




























ECE253: Cheat Sheet

Module A: Intro to Digital Logic

https://hdlbits.01xz.net/wiki/Main_Page

A1 (1.1-1.4): Information Representation, Number Conversion, Binary Addition

A1.0 Terminology

Term	Definition																											
Abstraction	<p>Critical technique for managing complexity is abstraction: hiding details when they are not important.</p> <ul style="list-style-type: none">It is good to know something about the levels of abstraction immediately above and below where you are working to evaluate the impact of my design choices on other levels of abstraction. <div><table><tr><td>Application Software</td><td></td><td>Programs</td></tr><tr><td>Operating Systems</td><td></td><td>Device Drivers</td></tr><tr><td>Architecture</td><td></td><td>Instructions Registers</td></tr><tr><td>Micro-architecture</td><td></td><td>Datapaths Controllers</td></tr><tr><td>Logic</td><td></td><td>Adders Memories</td></tr><tr><td>Digital Circuits</td><td></td><td>AND Gates NOT Gates</td></tr><tr><td>Analog Circuits</td><td></td><td>Amplifiers Filters</td></tr><tr><td>Devices</td><td></td><td>Transistors Diodes</td></tr><tr><td>Physics</td><td></td><td>Electrons</td></tr></table></div>	Application Software		Programs	Operating Systems		Device Drivers	Architecture		Instructions Registers	Micro-architecture		Datapaths Controllers	Logic		Adders Memories	Digital Circuits		AND Gates NOT Gates	Analog Circuits		Amplifiers Filters	Devices		Transistors Diodes	Physics		Electrons
Application Software		Programs																										
Operating Systems		Device Drivers																										
Architecture		Instructions Registers																										
Micro-architecture		Datapaths Controllers																										
Logic		Adders Memories																										
Digital Circuits		AND Gates NOT Gates																										
Analog Circuits		Amplifiers Filters																										
Devices		Transistors Diodes																										
Physics		Electrons																										

Microarchitecture	Microarchitecture links the logic and architecture levels of abstraction.
Architecture	The architecture level of abstraction describes a computer from the programmer's perspective.
Discipline	<p><i>Discipline</i> is the act of intentionally restricting your design choices so that you can work more productively at a higher level of abstraction.</p> <ul style="list-style-type: none"> • Eg. Using interchangeable parts. • Eg. Rules of combinational composition
The Three Y's	<p>To manage complexity, we use the Three Y's:</p> <ol style="list-style-type: none"> 1. Hierarchy involves dividing a system into modules, then further subdividing each of these modules until the pieces are easy to understand. 2. Modularity states that modules have well-defined functions and interfaces so that they connect easily without unanticipated side effects. 3. Regularity seeks uniformity among modules. Common modules are reused many times, reducing the number of distinct modules that must be designed.
Discrete-Valued Variables	Digital systems represent information with discrete-valued variables —that is, variables with a finite number of distinct values
Binary Representation	Binary (two-valued) representation in which a high voltage indicates a "1" and a low voltage indicates a "0"
Amount of Information	<p>The amount of information D in a discrete valued variable with N distinct states is measured in units of <i>bits</i> (binary digits) as</p> $D = \log_2 N \text{bits}$
1 vs 0	We will use the terms "1," "True," and "High" synonymously. Similarly, we will use "0," "False," and "Low" synonymously.

Bytes	A group of eight bits is called a byte . It represents one of $2^8 = 256$ possibilities.
Nibbles	A group of four bits, or half a byte, is called a nibble . It represents one of $2^4 = 16$ <i>possibilities</i>
Words	Microprocessors handle data in chunks called words . The size of a word depends on the architecture of the microprocessor.
Least Significant Bit/Byte and Most Significant Bit/Byte	<p>The bit in the 1's column is called the least significant bit/byte (lsb/LSB).</p> <p>The bit at the other end is called the most significant bit/byte (msb/MSB)</p>
Kilo, Mega, Giga	<ul style="list-style-type: none"> • 1024 bytes is called a kilobyte (KB) or kibibyte (KiB). • 1024 bits is called a kilobit (Kb or Kbit) or kibibit (Kib or Kibit). • Similarly, MB/MiB, Mb/Mib, GB/GiB, and Gb/Gib are used for millions and billions of bytes and bits. • Bytes are used to measure memory capacity.
Overflow	<p>Digital systems usually operate on a fixed number of digits. Addition is said to overflow if the result is too big to fit in the available digits.</p> <ul style="list-style-type: none"> • A 4-bit number, for example, has the range [0, 15]. 4-bit binary addition overflows if the result exceeds 15.
Unsigned Binary Numbers	Unsigned binary numbers that represent positive quantities.
Signed Binary Numbers	Signed binary numbers represent both positive and negative quantities.

A1.1 Number Systems

Term	Definition																																																																																					
Decimal Numbers	<ul style="list-style-type: none">Decimal numbers are <i>base 10</i>.<ul style="list-style-type: none">Indicated with a subscript after the numberRange: An N-digit decimal number represents one of 10^N possibilities																																																																																					
Binary Numbers	<p>Bits represent one of two values, 0 or 1, and are joined together to form binary numbers.</p> <ul style="list-style-type: none">Binary numbers are <i>base 2</i><ul style="list-style-type: none">Indicated with “0b”Range: An N-bit binary number represents one of 2^N possibilities. <div><p>Table 1.1 Binary numbers and their decimal equivalent</p><table><tr><th>1-Bit Binary Numbers</th><th>2-Bit Binary Numbers</th><th>3-Bit Binary Numbers</th><th>4-Bit Binary Numbers</th><th>Decimal Equivalents</th></tr><tr><td>0</td><td>00</td><td>000</td><td>0000</td><td>0</td></tr><tr><td>1</td><td>01</td><td>001</td><td>0001</td><td>1</td></tr><tr><td></td><td>10</td><td>010</td><td>0010</td><td>2</td></tr><tr><td></td><td>11</td><td>011</td><td>0011</td><td>3</td></tr><tr><td></td><td></td><td>100</td><td>0100</td><td>4</td></tr><tr><td></td><td></td><td>101</td><td>0101</td><td>5</td></tr><tr><td></td><td></td><td>110</td><td>0110</td><td>6</td></tr><tr><td></td><td></td><td>111</td><td>0111</td><td>7</td></tr><tr><td></td><td></td><td></td><td>1000</td><td>8</td></tr><tr><td></td><td></td><td></td><td>1001</td><td>9</td></tr><tr><td></td><td></td><td></td><td>1010</td><td>10</td></tr><tr><td></td><td></td><td></td><td>1011</td><td>11</td></tr><tr><td></td><td></td><td></td><td>1100</td><td>12</td></tr><tr><td></td><td></td><td></td><td>1101</td><td>13</td></tr><tr><td></td><td></td><td></td><td>1110</td><td>14</td></tr><tr><td></td><td></td><td></td><td>1111</td><td>15</td></tr></table></div>	1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents	0	00	000	0000	0	1	01	001	0001	1		10	010	0010	2		11	011	0011	3			100	0100	4			101	0101	5			110	0110	6			111	0111	7				1000	8				1001	9				1010	10				1011	11				1100	12				1101	13				1110	14				1111	15
1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents																																																																																		
0	00	000	0000	0																																																																																		
1	01	001	0001	1																																																																																		
	10	010	0010	2																																																																																		
	11	011	0011	3																																																																																		
		100	0100	4																																																																																		
		101	0101	5																																																																																		
		110	0110	6																																																																																		
		111	0111	7																																																																																		
			1000	8																																																																																		
			1001	9																																																																																		
			1010	10																																																																																		
			1011	11																																																																																		
			1100	12																																																																																		
			1101	13																																																																																		
			1110	14																																																																																		
			1111	15																																																																																		
Hexadecimal Numbers	<p>A group of four bits represents one of $2^4 = 16$ possibilities.</p> <ul style="list-style-type: none">Hexadecimal are <i>base 16</i>.<ul style="list-style-type: none">Indicated with “0x”Hexadecimal numbers use digits 0-9 with letters A-FOne hexadecimal digit stores one nibble.																																																																																					

Table 1.2 Hexadecimal number system

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

A1.2 Number Conversion

Term	Definition
D -> B (Right-to-Left)	<ol style="list-style-type: none"> See if the number is odd or even. <ol style="list-style-type: none"> If odd, the digit is 1 If even, the digit is 0 Represent <i>New Number</i> = $\frac{\text{Number} - (1 \text{ or } 0 \text{ depending on binary digit})}{2}$ Repeat steps 1-2 with the new number
D -> B (Left-to-Right)	<ol style="list-style-type: none"> Determine the largest power of 2 that fits into the number. Then represent the difference with the largest power of 2 that fits into the difference of the two numbers. Repeat steps 1-2 For the powers of 2 that weren't needed, treat them as 0s, and the rest as 1s.

D -> H (Left-to-Right)	<ol style="list-style-type: none"> 1. Determine the largest power of 16 that fits into the number. 2. Determine how many times the power of 16 can go into the number. 3. Then find the difference between the largest power of 16 and the number 4. Repeat steps 1-3 5. Once you get to a number that can be represented with powers of 16, find the hexadecimal equivalent.
B -> D	<ol style="list-style-type: none"> 1. Expand the binary code as the full sum. 2. Add up all the terms.
B -> H (Right-to-left Always))	<ol style="list-style-type: none"> 1. Group the bits in 4s. 2. Find the hexadecimal equivalent starting from right to left. <ul style="list-style-type: none"> • Note: If there aren't exactly groups of 4, then add 0s to the left hand side to match it.
H -> B	<ol style="list-style-type: none"> 1. Just find the binary equivalent.
H -> D	<ol style="list-style-type: none"> 1. Convert to binary 2. Then use B -> D <p>OR</p> <ol style="list-style-type: none"> 1. Expand out H 2. And combine terms together.

A1.3 Powers of Two

$2^1 = 2$	$2^9 = 512$
$2^2 = 4$	$2^{10} = 1024$
$2^3 = 8$	$2^{11} = 2048$
$2^4 = 16$	$2^{12} = 4096$
$2^5 = 32$	$2^{13} = 8192$
$2^6 = 64$	$2^{14} = 16384$
$2^7 = 128$	$2^{15} = 32768$
$2^8 = 256$	$2^{16} = 65536$

A1.4 Binary Addition

In binary addition, if the sum of two numbers is greater than 1, we carry the 2's digit over to the next column.

- The bit that is carried over to the neighboring column is called the **carry bit**.

A1.5 Two's Complement Numbers

Two's complement numbers are identical to unsigned binary numbers except that the most significant bit position has a weight of -2^{N-1} instead of 2^{N-1} .

- Zero is written as all zeros: $00...000_2$.
- -1 is written as all ones: $11...111_2$

Sign bit:

- Positive numbers have a 0 in the most significant position.
- Negative numbers have a 1 in the most significant position.

Reversing the Sign Method (Only Used When Working With Negative Two's Complement Numbers)**:

The sign of a two's complement number is reversed by inverting the bits in the number and then adding 1 to the least significant bit position.

- To find a representation of a negative number or determine the magnitude of a negative number.

Range of an N-bit Two's Complement Number:

$$[-2^{N-1}, 2^{N-1} - 1]$$

Overflow:

Overflow occurs if the two numbers being added have the same sign bit and the result has the opposite sign bit.

Sign Extension:

When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions.

- **Example:** The numbers 3 and -3 are written as 4-bit two's complement numbers 0011 and 1101, respectively. They are sign-extended to seven bits by copying the sign bit into the three new upper bits to form 0000011 and 1111101, respectively, which are 7-bit representations of 3 and -3.

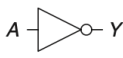
A2 (1.5): Introduction to Logic Gates

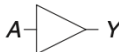
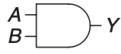

A2.0 Terminology




Term	Definition
Logic Gates	<p>Logic gates are simple digital circuits that take one or more binary inputs and produce a binary output.</p> <ul style="list-style-type: none"> • Logic gates are drawn with a symbol showing the input (or inputs) and the output. • The relationship between the inputs and the output can be described by a truth table or a Boolean equation.


Truth Table	<p>A truth table lists inputs on the left and the corresponding output on the right. It has one row for each possible combination of inputs.</p> <p>Convention for Truth Tables: The inputs are listed in the order 00, 01, 10, 11,..., as if you were counting in binary.</p>
Boolean Equation	A Boolean equation is a mathematical expression using binary variables.

A2.1 Types of Gates

Term	Definition						
NOT Gate (Inverter)	<p>A NOT gate is a one-input logic gate. The NOT gate’s output is the inverse of its input.</p> <div> <p>NOT</p>  <p>$Y = \overline{A}$</p> <table border="1"> <thead> <tr> <th>A</th><th>Y</th></tr> </thead> <tbody> <tr> <td>0</td><td>1</td></tr> <tr> <td>1</td><td>0</td></tr> </tbody> </table> <p>Figure 1.12 NOT gate</p> </div> <ul style="list-style-type: none"> • Notation for NOT in Boolean: ' or $\overline{}$ • A circle on the output is called a bubble and indicates inversion. 	A	Y	0	1	1	0
A	Y						
0	1						
1	0						
Buffer	A BUF is also a one-input logic gate. It simply copies the input to the output (ie. a wire).						

	<div>BUF</div> <div></div> <div>$Y = A$</div> <div><table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table></div> <div>Figure 1.13 Buffer</div>	A	Y	0	0	1	1									
A	Y															
0	0															
1	1															
AND Gate	<p>An AND gate is a two-input logic gate. AND gate produces a TRUE output, Y, if and only if both A and B are TRUE. Otherwise, FALSE.</p> <div>AND</div> <div></div> <div>$Y = AB$</div> <div><table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></div> <div>Figure 1.14 AND gate</div> <ul style="list-style-type: none">• Notation for AND in Boolean: \cdot or <i>nothing</i>• Note: ONLY FOR TWO INPUT GATE	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Y														
0	0	0														
0	1	0														
1	0	0														
1	1	1														
OR Gate	<p>OR gate produces a TRUE output, Y, if either A or B (or both) are TRUE. Otherwise, FALSE.</p> <div>OR</div> <div></div> <div>$Y = A + B$</div> <div><table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></div> <div>Figure 1.15 OR gate</div> <ul style="list-style-type: none">• Notation for AND in Boolean: $+$• Note: ONLY FOR TWO INPUT GATE	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y														
0	0	0														
0	1	1														
1	0	1														
1	1	1														
XOR	<p>XOR produces a TRUE if A or B, but not both, are TRUE. Otherwise, FALSE.</p>															

	<div><p>XOR</p><p>$Y = A \oplus B$</p><table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></div> <div><ul style="list-style-type: none">• Notation for AND in Boolean: \oplus• Note: ONLY FOR TWO INPUT GATE</div>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Y														
0	0	0														
0	1	1														
1	0	1														
1	1	0														
NAND	<p>NAND gate performs NOT AND. Its output is TRUE unless both inputs are TRUE, then FALSE.</p> <div><p>NAND</p><p>$Y = \overline{AB}$</p><table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></div> <div><ul style="list-style-type: none">• Note: ONLY FOR TWO INPUT GATE</div>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Y														
0	0	1														
0	1	1														
1	0	1														
1	1	0														
NOR	<p>NOR gate performs NOT OR. Its output is TRUE if neither A nor B is TRUE.</p> <div><p>NOR</p><p>$Y = \overline{A + B}$</p><table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table></div> <div><ul style="list-style-type: none">• An N-input XOR gate is sometimes called a parity gate and produces a TRUE output if an odd number of inputs are TRUE.</div>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Y														
0	0	1														
0	1	0														
1	0	0														
1	1	0														

	<ul style="list-style-type: none">● Note: ONLY FOR TWO INPUT GATE															
XNOR (Equality Gate)	<p>XNOR gate performs the inverse of XOR. XNOR output is TRUE if both inputs are FALSE or TRUE.</p> <p style="text-align: center;">XNOR</p> <div style="text-align: center;"> $Y = \overline{A \oplus B}$<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></div> <p style="text-align: center;">Figure 1.17 XNOR gate</p>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1
A	B	Y														
0	0	1														
0	1	0														
1	0	0														
1	1	1														

Convention for a Bubble:
Any gate can be followed by a bubble to invert its operation.

Note:
There can be three or more inputs that follow the same function.

A3 (2.1-2.3): Introduction to Boolean Algebra

A3.0 Terminology

Term	Definition
Circuit	<p>A circuit is a network that processes discrete-valued variables. A circuit can be viewed as a black box with</p> <ul style="list-style-type: none"> ● one or more discrete-valued input terminals ● one or more discrete-valued output terminals ● a functional specification describing the relationship between inputs and outputs ● a timing specification describing the delay between inputs changing and outputs responding.

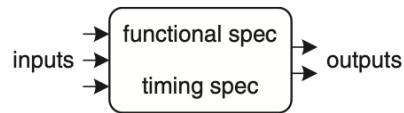


Figure 2.1 Circuit as a black box with inputs, outputs, and specifications

- Circuits are made up of nodes and elements.

Element and Node

An **element** is itself a circuit with inputs, outputs, and a specification.

A **node** is a wire, whose voltage conveys a discrete-valued variable.

Nodes are classified as **input, output, or internal**.

- Inputs receive values from the external world
- Outputs deliver values to the external world
- Wires that are not inputs/outputs are internal nodes.

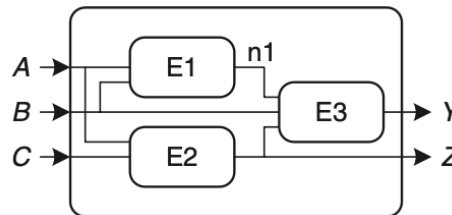


Figure 2.2 Elements and nodes

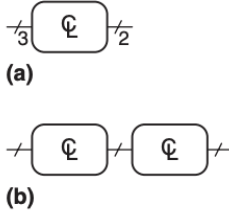
- 3 elements, E1, E2, and E3, and six nodes. Nodes A, B, and C are inputs. Y and Z are outputs. n1 is an internal node between E1 and E3.

Combinational vs. Sequential

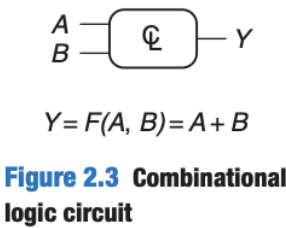
Digital circuits are classified as **combinational** or **sequential**.

A **combinational** circuit's outputs depend only on the current values of the inputs.

- AKA, It combines the current input values to compute the output.
- **Memoryless**
- **Eg.** A logic gate is a combinational circuit.
- The functional specification of a combinational circuit expresses the output values in terms of the current input values.
 - Expressed as a truth table or a Boolean equation.

	<ul style="list-style-type: none"> • The timing specification of a combinational circuit consists of lower and upper bounds on the delay from input to output. <p>A sequential circuit's outputs depend on both current and previous values of the inputs.</p> <ul style="list-style-type: none"> • AKA, it depends on the input sequence. • Has memory
Bus	<p>To simplify drawings, use a single line with a slash through it and a number next to it to indicate a bus, a bundle of multiple signals.</p> <ul style="list-style-type: none"> • The number specifies how many signals are in the bus. <div style="text-align: center;">  <p>(a)</p> <p>(b)</p> <p>Figure 2.6 Slash notation for multiple signals</p> </div> <ul style="list-style-type: none"> • a) Represents a block of combinational logic with 3 inputs and 2 outputs.

A3.1.0 Combinational Logic Design



- The symbol CL inside the box indicates that it is implemented using **only combinational logic**.
- You can have many implementations of this combinational logic circuit that depends on area, speed, power, and design time.

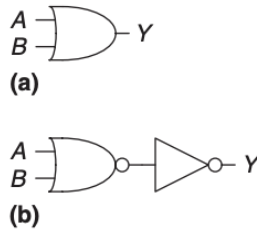


Figure 2.4 Two OR implementations

A3.1.2 Rules of Combinational Composition

A circuit is combinational if it consists of interconnected circuit elements such that

- Every circuit element is itself combinational.
- Every node of the circuit is either designated as an input to the circuit or connects to exactly one output terminal of a circuit element.
- The circuit contains no cyclic paths: every path through the circuit visits each circuit node at most once.

A3.2.0 Terminology for Boolean Equations

Term	Definition
Complement	The complement of a variable A is its inverse \bar{A} .
Literal	The variable or its complement is called a literal . <ul style="list-style-type: none"> • Eg. A, \bar{A}, B, and \bar{B}.
True Form	We call A the true form of the variable. <ul style="list-style-type: none"> • TRUE FORM DOESN'T MEAN THAT A IS TRUE. <ul style="list-style-type: none"> ◦ It just means A doesn't have a line over it.
Complementary Form	We call \bar{A} the complementary form .
Product/Implicant	The AND of one or more literals is called a product or an implicant . <ul style="list-style-type: none"> • Eg. $\bar{A}B$, $AB\bar{C}$, and B are all implicants for a function of three variables.

Minterm	<p>A minterm is a product involving all of the inputs to the function.</p> <ul style="list-style-type: none"> • Eg. $\overline{A}BC$ is a minterm for a function of A, B, and C. $\overline{A}B$ is not because it doesn't involve C. • If the input is 0, then \overline{x}_i (complementary form). If the input is 1, then x_i (true form). <ul style="list-style-type: none"> ○ This is because for SOP, we want to sum the minterms that give the output 1. So if input is 0, we want to reverse it (ie. use the complementary form). If the input is 1, we want to keep it (ie. true form). This ensures that the product (AND) is TRUE or 1.
Sum	The OR of one or more literals is called a sum .
Maxterm	<p>A maxterm is a sum involving all of the inputs to the function.</p> <ul style="list-style-type: none"> • Eg. $A + \overline{B} + C$ is a maxterm for a function of A, B, and C. • If the input is 1, then \overline{x}_i (complementary form). If the input is 0, then x_i (true form). <ul style="list-style-type: none"> ○ This is because for POS, we want to sum the maxterms that give the output 0. So if input is 1, we want to reverse it (ie. use the complementary form) to make 0. If the input is 0, we want to keep it (ie. true form). This ensures that the sum (OR) is 0 for each maxterm, ensuring 0 output.
Order of Operations	<p>The order from highest precedence to lowest:</p> <p style="text-align: center;">NOT, AND, OR</p> <ul style="list-style-type: none"> • Eg. $\overline{A}B + BCD = ((\overline{A})B) + (BC(\overline{D}))$
Minimized	<p>We define an equation in sum-of-products form to be minimized if it uses the fewest possible implicants.</p> <ul style="list-style-type: none"> • If there are several equations with the same number of implicants, the minimal one is the one with the fewest literals.
Prime Implicant	A prime implicant if it cannot be combined with any other implicants in the equation to form a new implicant with fewer literals.

- The implicants in a minimal equation must all be prime implicants.

A3.2.1 Sum-of-Products Form Canonical Form

Each row in a truth table is associated with a minterm that is TRUE **for that row**.

We can write a Boolean equation for any truth table by **summing each of the minterms for which the output is TRUE**. It is the sum (OR) of products (ANDs forming minterms)

- **Convention:** We will sort the minterms in the same order that they appear in the truth table
- SOP canonical form can be written in **sigma notation** by writing the rows at which the output is TRUE:

○ Eg. $F(A, B) = \sum(m_1, m_3) = \sum(1, 3)$

A	B	Y	minterm	minterm name
0	0	0	$\bar{A} \bar{B}$	m_0
0	1	1	$\bar{A} B$	m_1
1	0	0	$A \bar{B}$	m_2
1	1	1	$A B$	m_3

Figure 2.9 Truth table with multiple TRUE minterms

When to use SOP?

When the output is TRUE on only a few rows of a truth table.

A3.2.2 Product-of-Sums Form

Each row of a truth table corresponds to a maxterm that is FALSE for that row.

- Eg. The first row of a two-input truth table is $(A+B)$ because $(A+B)$ is FALSE when $A = 0 = B$.

We can write a Boolean equation for any circuit directly from the truth table as **the AND of each of the maxterms for which the output is FALSE.**

- POS canonical form can be written in **pi notation** by writing the rows at which the output is FALSE.

When to use POS?

When the output is FALSE on only a few rows of a truth table.

A3.3.0 Axioms Boolean Algebra

	Axiom		Dual	Name
A1	$B = 0 \text{ if } B \neq 1$	A1'	$B = 1 \text{ if } B \neq 0$	Binary field
A2	$\bar{0} = 1$	A2'	$\bar{1} = 0$	NOT
A3	$0 \cdot 0 = 0$	A3'	$1 + 1 = 1$	AND/OR
A4	$1 \cdot 1 = 1$	A4'	$0 + 0 = 0$	AND/OR
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$	AND/OR

- The practical significance is that they teach us how to **simplify logic to produce smaller and less costly circuits.**

A3.3.1 Duality

Axioms and theorems of Boolean algebra obey the principle of duality. If the symbols 0 and 1 and the operators \cdot (AND) and $+$ (OR) are interchanged, the statement will still be correct.

- Use the symbol ($'$) to denote the **dual** of a statement.

A3.3.2 Theorems of One Variable GOOD

Theorem	Theorem	Dual
1	$0 \cdot 0 = 0$	$1 + 1 = 1$
2	$1 \cdot 1 = 1$	$0 + 0 = 0$
3	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$

4	If $x = 0, \bar{x} = 0$	If $x = 1, \bar{x} = 0$
5	$x \cdot 0 = 0$	$x + 1 = 1$
6	$x \cdot 1 = x$	$x + 0 = x$
7	$x \cdot x = x$	$x + x = x$
8	$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$
9	$\overline{\overline{x}} = x$	No dual
10: Commutative	$x \cdot y = y \cdot x$	$x + y = y + x$
11: Associative	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x + (y + z) = (x + y) + z$
12: Distributive	$x \cdot (y + z) = xy + xz$	$x + (yz) = (x + y)(x + z)$
13: Absorption	$x + x \cdot y = x$	$x \cdot (x + y) = x$
14: Combining	$xy + x\bar{y} = x$	$(x + y)(x + \bar{y}) = x$
15: De Morgan's Theorem	$\overline{xy} = \bar{x} + \bar{y}$	$\overline{(x + y)} = \bar{x} \bar{y}$
16	$x + \bar{x}y = x + y$	$x \cdot (\bar{x} + y) = xy$

A3.3.3 Theorems of Several Variables

Table 2.3 Boolean theorems of several variables

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6'	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7'	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8'	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9'	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	T10'	$(B + C) \bullet (B + \bar{C}) = B$	Combining
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D)$ $= (B \bullet C) + (\bar{B} \bullet D)$	T11'	$(B + C) \bullet (\bar{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\bar{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \dots}$ $= (\bar{B}_0 + \bar{B}_1 + \bar{B}_2 \dots)$	T12'	$\overline{B_0 + B_1 + B_2 \dots}$ $= (\bar{B}_0 \bullet \bar{B}_1 \bullet \bar{B}_2 \dots)$	De Morgan's Theorem

- When you are thinking about the intuition of these theorems, plug in B=0 or B=1 and do the logic math.
- This can be expanded to having more complex inputs (ie. $B = (A + \bar{D}C)$)
 - So look out for these parts.

De Morgan's Theorem

The theorem explains that the complement of the product of all the terms is equal to the sum of the complement of each term. Likewise, the complement of the sum of all the terms is equal to the product of the complement of each term.

- De Morgan's Theorem can make **each function have duals** that are logically equivalent and can be used interchangeably.
 - **Eg.** a NAND gate is equivalent to an OR gate with inverted inputs
 - **Eg.** NOR gate is equivalent to an AND gate with inverted inputs.

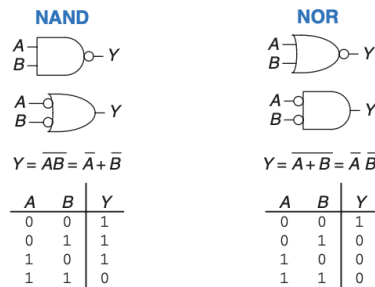


Figure 2.19 De Morgan equivalent gates

A3.3.4 Rules for Bubble Pushing:

- You can intuitively understand that the bubble is pushing through the gate that comes out the other side and flips the body of the gate.
1. Pushing bubbles backward (from the output) or forward (from the inputs) changes the body of the gate from AND to OR or vice versa.
 2. Pushing a bubble from the output back to the inputs puts bubbles on all gate inputs.
 3. Pushing bubbles on all gate inputs forward toward the output puts a bubble on the output.

A3.4 Perfect Induction

To prove a theorem is true, do a truth table with all the possible combinations for the LS and RS

A3.5 Simplifying Equations

Tips

Expanding an implicant (e.g., turning $AB = ABC + AB\overline{C}$) is sometimes useful in minimizing equations.

- By doing this, you can repeat one of the expanded minterms to be combined (shared) with another minterm.
- Idempotency allows us to duplicate terms.
- **For SOPs:** Combine terms using the relationship $PA + P\overline{A} = P$, where P may be any term by theorem 10.

A4 (2.4, 2.8, 5.2.1*): Example Logic Functions

✓ A4.0.0 Schematic Drawing

A **schematic** is a diagram of a digital circuit showing the elements and the wires that connect them.

✓ A4.0.1 Guidelines for Schematic Drawing and Truth Table (Last One)

- Inputs are on the left (or top) side of a schematic.
- Outputs are on the right (or bottom) side of a schematic.
- Whenever possible, gates should flow from left to right.
- Straight wires are better to use than wires with multiple corners.
- Wires always connect at a T junction.
- A dot where wires cross indicates a connection between the wires.
- Wires crossing without a dot make no connection.

✓ A4.0.2 Programmable Logic Array (PLA)

The inverters, AND gates, and OR gates are arrayed in a systematic fashion.

✓ A4.0.3 Overloaded Symbol X

We use the **symbol X** to describe inputs that the output doesn't care about, which are negligible such as in a priority circuit. This reduces the number of rows in the table when some variables do not affect the output.

✓ A4.1.0 Definition of Multiplexer

Multiplexers (mux) choose an output from among several possible inputs, based on the value of a **select** signal (or **control** signal).

- Select lines are control signals.

✓ A4.1.1 Ways that Multiplexers Can Be Built

1. **Normal:** For a 2:1 multiplexer with two data inputs D_0 and D_1 , a select input S , and one output Y .

- a. The multiplexer chooses between the two data inputs, based on the select: if $S = 0$, $Y = D_0$, and if $S = 1$, $Y = D_1$.

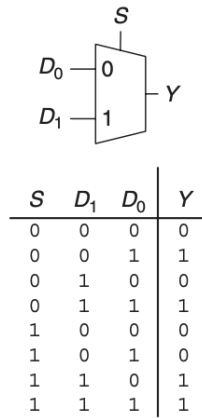


Figure 2.54 2:1 multiplexer symbol and truth table

2. **Tristate Buffers:** Multiplexers can be built from tristate buffers. The tristate enables are arranged such that exactly one tristate buffer is active at all times.

- a. When $S = 0$, tristate T_0 is enabled, allowing D_0 to flow to Y . When $S = 1$, tristate T_1 is enabled, allowing D_1 to flow to Y

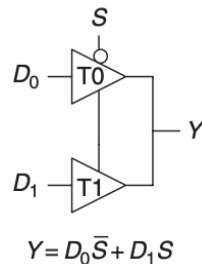
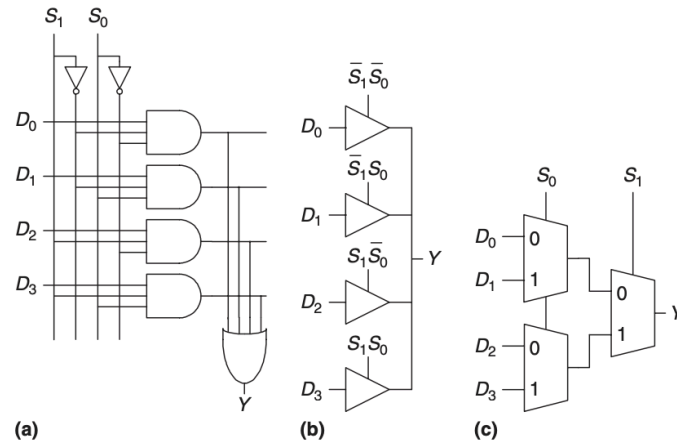


Figure 2.56 Multiplexer using tristate buffers

3. Sum-of-Products Logic
4. Decoder

✓ A4.1.2 Wider Multiplexers

An $N:1$ multiplexer needs $\log_2 N$ select lines.



- Wider multiplexers can be built by expanding the methods shown here of using SOP logic, tristates, or multiple 2:1 multiplexers.

✓ A4.1.3 Multiplexer Logic

Multiplexers can be used as **lookup tables** to perform logic functions. In general, a 2^N -input multiplexer can be programmed to perform any N-input logic function by applying 0's and 1's to the appropriate data inputs.

- **Eg.** 4:1 multiplexer implementation of two-input AND function by having the data inputs as 0 or 1.

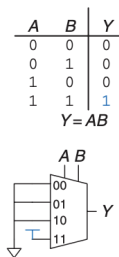


Figure 2.59 4:1 multiplexer implementation of two-input AND function

- Upside down triangle means 0
- Flat line means 1

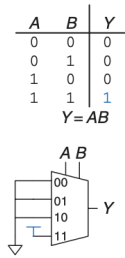
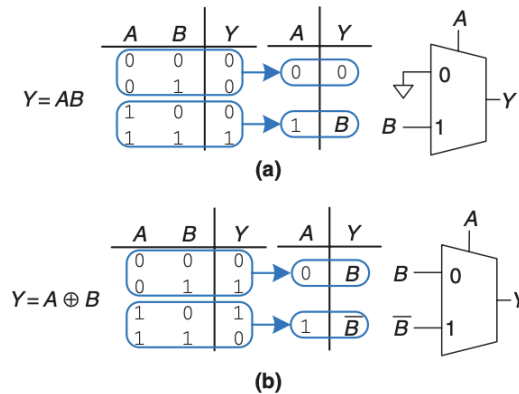


Figure 2.59 4:1 multiplexer implementation of two-input AND function

✓ A4.1.4 How to Cut the Multiplexer Size in Half?

We can cut the multiplexer size in half, using only a 2^{N-1} input multiplexer to perform any N-input logic function. The strategy is to **provide one of the literals, as well as 0's and 1's, to the multiplexer data inputs**.

- Eg.



- Use variable inputs (ie. one of the other variables as the input(s)) to express the output in terms of this variable

✓ A4.2.0 Decoders

A decoder has N inputs and 2^N outputs. It asserts exactly one of its outputs depending on the input combination.

- Outputs are called **one-hot**, because exactly one is “hot” (HIGH) at a given time.
- Eg. a 2:4 decoder.
 - When $A_1 = A_0 = 0$, $Y_0 = 1$.

- When $A_1 = 0, A_0 = 1, Y_1 = 1$
- When $A_1 = 1, A_0 = 0, Y_2 = 1$
- When $A_1 = 1, A_0 = 1, Y_3 = 1$

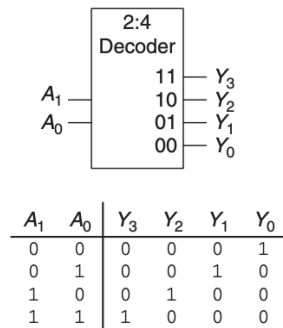


Figure 2.63 2:4 decoder

✓ A4.2.1 How to Build Decoders

In general, an $N:2^N$ (ie. Input:Output) decoder can be constructed from $2^N N$ – input AND gates that accept various combinations of true or complementary inputs.

- Each **output** in a decoder **represents a single minterm**.

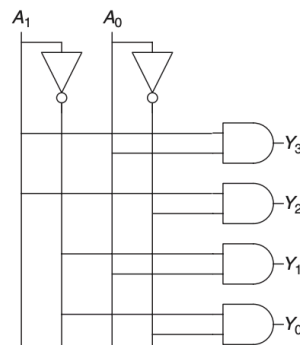


Figure 2.64 2:4 decoder implementation

✓ A4.2.2 Decoder Logic

Decoders can be combined with OR gates to build logic functions.

- Eg. Two-input XNOR function using a 2:4 decoder and a single OR gate.

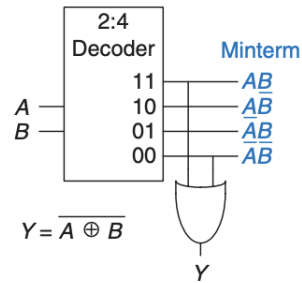


Figure 2.65 Logic function using decoder

- The function is built as the OR of all of the minterms in the function.
 - $Y = \overline{A}\overline{B} + AB = \overline{A \oplus B}$. This makes sense because XNOR outputs one if its not XOR.

✓ A4.2.3 Tips to Use Decoders to Build Logic

When using decoders to build logic, express functions as a truth table or in canonical SOP.

- An N-input function with M 1's in the truth table can be built with an $N: 2^N$ decoder and an M-input OR gate attached to all of the minterms containing 1's in the truth table

✓ A4.3.0 Types of Adders

Type	Definition
Half Adder	<ul style="list-style-type: none"> • The half adder has two inputs, A and B, and two outputs, S and C_{out}. • S is the sum of A and B. • If A and B are both 1, S is 2, which is indicated by a carry out C_{out}.

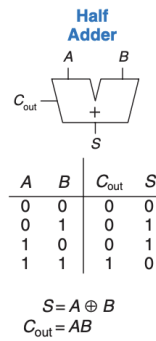


Figure 5.1 1-bit half adder

- Half adder lacks a C_{in} input to accept the C_{out} of the previous column.

Full Adder

A **full adder** accepts the carry in C_{in} , which solves the problem of half adders.

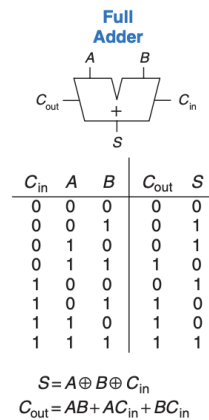


Figure 5.3 1-bit full adder

Carry Propagate Adder

An N-bit adder sums two N-bit inputs, A and B, and a carry in C_{in} to produce an N-bit result S and a carry out C_{out} .

- The carry out of one bit propagates into the next bit.

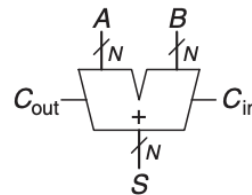


Figure 5.4 Carry propagate adder

- Three common CPA implementations are called **ripple-carry adders**, **carry-lookahead adders**, and **prefix adders**.

Ripple-Carry Adder

The simplest way to build an N-bit carry propagate adder is to chain together N full adders. The C_{out} of one stage acts as the C_{in} of the next stage.

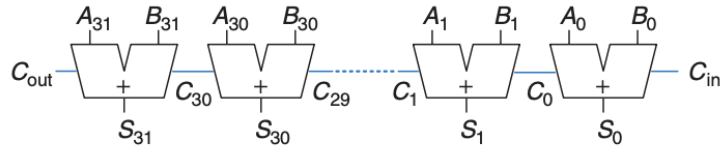


Figure 5.5 32-bit ripple-carry adder

- Disadvantage: When N is large, the adder is slow.

The carry **ripples** through the carry chain as they are all dependent on each other (ie. S_{31} depends on C_{30} , which depends on C_{29}, \dots , to C_{in}).

Delay of the Adder

$$t_{ripple} = N t_{FA}$$

- Where t_{FA} is the delay of a full adder.

A5 (2.6-2.7): Karnaugh Maps

✓ A5.1 Illegal Value - X

The symbol **X** indicates that the circuit node has an unknown or illegal value.

- This commonly happens if it is **being driven to both 0 and 1 at the same time**.

Contention:

A case where node Y is driven both HIGH and LOW, which is an error that must be avoided.

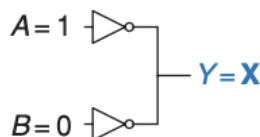


Figure 2.39 Circuit with contention

- **BE SURE NOT TO MIX UP the X (“don’t care”) in TRUTH TABLES WITH ILLEGAL VALUES.**

✓ A5.2.0 Floating Value - Z

The symbol **Z** indicates that a node is being driven neither HIGH nor LOW.

- The node is said to be **floating, high impedance, or high Z.**
- **Common Misconception:** A floating or undriven node is the same as a logic 0, however, a floating node can be 0, 1, or a voltage in-between.
- A floating node does not always mean there is an error in the circuit.
 - So long as some other circuit element does drive the node to a valid logic level when the value of the node is relevant to circuit operation.

Common Way to Produce a Floating Node:

If you forget to connect a voltage to a circuit input or to assume that an unconnected input is the same as an input with the value of 0.

✓ A5.2.1 Tristate Buffer

The **tristate buffer** has three possible output states: HIGH (1), LOW (0), and floating (Z) with input A, output Y, and **enable** E.

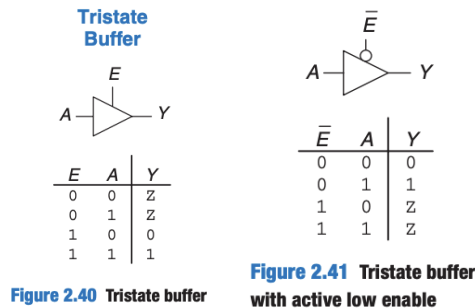
- When the enable is TRUE, the tristate buffer transfers the input value to the output. When the enable is FALSE, the output is allowed to float (Z).

Active High Enable vs. Active Low Enable Buffers

When the enable is HIGH (1), the buffer is enabled with **active high enable**.

When the enable is LOW (0), the buffer is enabled with **active low enable**.

- Show that the signal is active low by putting a bubble on its input wire and indicating with a bar, \overline{E} .



A5.3.1 Rules for Finding a Minimized Equation from a K-map

- Use the fewest circles necessary to cover all the 1's.
- All the squares in each circle must contain 1's.
- Each circle must span a rectangular block that is a power of 2 (i.e., 1, 2, or 4) squares in each direction.
- Each circle should be as large as possible.
- A circle may wrap around the edges of the K-map.
- A 1 in a K-map may be circled multiple times if doing so allows fewer circles to be used.
- Then read off the implicants circled

A5.3.2 Don't Cares

Don't cares also appear in truth table outputs where the output value is unimportant or the corresponding input combination can never happen. In a K-map, X's allow for even more logic minimization. They can be circled if they help cover the 1's with fewer or larger circles, but they do not have to be circled if they are not helpful.

A6 (4.1-4.2, 4.5.1-4.5.2, 5.6.2): Introduction to FPGAs and Verilog

✓ A6.0 Syntax

Term	Definition
AND	&

OR	
XOR	^
NOT	~
Module	<p>A block of hardware with inputs and outputs that starts off a code block</p> <p>Two General Styles for Describing Module Functionality:</p> <ol style="list-style-type: none"> 1. Behavioural models describe what a module does. 2. Structural models describe how a module is built from simpler pieces.
Endmodule	Ends the module.
Assign	Describes combinational logic
Logic	Logic is a type that indicates Boolean variables (ie. 1 or 0).
[X:Y]	If this is in the input and/or output section, this indicates the number of bits w.r.t to the variables afterwards
Internal Signals	These are signals that are in between the outputs and the inputs (think of them as an intermediary).

A6.1 Purposes of HDLs

1. Simulation

- a. Logic simulation is essential to test a system before it is built to test for bugs.

2. Synthesis

- a. Logic synthesis transforms HDL code into a **netlist** describing the hardware.
 - i. (e.g., the logic gates and the wires connecting them).

- ii. The logic synthesizer might perform optimizations to reduce the amount of hardware required.

A6.2 Tips

Think of your system in terms of blocks of combinational logic, registers, and finite state machines. Sketch these blocks on paper and show how they are connected before you start writing code.

0.1 Example Code

Mux 2 to 1

```
module mux2to1(input logic x, y, z, output logic f);  
    assign f = (~s & x) | (s & y);  
endmodule
```

Mux 2 to 1 with 2 bit inputs

```
module mux2to1__2bits(input logic [1: 0] x, y, input logic s, output logic [1: 0] f);  
    assign f[1] = (~s & x[1]) | (s & y[1]);  
    assign f[0] = (~s & x[0]) | (s & y[0]);  
endmodule
```

Half Adder

```
module HA(input logic a, b, output logic [1: 0] s);  
    assign s[1] = a & b;  
    assign s[0] = a ^ b;  
endmodule
```

Full Adder

```
module FA(input logic a, b, cin, output logic s, cout);  
    assign s = a ^ b ^ cin;  
    assign cout = (a & b) | (cin & a) | (cin & b);  
endmodule
```

endmodule

Hierarchical Verilog Code

```
module adder3(input logic [2: 0] A, B, input logic cin,  
               output logic [2: 0] S, output logic cout);  
    logic C1, C2; //internal signals  
    FA u0(A[0], B[0], cin, S[0], C1); // you can change the variable placement.  
    FA u1(A[1], B[1], C1, S[1], C2);  
    FA u2(A[2], B[2], C2, S[2], cout);  
endmodule
```

7 Seg Display

```
module seg7(input logic [0: 1] x, output logic [6: 0] H);  
    assign H[0] = ~x[1] & x[0];  
    assign H[1] = 1'b0; // "1" indicates # of bits (ie. h1 is 1 bit) c  
        // b indicates base(b = binary, d = dec. h = hexa.)  
        // "0" is value of h0  
  
    assign H[2] = x[1] & ~x[0]  
    assign H[3] = ~x[1] & x[0]  
    assign H[4] = x[0]  
    assign H[5] = x[1] | x[0]  
    assign H[6] = ~x[1]  
  
endmodule
```

A7 (2.9): Timing Analysis

Module B: Digital Storage Elements

B1: Latches 3.1-3.2.2

Bistable Element

An element with two stable states.

RS Latch - Cross Coupled NOR Gates

B2: Flip-Flops

B3: Counters and Registers

B4: Resets and Enables

Module C: Finite State Machines (FSMs)

Module D: Intro to Computer Organization & Assembly Language

Syntax

add
sub
addi

D1: Intro to Processors (6.1)

Terminology

Term	Definition
Architecture	The programmer's view of a computer <ul style="list-style-type: none">• Eg. RISC-V, x86
Instructions	The words in a computer's language

	<ul style="list-style-type: none"> • RISC-V represents each instruction as a 32-bit word.
Instruction Set	The computer's vocabulary.
Machine Language	<p>Instructions are encoded as binary numbers in a format called machine language.</p> <ul style="list-style-type: none"> • We use assembly language to turn machine language into a readable form. • Microprocessors read and execute machine language instructions.

D2: Signed Numbers (1.4.6)

D3: Instruction Set Architecture (6.2)

Terminology

Term	Definition
Assembly Language	The human-readable representation of the computer's native language.
Mnemonic	What operation to perform
Source Operands	The operands that have the operations performed on them (ie. after the destination operand)
Destination Operand	Result of the operation on the operands (ie. the first operand)
Operands	<p>Variables (ie. a, b, c)</p> <ul style="list-style-type: none"> • Can be stored in registers or memory, or be constants stored in the instruction itself. <ul style="list-style-type: none"> ◦ Operands stored in registers or constants are accessed quickly, but stored in memory are accessed slowly.
Register Set	The RISC-V architecture has 32 registers.

	<ul style="list-style-type: none"> This is stored in a small multi ported memory called a register file.
Immediates	<p>Their values are immediately available from the instruction and do not require a register or memory access</p> <ul style="list-style-type: none"> The immediate can be written in decimal, hexadecimal, or binary. <ul style="list-style-type: none"> Start with 0x for hexadecimal constants Start with 0b for binary constants Immediates are 12-bit two's complement numbers, so they are sign extended to 32 bits

Instructions

Term	High-Level Code	Definition in RISC-V Assembly Code
Add	a=b+c;	add a, b, c <ul style="list-style-type: none"> add is the mnemonic b and c are the source operands. a is the destination operand.
Sub	a=b-c;	sub a, b, c
Combination of Add and Sub	a=b+c-d;	add t, b, c # t = b + c sub a, t, d # a = t - d
Addi		<ul style="list-style-type: none"> addi adds an immediate to a register.
Lui		<ul style="list-style-type: none"> lui (ie. load upper immediate instruction) loads a 20-bit immediate into the most significant 20 bits of the instruction and places zeros in the least significant bits.

lw		<ul style="list-style-type: none"> ld (load word instruction) reads a data word from memory into a register.
----	--	---

Operands: Registers, Memory, and Constants

Term	High-Level Code	Definition in RISC-V Assembly Code
Register Operands	a = b + c;	# s0 = a, s1 = b, s2 = c add s0, s1, s2 <ul style="list-style-type: none"> Variables a,b,c arbitrarily placed in s0, s1, and s2. This adds the 32-bit values in s1 (b) and s2(b) and writes the 32-bit result to s0 (a).
Temporary Registers	a = b + c - d;	# s0 = a, s1 = b, s2 = c, s3 = d, t0 = t add t0, s1, s2 # t = b + c sub s0, t0, s3 # a = t - d

The Register Set

Table 6.1 RISC-V register set

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary registers
s0/fp	x8	Saved register/Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments/Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporary registers

- These registers are given a special name to indicate a register's conventional purpose.
 - s0-s11, and t0-t6 is for storing variables
 - ra, a0-a7 have special uses during function calls

Constants/Immediates

Term	High-Level Code	Definition in RISC-V Assembly Code
Immediate Operands	a = a + 4; b = a - 12;	# s0 = a, s1 = b addi s0, s0, 4 # a = a + 4

		addi s1, s0, -12 # b = a - 12
Initializing Values Using Immediates	i = 0; x = 2032; y = -78;	# s4 = i, s5 = x, s6 = y addi s4, zero, 0 # i = 0 addi s5, zero, 2032 # x = 2032 addi s6, zero, -78 # y = -78
32-Bit Constant Example	int a = 0xABCDE123;	lui s2, 0xABCDE # s2 = 0xABCDE000 addi s2, s2, 0x123 # s2 = 0xABCDE123
32-Bit Constant With a One in Bit 11	int a = 0xFEEDA987;	lui s2, 0xFEEDB # s2 = 0xFEEDB000 addi s2, s2, -1657 # s2 = 0xFEEDA987 <ul style="list-style-type: none"> 0xFEEDA is incremented by 1. and 0x987 is the 12-bit representation of -1657

Memory

- RISC-V uses a **byte-addressable memory**.

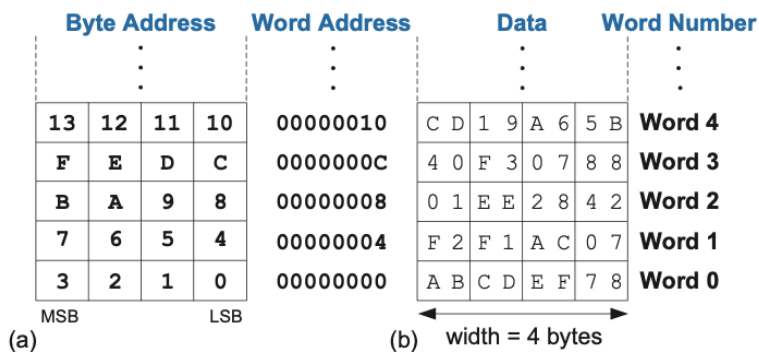


Figure 6.1 RISC-V byte-addressable memory showing: (a) byte address and (b) data

Term	High-Level Code	Definition in RISC-V Assembly Code
Reading Memory	a = mem[2];	# s7 = a; lw s7, 8(zero) # s7 = data at memory address (zero + 8)

Writing Memory	mem[5] = 42;	addi t3, zero, 42 # t3 = 42 sw t3, 20(zero) # data value at memory address 20 = 42.
---------------------------	--------------	---

D4: Basic Instruction Execution (6.2, 6.3.2-6.3.3)

Module E: Advanced Assembly Language

E1: Subroutines (6.3)

E2: I/O Devices (6.5, 9.1-9.2 **)

E3: Interrupts (6.6, 9.3)**