O1$	$O2$	$O3$	$O4$	$O5$	$O6$	$O7$
X	X	X	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0
0	0	1	1	0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0	0
1	0	1	1	0	0	0	0	0	1	0	0
1	1	0	1	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

Implementing a decoder with gates is straightforward. Since each output is high in only one case, a sum of products expression contains a single minterm. For a 2 to 4 decoder (the blue truth table), the output expressions are easily expressed and implemented.

$$O0 = \overline{I_1} \cdot \overline{I_0} \cdot En \qquad O1 = \overline{I_1} \cdot I_0 \cdot En \qquad O2 = I_1 \cdot \overline{I_0} \cdot En \qquad O3 = I_1 \cdot I_0 \cdot En$$



BCD to 7 Segment Decoder: Not all decoders assert one of M outputs. Sometimes the decoded outputs have a different requirement. For example, many numerical displays use a seven segment display to show a decimal digit. The digits are labeled a, b, c, d, e, f, and g. Any decimal digit (0-9) can be created by turning on different combination of these named segments. A four bit binary coded decimal (BCD) can be used as input to a decoder than switches on the proper segments for the corresponding digit character. When enable is low, all segments are switched off, blanking the display.

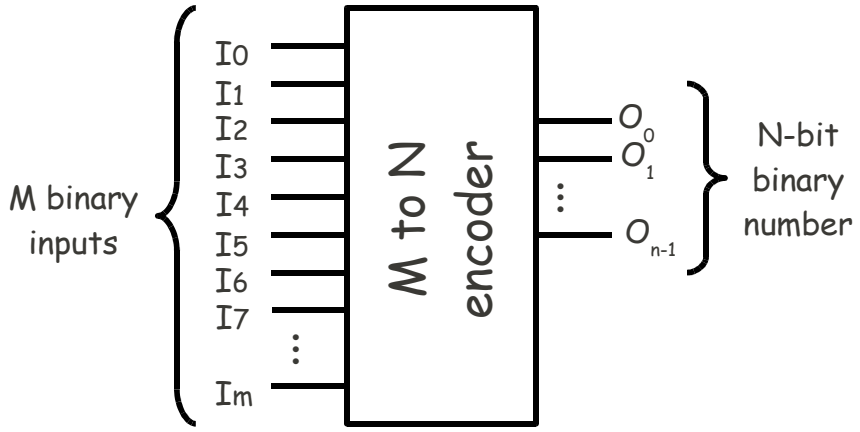


I_3	I_2	I_1	I_0	En	O_a	O_b	O_c	O_d	O_e	O_f	O_g
X	X	X	X	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	0
0	0	0	1	1	0	1	1	0	0	0	0
0	0	1	0	1	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	1	0	0	1
0	1	0	0	1	0	1	1	0	0	1	1
0	1	0	1	1	1	0	1	1	0	1	1
0	1	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	0	1	1

Here multiple outputs are asserted for each code. But the gate implementation is still direct. Each of the seven outputs can be expressed and simplified as a function of the four bit binary number and enable.

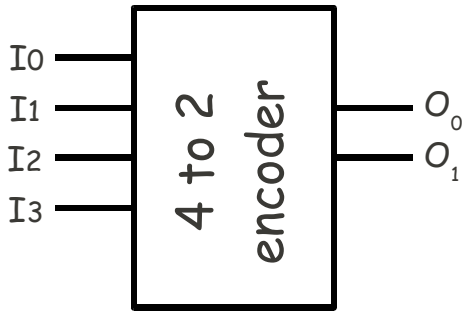
Decoders extract the *coded* information in a binary string to assert one or more outputs. They are a widely used building block. But how do we get the encoded word in the first place? Perhaps by using an *encoder !*

Encoders: If the job of decoders is to turn an N-bit coded binary string into M uncoded outputs, then an encoder must perform the reverse process: turning an asserted input into a coded binary string (a N-bit binary number). This is more complicated than is sounds.



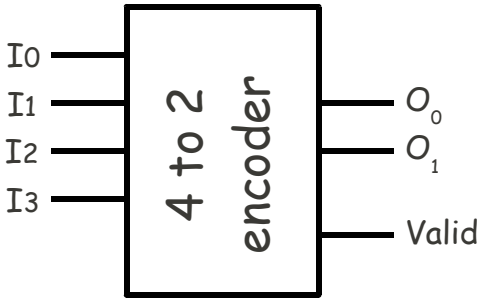
Consider the behavior of a 2 to 4 encoder.

I0	I1	I2	I3	O_1	O_0
1				0	0
	1			0	1
		1		1	0
			1	1	1



When a single input is asserted, the output string corresponds to the number of the asserted input. For example, when I2 is asserted, the output string is "10" which represents a binary "2". An ambiguity occurs when no inputs are asserted. What should the output be. Since all 2^N output values already have a defined meaning (i.e., the number of the asserted input), what remains to indicate no asserted inputs? Its time to add a new output: Valid.

I0	I1	I2	I3	V	O_1	O_0
0	0	0	0	0	X	X
1				1	0	0
	1			1	0	1
		1		1	1	0
			1	1	1	1



The valid output (V) indicates that an input is asserted and an valid encoded output is available. If no inputs are asserted, the valid signal is low and the outputs are

undefined. When using an encoder, the outputs should only be sampled when V is high. If V is low, no inputs are asserted to encode.

What's your priority?: But what happens when *more than one* input is asserted? In this case, the inputs require a priority scheme so that the highest priority input is encoded into a binary output value. Input priority can be used to expand the five case behavior table (show above) into the full 16 cases that can occur. That assume a simple priority scheme:

I3 > I2 > I1 > I0

Under this scheme, if I3 is asserted, the state of the other inputs is of no concern. I3 will be encoded as output 11. If I3 is zero, but I2 is asserted, the output will reflect this encoding: 10. If I1 is the asserted value being encoded (because I3 and I2 are zero), the output becomes 01. Finally, if only I0 is asserted, the output value is 00.

I0	I1	I2	I3	V	O_1	O_0
0	0	0	0	0	X	X
1	0	0	0	1	0	0
X	1	0	0	1	0	1
X	X	1	0	1	1	0
X	X	X	1	1	1	1

This leads to a implementation by simplifying the output behaviors as Boolean expression. Normally Karnaugh Maps are needed. But this behavior has obvious expressions.

$$V = I0 + I1 + I2 + I3$$

$$O_1 = I3 + I2$$

$$O_0 = I3 + I1 \cdot \overline{I2}$$

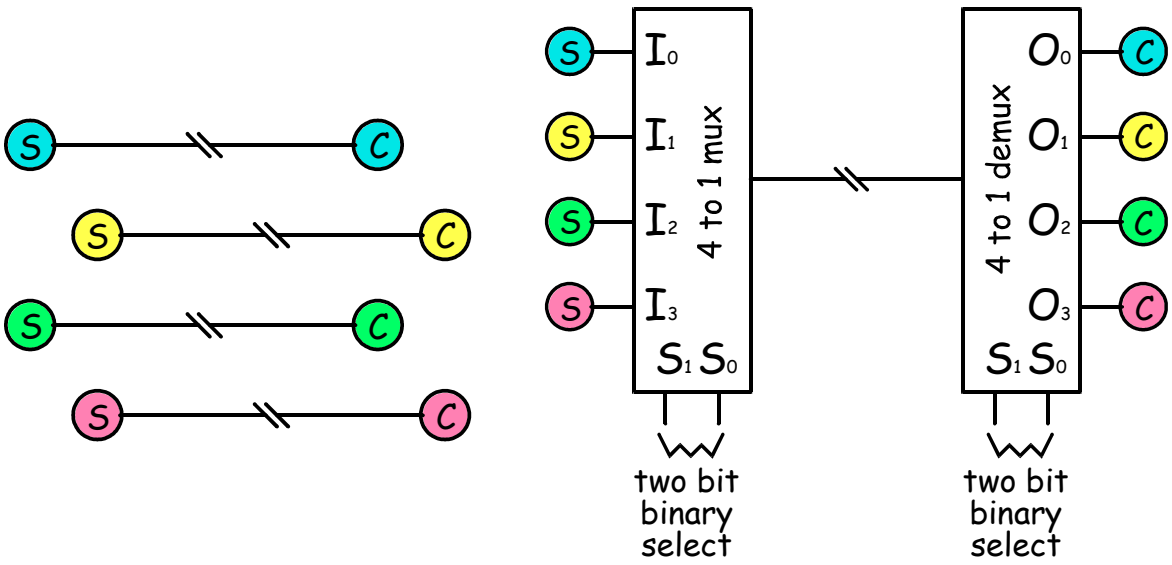
If the input priorities are changed, the Xs and 0s can be easily changed to reflect the new behavior. The rows are processed in a different order. But the same process is applied. In order for a given row to represent the encoded input, all higher priority inputs must be 0 while all lower priority inputs are ignored (don't cared). Here's another example.

I1 > I3 > I0 > I2

I0	I1	I2	I3	V	O_1	O_0
0	0	0	0	0	X	X
1	0	X	0	1	0	0
X	1	X	X	1	0	1
0	0	1	0	1	1	0
X	0	X	1	1	1	1

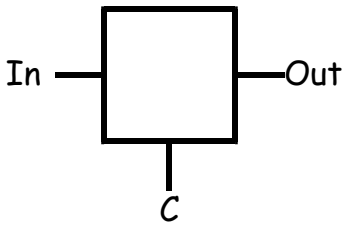
Summary: In general, decoders and encoders transform N-bit binary numbers into assertions of one of M outputs, and back. They change the how the value is represented.

Steering Logic: Sometimes the goal is not to transform data but rather to move it from one place to another. Wire, optical, and wireless channels do a good job of transporting data. But sometimes logic is required to steer data into and out of these channels. For example, we might want to connect multiple sensors that collect information to multiple controllers that process the data. Rather than connecting dedicated wires between each sensor and controller, we can multiplex the data (in time) on a single wire.



There are many uses for a digital block that can steer one of many inputs into an output (a multiplexer). Steering a single input to one on many outputs (a demultplexer) is also valuable. Let's explore their design.

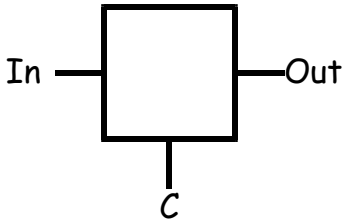
Two familiar gates, seen a new way: Before we start, let's revisit the functions of our more fundamental gates. Imagine a block that can pass or block an input signal depending on a control signal.



In	C	Out
X	0	0
A	1	A

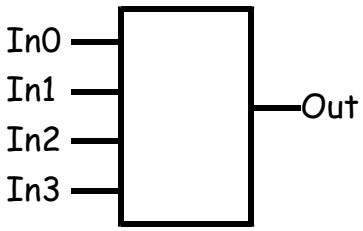
If the control signal is high, the input is passed on to the output. If the control signal is low, the output is masked. In binary, we only have two proper states: 0 and 1. So we'll define "masking" as setting to zero regardless of the input's value.

The implementation of this masking function can be seen by expanding the truth table of its behavior. A *masking gate* is really an AND gate.



In	C	Out
0	0	0
1	0	0
0	1	0
1	1	1

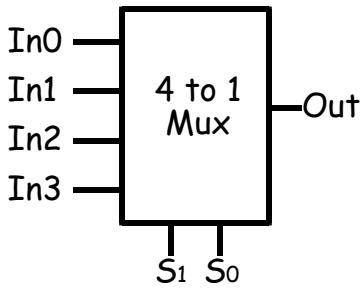
Now imagine a block that can take two or more binary inputs where exactly one of the inputs contains a value A, and all other inputs are zero. A four input version of the function would like like this:



In0	In1	In2	In3	Out
A	0	0	0	A
0	A	0	0	A
0	0	A	0	A
0	0	0	A	A

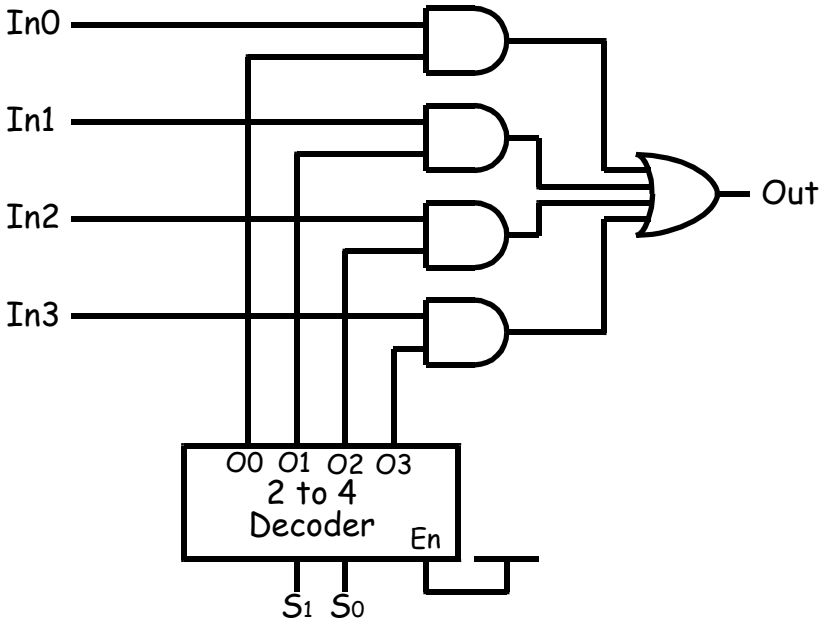
Regardless of which input receives the single, if A is zero, so is the output. If A is one, the output is one. This is the OR function since A + 0 = 0 + A = A. It serves as a *combining gate* for a single value and many zeros.

Multiplexer: A multiplexer or mux steers one of many inputs to the output. The input is selected by a binary number S. For example, a 4 to 1 mux uses a two bit binary number to steer one of four inputs to the output. Here's its behavior.



In0	In1	In2	In3	S_1	S_0	Out
A	B	C	D	0	0	A
A	B	C	D	0	1	B
A	B	C	D	1	0	C
A	B	C	D	1	1	D

Note that the binary number represented by S_1 and S_0 controls which input value is passed through to the output. The behavior can be realized using a 2 to 4 decoder and the masking and combining gates described above.

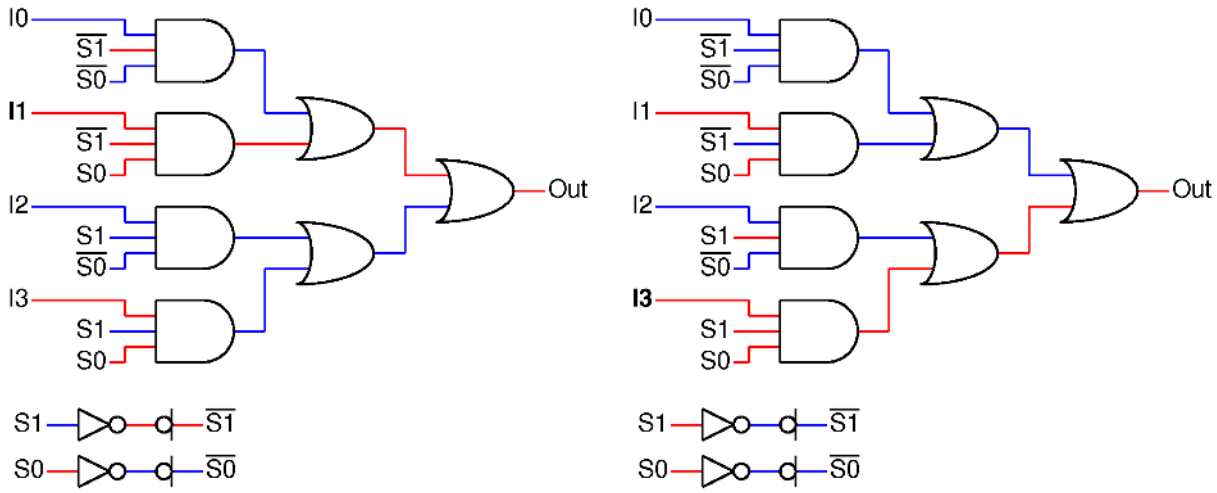


Each input is connected to a masking (AND) gate controlled by the corresponding decoder output. So only the input decoded from the binary input S will be passed through to the combining (OR) gate. All other inputs will be masked to zero. The combining gate ignores zeros, outputting the one passed input.

This behavior can also be expressed as Boolean expressions.

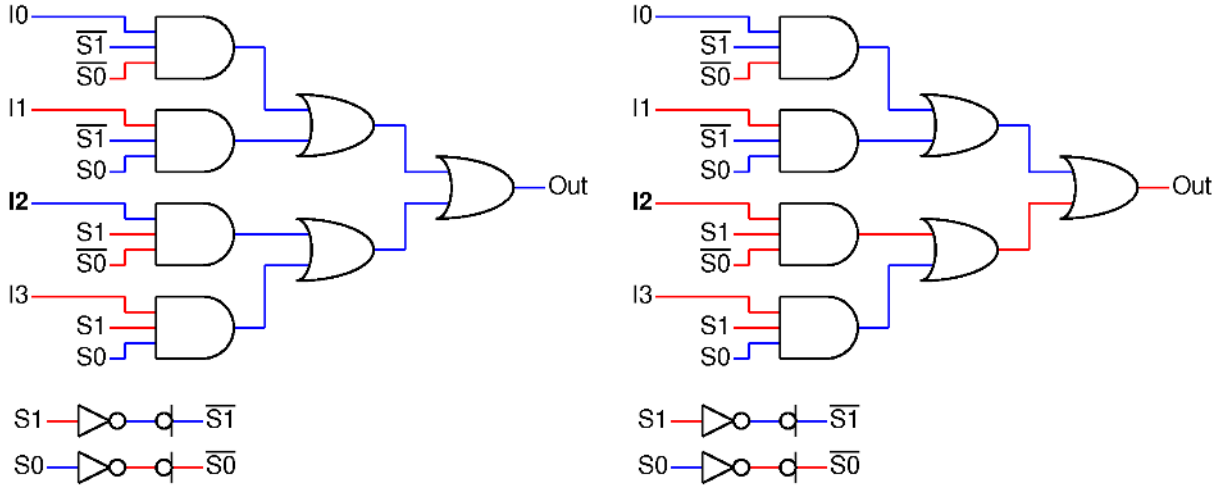
$$Out = In0 \cdot \overline{S_1} \cdot \overline{S_0} + In1 \cdot \overline{S_1} \cdot S_0 + In2 \cdot S_1 \cdot \overline{S_0} + In3 \cdot S_1 \cdot S_0$$

It can be implemented using AND and OR gates as shown below. Note that the four input OR gate can be broken into a combination of two input ORs.

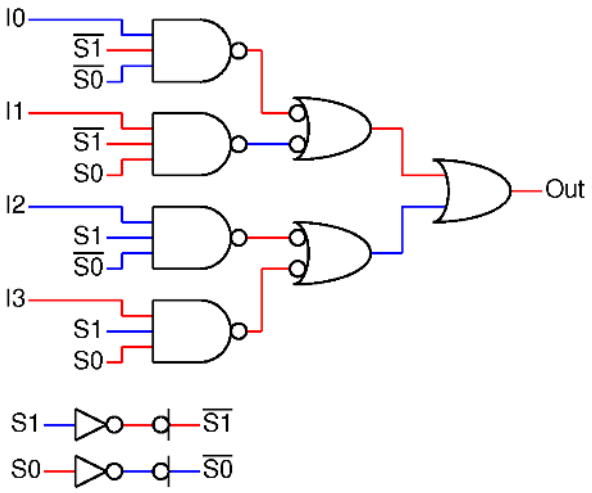


Here S_1 and S_0 contain the value 01 (shown on the left). I1's high value (red) is passed forward while all other masking gates are blocking inputs. The combining OR gates then pass this value forward to the output. If S is changed to the value 11, I3 becomes the signal that is steered to the output (shown on the right).

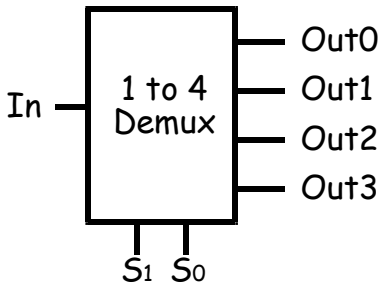
If the selected input happened to have a low value (I2, shown as blue on left), the output would be zero. However if I2 changes to a high value, this change is also be seen at the output (shown on right).



To reduce implementation cost, these AND and OR gates can be transformed to NAND gates using mixed logic. If three and four input gates are available (they are in VLSI), the implementation can be reduced to four 3-input NAND, one 4-input NAND, and two inverters for a total of 24 + 8 + 4 = 26 transistors. because of the inversions, this implementation is less easy to follow an input to the output. But it still works!



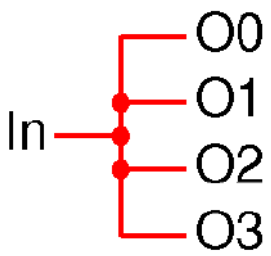
Demultiplexer: So what is the device that performs the reverse operation. A demultiplexer or demux takes a single input and, under control of a binary number, steers it to one of many outputs. The fundamental operation is easy to understand.



In	S_1	S_0	O0	O1	O2	O3
A	0	0	A			
A	0	1		A		
A	1	0			A	
A	1	1				A

However deciding the status of unselected outputs (off-diagonal values) dramatically affects the implementation and use of a demux. Suppose unselected outputs are don't cared. After all, they're not selected. In this case, the implementation can be extremely inexpensive!

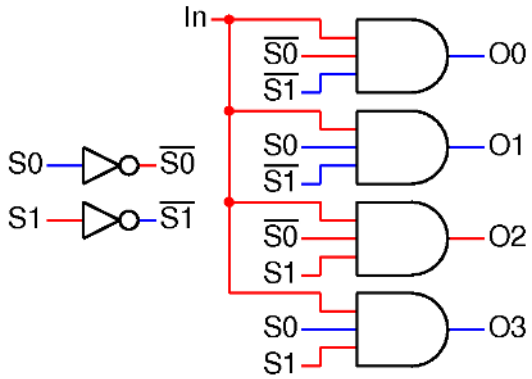
In	S_1	S_0	O0	O1	O2	O3
A	0	0	A	X	X	X
A	0	1	X	A	X	X
A	1	0	X	X	A	X
A	1	1	X	X	X	A



This is *fanout*. It is cheap and useful. Fanout is simply taking a value (in CMOS, a value is a high or low voltage), and connecting it to multiple inputs. Since using an input, or ignoring it does not affect the value, there is no obstacle to fanout (aside from parasitic loading that affects the wire's speed). But it marginally deserves the title "demux".

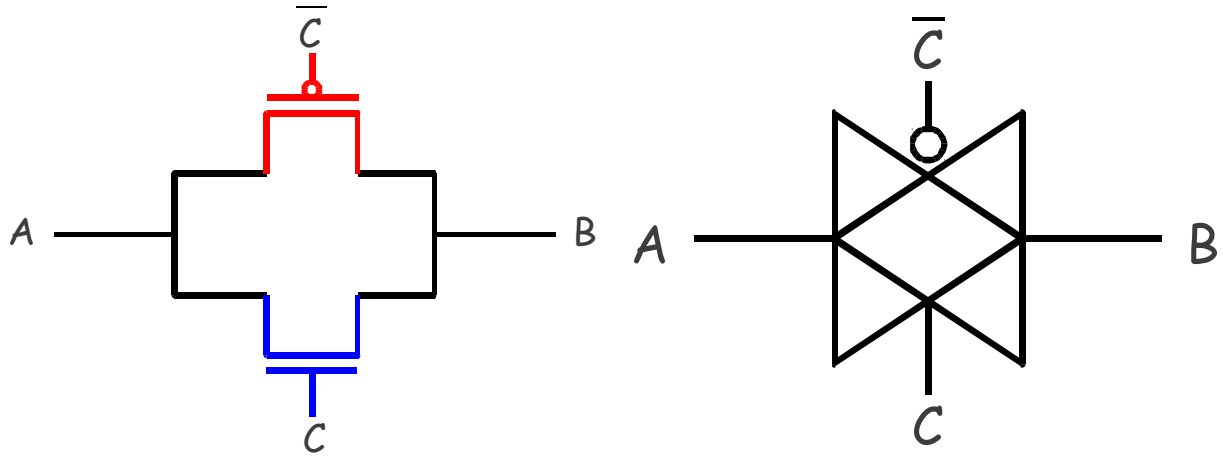
Unselected outputs can also be defined as zero. This could be useful since the input value is "masked" from unselected outputs. This is useful when multiple components produce a signal that must travel on the same wire (not at the same time). Unselected outputs can be combined with a signal (using an OR gate) without affecting the OR gate's output. The OR identity states that X + 0 = X.

In	S_1	S_0	O0	O1	O2	O3
A	0	0	A	0	0	0
A	0	1	0	A	0	0
A	1	0	0	0	A	0
A	1	1	0	0	0	A



Here the implementation is more complex and expensive; it costs 28 transistors for the the one to four demux. This added cost is only required to prevent a unselected output from interfering an OR gate combining many signals. There is a better way...

Pass Gates: Just when you thought you'd seen every gate, another one comes along ... and this one is amazing! So far, all gates have been *regenerative*; they use input signals to control switches that connect the output to either the high or low voltage source. This is a good idea to insure signal integrity. But suppose we just want to *pass* a signal through, or not. This would be far simpler and less expensive. If only there was an ideal switch that could connect a high or low signal. Unfortunately, P and N type switches can only do half the job. Let's use them both to create a *pass gate* (also known as transmission gate or T-Gate).

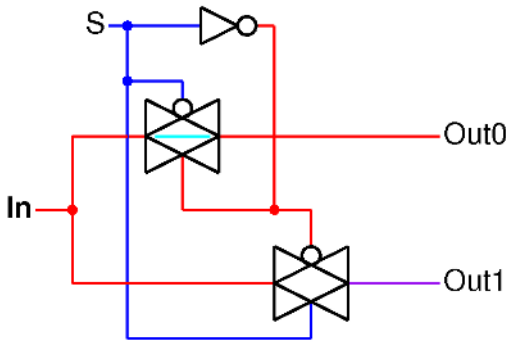


Two switches are connected at their switch points (source and drain) and opened or closed *together*. With this construction, a P-type switch is available to pull high while a N-type can pull low. With both switches closed, the signal level at A can be high or low and the most capable switch is there to connect the signal to B. Sometimes folks think that only one switch needs to be closed, depending on the signal being passed. But by closing both switches together, it really doesn't matter if the signal is high, low, or changing back ands forth!

Interestingly, this switch really doesn't have an input and an output. Instead it has two terminals (A and B) that can be connected together when the control signal C is high. When C is low, the terminals A and B are isolated. This component acts as an "ideal switch" and its extremely useful. It's gate icon looks like two overlapping buffers showing how signals can be passed bidirectionally. The control signals C and $\overline{C}$ arrive that mid point of the two buffers. The bubble indicates the active low control. Look carefully at the pass gate and its icon show side by side above. The icon is on the left; the implementation (using one P-type switch and one N-type switch) is on the right.

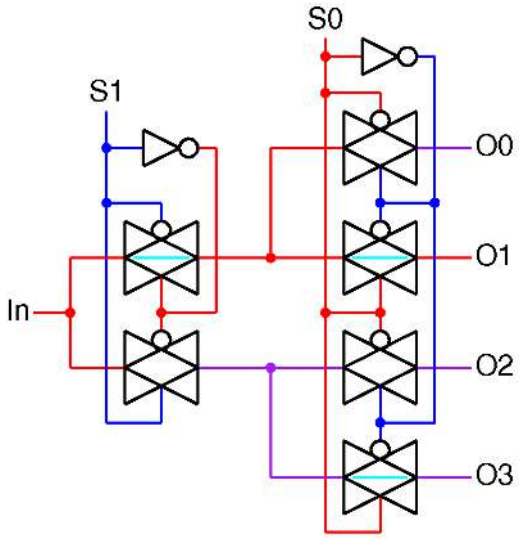
A Demux Using Pass Gates: We can use a couple of pass gates to create a demux with a twist.

In	S	O0	O1
A	0	A	Z_0
A	1	Z_0	A



Here the input value is passed to the selected output via a closed pass gate. The cyan bar on the pass gate indicates the gate is closed; it is not part of the icon. This output is like our previous demux implementations. However the unselected output isn't zero, it isn't connected to anything, Its *floating*. This condition is indicated in the truth table with the somewhat cryptic symbol Z_0 which means high impedance. But this is just a fancy way to say "floating". Because its floating, it can be connected to another signal with no risk of contention (and no OR gate required). If we want more outputs on our demux, we can replicate this demux in a binary tree.

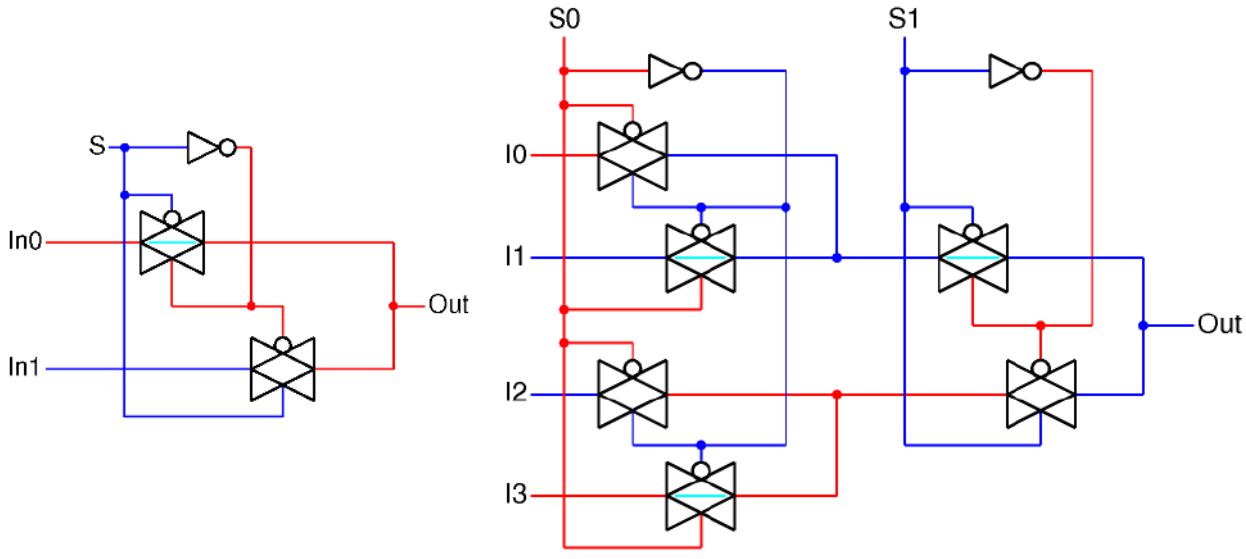
In	S_1	S_0	O0	O1	O2	O3
A	0	0	A	Z_0	Z_0	Z_0
A	0	1	Z_0	A	Z_0	Z_0
A	1	0	Z_0	Z_0	A	Z_0
A	1	1	Z_0	Z_0	Z_0	A



Here S_1 is low so the input travels through the top pass gate. Since S_0 is high, the bottom pass gate of each 2 to 1 demux is closed. But only the uppermost 2 to 1 demux has an input to connect to the output. So all but the selected output, O1, have floating outputs. O1 will follow In.

The function of this implementation is similar to the gate implementation. However, this version has floating unselected outputs (a good thing) and a lower implementation cost: 16 versus 28 switches (an even better thing).

A Mux Using Pass Gates: What's good for demuxes is also good for muxes. Here's a 2 to 1 and a 4 to 1 implemented using pass gates.



These muxes also enjoys a low implementation cost (6 and 16 switches respectively) and they behave exactly as the gate version. These implementations employ binary

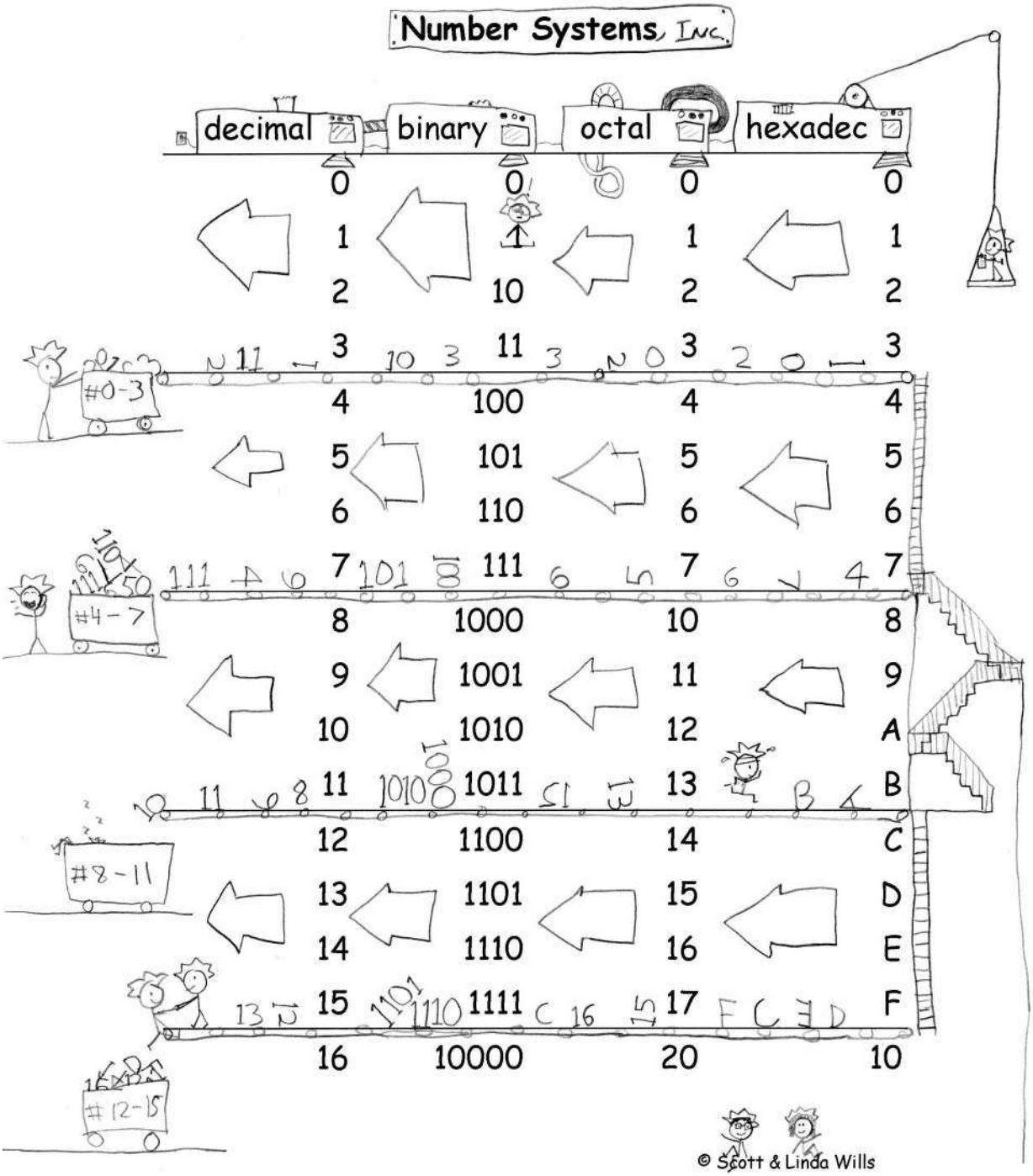
tree construction. To double the number of inputs, just replicate the current mux and then add one more on the end. An 8 to 1 mux would stack two 4 to 1 muxes and then add an extra 2 to 1 mux (controlled by S_2) to choose between their outputs. Using pass gates, muxes are as easy to build as demuxes … hey wait a minute.

Mirror Twins: Regarding the switch contacts, pass gates don't have inputs and outputs like other gates. Either terminal can pull the other. So muxes and demuxs built as binary trees of pass gates are the same thing, but for switching inputs and outputs. In fact they are mirror images of each other.

Summary: Building blocks provide a new abstraction for digital design. What decoders and muxes loss in the generality of gates, they gain in functionality. Its important to remember that, while they have similar appearances, they accomplish different objectives.

- Decoders and encoders perform translation between binary numbers and less compact, but valuable presentations (e.g., selecting one of eight outputs to be high).
- Muxes and demuxes are all about steering signals in and out of shared channels. They can also select a value, or help multiple components share a communications medium. They are controlled by a binary value. But they still just connect an input to an output.
- Pass gates are the ideal switch we wish we had all along. It can pull high and low. But it requires a control signal and its complement. It connect to wires. Or it can leave then floating. An most amazingly, it provide bidirectional connections for flexibility not achieved with other gates.

Designing Computer Systems

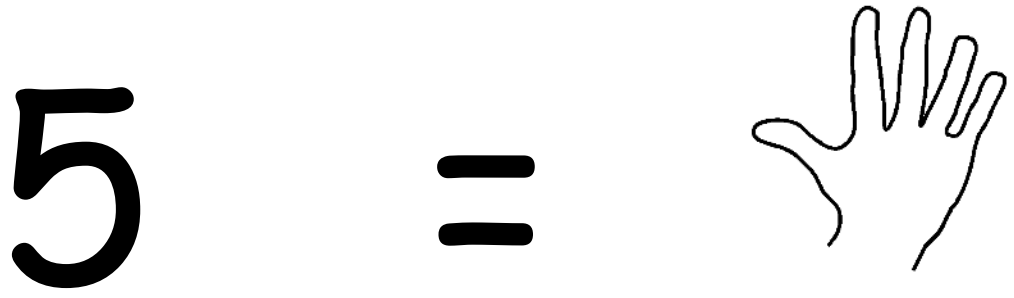


Designing Computer Systems

Number Systems

Most concepts are easier to learn when you're already familiar with them. But a few concepts are more difficult to learn because you know them so well. In our early childhood, we learn that abstract symbols represent real things in our world. The word "candy" represents something that tastes sweet. The word "bedtime" means you're about to leave the party. A symbol and its meaning are locked together in our brain.

This is especially true for qualitative symbols. Here we see the symbol "5" represents the quantity five. In fact, its difficult to describe the symbol with implying its meaning.



symbol meaning

But for computers, a symbol has no implicit meaning. It is a string of ones and zeros. Only when we instruct the computer on how to process a symbol does it have meaning. In many programming languages, you must declare the *type* of a variable, (i.e., an integer, a floating point, or a character string) before you can perform operations on it. This allows the compiler to assign the correct instruction for that interpretation of the variable's value.

Number systems separates a symbol and its meaning into two distinct concepts: a *notation* and a *representation*. Notations determine how symbols can be created using strings of characters from a given alphabet. Representations show how to assign real world meaning to a given string.

Native Notations: Humans around the world favor decimal (base 10) notation. An anthropologist might suggest this is because we have ten fingers. People define ten characters (0,1,2,3,4,5,6,7,8,9) to represent quantities. These characters form a notation *alphabet*. We use this alphabet to create multi-character *strings*, which provide a limitless number of intuitive, unique symbols. In base 10, a N character string can provide 10^N unique strings.

A computer also has a native notation. It uses binary (base 2) notation because the limited multiplicity of its "fingers" maintain digital states: 1 or 0, high or low, true or false. It also builds strings out of its two character alphabet (0 and 1). An N character binary string provides 2^N unique symbols.

Binary, requires longer strings to achieve the same number of symbols. A three character decimal string can represent 1000 symbols (000 - 999). It takes ten character binary string to achieve the same number of strings (0000000000 - 1111111111). To keep the length of written symbols manageable, we often use power of two bases octal (base 8) and hexadecimal (base 16).

The table below shows the ordered sequences in each notation. Notice that each digits counts through the base's alphabet. When a digit reaches the last character, it wraps back to zero and the next digit position is advanced. In all notations, leading zeros are implied, but not drawn.

decimal	binary	octal	hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Notational Conversion: Since all notations begin with zeros, strings on a row in the table are the same sequence number. Since we use the sequences in order, notational conversion of a string in one notation is accomplished by finding the

corresponding position in another notation. For example, the string 11 in decimal is 1011 in binary, 13 in octal, and B in hexadecimal. This conversion takes no position on the meaning of the string. Rather it shows string equivalence.

Since binary, octal, and hexadecimal are all power of two bases, they are more easily translated because they each can be represented as a whole number of binary digits or *bits*. Converting from one power of two notation to another is simply a matter of regrouping the bits. Here are a few examples:

10101110 (binary) = 010 101 110 = 256 (octal) = 1010 1110 = AE (hexadecimal)

153 (octal) = 001 101 011 = 1101011 (binary) = 0110 1011 = 6B (hexadecimal)

68A (hexadecimal) = 0110 1000 1010 = 0110 1000 1010 (binary) = 011 010 001 010 = 3212 (octal)

A conversion between a power of two bases (e.g., binary) and decimal is more complicated. A decimal digit is approximately three and a third bits, so bit regrouping will not work. Notational conversion between binary and decimal is accomplished by finding the string sequence position (how many strings is it from all zeros) and then converting the number between binary and decimal.

In an arbitrary base B, a N character string provides B^N unique symbols. The first digit on the right is the one's place. The second digit is the B's place, the third digit is the (B^2)'s place, the fourth digit is the (B^3)'s place etc. The familiar decimal places are 1s, 10s, 100s, 1000s, … In binary, the places 1s, 2s, 4s, 8s, 16s, … are less familiar, but more useful *powers of two*.

Powers of Two: When you work with computers, you must know the powers of two. Bad news: we have to memorize a few of them. Good news: we don't need to know very many. Here are the ones to learn:

$2^0 = 1$ $2^1 = 2$ $2^2 = 4$ $2^3 = 8$ $2^4 = 16$ $2^5 = 32$

$2^6 = 64$ $2^7 = 128$ $2^8 = 256$ $2^9 = 512$ $2^{10} = 1024 = \sim 1K$

Memorizing can be difficult … but not here. Most folks can compute through 2^4 in your head. 2^6 is 64. The sixes go together. 2^8 is 256. Eight bits is a byte so 256 shows up all the time. 2^5, 2^7, and 2^9 are either twice or half an easy one. And 2^{10} is the vehicle for all other powers of two! It is approximately 1000 (1K).

To find larger powers of two, recall that exponents can be reduced like this:

$$B^{X+Y} = B^X \cdot B^Y$$

We can break larger powers of two into groups in the table above. Exponent multiples of ten can be grouped to become 1000. Here are a few examples.

$2^{16} = 2^6 \times 2^{10} = 64 \times 1K = 64K$

$2^{24} = 2^4 \times 2^{10} \times 2^{10} = 16 \times 1K \times 1K = 16M$

$2^{25} = 2^5 \times 2^{10} \times 2^{10} = 32 \times 1K \times 1K = 32M$

$2^{32} = 2^4 \times 2^{10} \times 2^{10} \times 2^{10} = 4 \times 1K \times 1K \times 1K = 4G$

$2^{41} = 2^1 \times 2^{10} \times 2^{10} \times 2^{10} \times 2^{10} = 2 \times (1K)^4 = 16T$

$2^{-18} = 2^{-8} \times 2^{-10} = 1 / (256 \times 1K) = 1 / 256K$

Binary to Decimal: Using powers of two, binary numbers can be converted using the place values. Here's an example:

64's	32's	16's	8's	4's	2's	1's
1	1	1	1	0	0	1

In a base, the order of a string in a notation is found by summing the products of each character and its respective digit's significance. In binary, the digit values are powers of two. Since characters are either 0 or 1, multiplication is easy. In this example, the corresponding decimal string is computed as:

$$1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1$$

$$64 + 32 + 16 + 8 + 1$$

$$80 + 40 + 1$$

121 (decimal)

Note that many of the powers of two sum to form multiples of ten. Here are a few more examples. The bases are indicated here with subscript.

$110110_2 = 32 + 16 + 4 + 2 = 54_{10}$ $\quad$ $10101010_2 = 128 + 32 + 8 + 2 = 170_{10}$

$100001000_2 = 256 + 8 = 264_{10}$ $\quad$ $1111_2 = 8 + 4 + 2 + 1 = 15_{10}$

A string of ones always sums to next place value minus one.

Decimal to Binary: Notational conversion from decimal to binary is similar. Only here you subtract away powers of two until you reach zero.

78_{10}		165_{10}		500_{10}	
- 64	1000000	- 128	10000000	- 256	100000000
14		37		244	
- 8	+ 1000	- 32	+ 100000	- 128	10000000
6		5		116	
- 4	+ 100	- 4	+ 100	- 64	1000000
2		1		52	
- 2	+ 10	- 1	+ 1	- 32	100000
0	1001110_2	0	10100101_2	20	
				- 16	10000
				4	
				- 4	100
				0	111110100_2

Often there are tricky ways to do things. Sometimes they help. Sometimes they don't. For decimal to binary conversion, one can simply perform a series of halvings (dividing by two). If the number being halved is an even number, list a "0". If the

number being halved is odd, subtract one and list a "1". When you reach zero, the list of ones and zeros is the binary notation. Let's try 78 and 165 this way.

78	~~39~~ 38	~~19~~ 18	~~9~~ 8	4	2	~~1~~ 0	
0	10	110	1110	01110	001110	1001110_2	
~~165~~ 164	82	~~41~~ 40	20	10	~~5~~ 4	2	~~1~~ 0
1	01	101	0101	00101	100101	0100101	10100101_2

This trick works by deconstructing the decimal value from its binary components, from least significant to most significant. It gives the right result; but it sometimes requires more calculations and it is harder to double check the result.

It may appear that integer values are being translated between different bases. But we are only finding corresponding strings in different bases. Notations do not imply meaning.

Get to the Point: Sometimes strings include a point (a decimal point in base 10) as part of the notation. This point divides the string into two parts, a substring to the left of the point and a substring to the right. When performing notation conversion, start at the point and work left and then right. This addresses unwritten leading and trailing zeros. Let's try a few power of two conversion examples.

10100101.011011_2 = 1010 0101 . 0110 1100 = $A5.6C_{16}$

1101001.1111_2 = 001 101 001 . 111 100 = 151.74_8

$26.BC_{16}$ = 0010 0110 . 1011 1100 = 101 110 . 101 111 = 56.57_8

46.26_8 = 100 110 . 010 110 = 0010 0110 . 0101 1000 = 26.58_{16}

Sometimes leading and trailing zero are adding and subtracted to form necessary bit groupings. But notice that they always work out, left and right, from the point. Binary to decimal conversions with a point is the same, only the bit positions are fractions.

4's	2's	1's		1/2's	1/4's
1	0	1	.	1	1

4 + 1 + .5 + .25 = 5.75

8's	4's	2's	1's		1/2's	1/4's	1/8's	1/16's
1	0	1	0	.	0	1	0	1

8 + 2 + .25 + .0625 = 10.3125

8's	4's	2's	1's		1/2's	1/4's	1/8's	1/16's
1	1	0	1	.	1	0	1	1

8 + 4 + 1 + .5 + .125 + .0625 = 13.6875

Representations - Finding Meaning in a Digital World: Although the use of a point has implications to a sequence's value, the focus thus far has been on notational conversion. A given sequence is composed of a specified numbers of characters (N) in a given base (B) offering B^N unique codes. How those codes are used is dependent on *representations*.

Unsigned Integers: A representation begins with a requirement: what needs to be represented. Suppose a digital system is counting objects being manufactured in a factory. The counting numbers (0, 1, 2, …) are needed to maintain a tally. These unsigned integers can be associated with notational sequences in an intuitive way.

sequence	meaning
0000	"0"
0001	"1"
0010	"2"
0011	"3"
0100	"4"
0101	"5"
0110	"6"
0111	"7"

sequence	meaning
1000	"8"
1001	"9"
1010	"10"
1011	"11"
1100	"12"
1101	"13"
1110	"14"
1111	"15"

Perhaps this is too intuitive, since this looks like notational conversion from binary to decimal. But here the quoted value really does mean a quantity (remember the fingers). A four bit binary sequence is used to represent a quantity between "0" and "15". In general, when representing unsigned integers, an N-bit binary sequence can represent quantities between "0" and "2^N -1". So an eight bit unsigned integer can represent quantities between "0" and "255"; a 16 bit unsigned integer can represent "0" to "65,535" (around 64K), and a 32 bit unsigned integer can represent "0" to "4 billion". This process is nothing more than a uniform value sequence assignment. An integer value is assigned to each sequence.

Signed Integers: Some applications require negative as well as positive integers. While it doesn't have to be this way, a signed representation typically offers an equal number of positive and negative quantities.

signed	sequence	unsigned
"0"	0000	"0"
"1"	0001	"1"
"2"	0010	"2"
"3"	0011	"3"
"4"	0100	"4"
"5"	0101	"5"
"6"	0110	"6"
"7"	0111	"7"

signed	sequence	unsigned
"-8"	1000	"8"
"-7"	1001	"9"
"-6"	1010	"10"
"-5"	1011	"11"
"-4"	1100	"12"
"-3"	1101	"13"
"-2"	1110	"14"
"-1"	1111	"15"

Since half of the sequences are used to represent negative values, there are not as many to represent positive quantities. Here the 16 sequences represent "-8" to "+7". In general, this N-bit signed integer representation can represent quantities from "$-2^{(N-1)}$" to "$2^{(N-1)} - 1$". A eight bit signed integer can represent "-128" to "+127". A 16-bit signed integer can represent "-32,678" to "+32,767" (±32K). A 32-bit signed integer can represent "±2 billion" (±32G). Why isn't it symmetric? Because zero has to go somewhere (and use a sequence). Here it is counted as a positive value. This signed representation is called *two's complement*.

There are many choices for signed representations. But only one, two's complement is widely used, and for good reasons. As number systems and arithmetic are explored, two's complement has many significant advantages other other signed representations.

- Sign and Magnitude: This signed representation (used in floating point) employs all but one bits for an unsigned magnitude. The remaining bit indicates the sign. It problems include complex arithmetic logic (since addition sometimes becomes subtraction and vice versus) and two representations of zero (+0 and -0). This may seem like a small matter. But comparison to zero is the most commonly performed conditional operation. If there are two values representing zero, this operation become more complex.
- One's Complement: This signed representation has a simple negation: complement each bit. So +1 (0001) is negated to -1 (1110). This representation also introduces complexity is arithmetic. And it has two values for 0 (0000) and (1111).

Two's complement is related to one's complement. Negation involves complementing each bit in the representation. But then one is added: one's complement + one = two's complement. It only has one representation of zero (negating zero give zero). Sign is easy to determine; the most significant bit of the representation indicates the sign (0 = positive, 1 = negative). But it is not a sign bit. And arithmetic using two's complement couldn't be easier (one can ignore sign). Two's complement also works well with non-integer representations, which come next.

Fixed Point: Integer representations have a *fixed step size*, the value one. All adjacent sequences differ by the integer value one. This is its *resolution* and it is fixed. This step size can assume any value, depending on the position of the point (which separates whole and fractional parts of the representation). So if the point is fixed one bit position to the left of integers, the step becomes 0.5 instead of one. This four-bit, fixed point representation offers a different set of values.

signed	sequence	unsigned
"0.0"	000.0	"0.0"
"0.5"	000.1	"0.5"
"1.0"	001.0	"1.0"
"1.5"	001.1	"1.5"
"2.0"	010.0	"2.0"
"2.5"	010.1	"2.5"
"3.0"	011.0	"3.0"
"3.5"	011.1	"3.5"

signed	sequence	unsigned
"-4.0"	100.0	"4.0"
"-3.5"	100.1	"4.5"
"-3.0"	101.0	"5.0"
"-2.5"	101.1	"5.5"
"-2.0"	110.0	"6.0"
"-1.5"	110.1	"6.5"
"-1.0"	111.0	"7.0"
"-0.5"	111.1	"7.5"

For both unsigned and signed representations, there are the same number of sequences. With a smaller resolution (0.5 versus 1), the representation has a smaller range. In general, an N bit fixed point representation with K bits to the right of the binary point has a step size of $1/2^K$ and a range of $-2^{(N-1)}/2^K$ to $(2^{(N-1)}-1)/2^K$. The range is *divided* by the step size.

If the fixed point is set two bits from the left, the step size and range change. A smaller step, 0.25, yields higher resolution, but a smaller range.

signed	sequence	unsigned
"0.0"	00.00	"0.0"
"0.25"	00.01	"0.25"
"0.5"	00.10	"0.5"
"0.75"	00.11	"0.75"
"1.0"	01.00	"1.0"
"1.25"	01.01	"1.25"
"1.5"	01.10	"1.5"
"1.75"	01.11	"1.75"

signed	sequence	unsigned
"-2.0"	10.00	"2.0"
"-1.75"	10.01	"2.25"
"-1.5"	10.10	"2.5"
"-1.25"	10.11	"2.75"
"-1.0"	11.00	"3.0"
"-0.75"	11.01	"3.25"
"-0.5"	11.10	"3.5"
"-0.25"	11.11	"3.75"

Fixed point does not require a change to the arithmetic. It is only a matter of interpretation of the operands and the result. Fixed point is the presentation of choice for the financial world. All calculations must be accurate to the penny, regardless of the amount. This fixed resolution limits the range. Science and engineering often need something else.

Floating Point: Fixed point presentations have a problem in that their accuracy (the number of significant figures) is dependent on the magnitude of the represented value. The integer value 23,415,823 may have eight significant figures. But 16 has only two. Floating point has a different, more complex approach. Use a certain number of bits to represent the magnitude (the significant figures) of a value. Then use addition bits to scale it to the correct value. Most people have used this approach in scientific notation. The magnitude 6.022 is scaled by 10^{23} to express the number of molecules in a mole. This value would be difficult to express using a fixed point representation.

Floating point breaks the bits of the representation into fields: sign, mantissa, and exponent.

sign	mantissa	exponent

The sign field is a one bit field indicating the sign of the mantissa. This sign and magnitude representation makes sense when scaling the value. The mantissa is the largest field and contains the bits that provide the accuracy (significant figures) to the value being represented. Since the mantissa does not need to provide the scaling, its range is between zero and one. The exponent field is a signed integer that scales the mantissa to the proper value. In binary, the exponent is raised to a power of two, not ten. In general, a floating point value is computed as:

$$\text{sign} \times \text{mantissa} \times 2^{\text{exponent}}$$

where the sign is ±1, the mantissa is an unsigned fixed point value with the binary point at the right end of the sequence (K = N), and the exponent is a signed integer. Typical field lengths for an IEEE single precision floating point value is sign = one bit, mantissa = 23 bits, and exponent = 8 bits. This means that the unscaled step size is 1/8M of the mantissa. To find the equivalent decimal significant figures, consider the mantissa range (0 to 8,000,000). The first six digits can assume any value (0-9). The seventh decimal digit can assume 0-8. So this mantissa maintains between six and seven decimal significant figures. In general, every ten bits of mantissa provides three decimal significant figures.

The exponent field is a signed (two's complement) integer. Like scientific notation, it scales the mantissa to the proper value. It doesn't change the bits, rather it moves the binary point. Moving it right by one bit multiples the value by two. Moving right two bits multiples by four. Moving right by I bits multiples by 2^I. Moving left is similar, except it divides by 2^I. Because of this exponential scaling, a modest range in the exponent field can have an enormous effect on the value. An eight bit exponent has a range of -128 to +127. Since the mantissa is between zero and one, the final value an be as large as 2^{127} or as minuscule as $1/2^{128}$.

Floating points representations can assume smaller and larger number of bits. IEEE double precision floating point employs 64 bits including an eleven bit exponent and a 52 bit mantissa for approximately 15 significant figures. A 16 bit floating points might have a 10 bit mantissa (three significant figures) and a five bit exponent for values from 2^{15} (32K) to $1/2^{16}$ (1/64K).

Arithmetic operations in floating are more complicated since exponents must be adjusted before simple addition and subtraction can be performed in the mantissa. Afterwards, a process called *normalization* must be performed where the mantissa

and exponent are adjusted to keep a one in the most significant bit of the mantissa. This is necessary to maintain the full accuracy of the value.

In floating point, all values have a fixed accuracy (significant figures), but a varying resolution (step size). This contrasts with fixed point that has a fixed resolution, and a varying accuracy. Fixed point works for financial calculations. Floating point works for science and engineering. Both are important.

Full Disclosure: Floating point standards have many subtle complexities that are not covered here. For example, since normalization maintains a one on the most significant bit of the mantissa, it can be assume to effectively *add* a bit. Other field combinations are used for rare but important values like NaN (Not a Number). If interested, check out http://grouper.ieee.org/groups/754/.

Symbolic Values: Speaking of not a number, there is a large class of representation that don't represent quantities. Take this document, for example. Each character represents a letter of the alphabet, and sequences are strings of letters forming words, sentences, and paragraphs. One of the oldest and most common symbolic representation is ASCII (American Standard Code for Information Interchange). This seven bit representation includes the characters that appear on a keyboard: A-Z, 0-9, a-z, characters for punctuation, special symbols, etc. Plus some obsolete control characters like bell, ACK/NAK, etc. that date back to an era when mechanical teletypes were used to display text. This standard was latter expanded to eight bits (256 symbols) for CP/M, MS-DOS, etc. but it still lives on.

One limitation of ASCII is its inability to expand to international character sets. A modern alternative is Unicode, a 16-bit character code that embraces the diversity of symbols from around the world. While its larger 16 bits versus eight bits, its ability to international character sets justifies the extra storage. Still, ASCII is far from gone. It still is the primary representation used in text files under today's operating systems including Microsoft Windows, Mac OS X, and Linux.

Other Representations: There are hundreds of other representations to represent images (e.g., JPEG), videos (e.g., XviD), audio (e.g., mp3), vector graphics (e.g., postscript), and many other things. However the notations used generate the same patterns of sequences.

Summary: In digital computers, information is expressed in one of several notations, and its meaning is defined by one of many representations.

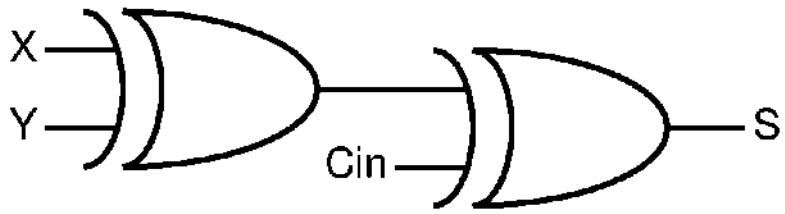
- Today's notations include binary, decimal, and hexadecimal. Powers of two fit the binary technology being used. Decimal fits ten fingered humans.

- Quantitative representations include signed and unsigned fixed point representations integers is a special case). For signed representations, two's complement is the representation of choice. Fixed point has a fixed step size (resolution), but varying accuracy. Floating point is a more complex representation with fixed accuracy, but a varying step size. Both representations have their place in digital systems.
- Symbolic representations are widely used in digital systems. ASCII is an old but widely used standard. Unicode allow representation of international characters.

ASCII Codes								
	0x00	0x10	0x20	0x30	0x40	0x50	0x60	0x70
0x0	NUL	DLE	SP	0	@	P	`	p
0x1	SOH	DC1	!	1	A	Q	a	q
0x2	STX	DC2	"	2	B	R	b	r
0x3	ETX	DC3	#	3	C	S	c	s
0x4	EOT	DC4	$	4	D	T	d	t
0x5	ENQ	NAK	%	5	E	U	e	u
0x6	ACK	SYN	&	6	F	V	f	v
0x7	BEL	ETB	'	7	G	W	g	w
0x8	BS	CAN	(	8	H	X	h	x
0x9	HT	EM	)	9	I	Y	i	y
0xA	LF	SUB	*	:	J	Z	j	z
0xB	VT	ESC	+	'	K	[	k	{
0xC	FF	FS	,	<	L	\	l	|
0xD	CR	GS	-	=	M	]	m	}
0xE	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

Arithmetic

```
  1   1
      1   1   0
+     0   1   1
-------------------
  1   0   0   1
```



Designing Computer Systems

Arithmetic

Arithmetic in digital systems involves a few familiar operations (addition, subtraction, multiplication, and division) on quantitative representations described in Number Systems. Everyone knows what the answer should be. The challenge is making hardware that performs these operations, and detecting when the representation cannot capture the result (overflow). This chapter combines Number Systems and Gate Design to define and implement addition and subtraction.

Addition: Addition is a simple *dyadic* operation in that operates on two operands, the addends, to produce a result, a sum. The rules of addition were first learned in an early grade on decimal values.

		1	1 1	1 1
6 8 3	6 8 3	6 8 3	6 8 3	6 8 3
+ 3 6 5	+ 3 6 5	+ 3 6 5	+ 3 6 5	+ 3 6 5
	8	4 8	0 4 8	1 0 4 8

When adding these values, one starts at the least significant digit. Adding three and five is easy. The result, eight is expressible within the significance of this digit (the one's place). So this place is done. The next digit, in the ten's place is more complicated. The sum of eight (80) and six (60) is 14 (140). But this cannot be expressed fully in the tens place. So the six (60) is recorded and the ten (100) is carried to the next place, the hundreds place. The sum of six (600), three (300), and the carried in one (100) sums to one thousand. Again, this cannot be captured in the hundred's place. So it is carried to the next digit, the thousand's place. This leads to a habit that humans have, but digital systems cannot support: The presumption of unlimited bit resolution. Humans assume that if there is space in the result line, it can be used to fully, and accurately express the result. Unfortunately digital systems must live within the available bits in the representation. If the representation if three decimal digits, 000 to 999, it cannot represent 1048. So there is an *overflow* error.

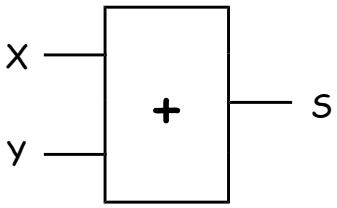
Moving to binary addition is simply a matter of employing a binary notation.

		1	1 1	1 1
1 1 0	1 1 0	1 1 0	1 1 0	1 1 0
+ 0 1 1	+ 0 1 1	+ 0 1 1	+ 0 1 1	+ 0 1 1
	1	0 1	0 0 1	1 0 0 1

Zero added to one results in one. There are fewer characters in the alphabet so this happens less frequently. Note also that places here are not powers of ten (1, 10, 100) but are instead powers of two (1, 2, 4, 8). When one (2) is added to one (2), the result, 10 (4) cannot be represented in the two's place. So zero is recorded and 10 (4) is carried to the next bit position, the 4's place. This 10 (4) is added to the one (4) and zero already in this place to produce a result 10 (8). The zero remains in the 4's place, and the 10 (8) is carried to the 8's place.

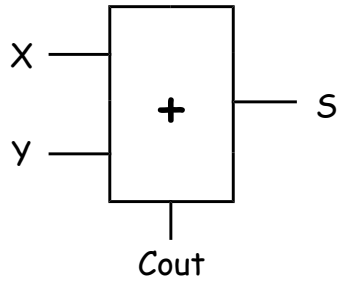
But as before, digital systems may not always have an extra bit position (here in the 8's place) to hold the one carried out of the 4's place. If it's available, 110 (6) added to 011 (3) results in 1001 (9). Otherwise the result is 001 (1), which is incorrect, at least for unsigned integers. Interestingly, in a three bit two's complement representation, 110 (-2) added to 011 (3) *is* 001 (1)! So overflow errors are dependent on the representation. More on this later.

Addition Hardware: The goal is to build hardware to perform arithmetic. It is important to note that the operation of binary addition is invariant to bit position. It's the same operation with respect to given inputs used to compute outputs regardless of whether it is performed in the 1's place, 2's place, etc. So building a one-bit adder is the place to start. Here's an adder for two one-bit binary operands, X and Y. The result is S.



X	Y	S
0	0	0
1	0	1
0	1	1
1	1	10

Again, the rules of addition are applied in binary. 0 + 0 = 0, 1 + 0 = 1, 0 + 1 = 1, 1 + 1 = 10. Only this is a *one-bit* adder. So 10 (2) cannot be represented in this place. An additional output is added to carry this value to the next bit position. This is called Carry Out (Cout). When the sum of X and Y exceeds one, this output signals that the bit position capacity exceeded, and a carry out takes excess output to the next bit position. When X and Y are one, the output is two and carry out transfers this excess. For all other addition cases, the sum can be represented in this bit position so carry out is zero.

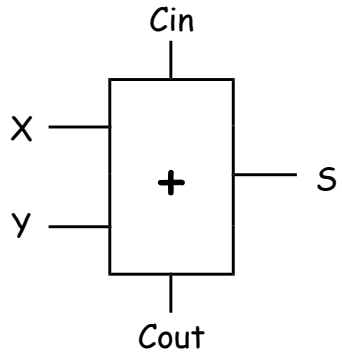


X	Y	Cout	S
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

This output is drawn on the bottom rather than the left side of the adder icon because it is used as an input for another one-bit adder. Since the next bit position must process this carry out signal, the adder also needs another input.

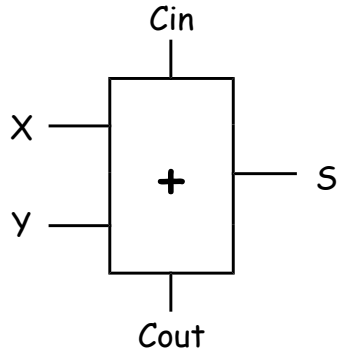
Two for You is One for Me: When a two in one bit position is carry into the next, what is it worth? Moving from least significant to most significant bit positions, each bit is twice the significance of the bit before it. So the two's place is twice the significance of the one's place. The four's place is twice the two's place. The eight's place is twice the four's place. In general, the i+1's place is twice the significance of i's place. So when two is carried out of the i's place, it becomes one in the i+1's place. So a carry out (two) from the lesser significant neighboring bit becomes one as a carry in.

This simplifies the behavior when a *Carry In* (Cin) is added. Now, rather than adding X and Y, the adder is adding X + Y + Cin. All three inputs have the same significance. Here's the behavior assuming the carry in is zero (from before).



X	Y	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1		
1	0	1		
0	1	1		
1	1	1		

To handle the new cases where carry in is one, the result includes the sum of X, Y, and Cin.



X	Y	Cin	Cout	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

The first case (X = 0, Y = 0, Cin = 1) produces a sum of one. No carry out required. When (X = 1, Y = 0, Cin = 1), the sum is 10 (2). This results in Sum = 0 and Cout = 1. The same outputs occur when (X = 1, Y = 0, Cin = 1). But when (X = 1, Y = 1, Cin = 1), the sum is 11 (3). Carry out moves 10 (2) to the next bit position. The remaining one becomes the Sum output. So the results are Sum = 1 and Cout = 1.

This one-bit adder, also known as *a Full Adder*, captures the behavior of binary addition. Multi-bit addition can be constructed from cascaded one-bit adders.

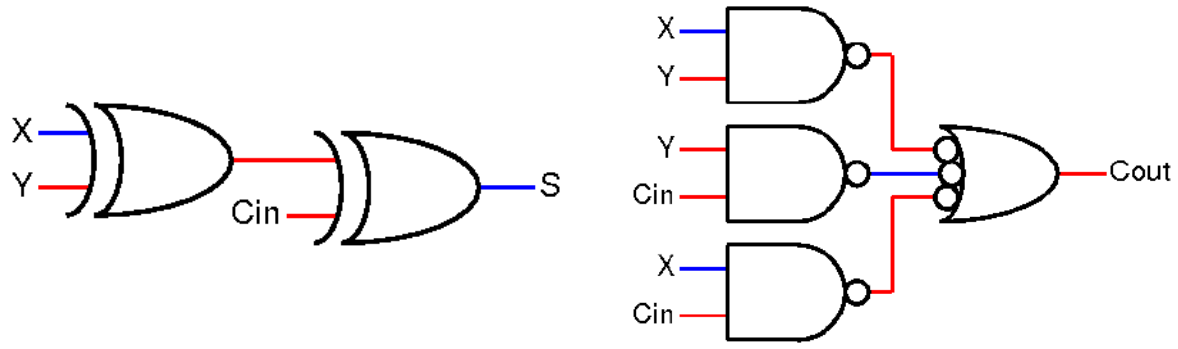
What's Inside a Full Adder?: The behavior of Sum (S) is expressed as odd parity (XOR) which is defined as "true when the number of high inputs is odd". As X, Y, or Cin transitions between zero and one, the number of high inputs changes from even to odd, or odd to even. Independent of the carry out signal, a transition of an input (X, Y, or Cin) results in a transition of the resulting sum, S. Carry out, Cout, has an equally intuitive definition. If there is a one on less than two inputs (X, Y, and Cin), the resulting sum can be handled within the bit position. However if two or more inputs are one, the sum will exceed the bit position's maximum value and carry out must be asserted. This happens in four cases:

$$X \cdot Y \qquad Y \cdot C_{in} \qquad X \cdot C_{in} \qquad X \cdot Y \cdot C_{in}$$

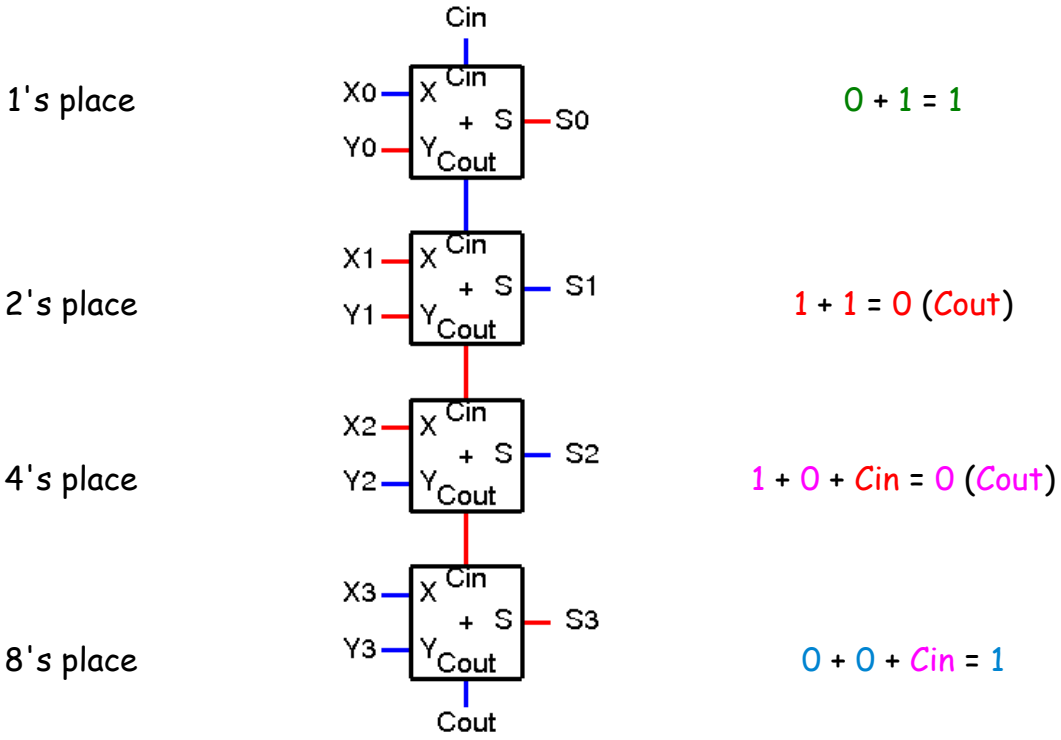
The expression for carry out can be simplified to a sum of three product terms. The behavior of a full adder can be expressed with these two boolean expressions.

$$S = X \oplus Y \oplus C_{in} \qquad C_{out} = X \cdot Y + X \cdot C_{in} + Y \cdot C_{in}$$

The implementation is straightforward. Odd parity (a checkerboard K-map) is difficult to simplify. The sum of products expression is implemented with NAND gates.



Going Multi-Bit: Since each bit position follows the same operations, a multi-bit adder is created by connecting several replicated one bit adders. The carry in of the least significant bit is set to zero. If it is one, an extra one is added to the sum. *This may come in handy later*. Each bit position is implicit relative to its position. Here's a four bit adder:



In this example X = 0110 (6) and Y = 0011 (3), Each one bit adder handles one bit position. The collection performs the word addition yielding the correct solution 1001 (9) … correct assuming an unsigned integer representation. But what about for other representations?

Fixed Point Arithmetic: A full adder works predictably for unsigned integers. It also supports unsigned fixed point representations, since the bit position

relationship does not change. The carry in comes from a bit position that has half the significance. The carry out goes to a bit position that has twice the significance. All that is required is a zero in the carry in to the least significant bit. For all unsigned representations, a carry out of the most significant bit in the representation indicates a result beyond the maximum expressible value. Otherwise the value is correct. If a fixed point is added in the middle of a four bit adder, its operation is unchanged. All that changes is the interpretation of the operands and the result. The operation adds 01.10 (1.5) and 00.11 (0.75) to produce 10.01 (2.25). Here are more examples.

integer		fixed point	
7 + 1 = 8	0111 + 0001 = 1000	01.11 + 00.01 = 10.00	1.75 + 0.25 = 2.0
9 + 6 = 15	1001 + 0110 = 1111	10.01 + 01.10 = 11.11	2.25 + 1.5 = 3.75
4 + 12 = 0 overflow	0100 + 1100 = 0000	01.00 + 11.00 = 00.00	1.0 + 3.0 = 0.0 overflow

In each of these examples, the same operand patterns are applied to the four bit adder, producing the same result. Only the representation differs. Note that for fixed point representations, overflow errors occur (or not) with the same operands, regardless of the position of the point. Fixed point is a scaling of the operands and the corresponding ranges of the representation.

Signed Arithmetic: In Number Systems, there were many advantages to two's complement representations for signed quantities: only one representation of zero, a simple negation, easy differentiation of positive and negative values. Here's one more reason to like two's complement: it employs the same rules for arithmetic. So the examples can be reconsidered as two's complement.

unsigned integer		two's complement integer	
7 + 1 = 8	0111 + 0001 = 1000	0111 + 0001 = 1000	7 + 1 = -8 overflow
9 + 6 = 15	1001 + 0110 = 1111	1001 + 0110 = 1111	-7 + 6 = -1
4 + 12 = 0 overflow	0100 + 1100 = 0000	0100 + 1100 = 0000	4 + -4 = 0.0

In both representations, the same four bit adder performs the same logical operations producing the correct result when the answer is within the representation's range. Different representations give different meanings to the operand sequences. And different ranges produce overflow errors in different places. The two's complement interpretation of the first example 7 + 1 = -8 results from the range of a four-bit two's complement integer: -8 to +7. This is not an

error for the unsigned integer representation with a range of 0 to 15. In the third example, the result in the unsigned integer representation overflows its range. The two's complement representation does not.

The four bit adder also performs properly for signed two's complement fixed point. Since this is just scaling of the values (and ranges), this is expected.

unsigned fixed point		two's complement fixed point	
01.11 + 00.01 = 10.00	1.75 + 0.25 = 2.0	01.11 + 00.01 = 10.00	1.75 + 0.25 = -4.0 overflow
10.01 + 01.10 = 11.11	2.25 + 1.5 = 3.75	10.01 + 01.10 = 11.11	-1.75 + 1.5 = -0.25
01.00 + 11.00 = 00.00	1.0 + 3.0 = 0.0 overflow	01.00 + 11.00 = 00.00	1.0 + -1.0 = 0.0

The carry out of the most significant bit indicates overflow with unsigned representations. It tells nothing about overflow in two's complement representations. So how can these errors be detected?

Overflow in Two's Complement: There are several ways to detect two's complement overflows. The most intuitive exploits the easy sign detection of the representation using only the most significant bit. If the MSB is zero, the value is positive. If the MSB is one, it is negative. When an overflow occurs, the inexpressible *wraps around* the range of the representation and ends up in the opposite signed values. In the first example, two positive numbers are added to produce a negative result. Because overflows wrap into the opposite sign, they can be detected using only the most significant bits of the operands and the result. Here are the eight cases when positive and negative values are added.

0 0 1 0	0 0 1 0	0 1 1 0	0 0 1 0	1 0 1 0	1 0 1 0	1 0 0 1	1 1 0 1
+ 0 0 1 1	+ 0 1 1 0	+ 1 1 0 0	+ 1 1 0 0	+ 0 1 1 0	+ 0 1 0 1	+ 1 0 1 0	+ 1 0 1 1
0 1 0 1	1 0 0 0	0 0 1 0	1 1 1 0	0 0 0 0	1 1 1 1	0 0 1 1	1 0 0 0
ok	overflow	ok	ok	ok	ok	overflow	ok

Overflows occur in two cases: when two positive values are added with a negative result, and when two negative values are added with a positive result. Here are the corresponding decimal values:

2	2	6	2	-6	-6	-7	-3
+ 3	+ 6	+ -4	+ -4	+ 6	+ 5	+ -6	+ -5
5	-8	2	-2	0	-1	3	-8
ok	overflow	ok	ok	ok	ok	overflow	ok

Two's complements overflows for addition occur when two positives produce a negative sum, or two negatives produce a positive sum. This can be expressed as a boolean expression, where the "m" subscript indicates the most significant bit:

$$Overflow = X_m \cdot Y_m \cdot \overline{S_m} + \overline{X_m} \cdot \overline{Y_m} \cdot S_m \quad .$$

The same multi-bit adder can be used for unsigned and signed two's complement representations. But different hardware is required to detect overflows. For unsigned addition, the carry out of the most significant bit indicates an overflow. For signed two's complement, hardware that implements this sum of products is needed. More of this hardware will follow an exploration of the next arithmetic operation, *subtraction*.

Giving and Taking Away: Subtraction is the converse of addition. But in addition, the early-learned rules of elementary school are directly applied, in subtraction, a more modern approach to borrowing is employed. Here's an example.

		1	1 1	1 1
4058	4058	4058	4058	4058
-2373	-2373	-2373	-2373	-2373
	5	85	685	1685

Like addition, one starts at the least significant bit. Here, 3 is subtracted from 8 leaving 5. But in the next digit, the ten's place, 7 is subtracted from 5. Like addition, sometimes the operation cannot be completed within the digit. Early on, one learns to search for the next non-zero digit, borrowing one in its place, and regrouping as needed. Here one would borrow from the 4 (4000). But this approach is expensive, since it requires a search though an indeterminate number of digits. A more efficient approach doesn't look for the needed value to borrow; it presumes it exists, passes a borrow signal to the next digit, and completes the operation. This closely resembles the modern approach to borrowing: one places a purchase on a credit card, and decides later whether needed funds are available. While this is an unsound fiscal policy, it works well in computer arithmetic.

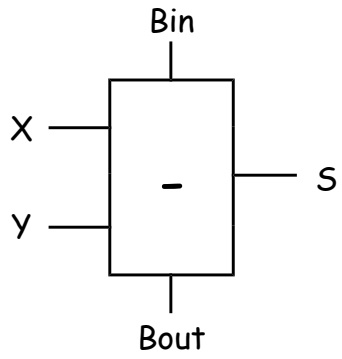
In this example, the process assumes the additional 10 (100) is available to regroup the 5 (50) as 15 (150). Then the 7 (70) is subtracted leaving 8 (80). A borrow out signal is passed to the next digit, the hundred's place. Here, 3 (300) is subtracted from 0. The the borrow in does not add, it subtracts as a 1 (100) in this digit. Again, the total of 4 (400) that is subtracted. So borrow out is asserted to the thousand's place. This provides 10 (1000) that the 4 (400) can be subtracted from. The operation in the thousand's place concludes with 2 (2000) being subtracted along with the borrow value 1 (1000) from the 4 (4000), leaving 1 (1000) remaining.

Binary subtraction follows this decimal example just as binary addition did.

		1	1 1	1 1
1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1	1 0 0 1
- 0 1 1 0	- 0 1 1 0	- 0 1 1 0	- 0 1 1 0	- 0 1 1 0
	1	1 1	0 1 1	0 0 1 1

In this example, barrow out is asserted when the operation cannot be completed with the bit position.

Subtraction Hardware: Subtraction logic resembles a full adder. Only it computes the difference, and it accepts borrow in and generate borrow out. In subtraction, the difference output (D) is computed as X = Y – Bin since both Y and Bin have the same significance. Here's the behavior of a full subtractor.

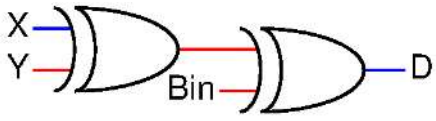


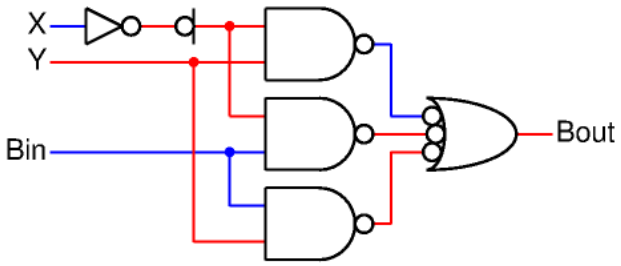
X	Y	Bin	Bout	D
0	0	0	0	0
1	0	0	0	1
0	1	0	1	1
1	1	0	0	0
0	0	1	1	1
1	0	1	0	0
0	1	1	1	0
1	1	1	1	1

It is interesting to note that difference (D) is exactly the same as sum (S) in addition: odd parity. This makes sense when one considers that, from a single bit position's standpoint, adding one has the same effect as subtracting one. The result toggles. The borrow out expression differs from carry out.

$$D = X \oplus Y \oplus B_{in} \qquad B_{out} = \overline{X} \cdot Y + \overline{X} \cdot B_{in} + Y \cdot B_{in}$$

The hardware implementation resembles the full adder, but for a small difference in the borrow out circuit.



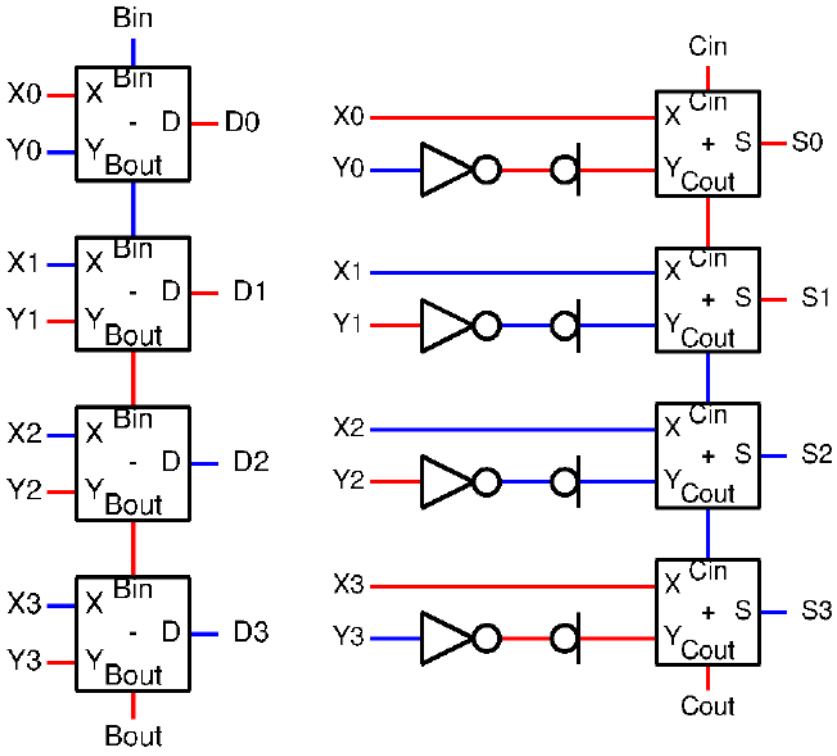


Two for One: Full subtracters can be replicated to form a multiple bit subtractor. But that's not how it's done. The hardware cost for subtraction would match the cost of addition. What if there were a way to get both operations using only one multi-bit arithmetic unit? How can an adder subtract? The answer comes from a little math and the magic of two's complement.

$$Z = X - Y \qquad\qquad Z = X + (-Y)$$

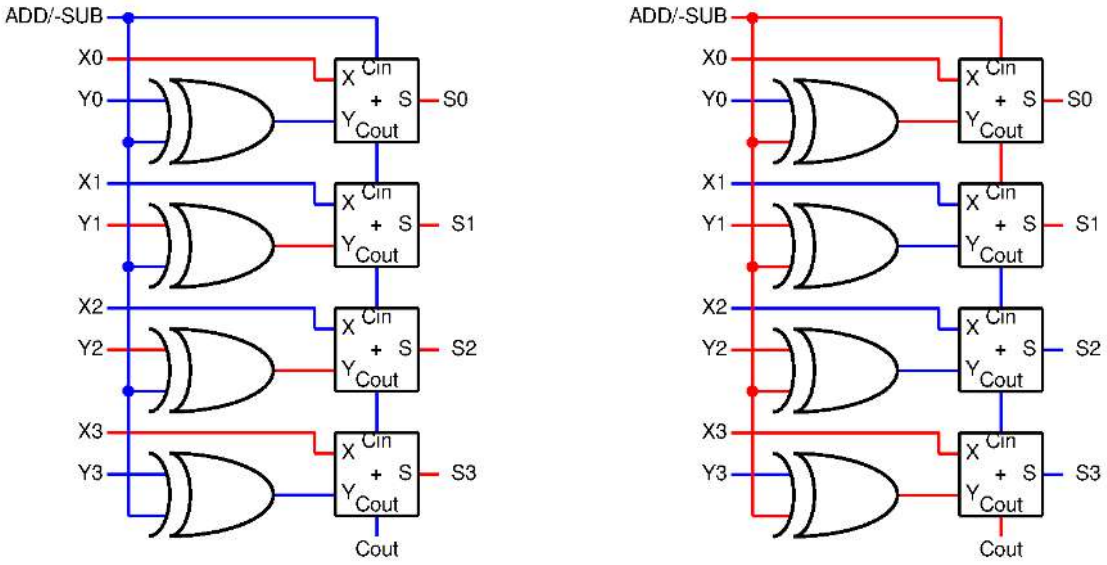
These two expressions compute the same result. So subtraction can be accomplished by adding the negation of the subtracted value. Negation is easy in two's complement. Just invert (complement) each bit and add one. The complement is done with inverters. The added one occurs by setting the carry in of the least significant bit to one. This "feature" was noted in the multi-bit adder.

Suppose the desired subtraction is 1001 – 0110. The result, 0011, can be computed using either full subtracters or full adders (with negation logic).



A multi-bit adder/subtracter can employ the adder hardware to perform addition and subtraction. A single control line $\overline{\text{ADD}}$/SUB determines which operation is to be performed. This rather unusual signal label declares that the signal is an active low *add* signal combined with an active high *subtract* signal. Since the signal is either low (zero) or high (one), the circuit is either adding or subtracting. To construct this, the inverters are replaced with selective inverters introduced in

Building Blocks, implemented as XOR gates. The $\overline{ADD}$/SUB signal also controls the carry in of the least significant bit of the adder. When high, it adds the one needed to negate the second operand being subtracted. The adder/subtractor implementation is shown below in both operation modes.

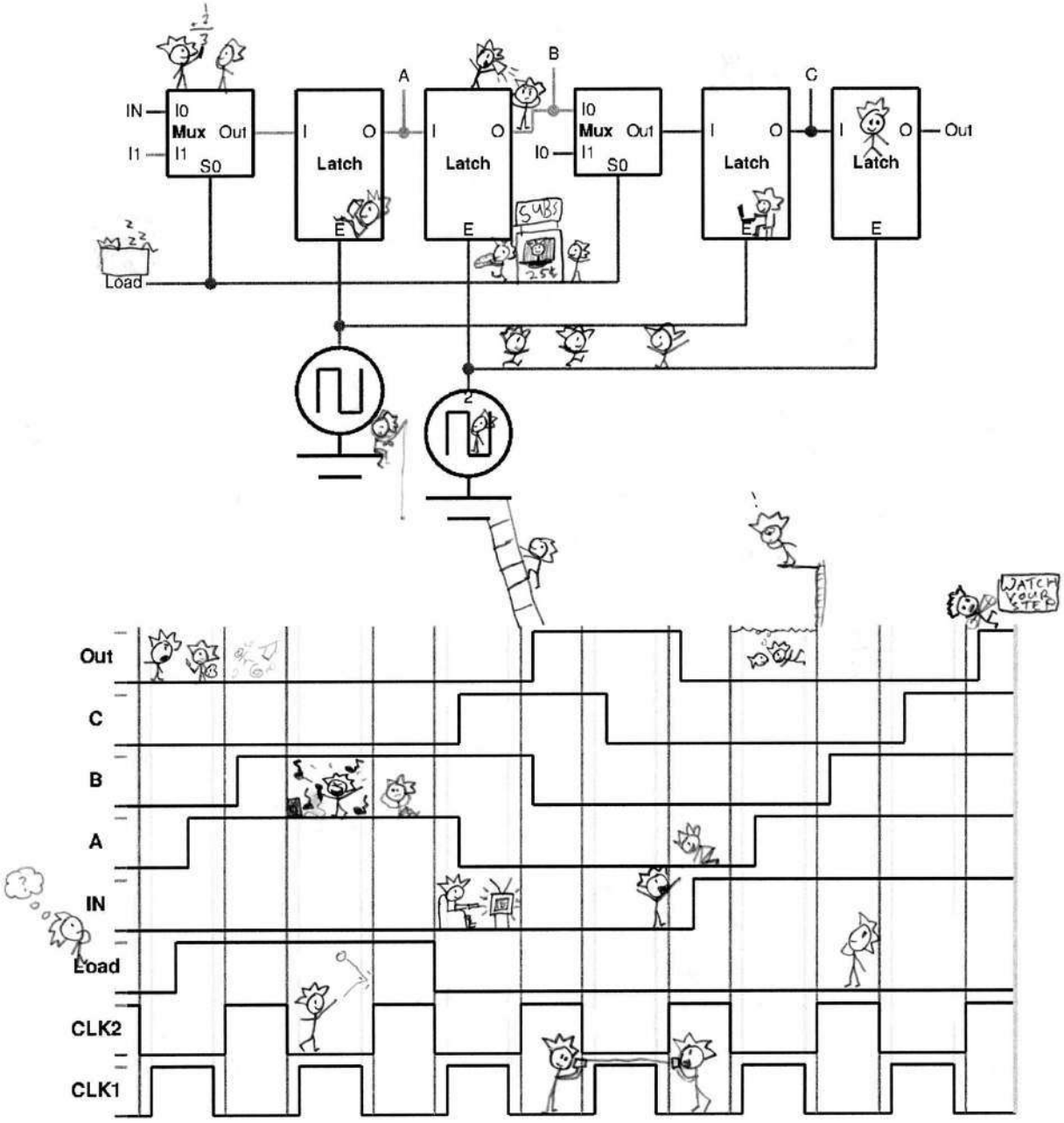


What About Overflows?: The signed overflow detection logic handles two's complement representations with this design. For unsigned, the carry out indicates addition overflow with a high value (One). Subtraction overflows are represented with a low value (zero) on carry out. The negation of the subtracted value wraps non-overflowing results into the unsigned overflow domain. Ironically, subtracted numbers larger than the first operand become small in negation and do not reach the unsigned overflow domain.

Summary:

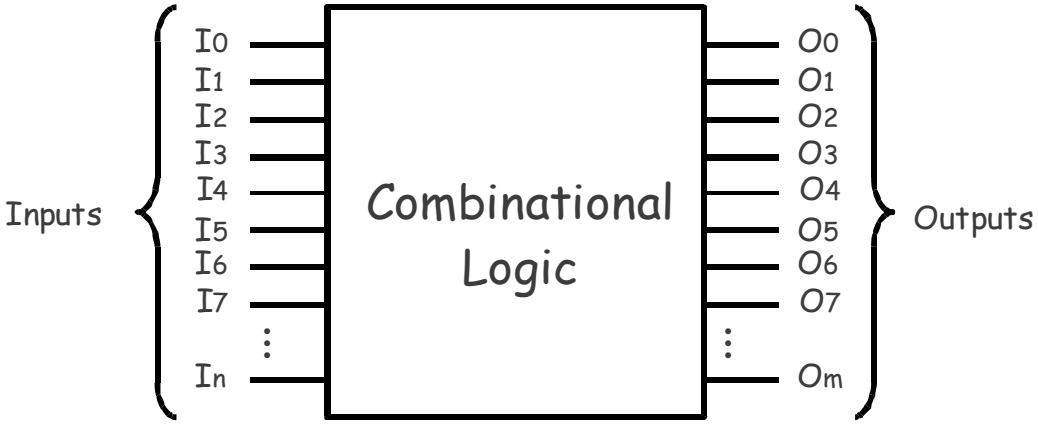
- Addition and subtraction follow the rules learned for decimal, but with a few small changes. One does not assume unlimited digits. And subtraction uses borrowing on credit to simplify protracted borrowing.
- Binary addition and subtraction are defined bit-wise, leading to the definition and construction of a full adder and a full subtractor. They are implemented using XOR and NAND gates.
- Overflows occur due to a bound word size. Error detection is straightforward and differs for unsigned and signed representations.
- An adder/subtractor can be constructed using a multi-bit adder and selectable negation circuitry for the second operand.

Latches and Registers



Latches and Registers

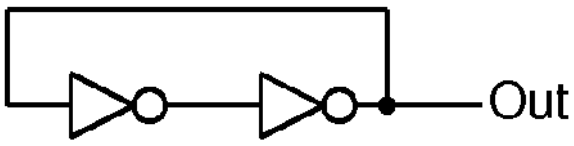
The potential to design functional blocks using switches and wire appears limitless. The analog quantities of our world can be represented using multi-bit words. Operations on these values, defined a Boolean expressions, can be constructed as *combinational logic* of unbound complexity. So what is missing? … *history.*



No matter how complex the implemented function is, it has no memory of previous values or results. All data to be processed must be presented to the combinational logic as inputs. Combinational logic has no state.

Building even the simplest digital system is impossible without state. Consider a basic four function calculator (a four banger). Although it can process big numbers rapidly, how can it solve a simple expression "5 + 3 =" without state? It would require the input keys "5", "+", "3", and "=" to be held simultaneously while the answer is displayed. Multi-digit math is out of the question.

One Bit Store: For useful digital computing systems, a simple block is needed that can store a bit of data for an extended period of time. That way data available now will persist, and can be used later. To best exploit the technology, this bit store must be built with switches and wire. And since many bits are needed, it must be implemented simply. Here's a starting point:



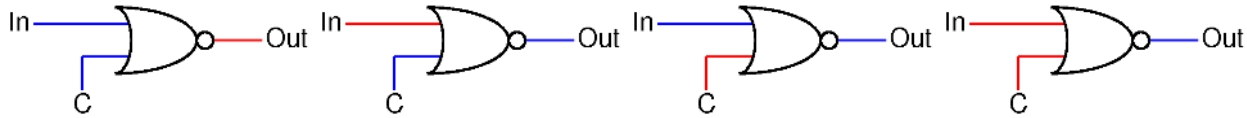
It certainly is simple, only two inverters (four switches). But why would somethings this simple have the ability to store data? The wire looping from the output to the

input is unusual: its uses its own output as an input. Analyzing this simple block is a challenge since, although it has only two nodes, it has no input values. So what are the nodes values?

Surprisingly, there are two answers. The output could be low (0) with the short wire between the inverters high (1). Or the output could be high with the short wire low. Which is right? They both are! This circuit is stable in two states; its *bi-stable*. Since a one bit store maintains one of two states, this simple block is ideal … except it has no input.

RS Latch: In order to use this bit store, it needs inputs that can set the output high, or reset the output low. But it still must retain the ability to hold a state, high or low, for an indefinite period. *Cross-coupled* inverting gates, where the output of each inverting gate is an input to the other, provides bi-stability. But inputs are need to Reset and Set it to a known state.

Consider a familiar gate, seen in a new way: a two input NOR. Assume one of the gate's input is a boolean variable In. The other is a control variable C.



C = 0 C = 1

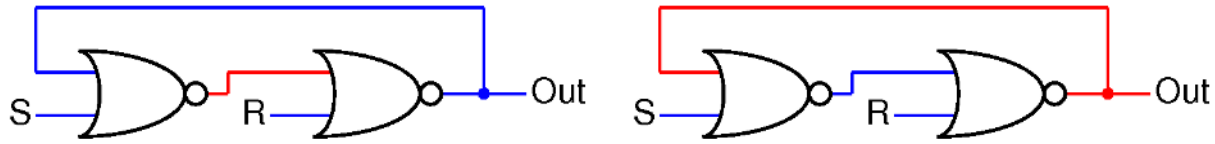
The control signal C determines whether output is related to In. If C is low, the output is the complement of In (i.e., it is an inverter). If C is high, the output is low no matter what the value of In is.

IN	C	Out	
A	0	$\overline{A}$	Out = $\overline{In}$
X	1	0	Out = 0

This is just what is needed. The heart of a bit store is two cross-coupled inverters. To force the output into one of two states, the inverter can be preempted. Using cross-coupled NOR gates, the control inputs turn off one of the two inverters, creating a known output (low or high).

When both R (reset) and S (set) are low, both NOR gates act as inverters. Their other input is complemented to become the output. So when Reset and Set are low,

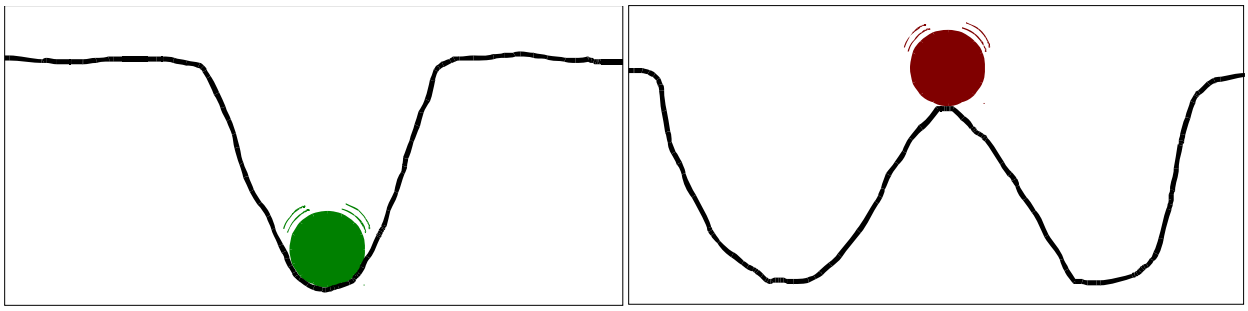
cross-coupled NOR gates become cross-coupled inverters used in the previous bit store implementation.



When either the Reset or Set inputs is temporary asserted (set high), it turns off an inverters, forcing the output into a known state. When Reset is high, Out is low independent of the first NOR gate's output. Since Out is low, the first NOR gate (acting as an inverter) makes the ignored input to the second NOR gate high. So when Reset returns low, and the second NOR gate becomes an inverter, Out remains low.

When Set is asserted (with Reset low), the output of the first NOR gate is low. With Reset low, the first NOR gate's output is inverted by the second NOR gate, setting Out high. Since this also sets the other input of the first NOR gate, the gates retain this state when Set is deasserted (set low).

Meta-Stability: Reset and Set can be asserted individually to force the RS latch into one of its two stable states. When both Reset and Set are low, this stable state remains unchanged. But what happens when Reset and Set are asserted simultaneously? Both NOR gates have low outputs. While this is logically correct, it can lead to an unpredictable state when Reset and Set are deasserted simultaneously. Which of the two stable states will it become? This condition is call *meta-stable* because the state of the latch is unknowable. It is also not knowable how long it will take for the latch to return to one of the two states.



"stable" "meta-stable"

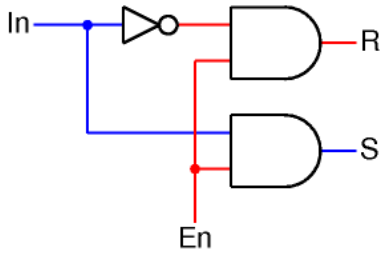
To appreciate the difference between stable and meta-stable, consider a ball in a valley versus a ball balanced in a peak. Small forces on the stable ball will not change its state. In contrast, a small force applied to the meta-stable ball will cause a significant state change. Needless to say, meta-stability should be avoided

when predictable storage is desired. So Reset and Set are only asserted individually.

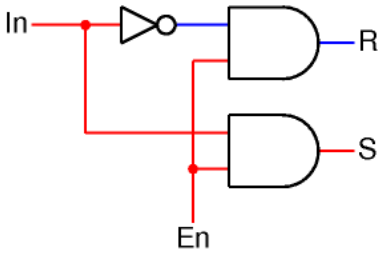
R	S	Out	
0	0	Q_o	hold
1	0	0	reset
0	1	1	set
1	1	0	avoid

The hold state employs a new symbol: Q_o to represent a previously defined state. It can be either 0 or 1, depending on whether Reset or Set was last asserted.

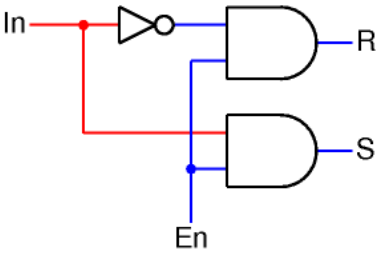
Transparent Latch: An RS latch can provide needed state. But it requires specific control signals to define and hold the storage. When a 0 is being stored, Reset is asserted and Set remains low. When a 1 is being stored, Set is asserted while R is low. When the value is held, neither Reset nor Set are asserted. This can be generated from an input In and an enable En that allows the input to be captured.



assert R

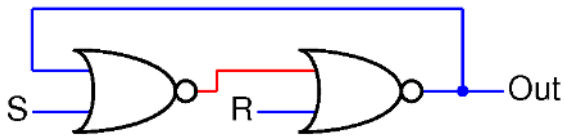


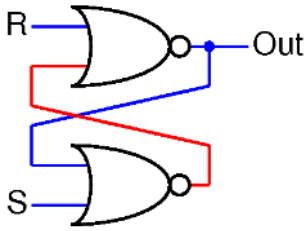
assert S



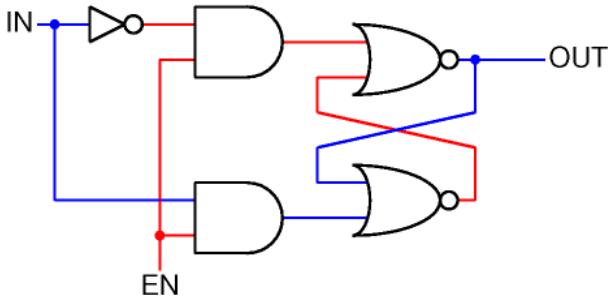
hold

Before connecting this circuity, the RS latch must be rearranged.

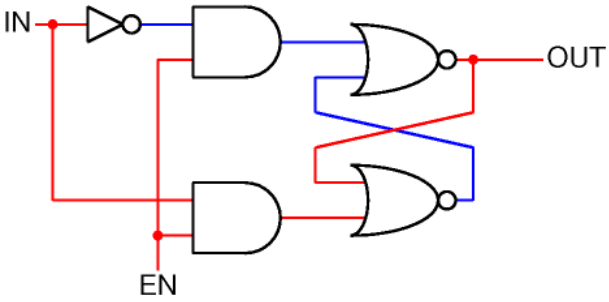




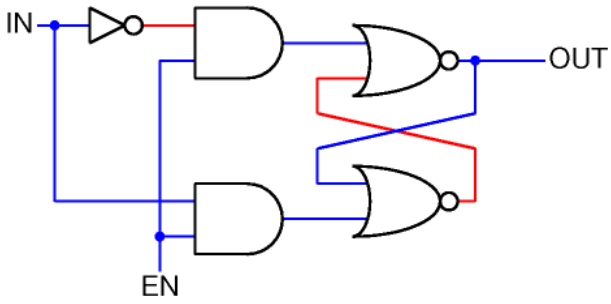
Combining both circuits produces a *transparent latch*. This transparent latch, shown below, can be in one of four different cases. Two are completely defined by the inputs: *storing 0* and *storing 1*. Two are defined by En and the internal state: *holding 0* and *holding 1*. In the second two cases, IN is ignored; it doesn't matter whether its one or zero because its masked by the AND gates.



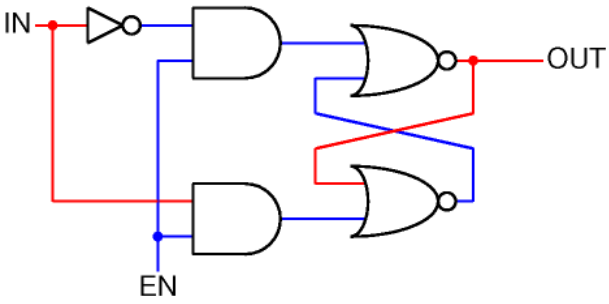
storing 0



storing 1



holding 0



holding 1

Here's the functional behavior of a transparent latch.

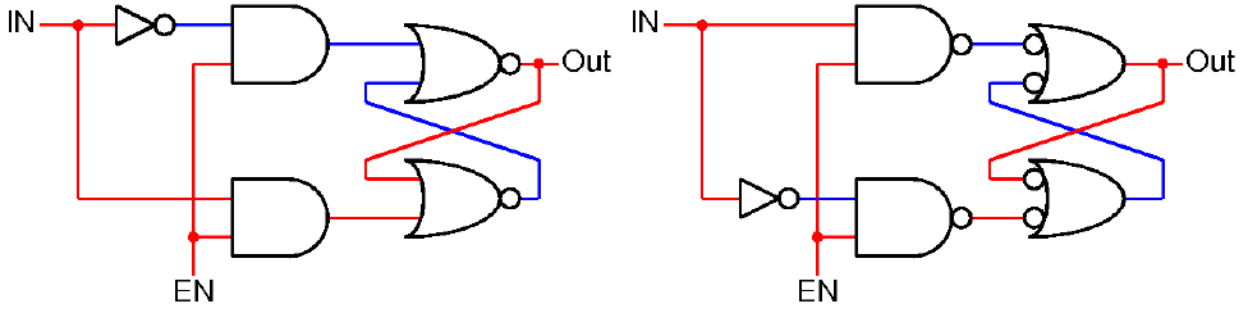
IN	En	Out	
X	0	Q_o	latch
A	1	A	transparent

When enable is asserted, the output follows the input so the latch becomes transparent. When enable is not asserted, the latch maintains the stored state on the output, independent of the input. Its value was defined at the last moment of transparency.

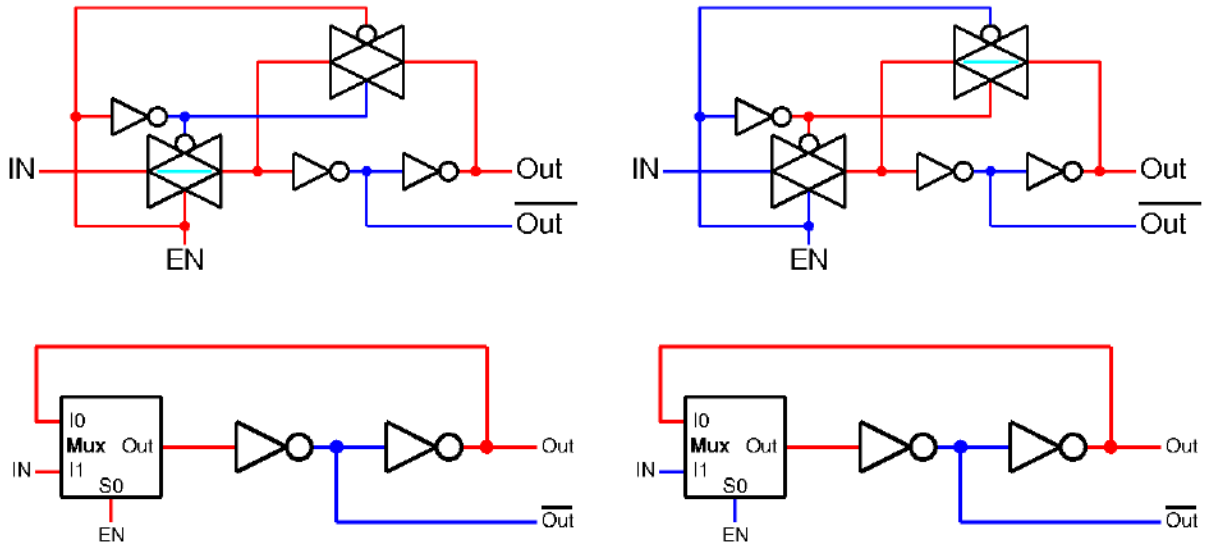
Implementation Costs: When implemented with switches and wire, this transparent latch requires two NOR gates (2 x 4 switches), two AND gates (2 x 6 switches), and one inverter (2 switches) for a total of 22 switches. Not bad. But digital systems require a lot of storage bits. Is there a cheaper implementation of this behavior using switches and wire?

Some tricks from mixed logic can help. If the bubbles on the NOR gates slide arround to the inputs and an extra pair of bubbles is added between the AND and OR, this implementation is transforms from two NOR and two AND gates to four NAND gates. Here the Set and Reset signal become active low (they are asserted when low, unasserted when high). Generating these signals requires the inverter to move down. But this latch implementation realizes the transparent latch behavior

with four NAND gates (4 x 4 switch) and one inverter (2 switches) for a total of 18 switches.



But can the count still be lowered? Let's go back to basics. Two cross coupled inverters are capable of storing a bit, but lack an input. Suppose pass gates are used to selective configure these inverters into either a transparent mode (where the the input is connected and the feedback pass is removed), or a hold mode (where the feedback path is connected and the input is removed). These pass gates serve as a two to one mux. It may be easier to understand when drawn as a mux.

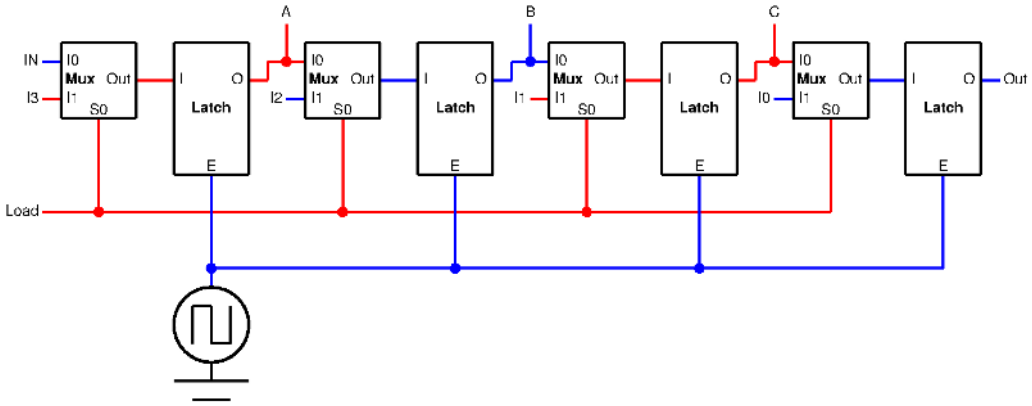


transparent mode hold mode

This implementation employs three inverters (2 x 2 switches) and two pass gates (2 x 2 switches) for a total of 10 switches. It is called a ten transistor latch and is the significant storage element in digital computation. Can we do better than ten transistors? Yes, but at a cost in speed, and only in dense arrays. More on this follows in the memory chapter.

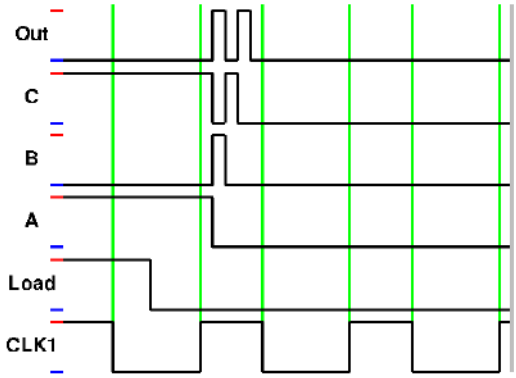
Latch Limitations: Latches can store a single bit of data, but with limitations. Consider a parallel to serial shift register. This is a device that can take parallel

word (in this case, four bits), and shift it down a serial wire in an orderly way. We use these devices to transfer data between our digital products. USB is short for "Universal Serial Bus". SATA means "Serial Advanced Technology Attachment". Strange as it may seem, serial buses often transfer data faster than parallel buses. So the need to load a parallel word into a clocked serial bus interface is widespread. Here's a first attempt.



Notice we are using latches to hold the word (I3:I0) when the Load signal is high. Then we use a simple clock to move bits along to Out.

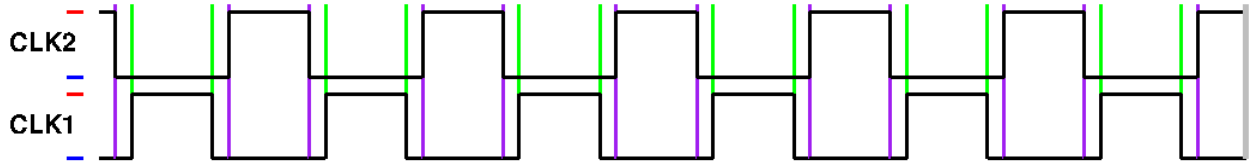
What is a Clock: In physics, the most used clock signal is a sinusoidal waveform. But in the digital world, everything is a one or a zero. A clock is a square wave that alternates between high and low at a defined period. A timing diagram shows the behavior as the load signal goes low and the data move serially through the latches. Time advances from left to right. Each signal is stacked with high and low values indicated by the red and blue marks. Note clock alternates between zero and one.



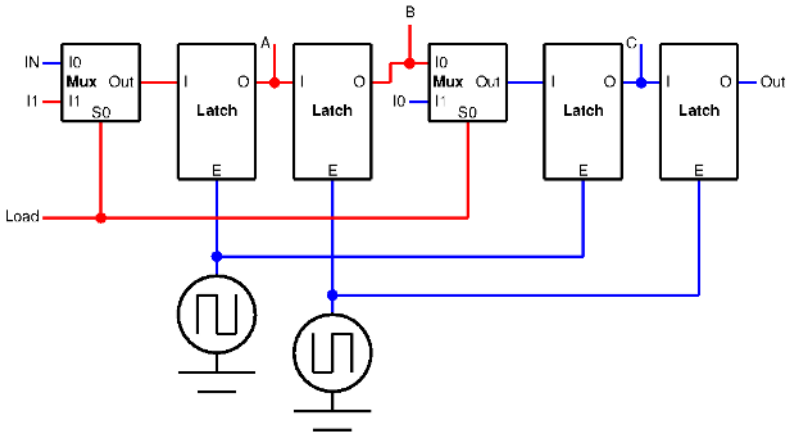
The problem occurs when the clock goes high (after Load goes low). All the enables on all of the latches go high and all latches become transparent. The stored data does travel to the output, but not in an orderly fashion. Instead bit race through

the latches independent of the clock. Not good. There is no way to capture and reconstruct the parallel word on the other end of the serial bus.

Two Phase, Non-Overlapping Clock: The problem with a latch is that is has no storage when its transparent. It can't hold an old value and accept a new value at the same time. To do accomplish this, there needs to be two latches and a special clocking scheme that allows one latch to hold the current value while a new value is being captured. The clocking scheme must allow each latch to be transparent independently, with a brief period in between where both latches are holding their value. Here's the clock that produces this behavior:

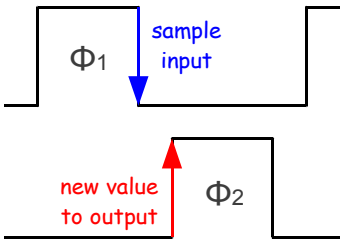


Both clocks have the same shape, including a small asymmetry of being low longer than being high. But the second clock is phase shifted by 180 degrees. Note also that the two clock are never high at the same time. This creates a two phase, non-overlapping clock. The clock signals are often named phi1 (Φ_1) and phi2 (Φ_2). It is widely used in digital computation where phase periods are set large enough to accommodate the gate depth x gate delay, and non-overlap periods are large enough to accommodate anticipated clock skew. This scheme will help create a workable shift register.

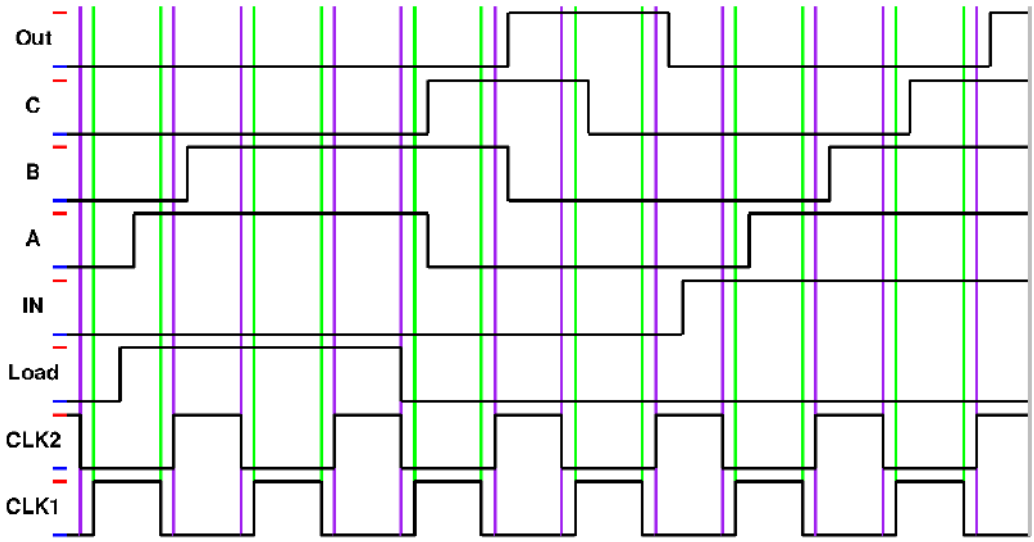


After data is loaded, it is advanced in the shift register in time with the clock frequency. This orderly movement is defined as the latch alternate between transparent and hold modes. During Φ_1, the first latch is transparent while new data is sampled. The falling edge of Φ_1 defines the sample point. Then on the rising

edge of Φ_2, this new value moves forward through the second (now transparent) latch.

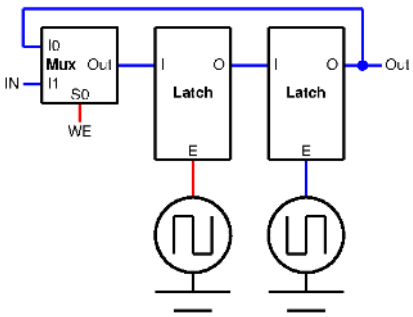


These two critical moments (the falling edge of Φ_1 and the rising edge of Φ_2) define this clock scheme behavior. Here's the timing diagram of this functional shift register:



Note the movement of ones and zeros through the monitor points A, B, C and Out. Of course, this four latch shift register can only maintain two bits. Half of the latches are transparent and cannot hold values.

Register: Two latches, plus the multiplexer form the core of a register.

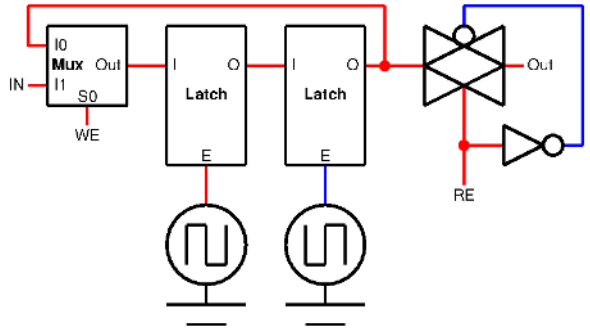


IN	WE	Clk	Out
X	0	→↓	Q_0
A	1	↑↓	A

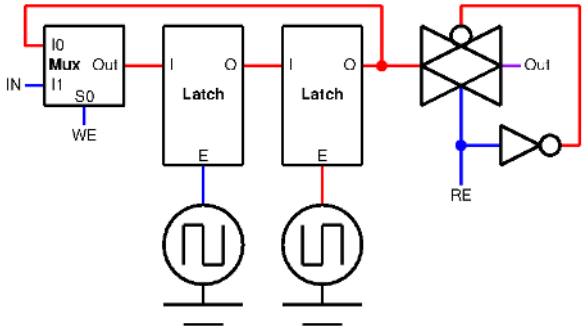
By connecting the output through a 2 to 1 mux to the input of the first latch, the ability to selectively write (or preserve) a register's value can be controlled. Like

the enable signal on the transparent latch, the write enable (WE) signal either selects a new input, or recycle the current output as the input. However this is a synchronous behavior in that the changing or preserving of a stored value is in sync with the clock signals. This selective write is call a *write port*.

Read Port: What would a *read port* be? Writing means changing the register state. Reading (or not) has no effect on its value. So what does a read port accomplish? Often registers are read onto a shared bus. Since only one value can be *read* onto the bus, a read port is a method of passing the register's contents onto a write (or not). This has been explored in demultiplexers, and is efficiently accomplished using a pass gate.



WE = RE = 1



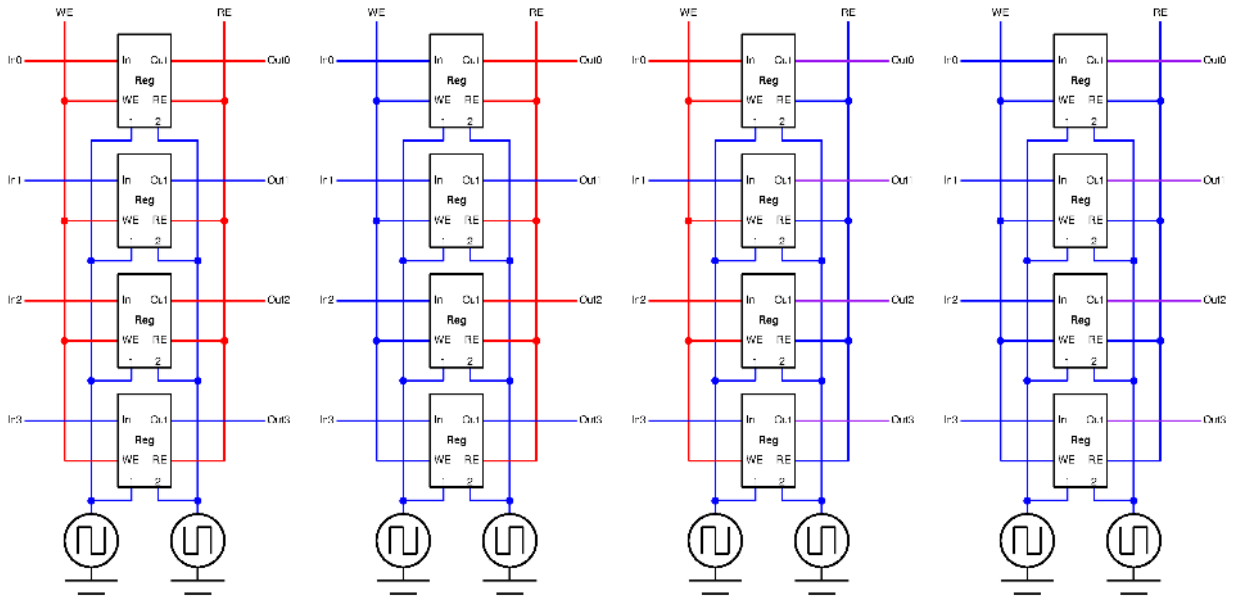
WE = RE = 0

The register on the left is being written and read. The register on the right is holding a value that is not being read (Out is floating). The behavior of this widely used storage element is shown below. Note that read and write are independent operations. Even when the register is not read (the output is floating), write operations can be performed. And when neither write or read operations are performed, a bit is still be stored.

IN	WE	RE	Clk	Out	
X	0	0	→↓	Z_0	hold
A	1	0	↑↓	Z_0	write
X	0	1	↑↓	Q_0	read
A	1	1	↑↓	*A*	write & read

Word-Wide Register: Once a one bit register is designed, it can be replicated to create a word wide register to store multi-bit values. These parallel registers are stored and loaded using multiple bit values. In this example, the word size is four bits. Control signals and clocks to read and write the register are shared. Individual lines for each input and out bit position are connected separately. Read

and write operations can be performed independently, something latches cannot do. In this example, the stored value of the register (0101) is unchanged in the four examples.



read & write | read | write | hold

Summary: Here's a summary of key points in digital storage:

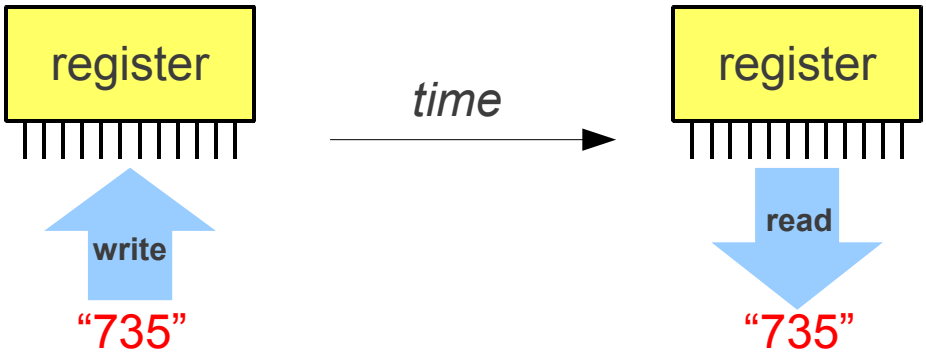
- Storage is needed for digital systems. It can created using simple cross-coupled inverting gates in a circuit that is bi-stable.
- A *Transparent Latch* is can store a bit of data, but it cannot hold data when a new bit is being stored. The 10T latch is the workhorse in digital systems.
- A *Register* can simultaneously be read and written. It is built of two latches, one to hold the current value while the other receives the new value.
- A *two-phase non-overlapping clock* provides necessary timing for digital systems. Its parameters (depth and non-overlap delay) determine performance of the digital system.
- A *shift register* shifts parallel words over a serial bus, often at high speeds. Serial interfaces are widely used in digital systems (USB, SATA, etc.).
- A timing diagram shows how sequential systems evolve in time. Behavioral tables cannot fully capture this information.

Designing Computer Systems

Counters

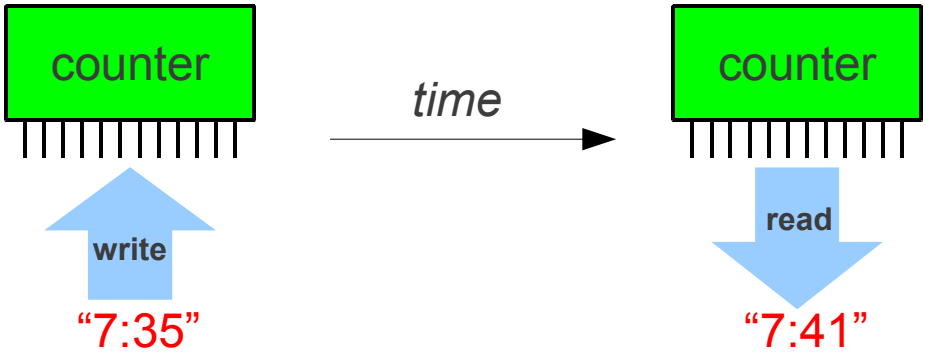
Counters

A register has a critical role to play in digital computing systems. Its stores data so it can be used in the future. Its singular purpose is to accurately preserve a value for an indeterminate period.



A value 735 is stored (written) in a register. Later, the register is read and the value 735 is obtained. The register's value is changed only when a new value is written or if the power source is removed.

Another form of storage is widely used in digital systems. It is constructed using similar components, but it has a different objective. Ironically, if it behaves like a register, *it is deemed faulty*.

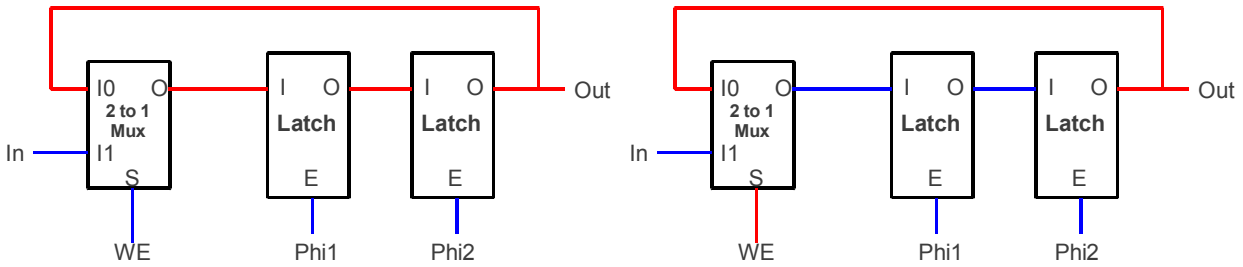


This storage is, of course, a *counter* and it is used for many purposes including time keeping. Here the clock is more than just a periodic sequencer of storage access. It also provides the time base that defines the counting interval. A counter is a multiple-bit storage element that follows a binary sequence in sync with a clock. It can also be described as counting the intervals of time defined by the clock's period. With a one Hertz clock, a counter *counts seconds*.

Stopwatch 101: Despite clocks being a prevalent example of counters in our world, its fraught with the peculiarities of time conventions. Time begins at 12:00. Then at 12:59, it *advances* to 1:00. Ante Meridian (AM) ends at 11:59 followed by 12:00 Post Meridian (PM). Who made this system up?

Stopwatches employ similar counter principles, but with more clarity. It begins at 0:00 when the clear button is pressed. When the start/stop button is pressed, the counter starts (or stops) measuring the passage of time. The lap button displays an interval while the counter continues.

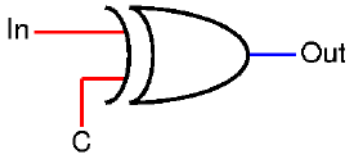
Remember the Register: Before exploring the construction of a counter, let's review the register. Because the counter must retain its current value while it is updated with its new value, it is based on a two latch cell that is similar to the register.



The 2-to-1 mux selects one of two values for the first latch's input: copy the current value from the second latch (right), or a new value from In (left). Because a register's roll is to preserve a value until it is overwritten, this is a good selection.

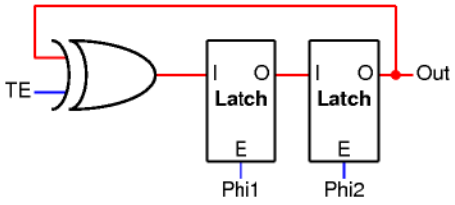
Counters need to count. In a binary world, a one-bit counter is rather dull: 0, 1, 0, 1, 0 ,1 , … this is well-described as toggling. But sometimes a one-bit counter needs to hold a value (1 or 0) for multiple cycles 0, 0, 0, … or 1, 1, 1, … The hold mode already exists in the register design. In this mode, on the left, the output value loops back to the input to *hold* the state. The toggling mode, on the right, is almost the same, except the complement of the output loops back to the input to toggle the state.

Since a cell will sometimes hold its value and sometime toggle its value, it must be able to do both, controlled by an input *toggle enable* (TE). This new input will control a selective inverter, otherwise known as a XOR gate (odd parity).

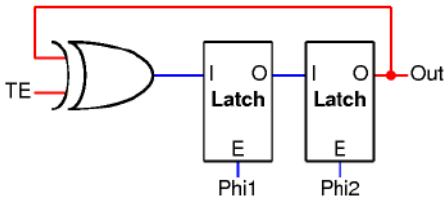


In	C	Out
A	0	A
A	1	$\overline{A}$

When the control input is low, A is passed through unchanged. When the control input is high, A is inverted $\overline{A}$. This gate can be used to create a cell that can toggle its output on each cycle of the clock (right). Or it can hold a value unchanged through a clock cycle (left).



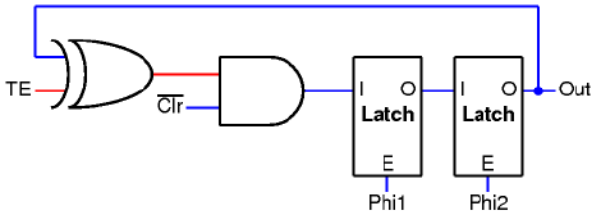
hold



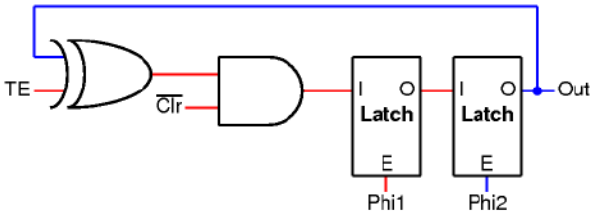
toggle

TE	Clk	Out
0	→↓	Q_o
1	↑↓	$\overline{Q}_o$

What State Am I: This cell has an essential feature, to selectively toggle. But it lacks definition. How can the state be set to a known value? The stopwatch must be able to be cleared, forcing all bits of the counter to zero. The current value must be *masked*. So an AND gate is used.



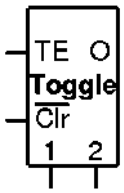
clear



toggle

The $\overline{Clr}$ signal is active low, indicated by the bar over the signal name. When the clear signal is asserted (low), the current value, toggled or held, is masked to zero. When the $\overline{Clr}$ signal is deasserted (high), the AND gate passes the toggled or held state from the second latch. In this example, it is toggled.

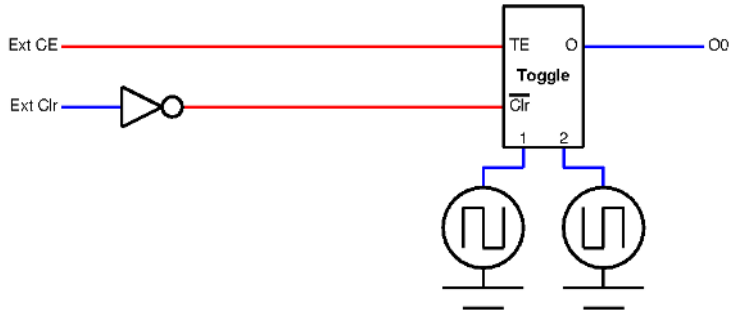
This design represents a one bit *toggle cell*. It can operate in three modes, clear, hold, and toggle. It is the foundation of multiple-bit counters. A simple icon is used to capture this implementation.

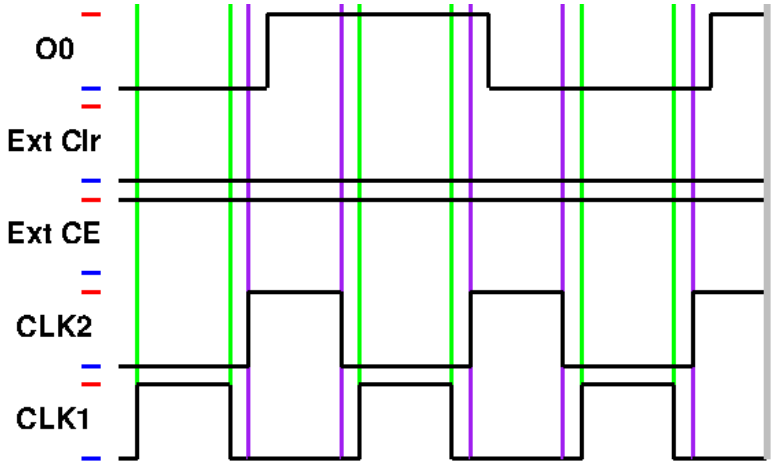


TE	$\overline{Clr}$	Clk	Out	
X	0	→↓	0	clear
0	1	↑↓	Q_o	hold
1	1	↑↓	$\overline{Q}_o$	toggle

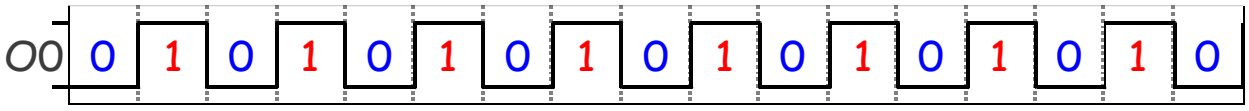
A stopwatch has two external controls. Count Enable (CE) allows the counter to advance with time. When it is not asserted, the counter is frozen at the current value. Clear (Clr) resets the counter value to zero. In both cases, the assertion of these signals is not immediate. The effect of these signals is seen at the counter's outputs at the end of each cycle. This is called *synchronous* operation since behavior is synchronized with the system clock.

N-bit Counter: A multi-bit counter begins with a single toggle cell connected to the external signals. The external count enable controls the toggle cell's toggle enable. The external clear controls the counter's clear; but it must be inverted to be active low. A **one-bit counter** is nothing more than a toggle cell.



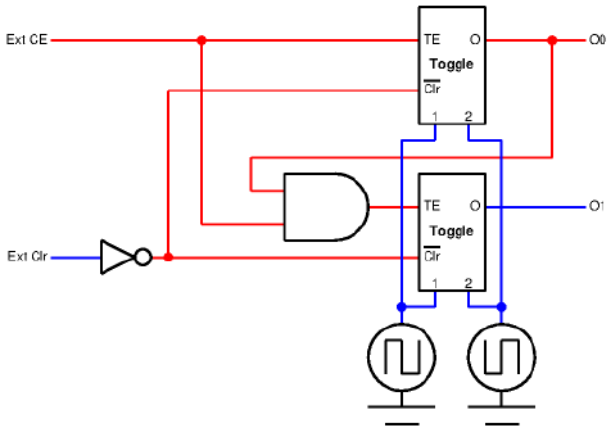


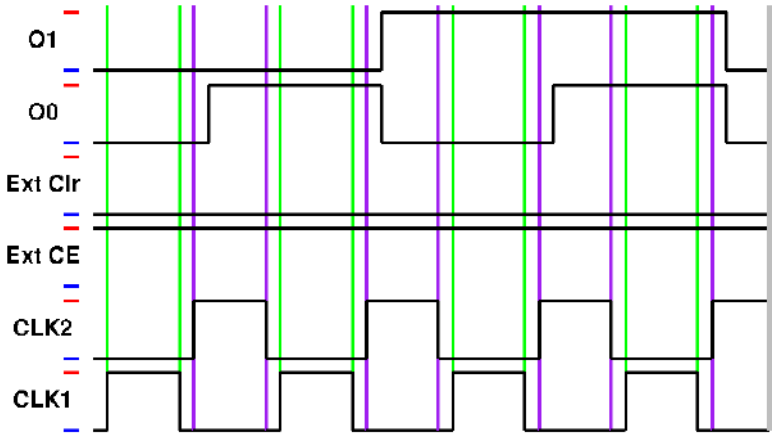
But when counting is enabled and the device is not being cleared, it counts with the system clock, its output toggling on each cycle. The timing diagram shows the one bit counter's output. The vertical dashed lines represent the system clock.



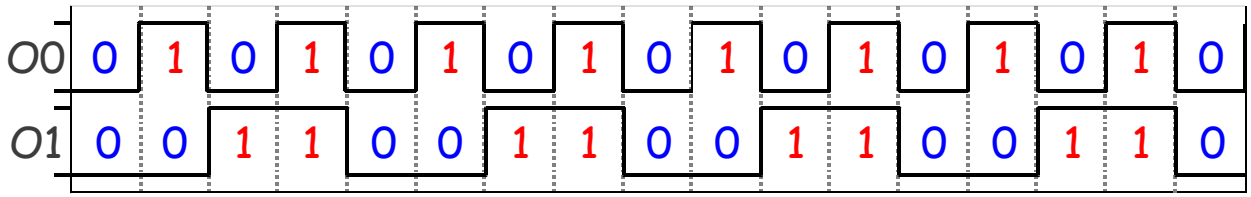
Because is *toggles* once per clock cycle, its cycle period is twice the system clock period, and its frequency is half the system clock. For this reason, a one bit counter is sometimes called a *divide by two* counter since it outputs a clock that is half the system clock frequency.

A **two bits counter** begins with two toggle cells. The first cell is connected directly to the external count enable and clear (as in the one bit counter). But the second toggle cell must be connected differently or it will toggle (count) identically to the first cell. For the second cell, it should toggle when the external count enable is asserted and the output of the first toggle cell is high.

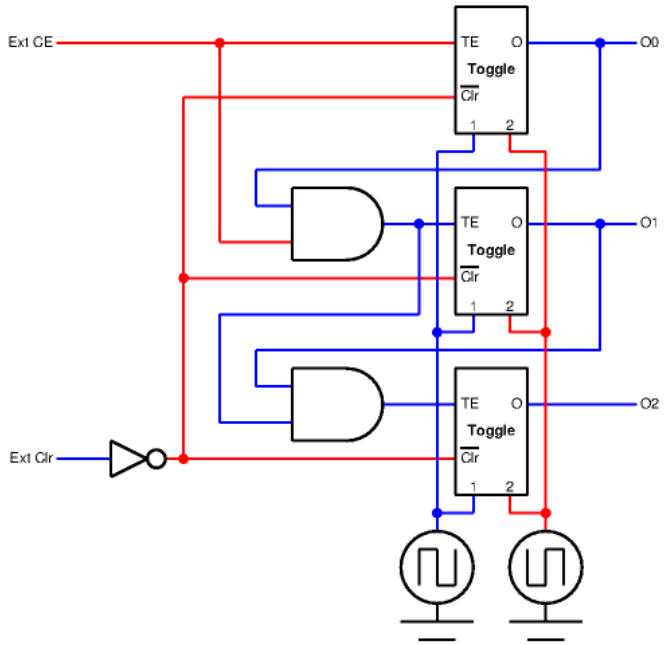


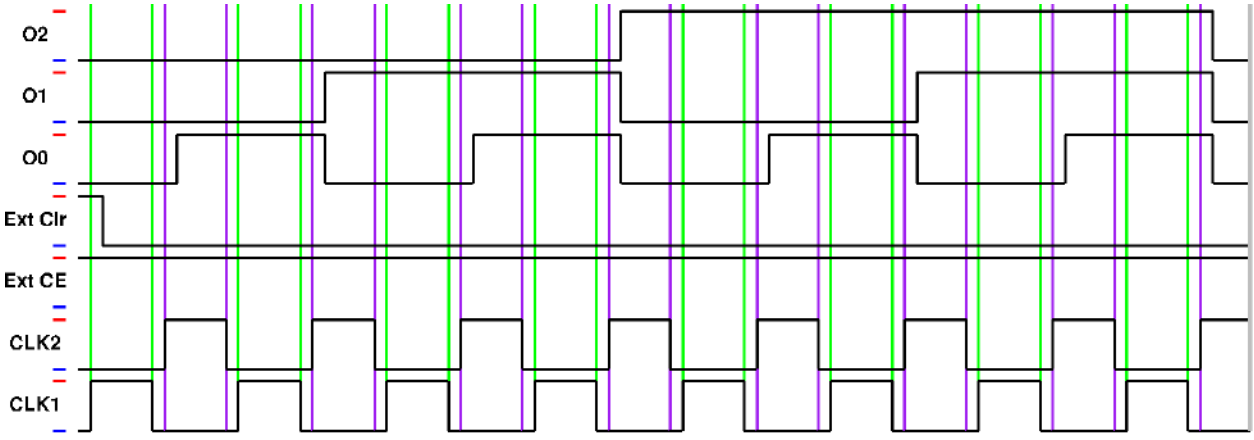


Since this is only true on half the cycles, the second cell will toggle *every other cycle*. Since the first output is already half the system clock, the second output is a divide by 4 (one fourth the system clock frequency).



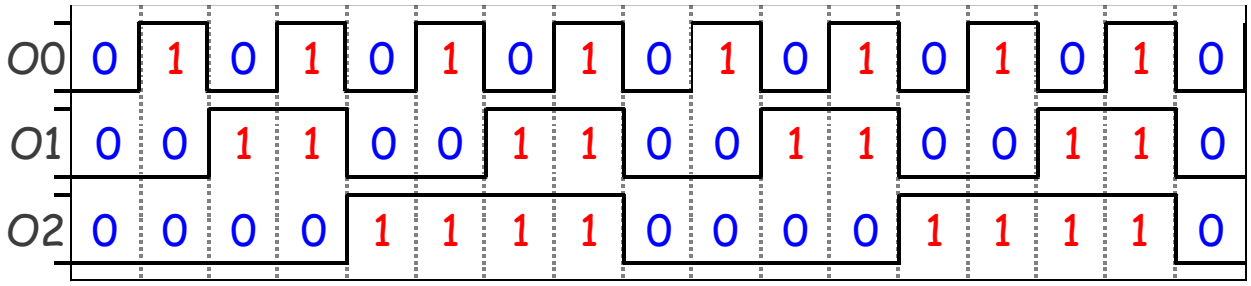
A **three bit counter** adds a third toggle cell.



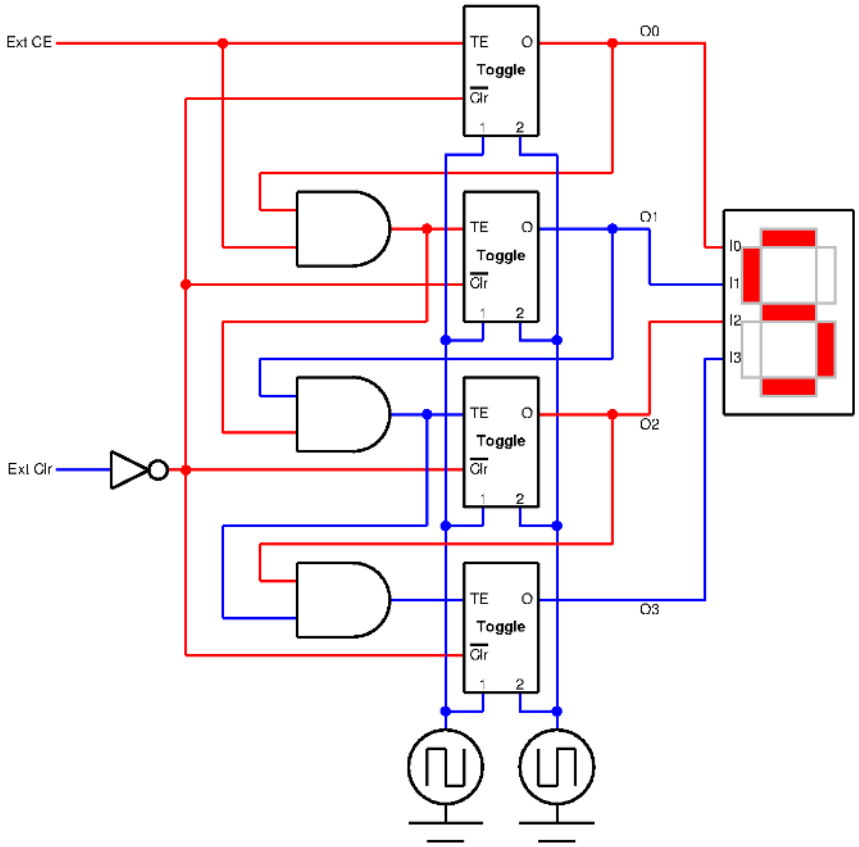


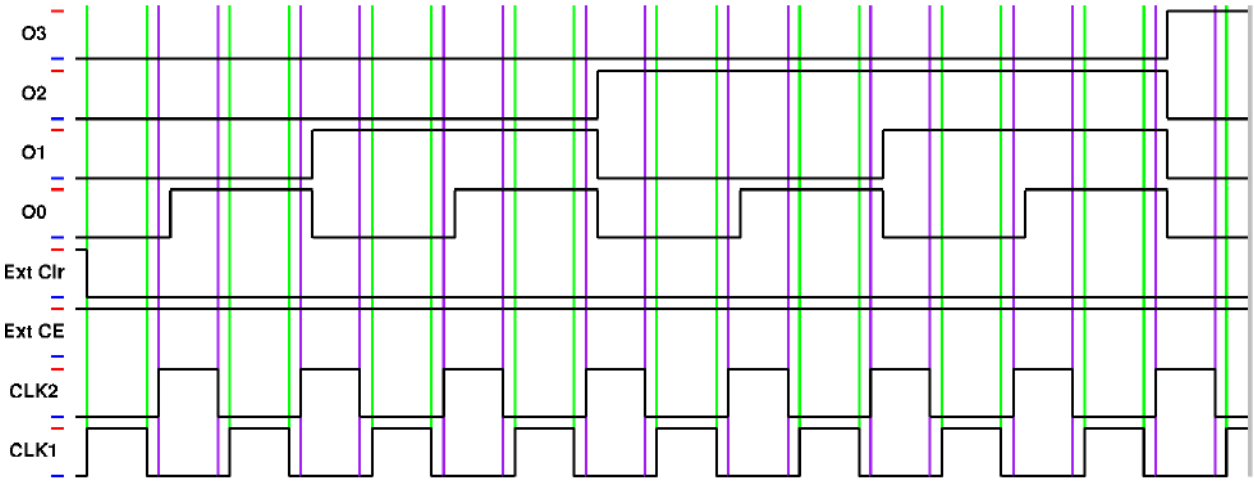
Whereas the first cell toggles every cycle, and the second toggle cell toggles every other cycle, the third toggle cell toggles *every fourth cycle*, when all lesser

significant bits are high. Its toggle enable is only high when O0 and O1 are high, along with the external count enable. Since two of these signals are already combined with an AND gate, its output can be ANDed with O1. The third output (O2) is a divide by 8 (one eighth the system clock frequency).

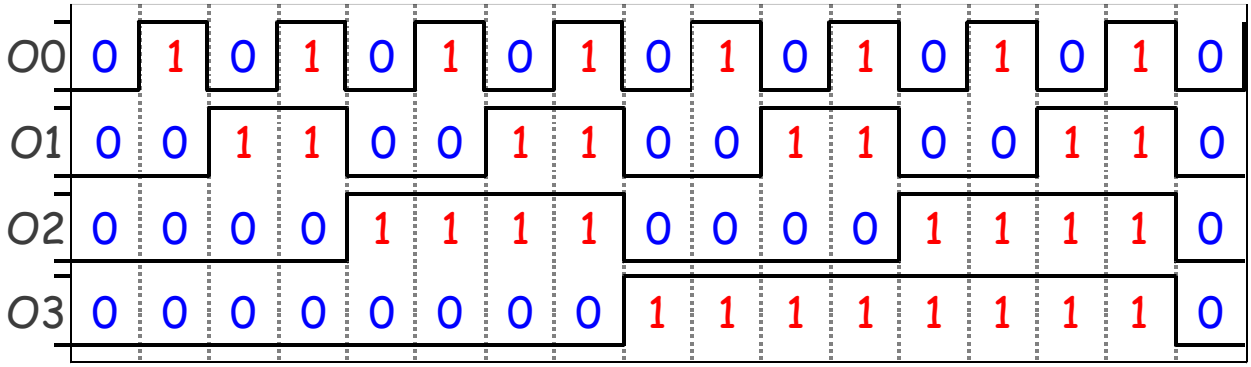


A **four bit counter** follows the same extension process. A fourth toggle cell is added and connected to toggle only when all lesser significant outputs are high, along with the external count enable. The new output, O3, toggles every sixteenth system clock cycle, as a divide by 16 counter. Here a decoder seven segment display has been added to show the numeric counting process.





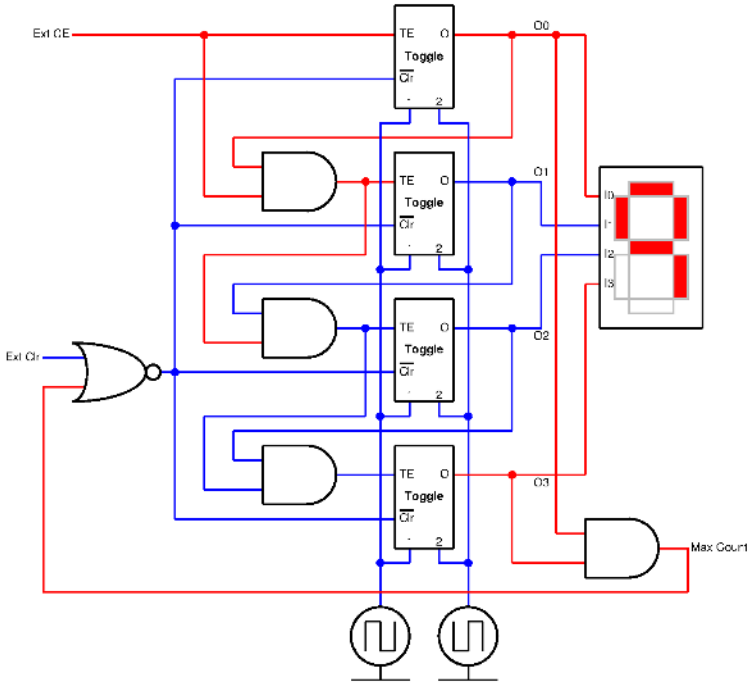
This process can be extended to an i-bit counter that divides the system clock by 2^i.



Divide by N: These counter divide by a power of two (two raised to the number of toggle cells). How would a counter count by an arbitrary value N? Since we live in decimal word (with ten fingers), a decade counter (divide by ten) would be a handy device. A four bit counter divides by 16, meaning it counts from '0' to '15' ('0' to 'F' in hexadecimal). A decade counter counts from '0' to '9' and then returns to '0'. Its maximum count is nine; when it reaches that count, to should return to zero on the next clock cycle. This can be accomplished using the max count detector and the external clear signal.

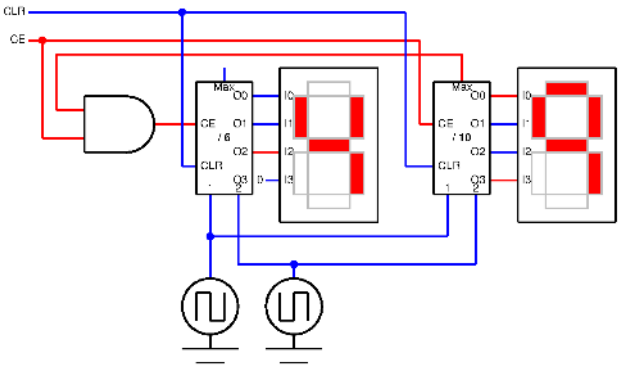
The max count detector monitors the counter outputs, detecting the maximum count. Normally, this would involve testing both high and low values in the maximum count. But since this is a up counter (it starts at zero and counts to all all ones), a more compact max count detector only compare *high values* in the max count. Differentiating low bits in the max count is pointless since it will never reach a high value; it will be cleared when it is low. A decade counter (divide by 10) has a

maximum value of nine. So a max count detector will monitor O0 and O3. When they are both high ('9'), the counter is cleared.

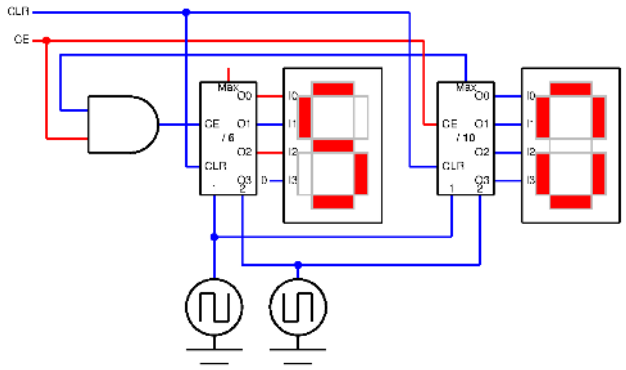


This method applies to an arbitrary value of N. For a divide by N counter, the maximum count is N-1. An N-1 detector will clear the counter on the cycle following the maximum count, restarting the counting sequence.

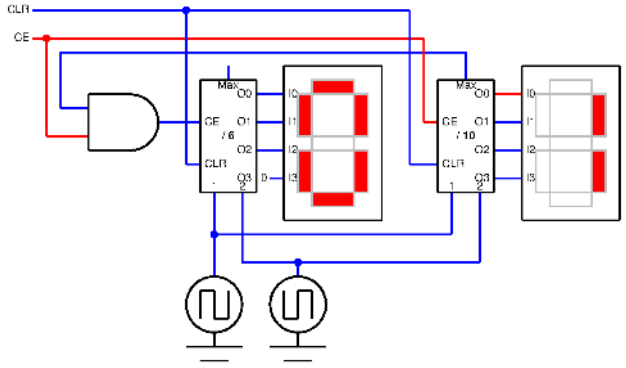
A Counting Flaw: This scheme works with single digit counters. But often multiple counter is required. For example, a stopwatch uses a decade counter to count seconds, and a divide by six counter to count tens of seconds. Note that the divide by six counter only counts when decade count has reached its maximum count.



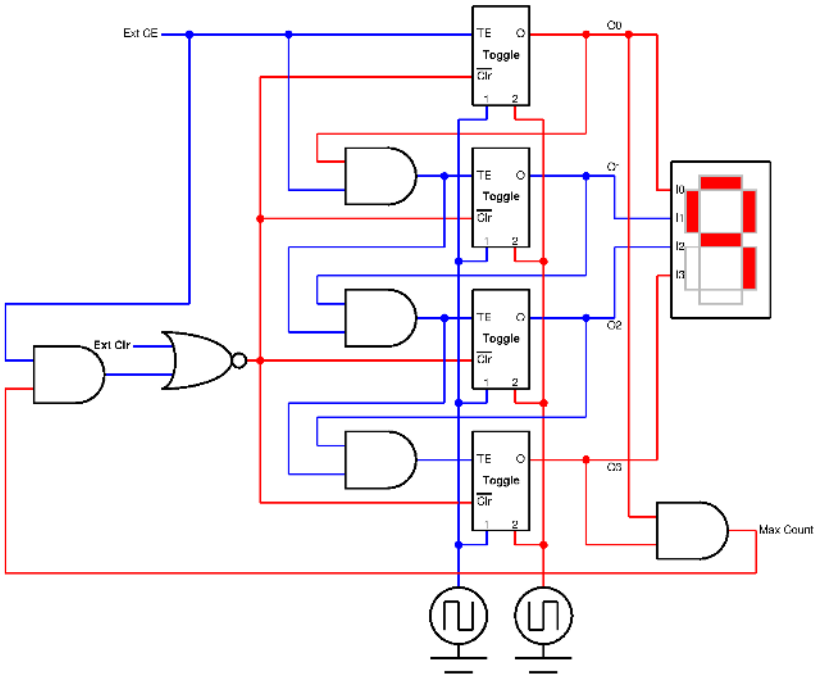
Using this technique will result in a counting flaw when it reaches '50'.



Here the decade counter clears to zero to begin the next decade. But since the divide by six counter has reached its maximum count, it automatically resets to zero on the next sequence.



A multi-digit counter reaches its maximum count does not *require* an automatic clear, since it can hold that maximum count until the next time it is incremented. The counter clear should when it has reached its maximum count *and* it ready to increment. Here's the correct divide by N counter. Note that the maximum count is reached, but the count enable is low. So the toggle cell clear signal is not asserted. Since the counter is frozen, the maximum count will remain until the external count enable is again asserted. At that time, the counter clear will be asserted and the counter will be cleared. Note also the external clear is not dependent on the value of the external count enable. When the user wants to clear the stopwatch, it is cleared imemdiately.



Summary: This chapter has introduced a new type of storage device that changes the stored value (in a predictable way) with time. Here are the main points:

- Counters are build from toggle cells that resemble the register cell used in the last chapter. However they don't have an input. Rather hey have a toggle enable (to count) and an active low clear (to reset them to a know value, zero).
- Toggle cells can be cascaded to produce a N bit counter. The counter will begin at zero and count to a maximum count of $2^N - 1$. These are called divide by 2^N counters since they divide the system clock (the two pahse non-overlapping clock) by 2^N.
- *A divide by N* counter can be built using a 2^N counter and a max count detector. The Max Count is N – 1. Since the counter starts at zero, only the high values in the maximum count need to be tested. This can be done with an AND gate.
- The Max Count signal can assert the Clear to reset the counter state. But only when Count Enable is high, when the counter is being instructed to counter *higher* than the maximum count.

