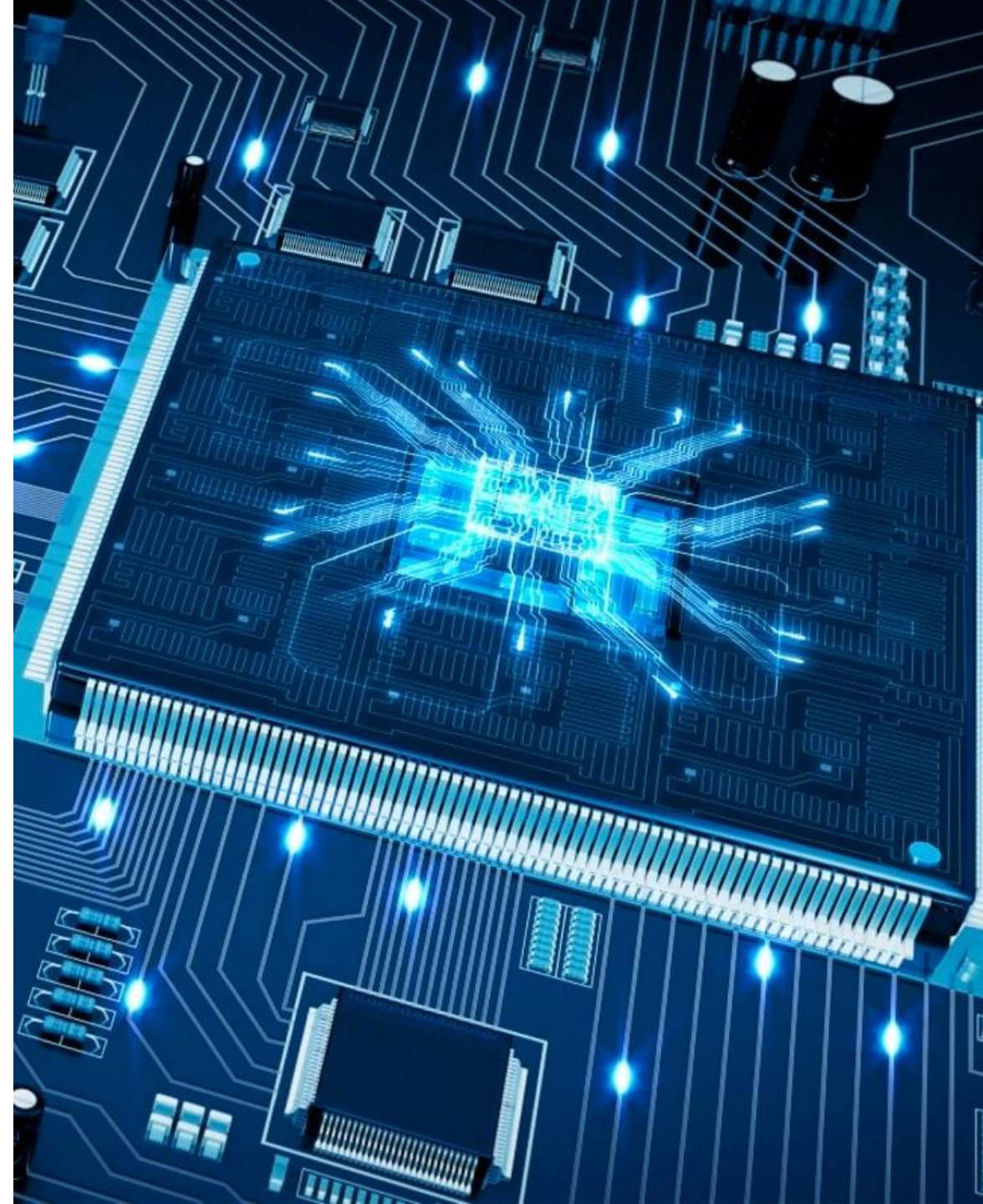


Verilog

Abdallah El Ghamry

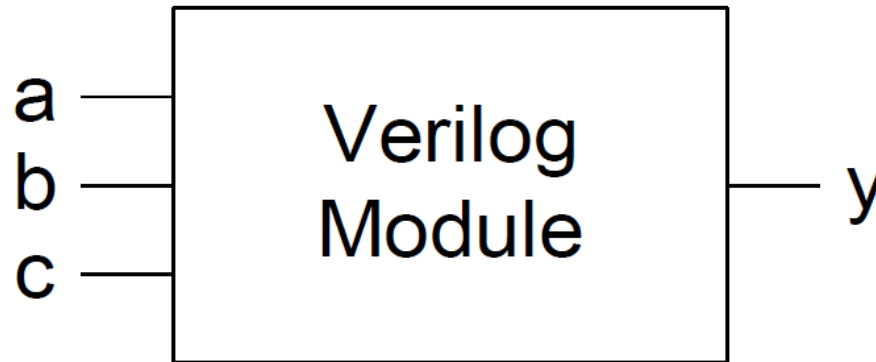


Simulation

- Humans routinely make **mistakes**. Such errors in hardware designs are called **bugs**.
- Eliminating the bugs from a digital system is obviously important, especially when customers are **paying money** and lives depend on the correct operation.
- Testing a system in the laboratory is **time-consuming**.
- Discovering the cause of errors **in the lab** can be extremely difficult.
- Correcting errors after the system is built can be **devastatingly expensive**.
- Intel's infamous FDIIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a **total cost of \$475 million**.

Modules

- A **block of hardware with inputs and outputs** is called a module.
- An AND gate, a multiplexer, and an adder are all examples of hardware modules.

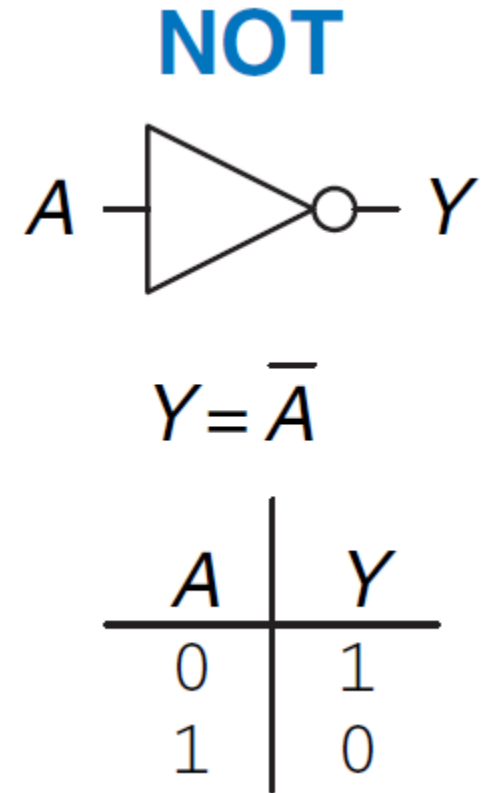


```
module ModuleName(a, b, c, y);  
    ...  
    ...  
endmodule
```

NOT Gate

The NOT gate's output is the inverse of its input.

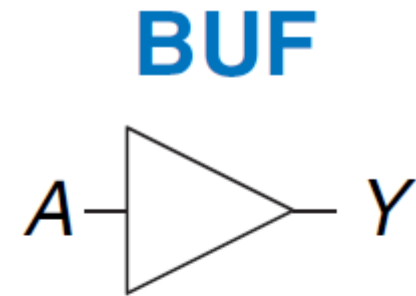
```
module NOT(A, Y);  
    input A;  
    output Y;  
  
    assign Y = ~A;  
endmodule
```



Buffer Gate

The buffer gate simply copies the input to the output.

```
module BUF(A, Y);  
    input A;  
    output Y;  
  
    assign Y = ~(~A);  
endmodule
```



$$Y = A$$

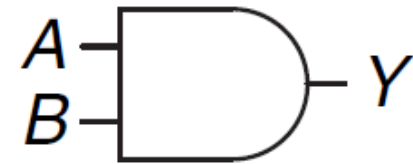
A	Y
0	0
1	1

AND Gate

A logic gate that produces a HIGH output only when all of the inputs are HIGH.

```
module AND2(A, B, Y);  
    input A, B;  
    output Y;  
  
    assign Y = A & B;  
endmodule
```

AND



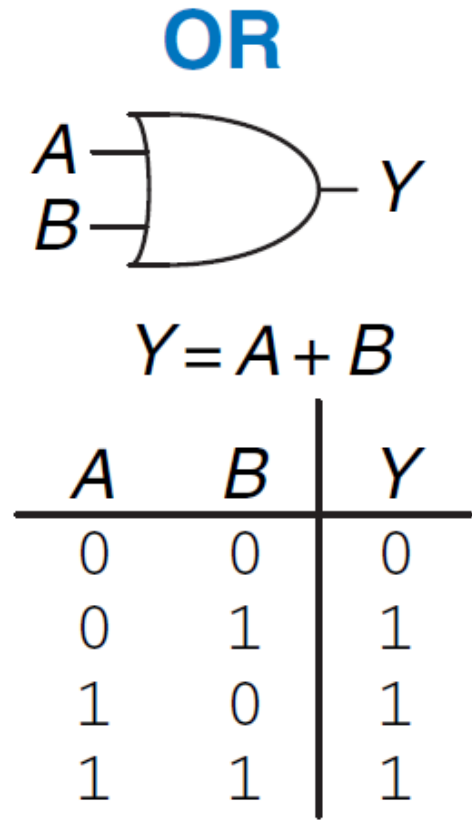
$$Y = AB$$

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

OR Gate

A logic gate that produces a HIGH output when one or more inputs are HIGH.

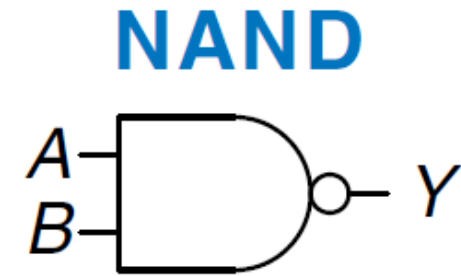
```
module OR2(A, B, Y);  
    input A, B;  
    output Y;  
  
    assign Y = A | B;  
endmodule
```



NAND Gate

A logic gate that produces a LOW output only when all the inputs are HIGH.

```
module NAND2(A, B, Y);  
    input A, B;  
    output Y;  
  
    assign Y = ~(A & B);  
endmodule
```



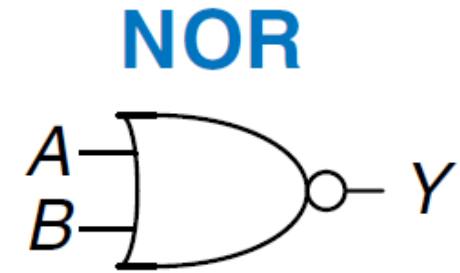
$$Y = \overline{AB}$$

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

NOR Gate

A logic gate in which the output is LOW when one or more of the inputs are HIGH.

```
module NOR2(A, B, Y);  
    input A, B;  
    output Y;  
  
    assign Y = ~(A | B);  
endmodule
```



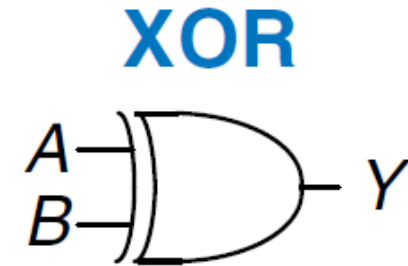
$$Y = \overline{A + B}$$

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

Exclusive-OR (XOR) Gate

A logic gate that produces a HIGH output only when its two inputs are at opposite levels.

```
module XOR2(A, B, Y);  
    input A, B;  
    output Y;  
  
    assign Y = A ^ B;  
endmodule
```



$$Y = A \oplus B$$

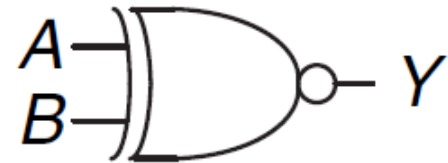
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Exclusive-NOR (XNOR) Gate

A logic gate that produces a LOW only when the two inputs are at opposite levels.

```
module XNOR2(A, B, Y);  
    input A, B;  
    output Y;  
  
    assign Y = ~(A ^ B);  
endmodule
```

XNOR



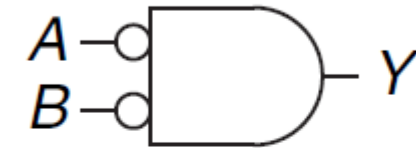
$$Y = \overline{A \oplus B}$$

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

Negative AND

```
module NegAND2(A, B, Y1, Y2);  
    input A, B;  
    output Y1, Y2;  
  
    assign Y1 = ~A & ~B;  
    assign Y2 = ~(A | B);  
endmodule
```

NOR



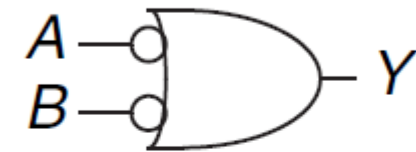
$$Y = \overline{A + B} = \bar{A} \bar{B}$$

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

Negative OR

```
module NegOR2(A, B, Y1, Y2);  
    input A, B;  
    output Y1, Y2;  
  
    assign Y1 = ~A | ~B;  
    assign Y2 = ~(A & B);  
endmodule
```

NAND



$$Y = \overline{AB} = \overline{A} + \overline{B}$$

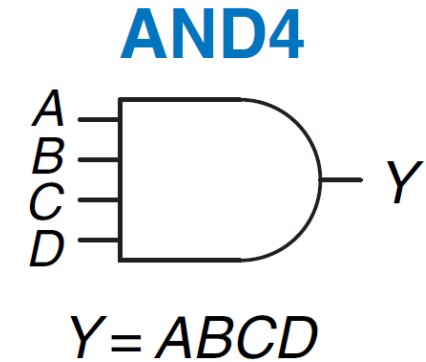
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

Quiz: Multiple-Input Gates

Write SystemVerilog code to implement the following functions in hardware:

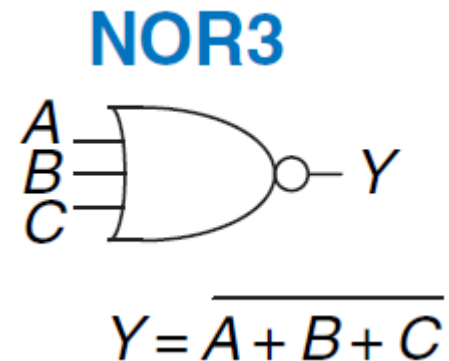
- Four-input AND gate.

```
module AND4(A, B, C, D, Y);  
    input A, B, C, D;  
    output Y;  
    assign Y = A & B & C & D;  
endmodule
```

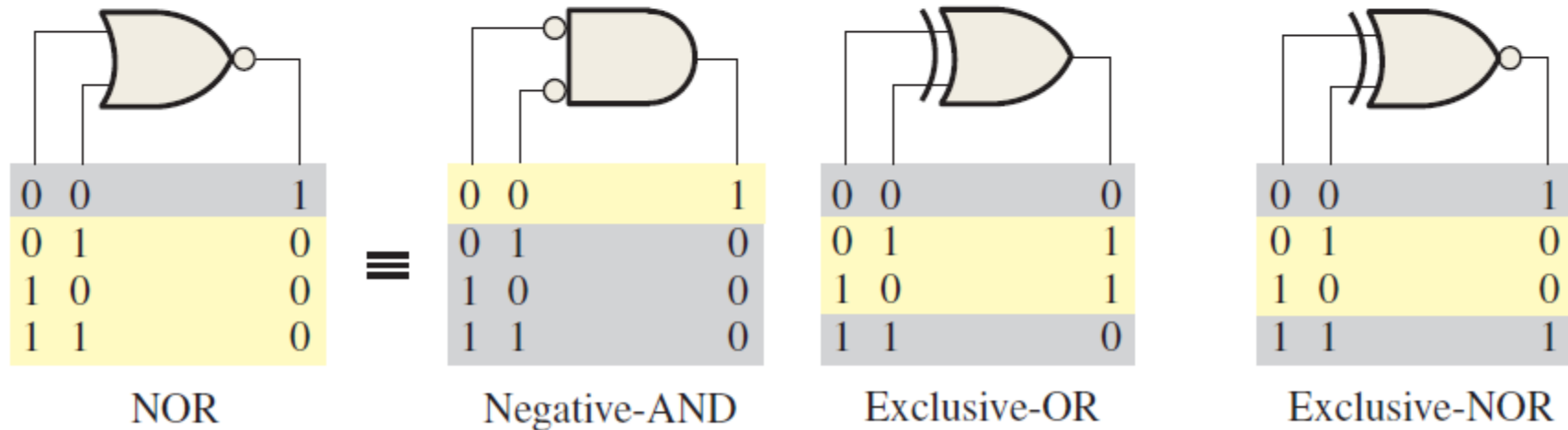
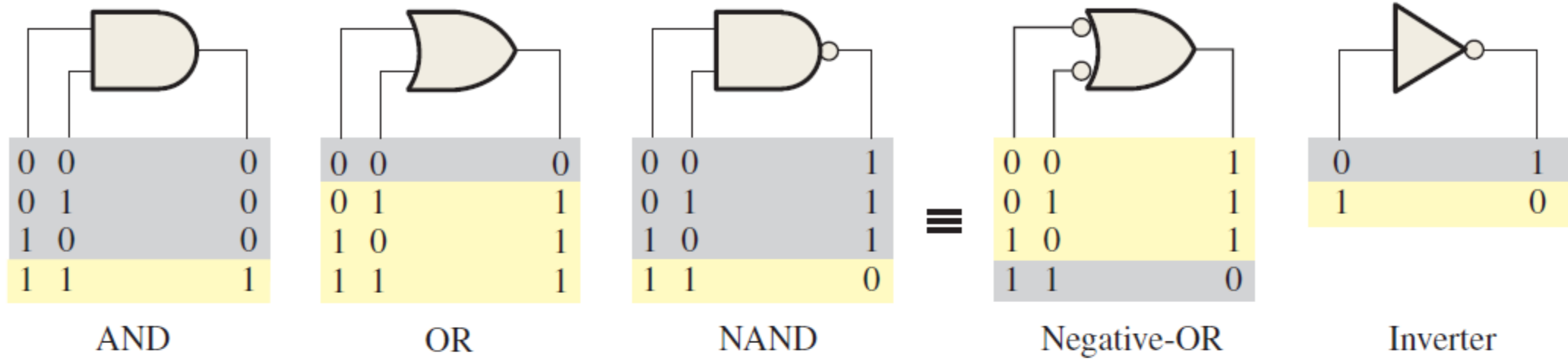


- Three-input NOR gate.

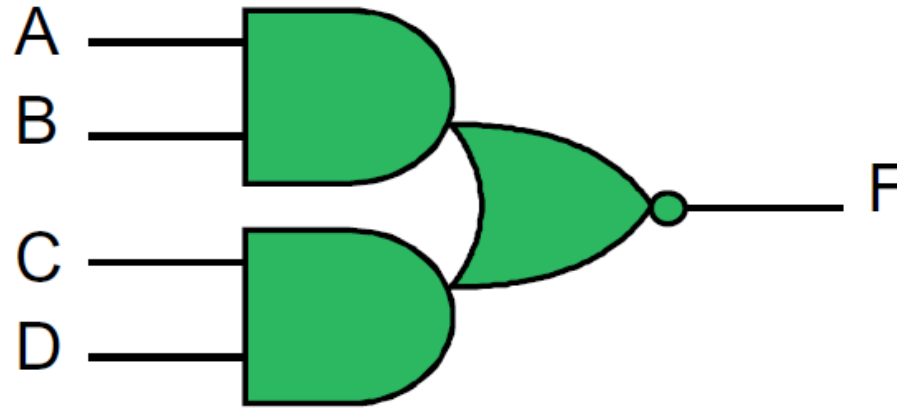
```
module NOR3(A, B, C, Y);  
    input A, B, C;  
    output Y;  
    assign Y = ~(A | B | C);  
endmodule
```



Logic Gates Summary



AND-OR-Inverter (AOI) Gate



```
module AOI (A, B, C, D, F);  
    input A, B, C, D;  
    output F;  
  
    assign F = ~((A & B) | (C & D));  
endmodule
```


Rules and Regulations

All definitions and statements go inside a module

```
// An and-or-invert gate
```

A comment

```
module AOI (A, B, C, D, F);
```

```
    input A, B, C, D;
```

Case sensitive names

```
    output F;
```

; at end of definition/statement

```
    assign F = ~( (A & B) | (C & D) );
```

```
/*
```

```
These lines are ignored
```

```
by the compiler
```

A block comment

```
*/
```

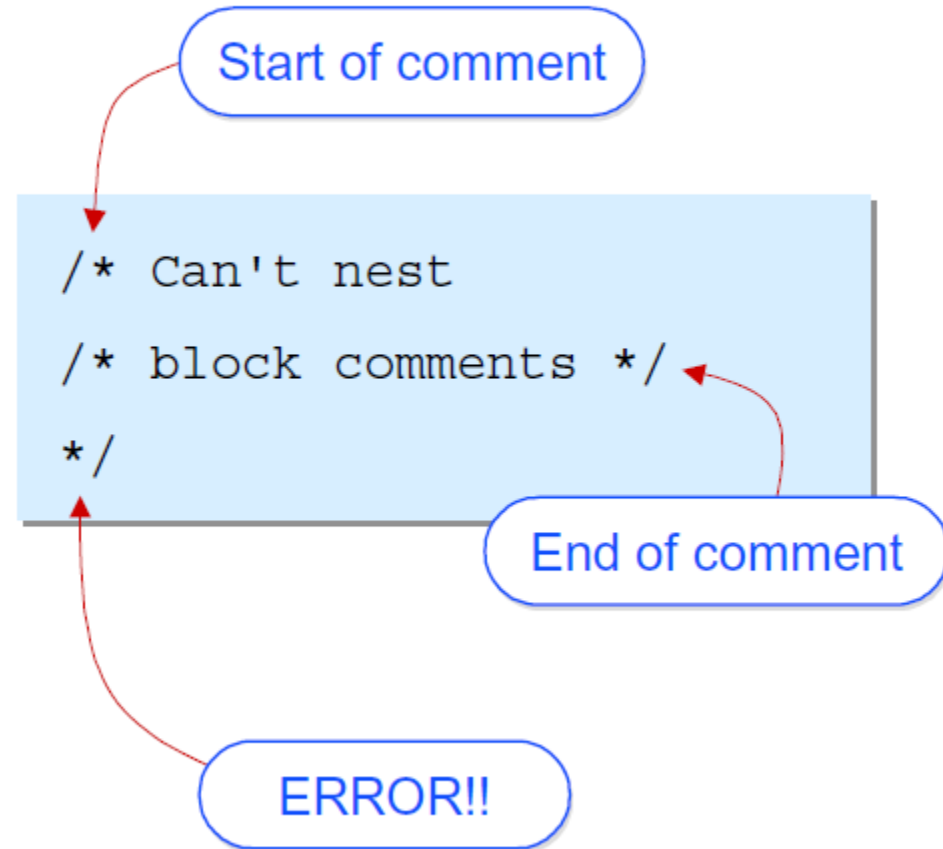
```
endmodule
```

Lower case keywords

Single-line vs. Block Comments

```
// Comment out  
// a number of lines  
// using single-line  
// comments
```

```
/* Can nest  
// single-line comments  
// within block comments  
*/
```



Names

◆ Identifiers

```
AB      Ab      aB      ab      // different!  
G4X$6_45_123$  
Y       Y_      _Y              //different!
```

◆ Illegal!

```
4plus4      $1
```

◆ Escaped identifiers (terminate with white space)

```
\4plus4      \$1      \a+b£$%^&* (
```

◆ Keywords

```
and      default      event      function      wire
```

Names

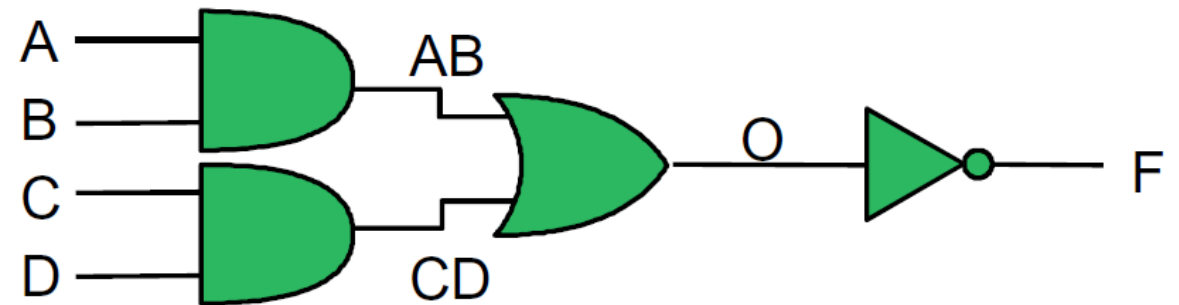
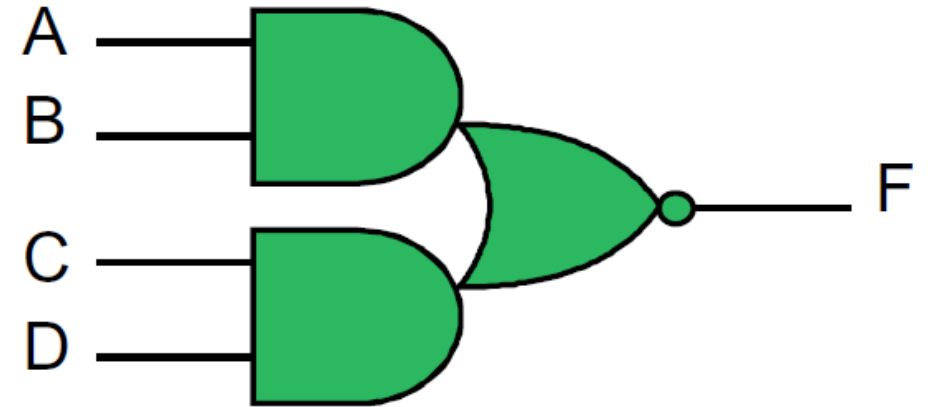
- Names of modules, ports etc. are properly called **identifiers**. Identifiers can be of any length, and consist of letters, digits, underscores (_) and dollars (\$). **The first character must be a letter or an underscore.**
- The **case of identifiers is significant**, so two identifiers that differ only in case do in fact name different items.
- There are a large number of **reserved identifiers that cannot be declared as names.**

SystemVerilog Operator Precedence

	Op	Meaning
H i g h e s t	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
L o w e s t	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	, ~	OR, NOR
	?:	Conditional

Wires

```
module AOI2 (A, B, C, D, F);  
  input A, B, C, D;  
  output F;  
  wire AB, CD, O;  
  
  assign AB = A & B;  
  assign CD = C & D;  
  assign O = AB | CD;  
  assign F = ~O;  
endmodule
```



Wire Assignments

```
module AOI2 (A, B, C, D, F);  
  input A, B, C, D;  
  output F;  
  wire AB, CD, O;  
  
  assign AB = A & B;  
  assign CD = C & D;  
  assign O = AB | CD;  
  assign F = ~O;  
endmodule
```

```
module AOI2 (A, B, C, D, F);  
  input A, B, C, D;  
  output F;  
  
  wire AB = A & B;  
  wire CD = C & D;  
  wire O = AB | CD;  
  wire F = ~O;  
endmodule
```

Wire Assignments

```
module AOI (A, B, C, D, F);  
    input A, B, C, D;  
    output F;  
  
    /*  
    wire F;  
    wire AB, CD, O;  
  
    assign AB = A & B;  
    assign CD = C & D;  
    assign O = AB | CD;  
    assign F = ~O;  
    */  
  
    // Equivalent...  
  
    wire AB = A & B;  
    wire CD = C & D;  
    wire O = AB | CD;  
    wire F = ~O;  
  
endmodule
```

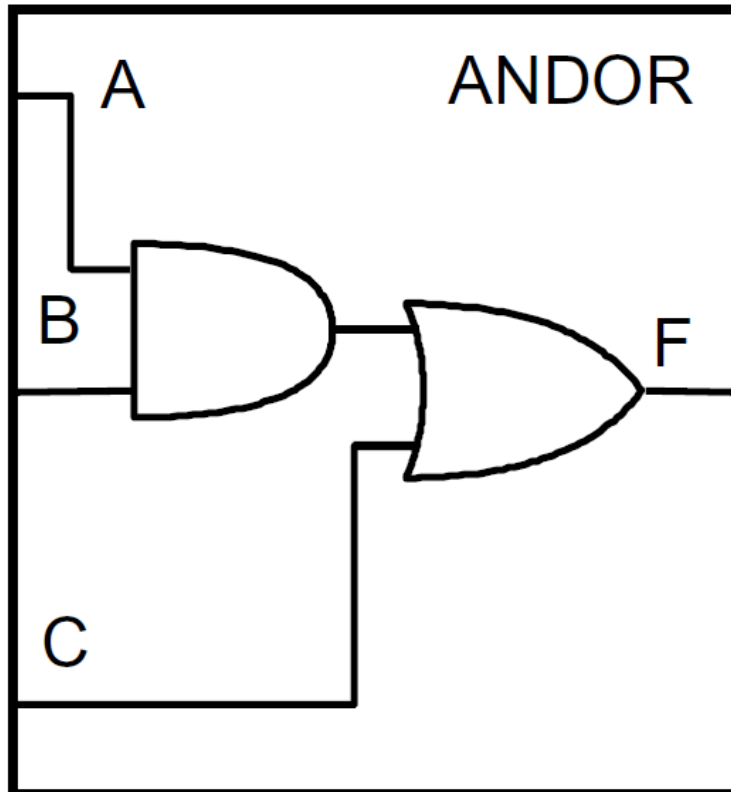
or

```
assign AB = A & B,  
        CD = C & D,  
        O  = AB | CD,  
        F  = ~O;
```

or

```
wire AB = A & B,  
      CD = C & D,  
      O  = AB | CD,  
      F  = ~O;
```


ANDOR



```
module ANDOR (A, B, C, F);  
  input A, B, C;  
  output F;  
  
  assign F = (A & B) | C;  
endmodule
```

```
module ANDOR (A, B, C, F);  
  input A, B, C;  
  output F;  
  wire AB;  
  assign AB = A & B;  
  assign F = AB | C;  
endmodule
```

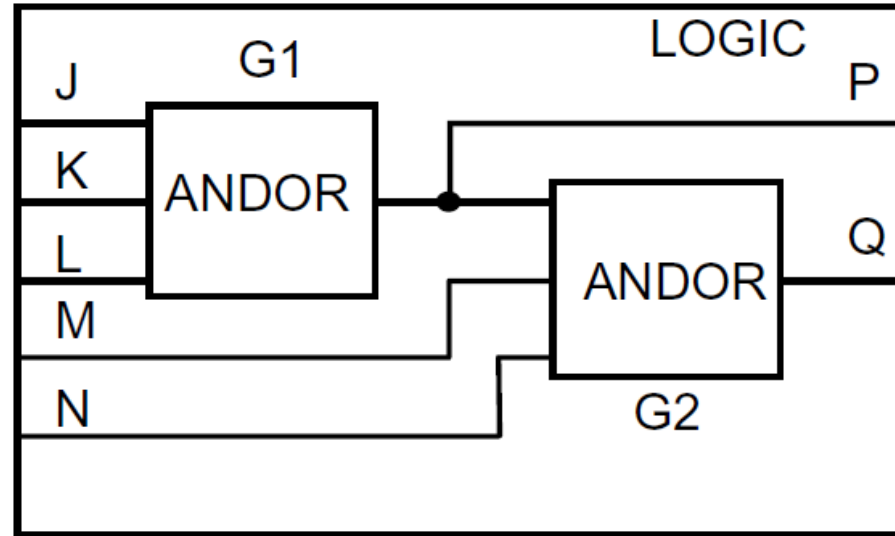
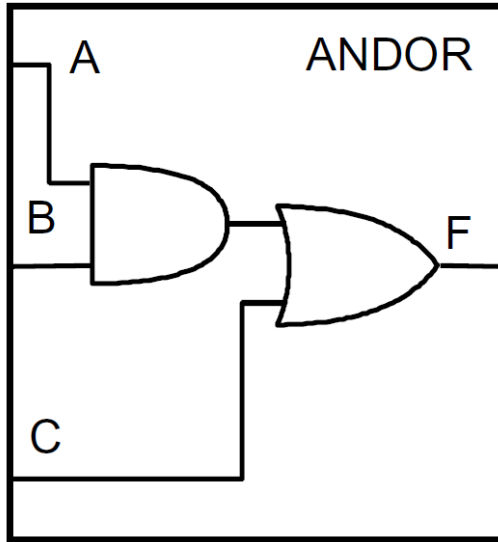
The Three-Y's

Hierarchy involves **dividing a system into modules**, then further subdividing each of these modules until the pieces are **easy to understand**.

Modularity states that the **modules have well-defined functions and interfaces**, so that they connect together easily without unanticipated side effects.

Regularity seeks **uniformity among the modules**. Common modules are **reused many times**, reducing the number of distinct modules that must be designed.

Hierarchy



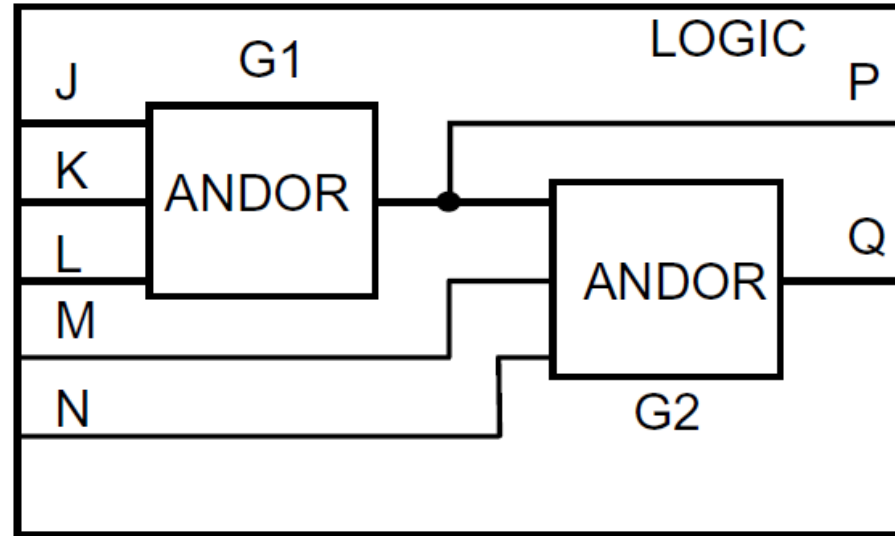
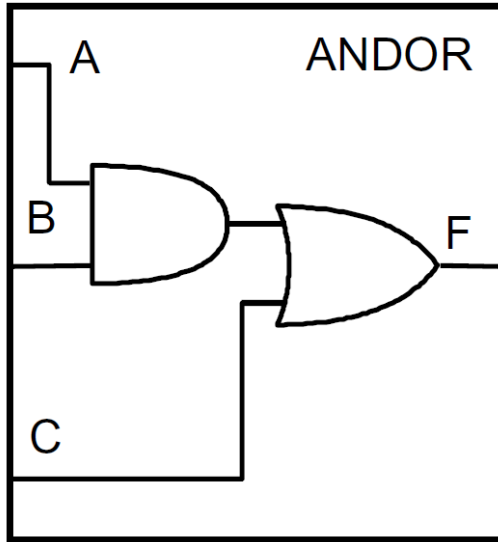
```
module LOGIC (J, K, L, M, N, P, Q);  
  input J, K, L, M, N;  
  output P, Q;
```

```
  ANDOR G1 (J, K, L, P);  
  ANDOR G2 (P, M, N, Q);  
endmodule
```

Instances of modules

Ordered mapping

Named Mapping



```
module LOGIC (J, K, L, M, N, P, Q);  
  input J, K, L, M, N;  
  output P, Q;
```

Named mapping

```
  ANDOR G1 (.A(J), .B(K), .C(L), .F(P));  
  ANDOR G2 (.B(M), .C(N), .A(P), .F(Q));  
endmodule
```

Order doesn't matter

Unconnected Ports

```
module AOI (A, B, C, D, F);  
  input A, B, C, D;  
  output F;  
  ...  
endmodule
```

```
module ...
```

```
  AOI AOI1 (W1, W2, , W3, W4);
```

```
  /* Equivalent...
```

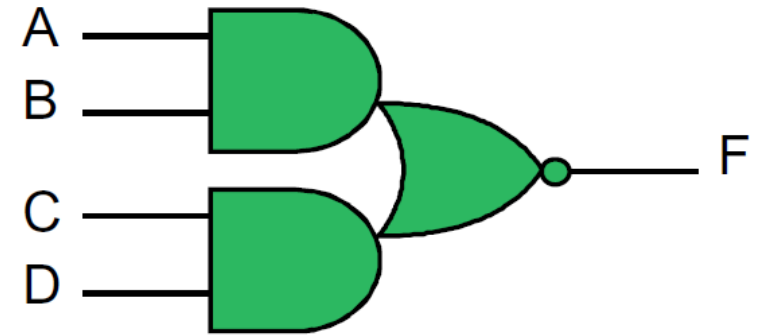
```
  AOI AOI1 (.F(W4), .D(W3), .B(W2), .A(W1));
```

```
  AOI AOI1 (.F(W4), .D(W3), .B(W2), .A(W1), .C());
```

```
*/
```

```
endmodule
```

Unconnected port C



Testbenches - Device Under Test (DUT)

```
module AND2 (A, B, Y);  
    input A, B;  
    output Y;  
    assign Y = A & B;  
endmodule
```

```
module AND2_DUT();  
    reg A, B;                // Registers  
    wire Y;  
  
    initial begin  
        A=1; B=1; #100;      // Delay 100ps  
        A=0; B=1; #100;      // Delay 100ps  
        A=1; B=0; #100;      // Delay 100ps  
        A=0; B=0; #100;      // Delay 100ps  
    end  
  
    AND2 G (A, B, Y);  
endmodule
```

vsim AND2_DUT

Truth Tables

```
module TestAND2();  
    reg A, B;                // Registers  
    wire Y;  
  
    initial begin  
        A=0; B=0; #100;      // Delay 100ps  
        A=0; B=1; #100;      // Delay 100ps  
        A=1; B=0; #100;      // Delay 100ps  
        A=1; B=1; #100;      // Delay 100ps  
    end  
  
    AND2 G (A, B, Y);  
  
    // Display Truth Table  
    initial begin  
        $display("\t\t Time A B Y");  
        $monitor("%d %b %b %b", $time, A, B, Y);  
    end  
endmodule
```

#	Time	A	B	Y
#	0	0	0	0
#	100	0	1	0
#	200	1	0	0
#	300	1	1	1

Primitives

Verilog includes a small number of **built-in primitives** or gates. These include models of simple logic gates, tristate buffers, pullup and pulldown resistors, and a number of unidirectional and bidirectional switches.

Built in

- **Gates**

and, nand, or, nor, xor, xnor, buf, not

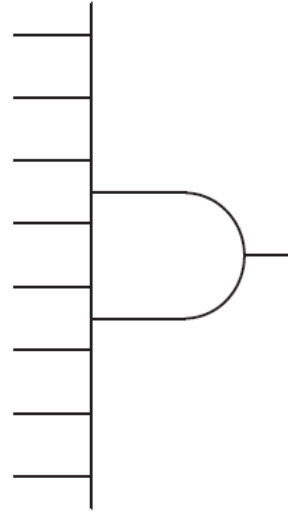
- **Pulls, tristate buffers**

pullup, pulldown, bufif0, bufif1, notif0, notif1

- **Switches**

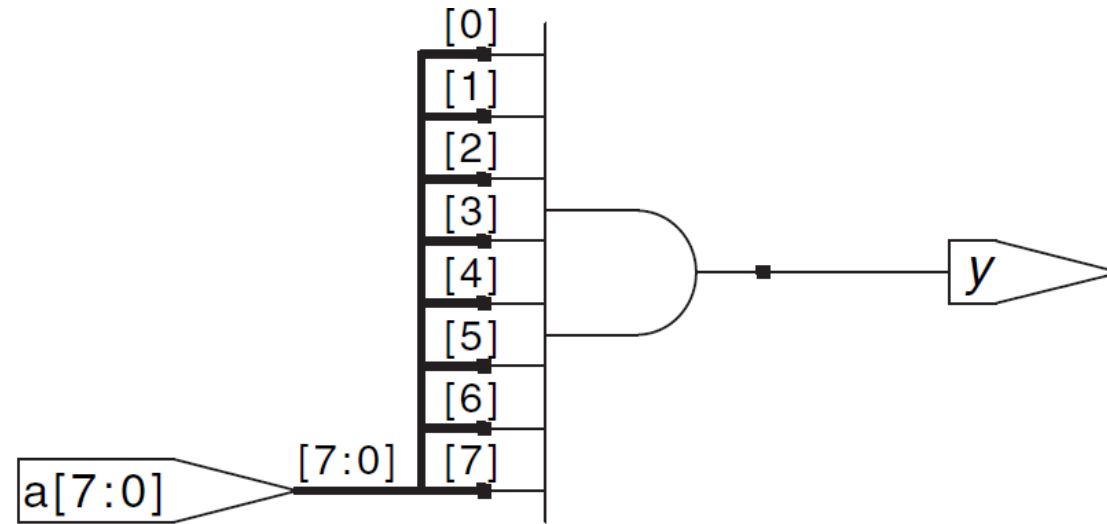
cmos, nmos, ... tran, ...

AND8



```
module AND8 (A, B, C, D, E, F, G, H, Y);  
  input A, B, C, D, E, F, G, H;  
  output Y;  
  
  assign Y = A & B & C & D & E & F & G & H;  
endmodule
```

Vector Ports



```
module AND8(A, Y);  
    input [7:0] A;  
    output Y;  
  
    assign Y = A[0] & A[1] & A[2] & A[3]  
               & A[4] & A[5] & A[6] & A[7];  
endmodule
```

2-input Logic Gates

```
module GATES2(A, B, Y0, Y1, Y2, Y3, Y4, Y5);  
    input A, B;  
    output Y0, Y1, Y2, Y3, Y4, Y5;  
  
    assign Y0 = A & B;           // AND  
    assign Y1 = ~(A & B);       // NAND  
    assign Y2 = A | B;          // OR  
    assign Y3 = ~(A | B);       // NOR  
    assign Y4 = A ^ B;          // XOR  
    assign Y5 = ~(A ^ B);       // XNOR  
endmodule
```

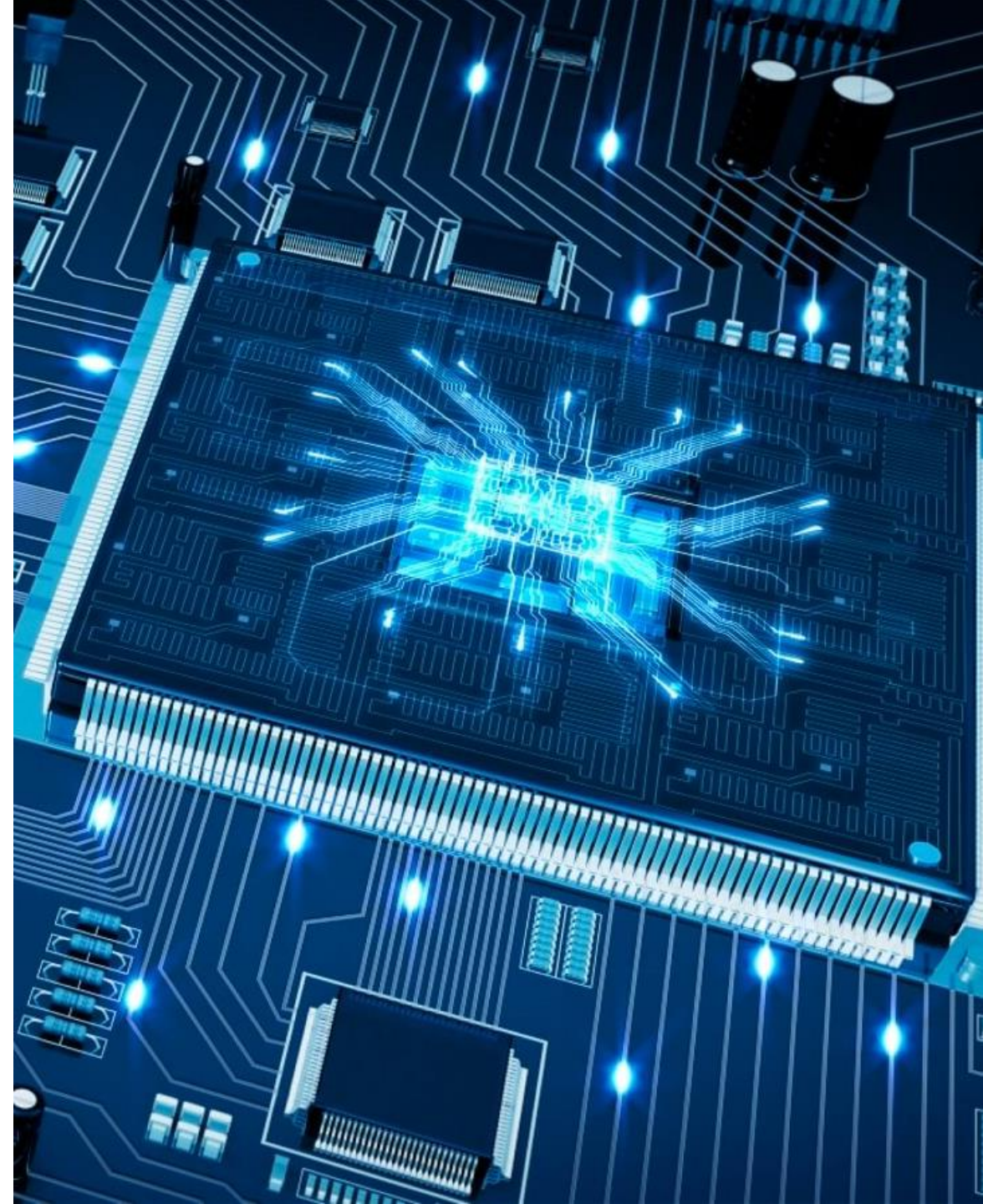
2-input Logic Gates Using Vectors

```
module VEC2(A, B, Y);  
  input A, B;  
  output [5:0] Y;  
  
  assign Y[0] = A & B;           // AND  
  assign Y[1] = ~(A & B);       // NAND  
  assign Y[2] = A | B;          // OR  
  assign Y[3] = ~(A | B);       // NOR  
  assign Y[4] = A ^ B;          // XOR  
  assign Y[5] = ~(A ^ B);       // XNOR  
endmodule
```

2-input Logic Gates Testbench

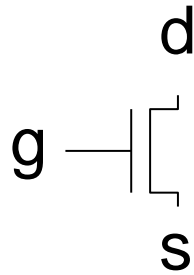
```
module TestVEC2();  
    reg A, B;                // Registers  
    wire [5:0] Y;  
  
    initial begin  
        A=0; B=0; #200;      // Delay 200ps  
        A=0; B=1; #200;      // Delay 200ps  
        A=1; B=0; #200;      // Delay 200ps  
        A=1; B=1; #200;      // Delay 200ps  
        A=0; B=0; #200;      // Delay 200ps  
    end  
  
    VEC2 G (A, B, Y);  
endmodule
```

*Mini Project With CMOS Transistors

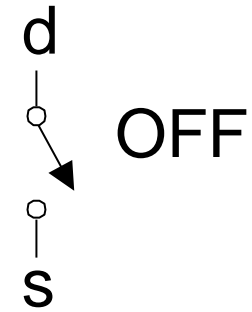


*CMOS Transistors

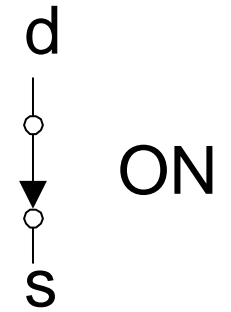
nMOS



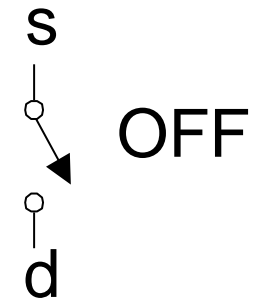
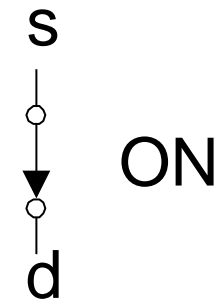
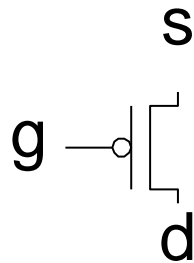
$g = 0$



$g = 1$

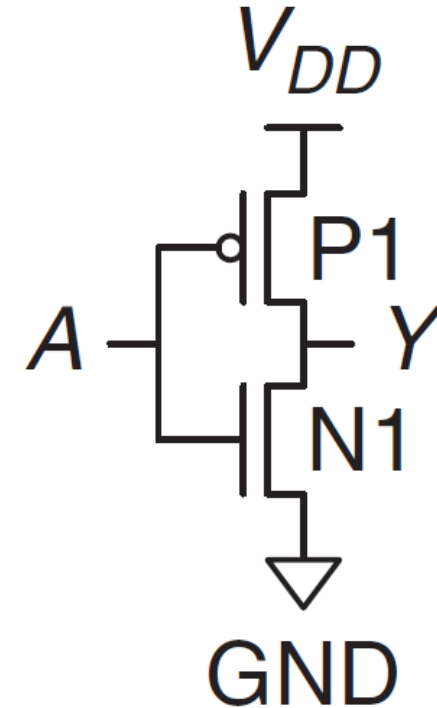


pMOS



*NOT Gate Built With CMOS Transistors

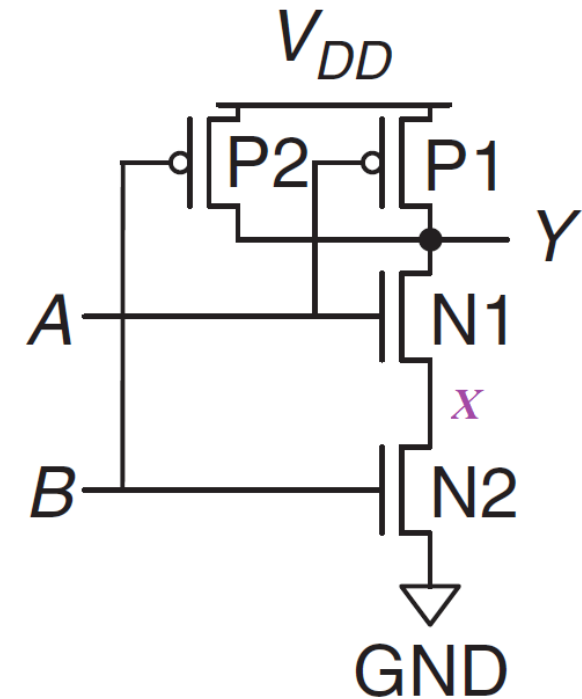
```
module TINV (A, Y);  
  input A;  
  output Y;  
  
  supply1 VDD;  
  supply0 GND;  
  
  pmos P1 (Y, VDD, A);  
  nmos N1 (Y, GND, A);  
endmodule
```



A	P1	N1	Y
0	ON	OFF	1
1	OFF	ON	0

*NAND2 Gate Built With CMOS Transistors

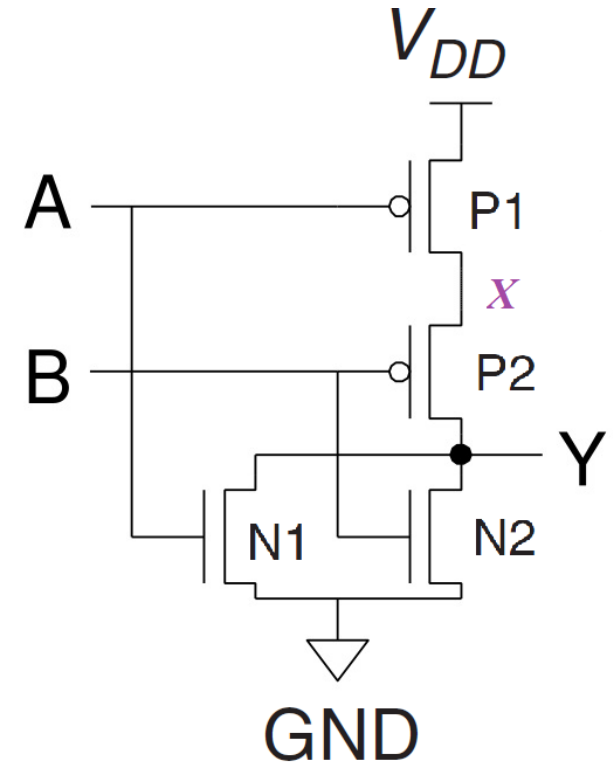
```
module TNAND2 (A, B, Y);  
  input A, B;  
  output Y;  
  
  supply1 VDD;  
  supply0 GND;  
  wire X;  
  
  pmos P1 (Y, VDD, A);  
  pmos P2 (Y, VDD, B);  
  nmos N1 (Y, X, A);  
  nmos N2 (X, GND, B);  
endmodule
```



A	B	P1	P2	N1	N2	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

*NOR2 Gate Built With CMOS Transistors

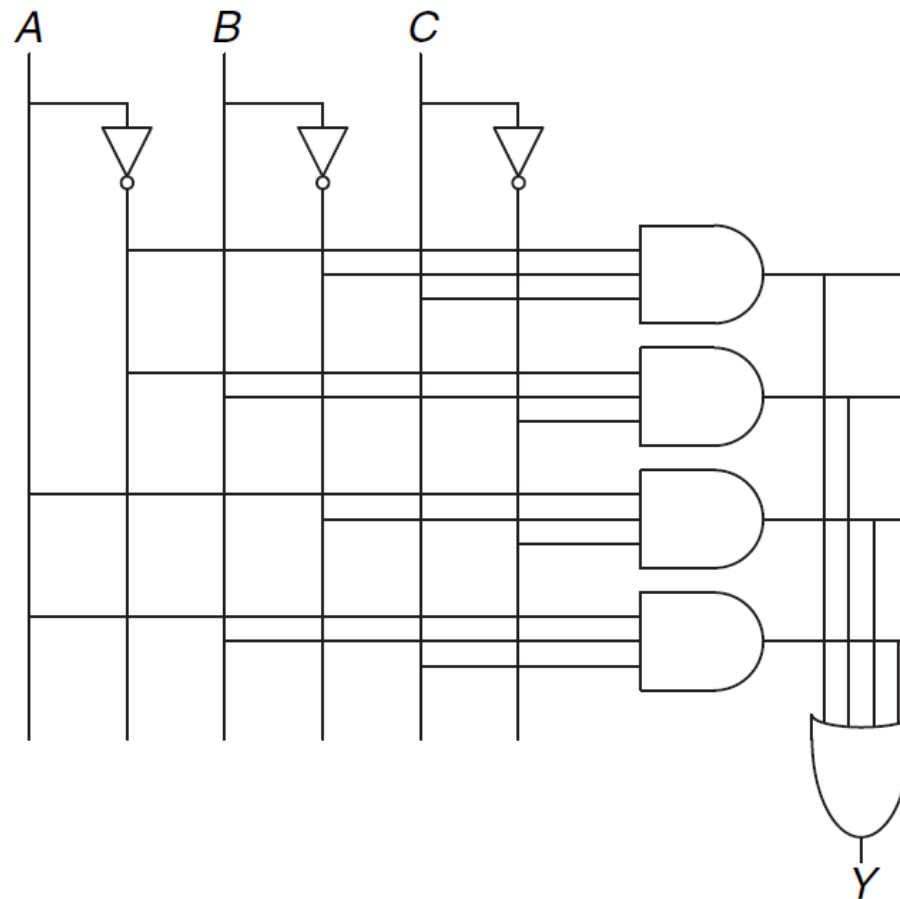
```
module TNOR2 (A, B, Y);  
  input A, B;  
  output Y;  
  
  supply1 VDD;  
  supply0 GND;  
  wire X;  
  
  pmos P1 (X, VDD, A);  
  pmos P2 (Y, X, B);  
  nmos N1 (Y, GND, A);  
  nmos N2 (Y, GND, B);  
endmodule
```



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

*Mini Project With CMOS Transistors

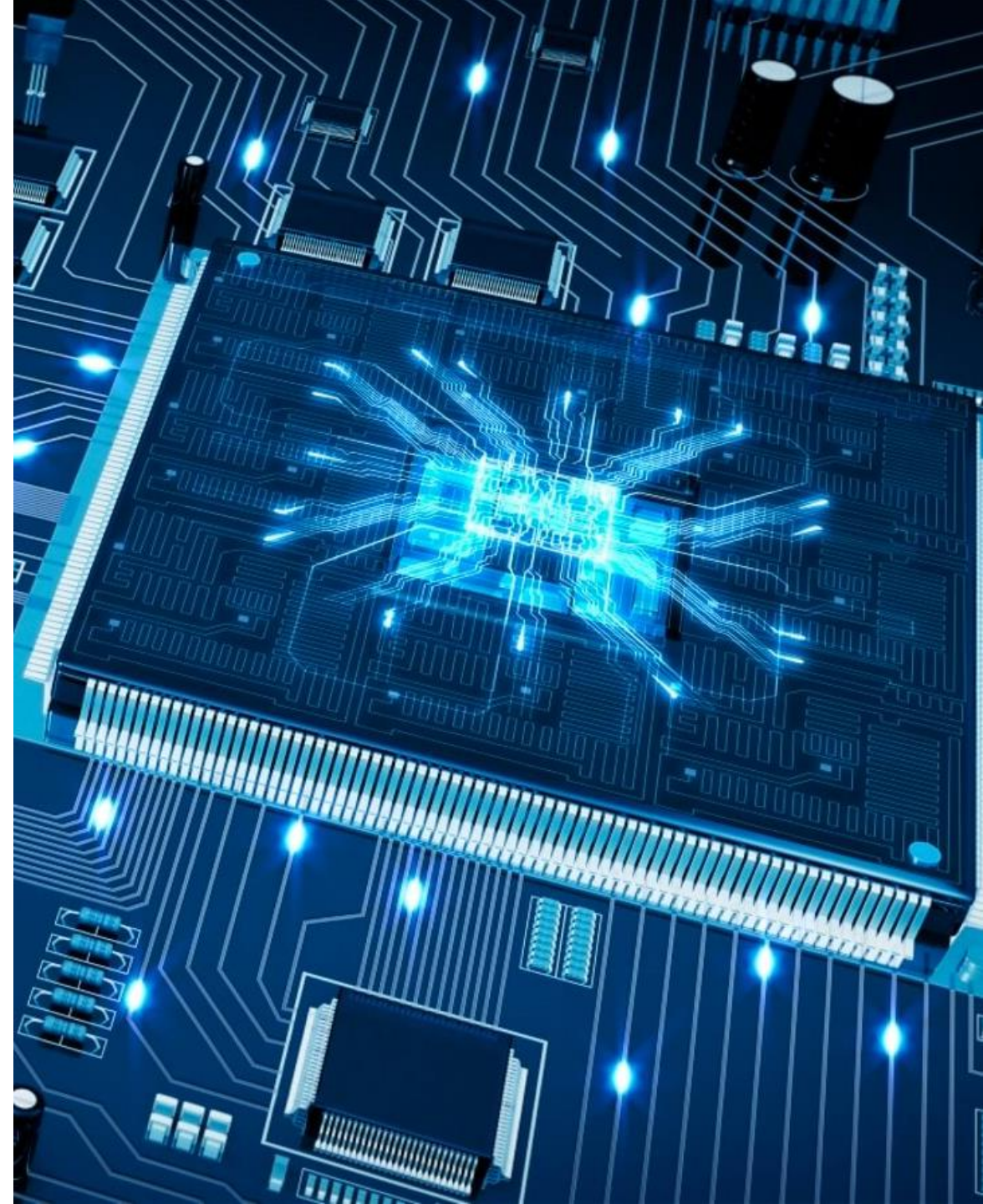
Write SystemVerilog code to implement the following combinational circuit **using only CMOS transistors**. Build the required logic gates from CMOS transistors and use module instances to implement the circuit. Don't use any primitives or operators.



*Mini Project With CMOS Transistors

- Answer the following question about your project.
 - 1) Show the truth table for the circuit.
 - 2) Write testbench and compare the results with the truth table.
 - 3) What is the function performed by the circuit?
 - 4) What is the number of transistors you used to build the circuit?
 - 5) How to implement the circuit using **only two logic gates**?

Introduction to Programmable Logic



Fixed-Function Logic

- A fixed-function IC comes with **logic functions that cannot be programmed** in and cannot be altered.

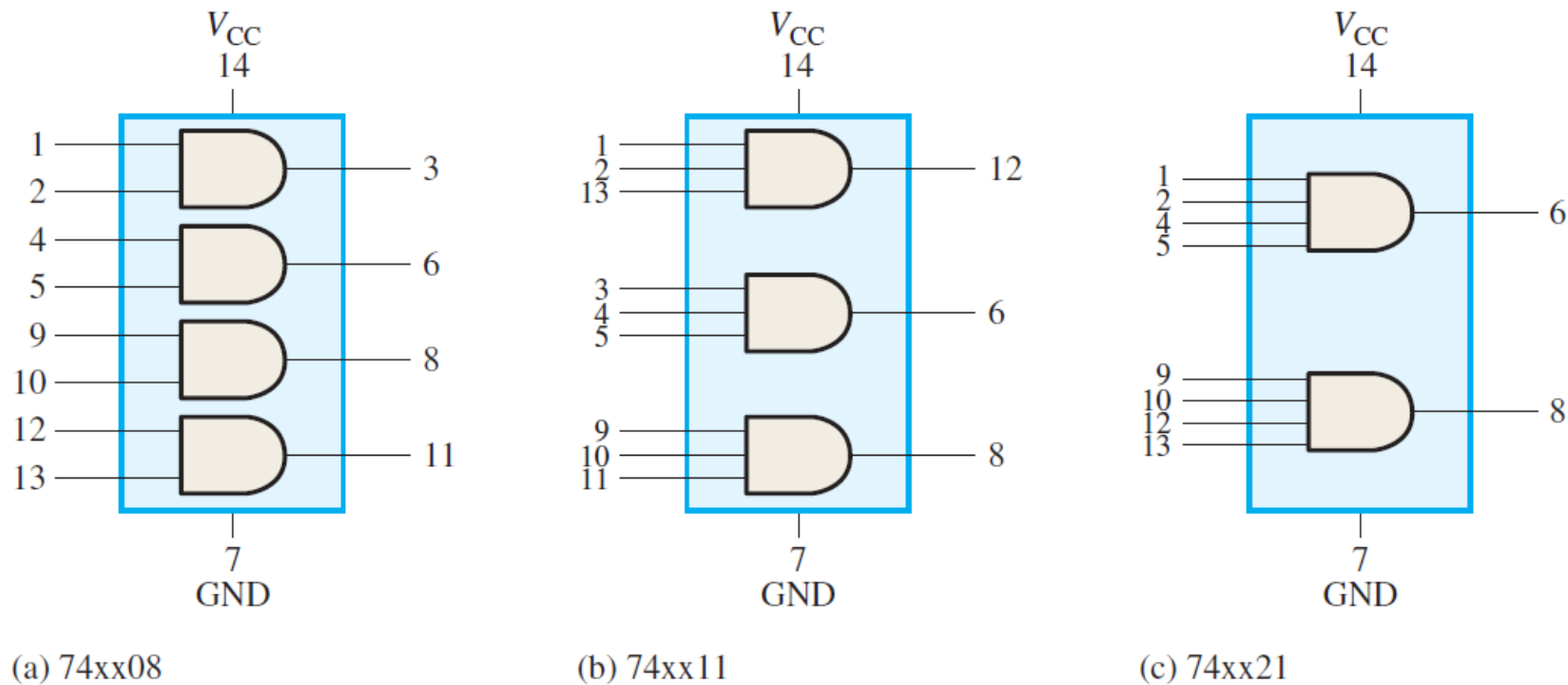


FIGURE 3-59 74 series AND gate devices with pin numbers.

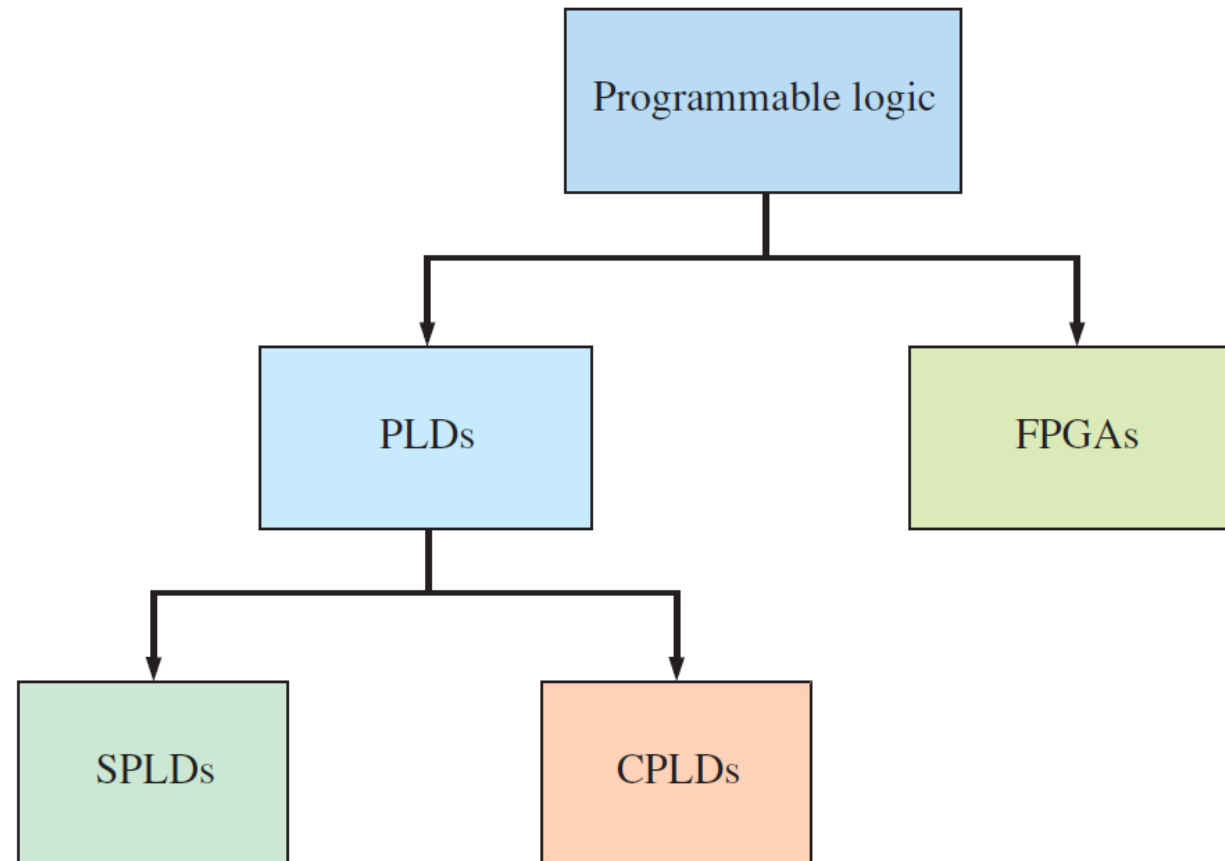
Programmable Logic

- Programmable logic requires both hardware and software.
- Programmable logic devices can be programmed to perform specified logic functions.
- With programmable logic, designs can be readily changed without rewiring or replacing components.
- A logic design can generally be implemented faster and with less cost with programmable logic than with fixed-function logic.
- To implement small segments of logic, it may be more efficient to use fixed-function logic.

Programmable Logic Hierarchy

Two major categories of user-programmable logic are

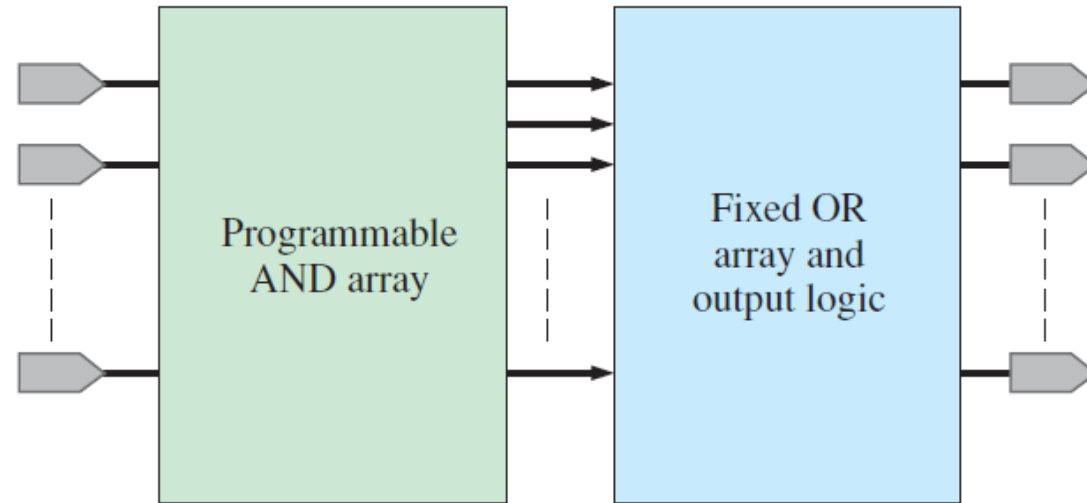
- PLD (programmable logic device) and
- FPGA (field-programmable gate array)



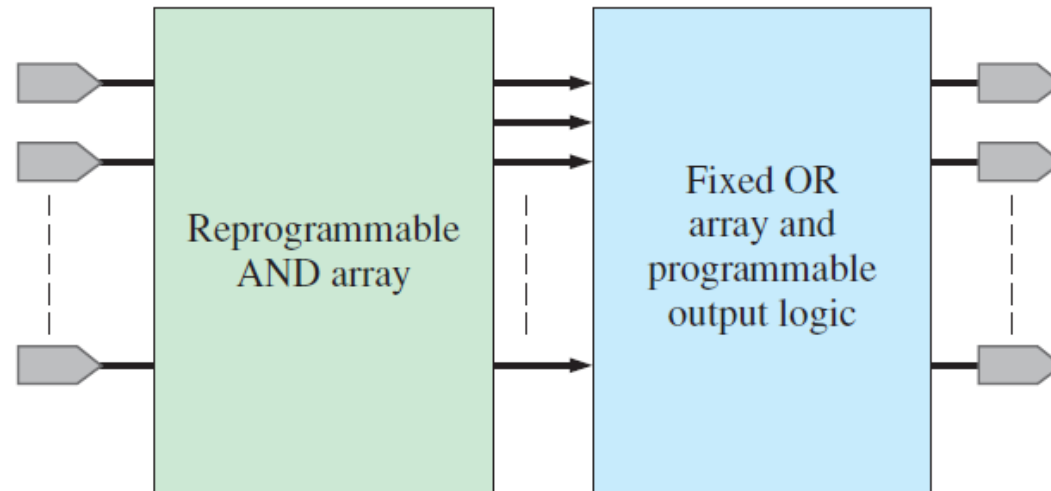
Simple Programmable Logic Device (SPLD)

- The SPLD is still available for **small-scale applications**.
- Most SPLDs are in one of two categories: **PAL** and **GAL**.
- A **PAL** (programmable array logic) is a device that can be **programmed one time**.
- It consists of a **programmable array of AND** gates and a fixed array of OR gates
- A **GAL** (generic array logic) is a device that is basically a PAL that can be **reprogrammed many times**.
- It consists of a **reprogrammable array of AND** gates and a fixed array of OR gates with **programmable outputs**.

Simple Programmable Logic Device (SPLD)

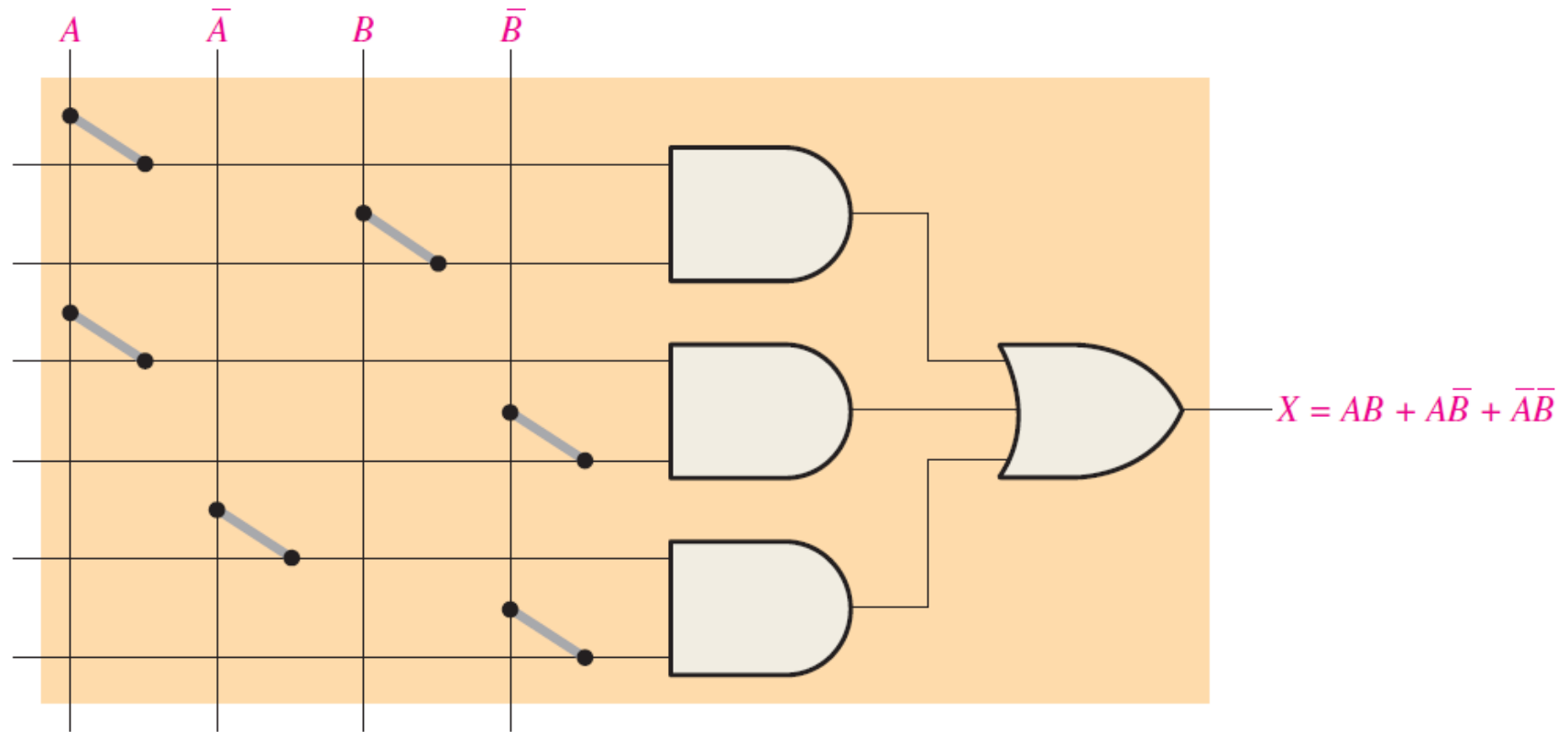


(a) PAL



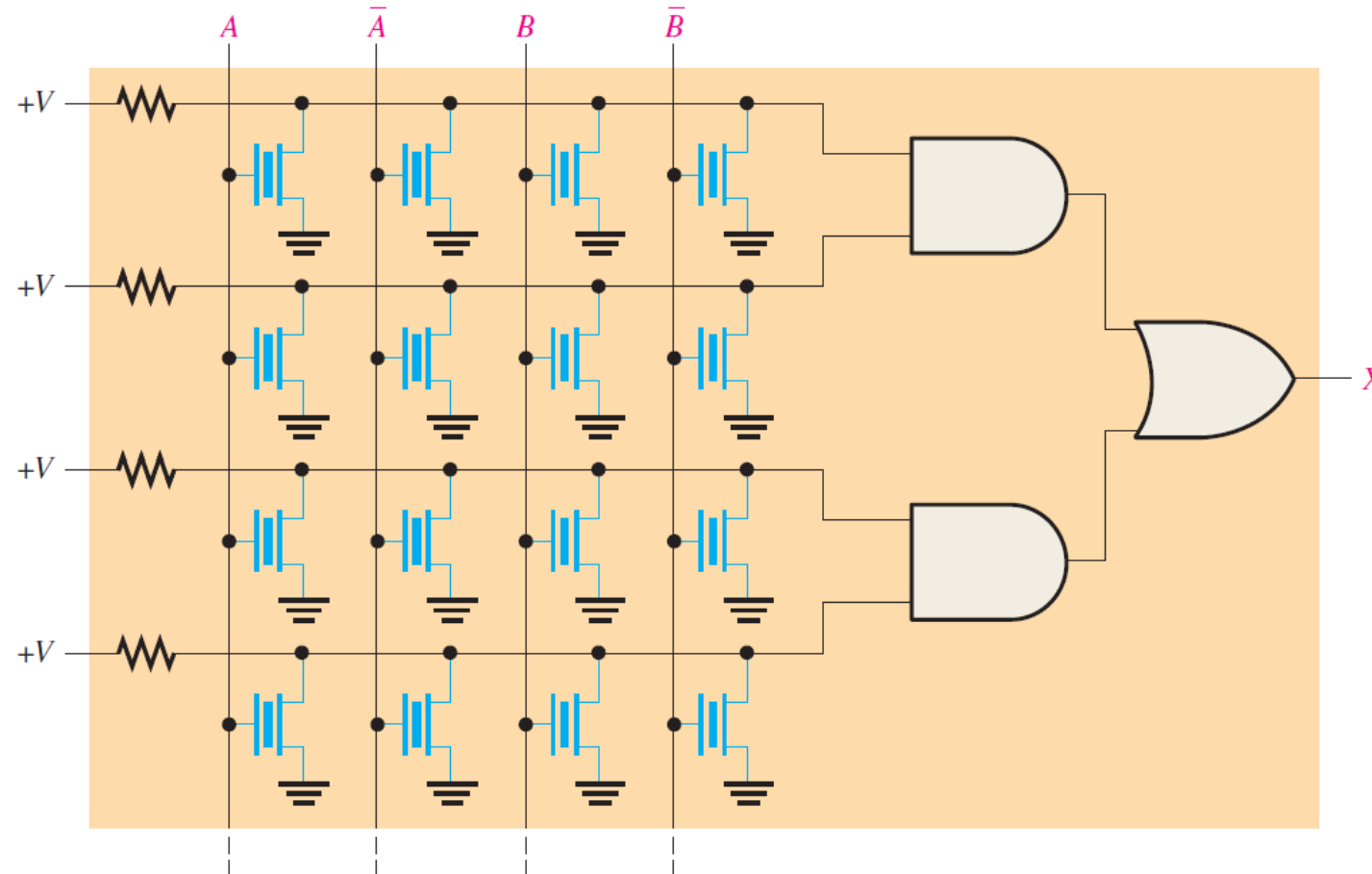
(b) GAL

SPLD: The PAL



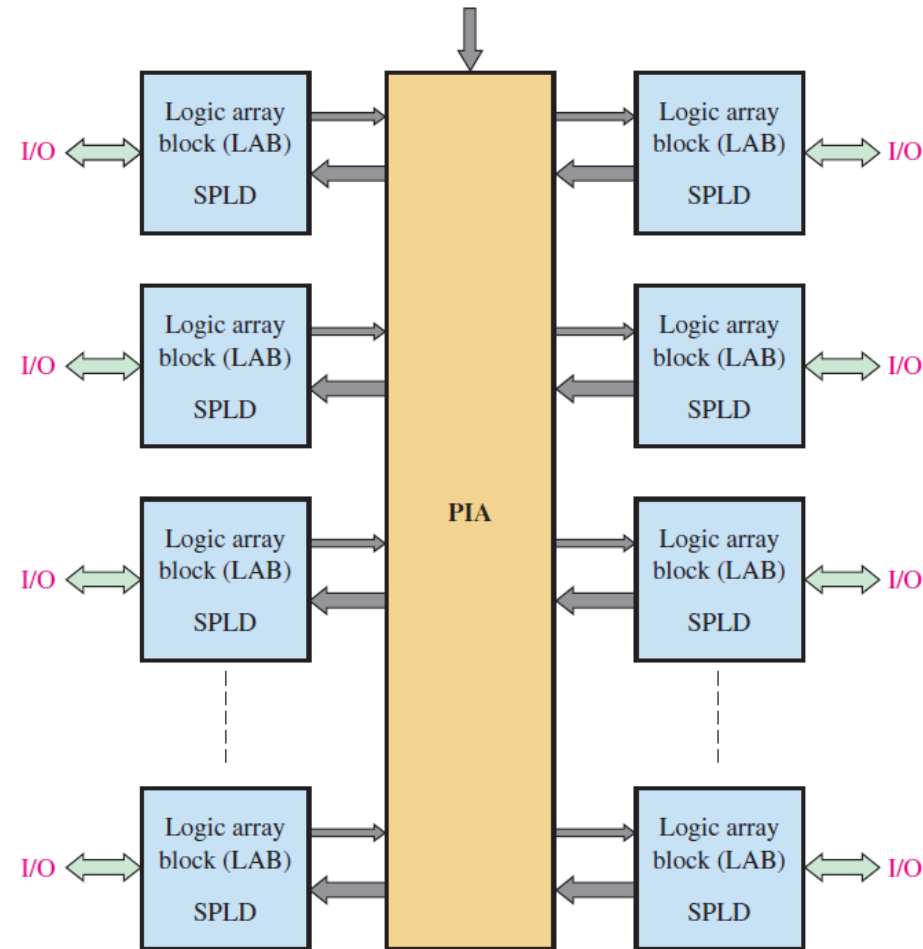
SPLD: The GAL

- The basic difference is that a GAL uses a reprogrammable process technology, such as **EEPROM** instead of fuses.



Complex Programmable Logic Device (CPLD)

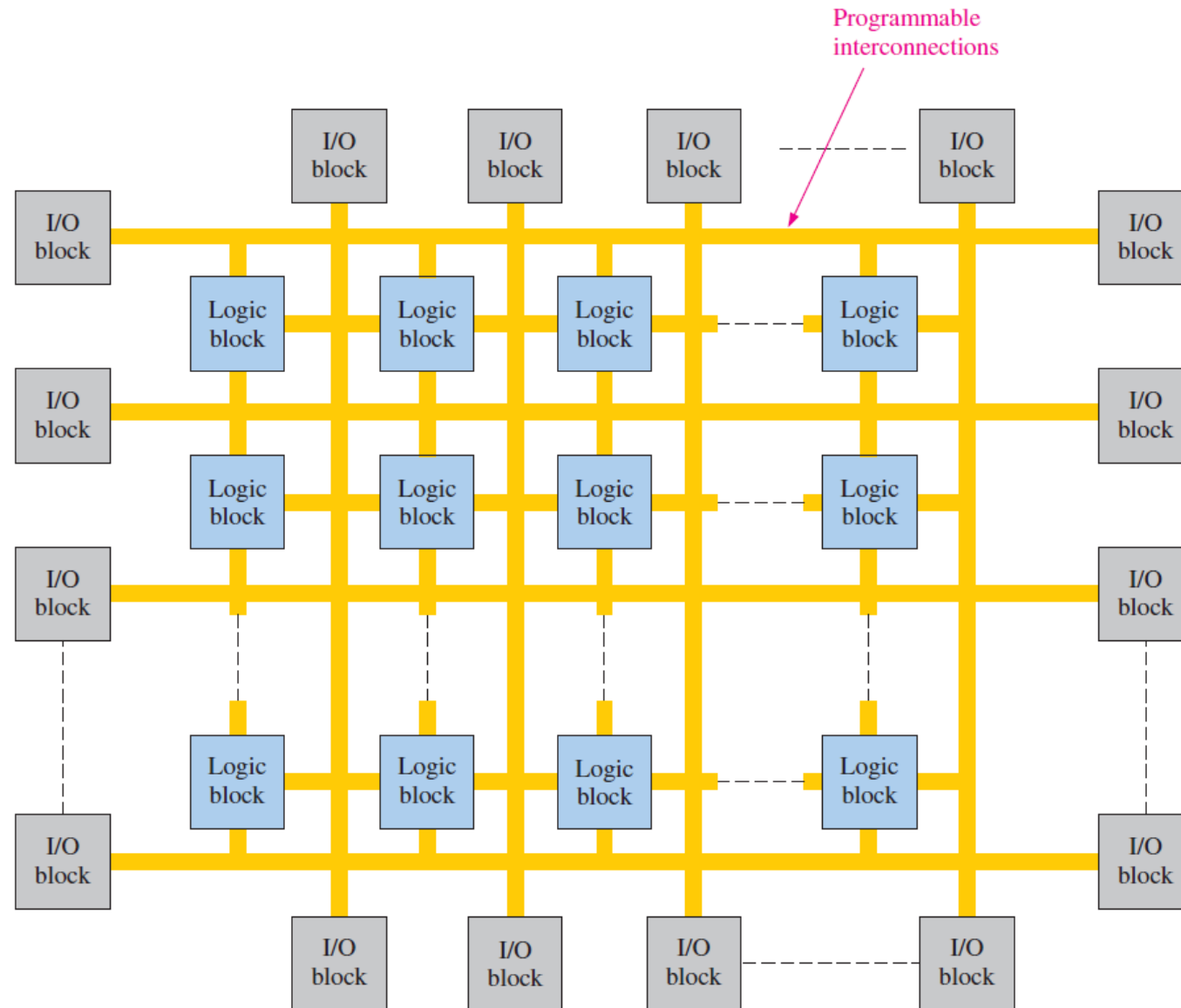
- The **CPLD** is a device **containing multiple SPLDs**.
- Each logic array block (**LAB**) is roughly equivalent to one SPLD.



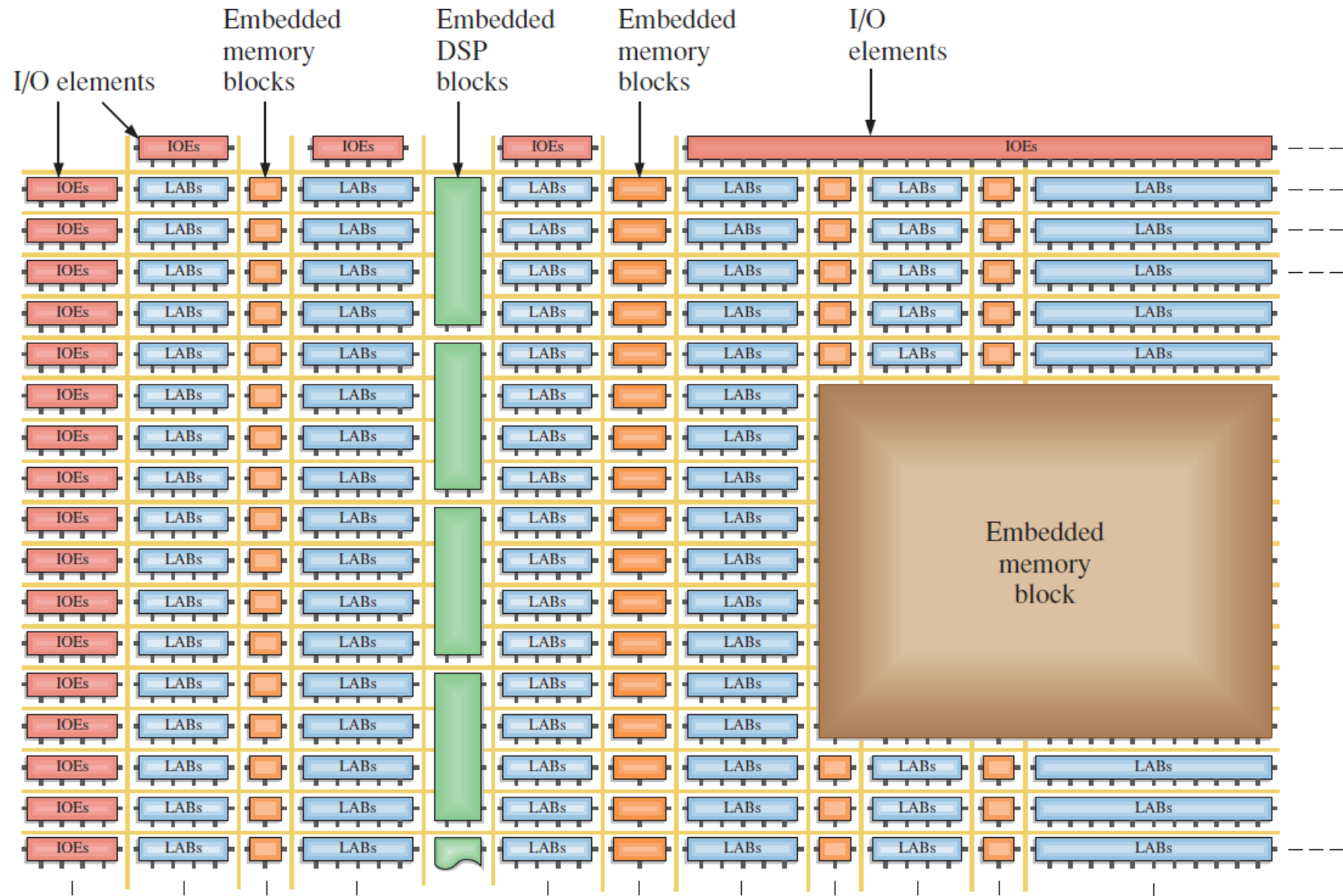
Field-Programmable Gate Array (FPGA)

- An **FPGA** is generally **more complex** and has a much higher density than a CPLD.
- As mentioned, the **SPLD and the CPLD are closely related** because the CPLD basically contains a number of SPLDs.
- The **three basic elements** in an FPGA are the logic block, the programmable interconnections, and the input/output (I/O) blocks.
- The **logic blocks** in an FPGA are not as complex as the logic array blocks (**LABs**) in a CPLD.

Field-Programmable Gate Array (FPGA)

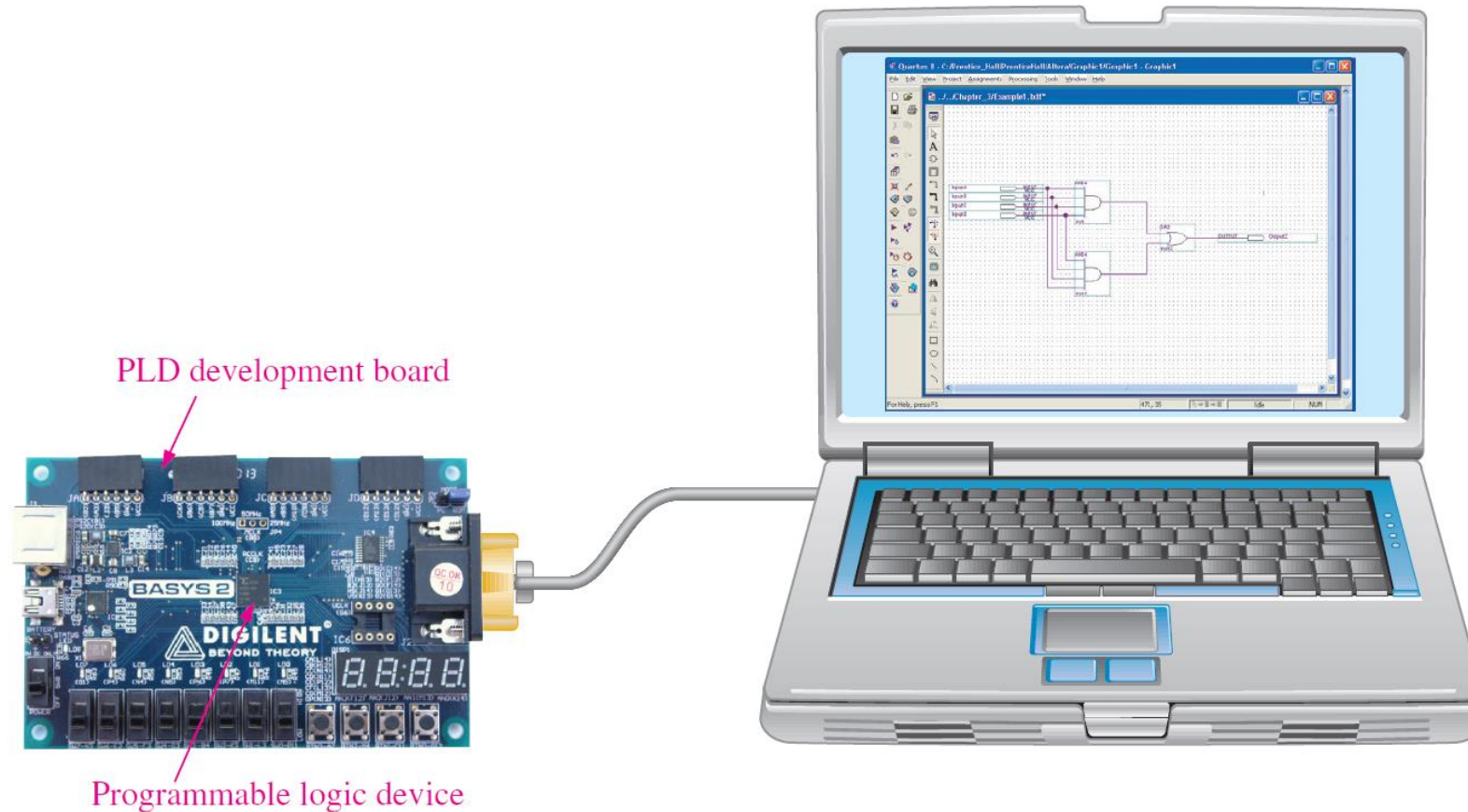


Field-Programmable Gate Array (FPGA)



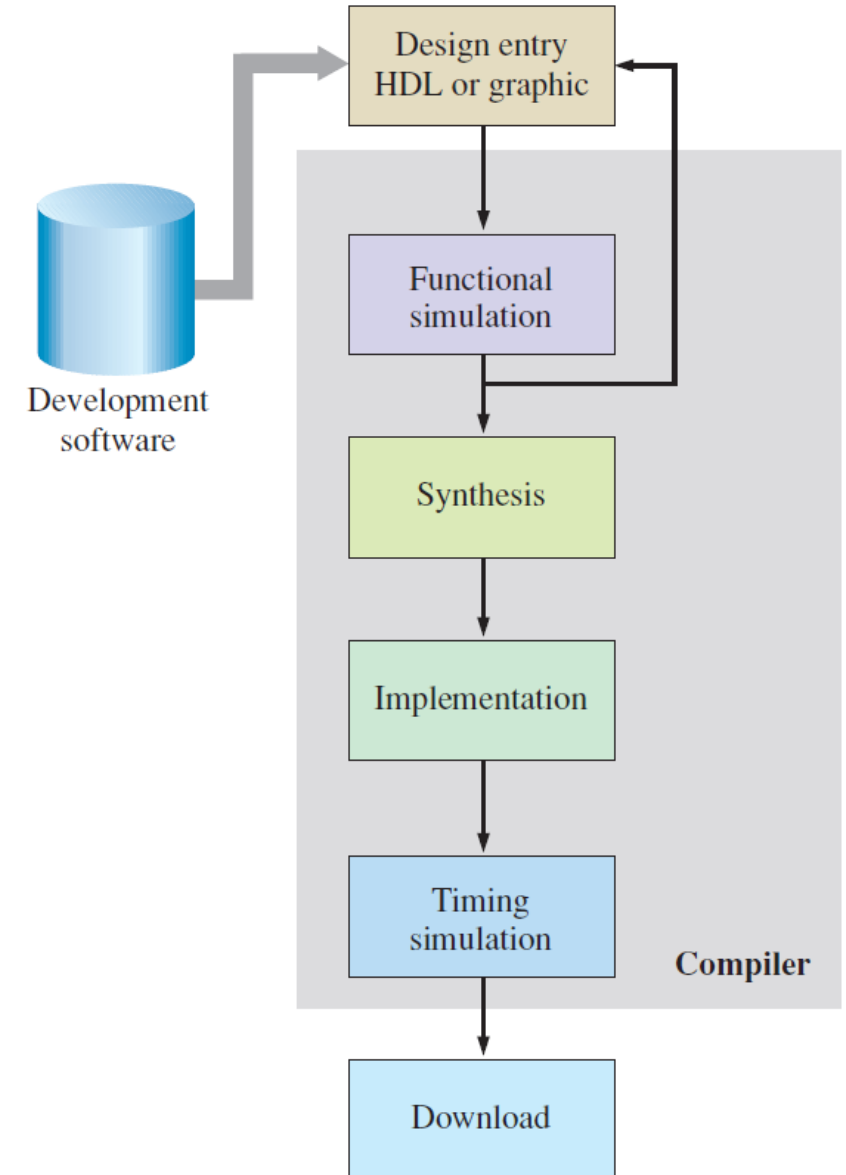
The Programming Process

- This process requires a software development package installed on a computer to implement a circuit design in the programmable chip.

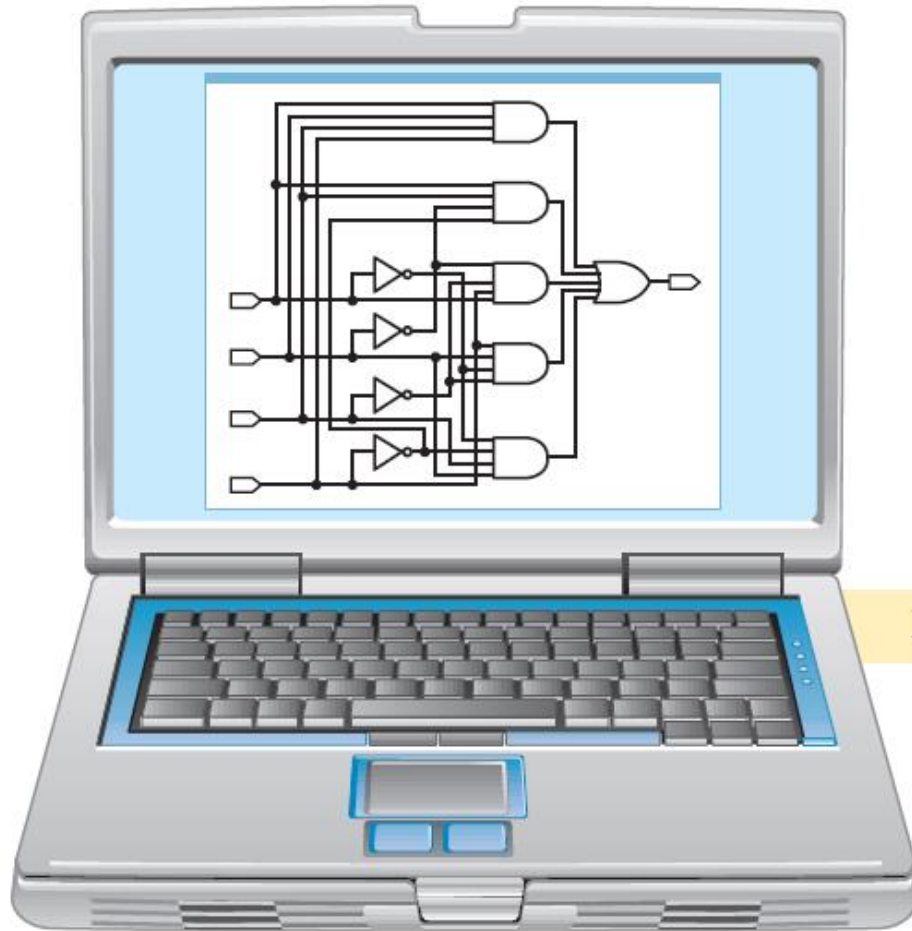


Programmable Logic Design Flow Block Diagram

- **Synthesis** is where the **design** is translated into a netlist.
- The **implementation** process is called **place and route** and results in an output called a **bitstream**, which is device dependent.
- Once a bitstream has been generated for a specific programmable device, **it has to be downloaded to the device**.

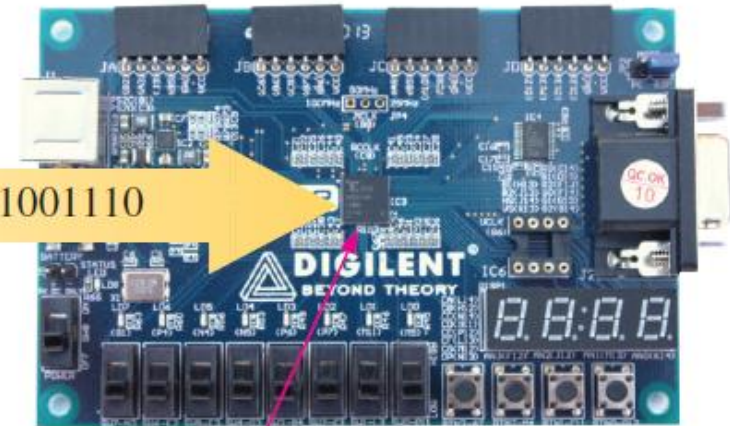


Downloading



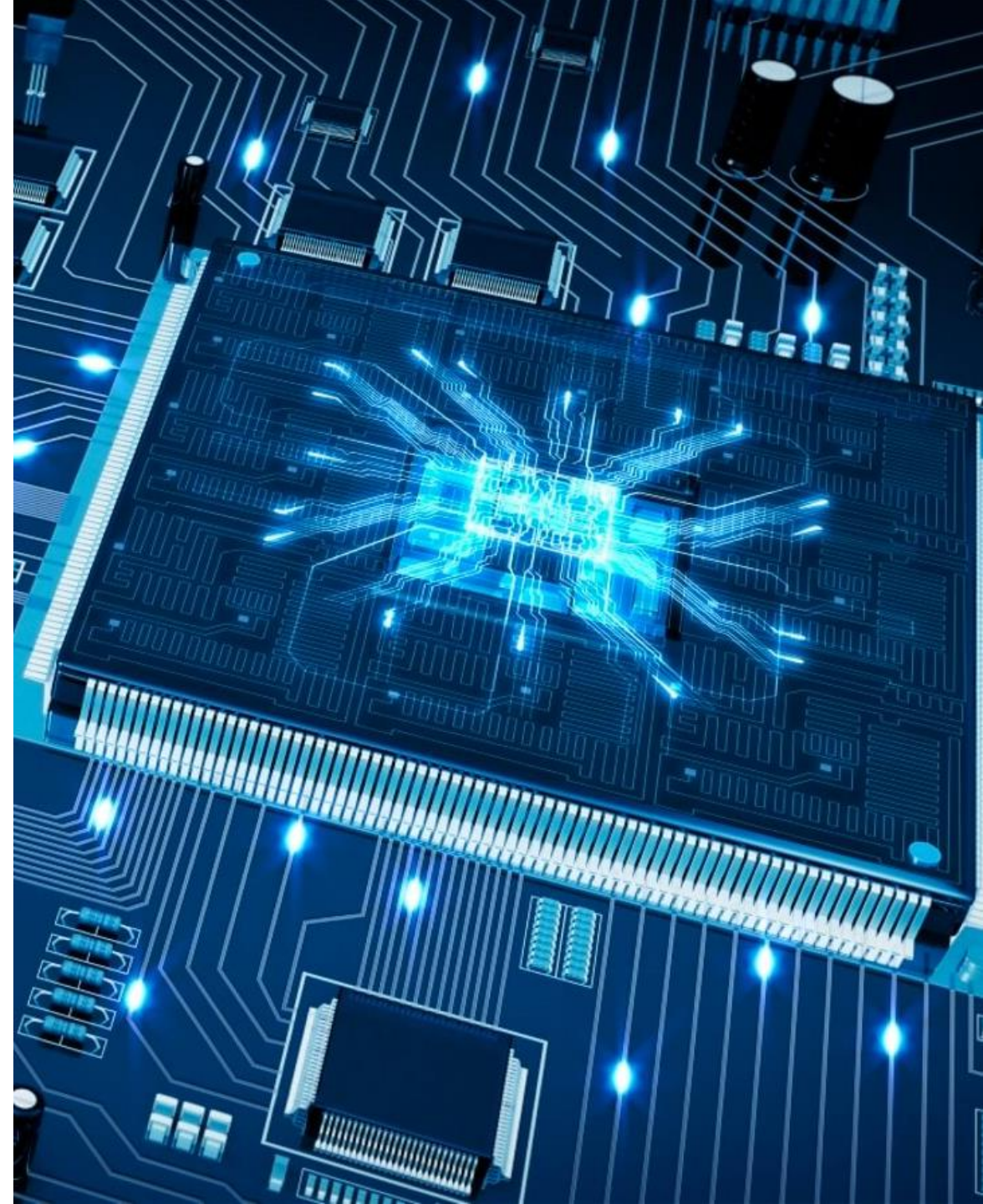
Bitstream

11010001101111101001110

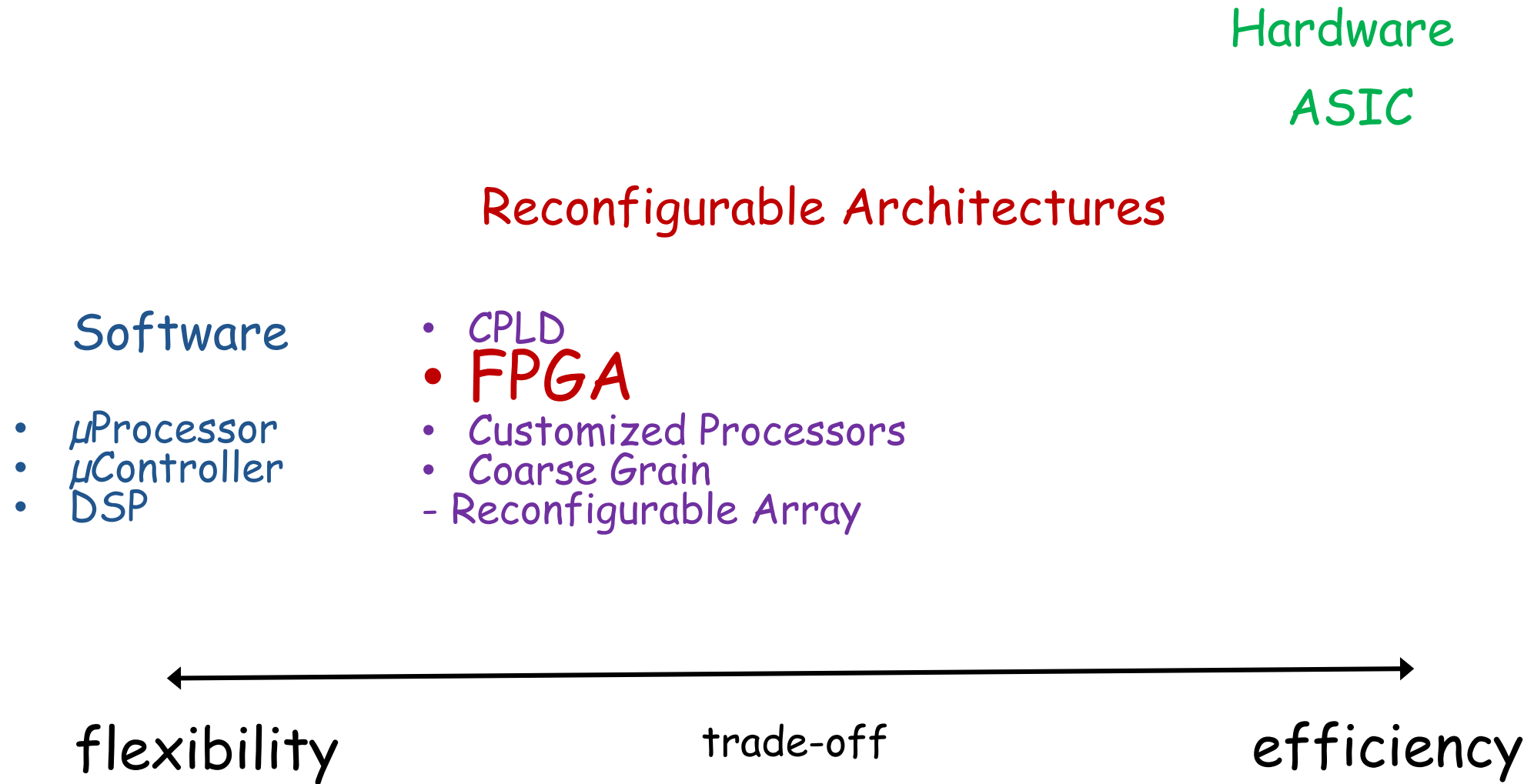


Target device

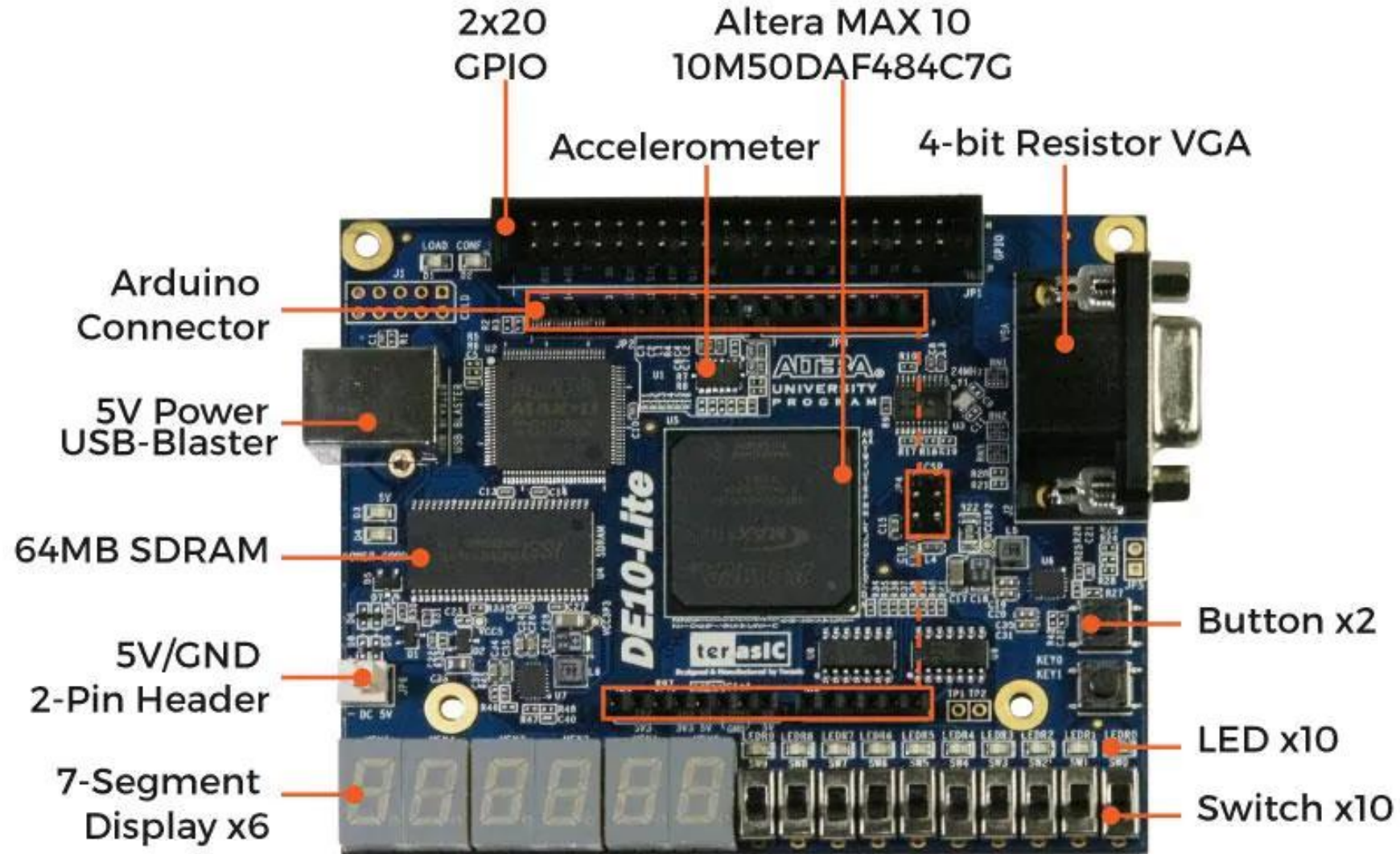
DE10-Lite FPGA



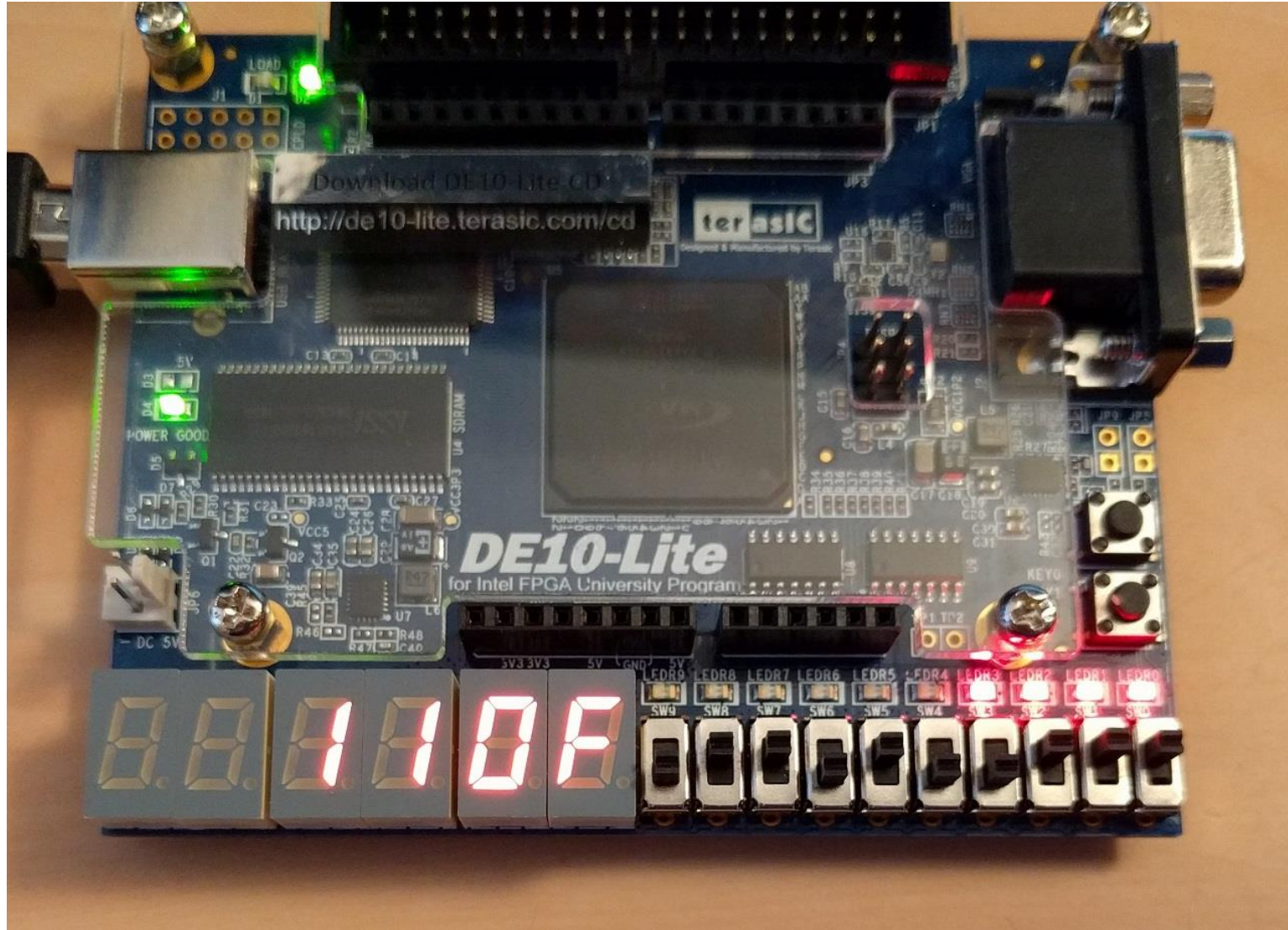
Digital System Implementation Spectrum



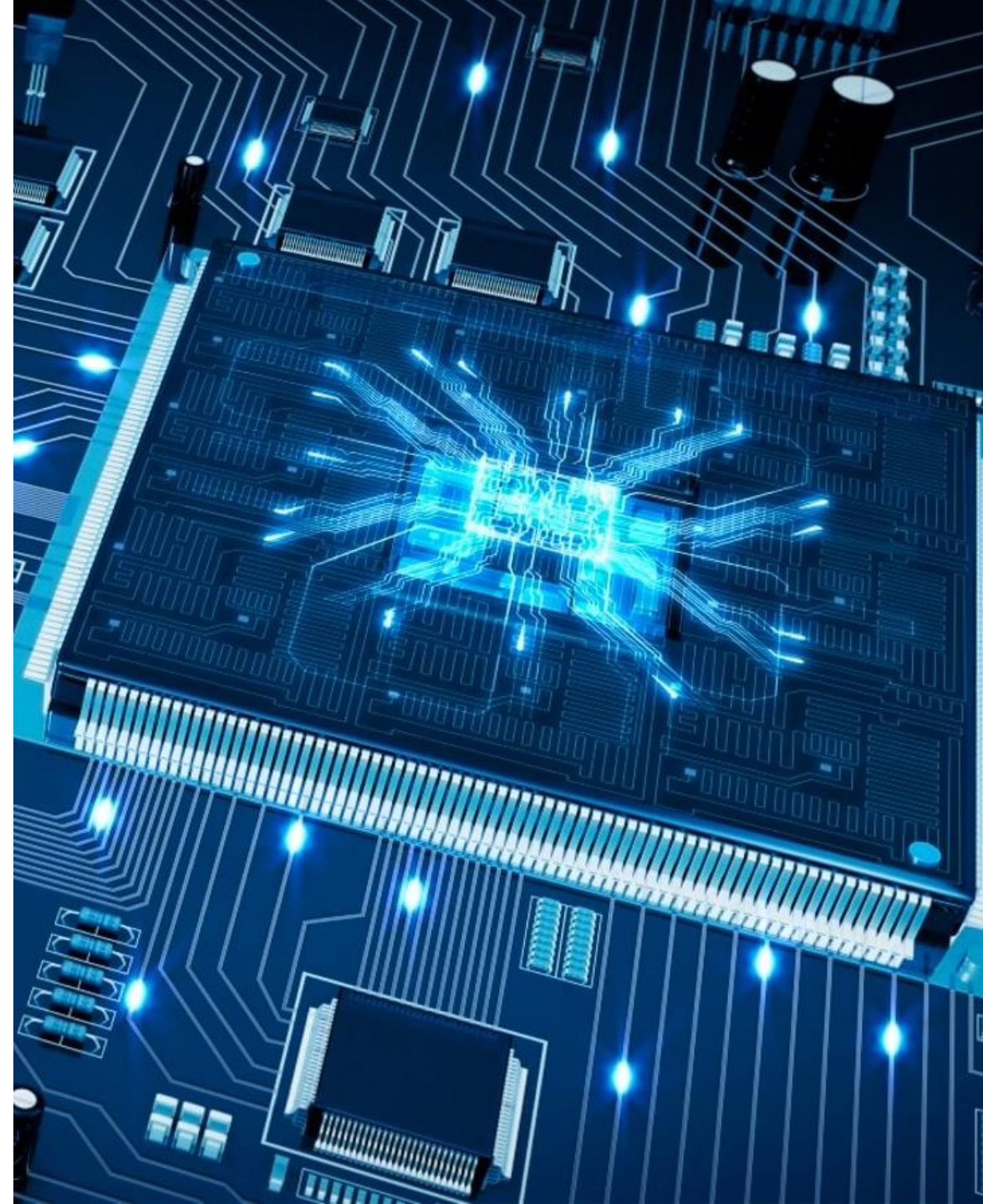
DE10-Lite



DE10-Lite



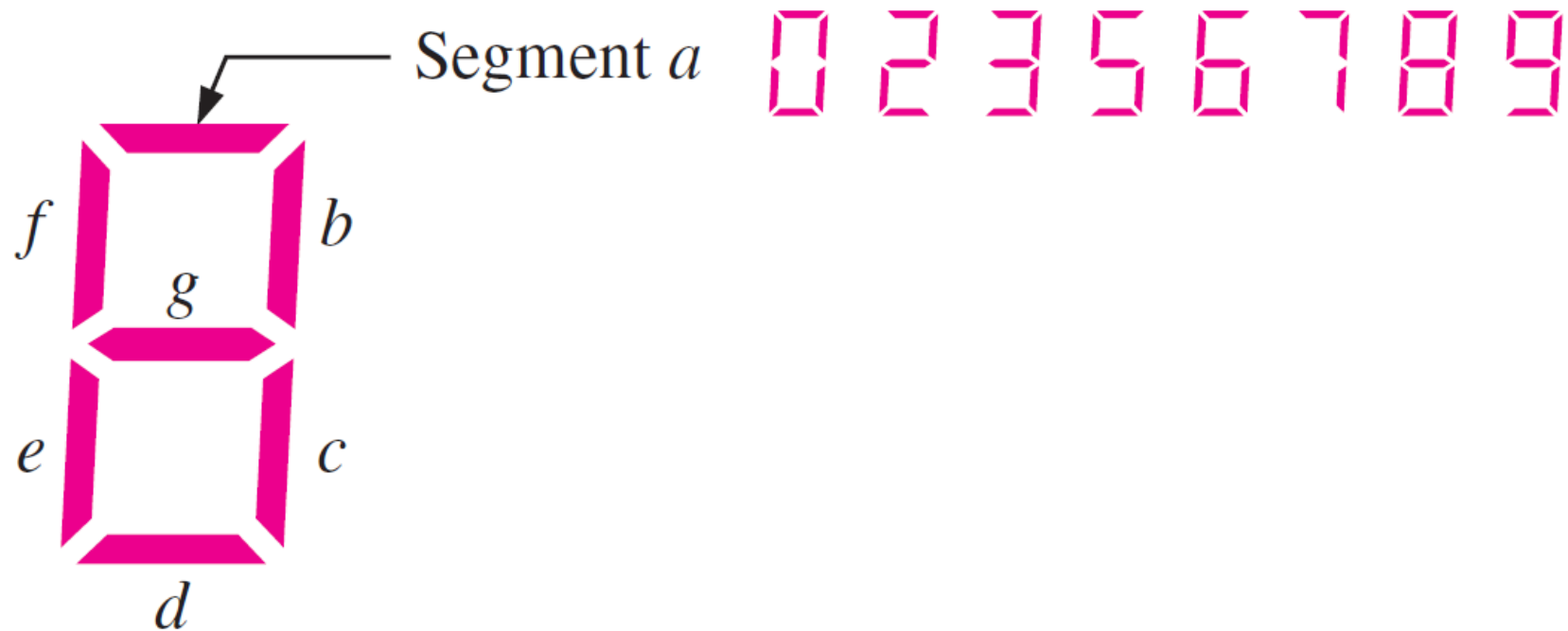
The BCD to 7-Segment Decoder



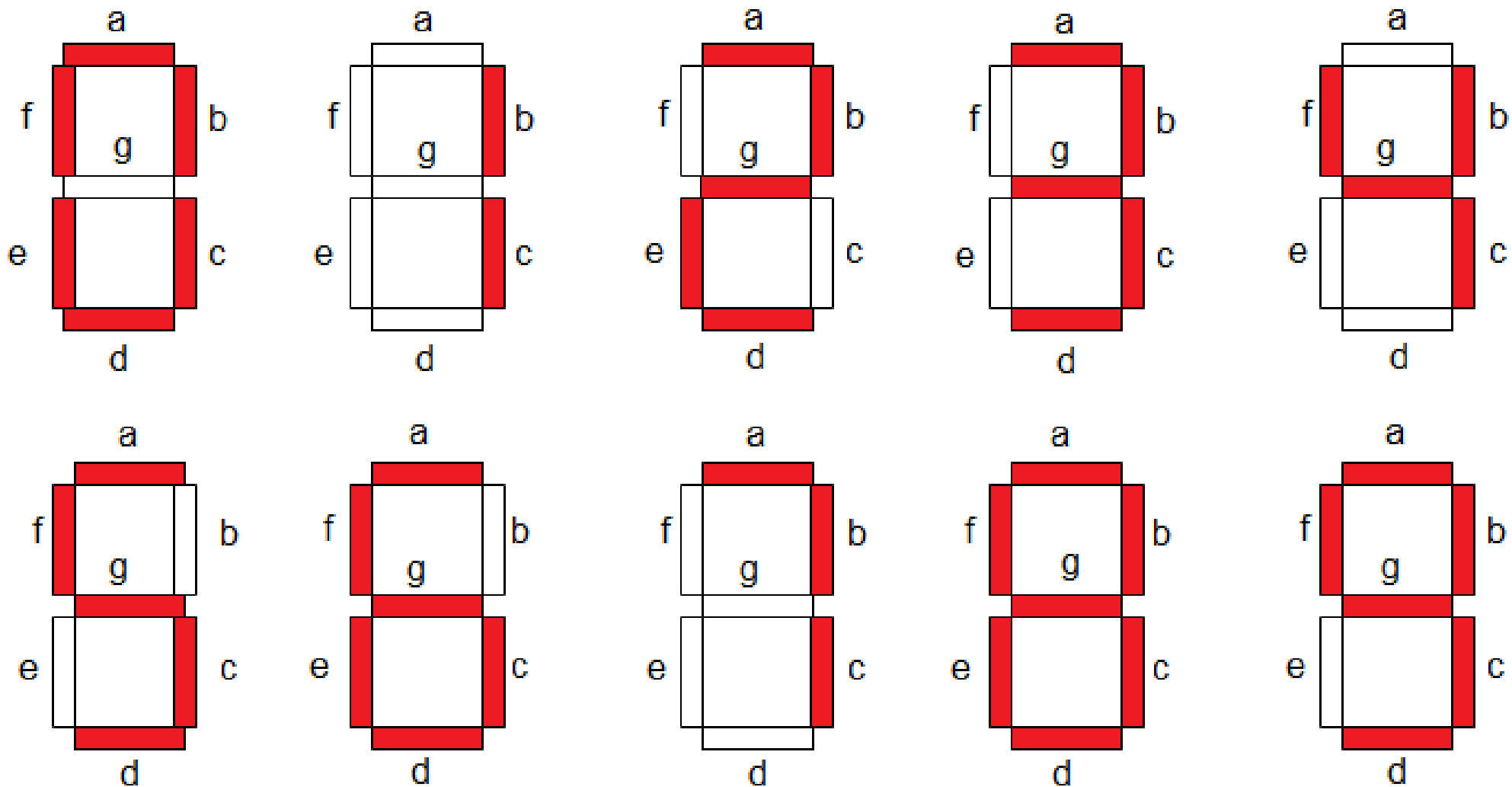
7-segment Display

In a 7-segment display, each of the seven segments is activated for various digits.











For example, segment *a* is activated for the digits 0, 2, 3, 5, 6, 7, 8, and 9, as illustrated in Figure



7-segment Display

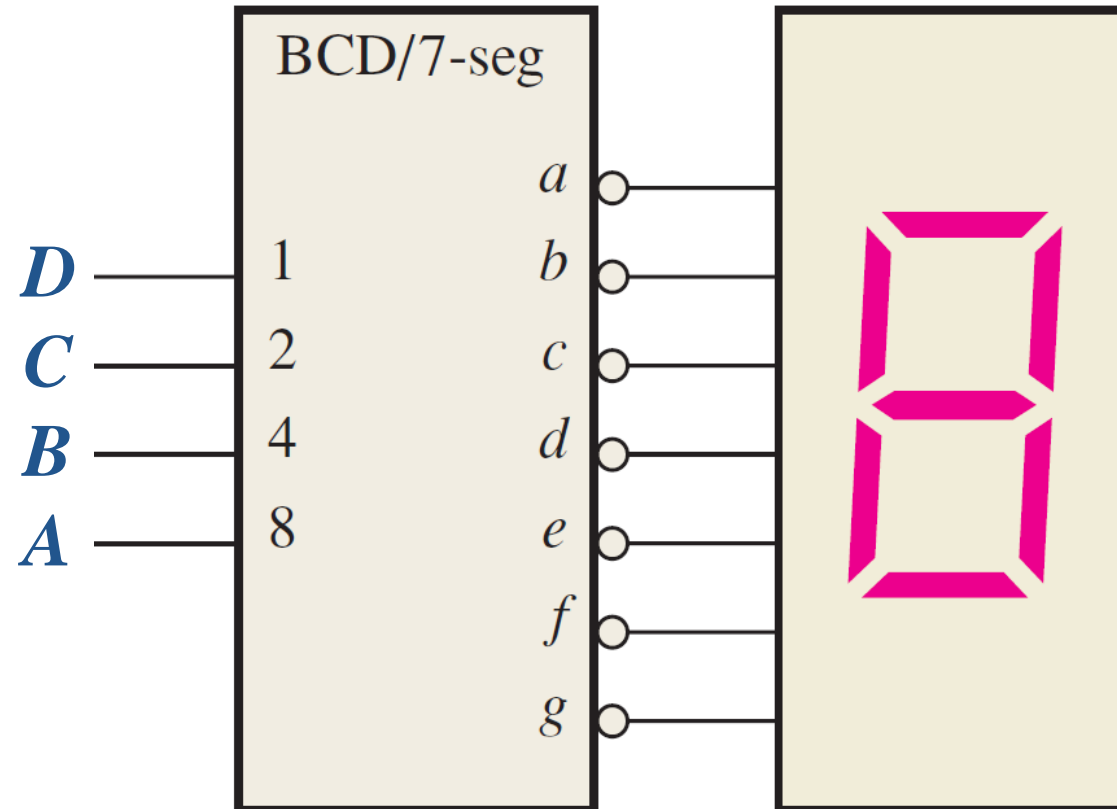


7-segment Display

Decimal Digit	Input lines				Output lines							Display pattern
	A	B	C	D	a	b	c	d	e	f	g	
0	0	0	0	0	1	1	1	1	1	1	0	
1	0	0	0	1	0	1	1	0	0	0	0	
2	0	0	1	0	1	1	0	1	1	0	1	
3	0	0	1	1	1	1	1	1	0	0	1	
4	0	1	0	0	0	1	1	0	0	1	1	
5	0	1	0	1	1	0	1	1	0	1	1	
6	0	1	1	0	1	0	1	1	1	1	1	
7	0	1	1	1	1	1	1	0	0	0	0	
8	1	0	0	0	1	1	1	1	1	1	1	
9	1	0	0	1	1	1	1	1	0	1	1	

The BCD-to-7-Segment Decoder

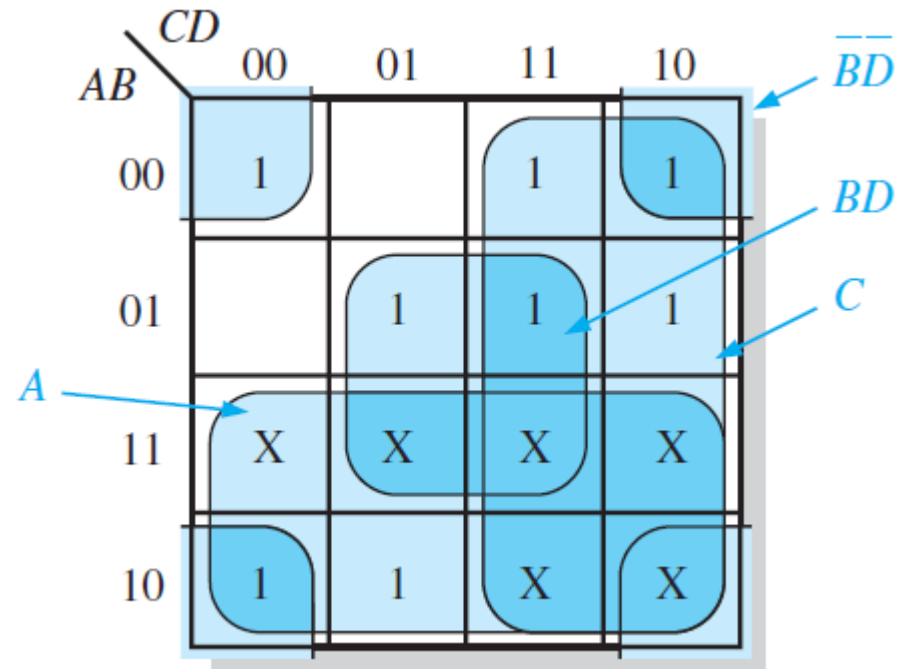
The BCD-to-7-segment decoder accepts the BCD code on its inputs and provides outputs to drive 7-segment display devices to produce a decimal readout.



The BCD-to-7-Segment Decoder

The expression for segment a is

$$a = \overline{A}\overline{B}\overline{C}\overline{D} + \overline{A}\overline{B}C\overline{D} + \overline{A}\overline{B}CD + \overline{A}B\overline{C}\overline{D} + \overline{A}B\overline{C}D + \overline{A}BC\overline{D} + \overline{A}BCD + A\overline{B}\overline{C}\overline{D}$$



From the Karnaugh map, the minimized expression for segment a is

$$a = A + C + BD + \overline{B}\overline{D}$$

The BCD-to-7-Segment Decoder

$$a = A + C + BD + \bar{B} \bar{D}$$

$$b = \bar{B} + \bar{C} \bar{D} + CD$$

$$c = B + \bar{C} + D$$

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

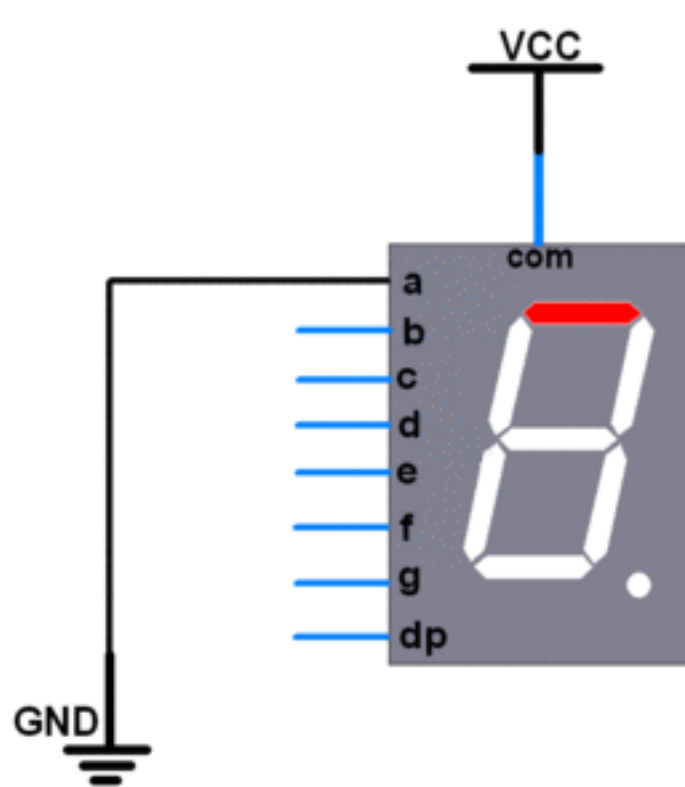
$$e = \bar{B} \bar{D} + C \bar{D}$$

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

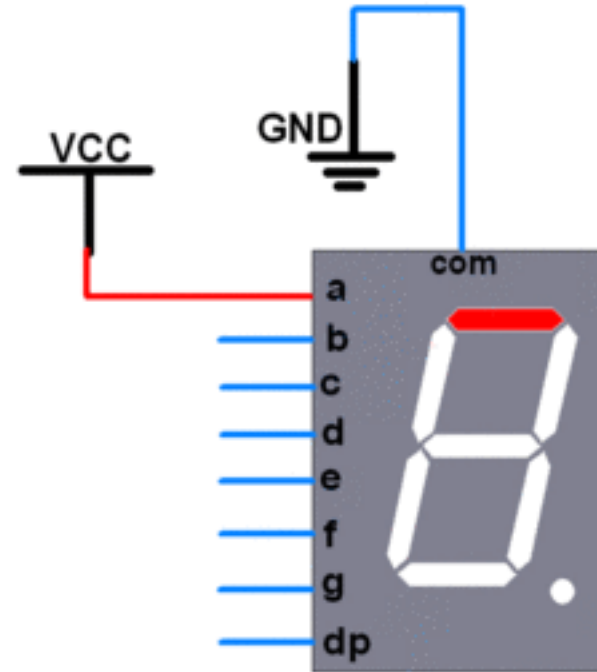
$$g = A + B \bar{C} + \bar{B} C + C \bar{D}$$

<https://www.electronicshub.org/bcd-7-segment-led-display-decoder-circuit/>

7-Segment



Common Anode



Common Cathode

The connection of 7-segments in DE10-Lite board is **common anode**.

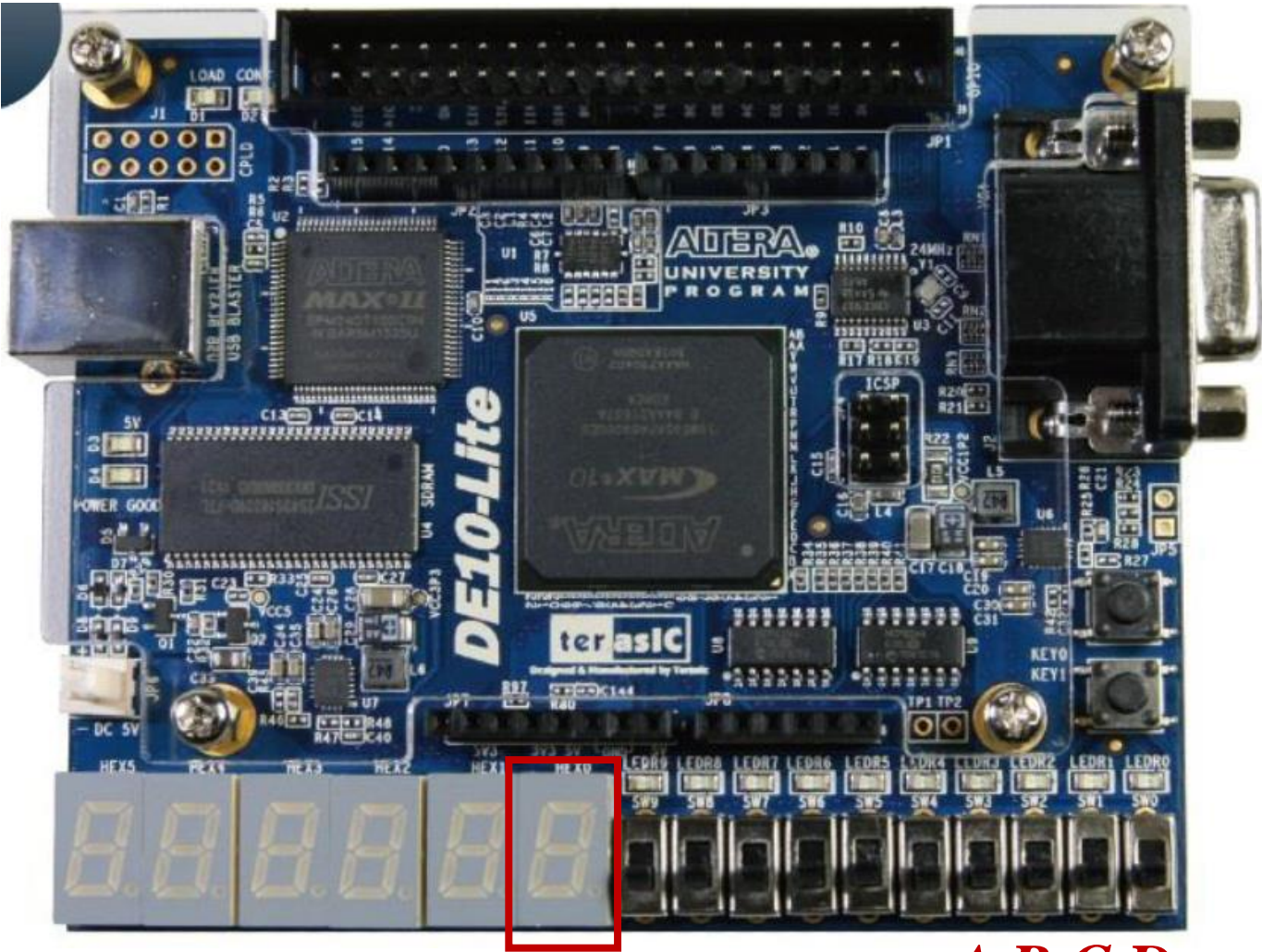
The BCD-to-7-Segment Decoder

```
module BCD7Seg(A, B, C, D, led_a, led_b, led_c, led_d,
led_e, led_f, led_g);

    input A, B, C, D;
    output led_a, led_b, led_c, led_d, led_e, led_f, led_g;

    assign led_a = ~(A | C | B&D | ~B&~D);
    assign led_b = ~(~B | ~C&~D | C&D);
    assign led_c = ~(B | ~C | D);
    assign led_d = ~(~B&~D | C&~D | B&~C&D | ~B&C | A);
    assign led_e = ~(~B&~D | C&~D);
    assign led_f = ~(A | ~C&~D | B&~C | B&~D);
    assign led_g = ~(A | B&~C | ~B&C | C&~D);
endmodule
```


FPGA Implementation

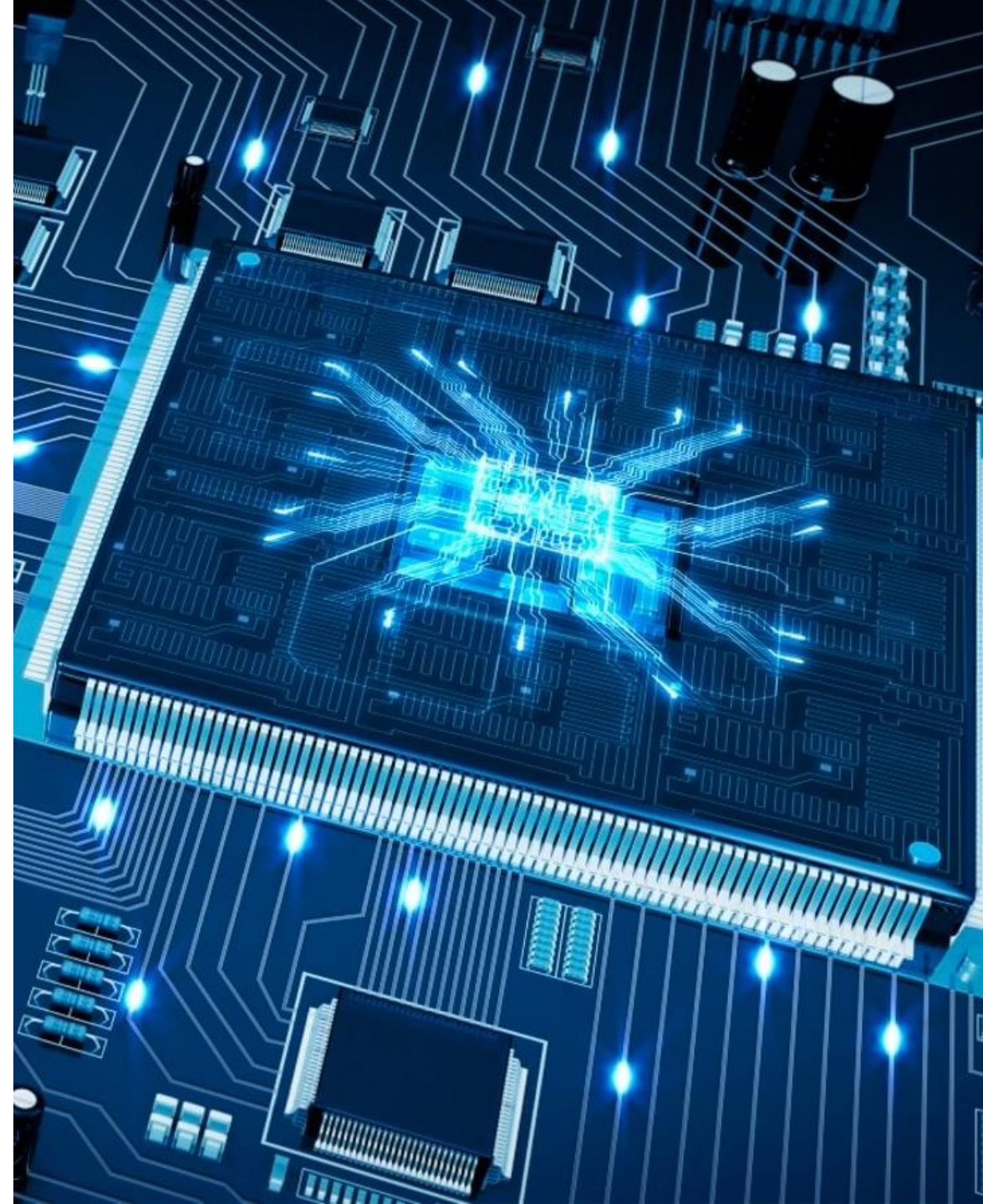


Output

A B C D

Node Name		Direction	Location
in	A	Input	PIN_C12
in	B	Input	PIN_D12
in	C	Input	PIN_C11
in	D	Input	PIN_C10
out	led_a	Output	PIN_C14
out	led_b	Output	PIN_E15
out	led_c	Output	PIN_C15
out	led_d	Output	PIN_C16
out	led_e	Output	PIN_E16
out	led_f	Output	PIN_D17
out	led_g	Output	PIN_C17

Verilog Basics



Verilog Logic Values

- A vector is a **row of logic values**.
- Verilog logic values are:
 - **1'b0** - logic 0, false, ground
 - **1'b1** - logic 1, true, power
 - **1'bX** - unknown or uninitialized
 - **1'bZ** - high impedance, floating
- The value 1 (indicating a single bit)
- 'b stands for "binary"

```
reg [7:0] V;  
V = 8'bXXXX0101;
```

V:

X	X	X	X	0	1	0	1
7	6	5	4	3	2	1	0

```
V[7] = 1'b0;  
V[6] = B & V[0];
```

Part Selects

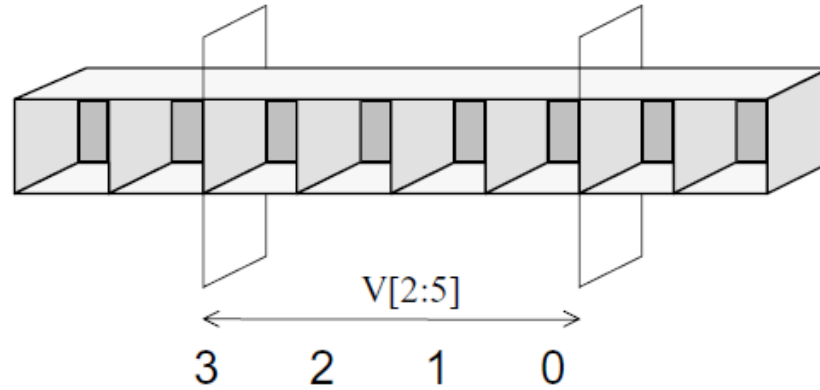
```
reg [0:7] V;  
reg [3:0] W;
```

```
W = V[2:5];
```

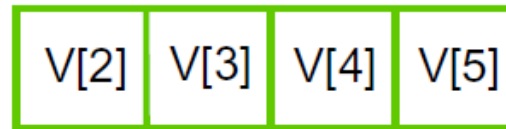
```
W[1:0] = 2'b11;
```

```
W[0:1] = 2'b11;
```

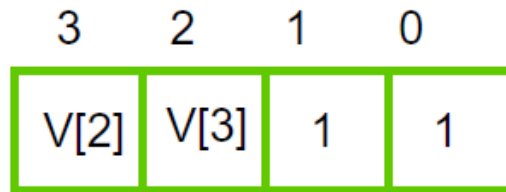
```
W[2:2] = 1'b1;
```



W:



W:



wrong direction - illegal

OK

Numbers

```
reg [7:0] V;
```

Numbers are not case sensitive

```
V = 8'b111XX000;
```

Binary 111XX000

```
V = 8'B111X_X000;
```

Binary 111XX000

_ is ignored in numbers

```
V = 8'hFF;
```

Hex = 8'b1111_1111

Leading 0's are added

```
V = 8'o77;
```

Octal = 8'b00_111_111

```
V = 8'd10;
```

Decimal = 8'b0000_1010

Numbers

- Underscores (_) are used for **formatting only** to make it easier to read.
SystemVerilog ignores them.
- Vector values can be written as integer literals, consisting of strings of 0s, 1s, Xs and Zs

Truncation and Extension

```
reg [7:0] V;
```

```
V = 8'bX;
```

8'bXXXXXXXX

```
V = 8'bZX;
```

8'bZZZZZZZX

```
V = 8'b1;
```

8'b00000001

No sign extension!!

```
V = 1'hFF;
```

1'hFF = 1'b1 (truncation) = 8'h01 (extension)

Truncation and Extension

- If all the digits of a number are not specified, the most significant bits are **filled with zeroes**. For example, `8'o77` represents `8'b00_111_111`.
- However, if the leftmost digit is X or Z, the missing bits are **filled with Xs or Zs** respectively.
- The **values of regs and wires** are considered to be **unsigned quantities**, even though negative numbers may be used in expressions.

“Disguised” Binary Numbers

- 10 and -3 is a **32-bit decimal number** and not a two bit binary one!
- -3 will be represented in **two's complement** format.
- Strings are converted to vectors by using the **8-bit ASCII code** for each character

```
reg [7:0] V;
```

Not binary!

```
V = 10;
```

$10 = 'd10 = 32'd10 = 32'b1010 \rightarrow 8'b00001010$

```
V = -3;
```

$-3 = 32'hFF_FF_FF_FD \rightarrow 8'b1111_1101$

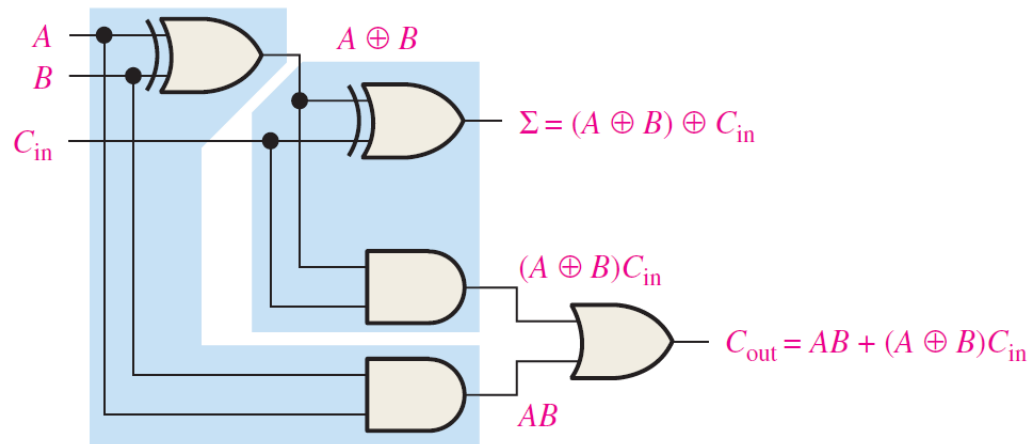
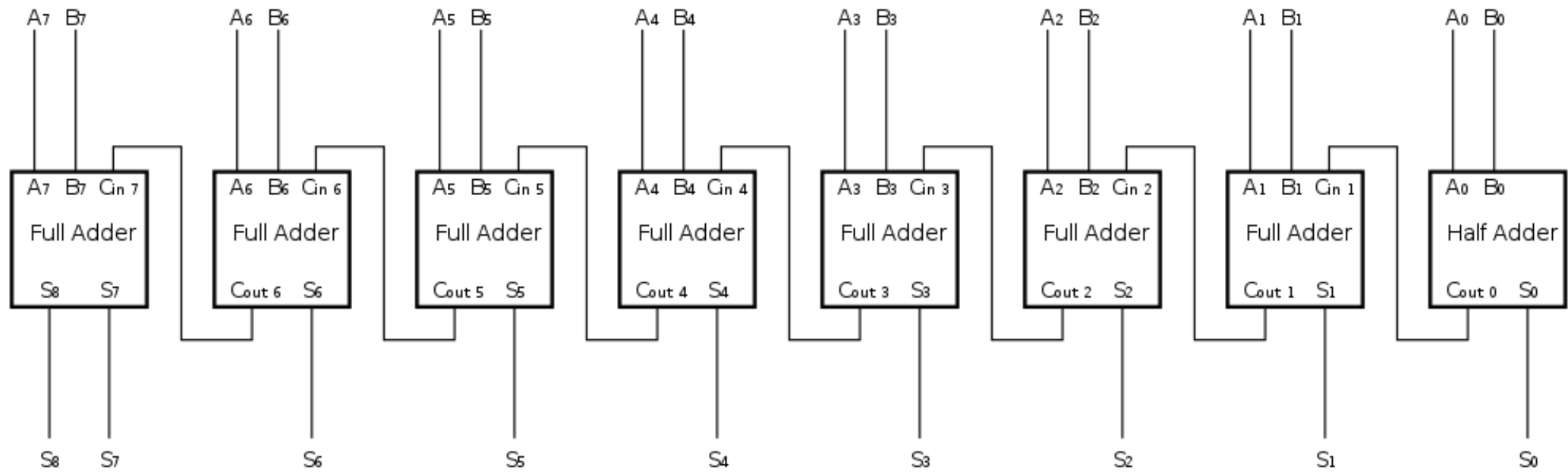
```
V = "A";
```

$ASCII = 8'd65 \rightarrow 8'b01000001$

SystemVerilog Numbers

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

8-bit Unsigned Addition



8-bit Unsigned Addition

```
module UADD8(A, B, Y);  
    input [7:0] A, B;  
    output [7:0] Y;  
  
    assign Y = A + B;  
endmodule
```

8-bit Unsigned Addition Testbench

```
module UADD8_DUT();
    reg [7:0] A, B;
    wire [7:0] Y;

    initial begin
        // 0000 0101 + 0000 0011 = 0000 1000      (5 + 3 = 8)
        A = 5; B = 3; #100;
        // 0110 1111 + 0000 0111 = 0111 0110      (111 + 7 = 118)
        A = 111; B = 8'b111; #100;
        // 1100 1000 + 0000 1010 = 1101 0010      (200 + 10 = 210)
        A = 200; B = 8'b0000_1010; #100;
    end

    UADD8 Add(A, B, Y);

    initial begin
        $monitor("%b + %b = %b \t %d + %d = %d", A, B, Y, A, B, Y);
    end
endmodule
```

8-bit Signed Addition

```
module SADD8(A, B, Y);  
    input signed [7:0] A, B;  
    output signed [7:0] Y;  
  
    assign Y = A + B;  
endmodule
```

8-bit Signed Addition Testbench

```
module SADD8_DUT();
    reg signed [7:0] A, B;
    wire signed [7:0] Y;

    initial begin
        // 0000 0101 + 0000 0011 = 0000 1000      (5 + 3 = 8)
        A = 5; B = 3; #100;
        // 0110 1111 + 0000 0111 = 0111 0110      (111 + 7 = 118)
        A = 111; B = 8'b111; #100;
        // 1100 1000 + 0000 1010 = 1100 1001      (-56 + 10 = -46)
        A = 200; B = 8'b0000_1010; #100;
    end

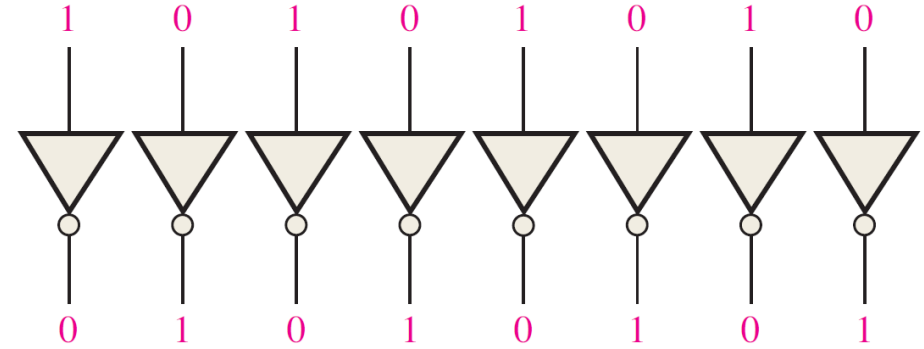
    SADD8 Add(A, B, Y);

    initial begin
        $monitor("%b + %b = %b \t %d + %d = %d", A, B, Y, A, B, Y);
    end
endmodule
```

8-bit 1's Complement With Testbench

```
module Complement1s (A, Y);  
    input [7:0] A;  
    output [7:0] Y;  
    assign Y = ~A;  
endmodule
```

```
module Complement1s_DUT();  
    reg [7:0] A;  
    wire [7:0] Y;  
  
    initial begin  
        A = 8'b1100_1001; #200;  
        A = 8'b1111_1111; #200;  
        A = 8'b0000_1111; #200;  
        A = 8'b0011_1100; #200;  
    end  
  
    Complement1s Ones (A, Y);  
endmodule
```



More About Formatting

- **Special formatting strings:**

%b	binary
%o	octal
%d	decimal
%h	hexadecimal
%s	ASCII string
%v	value and drive strength
%t	time (described later)
%m	module instance
\n	newline
\t	tab
\nnn	nnn is octal ASCII value of character
\\	\
\"	"

More About Formatting

- ◆ **One format string with correct number of values**

```
$display ("%b %b", Expr1, Expr2);
```

- ◆ **Two format strings**

```
$display ("%b", Expr1, " %b", Expr2);
```

Equivalent

- ◆ **Too many values**

```
$display ("%b", Expr1, Expr2);
```

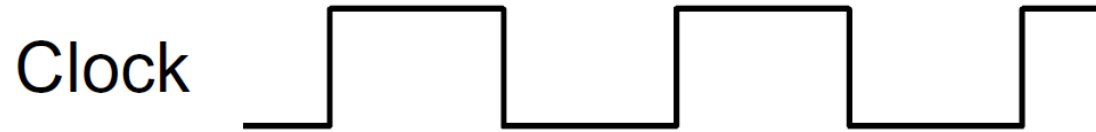
Expr2 is written in decimal

- ◆ **Too few values**

```
$display ("%b %b %b", Expr1, Expr2);
```

ERROR!!

Clock Generator



```
module ClockGen (CLK);  
    output reg CLK;  
  
    initial  
        CLK = 0;  
  
    always  
        #50 CLK = ~CLK;  
endmodule
```

Always

- An **always** block is very similar to an **initial** block, but instead of executing just once, it executes repeatedly throughout simulation.
- Having reached the end, it immediately starts again at the beginning, and so defines an **infinite loop**.
- Note that anything **assigned within the always block** must be defined as a **register**.

Clock Generator Testbench

```
module ClockGen_DUT();  
    wire CLK;  
    ClockGen CG1 (CLK);  
endmodule
```

```
module ClockGen_DUT2();  
    wire CLK;  
    ClockGen CG1 (CLK);  
  
    initial  
        #500 $stop;  
    initial  
        #1000 $finish;  
endmodule
```

\$stop and \$finish

- If a design contains always statements like those on the previous slide, then potentially it will **simulate indefinitely**.
- To prevent this, you can tell the simulator to execute for a **limited time**.
- The system task **\$stop** is used to set breakpoints. When **\$stop** is called, **simulation is suspended**.
- The system task **\$finish** is used to quit simulation.

Using Registers

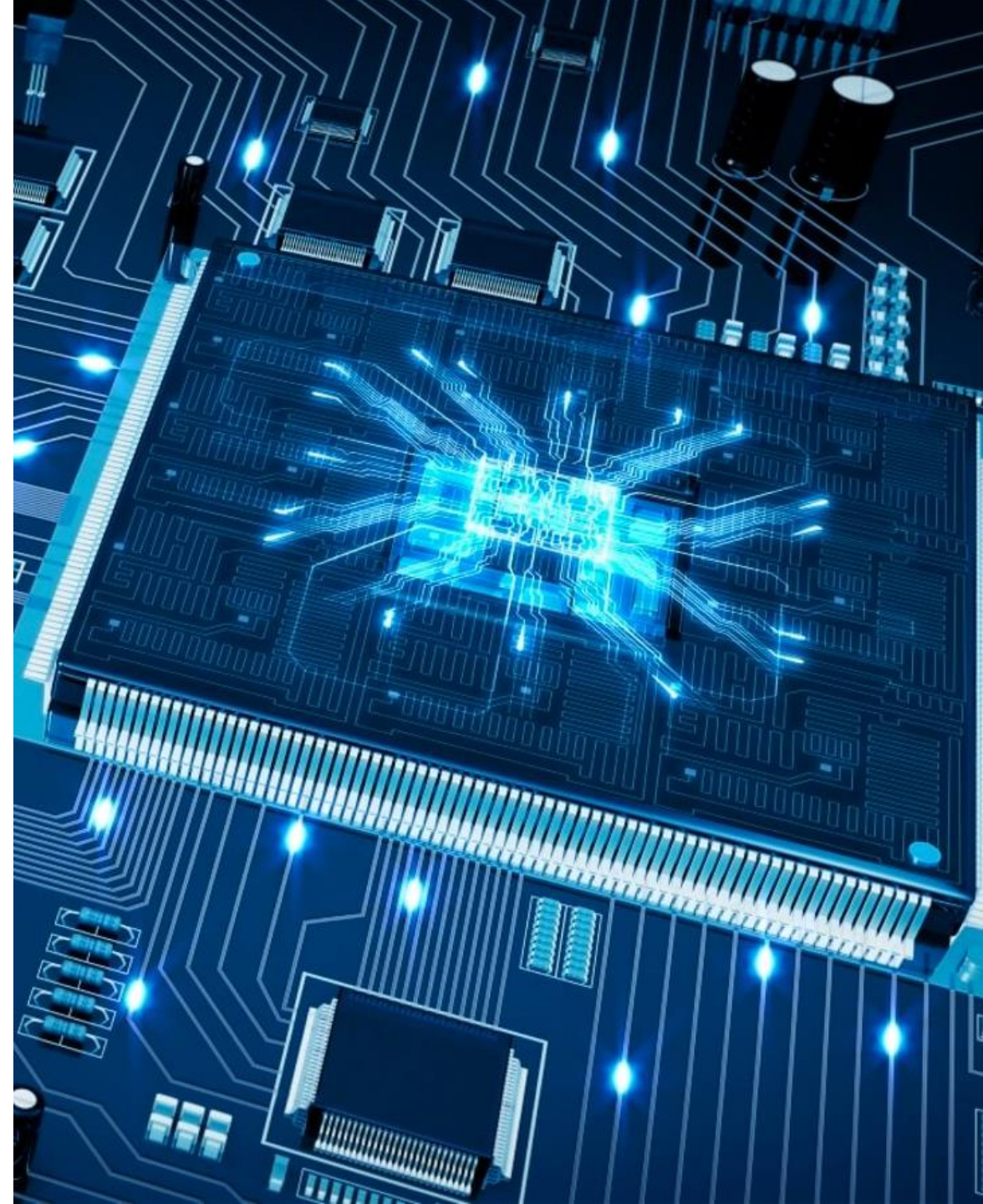
- Necessary when assigning in initial or always.
- Only needed when assigning in initial or always.
- Outputs can be regs.

```
module UsesRegs (OutReg);  
  output [7:0] OutReg;  
  reg      [7:0] OutReg;  
  
  reg R;  
  
  initial  
    R = ...  
  
  always  
    OutReg = 8'b...  
  
endmodule
```

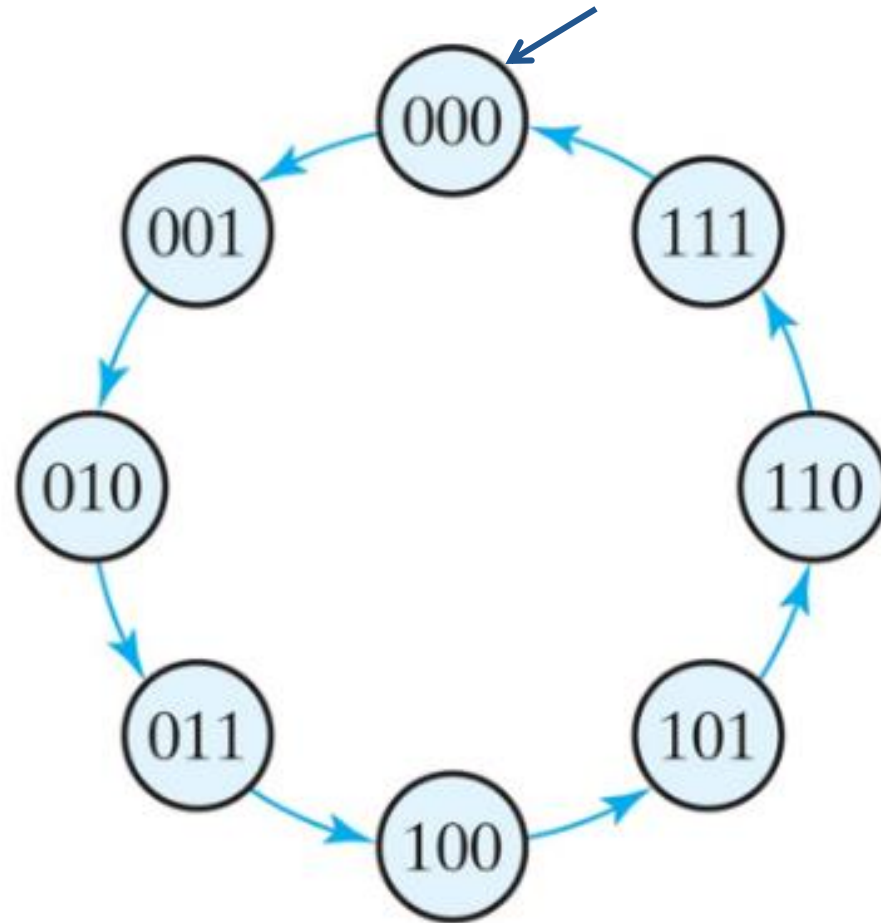
Declarations agree

Must be registers

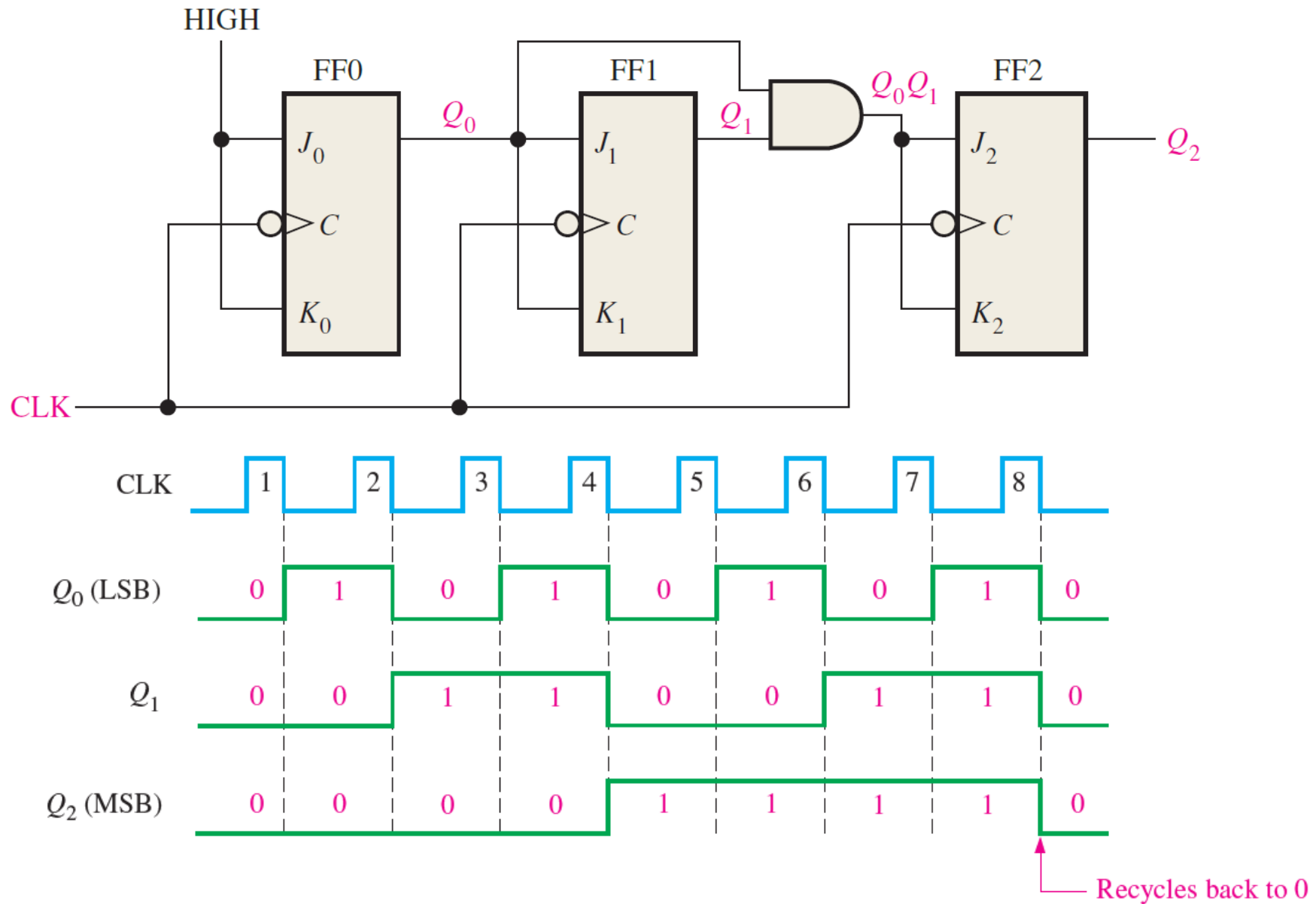
Counters



State Diagram



3-bit Counter



3-bit Counter

```
module counter (clk, count);  
    input clk;  
    output reg [2:0] count = 0;  
  
    always @(posedge clk)  
        count <= count + 1;  
endmodule
```

3-bit Counter

```
module counter (clk, reset, count);  
    input clk, reset;  
    output reg [2:0] count;  
  
    always @(posedge clk or posedge reset)  
        if(reset)  
            count <= 0;  
        else  
            count <= count + 1;  
endmodule
```

3-bit Counter

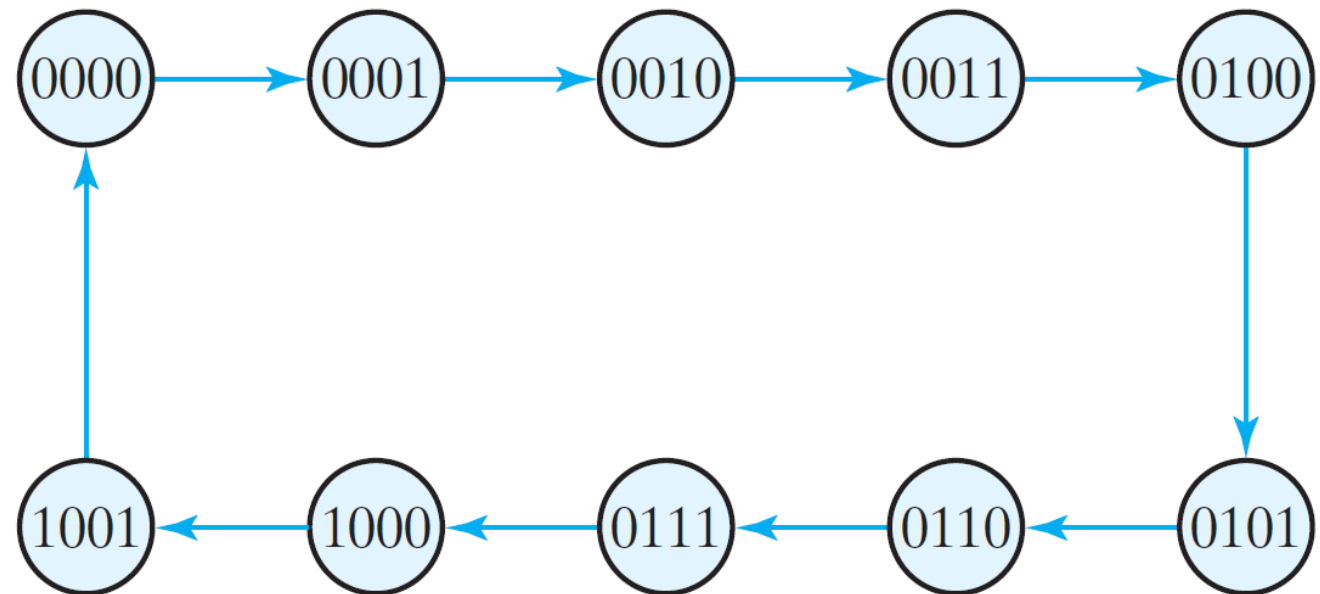
```
module counter (clk, reset, count);  
    input clk, reset;  
    output reg [2:0] count;  
  
    always @(posedge clk or posedge reset)  
        if(reset)  
            count <= 0;  
        else  
            count <= count + 1;  
endmodule
```

```
module counter_dut ();  
    wire clk;  
    reg reset;  
    wire [2:0] count;  
  
    initial begin  
        reset = 1; #50;  
        reset = 0;  
    end  
  
    counter cnt (clk, reset, count);  
endmodule
```

BCD Counter

States of a BCD decade counter.

Clock Pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10 (recycles)	0	0	0	0



BCD Counter

```
module bcd_counter (clk, reset, count);  
    input clk, reset;  
    output reg [3:0] count;  
  
    always @(posedge clk or posedge reset)  
        if(reset)  
            count <= 0;  
        else if(count == 9)  
            count <= 0;  
        else  
            count <= count + 1;  
endmodule
```










BCD Counter

```
module bcd_counter (clk, reset, count);
    input clk, reset;
    output reg [3:0] count;










    always @(posedge clk or posedge reset)
        if(reset)
            count <= 0;
        else if(count == 9)
            count <= 0;
        else
            count <= count + 1;
endmodule

module bcd_counter_dut ();
    wire clk;
    reg reset;
    wire [3:0] count;
    initial begin
        reset = 1; #50;
        reset = 0;
    end
    bcd_counter bcd_cnt (clk, reset, count);
endmodule
```


BCD Counter

Node Name	Direction	Location
 clk	Input	PIN_C11
 leds[6]	Output	PIN_C14
 leds[5]	Output	PIN_E15
 leds[4]	Output	PIN_C15
 leds[3]	Output	PIN_C16
 leds[2]	Output	PIN_E16
 leds[1]	Output	PIN_D17
 leds[0]	Output	PIN_C17
 reset	Input	PIN_C10

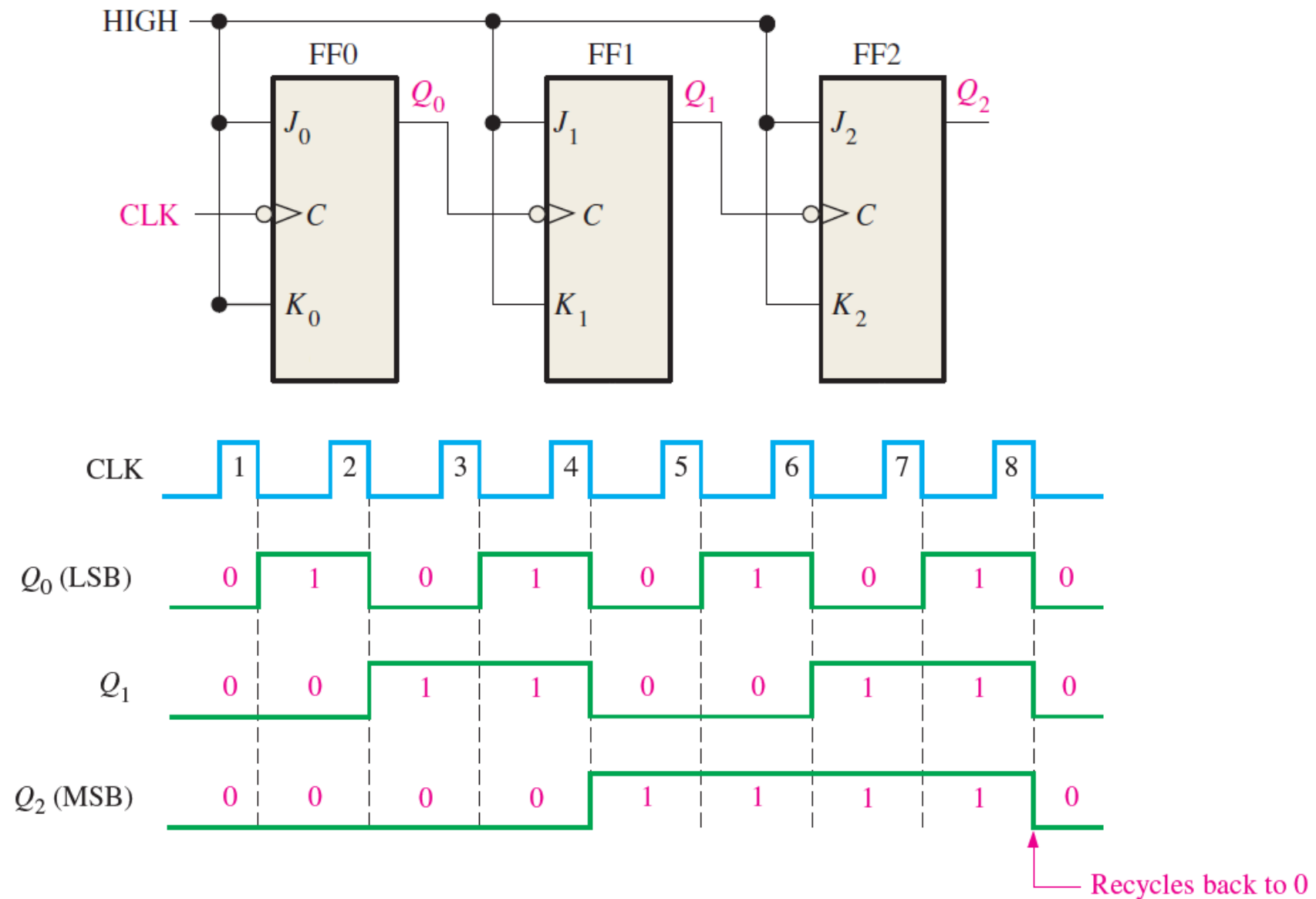
BCD Counter

Node Name	Direction	Location
 clk	Input	PIN_B8
 leds[6]	Output	PIN_C14
 leds[5]	Output	PIN_E15
 leds[4]	Output	PIN_C15
 leds[3]	Output	PIN_C16
 leds[2]	Output	PIN_E16
 leds[1]	Output	PIN_D17
 leds[0]	Output	PIN_C17
 reset	Input	PIN_C10

Clock Divider

- The clock frequency of the MAX 10 FPGA is 50MHz.
- To generate a 1-Hz clock, you need to implement a clock divider that slows down the FPGA clock.
- A clock divider is a counter.
- Our implementation divides the FPGA clock by 50 million counts.
- 25 million counts for One pulse, and 25 million counts for Zero pulse.

3-bit Asynchronous Counter



Clock Divider

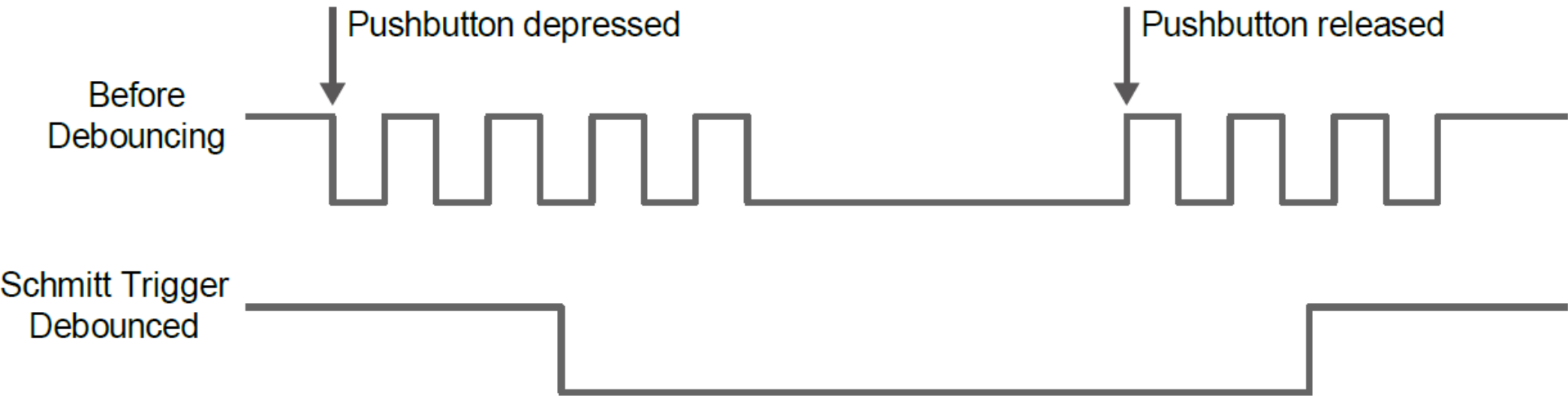
```
module clock_divider (clk, reset, CLK1Hz);
    input clk, reset;
    output reg CLK1Hz;
    reg [24:0] count;

    always @(posedge clk or posedge reset)
    begin
        if(reset)
            begin
                count <= 0;
                CLK1Hz <= 0;
            end
        else
            begin
                if(count < 25_000_000)
                    count <= count + 1;
                else
                    begin
                        CLK1Hz <= ~CLK1Hz;
                        count <= 0;
                    end
            end
        end
    end
endmodule
```

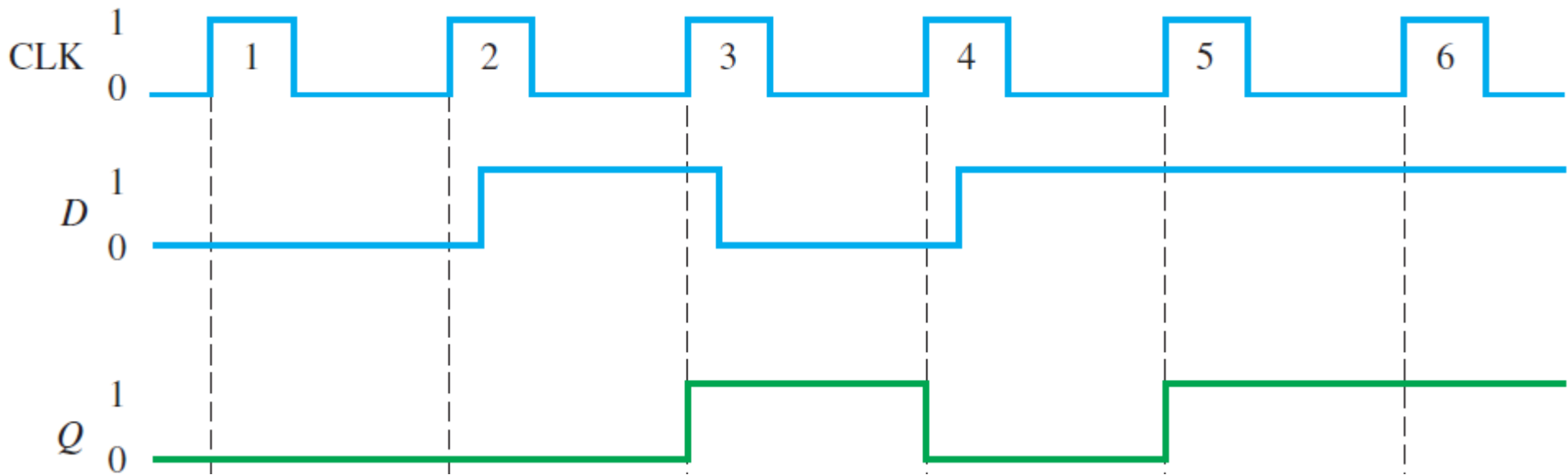
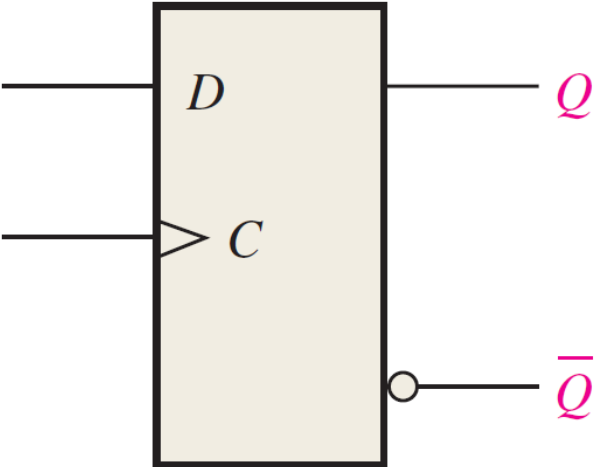
Bouncing



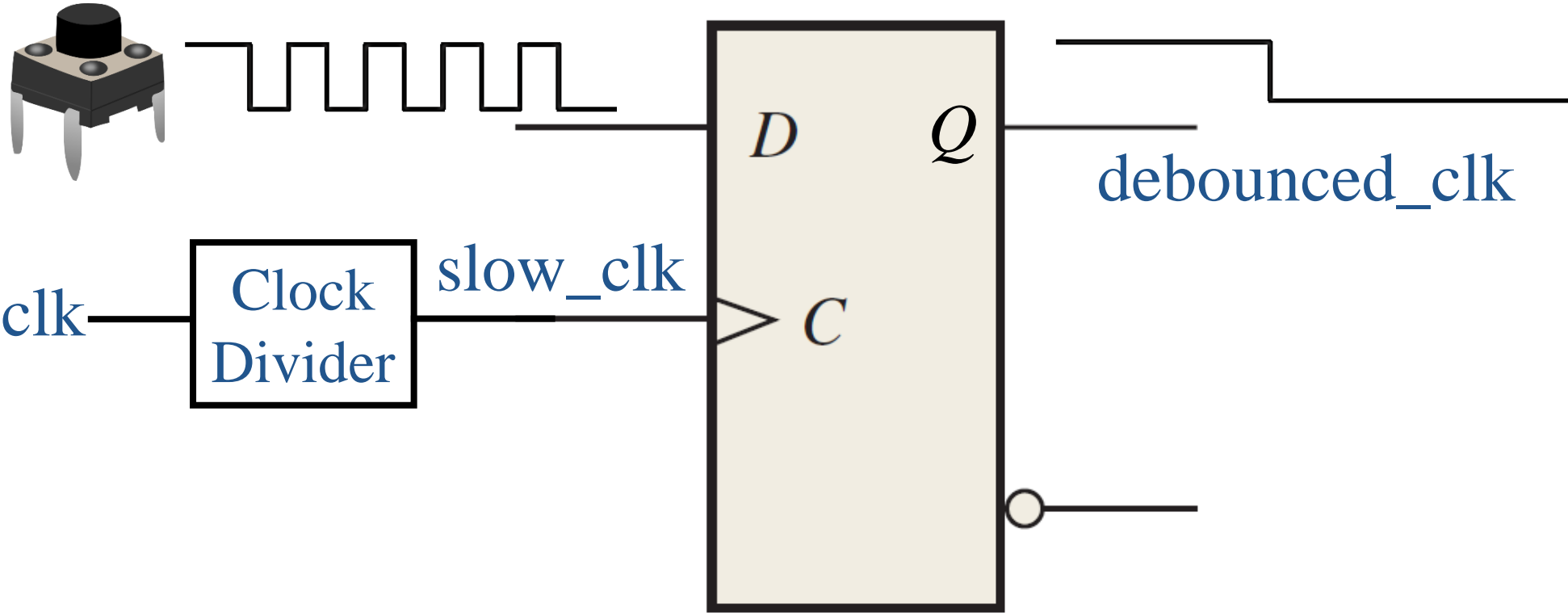
Debouncing



D Flip-Flop



Debouncing



Logical Shifter

- Shifts the number to the left (LSL) or right (LSR) and fills empty spots with 0's.
- Logical Shift Left (**LSL**) or <<
- Logical Shift Right (**LSR**) or >>

$$A * 2^N$$

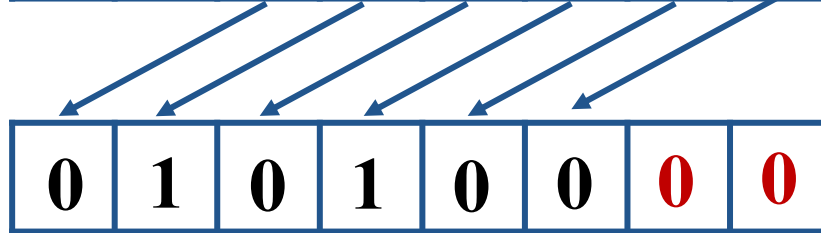
$$A / 2^N$$

Logical Shift Left

00010100 \ll 2



$(20)_{10}$

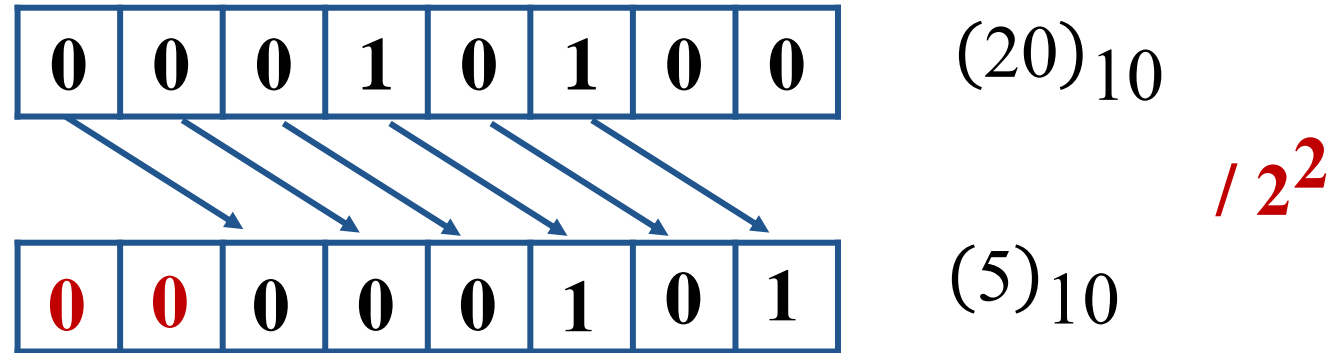


$(80)_{10}$

$\ast 2^2$

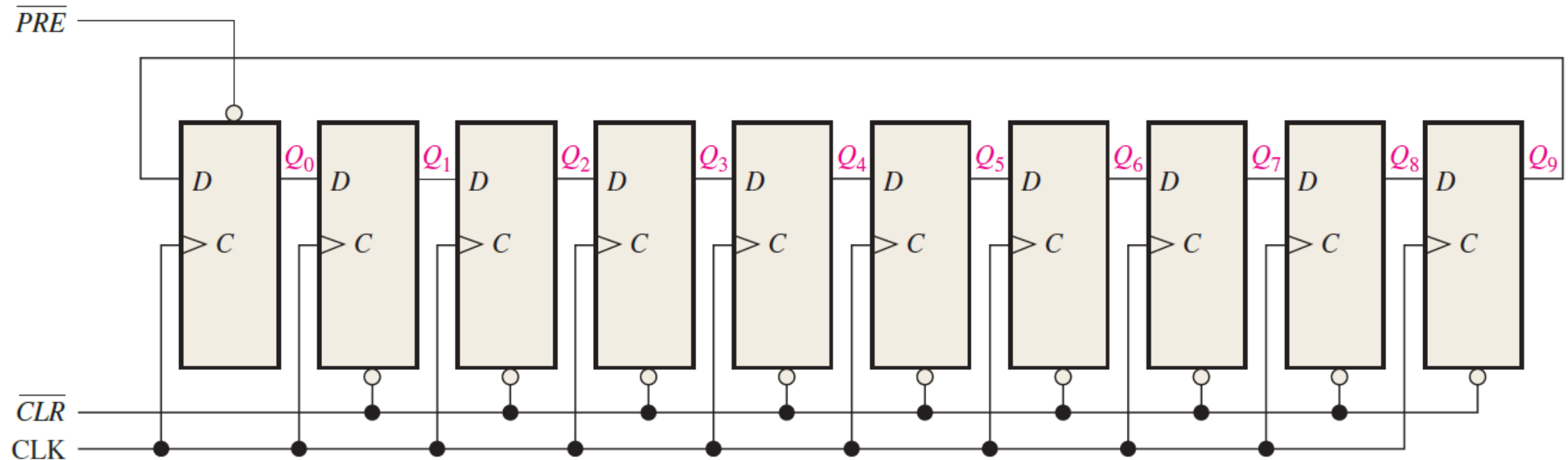
Logical Shift Right

00010100 >> 2



10-bit One-Hot Counter

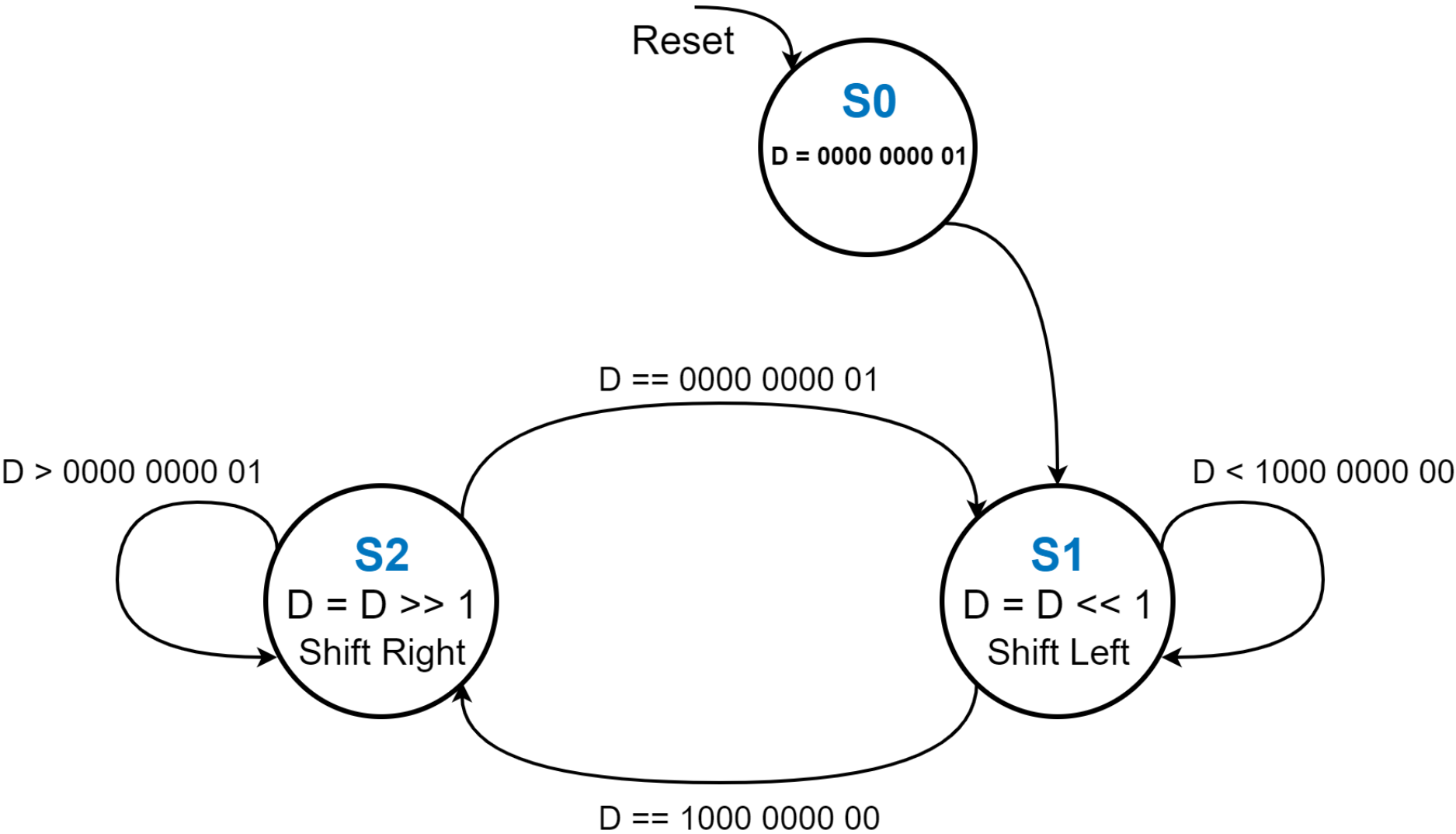
- In the case of a 10-bit ring counter, there is a **unique output** for each decimal digit.



10-bit Up/Down One-Hot Counter

[illegible]

10-bit Up/Down One-Hot Counter FSM



10-bit Up/Down One-Hot Counter FSM

```
module one_hot_counter_fsm (clk, reset, D);
    input clk, reset;
    Output reg [9:0] D;
    parameter S0 = 2'b00,    // initial state
              S1 = 2'b01,    // shift left state
              S2 = 2'b10;    // shift right state
    reg [1:0] State;

    always @(posedge clk or posedge reset)
    begin
        if(reset)
            begin
                State = S0;
                D <= 10'b0000_0000_01;
            end
        else
            begin
                case(State)
                    S0: State <= S1;
                    S1: if(D < 10'b1000_0000_00)
                        D <= D << 1;
                        else
                            State <= S2;
                    S2: if(D > 10'b0000_0000_01)
                        D <= D >> 1;
                        else
                            State <= S1;
                endcase
            end
        end
    end
```

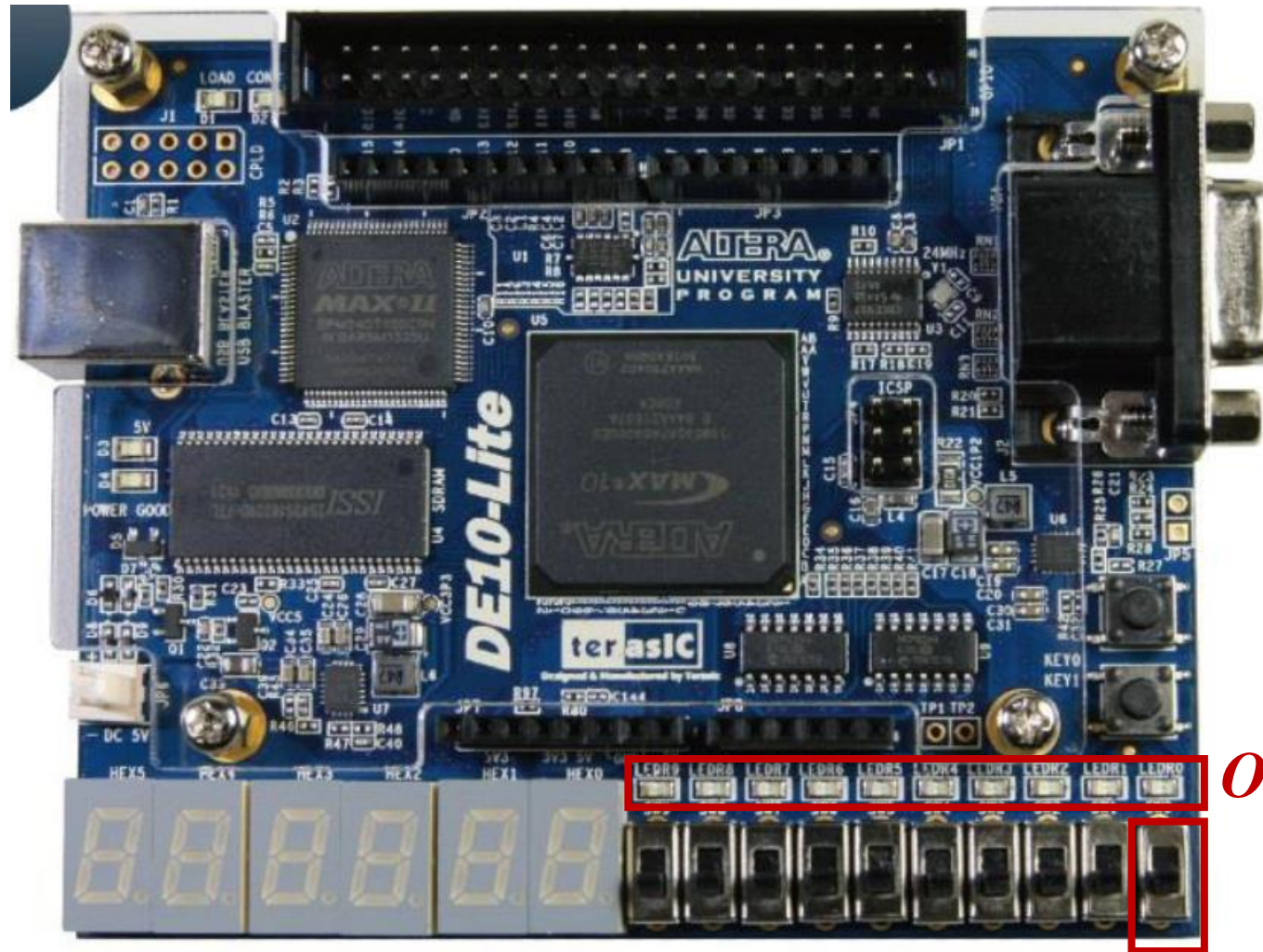
10-bit Up/Down One-Hot Counter FSM

```
module fsm_dut ();  
    reg clk, reset;  
    wire [9:0] D;  
  
    initial  
    begin  
        clk = 0;  
        reset = 1; #100;  
        reset = 0;  
    end  
  
    always  
        #100 clk = ~clk;  
  
    one_hot_counter_fsm fsm (clk, reset, D);  
endmodule
```

Do Files

```
vsim fsm_dut  
add wave -position insertpoint sim:/fsm_dut/*  
run 5000
```



























10-bit Up/Down One-Hot Counter FSM



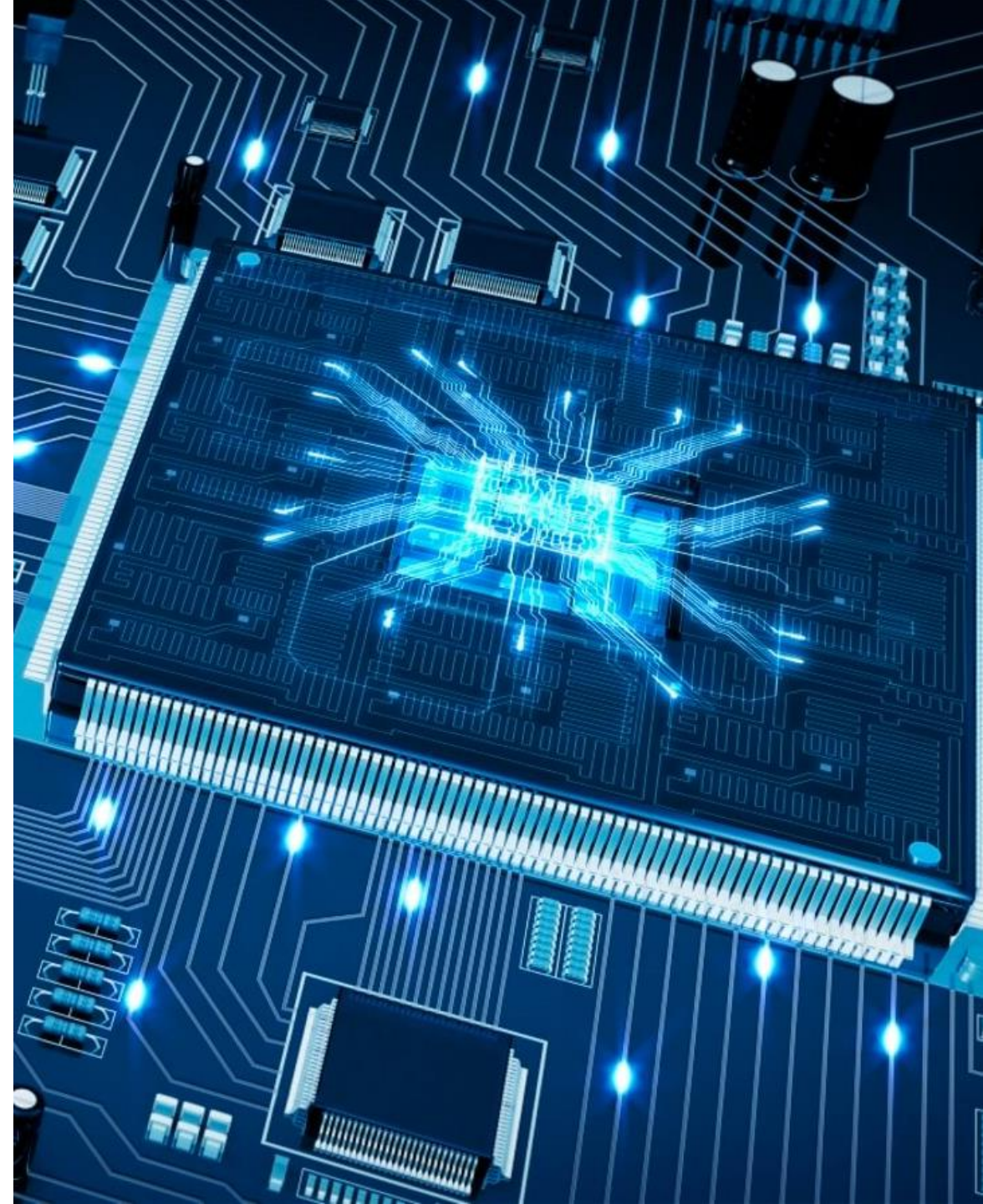
Output

Reset

One-Hot Counter FSM FPGA Implementation

 	Node Name	Direction	Location
	 CLK	Input	PIN_P11
	 D[9]	Output	PIN_B11
	 D[8]	Output	PIN_A11
	 D[7]	Output	PIN_D14
	 D[6]	Output	PIN_E14
	 D[5]	Output	PIN_C13
	 D[4]	Output	PIN_D13
	 D[3]	Output	PIN_B10
	 D[2]	Output	PIN_A10
	 D[1]	Output	PIN_A9
	 D[0]	Output	PIN_A8
	 Reset	Input	PIN_C10
	<<new node>>		

ALU



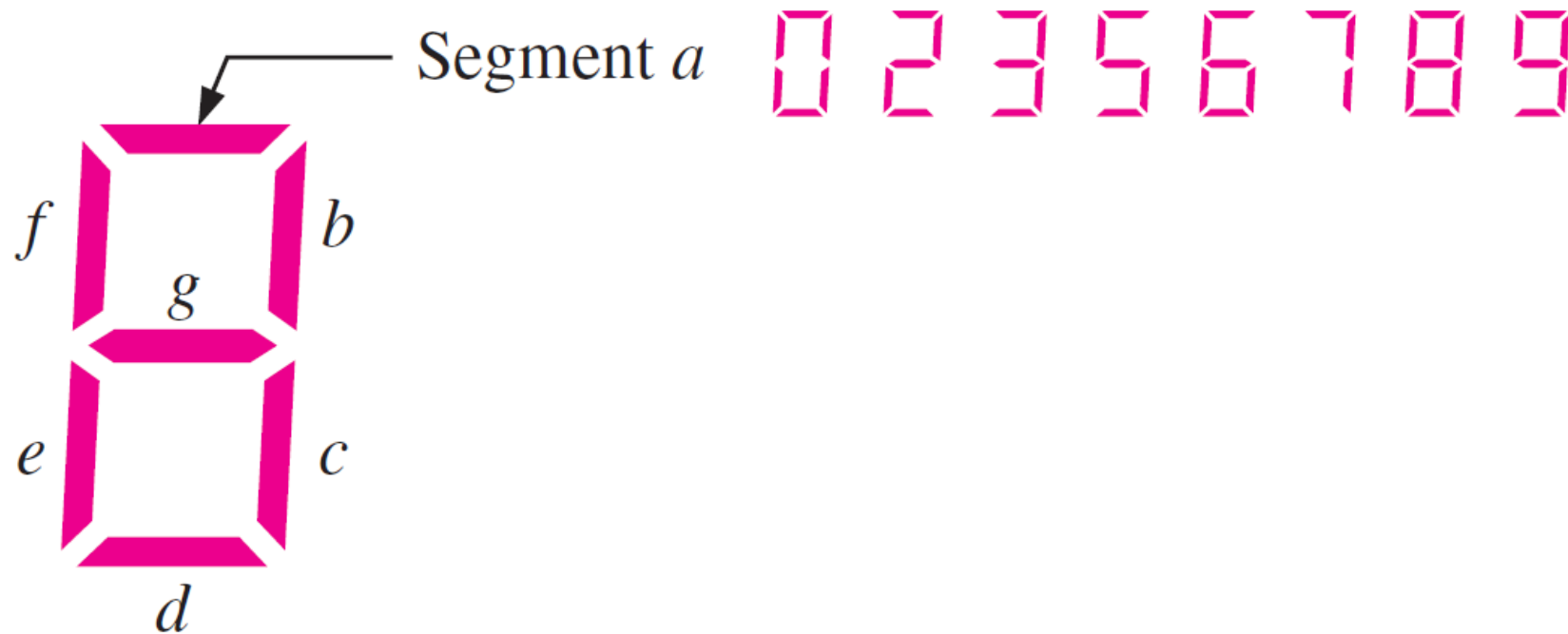
- An Arithmetic/Logical Unit (ALU) combines a variety of **mathematical and logical operations** into a single unit.
- For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations.
- The ALU forms the **heart of most computer systems**.
- A typical ALU is constructed of **many thousands of logic gates**.
- Logical operations are equivalent to the **basic gate operations that you are familiar with**.

$F_{2:0}$	Operation	Output
000	$A \& B$	On 2 Leds
001	$A B$	On 2 Leds
010	$A + B$	On 1 Seven Segments
011	$A - B$	On 1 Seven Segments
100	$A * B$	On 1 Seven Segments
101	$A > B$	Draw A on Seven Segment
110	$A < B$	Draw B on Seven Segment
111	$A = B$	Draw = on 1 Seven Segments

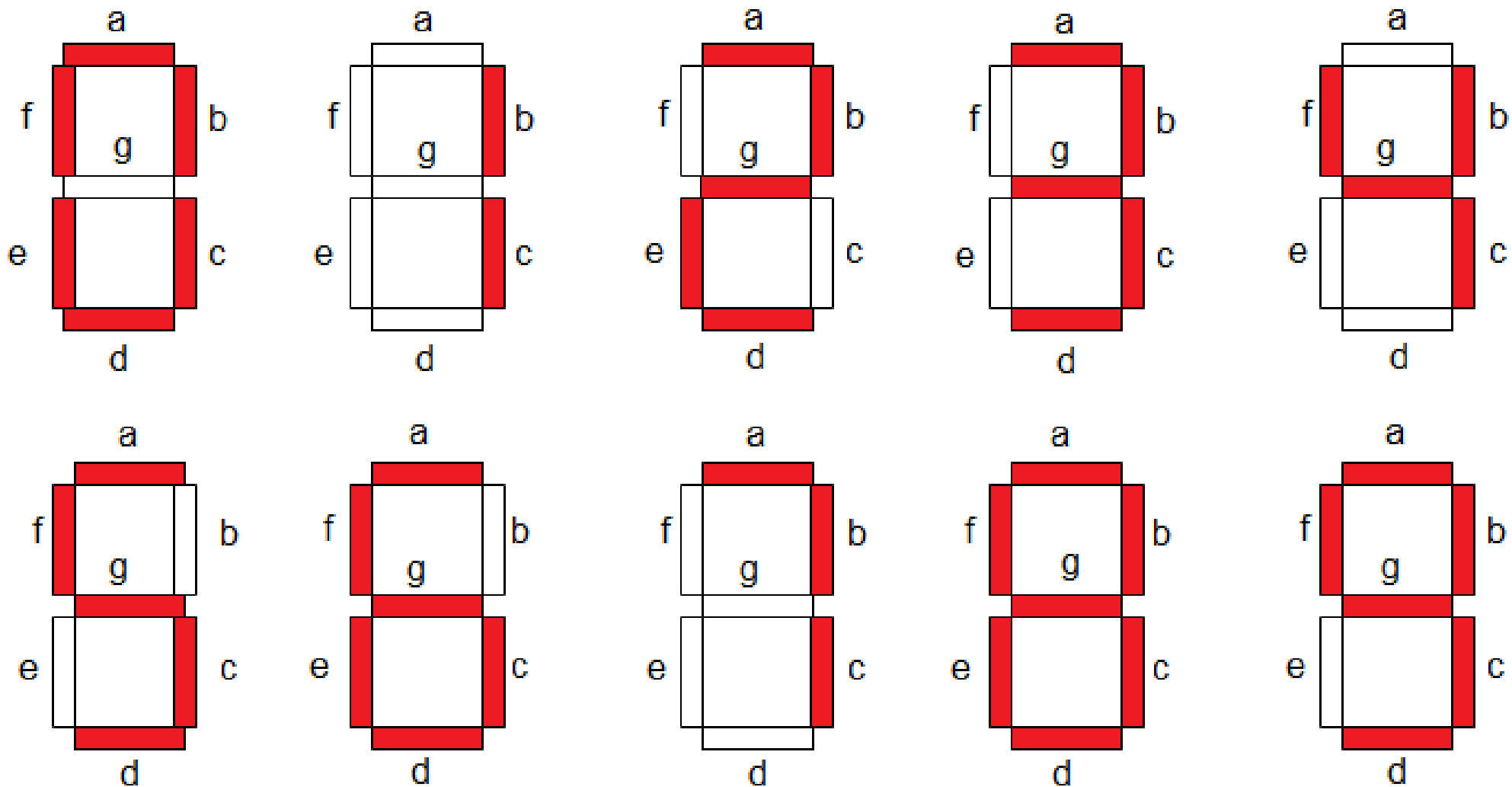
7-segment Display

In a 7-segment display, each of the seven segments is activated for various digits.

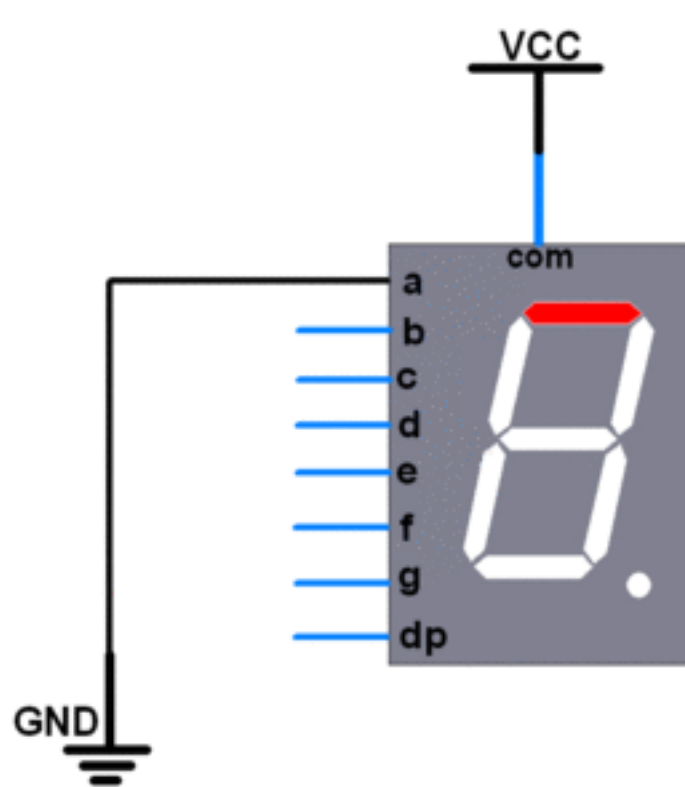
For example, segment *a* is activated for the digits 0, 2, 3, 5, 6, 7, 8, and 9, as illustrated in Figure



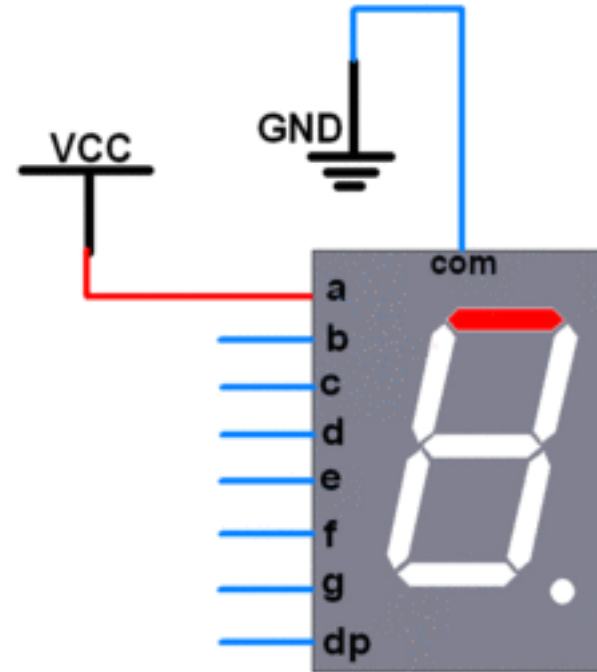
7-segment Display



7-Segment



Common Anode



Common Cathode

The connection of 7-segments in DE10-Lite board is **common anode**.

7-Segment Decoder

```
module sevenSegments (bcd, dec);
    input [3:0] bcd;
    output reg [6:0] dec;

    always @(bcd)
    begin
        case(bcd)
            4'b0000 : dec = ~7'b11111110;    // 0
            4'b0001 : dec = ~7'b01100000;    // 1
            4'b0010 : dec = ~7'b1101101;     // 2
            4'b0011 : dec = ~7'b11111001;    // 3
            4'b0100 : dec = ~7'b0110011;     // 4
            4'b0101 : dec = ~7'b1011011;     // 5
            4'b0110 : dec = ~7'b1011111;     // 6
            4'b0111 : dec = ~7'b1110000;     // 7
            4'b1000 : dec = ~7'b1111111;     // 8
            4'b1001 : dec = ~7'b1111011;     // 9
            4'b1010 : dec = ~7'b1110111;     // A
            4'b1011 : dec = ~7'b0011111;     // b
            4'b1100 : dec = ~7'b1001000;     // =
            default : dec = ~7'b11111110;    // 0
        endcase
    end
endmodule
```

ALU

```
module ALU (A, B, F, Y);  
    input [1:0] A, B;  
    input [2:0] F;  
    output reg [3:0] Y;  
  
    always @(*)  
    begin  
        case(F)  
            3'b000: Y = A & B;  
            3'b001: Y = A | B;  
            3'b010: Y = A + B;  
            3'b011: Y = A - B;  
            3'b100: Y = A * B;  
            3'b101: Y = A > B ? 4'b1010 : 4'b1011;  
            3'b110: Y = A < B ? 4'b1011 : 4'b1010;  
            3'b111: Y = A == B ? 4'b1100 : 4'b0000;  
            default: Y = 4'b0000 ;  
        endcase  
    end  
endmodule
```

ALU

```
module ALU_to_decoder (A, B, F, segment_leds, leds);  
    input [1:0] A, B;  
    input [2:0] F;  
    output [6:0] segment_leds;  
    output [1:0] leds;  
  
    wire [3:0] Y;  
  
    ALU ALU_dut (A, B, F, Y);  
  
    sevenSegments seg (Y, segment_leds);  
  
    assign leds = Y[1:0];  
endmodule
```

