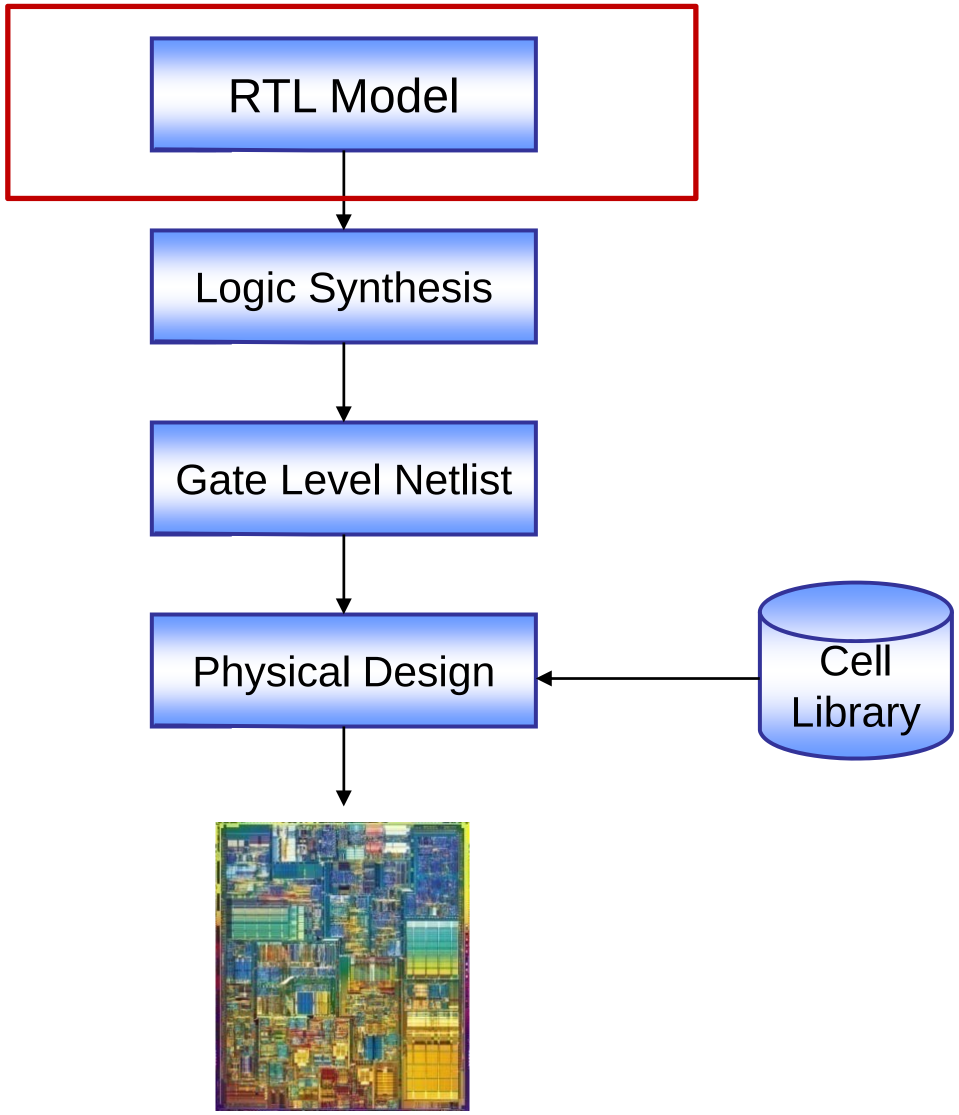
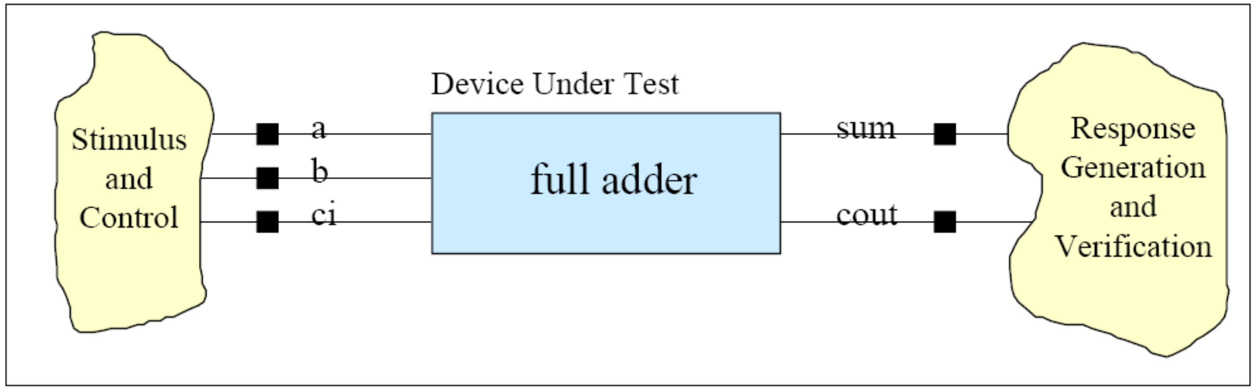
Verilog Tutorial

Outline

IC Design Flow



Full Adder for Example



module *module_name* (*port_name*);

port declaration

data type declaration

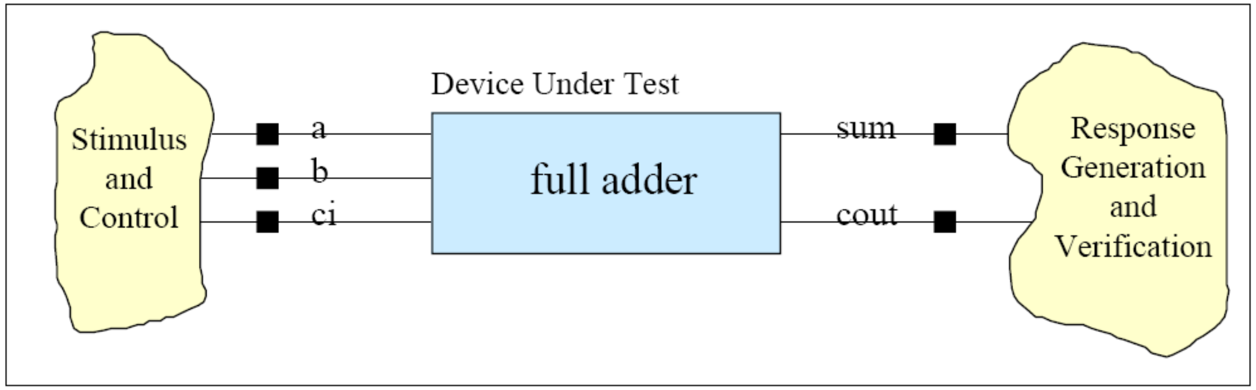
task & function declaration

module functionality or structure

timing specification

endmodule

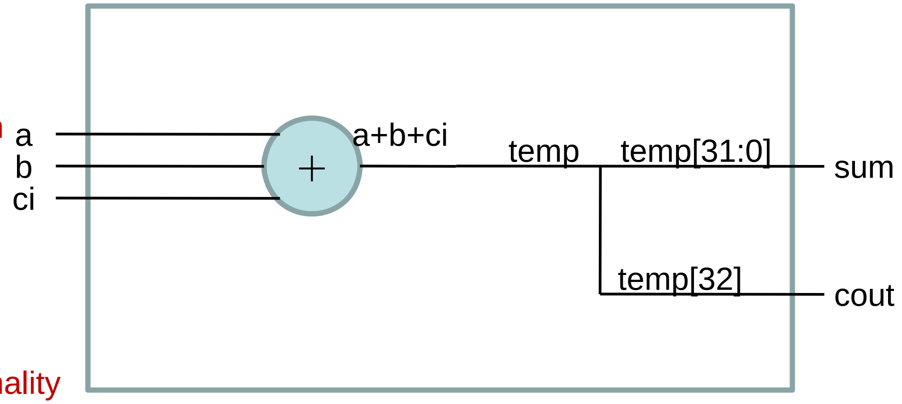
32-bit Full Adder: example 1



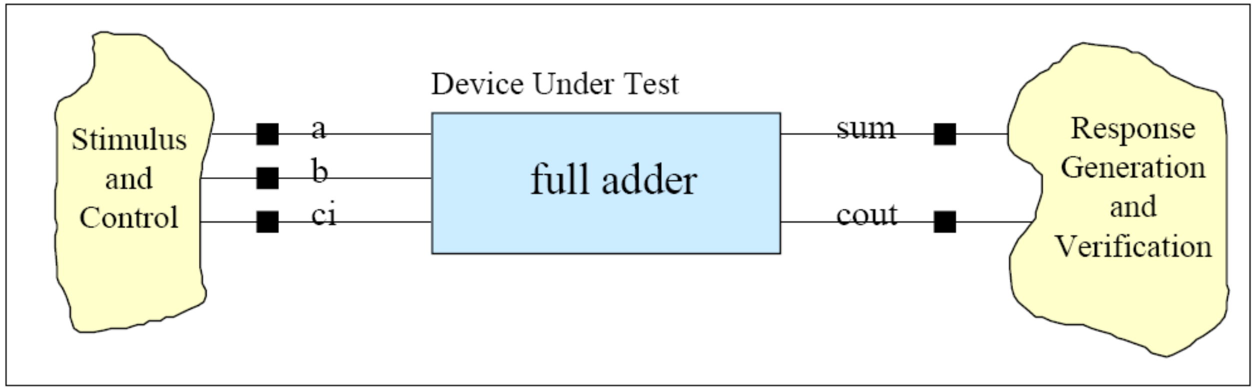
module_name
port_name

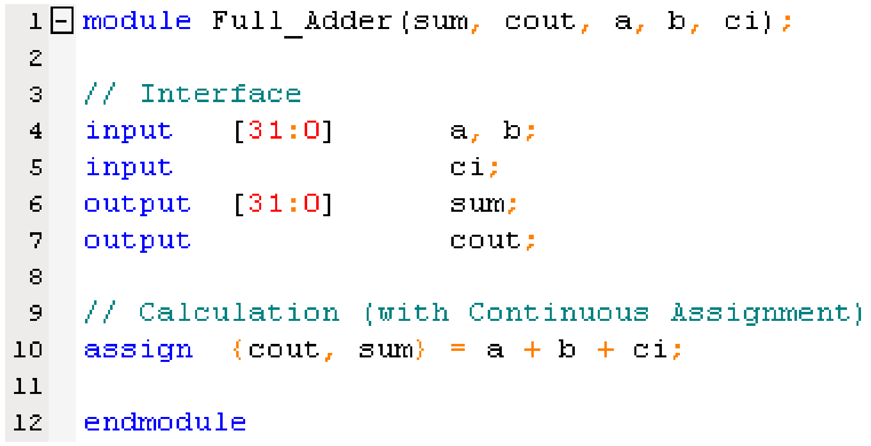
```
module Full_Adder(sum, cout, a, b, ci);

// Interface
input   [31:0]      a, b;
input               ci;
output  [31:0]      sum;
output              cout;

wire    [32:0]      temp;

// Calculation (with Continuous Assignment)
assign  temp = a + b + ci;
assign  sum  = temp[31:0];
assign  cout = temp[32];

endmodule
```

port declaration
data type declaration
module functionality

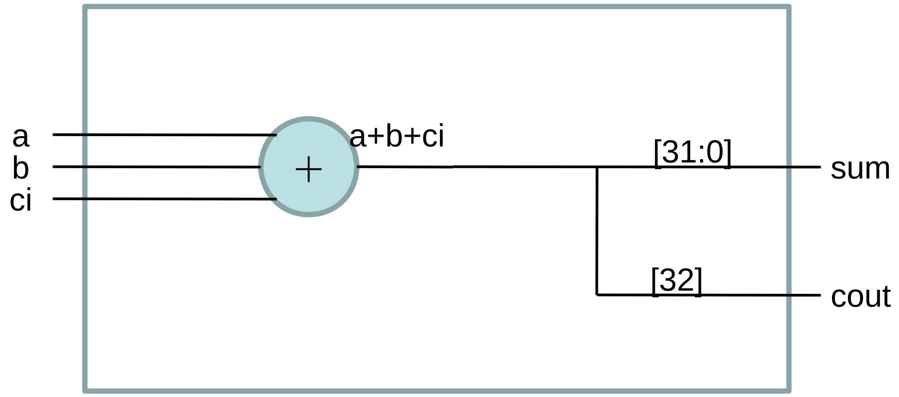


32-bit Full Adder: example 2

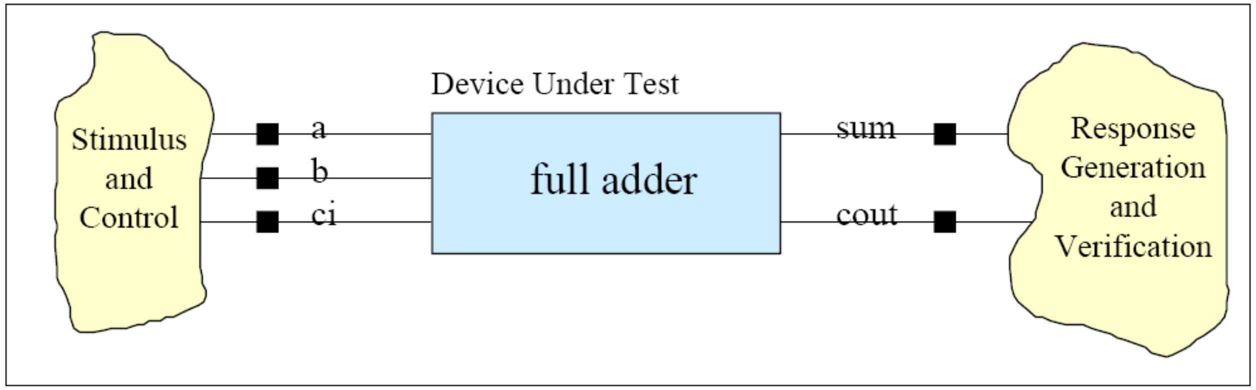


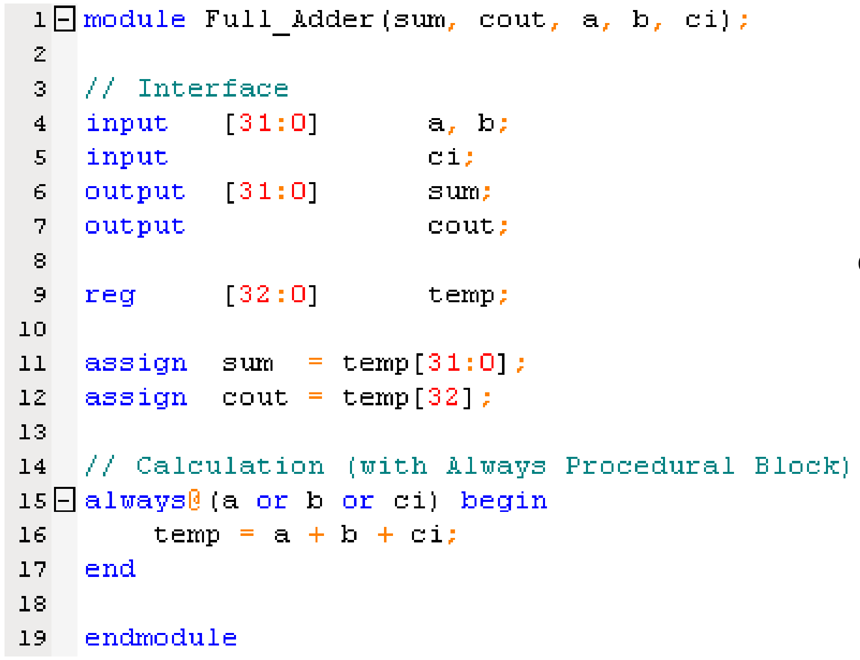


```verilog
module Full_Adder(sum, cout, a, b, ci);

// Interface
input   [31:0]      a, b;
input               ci;
output  [31:0]      sum;
output              cout;

// Calculation (with Continuous Assignment)
assign  {cout, sum} = a + b + ci;

endmodule
```

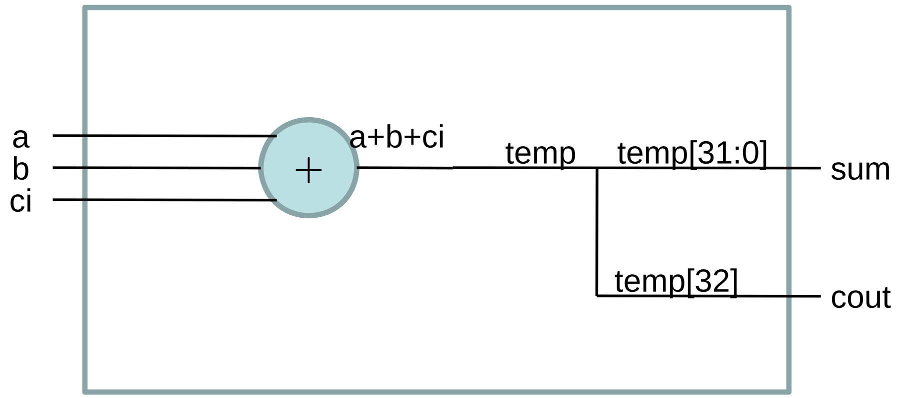


32-bit Full Adder: example 3



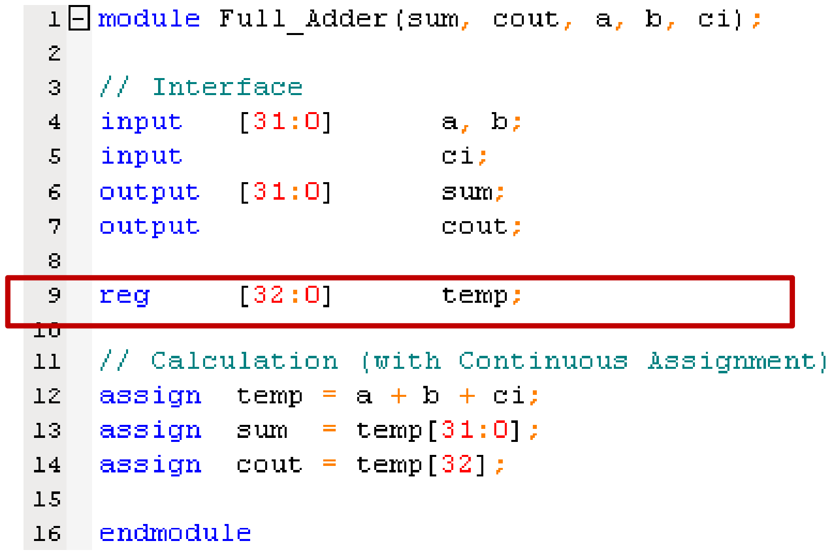


```verilog
module Full_Adder(sum, cout, a, b, ci);

// Interface
input   [31:0]      a, b;
input               ci;
output  [31:0]      sum;
output              cout;

reg     [32:0]      temp;

assign  sum  = temp[31:0];
assign  cout = temp[32];

// Calculation (with Always Procedural Block)
always@(a or b or ci) begin
    temp = a + b + ci;
end

endmodule
```



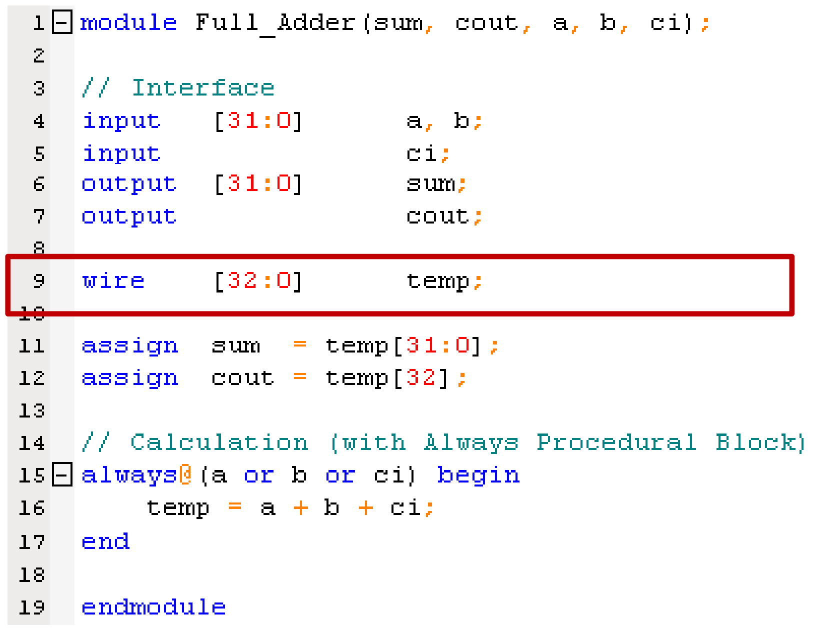
Wire & Register

- Can not use “reg” in left-hand side of continuous assignment.
- Can not use “wire” in left-hand side of assignment in procedural block.



```
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
```

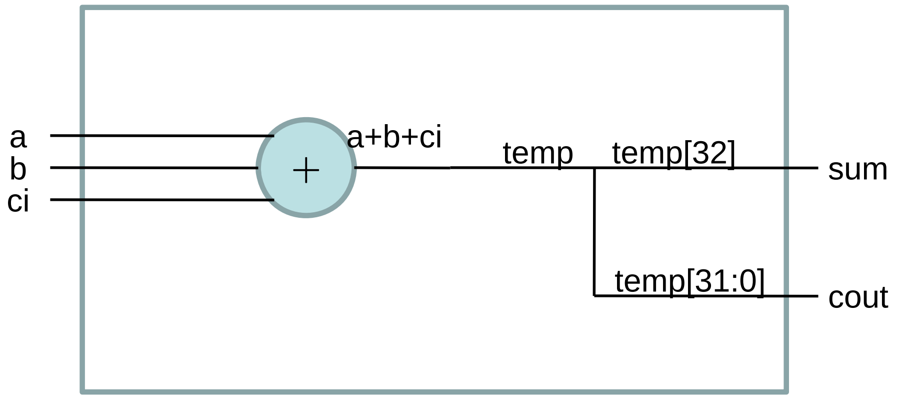
** Error: (12): Register is illegal in left-hand side of continuous assignment



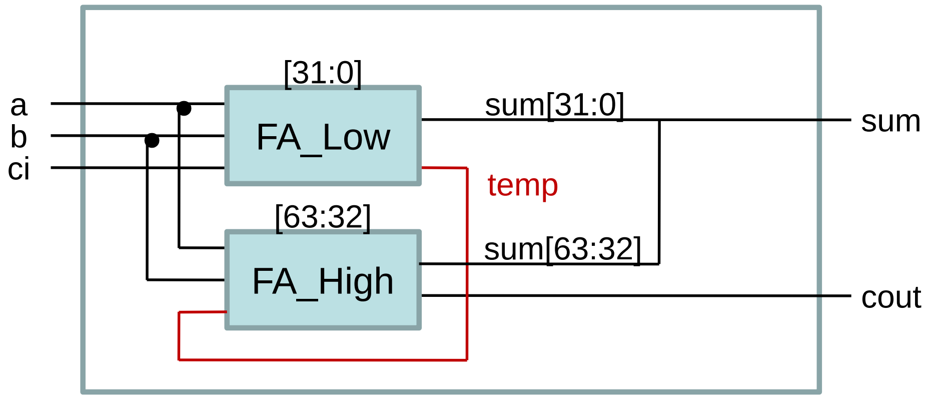
```
1  module Full_Adder(sum, cout, a, b, ci);
2
3  // Interface
4  input    [31:0]      a, b;
5  input                ci;
6  output   [31:0]      sum;
7  output               cout;
8
9  wire     [32:0]      temp;
10
11 assign  sum  = temp[31:0];
12 assign  cout = temp[32];
13
14 // Calculation (with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

64-bit Full Adder



Full_Adder

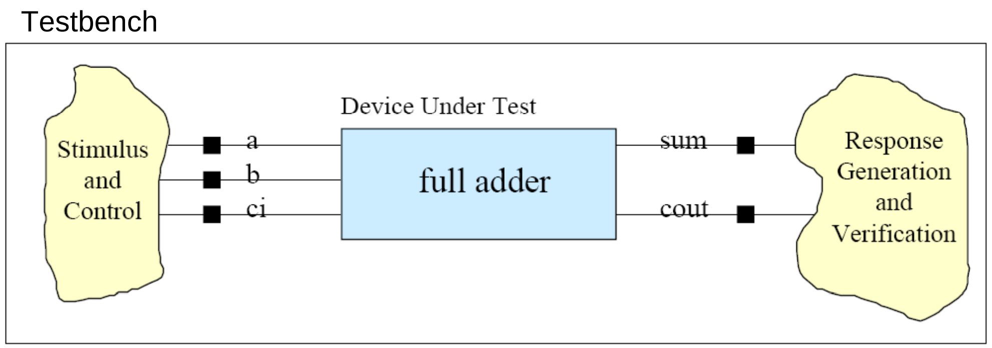


Full_Adder_64

```verilog
module Full_Adder(sum, cout, a, b, ci);

// Interface
input   [31:0]      a, b;
input               ci;
output  [31:0]      sum;
output              cout;

reg     [32:0]      temp;

assign  sum  = temp[31:0];
assign  cout = temp[32];

// Calculation (with Always Procedural Block)
always@(a or b or ci) begin
    temp = a + b + ci;
end

endmodule
```

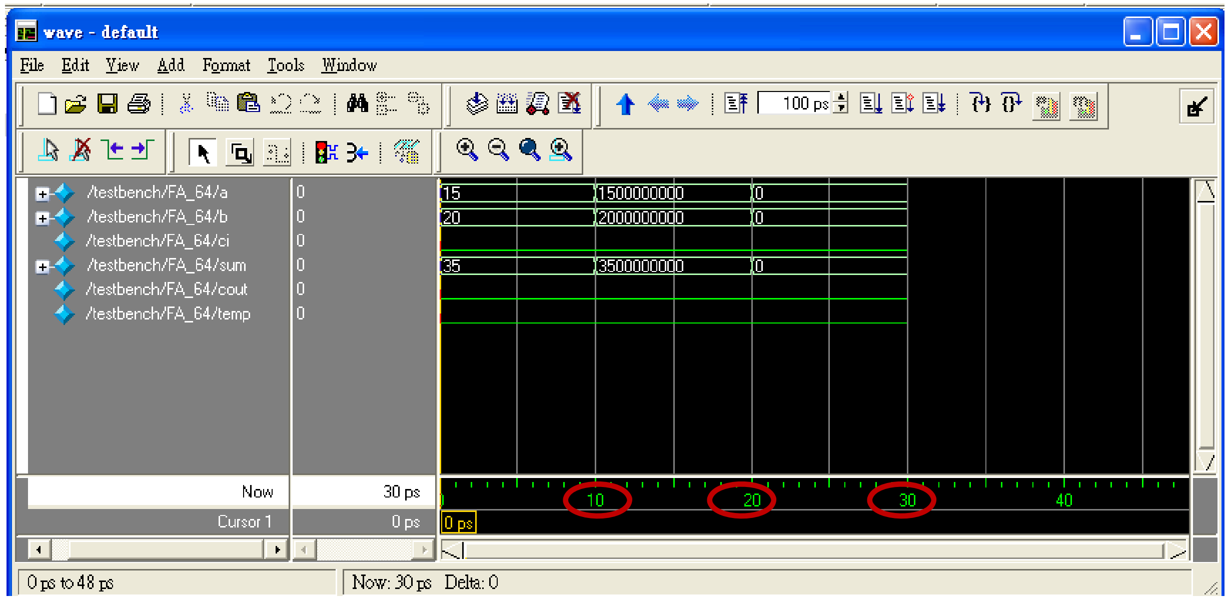
```verilog
module Full_Adder_64(sum, cout, a, b, ci);

// Interface
input   [63:0]      a, b;
input               ci;
output  [63:0]      sum;
output              cout;

wire                temp;

// Calculation (with 32-bit Full Adder Module)
Full_Adder       FA_Low (sum[31:0] , temp, a[31:0] , b[31:0] , ci  );
Full_Adder       FA_High(sum[63:32], cout, a[63:32], b[63:32], temp);

endmodule
```

Test Your 64-bit Full Adder Module



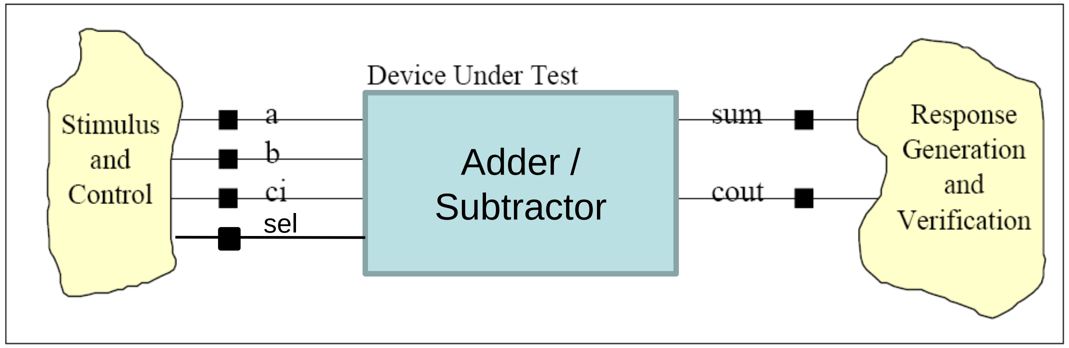
```verilog
module testbench();

reg    [63:0]        a, b;
reg                  ci;
wire   [63:0]        sum;
wire                 cout;

initial begin
   a = 15;
   b = 20;
   ci = 0;
   #10
   a = 1500000000;
   b = 2000000000;
   ci = 0;
   #10
   a = 0;
   b = 0;
   ci = 0;
   #10
   $stop();
end

// Add your modules here:
Full_Adder_64 FA_64(sum , cout, a, b, ci);

endmodule
```

Delay period

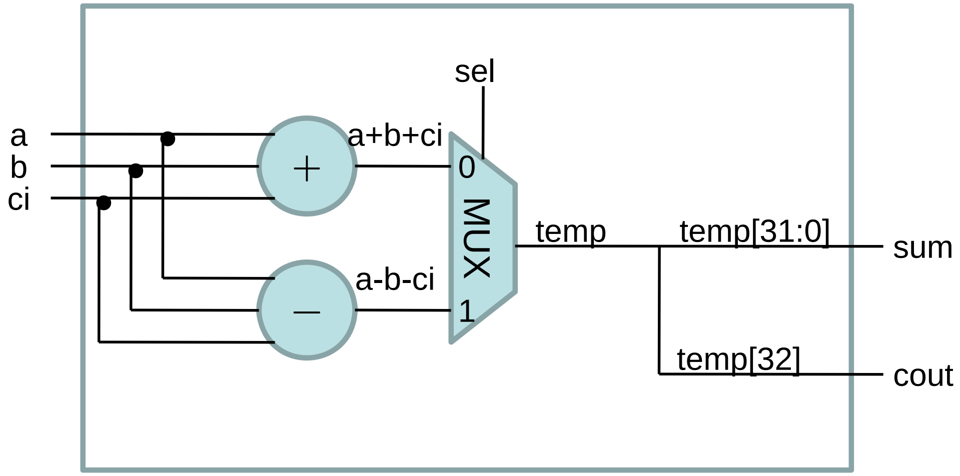


```
/testbench/FA_64/a      0    15   1500000000   0
/testbench/FA_64/b      0    20   2000000000   0
/testbench/FA_64/ci     0
/testbench/FA_64/sum    0    35   3500000000   0
/testbench/FA_64/cout   0
/testbench/FA_64/temp   0

Now        30 ps     10   20   30   40
Cursor 1    0 ps     0 ps

0 ps to 48 ps    Now: 30 ps  Delta: 0
```

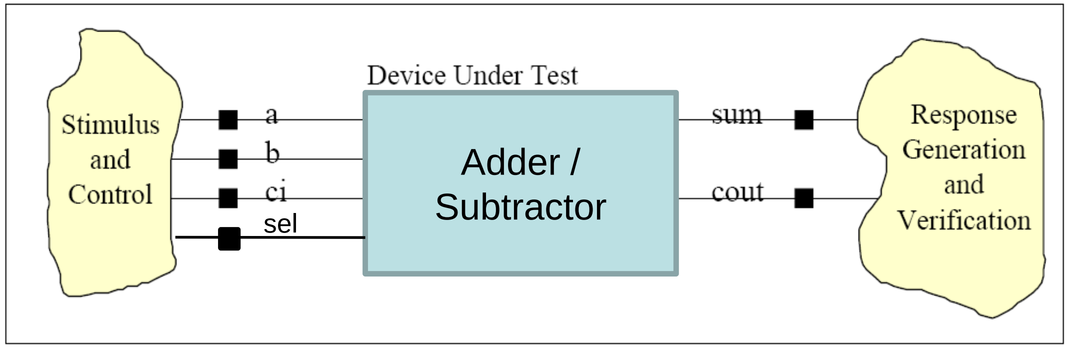
32-bit Adder Subtractor: example 1

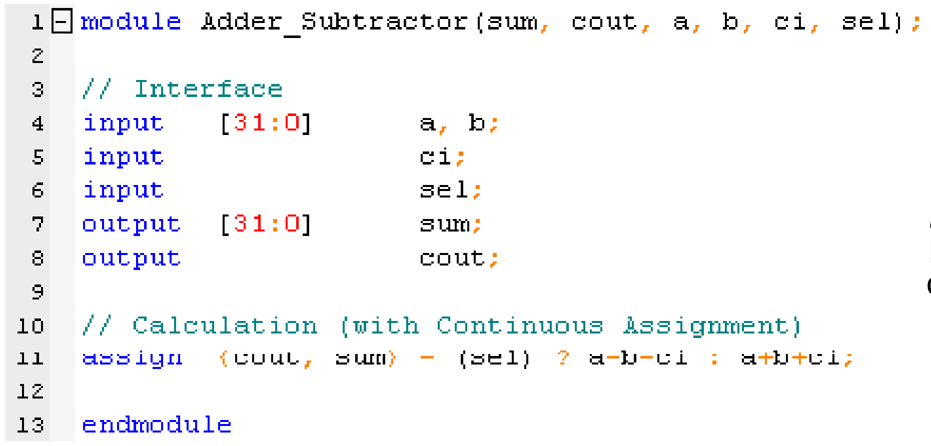


```verilog
module Adder_Subtractor(sum, cout, a, b, ci, sel);

// Interface
input    [31:0]       a, b;
input                 ci;
input                 sel;
output   [31:0]       sum;
output                cout;

reg    [32:0]         temp;

assign  sum  = temp[31:0];
assign  cout = temp[32];

// Calculation (with Always Procedural Block)
always@(a or b or ci or sel) begin
  if(sel) begin
    temp = a - b - ci;
  end
  else begin
    temp = a + b + ci;
  end
end

endmodule
```

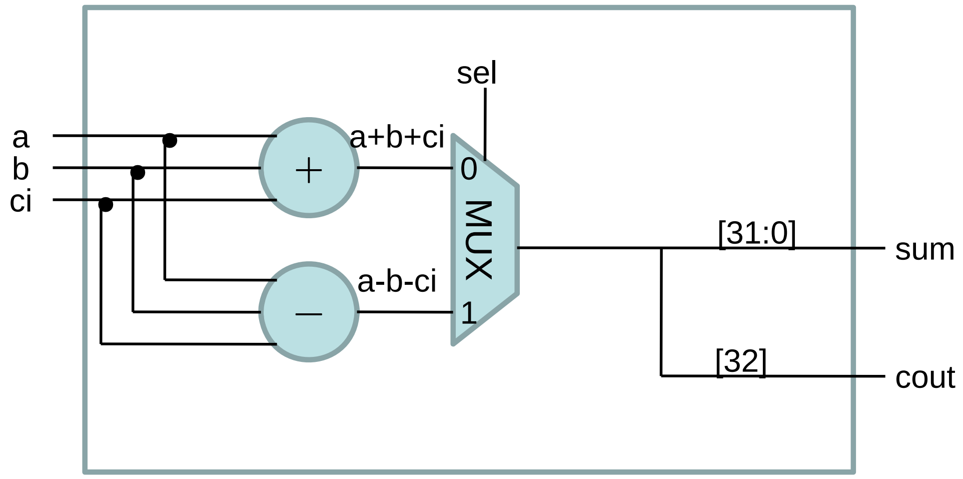


32-bit Adder Subtractor: example 2



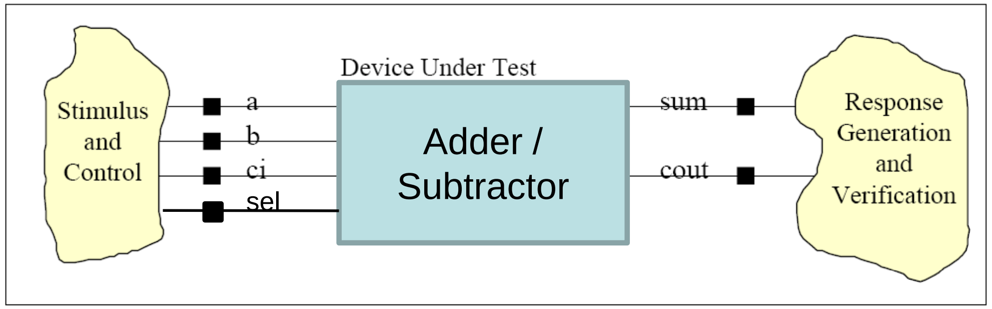


```verilog
module Adder_Subtractor(sum, cout, a, b, ci, sel);

// Interface
input   [31:0]      a, b;
input               ci;
input               sel;
output  [31:0]      sum;
output              cout;

// Calculation (with Continuous Assignment)
assign  {cout, sum} = (sel) ? a-b-ci : a+b+ci;

endmodule
```

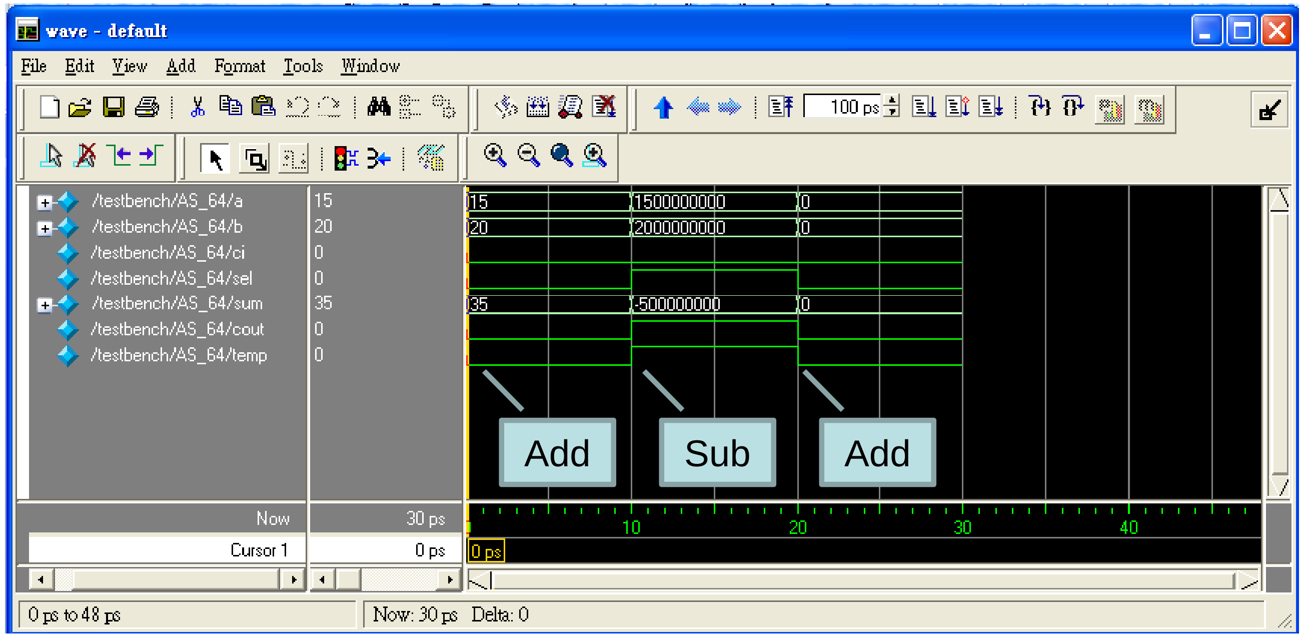


Test Your 64-bit Adder Subtractor Module

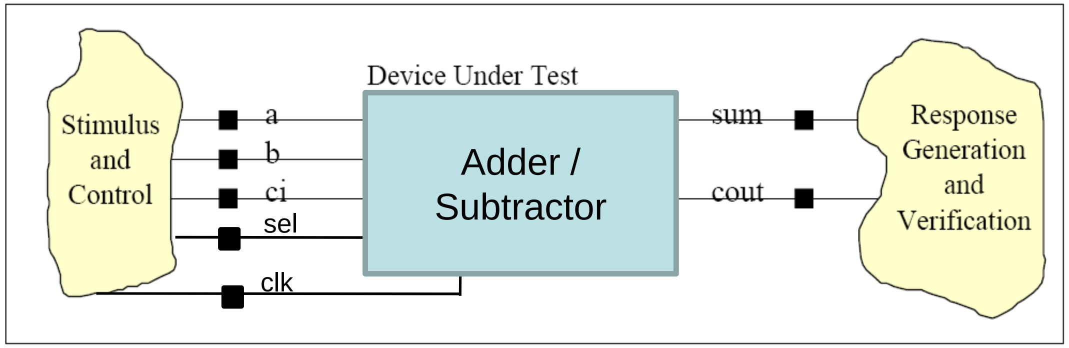
Testbench



```
module testbench();

reg    [63:0]        a, b;
reg                  ci;
reg                  sel;
wire   [63:0]        sum;
wire                 cout;

initial begin
   a = 15;
   b = 20;
   ci = 0;
   sel = 0;
   #10
   a = 1500000000;
   b = 2000000000;
   ci = 0;
   sel = 1;
   #10
   a = 0;
   b = 0;
   ci = 0;
   sel = 0;
   #10
   $stop();
end

// Add your modules here:
Adder_Subtractor_64 AS_64(sum , cout, a, b, ci, sel);

endmodule
```

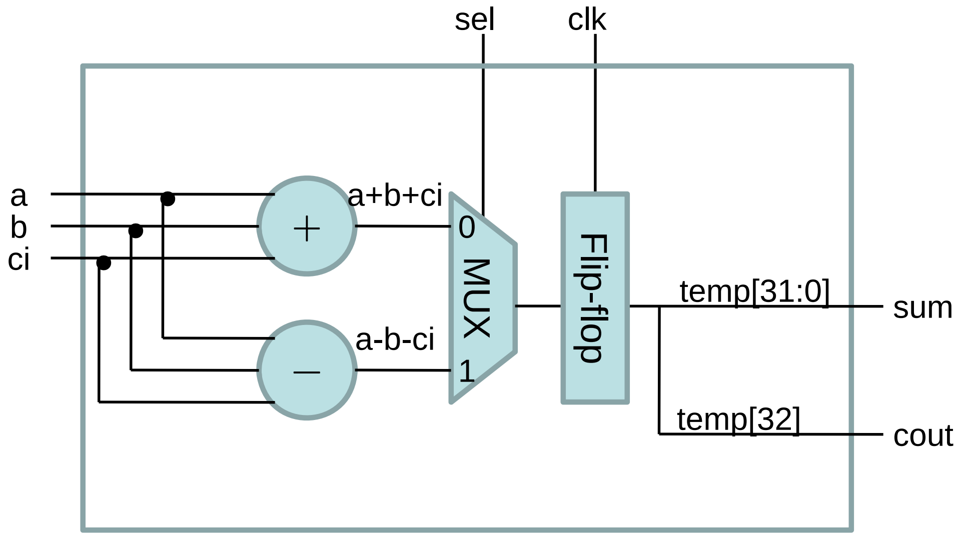


32-bit Adder Subtractor with Clock



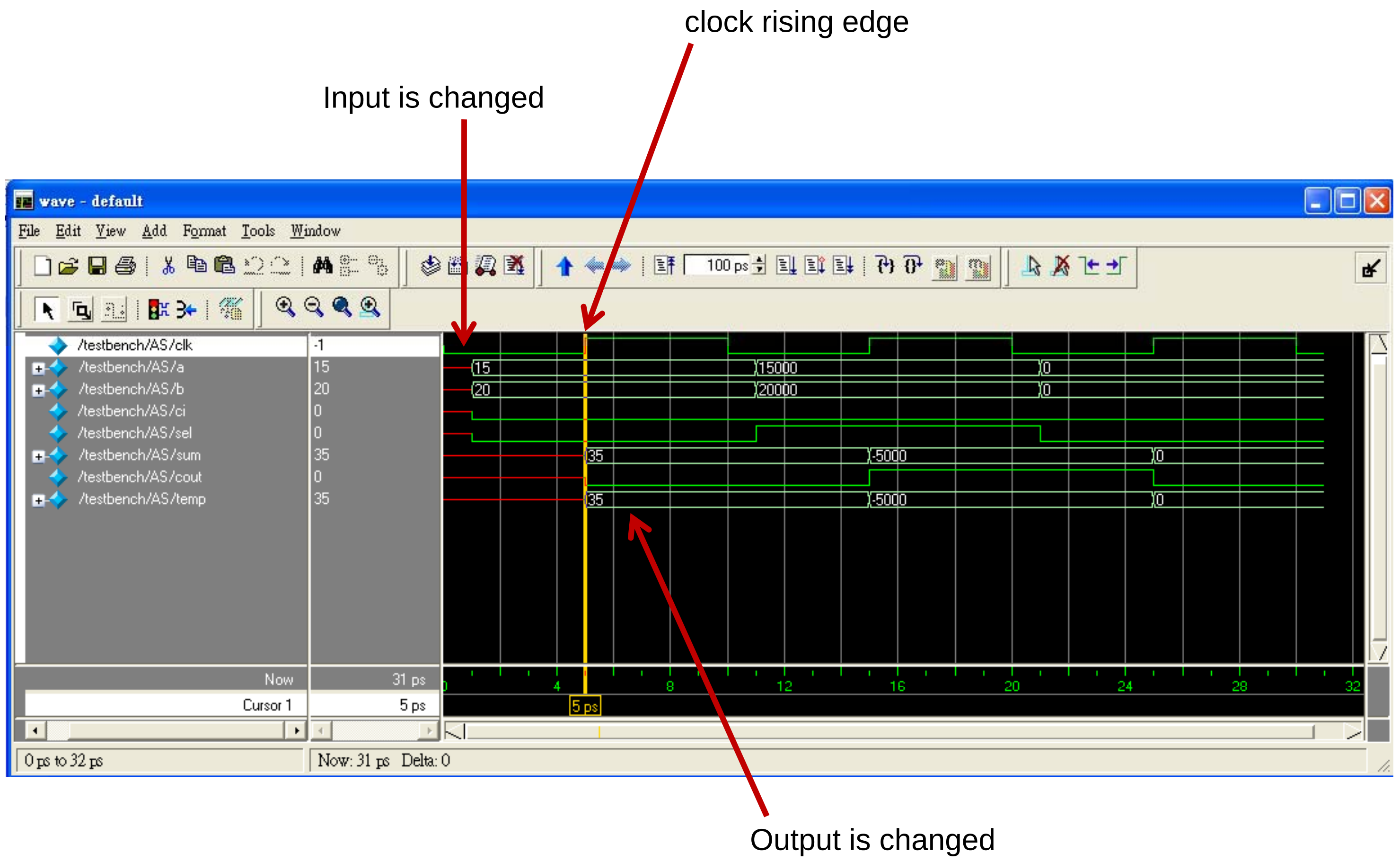
```verilog
module Adder_Subtractor(sum, cout, a, b, ci, sel, clk);

// Interface
input    [31:0]       a, b;
input                 ci;
input                 sel;
input                 clk;
output   [31:0]       sum;
output                cout;

reg    [32:0]         temp;

assign  sum  = temp[31:0];
assign  cout = temp[32];

// Calculation (with Always Procedural Block)
always@(posedge clk) begin
  if(sel) begin
    temp <= a - b - ci;
  end
  else begin
    temp <= a + b + ci;
  end
end

endmodule
```

Use non-blocking assignment !



Test 32-bit Adder Subtractor with Clock

```
module testbench();

reg    [31:0]       a, b;
reg                 ci;
reg                 sel;
reg                 clk;
wire   [31:0]       sum;
wire                cout;

always #5 clk = ~clk;

initial begin
  clk = 0;
  #1
  a = 15;
  b = 20;
  ci = 0;
  sel = 0;
  #10
  a = 15000;
  b = 20000;
  ci = 0;
  sel = 1;
  #10
  a = 0;
  b = 0;
  ci = 0;
  sel = 0;
  #10
  $stop();
end

// Add your modules here:
Adder_Subtractor  AS(sum , cout, a, b, ci, sel, clk);

endmodule
```



```
/testbench/AS/clk    -1
/testbench/AS/a      15     15      15000   0
/testbench/AS/b      20     20      20000   0
/testbench/AS/ci     0
/testbench/AS/sel    0
/testbench/AS/sum    35     35      -5000   0
/testbench/AS/cout   0
/testbench/AS/temp   35     35      -5000   0

Now       31 ps
Cursor 1  5 ps
0 ps to 32 ps    Now: 31 ps  Delta: 0
```

Outline

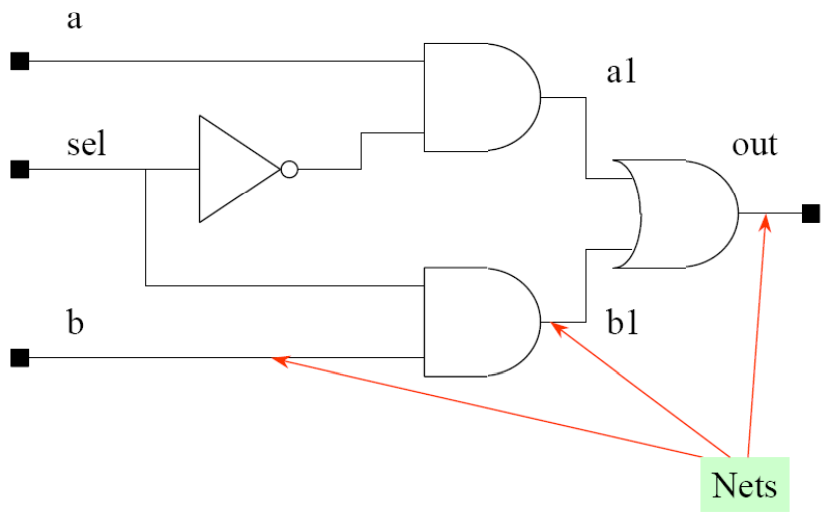
- Verilog & Examples
- Major Data Type
- Operations
- Behavior Modeling
- Structure Modeling

Major Data Type

- Nets (Wires)
- Registers
- Parameters

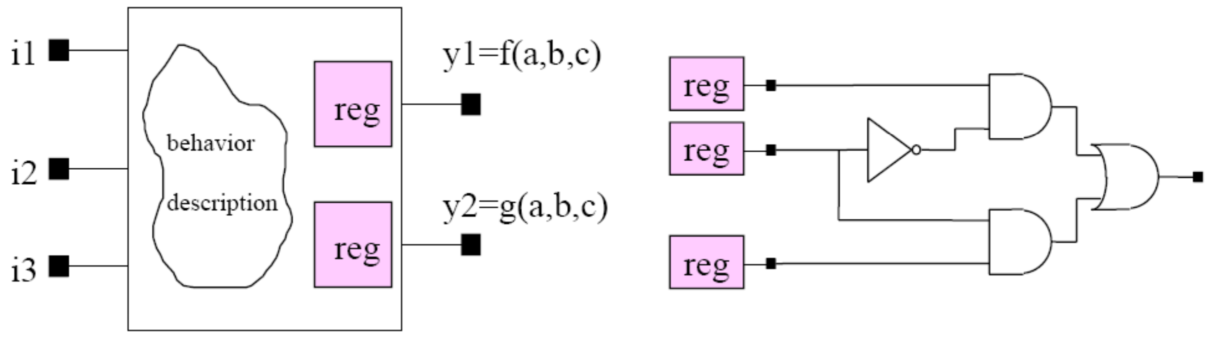
Nets

- *Net* data type represent physical connections between structural entities.
- A *net* must be driven by a driver, such as a gate or a continuous assignment.
- Verilog automatically propagates new values onto a net when the drivers change value.



Registers

- Registers represent abstract storage elements.
- A register holds its value until a new value is assigned to it.
- Registers are used extensively in behavior modeling.



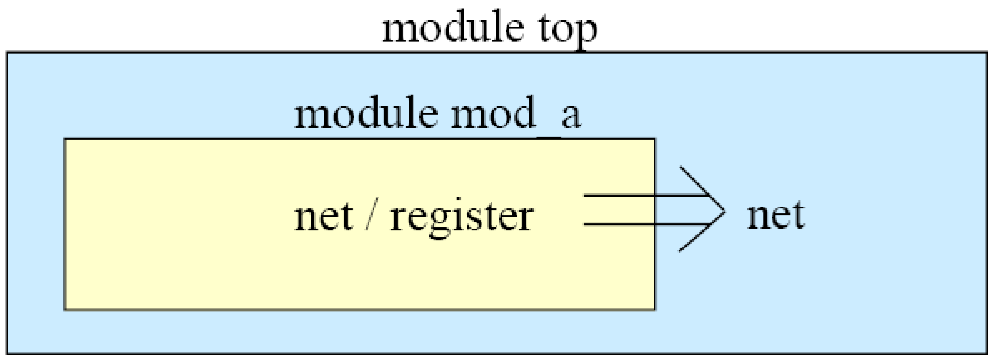
Common Mistake in Choosing Data Type

- An output port can be driven by a net or a register, but it can only drive a net.

Example

```
module top (...);
  ....
  reg  rega, regb;

  mod_a U1(rega,  regb);
......
```

instance *U1*'s output port connect to a register *rega*.



Parameters

- Parameters are not variables, they are constants.
- Typically parameters are used to specify delays and width of variables.

```
module var_mux(out, i0, i1, sel) ;
  parameter width = 2, delay = 1 ;
  output  [width-1:0] out ;
  input   [width-1:0] i0, i1 ;
  input   sel;

  assign #delay out = sel  ?  i1  : i0 ;
endmodule
```

- if *sel* = 1, then *i1* will be assigned to *out*;
- if *sel* = 0, then *i0* will be assigned to *out*;

Integer & Real Numbers

16 --- 32 bits decimal

8’d16

8’h10

8’b0001_0000

8’o20

32’bx --- 32 bits x

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

6.3

5.3e-4

6.2E3

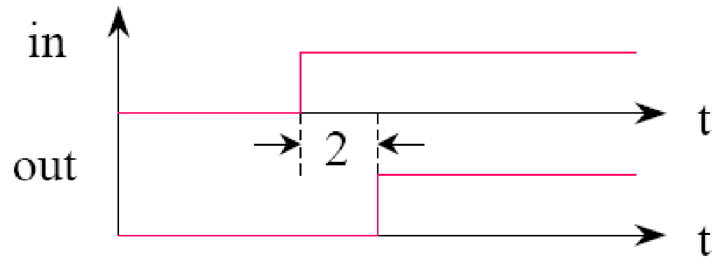
Outline

- Verilog & Examples
- Major Data Type
- Operations
- Behavior Modeling
- Structure Modeling

Continuous Assignments

- Any changes in the RHS of the continuous assignment are evaluated and the LHS is updates.

```
assign #2 out = in;
```



- RHS can be
  - Expression
    - assign and_out = i1 & i2;
  - Value
    - assign net_1 = 1;
  - Other net
    - assign net_a = net_b;

Example

continuous assignment

```
module inv_array(out, in);
   output [31:0] out;
   input   [31:0] in;

   assign out = ~in;
endmodule
```

gate-level modeling

```
module inv_array(out, in);
   output [31:0] out;
   input   [31:0] in;

   not  G0(out[0], in[0]);
   ......
   not  G31(out[31], in[31]);
endmodule
```

Operations

arithmetic operator

operator	operation
+	arithmetic addition
-	arithmetic substraction
*	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 preform 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 ……..

Shift Operator

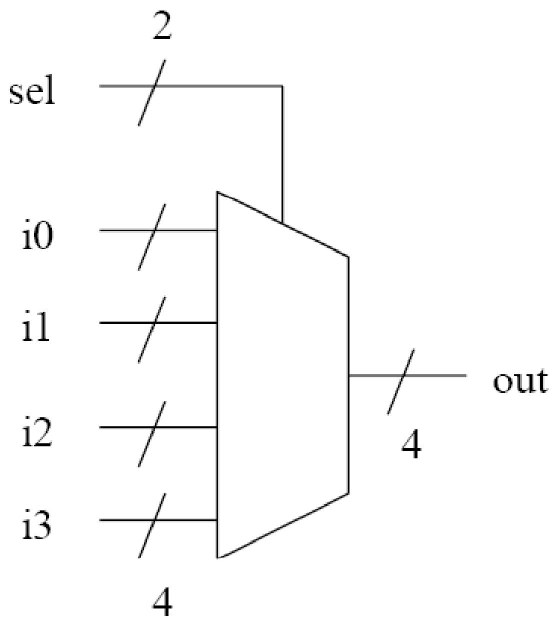
```
module shift_register(reg_out, reg_in);
  output [5:0] reg_out;
  input   [5:0] reg_in;

  parameter shift = 3;
    assign reg_out = reg_in << shift;
endmodule
```

examples :

```
reg_in = 6'b011100
reg_in << 3  →  100000
reg_in >> 3  →  000011
```

Conditional Operator

```
module MUX4_1(out, i0, i1, i2, i3, sel);
   output [3:0] out;
   input   [3:0] i0, i1, i2, i3;
   input   [1:0] sel;

   assign out = (sel == 2'b00) ? i0 :
                      (sel == 2'b01) ? i1 :
                      (sel == 2'b01) ? i2 :
                      (sel == 2'b01) ? i3 :
                      4'bx;
endmodule
```



Concatenation & Replication Operator

- **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}};
```

- **Sign Extension**

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

Outline

- Verilog & Examples
- Major Data Type
- Operations
- Behavior Modeling
- Structure Modeling

Behavior Modeling

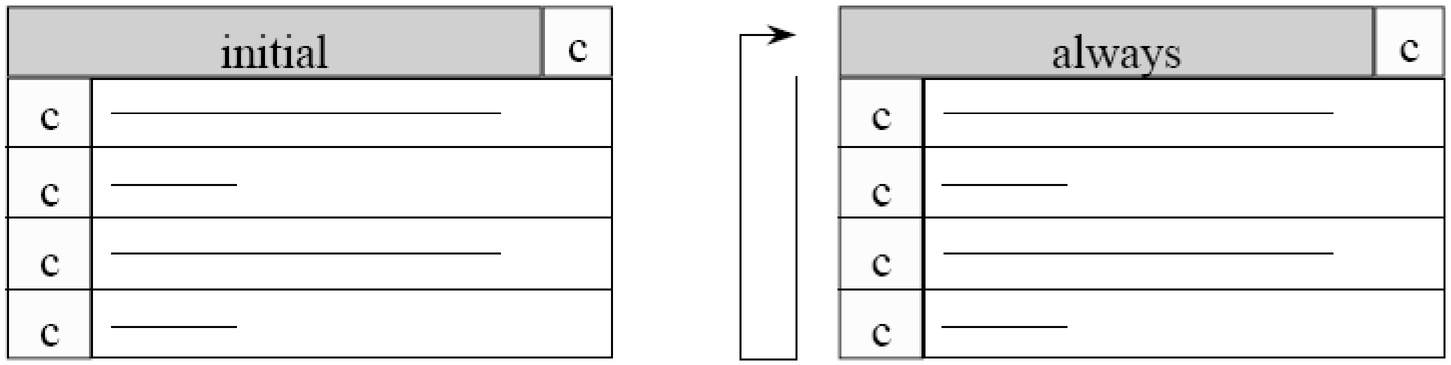
- At system level, system’s functional view is more important than implementation.
  - You do not have any idea about how to implement your netlist.
  - The data flow of this system is analyzed,
  - You may need to explore different design options.
- Behavior modeling enables you to describe the system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

Behavior Modeling

- In behavior modeling, you must describe your circuits'…
  - Action
    - How do you model your circuit's behaviors?
  - Timing control
    - At what time do what thing.
    - At what condition do what thing.
- Verilog supports the following constructs to model circuits' behavior
  - Procedural block
  - Procedural assignment
  - Timing control
  - Control statement

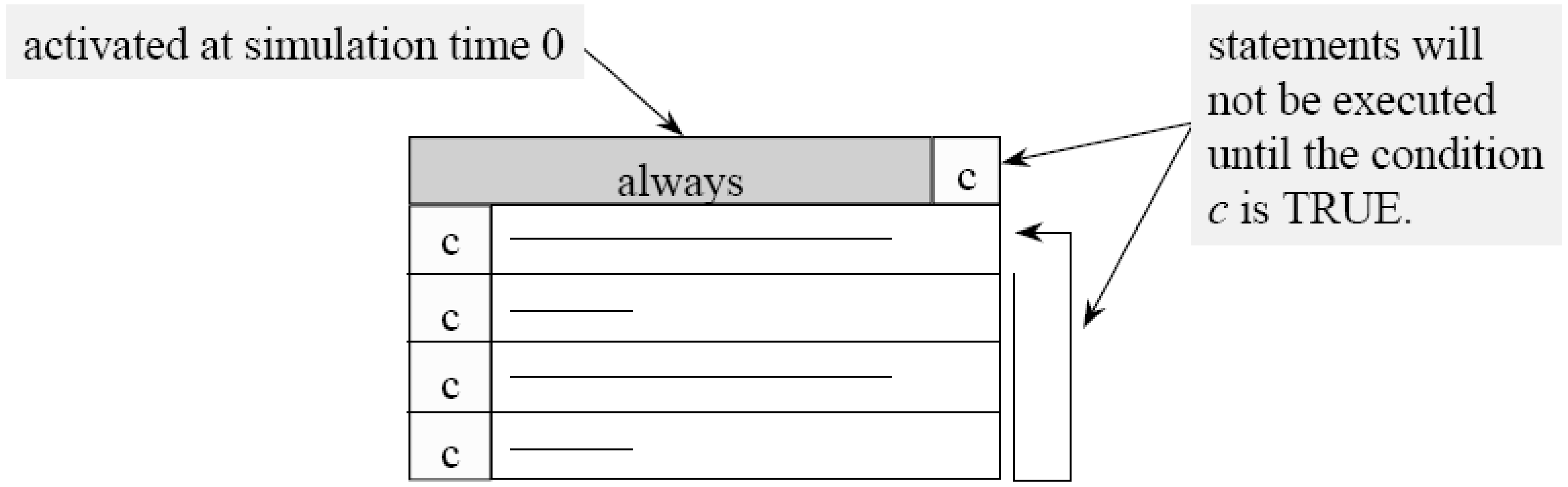
Procedural Blocks

- In Verilog, procedural blocks are the basic of behavior modeling.
  - You can describe one behavior in one procedural block
- Procedural blocks are of two types
  - Initial procedural block
    - Which execute only once
  - Always procedural block
    - Which execute in a loop



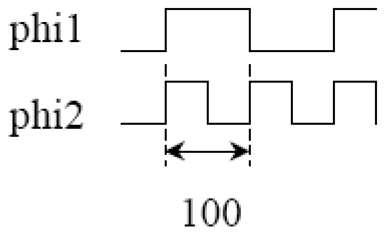
Procedural Blocks (Cont.)

- All procedural blocks are activated at simulation time 0
  - With enabling condition, the block will not ne execute until the enabling condition evaluates to TRUE
  - Without enabling condition, the block will be executed immediately.


activated at simulation time 0
always
c
c
c
c
c
statements will not be executed until the condition *c* is TRUE.

Procedural Blocks (Cont.)

```
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
```

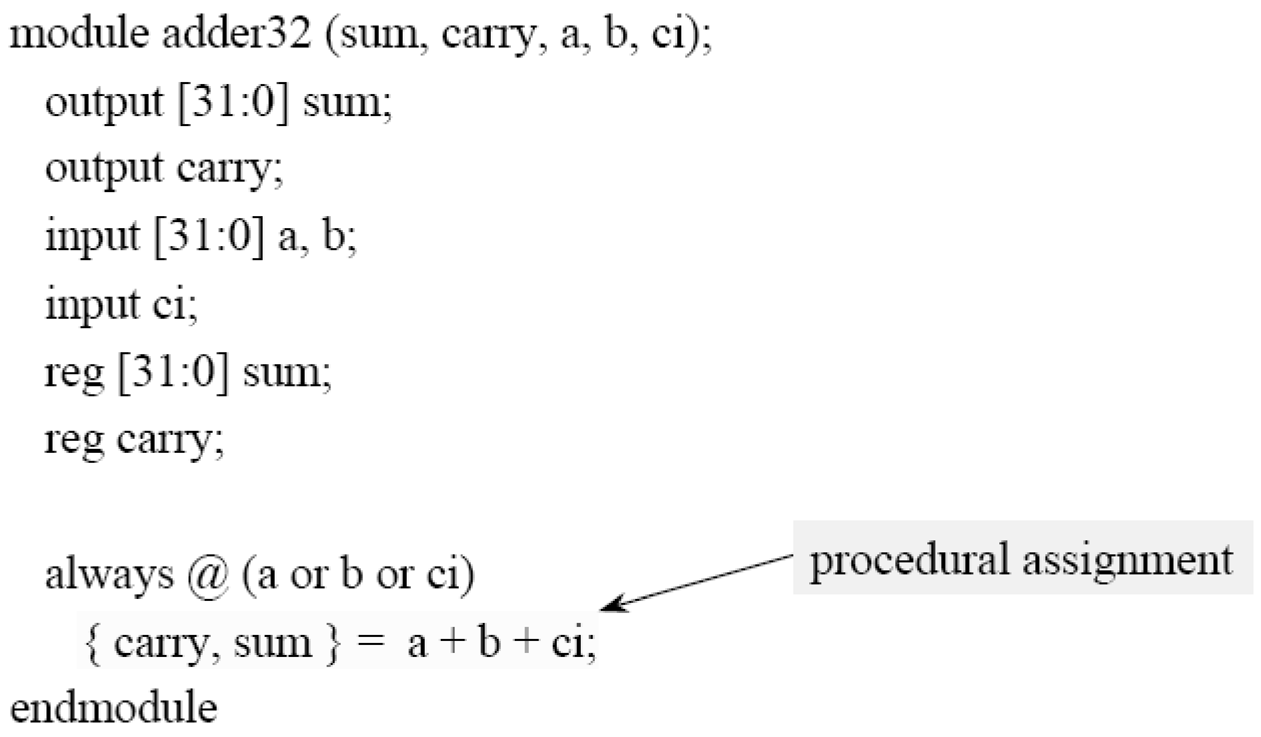


These procedural blocks are activated and executed at simulation time 0.

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

Procedural Assignment

- Procedural assignments drive values or expressions onto registers.

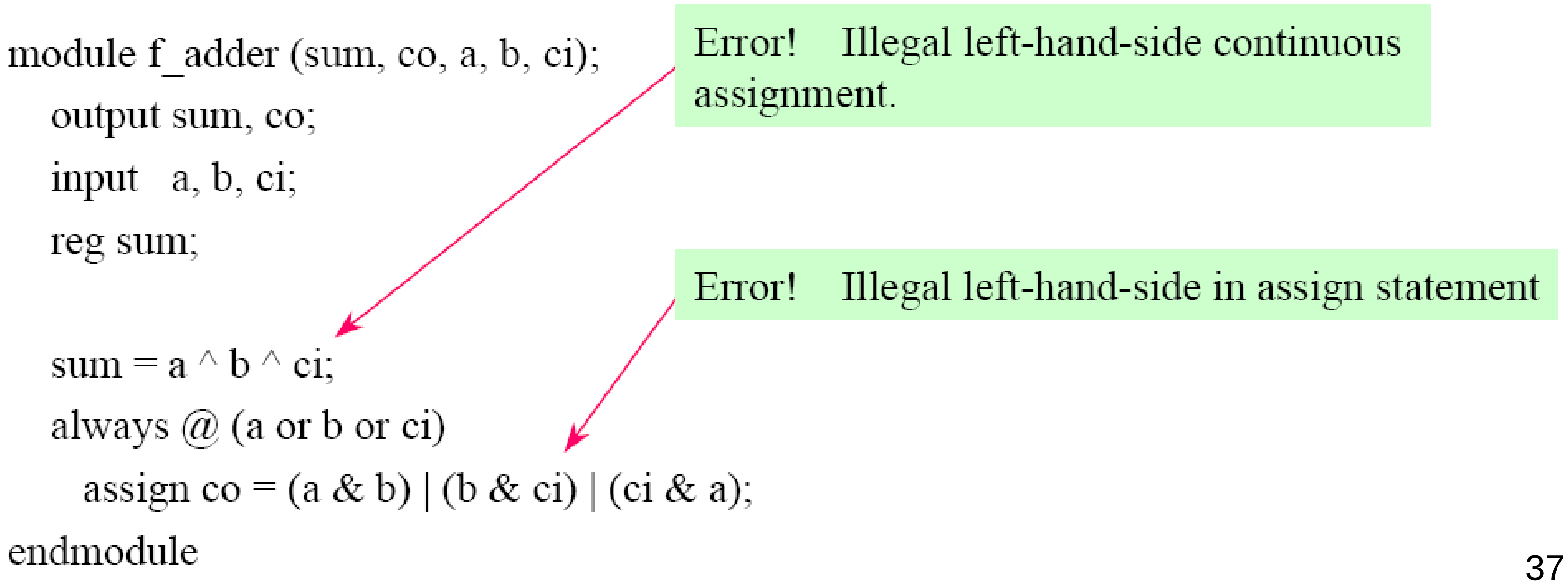


```
module adder32 (sum, carry, a, b, ci);
  output [31:0] sum;
  output carry;
  input [31:0] a, b;
  input ci;
  reg [31:0] sum;
  reg carry;

  always @ (a or b or ci)
    { carry, sum } =  a + b + ci;
endmodule
```

procedural assignment

Procedural Assignments

- A continuous assignment statement **cannont** be inside procedural blocks.
- A procedural assignment statement **must** be inside procedural blocks.



```
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

Non-blocking Procedural Assignment

- Blocking procedural assignment

```
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 procedural assignment

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

Shift register
```
 x = a
 y = x_old
 z = y_old
```

Conditional Statements

- If and If-Else Statement

```
if (expression)
   statement
else
   statement
```

Example

```
if (rega >= regb)
   result = 1;
else
   result = 0;
```

```
if (expression)
   statement
else if (expression)
   statement
else
   statement
```

```
if (index > 0)
   if (rega > regb)
      result = rega;
   else
      result = 0;
else
   $display("* Warning * index is equal or small than 0!");
```

Conditional Statements

- Case Statement

```
`define  pass_accum  4'b0000
`define  pass_data   4'b0001
`define  ADD         4'b0010
```

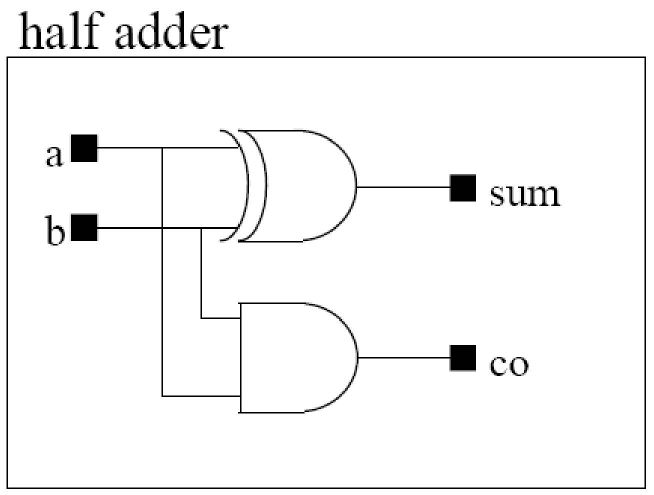
```
case (opcode)
  `pass_accum : #3.5 alu_out = accum;
  `pass_data  : #3.5 alu_out = data;
  `ADD        : #3.5 alu_out = accum + data;
  `AND        : #3.5 alu_out = accum & data;
  `XOR        : #3.5 alu_out = accum ^ data;
  default     : #3.5 alu_out = 8'bx;
endcase
```

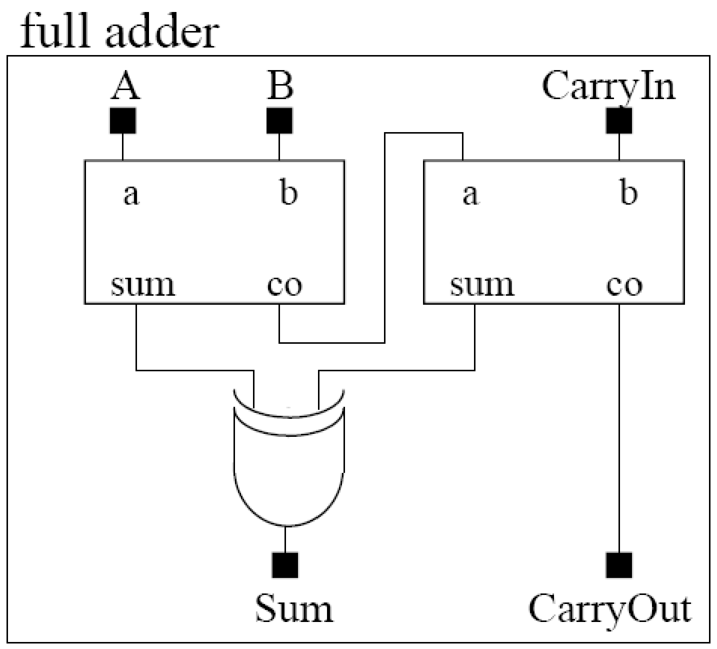
Outline

- Verilog & Examples
- Major Data Type
- Operations
- Behavior Modeling
- Structure Modeling

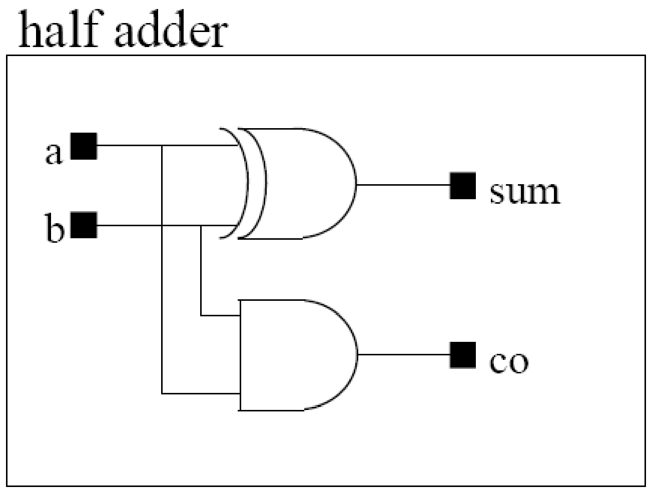
Structure Modeling

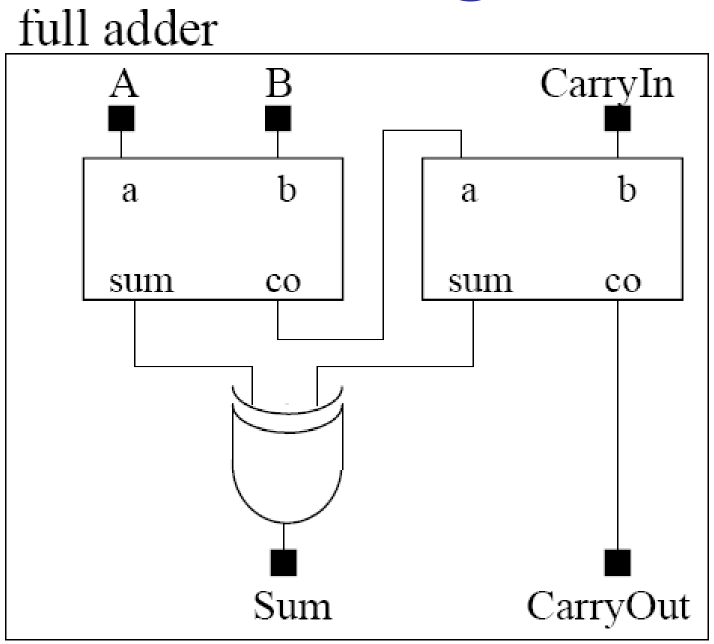
- In structural modeling, you connect components with each other to create a more complex component.





Structure Modeling





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

HA.v

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

FA.v