Introduction to Hardware Description Language (Verilog)

Kamyar Mirzazad
kammirzazad@ee.sharif.edu

Fall 2014

Digital vs. Analog Design

❖ Digital Design Advantages:

- More noise reliability
- Provides a perfect vehicle for digital signal processing
- Allows modular chip design
- Allows coding to achieve higher performance
- Allows encryption for higher security
- Can be verified on programmable devices before tape-out
- Enjoys the benefit of advanced sophisticated CAD tools

Digital Systems Components

- Printed Circuit Board (PCB)
- Software Oriented
  - Microprocessor (general purpose)
  - Microcontroller
  - Digital Signal Processor (DSP)
- Programmable Logic Devices
  - Simple Programmable Logic Device (SPLD)
  - Complex Programmable Logic Device (CPLD)
  - Field Programmable Gate Arrays (FPGAs)
- Application Specific Integrated Circuits (ASICs)

Computer Structure / Microprocessor

DSP Lab

Logic Circuits

ASIC/FPGA Chip Design

HDL Coding

- HDL allows us to describe the functionality of a logic circuit in a language that is
  - Easy to understand
  - Easy to share
  - Hides complicated implementation details

- Designer more concerned about the design functionality than the detailed design

Synthesis

- Synthesis tool:
  - Analyzes a piece of Verilog code and converts it into optimized logic gates
  - This conversion is done according to the **“language semantics”**
  - **Input :**
    - HDL Code
    - **“Technology library”** file -> Standard cells (known by transistor size)
      - Basic gates (AND , OR , NOR , …)
      - Macro cells (Adders , Muxes , Memory , Filp-Flops , …)
    - Constraint file (Timing , area , power , loading requirement , optimization Algorithms)
  - **Output:**
    - A gate-level **“Netlist”** of the design
    - Timing files (.sdf)

Introduction: Why HDL?

- Schematic entry not feasible for large designs:
  - Time consuming to draw the schematic for millions of gates
  - Subject to mistakes
  - Difficult design entry and sharing
  - Different design entry tools to learn
  - Lack of Vendor independency : Tools not compatible (hard to convert the design from one to another)
  - Not easy to modify

- Solution:
  - Describe the design in text -> Hardware Description Language
  - Just describe the design **“behavior”** not the detailed gate-level logic
  - Gate-level logic is generated automatically by a **“synthesis”** tool

Advantages of HDL Coding

- Designer describes what the hardware should do without actually describing the hardware itself
- HDL Coding allows designers to separate behavior from implementation *(abstraction)*
- Designers develop an executable functional specification that documents the exact behavior of all the components and their interfaces
- Designers can make decisions about cost, performance, power, and area earlier in the design process

HDL is NOT a Software Programming Language

- Software Programming Language
  - Language which can be translated with compiler into machine instructions (*assembly*) and then executed on a computer

- Hardware Description Language
  - A specialized computer language used to describe the structure, design and operation of digital logic circuit , translated with synthesizer into executable specifications of H/W
  - Includes an explicit notion of time

Hardware Description Language

- Verilog
  - Introduction
  - Language Fundamentals
  - Modeling Combinational & Sequential Logic Circuits
  - Modeling Finite State Machines
  - Verilog Operations

HDL Coding

- A hardware Description Language is a high-level programming language that offers special constructs, used to model microelectronic circuits

- Two standard HDLs:
  - VHDL (Very high- speed integrated circuit HDL)
  - Verilog

- Verilog:
  - Developed by Philip Moorby in 1985 as a proprietary language
  - Open to public by Cadence Design Systems in 1990
  - IEEE Standard in 1995 and revised in 2001

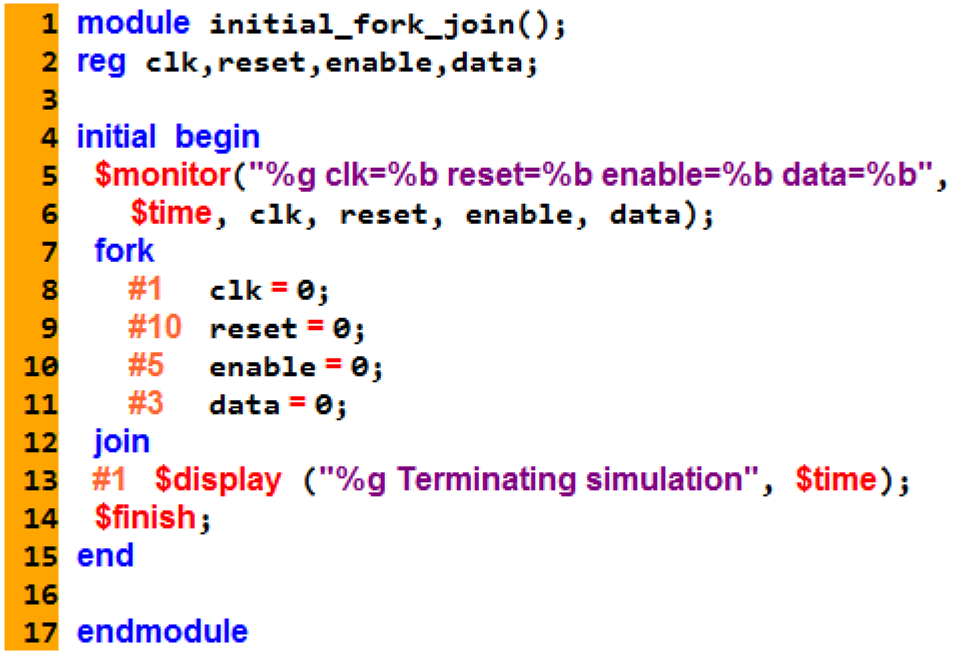
Abstraction Levels in Verilog

- Three are types of Verilog Coding:
  - Behavioral: *(Most Descriptive)*
    - Describes a system by the flow of data between its functions Blocks
    - Defines signal values when they change
  - RTL (Register Transfer Level): *(Somehow descriptive)*
    - Describe how data transfers between registers and input/outputs
    - Describes a system by the flow of data and control signals between and within its functional blocks
  - Structural: *(Least Descriptive)*
    - Shows detailed design components, nets, and interconnects
    - Uses technology-specific, low-level components
    - Used to pass netlist information between design tools

Behavioral Coding

- The behavioral level describes the behavior of a design without implying any specific internal architecture:
  - You use high level constructs, such as @, case, if, repeat, wait, while
  - You can use any behavioral constructs of the HDL in your testbench
  - Synthesis tools accept only a limited subset of these behavioral constructs

- This behavioral model defines the behavior of the design as seen at its port



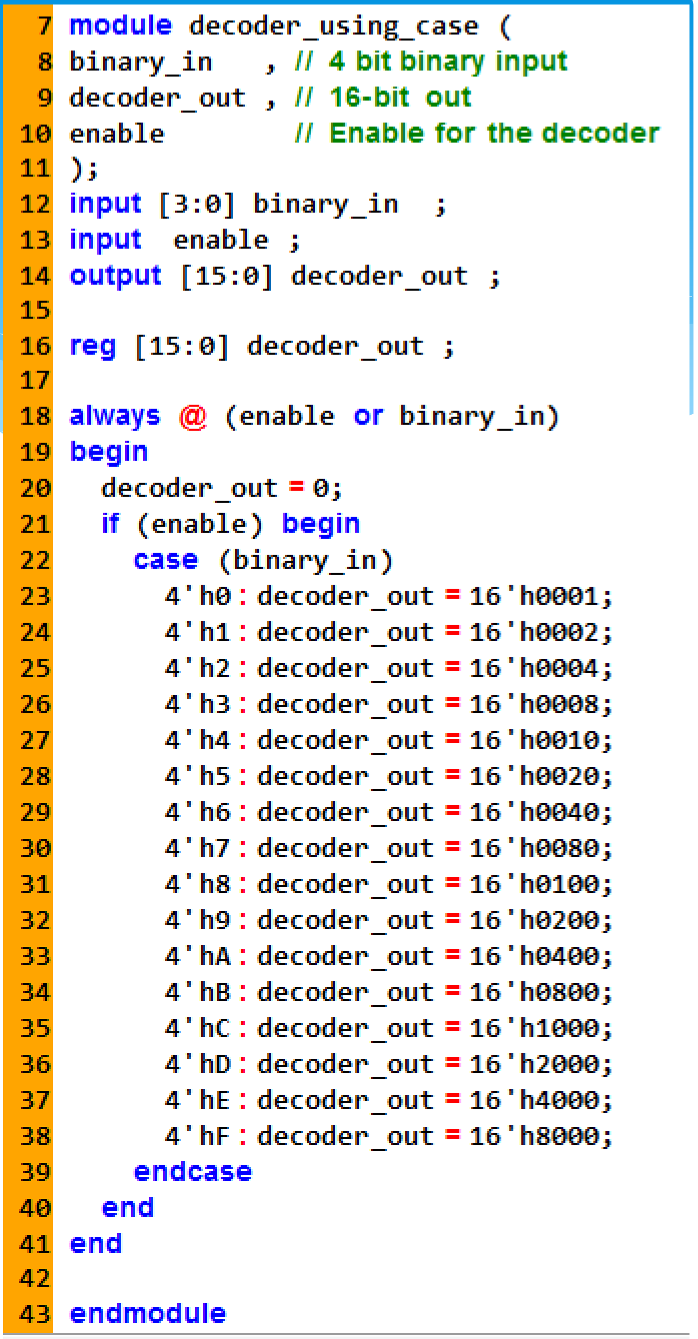
```
module initial_fork_join();
reg clk,reset,enable,data;

initial begin
 $monitor("%g clk=%b reset=%b enable=%b data=%b",
   $time, clk, reset, enable, data);
 fork
   #1   clk = 0;
   #10  reset = 0;
   #5   enable = 0;
   #3   data = 0;
 join
 #1  $display ("%g Terminating simulation", $time);
 $finish;
end

endmodule
```

RTL Coding

- ❖ The RTL (functional) level describes the design architecture in sufficient detail that a synthesis tool can construct the circuit

- ❖ RTL coding is the closest one to the actual hardware implementation

- ❖ RTL code includes a subset of all Verilog syntax
  - ○ Not all Verilog syntax are synthesizable



```
module decoder_using_case (
binary_in    , // 4 bit binary input
decoder_out , // 16-bit out
enable         // Enable for the decoder
);
input [3:0] binary_in   ;
input  enable ;
output [15:0] decoder_out ;

reg [15:0] decoder_out ;

always @ (enable or binary_in)
begin
  decoder_out = 0;
  if (enable) begin
    case (binary_in)
      4'h0 : decoder_out = 16'h0001;
      4'h1 : decoder_out = 16'h0002;
      4'h2 : decoder_out = 16'h0004;
      4'h3 : decoder_out = 16'h0008;
      4'h4 : decoder_out = 16'h0010;
      4'h5 : decoder_out = 16'h0020;
      4'h6 : decoder_out = 16'h0040;
      4'h7 : decoder_out = 16'h0080;
      4'h8 : decoder_out = 16'h0100;
      4'h9 : decoder_out = 16'h0200;
      4'hA : decoder_out = 16'h0400;
      4'hB : decoder_out = 16'h0800;
      4'hC : decoder_out = 16'h1000;
      4'hD : decoder_out = 16'h2000;
      4'hE : decoder_out = 16'h4000;
      4'hF : decoder_out = 16'h8000;
    endcase
  end
end

endmodule
```

Structural Coding

- Synthesis tools produce a purely structural design description.
- The structural level is also appropriate for small library components:
  - You can use built-in Verilog primitives, such as the **and** gate
  - You can describe your own User Defined Primitives (UDPs)
- This structural model instantiates predefined library components

```
module some_logic_component (c, a, b);
       // declare port signals
  output c;
  input a, b;
      // declare internal wire
  wire d;
      //instantiate structural logic gates
  and a1(d, a, b); //d is output, a and b are inputs
  not n1(c, d);    //c is output, d is input
endmodule
```

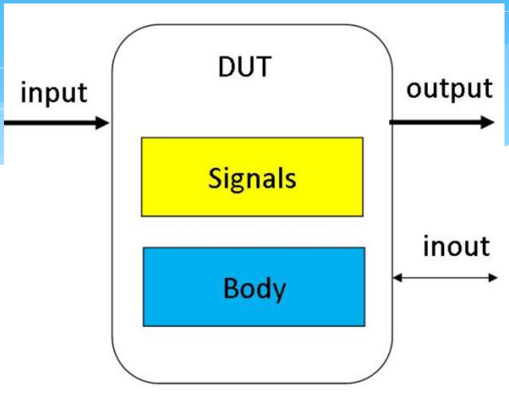
Verilog Fundamentals

- Module
- Signals
  - Signal Type
  - Signal Range
  - Signal Name
- Parameters
- Memories
- Operators
- Statements
  - Concurrent Statements
  - Procedural Statements

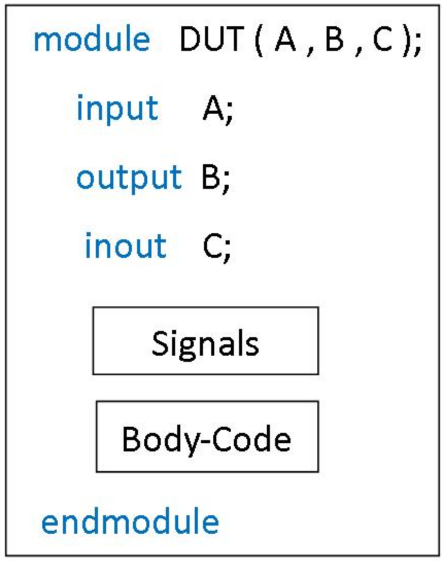
Verilog Fundamentals: Module

- Any circuit or subcircuit is declared as a "module" in Verilog.
- Each module may have:
  - Ports (Three possibilities)
    - Input
    - Output
    - Inout
  - Signals (main or intermediate)
  - Body-code

  (statements for module description)

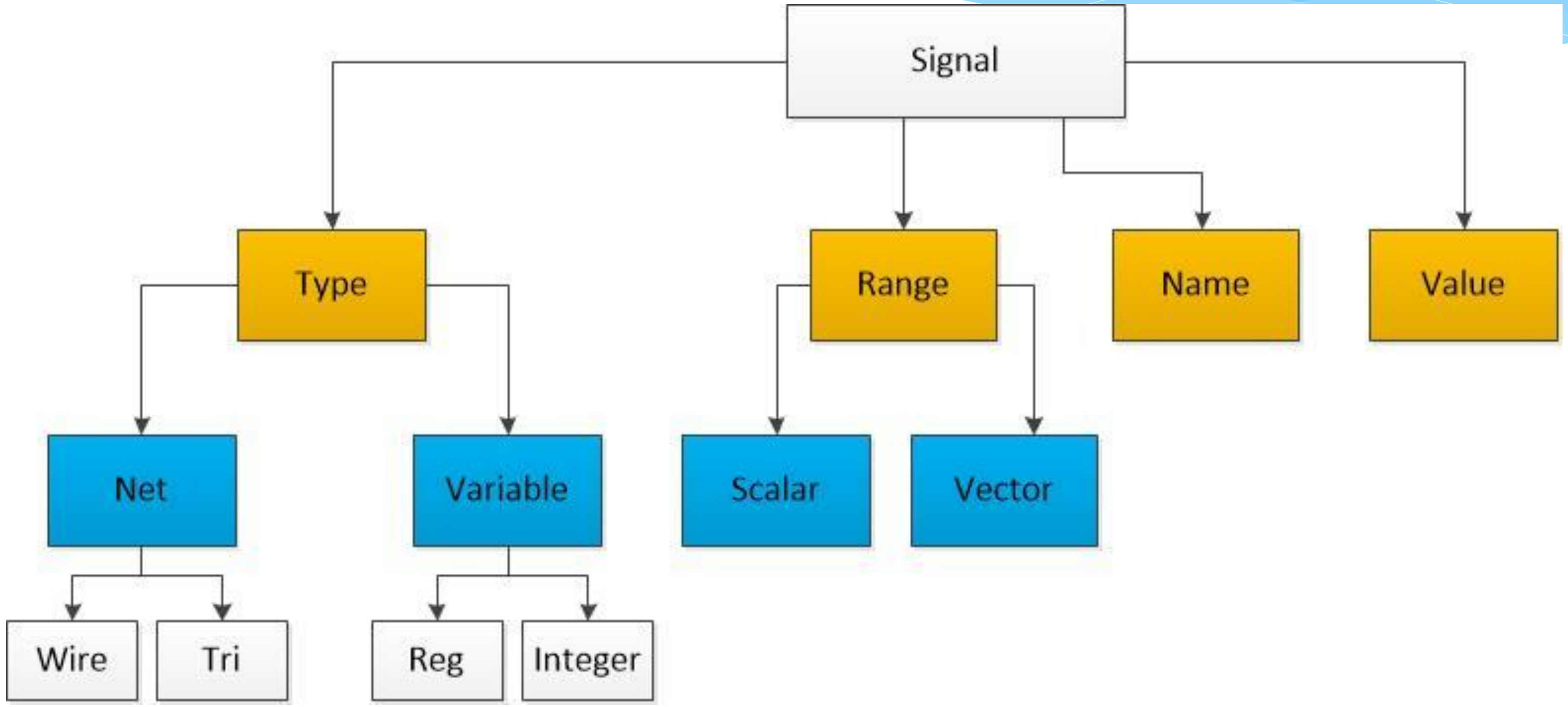


module



```
module   DUT ( A , B , C );
    input    A;
    output  B;
    inout    C;
             Signals
             Body-Code
 endmodule
```

Verilog Fundamentals: Signals



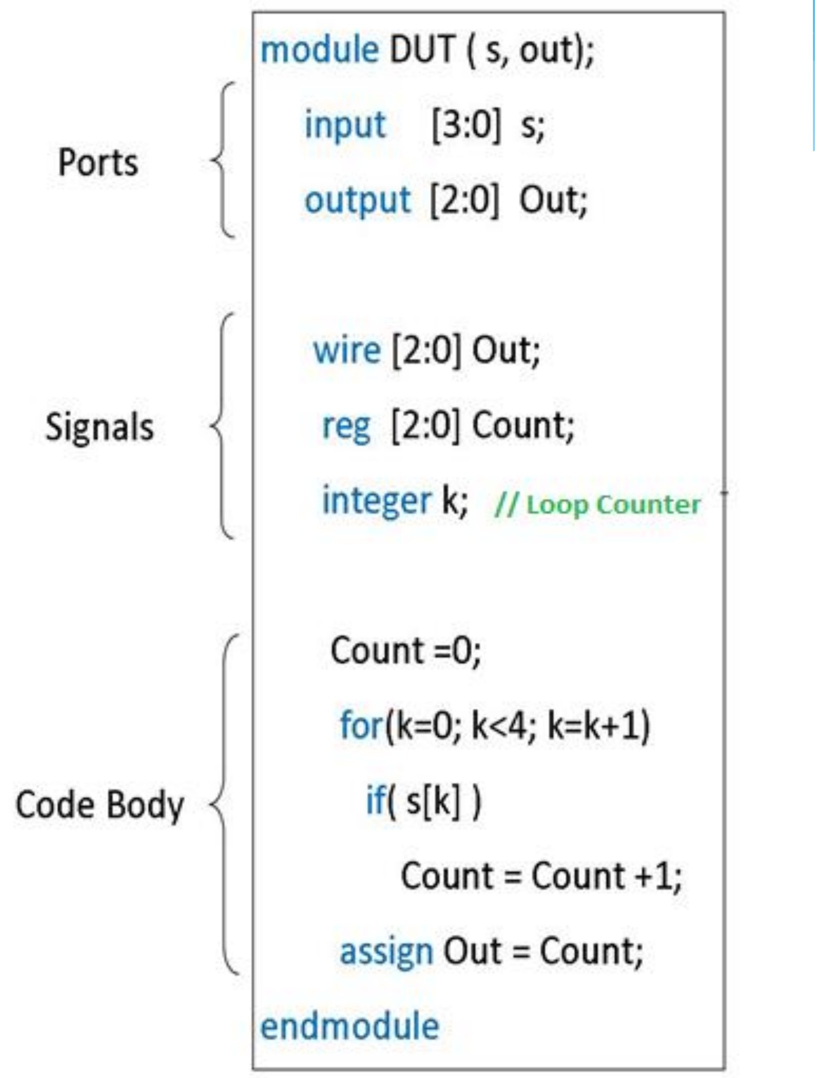
Verilog Fundamentals : Signal Type

- Net
  - wire:
    - For interconnecting logic elements (LEs)
    - To connect an output of a logic element to the input of another LE
  - Tri:
    - Circuit nodes that are connected in a tri-state fashion

- Variable
  - reg *(unsigned in general)*
    - Corresponds to a circuit node (*not necessarily a register!*)
    - Allow a circuit to be described in terms of its behavior
    - Retains its value until it is overwritten by a subsequent assignment
  - integer *(signed in general)*
    - Used for loop counters

Verilog Fundamentals : Signal Type

- The “wire” declarations are not necessary as Verilog assumes that signals are nets by default.
- The “reg” declaration is required.
- Don’t forget to use semicolon after module definition



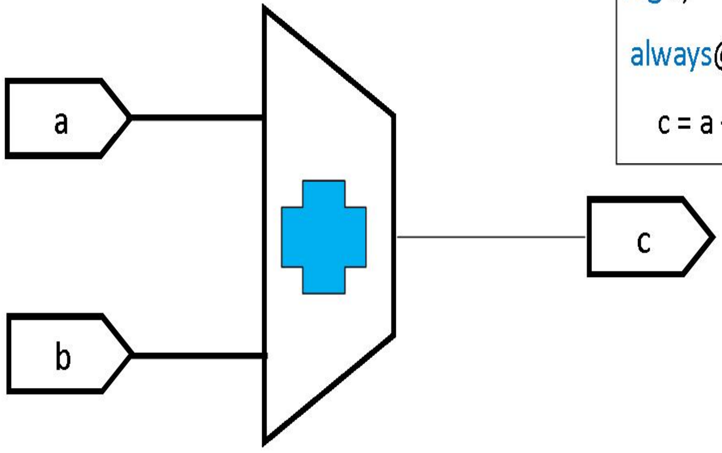
```
module DUT ( s, out);
   input   [3:0]  s;
   output  [2:0]  Out;

   wire [2:0] Out;
   reg  [2:0] Count;
   integer k;   // Loop Counter

    Count =0;
    for(k=0; k<4; k=k+1)
      if( s[k] )
          Count = Count +1;
    assign Out = Count;
endmodule
```

Reg Type

- The keyword “reg” does NOT necessarily denote a storage element or register.
- “reg” only models the behavior of a circuit.
- May or may not be synthesized as a register.

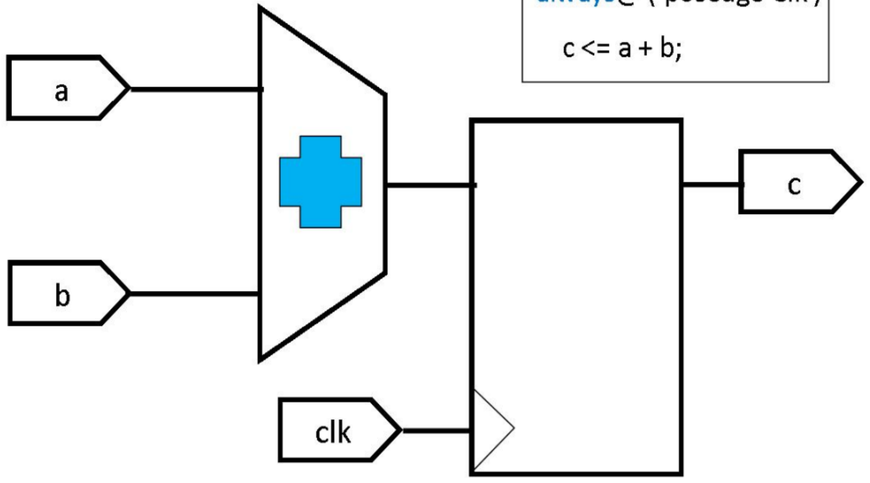
Not Register

```
reg c;
 always@ ( a, b)
   c = a + b;
```



Register

```
reg c;
 always@ ( posedge Clk )
   c <= a + b;
```



Verilog Fundamentals : Signal Range

- Signals in Verilog can be:
  - **Scalar** : representing a node

```
reg C;
wire B;
```

  - **Vector** : representing a bus

```
reg  [10:0] Data;
reg  [ 0:6]  S;
wire [ 7:4]  B;
```

- Each element of a bus can be accessed

```
assign a = Data[8];
```

Verilog Fundamentals: Signal Name

- Signal name may consist of:
  - Any letter
  - Any digit
  - Underscore (_) and $ sign

- DON'Ts:
  - Should not start with a digit
  - Should not be a Verilog keyword

Legal	Illegal
A_m	1xb
B1_signal	wire
My$	R&z

Verilog Fundamentals: Signal Value

- Scalar: each scalar signal can have four possible values:
  - 0 : Logic Value “0”
  - 1 : Logic Value “1”
  - Z(z) : Tri-State (high impedance) (such as open-circuit)
  - X(x) : Unknown value (such as conflict)

- Vector: <# of bits> <base> < number>
  - <# of bits> : number of bits for representation
  - <base> : “d” Decimal / “b” Binary / “h” Hexadecimal / “o” Octal
  - <number> : signal value in base

Verilog Fundamentals: Signal Value

- Example
  - A = 8’ha9 → A = 1010 1001
  - B = 4’d3 → B = 0011
  - C = 4’b100 → C = 0100
  - D = ‘b10x → D = (32-bit) 10X
  - E = -16’d47 → E = 1111 1111 1101 0001

Verilog Fundamentals : Parameters

- A parameter is used as a "constant" to facilitate coding
  - Example:

```
module DUT ( s , Out );

  parameter n   = 3;
  parameter S0 = 4'b1010;
  input   [n-1:0] s;
  output [   n:0] Out;
  wire     [n:0] Out;
  assign Out = S0;

endmodule
```

Verilog Fundamentals : Memories

- Memory:
  - A two dimensional array of bits
  - Declared in Verilog as a two-dimensional variable (reg)
  - Example: A 4-byte memory

$$\texttt{reg}\ \underbrace{[7:0]}_{\text{8-bit}}\ \texttt{R}\ \underbrace{[3:0]}_{\text{4 rows (cell)}}\ ;$$

	0	1	2	3	4	5	6	7
R[0]	R[0][0]							
R[1]						R[1][5]		
R[2]				R[2][3]				
R[3]								R[3][7]

Verilog Fundamentals : Memories

- A three-dimensional array may also be declared.
  - Example : reg [7:0] M [3:0][1:0];

- If an 8-bit A is declared then the legal assignment is:
  - reg [7:0] A → A = M[3][0];

Verilog Fundamentals : Operators

- Bitwise

Operation	Result
1010 & 1100	1000
1010 | 1100	1110
~ 1010	0101
1101 ^ 0100	1001

- Logical

Operation	Result
1010 && 1100	1
2’b11 || 2’b00	1
! 0010	0
2’b1X && 2’b11	X

If any operand is X/Z , result is Z

Verilog Fundamentals : Operators

❖Reduction

Operation	Result
& 1100	0
| 111	1
^ 0100	1

❖Relational

Operation	Result
2’b10 == 2’b10	1
2’b10 == 2’b11	0
2’b10 === 2’b1x	0
2’b10 <= 2’b11	1

== Used only with 0 and 1
=== Used with x and z

Verilog Fundamentals : Operators

❖Logical Shift

A = 6’b001100

Operation	Result
A >> 1	000110
A << 2	110000
A >> 3	000001

❖Concatenation

A = 2’b11
B = 3’b010

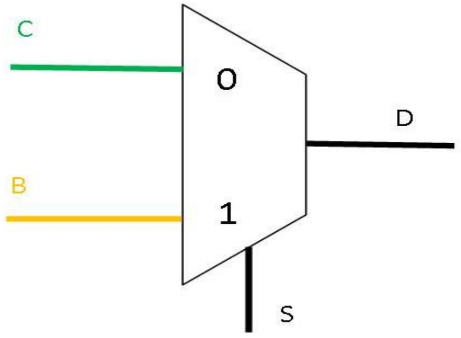
Operation	Result
{ A , B }	5’b11010
{ 3 { A } }	6’b111111
{ B , B }	6’b010010
{ { 3 { A } } , { 2 { B } } }	12’b111111010010

Verilog Fundamentals : Operators

- Conditional : (? , :)

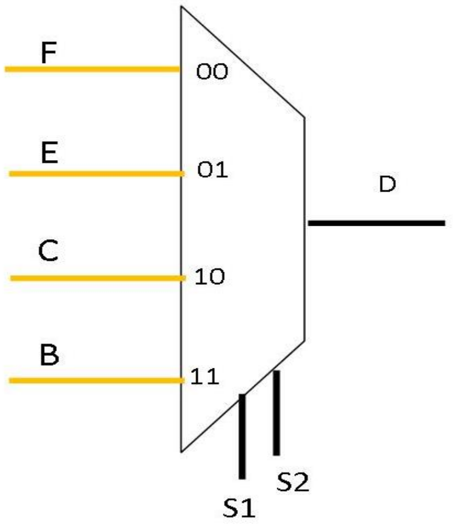
```
D = S ? B : C ;
```

$$D = \begin{cases} B \ \text{if} \ S = 1 ; \\ C \ \text{if} \ S = 0 ; \end{cases}$$



```
D = ( { S1 , S1 } == 2'b00 ) ? F:
    ( { S1 , S2 } == 2'b01 ) ? E :
    ( { S1 , S2 } == 2'b10 ) ? C : B ;
                                    default
```

```
D = ( { S1 , S1 } == 2'b00 ) ? F:
    ( { S1 , S2 } == 2'b01 ) ? E :
    ( { S1 , S2 } == 2'b10 ) ? C :
    ( { S1 , S2 } == 2'b11 )? B:B ;
                                    default
```



Verilog Fundamentals : Operators

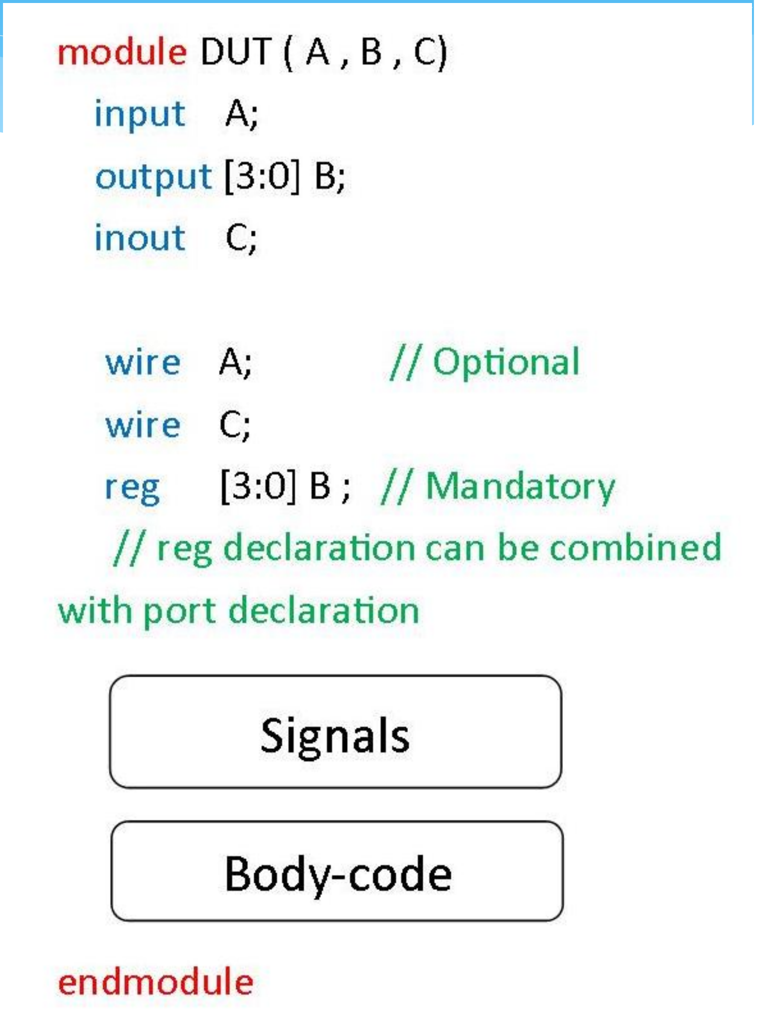
Type Of Operators	Symbols					
Concatenate & Replicate	{}	{{}}				
Unary	!	~	&	^	^~	|
Arithmetic	*	/	%			
	+	-				
Logical Shift	<<	>>				
Relational	<	<=	>	>=		
Equality	==	!=	===	!==		
Binary bit-wise	&	^	^~	|		
Binary logical	&&	||				
Conditional	?:					

Verilog Fundamentals : Module-Revisited

❖ Any circuit or subcircuit is declared as a "module" in Verilog

❖ There are three types of ports:

- input → type **"wire"**
- output → type **"wire"** or **"reg"**
- inout → type **"wire"**

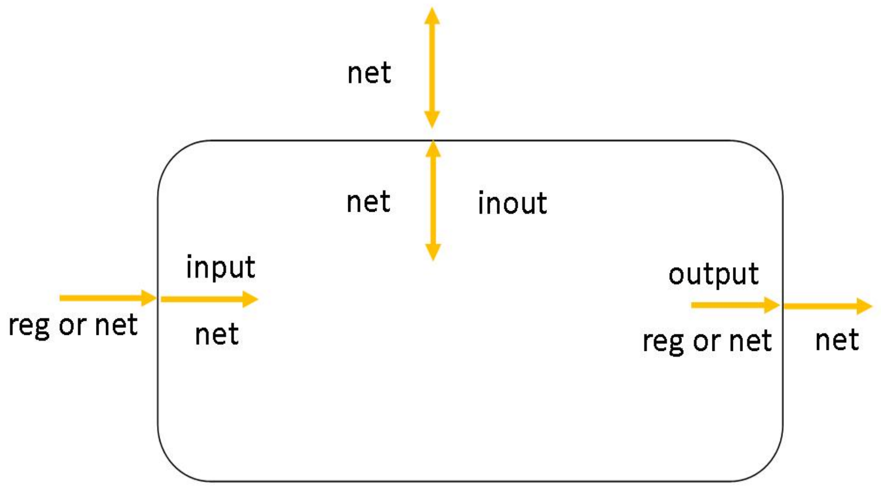


```
module DUT ( A , B , C)
   input   A;
   output [3:0] B;
   inout   C;

    wire   A;          // Optional
    wire   C;
    reg    [3:0] B ;  // Mandatory
     // reg declaration can be combined
with port declaration
```

```
endmodule
```

Verilog Fundamentals : Module Ports*

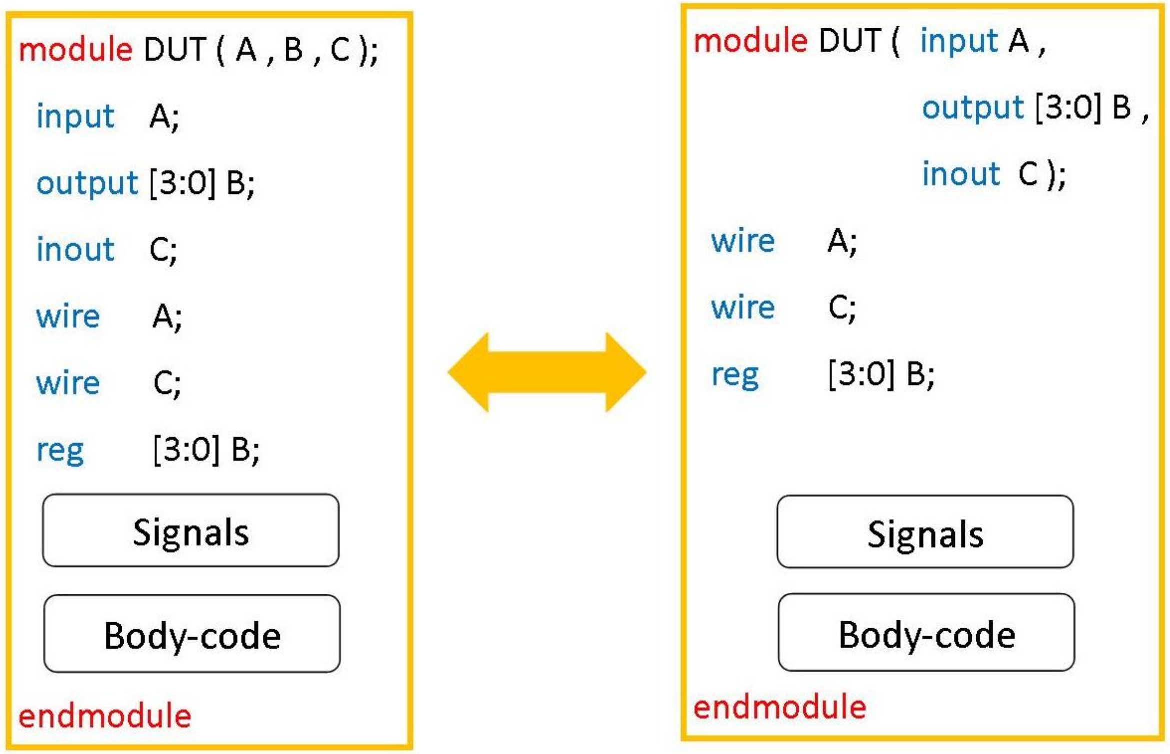
- Inside view of the module
  - inout : wire
  - input port : wire
  - **output port : wire or reg**
- Outside view of the module
  - inout : wire
  - **input port : wire or reg**
  - output port : wire



Verilog Fundamentals : Module-Revisited

- In Verilog-2001 the port list can directly follow the module declaration

- Body-code consist of some **“statements”**



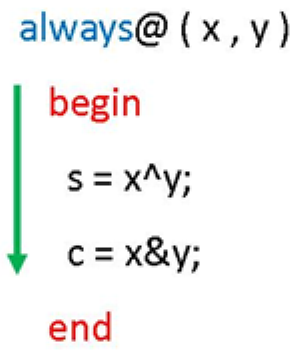
```
module DUT ( A , B , C );
  input   A;
  output [3:0] B;
  inout   C;
  wire    A;
  wire    C;
  reg     [3:0] B;
   Signals
   Body-code
endmodule
```

```
module DUT ( input A ,
             output [3:0] B ,
             inout  C );
  wire    A;
  wire    C;
  reg     [3:0] B;
   Signals
   Body-code
endmodule
```

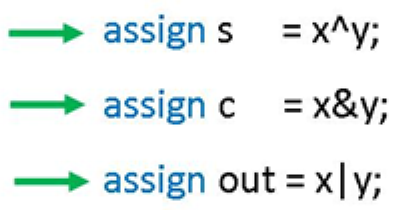
- Statements describe the circuit/module functionality

Verilog Fundamentals : Statements

- Programming languages:
  - **High-Level Language (HLL) :** C, Pascal, Matlab
  - **Hardware Description Language (HDL) :** Verilog, HDL
- In HLL programming all statements are sequential (procedural)
  - Statements evaluated in the order and one-bye-one
- Verilog Statements
  - **Procedural :** evaluated sequentially (Order IS important)
  - **Concurrent :** evaluated in parallel (Order is NOT important)



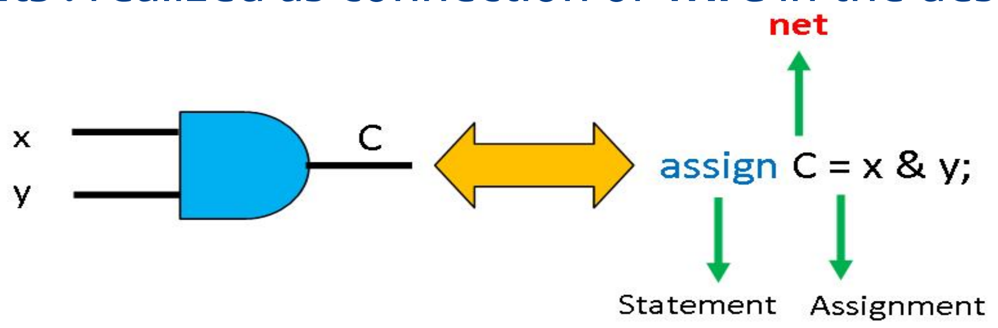
```
always@ ( x , y )
begin
  s = x^y;
  c = x&y;
end
```



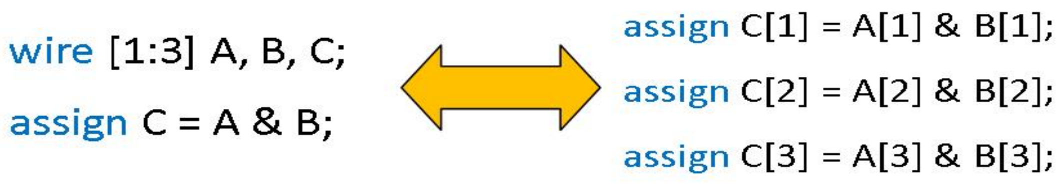
```
assign s   = x^y;
assign c   = x&y;
assign out = x|y;
```

Verilog Fundamentals : Concurrent Statements

- Evaluated in parallel
- Each statement describes part of the circuit, thus concurrent
- Most popular:
  - **Continuous statements :** realized as connection or **wire** in the design
  - **Format :**



```
assign C = x & y;
```



```
wire [1:3] A, B, C;
assign C = A & B;
```

```
assign C[1] = A[1] & B[1];
assign C[2] = A[2] & B[2];
assign C[3] = A[3] & B[3];
```

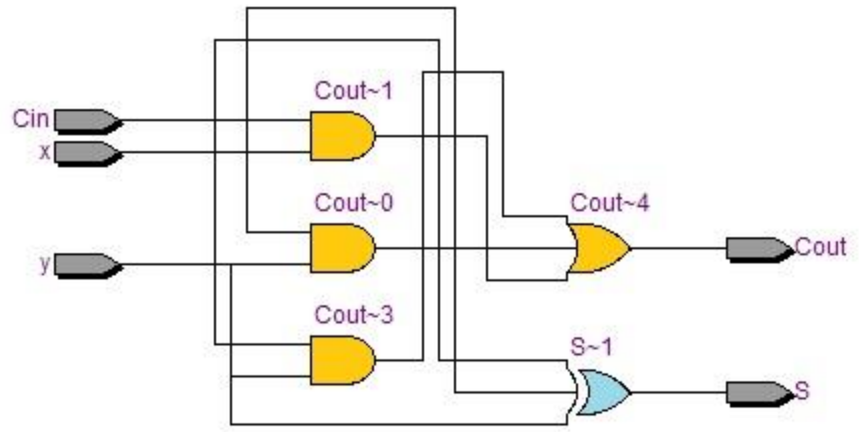
  - **assign** used only for nets (to be synthesizable)

Concurrent Statements

- Example : Full Adder, same circuit, two descriptions:

```
module Adder ( Cin, x, y, S, Cout );
   input    x, y, Cin;
   output  S, Cout;
   wire     S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x&y) | (x&Cin) | (y&Cin)

endmodule
```

```
module Adder ( Cin, x, y, S, Cout );
   input    x, y, Cin;
   output  S, Cout;
   wire     S, Cout;

   assign  { Cout , S } = x + y + Cin;

endmodule
```



x	y	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Concurrent Statements

- In Verilog “+” declares unsigned addition
- Signed addition has to be explicitly specified using the sign extension
- **Example:** Signed vs. unsigned addition

```
module Adder_sign ( X, Y, S_unsigned, S_signed );
  input   [3:0] X, Y;
  output [4:0] S_unsigned, S_signed;
  assign S_unsigned = X + Y ;
  assign S_signed    = {{X[3]},X} + {{Y[3]}+Y} // Sign Extension
endmodule
```

```
module Adder_sign ( X, Y, S_unsigned, S_signed );
  parameter n = 4;
  input   [n-1:0] X, Y ;
  output [n:0] S_unsigned , S_signed ;
  assign S_unsigned = X + Y;
  assign S_signed    = {{X[n-1]} , X} + {{Y[n-1] , Y};
endmodule
```

X = 0100 (unsigned 4) or (signed +4)
Y = 1100 (unsigned 12) or (signed -4)

S_unsigned = 10000 (unsigned 16)
S_signed = 00000 (signed 0)

Procedural Statements

- Evaluated in the order in which they appear in the code (sequential)
- Should be inside an **"always"** block
- An **"always"** block contains one or more procedural statements

always@ (sensitivity list) → List of all signals that trigger the evaluation inside the always block

begin

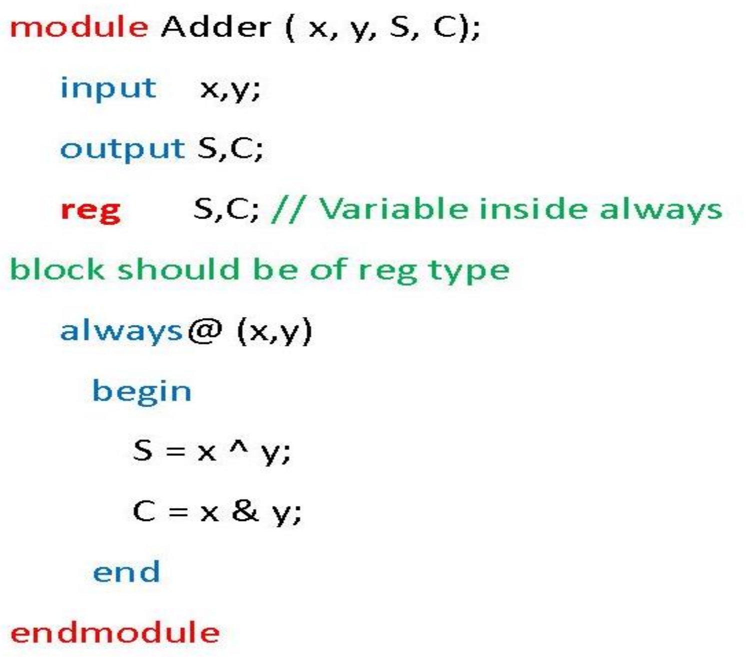
- Procedural assignments
- **if-else** statements
- **case** statements
- **while , repeat, for** loops

} Procedural Statements

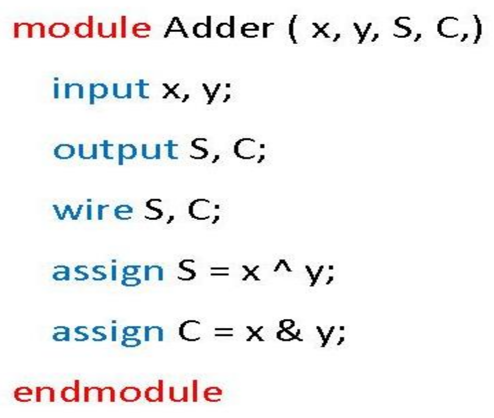
end

Procedural Statements : Half Adder

- Anything on the RHS should be on the sensitivity list
  - always @(*) → Automatically considers all signals on the RHS in sensitivity list
- Any signal assigned inside an always block has to be a variable of **reg** or **integer** type



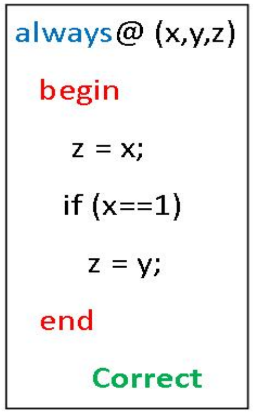
```
module Adder ( x, y, S, C);
   input    x,y;
   output S,C;
   reg      S,C; // Variable inside always
block should be of reg type
   always@ (x,y)
     begin
        S = x ^ y;
        C = x & y;
     end
endmodule
```



```
module Adder ( x, y, S, C,)
   input x, y;
   output S, C;
   wire S, C;
   assign S = x ^ y;
   assign C = x & y;
endmodule
```

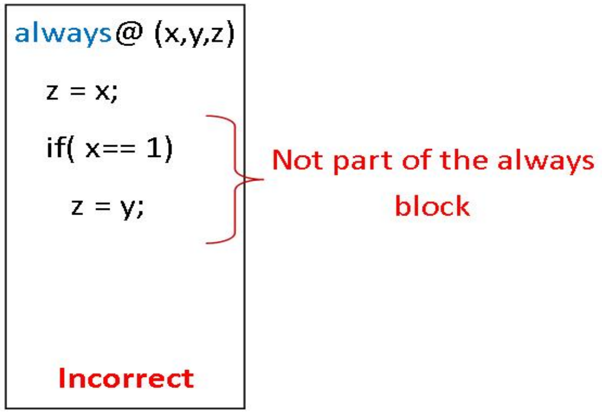
Always block

- The always construct requires begin-end only if there are multiple statements in the block



```
always@ (x,y,z)
 begin
    z = x;
    if (x==1)
      z = y;
 end
```

Correct

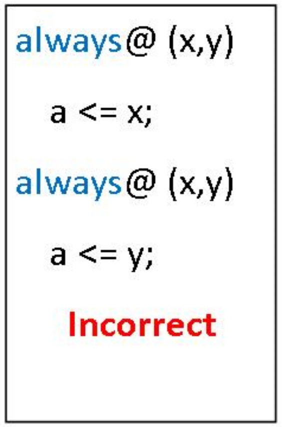


```
always@ (x,y,z)
   z = x;
   if( x== 1)
     z = y;
```

Not part of the always block

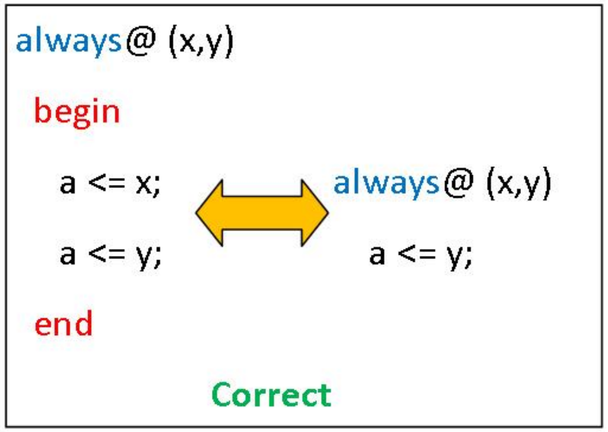
Incorrect

- A given variable should never be assigned a value in more than one always block because **always** blocks are **concurrent** with respect to **one another**



```
always@ (x,y)
   a <= x;
always@ (x,y)
   a <= y;
```

Incorrect



```
always@ (x,y)
 begin
   a <= x;               always@ (x,y)
   a <= y;                  a <= y;
 end
```

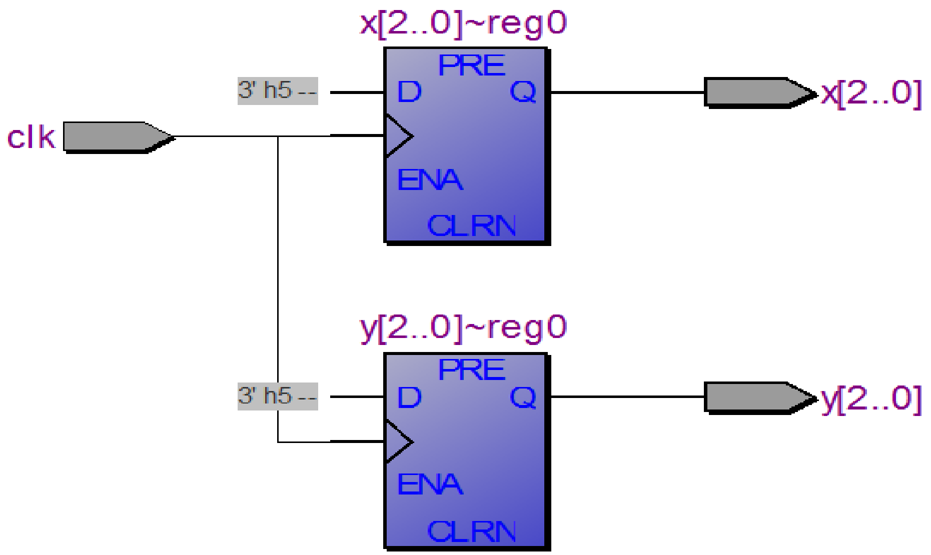
Correct

Procedural Assignments

- Used inside an always block and are of two types:
  - Blocking: denoted by “=“ token
    - Evaluation within the always block is “blocked” until this assignments is completed
  - Non-blocking: denoted by “<=“ token
    - Nothing is hold or blocked (parallel evaluation)

Blocking vs. Non-Blocking Assignments

- **Blocking**
  - Evaluated and assigned in a single step
  - Sequential nature
  - Assignment ordering IS important
  - S=5 “blocks” y=x to be executed
    - y=x has to wait for x=5 to be executed first



assume x=2 then

```
always@ (*)
  begin
    x = 5;
    y = x;
  end
```

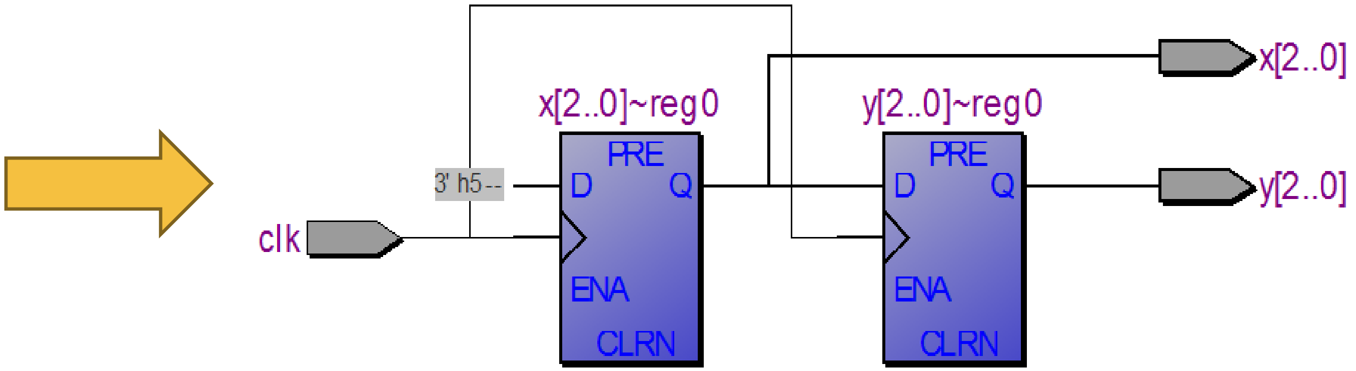
x = 5 & y = 5

(sequential)

Blocking vs. Non-Blocking Assignments

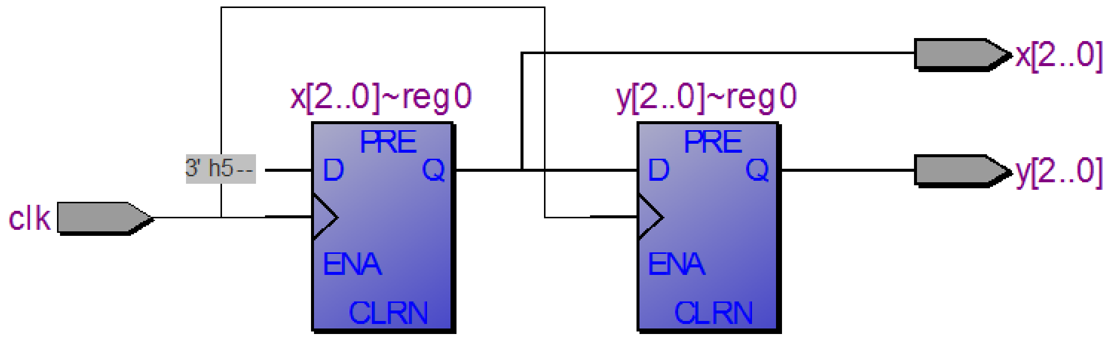
- What happens if we change order of two blocking assignments in previous slide?

```
assume x= 2 then
always@(*)
  begin
    y = x;
    x = 5;
  end
```



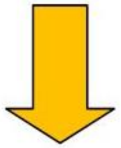
Blocking vs. Non-Blocking Assignments

- **Non-Blocking**
  - Evaluated and assigned in two steps
    1. All RHSs are evaluated in parallel
    2. Assignments to LHS are performed together
  - They all are executed all at once
  - Assignment ordering is NOT important
  - y <= x and x <= 5 executed in parallel



assume x=2 then

```
always@ (*)
  begin
    x <= 5;
    y <= x;
  end
```

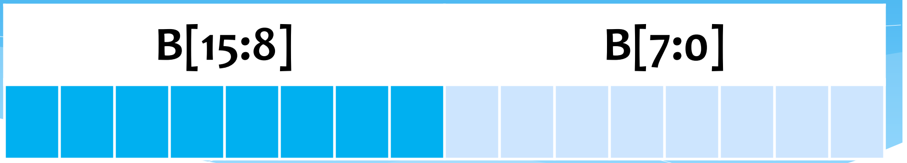


x = 5 & y = 2

(parallel)

Blocking vs. Non-Blocking Assignments

- Example: Swap bytes in words



Blocking

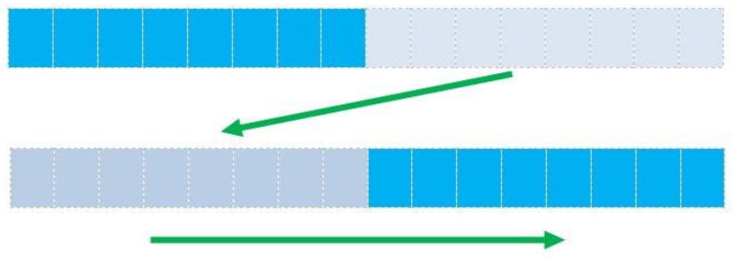
```
always @(*)
  begin
     B[15:8] = B[   7:0] ;
     B[ 7: 0]  = B[15:8] ;
  end
```

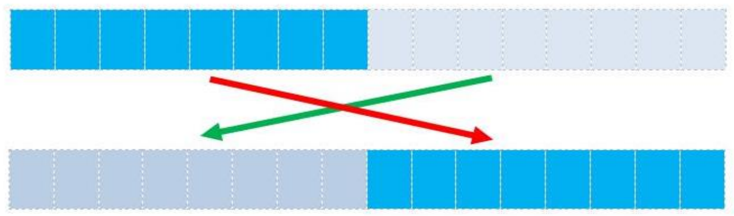
Incorrect

Non-Blocking

```
always @(*)
  begin
     B[15:8] <= B[   7:0] ;
     B[ 7: 0]  <= B[15:8] ;
  end
```

Correct

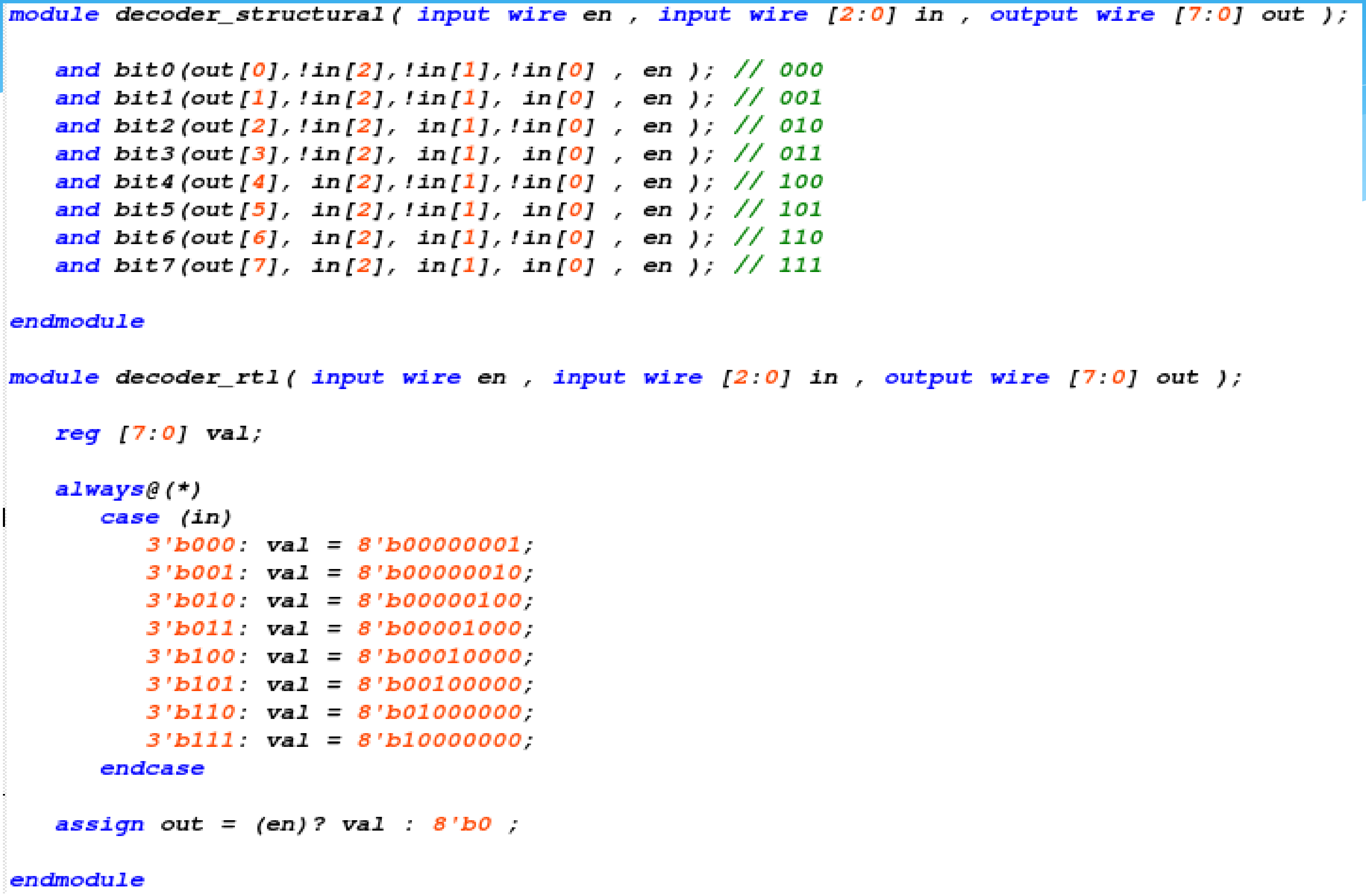




Description of some common logic circuits in Verilog

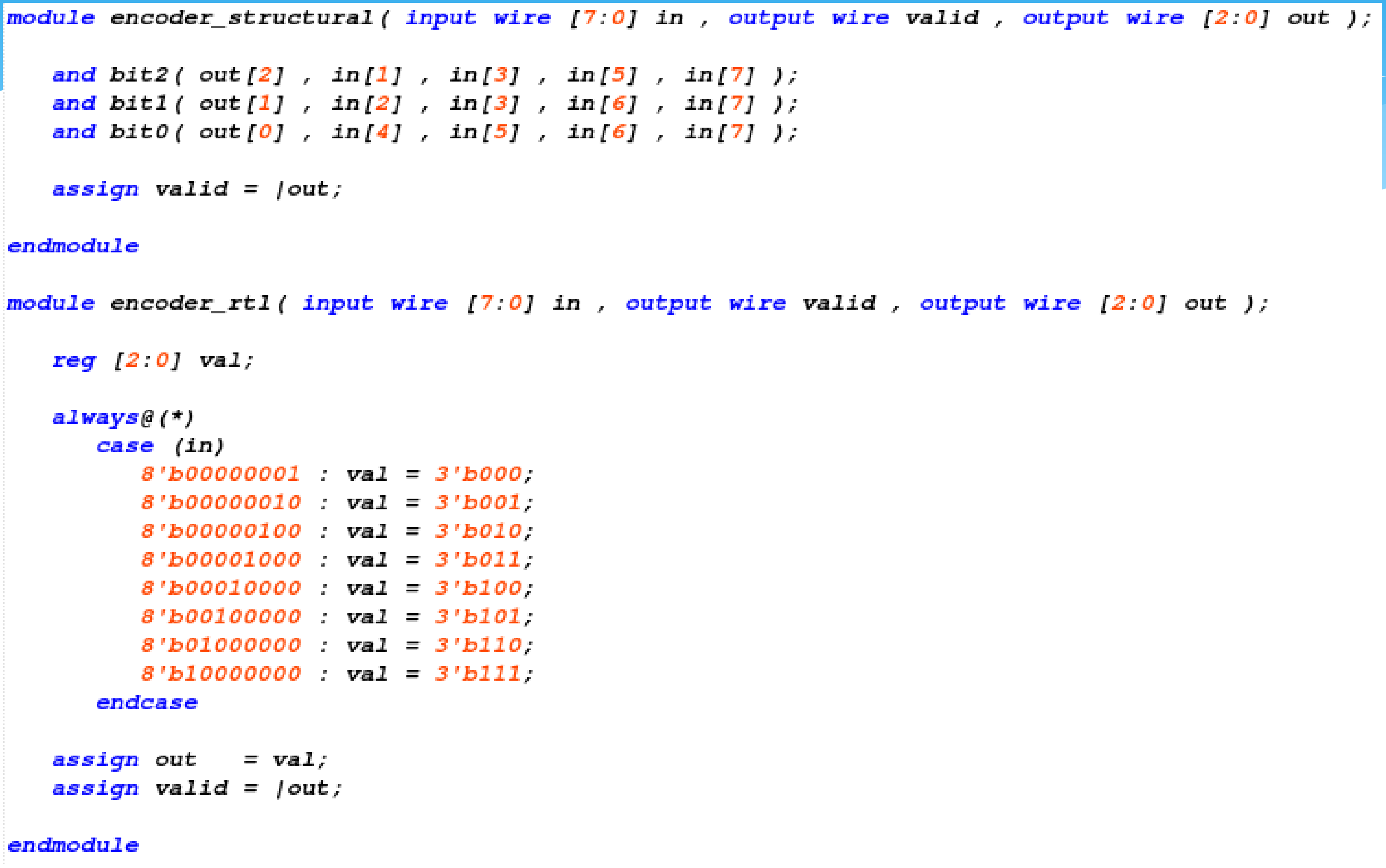
* Decoder
* Encoder
* Multiplexer
* Demultiplexer
* Parity Generator
* Arithmetic & Logic Unit (ALU)

Decoder



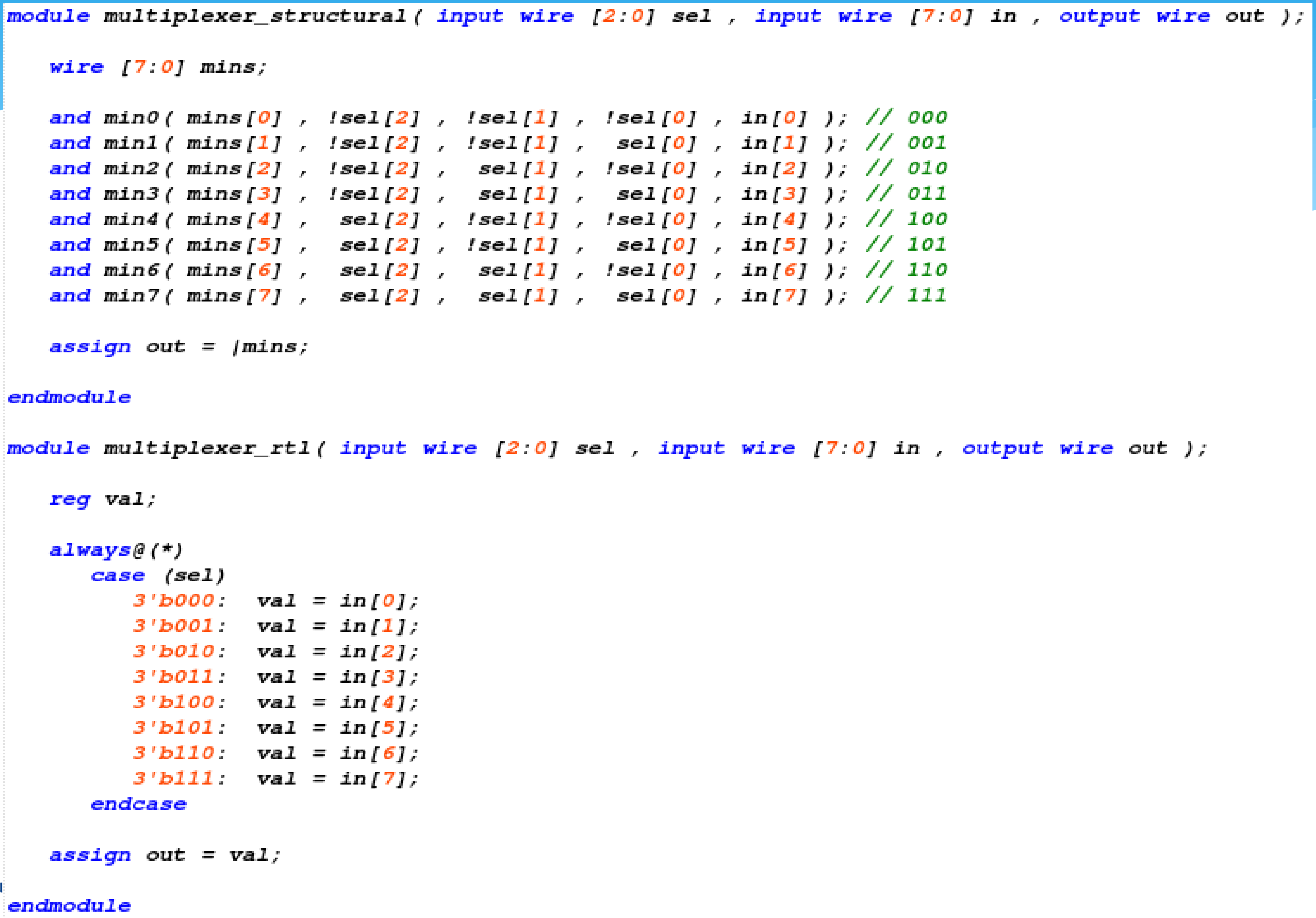
```
module decoder_structural( input wire en , input wire [2:0] in , output wire [7:0] out );

   and bit0(out[0],!in[2],!in[1],!in[0] , en ); // 000
   and bit1(out[1],!in[2],!in[1], in[0] , en ); // 001
   and bit2(out[2],!in[2], in[1],!in[0] , en ); // 010
   and bit3(out[3],!in[2], in[1], in[0] , en ); // 011
   and bit4(out[4], in[2],!in[1],!in[0] , en ); // 100
   and bit5(out[5], in[2],!in[1], in[0] , en ); // 101
   and bit6(out[6], in[2], in[1],!in[0] , en ); // 110
   and bit7(out[7], in[2], in[1], in[0] , en ); // 111

endmodule

module decoder_rtl( input wire en , input wire [2:0] in , output wire [7:0] out );

   reg [7:0] val;

   always@(*)
      case (in)
         3'b000: val = 8'b00000001;
         3'b001: val = 8'b00000010;
         3'b010: val = 8'b00000100;
         3'b011: val = 8'b00001000;
         3'b100: val = 8'b00010000;
         3'b101: val = 8'b00100000;
         3'b110: val = 8'b01000000;
         3'b111: val = 8'b10000000;
      endcase

   assign out = (en)? val : 8'b0 ;

endmodule
```

Encoder



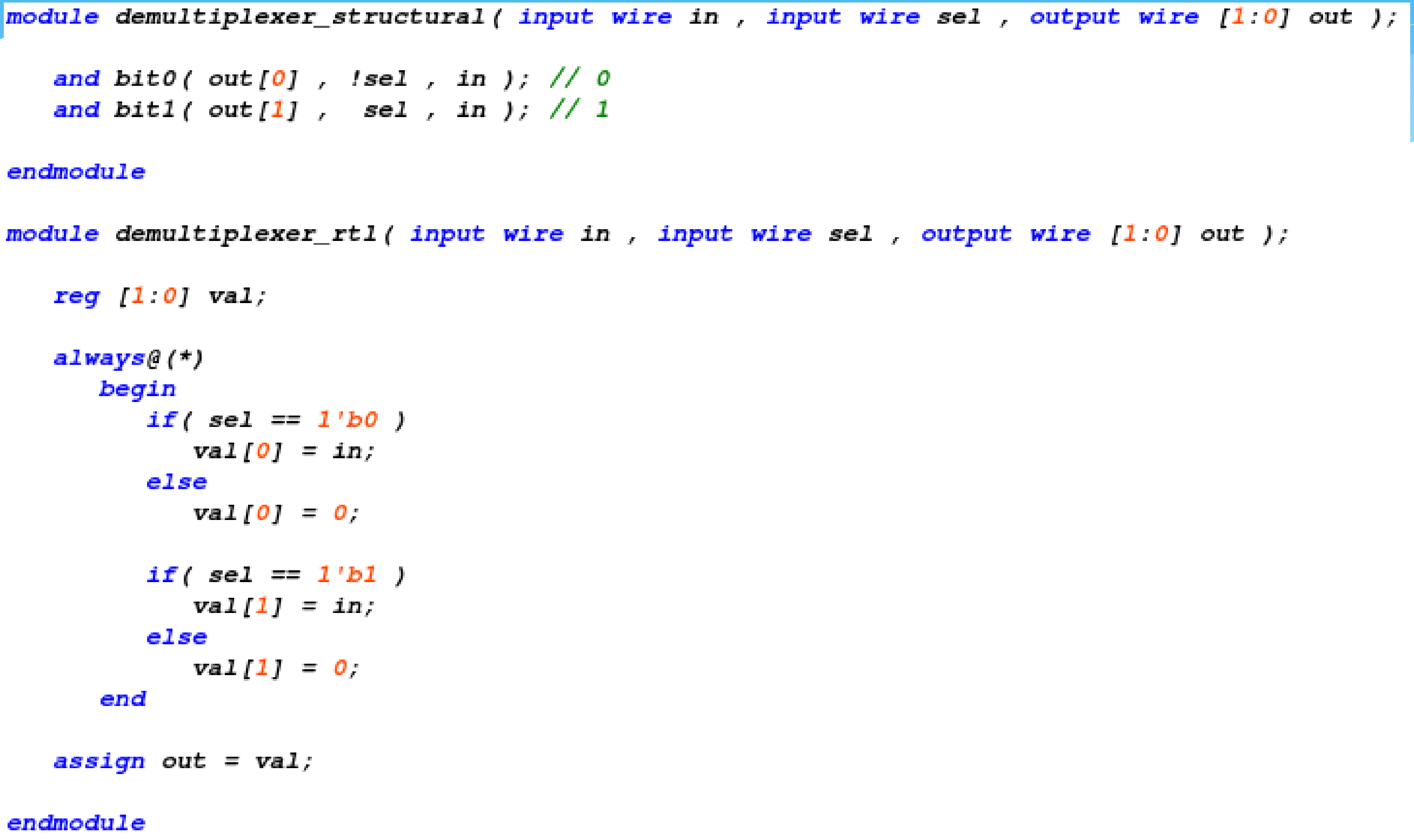
```verilog
module encoder_structural( input wire [7:0] in , output wire valid , output wire [2:0] out );

   and bit2( out[2] , in[1] , in[3] , in[5] , in[7] );
   and bit1( out[1] , in[2] , in[3] , in[6] , in[7] );
   and bit0( out[0] , in[4] , in[5] , in[6] , in[7] );

   assign valid = |out;

endmodule

module encoder_rtl( input wire [7:0] in , output wire valid , output wire [2:0] out );

   reg [2:0] val;

   always@(*)
      case (in)
         8'b00000001 : val = 3'b000;
         8'b00000010 : val = 3'b001;
         8'b00000100 : val = 3'b010;
         8'b00001000 : val = 3'b011;
         8'b00010000 : val = 3'b100;
         8'b00100000 : val = 3'b101;
         8'b01000000 : val = 3'b110;
         8'b10000000 : val = 3'b111;
      endcase

   assign out   = val;
   assign valid = |out;

endmodule
```

Multiplexer



```
module multiplexer_structural( input wire [2:0] sel , input wire [7:0] in , output wire out );

   wire [7:0] mins;

   and min0( mins[0] , !sel[2] , !sel[1] , !sel[0] , in[0] ); // 000
   and min1( mins[1] , !sel[2] , !sel[1] ,  sel[0] , in[1] ); // 001
   and min2( mins[2] , !sel[2] ,  sel[1] , !sel[0] , in[2] ); // 010
   and min3( mins[3] , !sel[2] ,  sel[1] ,  sel[0] , in[3] ); // 011
   and min4( mins[4] ,  sel[2] , !sel[1] , !sel[0] , in[4] ); // 100
   and min5( mins[5] ,  sel[2] , !sel[1] ,  sel[0] , in[5] ); // 101
   and min6( mins[6] ,  sel[2] ,  sel[1] , !sel[0] , in[6] ); // 110
   and min7( mins[7] ,  sel[2] ,  sel[1] ,  sel[0] , in[7] ); // 111

   assign out = |mins;

endmodule

module multiplexer_rtl( input wire [2:0] sel , input wire [7:0] in , output wire out );

   reg val;

   always@(*)
      case (sel)
         3'b000:  val = in[0];
         3'b001:  val = in[1];
         3'b010:  val = in[2];
         3'b011:  val = in[3];
         3'b100:  val = in[4];
         3'b101:  val = in[5];
         3'b110:  val = in[6];
         3'b111:  val = in[7];
      endcase

   assign out = val;

endmodule
```

Demultiplexer

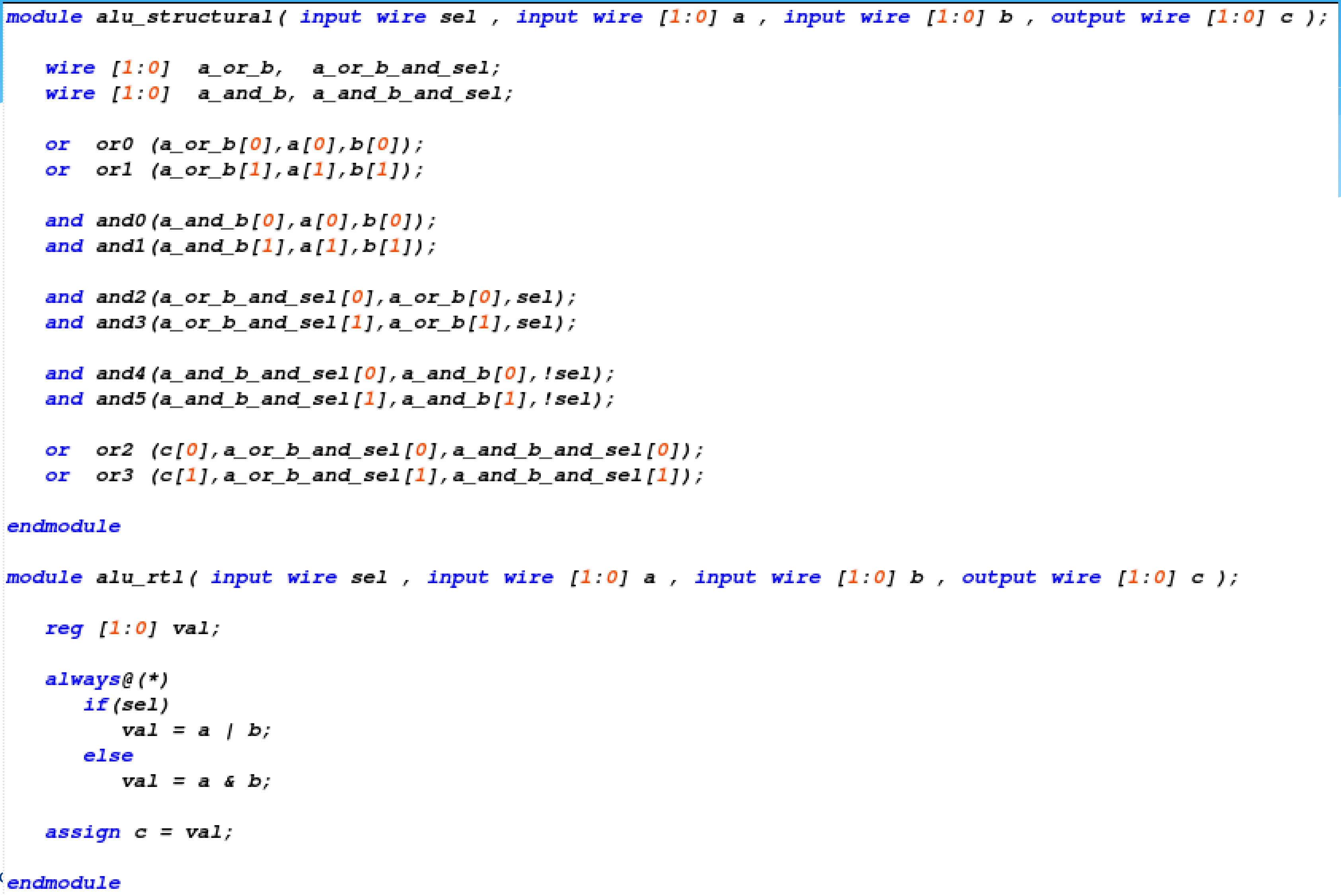


```
module demultiplexer_structural( input wire in , input wire sel , output wire [1:0] out );

   and bit0( out[0] , !sel , in ); // 0
   and bit1( out[1] ,  sel , in ); // 1

endmodule

module demultiplexer_rtl( input wire in , input wire sel , output wire [1:0] out );

   reg [1:0] val;

   always@(*)
      begin
         if( sel == 1'b0 )
            val[0] = in;
         else
            val[0] = 0;

         if( sel == 1'b1 )
            val[1] = in;
         else
            val[1] = 0;
      end

   assign out = val;

endmodule
```

Parity Generator

```
module parity_gen_structural( input wire [3:0] in , output wire even_parity , output wire odd_parity );

      xor xor1( even_parity , in[3] , in[2] , in[1] , in[0] );

      assign odd_parity = !even_parity;

endmodule

module parity_gen_rtl( input wire [3:0] in , output wire even_parity , output wire odd_parity );

   reg   val;

   always@(*)
      if( ^in )
         val = 1'b1;
      else
         val = 1'b0;

   assign odd_parity  = !val;
   assign even_parity = val;

endmodule
```

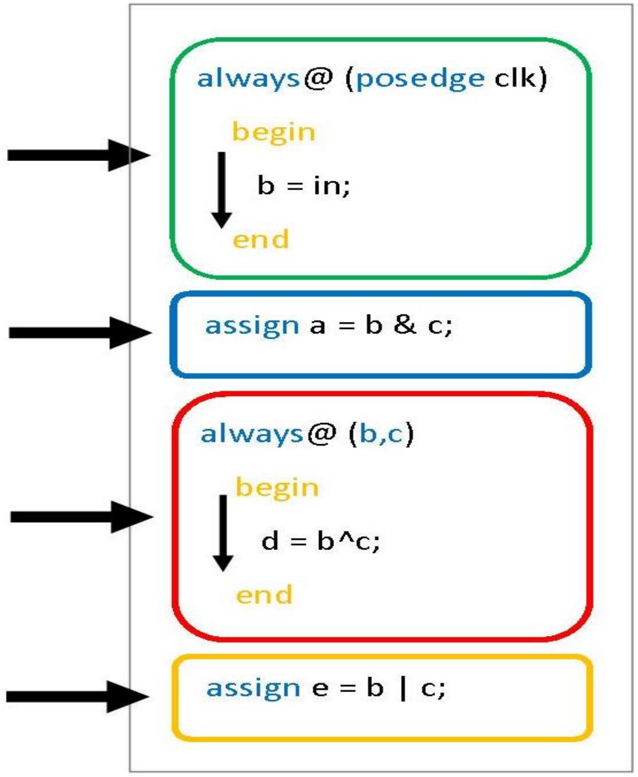
Arithmetic and Logic Unit



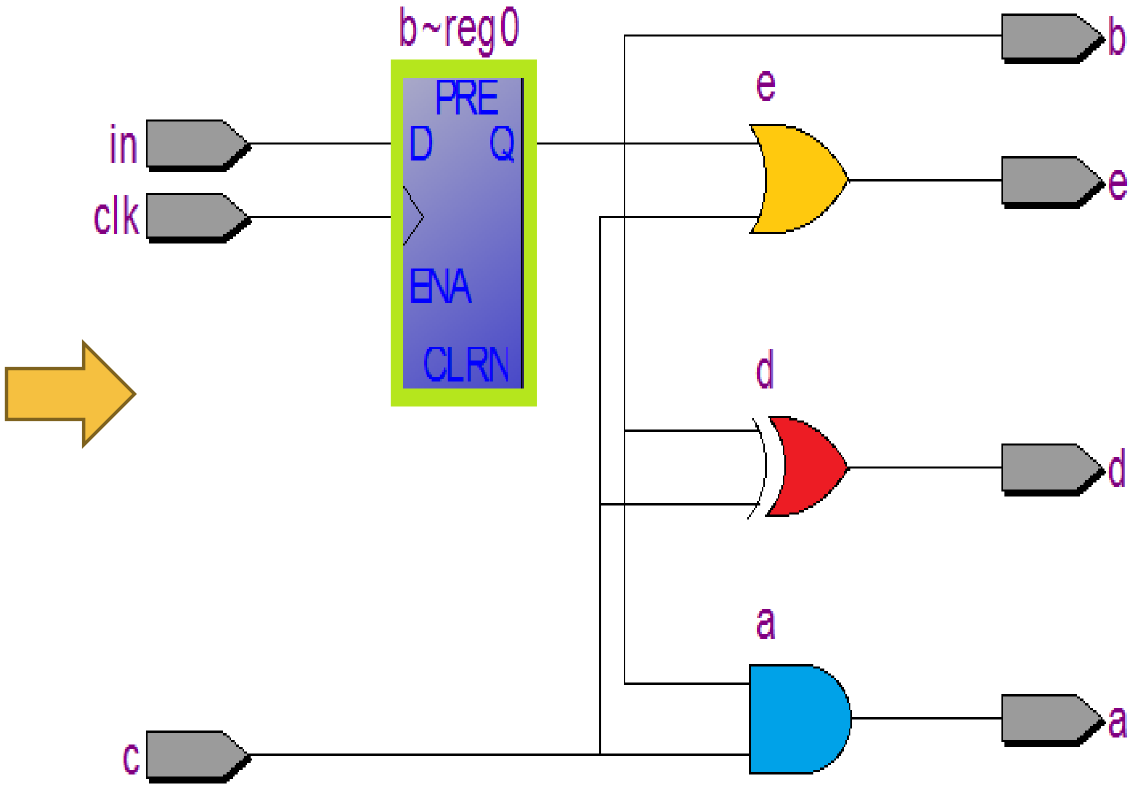
```
module alu_structural( input wire sel , input wire [1:0] a , input wire [1:0] b , output wire [1:0] c );

   wire [1:0]  a_or_b,  a_or_b_and_sel;
   wire [1:0]  a_and_b, a_and_b_and_sel;

   or   or0  (a_or_b[0],a[0],b[0]);
   or   or1  (a_or_b[1],a[1],b[1]);

   and  and0(a_and_b[0],a[0],b[0]);
   and  and1(a_and_b[1],a[1],b[1]);

   and  and2(a_or_b_and_sel[0],a_or_b[0],sel);
   and  and3(a_or_b_and_sel[1],a_or_b[1],sel);

   and  and4(a_and_b_and_sel[0],a_and_b[0],!sel);
   and  and5(a_and_b_and_sel[1],a_and_b[1],!sel);

   or   or2  (c[0],a_or_b_and_sel[0],a_and_b_and_sel[0]);
   or   or3  (c[1],a_or_b_and_sel[1],a_and_b_and_sel[1]);

endmodule

module alu_rtl( input wire sel , input wire [1:0] a , input wire [1:0] b , output wire [1:0] c );

   reg [1:0] val;

   always@(*)
      if(sel)
         val = a | b;
      else
         val = a & b;

   assign c = val;

endmodule
```

Overall Code Parallelism

- Statements inside an always block are evaluated sequentially
- However, all always blocks are evaluated concurrently
- All continuous assignments are evaluated concurrently too



```
always@ (posedge clk)
  begin
    b = in;
  end

assign a = b & c;

always@ (b,c)
  begin
    d = b^c;
  end

assign e = b | c;
```



Verilog Assignments

- Procedural

  Inside an always block

  - Blocking (=)
  - Non-Blocking (<=)
    - **Left** side of **procedural** assignment should be variable of **reg** or **integer** type

- Continuous

  Using assign statement (assign)

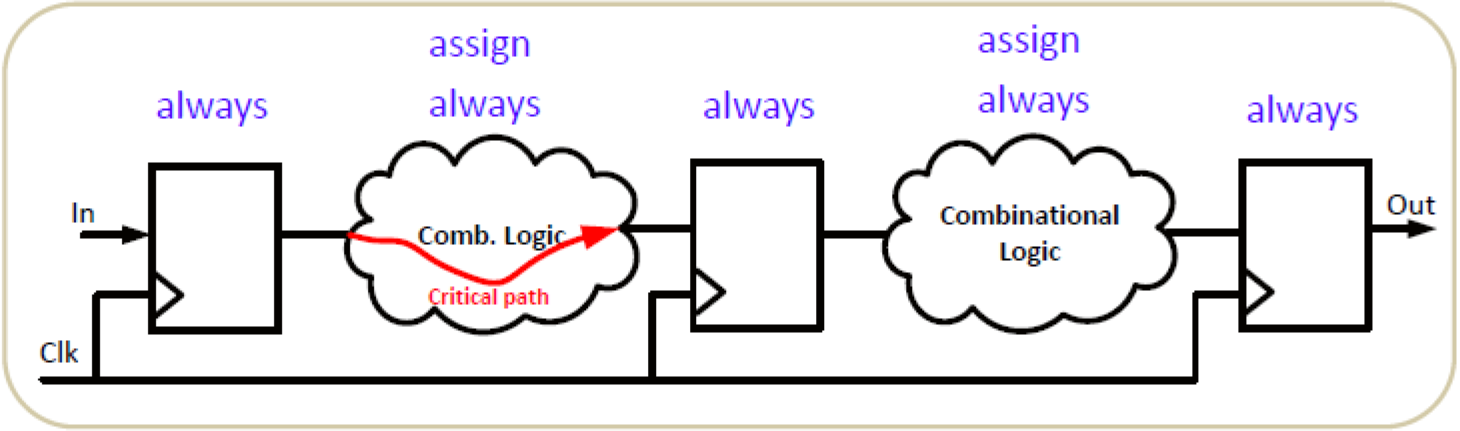
  - **Left** side of **continuous** assignment should be variable of **net** type

Logic Circuits Category

- Combinational Logic: (realized by **assign** and **always**)
  - Output depends on inputs
  - Inputs propagates to the output through some gates with delay
  - for example : adder, mux , multiplier, all logic gates
- Sequential Logic: (realized only by **always**)
  - Output depends on inputs and history circuit
  - Circuit history is kept using flip-flops, registers or latches
  - for example : Finite State Machines (FSM) , shift registers, Flip Flops
  - is divided into two groups:
    1. **Synchronous** : all registers controlled by a global clock
    2. **Asynchronous** : based on the handshaking process

Logic Circuits Category

- A general system consists of both combinational and sequential circuits

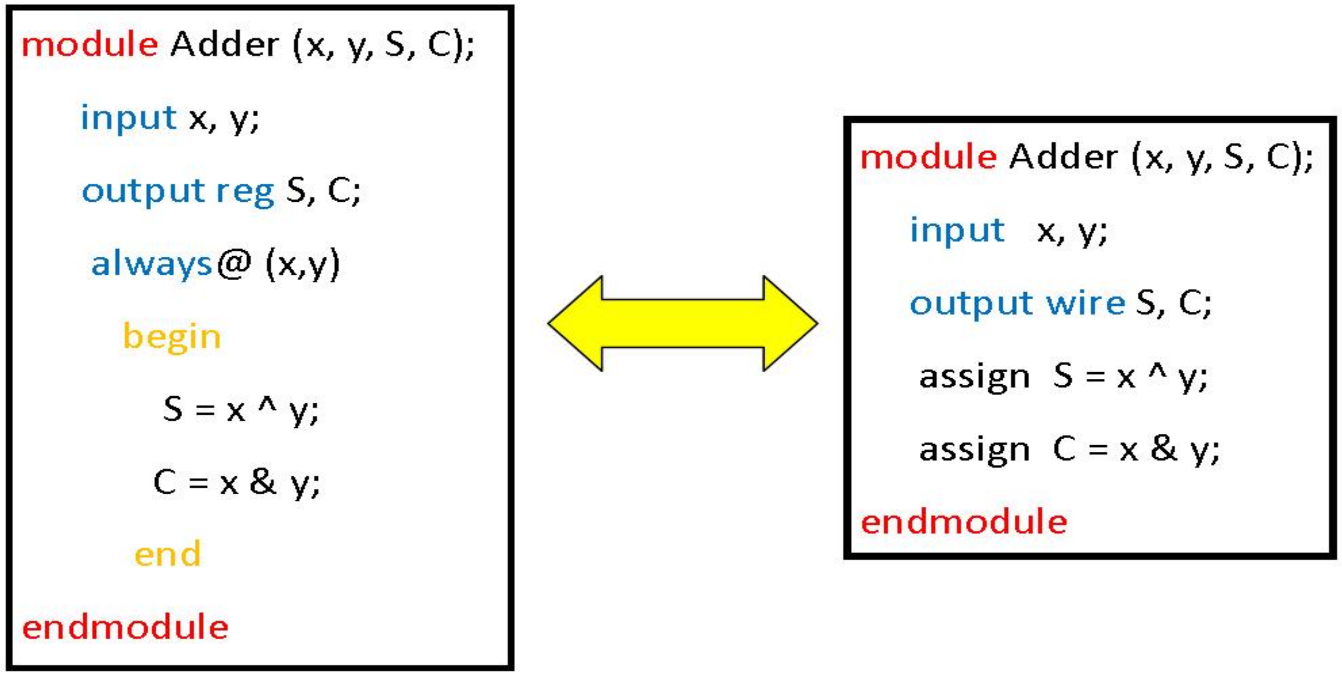


- Critical path of the Comb. Logic determines the max operating frequency
- Combinational logic can be realized using **assign** and **always** constructs
- Sequential logic can only be realized using **always** blocks

Combinational Logic

- When using always block for Com. Logic, **“blocking”** assignments are used
- When using an always block, time instant changes when one of the sensitivity list variables changes
- Example:

  Half Adder

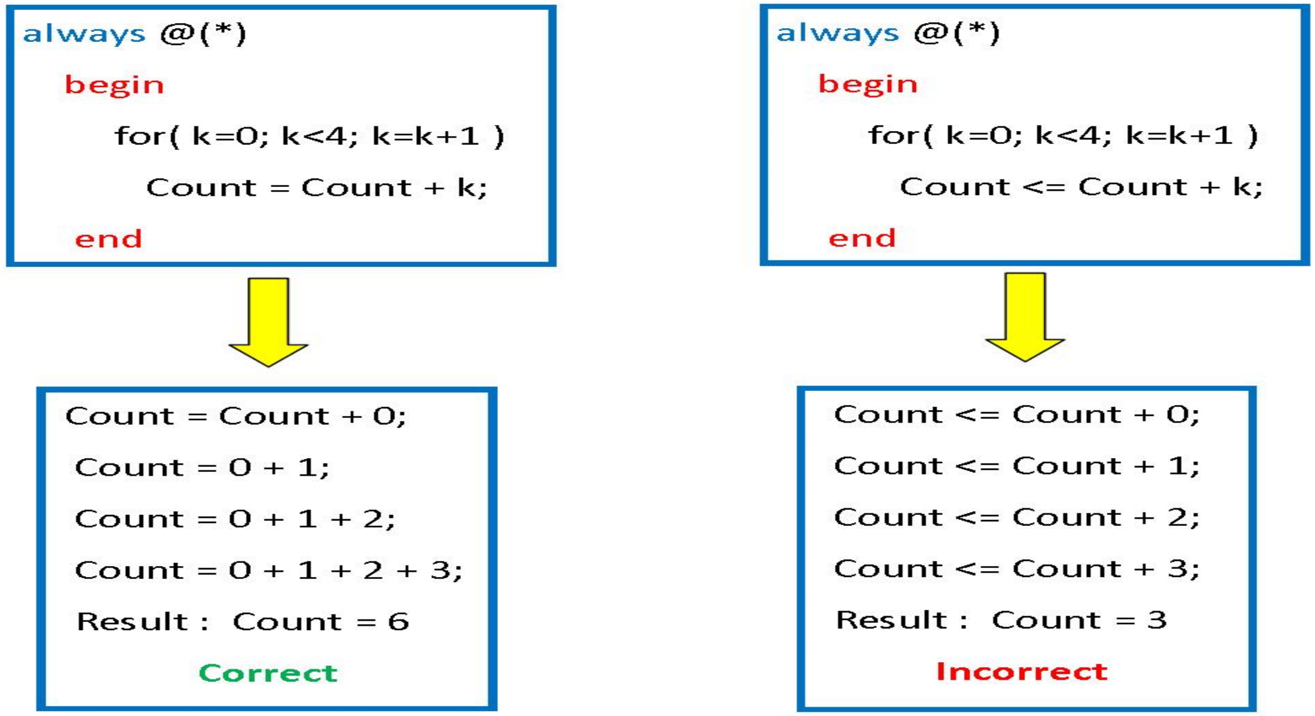


```
module Adder (x, y, S, C);
     input x, y;
     output reg S, C;
      always@ (x,y)
         begin
              S = x ^ y;
             C = x & y;
          end
endmodule
```

```
module Adder (x, y, S, C);
     input   x, y;
     output wire S, C;
      assign  S = x ^ y;
      assign  C = x & y;
endmodule
```

Blocking Assignment for Combinational Logic

- Use only **blocking** assignments for **combinational** logic.
- In **multiple concurrent non-blocking** assignments to a variable, the **last one** executes
- Example: Accumulator (Initial value of *Count* is zero)



```
always @(*)
   begin
       for( k=0; k<4; k=k+1 )
         Count = Count + k;
   end
```

```
Count = Count + 0;
 Count = 0 + 1;
 Count = 0 + 1 + 2;
 Count = 0 + 1 + 2 + 3;
 Result :  Count = 6
             Correct
```

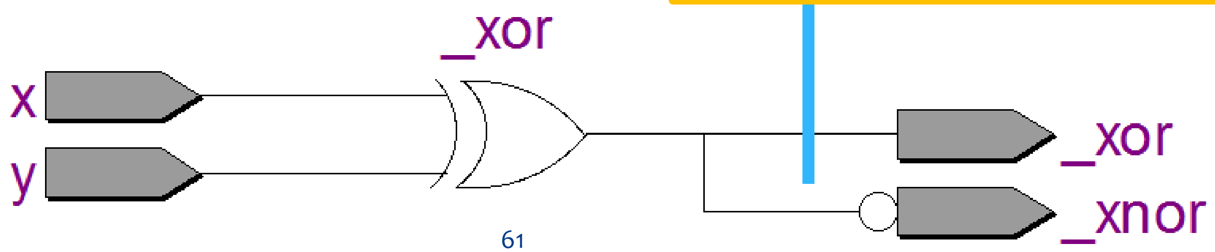
```
always @(*)
   begin
       for( k=0; k<4; k=k+1 )
         Count <= Count + k;
   end
```

```
Count <= Count + 0;
Count <= Count + 1;
Count <= Count + 2;
Count <= Count + 3;
Result :  Count = 3
          Incorrect
```

Combinational Logic

- Procedural assignments update the value of **reg.** The value will remain unchanged till another procedural assignment updates the variable. This is the main difference with continuous assignments in which the **right** hand expression is **constantly** placed on the **left** side

```
module DUT( input wire x, y,
            output reg   _xor ,
            output wire _xnor );

 always@(x,y)
        _xor = x ^ y;

 assign _xnor = ~_xor;

endmodule
```



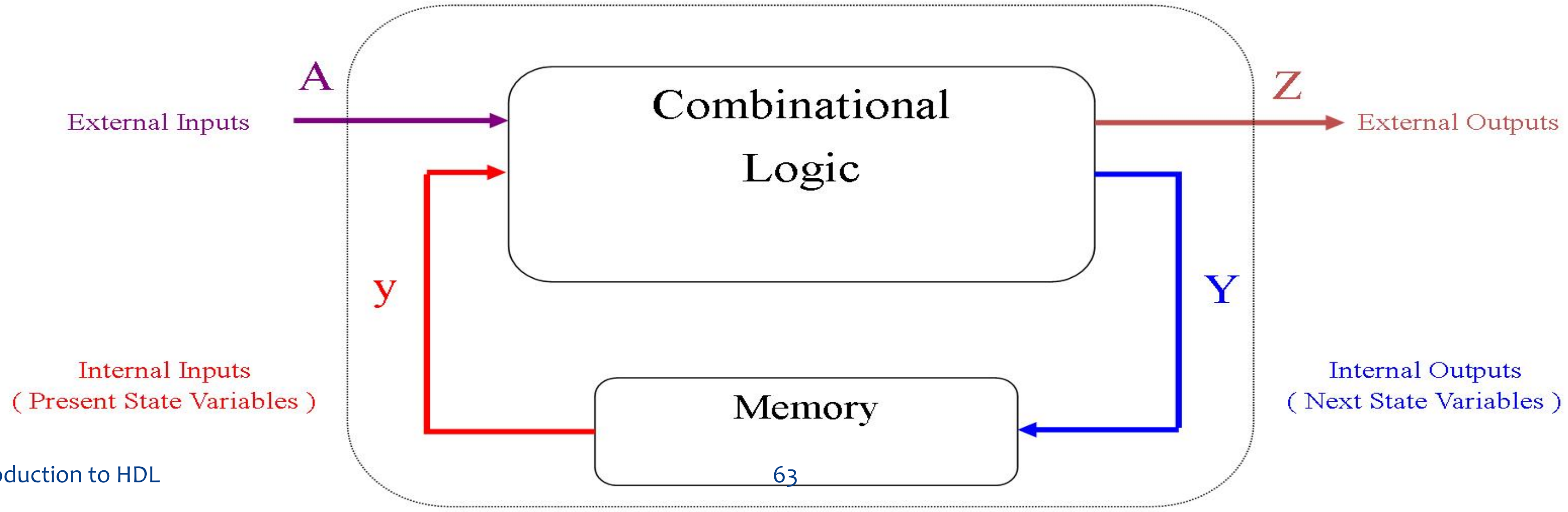
always vs. assign for describing combinational logic

- When Combinational Logic involves high complexity, it is preferred to use always block to describe it, because **always** block provides powerful statements like *if-else* , *case* and loop constructs

- Always block comes with more clarity and more concise description than **assign**

Sequential Logic

- Sequential logic is a type of logic circuit whose output depends not only on the present value of its input signals but on the past history of its inputs
- Current state is held in memory and the next state is computed through the combinational logic
- In a synchronous system , global clock signal organizes the flow of the data and the sequence of events



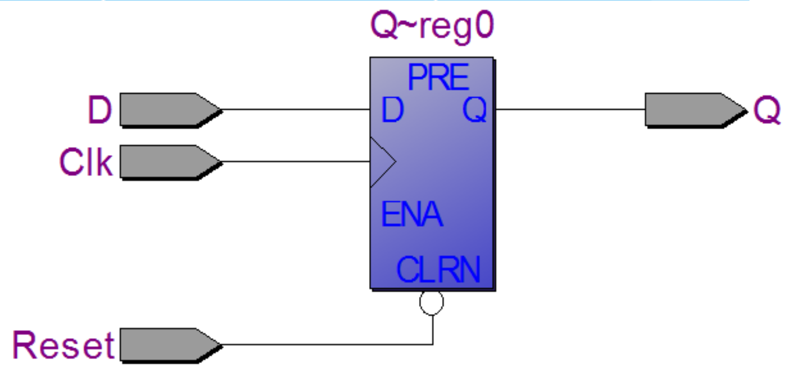
Sequential Logic

- Sequential logic can only be realized using an always block
- Consist of
  - Flip flops that are normally controlled by:
    - Positive edge of the clock (posedge)
    - Negative edge of the clock (negedge)
  - Latches
    - Transfers input to output when clock is “1” and otherwise stores the value
  - Finite State Machine (FSM)
- When using the always block for the sequential logic , “Non-blocking” assignments are used

Sequential Logic

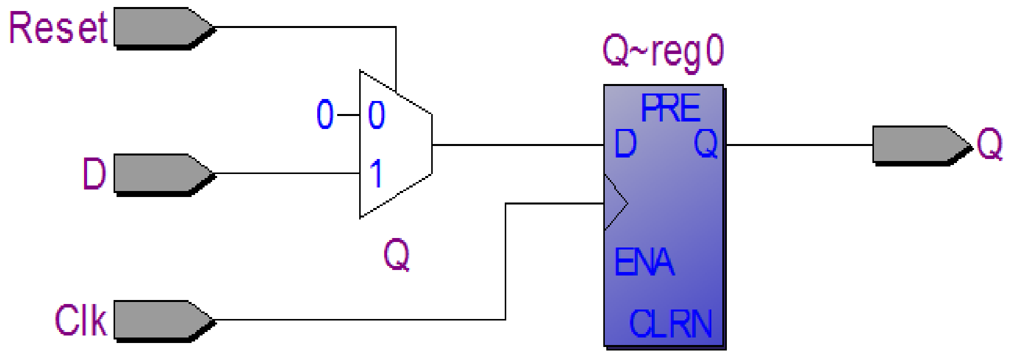
❖ Flip-flop with asynchronous Reset:

```
always@(posedge Clk, negedge Reset)
   if (Reset==0)
      Q <= 0;
   else
      Q <= D;
```



❖ Flip-flop with synchronous Reset:

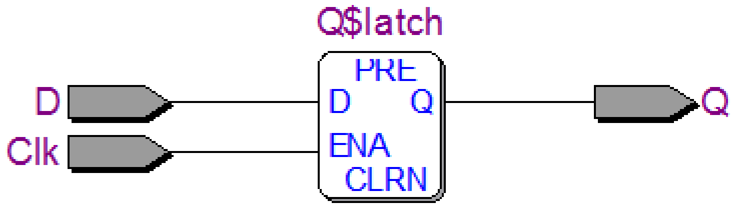
```
always@(posedge Clk)
   if (Reset==0)
      Q <= 0;
   else
      Q <= D;
```



Sequential Logic

- D-Latch

```
always@ ( D , Clk )
  if (Clk)
    Q <= D;
```



- What happens if we remove D from sensitivity list?

  Nothing , this results in same latch with warning saying D is not in the sensitivity list

Verilog Registers

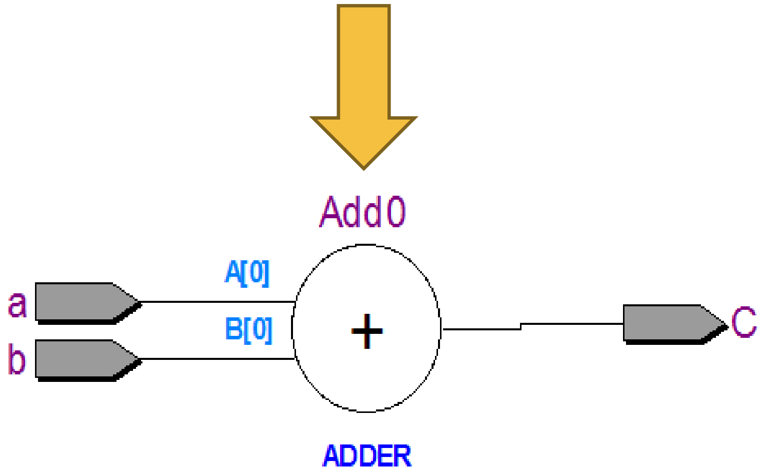
- The keyword “reg” does NOT necessarily denote a storage element or register.
- In digital design , registers represent memory elements
- Registers in Verilog should not be confused with hardware registers , the term register (reg) simply means a variable that can hold a value
- Verilog registers don’t need a clock and don’t need to driven like a net.

Verilog Registers

- May or may not be synthesized as a register

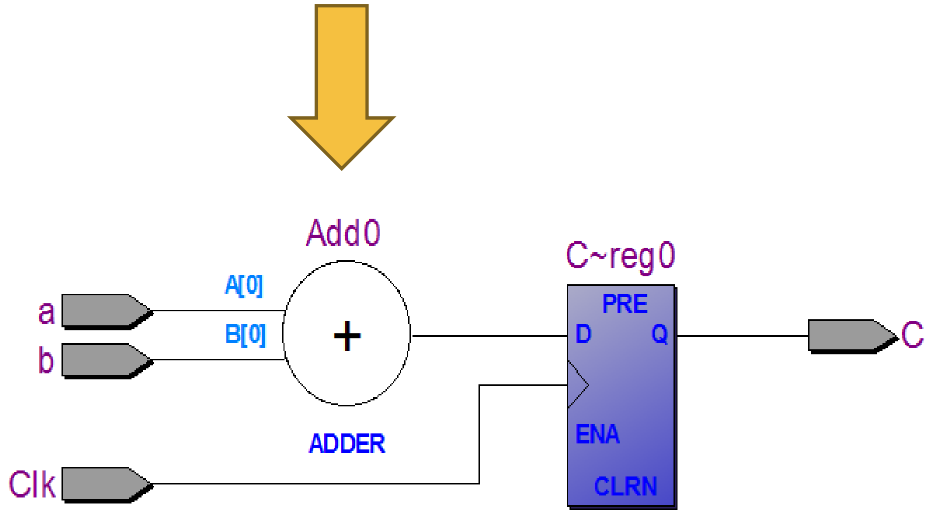
Not Register

```
always@ (a , b)
   C = a + b;
```



Register

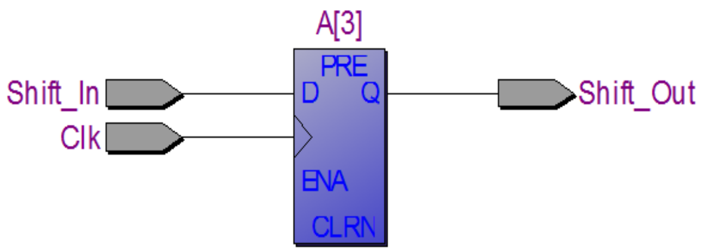
```
always@ (posedge Clk)
   C = a + b;
```



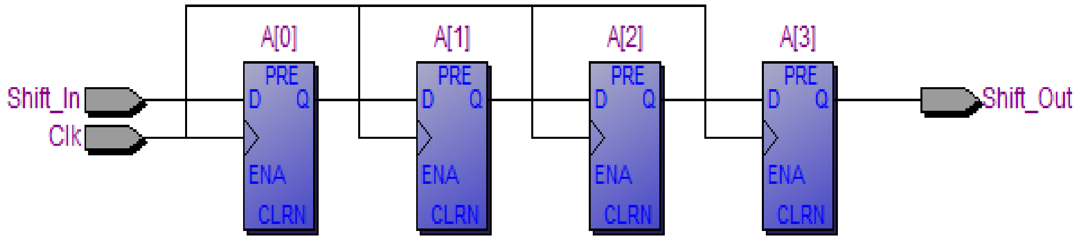
Sequential Logic

- When describing sequential logic using always block , “Non-Blocking” assignments are used.

```
integer k;
reg     [3:0] A;
assign  Shift_Out = A[3];

always@(posedge Clk)
  begin
    A[0] = Shift_In;
    for(k=1;k<4;k=k+1)
      A[k] = A[k-1];
  end
```



```
integer k;
reg     [3:0] A;
assign  Shift_Out = A[3];

always@(posedge Clk)
  begin
    A[0] <= Shift_In;
    for(k=1;k<4;k=k+1)
      A[k] <= A[k-1];
  end
```



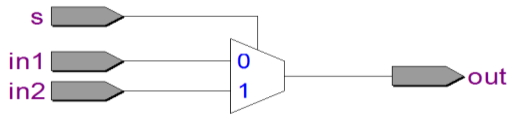
If-else statements

- Used only inside always block
- Format:

```
if ( expression1 )
   statement1;
else if ( expression2 )
   statement2;
else
    statement3;
```

Single statement no need for begin-end
Multiple statements , begin-end is needed

```
module Mux21(input  wire in1,in2,s,
             output reg  out);

  always@(in1,in2,s)
     if(s==0)
      out = in1;
     else
      out = in2;

endmodule
```



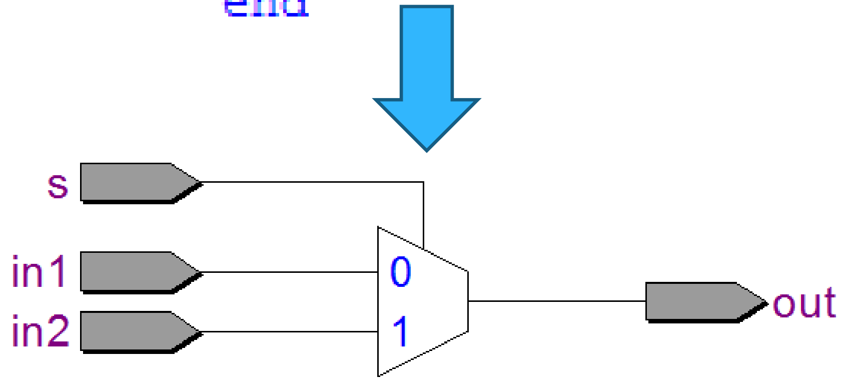
```
module Mux21(input  wire in1,in2,s,
             output reg  out);

  always@(in1,in2,s)
    begin
     out = in1;
     if(s==1)
      out = in2;
    end

endmodule
```

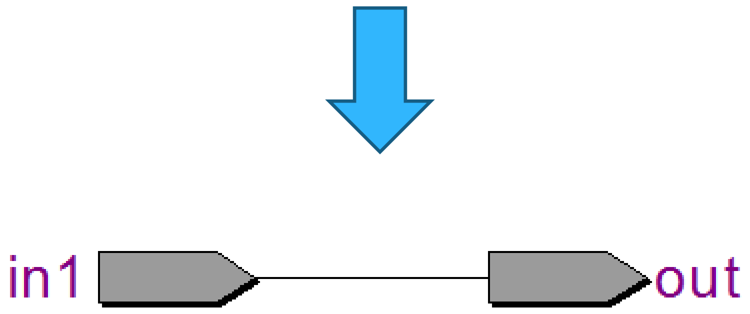
If-else statements

- If-else construct inside an always block have a sequential nature when used by blocking assignments.
- This means ordering is important

```
always@(*)
   begin
      out = in1;
      if(s==1)
         out = in2;
   end
```

```
always@(*)
   begin
      if(s==1)
         out = in2;
      out = in1;
   end
```





Case Statements

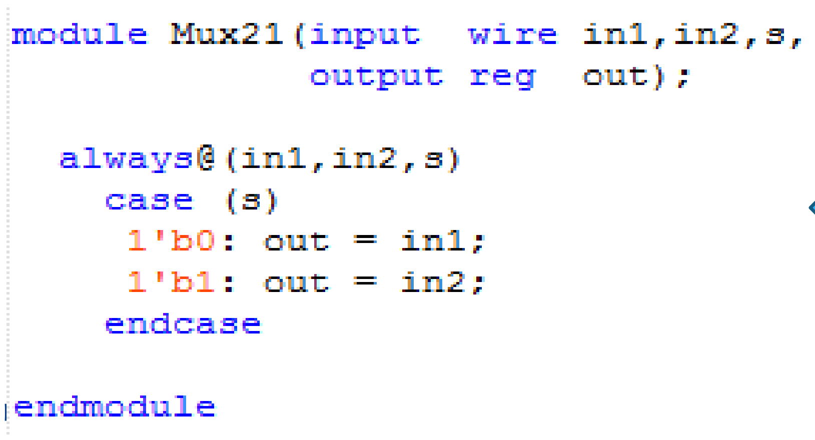
- Used only inside an always block
- Format:

```
case ( expression )
   option1 : statement1;
   option2 : begin
                  statement2;
               end
   default : statementn;
endcase
```

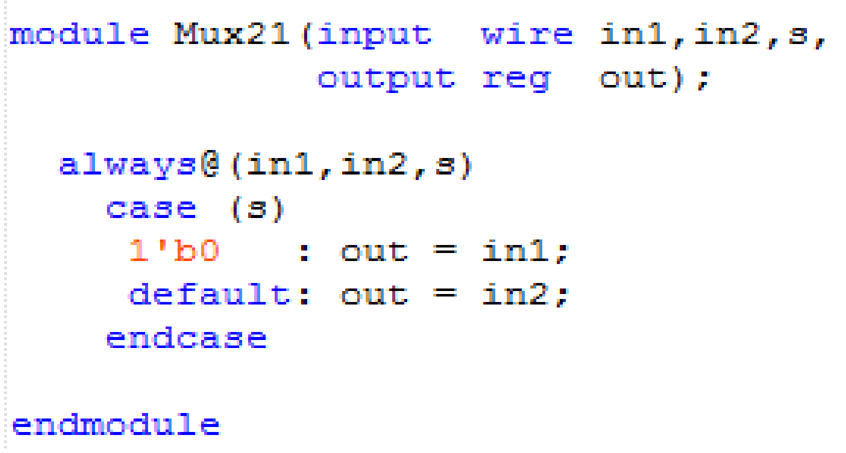
Single statement no need for begin-end

Multiple statements , begin-end is needed

- Example :



```
module Mux21(input  wire in1,in2,s,
                 output reg  out);

  always@(in1,in2,s)
    case (s)
     1'b0: out = in1;
     1'b1: out = in2;
    endcase

endmodule
```



```
module Mux21(input  wire in1,in2,s,
                 output reg  out);

  always@(in1,in2,s)
    case (s)
     1'b0    : out = in1;
     default: out = in2;
    endcase

endmodule
```

Case Statements

- Example: Combinational logic using both assign and always block

```
module HalfAdder(input  wire x,y,
                 output wire Cout,S);

   assign S    = x^y;
   assign Cout = x&y;

endmodule
```

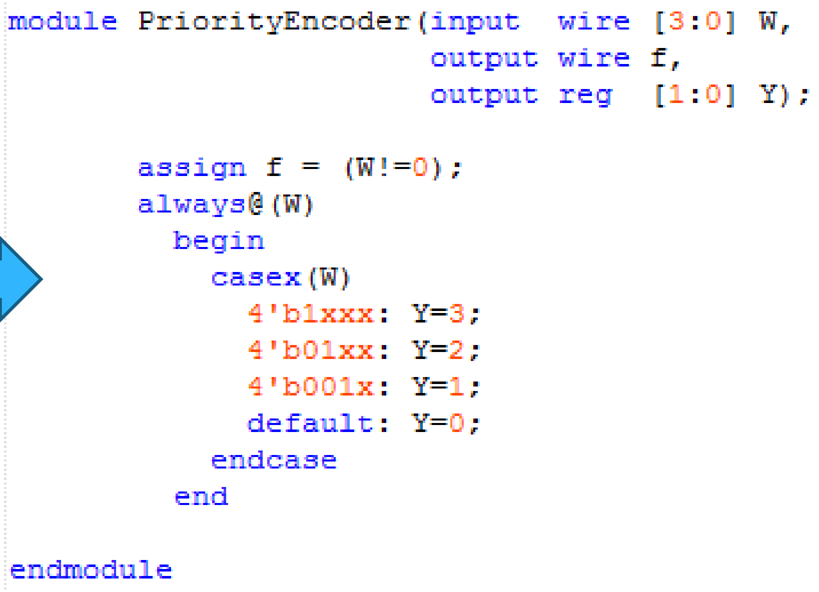
```
module HalfAdder( input  wire x,y,
                  output reg  Cout,S);

   always@(x,y)
     begin
       case ( {x,y} )
         2'b00: {Cout,S} = 2'b00;
         2'b01: {Cout,S} = 2'b01;
         2'b10: {Cout,S} = 2'b10;
         2'b11: {Cout,S} = 2'b11;
       endcase
     end

endmodule
```

Case Statements

- In case statements, each alternative is compared for an exact match
- Synthesis tools are only concerned about matching of "0" and "1" while "Z" and "X" are not important.
- If for simulation purposes, "X" or "Z" are needed to be added , casex is used which treats them as don't care.
- Example: 4-to-2 priority encoder

w3	w2	w1	w0
0	0	0	0
0	0	0	1
0	0	1	x
0	1	x	x
1	x	x	x

y1	y0	f
d	d	0
0	0	1
0	1	1
1	0	1
1	1	1



```
module PriorityEncoder(input  wire [3:0] W,
                       output wire f,
                       output reg  [1:0] Y);

    assign f = (W!=0);
    always@(W)
      begin
        casex(W)
          4'b1xxx: Y=3;
          4'b01xx: Y=2;
          4'b001x: Y=1;
          default: Y=0;
        endcase
      end

endmodule
```

Latch Inference in Combinational Logic

- When realizing combinational logic with always block using if-else or case constructs care has to be taken to avoid latch inference after synthesis
- The latch is inferred when “incomplete” if-else or case statements are declared
- This latch is “unwanted” as the logic is combinational not sequential
- A variable assigned within an always block that is not fully specified in all branches will infer a latch

Latch Inference in Combinational Logic

- To avoid latch inference make sure to specify all possible cases "explicitly"

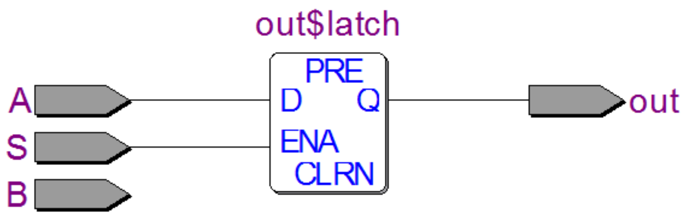
- Two practical approaches to avoid latch inference:
  - For if-else construct:
    1) Initialize the variable before the if-else construct
    2) Use else to explicitly list all possible cases
  - For case constructs:

    Use default to make sure all cases are covered.

Avoid Latch Inference in If-else Statements

✗ **Latch Inference**

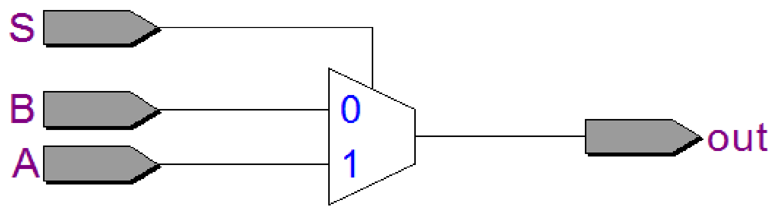
```
module DUT1(input  wire A,B,S,
            output reg  out);

       always@(*)
         begin
           if(S==1)
             out = A;
         end

endmodule
```



✓ **No Latch**

```
module DUT2(input  wire A,B,S,
            output reg  out);

       always@(*)
         begin
           out = B;
           if(S==1)
             out = A;
         end

endmodule
```

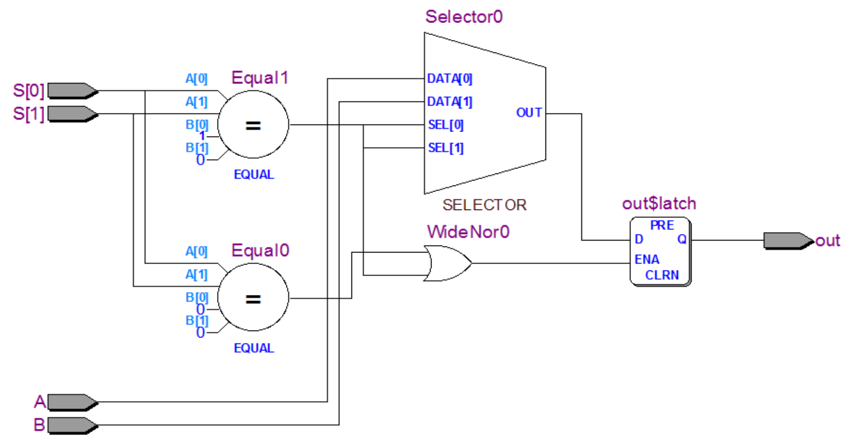
```
module DUT3(input  wire A,B,S,
            output reg  out);

       always@(*)
         begin
           if(S==1)
             out = A;
           else
             out = B;
         end

endmodule
```



Avoid Latch Inference in Case Statements

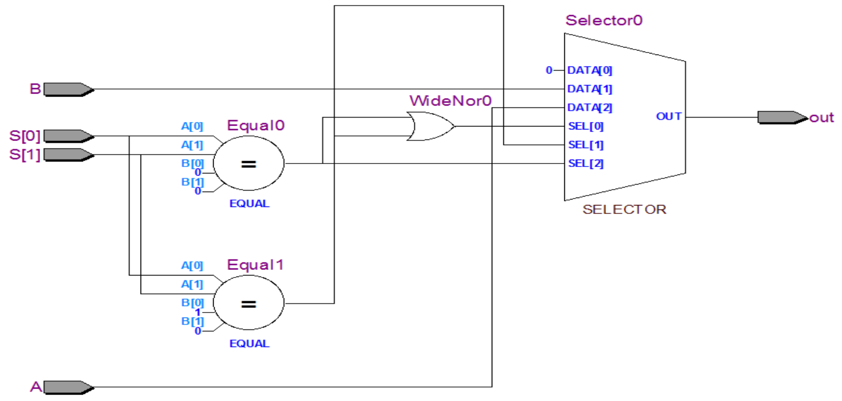
✗ Latch Inference

```
module  DUT1(input   wire  A,B,
             input   wire  [1:0]S,
             output  reg   out);

       always@(A,B,S)
         begin
           case  (S)
             2'b00:  out=A;
             2'b01:  out=B;
           endcase
         end

endmodule
```



✓ No Latch

```
module  DUT2(input   wire  A,B,
             input   wire  [1:0]S,
             output  reg   out);

       always@(A,B,S)
         begin
           case  (S)
             2'b00:     out=A;
             2'b01:     out=B;
             default:   out=1'b0;
           endcase
         end

endmodule
```



Using Sub-modules

- A design can use multiple sub-modules or a module multiple times
- Using a module in another is called “instantiation”
- Top-level module: the module that has not been instantiated
- To use a module inside another, it should be explicitly instantiated
- There are some built-in primitive logic gates in Verilog that can be instantiated
  - Built-in primitives means there is no need to define a module for these gates
  - and , nand , nor , or , xor , xnor , buf , not

Using Sub-modules

```
module Myand ( input  wire in1 , in2 , output  out );

   and myand ( out , in1 , in2 );              // instantiate

   assign out  =  in1 & in2;                   // assign

   reg  out ;
   always@ ( in1 , in2 )                       // always block
     out = in1 & in2;

endmodule
```

Sub-modules Instantiation

- To instantiate a module , two things have to be clearly specified
  - module's ports
  - module's parameters (considered as default if not specified)
- Format:

```
Module_name  #(parameter_value)  instance_name ( .port_name(port-connection)
, .port_name(port_connection) , … )
```

⇕

```
Module_name   instance_name ( .port_name(port-connection) ,
.port_name(port_connection) , … )
defparam instance_name.parameter_name  = parameter_value
```

- If port connections are in the same order as the original module **".port_name"** is not needed in the port list.

Sub-modules Instantiation

```
module Full_Adder ( input    wire Cin  ,  x  ,  y  ,
                             output  wire S ,  Cout   );

   assign S       = x ^ y ^ Cin;
   assign Cout = (x & y) | ( x & Cin ) | ( y & Cin );
endmodule
```

```
module RippleCarryAdder ( input    wire  [n-1:0]  X  , Y ,  input  wire  Cin ,
                                     output  wire [n-1:0] S        , output wire Cout );

  parameter n = 4;
  wire [n-1:0] C;

  Full_Adder  stage0 ( Cin , X[0]  , Y[0]  ,  S[0]  ,  C[1]  );
  Full_Adder  stage1  ( Cin ,  X[1]  , Y[1]  ,   S[1]  ,  C[2]  );
  Full_Adder  stage2  ( Cin ,  X[2] , Y[2]  ,  S[2]  ,  C[3] );
  Full_Adder  stage3  (.Cout(Cout) ,  .Cin(C[3]) , .x(X[3]) , .y(Y[3]) ,  .S(S[3]));

endmodule
```

Sub-modules Instantiation

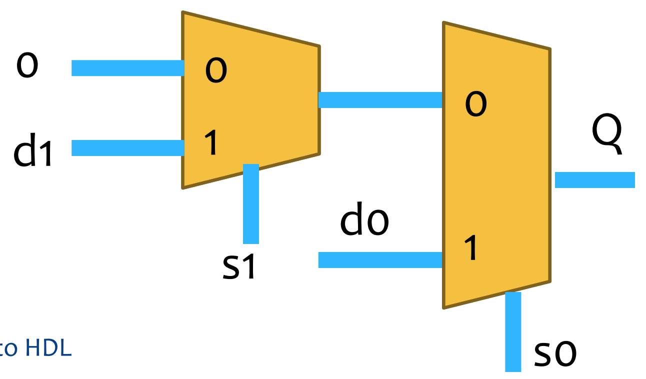
```
module  Add_Sub_Byte   ( input     wire [7:0]  X  , Y ,  input  wire  Op ,
                                         output  wire  [8:0] S          );
   // Subtract Y from X if Op = 1 otherwise add X + Y
   wire Cout;
   wire Y_2 [5:0] = Y ^ { 8'b111111};

   defparam    module0.n = 6;
   Full_Adder  module0( .Cin(op) , .x(X) , .y(Y) , .S(S[7:0]) , .Cout(S[8]) );

endmodule
```

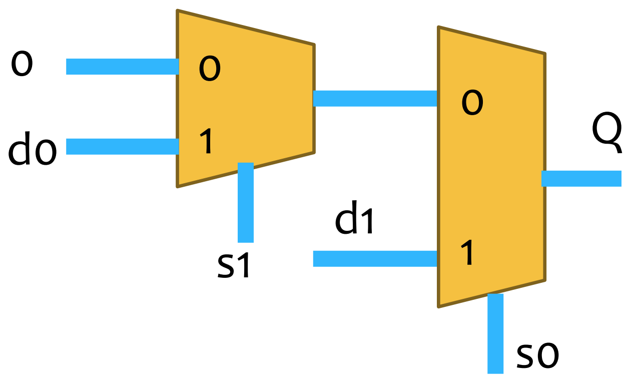
Priority Logic

- The order in which assignments are written in an always block may affect the logic that is synthesized.
- Example:

```
always@(s0,s1,d0,s1)
 begin
  Q = 0;
  if(s0) Q = d0;
  else if (s1) Q = d1;
 end
```

```
always@(s0,s1,d0,s1)
 begin
  Q = 0;
  if(s1) Q = d1;
  else if (s0) Q = d0;
 end
```





Counter

4-Bit Unsigned Down Counter
With Synchronous Set

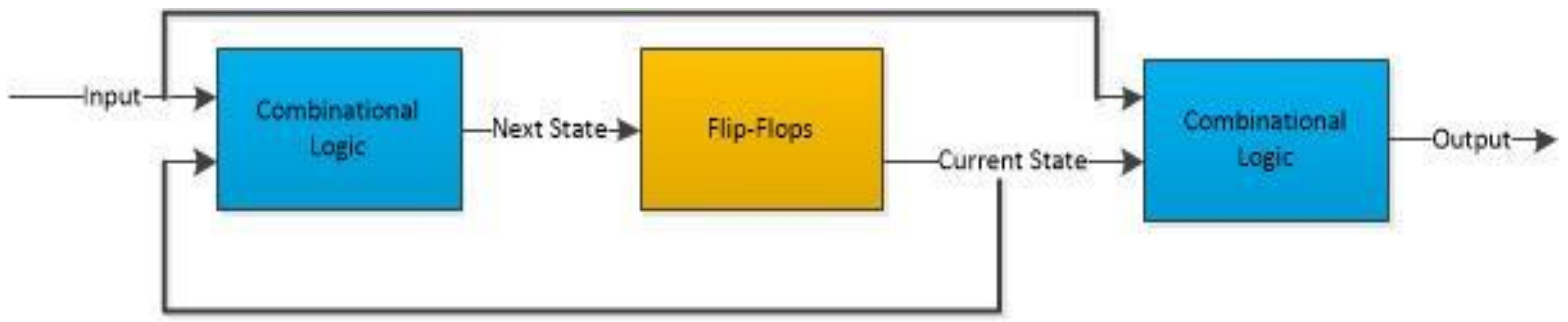
```
module D_counter ( input    Clk , S ,
                            output [3:0]Q);
  reg [3:0] tmp;
 always@(posedge Clk)
   begin
     if( S )
       tmp <= 4'b1111;
     else
       tmp <= tmp -1'b1;
   end
 assign Q = tmp;

endmodule
```

4-Bit Unsigned Down Counter
With Synchronous Set

```
module D_counter ( input    Clk , S ,
                            output [3:0]Q);
  reg [3:0] tmp;
 always@(posedge Clk or negedge Rst)
   begin
     if( Rst == 0)
       tmp <= 4'b0000;
     else
       tmp <= tmp +1'b1;
   end
 assign Q = tmp;

endmodule
```

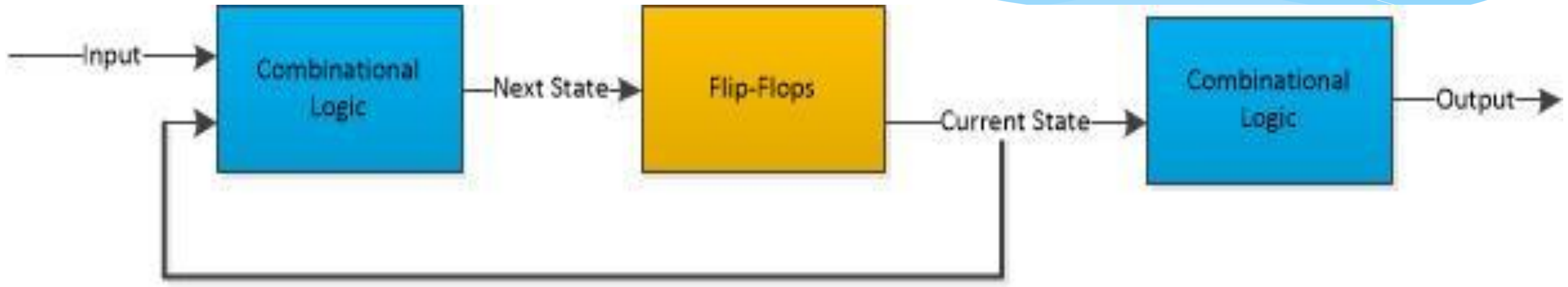
Finite State Machine (FSM)

- An abstract machine that can be in one of a finite number of states. The machine is in only one state at a time (current state)
- Two types :
  - Mealy: Output depends on the "current state" and the "input"



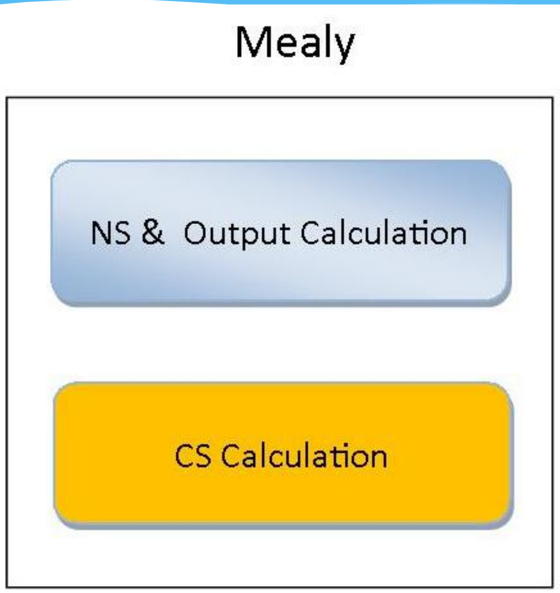
Finite State Machine (FSM)

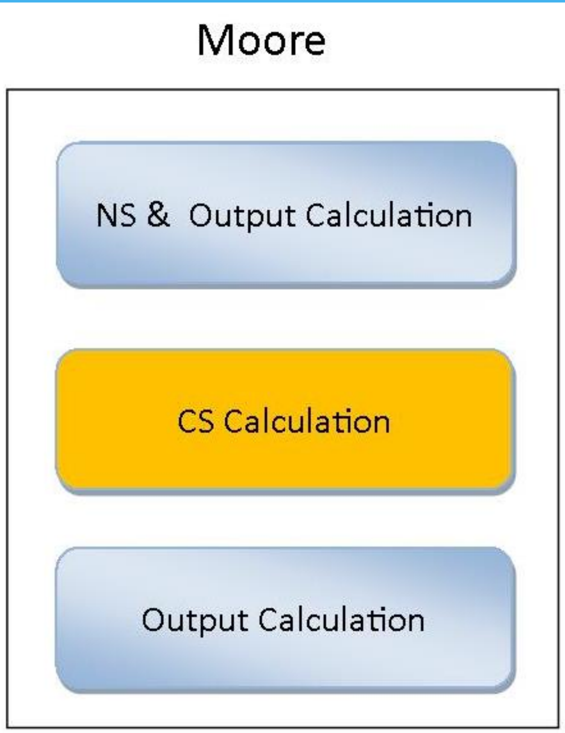
- Moore: Output depends only on the "current state"



- A FSM can be described in terms of its
  - Current State
  - Next State
  - Output

FSM Code Structure



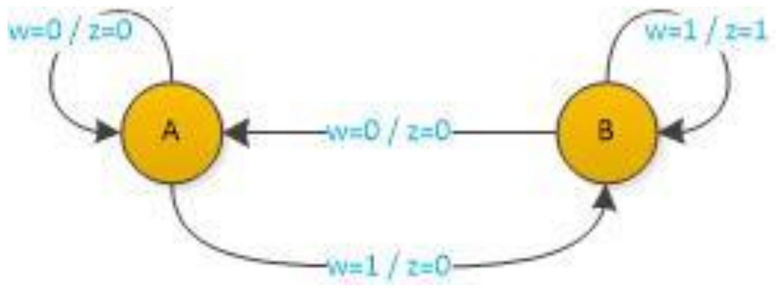


- NS & Output Calculation mainly implemented using blocking assignments & case struct inside always block

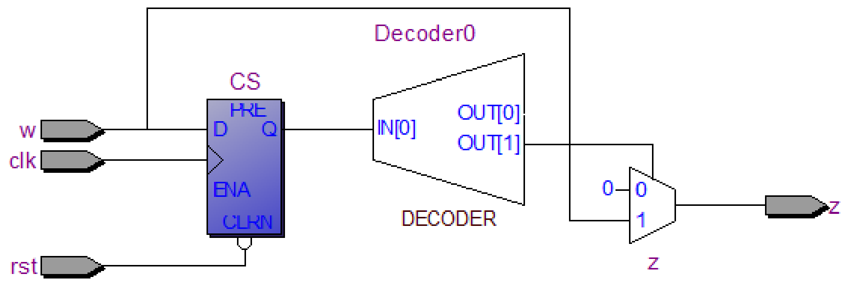
- NS Calculation mainly implemented using blocking assignments & case struct inside always block
- Output Calculation mainly implemented using assign statements

Mealy Machine

❖ State Diagram



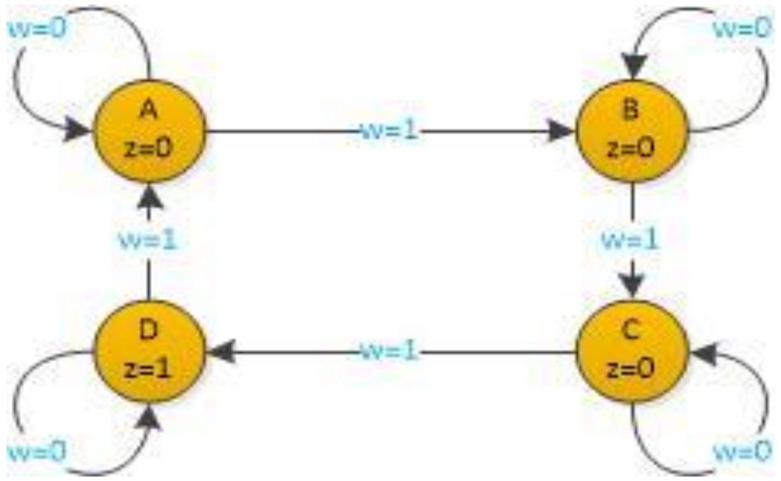
❖ RTL View



```
module mealy(input  wire clk, w, rst,
                   output reg  z);
  reg CS , NS;
  parameter A=1'b0, B=1'b1;

  // NS & Output Calculation ( Combinational )
  always@(w,CS)
    case (CS)
     A: if(w==0)
          begin
            NS = A; z = 0;
          end
        else
          begin
            NS = B; z = 0;
          end
     B: if(w==0)
          begin
            NS = A; z = 0;
          end
        else
          begin
            NS = B; z = 1;
          end
    endcase
  // CS Calculation ( Sequential )
  always@(posedge clk, negedge rst)
    if( rst == 0 )
      CS <= A;
    else
      CS <= NS;

endmodule
```

Moore Machine

- State Diagram



```
module mealy(input  wire clk, w, rst,
             output wire z);
  reg [1:0] CS , NS;
  parameter A=2'b00, B=2'b01, C=2'b10, D=2'b11;

  // NS Calculation ( Combinational )
  always@(w,CS)
    case (CS)
     A: NS = (w)? B : A;
     B: NS = (w)? C : B;
     C: NS = (w)? D : C;
     D: NS = (w)? A : D;
    endcase

  // CS Calculation ( Sequential )
  always@(posedge clk, negedge rst)
    if( rst == 0 )
      CS <= A;
    else
      CS <= NS;

  // Output Calculation
  assign z = ( CS == D );

endmodule
```

Verification

- Testbench
  - An unsynthesizable Verilog module that instantiates main Verilog design and examines behavior and output of design by applying predetermined inputs to it.
  - High-level unsynthesizable keywords such as initial , $display , $stop , $fopen , $readmem, are available for debugging purposes
  - Define **time unit** and **precision** by **`timescale** at the beginning of testbench
    - Time unit specifies time amount #1 refers to
    - Examples: `timescale 1ns / 1ns - `timescale 1ns / 100ps
  - Testbench module usually has **no ports**

Verification

- Value of signal “a” at time “t “ after start of simulation can be specified by initial #t a = val
- In more general form “#t” can be used to specify delay in any assignment
- Statement above can be used to generate clock with desired period to be applied to sequential design

```
reg clk;
initial clk = 1’b0;
always #T clk = ~ clk;
```

  - These generate clk with period equal to 2T whose initial value is 0.
  - Do not forget to initialize clk : If not initialized, clk value will be undefined (X) in simulation start and will persist through simulation

Simulation commands

* Register array used as memory can be initialized by variants of command $memread ($memreadb - $memreadh)

* $display command can be used anywhere inside always block or after initial statement to print signal
  * Arguments of command $display are **same** as those of printf in C
  * Example : $display(“ Clk value is %b” , clk);

* Command $stop can be used to stop simulation at any time
  * Example: To stop simulation at time 1000 => initial #1000 $stop;

Verification

```
`timescale 1 ns / 1 ns

module parity_gen_tb();

   reg [3:0] data_in;
   wire      even_parity, odd_parity;

   parity_gen_rtl pg0(.in(data_in),.even_parity(even_parity),.odd_parity(odd_parity));


   always@(*)
      begin
         if( ($time ==  5) && ( ( even_parity != 1'b0 ) || ( odd_parity != 1'b1 ) ) )
            $display(" data in = %b , even parity = %b , odd parity = %b " , data_in , even_parity , odd_parity );

         if( ($time == 15) && ( ( even_parity != 1'b1 ) || ( odd_parity != 1'b0 ) ) )
            $display(" data in = %b , even parity = %b , odd parity = %b " , data_in , even_parity , odd_parity );
      end

   initial  #0 data_in = 4'b1010;
   initial #10 data_in = 4'b1011;

   initial #50 $display( " Done verifying parity generator! ");
   initial #50 $stop;

endmodule
```

References

* Dr.Shabany's Slides for ASIC/FPGA Chip Design
  * Available online at ee.sharif.edu/~asic
* Introduction to Verilog
  * ee.mut.ac.th/course/eeet0413/ppt/Chapter10%20Verilog.ppt
* Asic - World Website
  * asic-world.com
* Wikipedia : Various Articles