Chapter 2

COMBINATIONAL CIRCUITS
SMALL DESIGNS

Signal Naming Standards

- **Active-high signal polarity**
  - **1 represents signal is <u>active</u>, <u>asserted</u>, <u>enabled</u>**
  - **0, otherwise**
  - **E.g., signal labeled as *x* without a pre- or post symbol**
- **Active-low signal polarity**
  - **0 represents signal is active, asserted, enabled**
  - **1, otherwise**
  - **E.g., signal labeled as _*x*, *x'*, /*x*, or *x*#**
    - **With a pre- or post-symbol**
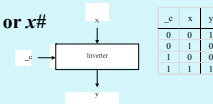
| _c	x	y
0	0	1
0	1	0
1	0	0
1	1	1

In this Chapter

- **Small Combinational Circuits**
  - **Fewer inputs (e.g., ≤ 4 inputs)**
  - **Circuits modeled as Truth Tables**
  - **Circuit minimization techniques**
- **Circuit implementation options**
  - **NANDs only**
  - **NORs only**
- **Timing diagram**
  - **Signal propagation delay**
  - **Understating signal hazards ("glitches")**
- **Other types of logic gates**
- **Design examples**
- **Introduction to design with HDL**

Primitive Logic Gates with Truth Tables
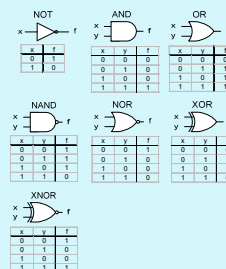
Small Combinational Circuits

- **Example: 2-bit unsigned multiplier, P = A * B**
- **Block diagram and truth table**
  - **Labeling of input and output signals**
- **Implementation options**
  - **LUT**
    - **Easier, slower, configurable**
  - **Logic circuit**
    - **Faster, less hardware**
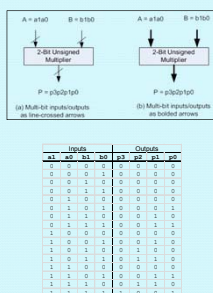
SOP Expressions

- **Based on input values that produce 1 as output**
- **Each such input is expressed as a product term**
- **Circuit performs AND-OR logic**

  $$f = \bar{x}\,\bar{y} + x\,y$$
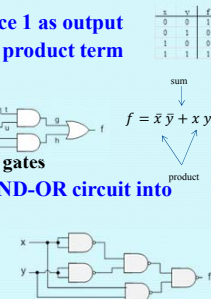
  - **Can be implemented with NAND gates**
- **DeMargan's theorems convert AND-OR circuit into NAND-only circuit**

  Theorem 1: $\overline{xy} = \bar{x} + \bar{y}$

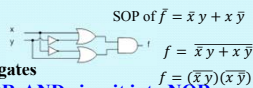 Theorem 2: $\overline{x + y} = \bar{x}\,\bar{y}$

POS Expressions

- **Based on input values that produce 0 for f (an output)**
  - **Same input values produce 1 for $\bar{f}$**
- **Find expression for f by complementing $\bar{f}$**
- **Each such input is expressed as a sum term**

$$f = (x + \bar{y})(\bar{x} + y)$$

x	y	f	$\bar{f}$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	0

- **Circuit performs OR-AND logic**

SOP of $\bar{f} = \bar{x}\,y + x\,\bar{y}$

$$f = \overline{\bar{x}\,y + x\,\bar{y}}$$
$$f = \overline{(\bar{x}\,y)(x\,\bar{y})}$$

  - **Can be implemented with NOR gates**
- **DeMargan's theorems convert OR-AND circuit into NOR-only circuit**
- **Also can use signal negation with Dual principle**

Dual of $\bar{f} = (\bar{x} + y)(x + \bar{y})$

Why minimize logic expressions

- **Eliminates redundancies**
- **Requires fewer gates**
- **Fewer inputs per gates**
- **Less wire**
- **Less power usage**
- **Reduces circuit delay**

How many gates and types for SOP?

Canonical SOP:
3 NOTs,
four 3-input ANDs,
one 4-input OR.

Minimal SOP:
One NOT,
two 2-input AND,
one 2-input OR

Implement with NAND gates
Canonical SOP: $f = \bar{x}\,\bar{y}z + \bar{x}\,yz + xy\bar{z} + xyz$
Minimal SOP: $f = \bar{x}z + xy$

Implement with NOR gates
Canonical POS: $f = (x + y + z)(x + \bar{y} + z)(\bar{x} + y + z)(\bar{x} + y + \bar{z})$
Minimal POS: $f = (x + z)(\bar{x} + y)$

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Show mathematically

- **NAND implementation**

$$f = \bar{\bar{f}} \quad\Longrightarrow\quad f = \overline{\overline{\bar{x}\,\bar{y} + x\,y}}$$

Theorem 2
$$\overline{x + y} = \bar{x}\,\bar{y}$$

$$f = \overline{(\overline{\bar{x}\,\bar{y}})(\overline{x\,y})}$$
NAND   NAND

NAND, again

- **NOR implementation**

$$f = (x + \bar{y})(\bar{x} + y) = \overline{\overline{(x + \bar{y})(\bar{x} + y)}} = \overline{\overline{(x + \bar{y})} + \overline{(\bar{x} + y)}}$$
NOR   NOR

NOR, again

Karnaugh map (K-Map) Layouts

yz:	00	01	11	10
x: 0	0	1	3	2
1	4	5	7	6

2 × 4

z:	0	1
xy: 00	0	1
01	2	3
11	6	7
10	4	5

4 × 2

4 × 4

yz:	00	01	11	10
wx: 00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Canonical Expression

$$g = x\bar{y} + \bar{x}z + xyz$$ Non-Canonical

$$g = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz$$ Canonical, every term has all the variable names

Min Terms vs. Canonical expression

For example, $g(x, y, z) = \sum(0, 1, 6, 7)$

$$g(x, y, z) = \sum((000)2, (001)2, (110)2, (111)2)$$

$$g = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + xy\bar{z} + xyz$$

	Inputs				Outputs			
	a1	a0	b1	b0	p3	p2	p1	p0
	0	0	0	0	0	0	0	0
	0	0	0	1	0	0	0	0
	0	0	1	0	0	0	0	0
	0	0	1	1	0	0	0	0
	0	1	0	0	0	0	0	0
	0	1	0	1	0	0	0	1
	0	1	1	0	0	0	1	0
	0	1	1	1	0	0	1	1
	1	0	0	0	0	0	0	0
	1	0	0	1	0	0	1	0
10	1	0	1	0	0	1	0	0
11	1	0	1	1	0	1	1	0
	1	1	0	0	0	0	0	0
	1	1	0	1	0	0	1	1
14	1	1	1	0	0	1	1	0
	1	1	1	1	1	0	0	1

$$p2(a1, a0, b1, b0) = \sum(10, 11, 14)$$

SOP and POS K-maps

yz:	00	01	11	10
x: 0				1
1		1	1	

$g(x, y, z) = \sum(2, 6, 7)$

SOP

yz:	00	01	11	10
x: 0	0	0	0	
1	0	0		

$g(x, y, z) = \Pi(0, 1, 3, 4, 5)$

POS

Minimizing SOP Expressions

$\sum (2,3,6,7) = \bar{x}y\bar{z} + \bar{x}yz + xy\bar{z} + xyz$

$= \bar{x}y(\bar{z} + z) + xy(\bar{z} + z)$ Factor out smaller terms and simplify

$= \bar{x}y + xy$ Factor out y and simplify

$= y(\bar{x} + x)$ Simplify

$= y$

| yz:	00	01	11	10
x: 0			1	1
1			1	1

Each pair of adjacent terms reduces to a simplified expression with one less variable.

Don't-Care Signal values

- **Example: Displaying BCD numbers**

A 7-segment display unit and converter

Inputs				Outputs						
w	x	y	z	fa	fb	fc	fd	fe	ff	fg
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	0	1	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0							
1	0	1	1							
1	1	0	0							
1	1	0	1							
1	1	1	0							
1	1	1	1							

?
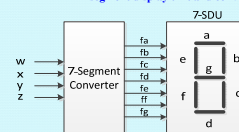
Assuming that w, x, y, and z will not exceed 9, what should we enter in the table for inputs 10 through 15?

K-Map Minimization Rules

1) **Min/max terms that differ in only one bit are adjacent (an Implicant). A K-map is assumed to wrap around on both sides.**
2) **A set of adjacent min/max terms may be combined to form a large group (a Prime Implicant). The number of terms in each group must be powers of 2**

   e.g., 2, 4, 8, or 16 terms.
3) **Each group of min/max terms must contain at least a single term that doesn't belong to any other group (no redundant groups), an Essential Prime Implicant**
4) **All terms must be grouped.**

K-Map with Don't Cares

$f(w, x, y, z) = \Sigma(1, 9, 14) + \Sigma_d(3, 7, 11)$

yz:	00	01	11	10
w:x 00		1	d	
01			d	
11				①
10		1	d	

$f(w, x, y, z) = \bar{x}z + wxy\bar{z}$

More K-map Examples
(no slides)

Logic Minimization Algorithm

- **Based on K-Map minimization technique**
  1. **Compare neighboring min/max terms two at a time (e.g., 0000 with 0001) to produce all Implicants**
  2. **Write the Implicant with a dash (e.g., 000-) for the bit that changes**
  3. **Repeat steps 1 and 2 for neighboring terms with matching dashes (e.g., 000- with 100- to get -00-)**
  4. **Prime implicants: Repeat step 3 until all prime implicants are identified**
  5. **Essential prime implicants: Choose a minimum set among the prime implicants**

Minimization Software

$f(w, x, y, z) = \Sigma\ (0, 2, 3, 4, 5, 6, 7, 8, 10, 12, 13)$

Input File

```
#Inputs: 4, Outputs: 1
.i 4
.o 1
#Input labels
.ilb w x y z
#output bit label
.ob f
#list of min-terms separated by space and a single output bit separated
by a tab
0 0 0 0      1
0 0 1 0      1
0 0 1 1      1
0 1 0 0      1
0 1 0 1      1
0 1 1 0      1
0 1 1 1      1
1 0 0 0      1
1 0 1 0      1
1 1 0 0      1
1 1 0 1      1
#end of list
.e
```

Output

```
#Inputs: 4, Outputs: 1
#Input signal labels
#output bit label
#list of min-terms and output
#end of list
.i 4
.o 1
.ilb w x y z
.ob f
.p 3
-10- 1
-0-0 1
0-1- 1
.e
```

$$\overline{x}\overline{z} + \overline{w}y + x\overline{y}$$

- **Can be used with don't care inputs too**

Digital Logic Design and Computer Organization with Computer Architecture for Security

19

Other Gates

- **Open collector (o.c.) buffer**
  - **Application: Wired-logic with a large fan-in**
    - **E.g., wired-AND or wired-OR logic**
  - **Many application areas**
- **Tri-state buffer**
  - **Used to create a bus for multiple modules to transmit data**
  - **Modules not outputting to the bus should be electrically isolated**

e	In	Out
0	0	Z
0	1	Z
1	0	0
1	1	1

Digital Logic Design and Computer Organization with Computer Architecture for Security

22

Circuit Timing Diagram

1. **Circuits have gate and signal wire delays**
2. **Gates may have different output signal *rise* and *fall* times**
3. **Circuits have different signal paths from inputs to outputs**
- **These may result in signals reaching each gate at different times**
- **Can cause unwanted signal change (glitch) at some outputs**
- **Must wait for the longest signal propagation delay before the output(s) of a circuit can be used (e.g., stored in a register)**

Digital Logic Design and Computer Organization with Computer Architecture for Security

20

Small combinational design examples

- **Full-adder circuit**
- **Multiplexer circuit**
  - **Selects data one from 2 or more inputs**
- **Decoder circuit**
  - **Translates an input value to a corresponding signal**
- **Encoder circuit**
  - **Translates an active input signal to a corresponding signal number**

Digital Logic Design and Computer Organization with Computer Architecture for Security

23

Circuit Fan-In and Fan-Out

- **Fan-in: Number of gate inputs**
- **Fan-out: Number of places a gate output can connect to**
- **Buffer gate to increase a gate's fan-out**
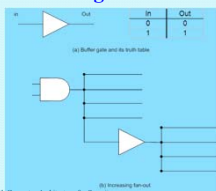
In	Out
0	0
1	1

(a) Buffer gate and its truth table
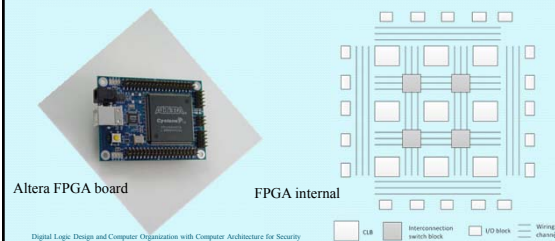
(b) Increasing fan-out

Digital Logic Design and Computer Organization with Computer Architecture for Security

21

Logic Implementation

- **ASIC (application specific integrated chip)**
- **FPGA (Field Programmable Gate Arrays)**

Altera FPGA board

FPGA internal

CLB   Interconnection switch block   I/O block   Wiring channel

Digital Logic Design and Computer Organization with Computer Architecture for Security
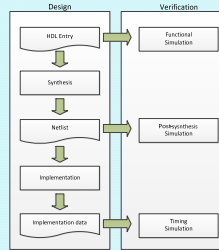
24

Design Flow

E.g., Verilog

Translate

Expressions

Map expressions to on-chip resources

Estimate signal delays

Verify circuit description

Verify netlist

Verify timing requirements

Behavioral Model

```
module full_adder
(
        input a, b, cin,
        output reg s, cout
);

always@(a or b or cin)
begin
        case ({a, b, cin})
        3'b000:     begin s = 0; cout = 0; end
        3'b001:     begin s = 1; cout = 0; end
        3'b010:     begin s = 1; cout = 0; end
        3'b011:     begin s = 0; cout = 1; end
        3'b100:     begin s = 1; cout = 0; end
        3'b101:     begin s = 0; cout = 1; end
        3'b110:     begin s = 0; cout = 1; end
        3'b111:     begin s = 1; cout = 1; end
        default:begin s = 0; cout = 0; end
        endcase
end
endmodule
```

Design Entry

- Schematic entry
- Hardware Description Language
  - Structural Model
    - Use gates
    - Use predefined modules
  - Behavior model
    - Use Boolean expressions, if-else, case (switch), for-loop, operators "+", "–", etc.
    - Not all behavioral models are synthesizable
    - Applications of non-synthesizable models
      - Generate test vectors for synthesizable models
      - Investigate computer architecture design ideas
  - Hybrid
    - Use both structural and behavioral models

Schematic Entry
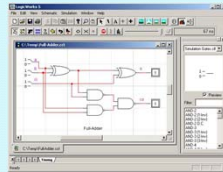
A summary of Verilog HDL operators

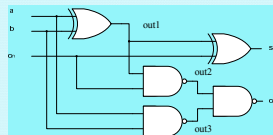| Precedence	Operator Type		Symbol	Example
Highest	Unary		+, -, !, ~	+a, -a (negate a), !a (logical not), ~a (bidwise not)
	Exponential		**	a ** 3 (a cubed)
	Arithmetic 1		*, /, %	a * b (multiply), a / b (divide), a % b (mod)
	Arithmetic 2		+, -	a + b (add), a - b (subtract)
	Shift:			
		Logical	<<, >>	a << 2 (shift left twice)
		Arithmetic	<<<, >>>	a >>> 3 (shift right 3 times extending the sign bit)
	Relational		<, <=, >, >=	a >= b (a greater or equal to b)
	Equality			
		Logical	==, !=	a == b if a is identical to b excluding x and z
		Case	===, !==	a === b is identical to b including x and z
	Bit-wise			
		Basics	&, |, ~, ^	a & b (and), a | b (or), ~a (not), a ^ b (xor)
		Combined	&~, |~, ~^, ^~	a &~ b (nand), a |~ b (nor), a ~^ b (xnor), a ^~ b (xnor)
	Logical		&&, ||, !	a &&b (and), a || b (or), !a (not)
Lowest	Conditional		?:	(a >= b) ? a - b : b - a

Structural Model Using Primitive Gates

```
module full_adder
(
        input a, b, cin,
        output s, cout
); //defines a module's name and its interface signals

wire out1, out2, out3;  //defines local signal names

xor     x1(out1, a, b);
xor     x2(s, out1, cin);
nand    n1(out2, out1, cin);
nand    n2(out3, a, b);
nand    n3(cout, out2, out3);

endmodule
```

5