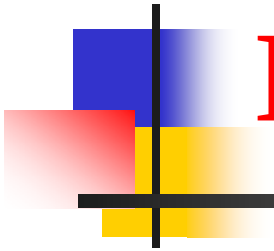
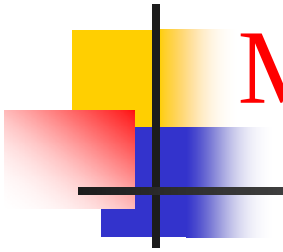


Hardware Description Languages (HDLs)-- Verilog

References:

1. M. Morris Mano and Michael D. Ciletti, *Digital Design*, 5th ed., 2013, Prentice Hall. (§3.9, §4.12, §5.6, §6.6, §8.3)
2. David Money Harris & Sarah L. Harris, *Digital Design and Computer Architecture*, 1st ed., 2007, Morgan Kaufmann (Ch4).

J.J. Shann



Mano & Ciletti: HDL Overview

3-9 Hardware Description Language

4-12 HDL Models of Combinational Circuits

5-6 Synthesizable HDL Models of Sequential Circuits

6-6 HDL for Registers and Counters

7-2 Random-Access Memory: Memory Description in HDL

8-3 Register Transfer Level in HDL

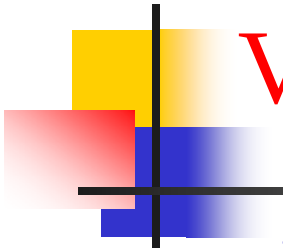
Mano 3-9

Hardware Description Language

J.J. Shann

3-9 Hardware Description Language

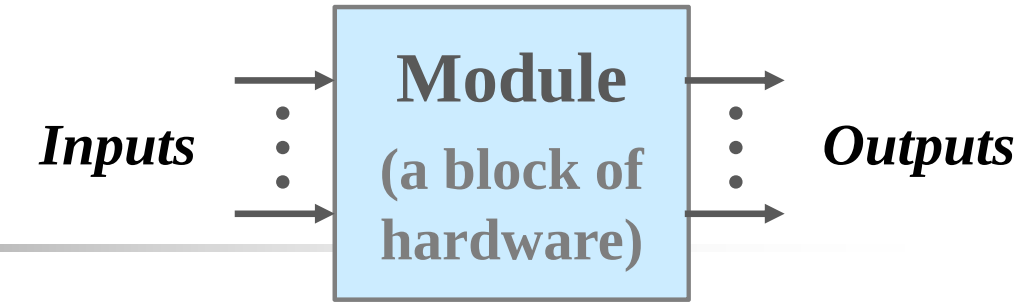
- Hardware description language (HDL)
  - Circuit descriptions in HDL resemble code in a programming language.
  - Note: The code is intended to represent ***digital hardware***.
- Modeling styles of HDL:
  - *gate-level modeling*
  - *dataflow modeling*
  - *behavioral modeling*
- 2 leading HDLs:
  - Verilog (✓ for this course)
  - VHDL



Verilog

- Verilog: (✓ for this course)
  - was developed by Gateway Design Automation for logic simulation in 1984.
    - Gateway was acquired by Cadence in 1989.
  - was made an open standard in 1990.
  - became an IEEE standard in 1995 & was updated in 2001.
    - IEEE: the institute of Electrical and Electronics Engineers
  - file name: **.v**
- SystemVerilog: (*)
  - was extended from Verilog in 2005
    to streamline idiosyncracies and
    to better support modeling and verification of systems.
  - file name: **.sv**

Modules

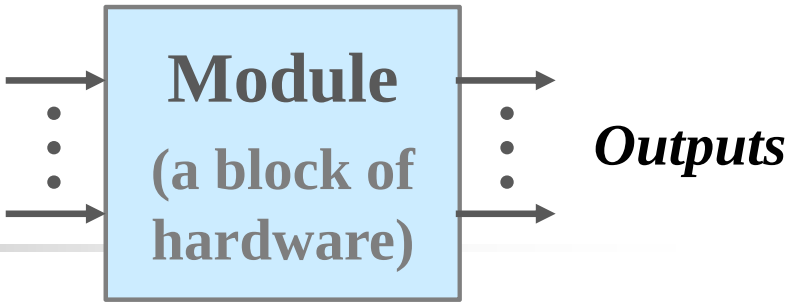


- Module:
  - a block of hardware w/ inputs and outputs
  - e.g.s: an AND gate, a MUX, a priority ckt, …
- Styles for describing module:
  - *gate-level*: use instantiations of predefined and user-defined primitive gates
  - *behavioral* model: describe what a module does
  - *structural* model: describe how a module is built from simpler pieces ⇒ an application of *hierarchy*

Design Flow of an Integrated Circuit (IC)

- Major steps in the design flow of an IC:
  - Design entry
    - creates an HDL-based description of the functionality that is to be implemented in hardware
  - Function simulation or verification
    - displays the behavior of a digital system through the use of a computer
  - Logic synthesis
    - derives a list of physical components and their interconnections, *netlist*, from the model of a digital system described in an HDL
  - Timing verification
    - confirms that the fabricated IC will operate at a specified speed
  - Fault simulation
    - compares the behavior of an ideal ckt w/ the behavior of a ckt that contains a process-induced flaw

Logic Simulation



- Simulation:
  - inputs are applied to a module and outputs are checked to verify that the module operates correctly
  - E.g.: Simulation waveforms (Harris, p.170, Fig 4.1)

$$y = a'b'c' + ab'c' + ab'c$$

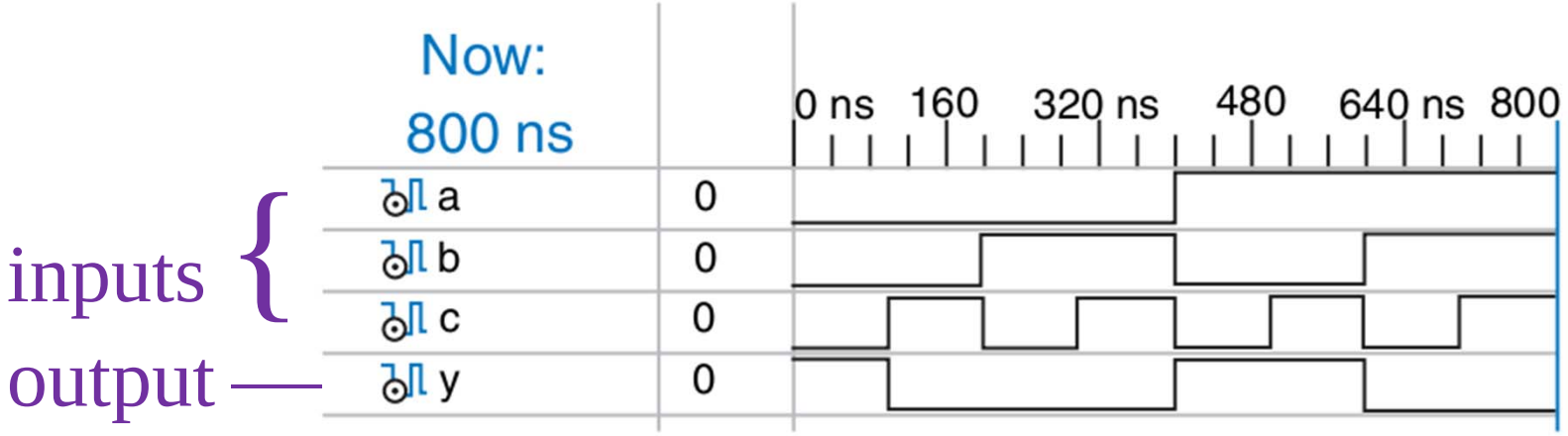


Figure 4.1 Simulation waveforms

Logic Synthesis

- Synthesis:
  - transforms HDL code into a *netlist* describing the hardware, e.g., the logic gates and the wires connecting them.
    - *netlist*: may be a text file, or may be drawn as a schematic
  - might perform optimizations to reduce the hardware.
  - E.g.: Synthesized ckt (Harris, p.170, Fig 4.2)

$$y = a'b'c' + ab'c' + ab'c$$
$$= b'c' + ab'$$

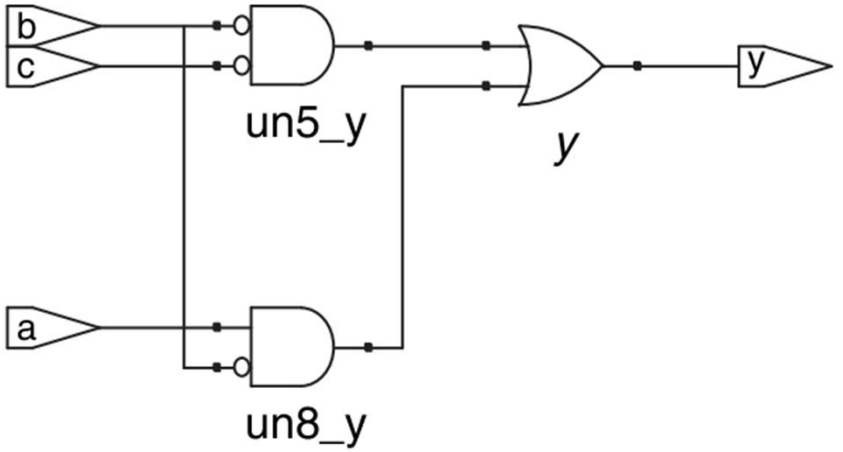
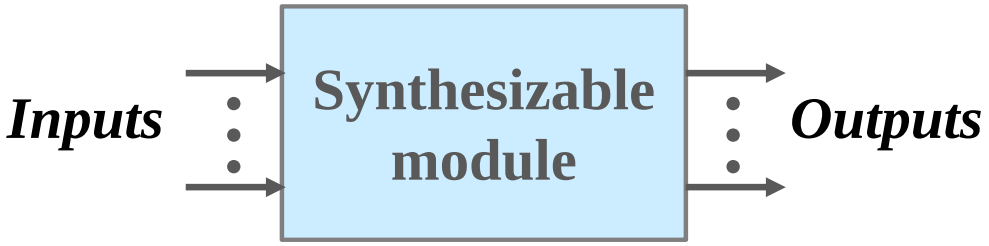


Figure 4.2 Synthesized circuit

Synthesizable Modules & Testbench

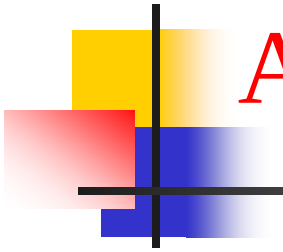
- **HDL code** may be divided into *synthesizable modules* and a *testbench*.
- Synthesizable modules:
  - describe the *hardware*



- Testbench:
  - contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies b/t expected and actual outputs.

* Testbench code is intended only for simulation and cannot be synthesized.

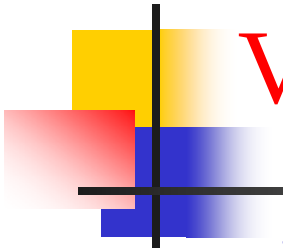
- E.g.: a command to print results on the screen during simulation



A. Module Declaration

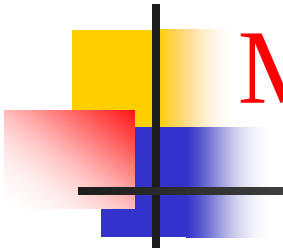
- HDL description forms:
  - a netlist of interconnected gates
  - Boolean logic equations
  - truth tables
  - an abstract behavioral model

- Gate-level modeling:
  - use instantiations of *predefined* and *user-defined primitive gates*



Verilog Model

- Verilog model:
  - is composed of text using keywords (100)
- Keywords:
  - are predefined *lowercase* identifiers that define the language constructs
  - E.g.s: **module**, **endmodule**, **input**, **output**, **wire**, **and**, **or**, **not**, …
- Comments:
  - // : single-line comment
  - /* …… */ : multiline comments
- Verilog is *case sensitive*.

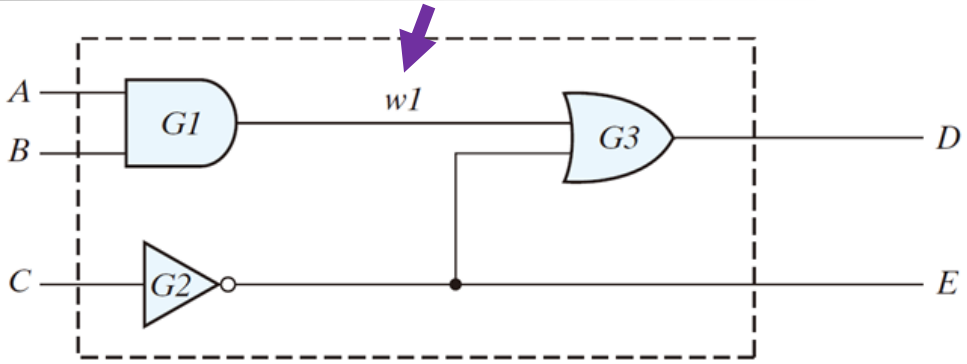


Module

- Module:
  - refers to the text enclosed by the keyword pair **module** … **endmodule**
  - is the fundamental descriptive unit
- HDL description of combinational logic:
  - a schematic connection of gates
  - a set of Boolean equations
  - a truth table
  - …

HDL Example 3.1: Gate-Level Model

- HDL Example 3.1: Gate-Level Model (p.127)
  - Combinational logic modeled w/ primitives
  - IEEE 1364-1995 Syntax

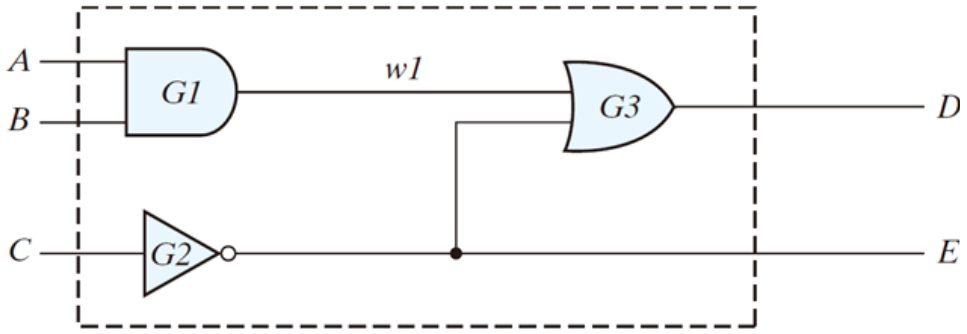


.v

```
module Simple_Circuit (A, B, C, D, E);
    output   D, E;
    input    A, B, C;
    wire     w1;

    and      G1(w1, A, B);   //Optional gate instance name
    not      G2(E, C);
    or       G3(D, w1, E);
endmodule
```

The output of a *primitive gate* is always listed first, followed by the inputs.

* Concurrence




```
module Simple_Circuit(A, B, C, D, E);
    output   D, E;
    input    A, B, C;
    wire     w1;

    and      G1(w1, A, B);
    not      G2(E, C);
    or       G3(D, w1, E);
endmodule
```

- Keywords in this example:
  - module: start the declaration (description) of a module
    - is followed by a name and a list of ports
    - The port list is the interface b/t the module and its environment.
    - The inputs and outputs of a module may be listed in any order.
  - endmodule: complete the declaration of a module
  - input, output: specify which of the ports are inputs/outputs
  - wire: declare internal connection
  - and, or, not: primitive gates
    - The output of a primitive gate is always listed first, followed by the inputs.

```
module Simple_Circuit(A, B, C, D, E);
    output   D, E;
    input    A, B, C;
    wire     w1;

    and      G1(w1, A, B);
    not      G2(E, C);
    or       G3(D, w1, E);
endmodule
```

- Identifiers:
  - are names given to modules, variables, and other elements of the language for reference in the design
  - are composed of alphanumeric characters and the underscore "_ ", but can not start w/ a number.
  - are *case sensitive*

  * Choose meaningful names for modules.
- Each statement must be terminated w/ a " **;** ", but not after endmodule.

Gate Delays

* Compiler directives start w/ the (`) back quote, or grave accent, symbol.

- Propagation delay: (in Verilog)
  - is specified in terms of *time units* and by the symbol # .
  - E.g.: and #(30) G1(w1, A, B);
- `timescale: a compiler directive
  - is made for association a time unit w/ physical time
  - is specified before the declaration of a module and applies to all numerical values of time in the code that follows
  - E.g.: `timescale 1ns/100ps

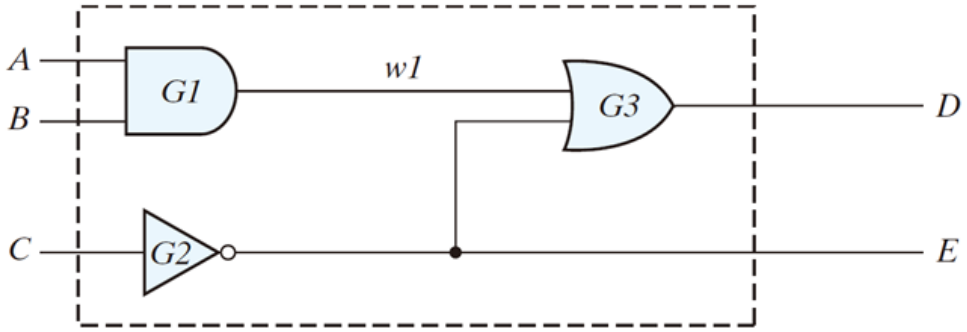
the unit of measurement for time delays

the precision for which the delays are rounded off

  - If no timescale is specified, a simulator may display dimensionless values or default to a certain time units, usually 1 ns (= 10^{-9} s).

HDL Example 3.2: Propagation Delays

- HDL Example 3.2:
  Gate-Level Model w/ Propagation Delays (p.130)



.V

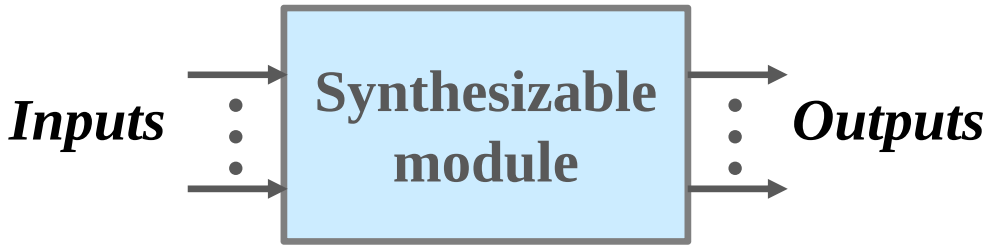
```
module Simple_Circuit_prop_delay (A, B, C, D, E);
    output   D, E;
    input    A, B, C;
    wire     w1;

    and      #(30) G1(w1, A, B);
    not      #(10) G2(E, C);
    or       #(20) G3(D, w1, E);
endmodule
```

Propagation delays of the primitive gates

– **#***n*: waits for *n* time units

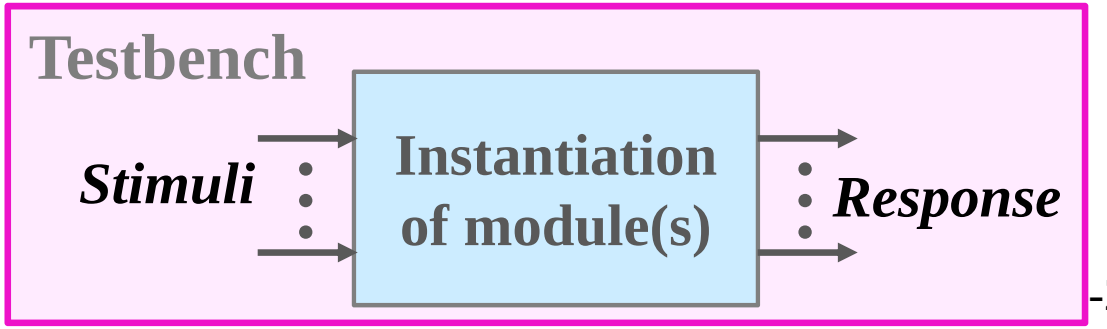
Testbench

- Recall: HDL code may be divided into *synthesizable modules* and a *testbench*.
- Synthesizable modules:
  - describe the hardware



- Testbench:
  - contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies b/t expected and actual outputs.

* Testbench code is intended only for simulation and cannot be synthesized.

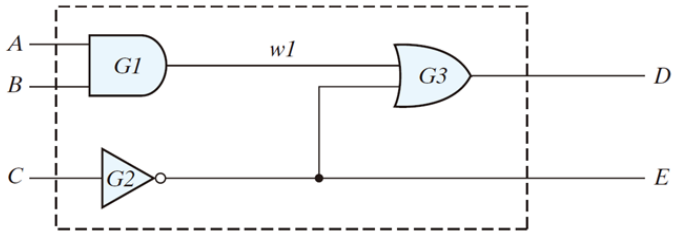


HDL Examples 3.3: Test Bench

* The test bench has no input or output ports ⇐ it does not interact w/ its environment.

- HDL Example 3.3 : Testbench (p.130)
  - Testbench of HDL Example 3.2

```
module Simple_Circuit_prop_delay (A, B, C, D, E);
    output   D, E;
    input    A, B, C;
    wire     w1;

    and      #(30) G1(w1, A, B);
    not      #(10) G2(E, C);
    or       #(20) G3(D, w1, E);
endmodule
```




```
module t_Simple_Circuit_prop_delay;
  wire D, E;
  reg  A, B, C;

  //instantiate device under test
  Simple_Circuit_prop_delay M1 (A, B, C, D, E);

  //apply inputs one at a time
  initial
    begin
      A = 1'b0; B = 1'b0; C = 1'b0;
      #100 A = 1'b1; B = 1'b1; C = 1'b1;
    end

  initial #200 $finish;
endmodule
```

The inputs to the ckt are declared w/ **reg** & the outputs are declared w/ **wire**.

an instantiation of the model to be verified

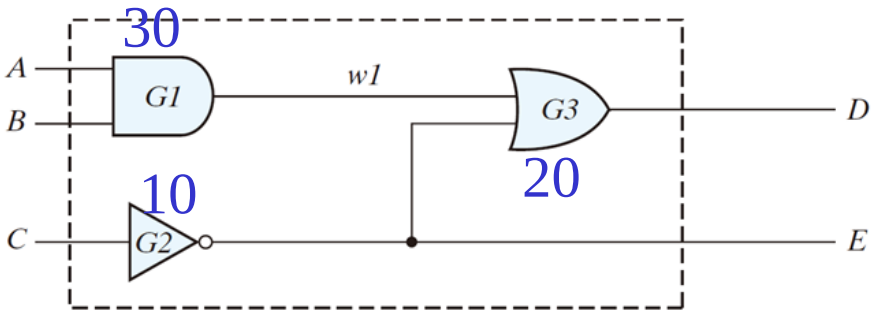
a signal generator

* The statements are executed ***in sequence***.

.V

```
module t_Simple_Circuit_prop_delay;
    wire D, E;
    reg  A, B, C;

    //instantiate device under test
    Simple_Circuit_prop_delay M1 (A, B, C, D, E);

    //apply inputs one at a time
    initial
        begin
            A = 1'b0; B = 1'b0; C = 1'b0;
            #100 A = 1'b1; B = 1'b1; C = 1'b1;
        end

     initial #200 $finish;
endmodule
```

- initial statement:
  - executes the statements in its body at the start of simulation
  - * should be used only in testbenches for simulation
- begin, end
- $finish: terminate the simulation
- reg: declares signals in initial statements
  - Variables of type reg retain their value until they are assigned a new value by an assignment statement




```
module Simple_Circuit_prop_delay(A, B, C, D, E);
    output   D, E;
    input    A, B, C;
    wire     w1;

    and      #(30) G1(w1, A, B);
    not      #(10) G2(E, C);
    or       #(20) G3(D, w1, E);
endmodule
```

```
module t_Simple_Circuit_prop_delay;
    wire D, E;
    reg  A, B, C;

    //instantiate device under test
    Simple_Circuit_prop_delay M1 (A, B, C, D, E);

    //apply inputs one at a time
    initial
        begin
            A = 1'b0; B = 1'b0; C = 1'b0;
            #100 A = 1'b1; B = 1'b1; C = 1'b1;
        end

    initial #200 $finish;
endmodule
```

- Simulation waveforms:

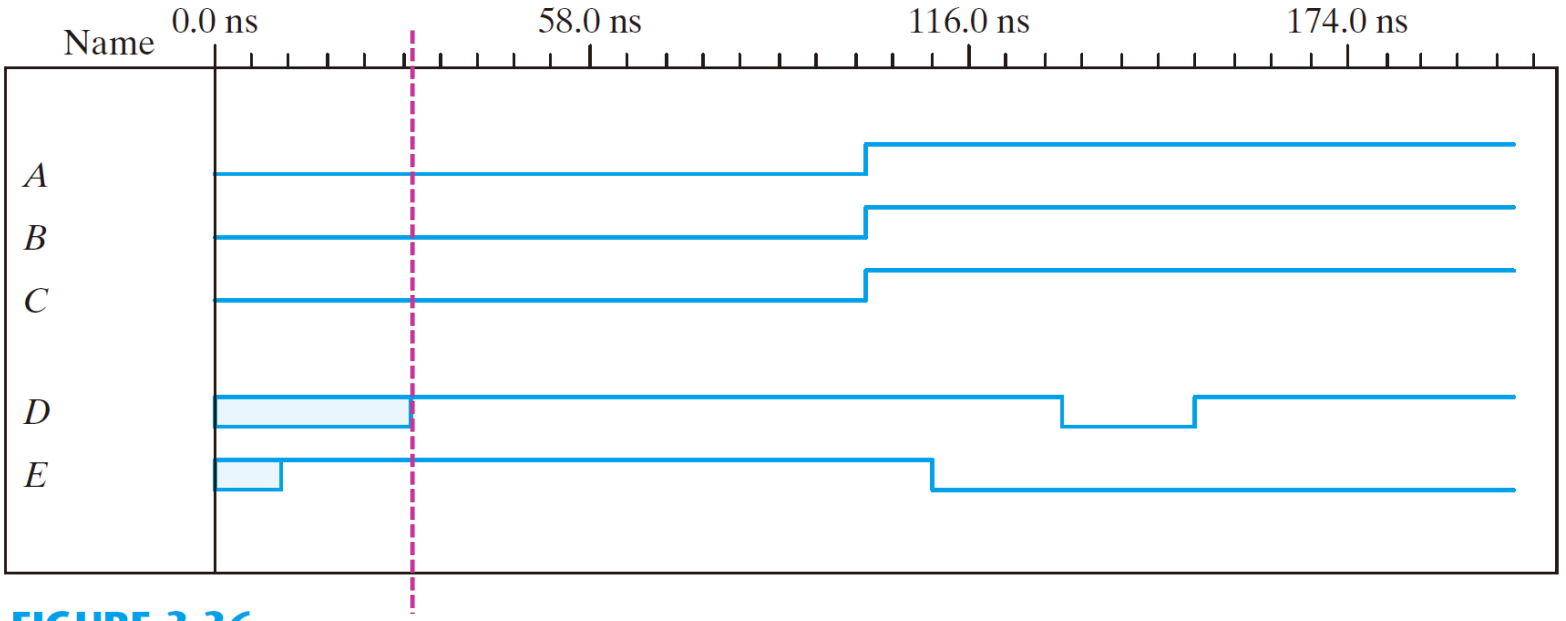


FIGURE 3.36
Simulation output of HDL Example 3.3

B. Boolean Expressions

- Boolean expressions of combinational logic:
  - are specified in Verilog w/ a ***continuous assignment statement*** consisting of the keyword **assign** followed by a Boolean expression
- Logic operators:

Operator	bitwise	logical
AND	&	&&
OR	|	||
NOT	~	!

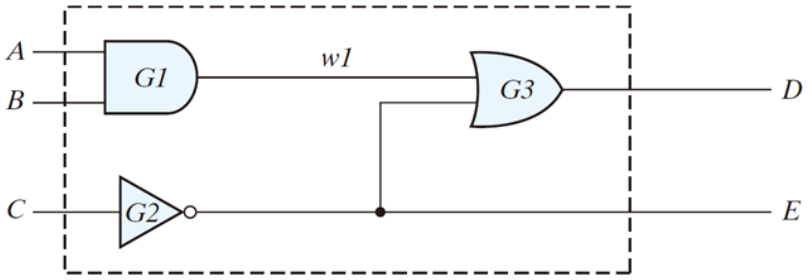
* If the operands are scalar, the results of bitwise and logical operators will be identical;

  if the operands are vectors, the results will not necessarily match.

- E.g.:

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

- E.g.s: p.HDL-52



HDL Example 3.4: Boolean Equations

- HDL Example 3.4: Combinational Logic Modeled w/ Boolean Equations (p.132)

Operator	bitwise	logical
AND	&	&&
OR	|	||
NOT	~	!

E = A + BC + B'D

F = B'C + BC'D'

.V

```
module Circuit_Boolean_CA (E, F, A, B, C, D);
   output    E, F;
   input     A, B, C, D;

   assign  E = A || (B && C) || ((!B) && D);
   assign  F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

– assign statement: describe *combinational* logic

* The simulator detects when the test bench changes a value of one or more of the inputs and updates the vales of the outputs.

C. User-Defined Primitives (Truth Tables)

- *system primitives*:
  - e.g.s: and, or, not, … (in Verilog)

- *user-defined primitives* (UDPs):
  - The user can create additional primitives by defining them in tabular form ⇒ truth table
  - are declared w/ the keyword pair

    primitive … **endprimitive**
  - A UDP can be instantiated in the construction of other modules, just as the system primitives are used.

General Rules for Defining a UDP

- General rules for defining a UDP:
  - It is declared w/ the keyword **primitive**, followed by a name and port list.
  - There can be only one output, and it must be listed first in the port list and declared w/ keyword **output**.
  - There can be any # of inputs. The order in which whey are listed in the **input** declaration must conform to the order in which they are given values in the table that follows.
  - The truth table is enclosed within the keywords **table** and **endtable**.
  - The values of the inputs are listed in order, ending w/ a colon (**:**). The output is always the last entry in a row and is followed by a semicolon (**;**).
  - The declaration of a UDP ends w/ the keyword **endprimitive**.

HDL Example 3.5: User-Defined Primitive

- HDL Example 3.5: User-Defined Primitive (p.133)

Define UPD_02467

$D(A, B, C)$

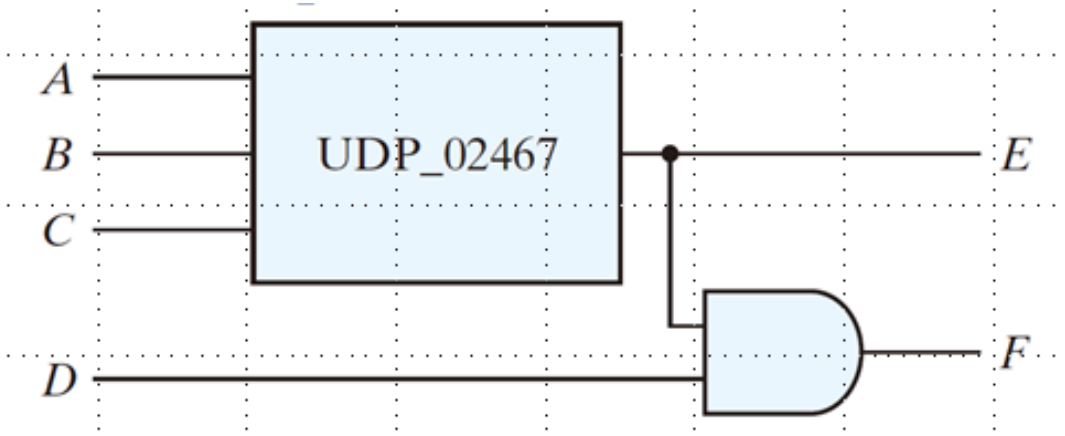
$= \Sigma m(0, 2, 4, 6, 7)$

.V

```
primitive UDP_02467 (D, A, B, C);
   output    D;
   input     A, B, C;

// Truth table for D = f(A, B, C) = Σm(0, 2, 4, 6, 7);
   table
//     A  B  C  :  D   // Column header comment
       0  0  0  :  1;
       0  0  1  :  0;
       0  1  0  :  1;
       0  1  1  :  0;
       1  0  0  :  1;
       1  0  1  :  0;
       1  1  0  :  1;
       1  1  1  :  1;
  endtable
endprimitive
```

HDL Example 3.5: (con't)

– Construct module w/ UPDs:



.v

```
// Verilog model: Circuit instantiation of Circuit_UDP_02467
module Circuit_with_UDP_02467 (e, f, a, b, c, d);
   output    e, f;
   input     a, b, c, d;

   UDP_02467  M0 (e, a, b, c);
   and  (f, e, d);        //Option gate instance name omitted
endmodule
```

Mano 4-12

HDL Models of Combinational Circuits

J.J. Shann

4-12 HDL Models of Combinational Ckts

* 1 module per file (.v) w/ the same name

- Module:
  - the basic building block for modeling hardware w/ the Verilog HDL
- Modeling styles of a module:
  - ***gate-level*** modeling: use instantiations of *predefined* and *user-defined primitive gates*
    - describes a ckt by specifying its gates and how they are connected w/ each other
  - ***dataflow*** modeling: use *continuous assignment statements* w/ the keyword **assign**
    - is used mostly for describing the Boolean equations of combinational logic
  - ***behavioral*** modeling: use *procedural assignment statements* w/ the keyword **always**
    - is used to describe combinational and sequential ckts at a higher level of abstraction

Verilog Model for Combinational Logic

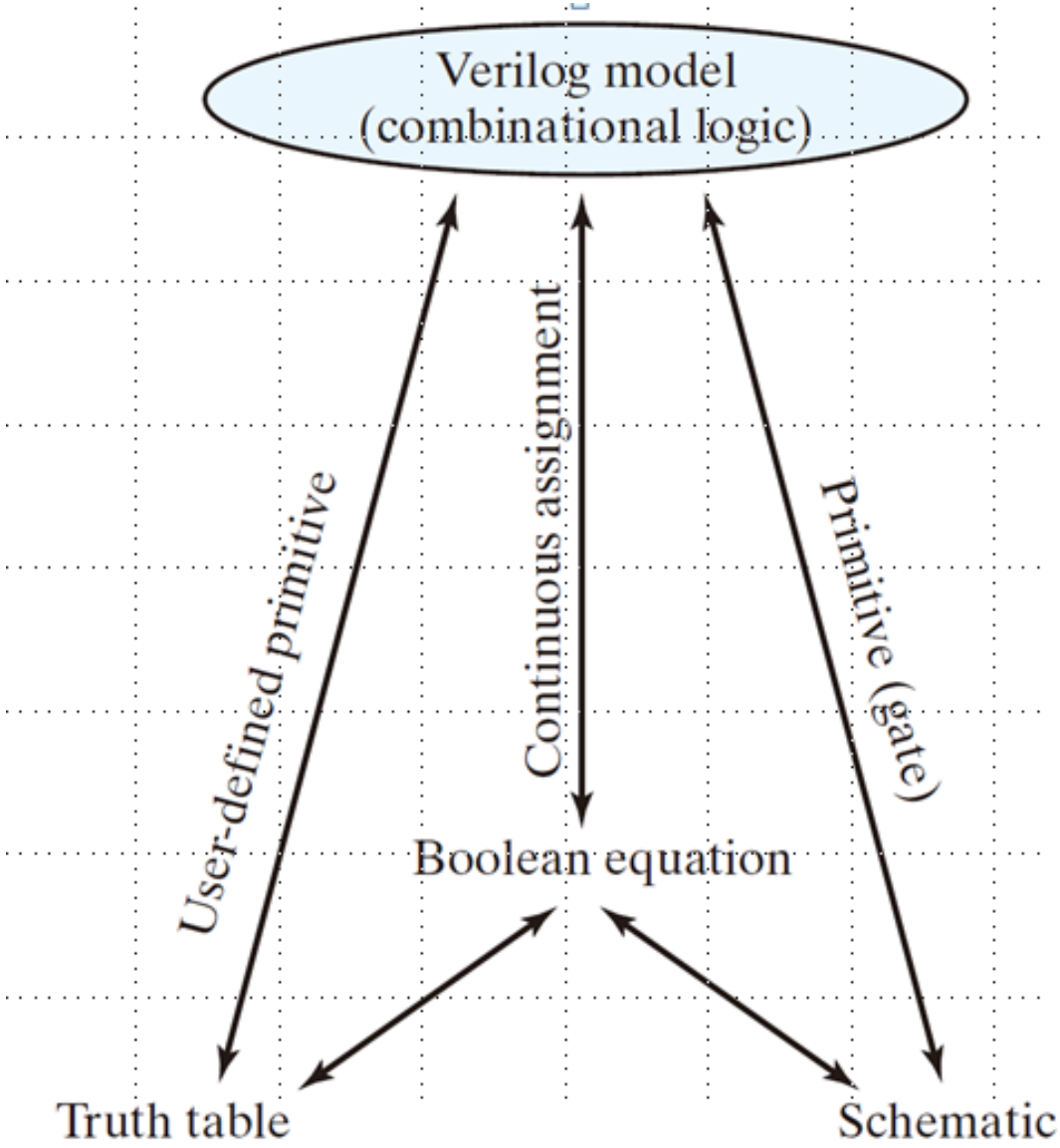


Figure 4.31

Relationship of Verilog constructs to truth tables, Boolean equations, and schematics

A. Gate-Level Modeling

* The output of a primitive is always listed first in the port list, followed by the inputs.

- Gate-level modeling:
  - Provides a texture description of a *schematic diagram*.
    - A ckt is specified by its *logic gates* and *their interconnections*.
- Predefined primitives of gates in Verilog: 12
  - 8 digital logic primitive gates: p.77, Fig 2.5

    and, **nand**, **or**, **nor**, **xor**, **nxor**, **not**, **buf**

    n-input 1-output primitives (can have any # of scalar inputs)

    1-input *n*-output primitives (can drive multiple output lines)
  - 4 three-state type primitive gates: p.HDL-44

    bufif1, **bufif0**, **notif1**, **notif0**

*** The logic of each gate is based on a 4-valued system:**

0, 1, *x* (*unknown*), *z* (*high impedance*)

4-Valued System

- **0**
- **1**
- ***x***: *unknown*
  - An unknown value is assigned during simulation when the *logic value of a signal is ambiguous* (can not be determined whether its value is 0 or 1).
- ***z***: *high impedance*
  - occurs at the *output of three-state gates that are not enabled* or if *a wire is inadvertently left unconnected*.
- E.g.: 4-valued logic truth tables (p.182, Table 4.9)

4-Valued Logic Truth Tables

- E.g.: 4-valued logic truth tables (p.182, Table 4.9)

Table 4.9
Truth Table for Predefined Primitive Gates

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

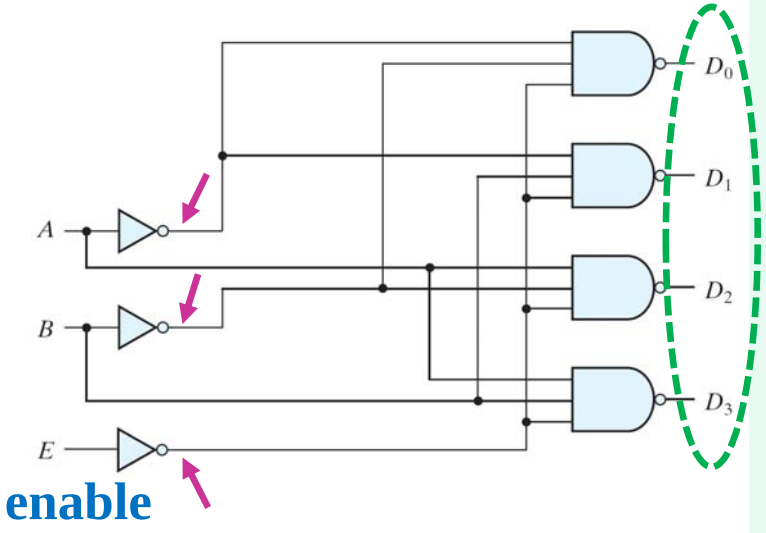
or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

not	input	output
	0	1
	1	0
	x	x
	z	x

HDL Example 4.1: 2-to-4 Line Decoder

- HDL Example 4.1: 2-to-4 Line Decoder
  - *gate-level* description of Fig 4.19




```
module decoder_2x4_gates (D, A, B, enable);
  output [0: 3]   D;
  input           A, B;
  input           enable;
  wire            A_not, B_not, enable_not;
  not
    G1  (A_not, A),
    G2  (B_not, B),
    G3  (enable_not, enable);
  nand
    G4  (D[0], A_not, B_not, enable_not),
    G5  (D[1], A_not, B, enable_not),
    G6  (D[2], A, B_not, enable_not),
    G7  (D[3], A, B, enable_not);
endmodule
```

The keyword not may be written only once for the three gates.

```
module decoder_2x4_gates (D, A, B, enable);
  output [0: 3]   D;
  input           A, B;
  input           enable;
  wire            A_not, B_not, enable_not;

  not
    G1  (A_not, A),
    G2  (B_not, B),
    G3  (enable_not, enable);
  nand
    G4  (D[0], A_not, B_not, enable_not),
    G5  (D[1], A_not, B, enable_not),
    G6  (D[2], A, B_not, enable_not),
    G7  (D[3], A, B, enable_not);
endmodule
```

- *vector*:
  - a multiple-bit width identifier
  - e.g.s:

    output [0: 3] D;

    wire [7: 0] SUM;
  - includes within "[ ]" 2 numbers separated w/ a " : "
    - The 1st number listed is always the MSB of the vector.
    - The individual bits are specified within [ ], D[2].
    - may address parts (contiguous bits) of vectors, SUM[2: 0].
- wire: for internal connections

* The output is always listed first in the port list of a primitive, followed by the inputs.

Design Methodologies of Digital Circuits

- 2 basic design methodologies: ***hierarchical***
  - ***bottom-up*** design: the building blocks are first identified and then combined to build the top-level block
    - E.g.:

      Half adder (HA): add two bits, 1 XOR gate + 1 AND gate

      $\Rightarrow$ Full adder (FA): add three bits, 2 HAs + 1 OR gate

      $\Rightarrow$ *n*-bit ripple-carry adder (RCA): add 2 *n*-bit numbers, *n* FAs
  - ***top-down*** design: the top-level block is defined and then the subblocks necessary to build the top-level block are identified
    - E.g.:

      n-bit ripple-carry adder (RCA) $\Rightarrow$ *n* full adders (FAs)

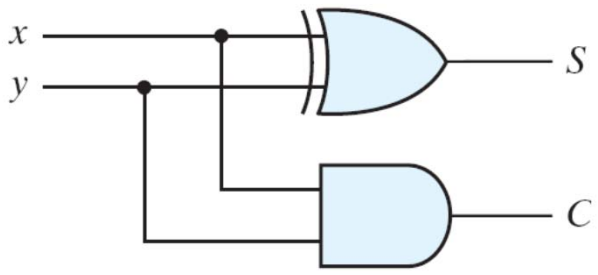
      Full adder $\Rightarrow$ 2 half adders (HAs) + 1 OR gate

      Half adder $\Rightarrow$ 1 XOR gate + 1 AND gate

HDL Example 4.2: Ripple-Carry Adder

- HDL Example 4.2: 4-bit Ripple-Carry Adder
  - *gate-level* description
  - *bottom-up* hierarchical description:

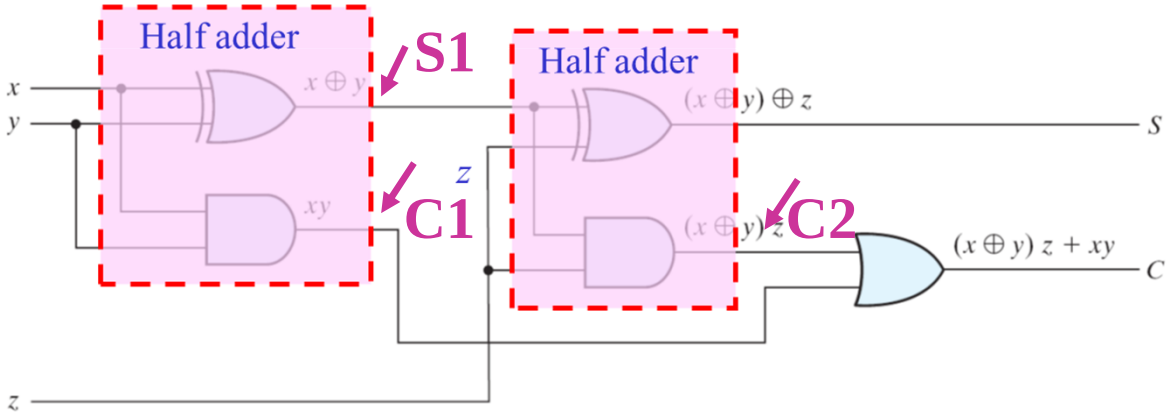
    HA $\rightarrow$ FA $\rightarrow$ RCA
  - HA: Fig 4.5(b)



(b) $S = x \oplus y$
$C = xy$

```
// Description of half adder

//module half_adder (S, C, x, y);
//  output    S, C;
//  input     x, y;

// Alternative Verilog 2005 syntax:

module half_adder (output S, C, input x, y);
 // Instantiate primitive gates
   xor (S, x, y);
   and (C, x, y);
endmodule
```

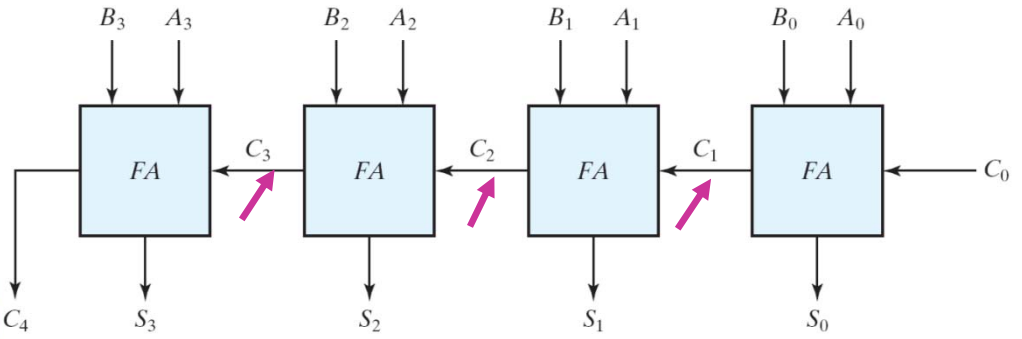
- FA: Fig 4.8



HA module

```
module half_adder (output S, C,
                   input x, y);
   xor (S, x, y);
   and (C, x, y);
endmodule
```

```
// Description of full adder
//module full_adder (S, C, x, y, z);
//  output    S, C;
//  input     x, y, z;

// alternative Verilog 2005 syntax:
module full_adder (output S, C, input x, y, z);
   wire        S1, C1, C2;

// Instantiate half adders
   half_adder HA1 (S1, C1, x, y);
   half_adder HA2 (S, C2, S1, z);
   or G1 (C, C2, C1);
endmodule
```

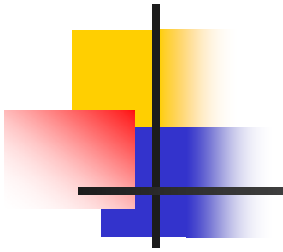


– 4-bit RCA: Fig 4.9

FA module

```
module full_adder (output S, C,
                   input x, y, z);
  wire        S1, C1, C2;

  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule
```

```
// Description of 4-bit ad
// module ripple_carry_4_bit_adder ( Sum, C4, A, B, C0);
//   output [3: 0]   Sum;
//   output          C4;
//   input   [3: 0]  A, B;
//   input           C0;

// Alternative Verilog 2005 syntax:
module ripple_carry_4_bit_adder ( output [3: 0] Sum,
   output C4,  input [3:0]  A, B, input C0);
   wire          C1, C2, C3;          // Intermediate carries
// Instantiate chain of full adders
   full_adder FA0 (Sum[0], C1, A[0], B[0], C0),
              FA1 (Sum[1], C2, A[1], B[1], C1),
              FA2 (Sum[2], C3, A[2], B[2], C2),
              FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```



- Module declarations **cannot** be nested.
  - A module definition (declaration) cannot be placed within another module declaration.

    ⇒ A module definition cannot be inserted into the text b/t the module and endmodule keywords of another module.
- Modules can be instantiated within other modules.
  - Creates a *hierarchical* decomposition of a design.

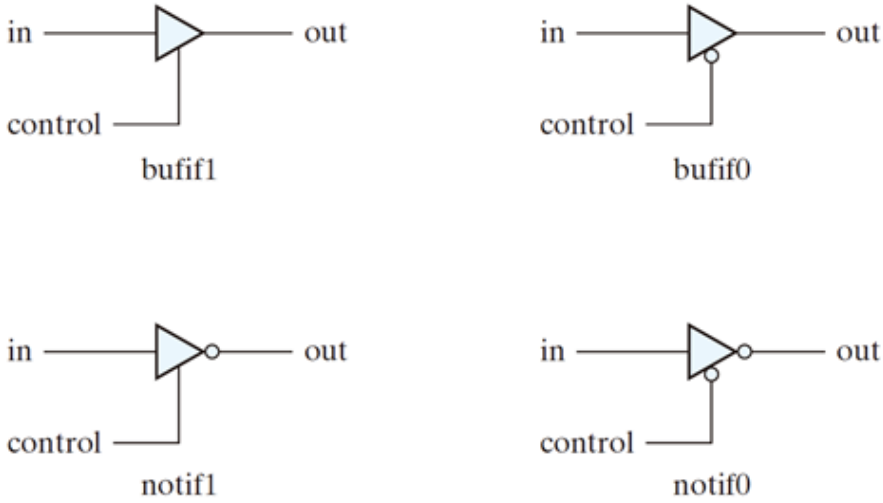
    ⇒ *structural* description

* One module per file (.v) & w/ the same name.

B. Three-State Gates

- Three-state gate:
  - has a control input that can place the gate into a *high-impedance* state (**z**).
- Types of three-state gates:
  - **bufif1**, **bufif0**,

    notif1, **notif0**



  - are instantiated w/ the statement

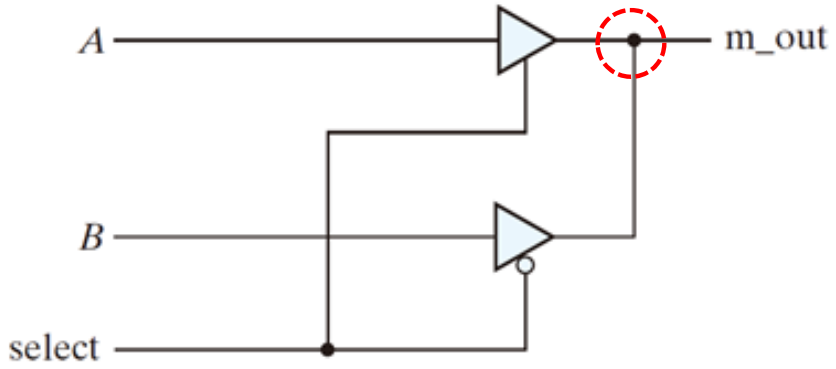
    gate name (output, input, control);
  - In simulation, the output can result in 0, 1, *x*, or *z*.
  - The outputs of 3-state gates can be connected together to form a common output line.

HDL Example: 2-to-1 Line Multiplexer

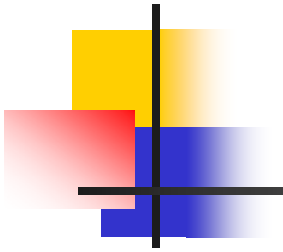
- HDL Example: 2-to-1 Line MUX w/ three-state buffers
  - p.179, Fig 4.30
  - **tri**: for tristate
    - indicate that the output has multiple drivers
  - Types of 3-state gates:

    bufif1, **bufif0**, **notif1**, **notif0**
  - Instantiate of a 3-state gate:

    ***gate name* (*output, input, control*);**




```
module mux_tri (m_out, A, B, select);
   output   m_out;
   input    A, B, select;
   tri      m_out;

   bufif1 (m_out, A, select);
   bufif0 (m_out, B, select);
endmodule
```



- *nets*: a class of data types
  - represent connections b/t hardware elements
  - e.g.s: **wire**, **wor**, **wand**, **tri**, **supply1**, **supply0**

wired-OR

wired-AND

power supply (1)

ground (0)

  - If an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a **wire**.

C. Dataflow Modeling

	Bitwise	Logical
AND	&	&&
OR	|	||
NOT	~	!

- Dataflow modeling:
  - uses a number of operators that act on binary operands to produce a binary result.
  - Commonly used Verilog HDL operators: p.187, Table 4.10

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

	Bitwise	Logical
AND	&	&&
OR	|	||
NOT	~	!

- *bitwise* operators: operate bit by bit on a pair of vector operands to produce a vector result
  - E.g.: ~(1010) is (0101)
- *logical* operators:
  - E.g.: !(1010) is 0

* If the operands are scalar, the results will be identical; if the operands are vectors, the result will not necessarily match.

- *concatenation* operator: **{}**, append multiple operands
- *conditional* operator: **?:**, acts like a multiplexer

condition **?** *true-expression* **:** *false-expression*

HDL operators (1/2)

Table 8.1
Verilog 2001 HDL Operators

Operator Type	Symbol	Operation Performed
Arithmetic	+	addition
	–	subtraction
	*	multiplication
	/	division
	%	modulus
	**	exponentiation
Bitwise or Reduction	~	negation (complement)
	&	AND
	|	OR
	^	exclusive-OR (XOR)

HDL operators (2/2)

Logical	!	negation
	&&	AND
	||	OR
Shift	>>	logical right shift
	<<	logical left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift
	{ , }	concatenation
Relational	>	greater than
	<	less than
	==	equality
	!=	inequality
	===	case equality
	!==	case inequality
	>=	greater than or equal
	<=	less than or equal

?

Operator Precedence

- Operator precedence from highest to lowest:

Precedence	Op	Meaning
Highest	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, NXOR
	|, ~|	OR, NOR
Lowest	?:	Conditional

Table 4.1, p.179 (Harris & Harris, 1st ed.)

* If the operands are scalar, the results will be identical; if the operands are vectors, the result will not necessarily match.

	Bitwise	Logical
AND	&	&&
OR	|	||
NOT	~	!

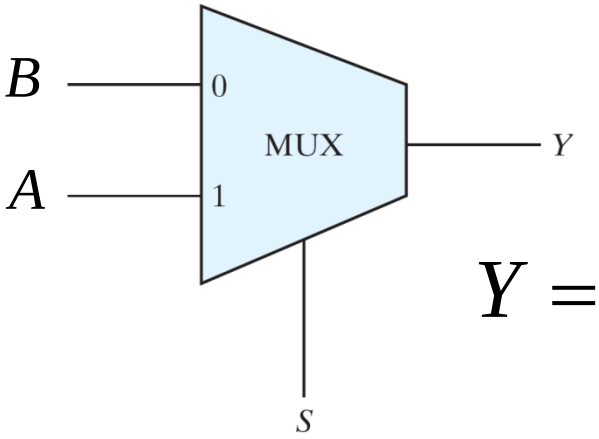
- Example: A =1010, B =0000
  - A has the Boolean value 1, B has Boolean value 0.
  - Results of other operation with these value:

```
A && B = 0          // Logical AND:          (1010) && (0000) = 0
A & B = 0000        // Bitwise AND:          (1010) & (1010) = (0000)
A || B = 1          // Logical OR:           (1010) || (0000) = 1
A | B = 1010        // Bitwise OR:           (1010) | (0000) = (1010)
!A = 0              // Logical negation      !(1010) = !(1) = 0
~A = 0101           // Bitwise negation      ~(1010) = (0101)
!B = 1              // Logical negation      !(0000) = !(0) = 1
~B = 1111           // Bitwise negation      ~(0000) = 1111
(A > B) = 1         // is greater than
(A == B) = 0        // identity (equality)
```

Dataflow Modeling of Comb. Logic

- Dataflow modeling of combinational logic:
  - Describes combinational ckts by their *function* rather than by their gate structure
  - uses *continuous assignments* and the keyword **assign**.
  - E.g.: a 2-to-1-line MUX w/ scalar data inputs *A* and *B*, select input *S*, and output *Y*

```
assign Y = (A && S) || (B && !S);
```

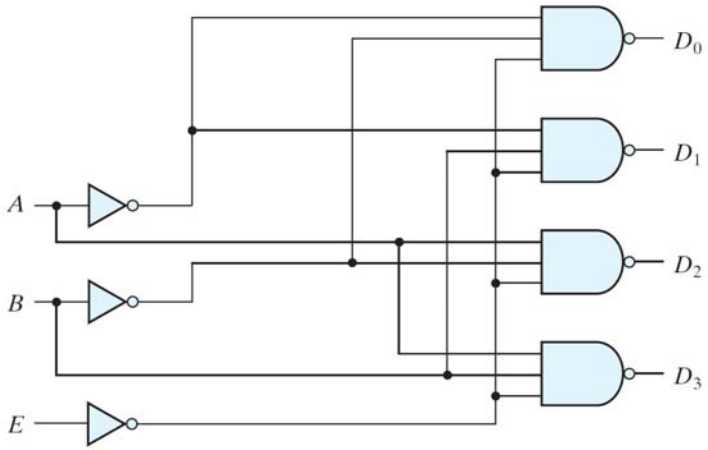


$$Y = SA + \overline{S}B$$

	Bitwise	Logical
AND	&	&&
OR	|	||
NOT	~	!

HDL Example 4.3: 2-to-4 Line Decoder (Dataflow)

- HDL Example 4.3: (Dataflow) 2-to-4 Line Decoder
  - *dataflow* description of Fig 4.19



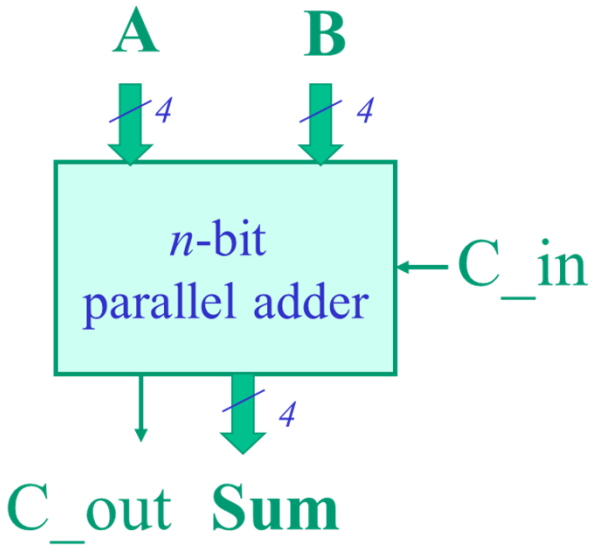
enable

	Bitwise	Logical
AND	&	&&
OR	|	||
NOT	~	!

```
//Verilog 2001, 2005 Syntax
module decoder_2x4_df (
   output [0: 3]   D,
   input           A, B,
   input           enable
);

   assign  D[0] = !((!A) && (!B) && (!enable)),
           D[1] = !((!A) && B & (!enable)),
           D[2] = !(A && (!B) && (!enable)),
           D[3] = !(A & B && (!enable));
endmodule
```

HDL Example 4.4: 4-bit Adder (Dataflow)

- HDL Example 4.4: (Dataflow) 4-bit adder
  - *dataflow* description of 4-bit adder



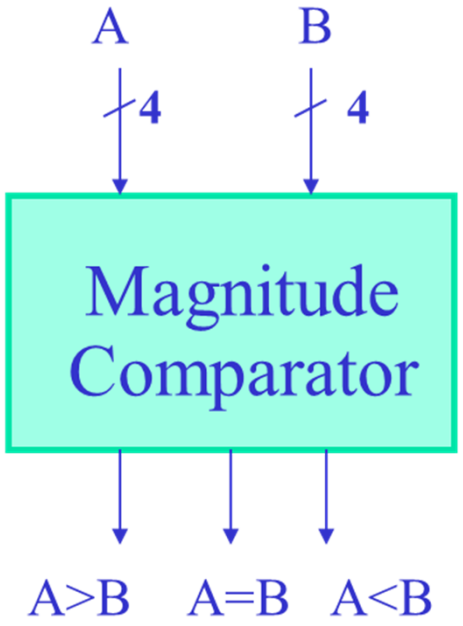
Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

```
// Verilog 2005 module port syntax
module binary_adder (
    output  [3:0]   Sum,
    output          C_out,
    input   [3: 0]  A, B,
    input           C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```

5-bit result of the addition operation

HDL Example 4.5: 4-bit Comparator (Dataflow)

- HDL Example 4.5: (Dataflow) 4-bit comparator

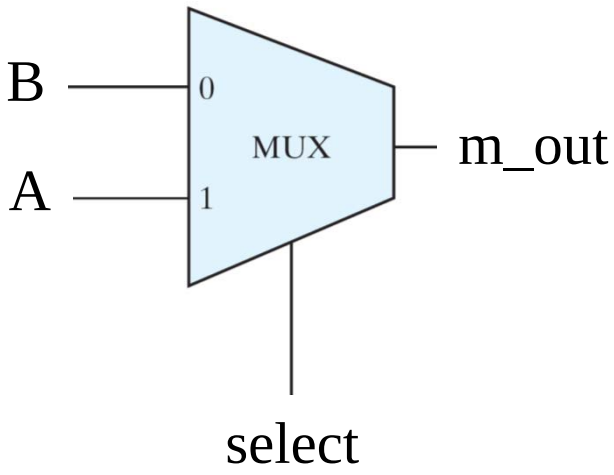


Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

```
// Verilog 2001, 2005 syntax
module mag_compare (
   output          A_lt_B, A_eq_B, A_gt_B,
   input   [3: 0]  A, B
);

   assign A_lt_B = (A < B);
   assign A_gt_B = (A > B);
   assign A_eq_B = (A == B);
endmodule
```

HDL Example 4.6: 2-to-1 MUX (Dataflow)

- HDL Example 4.6: (Dataflow) 2-to-1 MUX




```
// Verilog 2001, 2005 syntax
module mux_2x1_df(m_out, A, B, select);
  output      m_out;
  input       A, B;
  input       select;

  assign m_out = (select)? A : B;
endmodule
```

– *conditional* operator: **?:**

condition ? true-expression : false-expression

D. Behavioral Modeling

* **reg** data type:
  retains its value until a new value is assigned

- ***Behavioral*** modeling:
  - represents digital ckts at a *functional* and *algorithmic* level
  - is used mostly to describe *sequential* ckts, but can also be used to describe *combinational* ckts.
  - use keyword **always**, followed by an optional *event control expression* and a list of *procedural assignment statements*.

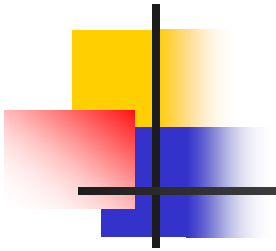
```
always @ (event control expression) begin
    //procedural assignment statements
end
```

specifies when the statements will execute

    - The procedural assignment statements inside the **always** block are executed every time there is a change in *any* of the variables listed after the **@** symbol.

keyword **or**

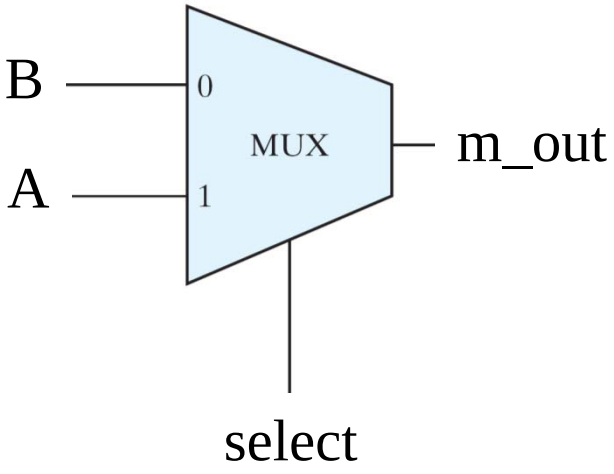
* The target output of the procedural assignment statement must be **reg** data type.



- **if-else** conditional statement
- **case** statement: **case** … **endcase**
  - a multiway conditional branch construct
  - The case items have an implied priority because the list is evaluated from top to bottom.

HDL Example 4.7: 2-to-1 MUX (Behavioral)

- HDL Example 4.7: (Behavioral) 2-to-1 Line MUX
  - *Behavioral* description of 2-to-1 line MUX




```
// Verilog 2001, 2005 syntax
module mux_2x1_beh(
   output reg m_out,
   input      A, B, select
);

   always @(A or B or select)     //@(A, B, select)
      if (select == 1) m_out = A; //(select)
      else m_out = B;
endmodule
```

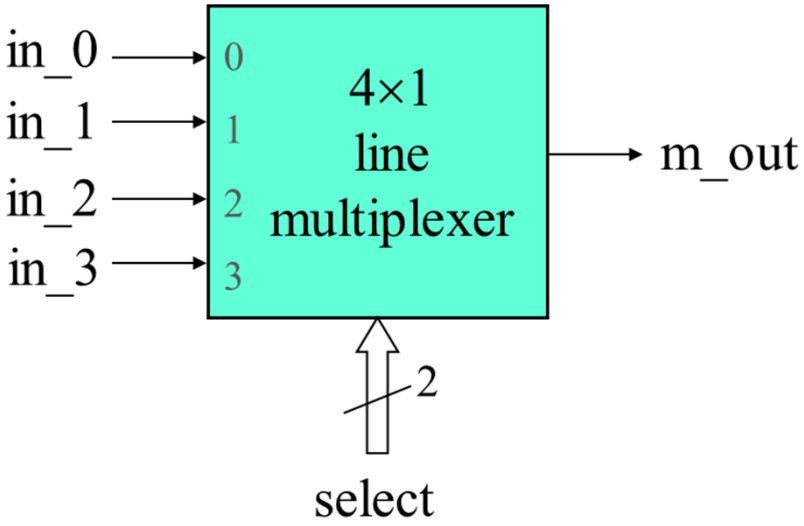
*** The target output of the procedural assignment statement inside always must be reg data type.**

*** Use *blocking assignments* (=) in always block for combinational logic.**

  - **if-else** conditional statement

HDL Example 4.8: 4-to-1 Line MUX (Behavioral)

- HDL Example 4.7: (Behavioral) 4-to-1 Line MUX
  - Behavioral description of 4-to-1 line MUX




```
// Verilog 2001, 2005 syntax
module mux_4x1_beh
( output  reg  m_out,
  input        in_0, in_1, in_2, in_3,
  input  [1:0] select
);

  always @(in_0, in_1, in_2, in_3, select)
     case (select)
       2'b00:   m_out = in_0;
       2'b01:   m_out = in_1;
       2'b10:   m_out = in_2;
       2'b11:   m_out = in_3;
     endcase
endmodule
```

case items, have implied priority (top to bottom)

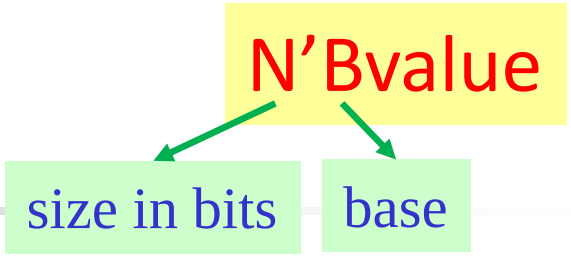
case Construct

- **case** statement: **case** … **endcase**
  - a multiway conditional branch construct
  - The case items have an implied priority because the list is evaluated from top to bottom.
- **default**: the last item in the list of case items, for unlisted case items
- 2 important variations of **case** construct:
  - **casex**: don't-cares any bits of the **case** expression or the **case** item that have logic value **x** or **z**
  - **casez**: don't-cares only the logic value **z**

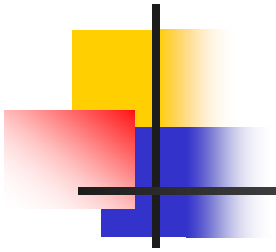
x: unknown

z: high-impendence

Numbers



- Numbers can be specified in *binary*, *octal*, *decimal*, or *hexadecimal* (bases 2, 8, 10, 16):
  - Underscores (_) in numbers are ignored and can be helpful in breaking long numbers into more readable chunks.
- Format for declaring constants: N'Bvalue
  - N: the size in bits, leading zeros are inserted to reach this size
    - optional, but better give the size explicitly.
    - If the size is not specified, the system assumes that it is the word length of the host simulator or at least 32 bits.
  - B: the letter indicating the base
    - 'b for binary, 'o for octal, 'd for decimal, and 'h for hexadecimal
    - If the base is omitted, it defaults to decimal.
  - value: gives the value



N'Bvalue

size in bits | base

- Examples: Harris, Table 4.3, p.180

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

* '0, '1: fill a bus w/ all 0s and all 1s

Summary

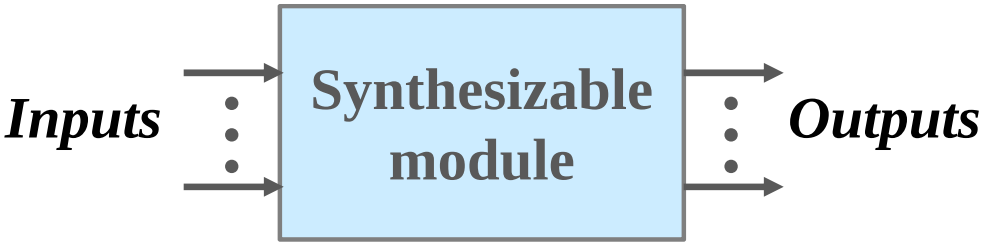
- Guidelines of modeling *combinational* logic:
  - Gate-level modeling
  - Dataflow modeling:
    - Use *continuous assignments* **assign** to model simple combinational logic.
  - Behavioral modeling:
    - Use **always @ (*)** and *blocking assignments* to model more complicated combinational logic where the **always** statement is helpful.
* Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

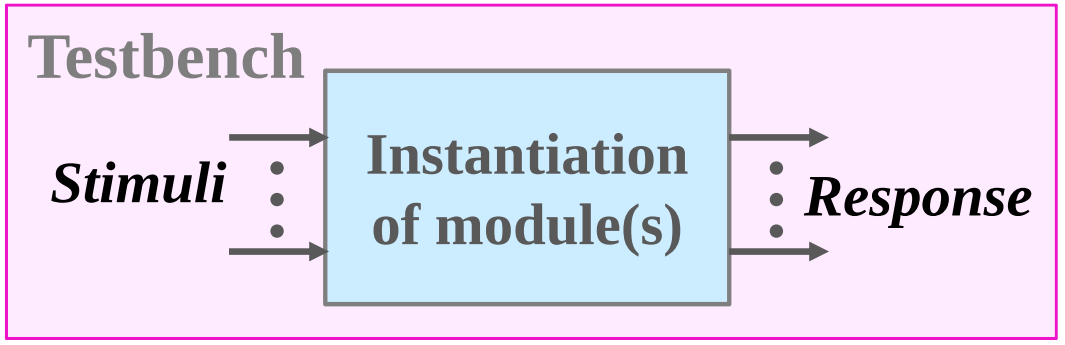
```
assign y = s ? d1 : d2;
```

```
always @ (*)
   begin
      p = a ^ b;  //blocking
      q = a & b;  //blocking
      s = p ^ cin;
      cout = q | (p & cin);
   end
```

E. Writing a Simple Test Bench

- Test bench:
  - is an HDL program used for describing and applying a *stimulus* to an HDL model of a ckt in order to test it and observe its *response* during *simulation*.
  - Write stimuli that will test a ckt thoroughly, exercising all of the operating features that are specified.
  - can be complex and lengthy and may take longer to develop than the design that is tested.
  - The results of a test are only as good as the test bench that is used to test a ckt.





(a) Providing Input Stimulus

- Statements used by a test bench to provide a stimulus to the ckt being tested:
  - **always** statement: executes repeatedly in a loop
    - specify how the associated statement is to execute (the *event control expression*)

```
always @ (event control expression) begin
    //procedural assignment statements
end
```

  - **initial** statement: executes only once
    - starting from simulation time 0, and may continue w/ any operations that are delayed by a given number of time units, as specified by the symbol **#**.

- Example:

stimulus	A	B
(t = 0)	0	0
(t = 10)	1	0
(t = 30)	0	1

```
Initial
  begin
            A = 0; B = 0;
    #10     A = 1;
    #20     A = 0; B = 1;
  end
```

- Example:

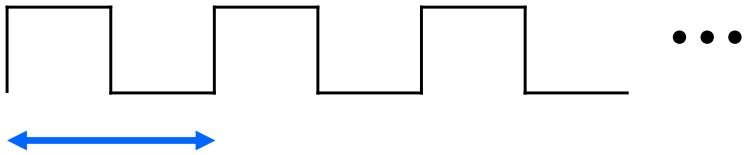
stimulus	D
(t = 0)	000
(t = 10)	001
…	
(t = 70)	111

```
Initial
  begin
    D = 3'b000;
    repeat (7)
    #10  D = D + 3'b001;
  end
```

  - **repeat**: specifies a looping statement

- Example:

generate periodic clock pulse (period = 10 time units)




```
always
   begin
        clock = 1;  #5;  clock = 0;  #5;
   end
```

(b) Displaying Output Response to the Stimulus

- The response of the stimulus generated by the initial and always blocks will appear:
  - in text format as standard output &
  - as waveforms (timing diagrams) in simulators having graphical output capability
- Numerical outputs are displayed by using Verilog *system tasks*.
- Verilog system tasks:
  - are built-in system functions that are recognized by keywords that begin with the symbol **$**.

System Tasks for Display

- **$display**: display a one-time value of variables or strings with an end-of-line return
- **$write**: same as **$display**, but w/o going to next line
- **$monitor**: display variables whenever a value changes during a simulation run
- **$time**: display the simulation time
- **$finish**: terminate the simulation

- Syntax for **$display**, **$write**, and **$monitor**:

  (next page)

- Syntax for **$display**, **$write**, and **$monitor**:

  Task-name (format specification, argumentlist);

  - Format specification:
    - uses the symbol **%** to specify the radix of the numbers that are displayed : **%b**, **%d**, **%h**, **%o** (%B, %D, %H, %O)
    - man have a string enclosed in quotes (**"....."**)
    - No commas in the format specification.
  - Argument list:
    - has commas b/t the variables.
  - E.g.s:

    $display ("%d %b %b", C, A, B);

    $display ("time = %0d A = %b B = %b", $time, A, B);

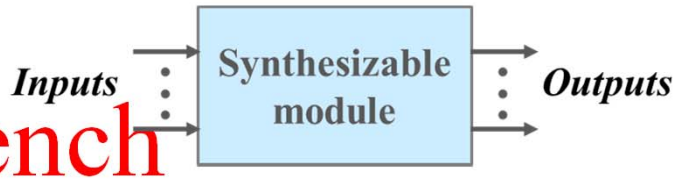
    format specification argument list

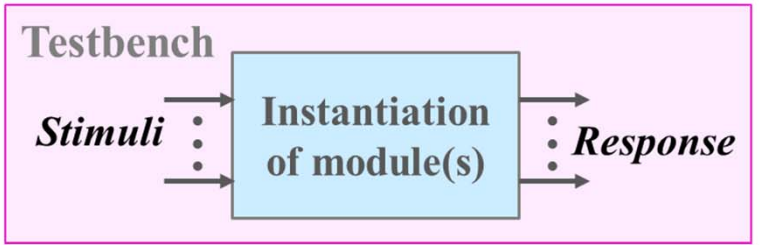
%d: w/ the leading spaces

%0d: w/o the leading spaces

* In displaying time values, use the format %0d instead of %d. (Time is calculated as a 32-bit number.)

(c) Testbench

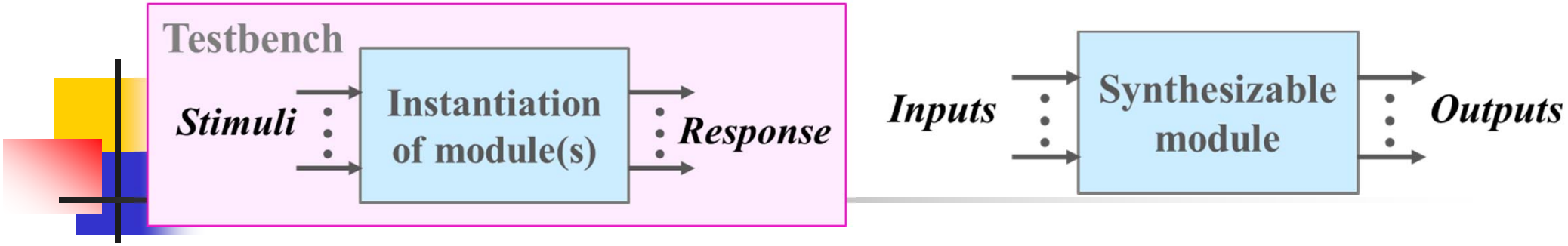




- Form of a test bench:

```
module test_module_name;      //no port list
    //Declare local reg and wire identifiers.
    //Instantiate the design module under test.
    //Specify a stopwatch, using $finish to terminate the simulation.
    //Generate stimulus, using initial and always statements.
    //Display the output response (text or graphics, or both).
endmodule
```

  - has no inputs or outputs.
  - The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type.
  - The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type.



- Interaction b/t stimulus and design modules:

Stimulus module (testbench)

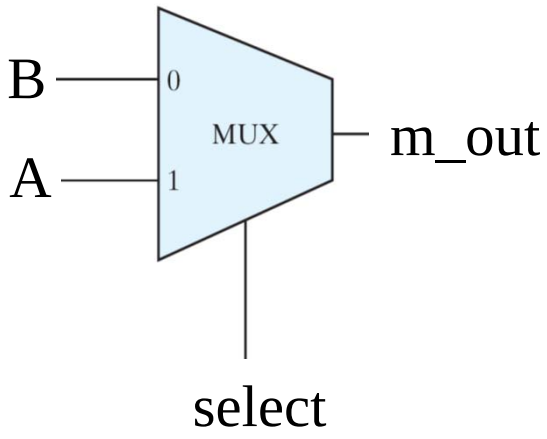
```
module t_circuit;
  reg t_A, t_B;
  wire t_C;
  parameter stop_time = 1000 ;

  circuit M ( t_C, t_A, t_B );

// Stimulus generators for
// t_A and t_B go here
  initial # stop_time $finish;
endmodule
```

Design module

```
module circuit ( C, A, B )

  input        A, B;

  output       C;

// Description goes here
endmodule
```

HDL Example 4.9: Test Bench (2-to-1 MUX)

- HDL Example 4.9: Test bench w/ stimulus for mux_2x1_df

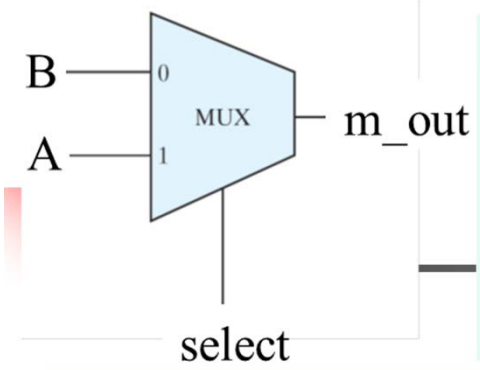



```
module mux_2x1_df(m_out, A, B, select);
   output   m_out;
   input    A, B;
   input    select;

   assign m_out = (select)? A : B;
endmodule
```

```
module  t_mux_2x1_df;
   wire      t_m_out;
   reg       t_A, t_B;
   reg       t_select;
   parameter  stop_time = 50;

   // Instantiation of circuit to be tested
   mux_2x1_df  M1 (t_m_out, t_A, t_B, t_select);

   initial # stop_time $finish;

   // Stimulus generator
   initial begin
            t_select = 1; t_A = 0; t_B = 1;
      #10   t_A = 1; t_B = 0;
      #10   t_select = 0;
      #10   t_A = 0; t_B = 1;
   end
   ...
```

parameter: a keyword, defines constant

$finish: a system task, terminates the simulation




```
module mux_2x1_df(m_out, A, B, select);
   output   m_out;
   input    A, B;
   input    select;

   assign m_out = (select)? A : B;
endmodule
```

Simulation log:

```
Select = 1  A = 0  B = 1  OUT = 0  time = 0
Select = 1  A = 1  B = 0  OUT = 1  time = 10
Select = 0  A = 1  B = 0  OUT = 0  time = 20
Select = 0  A = 0  B = 1  OUT = 1  time = 30
```

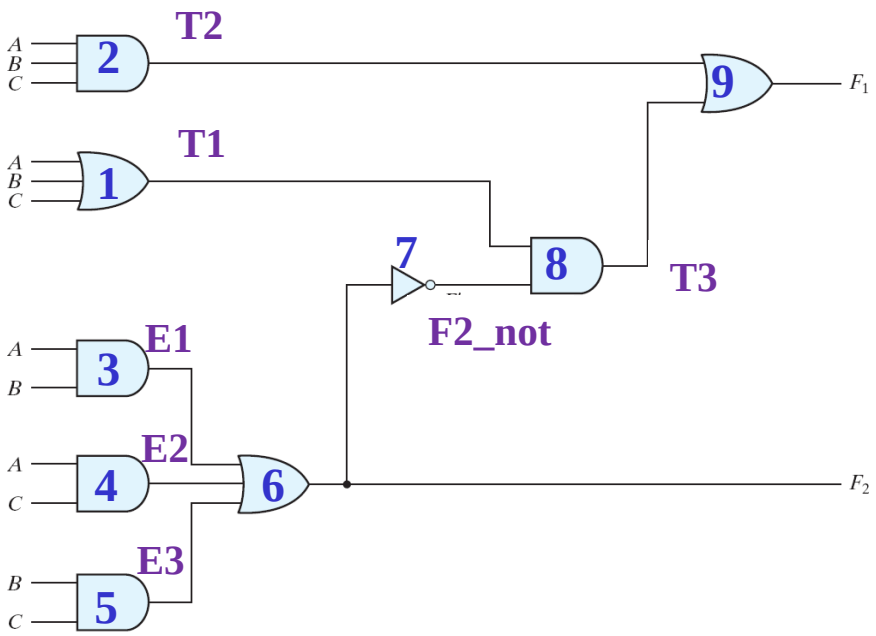
```
module  t_mux_2x1_df;
   //Declaration of local identifiers
   …
   // Instantiation of circuit to be tested
   …
   // Stimulus generator
   …

   //Response monitor
   initial begin
      $display ("     time   Select  A    B    m_out");
      $monitor ($time, "   %b     %b    %b    %b", t_select, t_A, t_B, t_m_out);
      //$monitor ("time=", $time, "select = %b A = %b B = %b m_out = %b",
      t_select, t_A, t_B, t_m_out);
   end
endmodule
```

$display: a system task, print in the simulator window

$monitor: a system task, displays the output caused by the given stimulus

HDL Example 4.10: Design Module & Test Bench (1/2)

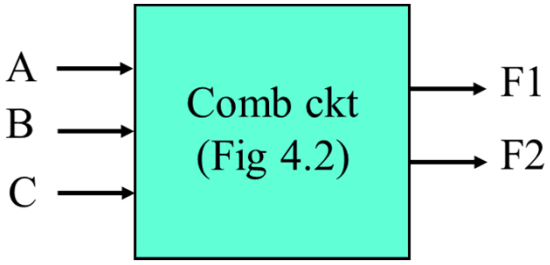
- HDL Example 4.10: Design Module & Test bench of Fig 4.2
  - *gate-level* description of ckt of Fig 4.2




```
module Circuit_of_Fig_4_2 (
   output  F1, F2,
   input   A, B, C);
   wire    T1, T2, T3, F2_not, E1, E2, E3;

   or   G1 (T1, A, B, C);
   and  G2 (T2, A, B, C);
   and  G3 (E1, A, B);
   and  G4 (E2, A, C);
   and  G5 (E3, B, C);
   or   G6 (F2, E1, E2, E3);
   not  G7 (F2_not, F2);
   and  G8 (T3, T1, F2_not);
   or   G9 (F1, T2, T3);
endmodule
```

HDL Example 4.10: Design Modul Bench (2/2)

- Stimulus to analyze the ckt of Fig 4.2




```
module Circuit_of_Fig_4_2 (
  output  F1, F2,
  input   A, B, C);
  wire    T1, T2, T3, F2_not, E1, E2, E3;

  or   G1 (T1, A, B, C);
  and  G2 (T2, A, B, C);
  and  G3 (E1, A, B);
  and  G4 (E2, A, C);
  and  G5 (E3, B, C);
  or   G6 (F2, E1, E2, E3);
  not  G7 (F2_not, F2);
  and  G8 (T3, T1, F2_not);
  or   G9 (F1, T2, T3);
endmodule
```

```
module t_Circuit_of_Fig_4_2;
  reg   [2: 0] D;
  wire F1, F2;
  parameter stop_time = 100;

  Circuit_of_Fig_4_2 M1 (F1, F2, D[2], D[1], D[0]);

  initial  # stop_time $finish;

  initial  begin              // Stimulus generator
     D = 3'b000;
     repeat (7) #10 D = D + 1'b1;
  end

  initial  begin              //Response monitor
     $display ("A     B     C     F1   F2");
    $monitor ("%b     %b     %b     %b     %b", D[2],
     D[1], D[0], F1, F2);
  end
endmodule
```

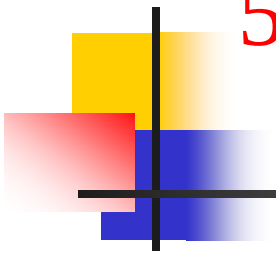
Simulation log:

```
ABC = 000  F1 = 0  F2 = 0
ABC = 001  F1 = 1  F2 = 0
ABC = 010  F1 = 1  F2 = 0
ABC = 011  F1 = 0  F2 = 1
ABC = 100  F1 = 1  F2 = 0
ABC = 101  F1 = 0  F2 = 1
ABC = 110  F1 = 0  F2 = 1
ABC = 111  F1 = 1  F2 = 1
```

Mano 5-6

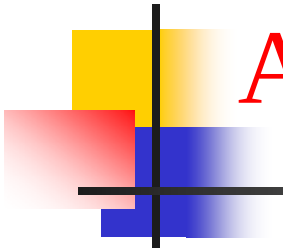
Synthesizable HDL Models of Sequential Circuits

J.J. Shann



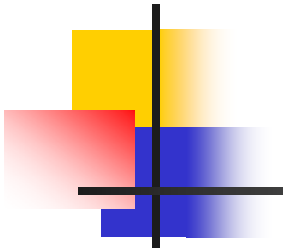
5-6 Synthesizable HDL Models of Sequential Circuits

- Behavioral Modeling of Sequential Circuit
- HDL Models of Flip-Flops and Latches
- State Diagram-Based HDL Models
- Structural Description of Clocked Sequential Circuits



A. Behavioral Modeling

- Behavioral models:
  - are abstract representations of the functionality of digital hardware.
  - describe how a ckt behaves, but don’t specify the internal details of the ckt.



- Two kinds of abstract behaviors in the Verilog HDL:
  - ***single-pass behavior*** : declared by the keyword **initial**, for prescribe stimulus signals in a ***test bench***
    - specifies a single statement or a block statement (by **begin … end** or **fork … join** keyword pair)
    - expires after the associated statement executes.
    - * **Never use single-pass behavior to model the behavior of a ckt!**
  - ***cyclic behavior***: declared by the keyword **always**
    - executes and reexecutes indefinitely, until the simulation is stopped.
  - A module may contain an arbitrary number of **initial** or **always** behavioral statements.
    - They execute concurrently w.r.t. each other, starting at time 0, and may interact through common variables.

initial statement

- **initial**: single-pass behavior, used in ***test bench***
  - E.g.: free-running clock

```
Initial
   begin
      clock = 1'b0;
      repeat (30)
         #10 clock = ~clock;
   end
```

```
Initial
   begin
      clock = 1'b0;
   end

Initial #300 $finish;

always #10 clock = ~clock;
```

* The 3 behavioral statements can be written in any order.

```
Initial
   begin
      clock = 1'b0;
      forever #10 clock = ~clock;
   end
```

an indefinite loop

always statement

- The general form of **always** statement:

```
always @ (event control expression) begin
    //Procedural assignment statements that execute
      when the condition is met
end
```

  - The variables in the left-hand side of the procedural statements must be declared as the **reg** data type.
  - *event control expression*: *sensitivity list*
    i. ***level sensitive events*** (in comb ckts and in latches)
       - E.g.: **always** @(A **or** B **or** C) //@(A, B, C)

    ii. ***edge sensitive events*** (in flip-flops): **posedge**, **negedge**
       - E.g.: **always** @(**posedge** clock **or negedge** reset) //Verilog 1995
         always @(**posedge** clock, **negedge** reset) //Verilog 2001

Procedural vs. Continuous Assignment

- Continuous assignment: **assign**
  - The updating of a continuous assignment is triggered whenever an event occurs in any variable included on the right-hand side of its expression.
- Procedural assignment:
  - is an assignment of a logic value to a variable within an **initial** or **always** statement.
  - A procedural assignment is made only when an assignment statement is executed and assigns value to it within a behavioral statement.
  - A variable having type **reg** remains unchanged until a procedural assignment is made to give it a new value.

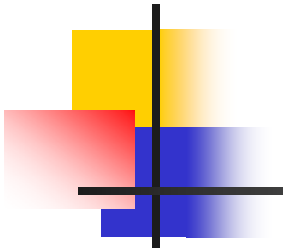
Procedural Assignments

- Two kinds of procedural assignments:
  - ***blocking*** assignment: **=** (for ***combinational logic***)
    - Blocking assignment statements are *executed sequentially* in the order they are listed in a block of statements.
    - E.g.: B = A;

      C = B + 1; //C = A + 1
  - ***nonblocking*** assingment: **<=**

    (for ***sync seq ckt*** and *latched behavioral*)
    - Nonblocking assignment statements are *executed concurrently* by evaluating the set of expressions on the right-hand side of the list of statements.
    - They do not make assignments to their left-hand sides until all of the expressions are evaluated.
    - E.g.: B <= A;

      C <= B + 1; //C will contain the original value of B, plus 1.



- General rule: (p.236)
  - Blocking assignment, **=** : Use blocking assignments when *sequential ordering* is imperative and in *cyclic behavior* that is *level sensitive* (i.e., in **combinational logic**).
  - Nonblocking assignment, **<=** : Use nonblocking assignments when modeling *concurrent execution* (e.g., *edge-sensitive behavior* such as **synchronous**, concurrent register transfers) and when modeling *latched behavior*.

* Prevent conditions that lead synthesis tools astray and create mismatches b/t the behavior of a model and the behavior of physical hardware produced by a synthesis tool.

Example

- Example: Full Adder

$p = a \oplus b$ … carry propagate function

$q = a \cdot b$ … carry generate function

$s = a \oplus b \oplus cin = p \oplus cin$

$cout = a \cdot b + (a \oplus b) \cdot cin = q + p \cdot cin$

@(a, b, cin)

```
always@ (*)
   begin
      p = a ^ b;  //blocking
      q = a & b;
      s = p ^ cin;
      cout = q | (p & cin);
   end
```

```
always@ (*)
   begin
      p <= a ^ b;  //nonblocking
      q <= a & b;
      s <= p ^ cin;
      cout <= q | (p & cin);
   end
```

@(a, b, cin, p, q)

✓

* For simulation & synthesis tools

Guidelines of Blocking and Nonblocking Assignments

- Use *continuous assignments* **assign** to model simple ***combinational*** logic.
- Use **always @ (*)** and *blocking assignments* to model more complicated ***combinational*** logic where the **always** statement is helpful.
- Use **always @ (posedge clk)** and *nonblocking assignments* to model synchronous ***sequential*** logic.
- Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

```
assign y = s ? d1 : d2;
```

```
always @ (*)
   begin
      p = a ^ b;  //blocking
      q = a & b;  //blocking
      s = p ^ cin;
      cout = q | (p & cin);
   end
```

```
always @ (posedge clk)
   begin
      n1 <= d;  //nonblocking
      q <= n1;  //nonblocking
   end
```

B. HDL Models of Flip-Flops and Latches

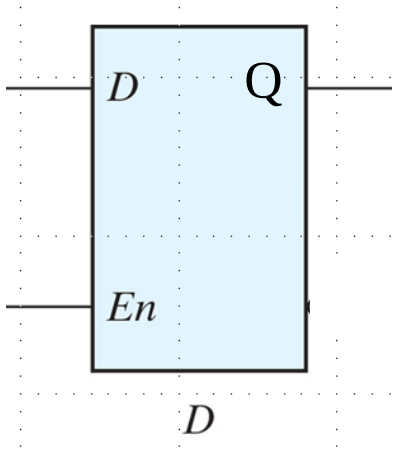
- Latch w/ control input:
  - responds to a change in data input w/ a change in the output as long as the enable input is asserted, i.e., in the active level.

    $\Rightarrow$ The latch is controlled by the "level" of its control input.
- Flip-Flop:
  - responses only to a "transition" of a triggering input called the "clock."
  - Positive-edge trigger: $0 \rightarrow 1$

    Negative-edge trigger: $1 \rightarrow 0$

HDL Example 5.1: D-Latch

- HDL Example 5.1: D-Latch



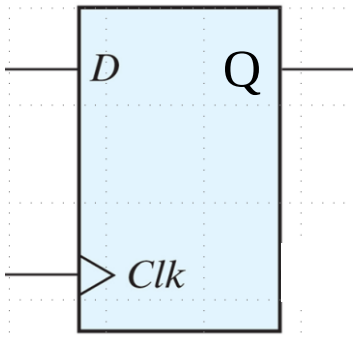

```
// Description of D latch (transparent latch), Fig. 5-6
//module D_latch (Q, D, enable);
//    output Q;
//    input    D, enable;
//    reg       Q;

//Verilog 2001, 2005

module D_latch (output reg Q, input D, enable);

    always @ (enable, D)          //(enable or D)
        if (enable) Q <= D;       //(enable == 1)
endmodule
```

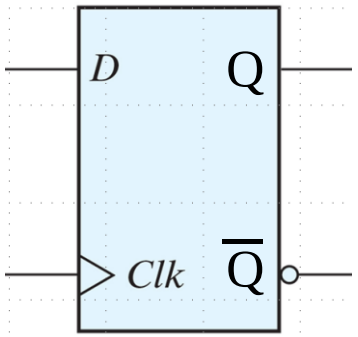
HDL Example 5.2: D-Type Flip-Flop

- HDL Example 5.2(a): D-Type Flip-Flop w/o Reset

D Flip-Flop

D	Q(t + 1)
0	0
1	1






```
module D_flip_flop (Q, D, Clk);
   output    Q;
   input     D, Clk;
   reg       Q;

   always @ (posedge Clk)
      Q <= D;
endmodule
```

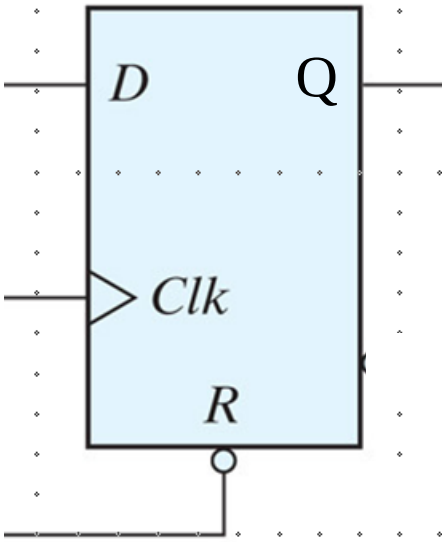
```
module D_flip_flop_b (Q, Q_b, D, Clk);
   output    Q, Q_b;
   input     D, Clk;
   reg       Q;

   assign    Q_b = ~Q;
   always @ (posedge Clk)
      Q <= D;
endmodule
```

D flip-flop w/o async reset

```
module D_flip_flop (Q, D, Clk);
   output   Q;
   input    D, Clk;
   reg      Q;

   always @ (posedge Clk)
      Q <= D;
endmodule
```

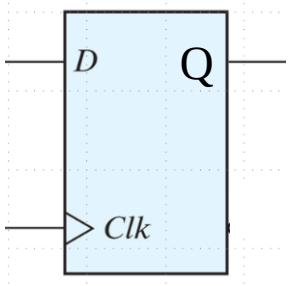
- HDL Example 5.2(b):

D Flip-Flop w/ Active-Low ***Asynchronous*** Reset

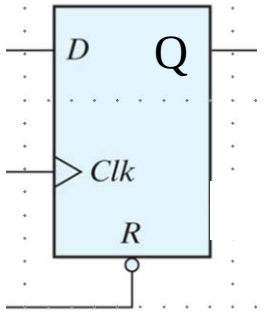



```
// Description of D flip-flop
// with active-low asynchronous reset

module D_flip_flop_AR (Q, D, Clk, rst);
   output   Q;
   input    D, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;    //if (rst == 0)
      else Q <= D;
endmodule
```

- Testbench of Example 5.2(a)(b):




```
module D_flip_flop (Q, D, Clk);
   output    Q;
   input     D, Clk;
   reg       Q;

   always @ (posedge Clk)
      Q <= D;
endmodule
```




```
module D_flip_flop_AR (Q, D, Clk, rst);
   output   Q;
   input    D, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;    //if (rst == 0)
      else Q <= D;
endmodule
```

```
module t_D_flip_flops;
   wire  Q, Q_AR;
   reg   D, Clk, rst;

   D_flip_flop M0 (Q, D, Clk);
   D_flip_flop_AR M1 (Q_AR, D, Clk, rst);

   initial #100 $finish;
   initial begin Clk = 0; forever #5 Clk = ~Clk; end
   initial fork
      D = 1;
      rst = 1;
      #20 D = 0;
      #40 D = 1;
      #50 D = 0;
      #60 D = 1;
      #70 D = 0;
      #42 rst = 0;
      #72 rst = 1;
   join
endmodule
```

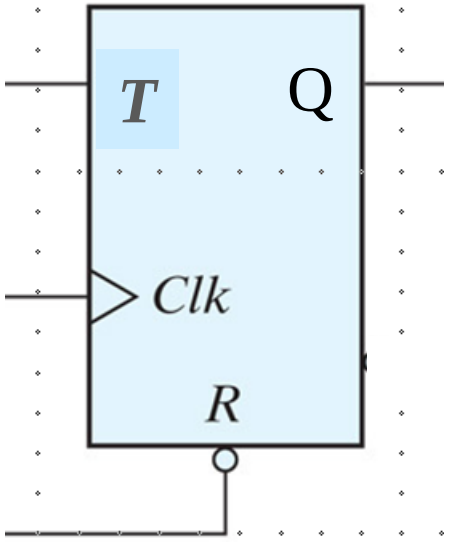
*** Statements within the fork ... join block execute in parallel.**

HDL Example 5.3 (a)(b): T Flip-Flop Models

```
module D_flip_flop_AR (Q, D, Clk, rst);
   output   Q;
   input    D, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;     //if (rst == 0)
      else Q <= D;
endmodule
```

D f-f w/ active-low async reset

- HDL Example 5.3(a): T flip-flop w/ Asynchronous Reset



T	Q(t + 1)
0	Q(t)
1	Q′(t)

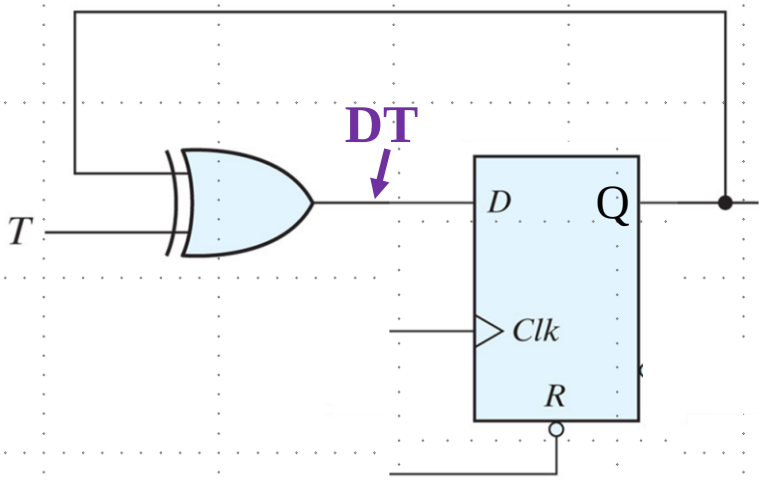
```
// Description of Toggle (T) flip-flop
module Toggle_flip_flop_1 (Q, T, Clk, rst);
   output   Q;
   input    T, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;
      else if (T) Q <= !Q;
endmodule
```

```
module D_flip_flop_AR (Q, D, Clk, rst);
   output   Q;
   input    D, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;     //if (rst == 0)
      else Q <= D;
endmodule
```

D f-f w/ active-low async reset

- HDL Example 5.3(b):

  T flip-flop from D flip-flop and gates



$DT = T \oplus Q$

```
//T flip-flop from D flip-flop and gates
module T_flip_flop_2 (Q, T, Clk, rst);
   output   Q;
   input    T, Clk, rst;
   wire     DT;

   assign  DT = T ^ Q;
   D_flip_flop_AR M0 (Q, DT, Clk, rst);
endmodule
```

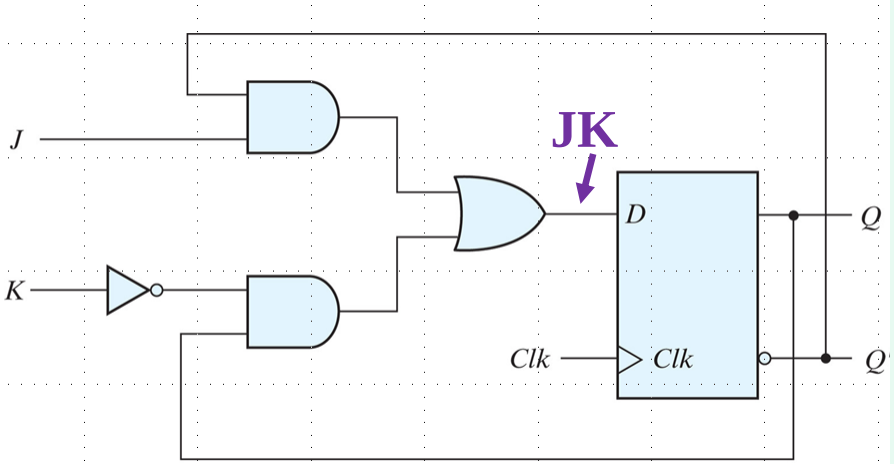
HDL Example 5.3 (c): JK Flip-Flop Models

- HDL Example 5.3(c):

  JK flip-flop from D flip-flop and gates

```
module D_flip_flop_AR (Q, D, Clk, rst);
   output   Q;
   input    D, Clk, rst;
   reg      Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;     //if (rst == 0)
      else Q <= D;
endmodule
```

D f-f w/ active-low async reset




```
//JK flip-flop from D flip-flop and gates
module JK_flip_flop (Q, J, K, Clk, rst);
   output   Q;
   input    J, K, Clk, rst;
   wire     JK;

   assign  JK = (J & ~Q) | (~K & Q);
   D_flip_flop_AR JK1 (Q, JK, Clk, rst);
endmodule
```

$JK = JQ' + K'Q$

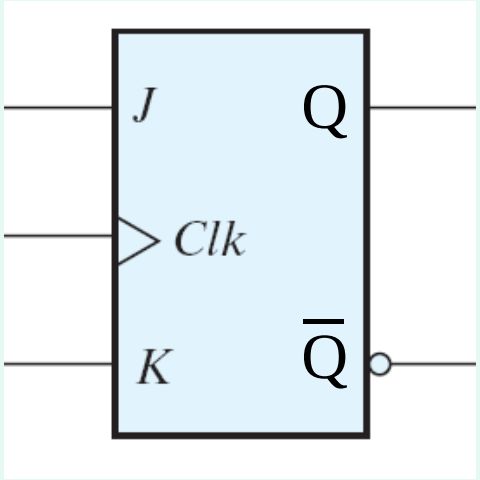
HDL Example 5.4: JK Flip-Flop

- HDL Example 5.4: JK flip-flop
  - *function description* of JK flip-flop

JK Flip-Flop

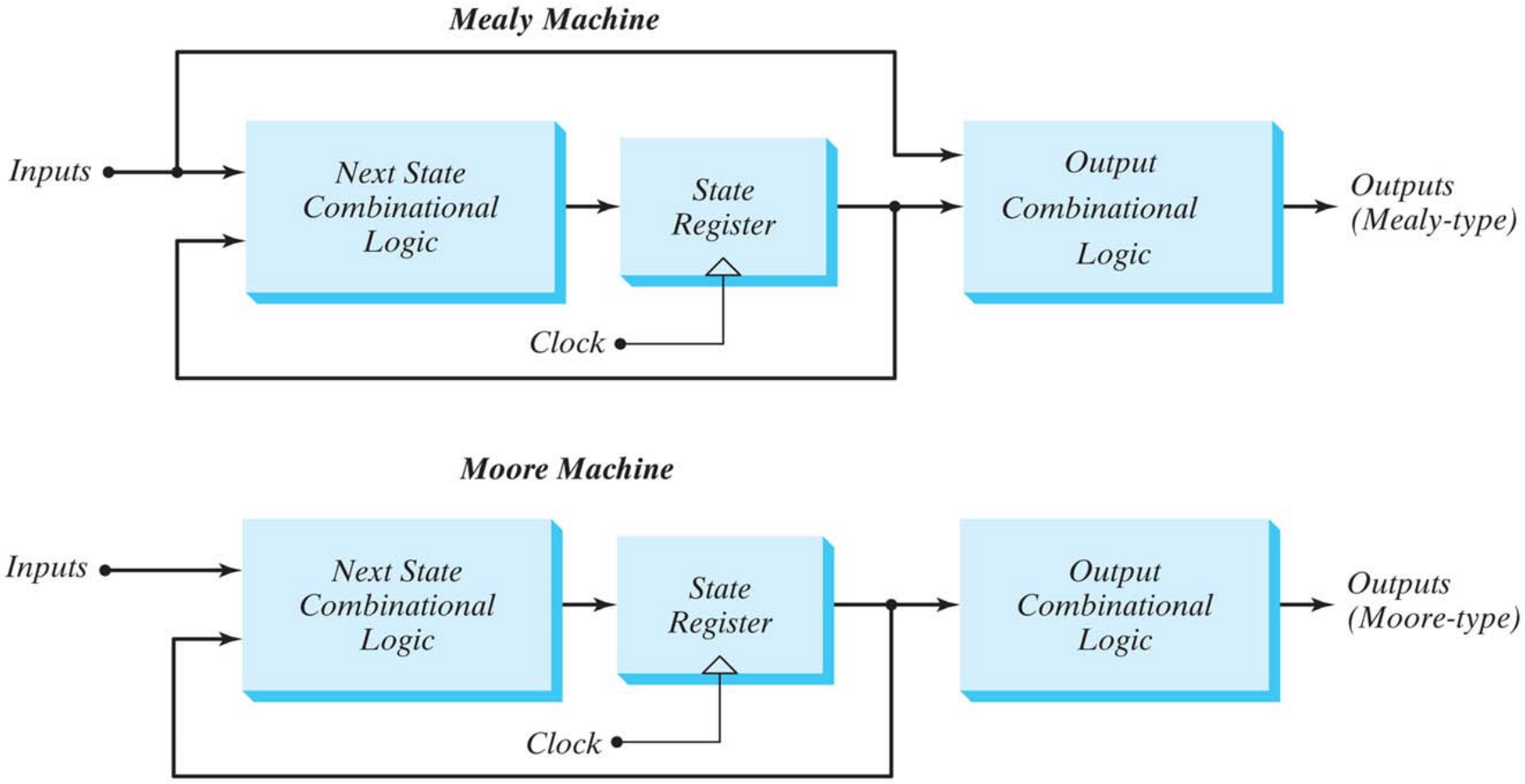
J	K	Q(t + 1)
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

```
//Function description of JK flip-flop
module JK_flip_flop_2 (Q, Q_not, J, K, Clk);
   output      Q, Q_not;
   input       J, K, Clk;
   reg         Q;

   assign  Q_not = ~Q;

   always @ (posedge Clk)
   case ({J, K})
      2'b00:   Q <= Q;
      2'b01:   Q <= 1'b0;
      2'b10:   Q <= 1'b1;
      2'b11:   Q <= ~Q;
   endcase
endmodule
```



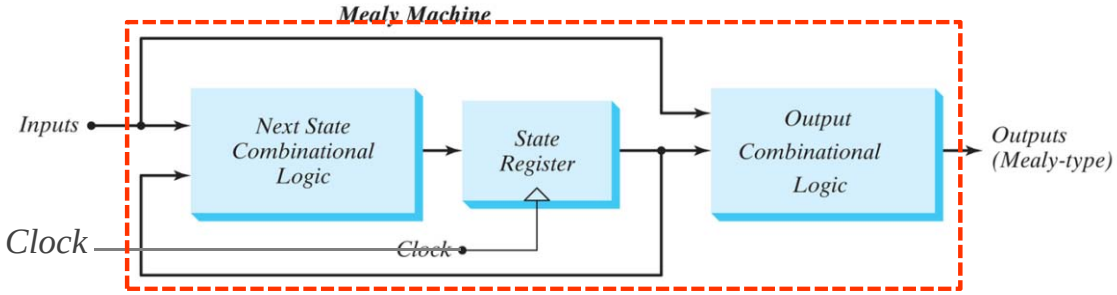
C. State Diagram-Based HDL Models of Synchronous Sequential Circuits

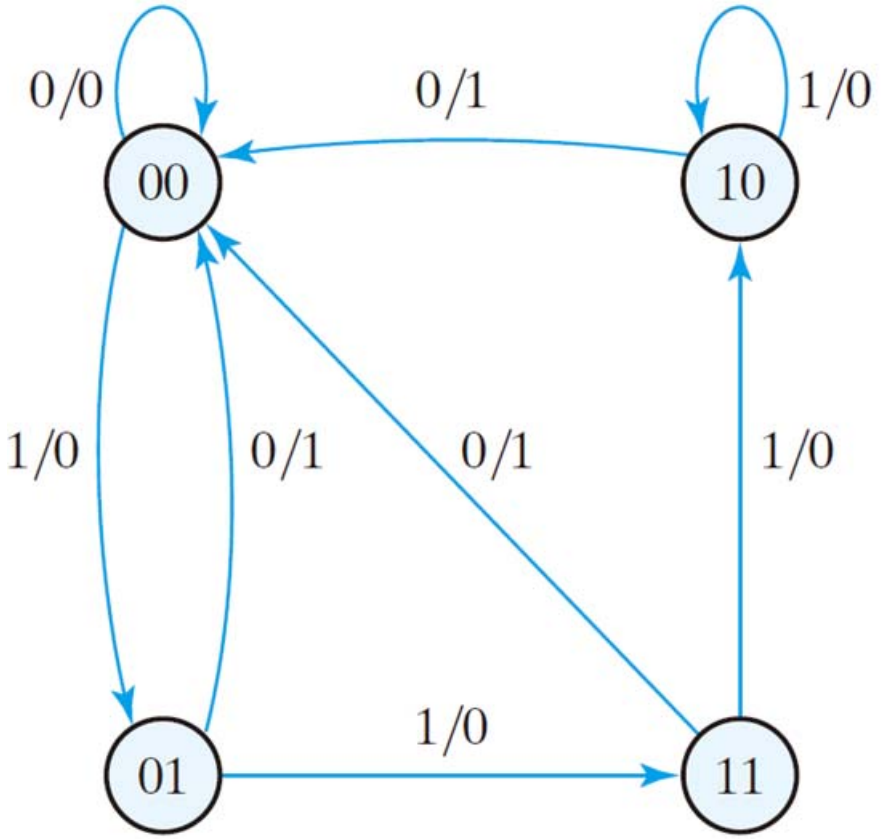
- Mealy HDL model
- Moore HDL model



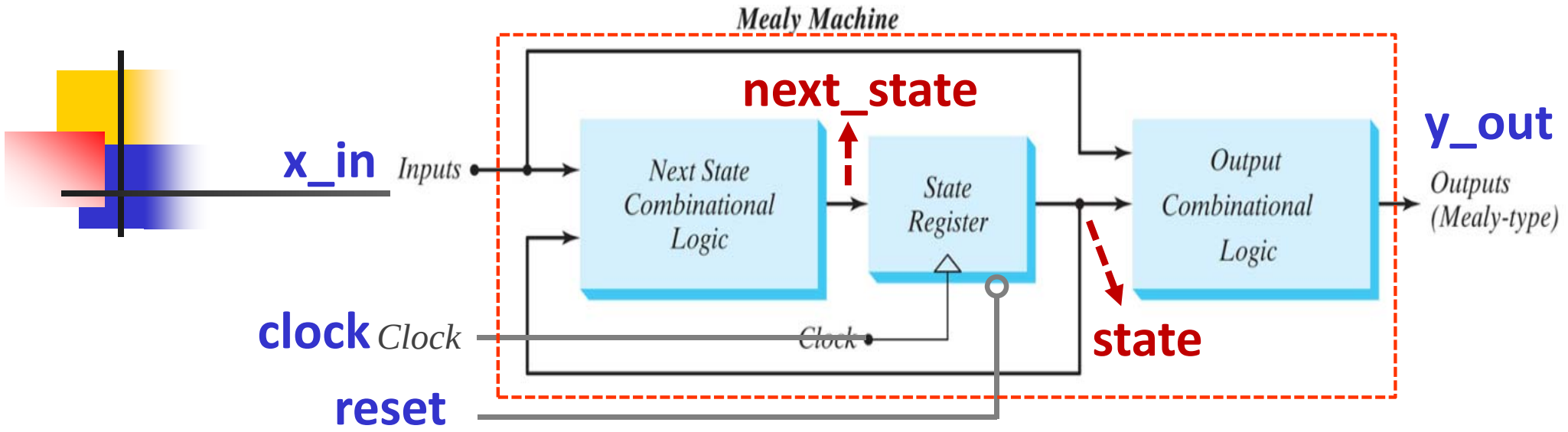
HDL Example 5.5: Mealy FSM Machine, Zero Detector

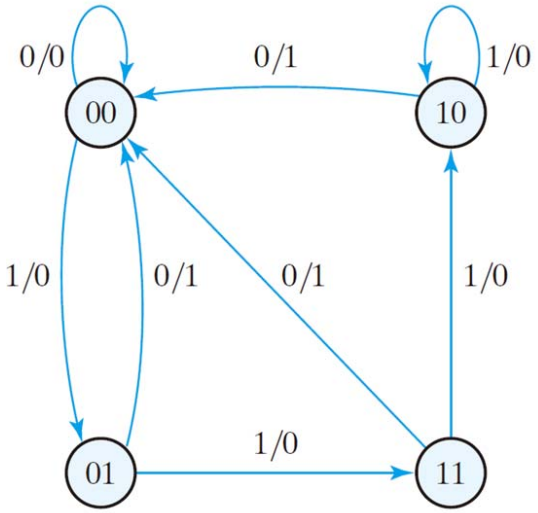
- HDL Example 5.5: Mealy Machine, Zero Detector
  - P.224, Fig 5.16, Table 5.3
  - *state diagram-based* model





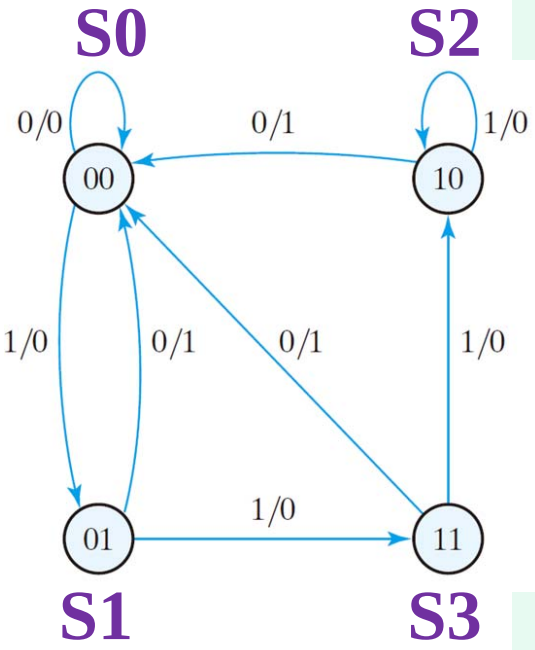
Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0





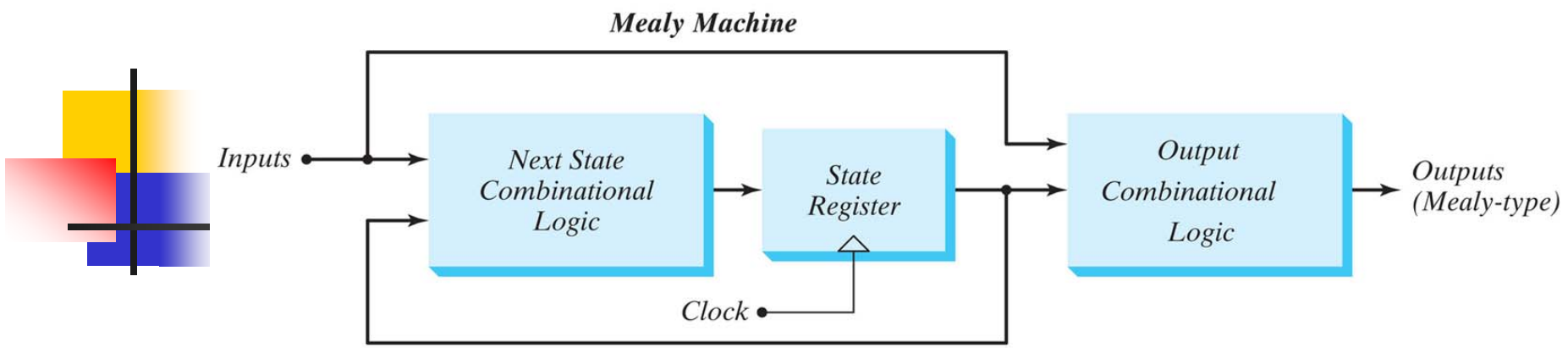
Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

```
module Mealy_Zero_Detector (output  reg  y_out,
                            input    x_in, clock, reset);
  reg [1: 0]     state, next_state;
  …

   //Form state transition-- State Register
  always @(posedge clock, negedge reset)
  …

   // Form the next state-- Next State Combinational Logic
  always @ (state, x_in)
  …

   // Form the output-- Output Combinational Logic
  always @ (state, x_in)
  …
endmodule
```



Present State		Next State				Output	
		$x = 0$		$x = 1$		$x = 0$	$x = 1$
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

```
module Mealy_Zero_Detector (output  reg  y_out,
                            input    x_in, clock, reset);
  reg [1: 0]    state, next_state;
  parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @(posedge clock, negedge reset) //state transition
     if (reset == 0) state <= S0;      //Active-LOW reset
     else state <= next_state;

  always @ (state, x_in)              // Form the next state
     case (state)
        S0: if (x_in)  next_state = S1; else next_state = S0;
        S1: if (x_in)  next_state = S3; else next_state = S0;
        S2: if (~x_in) next_state = S0; else next_state = S2;
        S3: if (x_in)   next_state = S2; else next_state = S0;
     endcase

  always @ (state, x_in)              // Form the output
     case (state)
        S0:   y_out = 0;
        S1, S2, S3: y_out = ~x_in;
     endcase
endmodule
```

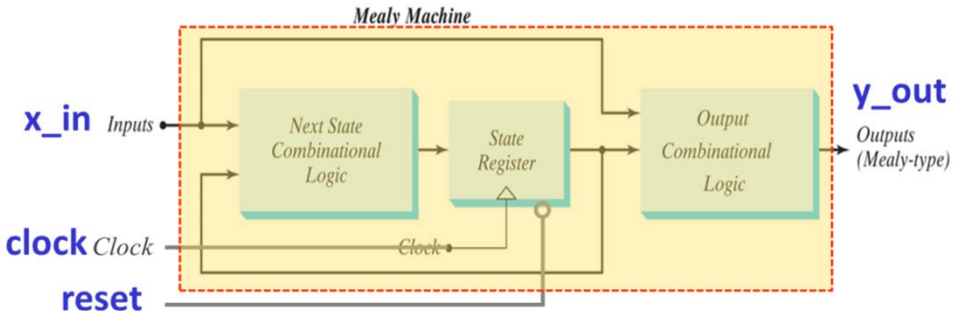



```
module Mealy_Zero_Detector (output  reg  y_out,
                            input    x_in, clock, reset);
  reg [1: 0]    state, next_state;
  parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
  always @(posedge clock, negedge reset) //state transition
     if (reset == 0) state <= S0;      //Active-LOW reset
     else state <= next_state;
  always @ (state, x_in)              // Form the next state
     case (state)
        S0: if (x_in)  next_state = S1; else next_state = S0;
        S1: if (x_in)  next_state = S3; else next_state = S0;
        S2: if (~x_in) next_state = S0; else next_state = S2;
        S3: if (x_in)   next_state = S2; else next_state = S0;
     endcase
  always @ (state, x_in)              // Form the output
     case (state)
        S0:   y_out = 0;
        S1, S2, S3: y_out = ~x_in;
     endcase
endmodule
```

state register

next state combinational logic

output combinational logic



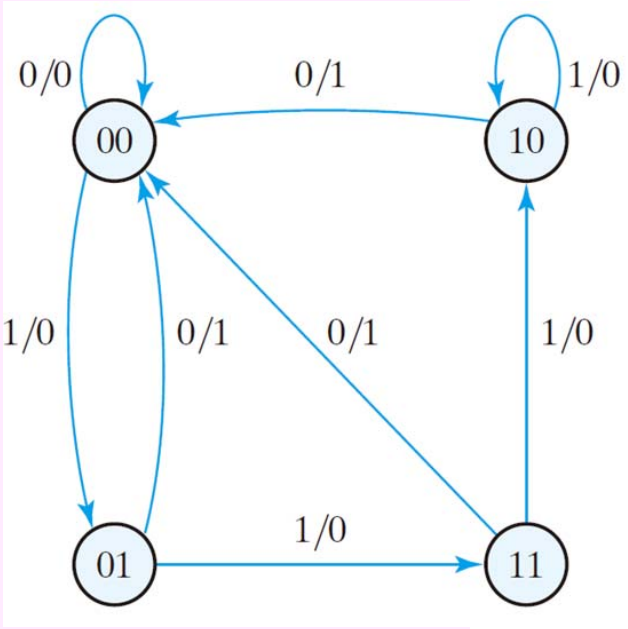
- Testbench of HDL Example 5.5:

```
module Mealy_Zero_Detector (output  reg  y_out,
                            input        x_in, clock, reset);
  reg [1: 0]    state, next_state;
  parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @(posedge clock, negedge reset) //state transition
     if (reset == 0) state <= S0;
     else state <= next_state;

  always @ (state, x_in)          // Form the next state
     case (state)
        S0: if (x_in)  next_state = S1; else next_state = S0;
        S1: if (x_in)  next_state = S3; else next_state = S0;
        S2: if (~x_in) next_state = S0; else next_state = S2;
        S3: if (x_in)   next_state = S2; else next_state = S0;
     endcase

  always @ (state, x_in)          // Form the output
     case (state)
        S0:   y_out = 0;
        S1, S2, S3: y_out = ~x_in;
     endcase
endmodule
```

```
module t_Mealy_Zero_Detector;
  wire t_y_out;
  reg   t_x_in, t_clock, t_reset;

  Mealy_Zero_Detector M0 (t_y_out,
   t_x_in, t_clock, t_reset);

  initial #200 $finish;

  //Generate stimulus for t_clock
  initial
     begin
        t_clock = 0;
        forever #5 t_clock = ~t_clock;
     end

  //Generate stimulus for t_x_in, t-reset
  initial fork
     ...
     join

endmodule
```

```
module Mealy_Zero_Detector (output  reg  y_out,
                            input    x_in, clock, reset);
  reg [1: 0]    state, next_state;
  parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @(posedge clock, negedge reset) //state transition
     if (reset == 0) state <= S0;
     else state <= next_state;

  always @ (state, x_in)            // Form the next st
     case (state)
        S0: if (x_in)  next_state = S1; else next_state
        S1: if (x_in)  next_state = S3; else next_state = S0;
        S2: if (~x_in) next_state = S0; else next_state = S2;
        S3: if (x_in)   next_state = S2; else next_state = S0;
     endcase

  always @ (state, x_in)            // Form the output
     case (state)
        S0:   y_out = 0;
        S1, S2, S3: y_out = ~x_in;
     endcase
endmodule
```

```
//Generate stimulus for t_x_in, t-reset
initial fork
        t_reset = 0;
#2      t_reset = 1;
#87     t_reset = 0;
#89     t_reset = 1;
#10     t_x_in = 1;
#30     t_x_in = 0;
#40     t_x_in = 1;
#50     t_x_in = 0;
#52     t_x_in = 1;
#54     t_x_in = 0;
#70     t_x_in = 1;
#80     t_x_in = 1;
#70     t_x_in = 0;
#90     t_x_in = 1;
#100    t_x_in = 0;
#120    t_x_in = 1;
#160    t_x_in = 0;
#170    t_x_in = 1;
join
endmodule
```

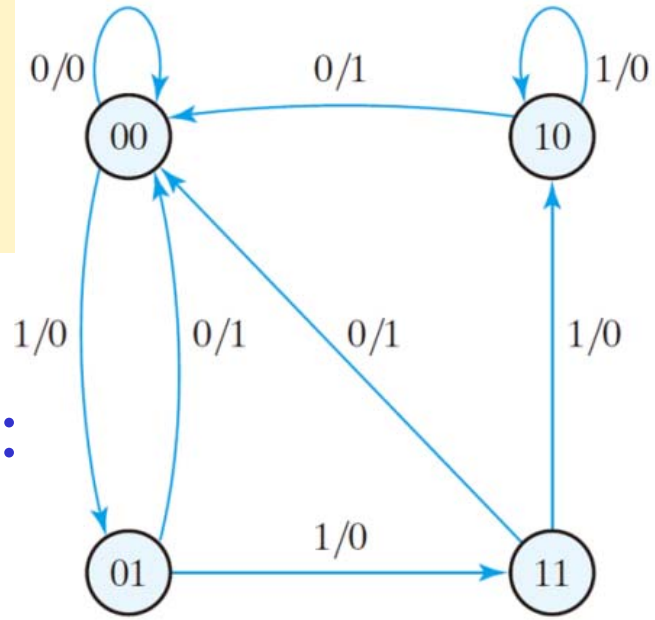


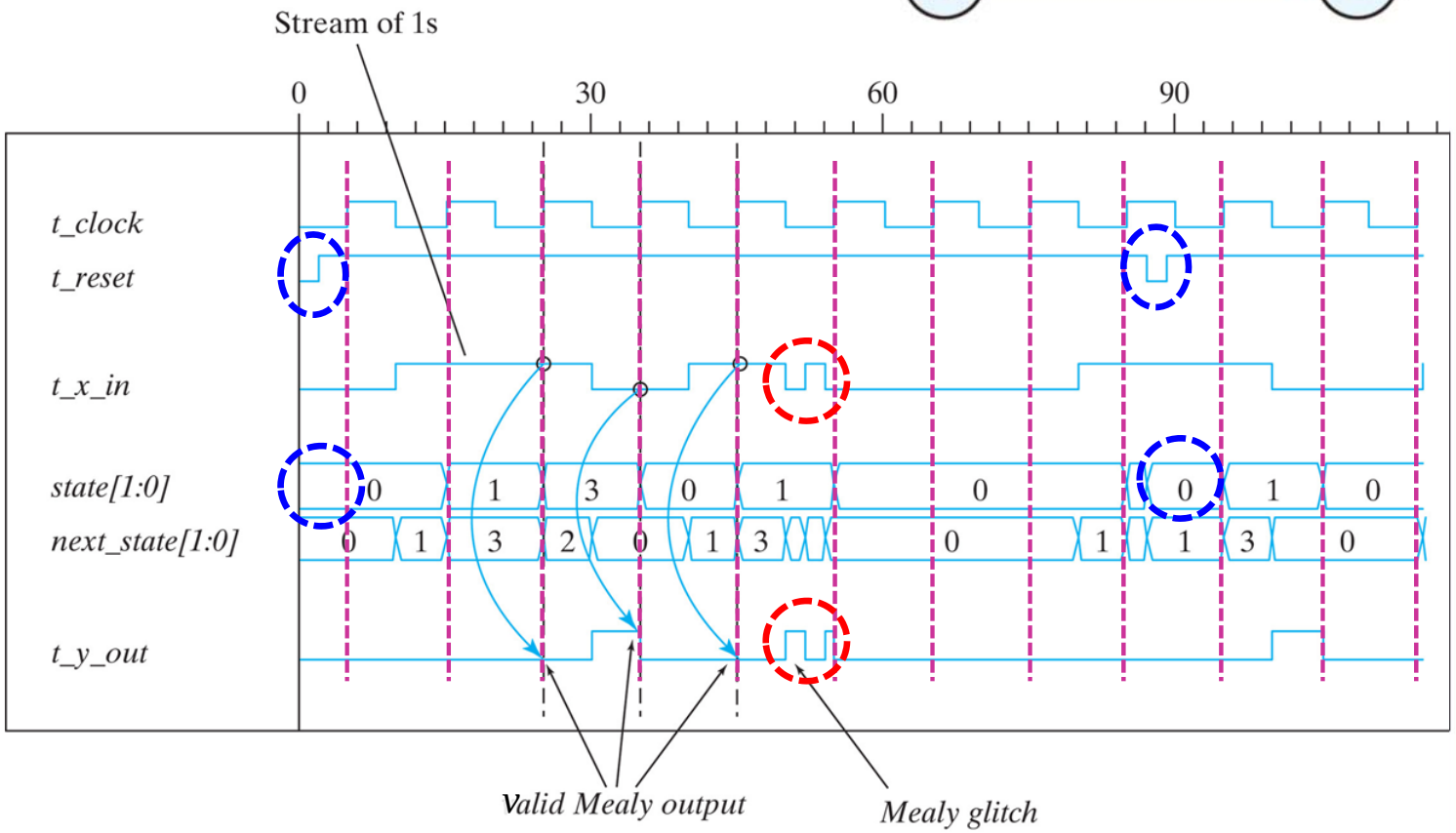
* Testbench: p.HDL-66

write stimuli that will test a ckt thoroughly

- Simulation output of Mealy_Zero_Detector:

p.242, Fig 5.22

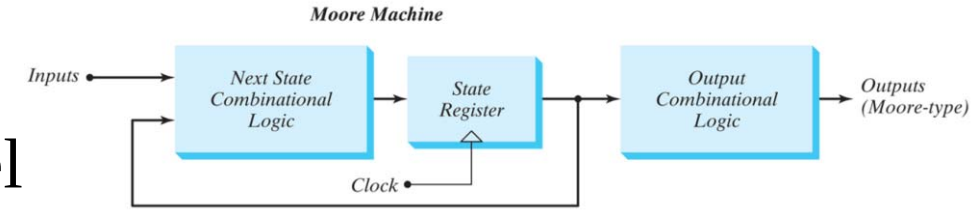


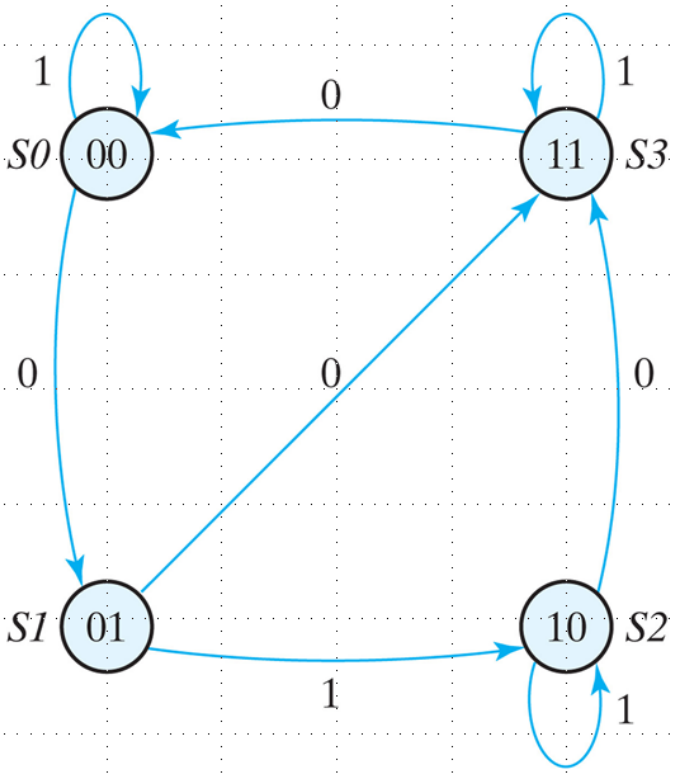



```
module t_Mealy_Zero_Detector;
  wire t_y_out;
  reg  t_x_in, t_clock, t_reset;

  Mealy_Zero_Detector M0 (t_y_out,
   t_x_in, t_clock, t_reset);

  initial #200 $finish;

//Generate stimulus for t_clock
  initial
     begin
        t_clock = 0;
        forever #5 t_clock = ~t_clock;
     end

  //Generate stimulus for t_x_in, t-reset
  initial fork
          t_reset = 0;
  #2      t_reset = 1;
  #87     t_reset = 0;
  #89     t_reset = 1;
  #10     t_x_in = 1;
  #30     t_x_in = 0;
  #40     t_x_in = 1;
  #50     t_x_in = 0;
  #52     t_x_in = 1;
  #54     t_x_in = 0;
  #70     t_x_in = 1;
  #80     t_x_in = 1;
  #70     t_x_in = 0;
  #90     t_x_in = 1;
  #100    t_x_in = 0;
  #120    t_x_in = 1;
  #160    t_x_in = 0;
  #170    t_x_in = 1;
  join
endmodule
```

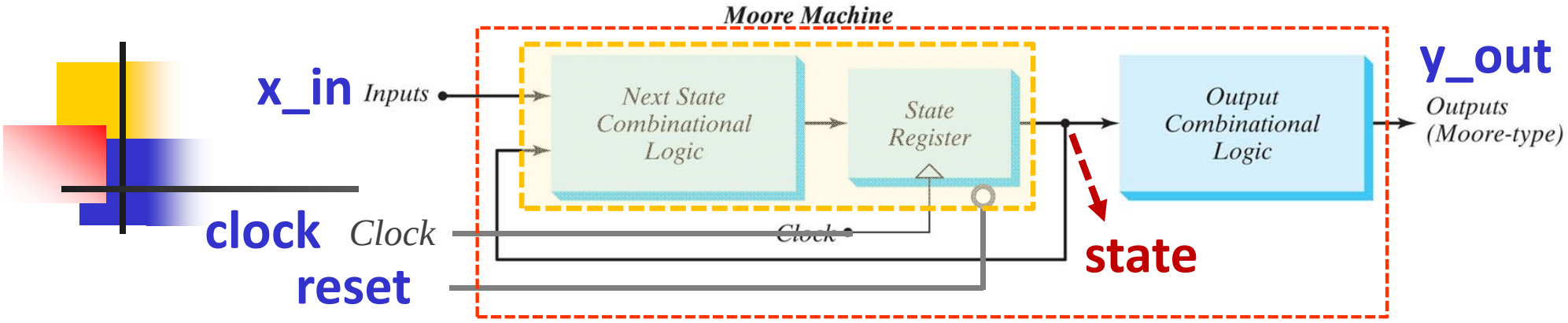
HDL Example 5.6: Moore FSM Machine, ~~Zero Detector~~

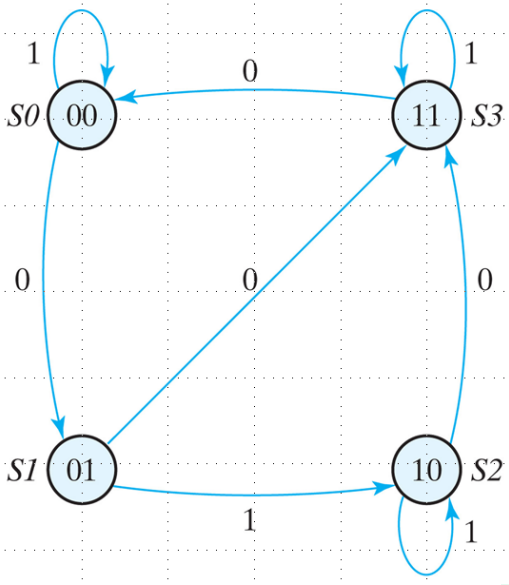
- HDL Example 5.6: Moore Machine, ~~Zero Detector~~
  - p.230, Fig 5.19
  - *state diagram-based* model





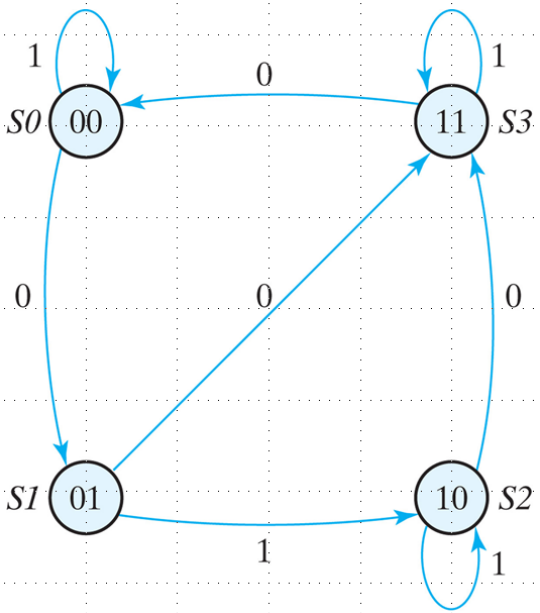
Present State		Input	Next State	
A	B	x	A	B
0	0	0	0	1
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

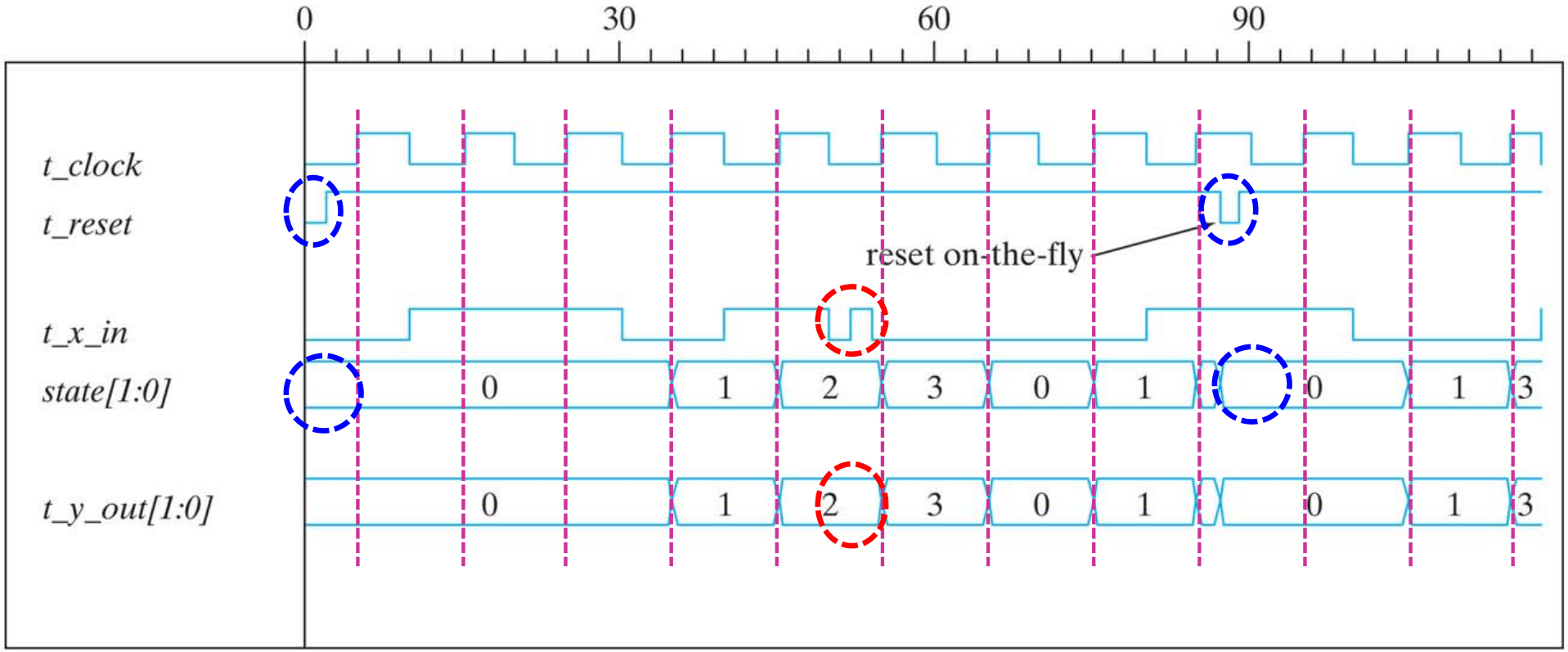





```
module Moore_Model_Fig_5_19 (output [1: 0] y_out,
                             input  x_in, clock, reset);
  reg [1: 0] state;
  parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

  always @ (posedge clock, negedge reset)
     if (reset == 0) state <= S0;   //Initialize to S0
     else case (state)
        S0:  if (~x_in) state <= S1; else state <= S0;
        S1:  if (x_in)   state <= S2; else state <= S3;
        S2:  if (~x_in) state <= S3; else state <= S2;
        S3:  if (~x_in) state <= S0; else state <= S3;
     endcase

  assign y_out = state;  // Output of flip-flops
endmodule
```

- Simulation output of Moore_Model_Fig_5_19:
  - p.244, Fig 5.23

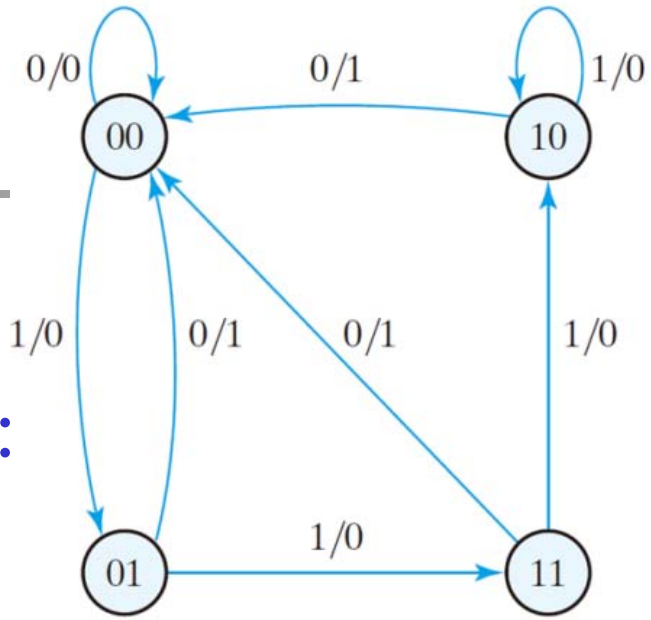


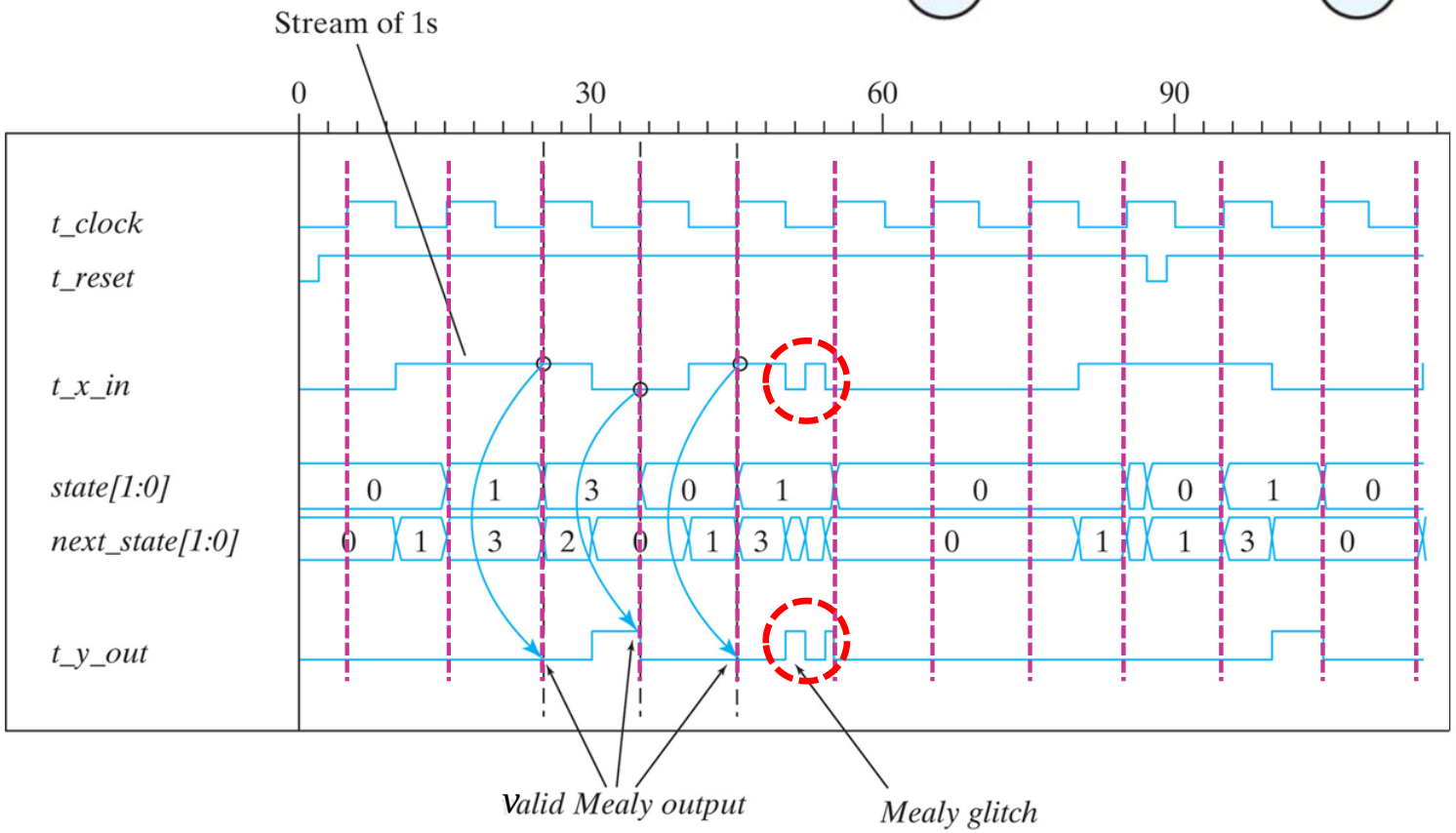


* No glitch

- Simulation output of Mealy_Zero_Detector:

p.242, Fig 5.22






```
module t_Mealy_Zero_Detector;
  wire t_y_out;
  reg  t_x_in, t_clock, t_reset;

  Mealy_Zero_Detector M0 (t_y_out,
   t_x_in, t_clock, t_reset);

  initial #200 $finish;

//Generate stimulus for t_clock
  initial
     begin
        t_clock = 0;
        forever #5 t_clock = ~t_clock;
     end

  //Generate stimulus for t_x_in, t-reset
  initial fork
          t_reset = 0;
  #2      t_reset = 1;
  #87     t_reset = 0;
  #89     t_reset = 1;
  #10     t_x_in = 1;
  #30     t_x_in = 0;
  #40     t_x_in = 1;
  #50     t_x_in = 0;
  #52     t_x_in = 1;
  #54     t_x_in = 0;
  #70     t_x_in = 1;
  #80     t_x_in = 1;
  #70     t_x_in = 0;
  #90     t_x_in = 1;
  #100    t_x_in = 0;
  #120    t_x_in = 1;
  #160    t_x_in = 0;
  #170    t_x_in = 1;
  join
endmodule
```

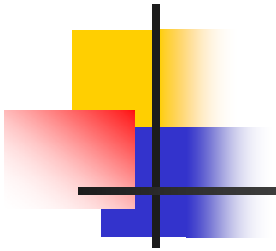
D. Structural Description of Clocked (Synchronous) Sequential Circuits

- Combinational logic ckts:
  - Theirs HDL models can be described

    by a *connection of gates* (*primitives* and *UDPs*),

    by *data flow statements* (*continuous assignments*), or

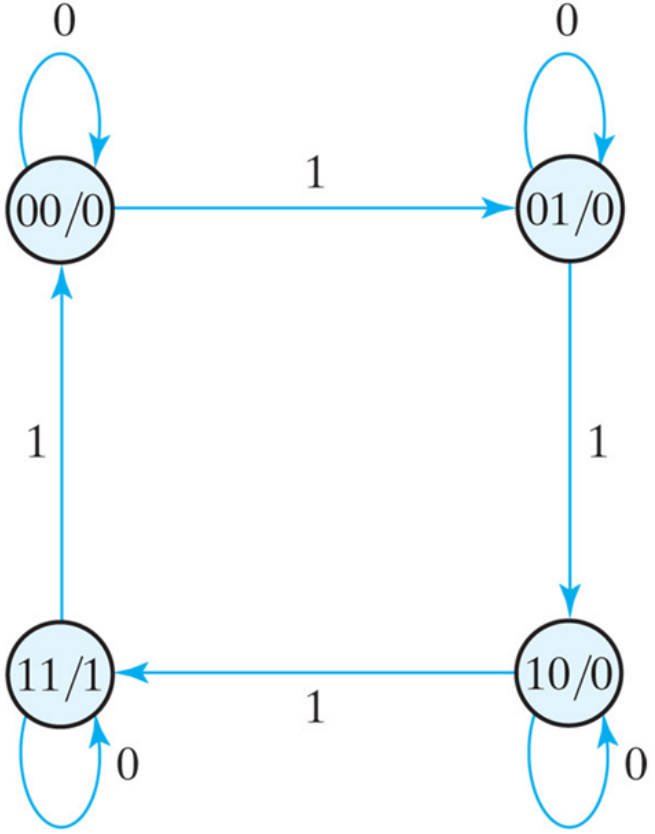
    by *level sensitive cyclic behaviors* (**always** blocks)
- Sequential ckts: Combinational logic + flip-flops
  - Theirs HDL models use *sequential UDPs* and *behavioral statements* (*edge-sensitive cyclic behaviors*) to describe the operation of flip-flops.
    - The flip-flops are described w/ an **always** statement.
    - The combinational part can be described w/ **assign** statements and Boolean equations.



- HDL models of sequential circuits:
  - State-diagram-based model (C.)
  - Structural model (D.)
- Structural model: combine separate modules by instantiation within a **module**.

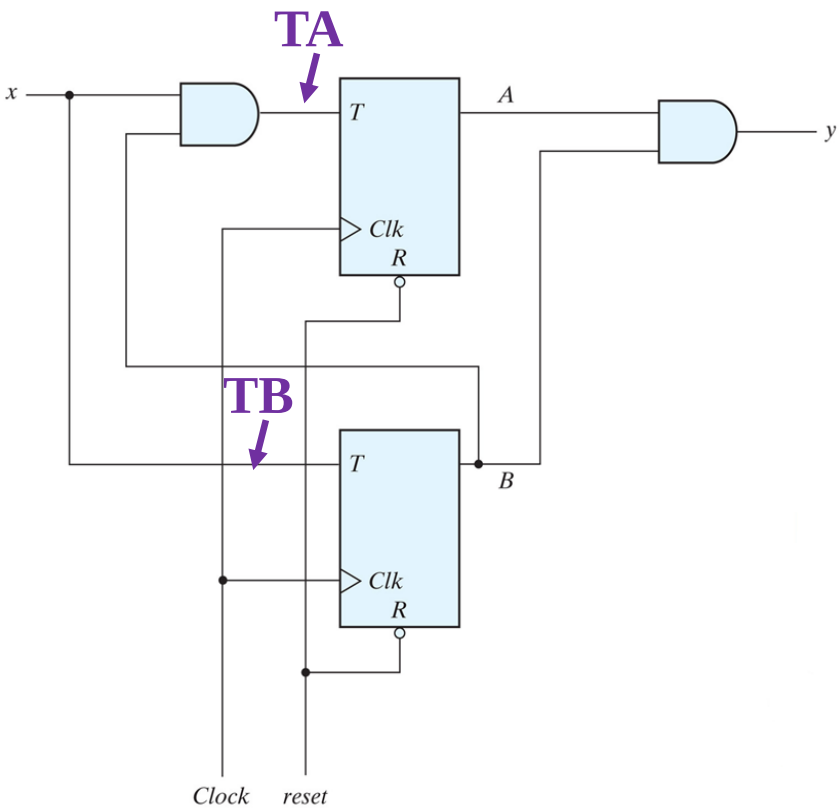
HDL Example 5.7: Binary Counter (Moore Machine)

- HDL Example 5.7(a)
  - *State-diagram-based* model
  - p.231, Fig 5.20(b)




```
module Moore_Model_Fig_5_20 (
   output  y_out,
   input    x_in, clock, reset
);
   reg [1: 0]    state;
   parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10,
                    S3 = 2'b11;

   always @ (posedge clock, negedge reset)
      if (reset == 0)  state <= S0;  //Initialize to S0
      else case (state)
         S0: if (x_in) state <= S1; else state <= S0;
         S1: if (x_in) state <= S2; else state <= S1;
         S2: if (x_in) state <= S3; else state <= S2;
         S3: if (x_in)   state <= S0; else state <= S3;
      endcase

   assign y_out = (state == S3);    //Output of f-fs
endmodule
```

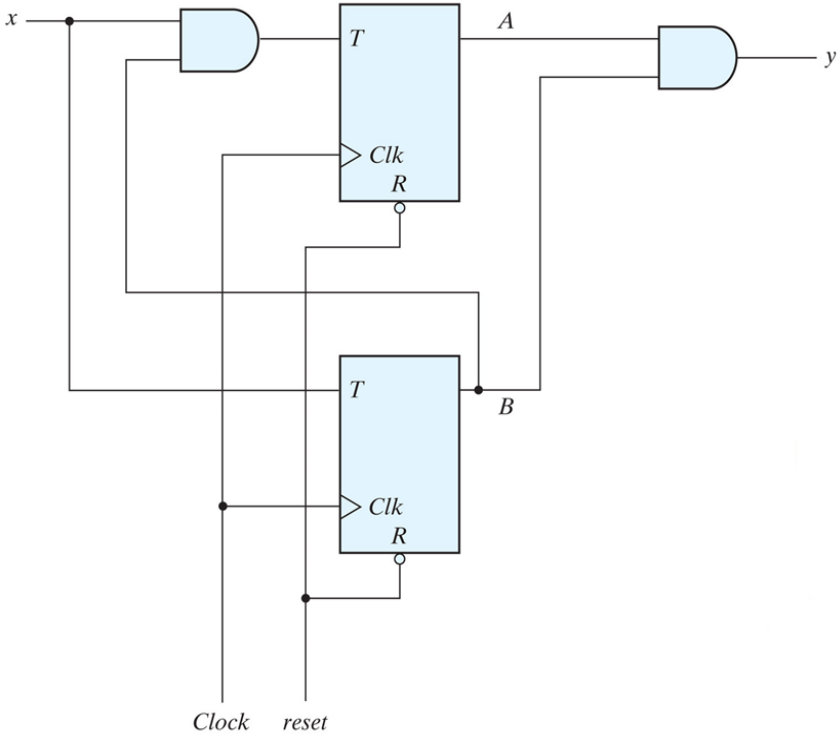
- HDL Example 5.7(b):
  - *Structural* model
  - p.231, Fig 5.20(a)




```
module Moore_Model_STR_Fig_5_20 (
   output   y_out, A, B,
   input    x_in, clock, reset
);
   wire TA, TB;

// Flip-flop input equations
   assign TA = x_in & B;
   assign TB = x_in;

//output equation
   assign y_out = A & B;

// Instantiate Toggle flip-flops
   Toggle_flip_flop_3 M_A (A, TA, clock, reset);
   Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule
```

- T flip-flop module for the circuit

```
module Moore_Model_STR_Fig_5_20 (
   output  y_out, A, B,
   input    x_in, clock, reset
);
   wire TA, TB;

// Flip-flop input equations
   assign TA = x_in & B;
   assign TB = x_in;
//output equation
   assign y_out = A & B;

// Instantiate Toggle flip-flops
  Toggle_flip_flop_3 M_A (A, TA, clock, reset);
  Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule
```




```
module Toggle_flip_flop_3 (Q, T, Clk, rst);
   output    Q;
   input     T, Clk, rst;
   reg       Q;

   always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;
      else  Q <= Q ^ T;
endmodule
```

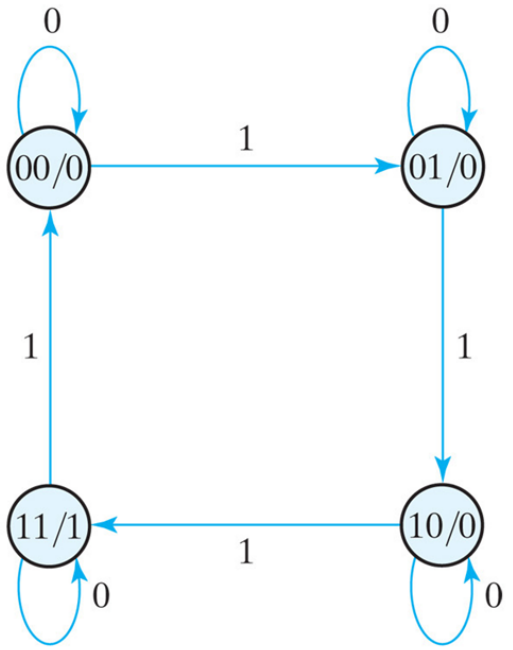
- Testbench of HDL Example 5.7(a)(b):

```
module Moore_Model_Fig_5_20 (
   output  y_out,
   input    x_in, clock, reset
);
   reg [1: 0]    state;
   parameter  S0 = 2'b00, S1 = 2'b01, S2 = 2'b10,
              S3 = 2'b11;

   always @ (posedge clock, negedge reset)
      if (reset == 0)  state <= S0;  //Initialize to S0
      else case (state)
         S0: if (x_in) state <= S1; else state <= S0;
         S1: if (x_in) state <= S2; else state <= S1;
         S2: if (x_in) state <= S3; else state <= S2;
         S3: if (x_in)   state <= S0; else state <= S3;
      endcase

   assign y_out = (state == S3);    //Output of f-fs
endmodule
```

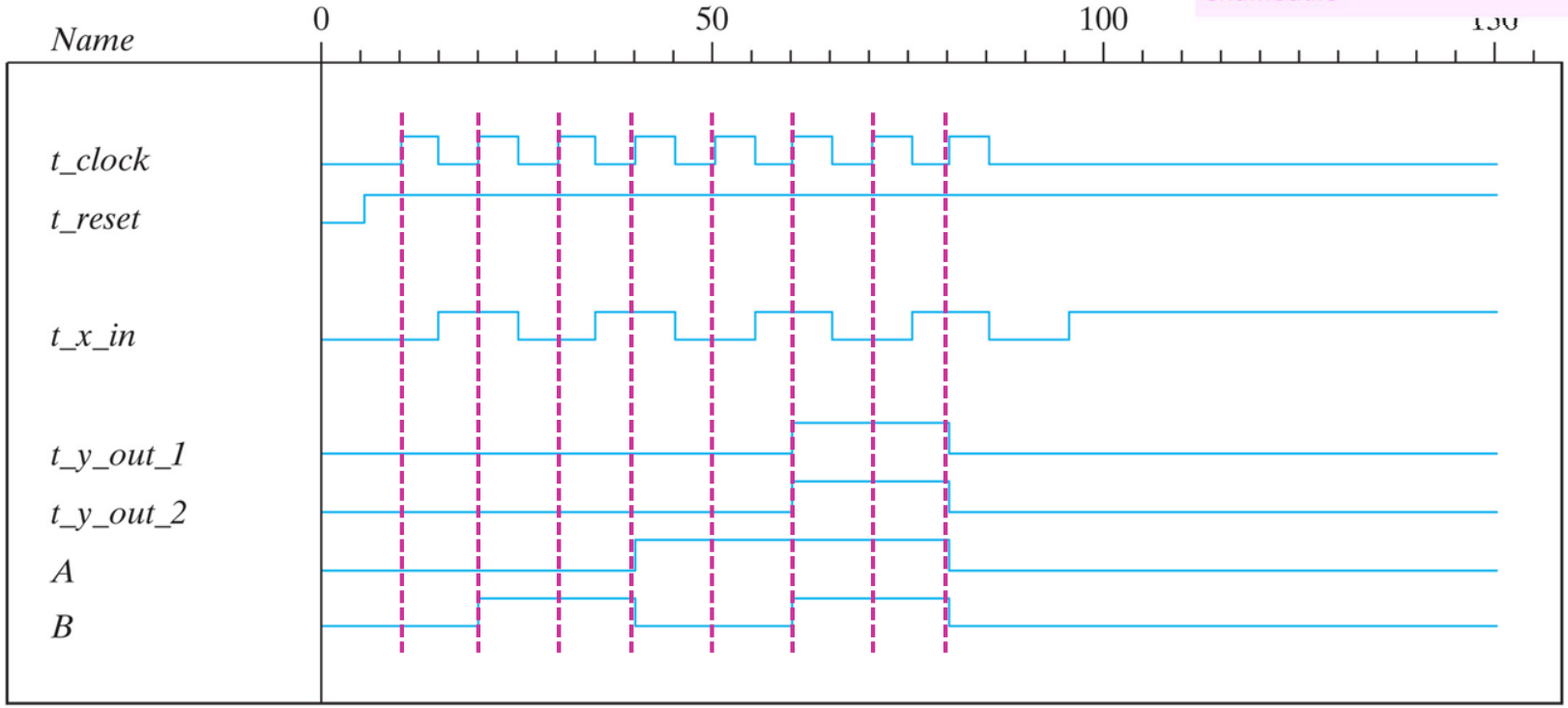
```
module Moore_Model_STR_Fig_5_20 (
   output  y_out, A, B,
   input    x_in, clock, reset
);
   wire TA, TB;

// Flip-flop input equations
   assign TA = x_in & B;
   assign TB = x_in;
//output equation
   assign y_out = A & B;

// Instantiate Toggle flip-flops
   Toggle_flip_flop_3 M_A (A, TA, clock, reset);
   Toggle_flip_flop_3 M_B (B, TB, clock, reset);
endmodule
```

```
module t_Moore_Fig_5_20;
   wire    t_y_out_2, t_y_out_1;
   reg   t_x_in, t_clock, t_reset;

   Moore_Model_Fig_5_20 M1
       (t_y_out_1, t_x_in, t_clock, t_reset);
   Moore_Model_STR_Fig_5_20 M2
       (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

   initial #200 $finish;

   initial begin
      t_reset = 0;
      t_clock = 0;
      #5  t_reset = 1;
      repeat (16)
         #5  t_clock = ~t_clock;
   end

   initial begin
      t_x_in = 0;
      #15 t_x_in = 1;
      repeat (8)
         #10 t_x_in = ~t_x_in;
   end
endmodule
```

- Simulation output of HDL Example 5.7:
  - p.247, Fig 5.24




```
module t_Moore_Fig_5_20;
   wire    t_y_out_2, t_y_out_1;
   reg   t_x_in, t_clock, t_reset;

   Moore_Model_Fig_5_20 M1
       (t_y_out_1, t_x_in, t_clock, t_reset);
   Moore_Model_STR_Fig_5_20 M2
       (t_y_out_2, A, B, t_x_in, t_clock, t_reset);

   initial #200 $finish;

   initial begin
      t_reset = 0;
      t_clock = 0;
      #5  t_reset = 1;
      repeat (16)
         #5  t_clock = ~t_clock;
   end

   initial begin
      t_x_in = 0;
      #15 t_x_in = 1;
      repeat (8)
         #10 t_x_in = ~t_x_in;
   end
endmodule
```



* Test the ckt thoroughly!

Mano 6-6

HDL for Registers and Counters

J.J. Shann

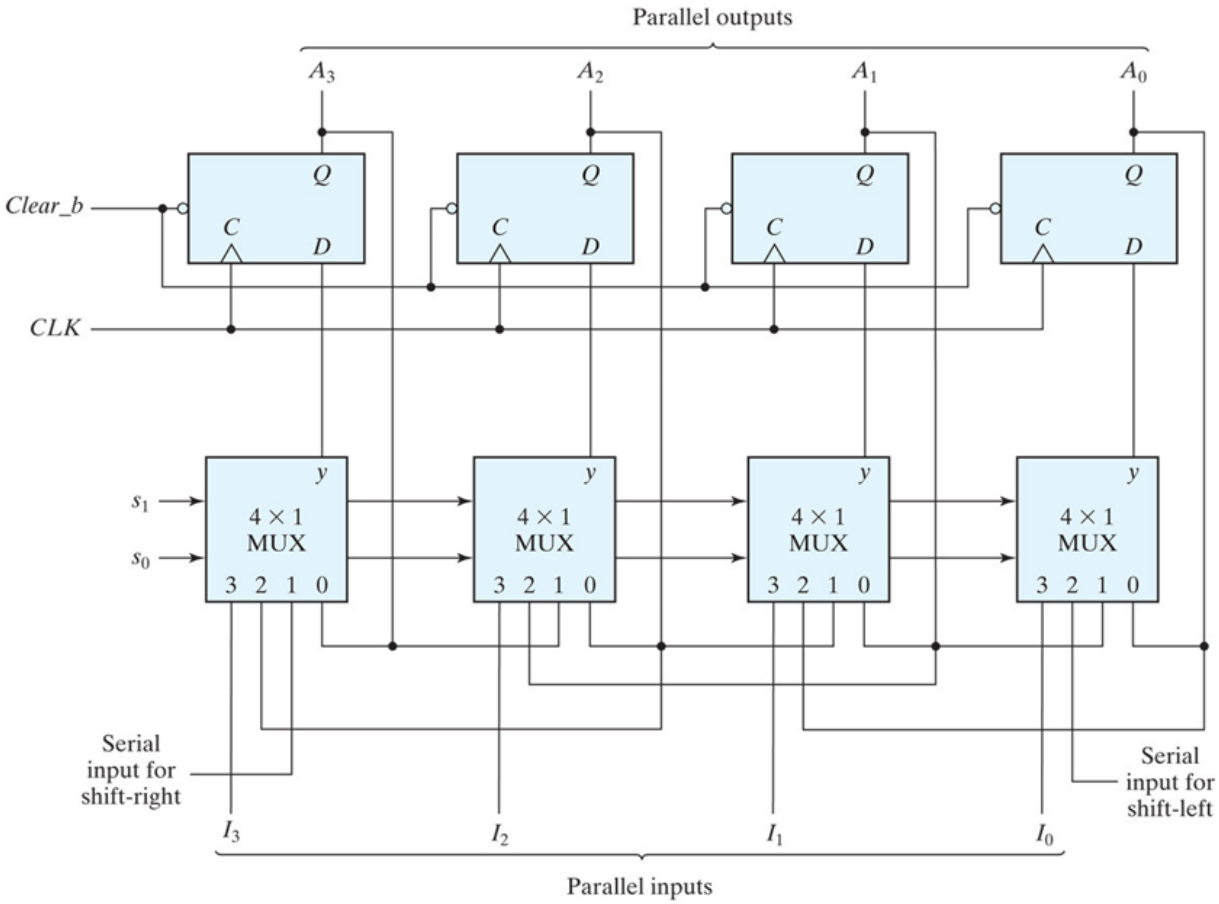
HDL Models of Registers and Counters

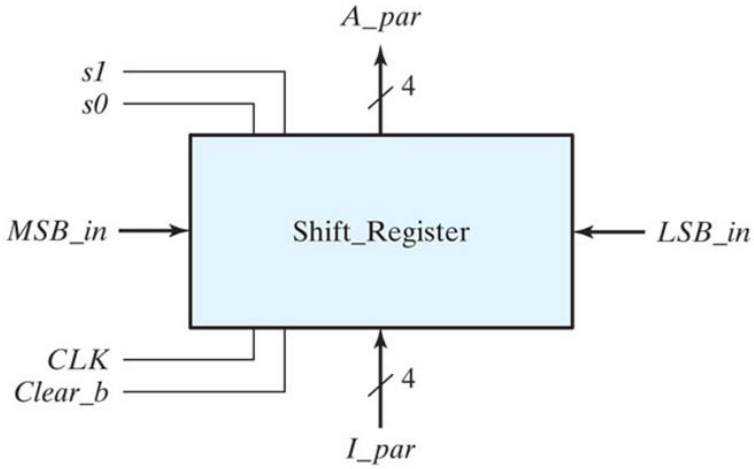
- HDL models of registers and counters:
  - Behavioral level: describes only the operations of the register, as prescribed by a function table, w/o a preconceived structure.
  - Structural level: shows the ckt in terms of a collection of components such as gates, flip-flops, and multiplexers.
    - The various components are instantiated to form a *hierarchical* description of the design.

* When a machine is complex, a *hierarchical* description creates a physical partition of the machine into simpler and more easily described units.

A. Shift Register

- University shift register:
  - p. 281, Fig 6.7, p.282, Table 6.3



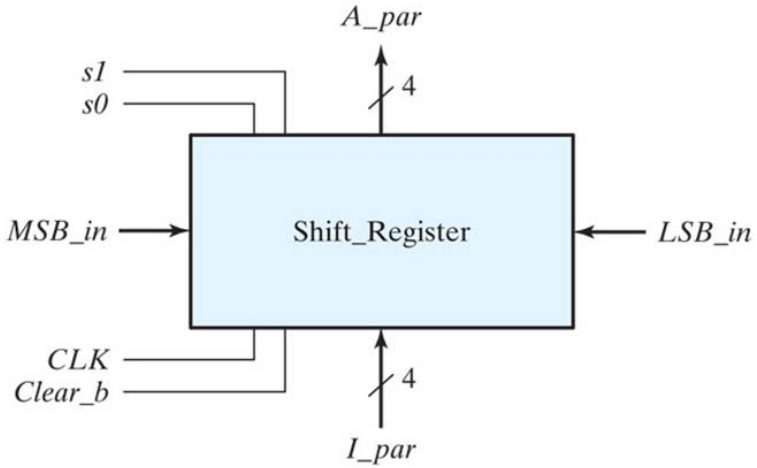


Function table

Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

HDL Example 6.1: University Shift Register (Behavioral Model)

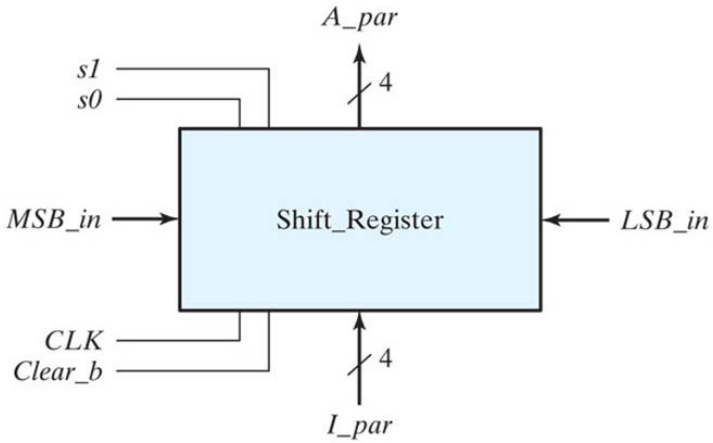
- HDL Example 6.1: 4-bit University Shift Register (Behavioral Model)-- p. 281, Fig 6.7(a), p.282, Table 6.3



Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

```
module Shift_Register_4_beh (          // V2001, 2005
   output  reg [3: 0]  A_par,          // Register output
   input       [3: 0]  I_par,          // Parallel input
   input               s1, s0,         // Select inputs
                       MSB_in, LSB_in, // Serial inputs
                       CLK, Clear_b    // Clock and Clear_b
);
```

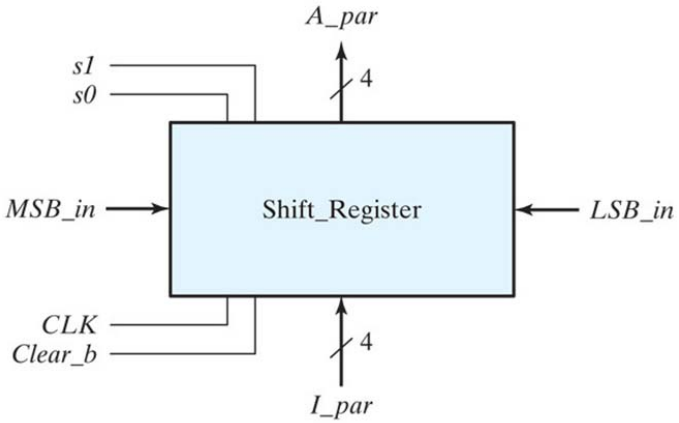
```
module Shift_Register_4_beh (
   output  reg [3: 0]  A_par,
   input       [3: 0]  I_par,
   input               s1, s0,
                       MSB_in, LSB_in,
                       CLK, Clear_b
);

   always @ (posedge CLK, negedge Clear_b)
      if (~Clear_b) A_par <= 4'b0000;
      else
         case ({s1, s0})
            2'b00: A_par <= A_par;    // No change
            2'b01: A_par <= {MSB_in, A_par[3: 1]};   // Shift right
            2'b10: A_par <= {A_par[2: 0], LSB_in};    // Shift left
            2'b11: A_par <= I_par;    // Parallel load of input
         endcase
endmodule
```

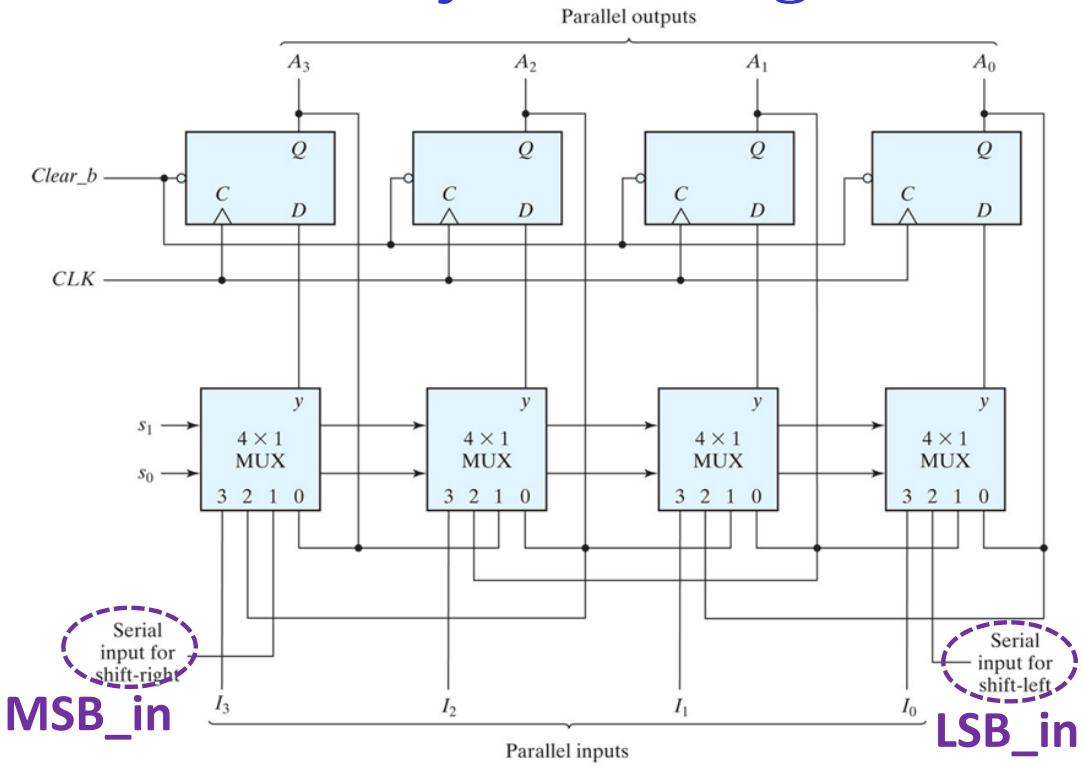


Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

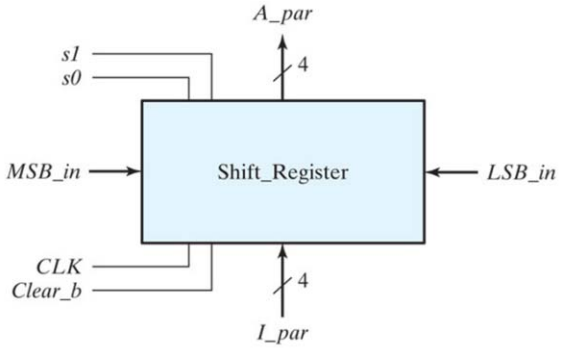
HDL Example 6.2: University Shift Register (Structural Model)

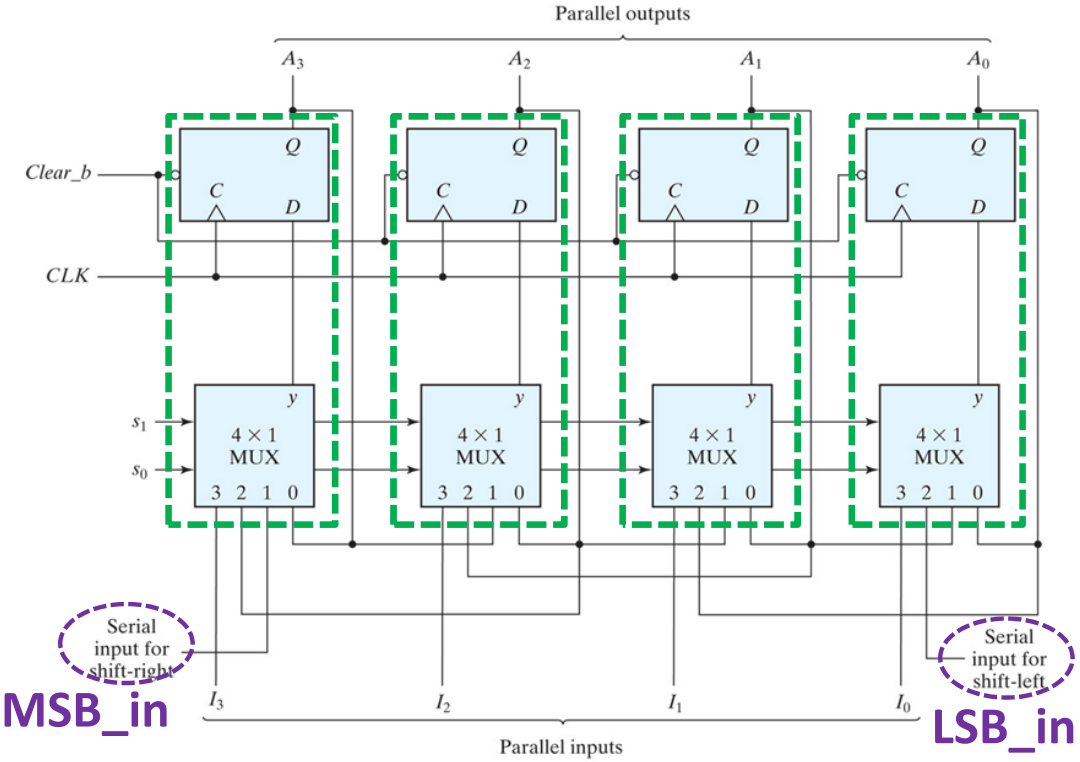
- HDL Example 6.2: 4-bit University Shift Register (Structural Model)
  - p. 281, Fig 6.7(a)(b)






```
module Shift_Register_4_str (                          // V2001, 2005
    output   [3: 0]   A_par,                           // Parallel output
    input    [3: 0]   I_par,                           // Parallel input
    input             s1, s0,                          // Mode select
    input             MSB_in, LSB_in, CLK, Clear_b     // Serial inputs, clock, clear
);
```

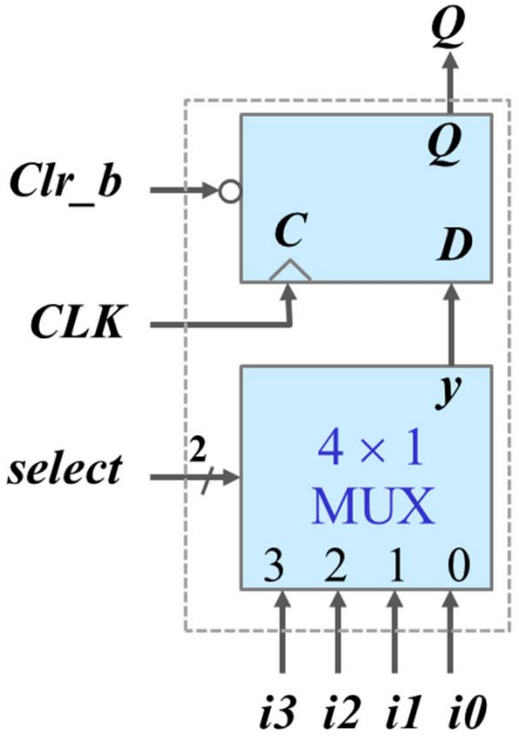




MSB_in

LSB_in

Shift_Register_4_str



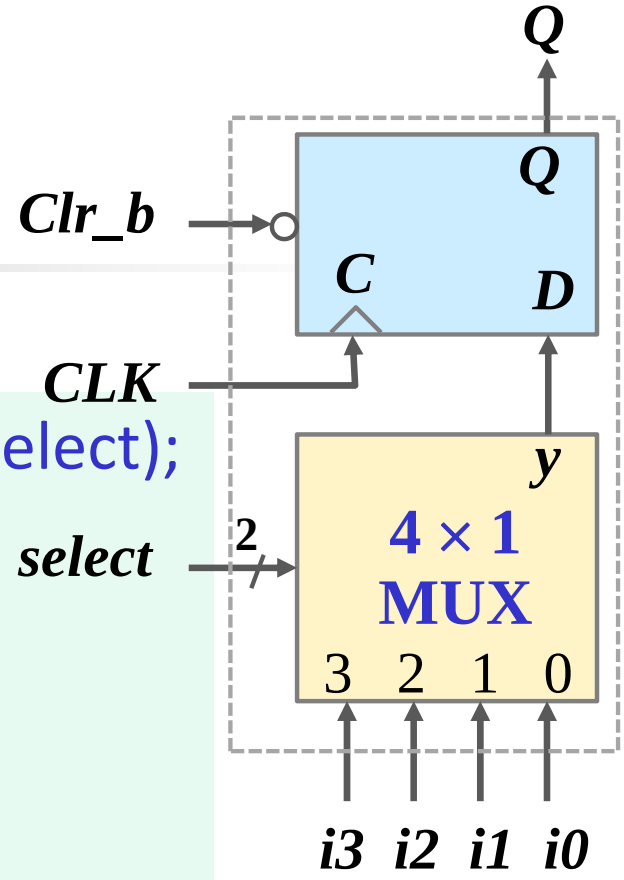
D_flip_flop

Mux_4_x_1

stage

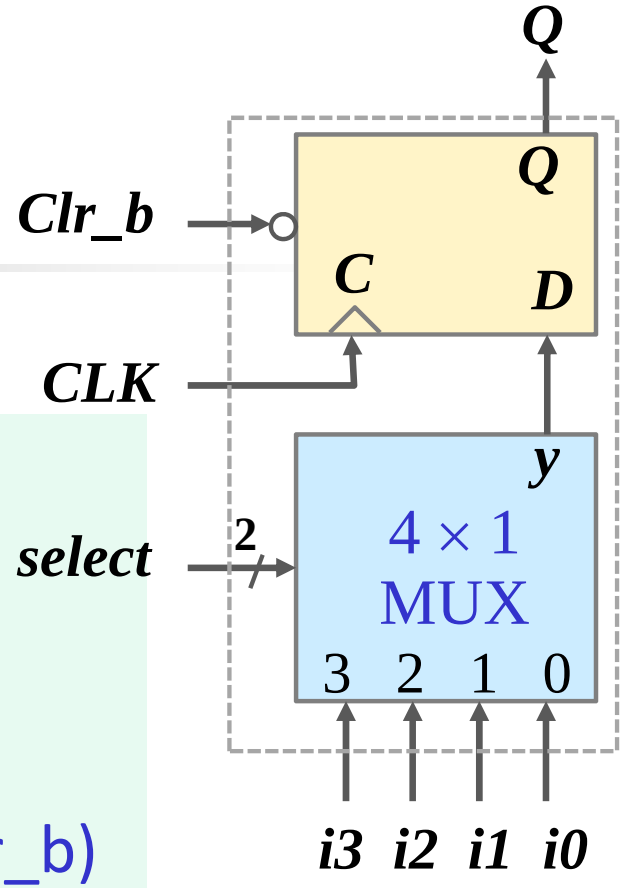
– 4x1 multiplexer (behavioral model):

```
module Mux_4_x_1 (mux_out, i0, i1, i2, i3, select);
   output          mux_out;
   input           i0, i1, i2, i3;
   input    [1: 0] select;
   reg             mux_out;

   always @ (select, i0, i1, i2, i3)
      case (select)
         2'b00:    mux_out = i0;
         2'b01:    mux_out = i1;
         2'b10:    mux_out = i2;
         2'b11:    mux_out = i3;
      endcase
endmodule
```

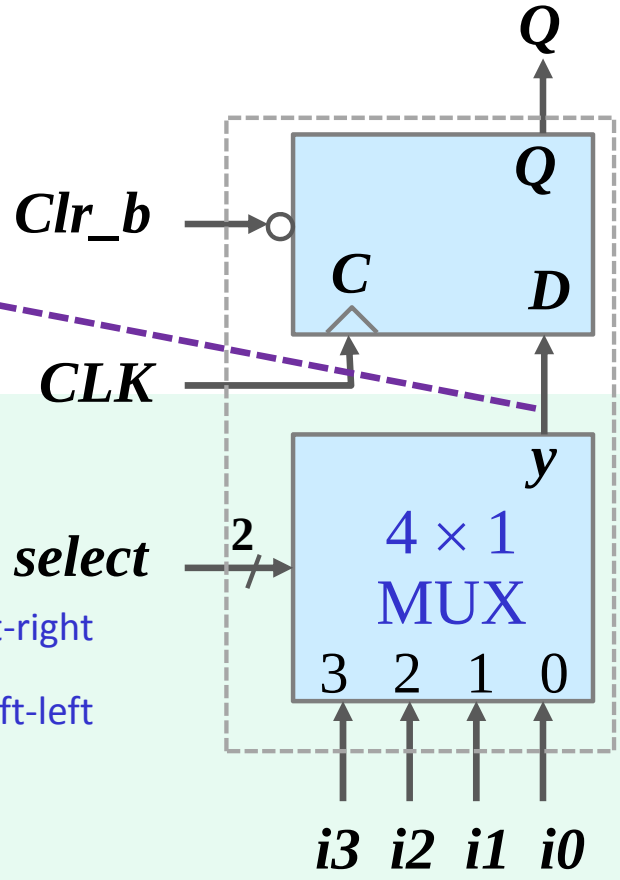


– D flip-flop (behavioral model):

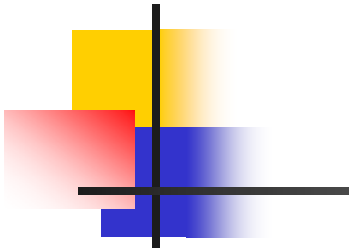
```
module D_flip_flop (Q, D, CLK, Clr_b);
   output   Q;
   input    D, CLK, Clr_b;
   reg      Q;

   always @ (posedge CLK or negedge Clr_b)
      if (!Clr_b) Q <= 1'b0; else Q <= D;
endmodule
```

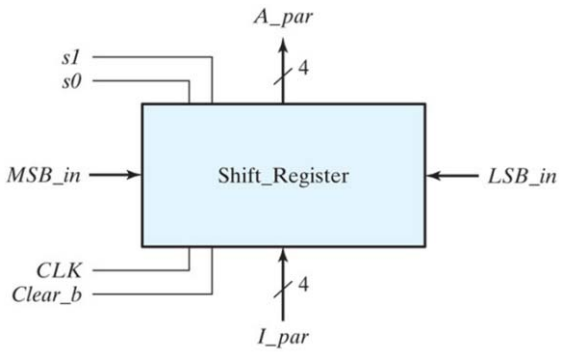


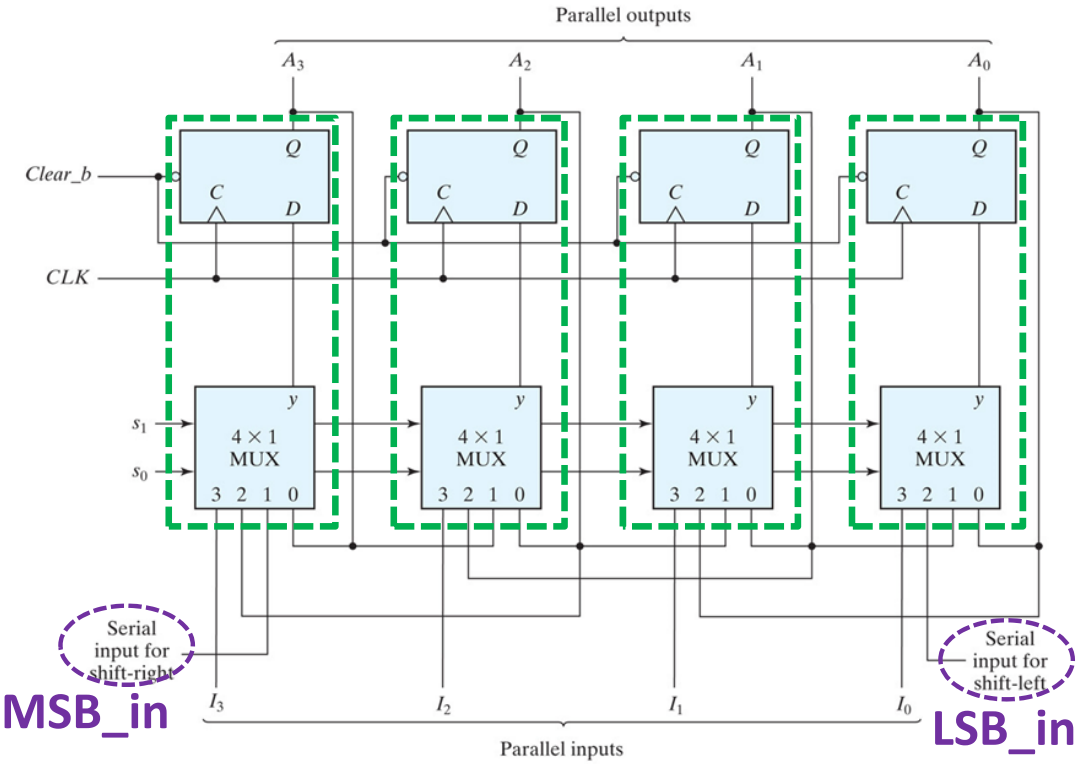
– 1-stage of shift register:




```
module stage (i0, i1, i2, i3, Q, select, CLK, Clr_b);
   input            i0,   //circulation bit selection
                    i1,   //data from left neighbor or serial input for shift-right
                    i2,   //data from right neighbor or serial input for shift-left
                    i3;   //data from parallel input
   output           Q;
   input    [1: 0]  select;        //stage mode control bus
   input            CLK, Clr_b;   //Clock, Clear for flip-flops
   wire             mux_out;

// instantiate mux and flip-flop
  Mux_4_x_1 M0 (mux_out, i0, i1, i2, i3, select);
  D_flip_flop M1 (Q, mux_out, CLK, Clr_b);
endmodule
```



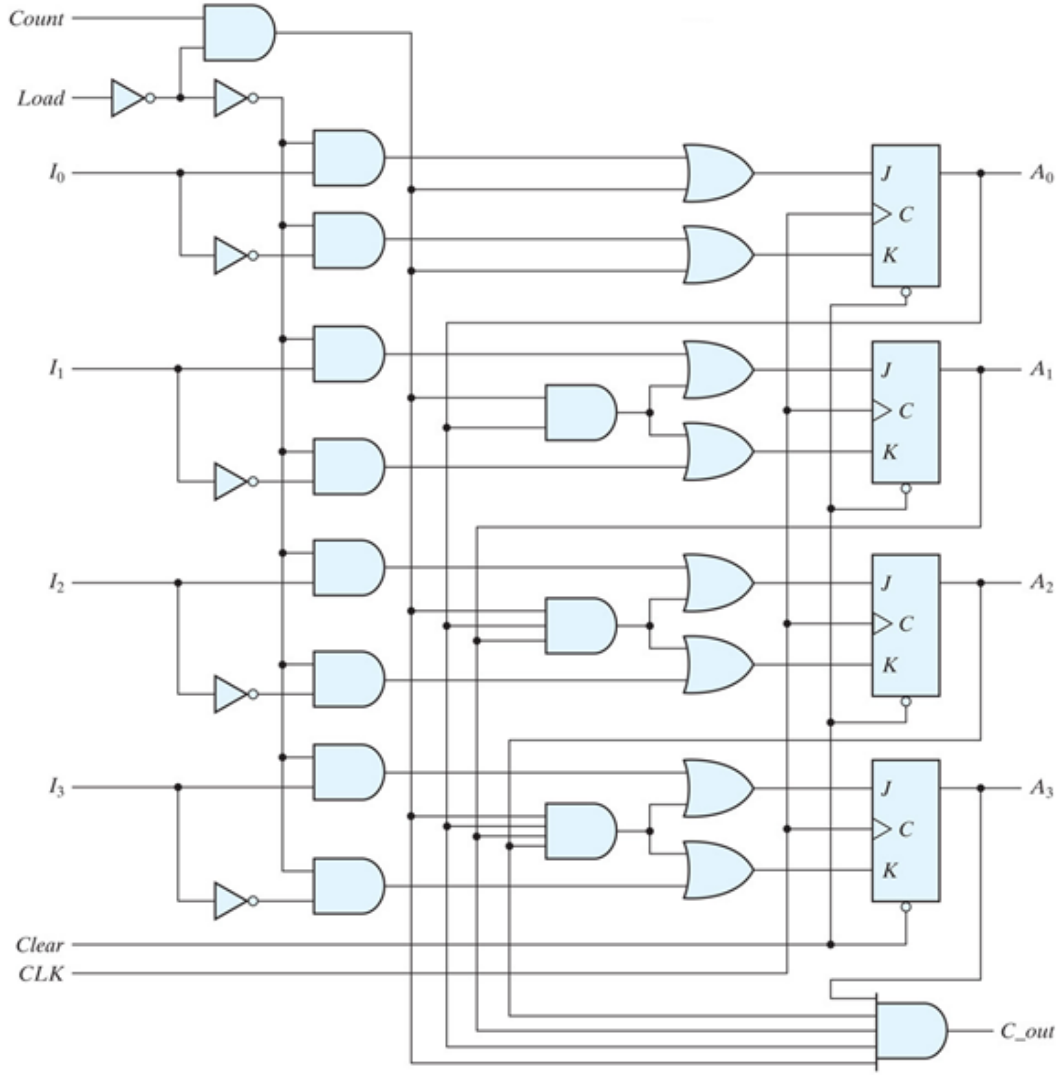


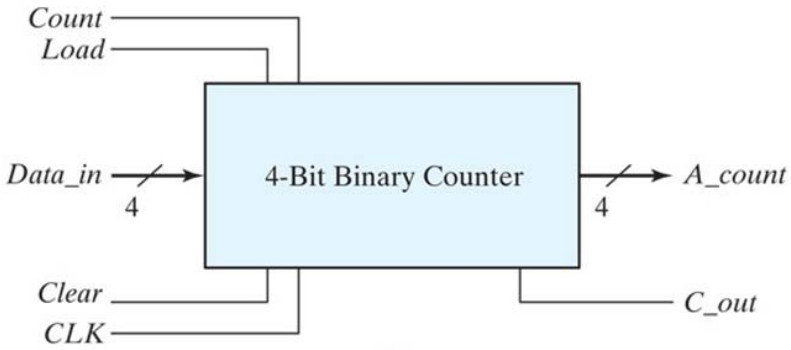



```
module Shift_Register_4_str (
    output   [3: 0]   A_par,
    input    [3: 0]   I_par,
    input             s1, s0,
    input             MSB_in, LSB_in, CLK, Clear_b      // Serial inputs, clock, clear
);

// bus for mode control
    wire  [1:0]     select = {s1, s0};

// Instantiate the four stages
    stage ST0 (A_par[0], A_par[1], LSB_in, I_par[0], A_par[0], select, CLK, Clear_b);
    stage ST1 (A_par[1], A_par[2], A_par[0], I_par[1],A_par[1], select, CLK, Clear_b);
    stage ST2 (A_par[2], A_par[3], A_par[1], I_par[2], A_par[2], select, CLK, Clear_b);
    stage ST3 (A_par[3], MSB_in, A_par[2], I_par[3], A_par[3], select, CLK, Clear_b);
endmodule
```

B. Synchronous Counter

- Synchronous counter:
  - p.292, Table 6.6, p. 293, Fig 6.14

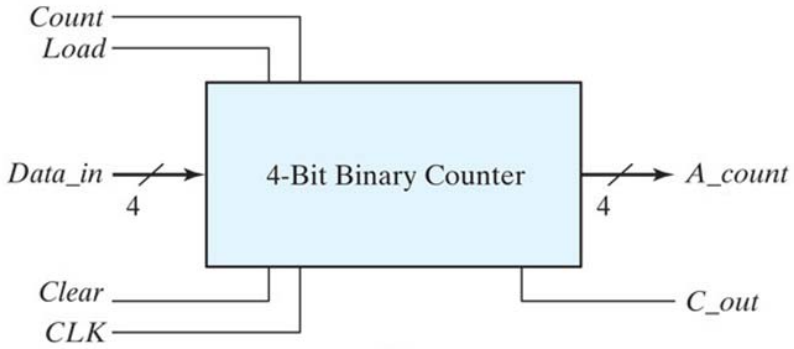




Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

HDL Example 6.3: Synchronous Counter

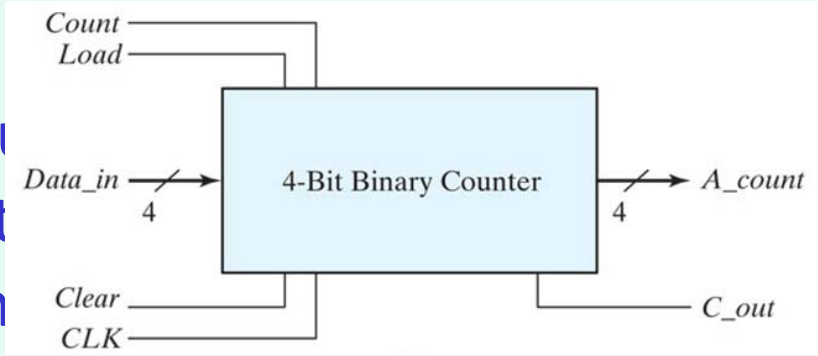
- HDL Example 6.3: 4-bit Binary Counter with Parallel Load-- p.292, Table 6.6, p. 293, Fig 6.14



Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

```
module Binary_Counter_4_Par_Load (
   output  reg  [3:0]  A_count,     // Data output
   output              C_out,       // Output carry
   input  [3:0]        Data_in,     // Data input
   input               Count,       // Active high to count
                       Load,        // Active high to load
                       CLK,         // Positive edge sensitive
                       Clear_b      // Active low
);
```

```
module Binary_Counter_4_Par_Load (
   output  reg  [3:0]  A_count,     // Data ou
   output               C_out,      // Output
   input  [3:0]         Data_in,    // Data in
   input                Count,      // Active high to count
                        Load,
                        CLK,
                        Clear_b
);
```

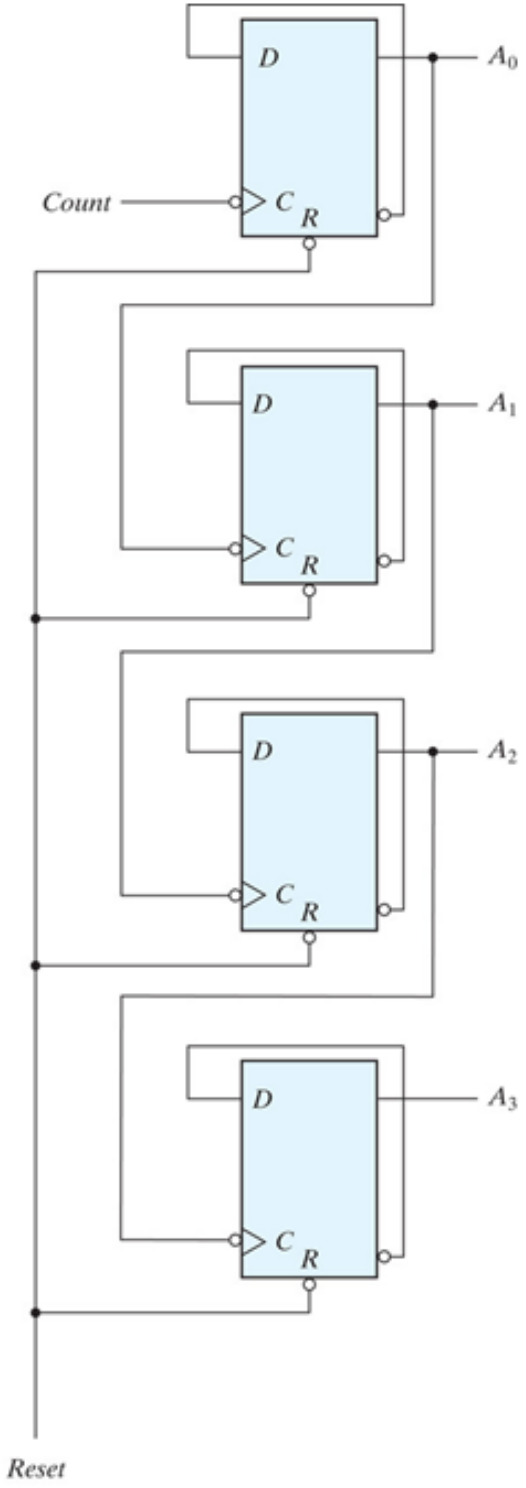


Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

```
   assign C_out = Count & (~Load) & (A_count == 4'b1111);
   always @ (posedge CLK, negedge Clear_b)
      if (~Clear_b)       A_count <= 4'b0000;
      else if (Load)      A_count <= Data_in;
      else if (Count)     A_count <= A_count + 1'b1;
      else                A_count <= A_count;   // redundant statement
endmodule
```

C. Ripple Counter

- Ripple counter:
  - p.284, Fig 6.8(b)

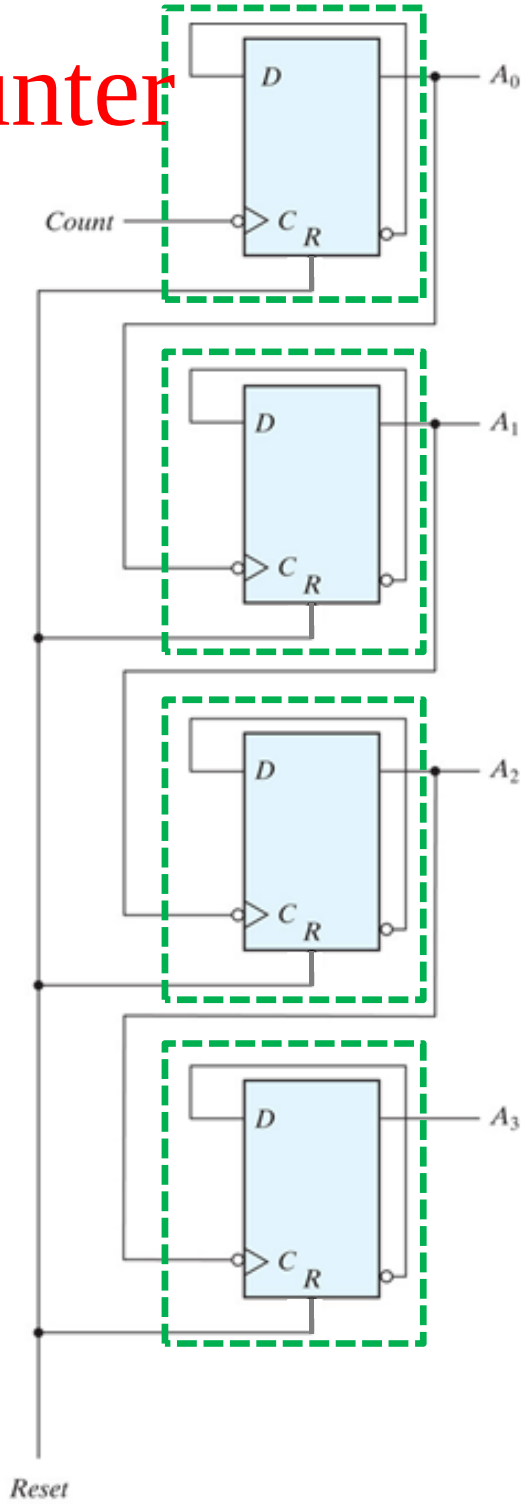


HDL Example 6.4: Ripple Counter

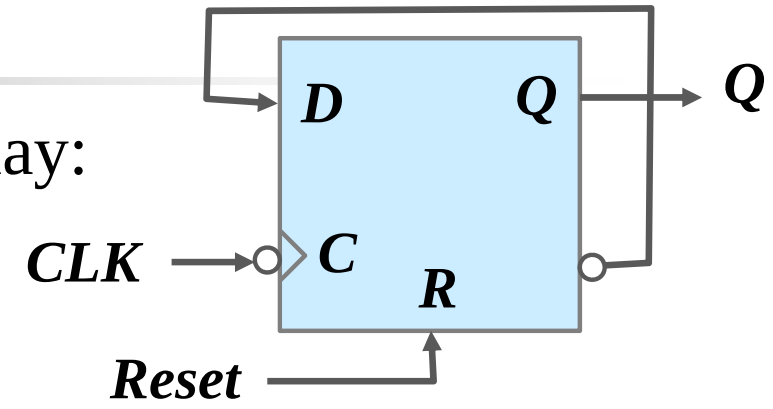
- HDL Example 6.4: 4-bit Binary Ripple Counter
  - p.284, Fig 6.8(b); w/ active-HIGH reset

```
`timescale 1ns / 100 ps                    //compiler directive
module Ripple_Counter_4bit (A3,A2,A1,A0, Count, Reset);
   output A3, A2, A1, A0;
   input   Count, Reset;

//Instantiate complementing flip-flop
   Comp_D_flip_flop F0 (A0, Count, Reset);
   Comp_D_flip_flop F1 (A1, A0, Reset);
   Comp_D_flip_flop F2 (A2, A1, Reset);
   Comp_D_flip_flop F3 (A3, A2, Reset);
endmodule
```

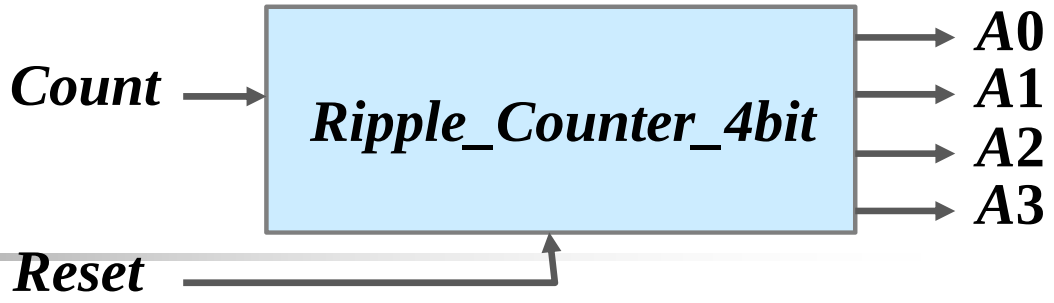
output **clock** **reset**



- Complementing flip-flop with delay:
  - Input to D flip-flop = Q′




```
module Comp_D_flip_flop (Q, CLK, Reset);
   output   Q;
   input    CLK, Reset;
   reg      Q;

   always @ (negedge CLK, posedge Reset)
      if (Reset) Q <= 1'b0; else Q <= #2 ~Q;   //Intra-assignment delay
endmodule
```

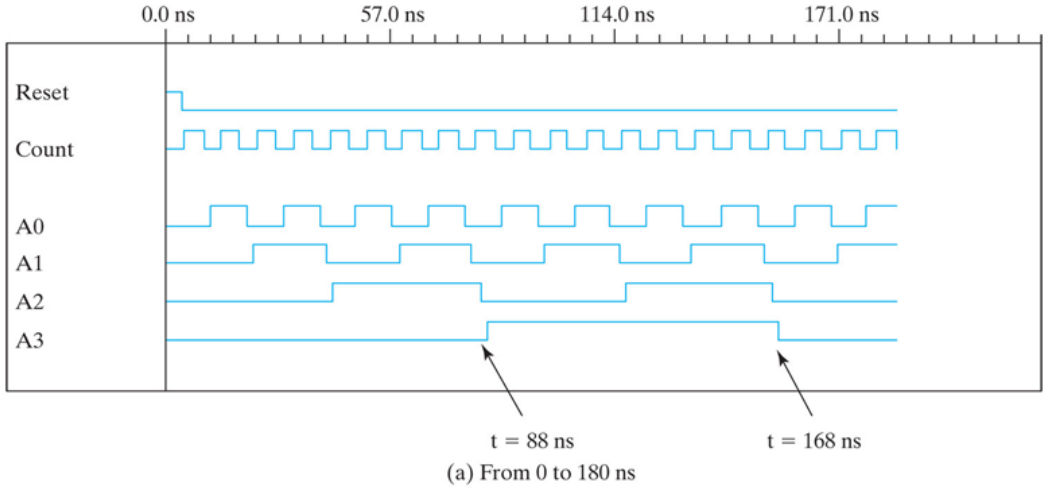


- Testbench of HDL Example 5.7(a)(b):

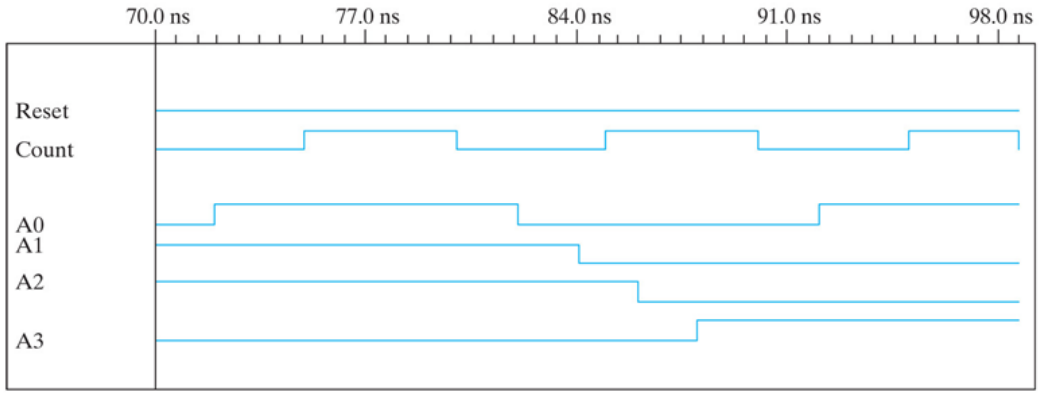
```
module t_Ripple_Counter_4bit;
   reg   Count;
   reg   Reset;
   wire A0, A1, A2, A3;

//Instantiate ripple counter
   Ripple_Counter_4bit M0 (A3, A2, A1, A0, Count, Reset);

   always
   #5 Count = ~Count;

   initial
      begin
         Count = 1'b0;
         Reset = 1'b1;
         #4 Reset = 1'b0;
      end

   initial  #200 $finish;
endmodule
```

```
`timescale 1ns / 100 ps
module Ripple_Counter_4bit (A3,A2,A1,A0, Count, Reset);
   output   A3, A2, A1, A0;
   input  Count, Reset;

//Instantiate complementing flip-flop
   Comp_D_flip_flop F0 (A0, Count, Reset);
   Comp_D_flip_flop F1 (A1, A0, Reset);
   Comp_D_flip_flop F2 (A2, A1, Reset);
   Comp_D_flip_flop F3 (A3, A2, Reset);
endmodule
```

- Simulation output of HDL Example 6.4:
  - p.306, Fig 6.19



(a) From 0 to 180 ns

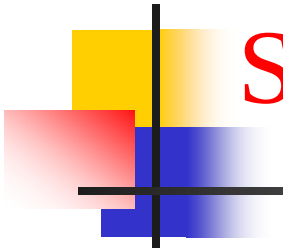


(b) From 70 to 98 ns

```
`timescale 1ns / 100 ps
module Ripple_Counter_4bit (A3,A2,A1,A0, Count, Reset);
   output   A3, A2, A1, A0;
   input  Count, Reset;

//Instantiate complementing flip-flop
   Comp_D_flip_flop F0 (A0, Count, Reset);
   Comp_D_flip_flop F1 (A1, A0, Reset);
   Comp_D_flip_flop F2 (A2, A1, Reset);
   Comp_D_flip_flop F3 (A3, A2, Reset);
endmodule
```

```
module Comp_D_flip_flop (Q, CLK, Reset);
   output   Q;
   input    CLK, Reset;
   reg      Q;

   always @ (negedge CLK, posedge Reset)
      if (Reset) Q <= 1'b0; else Q <= #2 ~Q;
endmodule
```

```
module t_Ripple_Counter_4bit;
   reg   Count;
   reg   Reset;
   wire A0, A1, A2, A3;

//Instantiate ripple counter
   Ripple_Counter_4bit M0 (A3, A2, A1, A0, Count, Reset);

   always
   #5 Count = ~Count;

   initial
      begin
         Count = 1'b0;
         Reset = 1'b1;
         #4 Reset = 1'b0;
      end

   initial  #200 $finish;
endmodule
```



Summary of HDL

- HDLs are extremely important tools for modern digital designers.
- HDLs are used for both *simulation* and *synthesis*.
- **Writing HDL code ⇒ describing real hardware!**
  1. Sketch a *block diagram* of your system.
  2. Identify which portions are *combinational logic* and which portions are *sequential circuits or FSMs*.
  3. Write HDL code for each portion, using the correct idioms to imply the kind of hardware needed.

Guidelines of Blocking and Nonblocking Assignments (p. HDL-89)

- Use *continuous assignments* **assign** to model simple *combinational* logic.
- Use **always @ (*)** and *blocking assignments* (=) to model more complicated *combinational* logic where the **always** statement is helpful.
- Use **always @ (posedge clk)** and *nonblocking assignments* (<=) to model synchronous *sequential* logic.
- Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

Mano 7-2

Random-Access Memory:
Memory Description in HDL

J.J. Shann

Memory Description in HDL

- Memory is modeled in the Verilog HDL by an array of registers:
  - is declared with a **reg** keyword, using a two-dimensional array
  - E.g.: a memory of 1024 words with 16 bits/word

    ⇒ a 2D array of 1024 registers, each containing 16 bits

```
reg[15: 0] memword [0: 1023];
```

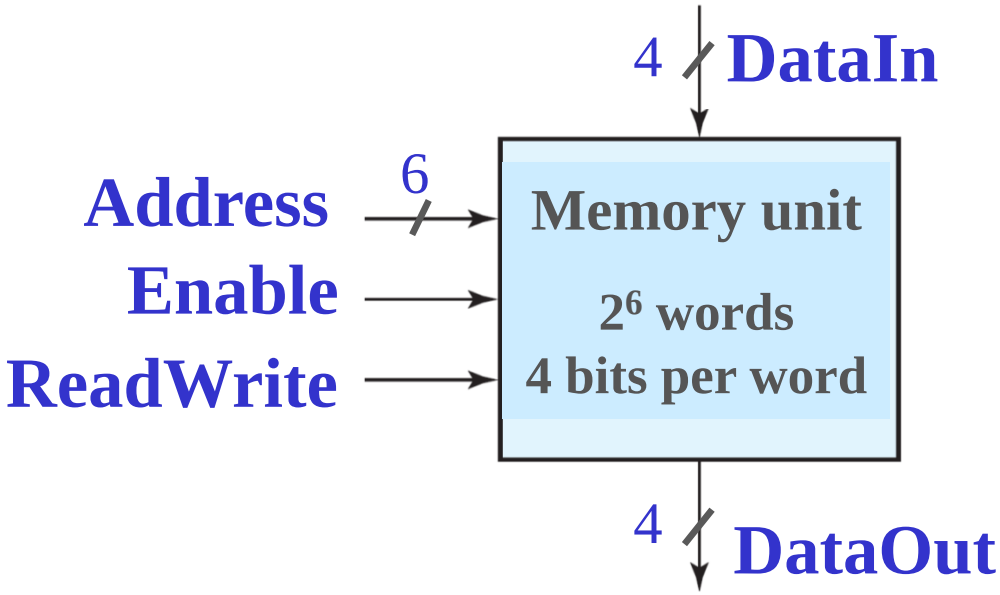
# of bits in a word
word length

# of words in memory
memory depth

* memword[512] refers to the 16-bit memory word at address 512.

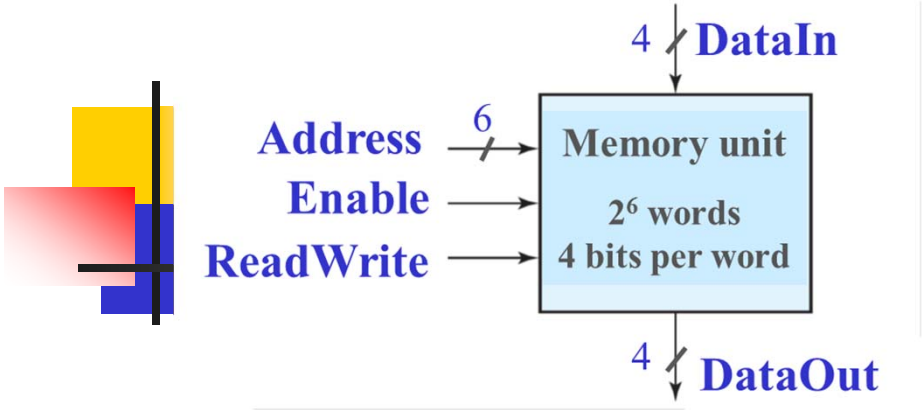
HDL Example 7.1: Operation of a Memory Unit

- HDL Example 7.1: Read and write operations of a memory with 64 words of 4 bits each



* The memory has three-state outputs.

Enable	ReadWrite	Memory operation
0	x	None (DataOut ← Hi-Z)
1	0	Write to selected word (Mem[Address] ← DataIn)
1	1	Read from selected word (DataOut ← Mem[Address])



Enable	ReadWrite	Memory operation
0	x	DataOut ← Hi-Z
1	0	Mem[Address] ← DataIn
1	1	DataOut ← Mem[Address]

```
module memory (Enable, ReadWrite, Address, DataIn, DataOut);
   input            Enable, ReadWrite;
   input   [3: 0]   DataIn;
   input   [5: 0]   Address;
   output  [3:0]    DataOut;
   reg     [3: 0]   DataOut;
   reg     [3: 0]   Mem [0: 63];                    //64 x 4 memory

   always @ (Enable or ReadWrite)
      if (Enable)
         if (ReadWrite) DataOut = Mem[Address];     //Read
         else Mem[Address] = DataIn;                //Write
      else DataOut = 4'bz;                          //High impedance state
endmodule
```