University of San Carlos | Department of
COMPUTER ENGINEERING

CpE 3101L – Introduction to HDL

Unit 5: Behavioral Modeling of Combinational Circuits

Lecture 1

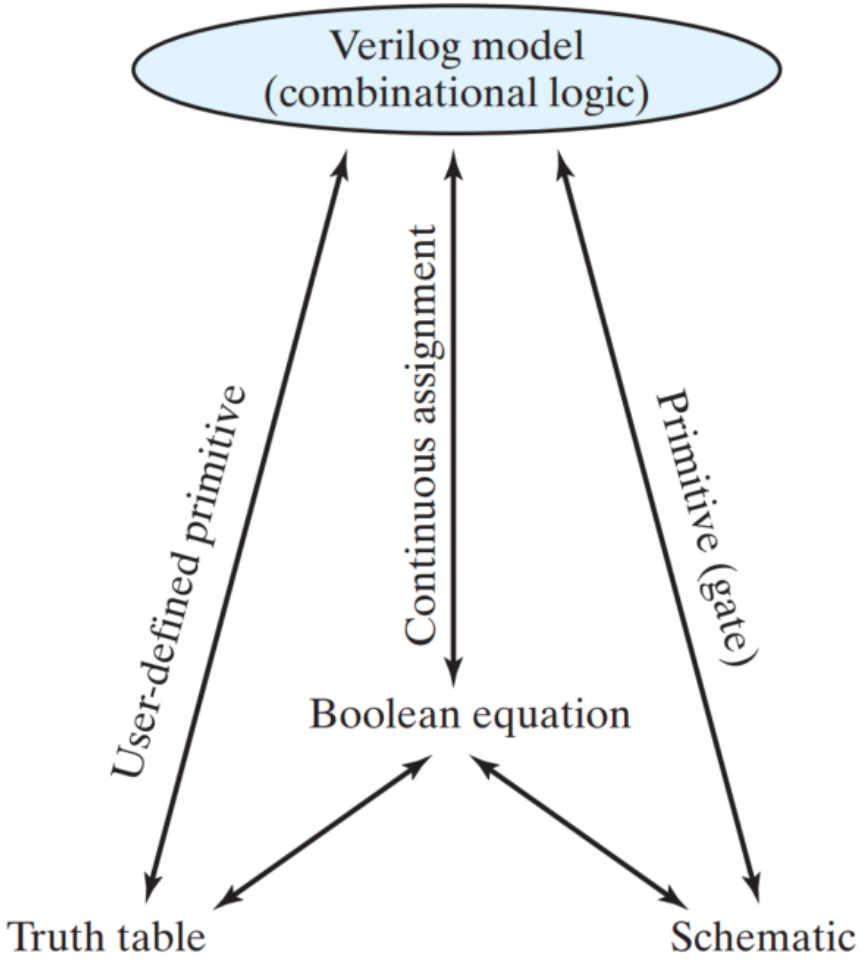
Outline

- Behavioral Modeling of Combinational Circuits
- Procedural Assignment: *Always* Block
- *Reg* Data Type and Sensitivity List
- Blocking Assignment
- *If-Else* and *Case* Statements
- Logic Synthesis of HDL-based Designs
- Synthesis of Operators
- Guidelines for Good Synthesis of Combinational Circuits

Combinational Logic in Verilog

- To create the HDL model of combinational circuits:

- Structural modeling *(Units 2-3)*
  - Gate primitives, ~~UDPs~~, other HDL modules
  - Needs schematic ~~or truth tables~~ or other HDL modules
- Dataflow modeling *(Unit 4)*
  - Needs Boolean equations
- Behavioral modeling *(Unit 5)*
  - Needs behavior or functionality of the circuit
  - *More frequently used in sequential logic (Unit 6)*



Behavioral Modeling of Combinational Circuits

- **Behavioral modeling** represents digital circuits at a functional and algorithmic level.
  - Involves describing at a higher level of abstraction
- Mostly used to describe sequential logic but can also be used for combinational logic.
- **Behavioral HDL models** describe combinational circuits by their ***behavior (functionality)*** rather than their gate structure or Boolean equations.

Procedural Assignment

- Behavioral descriptions use procedural assignment statements with keyword ***always***
  - ***always*** can also be used in simulations (together with ***initial***)
  - ***always*** is synthesizable (unlike ***initial***)

- Consists of a sensitivity list *(an event control expression)*

- **Sensitivity list** is a list of signals that is monitored for changes
  - Whenever one of the signals in the sensitivity list changes its value, the statements inside the ***always*** block are ***evaluated sequentially***.

Procedural Assignment

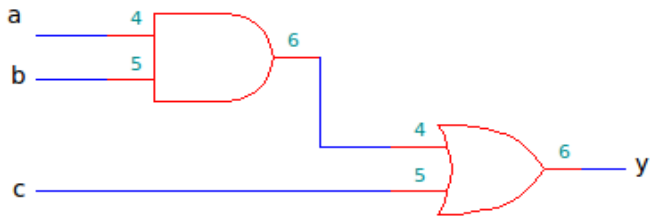
- All signals assigned inside the *always* block must be declared as *reg*
- Contrary to a *wire* (a *net* data type), a *reg* retains its value until a new value is assigned (*variable* data type)

```
// declare reg here;

always @( [sensitivity_list] )
begin
   // local variable declarations here; (optional)

   // body: composed of procedural statements;
end
```

- Note: Use of **begin-end** are required when the contents of the *always* block have *more than one statement*

Sensitivity List

- Sensitivity list is a list of signals and events to which the *always* block responds *(i.e.,* is *"sensitive to")*.
- *For combinational circuits*, all the input signals must be included in this list:
- Example:

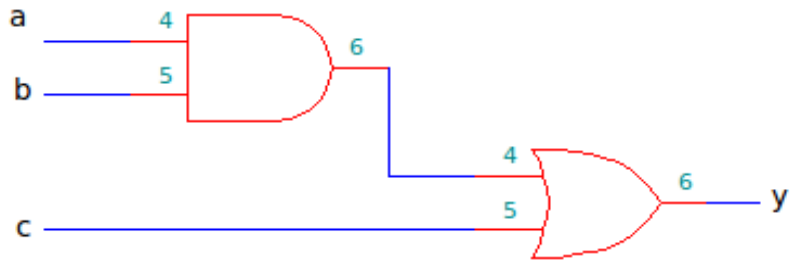



```
// Verilog 1995 syntax
reg y;

always @(a or b or c)
begin
    y = (a & b) | c;
end
```

```
// Verilog 2001 syntax
reg y;

always @(a, b, c)
begin
    y = (a & b) | c;
end
```

Sensitivity List

- Alternately, * can be used to implicitly include all input signals in the sensitivity list.
- Example:




```
// Verilog 2001 syntax
reg y;

always @(*)
begin
    y = (a & b) | c;
end
```

Blocking Assignment

- Uses = operator in assigning values in a ***reg*** *(inside an* ***always*** *block)*
- Evaluates and assigns values per single step *(sequential execution)*
- Execution flow within the ***always*** block is "***blocked***" until the assignment is completed
- Blocking assignment should be used in combinational logic
- Example:

Execution:

1. x is evaluated
2. y is evaluated using value from (1)

```
// using blocking statements
reg x, y;
always @(*)
begin
    x = a & b;
    y = x | c;
end
```

Example 1: 1-bit Equality Comparator

- Create a Verilog behavioral description of a 1-bit equality comparator:

- **Specifications:**
  - Inputs: i0, i1
  - Output: eq

- **Truth Table:**

i1	*i0*	*eq*
0	0	1
0	1	0
1	0	0
1	1	1

- **Logic Function:**

$$eq = (i1'\ i0') + (i1\ i0)$$

Example 1: Design Entry

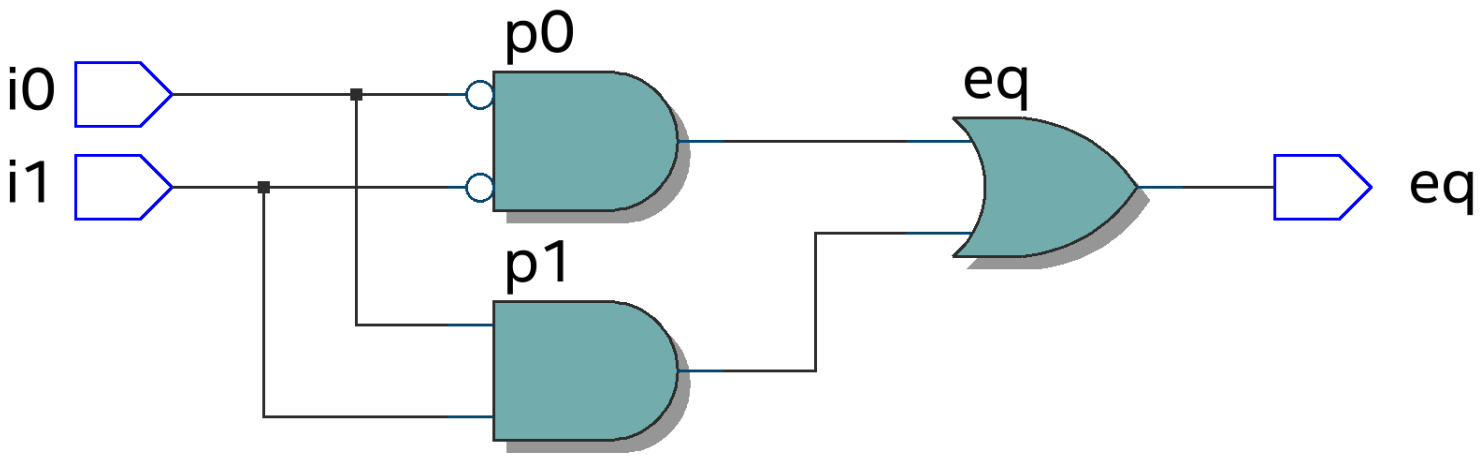
- Behavioral HDL Description (Verilog 2001 syntax):

```
// Behavioral description of a 1-bit Equality Comparator
//
module EqualityComparator_1bit (
    input wire  i0, i1,
    output reg  eq        // eq declared as reg
);

// internal signals p0 and p1 declared as reg
reg p0, p1;

// all signal assignments inside the always statement
// must be declared as reg types
//
always @(*)
begin
    // the order of statements are executed sequentially
    p0 = ~i0 & ~i1;
    p1 =  i0 &  i1;
    eq =  p0 |  p1;
end

endmodule
```

$$eq = (i1'\ i0') + (i1\ i0)$$

Example 1: Synthesized Design

- RTL Schematic View: $eq = (i1'\ i0') + (i1\ i0)$



Total logic elements	1
Total registers	0
Total pins	3

Example 1: Testbench File

```
// Testbench file of 1-bit Equality Comparator
//
`timescale 1 ns / 1 ps
module tb_EqualityComparator_1bit ();

    // declare inputs as reg types
    reg   i0, i1;

    // outputs as wire types
    wire  eq;

    // instantiate UUT
    EqualityComparator_1bit UUT ( .i0(i0), .i1(i1), .eq(eq) );

    // generate stimuli
    initial begin
        $display("Starting simulation at %0d ns...", $time);
        i1 = 1'b0;  i0 = 1'b0;  #5
        i1 = 1'b0;  i0 = 1'b1;  #5
        i1 = 1'b1;  i0 = 1'b0;  #5
        i1 = 1'b1;  i0 = 1'b1;  #5

        $display("Finished simulation at %0d ns.", $time);
        $stop;
    end

    // response monitor
    initial begin
        $monitor("Time = %2d ns\t i1 = %b\t i0 = %b\t eq = %b", $time, i1, i0, eq );
    end

endmodule
```

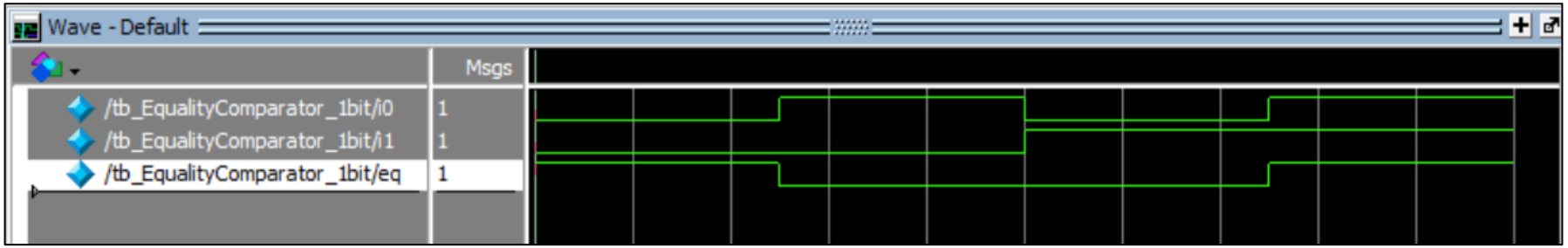
Example 1: Simulation Output

- Standard Output:

```
# Starting simulation at 0 ns...
# Time =  0 ns  i1 = 0  i0 = 0   eq = 1
# Time =  5 ns  i1 = 0  i0 = 1   eq = 0
# Time = 10 ns  i1 = 1  i0 = 0   eq = 0
# Time = 15 ns  i1 = 1  i0 = 1   eq = 1
# Finished simulation at 20 ns.
```

i1	*i0*	*eq*
0	0	1
0	1	0
1	0	0
1	1	1

- Waveform:



If-Else Statements

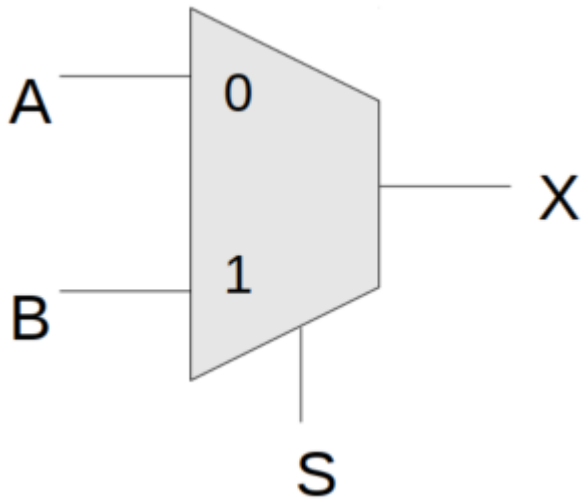
- Allow **conditional assignments**
- Can only be used inside procedural assignments (***always*** or *initial* blocks)
  - **Note:** Use of **begin-end** are required for *more than one statement*
- **Multiple *if* statements** can be **cascaded or nested** to evaluate multiple conditions
- **Syntax:**

```
if [condition]
  begin
   [procedural statements];
  end
else
  begin
   [procedural statements];
  end
```

```
if [condition_1]
   …
else if [condition_2]
   …
else if [condition_3]
   …
else
   …
```

Example 2: 2x1 Multiplexer

- Verilog behavioral description of a 2-to-1 line mux using *if-else statements*:



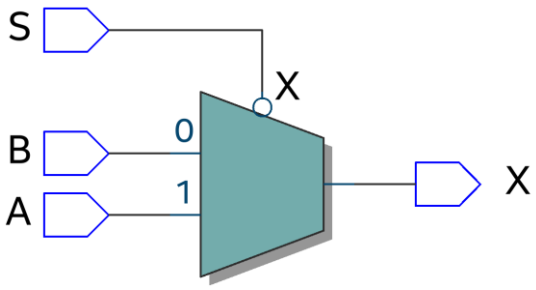

```
// Verilog 1995 syntax
reg X;
always @(A or B or S)
if (S == 0)
   X = A;
else
   X = B;
```

```
// Verilog 2001 syntax
reg X;
always @(*)
if (S == 0)
   X = A;
else
   X = B;
```

Example 2: Design Entry 1 (2x1 Mux)

- Using Verilog 1995 syntax:

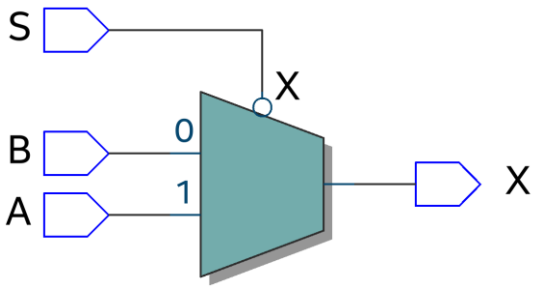
```
// Behavioral description of a 2x1 mux
//  using an if-else statement
//
module Mux_2x1 (A, B, S, X);
input    A, B, S;
output   X;

// all signals/ports assigned inside always statement
// must be declared as reg types
reg X;

always @(A or B or S)
begin
   if (S == 1'b0)
      X = A;
   else
      X = B;
end

endmodule
```



Example 2: Design Entry 2 (2x1 Mux)

- Using Verilog 2001 syntax:

```
// Behavioral description of a 2x1 mux
//  using an if-else statement
//
module Mux_2x1 (
    input wire  A, B, S,
    output reg  X          // X must be reg type
);
// all signals/ports assigned inside always statement
// must be declared as reg types

always @(*)
begin
    if (S == 1'b0)
        X = A;
    else
        X = B;
end

endmodule
```



Example 2: Testbench File (2x1 Mux)

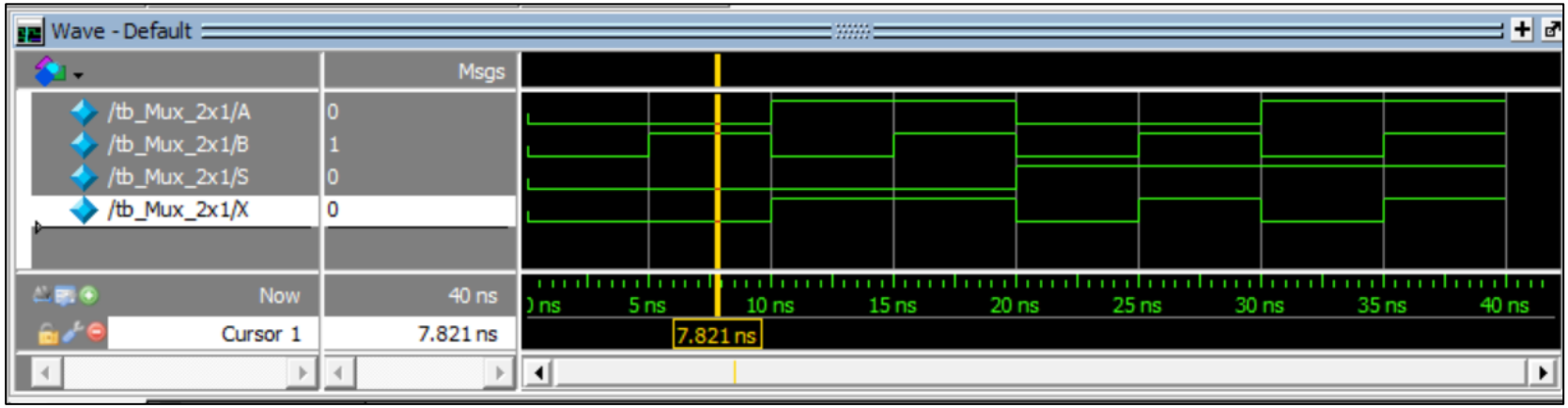
```
// Testbench file of 2x1 mux

`timescale 1 ns / 1 ps
module tb_Mux_2x1 ();

   // inputs as reg, outputs as wire
   reg   A, B, S;
   wire  X;

   // instantiate UUT
   Mux_2x1 UUT ( .A(A), .B(B), .S(S), .X(X) );

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);

      S  = 1'b0;  {A,B} = 2'b00; #5
                  {A,B} = 2'b01; #5
                  {A,B} = 2'b10; #5
                  {A,B} = 2'b11; #5

      S  = 1'b1;  {A,B} = 2'b00; #5
                  {A,B} = 2'b01; #5
                  {A,B} = 2'b10; #5
                  {A,B} = 2'b11; #5

      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial begin
      $monitor("Time = %2d ns\t S = %b AB = %b%b\t X = %b", $time, S, A, B, X);
   end

endmodule
```

Example 2: Simulation Output (2x1 Mux)

- Standard Output:

```
# Starting simulation at 0 ns...
# Time =  0 ns  S = 0 AB = 00  X = 0
# Time =  5 ns  S = 0 AB = 01  X = 0
# Time = 10 ns  S = 0 AB = 10  X = 1
# Time = 15 ns  S = 0 AB = 11  X = 1
# Time = 20 ns  S = 1 AB = 00  X = 0
# Time = 25 ns  S = 1 AB = 01  X = 1
# Time = 30 ns  S = 1 AB = 10  X = 0
# Time = 35 ns  S = 1 AB = 11  X = 1
# Finished simulation at 40 ns.
```

- Waveform:



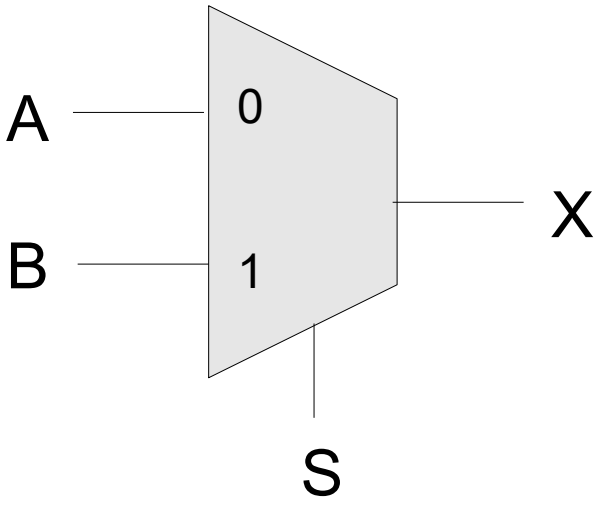
Case Statements

- Multiway decision statements
- Can only be used inside procedural assignments (*always* or *initial* blocks)
  - Note: Use of **begin-end** are required for *more than one statement*
- Syntax:

```
case [case_expression]
  [item_1]:
    begin
      [procedural statements];
    end
  [item_2]:
    begin
      [procedural statements];
    end
  default:
    begin
      [procedural statements];
    end
endcase
```

Example 3: 2x1 Multiplexer (v2)

- Verilog behavioral description of a 2-to-1 line mux using *case statements*:



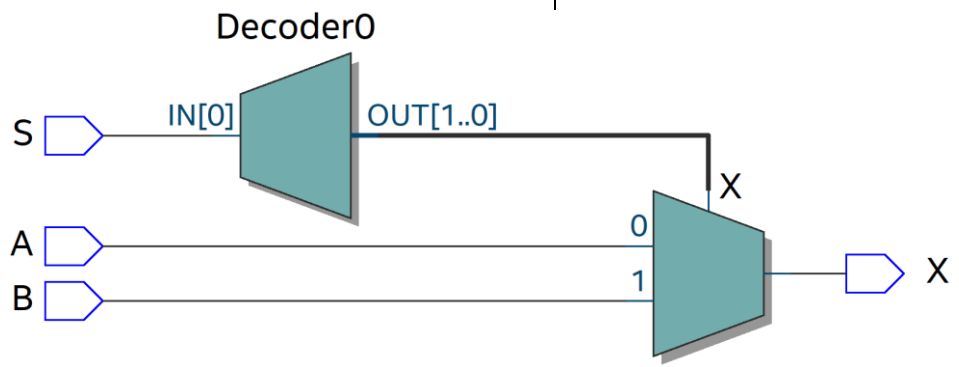

```
// Verilog 1995 syntax
reg X;
always @(A or B or S)
  case (S)
    1'b0 : X = A;
    1'b1 : X = B;
  endcase
```

```
// Verilog 2001 syntax
reg X;
always @(*)
  case (S)
    1'b0    : X = A;
    default : X = B;
  endcase
```

Example 3: Design Entry 1 (2x1 Mux v2)

- Using Verilog 1995 syntax:

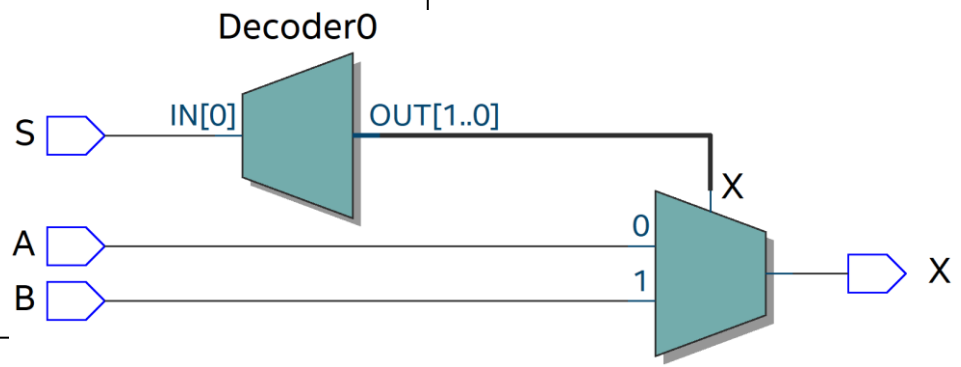
```
// Behavioral description of a 2x1 mux
//  using a case statement
//
module Mux_2x1 (A, B, S, X);
input       A, B, S;
output   X;

// all signals/ports assigned inside always statement
// must be declared as reg types
reg X;

always @(A or B or S)
begin
   case (S)
      1'b0  : X = A;
      1'b1  : X = B;
   endcase
end

endmodule
```



Example 3: Design Entry 2 (2x1 Mux v2)

- Using Verilog 2001 syntax:

```
// Behavioral description of a 2x1 mux
//  using a case statement
//
module Mux_2x1 (
    input wire  A, B, S,
    output reg  X          // X must be reg type
);
// all signals/ports assigned inside always statement
// must be declared as reg types

always @(*)
begin
    case (S)
       1'b0     :  X = A;
       default  :  X = B;
    endcase
end

endmodule
```

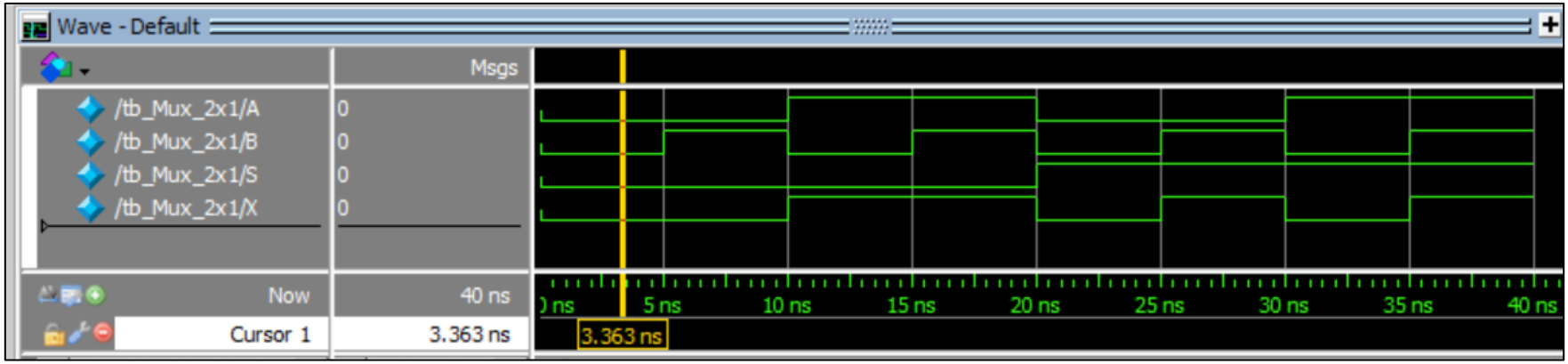


Example 3: Simulation Output (2x1 Mux v2)

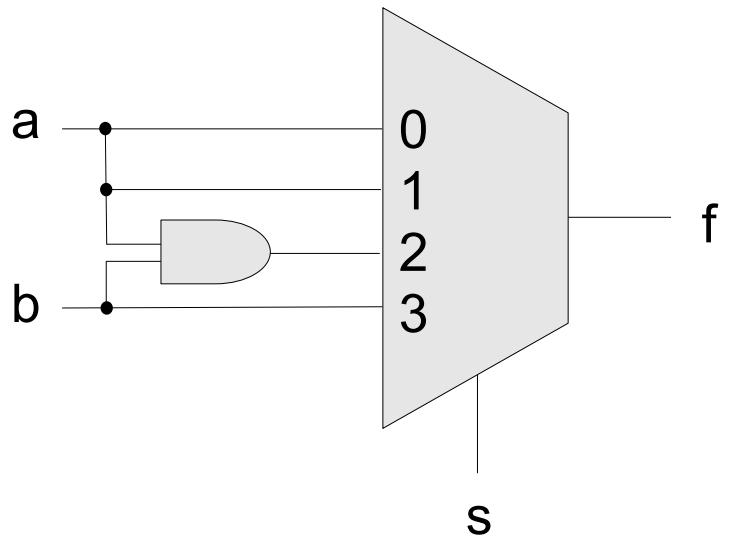
- *Using the same testbench file as Example 2*
- Standard Output:

```
# Starting simulation at 0 ns...
# Time =  0 ns  S = 0 AB = 00  X = 0
# Time =  5 ns  S = 0 AB = 01  X = 0
# Time = 10 ns  S = 0 AB = 10  X = 1
# Time = 15 ns  S = 0 AB = 11  X = 1
# Time = 20 ns  S = 1 AB = 00  X = 0
# Time = 25 ns  S = 1 AB = 01  X = 1
# Time = 30 ns  S = 1 AB = 10  X = 0
# Time = 35 ns  S = 1 AB = 11  X = 1
# Finished simulation at 40 ns.
```

- Waveform:



Incorporating Logic




```
// Verilog 2001 syntax

reg f;

always @(*)
  case (s)
    2'd0, 2'd1 : f = a;
    2'd2       : f = a & b;
    default    : f = b;
  endcase
```

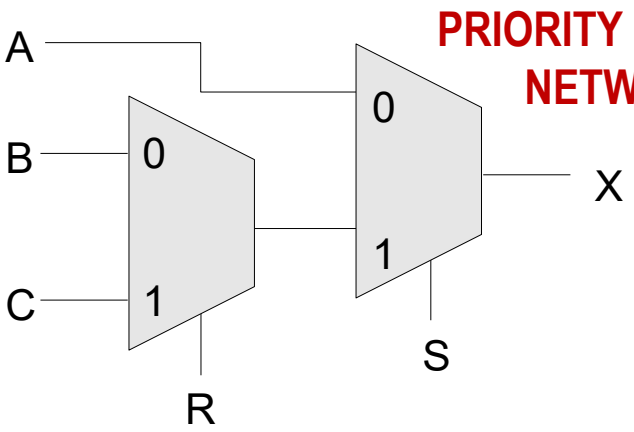
Nested If-Else vs. Case Statements

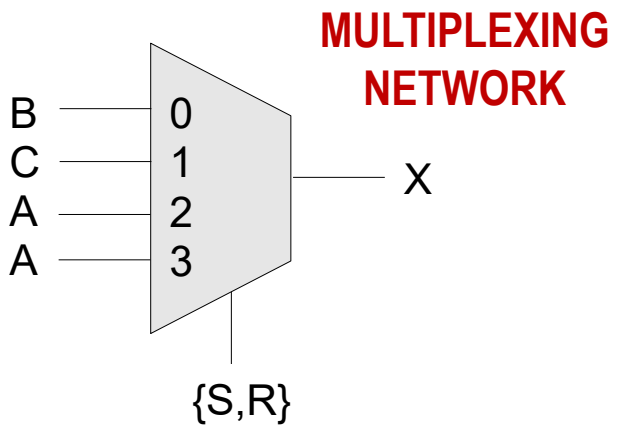
- Nesting *if-else* statements implies PRIORITY

```
reg X;
always @(*)
  if (S == 0)
    X = A;
  else
    if (R == 0)
      X = B;
    else
      X = C;
```

VS.

```
reg X;
always @(*)
  case ({S, R})
    2'b00      : X = B;
    2'b01      : X = C;
    default    : X = A;
  endcase
```





Example 4: 2x4 Line Decoder

- Verilog behavioral description of a 2-to-4 line decoder with active high enable *(unique 1 outputs)* using ***case statements***:

- Specifications:
  - Inputs: En, A [1:0]
  - Outputs: Y [3:0]

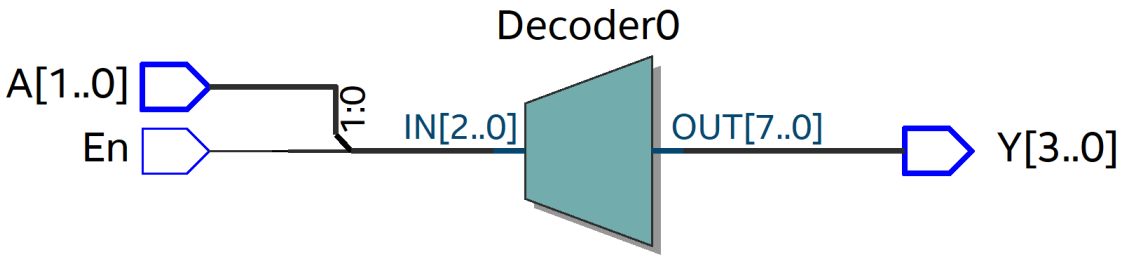
- Truth Table:

En	*A[1]*	*A[0]*	*Y*
0	-	-	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

Example 4: Design Entry (2x4 Dec)

```
// Behavioral description of a 2x4 Decoder using a case statement
//
module Decoder_2x4 (
    input wire        En,
    input wire  [1:0] A,
    output reg  [3:0] Y     // Y must be reg type (variable)
);

always @(*)
begin
    case ({En, A})
        3'b100   :  Y = 4'b0001;
        3'b101   :  Y = 4'b0010;
        3'b110   :  Y = 4'b0100;
        3'b111   :  Y = 4'b1000;
        default  :  Y = 4'b0000;
        // 3'b000, 3'b001, 3'b010, 3'b011 : Y = 4'b0000;
        endcase
end

endmodule
```

En	*A[1]*	*A[0]*	*Y*
0	-	-	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000



Example 4: Testbench File (2x4 Dec)

```
// Testbench file of 2x1 Decoder
//
`timescale 1 ns / 1 ps
module tb_Decoder_2x4 ();

   // inputs as reg, outputs as wire
   reg         En;
   reg   [1:0] A;
   wire  [3:0] Y;

   // instantiate UUT
   Decoder_2x4 UUT ( .A(A), .En(En), .Y(Y) );

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);

      {En, A}  = 3'b000;   #5
      {En, A}  = 3'b001;   #5
      {En, A}  = 3'b010;   #5
      {En, A}  = 3'b011;   #5

      {En, A}  = 3'b100;   #5
      {En, A}  = 3'b101;   #5
      {En, A}  = 3'b110;   #5
      {En, A}  = 3'b111;   #5

      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial begin
      $monitor("Time = %2d ns\t En = %b A = %b\t Y = %b", $time, En, A, Y);
   end

endmodule
```

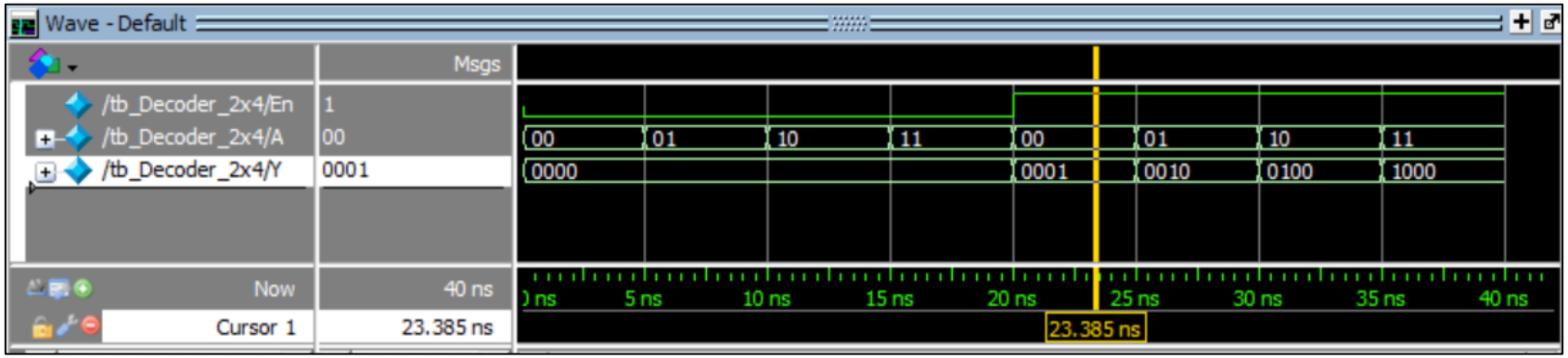
Example 4: Simulation Output (2x4 Dec)

- Standard Output:

```
# Starting simulation at 0 ns...
# Time =  0 ns  En = 0 A = 00  Y = 0000
# Time =  5 ns  En = 0 A = 01  Y = 0000
# Time = 10 ns  En = 0 A = 10  Y = 0000
# Time = 15 ns  En = 0 A = 11  Y = 0000
# Time = 20 ns  En = 1 A = 00  Y = 0001
# Time = 25 ns  En = 1 A = 01  Y = 0010
# Time = 30 ns  En = 1 A = 10  Y = 0100
# Time = 35 ns  En = 1 A = 11  Y = 1000
# Finished simulation at 40 ns.
```

En	*A[1]*	*A[0]*	*Y*
0	-	-	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

- Waveform:



Other Variants: casez and casex

- A normal **case** expression needs an **EXACT match** (0, 1, z, x)
  - **z and x** are treated as **it is**
- In a **casez** statement:
  - **z and ?** are treated as **don't care** *(i.e., the corresponding bit does not need to be matched)*
- In a **casex** statement:
  - **z, x, and ?** are treated as **don't care**

Example 5: 4x2 Priority Encoder

- Verilog behavioral description of a 4x2 priority encoder using *casez statements*:

- Specifications:
  - Inputs: D [3:0]
  - Outputs: Y [1:0] , V

- Notes:
  - *V* = valid output indicator
  - *D[3]* = highest priority

- Truth Table:

D[3]	*D[2]*	*D[1]*	*D[0]*	*Y*	*V*
0	0	0	0	xx	0
0	0	0	1	00	1
0	0	1	x	01	1
0	1	x	x	10	1
1	x	x	x	11	1

Example 5: Design Entry (4x2 Priority Encoder)

D[3]	*D[2]*	*D[1]*	*D[0]*	*Y*	*V*
0	0	0	0	xx	0
0	0	0	1	00	1
0	0	1	x	01	1
0	1	x	x	10	1
1	x	x	x	11	1

```
// Behavioral description of a 4x2 Priority Encoder using a case statement
//
module PriorityEncoder_4x2 (
    input wire  [3:0] D,
    output reg  [1:0] Y,     // Y must be reg type (variable)
    output reg        V      // V must be reg type (variable)
);

always @(*)
begin
    casez (D)
       4'b1???  :  begin    Y = 2'b11;  V = 1'b1;   end
       4'b01??  :  begin    Y = 2'b10;  V = 1'b1;   end
       4'b001?  :  begin    Y = 2'b01;  V = 1'b1;   end
       4'b0001  :  begin    Y = 2'b00;  V = 1'b1;   end
       default  :  begin    Y = 2'bxx;  V = 1'b0;   end
    endcase
end

endmodule
```

Example 5: Testbench File (4x2 PE)

```
// Testbench file of 4x2 Priority Encoder
//
`timescale 1 ns / 1 ps
module tb_PriorityEncoder_4x2 ();

   // inputs as reg, outputs as wire
   reg   [3:0] D;
   wire  [1:0] Y;
   wire        V;

   // instantiate UUT
   PriorityEncoder_4x2 UUT ( .D(D), .Y(Y), .V(V) );

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);

      D = 4'b0000;             // initial value of D

      repeat(16)               // looping statement
      #5    D = D + 4'b0001;   // D is incremented by 1 for 16 times

      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial begin
      $monitor("Time = %2d ns\t D = %b Y = %b\t V = %b", $time, D, Y, V);
   end

endmodule
```

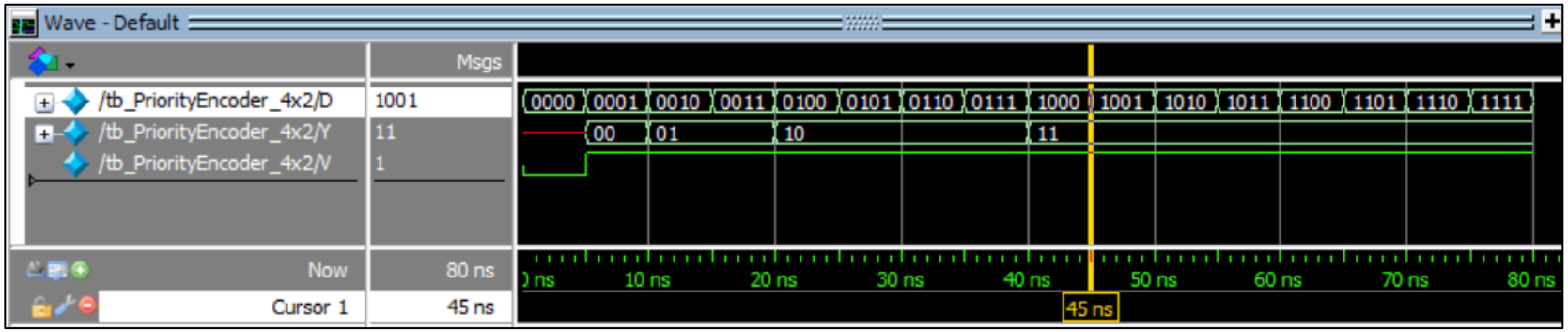
Example 5: Simulation Output (4x2 PE)

- Standard Output:

D[3]	*D[2]*	*D[1]*	*D[0]*	*Y*	*V*
0	0	0	0	xx	0
0	0	0	1	00	1
0	0	1	x	01	1
0	1	x	x	10	1
1	x	x	x	11	1

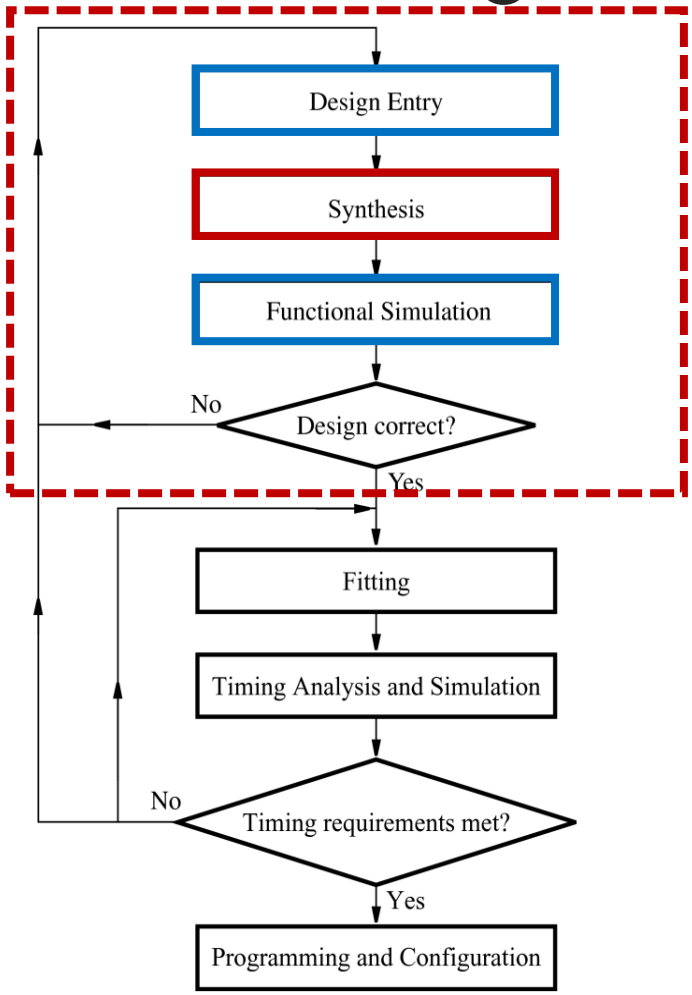
```
# Starting simulation at 0 ns...
# Time =  0 ns  D = 0000 Y = xx  V = 0
# Time =  5 ns  D = 0001 Y = 00  V = 1
# Time = 10 ns  D = 0010 Y = 01  V = 1
# Time = 15 ns  D = 0011 Y = 01  V = 1
# Time = 20 ns  D = 0100 Y = 10  V = 1
# Time = 25 ns  D = 0101 Y = 10  V = 1
# Time = 30 ns  D = 0110 Y = 10  V = 1
# Time = 35 ns  D = 0111 Y = 10  V = 1
# Time = 40 ns  D = 1000 Y = 11  V = 1
# Time = 45 ns  D = 1001 Y = 11  V = 1
# Time = 50 ns  D = 1010 Y = 11  V = 1
# Time = 55 ns  D = 1011 Y = 11  V = 1
# Time = 60 ns  D = 1100 Y = 11  V = 1
# Time = 65 ns  D = 1101 Y = 11  V = 1
# Time = 70 ns  D = 1110 Y = 11  V = 1
# Time = 75 ns  D = 1111 Y = 11  V = 1
# Finished simulation at 80 ns.
```

- Waveform:



Logic Synthesis of HDL-based Designs

- **Design or Logic synthesis:** derives a list of physical components and their interconnections (*netlist*) from the design files
  - Converting HDL descriptions into physical hardware components in the target device (FPGA)
  - Synthesizable Verilog code
  - The purpose of writing HDL code is **to infer hardware** *(rather than describing a sequential algorithm in C)*



Synthesis Pros and Cons

- Advantages:
  - Automatically manages many details of the design process, which results *in less human-related errors*, *faster implementation cycles,* and *increased productivity*.
  - Abstracts design (HDL code) from any particular technology.
  - Can sometimes lead to more optimal implementations than manual means.

- Disadvantages:
  - Because synthesis relies on a program, optimization is limited by the effectiveness of the algorithms used. This can sometimes lead to non-optimal implementations.

Synthesis of Operators

- **Logical operators** map into **logic gate primitives**.
- **Arithmetic operators** map into **adders, subtractors**.
  - *Unsigned 2's complement notation*
  - **, /, and % operators usually don't have physical counterparts*
- **Relational operators** generate **comparators**.
- **Conditional expressions** generate **logic** or **multiplexers**.

Guidelines for Good Synthesis

- For combinational always statements, specify complete sensitivity list.
  - Failure to do so may result in mismatch between functional simulations and post-synthesis simulations.
  - A simple solution is to use always @(*)

```
// Verilog 2001 syntax
reg X;
always @(*)
if (S == 0)
   X = A;
else
   X = B;
```

```
// Verilog 2001 syntax
reg X;
always @(*)
  case (S)
    1'b0    : X = A;
    default : X = B;
  endcase
```

Guidelines for Good Synthesis

- Ensure that all conditional statements are completely specified if your target is a combinational circuit.
  - For *if-else*: presence of the **else** condition
  - For *case*: all possible cases are listed or the use of the **default** case
  - This avoids inferring latches *(unintended memory)*.

✓
```
// Verilog 2001 syntax
reg X;
always @(*)
if (S == 0)
   X = A;
else
   X = B;
```

✓
```
// Verilog 2001 syntax
reg X;
always @(*)
  case (S)
    1'b0    : X = A;
    default : X = B;
  endcase
```

Guidelines for Good Synthesis

- Ensure that the outputs are assigned in all branches if your target is a combinational circuit.
  - This avoids inferring latches *(unintended memory)*.

✗

```
// missing output assignment
// in some branches
reg x, y;
always @(*)
   if (mode == 1'b0) begin
        x = 1'b1;
        y = 1'b0;
   end
   else begin
        y = a & b;
   end
```

✓

```
// outputs are assigned in
// all branches
reg x, y;
always @(*)
   if (mode == 1'b0) begin
        x = 1'b1;
        y = 1'b0;
   end
   else begin
        x = 1'b1;
        y = a & b;
   end
```

Guidelines for Good Synthesis

- Use blocking statements (=) to model combinational circuits.
  - The hardware generated by blocking assignments *is dependent on the order of assignments* (sequential order of execution).

```
// using blocking statements
reg x, y;
always @(*)
begin
   x = a & b;
   y = x | c;
end
```

Execution:

1. x is evaluated
2. y is evaluated using value from (1)

Guidelines for Good Synthesis

- Assign a variable only in a single always block.
  - This will *infer multiple drivers or sources* but is **unsynthesizable**.
  - This may lead to race conditions.

✗

```
// multiple always assignments
// of the same variable

reg y;

always @(*)
   if (clear) y = 1'b0;

always @(*)
   y = a & b;
```

✓

```
// single always assignment
// of the same variable

reg y;

always @(*)
   if (clear)
        y = 1'b0;
   else
        y = a & b;
```

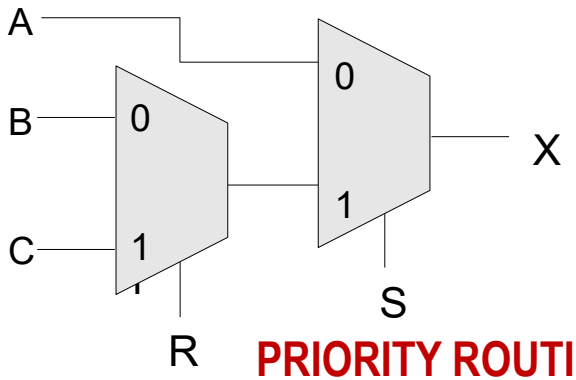
Guidelines for Good Synthesis

- Use case statements if you do not wish to imply priority.
  - If your *if* statement contains more than 3 conditions, consider using the *case* statement to improve the parallelism of your design and the clarity of your code.

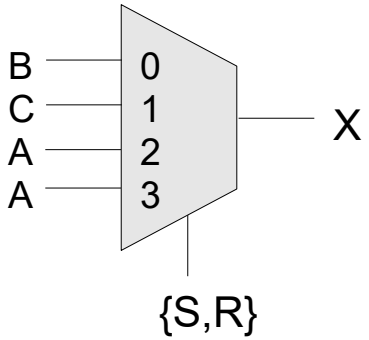
✓
```
reg X;
always @(*)
  if (S == 0)      X = A;
  else
    if (R == 0)    X = B;
    else           X = C;
```

VS.

✓
```
reg X;
always @(*)
  case ({S, R})
    2'b00        : X = B;
    2'b01        : X = C;
    default      : X = A;
  endcase
```



PRIORITY ROUTING NETWORK



MULTIPLEXING NETWORK

Guidelines for Good Synthesis

- Always think of HARDWARE when coding in HDL. HDL code is NOT A PROGRAM *(except for testbenches)*.

End of Unit 5