

Design of Digital Circuits

Lecture 6: Combinational Logic, Hardware Description Lang. & Verilog

Prof. Onur Mutlu

ETH Zurich

Spring 2018

9 March 2018

Required Lecture Video

- Why study computer architecture?
- Why is it important?
- Future Computing Architectures

- Required Assignment
 - Watch my inaugural lecture at ETH and understand it
 - <https://www.youtube.com/watch?v=kgiZISOcGFM>

- Optional Assignment
 - Write a summary of the lecture
 - What are your key takeaways?
 - What did you learn?
 - What did you like or dislike?

Required Readings

■ This week

□ Combinational Logic

- P&P Chapter 3 until 3.3 + H&H Chapter 2

□ **Hardware Description Languages and Verilog**

- **H&H Chapter 4 until 4.3 and 4.5**

■ Next week

□ Sequential Logic

- P&P Chapter 3.4 until end + H&H Chapter 3 in full

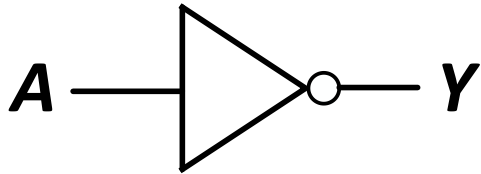
□ Hardware Description Languages and Verilog

- H&H Chapter 4 in full

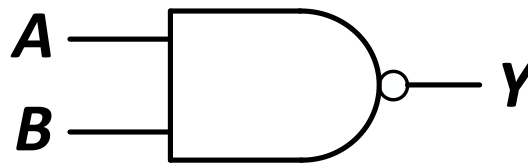
■ Make sure you are done with

- **P&P Chapters 1-3 + H&H Chapters 1-4**

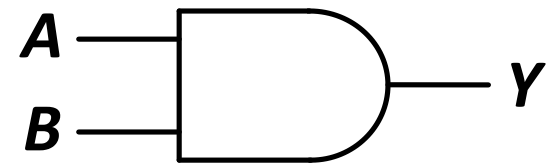
Recall: CMOS NOT, NAND, AND Gates



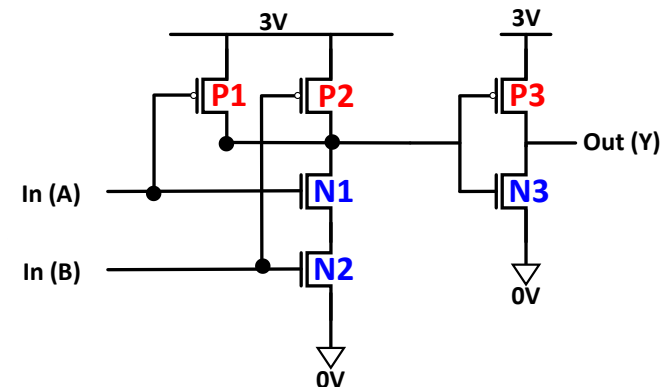
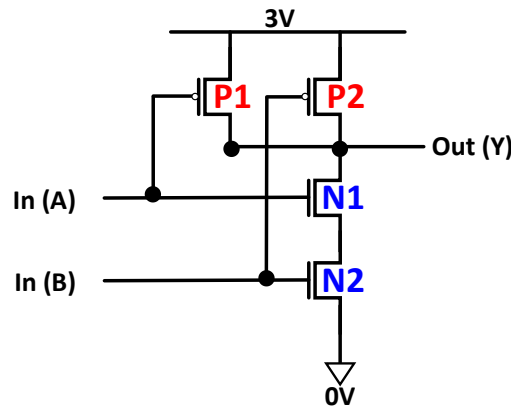
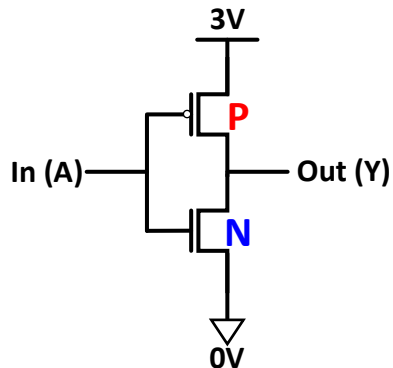
A	Y
0	1
1	0



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

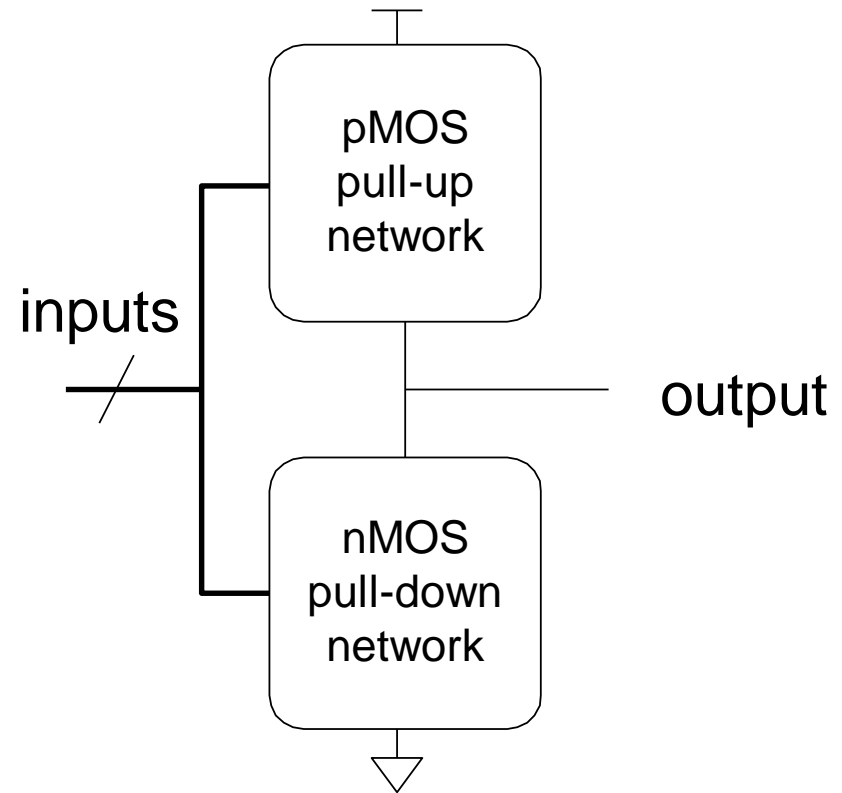


A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



Recall: General CMOS Gate Structure

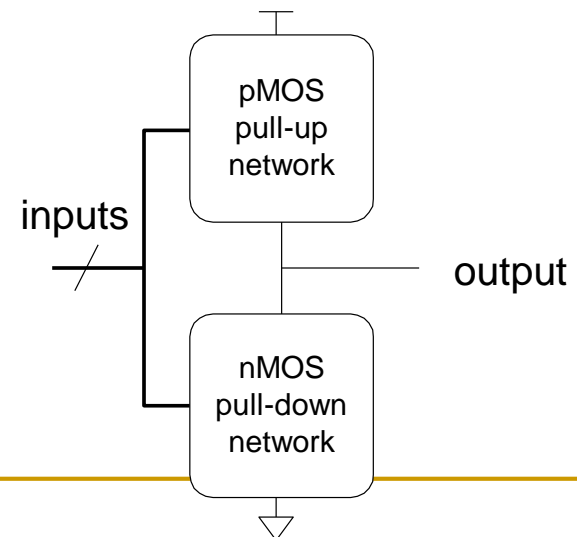
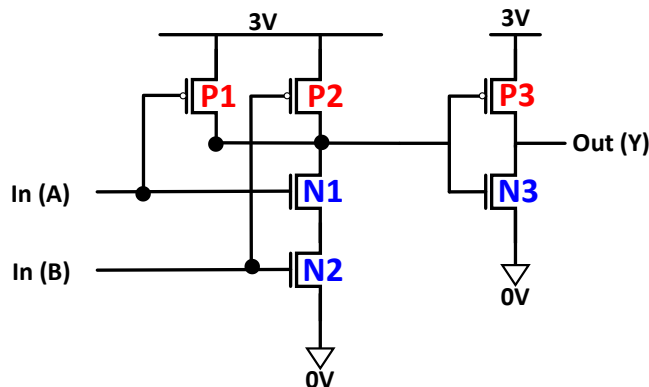
- The general form used to construct any inverting logic gate, such as: NOT, NAND, or NOR
 - The networks may consist of transistors in series or in parallel
 - When transistors are in parallel, the network is **ON** if one of the transistors is **ON**
 - When transistors are in series, the network is **ON** only if all transistors are **ON**



pMOS transistors are used for pull-up
nMOS transistors are used for pull-down

Digging Deeper: Why This Structure?

- MOS transistors are **not perfect** switches
- pMOS transistors pass 1's well but 0's poorly
- nMOS transistors pass 0's well but 1's poorly
- pMOS transistors are good at "pulling up" the output
- nMOS transistors are good at "pulling down" the output



Digging Deeper: Latency

- Which one is faster?
 - Transistors in series
 - Transistors in parallel
- Series connections are slower than parallel connections
 - More resistance on the wire
- How do you alleviate this latency?
 - See H&H Section 1.7.8 for an example: [pseudo-nMOS Logic](#)

Digging Deeper: Power Consumption

- Dynamic Power Consumption

- $C * V^2 * f$

- C = capacitance of the circuit (wires and gates)
 - V = supply voltage
 - f = charging frequency of the capacitor

- Static Power consumption

- $V * I_{\text{leakage}}$

- supply voltage * leakage current

- See more in H&H Chapter 1.8

Recall: DeMorgan's Law (Continued)

These are conversions between **different types of logic functions**
They can prove useful if you do not have **every type of gate**

$$A = \overline{(X + Y)} = \bar{X}\bar{Y}$$

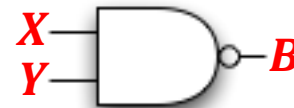
**NOR is equivalent to AND
with inputs complemented**



X	Y	$\overline{X + Y}$	\bar{X}	\bar{Y}	$\bar{X}\bar{Y}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0

$$B = \overline{(XY)} = \bar{X} + \bar{Y}$$

**NAND is equivalent to OR
with inputs complemented**



X	Y	\overline{XY}	\bar{X}	\bar{Y}	$\bar{X} + \bar{Y}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Using Boolean Equations to Represent a Logic Circuit

Sum of Products Form: Key Idea

- Assume we have the truth table of a Boolean Function
- How do we express the function in terms of the inputs in a **standard** manner?
- Idea: **Sum of Products** form
- Express the truth table as a two-level Boolean expression
 - that contains **all** input variable combinations that result in a 1 output
 - If ANY of the combinations of input variables that results in a 1 is TRUE, then the output is 1
 - $F = \text{OR of all input variable combinations that result in a 1}$

Some Definitions

- **Complement:** variable with a bar over it
 $\bar{A}, \bar{B}, \bar{C}$
- **Literal:** variable or its complement
 $A, \bar{A}, B, \bar{B}, C, \bar{C}$
- **Implicant:** product (AND) of literals
 $(A \cdot B \cdot \bar{C}), (\bar{A} \cdot C), (B \cdot \bar{C})$
- **Minterm:** product (AND) that includes **all** input variables
 $(A \cdot B \cdot \bar{C}), (\bar{A} \cdot \bar{B} \cdot C), (\bar{A} \cdot B \cdot \bar{C})$
- **Maxterm:** sum (OR) that includes **all** input variables
 $(A + \bar{B} + \bar{C}), (\bar{A} + B + \bar{C}), (A + B + \bar{C})$

Two-Level Canonical (Standard) Forms

- **Truth table** is the unique **signature** of a Boolean *function* ...
 - But, it is an expensive representation
- A Boolean function can have many alternative Boolean expressions
 - i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function)
- **Canonical** form: **standard form for a Boolean expression**
 - Provides a unique algebraic signature
 - If they all say the same thing, why do we care?
 - Different Boolean expressions lead to different gate realizations

Two-Level Canonical Forms

Sum of Products Form (SOP)

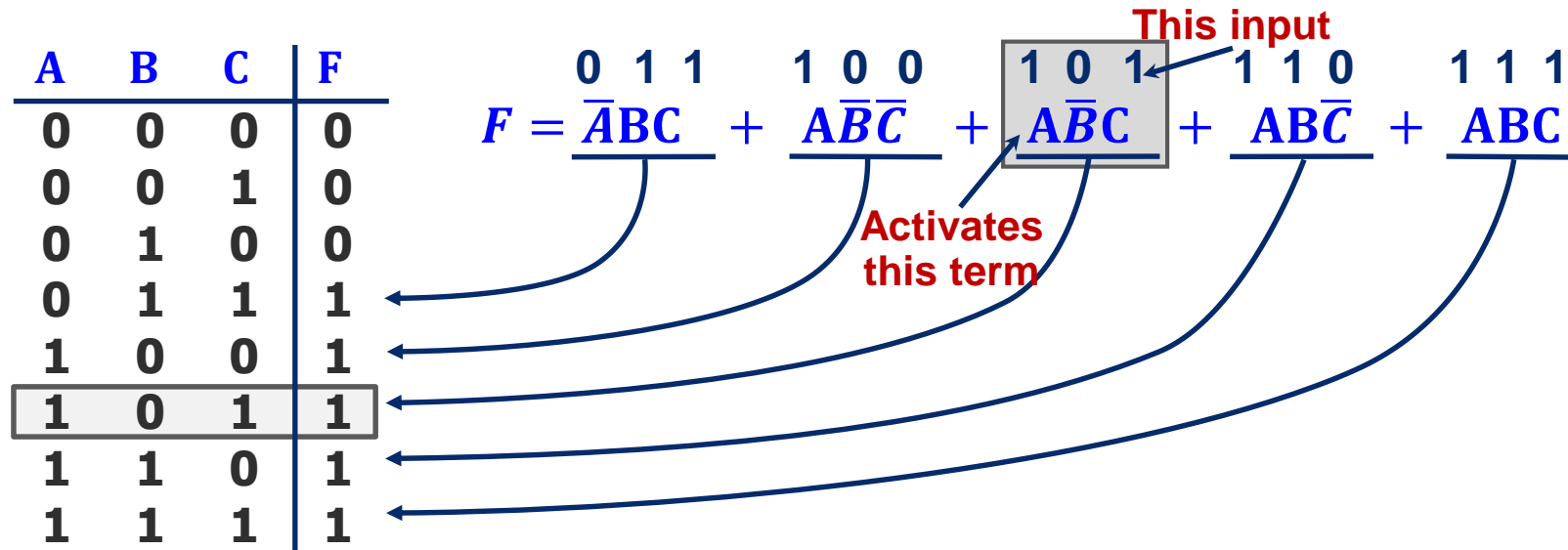
Also known as **disjunctive normal form** or **minterm expansion**

A	B	C	F		0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
0	0	0	0		$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
0	0	1	0						
0	1	0	0						
0	1	1	1	←					
1	0	0	1	←					
1	0	1	1	←					
1	1	0	1	←					
1	1	1	1	←					

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

All Boolean equations can be written in SOP form

SOP Form — Why Does It Work?



- Only the shaded product term — $\bar{A}\bar{B}C = 1 \cdot 0 \cdot 1$ — will be 1
- No other product terms will “turn on” — they will all be 0
- So if inputs A B C correspond to a product term in expression,
 - We get $0 + 0 + \dots + 1 + \dots + 0 + 0 = 1$ for output
- If inputs A B C do not correspond to any product term in expression
 - We get $0 + 0 + \dots + 0 = 0$ for output

Aside: Notation for SOP

- Standard “shorthand” notation
 - If we agree on the **order** of the variables in the rows of truth table...
 - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

100 = decimal 4 so this is minterm #4, or m4

111 = decimal 7 so this is minterm #7, or m7

f =

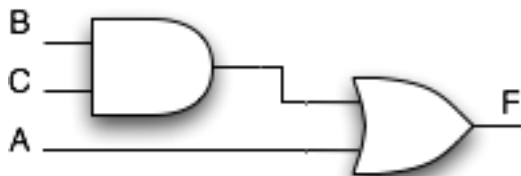
We can write this as a sum of products

Or, we can use a summation notation

Canonical SOP Forms

A	B	C	minterms
0	0	0	$\overline{A}\overline{B}\overline{C} = m_0$
0	0	1	$\overline{A}\overline{B}C = m_1$
0	1	0	$\overline{A}B\overline{C} = m_2$
0	1	1	$\overline{A}BC = m_3$
1	0	0	$A\overline{B}\overline{C} = m_4$
1	0	1	$A\overline{B}C = m_5$
1	1	0	$AB\overline{C} = m_6$
1	1	1	$ABC = m_7$

Shorthand Notation for
Minterms of 3 Variables



2-Level AND/OR
Realization

F in canonical form:

$$F(A,B,C) = \sum m(3,4,5,6,7) \\ = m_3 + m_4 + m_5 + m_6 + m_7$$

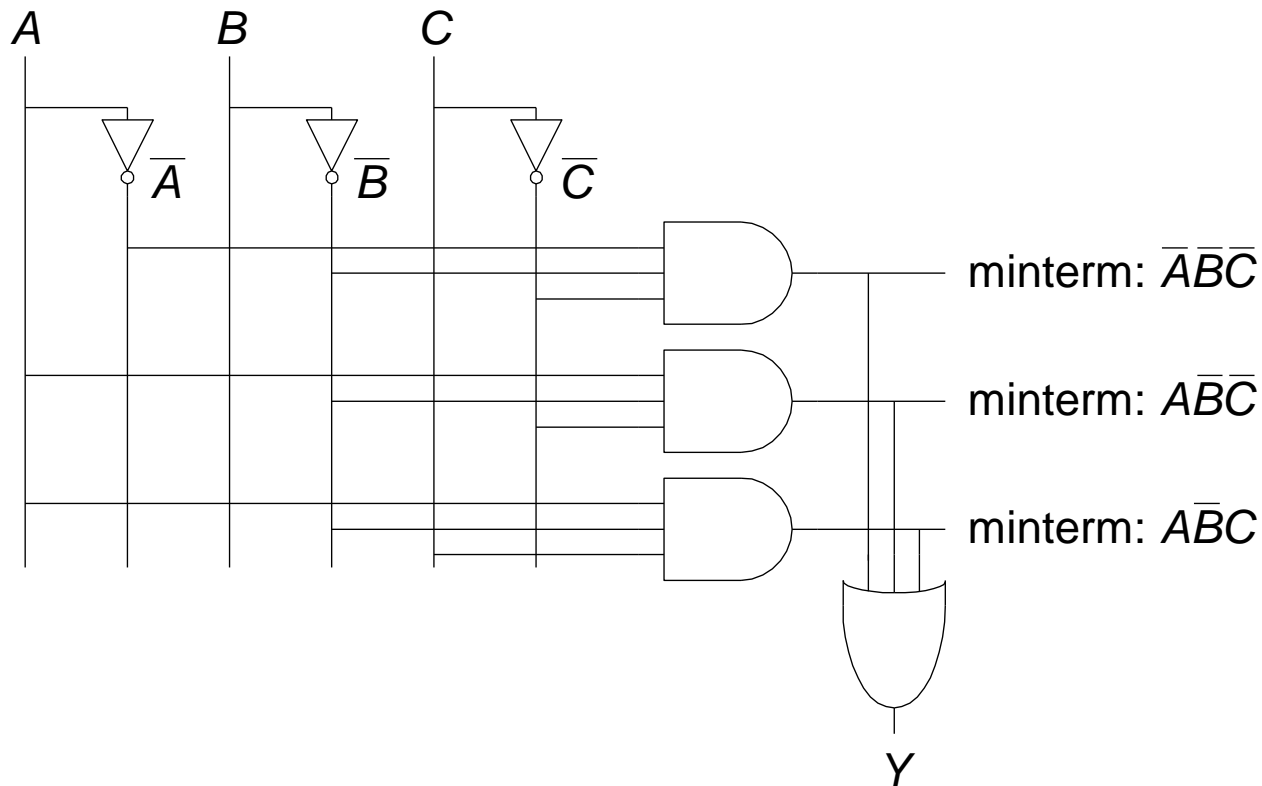
$F =$

canonical form \neq minimal form

F

From Logic to Gates

- **SOP (sum-of-products) leads to two-level logic**
- Example: $Y = (\bar{A} \cdot \bar{B} \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot \bar{B} \cdot C)$



Alternative Canonical Form: POS

We can have another form of representation

DeMorgan of SOP of \bar{F}

A product of sums (POS)

$$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$$

Diagram showing the structure of the POS expression. Arrows point from the words "products" and "sums" to the individual sum terms in the expression.

Each sum term represents one of the "zeros" of the function

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Diagram illustrating the POS expression $F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$ and its evaluation for the input $A=0, B=1, C=0$.

The expression is shown with the inputs substituted: $F = (0 + 0 + 0)(0 + 0 + \bar{1})(0 + \bar{1} + 0)$. The third sum term, $(0 + \bar{1} + 0)$, is shaded and labeled "Activates this term".

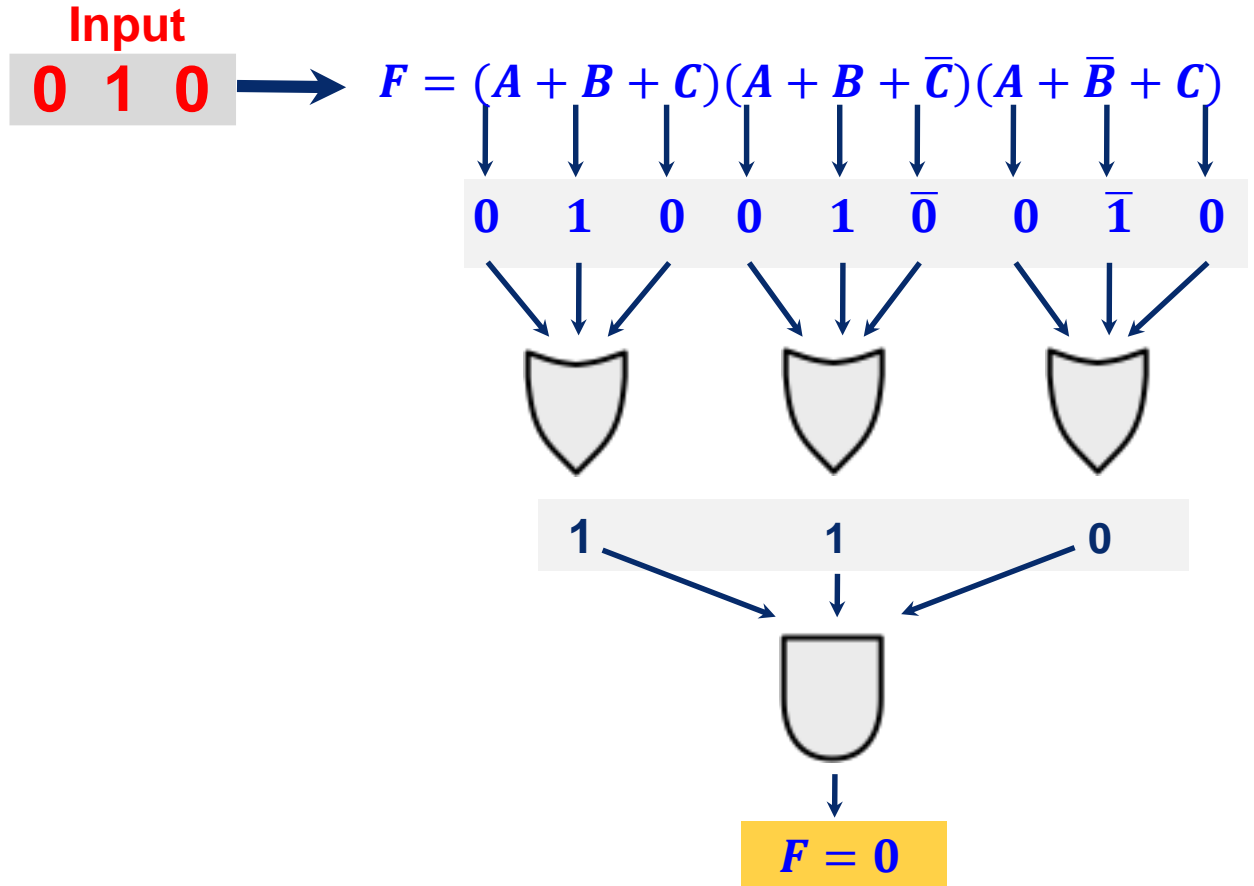
For the given input, only the shaded sum term will equal 0

$$A + \bar{B} + C = 0 + \bar{1} + 0$$

Anything ANDed with 0 is 0; Output F will be 0

Consider $A=0, B=1, C=0$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is $F = 0$

POS: How to Write It

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$

$A \quad \bar{B} \quad C$

$A + \bar{B} + C$

Maxterm form:

1. Find truth table rows where F is 0
2. 0 in input col → true literal
3. 1 in input col → complemented literal
4. OR the literals to get a Maxterm
5. AND together all the Maxterms

Or just remember, POS of F is the same as the DeMorgan of SOP of \bar{F} !!

Canonical POS Forms

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

$$F = (A + B + C)(A + B + \bar{C})(A + \bar{B} + C)$$

$$\prod M(0, 1, 2)$$

A	B	C	Maxterms
0	0	0	$A + B + C = M0$
0	0	1	$A + B + \bar{C} = M1$
0	1	0	$A + \bar{B} + C = M2$
0	1	1	$A + \bar{B} + \bar{C} = M3$
1	0	0	$\bar{A} + B + C = M4$
1	0	1	$\bar{A} + B + \bar{C} = M5$
1	1	0	$\bar{A} + \bar{B} + C = M6$
1	1	1	$\bar{A} + \bar{B} + \bar{C} = M7$

Maxterm shorthand notation
for a function of three variables

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Note that you
form the
maxterms around
the “zeros” of the
function

This is **not** the
complement of
the function!

Useful Conversions

1. **Minterm to Maxterm conversion:**

rewrite minterm shorthand using maxterm shorthand
replace minterm indices with the indices not already used

E.g., $F(A, B, C) = \sum m(3, 4, 5, 6, 7) = \prod M(0, 1, 2)$

2. **Maxterm to Minterm conversion:**

rewrite maxterm shorthand using minterm shorthand
replace maxterm indices with the indices not already used

E.g., $F(A, B, C) = \prod M(0, 1, 2) = \sum m(3, 4, 5, 6, 7)$

3. **Expansion of F to expansion of \bar{F} :**

$$\begin{array}{ll} \text{E. g., } F(A, B, C) = \sum m(3, 4, 5, 6, 7) & \longrightarrow \bar{F}(A, B, C) = \sum m(0, 1, 2) \\ = \prod M(0, 1, 2) & \longrightarrow = \prod M(3, 4, 5, 6, 7) \end{array}$$

4. **Minterm expansion of F to Maxterm expansion of \bar{F} :**

rewrite in Maxterm form, using the same indices as F

$$\begin{array}{ll} \text{E. g., } F(A, B, C) = \sum m(3, 4, 5, 6, 7) & \longrightarrow \bar{F}(A, B, C) = \prod M(3, 4, 5, 6, 7) \\ = \prod M(0, 1, 2) & \longrightarrow = \sum m(0, 1, 2) \end{array}$$

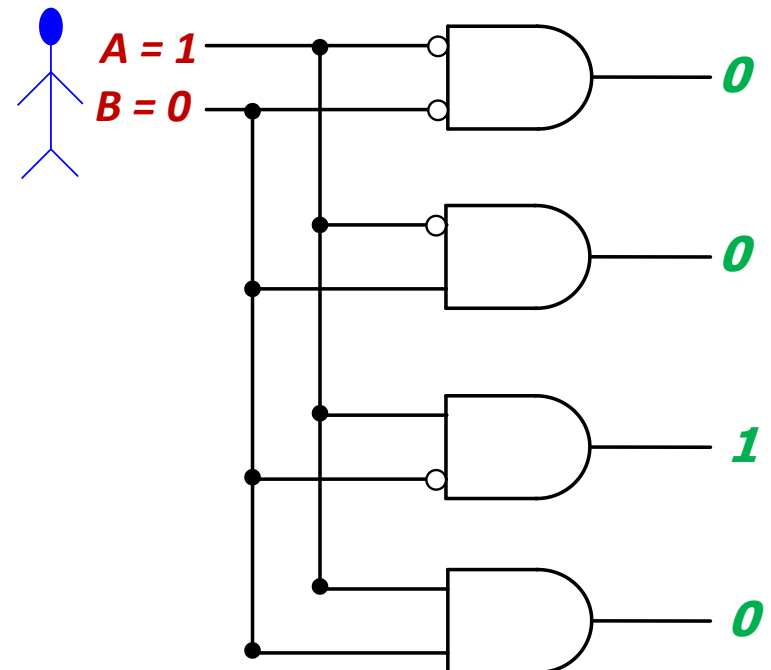
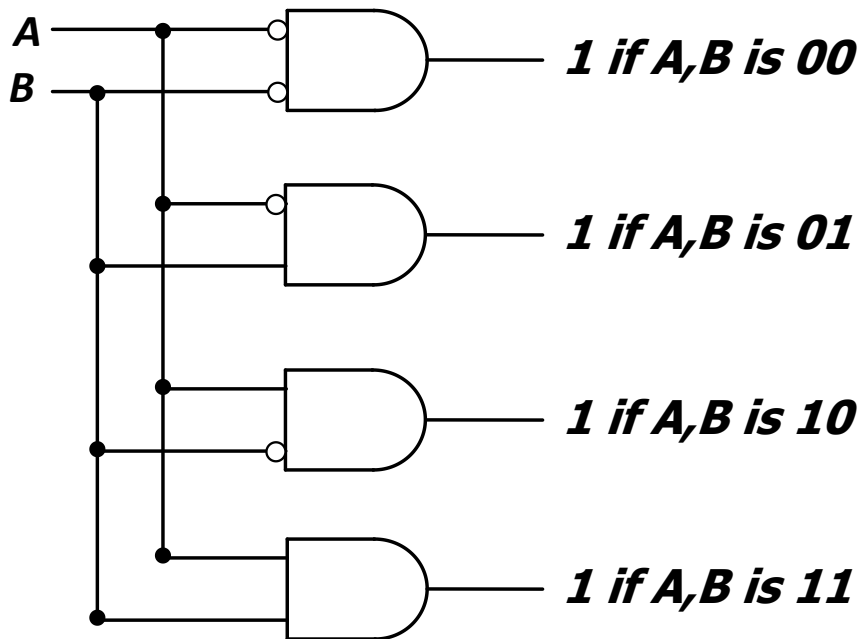
Combinational Building Blocks used in Modern Computers

Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more **complex systems**
- Hides the **unnecessary gate-level details** to emphasize the function of the building block
- We now look at:
 - Decoders
 - Multiplexers
 - Full adder
 - PLA

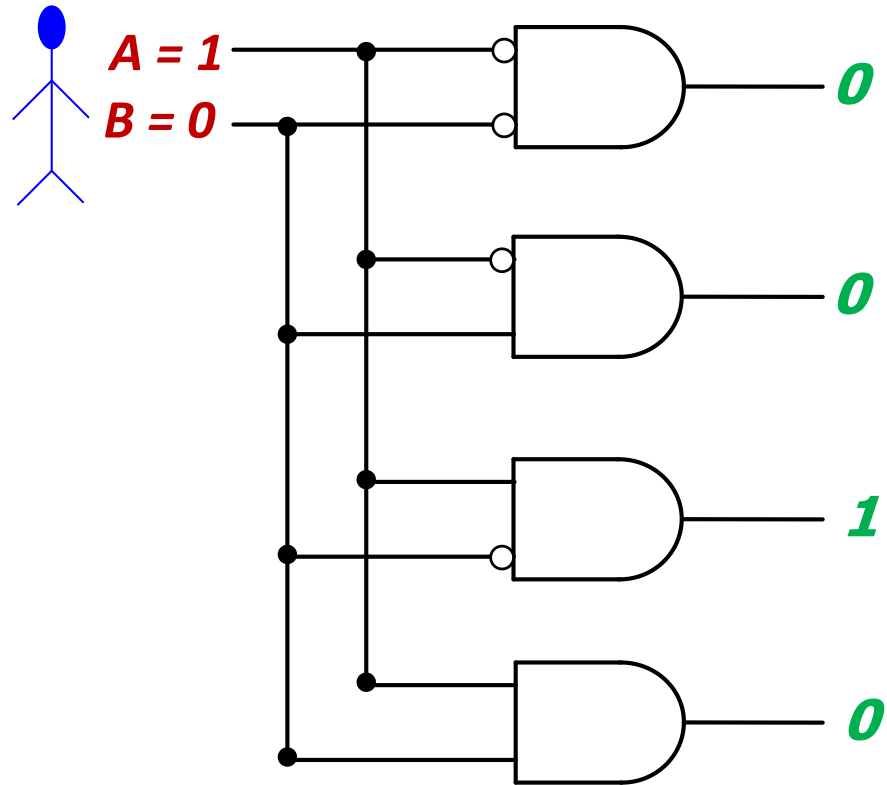
Decoder

- Exactly one of the outputs is 1 and all the rest are 0s
- n inputs and 2^n outputs
- The **one output** that is logically 1 is the output corresponding to the input **pattern** that the logic circuit is expected to detect



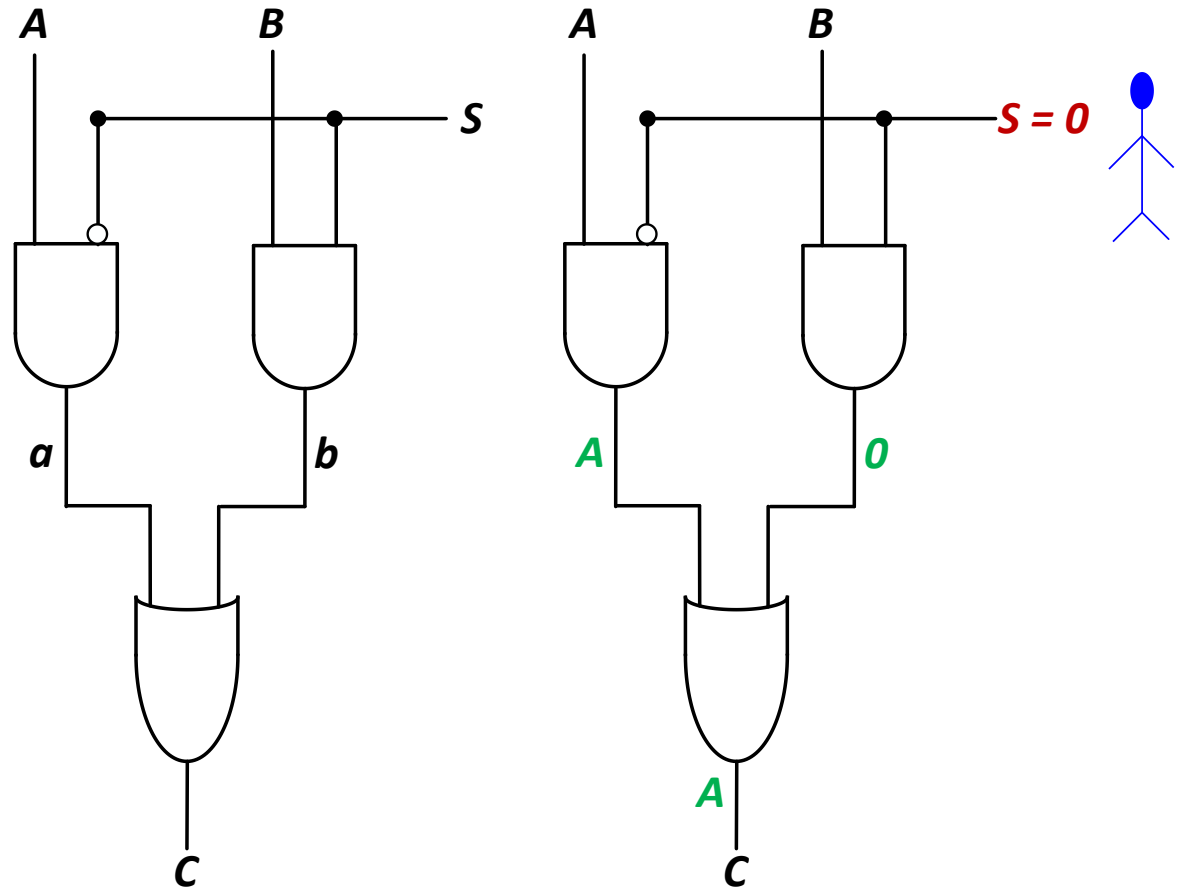
Decoder

- The decoder is useful in determining how to interpret a bit pattern
 - **It could be the address of a row in DRAM, that the processor intends to read from**
 - **It could be an instruction in the program and the processor has to decide what action to do! (based on *instruction opcode*)**



Multiplexer (MUX), or Selector

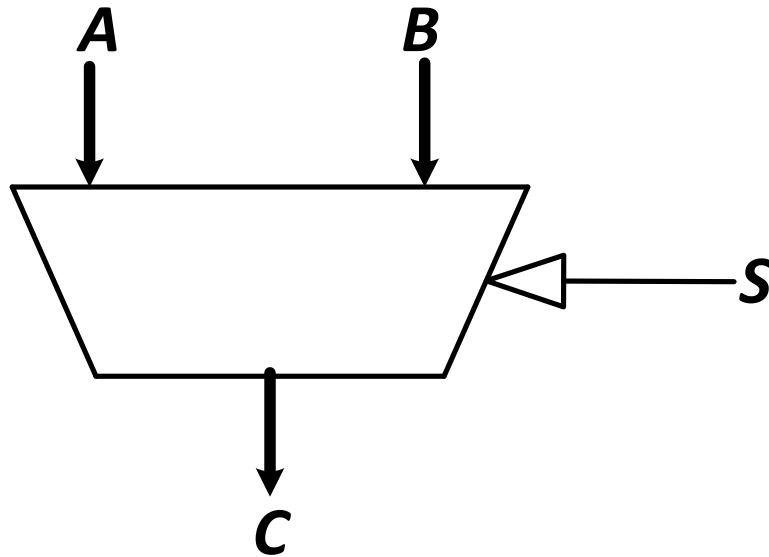
- **Selects** one of the N inputs to connect it to the output
- Needs $\log_2 N$ -bit control input
- 2:1 MUX



Multiplexer (MUX)

- The output C is always connected to either the input A or the input B
 - Output value depends on the value of the **select line S**

S	C
0	A
1	B



- **Your task:** Draw the schematic for an 8-input (8:1) MUX
 - Gate level: as a combination of basic AND, OR, NOT gates
 - Module level: As a combination of 2-input (2:1) MUXes

Full Adder (I)

■ Binary addition

- Similar to decimal addition
- From right to left
- One column at a time
- One sum and one carry bit

$$\begin{array}{r}
 a_{n-1}a_{n-2} \dots a_1a_0 \\
 b_{n-1}b_{n-2} \dots b_1b_0 \\
 \underline{C_n C_{n-1} \dots C_1} \\
 S_{n-1} \dots S_1S_0
 \end{array}$$

↓

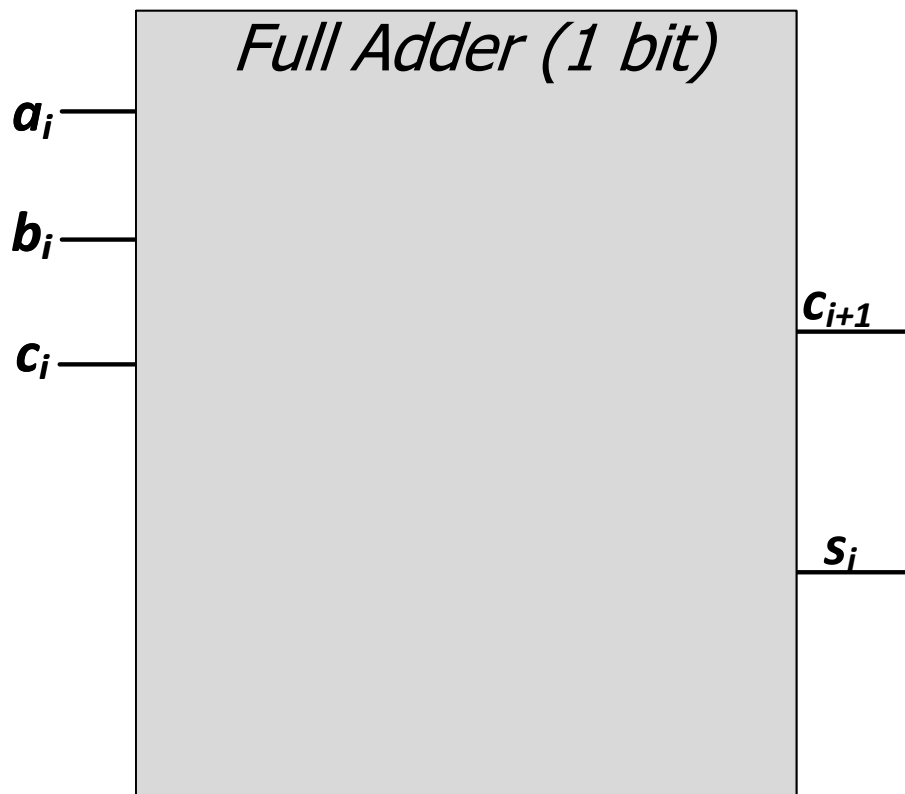
- Truth table of binary addition on **one column** of bits within two n-bit operands

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder (II)

■ Binary addition

- N 1-bit additions
- **SOP of 1-bit addition**



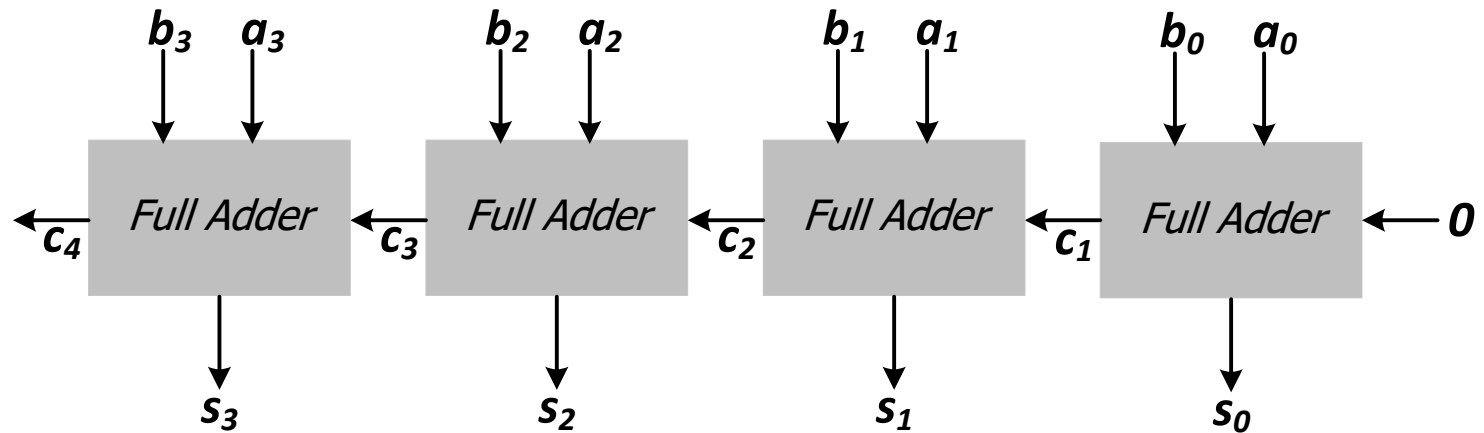
$$\begin{array}{r} a_{n-1}a_{n-2} \dots a_1a_0 \\ b_{n-1}b_{n-2} \dots b_1b_0 \\ \hline C_n C_{n-1} \dots C_1 \\ \hline S_{n-1} \dots S_1S_0 \end{array}$$

A vertical arrow points downwards from the right side of the addition, indicating the progression of bits from least significant to most significant.

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

4-Bit Adder from Full Adders

- Creating a **4-bit adder** out of 1-bit full adders
 - To add two 4-bit binary numbers A and B

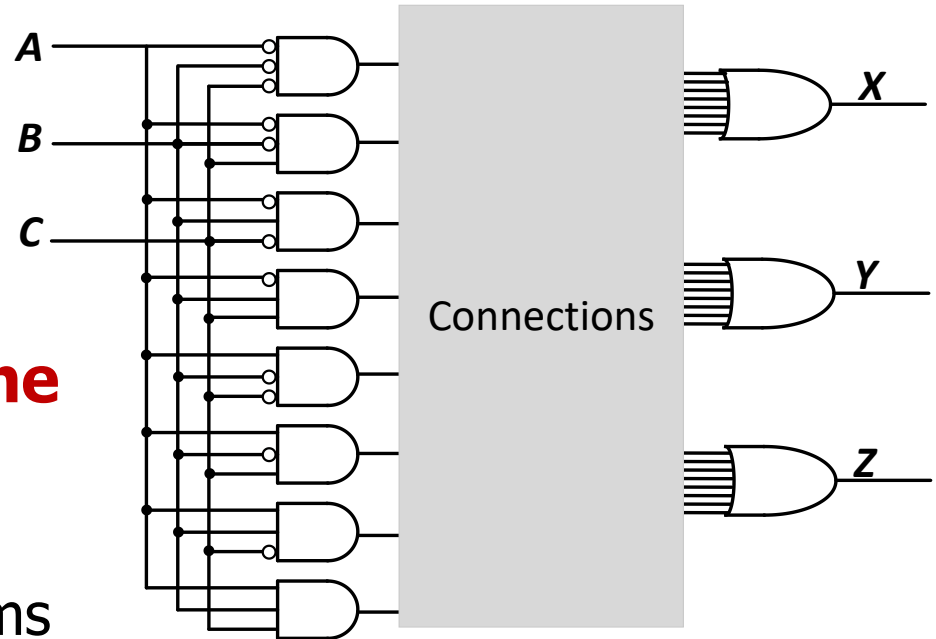


$$\begin{array}{rcccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline c_4 & c_3 & c_2 & c_1 & \\ \hline s_3 & s_2 & s_1 & s_0 & \end{array}$$

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & \\ \hline 0 & 1 & 0 & 0 & \end{array}$$

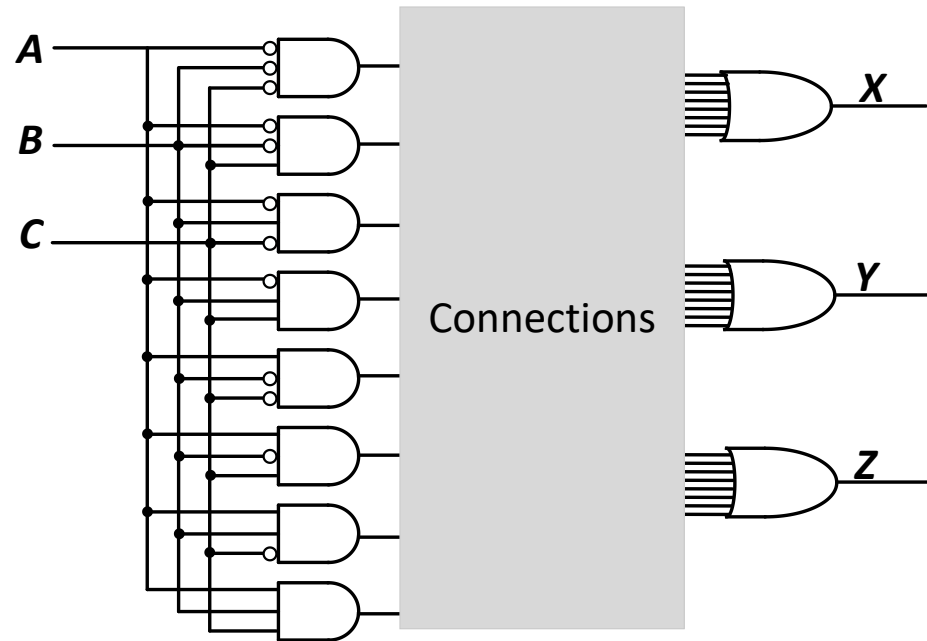
The Programmable Logic Array (PLA)

- The below logic structure is a very **common** building block for implementing any collection of logic functions one wishes to
- An **array** of AND gates followed by an **array** of OR gates
- **How do we determine the number of AND gates?**
 - **Remember SOP:** the number of possible minterms
 - For an n -input logic function, we need a PLA with 2^n n -input AND gates
- **How do we determine the number of OR gates?** The number of output columns in the truth table

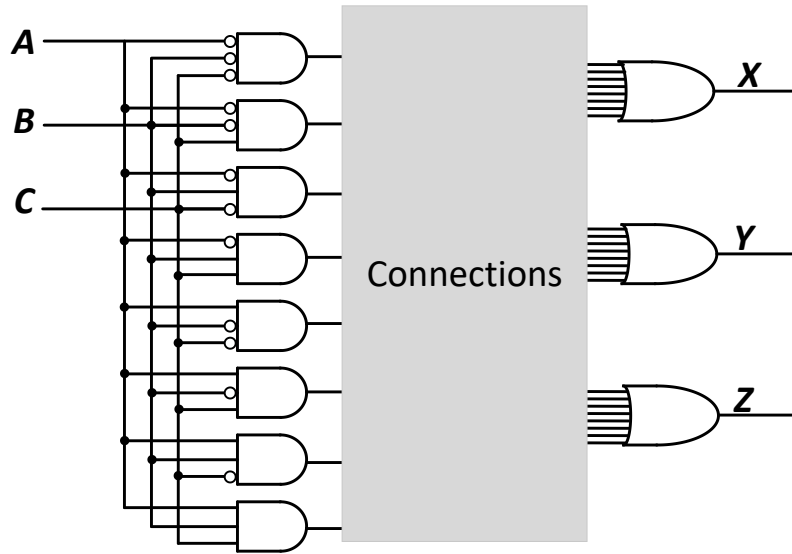


The Programmable Logic Array (PLA)

- How do we implement a logic function?
 - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
 - This is a simple programmable logic
- **Programming a PLA:** we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function
- Have you seen any other type of programmable logic?
 - Yes! An FPGA...
 - An FPGA uses more advanced structures, as we saw in Lecture 3



Implementing a Full Adder Using a PLA

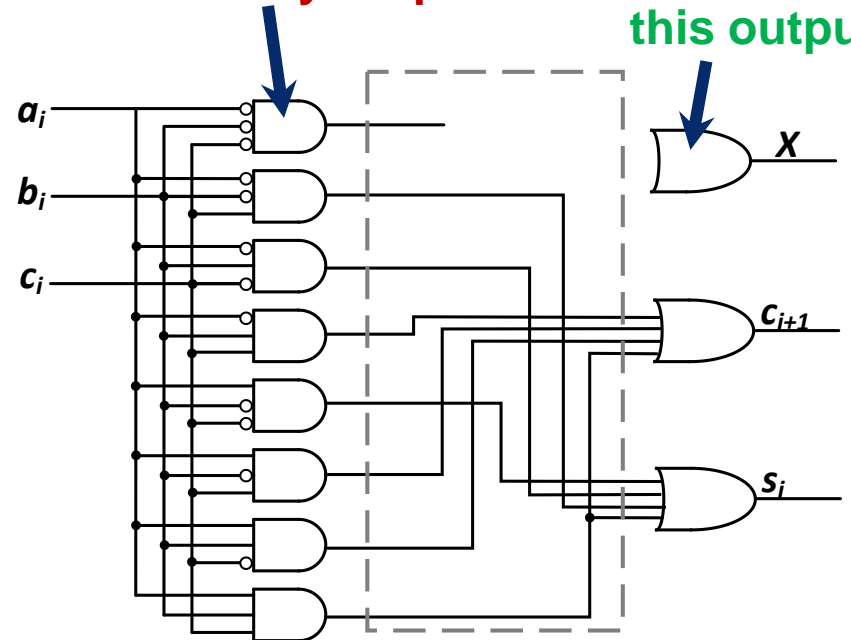


Truth table of a full adder

a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

This input should not be connected to any outputs

We do not need this output



Logical (Functional) Completeness

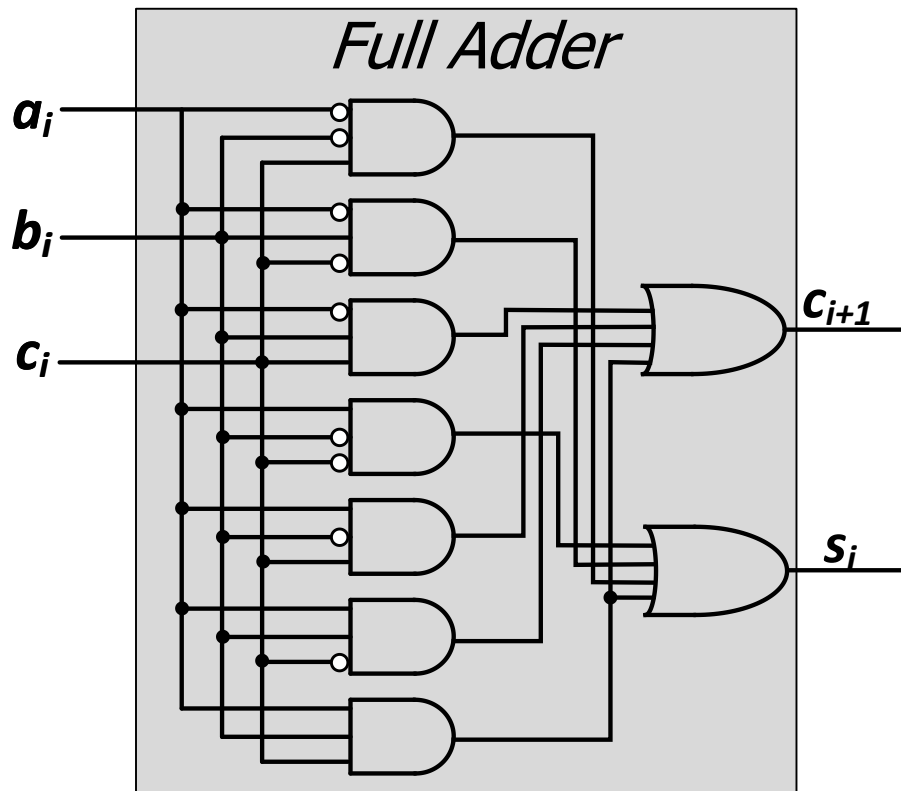
- **Any logic function** we wish to implement could be accomplished with PLA
 - PLA consists of **only** AND gates, OR gates, and inverters
 - We just have to program connections based on SOP of the intended logic function
- The set of gates {AND, OR, NOT} is **logically complete** because we can build a circuit to carry out the specification of **any truth table** we wish, without using any other kind of gate
- NAND is also logically complete. So is NOR.
 - **Your task:** Prove this.

More Combinational Building Blocks

- H&H Chapter 5
- Will be required reading soon.
- You will benefit greatly by reading the “combinational” parts of that chapter soon.
 - Sections 5.1 and 5.2

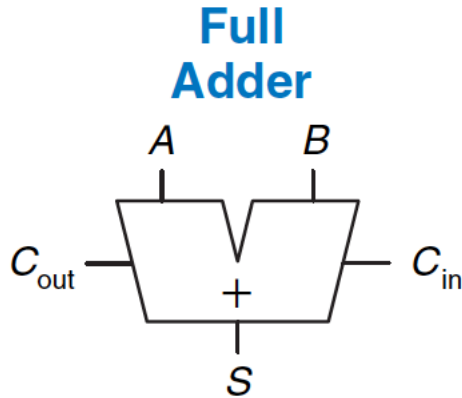
Logic Simplification: Karnaugh Maps (K-Maps)

Recall: Full Adder in SOP Form Logic



a_i	b_i	$carry_i$	$carry_{i+1}$	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Goal: Simplified Full Adder



$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

How do we simplify Boolean logic?

Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal

$$F = \sim A(A + B) + (B + AA)(A + \sim B)$$

- Can we reduce a given Boolean expression to an equivalent expression **with fewer terms**?

$$F = A + B$$

- The **goal** of logic simplification:
 - **Reduce** the number of gates/inputs
 - **Reduce** implementation cost

A basis for what the automated design tools are doing today

Logic Simplification

■ Systematic techniques for simplifications

- amenable to automation

Key Tool: The Uniting Theorem — $F = A\bar{B} + AB$

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

$$F = A\bar{B} + AB = A(\bar{B} + B) = A(1) = A$$

Essence of Simplification:

Find two element subsets of the ON-set where only one variable changes its value. This single varying variable *can be eliminated!*

→ *B is eliminated, A remains*

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

$$G = \bar{A}\bar{B} + A\bar{B} = (\bar{A} + A)\bar{B} = \bar{B}$$

B's values stay the same within the ON-set rows

A's values change within the ON-set rows

→ *A is eliminated, B remains*

Complex Cases

■ One example

$$Cout = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

■ Problem

- ❑ Easy to see how to apply Uniting Theorem...
- ❑ Hard to know if you applied it in all the right places...
- ❑ ...especially in a function of many more variables

■ Question

- ❑ Is there an easier way to potential simplifications?
- ❑ i.e., potential applications of Uniting Theorem...?

■ Answer

- ❑ Need an intrinsically *geometric* representation for Boolean $f()$
- ❑ Something we can draw, see...

Karnaugh Map

- Karnaugh Map (K-map) method
 - K-map is an alternative method of representing the **truth table** that helps **visualize adjacencies** in up to 6 dimensions
 - Physical adjacency \leftrightarrow Logical adjacency

2-variable K-map

$A \backslash B$	0	1
0	00	01
1	10	11

3-variable K-map

$A \backslash BC$	00	01	11	10
0	000	001	011	010
1	100	101	111	110

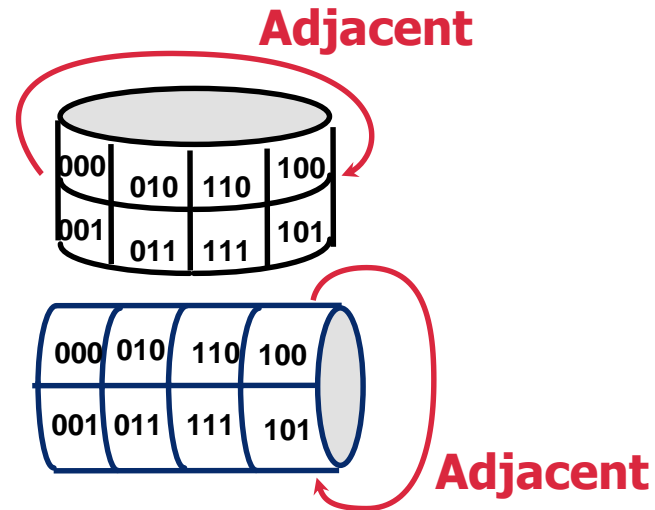
4-variable K-map

$AB \backslash CD$	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

Numbering Scheme: 00, 01, 11, 10 is called a “Gray Code” — only a *single bit changes* from code word to next code word

Karnaugh Map Methods

<i>A</i> \ <i>BC</i>	00	01	11	10
0	000	001	011	010
1	100	101	111	110



K-map adjacencies go “around the edges”
Wrap around from first to last column
Wrap around from top row to bottom row

K-map Cover - 4 Input Variables

$CD \backslash AB$	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

$$F(A, B, C, D) = \sum m(0, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$F = A + \bar{B}\bar{D} + B\bar{C}D$$

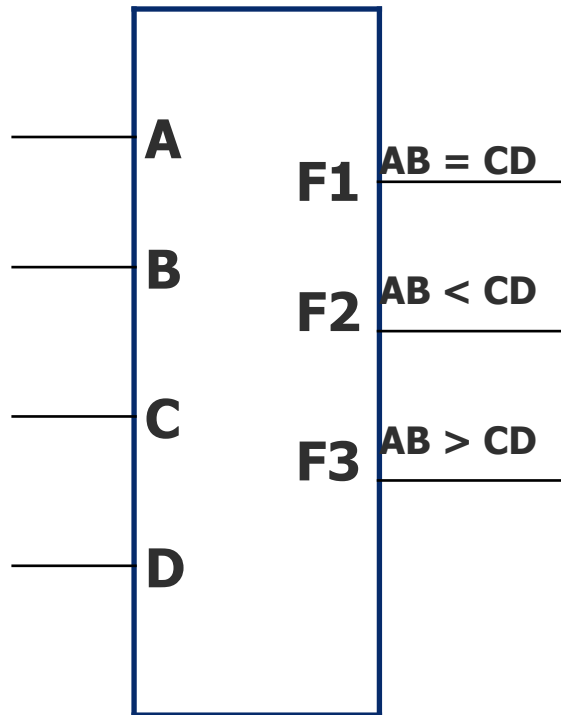
Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:

K-map Rules

- **What can be legally combined (circled) in the K-map?**
 - Rectangular groups of size 2^k for any integer k
 - Each cell has the same value (1, for now)
 - All values must be adjacent
 - Wrap-around edge is okay
- **How does a group become a term in an expression?**
 - Determine which literals are constant, and which vary across group
 - Eliminate varying literals, then AND the constant literals
 - constant 1 → use X , constant 0 → use \bar{X}
- **What is a good solution?**
 - Biggest groupings → eliminate more variables (literals) in each term
 - Fewest groupings → fewer terms (gates) all together
 - OR together all AND terms you create from individual groups

K-map Example: Two-bit Comparator



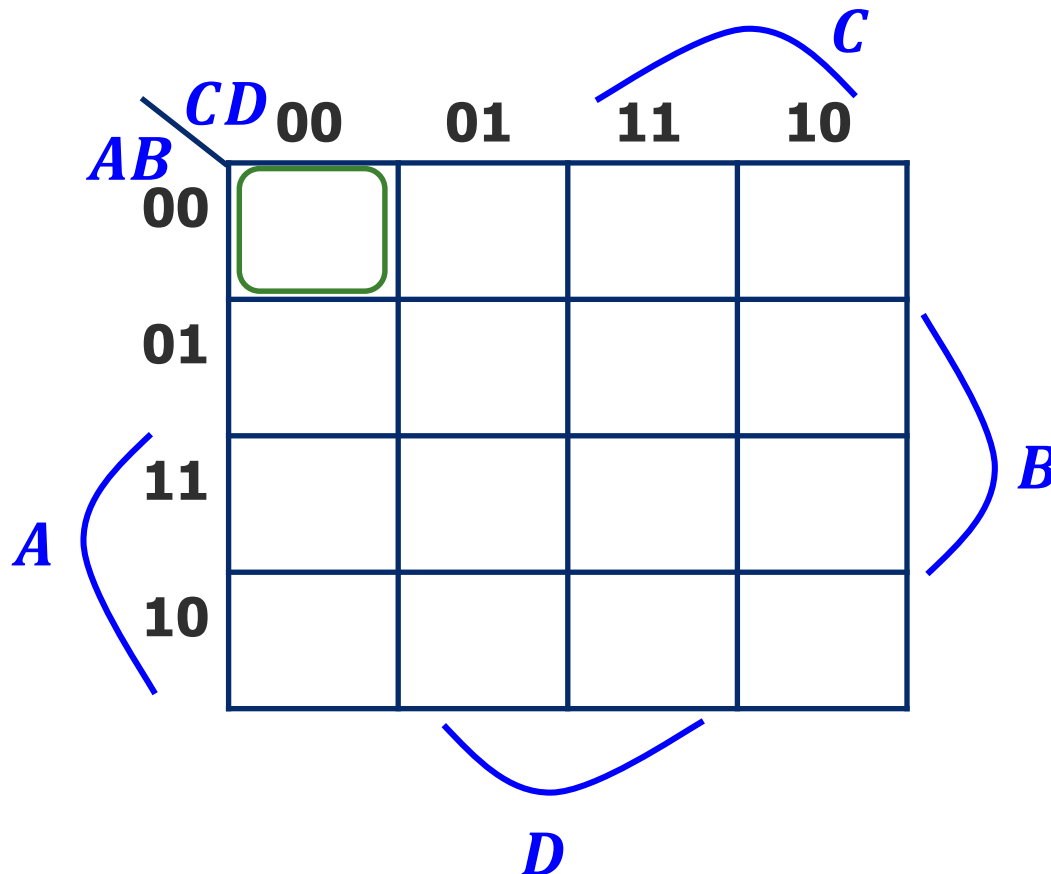
Design Approach:

**Write a 4-Variable K-map
for each of the 3
output functions**

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-map Example: Two-bit Comparator (2)

K-map for F1

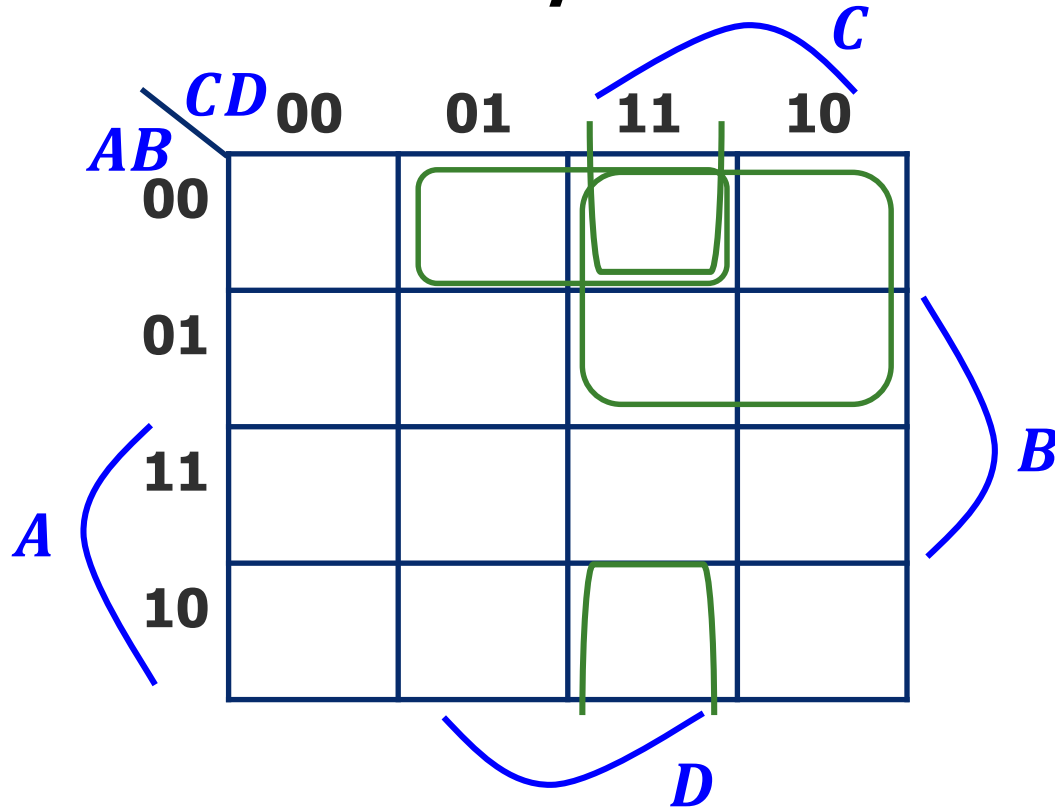


F1 =

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-map Example: Two-bit Comparator (3)

K-map for F2



F2 =

F3 = ? (Exercise for you)

A	B	C	D	F1	F2	F3
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

K-maps with “Don’t Care”

- **Don’t Care** really means *I don’t care what my circuit outputs if this appears as input*
 - You have an engineering choice to use DON’T CARE patterns intelligently as 1 or 0 to better **simplify** the circuit

A	B	C	D	F	G
...					
0	1	1	0	X	X
0	1	1	1		
1	0	0	0	X	X
1	0	0	1		
...					

I can pick 00, 01, 10, 11 independently of below

I can pick 00, 01, 10, 11 independently of above

Example: BCD Increment Function

- BCD (Binary Coded Decimal) digits
 - Encode decimal digits 0 - 9 with bit patterns 0000_2 — 1001_2
 - When **incremented**, the decimal sequence is 0, 1, ..., 8, 9, 0, 1

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

These input patterns **should never be encountered** in practice (hey -- it's a BCD number!)
So, associated output values are **"Don't Cares"**

K-map for BCD Increment Function

A B

+

W X

Z (without don't cares) =

Z (with don't cares) =

10	1		X	X
----	---	--	---	---

10			X	X
----	--	--	---	---

Y

		CD			
		00	01	11	10
AB	00		1		1
	01		1		1
	11	X	X	X	X
	10			X	X

Z

		CD			
		00	01	11	10
AB	00	1			1
	01	1			1
	11	X	X	X	X
	10	1		X	X

A *B* *C* *D*

K-map Summary

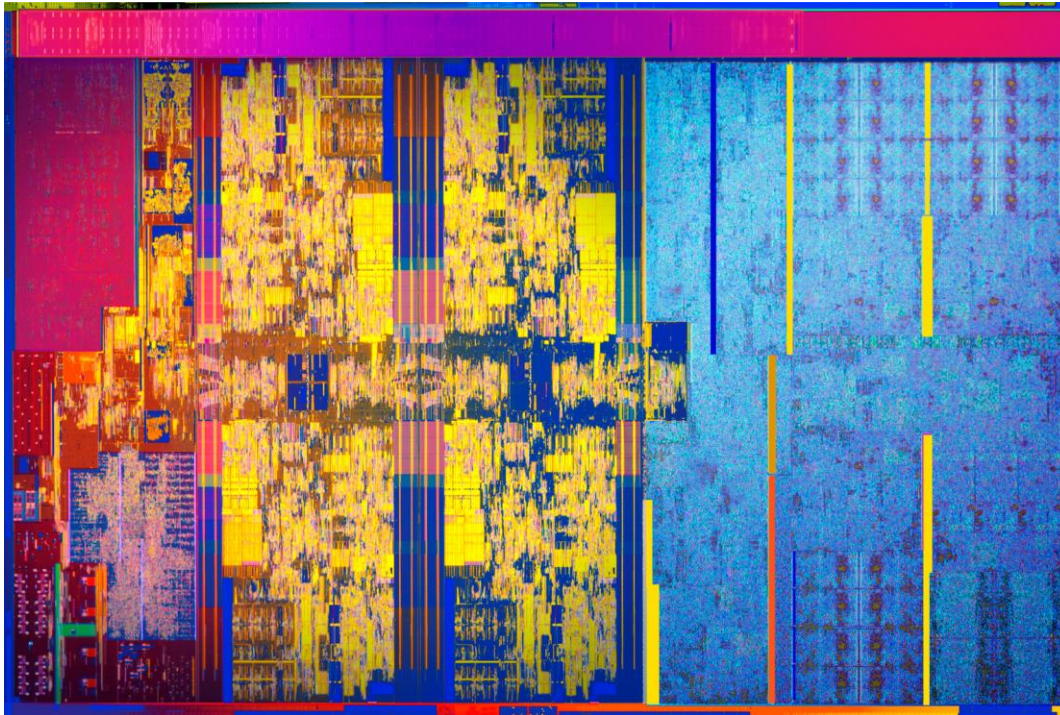
- Karnaugh maps as a formal systematic approach for logic simplification
- 2-, 3-, 4-variable K-maps
- K-maps with “Don’t Care” outputs

Hardware Description Languages & Verilog (Combinational Logic)

Agenda

- Implementing Combinational Logic
 - Hardware Description Languages
 - Hardware Design Methodologies
 - Verilog

2017: Intel Kaby Lake



https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake

- **64-bit processor**
- **4 cores, 8 threads**
- **14-19 stage pipeline**
- **3.9 GHz clock**
- **1.75B transistors**
- **In ~47 years, about 1,000,000-fold growth in transistor count and performance!**

How to Deal with This Complexity?

- Hardware Description Languages!
- A fact of life in computer engineering
 - Need to be able to **specify complex designs**
 - communicate with others in your design group
 - ... and to **simulate** their behavior
 - *yes, it's what I want to build*
 - ... and to **synthesize** (automatically design) portions of it
 - have an error-free path to implementation
- Hardware Description Languages
 - Many similarly featured **HDLs** (e.g., **Verilog**, VHDL, ...)
 - if you learn one, it is **not hard to learn** another
 - mapping between languages is typically **mechanical**, especially for the commonly used subset

Hardware Description Languages

- **Two well-known hardware description languages**
- **Verilog**
 - ❑ Developed in 1984 by Gateway Design Automation
 - ❑ Became an IEEE standard (1364) in 1995
 - ❑ More popular in US
- **VHDL (VHSIC Hardware Description Language)**
 - ❑ Developed in 1981 by the Department of Defense
 - ❑ Became an IEEE standard (1076) in 1987
 - ❑ More popular in Europe
- In this course we will use Verilog

Hardware Design Using Verilog

Hierarchical Design

- **Design hierarchy of modules is built using instantiation**

- ❑ Predefined “primitive” gates (AND, OR, ...)
- ❑ Simple modules are built by instantiating these gates (components like MUXes)
- ❑ Other modules are built by instantiating simple components, ...

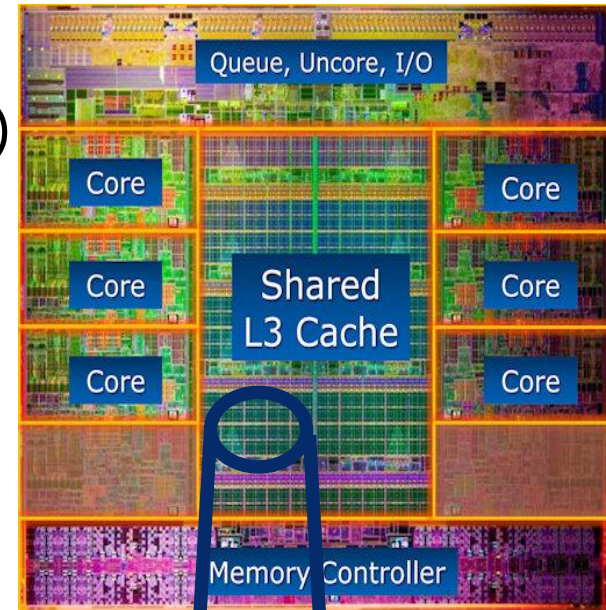
- **Hierarchy controls complexity**

- ❑ Analogous to the use of function abstraction in SW

- **Complexity is a BIG deal**

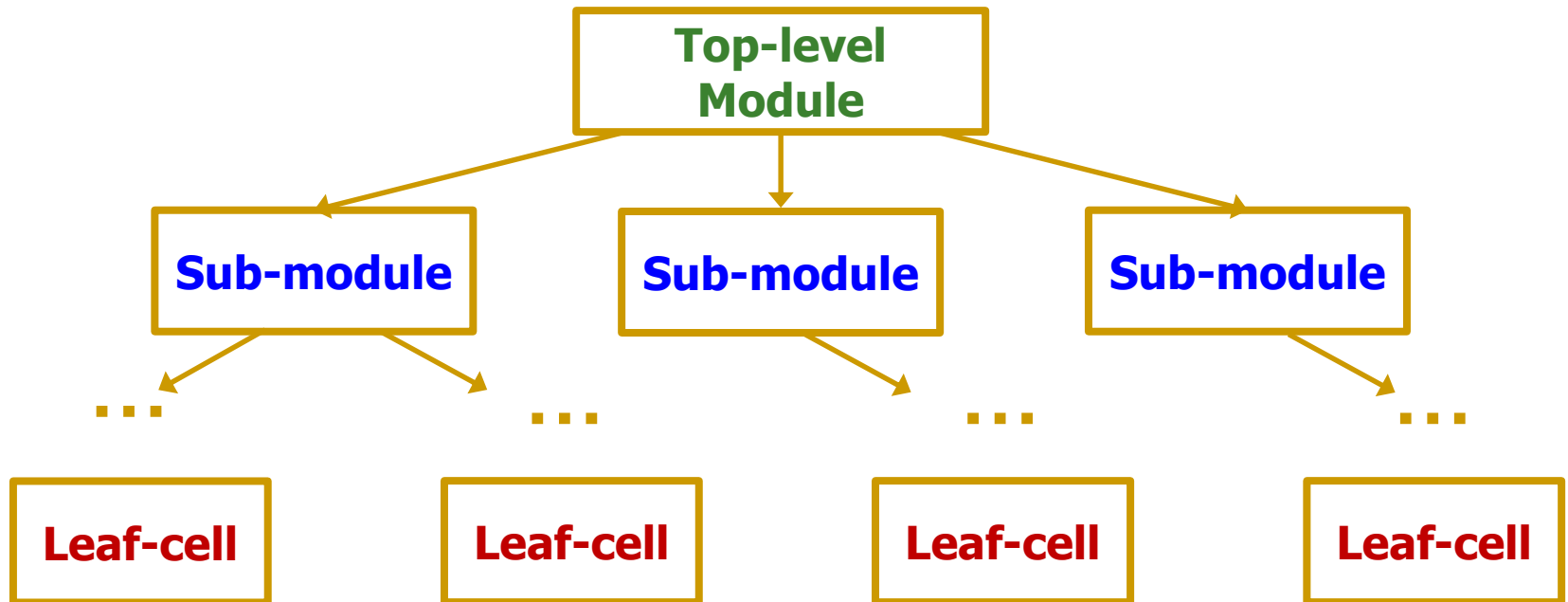
- ❑ In real world how big is size of one “blob” of random logic that we would describe as an HDL, then synthesize to gates?

<https://techreport.com/review/21987/intel-core-i7-3960x-processor>



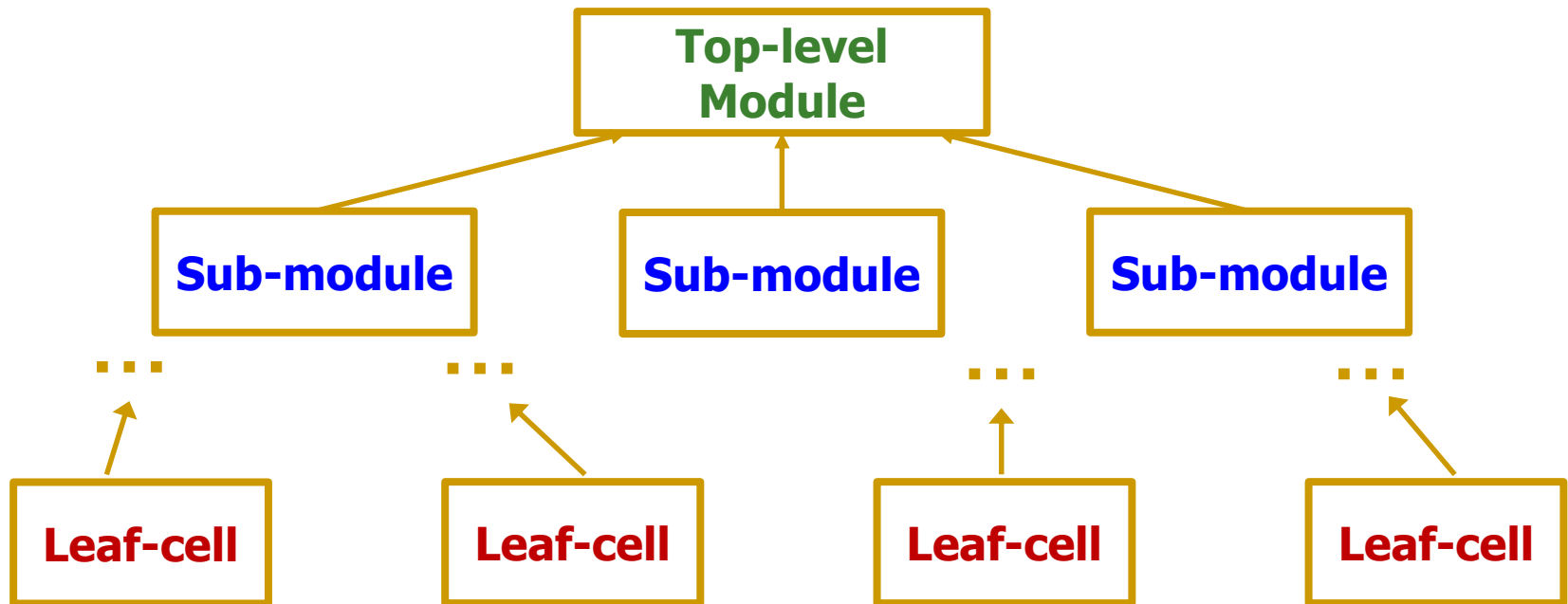
Top-Down Design Methodology

- We define the **top-level module** and identify the **sub-modules** necessary to build the top-level module
- Subdivide the sub-modules until we come to **leaf cells**
 - **Leaf cell**: circuit components that cannot further be divided (e.g., *logic gates, cell libraries*)



Bottom-Up Design Methodology

- We first identify the **building blocks** that are available to us
- **Build bigger modules**, using these building blocks
- These modules are then used for higher-level modules until we build the **top-level module** in the design

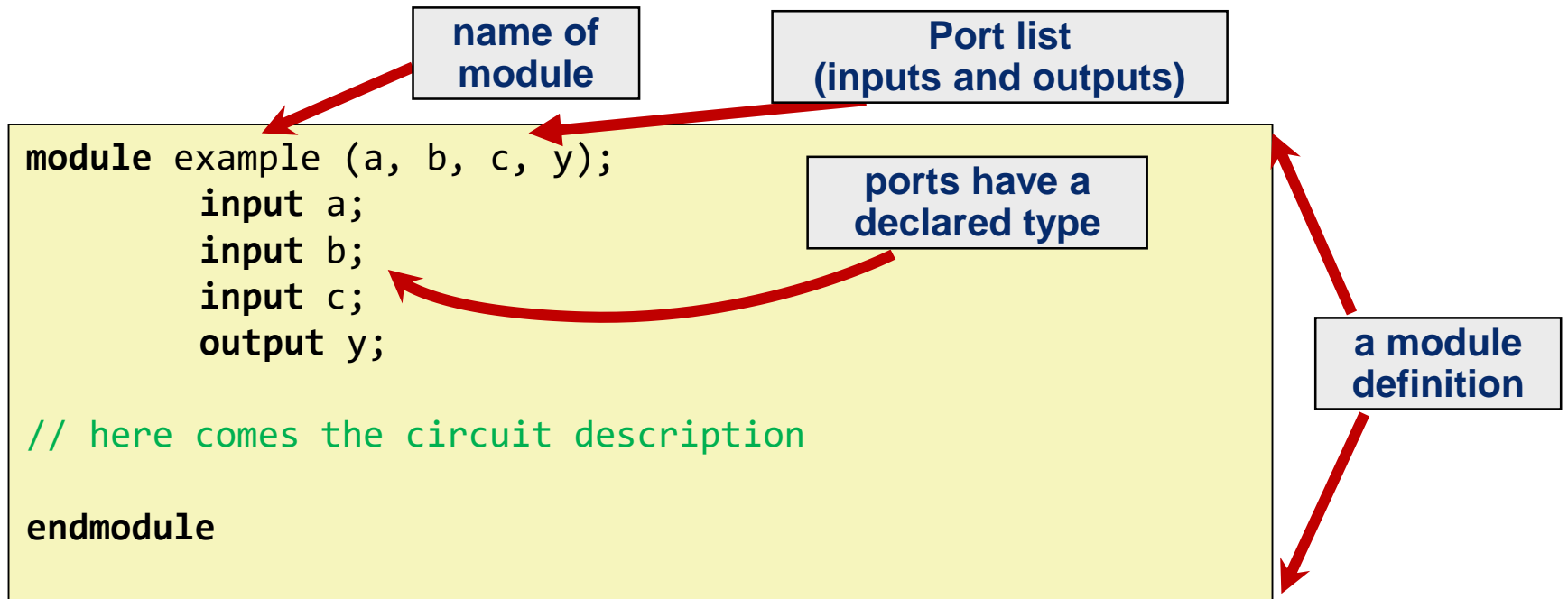
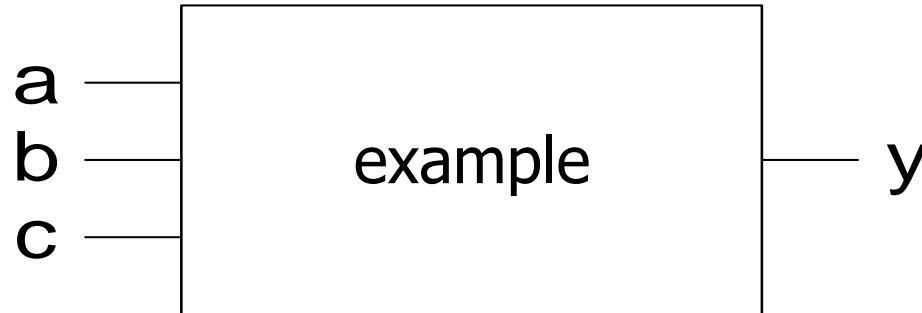


Defining a Module in Verilog

- A **module** is the main building block in Verilog
- We first need to define:
 - **Name** of the module
 - **Directions** of its **ports** (e.g., **input**, **output**)
 - **Names** of its **ports**
- Then:
 - Describe the **functionality** of the module



Implementing a Module in Verilog



A Question of Style

- **The following two codes are functionally identical**

```
module test ( a, b, y );  
    input a;  
    input b;  
    output y;  
  
endmodule
```

```
module test ( input a,  
              input b,  
              output y );  
  
endmodule
```

port name and direction declaration
can be combined

What If We Have Multi-bit Input/Output?

- **You can also define multi-bit Input/Output (Bus)**

- [range_end : range_start]
- **Number of bits:** range_end – range_start + 1

- **Example:**

```
input  [31:0] a;    // a[31], a[30] .. a[0]
output [15:8] b1;   // b1[15], b1[14] .. b1[8]
output [7:0]  b2;   // b2[7], b2[6] .. b2[0]
input          c;   // single signal
```

- **a** represents a 32-bit value, so we prefer to define it as:
[31:0] a
- It is preferred over [0:31] a which resembles *array* definition
- It is a good practice to **be consistent** with the representation of multi-bit signals, i.e., always [31:0] or always [0:31]

Manipulating Bits

- Bit Slicing
- Concatenation
- Duplication



Basic Syntax

- Verilog is case sensitive
 - `SomeName` and `somename` are not the same!
- Names cannot start with numbers:
 - `2good` is not a valid name
- Whitespaces are ignored

```
// Single line comments start with a //  
  
/* Multiline comments  
   are defined like this */
```

Two Main Styles of HDL Implementation

■ **Structural (Gate-Level)**

- ❑ The module body contains **gate-level description** of the circuit
- ❑ Describe how modules are interconnected
- ❑ Each module contains other modules (instances)
- ❑ ... and interconnections between these modules
- ❑ Describes a hierarchy

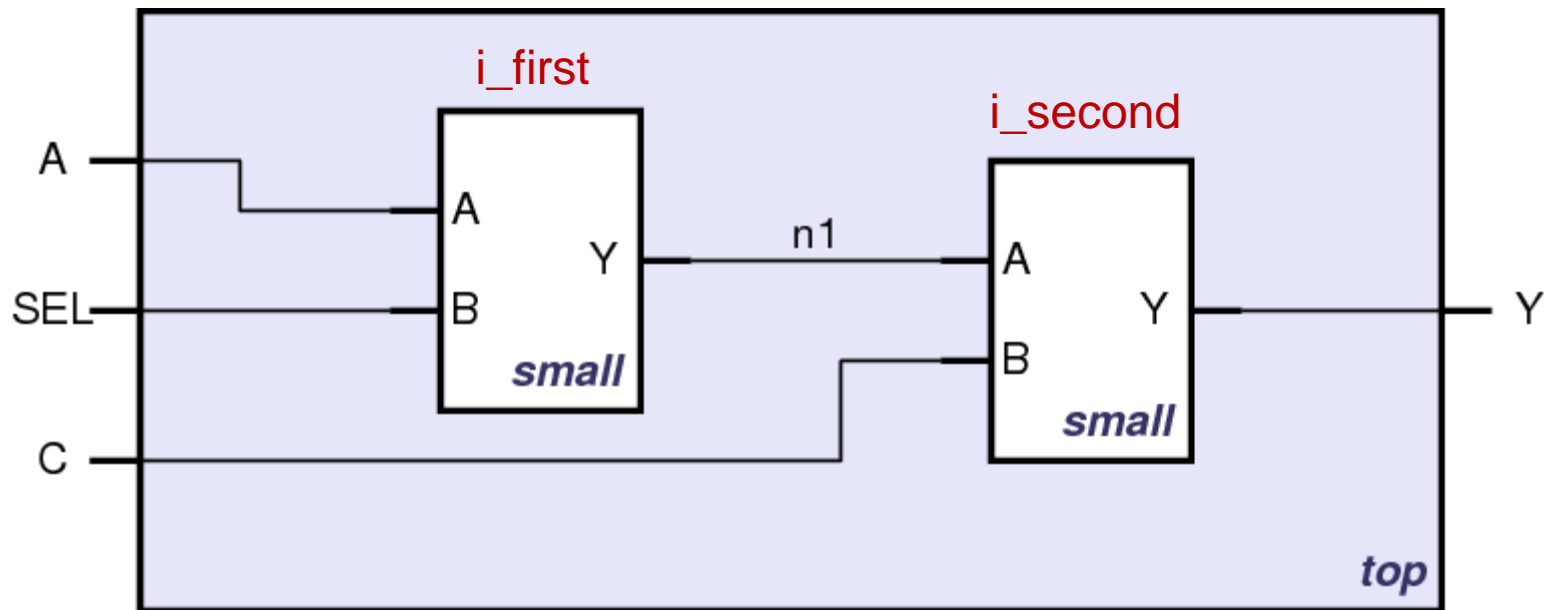
■ **Behavioral**

- ❑ The module body contains **functional description** of the circuit
- ❑ Contains logical and mathematical **operators**
- ❑ Level of abstraction is higher than gate-level
 - Many possible gate-level realizations of a behavioral description

■ **Practical circuits would use a combination of both**

Structural HDL

Structural HDL: Instantiating a Module

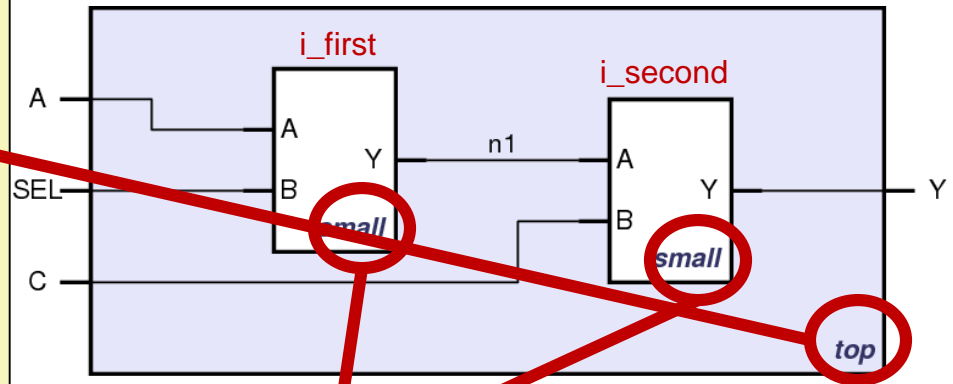


Schematic of module "top" that is built from two instances of module "small"

Structural HDL Example

■ Module Definitions in Verilog

```
module top(A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;  
  
endmodule
```



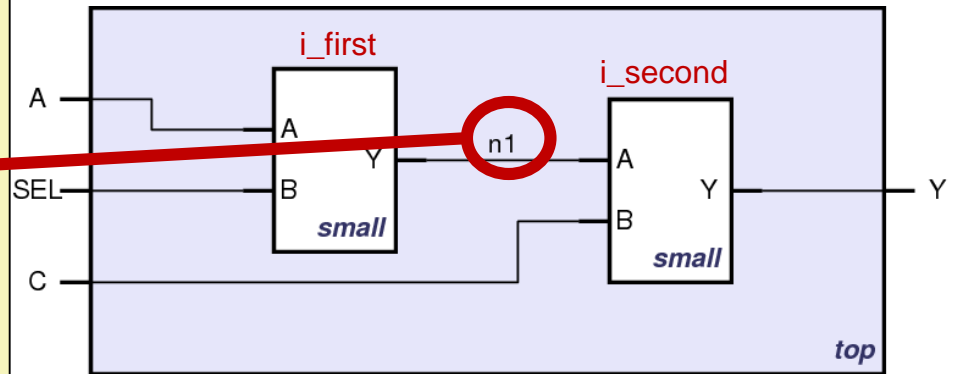
```
module small(A, B, Y);  
  input A;  
  input B;  
  output Y;  
  
  // description of small  
  
endmodule
```

Structural HDL Example

■ Defining wires (module interconnections)

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```

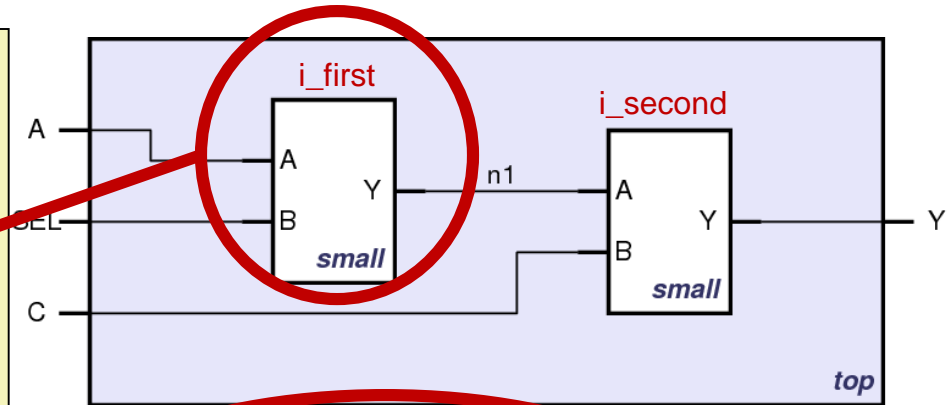
Structural HDL Example

■ The first instantiation of the “small” module

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
// instantiate small once  
small i_first ( .A(A),  
                .B(SEL),  
                .Y(n1) );
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

```
// description of small
```

```
endmodule
```

Structural HDL Example

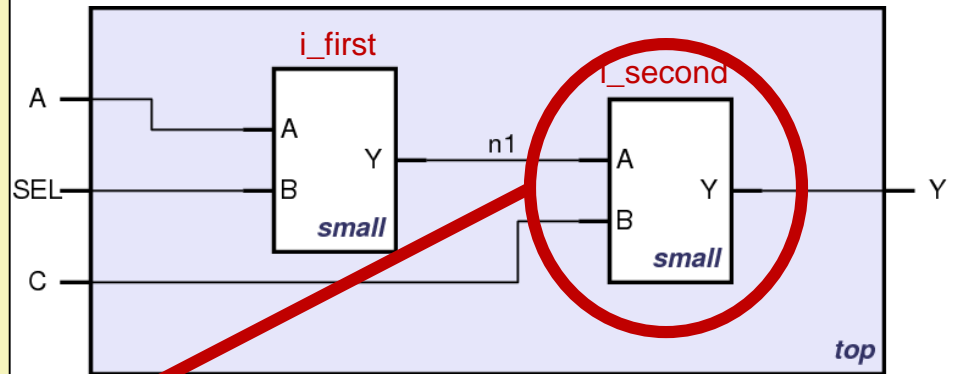
■ The second instantiation of the “small” module

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;
```

```
  // instantiate small once  
  small i_first ( .A(A),  
                  .B(SEL),  
                  .Y(n1) );
```

```
  // instantiate small second time  
  small i_second ( .A(n1),  
                   .B(C),  
                   .Y(Y) );
```

```
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;
```

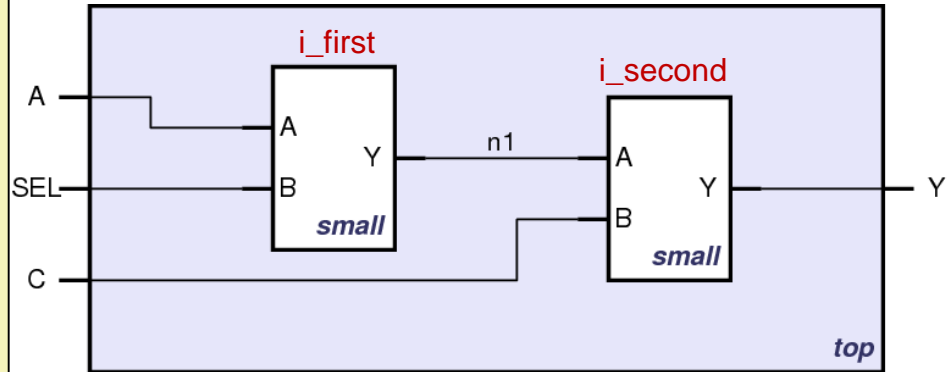
```
  // description of small
```

```
endmodule
```

Structural HDL Example

■ Short form of module instantiation

```
module top (A, SEL, C, Y);  
  input A, SEL, C;  
  output Y;  
  wire n1;  
  
  // alternative  
  small i_first ( A, SEL, n1 );  
  
  /* Shorter instantiation,  
     pin order very important */  
  
  // any pin order, safer choice  
  small i_second ( .B(C),  
                  .Y(Y),  
                  .A(n1) );  
  
endmodule
```



```
module small (A, B, Y);  
  input A;  
  input B;  
  output Y;  
  
  // description of small  
  
endmodule
```

Structural HDL Example 2

- Verilog supports basic logic gates as predefined *primitives*
 - These primitives are **instantiated** like modules except that they are predefined in Verilog and *do not need a module definition*

```
module mux2(input  [3:0] d0, d1,
             input          s,
             output [3:0] y);

    and  g1 (y1, d0, ns);
    and  g2 (y2, d1, s);
    or   g3 (y, y1, y2);
    not  g4 (ns, s);
endmodule
```

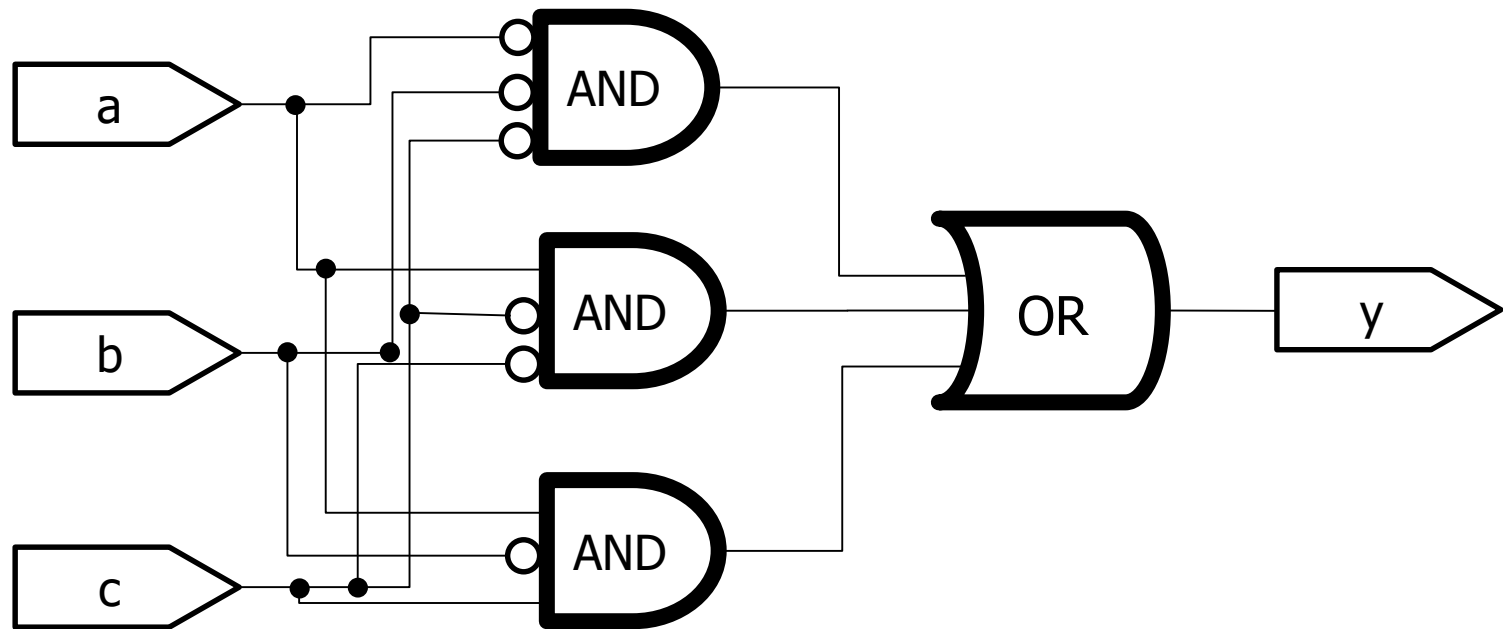
Behavioral HDL

Behavioral HDL: Defining Functionality

```
module example (a, b, c, y);  
    input a;  
    input b;  
    input c;  
    output y;  
  
    // here comes the circuit description  
    assign y = ~a & ~b & ~c |  
               a & ~b & ~c |  
               a & ~b & c;  
  
endmodule
```

Behavioral HDL: Schematic View

A behavioral implementation still models a hardware circuit!



Bitwise Operators in Behavioral Verilog

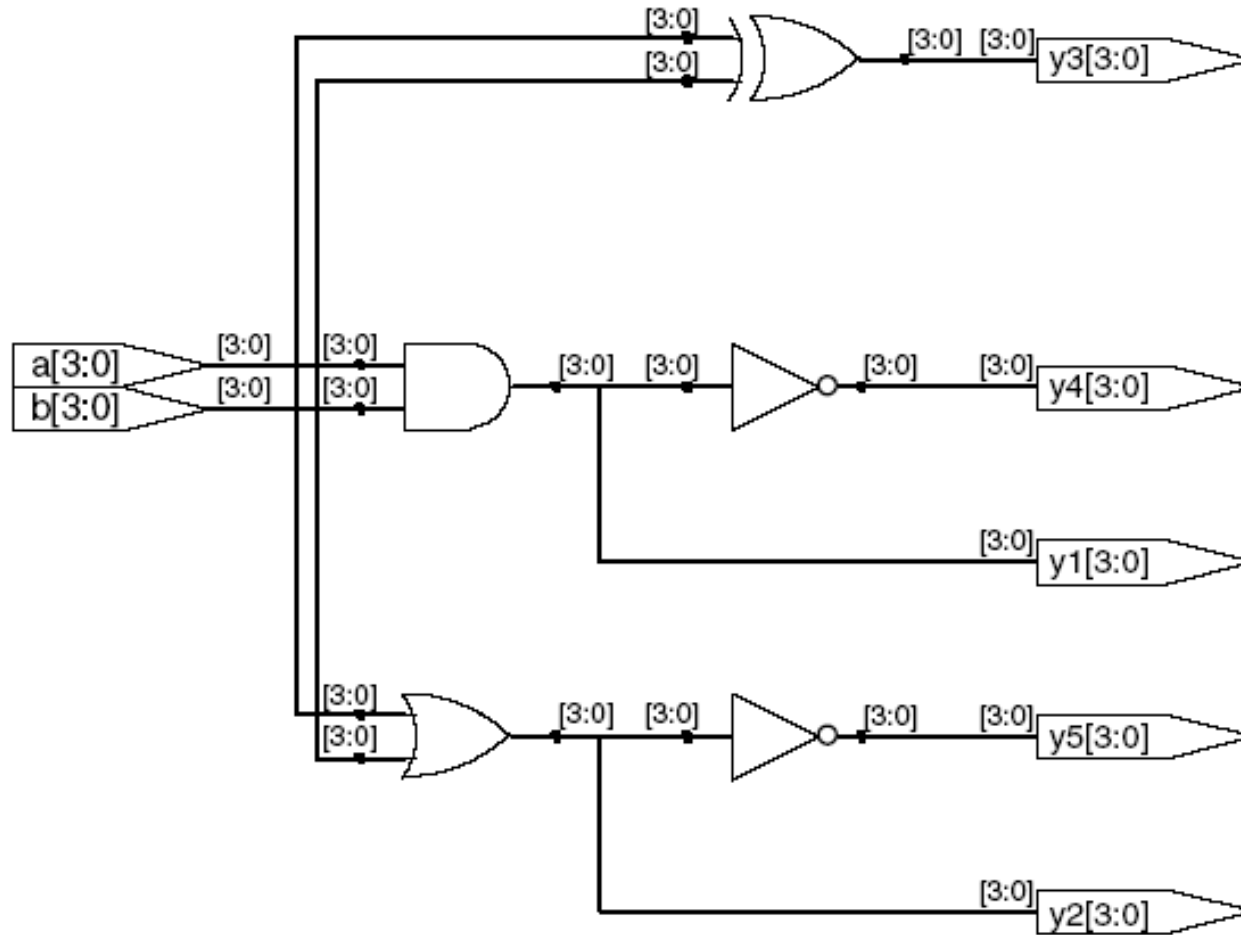
```
module gates(input  [3:0]  a, b,
              output [3:0] y1, y2, y3, y4, y5);

    /* Five different two-input logic
       gates acting on 4 bit buses */

    assign y1 = a & b;      // AND
    assign y2 = a | b;      // OR
    assign y3 = a ^ b;      // XOR
    assign y4 = ~(a & b);   // NAND
    assign y5 = ~(a | b);   // NOR

endmodule
```

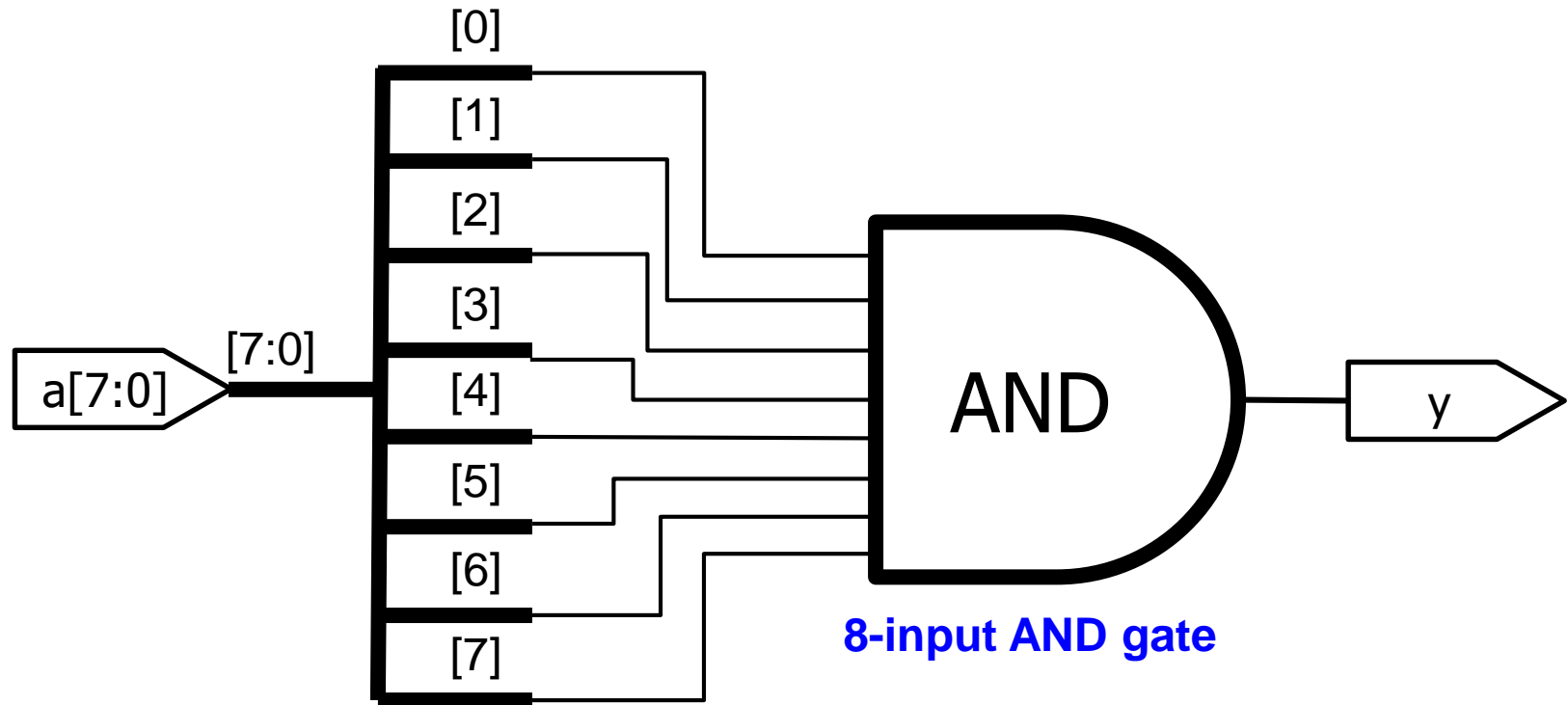
Bitwise Operators: Schematic View



Reduction Operators in Behavioral Verilog

```
module and8(input  [7:0] a,  
            output          y);  
  
    assign y = &a;  
  
    // &a is much easier to write than  
    // assign y = a[7] & a[6] & a[5] & a[4] &  
    //           a[3] & a[2] & a[1] & a[0];  
  
endmodule
```

Reduction Operators: Schematic View

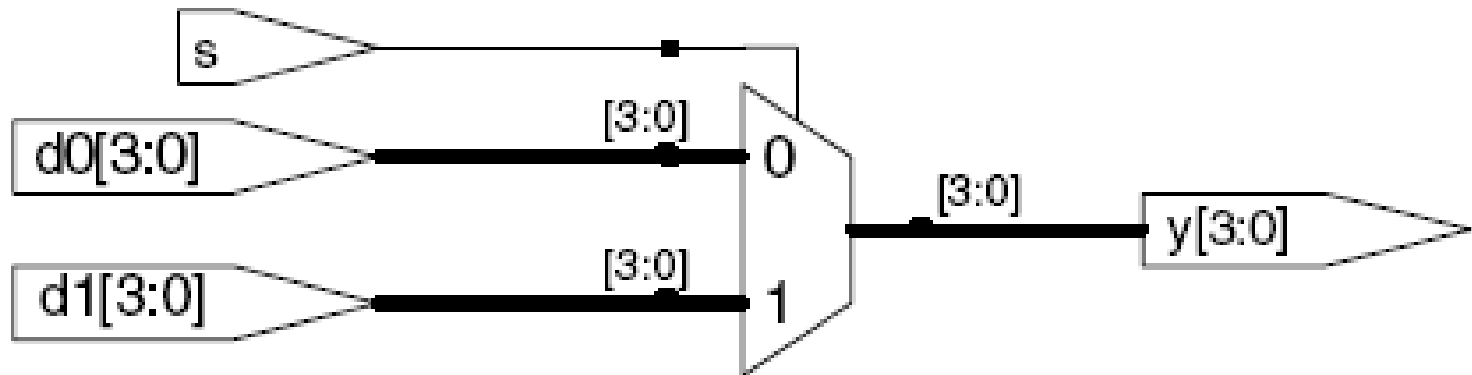


Conditional Assignment in Behavioral Verilog

```
module mux2(input  [3:0] d0, d1,  
            input      s,  
            output [3:0] y);  
  
    assign y = s ? d1 : d0;  
    // if (s) then y=d1 else y=d0;  
  
endmodule
```

- ? : is also called a **ternary operator** as it operates on three inputs:
 - ❑ s
 - ❑ d1
 - ❑ d0

Conditional Assignment: Schematic View



More Complex Conditional Assignments

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

    assign y = s[1] ? ( s[0] ? d3 : d2)
               : ( s[0] ? d1 : d0);

    // if (s1) then
    //     if (s0) then y=d3 else y=d2
    // else
    //     if (s0) then y=d1 else y=d0

endmodule
```

Even More Complex Conditional Assignments

```
module mux4(input  [3:0] d0, d1, d2, d3
            input  [1:0] s,
            output [3:0] y);

    assign y = (s == 2'b11) ? d3 :
               (s == 2'b10) ? d2 :
               (s == 2'b01) ? d1 :
               d0;

    // if      (s = "11" ) then y= d3
    // else if (s = "10" ) then y= d2
    // else if (s = "01" ) then y= d1
    // else                      y= d0

endmodule
```

Precedence of Operations in Verilog

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
, ~	OR, NOR
?:	ternary operator

Lowest

How to Express Numbers ?

N' **Bxx**

8' **b0000_0001**

- **(N) Number of bits**

- Expresses how many bits will be used to store the value


- **(B) Base**

- Can be b (binary), h (hexadecimal), d (decimal), o (octal)

- **(xx) Number**

- The value expressed in base
- Apart from numbers, it can also have X and Z as values
- Underscore _ can be used to improve readability

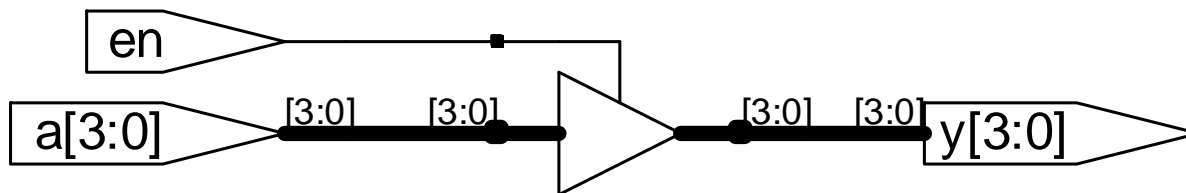
Number Representation in Verilog

Verilog	Stored Number	Verilog	Stored Number
4'b1001	1001	4'd5	0101
8'b1001	0000 1001	12'hFA3	1111 1010 0011
8'b0000_1001	0000 1001	8'o12	00 001 010
8'bxX0X1zZ1	XX0X 1ZZ1	4'h7	0111
'b01	0000 .. 0001  32 bits (default)	12'h0	0000 0000 0000

Floating Signals (Z)

- **Floating signal:** Signal that is not driven by any circuit
 - ❑ Open circuit, floating wire
- Also known as: **high impedance, hi-Z, tri-stated** signals

```
module tristate_buffer(input  [3:0] a,  
                      input      en,  
                      output [3:0] y);  
  
    assign y = en ? a : 4'bz;  
  
endmodule
```



Aside: Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

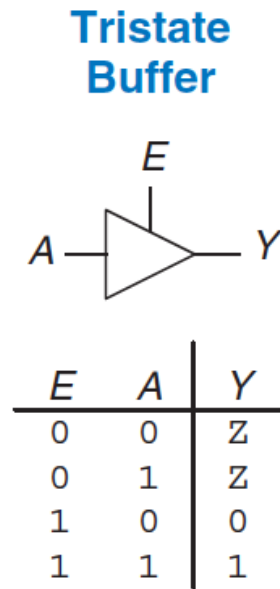
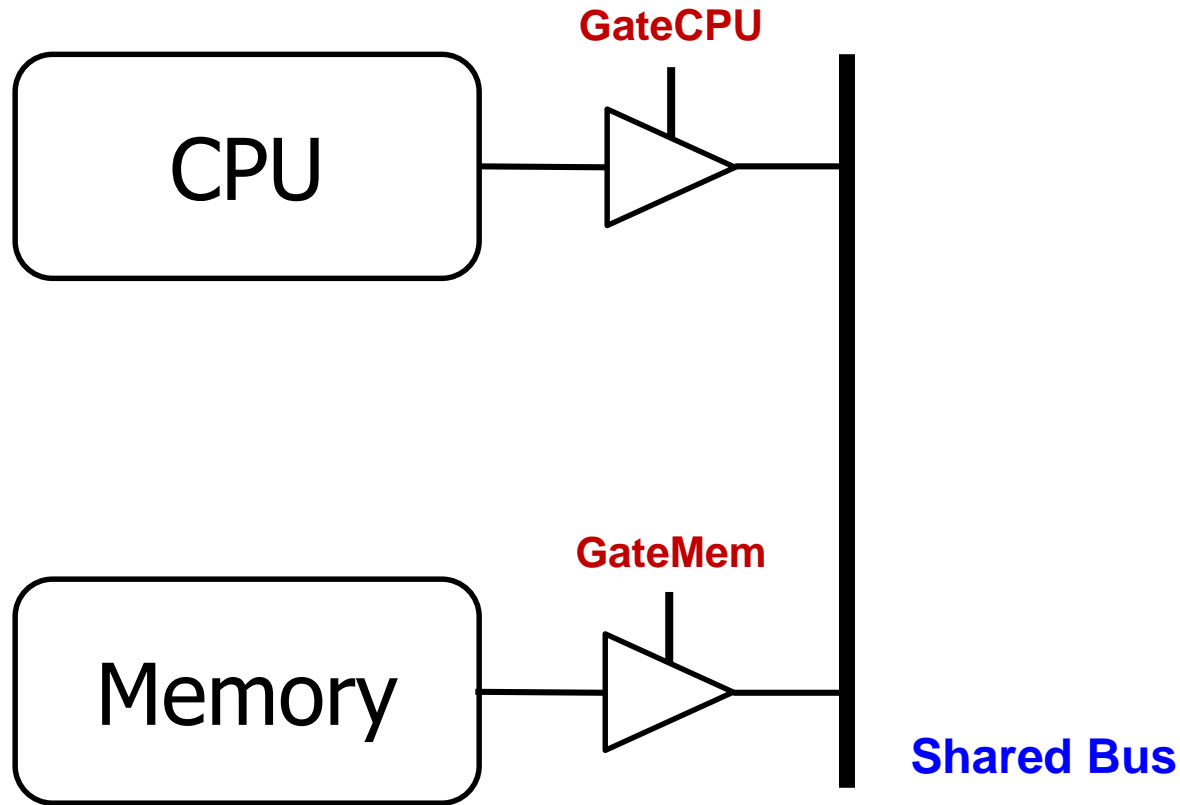


Figure 2.40 Tristate buffer

Example: Use of Tri-State Buffers

- Imagine a wire connecting the CPU and memory
 - At any time only the CPU or the memory can place a value on the wire, both not both
 - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time

Example Design with Tri-State Buffers



Truth Table for AND with Z and X

AND		A			
		0	1	Z	X
B	0	0	0	0	0
	1	0	1	X	X
	Z	0	X	X	X
	X	0	X	X	X

What Happens with HDL Code?

■ Synthesis

- ❑ Modern tools are able to **map** a *HDL code* into low-level *cell libraries*
- ❑ They can perform many **optimizations**
- ❑ ... however they **can not guarantee** that a solution is optimal
 - Mainly due to **computationally expensive placement** and **routing** algorithms
- ❑ Most common way of Digital Design these days

■ Simulation

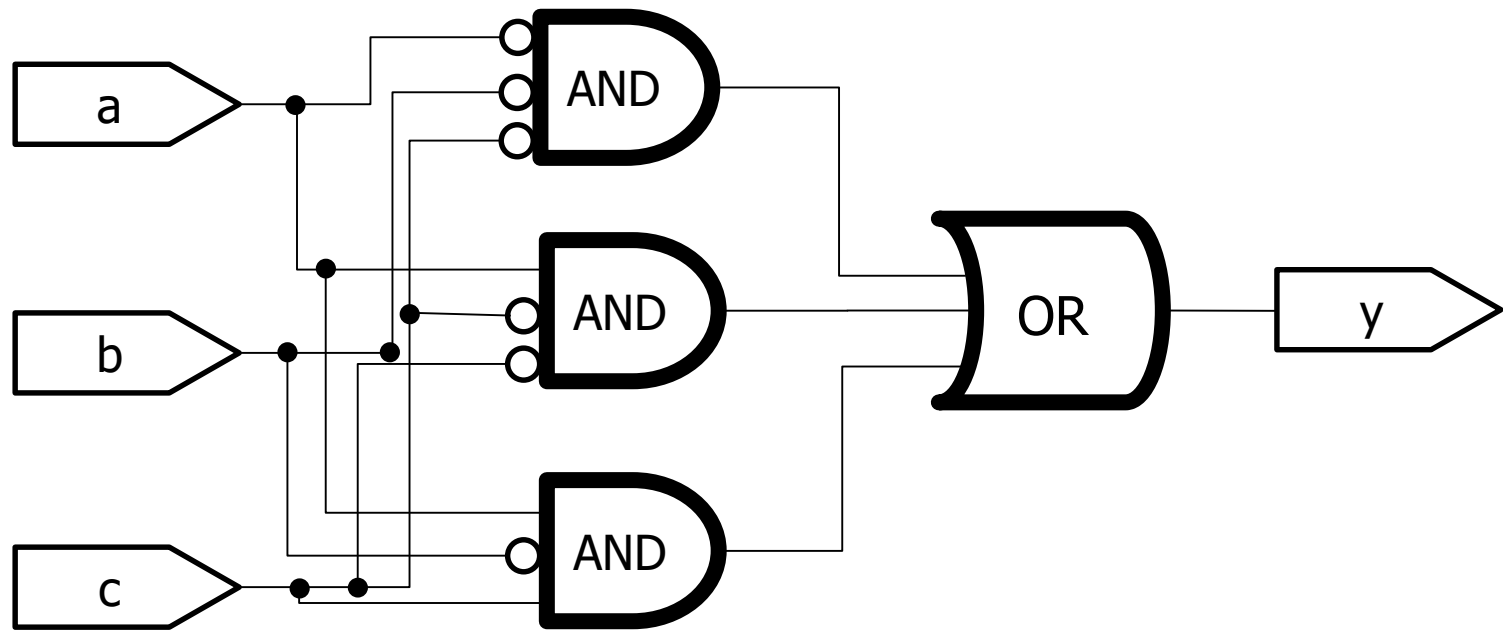
- ❑ Allows the behavior of the circuit to be **verified without actually manufacturing the circuit**
- ❑ Simulators can work on *structural* or *behavioral* HDL

We did not cover the following.
They are for your preparation.

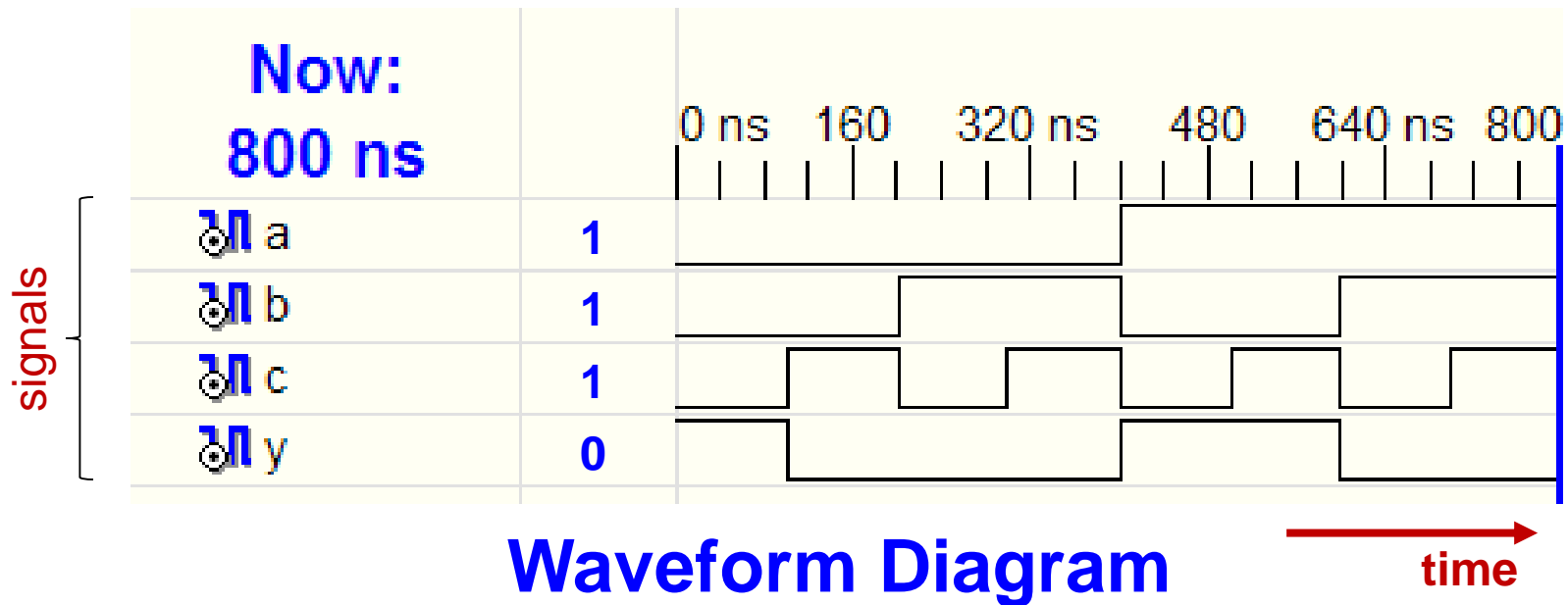
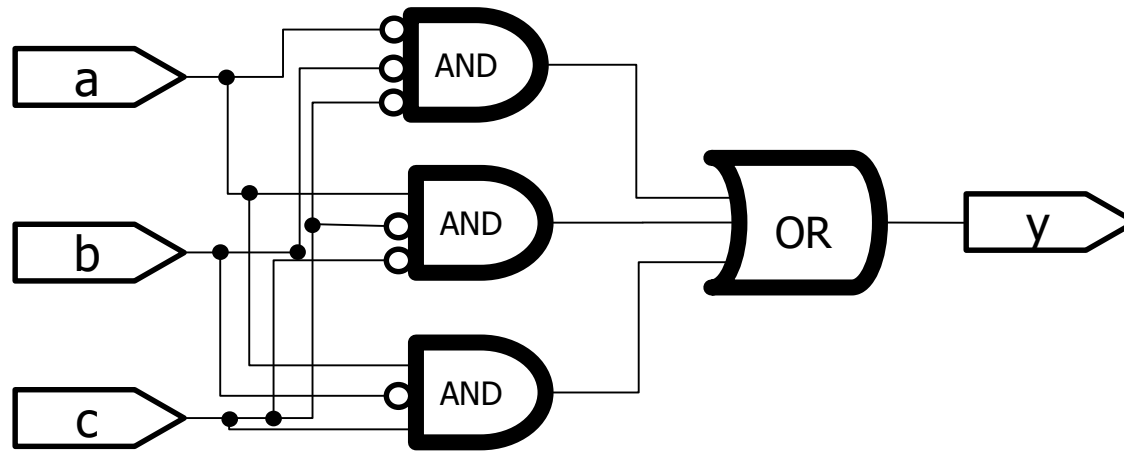
Recall This “example”

```
module example (a, b, c, y);  
    input a;  
    input b;  
    input c;  
    output y;  
  
    // here comes the circuit description  
    assign y = ~a & ~b & ~c |  
               a & ~b & ~c |  
               a & ~b & c;  
  
endmodule
```

Synthesizing the “example”



Simulating the “example”



What We Have Seen So Far

- **Describing structural hierarchy with Verilog**
 - Instantiate modules in an other module
- **Describing functionality using behavioral modeling**
- **Writing simple logic equations**
 - We can write AND, OR, XOR, ...
- **Multiplexer functionality**
 - If ... then ... else
- **We can describe constants**
- **But there is more...**

More Verilog Examples

- We can write Verilog code in many different ways
- Let's see how we can express the same functionality by developing Verilog code
 - At low-level
 - Poor readability
 - More optimization opportunities
 - At a higher-level of abstraction
 - Better readability
 - Limited optimization

Comparing Two Numbers

- **Defining your own gates as new modules**
- We will use our gates to show the different ways of implementing a 4-bit comparator (equality checker)

An XNOR gate

```
module MyXnor (input a, b,  
               output z);  
  
    assign z = ~(a ^ b); //not XOR  
  
endmodule
```

An AND gate

```
module MyAnd (input a, b,  
              output z);  
  
    assign z = a & b;    // AND  
  
endmodule
```

Gate-Level Implementation

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    MyAnd haha (.A(c0), .B(c1), .Z(c01) ); // AND
    MyAnd hoho (.A(c2), .B(c3), .Z(c23) ); // AND
    MyAnd bubu (.A(c01), .B(c23), .Z(eq) ); // AND

endmodule
```

Using Logical Operators

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
    wire c0, c1, c2, c3, c01, c23;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    assign c01 = c0 & c1;
    assign c23 = c2 & c3;
    assign eq  = c01 & c23;

endmodule
```

Eliminating Intermediate Signals

```
module compare (input a0, a1, a2, a3, b0, b1, b2, b3,
                output eq);
    wire c0, c1, c2, c3;

    MyXnor i0 (.A(a0), .B(b0), .Z(c0) ); // XNOR
    MyXnor i1 (.A(a1), .B(b1), .Z(c1) ); // XNOR
    MyXnor i2 (.A(a2), .B(b2), .Z(c2) ); // XNOR
    MyXnor i3 (.A(a3), .B(b3), .Z(c3) ); // XNOR
    // assign c01 = c0 & c1;
    // assign c23 = c2 & c3;
    // assign eq  = c01 & c23;
    assign eq    = c0 & c1 & c2 & c3;

endmodule
```

Multi-Bit Signals (Bus)

```
module compare (input [3:0] a, input [3:0] b,  
                output eq);  
    wire [3:0] c; // bus definition  
  
    MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) ); // XNOR  
    MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) ); // XNOR  
    MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) ); // XNOR  
    MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) ); // XNOR  
  
    assign eq  = &c; // short format  
  
endmodule
```

Bitwise Operations

```
module compare (input [3:0] a, input [3:0] b,  
                output eq);  
    wire [3:0] c; // bus definition  
  
    // MyXnor i0 (.A(a[0]), .B(b[0]), .Z(c[0]) );  
    // MyXnor i1 (.A(a[1]), .B(b[1]), .Z(c[1]) );  
    // MyXnor i2 (.A(a[2]), .B(b[2]), .Z(c[2]) );  
    // MyXnor i3 (.A(a[3]), .B(b[3]), .Z(c[3]) );  
  
    assign c = ~(a ^ b); // XNOR  
  
    assign eq = &c; // short format  
  
endmodule
```

Highest Abstraction Level: Comparing Two Numbers

```
module compare (input [3:0] a, input [3:0] b,  
               output eq);
```

```
// assign c = ~(a ^ b); // XNOR
```

```
// assign eq = &c; // short format
```

```
assign eq = (a == b) ? 1 : 0; // really short
```

```
endmodule
```

Writing More Reusable Verilog Code

- We have a module that can compare two 4-bit numbers
- What if in the overall design we need to compare:
 - **5**-bit numbers?
 - **6**-bit numbers?
 - ...
 - **N**-bit numbers?
 - Writing code for each case looks tedious
- What could be a better way?

Parameterized Modules

In Verilog, we can define **module parameters**

```
module mux2
  #(parameter width = 8)  // name and default value
  (input  [width-1:0] d0, d1,
   input                                     s,
   output [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

We can set the parameters to different values
when instantiating the module

Instantiating Parameterized Modules

```
module mux2
  #(parameter width = 8) // name and default value
  (input [width-1:0] d0, d1,
   input          s,
   output [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

What About Timing ?

- It is possible to define *timing relations* in Verilog. **BUT:**
 - ❑ These are **ONLY** for simulation
 - ❑ They **CAN NOT** be synthesized
 - ❑ They are used for *modeling delays* in a circuit

```
'timescale 1ns/1ps
module simple (input a, output z1, z2);

assign #5 z1 = ~a; // inverted output after 5ns
assign #9 z2 = a;  // output after 9ns

endmodule
```

More to come in later lectures!

Good Practices

- Develop/use a **consistent** naming style
- Use **MSB to LSB ordering** for buses
 - Use “**a[31:0]**”, **not** “**a[0:31]**”
- Define **one module per file**
 - Makes managing your design hierarchy easier
- Use a file name that equals module name
 - i.e., module **TryThis** is defined in a file called **TryThis.v**
- Always keep in mind that **Verilog describes hardware**

Summary

- We have seen an overview of Verilog
- Discussed structural and behavioral modeling
- Showed combinational logic constructs

Next Lecture: Sequential Logic

Design of Digital Circuits

Lecture 6: Combinational Logic, Hardware Description Lang. & Verilog

Prof. Onur Mutlu

ETH Zurich

Spring 2018

9 March 2018