EE312 Introduction to Computer Architecture

Introduction to Verilog

KAIST, SChool of EE

Computer Architecture and Memory systems Laboratory

KAIST EE

CAMELab

Outline

- **Lab Class Operating Plan**
- **Verilog HDL: Introduction**
- **Verilog Syntax**
  - Module Body
  - Assignments
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- **Verilog Simulation**
  - Verilog RTL Simulation
  - Simulation using Modelsim

Outline

- **Lab Class Operating Plan**
- Verilog HDL: Introduction
- Verilog Syntax
- Verilog Simulation

Basic Direction

- **Lab Class**
  - Dealing with practical implementation of what you have learned in the class

- **Lab Assignment**
  - Verilog implementation of what you learned in lab class
    - Verilog: A programming language for hardware design

QnA on Class and Assignment

- **Send an e-mail to TAs**
  - Note that we have designated TA for each lab assignment

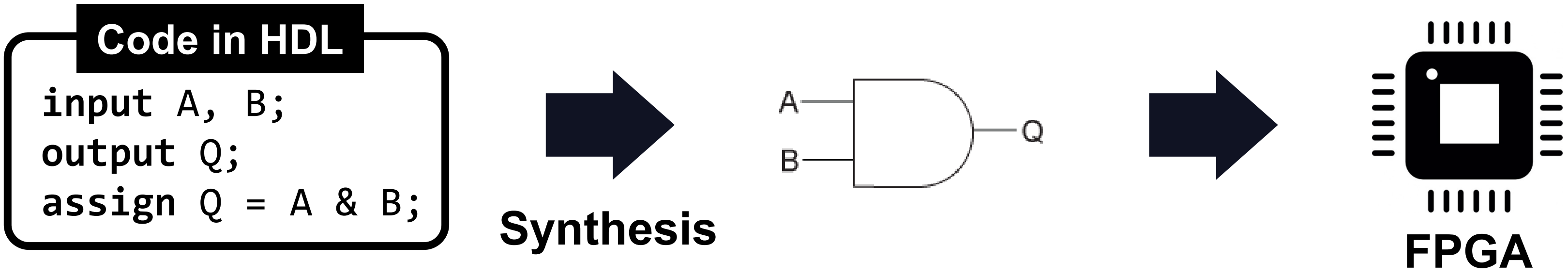
- **Use Office Hours of TAs**
  - Refer to the notice on KLMS

Outline

- Lab Class Operation Plan
- **Verilog HDL: Introduction**
- Verilog Syntax
- Verilog Simulation

Introduction to Verilog

- **What is Verilog HDL?**
  - Hardware Description Language (HDL) for hardware (ASIC or FPGA) design/simulation/verification
  - Similar to C language
  - Providing a range of abstractions
  - RTL: Register Transfer Level
    - Abstract hardware model described in Verilog



Code in HDL

```
input A, B;
output Q;
assign Q = A & B;
```

Example HDLs: Verilog, VHDL..

In this lab, *Verilog* is used!

Differences from Software

- Software is inherently sequential
  - Operation executed in sequential order
- Hardware blocks run in parallel (at the same time)
  - Use event-driven paradigm
  - HDL provide constructs for both parallel & sequential oper.

Software

execute sequential

```
ret1   = x + y;
ret2 = x - y;
```

Hardware

2 gates work at the same time

```
ret1   = a & b;
ret2 = a | b;
```

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
    input       [3:0] A,
    input       [3:0] B,
    output      [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

Starting from here.
(In case of designing
Multi-bit AND chip)

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
    input       [3:0] A,
    input       [3:0] B,
    output      [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

← 1ns: the min. unit of delay
100ps: the min. unit of simulation

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
      input        [3:0] A,
      input        [3:0] B,
      output       [3:0] Q);

      reg [3:0] Q;

      always begin
            Q = A & B;
      end
endmodule
```

← Module start

Verilog Example

- **Simple AND Module**

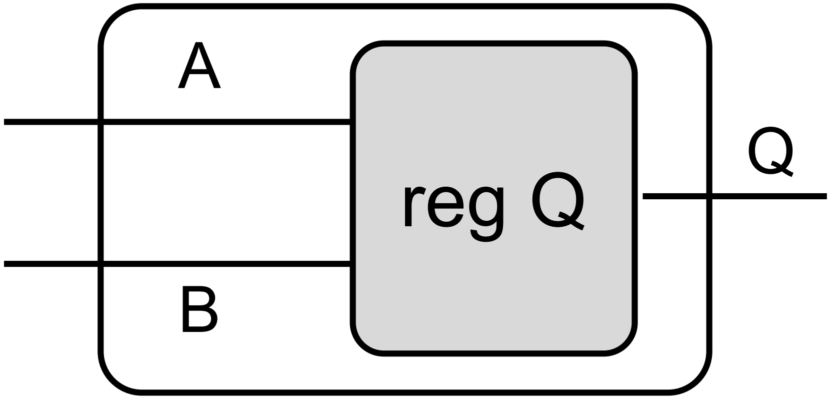
```
`timescale 1ns / 100ps

module AND (
    input       [3:0] A,
    input       [3:0] B,
    output      [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

← I/O ***ports*** declaration

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
    input       [3:0] A,
    input       [3:0] B,
    output      [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```



← Declaration of Output Q as a register

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
    input        [3:0] A,
    input        [3:0] B,
    output       [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

← If there are any change of values in a block (from ***begin*** to ***end***, like { } in C language), the statement Q = A&B is always executed.

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
    input        [3:0] A,
    input        [3:0] B,
    output       [3:0] Q);

    reg [3:0] Q;

    always begin
         Q = A & B;
    end
endmodule
```

← Assign A&B in the register Q

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (
    input       [3:0] A,
    input       [3:0] B,
    output      [3:0] Q);

    reg [3:0] Q;

    always begin
        Q = A & B;
    end
endmodule
```

← 'AND' Module ends here.

Verilog Example

- **Simple AND Module**

```
`timescale 1ns / 100ps

module AND (A, B, Q);
    input       [3:0] A;
    input       [3:0] B;
    output reg  [3:0] Q;



    always begin
        Q = A & B;
    end
endmodule
```

← Looks different, but functionally equivalent to the codes shown in previous slide

Timescale

`timescale 1ns / 100ps

- **Declaration of Time Units for Simulation**
  - Generally inserted at the beginning of the code
    - `timescale <Delay Unit> / <Simulation Unit>

- **Delay Unit: The min. time unit of Delay cmd.**

```
#100;      // Delay for 100 * 1 ns
```

- **Simulation Unit: The min. time unit of Sim.**
  - The timing resolution of simulation

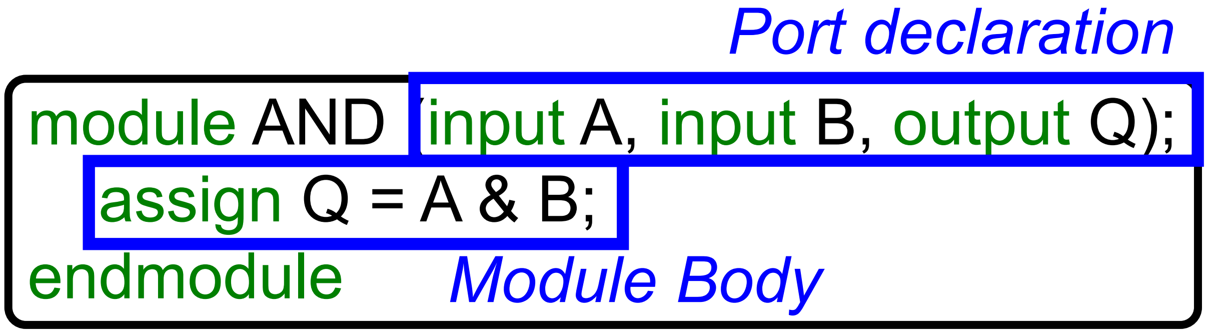
Outline

- Lab Class Operating Plan
- Verilog HDL: Introduction
- **Verilog Syntax**
  - Module Body
  - Assignments
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- Verilog Simulation

Module Concept in Verilog

- **Building Blocks for Verilog Hardware Design**
  - A chip itself or a component of a chip
  - Similar to “class” in programming languages

- **Interface + Behavior**
  - Interface: Input and Output ***(Ports)***
  - Behavior: How does the module work?
  - How are the outputs generated w.r.t. the input?



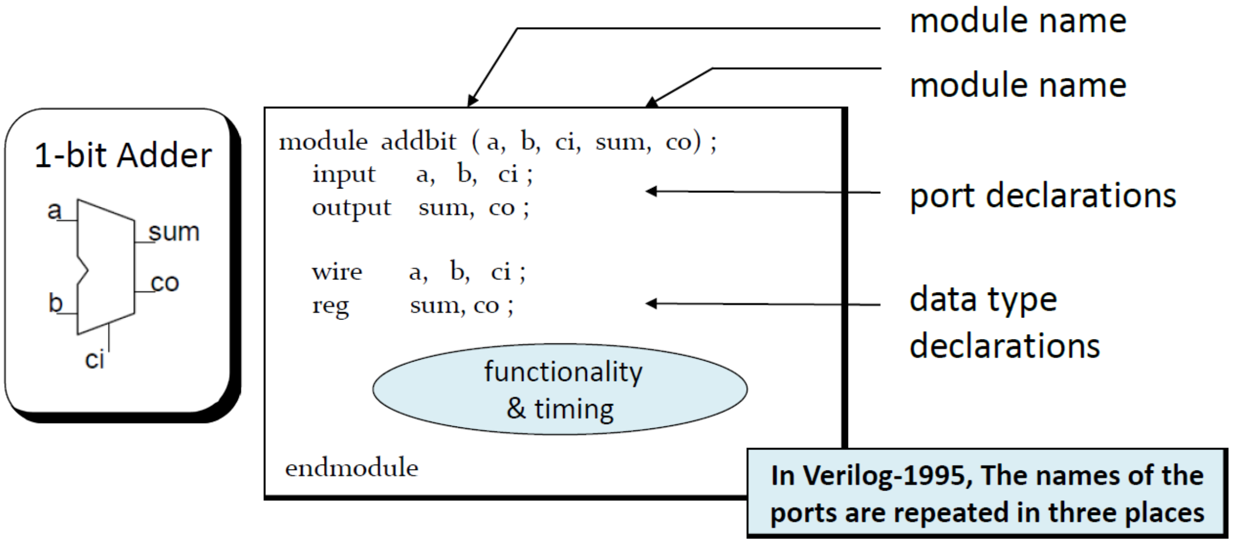
Port declaration

```
module AND (input A, input B, output Q);
   assign Q = A & B;
endmodule
```

Module Body

RTL Module Structure

- **Module Declaration**



```
module  addbit  ( a,  b,  ci,  sum,  co) ;
   input     a,   b,   ci ;
   output   sum,  co ;

   wire      a,   b,   ci ;
   reg       sum, co ;

endmodule
```

In Verilog-1995, The names of the ports are repeated in three places

- Module name: a user-defined name for the model
- Module ports: signals coming in and going out
  - Port declaration: direction, bit width, wire/reg
- Data type: wire vs reg

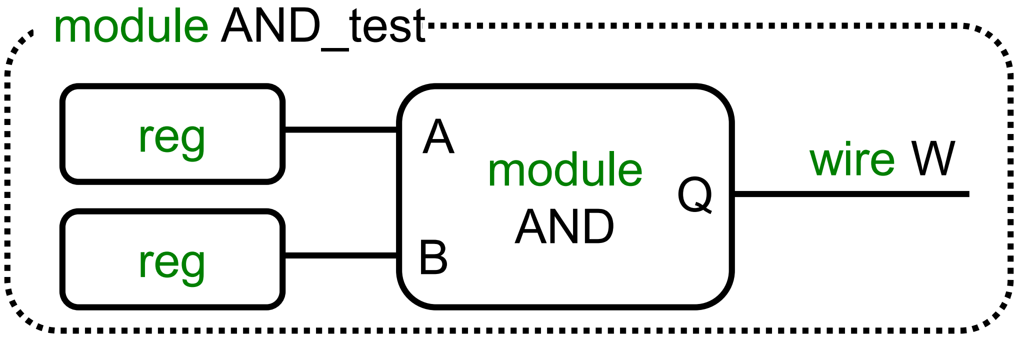
Module Instantiation

- **Instantiation**: **Similar to a class call**

```
module AND (input A, input B, output Q);
    assign Q = A & B;
endmodule
```

```
module AND_test ();
    reg X, Y;   wire W;
    AND aaa (X, Y, W);
endmodule
```

Instantiation (Positional mapping)



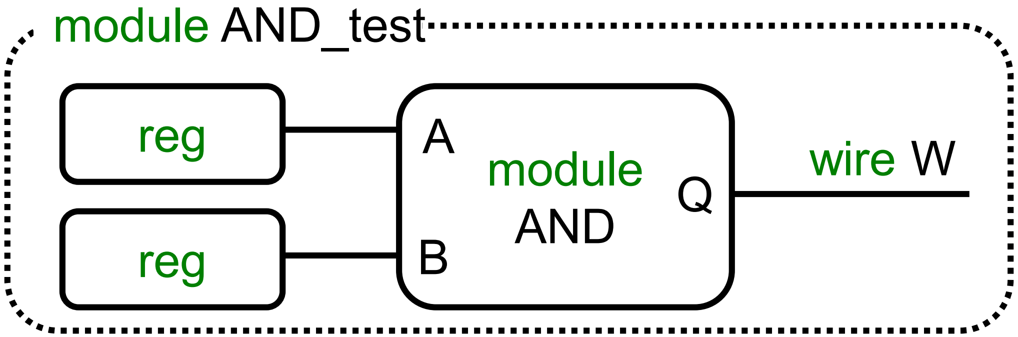
Module Instantiation

- **Another way of instantiating modules**

```
module AND (input A, input B, output Q);
    assign Q = A & B;
endmodule
```

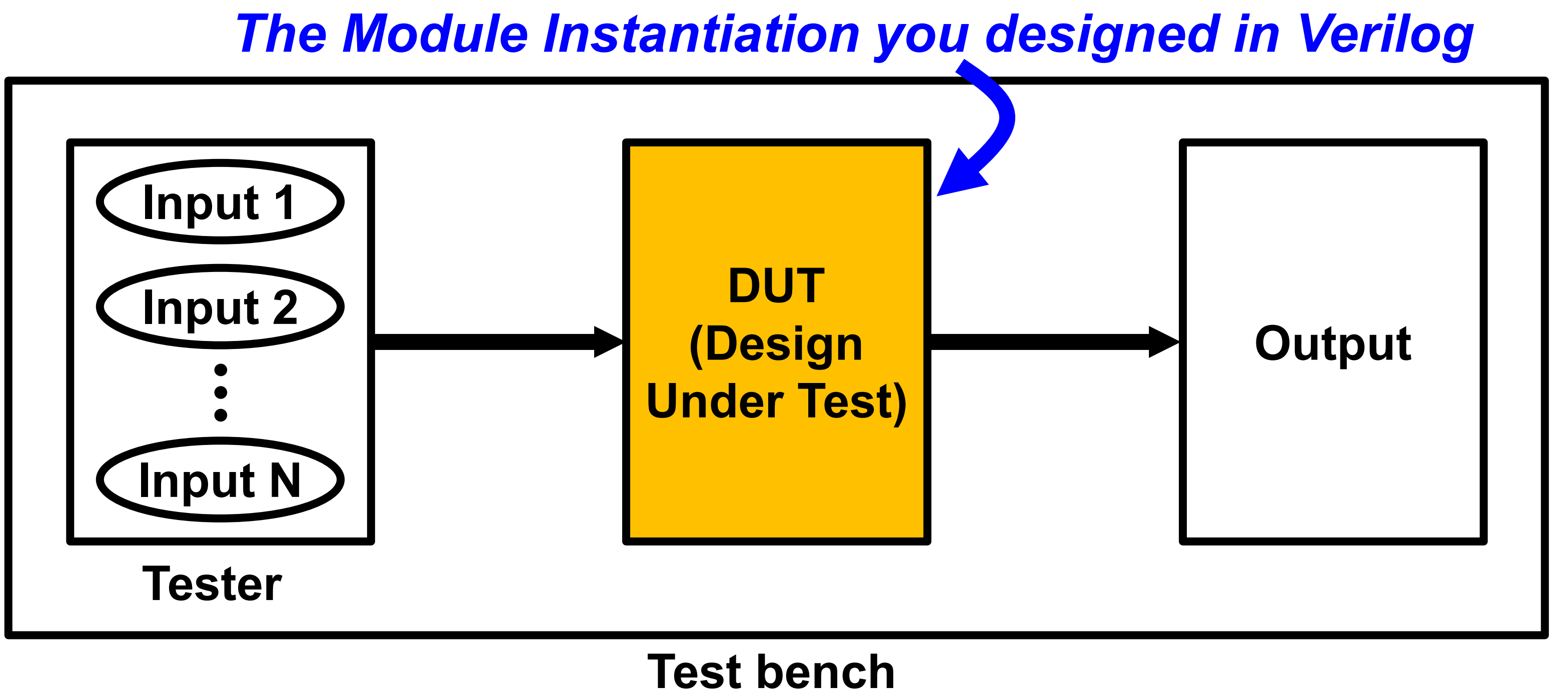
```
module AND_test ();
    reg X, Y;   wire W;
    AND aaa (.A(X), .B(Y), .Q(W));
endmodule
```

Instantiation (Named mapping)



Testbench for Test Designed Module

- **Testbench**
  - Test signal generation and output data recording with instantiation of the designed module
  - Test signal (stimulus) generation as input to DUT



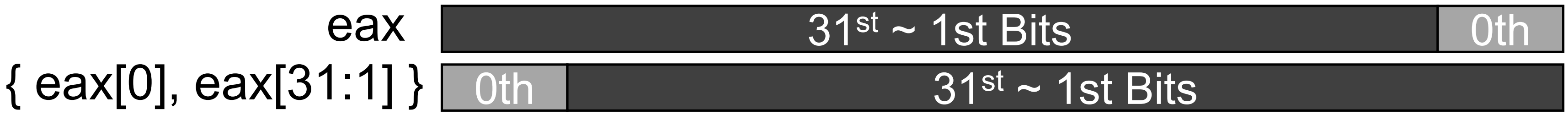
Port Declaration

- **I/O of Module**
  - Basically, regarded as ***wire*** *(input & output)*
  - Optionally, can be also be declared as ***reg*** *(output)*
  - ***wire-wire*** *or* ***reg-wire*** *connection* ➔ *possible*
  - ***reg-reg*** *connection* ➔ *impossible*

```
module always_one (output Q);
  reg Q;      Declaring output Q as reg
  initial Q = 1;
endmodule
```

Data Type

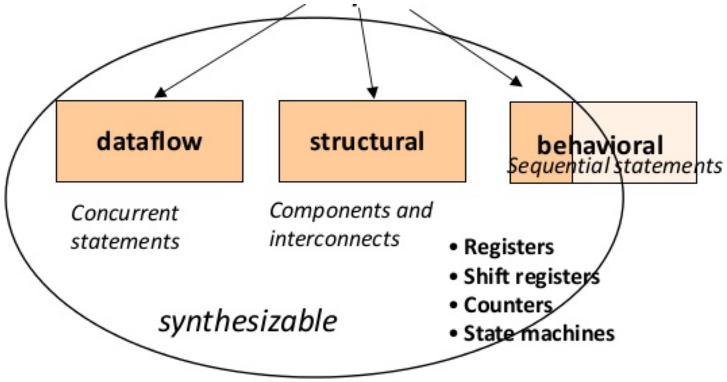
- **wire**
  - Literally a piece of wire as physical connection
- **reg**
  - 1-bit D-flip flop (1-bit register), data storage element
- **Bit Vector**
  - ex) wire [0:31] addr; // Big-endian
  - ex) reg [31:0] eax; // Little-endian (recommended)
- **Reference & Concatenation**
  - ex) addr // Entire addr bit vector
    addr[0] // 0th bit of addr
    addr[3:0] // 3rd ~ 0th bits of addr
    {eax[0], eax[31:1]} // eax's 0th bit + eax 31st ~ 1st bits



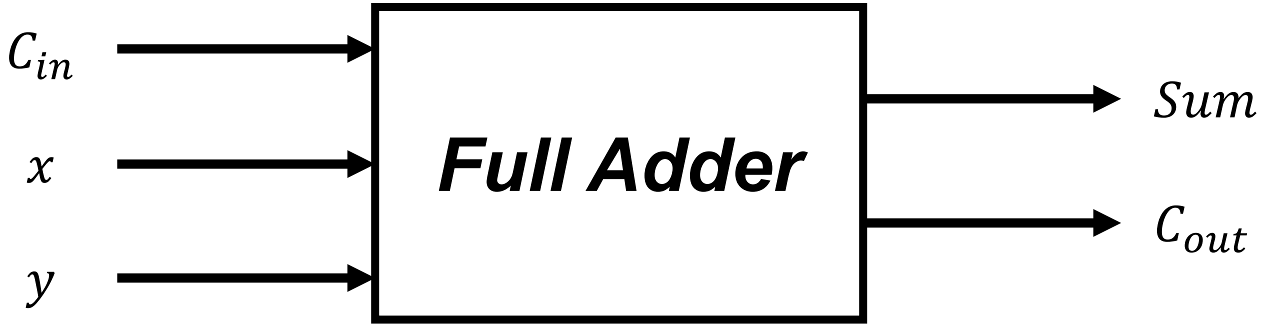
Verilog Modeling Concepts

- **Structural (Gate-level) Modeling**
  - Description using logic primitives
  - *and, or, nand, not* ... and so on
- **Dataflow Modeling**
  - Only for combinational functions
  - Using statement ***assign***
- **Behavioral Modeling**
  - Providing high-level abstract description like C
  - Using statement ***always***
  - Beware of "synthesizability"
    - ➔ Not all behavioral models are synthesizable
- **RTL: Usally, Mix of Dataflow & Behavioral**

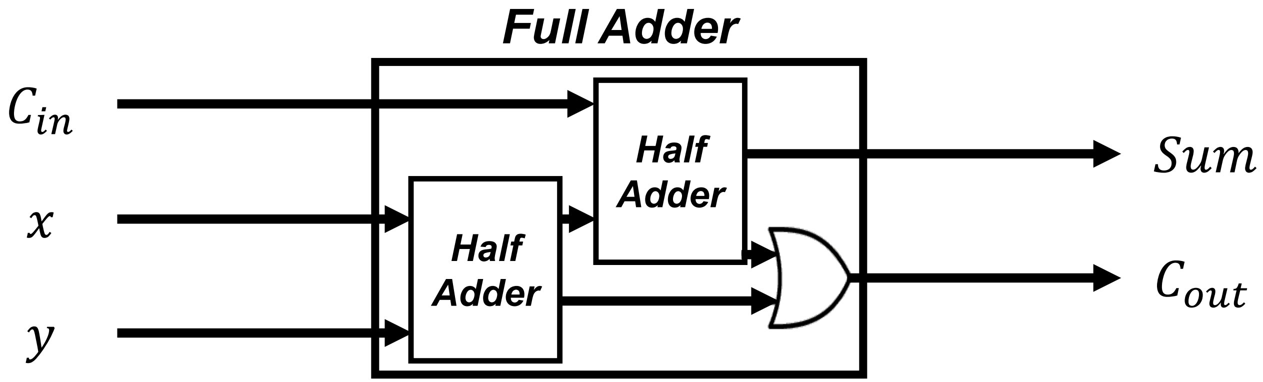
Verilog Design Styles



Modeling Example



Structural Modeling Example

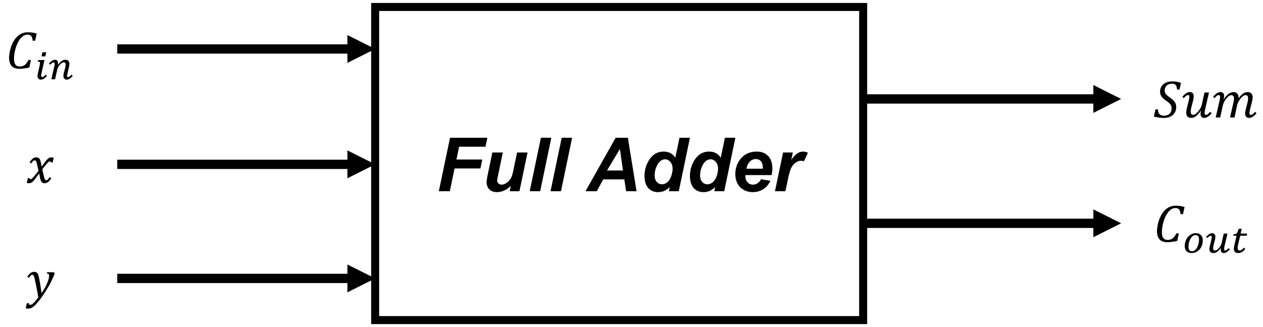


```
module Half_adder (x, y, sum, c_out);
    input       x, y;
    output      sum, c_out;

    xor (s,x,y);
    and (c,x,y);
endmodule
```

→

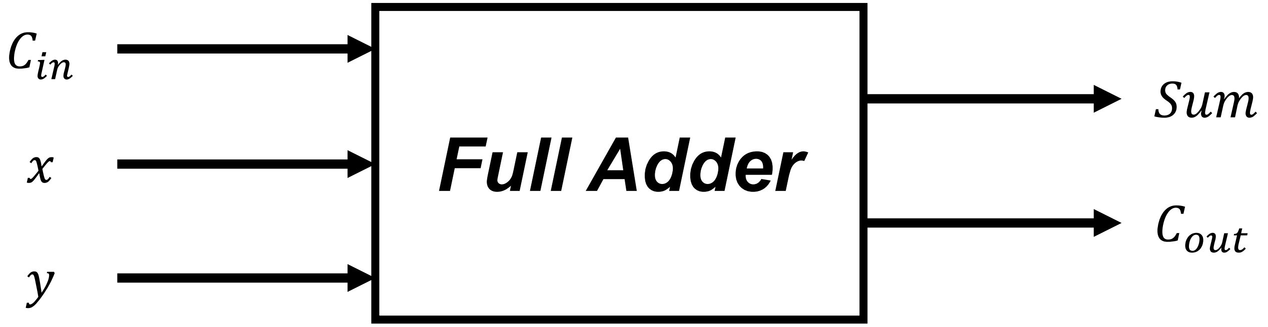
```
module Full_adder (x, y, c_in, sum, c_out);
    input       x, y, c_in;
    output      sum, c_out;
    wire        c1, s1, c2;

    Half_adder ha_1 (x, y, c1, s1);
    Half_adder ha_2 (c_in, s1, c2, sum);
    or(c_out, c1, c2);
endmodule
```

Dataflow Modeling Example



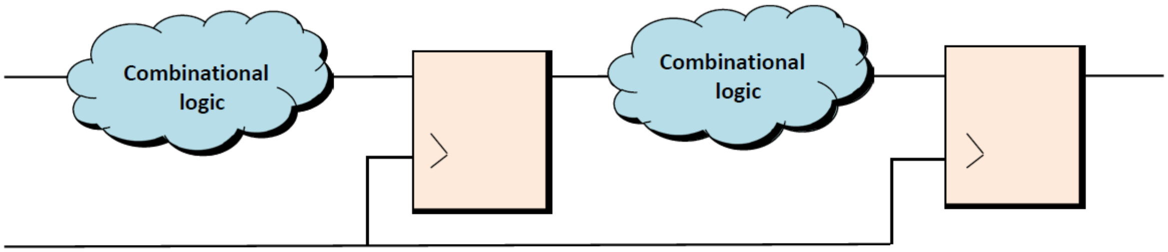
```
module Full_adder (x, y, c_in, sum, c_out);
   input       x, y, c_in;
   output      sum, c_out;

   assign {c_out, sum} = x + y + c_in;
endmodule
```

Behavioral Modeling Example



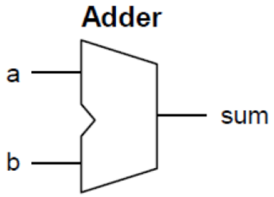
```
module Full_adder (x, y, c_in, sum, c_out);
   input      x, y, c_in;
   output     sum, c_out;
   reg        sum, c_out; // need to be declared as reg type

   always @ (x, y, c_in) begin // also @(*) or @(x or y or c_in)
      {c_out, sum} = x + y + c_in;
   end
endmodule
```

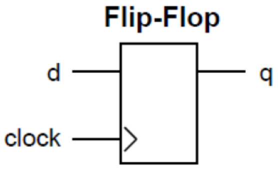
Sequential & Combinational Logic



Guidelines:
- All module outputs should be registered
- Only use one clock per module

- **Combinational Logic**
  - ***always @ (*)*** and blocking assignment (=)
  - The output as a direct reflection of the input values
  - Unable to use latching (registers)
- **Sequential Logic**
  - ***always @ (posedge clk or negedge rst_n)***
  - Non-blocking assignment (<=)
  - Latching: synchronized with clock ➔ data storing





Procedural Statements

- **Procedural Statements as Code Block**

```
module AND (input A, input B, output Q);
    reg Q;   wire AnB;
    assign AnB = A & B;

    always begin
        Q = AnB;
    end
endmodule
```

Procedural Statement

- Can be regarded as a code block like { } in C
- ***always*** and ***initial***

Initial Procedure

- **Initial Procedure at Testbench Initialization**

```
initial begin
   clk = 0;
   x = 1;
end
```

=

```
initial clk = 0;
initial x = 1;
```

- Executed just once when module is instantiated
- Several statements can be contained b/w ***begin & end***
- Usually used when writing testbench to generate stimulus signals or to set initial values

Always Procedure

- **Always Procedure at Testbench Clock Gen.**

```
reg clk;
initial clk = 0;
always begin
   #50;
   clk = ~clk;
end
```

```
reg clk;
initial clk = 0;
always @(posedge clk) begin
   #50;
   a = ~a;
end
```

  - Always called when certain condition is satisfied
  - Infinitely called when there is no condition
  - Used in both module design and testbench
  - Usually used when writing testbench to generate clock signals

Outline

- Lab Class Operating Plan
- Verilog HDL: Introduction
- **Verilog Syntax**
  - Module Body
  - **Assignments**
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- Verilog Simulation

Assignments

- **Continuous and Procedural Assignments**

```
module AND (input A, input B, output Q);
   reg Q;   wire AnB;
   assign AnB = A & B;

   always begin
      Q = AnB;
   end
endmodule
```

Continuous Assignments

Procedural Assignments

- Continuous: defining **wire** connections
- Procedural: allocating values at **registers**

Continuous Assignments

- **Continuous Assignment for wire (assign)**
  - Allocating connections b/w wires outside of procedure

```
wire a, b, c;
assign c = a&b;  // wire c is defined as a&b
```

  - Can be used with declaration simultaneously

```
wire a, b;
wire c = a&b;
```

  - Not changed in the middle of simulation
  - Used with delay for wire delay modeling

```
wire a, b, c;
assign #10 c = a&b;
     // c is update to a&b after 10 delay unit
```

Procedural Assignments

- **Prodedural Assignment for Register (<=, =)**

```
reg clk;
initial begin
    clk = 0;
end
```

  - Storing values in reg
  - Use with delay for delay modeling
    - ex) #10 clk = 0;   *//assign 0 at clk after 10 delay*
      #10 clk = #20 tmp;   *// read tmp after 20 delay*
                                           // then store it in clk after 10 delay
  - **Blocking (=) & Non-blocking (<=)**

Types of Procedural Assignments

- **Blocking Assignments**

```
reg a, b;
initial begin
    a = 0;
    b = 1;
end
```

In the both codes,
a=0 is executed first,
and then b=1 is executed

```
reg a, b;
initial begin
    a = 0;
end
Initial begin
    b = 1;
end
```

- Executed sequentially line by line (like C)
- Use “**=**“ operator
- The order b/w procedures is also valid

Types of Procedural Assignments

- **Non-Blocking Assignments**

```
reg a, b;
initial begin
    a <= 0;
    b <= 1;
end
```

In the both codes,
a<=0 and b<=1
are executed
simultaneously

```
reg a, b;
initial begin
    a <= 0;
    b <= 1;
end
```

  - Executed concurrently in parallel
  - Use “**<=**“ operator
  - Does not matter with order

DO NOT USE blocking (=) and non-blocking (<=) TOGETHER in the same procedure!

Blocking vs Non-blocking

- **Data Flow w.r.t Time (Initialized as a=0, b=1)**

Blocking Assn.

```
reg a, b, c;
initial begin
    #50;
    b = a;
    c = b;
end
```

Cycle	a	b	c
0	0	1	
50	0	0	0

Non-blocking Assn.

```
reg a, b, c;
initial begin
    #50;
    b <= a;
    c <= b;
end
```

Cycle	a	b	c
0	0	1	
50	0	0	1

Outline

- **Lab Class Operating Plan**
- **Verilog HDL: Introduction**
- **Verilog Syntax**
  - Module Body
  - Assignments
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- **Verilog Simulation**

Delay

- **Purpose of Delay Assignment**
  - To Imposing delay during simulation
    - clock signal generation, stimulus signal input
  - Realistic hardware delay modeling
    - wire RC timing delay, gate delay
  - In this course, use delay only at testbench for sim.
- **Accumulation of delay**
  - Delays are accumulated in a procedure

```
initial begin
   #50 clk = 0;
   #50 x = 1;
end
```

=

```
initial #50 clk = 0;
initial #100 x = 1;
```

Event

- **Purpose of Event Assignment**

```
module dummy (input a,
        output b);
    reg b;
    event e;
    initial begin
        #50;
        -> e;
    end
    always @e begin
        b <= 1;
    end
endmodule
```

After 50 delay units, e is executed

Every time the event e happens procedure is always executed

- Assigning executing condition to peocedures

Cases of Event

- **How to Assign Events**
  - Data change as event

```
reg a;
always @ (a) begin
    ...
end
```

  - The moment of value change as event

```
reg a;
always @ (posedge a) begin // when a 0→1
always @ (negedge a) begin // when a 1→0
```

  - Logically combining events

```
always @ (posedge clk or negedge rst_n)
      // very typical structure in sequential logic
```

Outline

- Lab Class Operating Plan
- Verilog HDL: Introduction
- **Verilog Syntax**
  - Module Body
  - Assignments
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- Verilog Simulation

Flow Control Statements: If

- **If Statement**
  - Conditional execution of statement
  - Trinomial operator in C

```
initial begin
    if (a == 5) begin
        b <= 15;
    end
    else begin
        b <= 25;
    end
end
```

=

```
wire c = (a == 5) ? 15 : 25;
initial begin
    b <= c;
end
```

Flow Control Statements: Case

- **Case Statement**
  - Multi-conditional execution of statement

```
initial begin
    if (a == 5) begin
        b <= 15;
    end
    else begin
        b <= 25;
    end
end
```

=

```
initial begin
    case (a)
        5 : b <= 15;
        default : b <= 25;
    endcase
end
```

Flow Control Statements: Loop

- **Loop Statement: Usally used in testbench**
  - Repeat: execute statement for finite times

```
initial begin
    repeat (4) c = c + 1;
end
```

=

```
initial begin
    c = c + 1;
    c = c + 1;
    c = c + 1;
    c = c + 1;
end
```

```
initial begin
    forever #5 c = c + 1;
end
```

=

```
initial begin
    #5 c = c + 1;
    #5 c = c + 1;
    #5 c = c + 1;
    #5 c = c + 1;
End// inifinitely repeat
```

Flow Control Statements: Loop

- **Conditional Loop: while, for**
  - Used like in C
  - Generally not synthesizable
  - Usually only used in testbench

```
reg c;
initial begin
    while (c < 10)
        c = c + 1;
end
```

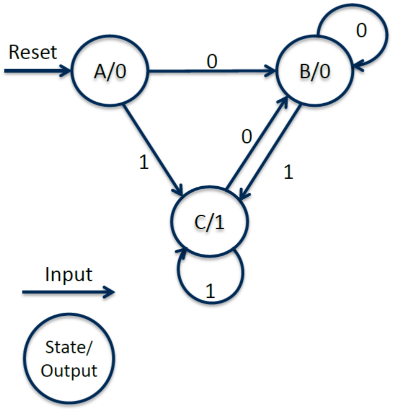
```
reg c;
initial begin
    for (c = 0; c < 10; c = c + 1)
        // Statement
end
```

Outline

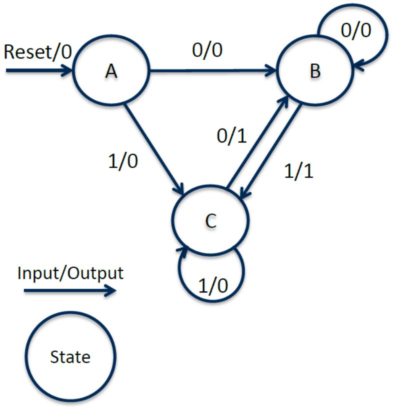
- Lab Class Operating Plan
- Verilog HDL: Introduction
- **Verilog Syntax**
  - Module Body
  - Assignments
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- Verilog Simulation

Concept of FSM

- **Finite State Machine**
  - Deterministic Finite Automata
  - A machine with only deterministic finite states
  - Mealy machine vs Moore machine

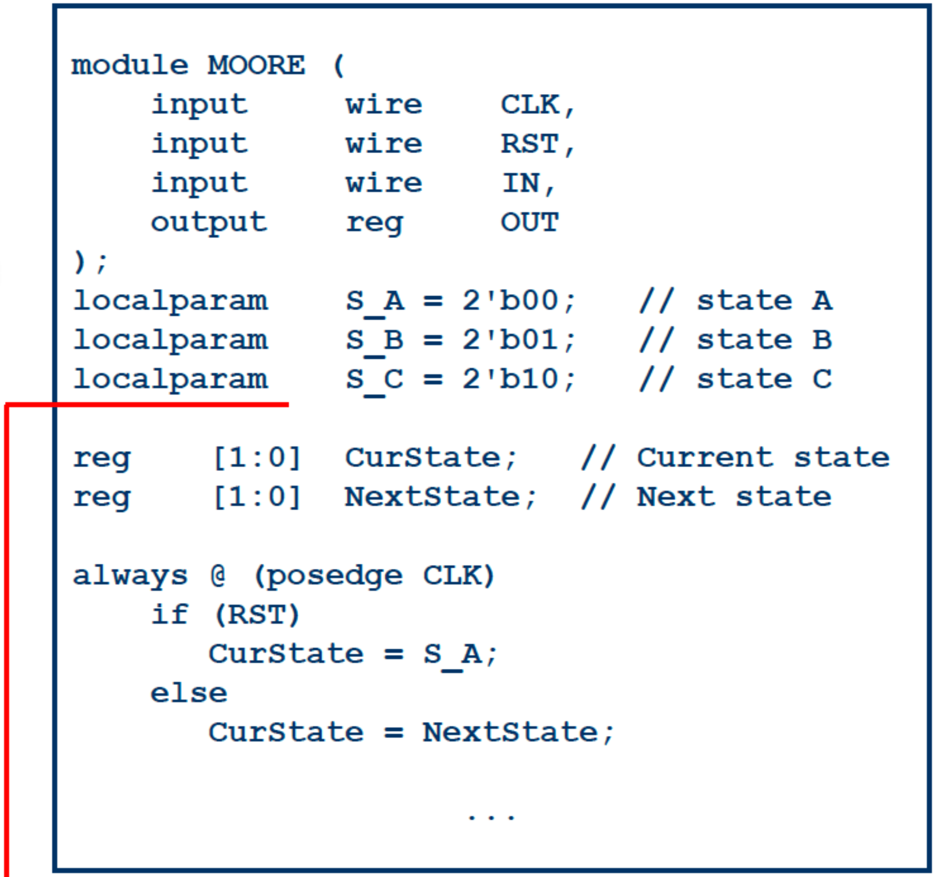


- **Moore Machine**
  - Output is determined
    - by current state
  - Next state is determined
    - by current state & input



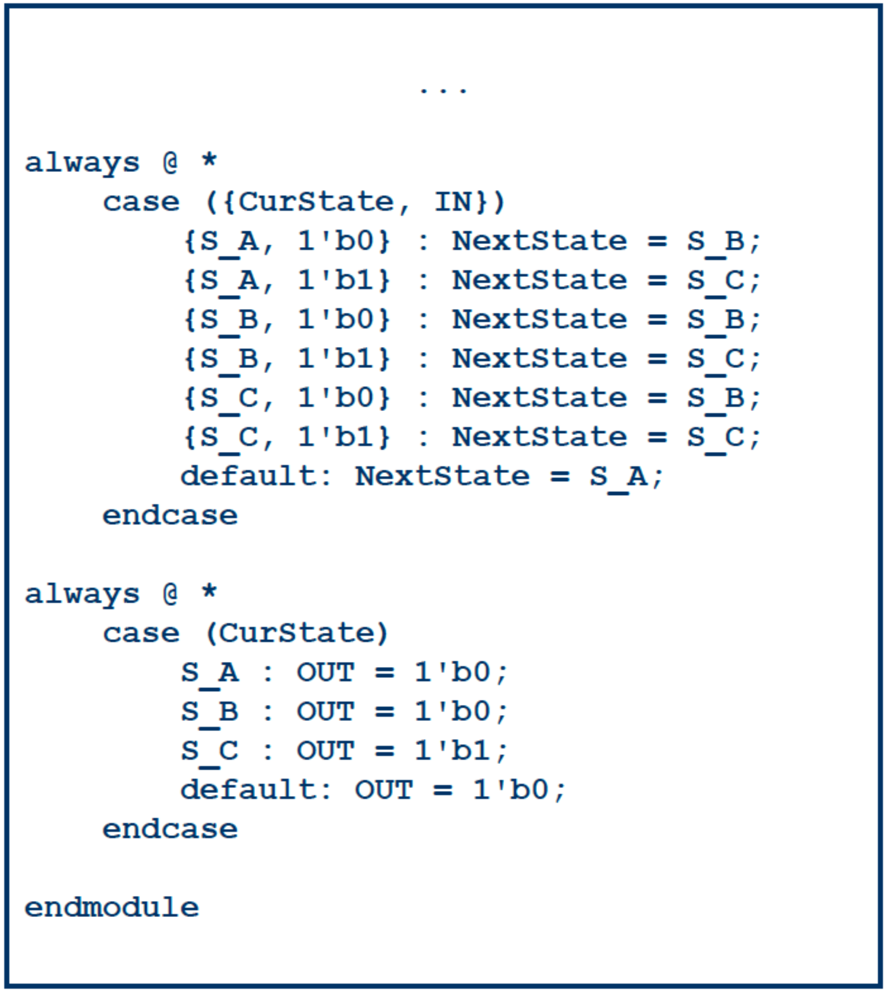
- **Mealy Machine**
  - Output is determined
    - by current state & input
  - Next state is determined
    - by current state & input

Moore Machine Example



```
module MOORE  (
    input     wire    CLK,
    input     wire    RST,
    input     wire    IN,
    output    reg     OUT
);
localparam    S_A = 2'b00;   // state A
localparam    S_B = 2'b01;   // state B
localparam    S_C = 2'b10;   // state C

reg    [1:0]  CurState;   // Current state
reg    [1:0]  NextState;  // Next state

always @ (posedge CLK)
    if (RST)
       CurState = S_A;
    else
       CurState = NextState;

                     ...
```

Parameter definition which can not be changed from outside



```
                     ...

always @ *
    case ({CurState, IN})
        {S_A, 1'b0} : NextState = S_B;
        {S_A, 1'b1} : NextState = S_C;
        {S_B, 1'b0} : NextState = S_B;
        {S_B, 1'b1} : NextState = S_C;
        {S_C, 1'b0} : NextState = S_B;
        {S_C, 1'b1} : NextState = S_C;
        default: NextState = S_A;
    endcase

always @ *
    case (CurState)
        S_A : OUT = 1'b0;
        S_B : OUT = 1'b0;
        S_C : OUT = 1'b1;
        default: OUT = 1'b0;
    endcase

endmodule
```

Mealy Machine Example

```
module MEALY (
    input    wire    CLK,
    input    wire    RST,
    input    wire    IN,
    output   reg     OUT
);
localparam    S_A = 2'b00;    // state A
localparam    S_B = 2'b01;    // state B
localparam    S_C = 2'b10;    // state C

reg    [1:0]  CurState;    // Current state
reg    [1:0]  NextState;   // Next state

always @ (posedge CLK)
    if (RST)
        CurState = S_A;
    else
        CurState = NextState;


                          ...
```

```
                        ...

always @ *
    case ({CurState, IN})
        {S_A, 1'b0} :
            {NextState, OUT} = {S_B, 1'b0};
        {S_A, 1'b1} :
            {NextState, OUT} = {S_C, 1'b0};
        {S_B, 1'b0} :
            {NextState, OUT} = {S_B, 1'b0};
        {S_B, 1'b1} :
            {NextState, OUT} = {S_C, 1'b1};
        {S_C, 1'b0} :
            {NextState, OUT} = {S_B, 1'b1};
        {S_C, 1'b1} :
            {NextState, OUT} = {S_C, 1'b0};
        default:
            {NextState, OUT} = {S_A, 1'b0};
    endcase

endmodule
```

Outline

- **Lab Class Operating Plan**
- **Verilog HDL: Introduction**
- **Verilog Syntax**
  - Module Body
  - Assignments
  - Delay & Event
  - Flow Control
  - Design Method: FSM
  - More on Verilog Syntax
- **Verilog Simulation**

More on Verilog Syntax

- **Comment**
  - // or /* ... */ ➔ the same with C language
- **Constant**
  - Format: <the num. of bits>’<number system><value>
  - <number system>
    - h(hexadecimal), d(decimal), o(octal), b(binary)
  - <value>
    - 0~9, A~F, X(undefined), Z(High Impedance)
  - _(underbar): for better visibility of long digit values
  - Just number can be used for decimal numbers
    - ex) 8’b0100_1111 (8-bit binary 01001111)
      16’hAF (16-bit hexadecimal AF)

More on Verilog Syntax

- **Operators**
  - Similar to C language operators except for ++ and --
  - Monomial / binomial logical opertors:

    and(&), nand(~&), or(|), nor(~|), xor(^), xnor(~^), not(~)
  - Equality operators

    a==b, a!=b ➔ return x if a or b is x or z

    a===b, a!===b ➔ comparison including x and z
- **Array and Vectors**

```
reg  [31:0] memory   [0:511];        // 32-bit memory w/ 512 elem.
reg         regA     [3:0][0:511];   // 2D array of 1-bit reg
wire [7:0] bus_array [0:511];        // 8-bit bus array of size 12
```

Reference

- Verilog Study Webpages
  - Verilog Tutorial & Examples

    http://www.asic-world.com/verilog/index.html
  - Verilog Syntax: Compact Summary

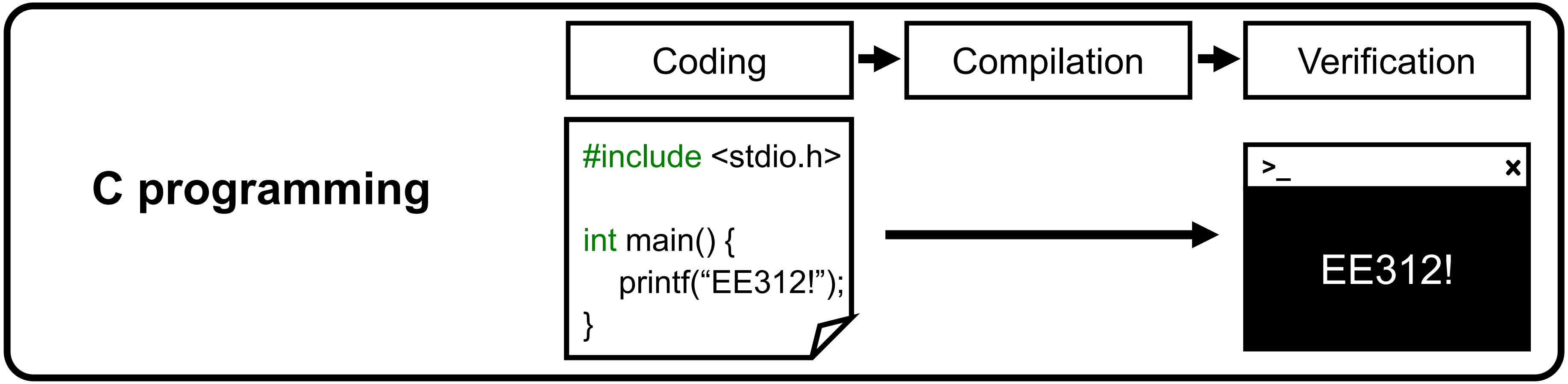
    https://www.csee.umbc.edu/portal/help/VHDL/verilog/summary.html

Outline

- Lab Class Operating Plan
- Verilog HDL: Introduction
- Verilog Syntax
- **Verilog Simulation**
  - Verilog RTL Simulation
  - Simulation using ModelSim

Verilog RTL Simulation

- **Procedures of Verilog Simulation**
  - Similar to C programming

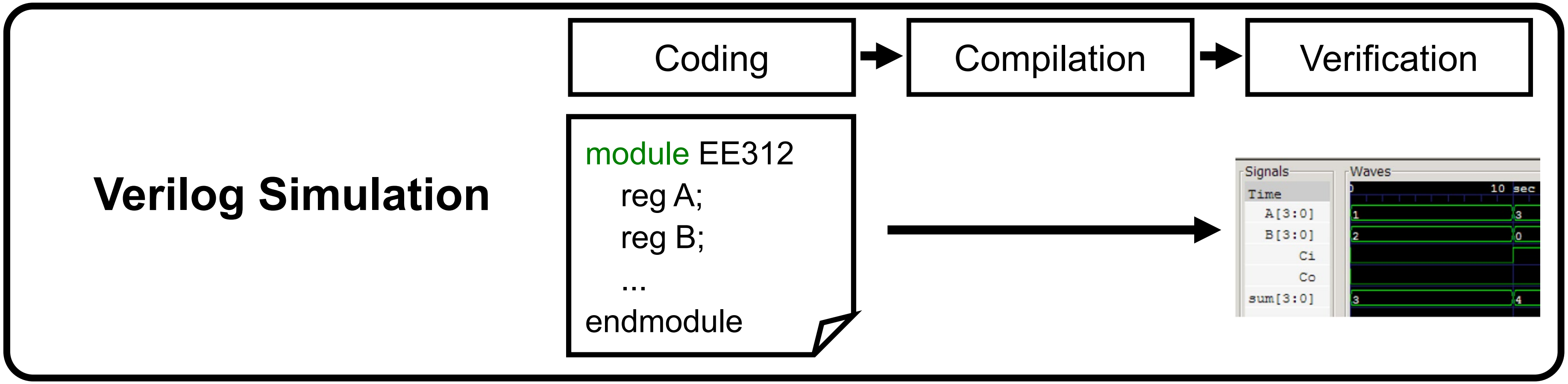


C programming

Coding → Compilation → Verification

```
#include <stdio.h>

int main() {
    printf(“EE312!”);
}
```

```
EE312!
```



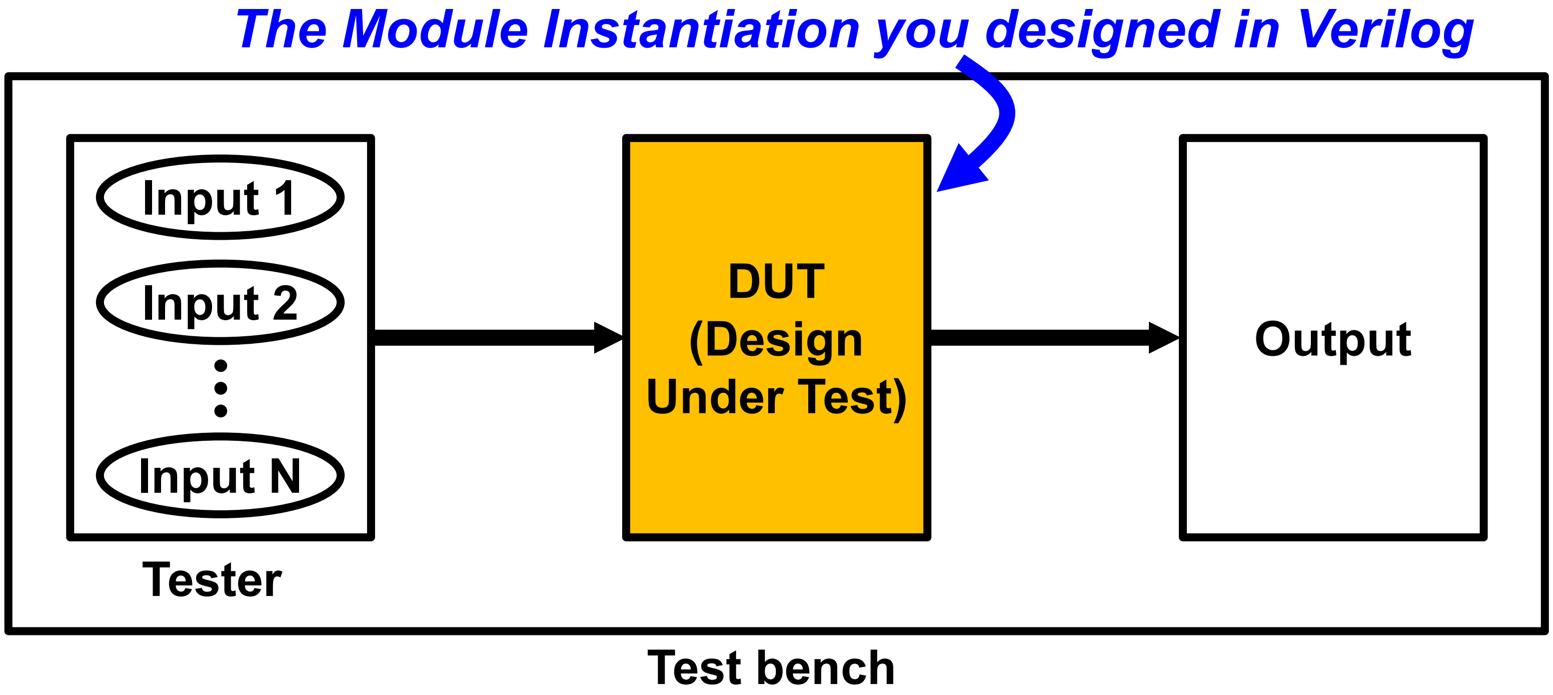
Verilog Simulation

Coding → Compilation → Verification

```
module EE312
    reg A;
    reg B;
    ...
endmodule
```

Testbench for Test Designed Module

- **Testbench**
  - Test signal generation and output data recording with instantiation of the designed module
  - Test signal (stimulus) generation as input to DUT



Outline

- Lab Class Operating Plan
- Verilog HDL: Introduction
- Verilog Syntax
- **Verilog Simulation**
  - Verilog RTL Simulation
  - Simulation using ModelSim

ModelSim Guide

- Please refer to
  - “[EE312]ModelSim_guide.pdf”