

Verilog Tutorial

Chao-Hsien, Hsu

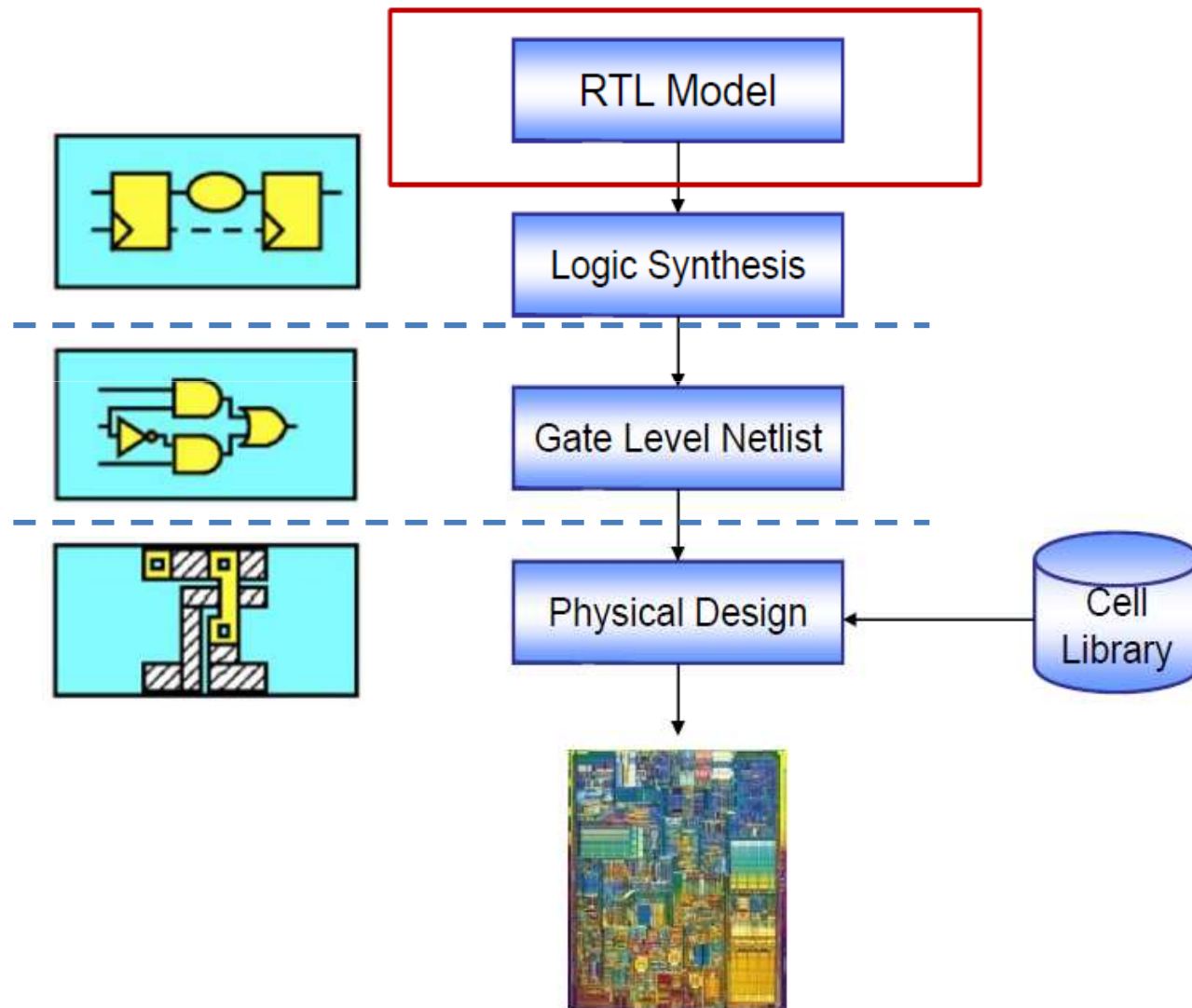
Computer Architecture

Date: 2011/4/12

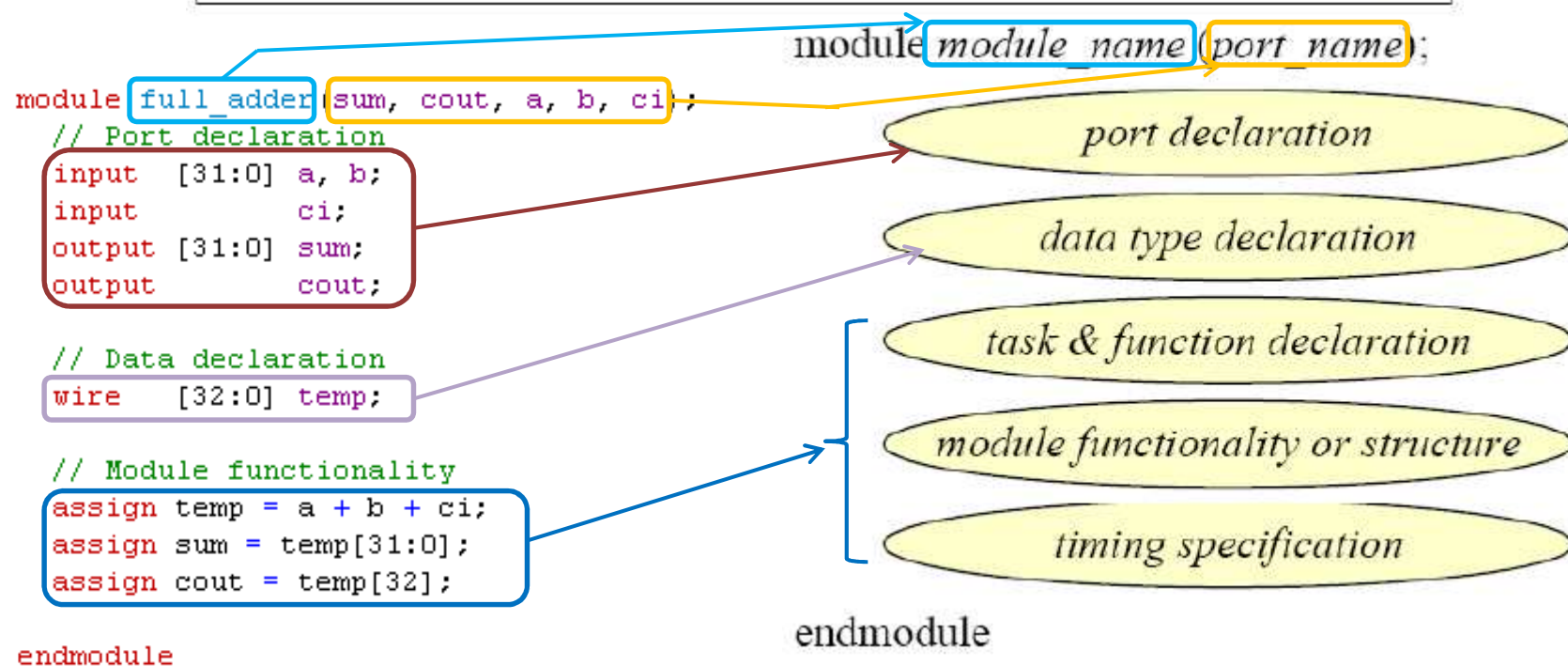
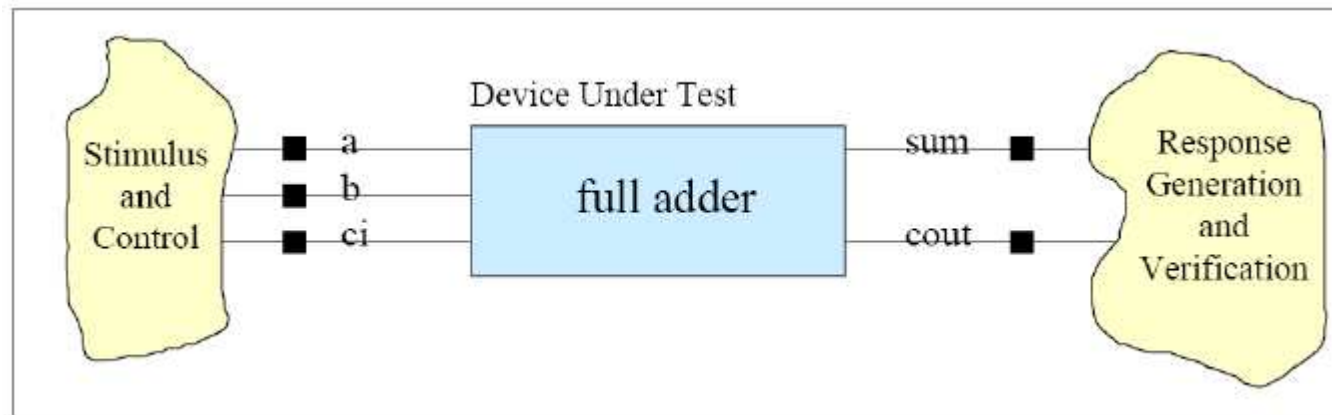
Outline

- Verilog & Example
- Major Data Type
- Operators
- Conditional & Looping Statements
- Behavior Modeling
- Structure Modeling
- Verification Methodology

IC Design Flow



Example: 32 bits Full Adder



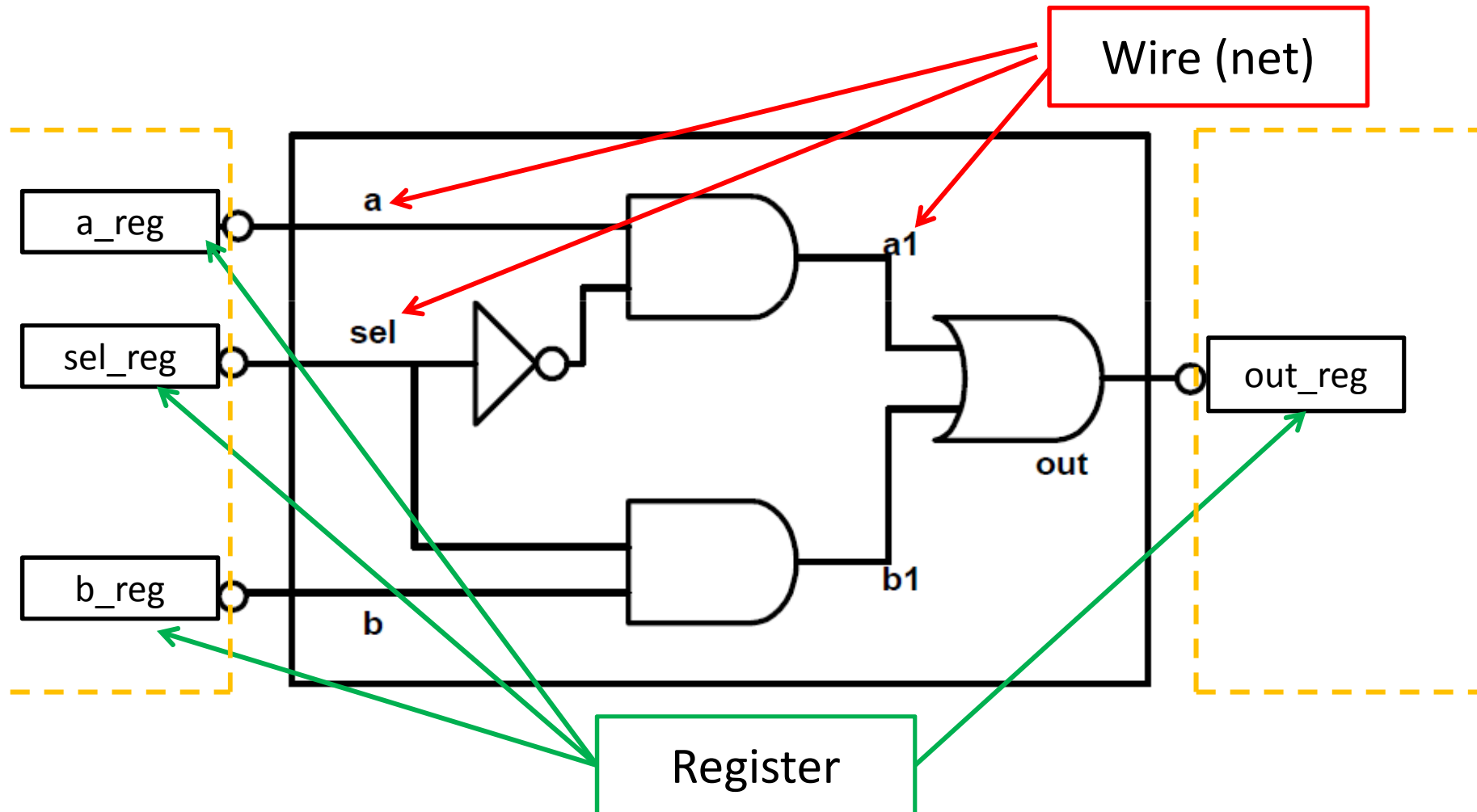
Outline

- Verilog & Example
- Major Data Type
- Operators
- Conditional & Looping Statements
- Behavior Modeling
- Structure Modeling
- Verification Methodology

Wire vs. Register (1/4)

- Wire
 - Physical wire in the circuit
 - A wire does not store its value, it must be driven by
 - connecting the wire to the output of a gate or module
 - assigning a value to the wire in a **continuous assignment**
 - Can not use “wire” in left-hand-side of assignment in **procedural block**
- Register
 - Not “register” of CPU
 - No guarantee to be a DFF(D-flip flop)
 - Maybe a physical wire
 - Holding its value until a new value is assigned to it.
 - It is event-driven.
 - Can not use “reg” in left-hand side of **continuous assignment**

Wire vs. Register (2/4)

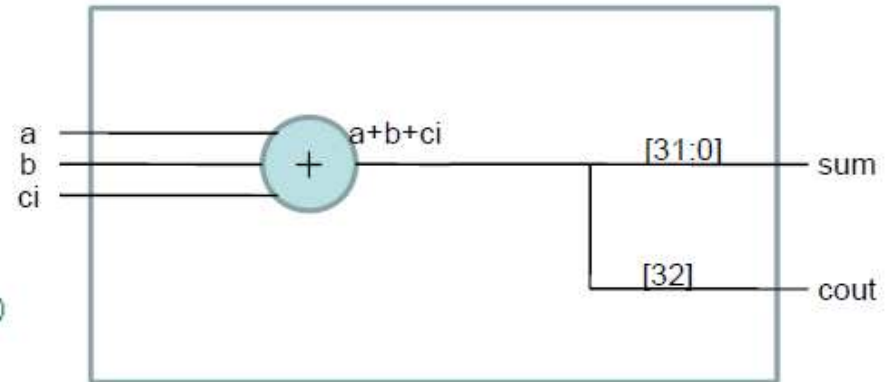


Wire vs. Register (3/4)

```

1 module Full_Adder(sum, cout, a, b, ci);
2
3 // Interface
4 input  [31:0]    a, b;
5 input          ci;
6 output [31:0]    sum;
7 output          cout;
8
9 // Calculation (with Continuous Assignment)
10 assign {cout, sum} = a + b + ci;
11
12 endmodule

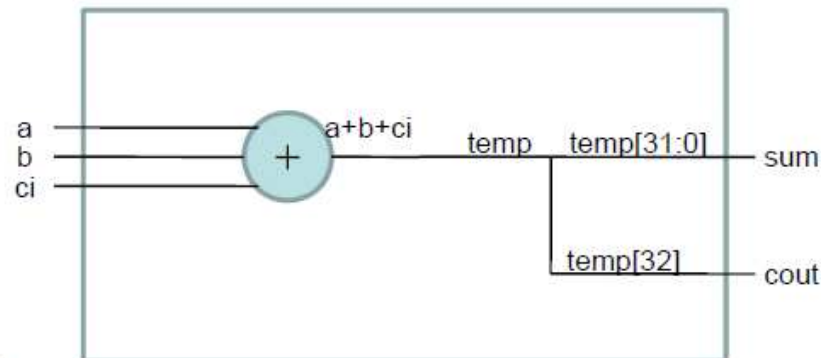
```



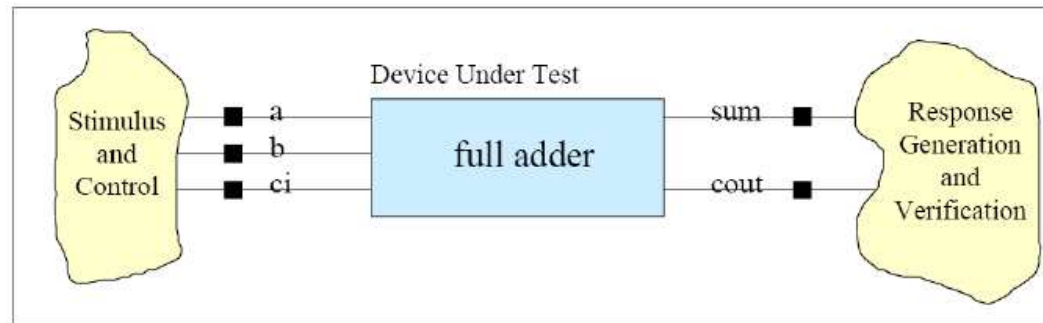
```

1 module Full_Adder(sum, cout, a, b, ci);
2
3 // Interface
4 input  [31:0]    a, b;
5 input          ci;
6 output [31:0]    sum;
7 output          cout;
8
9 reg  [32:0]      temp;
10
11 assign sum = temp[31:0];
12 assign cout = temp[32];
13
14 // Calculation (with Always Procedural Block)
15 always@(a or b or ci) begin
16     temp = a + b + ci;
17 end
18
19 endmodule

```



Wire vs. Register (4/4)



```
1 module Full_Adder(sum, cout, a, b, ci);
2
3 // Interface
4 input  [31:0]    a, b;
5 input          ci;
6 output [31:0]    sum;
7 output          cout;
8
9 reg  [32:0]      temp;
10
11 // Calculation (with Continuous Assignment)
12 assign temp = a + b + ci;
13 assign sum  = temp[31:0];
14 assign cout = temp[32];
15
16 endmodule
```

**** Error: (12): Register is illegal in left-hand side of continuous assignment**

```
1 module Full_Adder(sum, cout, a, b, ci);
2
3 // Interface
4 input  [31:0]    a, b;
5 input          ci;
6 output [31:0]    sum;
7 output          cout;
8
9 wire [32:0]      temp;
10
11 assign sum  = temp[31:0];
12 assign cout = temp[32];
13
14 // Calculation (with Always Procedural Block)
15 always@(a or b or ci) begin
16     temp = a + b + ci;
17 end
18
19 endmodule
```

**** Error: (13): (vlog-2110) Illegal reference to net "temp"**

Integer & Real Numbers

16 --- 32 bits decimal

8'd16

8'h10

8'b0001_0000

8'o20

32'bx --- 32 bits x

X: unknown

2'b1? --- “?” represents a high impedance bit

6.3

5.3e-4

6.2E3

Outline

- Verilog & Example
- Major Data Type
- **Operators**
- Conditional & Looping Statements
- Behavior Modeling
- Structure Modeling
- Verification Methodology

Operators

arithmetic operator

operator	operation
+	arithmetic addition
-	arithmetic subtraction
*	arithmetic multiplication
/	arithmetic division
%	arithmetic modulus

other operators

operator	operation
>>	logical shift right
<<	logical shift left
==, !=	equality
===, !==	identity
?:	conditional
{}	concatenate
{() }	replicate

unary operator (1-bit result)

operator	operation
&	unary reduction AND
~&	unary reduction NAND
	unary reduction OR
~	unary reduction NOR
^	unary reduction XOR
~^	unary reduction XNOR

- unary operation will perform the operation on each bit of the operand and get a one-bit result.

[8'b00101101 is 1'b1

bit-wise operators

operator	operation
~	bit-wise NOT
&	bit-wise AND
	bit-wise OR
^	bit-wise XOR
~^	bit-wise XNOR

- binary bit-wise operation will perform the operation one bit of a operand and its equivalent bit on the other operand to calculate one bit for the result.

(8'b11110000 & 8'b00101101) is 8'b00100000

logical operators

operator	operation
!	logical NOT
&&	logical AND
	logical OR
==	logical equality
!=	logical inequality
===	logical identity
!==	the inverse of ===

- logical operator operate with logic values. (non-zero is true, and zero value is false).

if(sel == 4'h03)..... else

Operators: Example

- Example
 - $A = 4'b1101$, $B = 4'b1010$
 - Logical
 - $A || B = 1$
 - $A \& B = 1$
 - Bit-wised
 - $A | B = 4'b1111$
 - $A \& B = 4'b1000$
 - Unary
 - $|A = 1$, $\&A = 0$
 - $|B = 1$, $\&B = 0$

Equality vs. Identity

- “**=**” is the assignment operator.

- “**==**” is the equality operator

– $A = 2'b1x$; $B = 2'b1x$;

- $A == B$ (?)

==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

- “**===**” is the identity operator

– $A = 2'b1x$; $B = 2'b1x$;

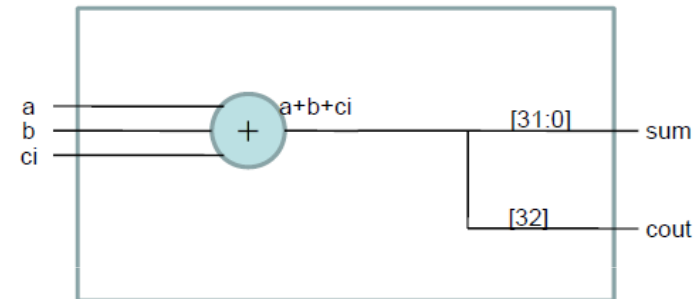
- $A === B$ (?)

===	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

Concatenation & Replication

▼ Concatenation operator in LHS

```
module add_32 (co, sum, a, b, ci);  
    output co;  
    output [31:0] sum;  
    input [31:0] a, b;  
    input ci;  
    assign #100 {co, sum} = a + b + ci;  
endmodule
```



▼ Bit replication to produce *01010101*

```
assign byte = {4{2'b01}};    // 8'b01010101
```

▼ Sign Extension

```
assign word = {{8{byte[7]}}, byte};
```

Outline

- Verilog & Example
- Major Data Type
- Operators
- Conditional & Looping Statements
- Behavior Modeling
- Structure Modeling
- Verification Methodology

Behavior Model (1/3)

- At system level, system's **function view** is more important than implementation.
 - You do not have any idea about how to implement your net-list.
 - The data flow of this system is analyzed.
 - You may need to explore different design options.
- Behavior modeling enables you to describe the system at a **high-level of abstraction**.
- All you need to do is to describe the behavior of your design.

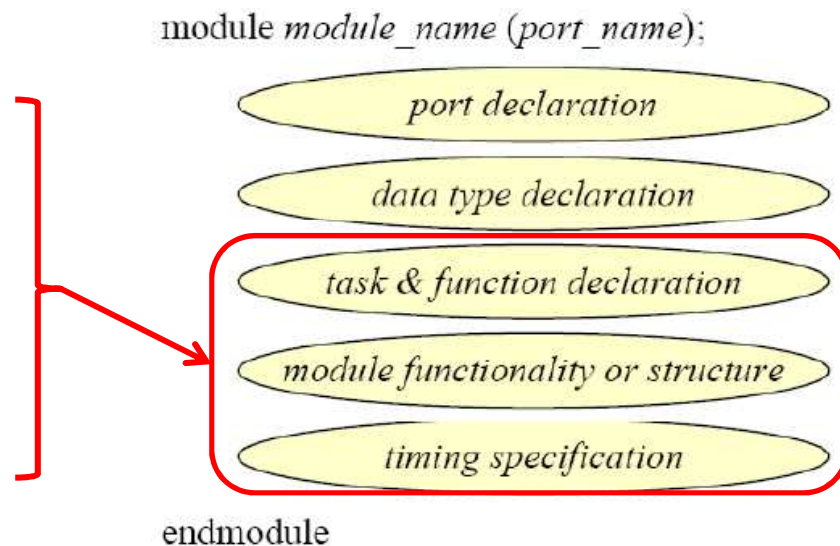
Behavior Model (2/3)

- Describing the behavior of your design(circuits).
 - Action
 - How do you model your circuit's behaviors?
 - Timing control
 - What time to do what thing
 - What condition to do what thing
 - You may need to explore different design options.
- Behavior modeling enables you to describe the system at a **high-level of abstraction**.

Behavior Model (3/3)

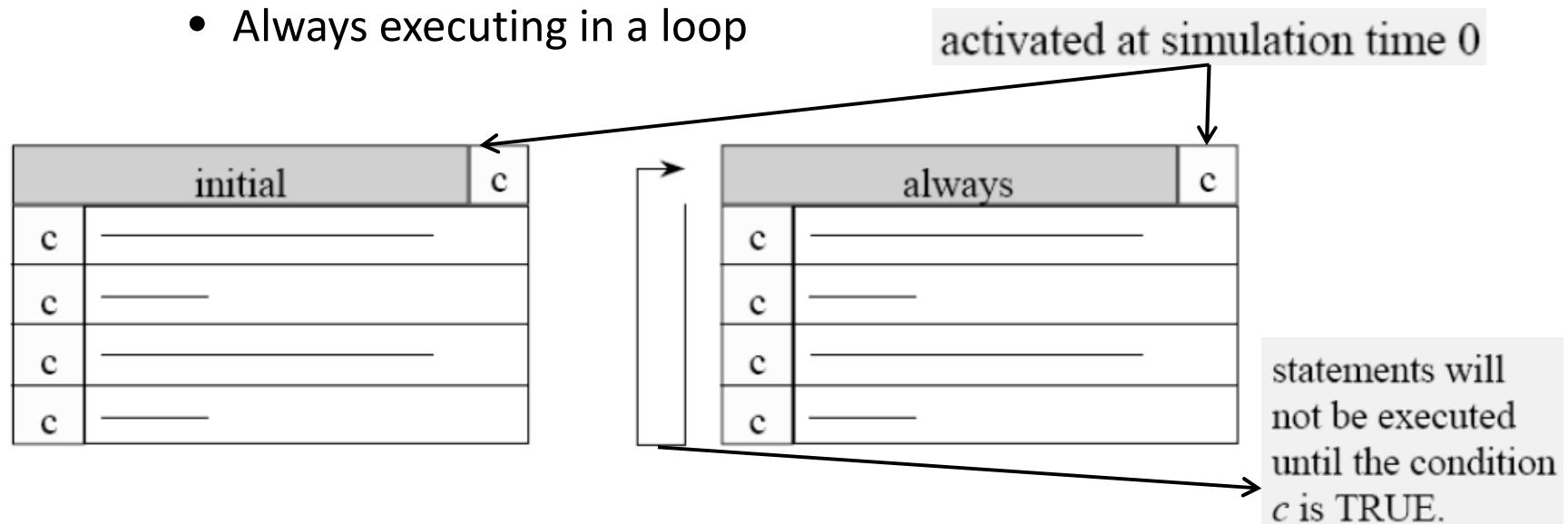
- All you need to do is to describe the behavior of your design.

- Procedural block
- Procedural assignment
- Timing control
- Control statement



Procedural Blocks

- Procedural block is the basic of behavior modeling.
 - One behavior in one procedural block
- Two types
 - **“Initial”** procedural block
 - Execute only once
 - **“Always”** procedural block
 - Always executing in a loop



Procedural Block: Example

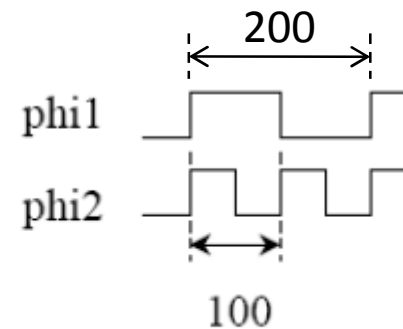
```
module clock_gen (phi1, phi2);  
  output phi1, phi2;  
  reg phi1, phi2;
```

```
  initial begin  
    phi1 = 0; phi2 = 0;  
  end
```

```
  always #100 phi1 = ~ phi1
```

```
  always @(posedge phi1)  
  begin  
    phi2 = 1;  
    #50 phi2 = 0;  
    #50 phi2 = 1;  
    #50 phi2 = 0;  
  end
```

```
endmodule
```



These procedural blocks are activated and executed at simulation time 0.

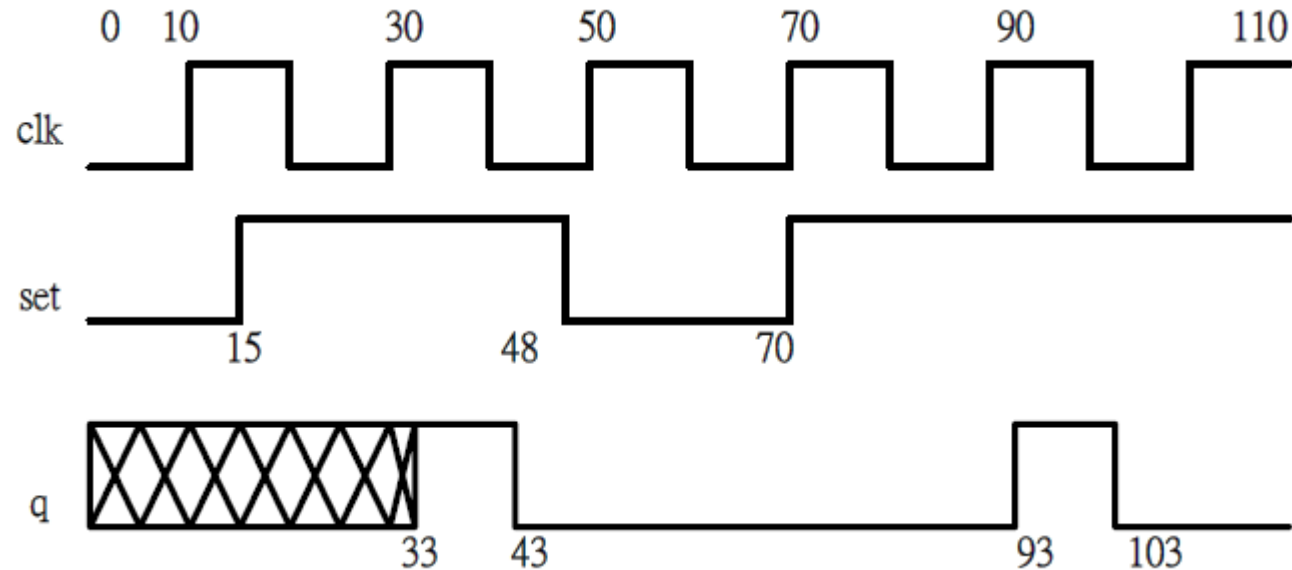
This procedural block is activated at simulation time 0 but executed at positive edge of *phi1*.

Timing Control

- Simple Delay
 - **#10** rega = regb;
 - **#(cycle/2)** clk = ~clk; // cycle is declared as parameter
- Edge-Trigger Timing Control
 - **@(r or q)** rega = regb; // Controlled by “r” or “q”
 - **@(posedge clk)** rega = regb; // positive edge
 - **@(negedge clk)** rega = regb; // negative edge
- Level-Triggered Timing Control
 - **wait (!enable)** rega = regb; // will wait until enable = 0

Example

```
always wait(set)
begin
  @(posedge clk)
    #3 q = 1;
    #10 q = 0;
    wait(!set);
end
```



Procedural Assignment

```
1 module Full_Adder(sum, cout, a, b, ci);
2
3 // Interface
4 input  [31:0]    a, b;
5 input          ci;
6 output [31:0]    sum;
7 output          cout;
8
9 reg  [32:0]      temp;
10
11 assign sum = temp[31:0];
12 assign cout = temp[32];
13
14 // Calculation (with Always Procedural Block)
15 always@(a or b or ci) begin
16     temp = a + b + ci;
17 end
18
19 endmodule
```

- Continuous Assignment
 - Cannot be inside procedural block
- Procedural Assignment
 - Must be inside procedural block
 - Blocking
 - Non-blocking

```
module f_adder (sum, co, a, b, ci);
```

```
    output sum, co;
```

```
    input  a, b, ci;
```

```
    reg sum;
```

```
    sum = a ^ b ^ ci;
```

```
    always @(a or b or ci)
```

```
        assign co = (a & b) | (b & ci) | (ci & a);
```

```
endmodule
```

Error! Illegal left-hand-side continuous assignment.

Error! Illegal left-hand-side in assign statement

Blocking and Non-blocking Procedural Assignment (1/2)

- Blocking

```
always@(posedge clock) begin
    x = a;
    y = x;
    z = y;
end
```

x = a x = a
y = x → y = x
z = y z = x

- Non-blocking

```
always@(posedge clock) begin
    x <= a;
    y <= x;
    z <= y;
end
```

Shift register
x = a
y = x_old
z = y_old

Blocking and Non-blocking Procedural Assignment (2/2)

- Usage Policies
 - Non-blocking is only used in “Always” block with “clock”

```
always@(posedge clk or negedge reset_n) begin
    if (!reset_n)
        counter <= 8'b00;
    else
        counter <= counter + 1;
end
```

- Blocking is used in “Always” block without “clock”

```
always@(sel or a or b) begin
    case (sel)
        2'b00 : c = a;
        2'b01 : c = b;
    endcase
end
```

- Blocking is used in continuous assignment
- Only one assignment is used in “Always” block

```
assign y = a&b;
```

Conditional Statements (1/2)

- If and if-else statement

```
if (expression)  
begin  
  // statement  
end  
else  
begin  
  // statement  
end
```



```
if (a >= b)  
begin  
  result <= 1;  
end  
else  
begin  
  result <= 0;  
end
```

```
if (expression)  
begin  
  // statement  
end  
else if (expression)  
begin  
  // statement  
End  
else  
begin  
  // statement  
end
```



```
if (a > b)  
begin  
  result <= 2;  
end  
else if (a < b)  
begin  
  result <= 1;  
end  
else  
begin  
  result <= 0;  
end
```

Conditional Statements (2/2)

- Case statement

```
`define pass_accum 4'b0000
`define pass_data 4'b0001
`define ADD 4'b0010
`define AND 4'b0011
`define XOR 4'b0100

case(opcode)
  `pass_accum: alu_out = accum;
  `pass_data: alu_out = data;
  `ADD: alu_out = accum + data;
  `AND: alu_out = accum & data;
  `XOR: alu_out = accum ^ data;
  default: alu_out = 8'b11111111;
```

Looping Statements (1/2)

■ For Loops

```
for (index = 0; index < size; index = index + 1)
    if (val[index] == 1'bx)
        $display ("found an X");
```

```
for (i = 10; i > index; i = i - 1)
    memory[i] = 0;
```

Looping Statements (2/2)

■ While Loops

```
...  
reg [7:0] tempreg;  
reg count;  
...  
...  
count = 0;  
while (tempreg)  
begin  
    if (tempreg[0]) count = count + 1;  
    tempreg = tempreg >> 1; //Right Shift  
end  
endmodule
```

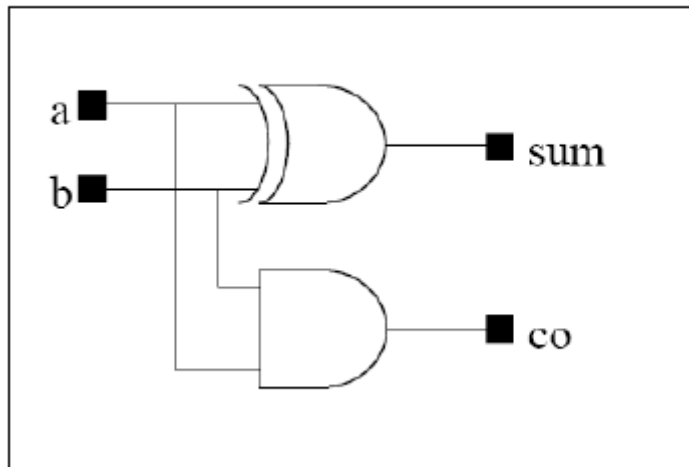
Outline

- Verilog & Example
- Major Data Type
- Operators
- Conditional & Looping Statements
- Behavior Modeling
- Structure Modeling
- Verification Methodology

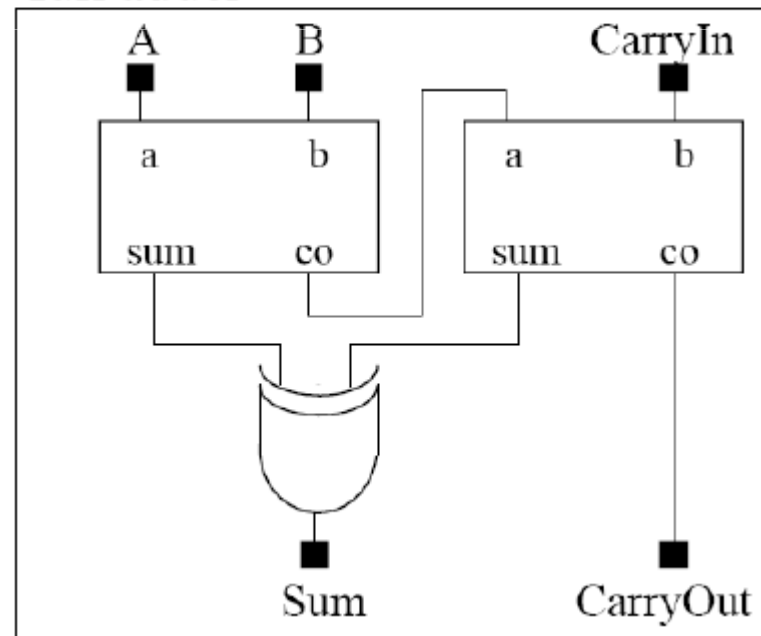
Structure Modeling (1/4)

- In structural modeling, you connect components with each other to create a more complex component.

half adder

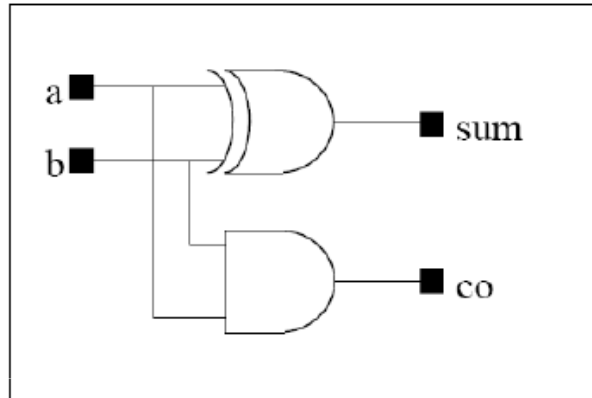


full adder



Structure Modeling (2/4)

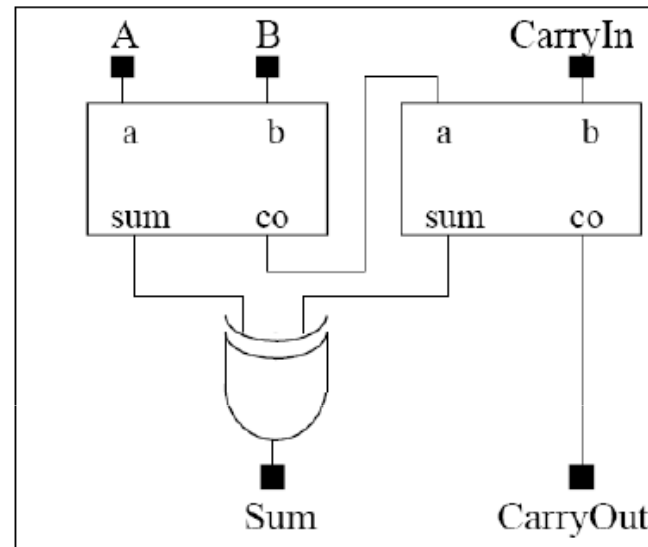
half adder



```
module HA(a,b,sum,co);  
  input a, b;  
  output sum, co;  
  assign sum = a ^ b;  
  assign co = a & b;  
endmodule
```

HA.v

full adder

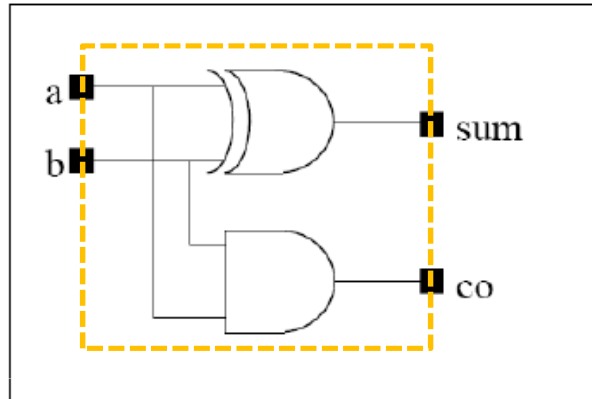


```
module FA(A,B,CarryIn,Sum,CarryOut);  
  input A, B, CarryIn;  
  output Sum, CarryOut;  
  wire sum0, sum1, co0;  
  HA ha0(A,B,sum0,co0);  
  HA ha1(co0,CarryIn,sum1,CarryOut);  
  assign Sum = sum0 ^ sum1;  
endmodule
```

43
FA.v

Structure Modeling (3/4)

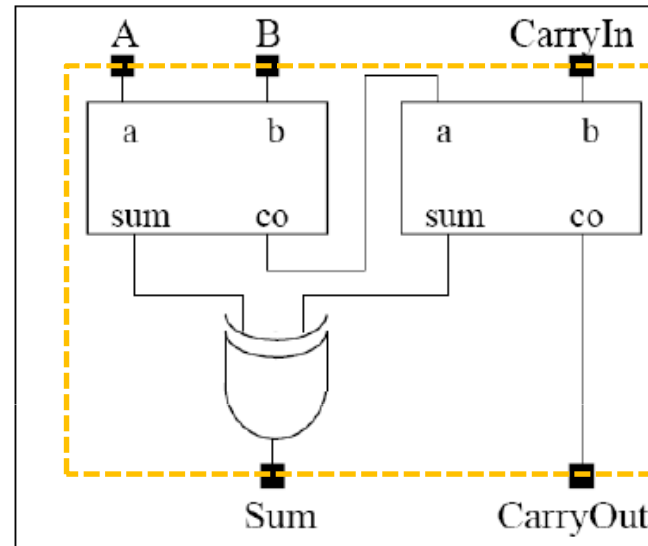
half adder



```
module HA(a, b, sum, co);
  input a, b;
  output sum, co;
  assign sum = a ^ b;
  assign co = a & b;
endmodule
```

HA.v

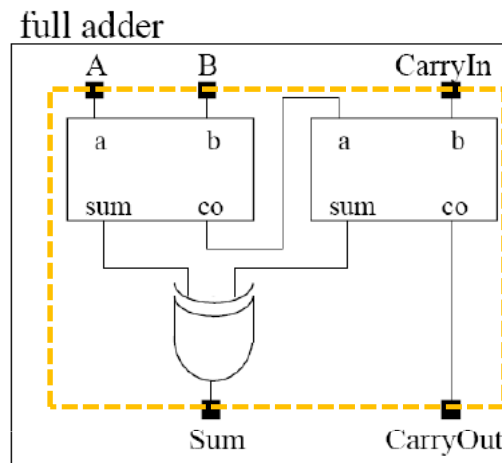
full adder



```
module FA(A, B, CarryIn, Sum, CarryOut);
  input A, B, CarryIn;
  output Sum, CarryOut;
  wire sum0, sum1, co0;

  HA ha0(.a(A), .b(B), .sum(sum0), co(co0));
  HA ha1(.a(co0), .b(CarryIn), .sum(sum1), co(CarryOut));
  assign Sum = sum0 ^ sum1;
endmodule
```

Structure Modeling (4/4)



```

module FA(A,B,CarryIn,Sum,CarryOut);
  input A, B, CarryIn;
  output Sum, CarryOut;
  wire sum0, sum1, co0;
  HA ha0(A,B,sum0,co0);
  HA ha1(co0,CarryIn,sum1,CarryOut);
  assign Sum = sum0 ^ sum1;
endmodule
  
```

```

module FA(A, B, CarryIn, Sum, CarryOut);
  input A, B, CarryIn;
  output Sum, CarryOut;
  wire sum0, sum1, co0;

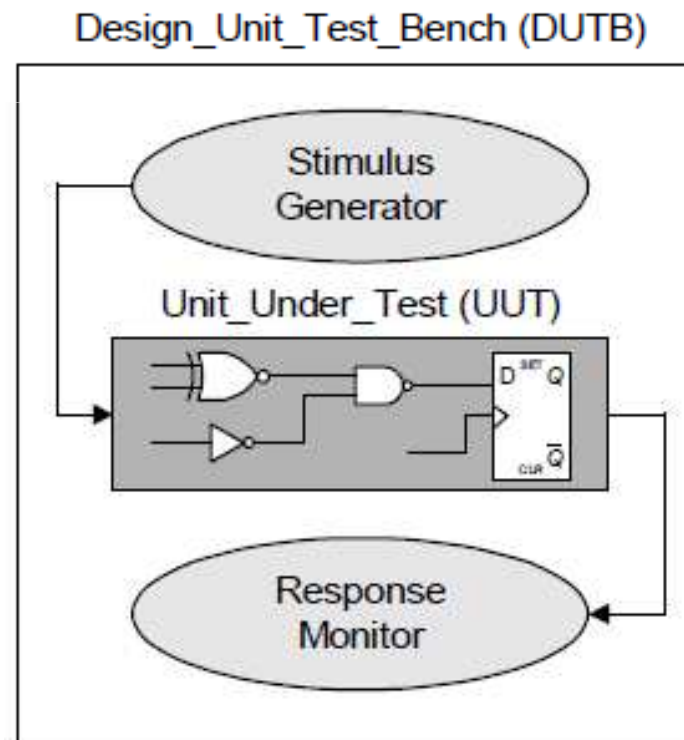
  HA ha0(.a(A), .b(B), .sum(sum0), co(co0));
  HA ha0(.a(co0), .b(CarryIn), .sum(sum1), co(CarryOut));
  assign Sum = sum0 ^ sum1;
endmodule
  
```

Outline

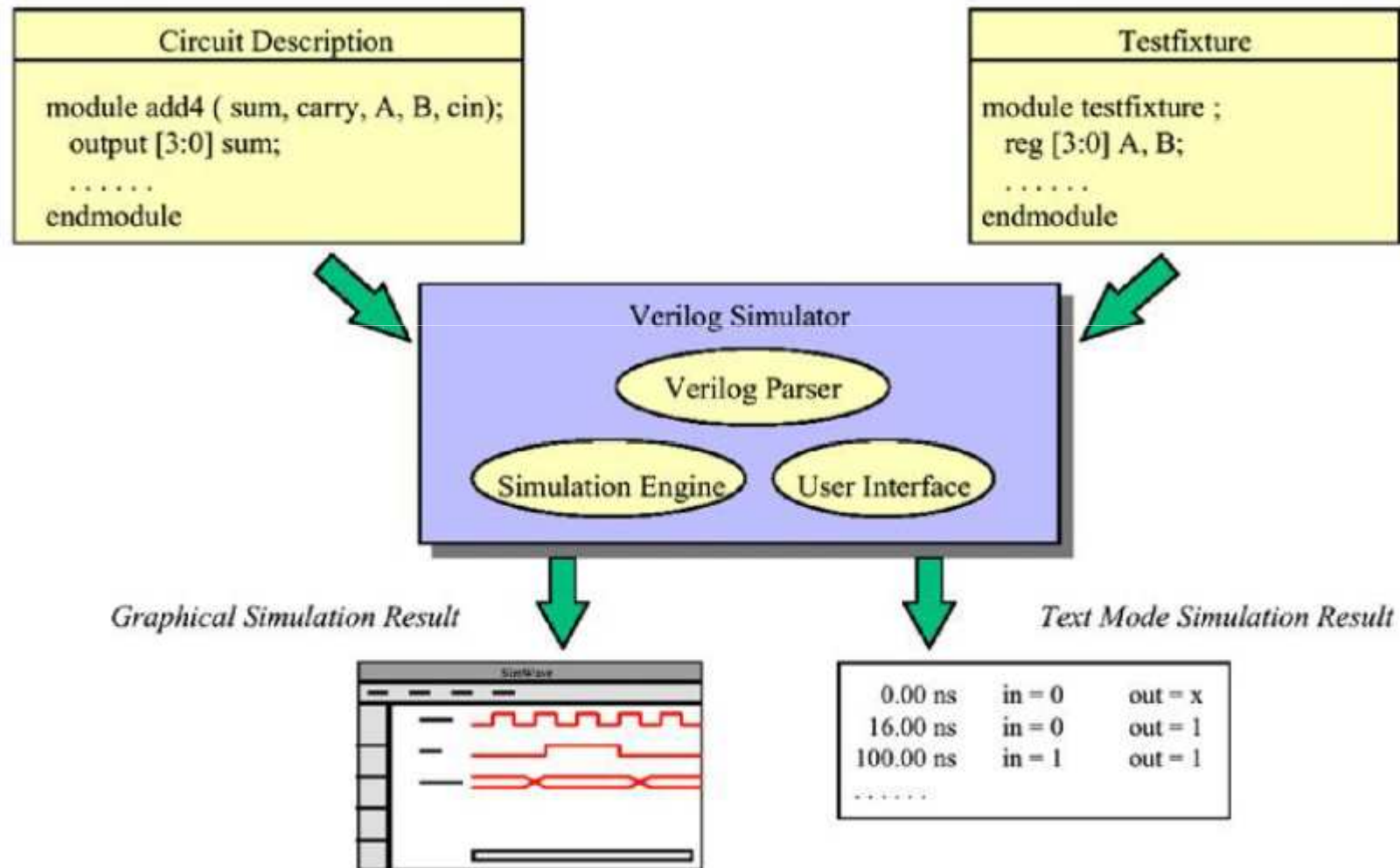
- Verilog & Example
- Major Data Type
- Operators
- Conditional & Looping Statements
- Behavior Modeling
- Structure Modeling
- Verification Methodology

Verification Methodology

- Simulation
 - Detect syntax violations in source code
 - Simulate behavior
 - Monitor results



Verilog Simulator (ModelSim)



VHDL & Verilog

- Process Block
- Signal Assignment
- Interface Declaration
- Buses

Process Block

Process Block

VHDL:

```
    process (siga, sigb)
    begin
        .....
    end;
```

Verilog:

```
    always @ (siga or sigb)
    begin
        ....
    end
```

Both used to specify blocks
of logic with multiple
inputs/outputs

Signal Assignment

VHDL:

```
signal a, b, c, d: std_logic;  
  
begin  
  
    a <= b and c;  
    d <= (c or b) xor (not (a) and b);  
end;
```

VERILOG:

```
wire a,b,c,d;  
  
assign a = b & c;  
assign d = (c | b) ^ (~a & b);
```

Declarations for
one-bit wires are
optional.

Logical operators
same as in C.

Interface Declaration

VHDL:

```
entity mux is
  port ( a,b, s: in std_logic;
         y : out std_logic);

architecture a of mux is
begin
  .....
end;
```

VERILOG:

```
module mux (a,b,s,y);
  input a,b,s;
  output y;

  ....
endmodule
```

Buses

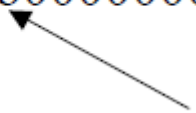
VHDL:

```
signal a,c: std_logic_vector(7 downto 0);  
begin  
    a(3 downto 0) <= c (7 downto 4);  
    c(0) <= '0';  
    c<= "00001010";  
  
end;
```

Verilog:

```
wire [7:0] ;  
begin  
    assign a[3:0] = b[7:4];  
    assign a[0] <= 0;  
    assign a = 'b00000000;  
  
end;
```

Value specified in binary.



Reference

- Some contents are referenced from the below materials
 - The slides of the “VLSI System Design” course by **Prof. Kuen-Jong Lee**
 - The slides of the “Digital System Design” course by **Prof. An-Yeu (Andy) Wu**
 - http://www.ece.msstate.edu/~reese/EE4743/lectures/verilog_intro_2002/verilog_intro_2002.pdf