Digital Systems
Section 2

Verilog (Combinational)

Programming: Bridging Software and Hardware

- In the world of programming, tasks can be executed in two ways:
  - **Serial Programming:** Operations are executed one after another, step-by-step, in a sequential order. This is common in **software programming**, where the focus is on algorithms and data manipulation (e.g., Python, C++).
  - **Parallel Programming:** Multiple operations happen simultaneously. This is essential in **hardware programming**, where circuits can handle multiple tasks at the same time (e.g., multiple logic gates working concurrently).

✪ **Hardware programming** languages like **Verilog** and **VHDL** help describe how digital circuits perform these parallel operations efficiently. This shift from sequential software logic to parallel hardware logic is key to understanding HDLs.

- **What is Hardware Programming?**
  - Describes the behavior and structure of digital circuits.

- **Key Languages:**
  - VHDL: Strongly typed, verbose, suited for large, complex systems.
  - **Verilog**: Compact syntax, simpler, often used in commercial applications.

- **Difference from Software Programming:**
  - HDLs describe hardware **behavior**, not algorithms.
  - Parallel operations are at the core, unlike the sequential nature of traditional software.

- **What is Verilog?**
  - A hardware description language used to **model** digital circuits.
  - Popular in both academia and industry for **designing** FPGAs and ASICs.

- **Why Use Verilog?**
  - **Compact** and easy to learn.
  - Allows testing and simulating circuit designs before physical implementation.

- **Key Concepts Covered:**
  - Modules, gates, and behavioral/structural modeling.

Verilog ≡ Verifying Logic

Logic Simulation

- A technique used to **verify** the **behavior** and **functionality** of digital circuits before physical implementation.

- **Purpose:**
  - Identifies **errors early** in the design process.
  - Ensures the circuit works **as expected** under different input conditions.

- **Applications in Hardware Design:**
  - Used extensively with Verilog and VHDL models.
  - **Simulates** gate-level designs, timing constraints, and functional behavior.
  - **Timing Diagrams:** Visual representation of the state of a digital signal over time, illustrating how signals interact within the circuit.
  - **Test Bench:** A simulation environment where input signals are applied to the design under test (DUT) to verify its operation.

- **Tools:**
  - ModelSim, Cadence, **Quartus II**

- **Key Benefit:**
  - Saves time and cost by avoiding rework during fabrication

ENCS 2340

- **Overview:**
  - Logic synthesis is similar to **translating** a program.
  - The output of logic synthesis is a **digital circuit**.
- **Key Concepts:**
  - A digital circuit modeled in Verilog can be translated into a list of components and their interconnections, known as a **netlist**.
  - Synthesis can be used to **fabricate** an integrated circuit.
  - Synthesis can also target a Field Programmable Gate Array (**FPGA**).
- **FPGA Configuration:**
  - An FPGA chip can be **configured** to **implement** a digital circuit.
  - The digital circuit can be **modified** by **reconfiguring** the FPGA.
- **Automation in Design:**
  - Logic simulation and synthesis are **automated** processes.
  - This is accomplished using special software known as Electronic Design Automation (**EDA**) tools.

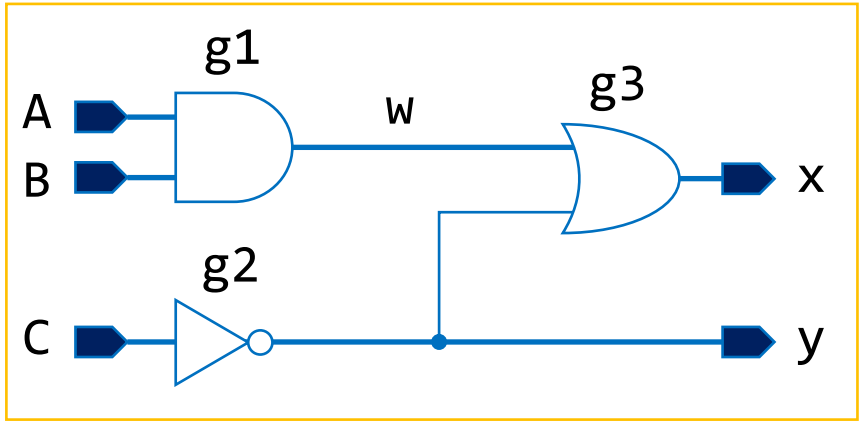
- A **digital circuit** is described in Verilog as a **set of modules**
- A **module** is the **design entity** in Verilog
- A module is declared using the ***module*** keyword
- A module is terminated using the ***endmodule*** keyword
- Each module has a **name** and a list of **input** and **output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

Example:

```
module simple_circuit(input A, B, C, output x, y);
  wire w;
  and g1(w, A, B);
  not g2(y, C);
  or  g3(x, w, y);
endmodule
```

Optional

Order is **NOT** important



- The **input** keyword **defines** the input **ports**: A, B, C
- The **output** keyword **defines** the output **ports**: x, y
- The **wire** keyword **defines** an **internal connection**: w
- The **structure** of simple_circuit is defined by three gates: and, not, or
- Each gate has an **optional name**, followed by the gate **output** then **inputs**

Verilog Syntax

- **Keywords:** have special meaning in Verilog
  - Many keywords: **module**, **input**, **output**, **wire**, **and**, **or**, etc.
  - Keywords **can NOT** 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 Syntax

- 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 (**Unary**)

Examples:

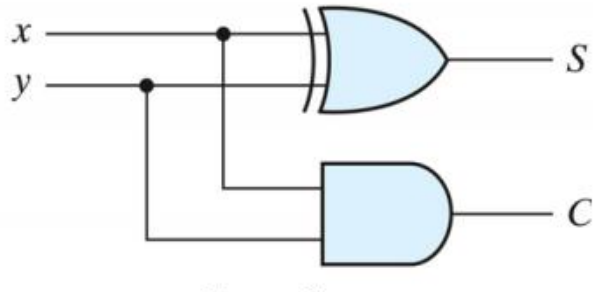
```
and g1(x,a,b);          // 2-input and gate named g1
or  g2(y,a,b,c);        // 3-input or  gate named g2
nor g3(z,a,b,c,d);      // 4-input nor gate named g3
```

Half Adder Modeling


```
module Half_Adder(x, y, C, S);
  input x, y;
  output S, C;
  and (C, x, y);
  xor (S, x, y);
endmodule
```

Alternative way to define **Input/Output**

x	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

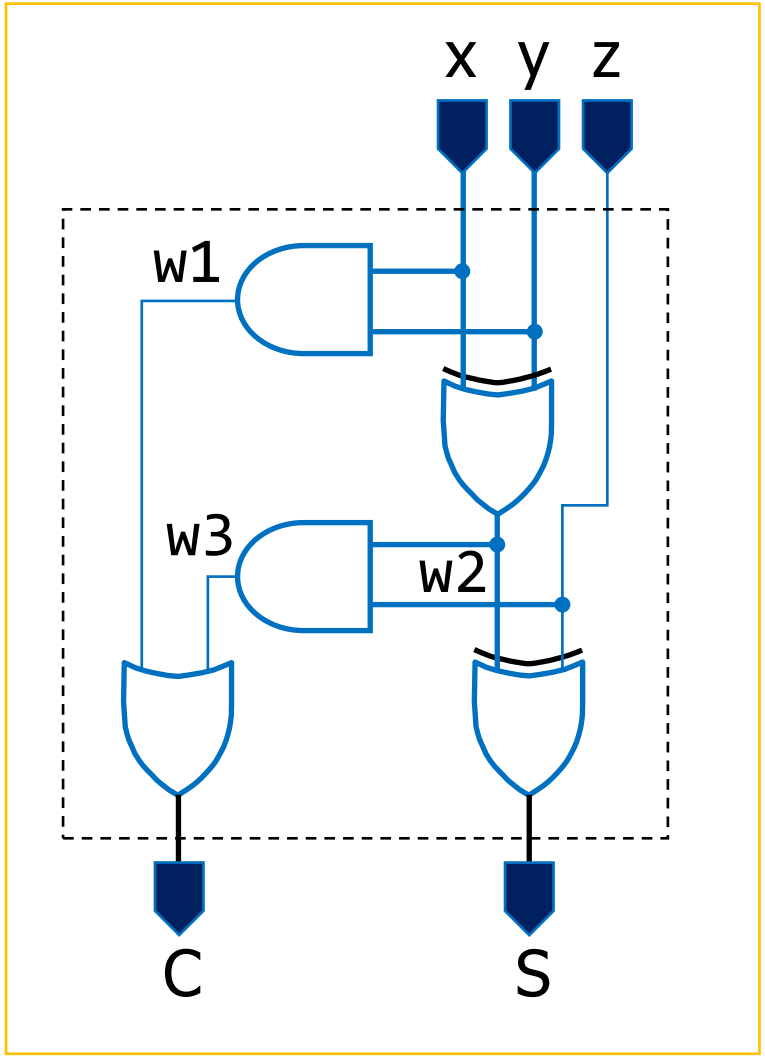


$S = x \oplus y$

$C = xy$

Full Adder Modeling


```
module Full_Adder(input x, y, z, output C, S);
  wire w1, w2, w3;
  and (w1, x, y);
  xor (w2, x, y);
  and (w3, w2, z);
  xor (S, w2, z);
  or  (C, w1, w3);
endmodule
```



Gate Delay

- When simulating Verilog modules, it is sometime necessary to specify the **delay** of gates using the # symbol
- The **timescale** directive specifies the time **unit** and **precision**
- **timescale** is also used as a simulator option

timescale 1ns/100ps

Time unit = 1ns = 10^{-9} sec

Precision = 100ps = 0.1ns

```
module Half_Adder(input a, b, output cout, sum);
  and #2 (cout, a, b);          // gate delay = 2ns
  xor #3 (sum, a, b);           // gate delay = 3ns
endmodule
```

Gate Delay


```
module Full_Adder(input x, y, z, output C, S);
  wire  w1, w2, w3;
  and  #2 (w1, x, y);
  xor  #3 (w2, x, y);
  and  #2 (w3, w2, z);
  xor  #3 (S, w2, z);
  or   #2 (C, w1, w3);
endmodule
```

Continuous Assignment

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

Examples:

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

- Verilog uses the **bit** operators: **~** (not), **&** (and), **|** (or), **^** (xor)
- Operator **precedence**: **(**parentheses**)**, **~** , **&** , **|** , **^**

Continuous Assignment with Delay

Syntax: ***assign #delay name = expression;***

Example:

```
module Full_Adder (input x, y, z, output C, S);
 assign #6 S = (x^y)^z;                    // delay = 6
 assign #7 C = x&y | (x^y)&z;              // delay = 7
endmodule
```

- The **order** of the assign statements does **NOT matter**
- They are **sensitive** to **inputs** (x, y, z) that appear in the expressions
- Any **change** in value of the input ports (x, y, z) will **re-evaluate** the outputs S and C of the assign statements

Verilog Data Types

- 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
  - A wire **cannot store** a value
  - Its value is determined by its driver, such as a gate, a module output, or continuous assignment

- The **reg** is a **variable** data type
  - **Can store** a value from one assignment to the next
  - Used only in procedural blocks, such as the **initial block**

The initial Statement

- The **initial** statement is a **procedural block** of **statements**
- The **body** of the initial statement surrounded by **begin-end** is **sequential**
- **Procedural** assignments are used **inside** the **initial** block
- Procedural assignment statements are **executed in sequence**
- Syntax: #delay **variable = expression;**
- Procedural assignment statements **can be delayed**
- The optional #delay indicates that the variable (of reg type) should be **updated after the time delay**

Test Bench

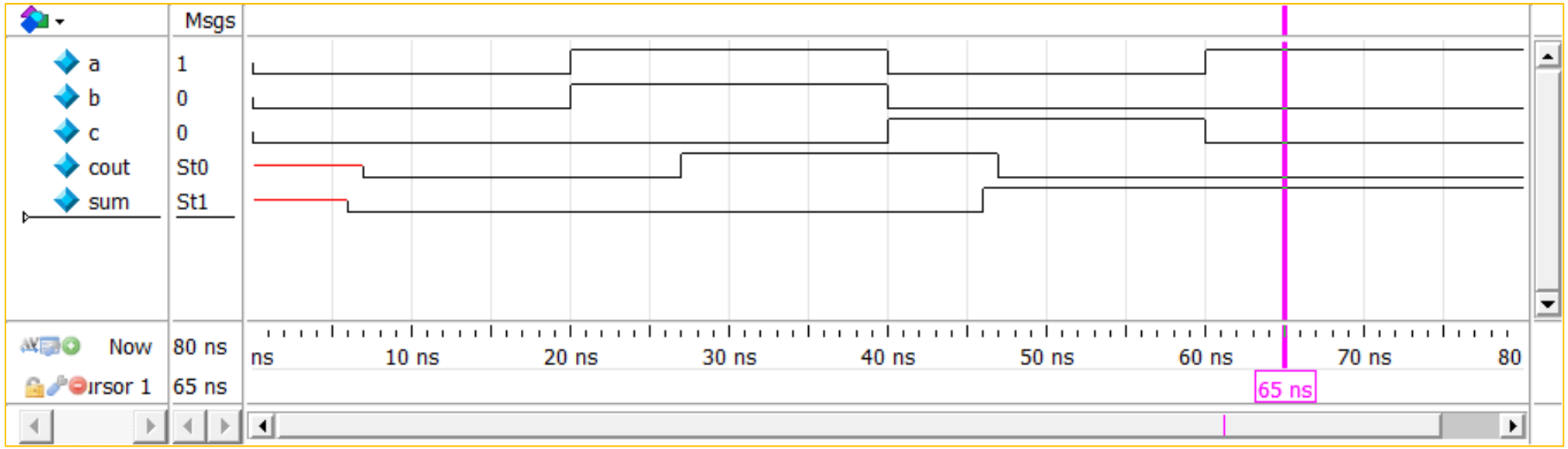
- In order to **simulate** a circuit, it is necessary to **apply inputs** to the circuit for the simulator to **generate** an **output response**
- A **test bench** is written to **verify** the **correctness** of a design
- A **test bench** is written as a Verilog **module with no ports**
- It **instantiates** the module that should be tested
- It **provides inputs** to the module that should be tested
- **Test benches** can be **complex and lengthy**, depending on the complexity of the design

Example:

```
module Test_Full_Adder;                  // No need for Ports
  reg  x, y, z;                          // reg (variable) inputs
  wire S, C;                             // wire (net) outputs

// Instantiate the module to be tested
  Full_Adder FA (x, y, z, C, S);
  initial begin                          // initial block
    x=0; y=0; z=0;                       // at t=0  time units
    #20 x=1; y=1;                        // at t=20 time units
    #20 x=0; y=0; z=1;                   // at t=40 time units
    #20 x=1; z=0;                        // at t=60 time units
    #20 $finish;                         // at t=80 finish simulation
  end                                    // end of initial block

endmodule
```

Change the Inputs
&
Monitor the Effect on Outputs

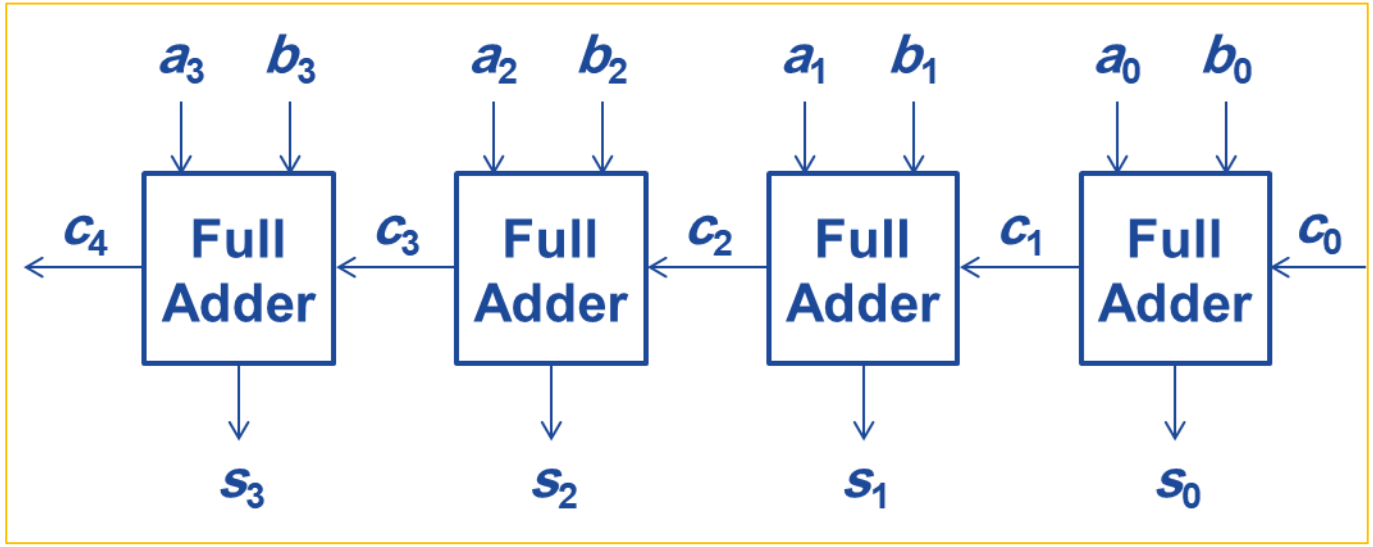
Simulator Run:



- **Examine** the **waveforms** to verify the correctness of your design
- At ***t* = 0** ns, the values of **cout** and **sum** are **unknown** (shown in red)
- The **cout** and **sum** signals are **delayed** by **7ns** and **6ns**, respectively

Modular Design: 4-bit Adder

- Uses **identical copies** of a full adder to **build** a **large adder**
- Simple to implement: the cell (**iterative block**) is a full adder
- **Carry-out** of cell **i** becomes **carry-in** to cell (**i +1**)
- Can be **extended** to add **any number of bits**



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** 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**

4-bit Adder using Module Instantiation


```
module Adder4 (input a0, a1, a2, a3, b0, b1, b2, b3, c0, output s0, s1, s2, s3, c4);
  wire c1, c2, c3;          // Internal wires for the carries
 // Instantiate Four Full Adders: FA0, FA1, FA2, FA3
 // The ports are matched by position
  Full_Adder FA0 (a0, b0, c0, c1, s0);
  Full_Adder FA1 (a1, b1, c1, c2, s1);
  Full_Adder FA2 (a2, b2, c2, c3, s2);
  Full_Adder FA3 (a3, b3, c3, c4, s3);
 // Can also match the ports by name
 // Full Adder FA0 (.a(a0), .b(b0), .c(c0), .cout(c1), .sum(s0));
endmodule
```

Test Bench for the 4-bit Adder


```
module Adder4_TestBench;                    // No Ports
  reg  a0, a1, a2, a3;                      // variable inputs
  reg  b0, b1, b2, b3, cin;                 // variable inputs
  wire s0, s1, s2, s3, cout;                // net outputs
  // Instantiate the module to be tested
  Adder4 Add4 (a0,a1,a2,a3, b0,b1,b2,b3, cin, s0,s1,s2,s3, cout);
  initial begin                             // initial block
    a0=0;a1=0;a2=0;a3=0;                    // at t=0
    b0=0;b1=0;b2=0;b3=0;cin=0;              // at t=0
    #100 a1=1;a3=1;b2=1;b3=1;               // at t=100
    #100 a0=1;a1=0;b1=1;b2=0;               // at t=200
    #100 a2=1;a3=0;cin=1;                   // at t=300
    #100 $finish;                           // at t=400 finish simulation
  end                                       // end of initial block
endmodule
```

Four-Valued Logic

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

Data Types (Revision)

- **Recall**: Verilog has **two** major **data types**:
  - **Net** data types: are **connections** between parts of a design
  - **Variable** data types: can **store** data values
- The **wire** is a **net** data type (**physical** connection)
  - A wire can **NOT** 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, can **NOT** be driven by continuous assignment
- Other **variable** types: **integer, time, real, and realtime**

- **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

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

Example:

```
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
```

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)

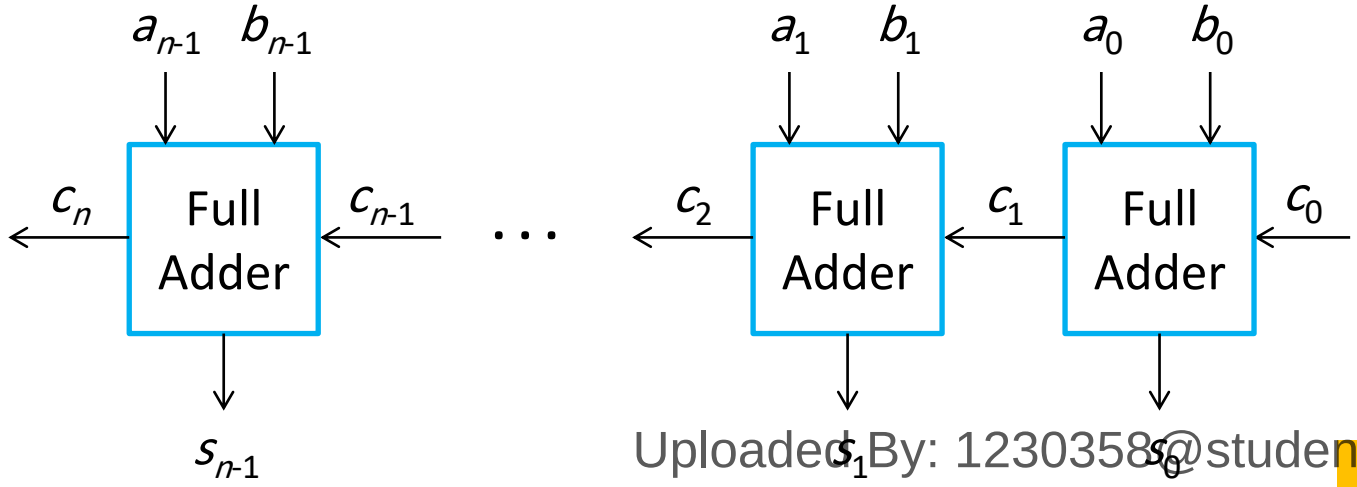
- Syntax: [size]['base] value
  - **size** (**optional**) is the **number** of bits in the value
  - **'base** can be: 'b(binary), 'o(octal), 'd(decimal), or 'h(hex)
  - **value** can be in binary, octal, decimal, or hexadecimal
  - If the **'base** is **NOT** specified then the **default** is **decimal** value

- **Examples:**
  - 8'**b**1011_1101 (8-bit **binary**), '**h**A3F0 (16-bit hexadecimal)
  - 16'**o**56377 (16-bit **octal**), 32'**d**999 (32-bit decimal)
- The underscore _ can be used to enhance **readability** of value
- When **size** is **fewer** bits than **value**, **upper** bits are **truncated**

16-bit Binary Adder


```
// 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 Binary Adder using 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 Binary Adder Using + 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

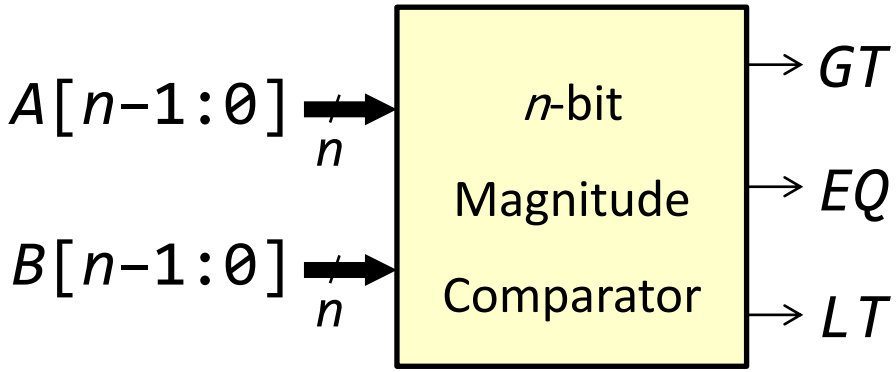

```
// Parametric n-bit adder, default value for n = 16
module Adder #(parameter n = 16)
  ( input [n-1:0] A, B, input cin, output [n-1:0] Sum, output cout );

  // 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

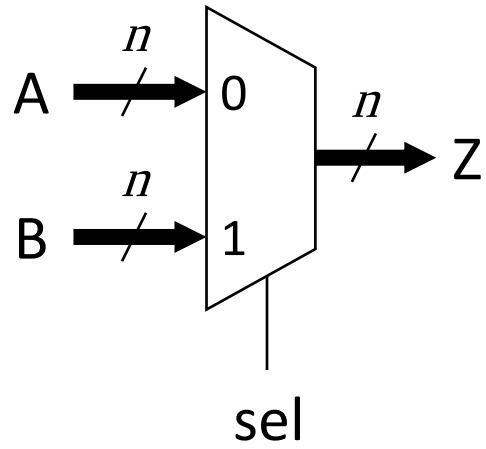
- **Syntax**:
  - Boolean_expr **?** True_expression **:** False_expression
  - **If** Boolean_expr is true then select True_expression
  - **Else** select False_Expression
- Conditional operators can be **nested**

Example:

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

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

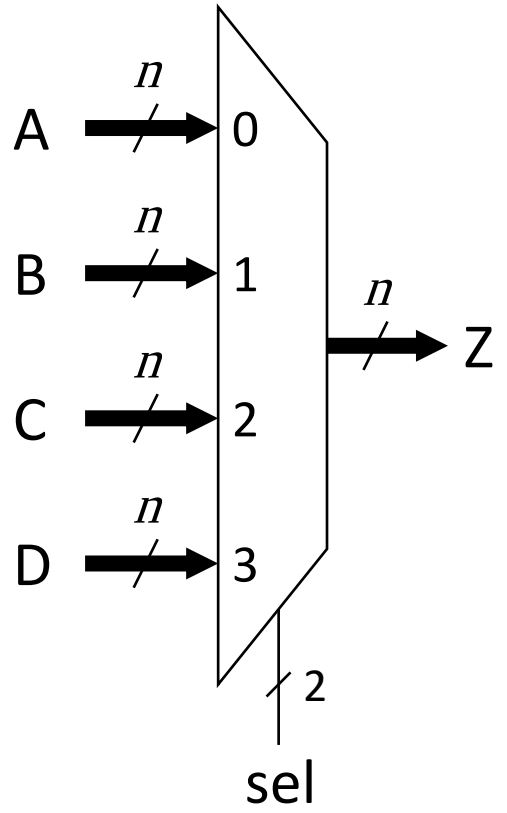
Modeling 2x1 Multiplexer


```
// 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 4x1 Multiplexer


```
// Parametric 2x1 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
  - The **initial** block
    - Executes the enclosed statement(s) **one time only**
  - 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

Behavioral Modeling - Example


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

  @(*) is **automatically** sensitive to 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

If Statement

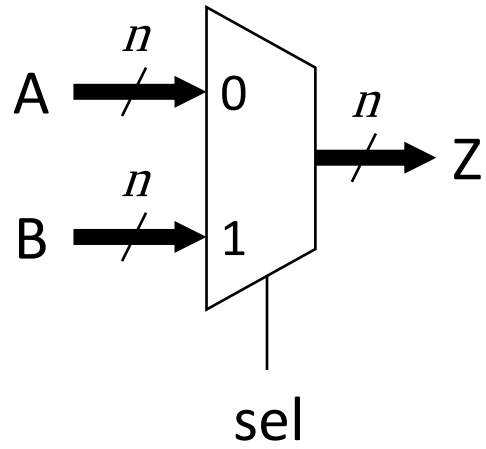
- The **if** statement is **procedural**
  - Can only be used inside a procedural block
- **Syntax:**

  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 2x1 Multiplexer Using If Statement


```
// 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 3x8 Decoder Using If Statement


```
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 4x2 Priority Encoder Using If Statement


```
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
```

Case Statement

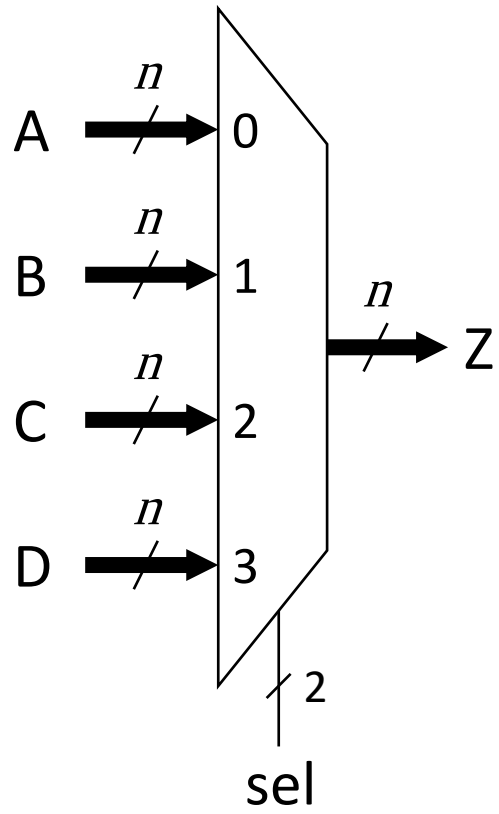
- The **case** statement is procedural (used inside always block)
- **Syntax**:

```
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**

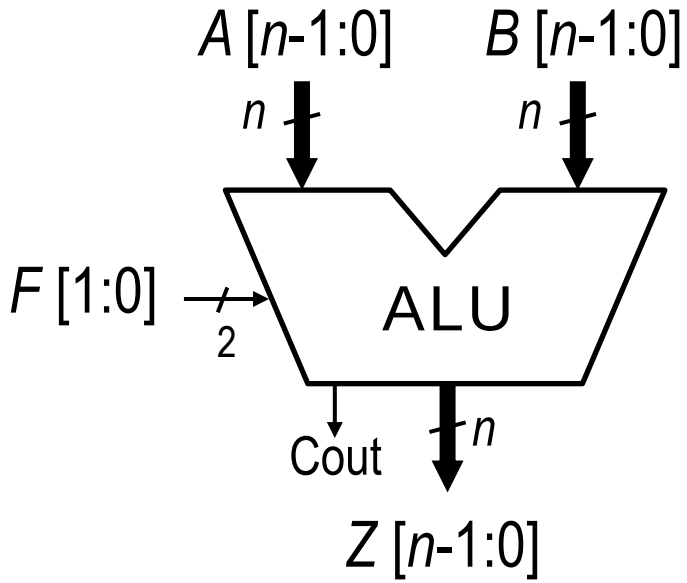
Modeling 4x1 Multiplexer Using Case Statement


```
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
```



```
// 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



```
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
```