

Verilog

Part I

Presentation Outline

- ❖ Hardware Description Language
- ❖ Logic Simulation versus Synthesis
- ❖ Verilog Module
- ❖ Gate-Level Description and Gate Delays
- ❖ Module Instantiation
- ❖ Continuous Assignment
- ❖ Writing a Simple Test Bench

Hardware Description Language

- ❖ Describes the hardware of digital systems in a textual form
- ❖ Describes the hardware structures and behavior
- ❖ Can represent logic diagrams, expressions, and complex circuits
- ❖ NOT a software programming language
- ❖ Two standard hardware description languages (HDLs)
 - 1. Verilog** (will be studied in this course)
 - 2. VHDL** (harder to learn than Verilog)

Verilog = "Verifying Logic"

- ❖ Invented as a simulation language in 1984 by Phil Moorby
- ❖ Opened to public in 1990 by Cadence Design Systems
- ❖ Became an IEEE standard in 1995 (Verilog-95)
- ❖ Revised and upgraded in 2001 (Verilog-2001)
- ❖ Revised also in 2005 (Verilog-2005)
- ❖ Verilog allows designers to describe hardware at different levels
 - ✧ Can describe anything from a single gate to a full computer system
- ❖ Verilog is supported by the majority of electronic design tools
- ❖ Verilog can be used for logic simulation and synthesis

Logic Simulation

- ❖ Logic simulator interprets the Verilog (HDL) description
- ❖ Produces **timing diagrams**
- ❖ Predicts how the hardware will behave before it is fabricated
- ❖ Simulation allows the detection of functional errors in a design
 - ✧ Without having to physically implement the circuit
- ❖ Errors detected during the simulation can be corrected
 - ✧ By modifying the appropriate statements in the Verilog description
- ❖ Simulating and verifying a design requires a **test bench**
- ❖ The test bench is also written in Verilog

Logic Synthesis

- ❖ Logic synthesis is similar to translating a program
- ❖ However, the output of logic synthesis is a digital circuit
- ❖ A digital circuit modeled in Verilog can be translated into a list of components and their interconnections, called **netlist**
- ❖ Synthesis can be used to fabricate an integrated circuit
- ❖ Synthesis can also target a Field Programmable Gate Array
 - ✧ An FPGA chip can be configured to implement a digital circuit
 - ✧ The digital circuit can also be modified by reconfiguring the FPGA
- ❖ Logic simulation and synthesis are automated
 - ✧ Using special software, called Electronic Design Automation (EDA) tools

HDL Verilog

❖ A module can be described in any one (or a combination) of the following modeling techniques:

1. **Gate – level modeling** using instantiation of primitive gates and user - defined modules
2. **Data flow modeling** using continuous assignment statements with keyword **assign**
3. **Behavioral modeling** using procedural assignment statements with keyword **always**

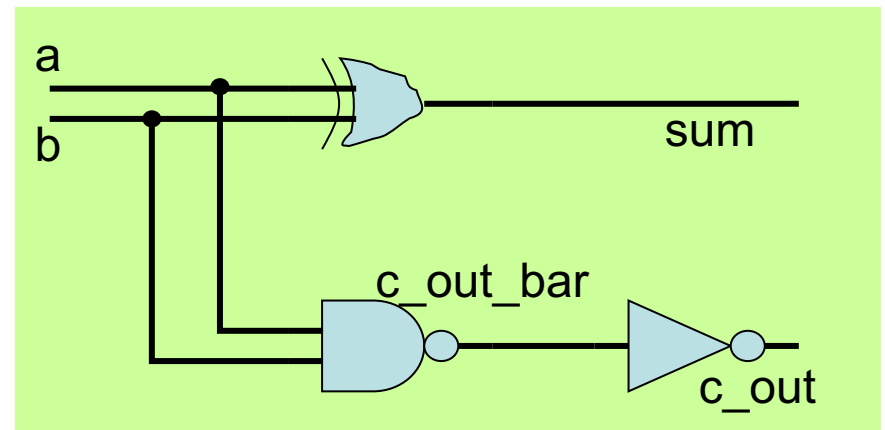
Structure Description in Verilog

```
module Add_half (sum, c_out, a, b);  
  input  a, b;  ← Declaration of port modes  
  output sum, c_out; ← Declaration of port modes  
  wire  c_out_bar; ← Declaration of internal signal  
  
  xor Gate1 (sum, a, b); ← Instantiation of primitive gates  
  nand (c_out_bar, a, b); ← Instantiation of primitive gates  
  not  (c_out, c_out_bar); ← Instantiation of primitive gates  
endmodule
```

Module name *Module ports*

Instance name *Instantiation of primitive gates*

Verilog keywords



Basic Gates

- ❖ Basic gates: **and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf**
- ❖ Verilog define these gates as **keywords**
- ❖ Each gate has an optional name
- ❖ Each gate has an output (listed first) and one or more inputs
- ❖ The **not** and **buf** gates can have only one input
- ❖ Examples:

and g1(x,a,b); // 2-input and gate named g1

or g2(y,a,b,c); // 3-input or gate named g2

nor g3(z,a,b,c,d); // 4-input nor gate named g3

↑ ↑ └───┘
name output inputs

Verilog Module

- ❖ A digital circuit is described in Verilog as a set of modules
- ❖ A module is the design entity in Verilog
- ❖ A module is declared using the **module** keyword
- ❖ A module is terminated using the **endmodule** keyword
- ❖ A module has a name and a list of **input** and **output** ports
- ❖ A module is described by a group of statements
- ❖ The statements can describe the module structure or behavior

Verilog Syntax

❖ **Keywords:** have special meaning in Verilog

Many keywords: **module**, **input**, **output**, **wire**, **and**, **or**, etc.

Keywords cannot be used as identifiers

❖ **Identifiers:** are user-defined names for modules, ports, etc.

Verilog is **case-sensitive**: **A** and **a** are different names

❖ **Comments:** can be specified in two ways (similar to C)

✧ Single-line comments begin with **//** and terminate at end of line

✧ Multi-line comments are enclosed between **/*** and ***/**

❖ **White space:** space, tab, newline can be used freely in Verilog

❖ **Operators:** operate on variables (similar to C: **~ & | ^ + -** etc.)

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

Gate level representation example: Half Adder

A half adder adds two bits: **a** and **b**

Two output bits:

1. Carry bit: **cout** = **a** · **b**

2. Sum bit: **sum** = **a** ⊕ **b**

```
module Half_Adder(a, b, cout, sum);
```

```
    input a, b;
```

```
    output sum, cout;
```

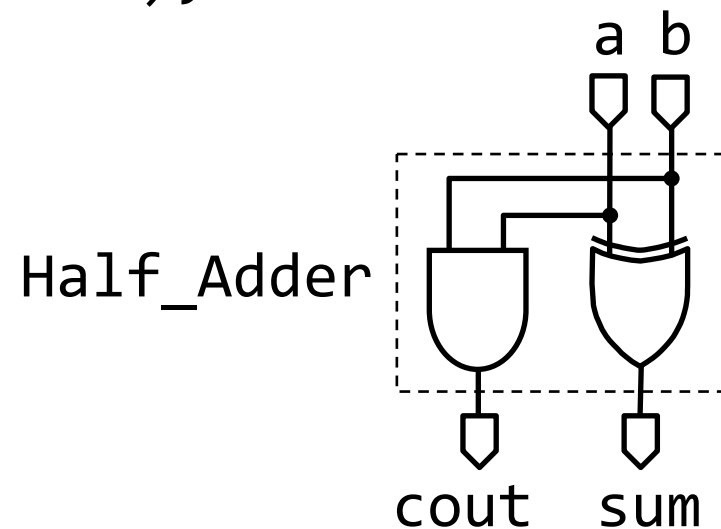
```
    and (cout, a, b);
```

```
    xor (sum, a, b);
```

```
endmodule
```

Truth Table

a	b	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Full Adder

- ❖ Full adder adds 3 bits: **a**, **b**, and **c**
 - ❖ Two output bits:
 1. Carry bit: **cout**
 2. Sum bit: **sum**
 - ❖ Sum bit is 1 if the number of 1's in the input is odd (odd function)
- sum** = $(a \oplus b) \oplus c$
- ❖ Carry bit is 1 if the number of 1's in the input is 2 or 3

$$\text{cout} = a \cdot b + (a \oplus b) \cdot c$$

Truth Table

a	b	c	cout	sum
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	1	1

Full Adder Module

```
module Full_Adder(input a, b, c, output cout, sum);
```

```
    wire w1, w2, w3;
```

```
    and (w1, a, b);
```

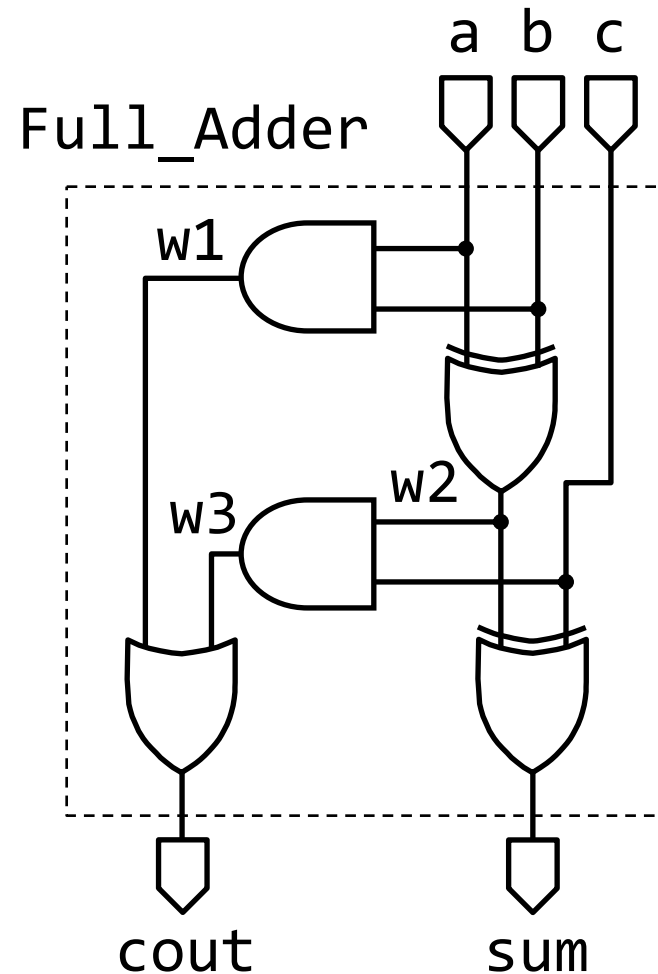
```
    xor (w2, a, b);
```

```
    and (w3, w2, c);
```

```
    xor (sum, w2, c);
```

```
    or  (cout, w1, w3)
```

```
endmodule
```



Modular Design

A full adder can be designed using two half adders and one OR gate

First Half Adder: HA1

$$w1 = a \cdot b$$

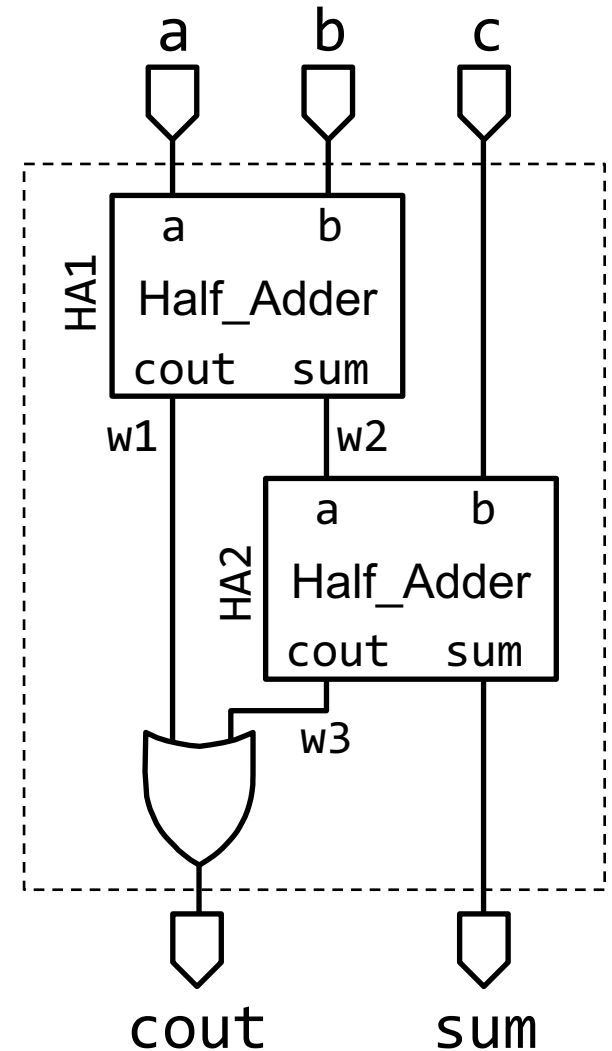
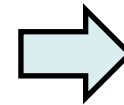
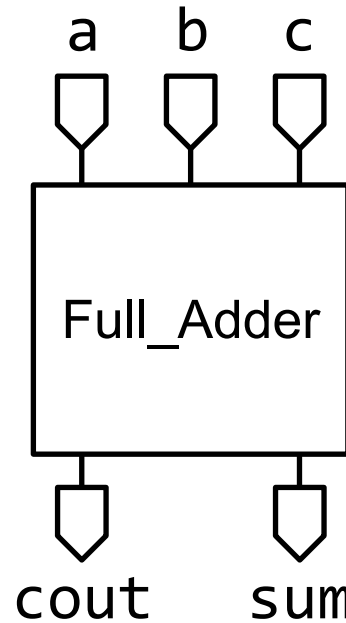
$$w2 = a \oplus b$$

Second Half Adder: HA2

$$w3 = w2 \cdot c = (a \oplus b) \cdot c$$

$$\text{sum} = w2 \oplus c = (a \oplus b) \oplus c$$

$$\text{cout} = w1 + w3 = a \cdot b + (a \oplus b) \cdot c$$

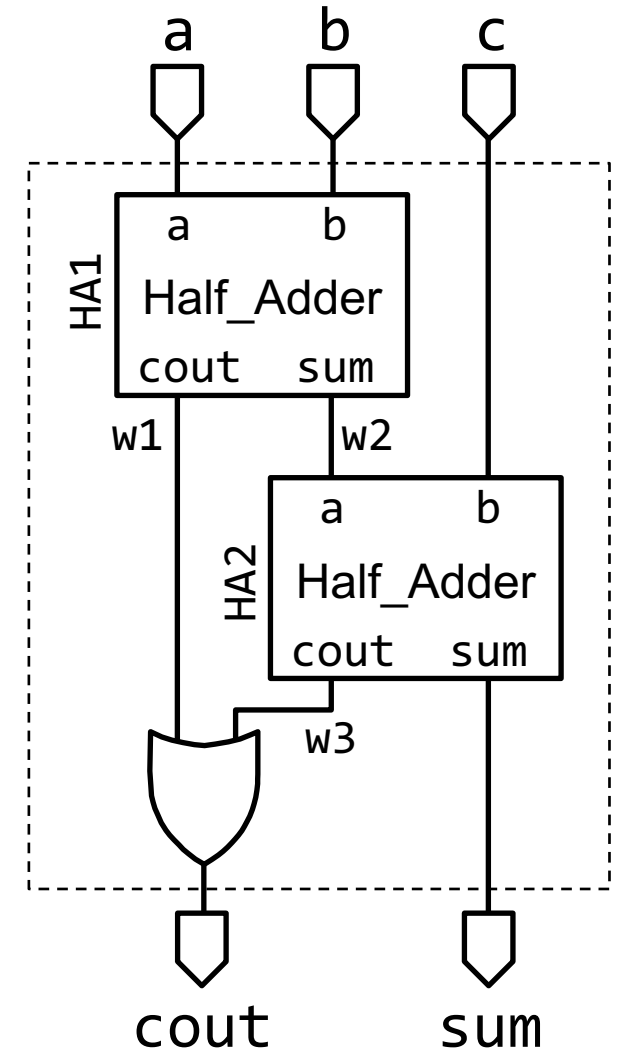


Module Instantiation

- ❖ Module declarations are like **templates**
- ❖ Module instantiation is like creating an **object**
- ❖ Modules are instantiated inside other modules at different levels
- ❖ The top-level module does not require instantiation
- ❖ Module instantiation defines the **structure** of a digital design
- ❖ It produces a **tree of module instances** at different levels
- ❖ The ports of a module instance must match those declared
- ❖ The matching of the ports can be done **by name** or **by position**

Example of Module Instantiation

```
module Full_Adder (input a, b, c, output cout, sum);  
    wire w1, w2, w3;  
  
    // Instantiate two Half Adders: HA1, HA2  
    // The ports are matched by position  
    Half_Adder HA1 (a, b, w1, w2);  
    Half_Adder HA2 (w2, c, w3, sum);  
    or (cout, w1, w3);  
  
    // Can also match the ports by name  
    // Half_Adder HA2  
    // (.a(w2), .b(c), .cout(w3), .sum(sum));  
  
endmodule
```



Number Representation in Verilog

Numbers are represented as:

`<size>'<signed><radix>value` ("`<>`" indicates optional part)

size The number of binary bits the number is comprised of. Not the number of hex or decimal digits. Default is 32 bits.

' A separator, single quote, not a backtick

signed Indicates if the value is signed. Either `s` or `S` can be used. Not case dependent. Default is unsigned.

radix Radix of the number

`'b` or `'B` : binary

`'o` or `'O` : octal

`'h` or `'H` : hex

`'d` or `'D` : decimal

default is decimal

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
```

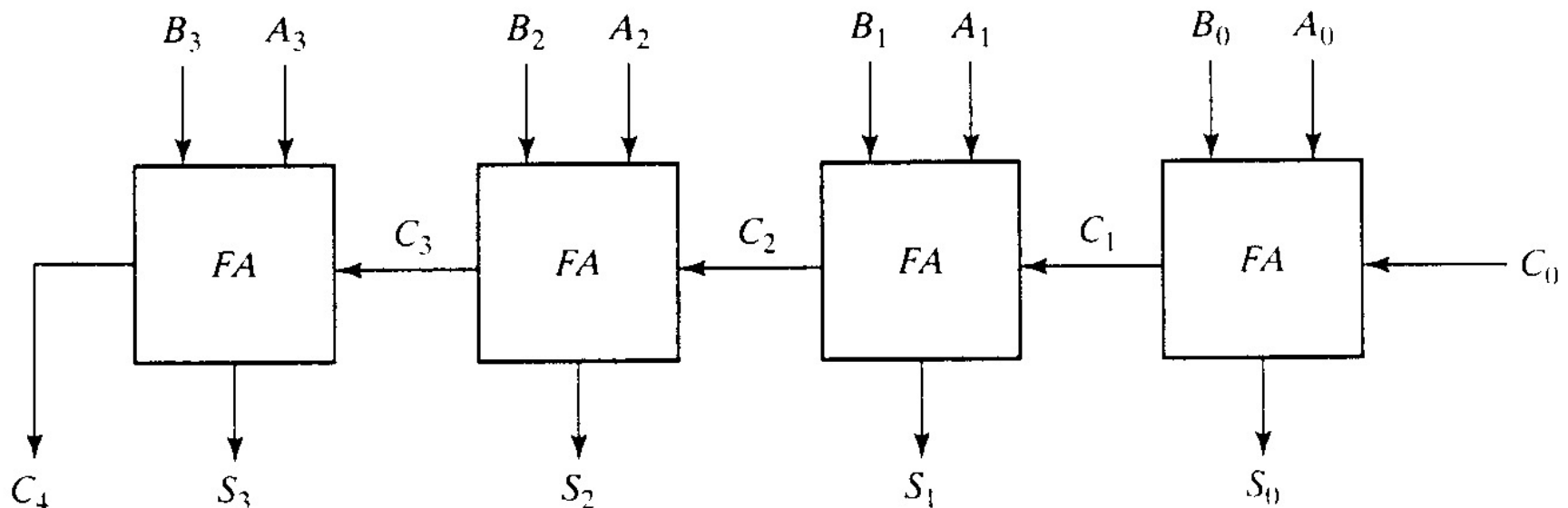
```
output [0:15] B;    // Bit 0 is most-significant bit
```

```
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

4-bit Binary Adder Example

<i>Subscript i:</i>	3	2	1	0	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}



4-bit Binary Adder Example

```
module adder4 (sum, C4, A, B, C0) ;  
    output    [3:0]  sum;  
    output    C4;  
    input     [3:0]  A, B;  
    input     C0;  
    wire      C1, C2, C3;    // Intermediate carries  
    // Instantiate chain of full adders  
    Add_full FA0 (sum[0], C1, A[0], B[0], C0) ;  
    Add_full FA1 (sum[1], C2, A[1], B[1], C1) ;  
    Add_full FA2 (sum[2], C3, A[2], B[2], C2) ;  
    Add_full FA3 (sum[3], C4, A[3], B[3], C3) ;  
endmodule
```

Dataflow Modeling

- ❖ Dataflow modeling provides the means of describing combinational circuits by **their function** rather than by their gate structure.
- ❖ Used mostly for describing Boolean equations and combinational logic
- ❖ Synthesis tool can map a dataflow model into a target technology
- ❖ Can describe: adders, comparators, multiplexers, etc.
- ❖ Dataflow modeling uses a number of operators that act on operands to produce desired results
- ❖ Dataflow modeling uses continuous assignments and the keyword **assign**.

Continuous Assignment

- ❖ The **assign** statement defines continuous assignment
- ❖ Syntax: **assign** *net_name* = *expression*;
- ❖ Assigns *expression* value to *net_name* (wire or output port)
- ❖ 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

Continuous Assignment

❖ Examples:

<code>assign x = a&b c&~d;</code>	<code>// x = ab + cd'</code>
<code>assign y = (a b) & ~c;</code>	<code>// y = (a+b)c'</code>
<code>assign z = ~(a b c);</code>	<code>// z = (a+b+c)'</code>
<code>assign sum = (a^b) ^ c;</code>	<code>// sum = (a ⊕ b) ⊕ c</code>

❖ Verilog uses the bit operators: ~ (not), & (and), | (or), ^ (xor)

❖ Operator precedence: (parentheses), ~ , & , | , ^

Verilog Operators

Bitwise Operators		Arithmetic Operators		Relational Operators	
<code>~a</code>	Bitwise NOT	<code>a + b</code>	ADD	<code>a == b</code>	Equality
<code>a & b</code>	Bitwise AND	<code>a - b</code>	Subtract	<code>a != b</code>	Inequality
<code>a b</code>	Bitwise OR	<code>-a</code>	Negate	<code>a < b</code>	Less than
<code>a ^ b</code>	Bitwise XOR	<code>a * b</code>	Multiply	<code>a > b</code>	Greater than
<code>a ~^ b</code>	Bitwise XNOR	<code>a / b</code>	Divide	<code>a <= b</code>	Less or equal
<code>a ^~ b</code>	Same as <code>~^</code>	<code>a % b</code>	Remainder	<code>a >= b</code>	Greater or equal
Reduction Operators		Shift Operators		Miscellaneous Operators	
<code>&a</code>	AND all bits	<code>a << n</code>	Shift Left	<code>sel?a:b</code>	Conditional
<code> a</code>	OR all bits	<code>a >> n</code>	Shift Right	<code>{a, b}</code>	Concatenate
<code>^a</code>	XOR all bits				
<code>~&a</code>	NAND all bits				
<code>~ a</code>	NOR all bits				
<code>~^a</code>	XNOR all bits				

Reduction operators produce a 1-bit result

Relational operators produce a 1-bit result

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

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

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
```

Modeling a 16-bit Adder

```
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
```

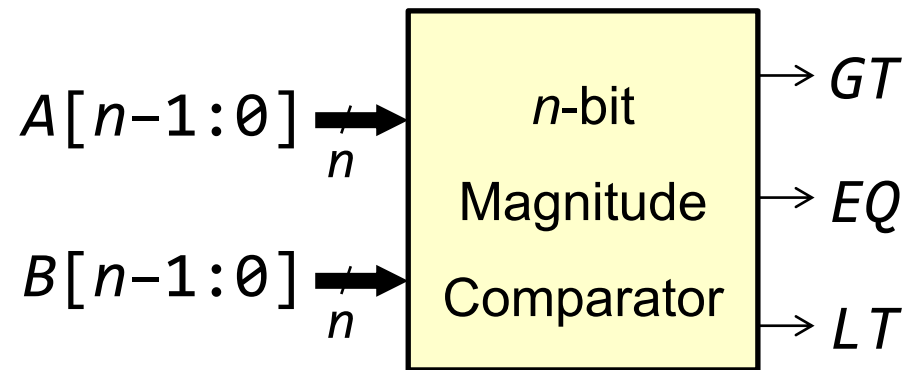
Modeling a Magnitude Comparator

```
// n-bit magnitude comparator, No default value for n
module Comparator (input [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
```



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 2-Input Multiplexer

// Parametric 2-input Mux, default value for n = 1

```
module Mux2( input  [1:0] A, B, input sel,  
            output [1:0] Z);
```

// A and B are n-bit input vectors

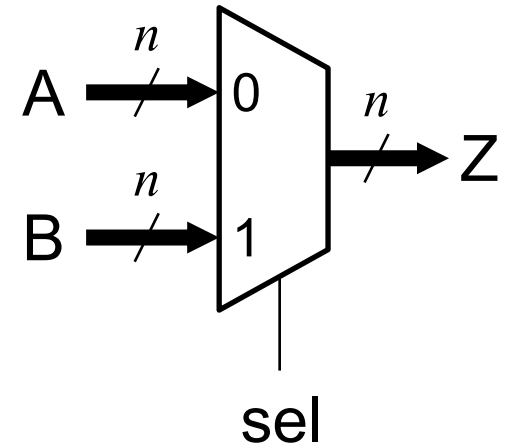
// Z is the n-bit output vector

// if (sel==0) Z = A; else Z = B;

// Conditional operator used for selection

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

```
endmodule
```



Modeling a 4-Input Multiplexer

// Parametric 4-input 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

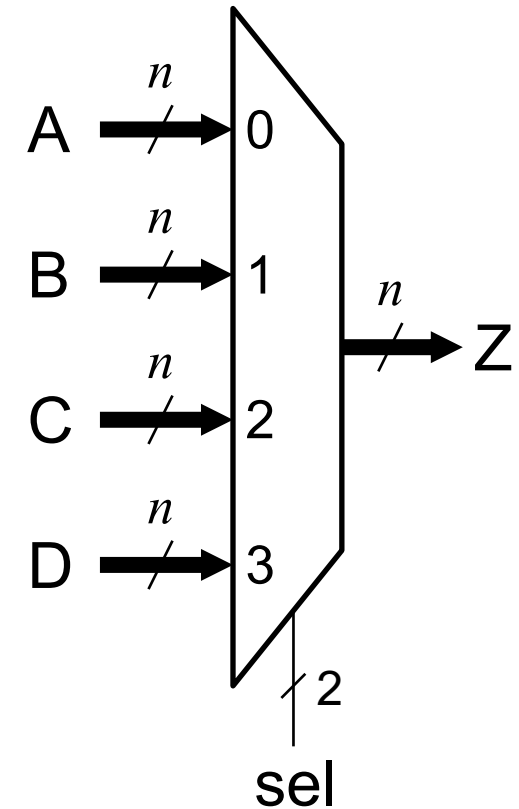
// Nested conditional operators

```
assign Z = (sel == 'b00)? A :
```

```
          (sel == 'b01)? B :
```

```
          (sel == 'b10)? C : D;
```

```
endmodule
```



Dataflow and Behavioral Modeling

- ❖ 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)

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
```

The **initial** Statement

- ❖ The **initial** statement is a procedural block of statements
- ❖ The body of the **initial** statement surrounded by **begin-end** is sequential, like a sequential block in a programming language
- ❖ Procedural assignments are used inside the **initial** block
- ❖ Procedural assignment statements are executed in sequence

Syntax: *variable = expression;*

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

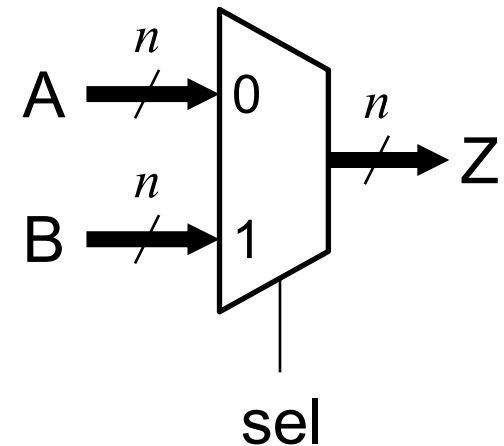
A *compound statement* is surrounded by **begin** ... **end**

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

Modeling a 2-Input Multiplexer

```
// Behavioral Modeling of a Parametric 2-input Mux
module Mux2 ( input [1:0] A, B, input sel,
              output reg [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)
```

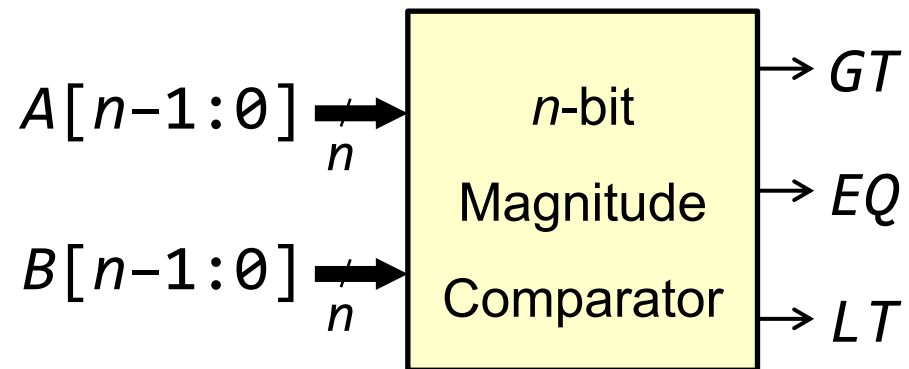
```
      {GT, EQ, LT} = 'b010;
```

```
    else
```

```
      {GT, EQ, LT} = 'b001;
```

```
  end
```

```
endmodule
```



Modeling a 4-Input Multiplexer

// Behavioral Modeling of a 4-input Mux

```
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
```

```
    if (sel == 'b00) Z = A;
```

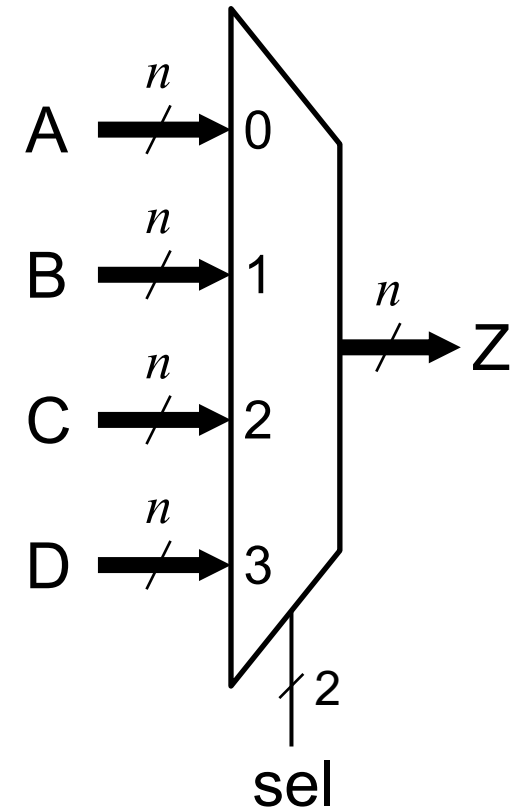
```
    else if (sel == 'b01) Z = B;
```

```
    else if (sel == 'b10) Z = C;
```

```
    else Z = D;
```

```
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( input [1:0] A, B, C, D, input [1:0]  
            sel, output reg [1:0] Z );
```

```
// @(*) is @(A, B, C, D, sel)
```

```
always @(*) begin
```

```
    case (sel)
```

```
        2'b00:    Z = A;
```

```
        2'b01:    Z = B;
```

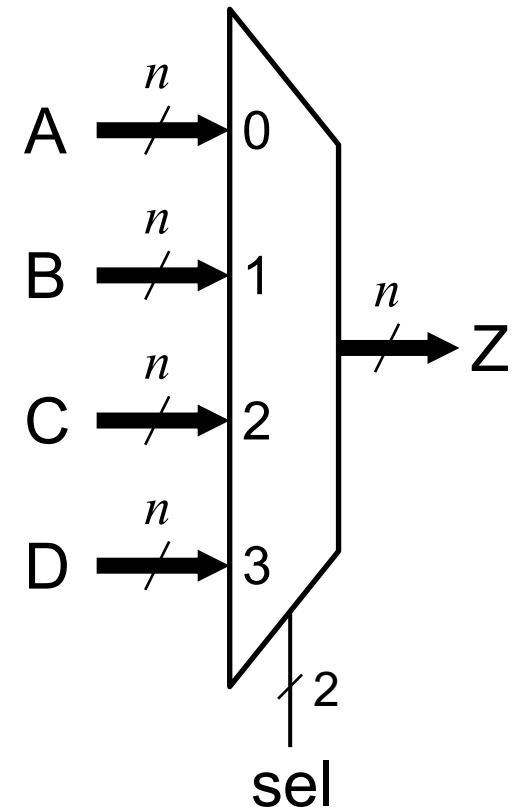
```
        2'b10:    Z = C;
```

```
        default:  Z = D;
```

```
    endcase
```

```
end
```

```
endmodule
```

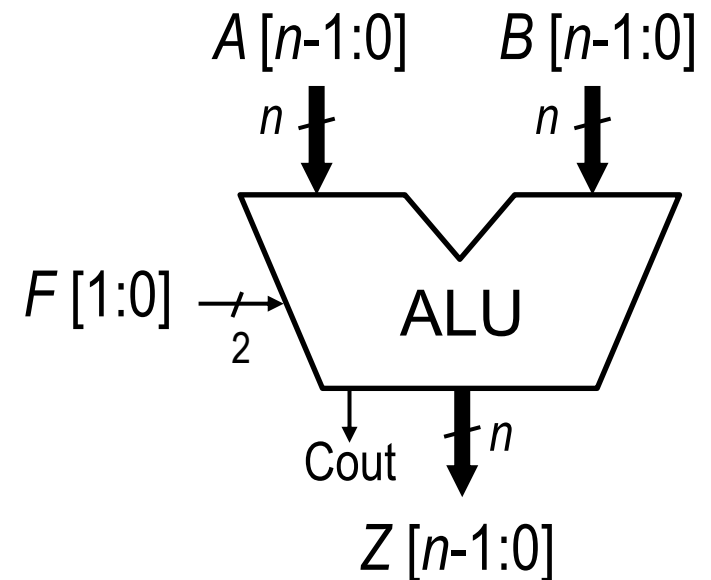


Modeling a Multifunction ALU

// Behavioral Modeling of an ALU

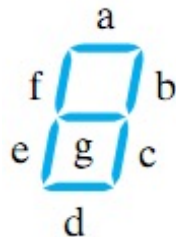
```
module ALU #(parameter n = 16)
  ( input  [n-1:0] A, B, input [1:0] F,
    output reg [n-1:0] Z, output reg Cout );
  // @(*) is @(A, B, F)
  always @(*) begin
    case (F)
      2'b00: {Cout,Z} = A+B;
      2'b01: {Cout,Z} = A-B;
      2'b10: {Cout,Z} = A&B;
      default: {Cout,Z} = A|B;
    endcase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



0 1 2 3 4 5 6 7 8 9