Combinational Circuit Design

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- How to Design a Combinational Circuit
- Designing a BCD to Excess-3 Code Converter
- Designing a BCD to 7-Segment Decoder
- Hierarchical Design
- Iterative Design

Combinational Circuit

- A combinational circuit is a block of logic gates having:

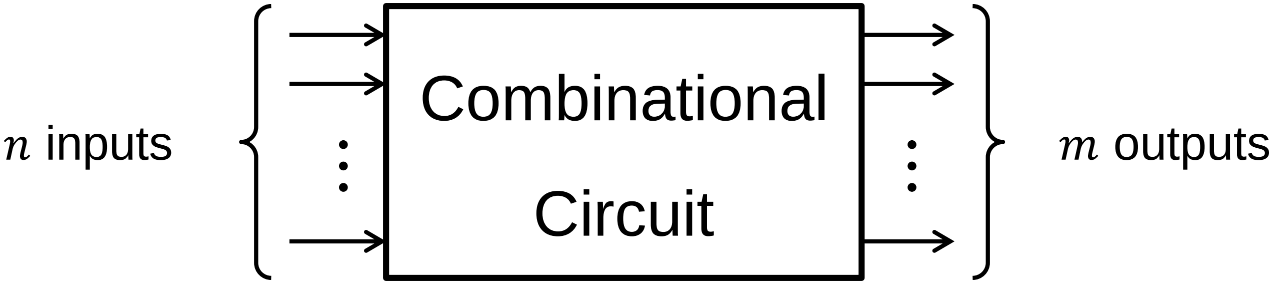
  n inputs: $x_1, x_2, \dots, x_n$

  m outputs: $f_1, f_2, \dots, f_m$

- Each output is a function of the input variables

- Each output is determined from **present combination** of inputs

- Combination circuit performs operation specified by logic gates



How to Design a Combinational Circuit

1. **Specification**
    - ✧ Specify the inputs, outputs, and what the circuit should do
2. **Formulation**
    - ✧ Convert the specification into truth tables or logic expressions for outputs
3. **Logic Minimization**
    - ✧ Minimize the output functions using K-map or Boolean algebra
4. **Technology Mapping**
    - ✧ Draw a logic diagram using ANDs, ORs, and inverters
    - ✧ Map the logic diagram into the selected technology
    - ✧ Considerations: cost, delays, fan-in, fan-out
5. **Verification**
    - ✧ Verify the correctness of the design, either manually or using simulation

Designing a BCD to Excess-3 Code Converter

1. Specification

✧ Convert BCD code to Excess-3 code

✧ Input: BCD code for decimal digits 0 to 9

✧ Output: Excess-3 code for digits 0 to 9

2. Formulation

✧ Done easily with a truth table

✧ BCD input: a, b, c, d

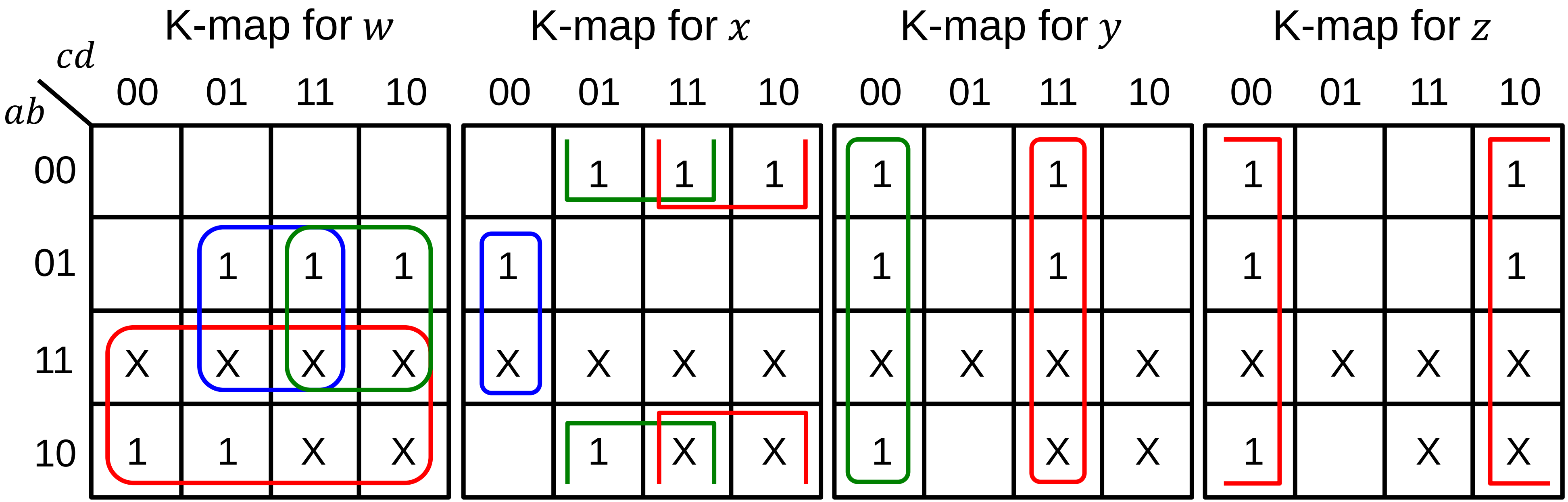
✧ Excess-3 output: w, x, y, z

✧ Output is don't care for 1010 to 1111

BCD	Excess-3
a b c d	w x y z
0 0 0 0	0 0 1 1
0 0 0 1	0 1 0 0
0 0 1 0	0 1 0 1
0 0 1 1	0 1 1 0
0 1 0 0	0 1 1 1
0 1 0 1	1 0 0 0
0 1 1 0	1 0 0 1
0 1 1 1	1 0 1 0
1 0 0 0	1 0 1 1
1 0 0 1	1 1 0 0
1010 to 1111	X X X X

Designing a BCD to Excess-3 Code Converter

3. Logic Minimization using K-maps



K-map for w

$ab \backslash cd$	00	01	11	10
00				
01		1	1	1
11	X	X	X	X
10	1	1	X	X

K-map for x

$ab \backslash cd$	00	01	11	10
00		1	1	1
01	1			
11	X	X	X	X
10		1	X	X

K-map for y

$ab \backslash cd$	00	01	11	10
00	1		1	
01	1		1	
11	X	X	X	X
10	1		X	X

K-map for z

$ab \backslash cd$	00	01	11	10
00	1			1
01	1			1
11	X	X	X	X
10	1		X	X

Minimal Sum-of-Product expressions:

$$w = a + bc + bd \text{ , } x = b'c + b'd + bc'd' \text{ , } y = cd + c'd' \text{ , } z = d'$$

Additional 3-Level Optimizations: extract common term $(c + d)$

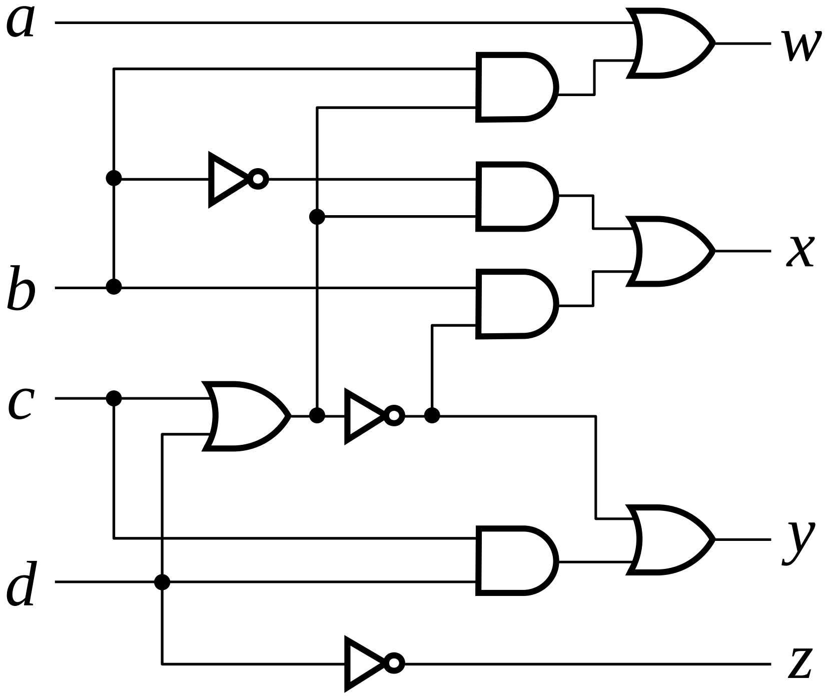
$$w = a + b(c + d) \text{ , } x = b'(c + d) + b(c + d)' \text{ , } y = cd + (c + d)'$$

Designing a BCD to Excess-3 Code Converter

4. Technology Mapping

Draw a logic diagram using ANDs, ORs, and inverters

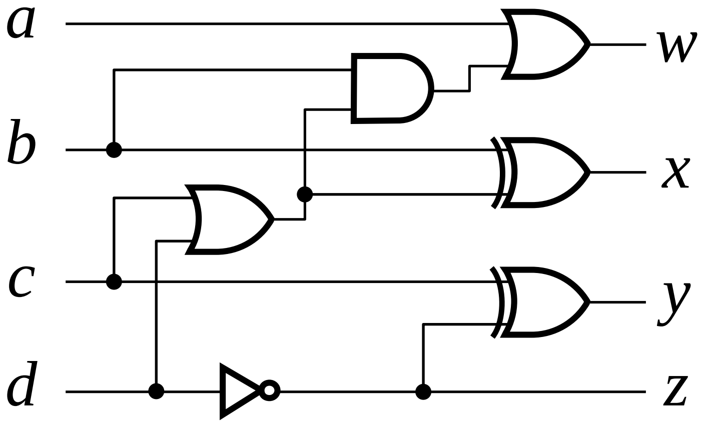
Other gates can be used, such as NAND, NOR, and XOR



Using XOR gates

$$x = b'(c+d) + b(c+d)' = b \oplus (c+d)$$

$$y = cd + c'd' = (c \oplus d)' = c \oplus d'$$



Designing a BCD to Excess-3 Code Converter

5. Verification

Can be done manually

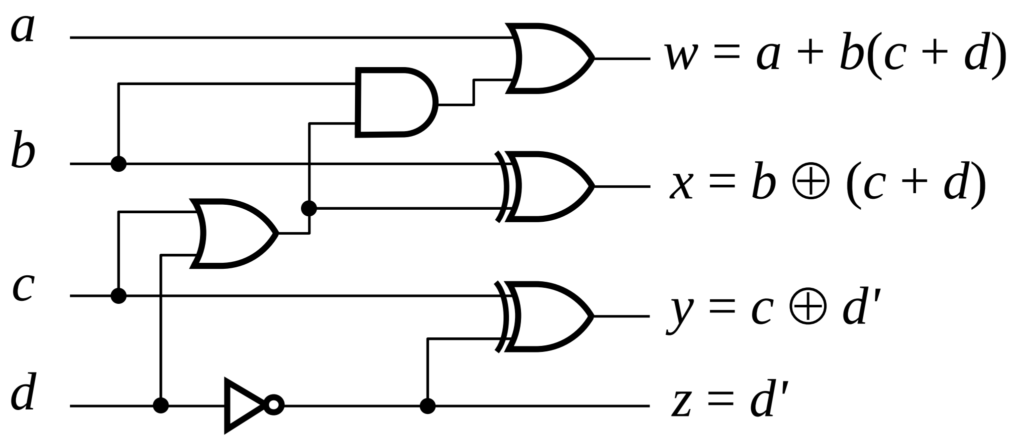
Extract output functions from circuit diagram

Find the truth table of the circuit diagram

Match it against the specification truth table

Verification process can be automated

Using a simulator for complex designs

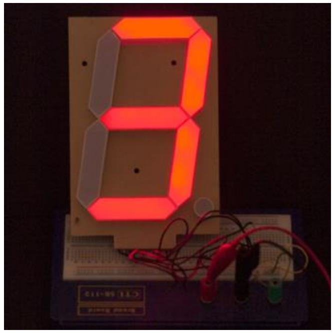


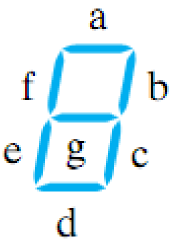
Truth Table of the Circuit Diagram

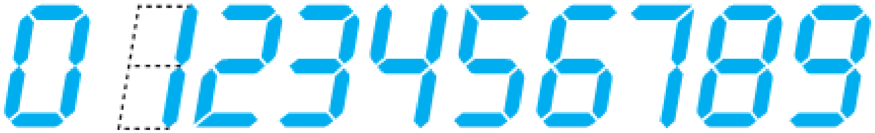
BCD a b c d	c+d	b(c+d)	Excess-3 w x y z
0 0 0 0	0	0	0 0 1 1
0 0 0 1	1	0	0 1 0 0
0 0 1 0	1	0	0 1 0 1
0 0 1 1	1	0	0 1 1 0
0 1 0 0	0	0	0 1 1 1
0 1 0 1	1	1	1 0 0 0
0 1 1 0	1	1	1 0 0 1
0 1 1 1	1	1	1 0 1 0
1 0 0 0	0	0	1 0 1 1
1 0 0 1	1	0	1 1 0 0

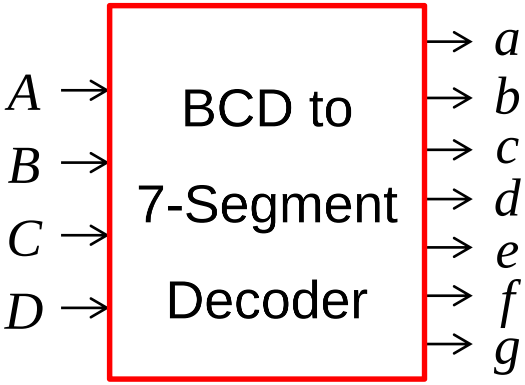
BCD to 7-Segment Decoder

- Seven-Segment Display:
  - Made of Seven segments: light-emitting diodes (LED)
  - Found in electronic devices: such as clocks, calculators, etc.









- BCD to 7-Segment Decoder
  - Accepts as input a BCD decimal digit (0 to 9)
  - Generates output to the seven LED segments to display the BCD digit
  - Each segment can be turned on or off separately

Designing a BCD to 7-Segment Decoder

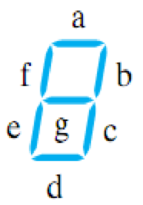
1. Specification:

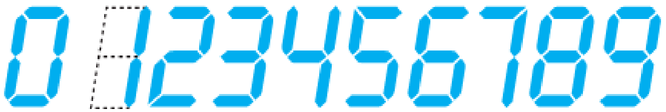
- Input: 4-bit BCD (*A*, *B*, *C*, *D*)
- Output: 7-bit (*a*, *b*, *c*, *d*, *e*, *f*, *g*)
- Display should be OFF for

  Non-BCD input codes

2. Formulation

- Done with a truth table
- Output is zero for 1010 to 1111





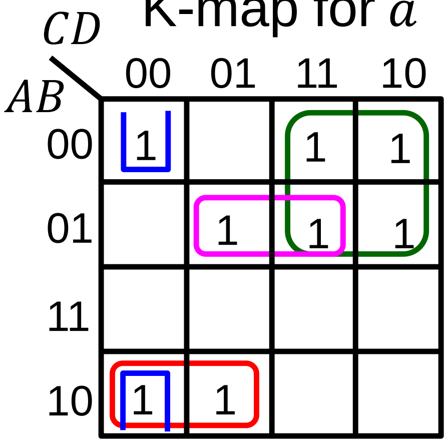
Truth Table

BCD input				7-Segment decoder						
A	B	C	D	a	b	c	d	e	f	g
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	1	0	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	1	0	1	1
1010 to 1111				0	0	0	0	0	0	0

Designing a BCD to 7-Segment Decoder

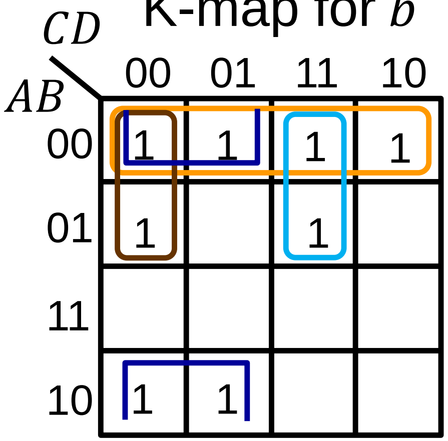
3. Logic Minimization Using K-Maps

K-map for a



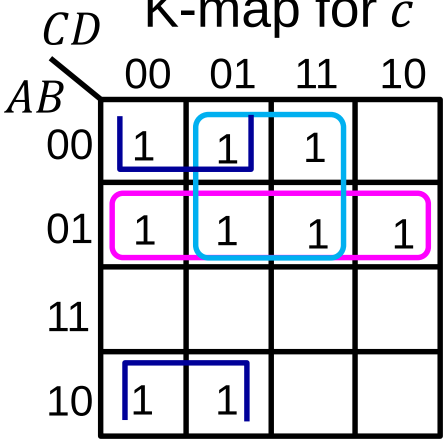
AB \ CD	00	01	11	10
00	1		1	1
01		1	1	1
11				
10	1	1		

K-map for b



AB \ CD	00	01	11	10
00	1	1	1	1
01	1		1	
11				
10	1	1		

K-map for c



AB \ CD	00	01	11	10
00	1	1	1	
01	1	1	1	1
11				
10	1	1		

$$a = A'C + A'BD + AB'C' + B'C'D'$$

$$b = A'B' + B'C' + A'C'D' + A'CD$$

$$c = A'B + B'C' + A'D$$

Extracting common terms

Let $T_1 = A'B$, $T_2 = B'C'$, $T_3 = A'D$

Optimized Logic Expressions

$$a = A'C + T_1\, D + T_2\, A + T_2\, D'$$

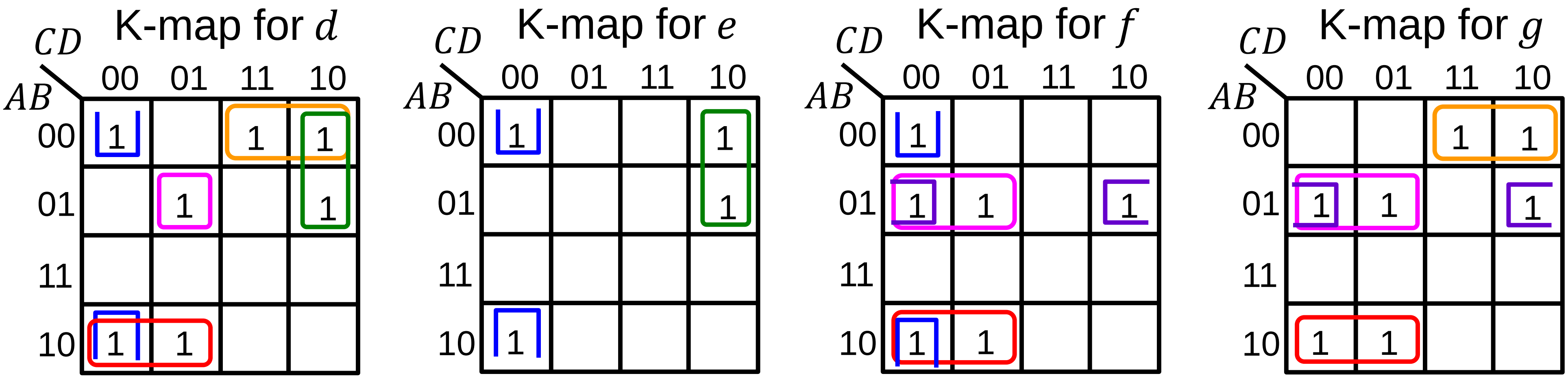
$$b = A'B' + T_2 + A'C'D' + T_3 C$$

$$c = T_1 + T_2 + T_3$$

T_1, T_2, T_3 are **shared gates**

Designing a BCD to 7-Segment Decoder

3. Logic Minimization Using K-Maps



K-map for d

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

K-map for e

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

K-map for f

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

K-map for g

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

Common AND Terms

➔ Shared Gates

$T_4 = AB'C'$, $T_5 = B'C'D'$

$T_6 = A'B'C$, $T_7 = A'CD'$

$T_8 = A'BC'$, $T_9 = A'BD'$

Optimized Logic Expressions

$d = T_4 + T_5 + T_6 + T_7 + T_8\, D$

$e = T_5 + T_7$

$f = T_4 + T_5 + T_8 + T_9$

$g = T_4 + T_6 + T_8 + T_9$

Designing a BCD to 7-Segment Decoder

4. Technology Mapping

Many Common AND terms: T_0 thru T_9

$T_0 = A'C$, $T_1 = A'B$, $T_2 = B'C'$

$T_3 = A'D$, $T_4 = AB'C'$, $T_5 = B'C'D'$

$T_6 = A'B'C$, $T_7 = A'CD'$

$T_8 = A'BC'$, $T_9 = A'BD'$

Optimized Logic Expressions

$a = T_0 + T_1\,D + T_4 + T_5$

$b = A'B' + T_2 + A'C'D' + T_3C$

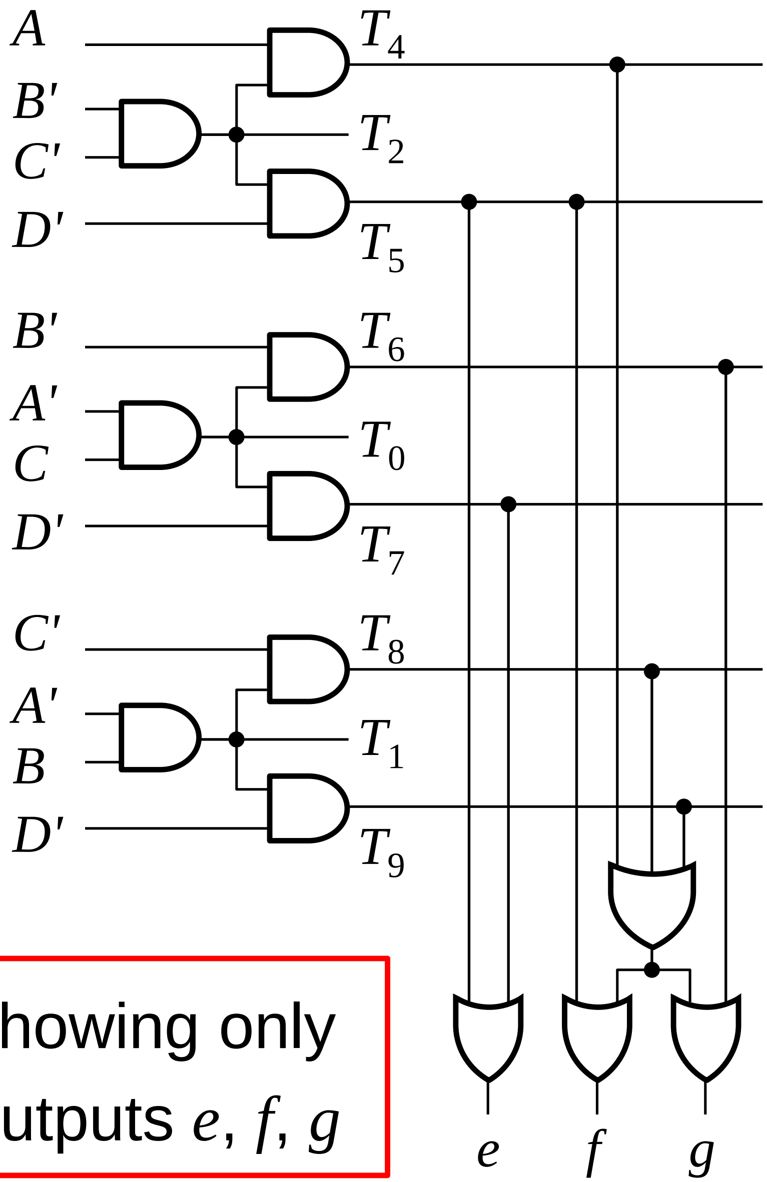
$c = T_1 + T_2 + T_3$

$d = T_4 + T_5 + T_6 + T_7 + T_8\,D$

$e = T_5 + T_7$

$f = T_4 + T_5 + T_8 + T_9$

$g = T_4 + T_6 + T_8 + T_9$



Showing only Outputs e, f, g

Verification Methods

- Manual Logic Analysis
  - Find the logic expressions and truth table of the final circuit
  - Compare the final circuit truth table against the specified truth table
  - Compare the circuit output expressions against the specified expressions
  - Tedious for large designs + Human Errors
- Simulation
  - Simulate the final circuit, possibly written in HDL (such as Verilog)
  - Write a test bench that automates the verification process
  - Generate test cases for ALL possible inputs (exhaustive testing)
  - Verify the output correctness for ALL input test cases
  - Exhaustive testing can be very time consuming for many inputs

Modeling the 7-Segment Decoder

```
// Module BCD_to_7Segment: Modeled using continuous assignment
module BCD_to_7Segment (input A,B,C,D, output a,b,c,d,e,f,g);
  wire T0, T1, T2, T3, T4, T5, T6, T7, T8, T9;
  assign T0=~A&C;   assign T1=~A&B;    assign T2=~B&~C;
  assign T3=~A&D;   assign T4=T2&A;    assign T5=T2&~D;
  assign T6=T0&~B;  assign T7=T0&~D;   assign T8=T1&~C;
  assign T9=T1&~D;
  assign a = T0 | T1&D | T4 | T5;
  assign b = ~A&~B | T2 | ~A&~C&~D | T3&C;
  assign c = T1 | T2 | T3;
  assign d = T4 | T5 | T6 | T7 | T8&D;
  assign e = T5 | T7;
  assign f = T4 | T5 | T8 | T9;
  assign g = T4 | T6 | T8 | T9;
endmodule
```

assign statements can appear in any order

Testing the BCD to 7-Segment Decoder

```
module Test_BCD_to_7Segment;   // No need for Ports

  reg  A, B, C, D;             // variable inputs
  wire a, b, c, d, e, f, g;    // wire outputs
  // Instantiate the module to be tested
  BCD_to_7Segment BCD_7Seg (A, B, C, D, a, b, c, d, e, f, g);

  initial begin                // initial block
    A=0; B=0; C=0; D=0;        // at t=0
    #200 $finish;              // at t=200 finish simulation
  end                          // end of initial block

  always #10 D=~D;             // invert D every 10 time units
  always #20 C=~C;             // invert C every 20 time units
  always #40 B=~B;             // invert B every 40 time units
  always #80 A=~A;             // invert A every 80 time units

endmodule
```

The `initial` and `always` Blocks

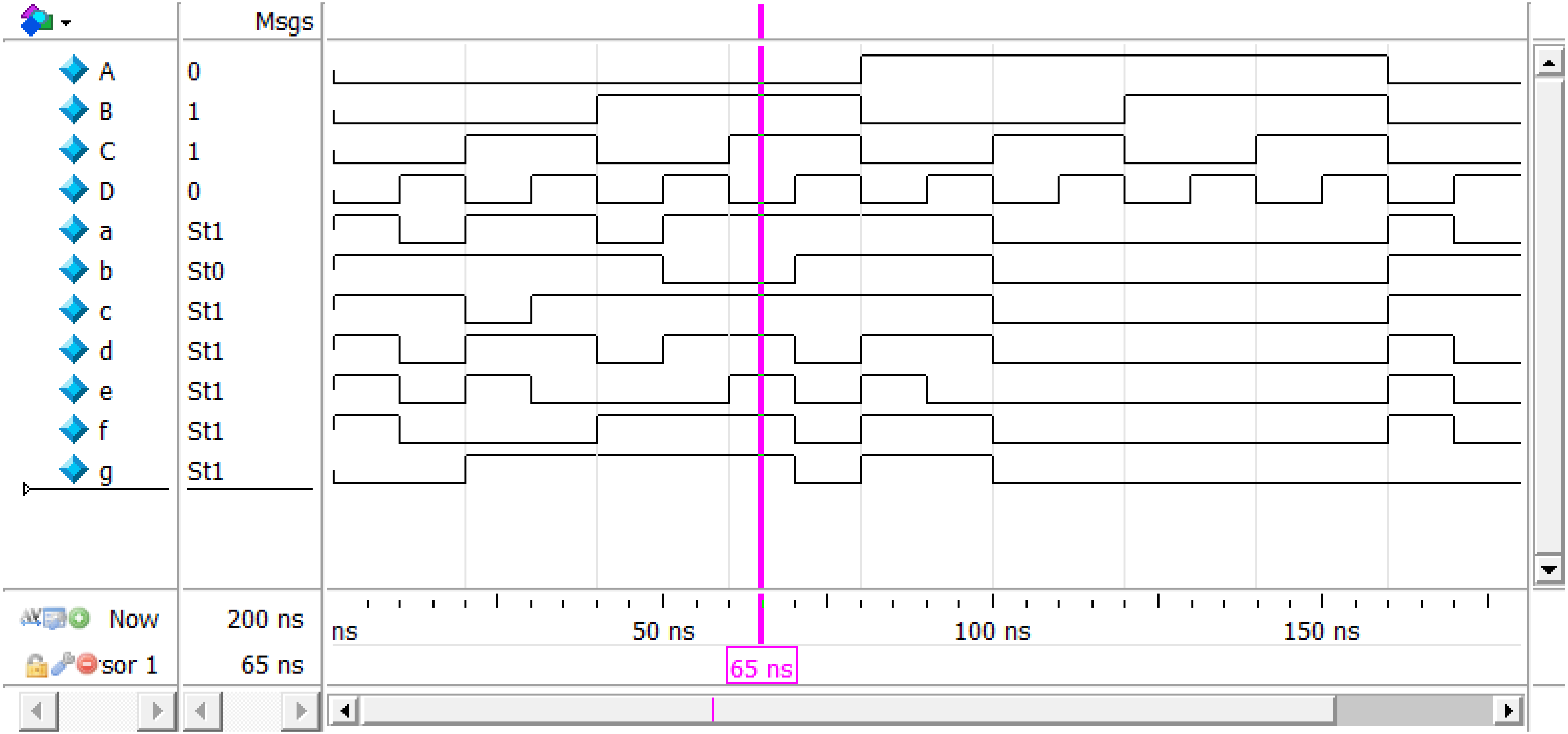
- There are two types of **procedural** blocks in Verilog

1. The **`initial`** block
   - Executes the enclosed statement(s) once
2. The **`always`** block
   - Executes the enclosed statement(s) repeatedly until simulation terminates

- The body of the **`initial`** and **`always`** blocks is **procedural**
  - Can enclose one or more **procedural statements**
  - Procedural statements surrounded by **`begin`** … **`end`** execute **sequentially**
  - **`#delay`** is used to delay the execution of the procedural statement
- Procedural blocks can appear in any order inside a module
- Multiple procedural blocks run **in parallel** inside the simulator

Simulator Waveforms

All sixteen input test cases of *A*, *B*, *C*, *D* are generated between *t*=0 and *t*=160ns. Verify that outputs *a* to *g* match the truth table.



Hierarchical Design

- Why Hierarchical Design?

  To simplify the implementation of a complex circuit

- What is Hierarchical Design?

  Decompose a complex circuit into smaller pieces called blocks

  Decompose each block into even smaller blocks

  Repeat as necessary until the blocks are small enough

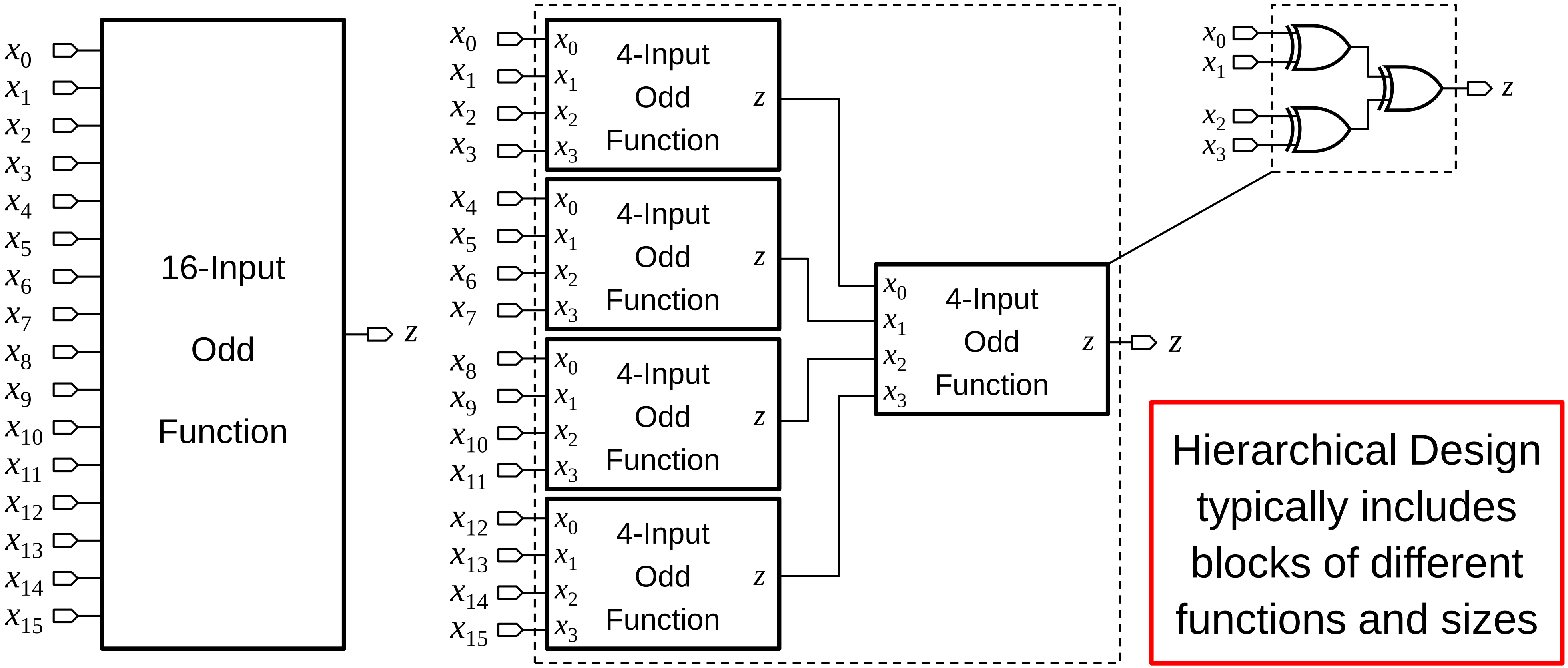
  Any block not decomposed is called a primitive block

  The hierarchy is a tree of blocks at different levels

- The blocks are verified and well-document
- They are placed in a library for future use

Example of Hierarchical Design

- Top Level: 16-input odd function: 16 inputs, one output
  - Implemented using Five 4-input odd functions
- Second Level: 4-input odd function that uses three XOR gates


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
16-Input
Odd
Function
z
x_0 x_1 x_2 x_3
4-Input
Odd
Function
z
x_4 x_5 x_6 x_7
x_8 x_9 x_{10} x_{11}
x_{12} x_{13} x_{14} x_{15}
z

Hierarchical Design typically includes blocks of different functions and sizes

Top-Down versus Bottom-Up Design

- A **top-down design** proceeds from a high-level specification to a more and more detailed design by decomposition and successive refinement
- A **bottom-up design** starts with detailed primitive blocks and combines them into larger and more complex functional blocks
- Design usually proceeds top-down to a known set of building blocks, ranging from complete processors to primitive logic gates

Hierarchical Design in Verilog

```
// Module Odd_4: 4-input Odd function uses three xor gates
module Odd_4 (input [0:3] x, output z);
  wire [0:1] w;
  xor g1(w[0], x[0], x[1]);
  xor g2(w[1], x[2], x[3]);
  xor g3(z, w[0], w[1]);
endmodule

// Module Odd_16: 16-input Odd function
module Odd_16 (input [0:15] x, output z);
  wire [0:3] w;
  Odd_4 block0 (x[0:3], w[0]);
  Odd_4 block1 (x[4:7], w[1]);
  Odd_4 block2 (x[8:11],  w[2]);
  Odd_4 block3 (x[12:15], w[3]);
  Odd_4 block4 (w[0:3], z);
endmodule
```

Five instances of the **Odd_4** module

Bit Vectors in Verilog

- A Bit Vector is multi-bit declaration that uses a single name
- A Bit Vector is specified as a Range **`[msb:lsb]`**
- **msb** is **most-significant bit** and **lsb** is **least-significant bit**
- Examples:

```
input [0:15] x;      // x is a 16-bit input vector
wire [0:3] w;        // Bit 0 is most-significant bit
reg [7:0] a;         // Bit 7 is most-significant bit
```

- **Bit select**: **`w[1]`** is bit **`1`** of vector **`w`**
- **Part select**: **`x[8:11]`** is a 4-bit select of **`x`** with range **`[8:11]`**
- The part select range must be consistent with vector declaration

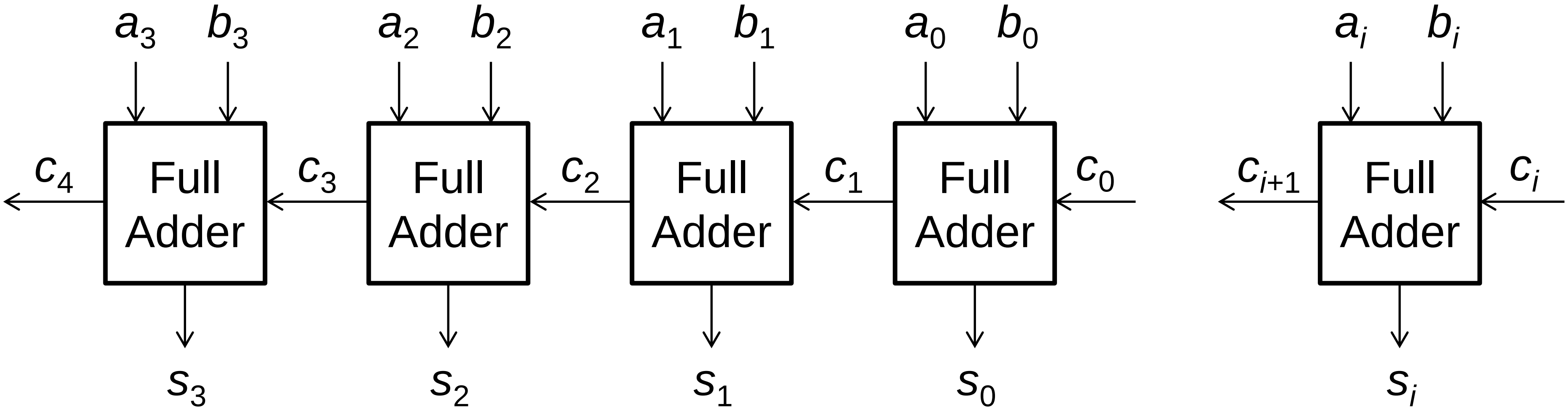
Testing Hierarchical Design

- Exhaustive testing can be very time consuming (or impossible)
  - For a 16-bit input, there are 2^{16} = 65,536 test cases (combinations)
  - For a 32-bit input, there are 2^{32} = 4,294,967,296 test cases
  - For a 64-bit input, there are 2^{64} = 18,446,744,073,709,551,616 test cases!
- Testing a hierarchical design requires a different strategy
- Test each block in the hierarchy separately
  - For smaller blocks, exhaustive testing can be done
  - It is easier to detect errors in smaller blocks before testing complete circuit
- Test the top-level design by applying selected test inputs
- Make sure that the test inputs exercise all parts of the circuit

Iterative Design

- Using **identical copies** of a smaller circuit to build a large circuit
- Example: Building a 4-bit adder using 4 copies of a full-adder
- The **cell** (iterative block) is a **full adder**

  Adds 3 bits: a_i, b_i, c_i, Computes: Sum s_i and Carry-out c_{i+1}
- Carry-out of cell *i* becomes carry-in to cell (*i*+1)



Full Adder

- Full adder adds 3 bits: **a**, **b**, and **c**
- Two output bits:

  1. Carry bit: **cout**
  2. Sum bit: **sum**

- Sum bit is 1 if the number of 1's in the input is odd (odd function)

  $$\mathbf{sum} = (a \oplus b) \oplus c$$

- Carry bit is 1 if the number of 1's in the input is 2 or 3

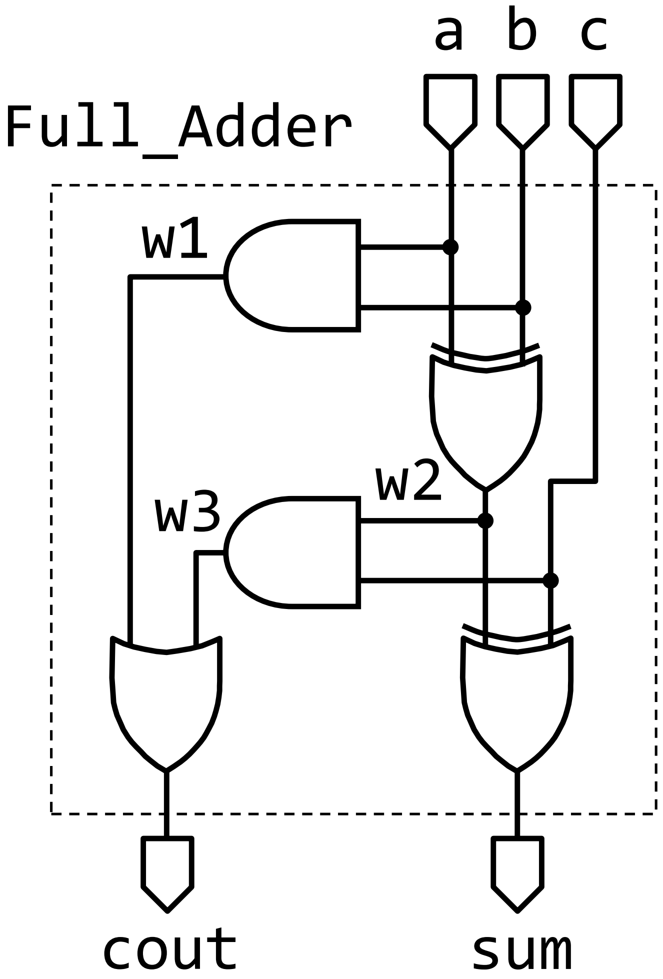
  $$\mathbf{cout} = a \cdot b + (a \oplus b) \cdot c$$

Truth Table

a b c	cout	sum
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 Module (Gate-Level Description)

```
module Full_Adder(input a, b, c, output cout, sum);

  wire w1, w2, w3;

  and (w1, a, b);

  xor (w2, a, b);

  and (w3, w2, c);

  xor (sum, w2, c);

  or  (cout, w1, w3);

endmodule
```



16-Bit Adder with Array Instantiation

```
// Input  ports: 16-bit a and b, 1-bit cin (carry input)
// Output ports: 16-bit sum, 1-bit cout (carry output)

module Adder_16 (input  [15:0] a, b, input cin,
                 output [15:0] sum, output cout);

  wire [16:0] c;                 // carry bits
  assign c[0] = cin;             // carry input
  assign cout = c[16];           // carry output

  // Instantiate an array of 16 Full Adders
  // Each instance [i] is connected to bit select [i]

  Full_Adder adder [15:0] (a[15:0], b[15:0], c[15:0],
                           c[16:1], sum[15:0]);
endmodule
```

Array Instantiation of identical modules by a single statement

16-Bit Adder with Continuous Assignment

```
// Input  ports: 16-bit a and b, 1-bit cin (carry input)
// Output ports: 16-bit sum, 1-bit cout (carry output)

module Adder_16b (input  [15:0] a, b, input cin,
                  output [15:0] sum, output cout);

  wire [16:0] c;               // carry bits
  assign c[0] = cin;           // carry input
  assign cout = c[16];         // carry output

  // assignment of 16-bit vectors
  assign sum[15:0] = (a[15:0] ^ b[15:0]) ^ c[15:0];
  assign c[16:1]   = (a[15:0] & b[15:0]) |
                     (a[15:0] ^ b[15:0]) & c[15:0];
endmodule
```

Testing the 16-bit Adder

- Exhaustive testing: 2^{16} × 2^{16} × 2 = 8,589,934,592 test cases
- Let us choose only: 3 × 3 × 2 = 18 test cases
- Input test cases for **`a[15:0]`** = **`'h158A, 'h52AF, 'hB903`**

  Chosen randomly with values shown in hexadecimal

  `'h158A` (hexadecimal) = **`'b0001_0101_1000_1010`** (binary)

  Underscores are ignored (used to enhance readability)
- Input test cases for **`b[15:0]`** = **`'h7095, 'h9A4E, 'hC6BD`**

  Also chosen randomly with values shown in hexadecimal

  Radix symbol: **`'b`** (binary), **`'o`** (octal), **`'d`** (decimal), **`'h`** (hex)
- Input test cases for **`cin`** = **`0, 1`**

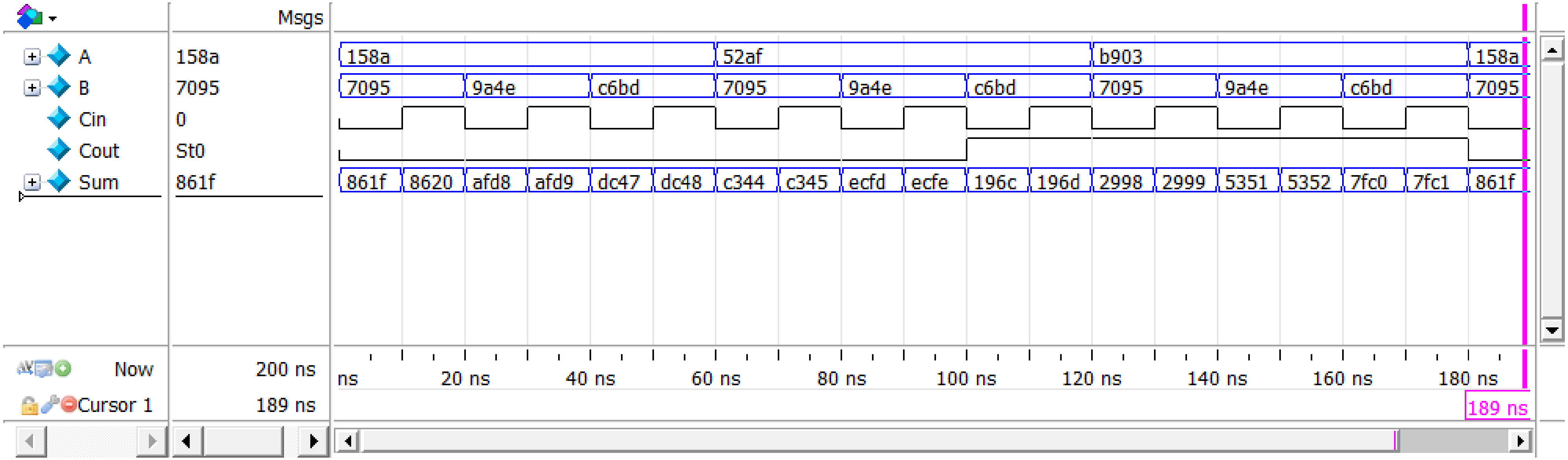
Writing a Test bench for the 16-bit Adder

```
module Test_Adder_16;                   // Test bench for Adder_16

  reg [15:0] A, B; reg Cin;             // Data and Carry inputs
  wire [15:0] Sum; wire Cout;           // Sum and Carry outputs

  Adder_16 Test (A, B, Cin, Sum, Cout); // Instantiate 16-bit adder

  initial begin
    A='h158A; B='h7095; Cin=0;          // Initialize A, B, Cin
    #10 Cin=1;                          // t=10, Change Cin
    #10 B='h9A4E; Cin=0;                // t=20, Change B and Cin
    #10 Cin=1;                          // t=30, Change Cin
    #10 B='hC6BD; Cin=0;                // t=40, Change B and Cin
    #10 Cin=1;                          // t=50, Change Cin
    #10 A='h52AF; B='h7095; Cin=0;      // t=60, Change A, B and Cin
    #10 Cin=1;                          // t=70, Change Cin
    #10 B='h9A4E; Cin=0;                // t=80, Change B and Cin
    . . .                               // more test cases
    #10 $finish;                        // End simulation
  end

endmodule
```

Generating Test Cases in parallel with always

```
module Test_Adder_16;                   // Test bench for Adder_16

  reg [15:0] A, B; reg Cin;             // Data and Carry inputs
  wire [15:0] Sum; wire Cout;           // Sum and Carry outputs

  Adder_16 Test (A, B, Cin, Sum, Cout); // Instantiate 16-bit adder

  initial begin
    A='h158A; B='h7095; Cin=0;          // Initialize A, B, Cin
    #200 $finish;                       // At t=200 end simulation
  end

  always begin                          // Change A every 60 ns
    #60 A='h52AF; #60 A='hB903; #60 A='h158A;
  end

  always begin                          // Change B every 20 ns
    #20 B='h9A4E; #20 B='hC6BD; #20 B='h7095;
  end

  always #10 Cin = ~Cin;                // Invert Cin every 10 ns

endmodule
```

Simulator Waveforms


Msgs
A 158a
B 7095
Cin 0
Cout St0
Sum 861f
158a 52af b903 158a
7095 9a4e c6bd 7095 9a4e c6bd 7095 9a4e c6bd 7095
861f 8620 afd8 afd9 dc47 dc48 c344 c345 ecfd ecfe 196c 196d 2998 2999 5351 5352 7fc0 7fc1 861f
Now 200 ns
Cursor 1 189 ns
ns 20 ns 40 ns 60 ns 80 ns 100 ns 120 ns 140 ns 160 ns 180 ns
189 ns

- ❖ The values of A, B, and Sum are shown in hexadecimal

  Can change the radix to binary, octal, and decimal

- ❖ The values of Sum and Cout can be verified easily

- ❖ All 18 test cases of A, B, and Cin are generated (t=0 to 180 ns)