Behavioral Modeling in Verilog

COE 202

Digital Logic Design

Dr. Muhamed Mudawar

King Fahd University of Petroleum and Minerals

Presentation Outline

- Introduction to Dataflow and Behavioral Modeling
- Verilog Operators
- Module Parameters
- Modeling Adders, Comparators, Multiplexers
- Always Block with Sensitivity List
- Procedural Statements: IF and CASE
- Modeling Decoder, Priority Encoder, and ALU

Verilog Four-Valued Logic

- Verilog Value Set consists of four basic values:
  - **0** – represents a logic zero, or false condition
  - **1** – represents a logic one, or true condition
  - **X** – represents an unknown logic value
  - **Z** – represents a high-impedance value

x or **X** represents an unknown or uninitialized value

z or **Z** represents the output of a disabled tri-state buffer

Nets and Variables

Verilog has two major data types:

1. **Net data types:** are connections between parts of a design
2. **Variable data types:** can store data values

- The **`wire`** is a net data type (physical connection)
  - A wire cannot store the value of a procedural assignment
  - However, a wire can be driven by continuous assignment
- The **`reg`** is a variable data type
  - Can store the value of a procedural assignment
  - However, cannot be driven by continuous assignment
  - Other variable types: **`integer`**, **`time`**, **`real`**, and **`realtime`**

Modeling Circuits in Verilog

Four levels of modeling circuits in Verilog

1. **Gate-Level Modeling**

   Lowest-level modeling using Verilog primitive gates

2. **Structural Modeling** using module instantiation

   Describes the structure of a circuit with modules at different levels

3. **Dataflow Modeling** using concurrent assign statements

   Describes the flow of data between input and output

4. **Behavioral Modeling** using procedural blocks and statements

   Describes what the circuit does at a higher level of abstraction

Can also mix different models in the same design

Dataflow and Behavioral Modeling

- Dataflow Modeling using Continuous Assignment
  - Used mostly for describing Boolean equations and combinational logic
  - Verilog provides a rich set of operators
  - Can describe: adders, comparators, multiplexers, etc.
  - Synthesis tool can map a dataflow model into a target technology
- Behavioral Modeling using Procedural Blocks and Statements
  - Describes what the circuit does at a functional and algorithmic level
  - Encourages designers to rapidly create a prototype
  - Can be verified easily with a simulator
  - Some procedural statements are synthesizable (Others are NOT)

Continuous Assignment

- The **assign** statement defines continuous assignment
- Syntax: **assign** [**#delay**] ***net_name = expression;***
- Assigns ***expression*** value to ***net_name*** (wire or output port)
- The optional **#delay** specifies the delay of the assignment
- Continuous assignment statements are **concurrent**
- Can appear in any order inside a module
- Continuous assignment can model combinational circuits
- Describes the flow of data between input and output

Verilog Operators

Bitwise Operators	
`~a`	Bitwise NOT
`a & b`	Bitwise AND
`a | b`	Bitwise OR
`a ^ b`	Bitwise XOR
`a ~^ b`	Bitwise XNOR
`a ^~ b`	Same as ~^

Arithmetic Operators	
`a + b`	ADD
`a - b`	Subtract
`-a`	Negate
`a * b`	Multiply
`a / b`	Divide
`a % b`	Remainder

Relational Operators	
`a == b`	Equality
`a != b`	Inequality
`a < b`	Less than
`a > b`	Greater than
`a <= b`	Less or equal
`a >= b`	Greater or equal

Reduction Operators	
`&a`	AND all bits
`|a`	OR all bits
`^a`	XOR all bits
`~&a`	NAND all bits
`~|a`	NOR all bits
`~^a`	XNOR all bits

Shift Operators	
`a << n`	Shift Left
`a >> n`	Shift Right

Miscellaneous Operators	
`sel?a:b`	Conditional
`{a, b}`	Concatenate

Reduction operators produce a 1-bit result

Relational operators produce a 1-bit result

`{a, b}` concatenates the bits of **a** and **b**

Bit Vectors in Verilog

- A Bit Vector is multi-bit declaration that uses a single name
- A Bit Vector is specified as a Range **[msb:lsb]**
- **msb** is ***most-significant bit*** and **lsb** is ***least-significant bit***
- Examples:

```
input  [15:0] A;    // A is a 16-bit input vector
output [0:15] B;    // Bit 0 is most-significant bit
wire   [3:0]  W;    // Bit 3 is most-significant bit
```

- **Bit select**: **W[1]** is bit **1** of **W**
- **Part select**: **A[11:8]** is a 4-bit select of **A** with range **[11:8]**
- The part select range must be consistent with vector declaration

Reduction Operators

```
module Reduce
   ( input [3:0] A, B, output X, Y, Z );
   // A, B are input vectors, X, Y, Z are 1-bit outputs

   // X = A[3] | A[2] | A[1] | A[0];
   assign X = |A;

   // Y = B[3] & B[2] & B[1] & B[0];
   assign Y = &B;

   // Z = X & (B[3] ^ B[2] ^ B[1] ^ B[0]);
   assign Z = X & (^B);

endmodule
```

Concatenation Operator { }

```
module Concatenate
  ( input [7:0] A, B, output [7:0] X, Y, Z );
  // A, B are input vectors, X, Y, Z are output vectors

  // X = A is right-shifted 3 bits using { } operator
  assign X = {3'b000, A[7:3]};

  // Y = A is right-rotated 3 bits using { } operator
  assign Y = {A[2:0], A[7:3]};

  // Z = selecting and concatenating bits of A and B
  assign Z = {A[5:4], B[6:3], A[1:0]};

endmodule
```

Integer Literals (Constant Values)

- Syntax: **[size]['base]value**

  size (optional) is the number of bits in the value

  'base can be: **'b**(binary), **'o**(octal), **'d**(decimal), or **'h**(hex)

  value can be in binary, octal, decimal, or hexadecimal

- If the **'base** is not specified then decimal **value**
- Examples:

  8'b1011_1101 (8-bit binary), **'hA3F0** (16-bit hexadecimal)

  16'o56377 (16-bit octal), **32'd999** (32-bit decimal)

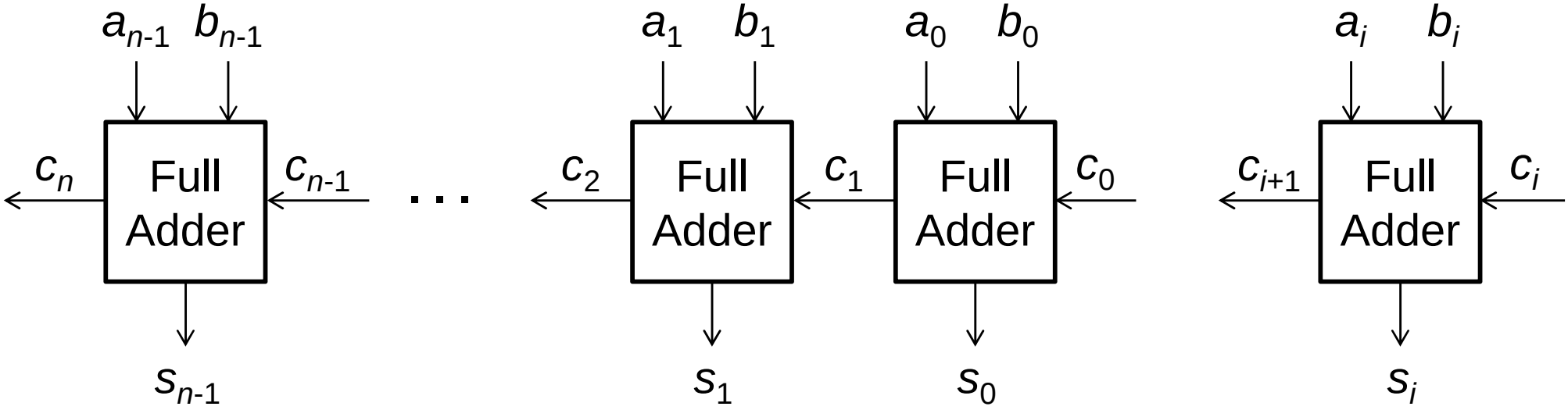
- The underscore _ can be used to enhance readability of **value**
- When **size** is fewer bits than **value**, upper bits are truncated

Ripple Carry Adder

- Using **identical copies** of a full adder to build a large adder
- The **cell** (iterative block) is a **full adder**

  Adds 3 bits: a_i, b_i, c_i, Computes: Sum s_i and Carry-out c_{i+1}

- Carry-out of cell i becomes carry-in to cell (i +1)



16-Bit Adder with Array Instantiation

```
// Input  ports: 16-bit a and b, 1-bit cin (carry input)
// Output ports: 16-bit sum, 1-bit cout (carry output)

module Adder_16 (input  [15:0] a, b, input cin,
                 output [15:0] sum, output cout);

  wire [16:0] c;                  // carry bits
  assign c[0] = cin;              // carry input
  assign cout = c[16];            // carry output

  // Instantiate an array of 16 Full Adders
  // Each instance [i] is connected to bit select [i]

  Full_Adder FA [15:0] (a[15:0], b[15:0], c[15:0],
                        c[16:1], sum[15:0]);
endmodule
```

Array Instantiation of identical modules by a single statement

16-Bit Adder with Continuous Assignment

```
// Input  ports: 16-bit a and b, 1-bit cin (carry input)
// Output ports: 16-bit sum, 1-bit cout (carry output)

module Adder_16 (input  [15:0] a, b, input cin,
                 output [15:0] sum, output cout);

  wire [16:0] c;                  // carry bits
  assign c[0] = cin;              // carry input
  assign cout = c[16];            // carry output


  // assignment of 16-bit vectors
  assign sum[15:0] = (a[15:0] ^ b[15:0]) ^ c[15:0];
  assign c[16:1]   = (a[15:0] & b[15:0]) |
                     (a[15:0] ^ b[15:0]) & c[15:0];

endmodule
```

16-bit Adder with the + Operator

```
module Adder16
  ( input [15:0] A, B, input cin,
    output [15:0] Sum, output cout );

  // A and B are 16-bit input vectors
  // Sum is a 16-bit output vector

  // {cout, Sum} is a concatenated 17-bit vector
  // A + B + cin is 16-bit addition + input carry
  // The + operator is translated into an adder

  assign {cout, Sum} = A + B + cin;

endmodule
```

Modeling a Parametric n-bit Adder

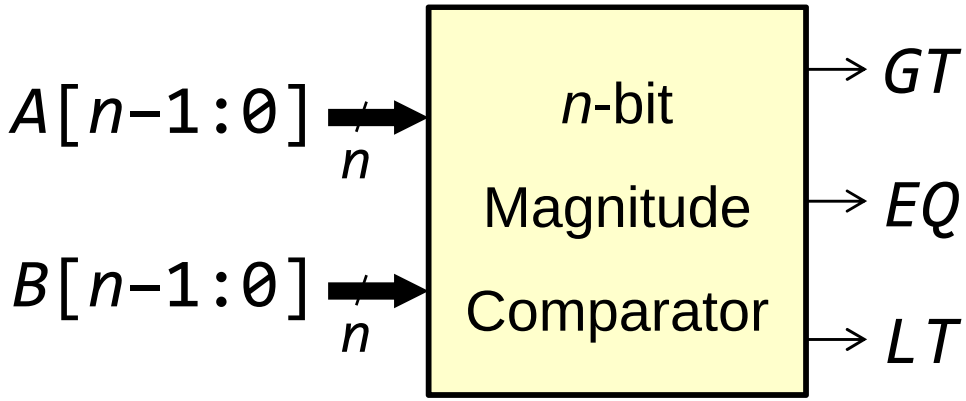
```
// Parametric n-bit adder, default value for n = 16
module Adder #(parameter n = 16)
  ( input [n-1:0] A, B, input cin,
    output [n-1:0] Sum, output cout );

  // A and B are n-bit input vectors
  // Sum is an n-bit output vector

  // The + operator is translated into an n-bit adder
  // Only one assign statement is used

  assign {cout, Sum} = A + B + cin;

endmodule
```

Instantiating Adders of Various Sizes

```
// Instantiate a 16-bit adder (parameter n = 16)
// A1, B1, and Sum1 must be 16-bit vectors
Adder #(16) adder16 (A1, B1, Cin1, Sum1, Cout1);

// Instantiate a 32-bit adder (parameter n = 32)
// A2, B2, and Sum2 must be 32-bit vectors
Adder #(32) adder32 (A2, B2, Cin2, Sum2, Cout2);

// If parameter is not specified, it defaults to 16
Adder adder16 (A1, B1, Cin1, Sum1, Cout1);
```

Modeling a Magnitude Comparator

```
// n-bit magnitude comparator, No default value for n
module Comparator #(parameter n)
  (input [n-1:0] A, B,
   output GT, EQ, LT);

  // A and B are n-bit input vectors (unsigned)
  // GT, EQ, and LT are 1-bit outputs

  assign GT = (A > B);
  assign EQ = (A == B);
  assign LT = (A < B);

endmodule
```



Instantiating Comparators of Various Sizes

```
// Instantiate a 16-bit comparator (n = 16)
// A1 and B1 must be declared as 16-bit vectors
Comparator #(16) comp16 (A1, B1, GT1, EQ1, LT1);

// Instantiate a 32-bit comparator (n = 32)
// A2 and B2 must be declared as 32-bit vectors
Comparator #(32) comp32 (A2, B2, GT2, EQ2, LT2);

// WRONG Instantiation: Must specify parameter n
Comparator comp32 (A2, B2, GT2, EQ2, LT2);
```

Conditional Operator

- Syntax:

  Boolean_expr ? True_expression : False_expression

  If ***Boolean_expr*** is true then select ***True_expression***

  Else select ***False_Expression***

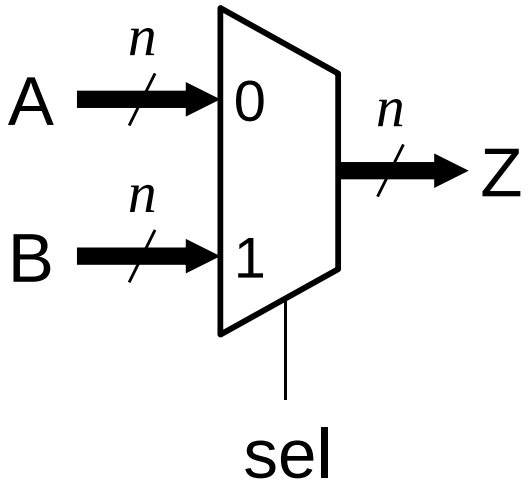
- Examples:

```
assign max = (a>b)? a : b;  // maximum of a and b

assign min = (a>b)? b : a;  // minimum of a and b
```

- Conditional operators can be nested

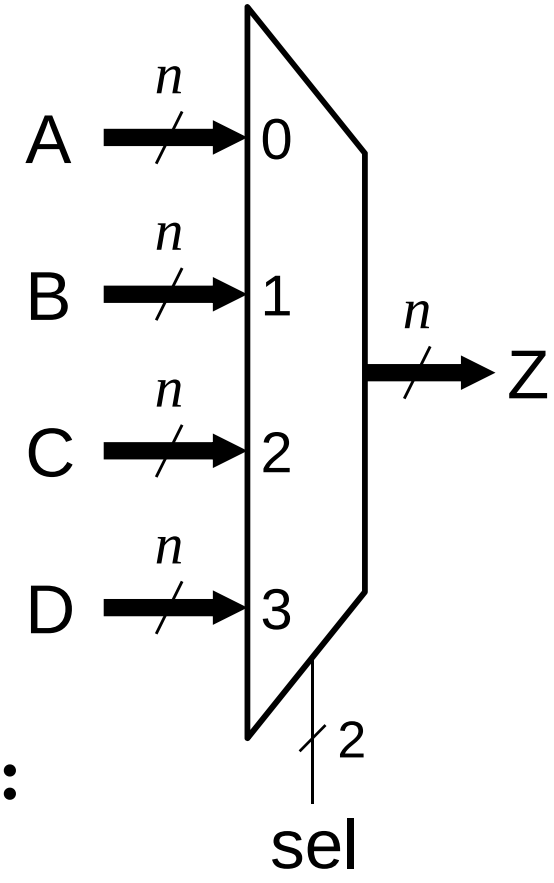
Modeling a 2x1 Multiplexer

```
// Parametric 2x1 Mux, default value for n = 1
module Mux2 #(parameter n = 1)
  ( input  [n-1:0] A, B, input sel,
    output [n-1:0] Z);

  // A and B are n-bit input vectors
  // Z is the n-bit output vector
  // if (sel==0) Z = A; else Z = B;
  // Conditional operator used for selection

  assign Z = (sel == 0)? A : B;

endmodule
```



Modeling a 4x1 Multiplexer

```
// Parametric 4x1 Mux, default value for n = 1
module Mux4 #(parameter n = 1)
  ( input  [n-1:0] A, B, C, D,
    input  [1:0] sel,
    output [n-1:0] Z );

  // sel is a 2-bit vector
  // Nested conditional operators
  assign Z = (sel[1] == 0) ?
             ((sel[0] == 0) ? A : B) :
             ((sel[0] == 0) ? C : D);
endmodule
```



Behavioral Modeling

- Uses procedural blocks and procedural statements
- There are two types of **procedural** blocks in Verilog

1. The **`initial`** block
   - Executes the enclosed statement(s) one time only
2. The **`always`** block
   - Executes the enclosed statement(s) repeatedly until simulation terminates

- The body of the **`initial`** and **`always`** blocks is **procedural**
  - Can enclose one or more **procedural statements**
  - Procedural statements are surrounded by **`begin`** … **`end`**
- Multiple procedural blocks can appear in any order inside a module and run in parallel inside the simulator

Example of Initial and Always Blocks

```
module behave;
  reg clk;                // 1-bit variable
  reg [15:0] A;           // 16-bit variable
  initial begin           // executed once
    clk = 0;              // initialize clk
    A = 16'h1234;         // initialize A
    #200 $finish
  end
  always begin            // executed always
    #10 clk = ~clk;       // invert clk every 10 ns
  end
  always begin            // executed always
    #20 A = A + 1;        // increment A every 20 ns
  end
endmodule
```

Always Block with Sensitivity List

- Syntax:

```
always @(sensitivity list) begin
   procedural statements
end
```

- An **always** block can have a ***sensitivity list***
- Sensitivity list is a list of signals: **@(*signal1, signal2,* …)**

  The sensitivity list triggers the execution of the **always** block

  When there is a ***change of value in any listed signal***

  Otherwise, the **always** block does nothing until another change occurs on a signal in the sensitivity list

Sensitivity List for Combinational Logic

- For combinational logic, the sensitivity list **must include**:

  ALL the signals that are read inside the **always** block

  Example: **A**, **B**, and **sel** must be in the sensitivity list below:

```
always @(A, B, sel) begin
  if (sel == 0) Z = A;
  else Z = B;
end
```

A, **B**, and **sel** are read inside the **always** block

- Combinational logic can also use: **@(*)** or **@***

  @(*) is automatically sensitive to all the signals that are read inside the **always** block

If Statement

- The **`if`** statement is procedural
- Can only be used inside a procedural block
- Syntax:

  `if` **(*`expression`*)** ***`statement`***

  [**`else`** ***`statement`*** **]**

- The **`else`** part is optional

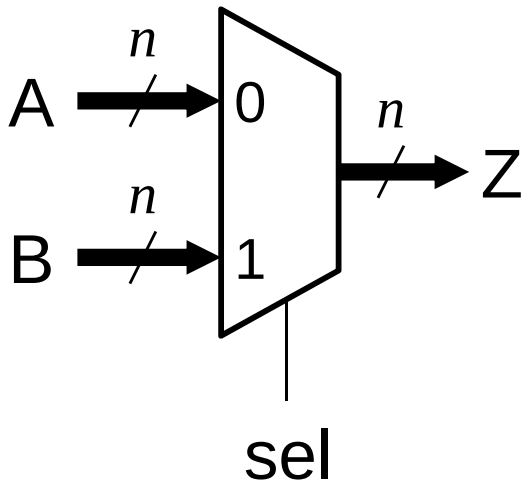
  A ***`statement`*** can be simple or compound

  A ***`compound statement`*** is surrounded by **`begin ... end`**

- **`if`** statements can be nested
- Can be nested under **`if`** or under **`else`** part

Modeling a 2x1 Multiplexer

```
// Behavioral Modeling of a Parametric 2x1 Mux
module Mux2 #(parameter n = 1)
  ( input [n-1:0] A, B, input sel,
    output reg [n-1:0] Z);

  // Output Z must be of type reg
  // Sensitivity list = @(A, B, sel)
  always @(A, B, sel) begin
    if (sel == 0) Z = A;
    else Z = B;
  end

endmodule
```



Modeling a 3x8 Decoder

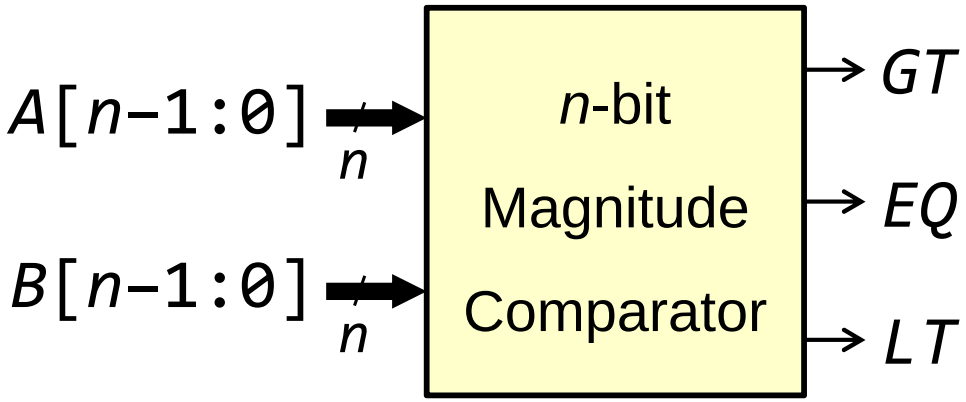
```
module Decoder3x8 (input [2:0] A, output reg [7:0] D);
  // Sensitivity list = @(A)
  always @(A) begin
    if (A == 0)       D = 8'b00000001;
    else if (A == 1) D = 8'b00000010;
    else if (A == 2) D = 8'b00000100;
    else if (A == 3) D = 8'b00001000;
    else if (A == 4) D = 8'b00010000;
    else if (A == 5) D = 8'b00100000;
    else if (A == 6) D = 8'b01000000;
    else             D = 8'b10000000;
  end
endmodule
```

Modeling a 4x2 Priority Encoder

```
module Priority_Encoder4x2
  (input [3:0] D, output reg V, output reg [1:0] A);

  // sensitivity list = @(D)
  always @(D) begin
    if (D[3])      {V, A} = 3'b111;
    else if (D[2]) {V, A} = 3'b110;
    else if (D[1]) {V, A} = 3'b101;
    else if (D[0]) {V, A} = 3'b100;
    else           {V, A} = 3'b000;
  end
endmodule
```

Modeling a Magnitude Comparator

```
// Behavioral Modeling of a Magnitude Comparator
module Comparator #(parameter n = 1)
  (input [n-1:0] A, B, output reg GT, EQ, LT);
  // Sensitivity list = @(A, B)
 always @(A, B) begin
    if (A > B)
      {GT,EQ,LT}='b100;
    else if (A == B)
      {GT,EQ,LT}='b010;
    else
      {GT,EQ,LT}='b001;
  end
endmodule
```



Case Statement

- The **case** statement is procedural (used inside **always** block)
- Syntax:

```
case (expression)
   case_item1: statement
   case_item2: statement
   . . .
   default:    statement
endcase
```

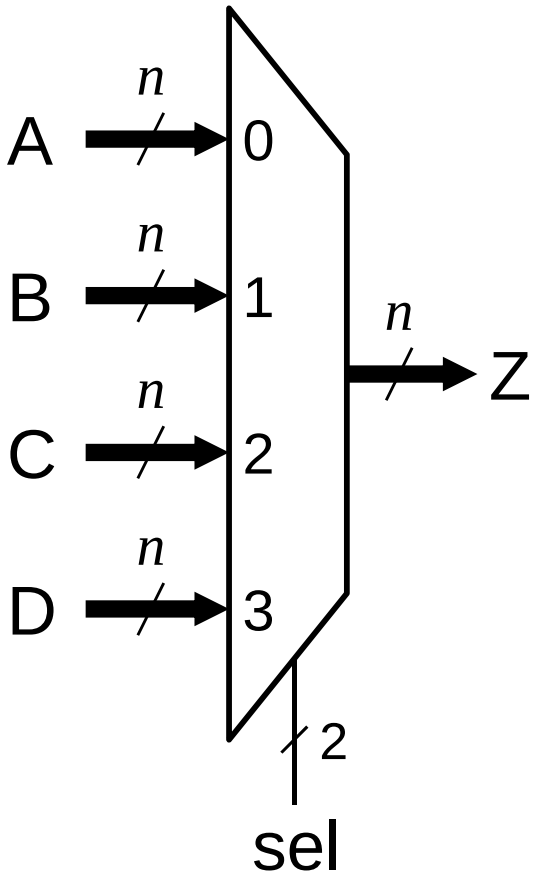
The **default** case is optional

A ***statement*** can be simple or compound

A ***compound statement*** is surrounded by **begin ... end**

Modeling a Mux with a Case Statement

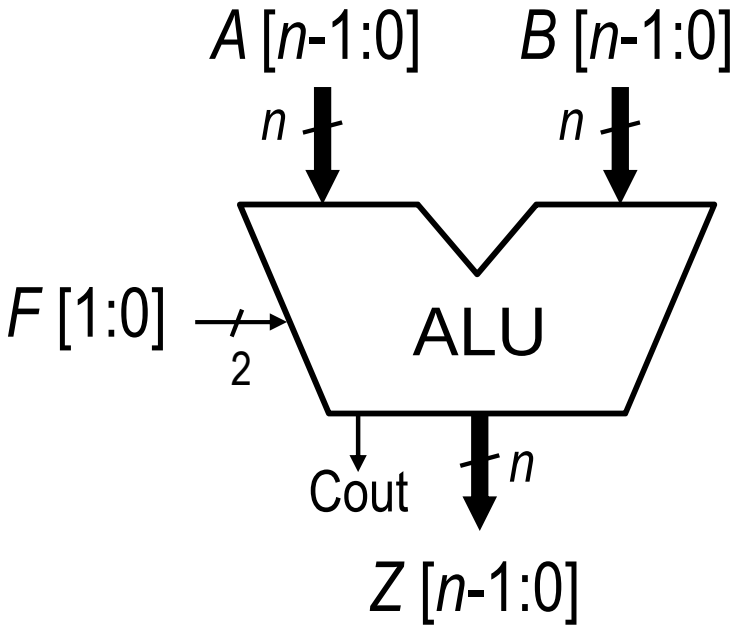
```
module Mux4 #(parameter n = 1)
  ( input  [n-1:0] A, B, C, D, input [1:0] sel,
    output reg [n-1:0] Z );

  // @(*) is @(A, B, C, D, sel)
  always @(*) begin
    case (sel)
      2'b00:   Z = A;
      2'b01:   Z = B;
      2'b10:   Z = C;
      default: Z = D;
    endcase
  end
endmodule
```



Modeling a Multifunction ALU

```
// Behavioral Modeling of an ALU
module ALU #(parameter n = 16)
  ( input  [n-1:0] A, B, input [1:0] F,
    output reg [n-1:0] Z, output reg Cout );
  // @(*) is @(A, B, F)
  always @(*) begin
    case (F)
      2'b00:   {Cout,Z} = A+B;
      2'b01:   {Cout,Z} = A-B;
      2'b10:   {Cout,Z} = A&B;
      default: {Cout,Z} = A|B;
    endcase
  end
endmodule
```

ALU Symbol



Modeling a BCD to 7-Segment Decoder

```
module BCD_to_7Seg_Decoder
  ( input  [3:0] BCD, output reg [6:0] Seg )
  always @(BCD) begin
    case (BCD)
      0: Seg = 7'b1111110;   1: Seg = 7'b0110000;
      2: Seg = 7'b1101101;   3: Seg = 7'b1111001;
      4: Seg = 7'b0110011;   5: Seg = 7'b1011011;
      6: Seg = 7'b1011111;   7: Seg = 7'b1110000;
      8: Seg = 7'b1111111;   9: Seg = 7'b1111011;
      default: Seg = 7'b0000000;
    endcase
  end
endmodule
```