Chapter 2

Introduction to Verilog

J.J. Shann

Chapter Overview (1/2)

Chapter Overview (2/2)

2-11 Verilog Data Types and Operators
2-12 Simple Synthesis Examples
2-13 Verilog Models for Multiplexers
2-14 Modeling Registers and Counters Using Verilog Always Statements
2-15 Behavioral and Structural Verilog
2-16 Constants
2-17 Arrays
2-18 Loops in Verilog
2-19 Testing a Verilog Model (Test bench)
2-20 A Few Things to Remember

Exercises in Textbook

Sections	Exercises
§2-2	2.60
§2-3	2.61, 2.62
§2-4	2.63, 2.64
§2-6	2.65
§2-7	2.2~2.5, 2.42
§2-8	2.7, 2.22
§2-9	2.1, 2.9, 2.10, 2.16~2.20, 2.24
§2-10	2.8, 2.40
§2-11	2.15, 2.21, 2.23, 2.25, 2.26, 2.28, 2.30, 2.54, 2.56, 2.58
§2-12	2.12~2.14, 2.27, 2.29, 2.31
§2-13	2.6, 2.11, 2.50
§2-14	2.33, 2.35, 2.36, 2.38, 2.51~2.53, 2.55, 2.57, 2.59
§2-15	2.32, 2.34, 2.37, 2.39, 2.41
§2-17	2.43, 2.44
§2-18	2.45
§2-19	2.46~2.49

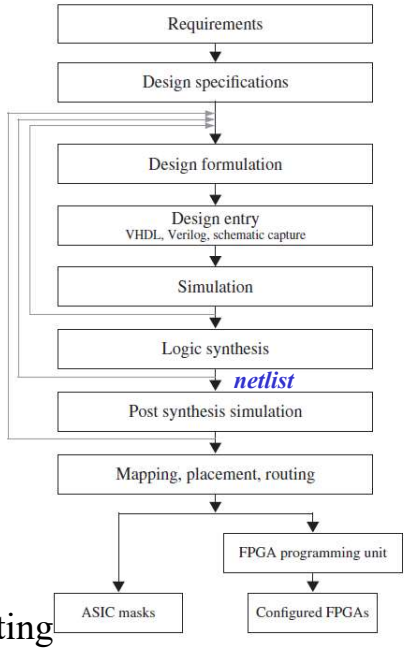
2-1

Computer-Aided Design (CAD)

J.J. Shann

CAD

- Steps in modern digital system design:
  - Design requirement
  - Design specification
  - Design formulation
  - Design entry
  - Simulation
  - Logic synthesis
  - Post synthesis simulation
  - Mapping, placement, routing



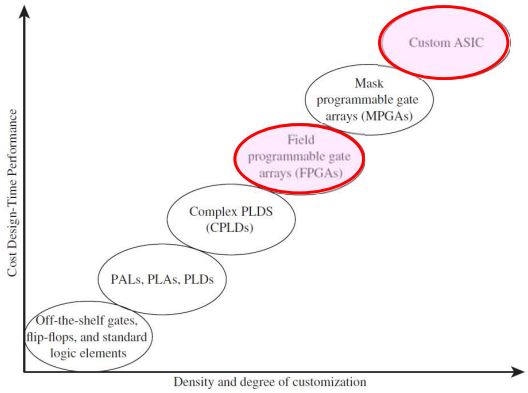
2-7

Hardware Description Language

- Hardware description languages (HDL):
  - allows a digital system to be designed and debugged at a ***higher level of abstraction*** than *schematic capture*.
  - **Behavioral** description
    - Specifies only the general working of the design at a *flow-chart* or *algorithmic level*.
  - **Structural** description
    - Specifies *components* or *specific implementations of components* associated w/ the design.

Design Technologies

- Target technologies:



  - Most common:
    - FPGAs: field programmable gate arrays
    - ASICs: application-specific integrated circuits

2-2

Hardware Description Languages

J.J. Shann

Hardware Description Languages (HDLs)

- HDLs:
  - can describe a digital system at several different levels: ***behavioral***, ***data flow***, ***structural***
  - lead naturally to a **top-down** design methodology
- Two popular HDLs: VHDL, Verilog
  - VHDL:
    - Slightly more elegant for simulation of very large systems at a higher level of abstraction
  - Verilog: (v)
    - slightly more supportive for synthesis
    - syntactic similarity to C

2-11

Verilog

- Some history on Verilog:
  - Gateway Design Automation: 1984
  - Cadence: 1990
    - make Verilog an open architecture
  - IEEE standard: 1995; revised in 2001, 2005
- Purposes of Verilog:
  - describe, document, simulate, and automatically generate hardware

  * Syntactic roots: C

  * Has statements that execute concurrently!

Learning Verilog

- Process of learning Verilog:
  - Alphabet
  - Vocabulary or lexical elements of the language
    - Identifiers, reserved words, special symbols, and literals
  - Syntax (grammar and rules)
  - Semantics (meaning of descriptions)

HDL vs. Ordinary Programming Language

- Differences b/t Verilog & ordinary programming language:
  - Verilog has statements that execute concurrently to model real hardware in which the components are all in operation at the same time.
  - The constructs of Verilog are tailored for the purposes of describing, documenting, simulating, and automatically generating hardware.

2-3

Verilog Description of Combinational Circuits

Verilog Description of Combinational Circuits

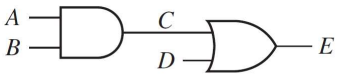
- ***Concurrent statements*: *continuous assignments***
  - Statements that are always ready to execute.
  - Evaluated any time and every time a signal on the right side of the statement changes.
  - Execute repeatedly as if they were in a loop.
- Form of a signal assignment statement:

  assign [#delay] signal_name = expression**;**

  - Square brackets "[...]" indicate that #delay is optional.

2-17

Example: Concurrent statements



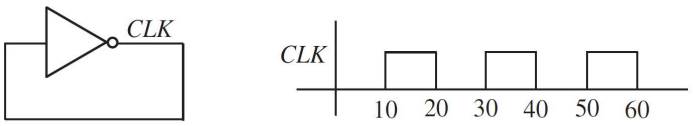

```
assign #5 C = A && B;
assign #5 E = C || D;
```

- The **assign** statement is used to assign a value.
- A, B, C, D, E: are signals in a Verilog physical system
- "**&&**": represents the AND gate
- "||": represents the OR gate
- #5: indicates a delay symbol of 5 ns.
- "=": the signal assignment operator, indicates that the value computed on the right side is assigned to the signal on the left side

2-18

Example: Inverter w/ Feedback

- E.g.: **assign** #10 CLK = ~CLK;



- E.g.: **assign** CLK = ~CLK;
  - Time will never advance to 1 ns.

Some Verilog Syntax (Rules)

- Verilog is *case sensitive*.
  - **assign** #10 Clk = ~Clk;
    assign #10 CLK = ~CLK;
    ⇒ Result in two different clocks.
- Verilog signal names and identifiers:
  - may contain ***letters***, ***numbers***, the ***underscore character*** (_), and the ***dollar sign*** ($).
  - must start w/ a ***letter*** or ***underscore*** character, and cannot start w/ a number or a $ sign.
  - The dollar sign ($) is reserved as the 1st character for ***system tasks***.

- Every Verilog statement must be terminated with a semicolon (;).
- Spaces, tabs, carriage returns treated the same way.
- Comments: // , /* … */
- Reserved words (keywords):
  - **and**, **or**, **always**, …
- Numbers or net values are represented as:
  <number_of_bits> '<base> <value>
  - base: ***b***, ***d***, or ***h*** for binary, decimal, or hex

2-21

Example: Identifiers

- Egs of valid identifiers:

  adder
  Mux_input
  _error_code
  Index_bit
  vector_sz
  _$five
  Count
  XOR

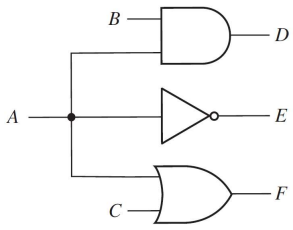
  * An ***identifier*** must start with a ***letter*** or ***underscore*** character, and it cannot start with a number or a $ sign.

- Egs of invalid identifiers:

  4bitadder
  $error_code
  xor (a keyword)

2-22

Example: Comments




```
// when A changes, these concurrent
// statements all execute at the
// same time
  assign #2 D = A && B;
  assign #1 E = ~A;
  assign #3 F = A || C;
```

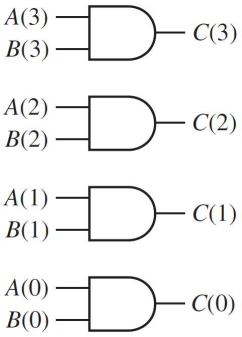
Shann 2-23

Data Types

- Data types: **reg**, **wire**, ...
  - **reg** (register): is used to store value
  - **wire**: cannot hold a value; is used to link modules of combinational logic
- Data structures: ***arrays***, ***memory arrays***, ***vectors***
  - Array: a collection of objects w/ the same attributes
  - Vector: a one-dimensional array of bit signals
  - E.g.: declare a multiple bit wire ⇒ **wire** B[3:0]

2-24

Example: Vectors



&: bit-wised AND (binary operator)
&&: logical AND

```
// the hard way
assign C[3] = A[3] && B[3];
assign C[2] = A[2] && B[2];
assign C[1] = A[1] && B[1];
assign C[0] = A[0] && B[0];
```

```
// the easy way assuming C, A and
// B are 4-bit vectors
assign C = A & B;
```


Built-in Primitives (§8-4)

- ***Built-in primitives***: provide the ***gate***- and ***switch***-level modeling facility
  - predefined ***logic gate*** primitives: 12
  - predefined ***switch*** primitives: 14
- Advs. of modeling at gate/switch level:
  - Synthesis tools can easily map it to the desired circuitry since gates provide a very close one-to-one mapping b/t the intended ckt and the model.
  - There are primitives, e.g., the bidirectional transfer gate, that cannot be otherwise modeled using continuous assignments.

Gate-level Primitives

* Output port(s) first followed by input(s).

Gate-level Primitive (12)	Type
and, **nand**, **or**, **nor**, **xor**, **xnor**	n-input gates (1 output)
buf, **not**	n-output gates (1 input)
bufif0, **bufif1**, **notif0**, **notif1**	Three-state gates

8-27

Gate-level Primitive (12)	Type
and, **nand**, **or**, **nor**, **xor**, **xnor**	n-input gates (1 output)
buf, **not**	n-output gates (1 input)
bufif0, **bufif1**, **notif0**, **notif1**	Three-state gates

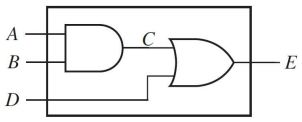
- **and**, **or**: ***one output*** and multiple inputs
  - The ***output*** must be listed first followed by ***inputs***

    and (out, in_1, in_2, …, in_n);
- **not**, **buf**: ***one input*** and multiple outputs
  - The ***output(s)*** must be listed first followed by ***input***

    not (out_1, out_2, … , iout_n, in);
- **bufif0**, **bufif1**, **notif0**, **notif1**: tristate gates
  - **bufif0**/**bufif1**: non-inverting buffer primitive w/ active-low/high control input
  - **notif0**/**notif1**: inverting buffer primitive w/ active-low/high control input
  - The ***output(s)*** must be listed first followed by ***input*** and finally the ***tristate control input***.

J.J. Shann 8-28

Example



- Verilog code using ***concurrent statements***:

  assign C = A && B;

  assign E = C || D;

- Verilog code using ***built-in primitives*** (**and**, **or**):

  or (E,C,D);

  and (C,A,B);

* Output port first followed by inputs.

2-4

Verilog Modules

Verilog Modules

- Module:
  - a basic building block that declares the input and output signals and specifies the internal operation of the module.
  - A module starts with the keyword **module** followed by an optional ***module name*** and an optional ***port list***.
  - The key word **endmodule** ends a module.
  - Each module declaration includes a ***list of interface signals*** that can be used to connect to other modules or to the outside world.

2-31

Module Declaration

- Form of a module declaration:

port list

```
module module-name (module interface list);
[list-of-interface-ports]
...
[port-declarations]
...
[functional-specification-of-module]
...
endmodule
```

  - Items enclosed in square brackets ([ ]) are optional.

2-32

module module-name (module interface list);
[list-of-interface-ports]
[port-declarations]
[functional-specification-of-module]
endmodule

- Form of the ***list-of-interface-ports***:

  type-of-port list-of-interface-signals

  {; type-of-port list-of-interface-signals};

  - curly brackets { }: indicate zero or more repetitions of the enclosed clause
  - ***type-of-port***: **input**, **output**, **inout**
    - specify the direction of port signals
  - Can be combined w/ the ***module interface list***.
- ***port-declarations*** section:
  - declare ***internal signals*** that are used within the module.

2-33

Types of Interface Ports (input, output, inout)

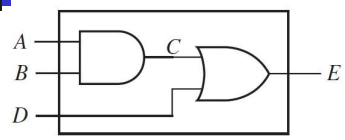
- input
- output
  - An **output** port can also be used as an input in a statement inside the same module.
  - E.g.:

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

   assign #5 D = A || B;
   assign #5 E = C || D; // uses D as an input

endmodule
```

- **inout:** used for a variable if it has to be used as input and output by other modules

J.J. Shann 2-34

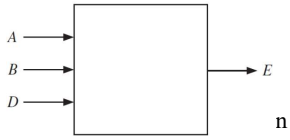
Example



module module-name (module interface list);
[list-of-interface-ports]
...
[port-declarations]
...
[functional-specification-of-module]
...
endmodule

```
module two_gates (A, B, D, E);
  output E;
  input A, B, D;

  wire C;   // an internal signal

  //concurrent statements
  assign C = A && B;
  assign E = C || D;

endmodule
```

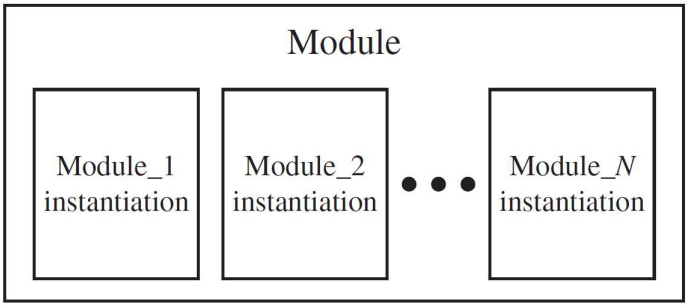
* Module I/O declaration part can be considered as the black box picture of the module being designed and its external interface. ⇒ It represents the interconnections from this module to the external world.



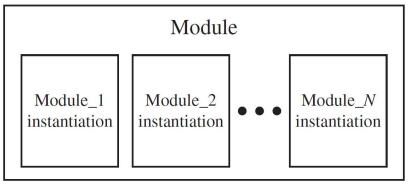
n 2-36

Verilog Program Structure

- Verilog program structure:
  - A module may contain other ***module instances*** that describe the operation of the module.



J.J. Shann 2-38



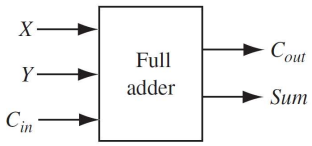
- Module instantiation:
  - ***Positional association***: listing the ports in order
    - While instantiating a module, the order of the signals in the port map is the same as the order of the signals in the port of the module declaration of the instantiated module.
  - ***Named association*** (Ch8):
    - signals can be in any order as long as the signals in the module are connected to the ports by name.

Example: Full Adder

The logic equations for the full adder:

$$Sum = X \oplus Y \oplus C_{in}$$

$$C_{out} = XY + YC_{in} + Xc_{in}$$

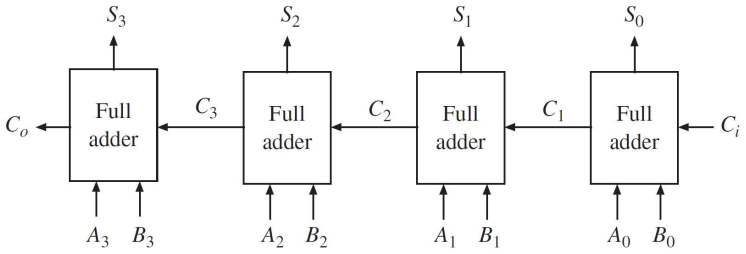


The Verilog code for the full adder:

```
module FullAdder(X, Y, Cin, Cout, Sum);
   output Cout, Sum;
   input X, Y, Cin;

   assign #10 Sum = X ^ Y ^ Cin;
   assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);

endmodule
```

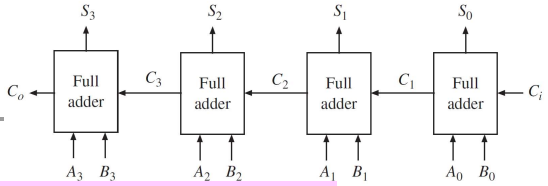
Example: 4-bit Binary Adder

- Use the FullAdder module to form a 4-bit binary adder:



1. Declare the 4-bit adder as another module, Adder4.
2. Instantiate four FullAdder modules within Adder4.

<Ans.>




```
module Adder4 (S, Co, A, B, Ci);
   output [3:0] S;  // 4-bit vector
   output Co;
   input [3:0] A, B;
   input Ci;

   wire [3:1] C; // an internal signal

   // instantiate four copies of the FullAdder
   //                  X    Y    Cin  Cout  Sum
   FullAdder FA0 (A[0], B[0],  Ci,  C[1], S[0]);
   FullAdder FA1 (A[1], B[1], C[1], C[2], S[1]);
   FullAdder FA2 (A[2], B[2], C[2], C[3], S[2]);
   FullAdder FA3 (A[3], B[3], C[3],  Co,  S[3]);

endmodule
```

```
module FullAdder(X, Y, Cin, Cout, Sum);
   output Cout, Sum;
   input X, Y, Cin;
   …
endmodule
```

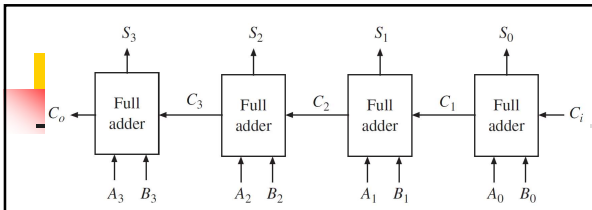
* Position association

Simulator

- **ModelSim** Verilog simulator:
  - from Mentor Graphics
- ***Simulator command file***: *do file*
  - E.g.: simulator commands used to test Adder4

```
add list A B Co C Ci S      //put signals on the output list
force A 1111                // set the A inputs to 1111
force B 0001                // set the B inputs to 0001
force Ci 1                  // set the Ci inputs to 1
run 50 ns                   //run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

```
module Adder4 (S, Co, A, B, Ci);
    output [3:0] S;
    output Co;
    input [3:0] A, B;
    input Ci;
    wire [3:1] C;
    …
endmodule
```


```
module Adder4 (S, Co, A, B, Ci);
    output [3:0] S;
    output Co;
    input [3:0] A, B;
    input Ci;
    wire [3:1] C;
    …
endmodule
```

```
add list A B Co C Ci S      //put signals on the output list
force A 1111                // set the A inputs to 1111
force B 0001                // set the B inputs to 0001
force Ci 1                  // set the Ci inputs to 1
run 50 ns                   //run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```


```
module FullAdder(X, Y, Cin, Cout, Sum);
    output Cout, Sum;
    input X, Y, Cin;
    assign #10 Sum = X ^ Y ^ Cin;
    assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);
endmodule
```

Example: Simulation Results

```
add list A B Co C Ci S
force A 1111
force B 0001
force Ci 1
run 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

Simulation results for the command list used to test Adder4:

ns	delta	a	b	co	c	ci	s
0	+0	1111	0001	x	xxx	1	xxxx
10	+0	1111	0001	x	xx1	1	xxx1
20	+0	1111	0001	x	x11	1	xx01
30	+0	1111	0001	x	111	1	x001
40	+0	1111	0001	1	111	1	0001
50	+0	0101	1110	1	111	0	0001
60	+0	0101	1110	1	110	0	0101
70	+0	0101	1110	1	100	0	0111
80	+0	0101	1110	1	100	0	0011

2-5 & 2-6

Verilog Assignments

Verilog Assignments

- Two types of assignments in Verilog:
  - ***Continuous assignments***: used to assign values for ***comb logic*** ckts (§2-3 & §2-5)
    - Model constant reaction to input changes.
  - ***Procedural assignments***: may be used to model ***registers*** and ***finite state machines*** (§2-6)
    - Model selective activity conditional on clock, edge-triggered devices, sequence of operations, etc.

A. Continuous Assignments

- Continuous assignments: used to assign values for ***comb*** logic circuits.
  - Model constant reaction to input changes.
- Two types of continuous assignments:
  - ***Explicit*** continuous assignments: uses the **assign** keyword after the net is separately declared.
    - E.g.: **wire** C;
      assign C = A || B;
  - ***Implicit*** continuous assignments: assign the value in declaration w/o using the **assign** keyword.
    - E.g.: **wire** D = E && F;

B. Procedural Assignments

- Procedural assignments: may be used to model ***registers*** and ***finite state machines***
  - Model selective activity conditional on clock, edge-triggered devices, sequence of operations, etc.
- Two types of procedural assignments:
  - **initial** blocks: execute ***only once at time zero***
    - are useful in *simulation* and *verification*.
  - **always** blocks: ***loop*** to execute over and over again when a ***specific event*** occurs, ***starts at time zero***
    - Only always blocks are *synthesized*

2-52

Initial Statements

- Basic form of an initial statement:

```
initial
begin
   sequential-statements
end
```

2-53

Always Statements

- Basic form of an always statement:

```
always @(sensitivity-list)
begin
    sequential-statements
end
```

  - The process executes whenever any signal in the *sensitivity list* changes.
  - The statements b/t **begin** and **end** of the **always** are executed *sequentially* rather than concurrently.
  - When a process finishes executing, it goes back to the beginning and waits for a signal on the sensitivity list to change again.

2-54

Sensitivity List

```
always @(sensitivity-list)
begin
    sequential-statements
end
```

- Sensitivity list:
  - An **always** block will be activated if one of the events occurs. (Events are separated by "**or**" or "**,**".)
  - ***Comb* always *blocks***: list includes **all signals** that are used in the condition statement and all signals on the right-hand side of the assignment

    ⇒ **always @***
  - ***Seq* always *blocks***: list contains three kinds of **edge-triggered events-- clock**, **reset**, and **set** signal event
  - If sensitivity list is omitted at the **always** keyword, ***delays*** or ***time-controlled events*** must be specified inside the **always** block. (§2-8)

2-57

Sequential Statement

```
always @(sensitivity-list)
begin
  sequential-statements
end
```

- 2 different ways for evaluating sequential statements:
  - ***Blocking assignment***: must complete the evaluation of the right-hand side of a statement before the next statements in a sequential block are executed.
    - Uses the "=" operator
  - ***Non-blocking assignment***: allows several assignment evaluations to be ***assessed at the same time***.
    - Uses the "<=" operator

* The variables on the left-hand side element of an = or <= in an **always** block should be defined as **reg** data type.

2-58

Examples

- E.g.:

```
always @(A, B, D)
begin
   C = A && B;   // Blocking operator is used
   E = C || D;   // Statements execute sequentially
end
```

- E.g.:

```
always @(A, B, D)
begin
   C <= A && B;  // non-blocking operator is used
   E <= C || D;  // Statements execute simultaneously
end
```

J.J. Shann 2-59

Note

- **always** statements can be used for modeling *comb* logic and *seq* logic.
  - For ***comb*** logic:
    - Use the **always @*** statement. ⇐ If any of the input signals are accidentally omitted from the sensitivity list, there can be mismatches b/t synthesis and simulation and a lot of confusion.
    - * It is not necessary to be modeled by **always** statements.
  - For ***seq*** logic:
    - * It is required to be modeled by **always** statements.

Example: Seq Logic w/ Blocking & Non-blocking Assignments

```
module sequential_module (A, B, C, D, clk);
   input clk;
   output A, B, C, D;

   reg A, B, C, D;

   always @(posedge clk)
   begin
      A = B;        // blocking statement 1
      B = A;        // blocking statement 2
   end

   always @(posedge clk)
   begin
      C <= D;       // non-blocking statement 1
      D <= C;       // non-blocking statement 2
   end

endmodule
```

* **reg** is the only legal type on the left-hand side element of an = or <= in an always block.

* **posedge**: a keyword is used for an edge-triggered functionality in the sensitivity list

* **Discussion:**
  Difference b/t these two always blocks?

Example: Comb Logic w/ Blocking Assignments in an Always Block

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

   reg D, E;

   always @(*)
   begin
      #5 D = A || B; // blocking statement 1
      #5 E = C || D; // blocking statement 2
   end

endmodule
```

* If **inout** is used for D, there will have compiler error, since D is of **reg** type.

* **input** and **inout** ports can be only net (or **wire**) data type.

Brief Summary

- Writing ***synthesizable*** code:
  - Use ***non-blocking assignments*** "<=" in **always** blocks intended to create ***seq logic***
  - Use ***blocking operator*** "=" in **always** blocks intended to create ***comb logic***.
  - **Do not** mix blocking and non-blocking assignments in the same **always** block.

  ⇒ When each **always** block is written, think whether you want seq logic or comb logic carefully.

Verilog Data Value Set

- Default Verilog HDL data value set:
  a ***4-value system*** (Ch8)
  - **0**: represents a ***logic zero***, or a ***false*** condition
  - **1**: represents a ***logic one***, or a ***true*** condition
  - ***x***: represents an ***unknown*** logic value
  - ***z***: represents a ***high-impedance*** state (a tristated value)

Commonly-Used Verilog Data Types

- Two commonly-used Verilog data types:
  - **wire**: acts as real wires in ckt designs
  - **reg**: is similar to wires, but can store information just like registers
  - Declarations for **wire** and **reg** signals: should be done inside a module but outside any **initial** or **always** block.

Wire

* **input** and **inout** ports can be only net (or **wire**) data type.

- **wire**:
  - Acts as real wires in ckt designs.
  - Cannot store information.
  - Initial value is *z* (high impedance).
  - Either single bit or multiple bits.
  - Can be used in modeling ***comb logic only*** and must be driven by something.
    - Can be used on the left-hand side of an **assign** statement but cannot be used on the left-hand side of "=" or "<=" in an **always** @ block.

2-66

Reg

- **reg**:
  - are used to store values, i.e., the assigned data needs to be stored until the next assignment.
  - Initial value is ***x*** (unknown).
  - Can be used in modeling ***comb*** and ***sequ*** logic.
  - is the only legal type to be used on the left-hand side of "=" or "<=" in an **always** @ block or **initial** block but cannot be used on the left side of an **assign** statement.

2-67

Guidelines of Blocking & Nonblocking Assignments

- Use *continuous assignments* for simple ***comb*** logic.
- Use **always @ (*)** and *block* to model more complicated (the **always** statement is help
- Use **always @ (posedge clk)** *assignments* (<=) to model s
* Do not mix blocking and no assignments in the same **alw**
* Do not make assignments to more than one **always** stater 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
```


2-7

Modeling Flip-Flops Using Always Block

J.J. Shann

Modeling Flip-Flops Using Always Block

- Modeling flip-flops using **always** block:
  - Flip-flop can change state either on the ***rising*** or the ***falling*** edge of the clock input
  - Expressions **posedge** or **negedge** are used to accomplish the functionality of an edge-triggered device.
- Types of sequential statements:
  - signal assignment statements
  - **if** statements
  - ...

2-70

If Statement

- **if** statements:
  - are sequential statements that can be used within an **always** or **initial** block
  - Can not be used as concurrent statements outside of an **always/initial** block.
- Form of basic **if** statement:

```
if (condition)
   sequential statements1
else
   sequential statements2
```

  - The condition is a Boolean expression that evaluates to TRUE or FALSE.

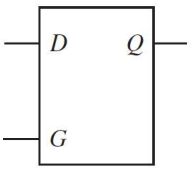
2-71

- General **if** statement has the form:

if (condition)
  sequential statements
  //0 or more else if clauses may be included
{**else if** (condition)
  sequential statements}
[**else** sequential statements] // else clause is optional

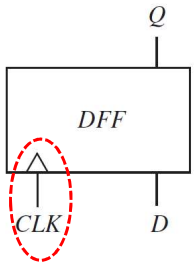
  - Curly brackets "{ }": indicate that any # of the clauses may be included
  - Square brackets "[ ]": indicate that the clause is optional

Example: Transparent Latch



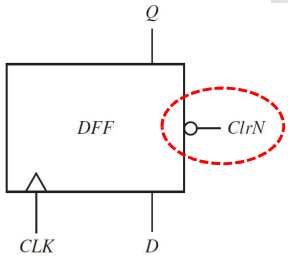

```
always @ (G or D)
begin
 if (G)
   Q <= D;
end
```

Example: Simple D Flip-Flop




```
always @ (posedge CLK)
begin
   Q <= D;
end
```

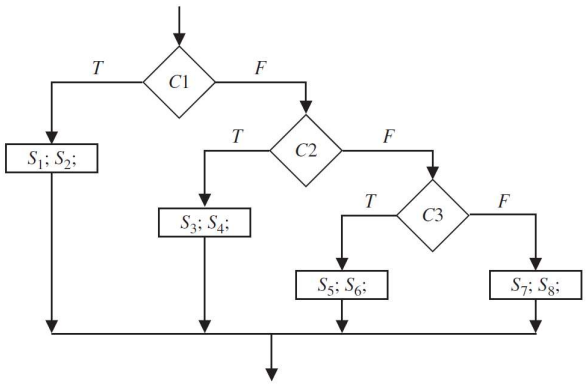

Example: D Flip-Flop w/ Async Clear




```
always @ (posedge CLK or negedge ClrN)
begin
   if (~ClrN)
     Q <= 0;
   else
     Q <= D;
end
```

Example: Nested Ifs and Else Ifs

- Flowchart depiction:

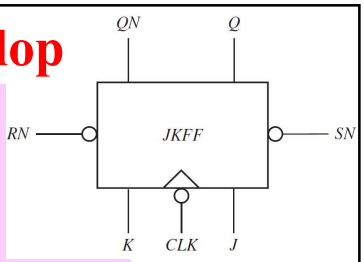



```
if (C1)
begin
  S1; S2;
end
else if (C2)
begin
  S3; S4;
end
else if (C3)
begin
  S5; S6;
end
else
begin
  S7; S8;
end
```

2-76

Example: J-K Flip-Flop

```
module JKFF (SN, RN, J, K, CLK, Q, QN);
   input SN, RN, J, K, CLK;
   output Q, QN;

   reg Qint;

   always @(negedge CLK or RN or SN)
   begin
      if (~RN)
         #8 Qint <= 0;
      else if (~SN)
         #8 Qint <= 1;
      else
         Qint <= #10 ((J && ~Qint) || (~K && Qint));
   end

   assign Q = Qint;
   assign QN = ~Qint;

endmodule
```



$Q^+ = JQ' + K'Q$

J.J. Shann 2-77

2-8

Always Blocks Using Event Control Statements

J.J. Shann

Always Blocks Using Event Control Statements

- Always blocks using event control statements:
  - If a sensitivity list is omitted, ***delays*** or ***time-controlled events*** must be specified inside the block.
  - E.g.:
```
always
begin
    #10 clk <= ~clk;
end
```
  - ***Time-controlled events***: **wait** or event control statements

* An **always** block cannot have both sensitivity list and **wait** statements.

2-79

Always Blocks with Wait Statement

- Form of an **always** block w/ **wait** statements:

```
always
begin
   sequential-statements
   wait-statement (boolean-expression)
   sequential-statements
   wait-statement (boolean-expression)
   . . .
end
```


Wait Statement

- **wait** statement: is used as a ***level-sensitive*** event control
- General syntax of the **wait** statement:

  wait (Boolean-expression)

  - A procedural statement waits when the Boolean expression is FALSE.
    - The logic values 0, "x", and "z" are treated as FALSE.
  - When the expression is TRUE, the statement is executed.
    - Logic 1 is TRUE.

Examples

- E.g.:

```
always
begin
    rst = 1; // sequential statements
    wait (posedge CLK); //wait until posedge CLK
    // more sequential statements
end
```

- E.g.:

```
always
begin
    wait (WR)
        MEM = DATA_IN;
    wait (~WR)
        DATA_OUT = MEM;
end
```


Examples: p.83 & p.84

- For a half adder (HA):

  $sum = x \text{ XOR } y, \quad carry = x \text{ AND } y$

- Assume that the HA must add if *add* signal = 1.
- What is wrong w/ each of the following codes?

  (a) It will not even compile.

  (b) It will compile but not simulate correctly.

  (c) It will compile and simulate correctly but not synthesize correctly.

  (d) It will work correctly in simulation and synthesis.

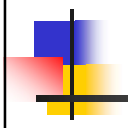
```
always @(*)
begin
if (add == 1)
    sum = x ^ y;
    carry = x & y;
end
```

```
always @(*)
begin
if (add == 1)
    sum = x ^ y;
    carry = x & y;
else
    sum = 0;
    carry = 0;
end
```

<Ans.>

- (a) This code will compile but will not simulate correctly.
  - Corrections:
    - Add **begin** and **end** for the **if** statement.
    - * ***Avoiding unwanted latches***:

      Latches can be avoided (i) by adding **else** clause or (ii) by initializing *sum* and *carry* to 0 at the beginning of the **always** statement. (§2-10)
- (d) This code will not even compile.
  - Corrections: Add **begin** and **end** for both of the **if** and **else** clauses.



2-9

Delays in Verilog

J.J. Shann

Delays in Verilog

- ***Inertial delay***
  - models ***gates*** and other ***devices*** that do not propagate ***short pulses*** from the input to the output.
- ***Transport delay***
  - models the delay introduced by ***wiring***
  - it simply delays an input signal by the specified delay time.
- ***Net delay***
  - the time it takes from any ***driver*** on the net to change value to the time when the net value is updated and propagated further

Initial Delay

- ***Inertial delay***:
  - models ***gates*** and other ***devices*** that do not propagate ***short pulses*** from the input to the output.
  - If a gate has an ideal inertial delay *T*, in addition to delaying the input signals by time *T*, any pulse w/ a width less than *T* is rejected.

Examples: Initial Delay for Combinational Blocks

- 3 ways to express ***inertial delay*** for comb blocks:

```
// explicit continuous assignment
wire D;
assign #5 D = A && B;  //inertial delay is 5 time units

// implicit continuous assignment
wire #5 D = A && B;  //inertial delay is 5 time units

// net declaration
wire #5 D;
assign D = A && B;  //inertial delay is 5 time units
```


Transport Delay

- ***Transport delay***:
  - models the delay introduced by ***wiring***.
  - It simply delays an input signal by the specified delay time.
  - ***Intra-assignment delay*** : The delay value is specified on the right-hand side of the statement

*** Can not be done w/ continuous assign statements**

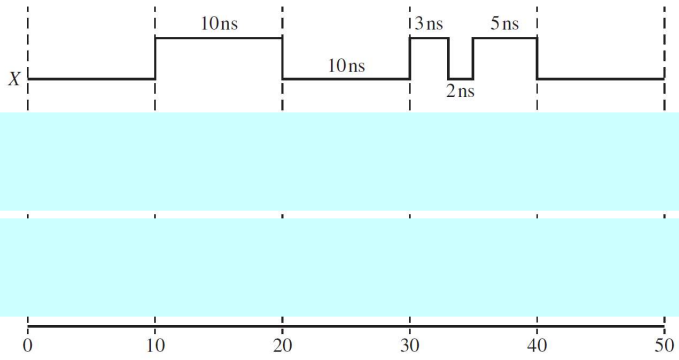
- E.g.: **assign** a = #10 b; // × compile-time error

Example: Initial and Transport Delays

```
always @ (X)
begin
   Z1 <=#10 (X); // transport delay
end
assign #10 Z2 = X; // inertial delay
```

* ***Delay assignment***: The expression on the right hand side is evaluated but not assigned to Z_1 until the delay has elapsed.

* ***Delay evaluation***: The delay elapses first and then the expression is evaluated and assigned to Z_2.



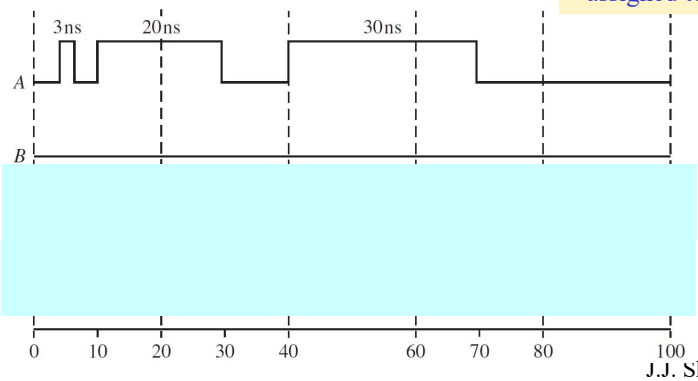
Net Delay

- ***Net delay***: the time it takes from any ***driver*** on the net to change value to the time when the ***net value*** is updated and propagated further.

Example: Net Delays (p.86)

```
wire C1;
wire #10 C2;  //net delay on wire C2
assign #30 C1 = A || B;  //inertial delay
assign #20 C2 = A || B;  //inertial delay
```

* Added to net delay before being assigned to wire C2



Notice

- These delays are relevant only for ***simulation***.
- The **pulse rejection** associated w/ ***inertial delay*** can inhibit many output changes.
- In simulation w/ basic gates and simple ckts, one should make sure that test sequences that you apply are wider than the ***inertial delays*** of the modeled devices.

* The focus of this book is ***synthesizable Verilog*** where all **specified delays are ignored**.

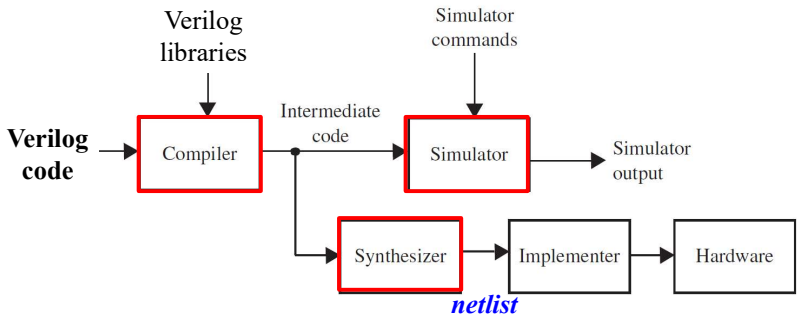
2-10

Compilation, Simulation, and Synthesis of Verilog Code

J.J. Shann

Compilation, Simulation, and Synthesis of Verilog Code

- Compilation, Simulation, and Synthesis of Verilog Code:

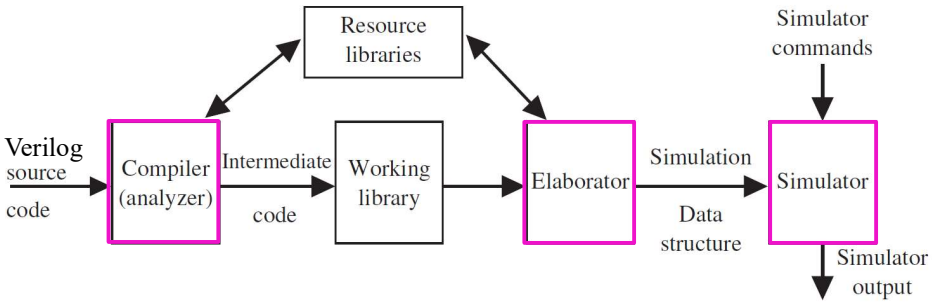


  - ***netlist*** : a list of required ***components*** and their ***interconnections***

2-100

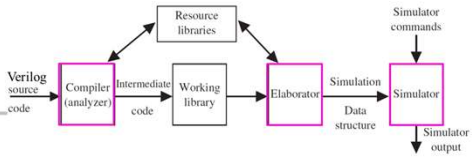
Simulation of Verilog Code

- 3 phases in the simulation of Verilog code:
  - analysis (compilation), elaboration, & simulation



2-101

Simulation Phases



- Analysis (compilation):
  - Checks source code; if a syntax or semantic error occurs, then the compiler gives error message.
  - Checks references to libraries.
- Elaboration:
  - creates a hierarchy of module instances, propagates parameters among modules, makes drivers, and constructs a design hierarchy.
- Simulation: ***discrete event simulation***
  - consists of an initialization phase and actual simulation.
  - ***Initialization phase***: gives an initial value to the signal
  - ***Simulation***: Events are kept on an event queue, ordered by simulation time.

2-102

Verilog Event Queue

- Five regions of the Verilog event queue:
  1. ***Active event region***
  2. ***Inactive event region***
  3. ***Non-blocking assign update region***
  4. ***Monitor event region***
  5. ***Future event region***

current simulation time

future simulation time

2-103

Scheduling Convention

Regions of Verilog event queue:
- Active event region
- Inactive event region
- Non-blocking assign update region
- Monitor event region
- Future event region

- Adding events to various queue regions for each type of statements:
  1. ***Continuous assignment***: evaluate RHS and add to active region as an ***active update event***
  2. ***Procedural continuous assign***: evaluate RHS and add to active region as an ***active update event***
  3. ***Blocking assignment with no delay***: compute RHS and put into inactive region as an ***inactive event*** for current time
  4. ***Blocking assignment with delay***: compute RHS and put into future event region for time after delay
  5. ***Non-blocking assignment with no delay***: compute RHS and schedule as ***non-blocking assign update event*** for current time
  6. **Non-blocking assignment with delay**: compute RHS and schedule as ***non-blocking assign update event*** for future time
  7. **$monitor** / **$strobe** system tasks: create ***monitor events*** which are continuously reenabled in every successive time step.

2-105

- Actions for each ***simulation cycle***:
  1. Process all active update events.
  2. Activate all inactive events for that time.
  3. Activate all non-blocking assign update events and process them.
  4. Activate all monitor events and process them.
  5. Advance time to the next event time and repeat from step 1.

* All of these 5 steps happen at the same time (one simulation cycle), but the events occur in the order *active*, *inactive*, *non-blocking update*, and *monitor* events.

Scheduling of Nonblocking (<=) Assignment

- Scheduling of simulator for "<=":
  - Whenever a component input changes, the output is scheduled to change after ***the specified delay*** or after ***an infinitesimal delay*** (***Δ delay***) if no delay is specified.
  - If two non-blocking updates are made to the same variable in the same time step, the 2nd one dominates by the end of the time step.

Examples: p.90, Fig 2-26

- (a) determinate

```
module determinate;
    reg a;
    initial a = 0;
    always begin
        a <= #5 0;
        a <= #5 1;
    end
endmodule
```

* The assigned value of *a* is deterministic because of ordering from **begin** to **end**.

- (b) nondetrminate

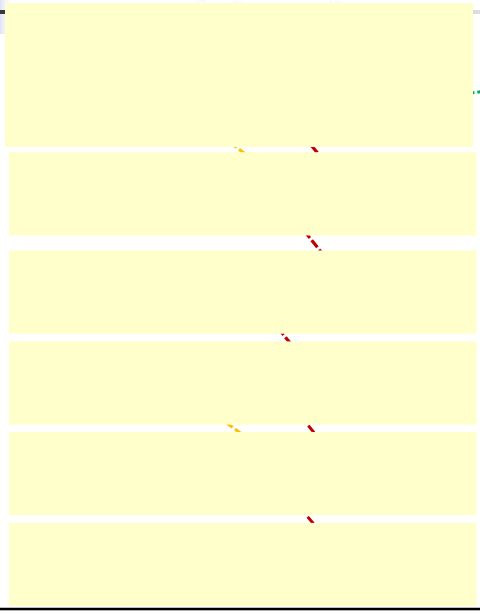
```
module nondeterminate;
    reg a;
    initial a = 0;
    always a <= #5 0;
    always a <= #5 1;
endmodule
```

* The assigned value of *a* is non-deterministic.

Simulation w/ Multiple Processes (Initial or Always Blocks)

- If a model contains more than one process (**initial**/**always** block), all processes execute concurrently w/ other processes.
- If there are ***concurrent statements*** outside **always** statements, they also execute concurrently.
- Statements inside of each **always** block execute ***sequentially***.
- A process takes no time to execute unless it has **wait** statements in it.

Example: Signal Drivers for Simulation



```
module twoprocess;
  reg A, B;

  initial
  begin
     A = 0;
     B = 0;
  end

  // process P1
  always @(B)
  begin
     A <= 1;
     A <= #5 0;
  end

  // process P2
  always @(A)
  begin
     if (A)
        B <= #10 ~B;
  end
endmodule
```

Simulation of Inertial Delays

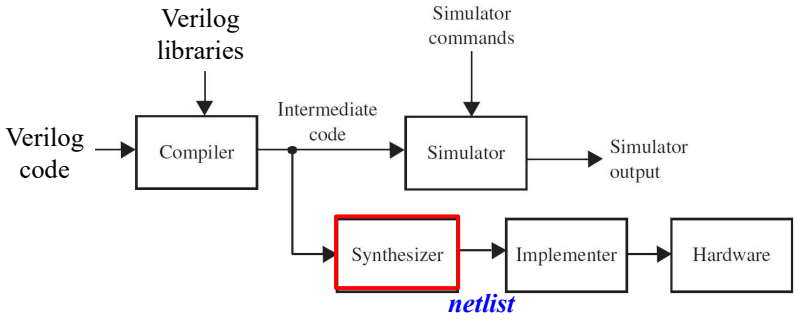
** Inertial delay, T:*
Delay the input signals by time *T*, and reject any pulse w/ a width less than *T*.

- Simulation of ***inertial delays***:
  - Each input change causes the simulator to schedule a change, which is scheduled to occur after the ***specified delay.***
  - If another input change happens before the specified delay has elapsed, the 1st change is ***dequeued*** from the simulation driver queue.

  ⇒ Only pulses wider than the specified inertial delay appear at the output.

Synthesis of Verilog Code

- Compilation, Simulation, and Synthesis of Verilog Code:



  - ***netlist*** : a list of required components and their interconnections

2-113

Synthesis

- Synthesis: automatically create hardware from a HDL description.
- Synthesis software for Verilog:
  - translates the Verilog code to a ckt description that specifies the needed ***components*** and the ***connections*** b/t the components. (***netlist***)
  - Synthesizer output: can be used to implement the digital system using specific hardware, such as a CPLD or an FPGA or as an ASIC

2-114

2-11

Verilog Data Types and Operators

J.J. Shann

A. Verilog Data Types

- Two main groups of data types:
  i. ***Variable*** data types
  ii. ***Net*** data types
- Differences b/t these two groups:
  i. Differ in the way that they are ***assigned*** and ***hold values***.
  ii. They represent different ***hardware structures***.
- All data types are predefined by the Verilog language (not by the user).

Net Data Types

- *Net* data types: **wire**, **tri** (tristate), **wand** (wired and), **wor** (wired or)
  - Represent ***physical connections*** b/t structural entities, i.e., connections b/t hardware elements.
  - **Does *not store values.***
  - Its value is determined by the values of its ***drivers*** (a ***continuous assignment*** or a ***gate***).

2-117

Variable Data Types

- *Variable* data types: **reg**, **time**, **integer**, **real**, **real-time**
  - is an abstraction of a ***data storage element***, i.e., a data storage element that ***can retain value***.
  - A variable should store a value from one assignment to the next.
  - An ***assignment statement in a procedure*** acts as a trigger that changes the value in the data storage element.

2-118

Definitions of Data Types

- Popular predefined data types:
  - ***nets***: ***connections*** b/t hardware elements, e.g., **wire**
  - ***variables***: ***data storage elements*** that can retain values, e.g., **reg**
    - **integer**: an ***integer***
    - **real**: real number constants and real variable data types for ***floating-point number***
    - **time**: a special variable data type to store ***time information***

* ***vectors***: **wire** or **reg** data types can be declared as vectors (multiple bits) (**[range1 : range2]**)

B. Predefined Verilog Operators (1/4)

Operator type	Operator symbols	Operation performed
Logical (bitwise)	~	Bitwise NOT (1's complement)
	&	Bitwise AND (binary operator)
	|	Bitwise OR (binary operator)
	^	Bitwise XOR (binary operator)
	~^ or ^~	Bitwise XNOR (binary operator)
Logical	!	Logical NOT
	&&	Logical AND
	||	Logical OR

Predefined Verilog Operators (2/4)

Operator type	Operator symbols	Operation performed
Sign	+ −	Unary sign operators
Reduction **(* unary operator)**	&	Reduction AND
	~&	Reduction NAND
	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR
Shift (unary operator)	>>	Logical right shift
	<<	Logical left shift
	>>>	Arithmetic right shift
	<<<	Arithmetic left shift

* unary operator to AND bits in a vector and reduce to one bit

* unary operator to OR bits in a vector and reduce to one bit

2-121

<< **Logical left shift**
>> **Logical right shift**
<<< **Arithmetic left shift**
>>> **Arithmetic right shift**

- Shift operators:
  - Can be applied to ***signed*** and ***unsigned*** registers.
  - If the register is ***unsigned***, ***arithmetic*** and ***logic shifts*** do the same operation.
  - E.g.: declare a register to be signed or unsigned

```
reg signed [7:0] A = 8'hA5;  //signed register A, 1010 0101
reg  [7:0] B = 8'hA5;  //unsigned register B, 1010 0101
```

  - E.g.s: signed and unsigned shifts on signed and unsigned data (A = 1010 0101, B = 1010 0101)

A >> 4	= 0000 1010	B >> 4	= 0000 1010
A >>> 4	= 1111 1010	B >>> 4	= 0000 1010
A << 4	= 0101 0000	B << 4	= 0101 0000
A <<< 4	= 0101 0000	B <<< 4	= 0101 0000

J.J. Shann 2-122

Predefined Verilog Operators (3/4)

Operator type	Operator symbols	Operation performed
Relational	>	* The result of applying a ***relational*** operator is always a **Boolean** (FALSE or TURE).
	<	
	>=	
	<=	

* Logical equality and inequality: == and !=
  If, due to ***unknown*** (*x*) or ***high-impedance*** (*z*) bits in the operands, the relation is ambiguous, then the result shall be a 1-bit unknown value (***x***).
* Case equality and inequality: === and !==
  Bits that are *x* or *z* shall be included in the comparison and shall match for the result to be considered equal. The result shall always be a known value, either 1 or 0.

Predefined Verilog Operators (4/4)

Operator type	Operator symbols	Operation performed
Arithmetic	+	Addition
	-	Subtraction (2's complement)
	*	Multiplication
	/	Division
	%	Modulus
	**	
Concatenation	{ }	
Replication	{{ }}	
Conditional	? :	Conditional

* Concatenate operator can be used on the ***left side*** of an assignment.
  E.g.: {Carry, SUM} = A + B;

Operator Precedence

- Operator precedence: p.94~96 (p.2-128~2-131)
  - Operators in **class 1** have ***highest*** precedence.
  - Operators in the same class have the same precedence and are applied ***from left to right*** in an expression.
  - The precedence order can be changed by using ***parentheses***.

  * Different languages have differences in the order of precedence.

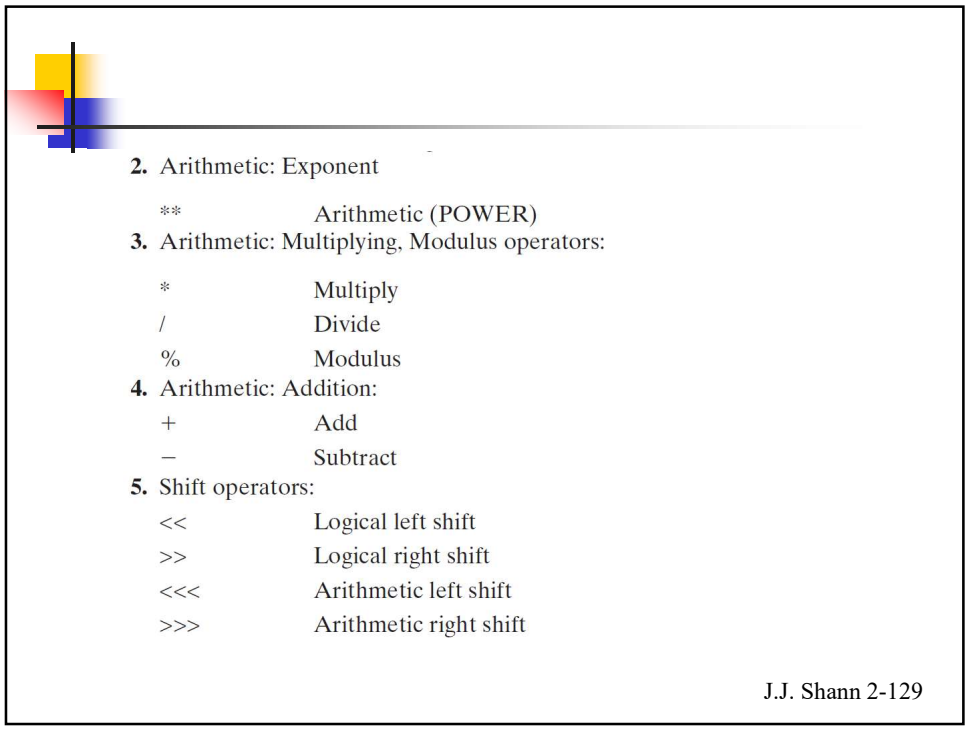
*** Use *parentheses* and make code unambiguous!**

J.J. Shann 2-127

1. Unary sign and reduction operators:

+ −	Unary sign operators
&	Reduction and (unary operator to and bits in a vector and reduce to one bit)
~&	Reduction NAND
|	Reduction or (unary operator to or bits in a vector and reduce to one bit)
~|	Reduction NOR
^	Reduction XOR
~^ or ^~	Reduction XNOR
!	Logical negation
~	Bit-wise negation

J.J. Shann 2-128



2. Arithmetic: Exponent

**	Arithmetic (POWER)

3. Arithmetic: Multiplying, Modulus operators:

*	Multiply
/	Divide
%	Modulus

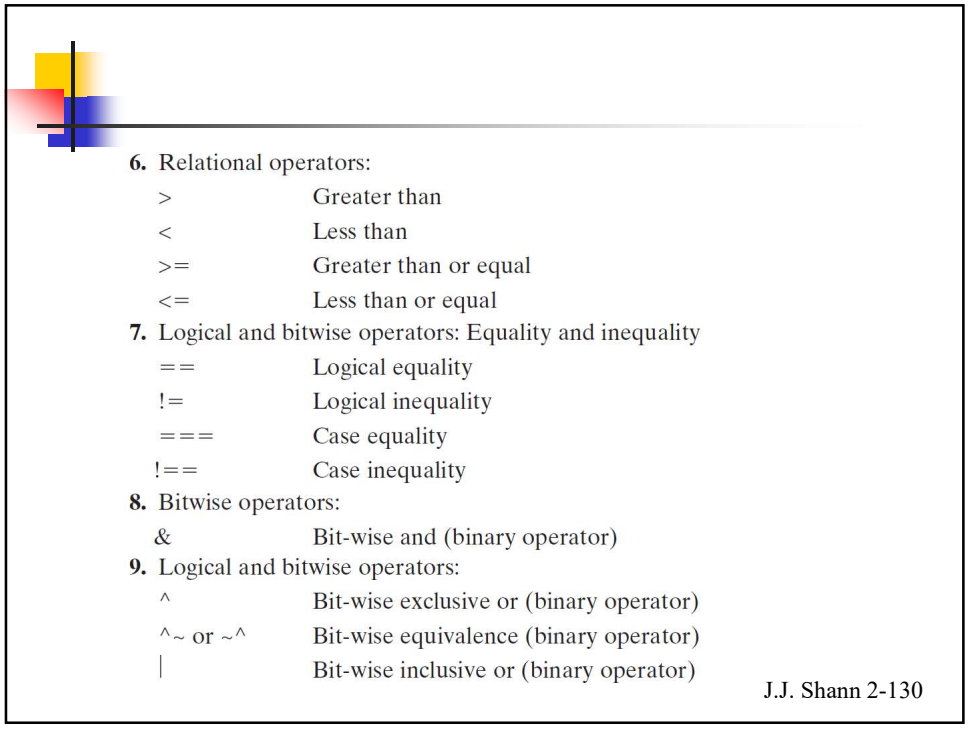
4. Arithmetic: Addition:

+	Add
−	Subtract

5. Shift operators:

<<	Logical left shift
>>	Logical right shift
<<<	Arithmetic left shift
>>>	Arithmetic right shift

J.J. Shann 2-129



6. Relational operators:

>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

7. Logical and bitwise operators: Equality and inequality

==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

8. Bitwise operators:

&	Bit-wise and (binary operator)

9. Logical and bitwise operators:

^	Bit-wise exclusive or (binary operator)
^~ or ~^	Bit-wise equivalence (binary operator)
|	Bit-wise inclusive or (binary operator)

J.J. Shann 2-130

10. Logical and:

&&	Logical and

11. Logical or:

||	Logical or

12. Conditional

? :	Conditional

13. Concatentation and replication

{}	Concatenation
{{}}	Replication

Examples

>>: logical right shift
~, &, | : bitwise NOT, AND, OR
{ }: concatenation
==: equality (logical and bitwise)

- A = 110, B = 111, C = 011000, D = 111011

(a) C >>2 = 000110
(b) C >> 2 & D = 000010
(c) ~B = 000
(d) {A, ~B} = 110000
(e) ({A, ~B}) | (C >> 2 & D) = 110010
(f) [({A, ~B} | C >> 2) & D) == 110010] = TRUE

2-12

Simple Synthesis Examples

J.J. Shann

Synthesis

- Synthesis tools:
  - infer the ***hardware components*** needed by "looking" at the ***Verilog code***
    - Each Verilog statement implies certain hardware requirements.

  * In order for code to synthesize correctly, certain conventions must be followed.

* Even if Verilog code gives the correct result when simulated, it may not result in hardware that works correctly when synthesized.

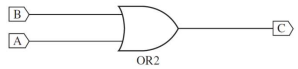
2-134

Example: Simulation and Synthesis Results in Different Outputs

- Considering the code below, what hardware will you get if you synthesized this code?
  - Most synthesizers will output an OR gate:
    - Warning messages: B is missing from the sensitivity list in **always** statement
    - Ignore the 5 ns delay in the statement.
  - The simulator output will not match the synthesizer's output.
    ⇐ The **always** statement will not execute when B changes.

```
module Q1 (A, B, C);
input  A;
input  B;
output C;
reg  C;
always @(A)
   C = #5 A | B;
endmodule
```

Synthesizer warning: B is missing from the sensitivity list

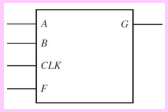


*** Always check for synthesizer warnings of missing signals in the sensitivity list.**

Example: p.100

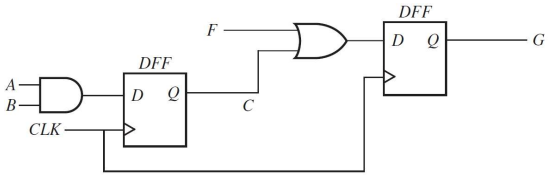
- Considering the code below, what hardware will you get if you synthesized this code?

```
module Q3 (A, B, F, CLK, G);
   input   A;
   input   B;
   input   F;
   input   CLK;
   output  G;

   reg     G;
   reg     C;

   always @(posedge CLK)
   begin
      C <= A & B;  //statement 1
      G <= C | F;  //statement 2
   end
endmodule
```



* Since the values of C and G need to be retained after the clock edge, flip-flops are required for both C and G.

* A change in the value of C from statement 1 will not be considered during the execution of statement 2 in that pass of the process.



Example

- Considering the code below, does it represent an AND gate w/ G and D as inputs?

```
always @(G or D)
begin
   if (G) Q <= D;
end
```

<Ans.>

In the device modeled, it is expected to make no changes to the output if G is not equal to 1

If currently Q = 1 and then G changes to 0:

For an AND gate: Q = G·D, it would propagate that to the output.

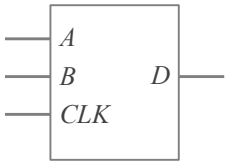
For a D latch: it make no changes to the output if G is not equal to 1.

⇒ It has to be a D-latch and not an AND gate.

Example

- Considering the code below, what hardware will you get if you synthesized this code?

```
module no_syn (A, B, CLK, D);
   input A;
   input B;
   input CLK;
   output  D;
   reg   C;
   always @(posedge CLK)
      C <= A & B;
endmodule
```



<Ans.>

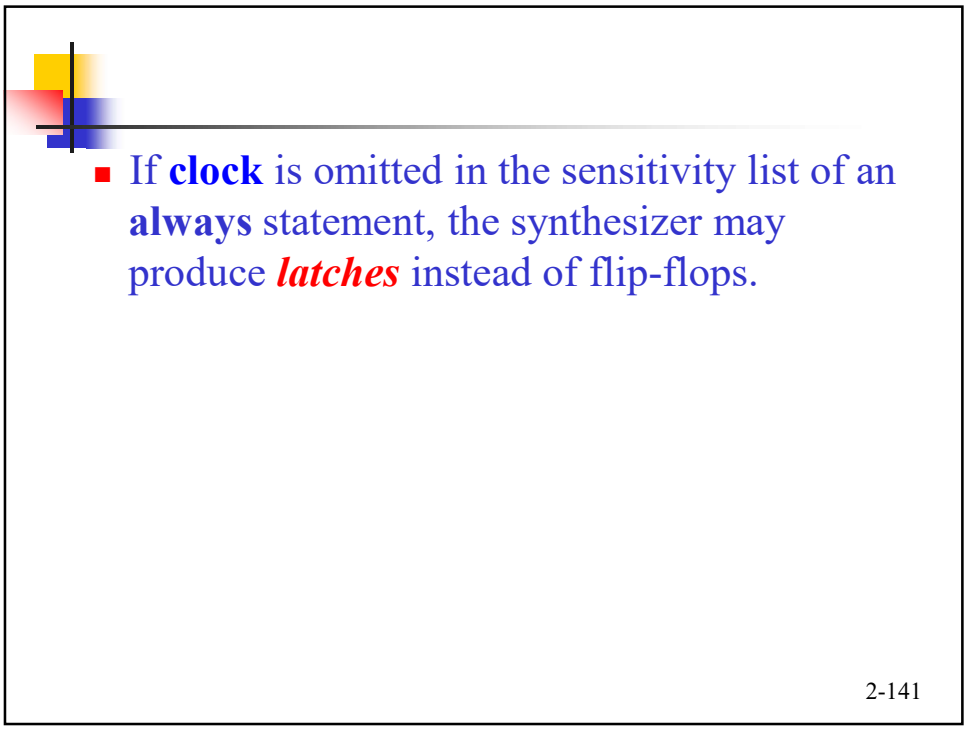
```
module no_syn (A, B, CLK, D);
    input  A;
    input  B;
    input  CLK;
    output D;
    regC;
    always @(posedge CLK)
        C <= A & B;
endmodule
```

- The synthesizer will generate an empty block diagram.
  ⇐ D, the output of the block, is never assigned.
- Warning messages:
  Input <CLK> is never used.
  Input <A> is never used.
  Input <B> is never used.
  Output <D> is never assigned.

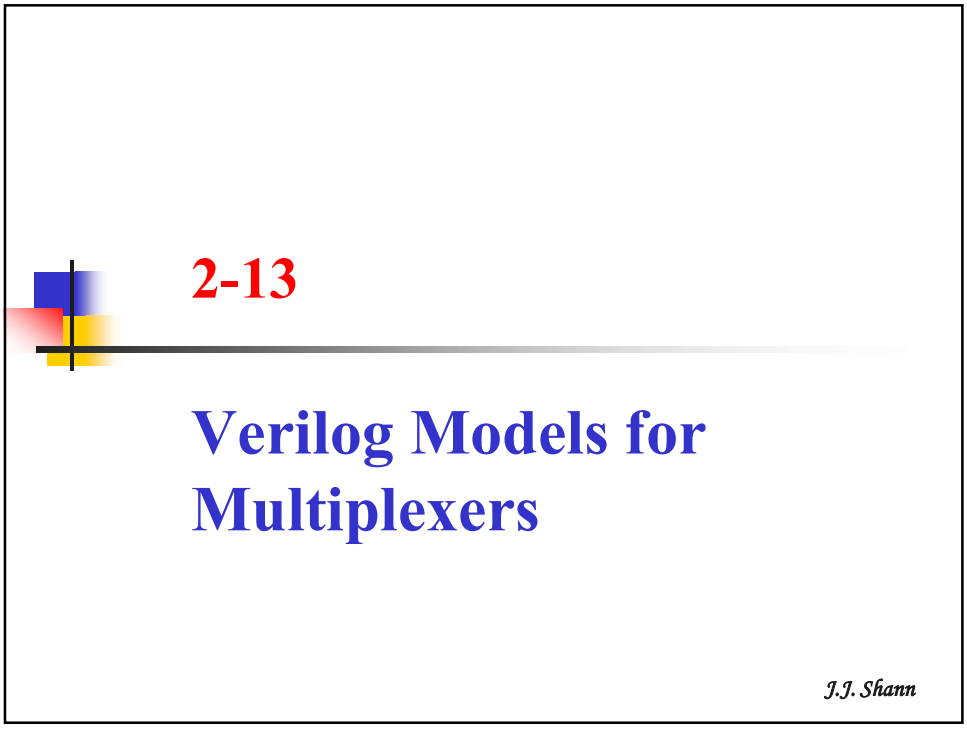
Notice: Code conventions for Synthesizers

- For inferring ***flip-flops*** or ***registers*** that change state on the ***rising/falling edge*** of a **clock** signal:
  - the ***sensitivity list*** in an **always** statement should include an ***edge-triggered signal***.
    - E.g.: always @(posedge CLK)
  - For every assignment statement in a ***clocked*** **always** statement, a signal on the left side of the assignment will cause creation of a ***register*** or ***f-f***.

* If you don't want to create unnecessary f-fs, do not put the signal assignments in a ***clocked*** **always** statement.



- If **clock** is omitted in the sensitivity list of an **always** statement, the synthesizer may produce ***latches*** instead of flip-flops.



2-13

Verilog Models for Multiplexers

J.J. Shann

Verilog Models for Multiplexers

- Multiplexer, MUX: a comb ckt
  - can be modeled using:
    i. a conditional operator with **assign** statement
    ii. a **case** statement or **if-else** statement within an **always** statement

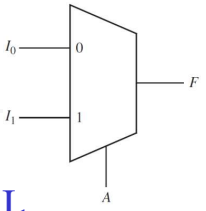
Using Conditional Operator

- Form of a ***conditional signal assignment statement***:

```
assign signal_name = condition ? expression_T : expression_F;
```

  - If *condition* is true, *signal_name* is set equal to the value of *expression_T*;
    otherwise, *signal_name* is set equal to the value of *expression_F*.
  - This concurrent statement is executed whenever a change occurs in a signal used in one of the *expressions* or *conditions*.

Example: a 2-to-1 MUX



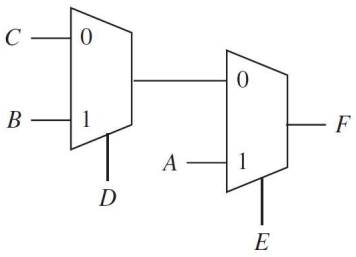
- A 2-to-1 MUX w/ 2 data inputs and one control input: $F = A' I_0 + A I_1$
  - Modeled as a single concurrent signal assignment statement:

    assign F = (~A && I0) || (A && I1);
  - Modeled by a conditional signal assignment statement:

    assign F = (A) ? I1 : I0;

Example: Cascaded 2-to-1 MUXes

- Cascaded 2-to-1 MUXes:

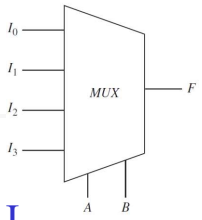


  - Modeled by a conditional signal assignment statement:

    assign F = E ? A : (D ? B : C);

    //nested conditional assignment

Example: a 4-to-1 MUX



- A 4-to-1 :

  $F = A'B' I_0 + A'B I_1 + AB' I_2 + AB I_3$

  - Modeled as a single concurrent signal assignment statement:

```
assign F = (~A && ~B && I0) || (~A && B && I1) ||
           (A && ~B && I2) || (A && B && I3);
```

  - Modeled by a conditional signal assignment statement:

```
assign F = (A) ? (B? I3 : I2) : (B ? I1 : I0);
//nested conditional assignment
```


Using If-else or Case Statement in an Always Block

- The form of a **case** statement:

```
case expression
   choice1 : sequential statements1
   choice2 : sequential statements2
   . . .
   [default : sequential statements]
endcase
```

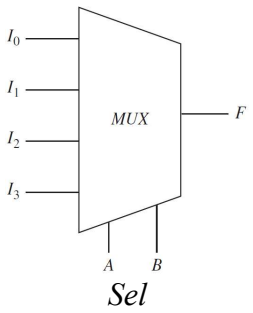
  * All possible values of the expression must be included in the choices.
  * If all values are not explicitly given, a **default** clause is required in the **case** statement.

- Alternatives: use an **if-else** statement within an **always** block

Example: a 4-to-1 MUX

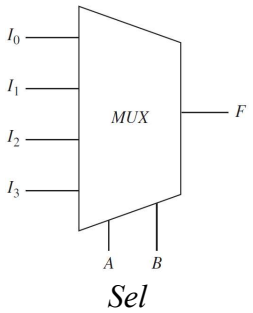
- A 4-to-1 MUX:
  - Modeled using **a case** statement within an **always** block:

```
always @ (Sel or I0 or I1 or I2 or I3)
    case (Sel)
        2'b00 :  F = I0;
        2'b01 :  F = I1;
        2'b10 :  F = I2;
        2'b11 :  F = I3;
    endcase
```



  - Modeled using an **if-else** statement within an **always** block:

```
always @ (Sel or I0 or I1 or I2 or I3)
begin
    if      (Sel == 2'b00)   F = I0;
    else if (Sel == 2'b01)   F = I1;
    else if (Sel == 2'b10)   F = I2;
    else if (Sel == 2'b11)   F = I3;
end
```



Verilog for Hardware

- ***Combinational ckts***: can be described using ***concurrent*** or ***sequential*** statements.
- ***Sequential ckts***: generally require an **always** statement.
- **always** statements can be used to make seq or comb ckts.

Writing Synthesizable Verilog for Combinational Hardware

- Important coding practices while writing ***synthesizable*** Verilog for ***comb hardware***:
  1. If possible use ***concurrent assignments*** (e.g., **assign**) to design comb logic.
  2. When procedural assignments (**always** blocks) are used for comb logic, use ***blocking assignments*** ("=").
  3. If Verilog 2001 or later is used, instead of specifying contents of sensitivity lists, use **always@*** to avoid accidental omission of inputs from sensitivity lists.

2-14

Modeling Registers and Counters Using Verilog Always Statements

J.J. Shann

Modeling Registers and Counters Using Verilog Always Statements

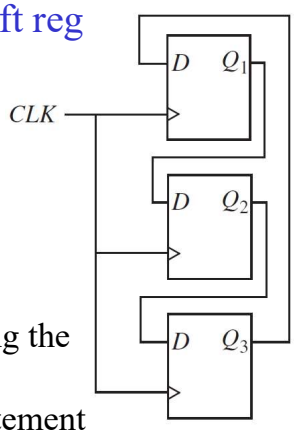
- When several f-fs change state on the same ***clock edge***, statements representing these f-fs can be placed in the same clocked **always** statements.
  - The order of the statements is not important when the ***non-blocking assignment*** operator "<=" is used.

2-154

Example: a Cyclic Shift Register

- Cyclic shift register: rotating shift reg

```
always @ (posedge CLK)
begin
   Q1 <= #5  Q3;
   Q2 <= #5  Q1;
   Q3 <= #5  Q2;
end
```



  - These f-fs all change state following the ***rising edge*** of the clock.
  - The 3 statements in the **always** statement execute in sequence with no delay.
  - The new values of the three Qs are then scheduled to change after 5 ns.

2-155

```
always @ (posedge CLK)
begin
   Q1 <= #5  Q3;
   Q2 <= #5  Q1;
   Q3 <= #5  Q2;
end
```

- If we omit the delay in the code and replace the sequential statements w/:
  Q1 <= Q3; Q2 <= Q1; Q3 <= Q2;
  - The operation would be essentially the same.
  - The 3 statements execute in sequence in zero time, and then the Qs values change after a delta delay.
- The order of the statements is not important when the non-blocking assignment operator "<=" is used;
  - The result is the same even if statements are reversed.

2-156

Example: p.105

- What is the hardware obtained if the following code is synthesized?

```
module reg3 (Q1, Q2, Q3, A, CLK);
input  A;
input  CLK;
output Q1, Q2, Q3;
reg    Q1, Q2, Q3;

always @(posedge CLK)
begin
    Q3 = Q2;  // statement 1
    Q2 = Q1;  // statement 2
    Q1 = A;   // statement 3
end
endmodule
```

→ A 3-bit shift register

* While a register can be modeled using the ***blocking operator*** "=", it is generally advised to **not** do so.

2-157

Example: p.105

- What is the hardware obtained if the following code is synthesized?

```
module reg31 (Q1, Q2, Q3, A, CLK);
input  A;
input  CLK;
output Q1, Q2, Q3;
reg    Q1, Q2, Q3;

always @(posedge CLK)
begin
    Q1 = A;   // statement 1
    Q2 = Q1;  // statement 2
    Q3 = Q2;  // statement 3
end
endmodule
```

→ i. A single f-f

ii. 3 parallel f-fs, each w/ the same input A but w/ outputs Q1, Q2, Q3, respectively

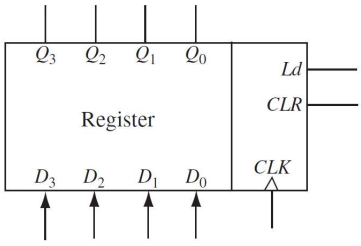
* If ***non-blocking*** statements were used in the code, the order of the statements would not have mattered.

2-158

Notice: Code conventions for Synthesizers

- While writing ***synthesizable code*** :
  - use ***non-blocking assignments*** "<=" in **always** blocks intended to create ***seq logic***
  - use ***blocking assignments*** "=" in **always** blocks intended to create ***comb logic***

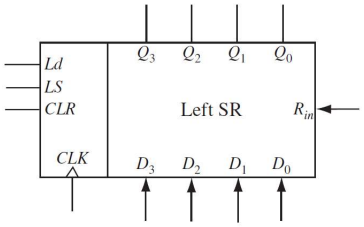
Example: Register w/ Sync Clear & Load



CLR	Ld	CLK	Q^+
1	×	↑	0
0	1	↑	D
0	0	↑	Q

```
always @ (posedge CLK)
begin
 if (CLR)          Q <= 4'b0000;
 else if (Ld)      Q <= D;
end
```

Example: Left Shift Register w/ Sync Clear & Load



CLR	Ld	LS	CLK	Q^+
1	×	×	↑	0
0	1	×	↑	D
0	0	1	↑	Left shift
0	0	0	↑	Q

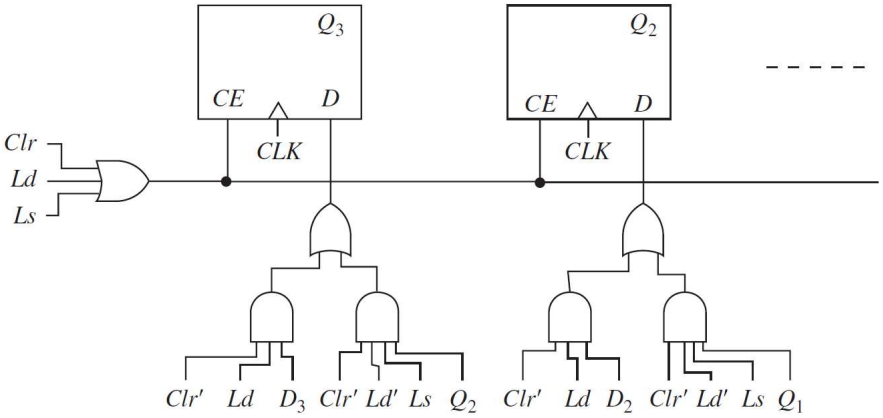
```
always @ (posedge CLK)
begin
  if (CLR)        Q <= 4'b0000;
  else if (Ld)    Q <= D;
  else if (LS)    Q <= {Q[2:0], Rin};
end
```



```
always @ (posedge CLK)
begin
  if (CLR)        Q <= 4'b0000;
  else if (Ld)    Q <= D;
  else if (LS)    Q <= {Q[2:0], Rin};
end
```

CLR	Ld	LS	CLK	Q^+
1	×	×	↑	0
0	1	×	↑	D
0	0	1	↑	Left shift
0	0	0	↑	Q

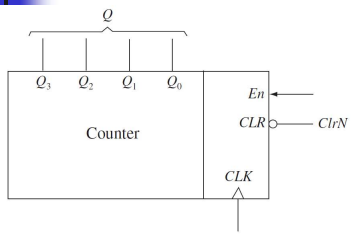
- Synthesis result:



Notice: Code conventions for Synthesizers

- A Verilog synthesizer **cannot** synthesize ***delays***.
  - Clauses of the form "# time" will be ignored by most synthesizers.
- ***Initial blocks*** and ***initial values*** for signals specified in port and signal declarations are usually ignored by synthesis tools.
  - A ***reset*** signal should be provided if the hardware must be set to a specific ***initial state***.
  - Exception: some synthesis tools for FPGAs may utilize the initial blocks and initial values in the initial bit stream that is downloaded into the FPGA.

Example: A Simple Sync Counter

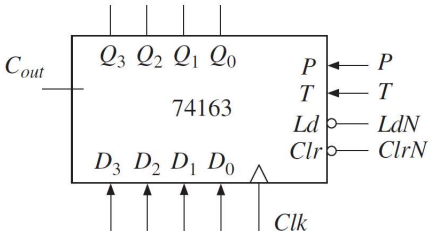


CLR	En	CLK	Q^+
0	×	↑	0
1	1	↑	+1
1	0	↑	Q

```
reg Q[3:0];

always @ (posedge CLK)
begin
 if (~ClrN)     Q <= 4'b0000;
 else if (En)   Q <= Q + 1;
end
```

Example: 74163 Counter

- 74163 counter: fully synchronous counter



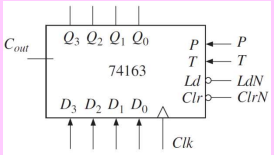
* *ClrN*: sync clear

* If $T = 1$, the counter generates a carry (C_{out}) in state 15

Control Signals			Next State				
ClrN	*LdN*	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

J.J. Shann 2-165

```
module c74163(LdN, ClrN, P, T, Clk, D, Cout, Qout);
input     LdN;
input     ClrN;
input     P;
input     T;
input     Clk;
input [3:0]     D;
output    Cout;
output [3:0]    Qout;

reg [3:0] Q;

assign Qout = Q;
assign Cout = Q[3] & Q[2] & Q[1] & Q[0] & T;

always @(posedge Clk)
begin
   if (~ClrN) Q <= 4'b0000;
   else if (~LdN) Q <= D;
   else if (P & T) Q <= Q + 1;
end
endmodule
```



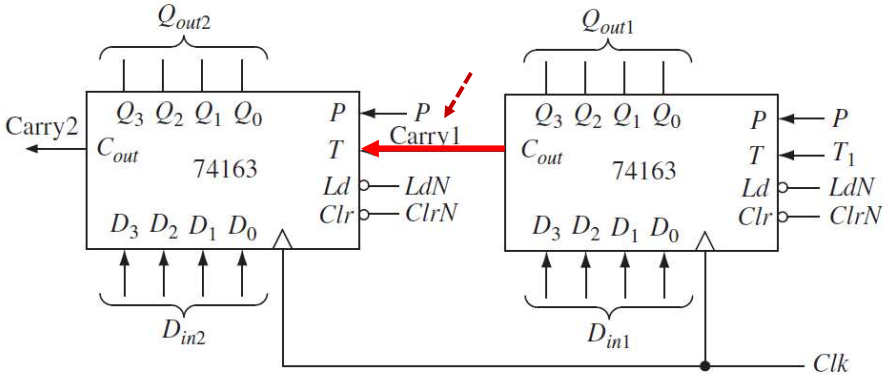
* If $T = 1$, the counter generates a carry (C_{out}) in state 15

Control Signals			Next State				
ClrN	*LdN*	*PT*	Q_3^+	Q_2^+	Q_1^+	Q_0^+	
0	X	X	0	0	0	0	(clear)
1	0	X	D_3	D_2	D_1	D_0	(parallel load)
1	1	0	Q_3	Q_2	Q_1	Q_0	(no change)
1	1	1	present state + 1				(increment count)

J.J. Shann 2-167

Example: 8-bit Counter

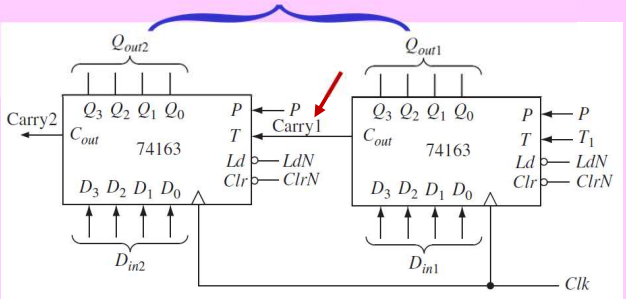
- Cascade two 74163 counter to form an 8-bit counter: * Structural modeling



J.J. Shann 2-168

```
// 8-Bit counter using two 74163 counters of the model in Fig 2-44
module eight_bit_counter (ClrN, LdN, P, T1, Clk, Din1, Din2, Count, Carry2);
input     ClrN;
input     LdN;
input     P;
input     T1;
input     Clk;
input [3:0] Din1;
input [3:0] Din2;
output [7:0]   Count;
output   Carry2;

wire Carry1;
wire [3:0] Qout1;
wire [3:0] Qout2;

c74163 ct1 (LdN, ClrN, P, T1, Clk, Din1, Carry1, Qout1);  // right
c74163 ct2 (LdN, ClrN, P, Carry1, Clk, Din2, Carry2, Qout2);   // left

assign Count = {Qout2, Qout1};

endmodule
```

Count




```
module c74163(LdN, ClrN, P, T, Clk, D, Cout, Qout);
…
endmodule
```

Unwanted latches

- Verilog signals retain their current values until they are changed.
  - This can result in the creation of ***unwanted latches*** when the Verilog code is synthesized.
  - The latch creates a variety of timing problems and unexpected behavior.

Example: Unwanted Latches

- Verilog code intending a MUX:

```
always @ (Sel or I0 or I1 or I2)
begin
   if       (Sel == 2'b00)   F = I0;
   else if  (Sel == 2'b01)   F = I1;
   else if  (Sel == 2'b10)   F = I2;
end
```

  - There would be latches to hold the value of *F* when *Sel* changes to 2'b11.

    ⇒ ***Not comb hardware any more!***
  - If *F* is multiple-bit wide, then multiple latches would be created.

Avoiding unwanted latches

- Avoiding unwanted latches:
  1. Assign a value to ***comb signals*** in ***every possible execution path*** in an **always** block intended to create ***comb hardware***.
     - One should include an **else** clause in every **if** statement or explicitly include ***all possible cases*** of the inputs.
  2. ***Initialize*** at the beginning of the **always** statement.

* For **if** then **else** statements and **case** statements, have all cases specified, or initialize at the beginning of the **always** statement.

Example: Avoiding Unwanted Latches

Verilog code w/ unwanted latches:

```
always @ (Sel or I0 or I1 or I2)
begin
   if       (Sel == 2'b00)  F = I0;
   else if  (Sel == 2'b01)  F = I1;
   else if  (Sel == 2'b10)  F = I2;
end
```

(i)

```
always @ (Sel or I0 or I1 or I2)
begin
   if       (Sel == 2'b00)  F = I0;
   else if  (Sel == 2'b01)  F = I1;
   else if  (Sel == 2'b10)  F = I2;
   else if  (Sel == 2'b11)  F = 0;
end
```

(ii)

```
always @ (Sel or I0 or I1 or I2)
F = 0;
begin
   if       (Sel == 2'b00)  F = I0;
   else if  (Sel == 2'b01)  F = I1;
   else if  (Sel == 2'b10)  F = I2;
end
```

Synthesizable Verilog for Sequential Hardware

- Important coding practices while writing ***synthesizable* Verilog** for ***sequ hardware***:
  1. Use an ***edge-triggered clock*** in the ***sensitivity list*** using the **posedge** or **negedge** keywords.
  2. Use ***non-blocking assignments***: use "<=" inside **always** blocks.
  3. Do not mix blocking and non-blocking statements in an **always** block.
  4. Do not make assignments to the same variable from more than one **always** block.

2-174

Synthesizable Verilog for Combinational Hardware

- Important coding practices while writing ***synthesizable* Verilog** for ***comb hardware***:
  1. Use *continuous assignments* **assign** to model simple ***comb*** logic.
  2. Use **always @ (*)** and ***blocking assignments*** "=" to model more complicated ***comb*** logic.
  3. ***Avoid unwanted latches*** by assigning a value to ***comb output signals*** in every possible execution path in the **always** block.
  4. Do not mix blocking and non-blocking assignments in the same **always** block.
  5. Do not make assignments to the same signal in more than one **always** statement or ***continuous assignment statement***.

2-175

- Avoiding unwanted latches:
  i. including **else** clauses for **if** statements
  ii. specifying all cases for **case** statements or have a **default** clause at the end
  iii. unconditionally assigning ***default values*** to all ***comb output signals*** at the beginning of the **always** block, i.e., initializing at the beginning of the **always** statement.

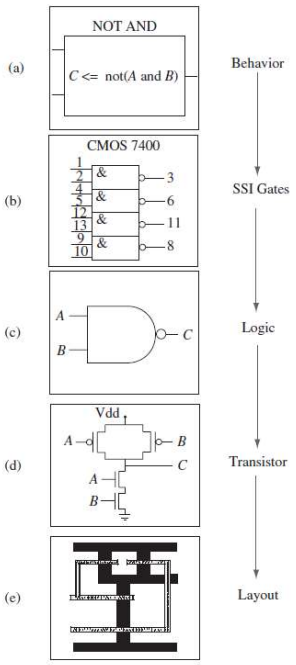
2-15

Behavioral and Structural Verilog

J.J. Shann

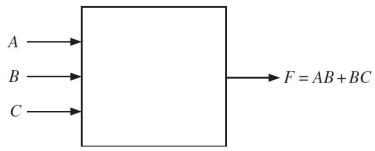
Abstraction

- Any ckt or device can be represented in multiple forms of abstraction.
  - E.g.: different levels of abstraction of a NAND device

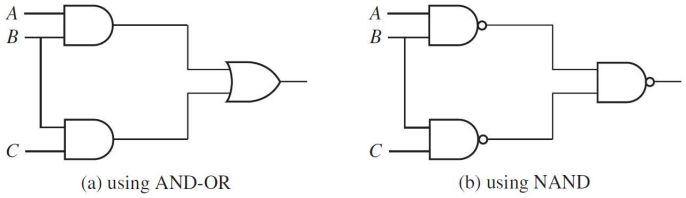


Example: $F = ab + bc$

- Behavioral level representation:



- Structural representations:



(a) using AND-OR (b) using NAND

J.J. Shann 2-179

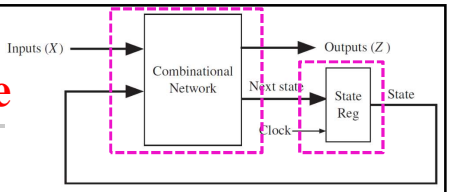
Abstraction of Verilog

- Multiple levels of abstraction:
  - ***Behavioral model***:
    - Describes the ckt or system at a high w/o implying any particular structure
    - Specifies only the overall behavior.
  - ***Data flow*** (Register Transfer Language, RTL) ***model***:
    - ***Data path*** and ***control signals*** are specified.
    - System is described in terms of the data transfer b/t registers.
  - ***Structural model***: at a low level of abstraction
    - ***Components*** used and the structure of the ***interconnection*** b/t the components are clearly specified.
    - May be detailed enough to specify use of particular ***gates*** and ***flip-flops***.

* ***Behavioral*** and ***structural*** design techniques are often combined.
* Different parts of the design are often done w/ different techniques.

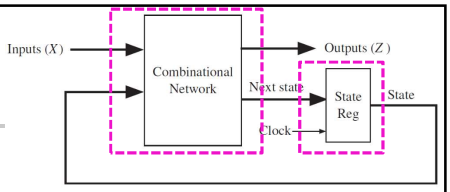
2-180

Modeling a Sequential Machine



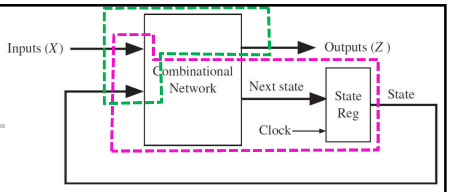
- 3 approaches for modeling a seq machine:
  1. ***Behavioral model***:
     (a) use two **always** blocks to represent the two parts of the ckt
     (b) use a single **always** block
  2. ***Data flow model***
  3. ***Structural model***

2-181



- 3 approaches for modeling a seq machine:
  1. ***Behavioral model***:
     (a) use two **always** blocks to represent the two parts of the ckt:
     - One **always** block models the ***comb*** part of the ckt ⇒ generates the ***next state information*** and ***outputs***
     - The other **always** block models the ***state register*** ⇒ updates the state at the appropriate edge of the clock

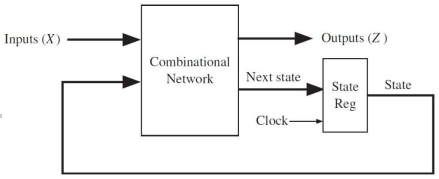
2-182



1. ***Behavioral model***: (cont'd)
   (b) use a single **always** block
   - The ***state register*** is updated directly to the proper next state value on the appropriate edge of the clock in the **always** block.
   - Use ***continuous assignment statements*** to compute the ***outputs*** out of the **always** block.

* The two **always** block model for a state machine is preferable to the one **always** block model.

2-183

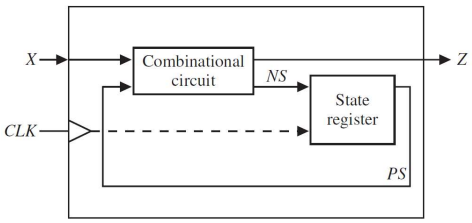


2. ***Data flow model***: based on the ***next state*** and ***output equations*** derived for the ckt.
3. ***Structural model***: describing the ***gates*** and ***f-fs*** in the ckt
   - When primitive components are required, each of these components can be defined in a ***separate Verilog module***, or by using the ***built-in primitives*** of Verilog.
   - The component modules must be included in the same file as the main Verilog description or they must be inserted as ***separate files*** in a Verilog ***project***.

Example: a Mealy Seq Ckt for BCD to Excess-3 Code Converter

- Write a Verilog model for a Mealy seq ckt: BCD to Excess-3 code converter
  - 3 approaches:
    i. ***behavioral*** (two/one **always** blocks)
    ii. ***data flow***
    iii. ***structural***

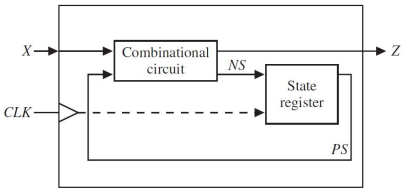
	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



<Approach 1a> Behavioral Model

- ***Behavior level*** based on the **state table** (or **state diagram**)
- Use two **always** blocks to represent the two parts of the ckt:
  - One for the ***comb ckt***, another for the ***state register***.

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

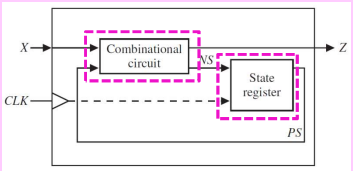



```
// This is a behavioral model of a Mealy state machine based on its state table.
// The output (Z) and next state are computed before the active edge of the clock.
// The state change occurs on the rising edge of the clock.
module Code_Converter (X, CLK, Z);
input  X, CLK;
output Z;
reg  Z;
reg  [2:0] State;
reg  [2:0] Nextstate;
initial
begin
   State = 0;
   Nextstate = 0;
end
always @(State or X)     //Combinational circuit
…
always @(posedge CLK)  // State register
…
endmodule
```

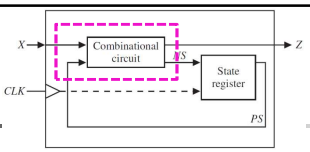
List of interface ports

Port declarations

Initial values:
is meaningful only for simulations!



	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

- 1st always block: comb ckt

```
always @(State or X)
begin
case (State)
  0 :  begin
       if (X == 1'b0)
       begin
            Z = 1'b1;
            Nextstate = 1;
       end
       else
       begin
            Z = 1'b0;
            Nextstate = 2;
       end
  end
```

```
  1 :  begin
       if (X == 1'b0)
       begin
            Z = 1'b1;
            Nextstate = 3;
       end
       else
       begin
            Z = 1'b0;
            Nextstate = 4;
       end
  end
```

```
  2 :  begin
       if (X == 1'b0)
       begin
            Z = 1'b0;
            Nextstate = 4;
       end
       else
       begin
            Z = 1'b1;
            Nextstate = 4;
       end
  end
```

PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

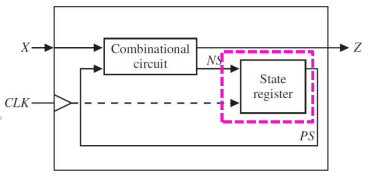
```
  3 :  begin
       if (X == 1'b0)
       begin
            Z = 1'b0;
            Nextstate = 5;
       end
       else
       begin
            Z = 1'b1;
            Nextstate = 5;
       end
  end
```

```
  4 :  begin
       if (X == 1'b0)
       begin
            Z = 1'b1;
            Nextstate = 5;
       end
       else
       begin
            Z = 1'b0;
            Nextstate = 6;
       end
  end
```

```
  5 :  begin
       if (X == 1'b0)
       begin
            Z = 1'b0;
            Nextstate = 0;
       end
       else
       begin
            Z = 1'b1;
            Nextstate = 0;
       end
  end
```

```
    6 :  begin
         if (X == 1'b0)
         begin
              Z = 1'b1;
              Nextstate = 0;
         end
         else
         begin
              Z = 1'b0;
              Nextstate = 0;
         end
    end
    default : begin
    // should not occur
    end
endcase
end
```

PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



– 2nd always block: state reg

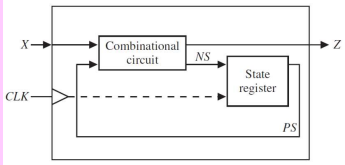
```
always @(CLK)          // State register
begin                  // rising edge of clock (simulation)
   State <= Nextstate;
end
```

(×)

```
always @(posedge CLK)  // State register
begin                  // rising edge of clock (synthesis)
   State <= Nextstate;
end
```

(✓)

```
module Code_Converter (X, CLK, Z);
input  X, CLK;
output  Z;

reg  Z;
reg  [2:0] State;
reg  [2:0] Nextstate;

initial
begin
   State = 0;
   Nextstate = 0;
end

always @(State or X)
begin
case (State)
   0 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b1;
             Nextstate = 1;
        end
        else
        begin
             Z = 1'b0;
             Nextstate = 2;
        end
   end
   1 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b1;
             Nextstate = 3;
        end
        else
        begin
             Z = 1'b0;
             Nextstate = 4;
        end
   end
```



PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

```
   2 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b0;
             Nextstate = 4;
        end
        else
        begin
             Z = 1'b1;
             Nextstate = 4;
        end
   end
   3 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b0;
             Nextstate = 5;
        end
        else
        begin
             Z = 1'b1;
             Nextstate = 5;
        end
   end
```

```
   4 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b1;
             Nextstate = 5;
        end
        else
        begin
             Z = 1'b0;
             Nextstate = 6;
        end
   end
   5 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b0;
             Nextstate = 0;
        end
        else
        begin
             Z = 1'b1;
             Nextstate = 0;
        end
   end
```

```
   6 :  begin
        if (X == 1'b0)
        begin
             Z = 1'b1;
             Nextstate = 0;
        end
        else
        begin
             Z = 1'b0;
             Nextstate = 0;
        end
   end
   default : begin
   // should not occur
   end
endcase
end

always @(posedge CLK)
begin
   State <= Nextstate;
end

endmodule
```

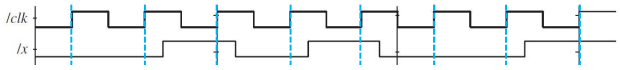
*** force signal_name v1 t1, v2 t2, …**

⇒ signal_name gets the value v1 at time t1, …

- Simulator command file:

  add wave CLK X State Nextstate Z
  force CLK 0 0, 1 100 –repeat 200
  force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
  run 1600

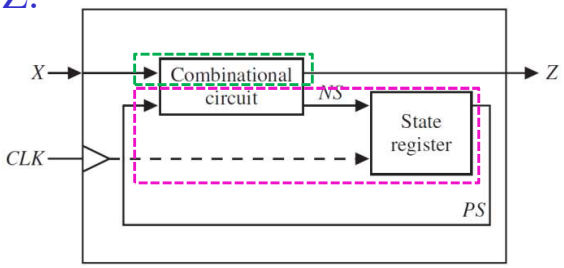
- Simulator output for the converter:



- Synthesis result: using Xilinx ISE tools
  - 7 D flip-flops, 15 2-input AND gates, 3 2-input OR gates, and 1 7-input OR gate. (using ***one-hot design*** by default)

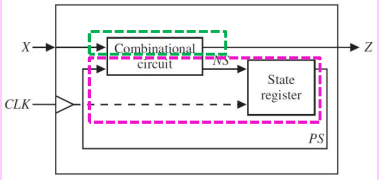
<Approach 1b> Behavioral level

- Use a single **always** block:
  - The state register is updated directly to the proper next state value on the rising edge of the clock.
- Use a ***continuous assignment statement*** to compute Z.



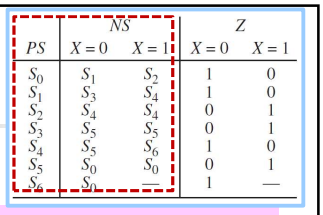
2-195

```
// This is a behavioral model of a Mealy state machine based on its state table.
// The output (Z) and next state are computed before the active edge of the clock.
// The state change occurs on the rising edge of the clock.

module Code_Converter (X, CLK, Z);
input  X, CLK;
output Z;                          --> List of interface ports
reg [2:0] State;                   --> Port declarations
initial                            --> Initial values: is meaningful
begin                                  only for simulations!
   State = 0;
end
always @(posedge CLK)  // State register
...
assign ...                         //Combinational circuit
endmodule
```



PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

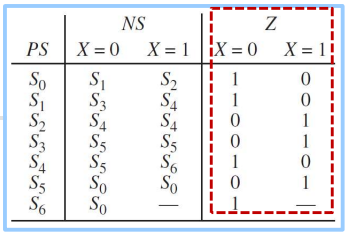
2-196



PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

– The **always** block: state register

```
always @(posedge CLK)
begin
case (State)
   0 :  begin
        if (X == 1'b0)
             State <= 1;
        else
             State <= 2;
   end
   1 :  begin
        if (X == 1'b0)
             State <= 3;
        else
             State <= 4;
   end
   2 :
        State <= 4;
   3 :
        State <= 5;
   4 :  begin
        if (X == 1'b0)
             State <= 5;
        else
             State <= 6;
   end
   5 :
        State <= 0;
   6 :
         State <= 0;
   default : begin
   end
endcase
end
```

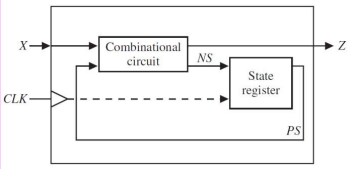



PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

– The conditional assignment statement: ***comb ckt***

```
assign Z = (State == 0 && X == 1'b0) || (State == 1 && X == 1'b0) ||
           (State == 2 && X == 1'b1) || (State == 3 && X == 1'b1) ||
           (State == 4 && X == 1'b0) || (State == 5 && X == 1'b1) ||
           State 6  ?  1'b1 : 1'b0
```

```
module Code_Converter (X, CLK, Z);

input  X, CLK;
output  Z;

reg  [2:0] State;

initial
begin
    State = 0;
end

always @(posedge CLK)
begin
case (State)
    0 :    begin
           if (X == 1'b0)
                State <= 1;
           else
                State <= 2;
    end
    1 :    begin
           if (X == 1'b0)
                State <= 3;
           else
                State <= 4;
    end
    2 :
           State <= 4;
    3 :
           State <= 5;
    4 :    begin
           if (X == 1'b0)
                State <= 5;
           else
                State <= 6;
    end
    5 :
           State <= 0;
    6 :
           State <= 0;
    default : begin
    end
endcase
end

assign Z = (State == 0 && X == 1'b0) || (State == 1 && X == 1'b0) ||
           (State == 2 && X == 1'b1) || (State == 3 && X == 1'b1) ||
           (State == 4 && X == 1'b0) || (State == 5 && X == 1'b1) ||
           State 6  ?  1'b1 : 1'b0

endmodule
```



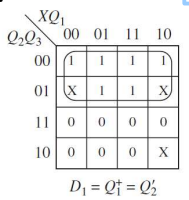
PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—

J.J. Shann 2-199

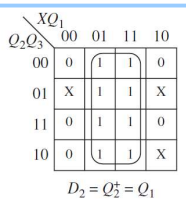
<Approach 2> Data Flow Model

- ***Data flow model*** based on the ***next state*** and ***output equations***
  - State assignment used:
    - S0 → 0 (000)
    - S1 → 4 (100)
    - S2 → 5 (101)
    - S3 → 7 (111)
    - S4 → 6 (110)
    - S5 → 3 (011)
    - S6 → 2 (010)

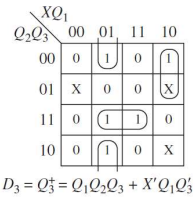
PS	NS X = 0	NS X = 1	Z X = 0	Z X = 1
S_0	S_1	S_2	1	0
S_1	S_3	S_4	1	0
S_2	S_4	S_4	0	1
S_3	S_5	S_5	0	1
S_4	S_5	S_6	1	0
S_5	S_0	S_0	0	1
S_6	S_0	—	1	—



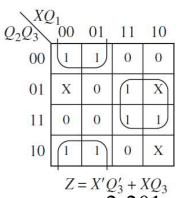
$D_1 = Q_1^+ = Q_2'$



$D_2 = Q_2^+ = Q_1$



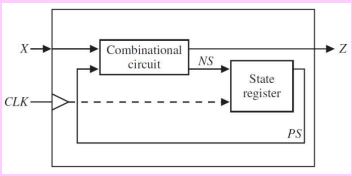
$D_3 = Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$



$Z = X'Q_3' + XQ_3$

2-201

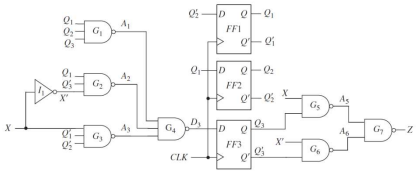
```
module Code_Converter(X, CLK, Z);

input  X;
input  CLK;
output  Z;

reg  Q1;
reg  Q2;
reg  Q3;

always @(posedge CLK)
begin
    Q1 <= #10 (~Q2);
    Q2 <= #10 Q1;
    Q3 <= #10 (Q1 & Q2 & Q3) | ((~X) & Q1 & (~Q3)) |
                (X & (~Q1) & (~Q2));
end

assign #20 Z =  ((~X) & (~Q3)) | (X & Q3);

endmodule
```



$$Q_1^+ = Q_2'$$
$$Q_2^+ = Q_1$$
$$Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$$
$$Z = X'Q_3' + XQ_3$$

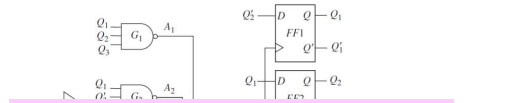
<Approach 3> Structural Model

- ***Structural model***: describing the ***gates*** and ***flip-flops*** in the ckt



  - When primitive components are required, each of these components can be defined in a separate ***Verilog module***, or by using the ***built-in primitives*** defined in Verilog.
  - The component modules must be included in the same file as the main Verilog description or they must be inserted as ***separate files*** in a ***Verilog project***.

```
// D flip-flop
module DFF (D, CLK, Q, QN);

input  D;
input  CLK;
output Q;
output QN;

reg    Q;
reg    QN;

initial
begin
    Q  = 1'b0;
    QN = 1'b1;
end

always @(posedge CLK)
    begin
        Q  <= #10 D;
        QN <= #10 (~D);
    end

endmodule
```

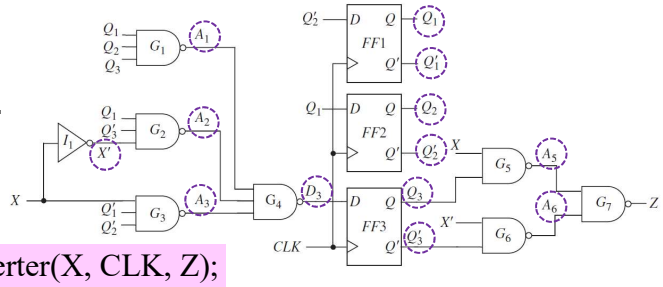


```
// 3-input NAND gate
module Nand3 (A1, A2, A3, Z);

input  A1;
input  A2;
input  A3;
output Z;

assign #10  Z = (~ (A1 & A2 & A3));

endmodule
```

```
// 2-input NAND gate
module Nand2 (A1, A2, Z);
…
endmodule
```

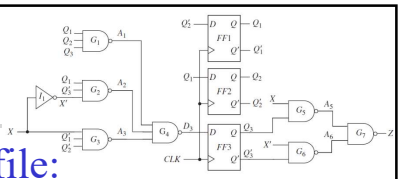
```
// Inverter
module Inverter (A, Z);
…
endmodule
```

* Built-in primitives



```
module Code_Converter(X, CLK, Z);

input  X, CLK;
output Z;

wire  A1, A2, A3, A5, A6, D3, Q1,
      Q2, Q3, Q1N, Q2N, Q3N, XN;

Inverter I1(X, XN);
Nand3 G1(Q1, Q2, Q3, A1);
Nand3 G2(Q1, Q3N, XN, A2);
Nand3 G3(X. Q1N, Q2N, A3);
Nand3 G4(A1, A2, A3, D3);
```

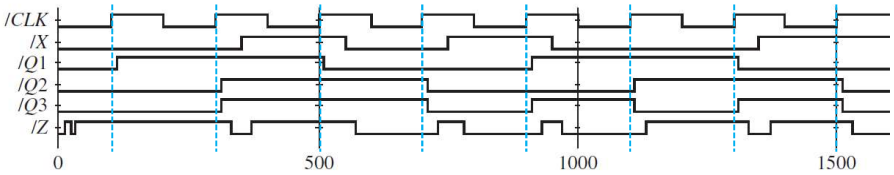
```
DFF FF1(Q2N, CLK, Q1, Q1N);
DFF FF2(Q1, CLK, Q2, Q2N);
DFF FF3(D3, CLK, Q3, Q3N);
Nand2 G5(X, Q3, A5);
Nand2 G6(XN, Q3N, A6);
Nand2 G7(A5, A6, Z);

endmodule
```



- The simulator command file:

```
add wave CLK X Q1 Q2 Q3 Z
force CLK  0 0, 1 100 –repeat 200
force X  0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600
```

- Simulation results:



- Synthesis result:
  - Get exactly the same ckt

Synthesis

- Synthesis w/ integers:
  - ***Integers*** are generally treated as ***32-bit*** quantities.
  - While writing ***synthesizable code***, use the appropriate # of bits for your variables as needed as opposed to using integers.
  - One should use integers only where they are ***not explicitly synthesized***, e.g., looping variables.
  - If integers are used for ***state variables*** in the state machine, some synthesis tools may be able to figure out the # of bits actually required by the state variable, but some may generate 32-bit registers.

- ***Structural*** model vs. ***behavioral*** model:
  - Structural model:
    - has more control over the generated circuitry
    - takes a lot more effort to produce a structural model.
  - Behavioral model:
    - Designers often use behavioral design in order to achieve quick ***time to market***.
    - CAD tools have matured significantly, and most modern synthesis tools are capable of producing efficient hardware for arithmetic and logic ckts.

2-16

Constants

Constants

’ : apostrophe
` : back quote

- 3 different ways to define ***constant values***:
  1. **`define**: ***compiler directive***, define a number or an expression for a meaningful string

     `define constant_name constant_value
     - E.g.: **`define** wordsize 16
       reg [1:`wordsize] data;
  2. **parameter**: define constants that should not be changed

     parameter constant_name = constant_value;
     - E.g.s:
       parameter msb = 15; // defines msb as a constant value 15
       parameter [31:0] decim = 1'b1; //value converted to 32 bits
  3. **localparam**: similar to **parameter**

     localparam constant_name = constant_value;

2-211

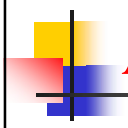
- Parameters: (Ch8)
  - are used to define ***constant values*** in a module
  - can be used to customize the module instances
  - Typical uses of parameters: specify ***delays*** and ***width of variables***
  - Do not belong to either the *variable* or the *net* group.
  - Module parameters can be modified at ***compilation time***, but is illegal to modify their values at run time.

2-212

2-17

Arrays

J.J. Shann



Arrays

- Key feature of VLSI circuits is the repeated use of similar structures.
- ***Arrays*** in Verilog:
  - can be used while modeling the repetition.
  - can be used to create ***memory arrays*** and specify the ***values to be stored in these arrays***.
  - must declare the array ***upper*** and ***lower bound***.
  - can be created of various ***data types***.

2-214

Declaring Array Bounds

- 2 positions to declare the ***array bounds***:
  1. declared *b/t* the ***variable type*** (**reg** or **net**) and the ***variable name***
     - The array bound means the # of bits for the declared variable.
     - E.g.: **reg** [7:0] eight_bit_register;
       eight_bit_register = 8'b00000001; // initialize
  2. declared *after the name of the array*
     - E.g.: **reg** rega [1:n]; // an array of n 1-bit registers
       reg [1:n] regb; // an n-bit register
- E.g.: declare 16 8-bit registers

  reg [7:0] eight_bit_register_array [15:0];

2-215

Matrices

- Matrices: multidimensional array types
  - may be defined w/ two or more dimensions
- E.g.:

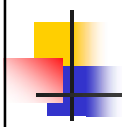
  Define a 2-D array variable in an **initial** statement, which is a matrix of integers w/ four rows and three columns w/ 8-bit elements.

```
matrixA [0:3][0:2] = {{ 1,  2,  3},
                      { 4,  5,  6},
                      { 7,  8,  9},
                      {10, 11, 12}};
```

  row column

  ⇒ *matrixA[3][1]* = 11

2-216



Look-Up Table Method

- Look-up table (LUT):
  - Use ***array construct*** together w/ **parameter** to create ***look-up tables***.
- Look-up table (LUT) method: ROM method
  - can be used to create ***comb ckts***

Example: Odd-Parity Generator

- Use Verilog arrays to represent a parity generator that generates a 5-bit-odd-parity generation for a 4-bit input number using the look-up table (LUT) method.
- Accomplished by the ***ROM*** or ***LUT method*** using a look-up table of size 16 entries × 5 bits.

<Ans.>

- The look-up table: odd-parity generation

Input (LUT Address)				Output (LUT Data)				
A	B	C	D	P	Q	R	S	T
0	0	0	0	0	0	0	0	
0	0	0	1	0	0	0	1	
0	0	1	0	0	0	1	0	
0	0	1	1	0	0	1	1	
0	1	0	0	0	1	0	0	
0	1	0	1	0	1	0	1	
0	1	1	0	0	1	1	0	
0	1	1	1	0	1	1	1	
1	0	0	0	1	0	0	0	
1	0	0	1	1	0	0	1	
1	0	1	0	1	0	1	0	
1	0	1	1	1	0	1	1	
1	1	0	0	1	1	0	0	
1	1	0	1	1	1	0	1	
1	1	1	0	1	1	1	0	
1	1	1	1	1	1	1	1	

- The Verilog code for the parity generator:

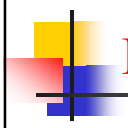
Input (LUT Address)				Output (LUT Data)				
A	B	C	D	P	Q	R	S	T
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	1
0	1	0	0	0	1	0	0	0
0	1	0	1	0	1	0	1	1
0	1	1	0	0	1	1	0	1
0	1	1	1	0	1	1	1	0
1	0	0	0	1	0	0	0	0
1	0	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	0	1	1	0	0	1
1	1	0	1	1	1	0	1	0
1	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1

```
module parity_gen (X, Y);

input   [3:0] X;
output  [4:0] Y;

wire    ParityBit;

parameter [0:15] OT= {1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 1'b1, 1'b1, 1'b0,
1'b0, 1'b1, 1'b1, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1};

assign  ParityBit = OT[X];
assign  Y = {X, ParityBit};

endmodule
```

2-18

Loops in Verilog

J.J. Shann



Loops in Verilog

- Loops:
  - Activity occurring in a ***repetitive*** way.
  - A loop statement is a ***sequential statement***.
  - Kinds of loop statements: **for**, **while**, **repeat**

Forever Loop (Infinite Loop)

- Infinite loops:
  - can be useful in hardware modeling where a device works continuously until the power is off
  - E.g.:

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

J.J. Shann 2-223

For Loops

- The general form of a **for** loop:

```
for (initial_statement; expression; incremental_statement)
begin
  sequential statement(s);
end
```

- E.g.: initialize array variable using **for** loop

```
reg [7:0] eight_bit_register_array [15:0];
for (i=0; i<16; i=i+1)
begin
   eight_bit_register_array[i] = 8'b00000000;
end
```

2-224

Example: a 4-bit Parallel Adder

- Use the **for** loop to create multiple copies of a basic cell.

```
for (i=0; i<4; i=i+1)
begin
    Cout = (A[i] && B[i]) || (A[i] && Cin) || (B[i] && Cin);
    sum[i] = A[i] ^ B[i] ^ Cin;
    Cin = Cout;
end
```

  - The carry out (C_{out}) from one iteration is copied to the carry in (C_{in}) before the end of the loop.

2-225

While Loops

- While loop:
  - A condition is tested before each iteration.
  - It is used mainly for ***simulation***.
- The general form of a while loop:

```
while condition
begin
    sequential statements;
end
```

2-226

Example: an up counter

- Use the **while** statement to continue the incrementing process until the stop is encountered or the counter reaches the maximum.

```
`define  MAX 100

module counter_100;
integer count;

initial  begin
   count = 0;
   while (count < `MAX)  begin
       count = count + 1;
   end   // while
   $display("number = %d ", count);
end     // initial

endmodule
```

2-227

Repeat Loops

- Repeat loop:
  - repeats the sequential statement(s) for specified times.
  - The # of repetitions is set by a constant value or a logical expression.
- E.g.:

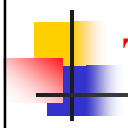
```
repeat ( 8 )
begin
   x = x + 1;
   y = y + 2;
end
```

2-228

2-19

Testing a Verilog Model

J.J. Shann



Testing a Verilog Model

- A model has to be tested and validated before it can be successfully used.
- ***Test bench***: a piece of ***Verilog code*** that can provide input combinations to test a Verilog model for the system under test.
  - Test benches are frequently used during ***simulation*** to provide sequences of inputs to the circuit or Verilog model under test and observe the outputs.

2-230

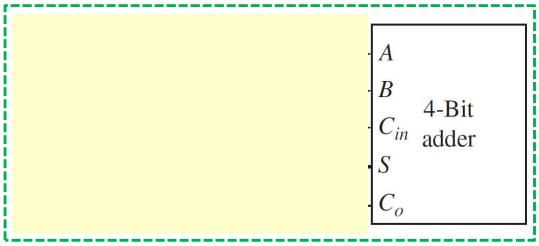
System Tasks for Observing Outputs

- **$display**: outputs/prints values exactly where it is executed and adds a ***new line character*** to the end of its output
- **$write**: works the same as **$display**, but does not add a new line character to the end of its output
- **$strobe**: displays at the very end of the current ***simulation time unit***
- **$monitor**: displays ***every time*** one of its parameters changes

2-231

Example: Testing a 4-bit Binary Adder

- Test bench for testing a 4-bit binary adder:




```
module Adder4 (S, Co, A, B, Ci);
output [3:0] S;
output Co;
input  [3:0] A, B;
input  Ci;

wire   [3:1] C;  // C is an internal signal

// instantiate four copies of the FullAdder

FullAdder FA0 (A[0], B[0], Ci,   C[1], S[0]);
FullAdder FA1 (A[1], B[1], C[1], C[2], S[1]);
FullAdder FA2 (A[2], B[2], C[2], C[3], S[2]);
FullAdder FA3 (A[3], B[3], C[3], Co,   S[3]);

endmodule
```

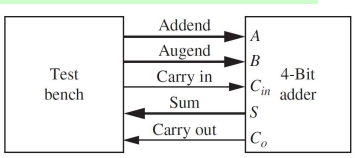
```
module FullAdder(X, Y, Cin, Cout, Sum);
output Cout, Sum;

input X, Y, Cin;

  assign #10 Sum = X ^ Y ^ Cin;
  assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);
endmodule
```

2-232

```
module TestAdder_v2;
parameter N = 11;  // a random set of 11 tests
reg    [3:0]  addend;
reg    [3:0]  augend;
reg           cin;
wire   [3:0]  sum;
wire          cout;
reg    [3:0]  addend_array[1:N];
reg    [3:0]  augend_array[1:N];
reg    [1:N]  cin_array;
reg    [3:0]  sum_array[1:N];
reg    [1:N]  cout_array;
initial  begin  …
// initialization of addend_array, cin_array, augend_array,
// initialization of sum_array and cout_array (expected sum and carry outputs)
end
integer  i;
always begin  // check whether the module outputs match the expected outputs
…
end
Adder4 add1(addend, augend, cin, sum, cout);  //module under test instantiated
endmodule
```

* Inputs of the module under test

* Outputs of the module under test




```
module Adder4 (S, Co, A, B, Ci);
output [3:0] S;
output Co;
input  [3:0] A, B;
input  Ci;

wire   [3:1] C;  // C is an internal signal

// instantiate four copies of the FullAdder

FullAdder FA0 (A[0], B[0], Ci,   C[1], S[0]);
FullAdder FA1 (A[1], B[1], C[1], C[2], S[1]);
FullAdder FA2 (A[2], B[2], C[2], C[3], S[2]);
FullAdder FA3 (A[3], B[3], C[3], Co,   S[3]);

endmodule
```

– ***Initial block***: initialization of ***inputs*** (addend_array, cin_array, augend_array) and expected ***outputs*** (sum_array, and cout_array)

```
initial
begin
  //initialization of addend_array
  addend_array[1] = 4'b0111;
  addend_array[2] = 4'b1101;
  addend_array[3] = 4'b0101;
  addend_array[4] = 4'b1101;
  addend_array[5] = 4'b0111;
  addend_array[6] = 4'b1000;
  addend_array[7] = 4'b0111;
  addend_array[8] = 4'b1000;
  addend_array[9] = 4'b0000;
  addend_array[10] = 4'b1111;
  addend_array[11] = 4'b0000;
  //initialization of augend_array
  augend_array[1] = 4'b0101;
  augend_array[2] = 4'b0101;
  augend_array[3] = 4'b1101;
  augend_array[4] = 4'b1101;
  augend_array[5] = 4'b0111;
  augend_array[6] = 4'b0111;
  augend_array[7] = 4'b1000;
  augend_array[8] = 4'b1000;
  augend_array[9] = 4'b1101;
  augend_array[10] = 4'b1111;
  augend_array[11] = 4'b0000;
```

```
  //initialization of cin_array
  cin_array[1] = 1'b0;
  cin_array[2] = 1'b0;
  cin_array[3] = 1'b0;
  cin_array[4] = 1'b0;
  cin_array[5] = 1'b1;
  cin_array[6] = 1'b0;
  cin_array[7] = 1'b0;
  cin_array[8] = 1'b0;
  cin_array[9] = 1'b1;
  cin_array[10] = 1'b1;
  cin_array[11] = 1'b0;
  //initialization of sum_array
  sum_array[1] = 4'b1100;
  sum_array[2] = 4'b0010;
  sum_array[3] = 4'b0010;
  sum_array[4] = 4'b1010;
  sum_array[5] = 4'b1111;
  sum_array[6] = 4'b1111;
  sum_array[7] = 4'b1111;
  sum_array[8] = 4'b0000;
  sum_array[9] = 4'b1110;
  sum_array[10] = 4'b1111;
  sum_array[11] = 4'b0000;
```

```
  //initialization of cout_array
  cout_array[1] = 1'b0;
  cout_array[2] = 1'b1;
  cout_array[3] = 1'b1;
  cout_array[4] = 1'b1;
  cout_array[5] = 1'b0;
  cout_array[6] = 1'b0;
  cout_array[7] = 1'b0;
  cout_array[8] = 1'b1;
  cout_array[9] = 1'b0;
  cout_array[10] = 1'b1;
  cout_array[11] = 1'b0;
end
```

J.J. Shann 2-234

– ***Always block***: check whether the module outputs match the expected outputs:

```
integer i;

always
begin
  for(i = 1 ; i <= N ; i = i + 1)
  begin
      $display(i);
      addend <= addend_array[i];//apply an addend test vector
      augend <= augend_array[i];//apply an augend test vector
      cin <= cin_array[i];//apply a carry in
      #(40);//adder expected to take 40 time units
    if(!(sum == sum_array[i] & cout == cout_array[i]))
    begin
        $write("ERROR: ");
        $display("Wrong Answer ");
    end
    else begin
        $display("Correct!!");
    end
  end
   $display("Test Finished");
end
```

$display: outputs/prints values exactly where it is executed and adds a ***new line character*** to the end of its output.

$write: works the same as **$display**, but does not add a new line character to the end of its output.

2-20

A Few Things to Remember

J.J. Shann

Purposes of Writing Verilog

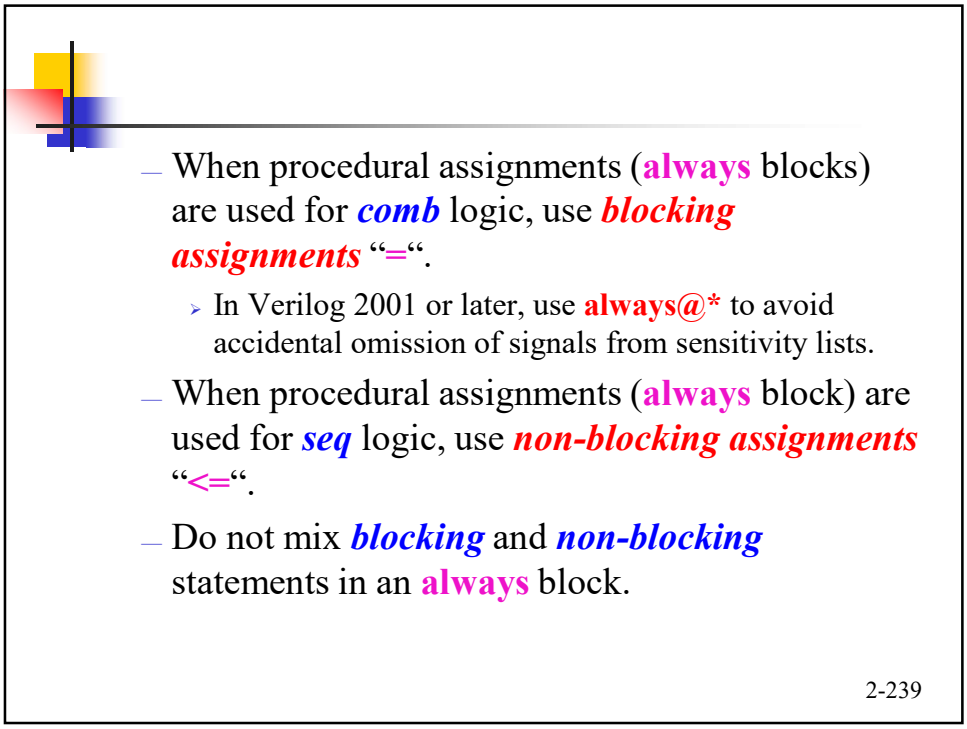
- Verilog is typically written for the following three reasons:
  1. To design hardware (i.e., to ***model*** and ***synthesize*** hardware)
  2. To model hardware (i.e., to create ***simulation*** models that are not necessarily synthesizable)
  3. To verify hardware (i.e., to ***test*** designs)

2-237

Designing (*Synthesizing*) Hardware

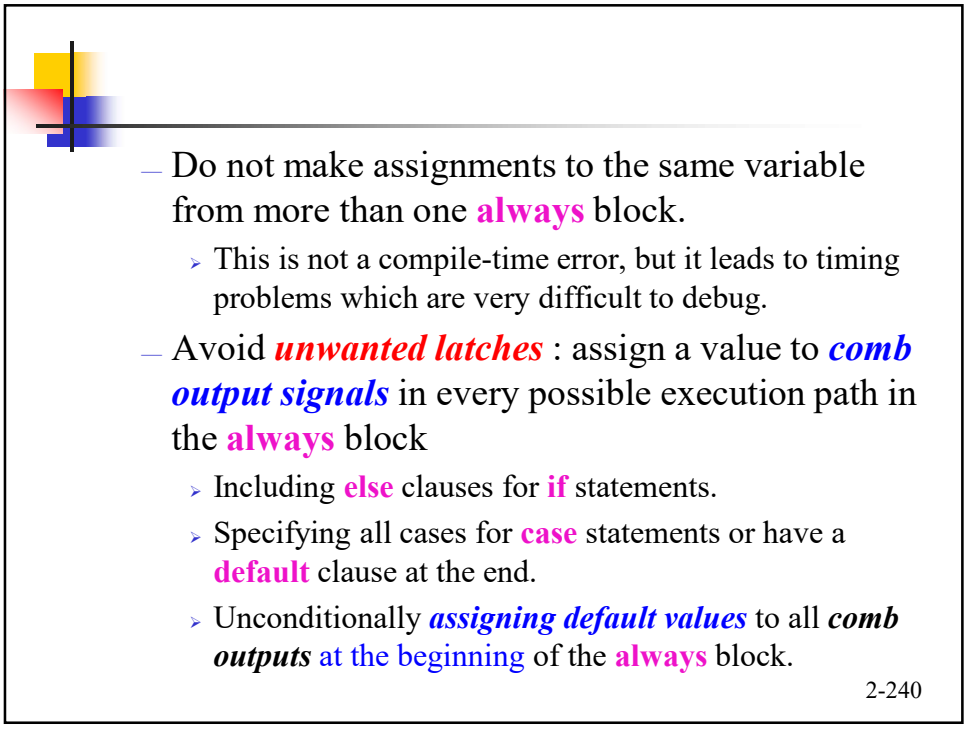
- Important guidelines of writing Verilog code for ***designing***, i.e., ***modeling*** + ***synthesizing***, hardware:
  - Do not use **initial** blocks.
    - Initial blocks are usually ignored during synthesis.
  - Do not use ***delays*** (either *delayed assignment* or *delayed evaluation*).
    - Delays are ignored during synthesis.
  - If possible, use ***concurrent assignments*** (**assign**) to design ***comb*** logic.

2-238



- When procedural assignments (**always** blocks) are used for ***comb*** logic, use ***blocking assignments*** "=".
  - In Verilog 2001 or later, use **always@*** to avoid accidental omission of signals from sensitivity lists.
- When procedural assignments (**always** block) are used for ***seq*** logic, use ***non-blocking assignments*** "<=".
- Do not mix ***blocking*** and ***non-blocking*** statements in an **always** block.

2-239



- Do not make assignments to the same variable from more than one **always** block.
  - This is not a compile-time error, but it leads to timing problems which are very difficult to debug.
- Avoid ***unwanted latches*** : assign a value to ***comb output signals*** in every possible execution path in the **always** block
  - Including **else** clauses for **if** statements.
  - Specifying all cases for **case** statements or have a **default** clause at the end.
  - Unconditionally ***assigning default values*** to all ***comb outputs*** at the beginning of the **always** block.

2-240

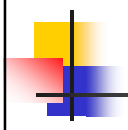
Modeling (*Simulating*) Hardware

- Important guidelines of writing Verilog for ***modeling***, i.e., creating ***simulation models*** which are not necessarily synthesizable, hardware:
  - If delays are not to be modeled, use blocking assignments "=" for *comb* logic and non-blocking assignments "<=" for *seq* logic.
  - In blocking assignments with no delay specification, the new values are assigned immediately w/o any delta delays.
  - In non-blocking assignments with no delay specification, the change is scheduled to occur after a delta time.

2-241

- To model *comb logic* w/ ***inertial delays***, use ***delayed evaluation blocking statements***.
  - E.g.: #10 A = B;
- To model *comb logic* with ***transport delays***, use ***delayed assignment non-blocking assignments***.
  (* inside an **always** statement.)
  - E.g.: A <= #10 B;
- To model *seq logic* w/ delays, use ***delayed assignment non-blocking assignments***.
  (* inside an **always** statement.)
  - E.g.: A <= #10 B;

2-242



- Use ***inertial delays*** if pulse-rejection behavior is required.
  - If input pulses are narrowed than the inertial delay values, output changes will not occur.
  - Remember this fact when checking simulation outputs.
- Do not make assignments to the same variable from more than one **always** block.
  - This is not a compile-time error, but it leads to timing problems which are very difficult to debug.
  - It may appear that the ckt is working at times and not working at times.

2-243

Verifying (*Testing*) Hardware

- ***Verification models*** and ***test benches*** mostly use **initial** blocks w/ ***blocking assignments*** and ***delayed assignments***.
- ***Delays*** are often used to create ***test stimuli*** arrive to unit under test at precise times, and results are compared against expected results after the delays for the ckt to work.
- Good strategies for simulation apply here as well.

2-244

$display: outputs values exactly where it is executed.
$strobe: displays at the very end of the current simulation time unit.
$monitor: displays every time one of its parameters changes.

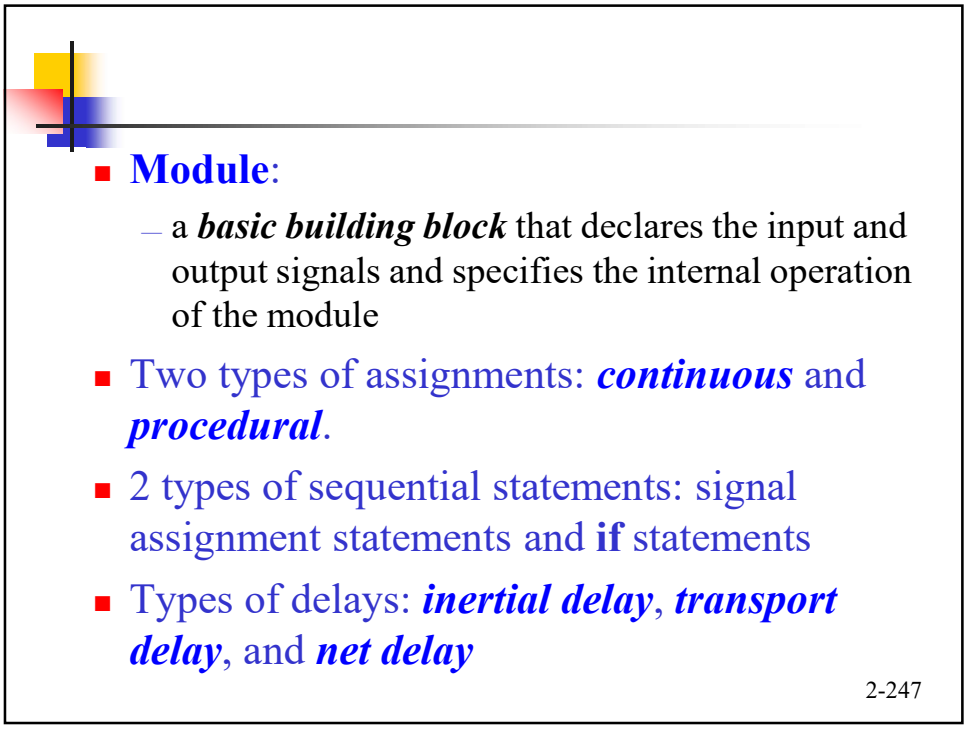
- Especially important strategies for verifying hardware:
  - Although all types of assignment statements can be used in verification models, use ***blocking assignments*** if possible.
  - When delays are used, pay attention to ***inertial*** behavior.
  - Use **initial** blocks to hard code ***test stimulus values***.
  - Use ***parameters*** for creating constants so test benches can easily be modified.
  - Be aware of the differences between **$display**, **$strobe** and **$monitor** so that wrong conclusions are not made about correctly working ckts.

2-245

Summary

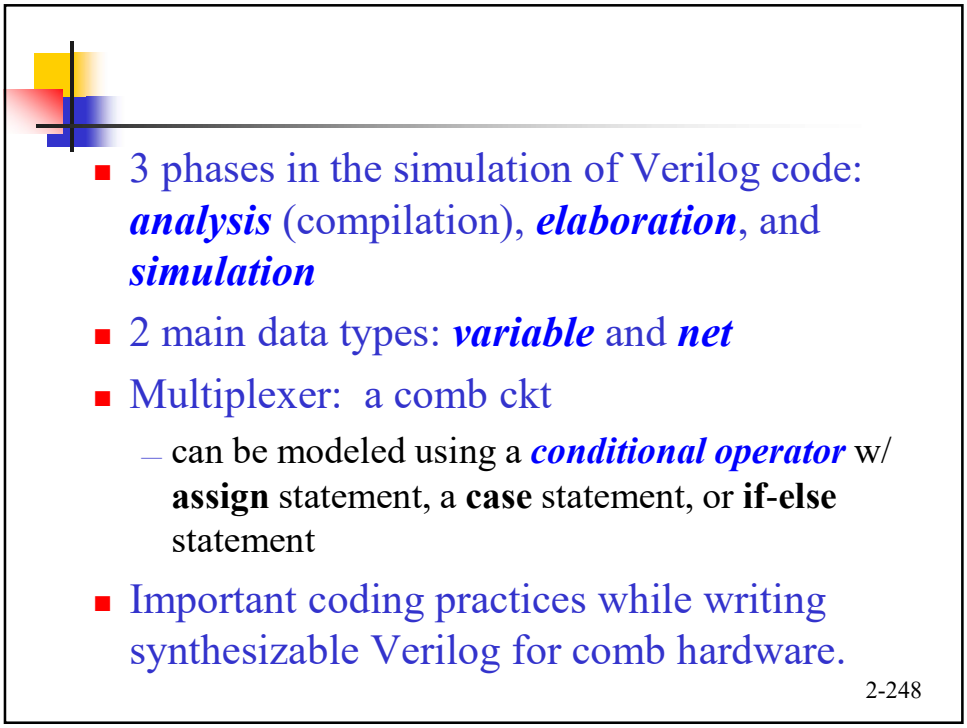
- Steps in design:
  - state the problem, design requirements, develop specs, design formulation, design entry, simulation, logic synthesis, post synthesis simulation, mapping, placement, and routing.
- Verilog models ***comb circuits*** by what are called ***concurrent statements*** or ***continuous assignments***.
- Data types: **reg** (register) and **wire**
- Keywords: words such as **and**, **or**, and **always** are reserved words (or keywords)

2-246



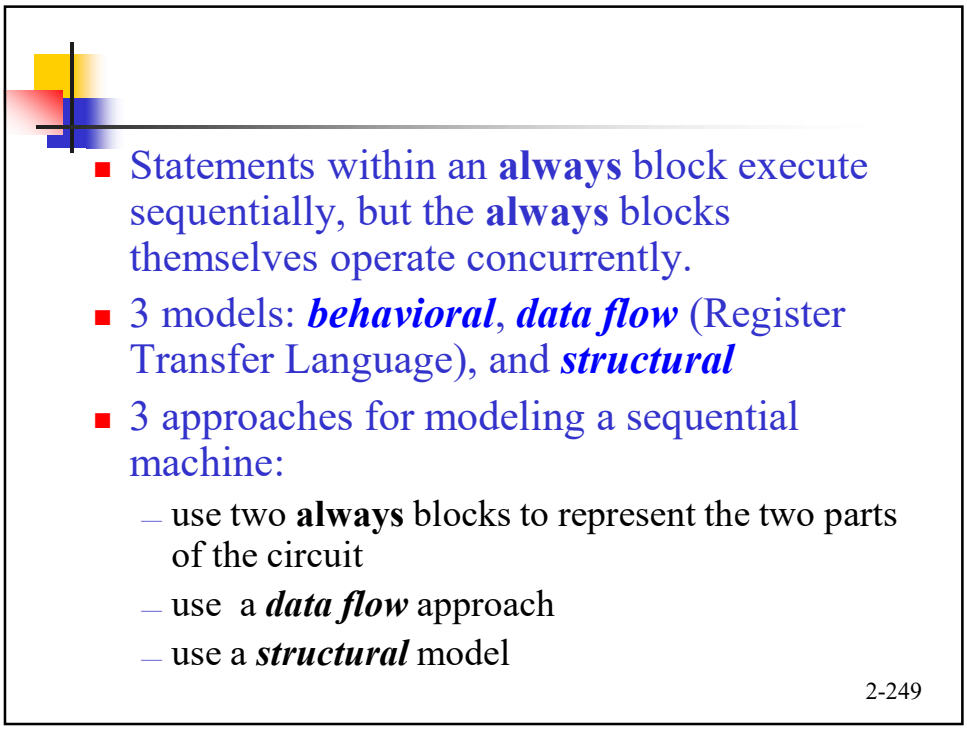
- **Module**:
  - a ***basic building block*** that declares the input and output signals and specifies the internal operation of the module
- Two types of assignments: ***continuous*** and ***procedural***.
- 2 types of sequential statements: signal assignment statements and **if** statements
- Types of delays: ***inertial delay***, ***transport delay***, and ***net delay***

2-247



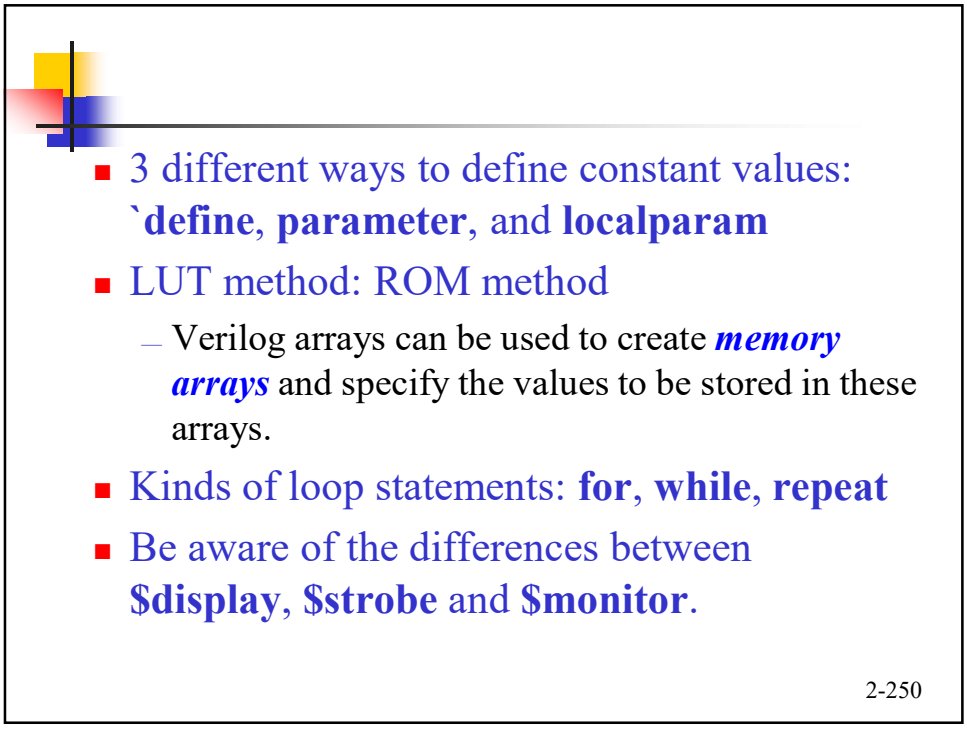
- 3 phases in the simulation of Verilog code: ***analysis*** (compilation), ***elaboration***, and ***simulation***
- 2 main data types: ***variable*** and ***net***
- Multiplexer: a comb ckt
  - can be modeled using a ***conditional operator*** w/ **assign** statement, a **case** statement, or **if-else** statement
- Important coding practices while writing synthesizable Verilog for comb hardware.

2-248



- Statements within an **always** block execute sequentially, but the **always** blocks themselves operate concurrently.
- 3 models: ***behavioral***, ***data flow*** (Register Transfer Language), and ***structural***
- 3 approaches for modeling a sequential machine:
  - use two **always** blocks to represent the two parts of the circuit
  - use a ***data flow*** approach
  - use a ***structural*** model

2-249



- 3 different ways to define constant values: **`define**, **parameter**, and **localparam**
- LUT method: ROM method
  - Verilog arrays can be used to create ***memory arrays*** and specify the values to be stored in these arrays.
- Kinds of loop statements: **for**, **while**, **repeat**
- Be aware of the differences between **$display**, **$strobe** and **$monitor**.

2-250