N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2010

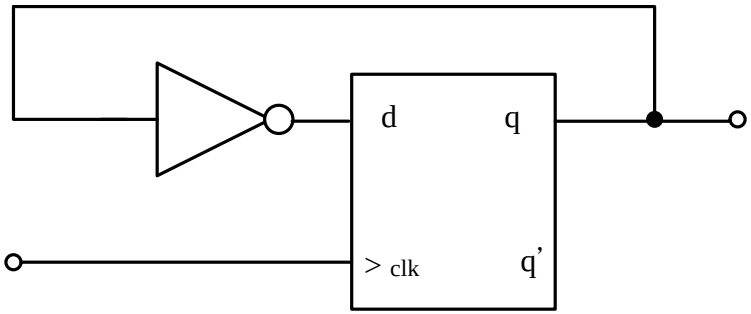
Verilog Basics

Outline

- Concepts of RTL
- First look of a simple Verilog code
- Testbench Template
- Basic elements of Verilog language
- Hierarchical example – 4-bit Ripple Carry Adder
- Parameter, Time scale, Text inclusion, Text substitution
- Special Language Tokens

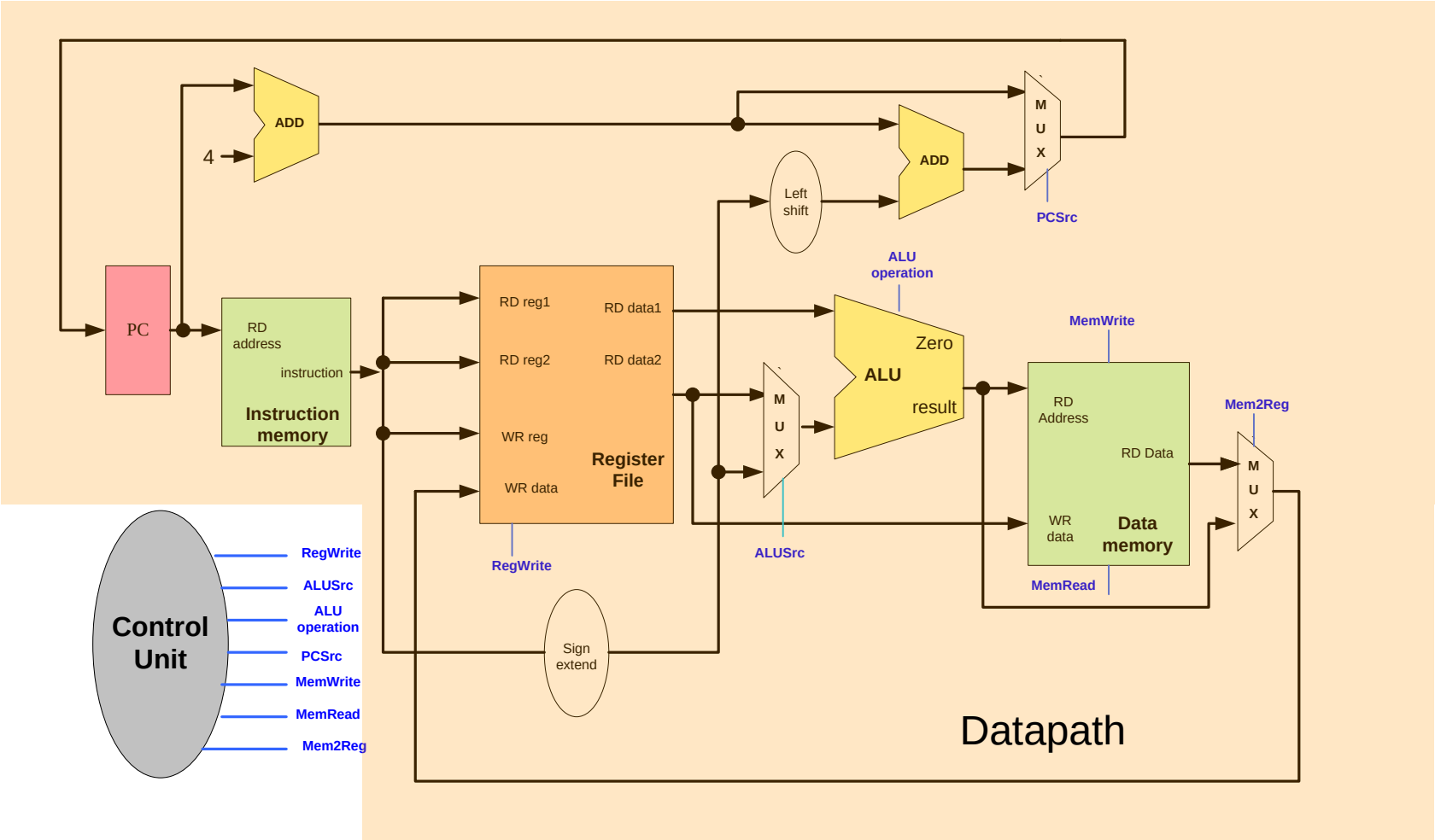
Register-Transfer Level

- RTL description is a way of describing the operation of a synchronous machine.
- The behavior of a machine is defined by the flow of signals (or transfer of data) between hardware registers and the logical operations (+,-, not,….)



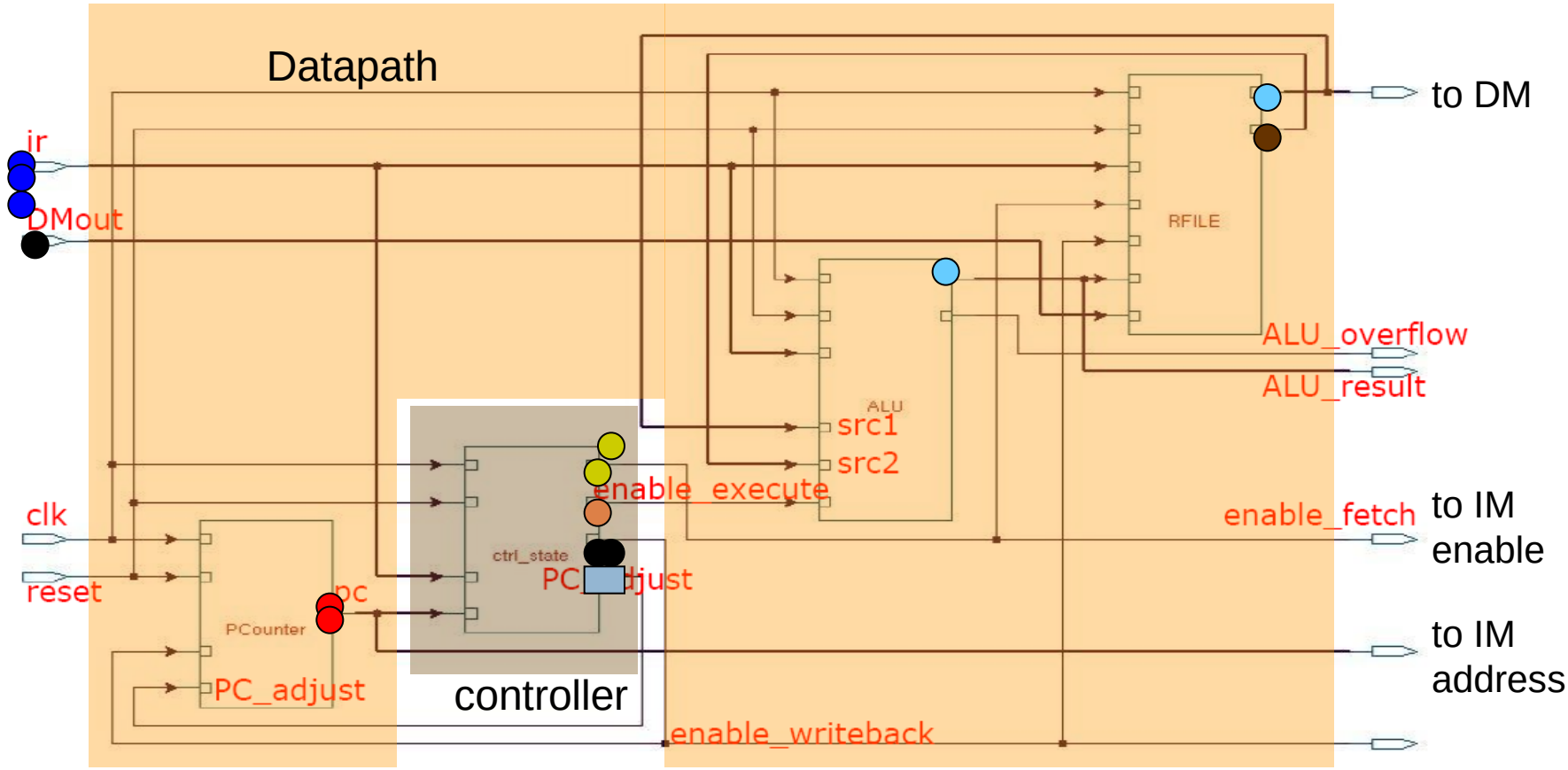
A simple RTL design

A Complex RTL Design -- CPU

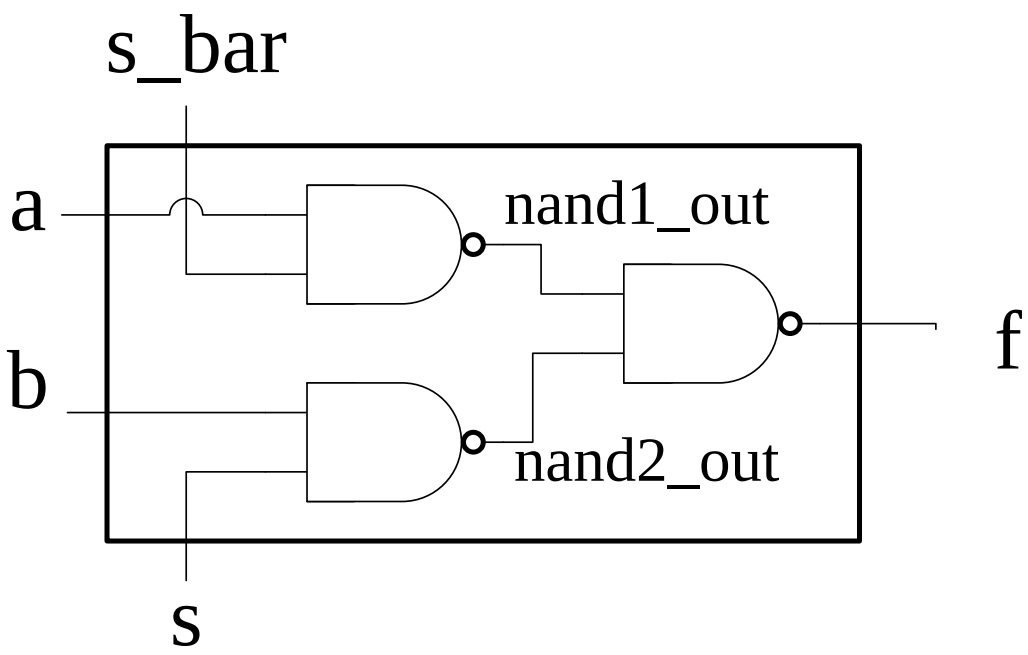


CPU in Action (ALU REG <- REG)

- RF[dst] <- ALU(RF(src),RF(dst))



One-bit Multiplexer



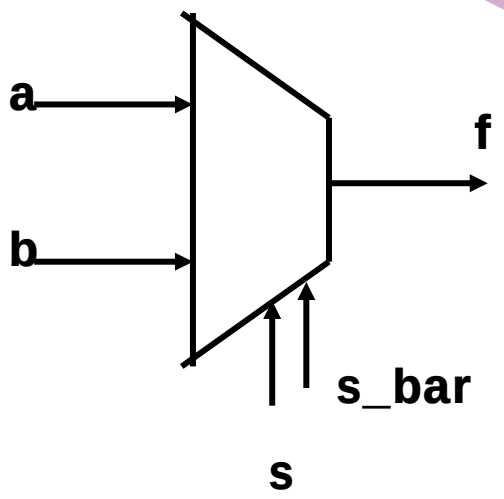
First Look at Verilog (1/2)

Main frame

Variable declaration

Main body

```
module mux2x1(f, s, s_bar, a, b);
    output  f;
    input  s, s_bar;
    input  a, b;
    wire  nand1_out, nand2_out;
    // boolean function
    nand(nand1_out, s_bar, a);
    nand(nand2_out, s, b);
    nand(f, nand1_out, nand2_out);
endmodule
```



First Look at Verilog (2/2)

Variable declaration

output

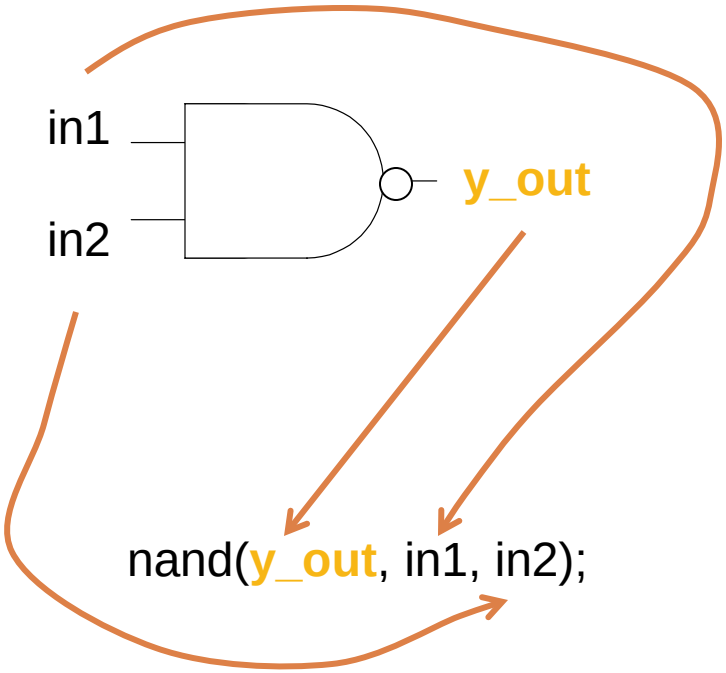
Input

wire

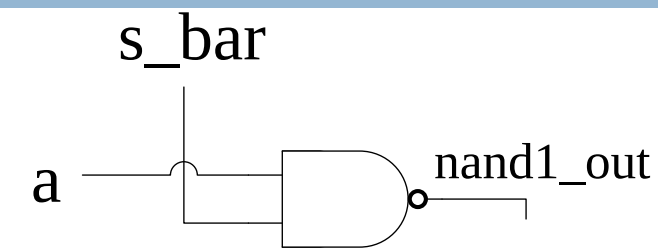
Main body

```
module mux2x1(f, s, s_bar, a, b);
    output  f;
    input  s, s_bar;
    input  a, b;
    wire  nand1_out, nand2_out;
    // boolean function
    nand(nand1_out, s_bar, a);
    nand(nand2_out, s, b);
    nand(f, nand1_out, nand2_out);
endmodule
```

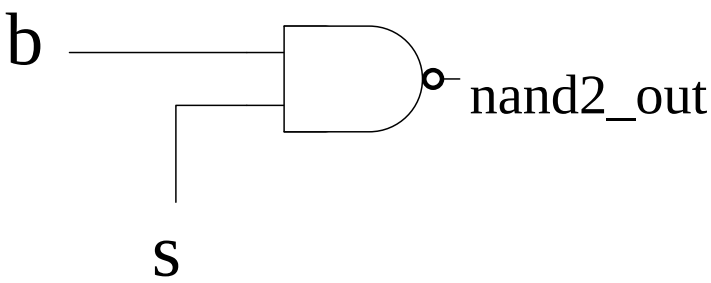
Mapping between a NAND gate and its Verilog nand



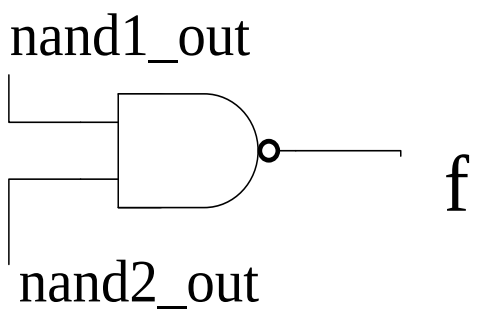
NANDs with different input names and output names



nand(**nand1_out**, a, s_bar);

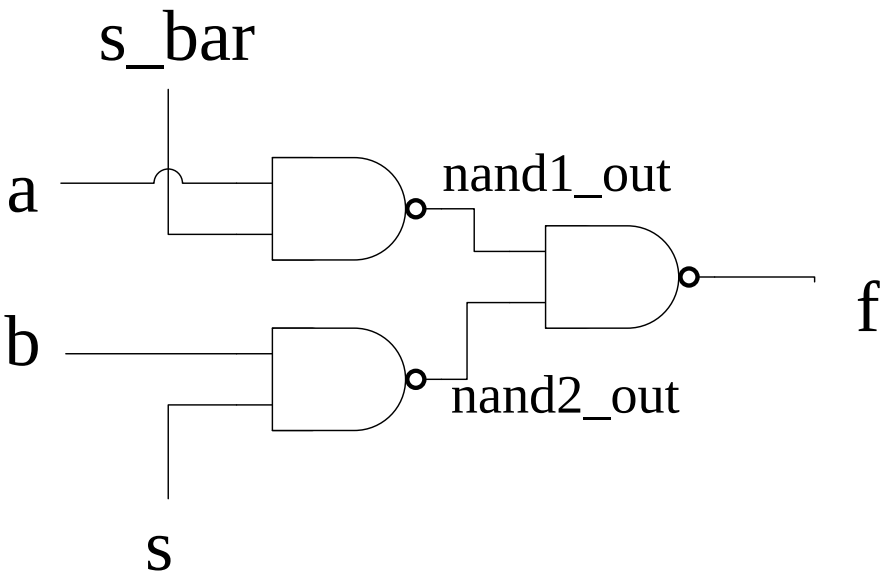


nand(**nand2_out**, s, b);



nand(**f**, nand2_out, nand1_out);

Putting Them All Together




```
nand(nand1_out, a, s_bar);
nand(nand2_out, s, b);
nand(f, nand2_out, nand1_out);
```

Basic Language Rules

- General
  - A statement shall terminate with semicolon (;), except *endmodule*
  - Comments: (//) for single line and {/*, */} for multiple lines
- Naming
- Number representation
- Primitive operators
- First examples

Naming

- Case sensitive
  - C_out_bar and C_OUT_BAR: two different identifiers
- NO whitespace
- Accepted chars
  - Lower/upper case letters
  - Digits (0,1,…,9)
  - Underscore (_)
  - Dollar sign ($)
  - Max characters: 1024

Number Representation

- May be represented using
  - Binary, octonary, decimal, hexadecimal,
- Format
  - <size>' <base_format> <number>
  - base_format:
    - b, o, d, h
- Example
  - 4'b1111; -16'd255
  - 23456 (32-bit decimal # by default); 'hc3 (32 bit)
  - 12'b1111_0000_1010

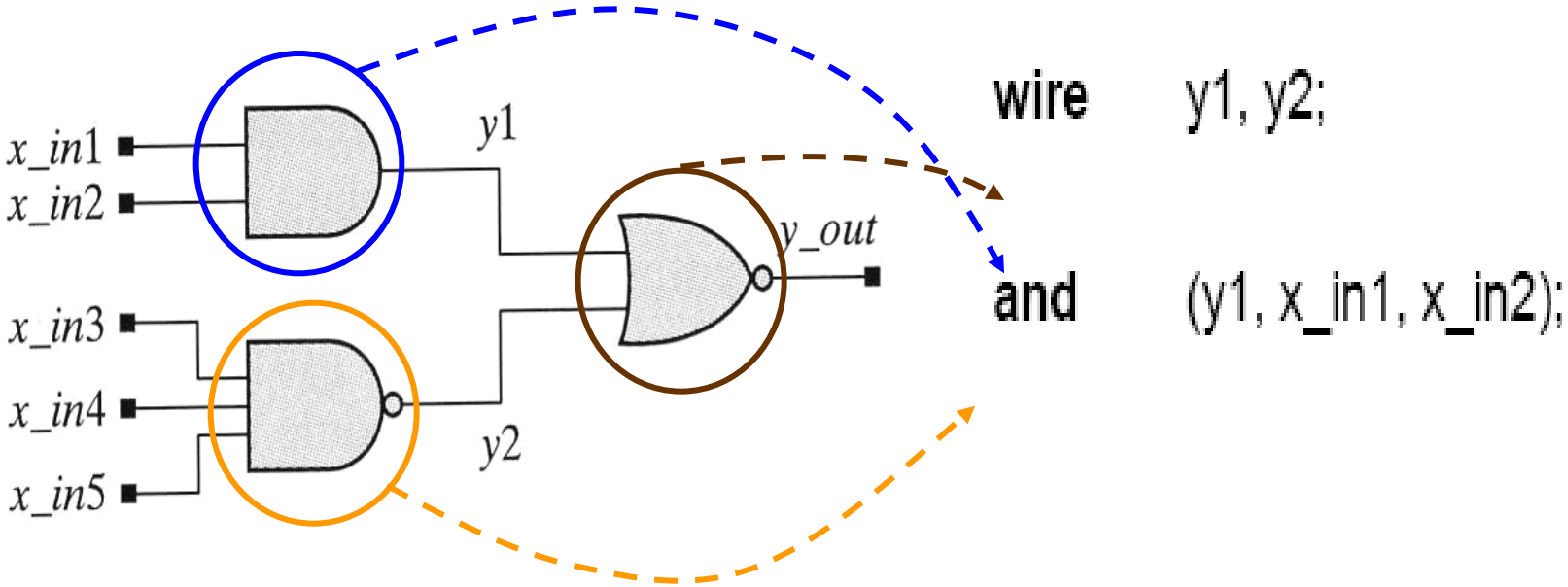
Primitive Operator

- Most basic functional models of combinational logic gates
- Built in the Verilog

TABLE 4-1 Verilog primitives for modeling combinational logic gates.

n-Input	*n*-Output, 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif0

Example: AOI Gate



Output ports of a primitive first, followed by its input port(s)!

Module Ports

- Interface to the environment
- Mode
  - Input
  - Output
  - Inout: bidirectional
- Not order sensitive in declaration

```
module AOI (y_out,x_in1,x_in2,x_in3,x_in4,x_in5);

  input x_in1,x_in2;
  input x_in5,x_in4,x_in3;
  output y_out;

  // ...
endmodule
```

Module Ports

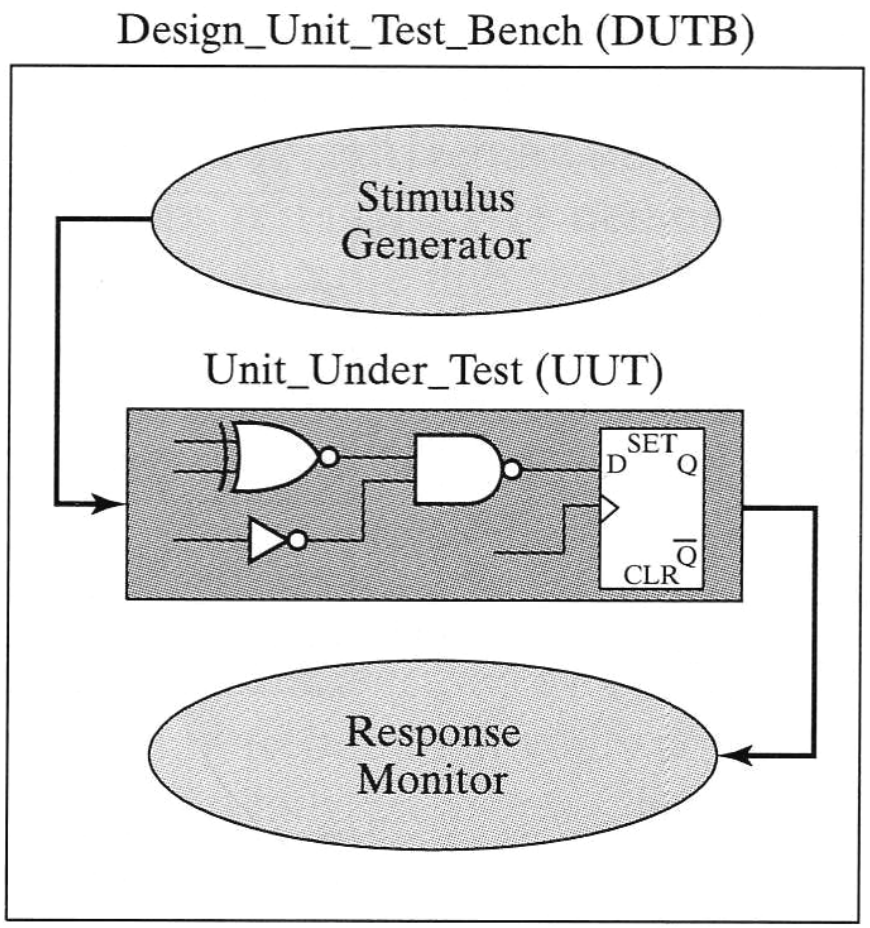
- Order sensitive when used
  - Position sensitive

    AOI M1(w1, a1, a2, b1, b2, b3);
  - Explicitly declared

    AOI M1 (.x_in3(b1), .y_out(w1), .x_in2(a2), .x_in1(a1),.x_in4(b2), .x_in5(b3));

```
module AOI (y_out,x_in1,x_in2,x_in3,x_in4,x_in5);

  input x_in1,x_in2;
  input x_in5,x_in4,x_in3;
  output y_out;

  // ...
endmodule
```

Testbench



- Generate test patterns
  - Waveforms to inputs
- Timing of applying patters
  - Delay in signals
  - Clock generation
- Start/end time for simulation
  - $initial
  - $finish

Test Fixture Template


```
module testfixture ;

    // Data type
    declaration

    // Instantiate modules

    // Apply stimulus

    // Display results
endmodule
```

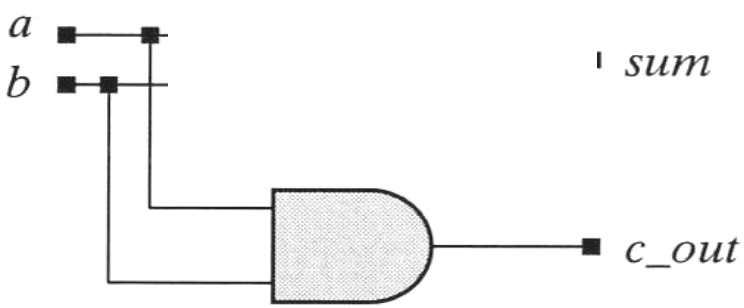
⇨ **There are no ports for test fixture**

Test Fixture – Making an Instance


```
module testfixture ;

// Data type declaration

// MUX instance
        MUX2_1 mux (out, a, b, sel) ;
// Apply stimulus

// Display results

endmodule
```

Example: Half-adder

a	b	sum	c_out
0	0		
0	1		
1	0		
1	1		




```
module   Add_half (sum, c_out, a, b);
   input       a, b;
   output      c_out, sum;

   and         (c_out, a, b);
endmodule
```

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);      // UUT
   initial begin                                  // Time Out
   #100   $finish;
   end

   initial begin
   #10   a = 0; b = 0;
   #10   b = 1;
   #10   a = 1;
   #10   b = 0;
   end
endmodule
```

Unit under test

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay  M1 (sum, c_out, a, b);      // UUT
   initial begin                                 // Time Out
   #100   $finish;
   end

   initial begin
   #10   a = 0; b = 0;
   #10   b = 1;
   #10   a = 1;
   #10   b = 0;
   end
endmodule
```

Module name:
a module called Add_half_0_delay is used

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay  M1 (sum, c_out, a, b);      // UUT
   initial begin                                 // Time Out
   #100   $finish;
   end

   initial begin
   #10   a = 0; b = 0;
   #10   b = 1;
   #10   a = 1;
   #10   b = 0;
   end
endmodule
```

User defined name:
M1 is for internal identification only.
you may name is as X1, or hadd1, or ….

Example: Half-adder


```
module t_Add_half();
   wire          sum, c_out;
   reg           a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);        // UUT
   initial begin                                    // Time Out
   #100   $finish;
   end

   initial begin
   #10    a = 0; b = 0;
   #10    b = 1;
   #10    a = 1;
   #10    b = 0;
   end
endmodule
```

Define length of simulation

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);      // UUT
   initial begin                                  // Time Out
   #100   $finish;
   end

   initial begin
   #10   a = 0; b = 0;
   #10   b = 1;
   #10   a = 1;
   #10   b = 0;
   end
endmodule
```

initial:

-- A single-pass behavior

-- Let the simulator starting from tsim = 0

-- Procedural statements enclosed in begin … end

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);      // UUT
   initial begin                                  // Time Out
   #100   $finish;
   end

   initial begin
   #10    a = 0; b = 0;
   #10    b = 1;
   #10    a = 1;
   #10    b = 0;
   end
endmodule
```

$finish

== STOP

== return control to the OS

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);      // UUT
   initial begin                                  // Time Out
   #100   $finish;
   end

   initial begin
   #10    a = 0; b = 0;
   #10    b = 1;
   #10    a = 1;
   #10    b = 0;
   end
endmodule
```

Delay time

-- Proceeding the statement: #10 b = 1;

-- Delay the execution until specified time

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);      // UUT
   initial begin                                  // Time Out
   #100   $finish;
   end

   initial begin
   #10   a = 0; b = 0;
   #10   b = 1;
   #10   a = 1;
   #10   b = 0;
   end
endmodule
```

Define Inputs

Example: Half-adder


```
module t_Add_half();
   wire        sum, c_out;
   reg         a, b;
   Add_half_0_delay   M1 (sum, c_out, a, b);      // UUT
   initial begin                                  // Time Out
   #100   $finish;
   end

   initial begin
   #10   a = 0; b = 0;
   #10   b = 1;
   #10   a = 1;
   #10   b = 0;
   end
endmodule
```

a	0	0	1	1	1	1
b	0	1	1	0	0	0

time

Signal Generator

- ***initial***:
  - A single-pass behavior
  - Let the simulator starting from $t_{sim} = 0$
  - Procedural statements enclosed in begin … end
- Delay time
  - Proceeding the statement: #10 b = 1;
  - Delay the execution until specified time
- Signal type: **reg**
  - Retain its value from the moment assigned by the procedural statement until change by the next statement
  - Initially given the value x
- ***$finish*** == return control to the OS

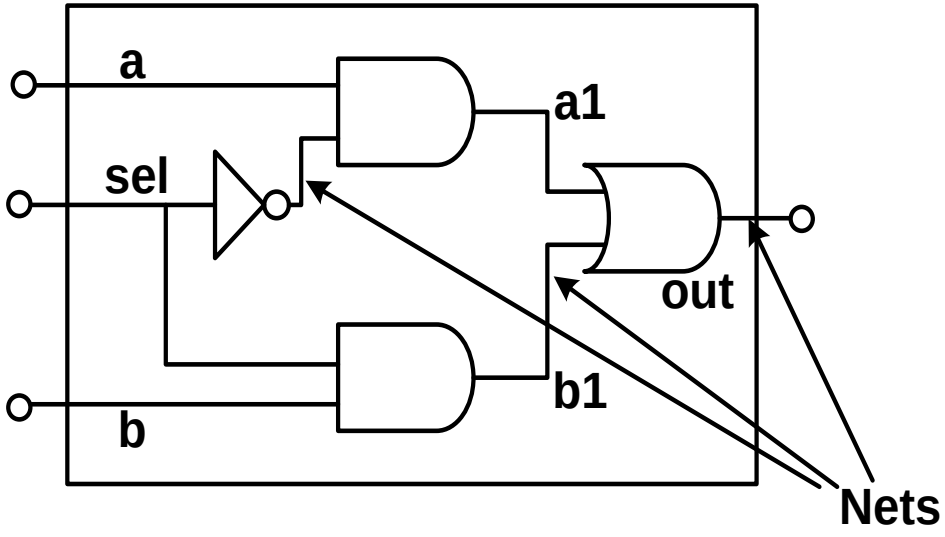
Example: Half-adder

- Net: ***wire***
  - Acts like wires in a physical circuit
  - Connects design objects
  - Needs for a driver

- Register: ***reg, integer***
  - Acts like variables in ordinary procedural languages
  - Stores information while the program executes
  - No needs for a driver, changes its value as wish

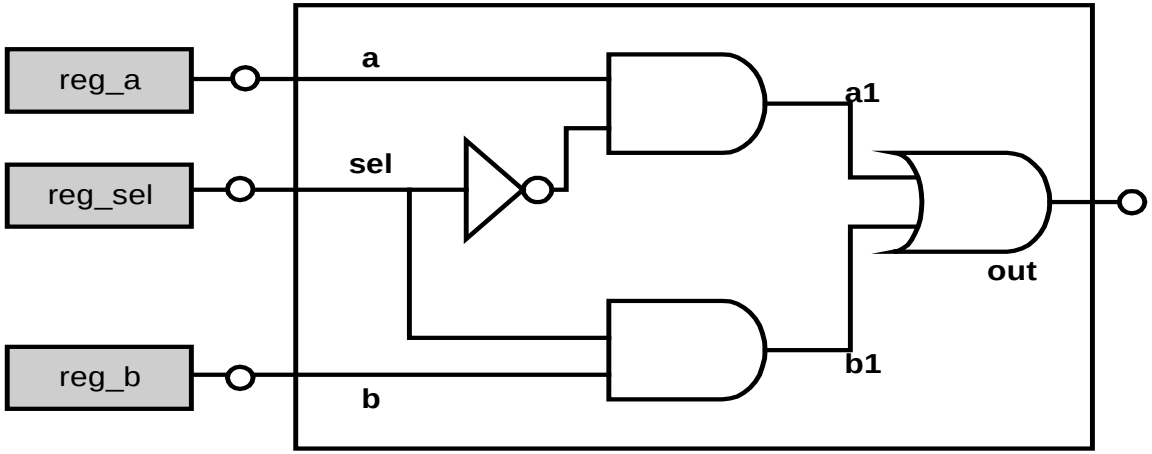
Nets

- Nets are continuously driven by the devices that drive them.
- Verilog automatically propagates a new value onto a net when the drivers on the net change value.

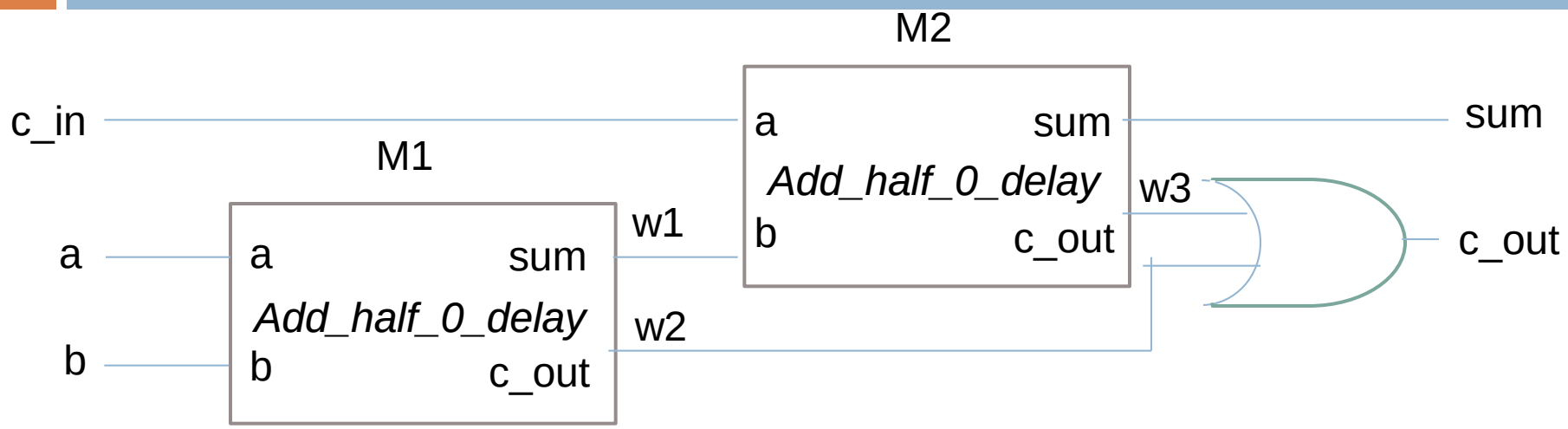


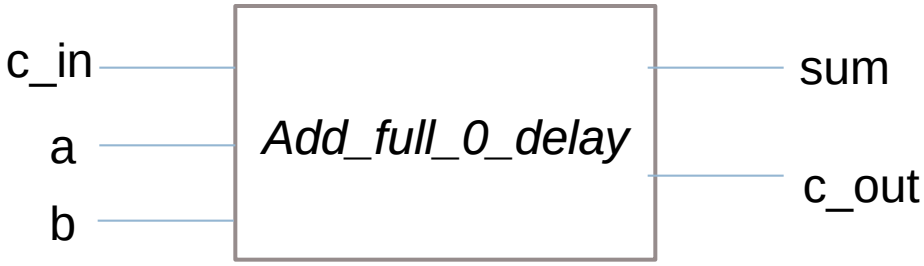
Registers

- A register is merely a variable, which holds its value until a new value is assigned to it. It is different from a hardware register.



Full adder


M2
c_in
a
sum
Add_half_0_delay
b
c_out
sum
w3
c_out
M1
a
a
sum
Add_half_0_delay
b
b
c_out
w1
w2


c_in
a
b
Add_full_0_delay
sum
c_out

Coding Notes


```
module Add_full_0_delay(sum, c_out, a, b, c_in);
   output          sum, c_out;
   input           a, b, c_in;
   wire            w1, w2, w3;

   Add_half_0_delay  M1 (w1, w2, a, b);
   Add_half_0_delay  M2 (sum, w3, w2, c_in);
   or                M3 (c_out, w2, w3);
endmodule
```

Module instance name

Example: 4-bit RCA


```
module Add_rca_4 (sum, c_out, a, b, c_in);
   output      [3: 0]     sum;
   output                 c_out;
   input       [3: 0]     a, b;
   input                  c_in;
   wire                   c_in2, c_in3, c_in4;

   Add_full M1 (sum[0],   c_in2,   a[0], b[0], c_in);
   Add_full M2 (sum[1],   c_in3,   a[1], b[1], c_in2);
   Add_full M3 (sum[2],   c_in4,   a[2], b[2], c_in3);
   Add_full M4 (sum[3],   c_out,   a[3], b[3], c_in4);
endmodule
```

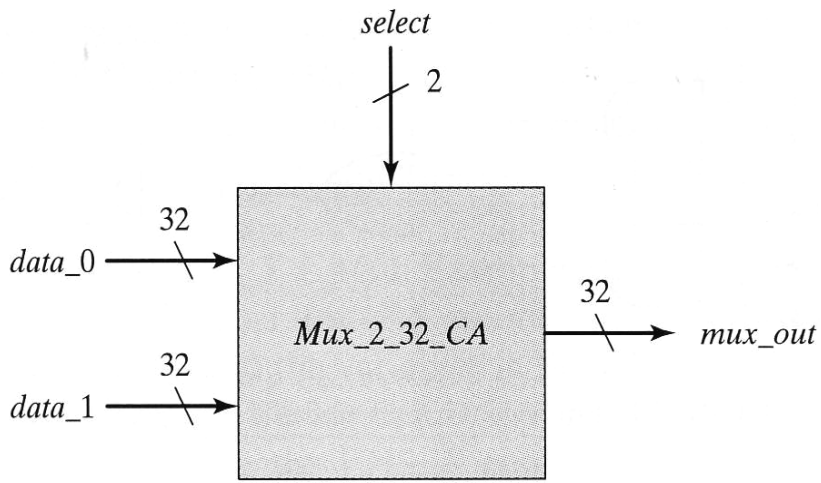
Vectors in Verilog

- e.g., output[15:0] sum;
- Leftmost index bit is the most significant bit
- Rightmost index bit is the least significant bit
- If out of bounds, x is returned
  - X: unknown value

Vectors

- [*high#: low#*] or [*low#: high#*]
- Left number in the [] ➔ the most significant bit
- Examples
  - reg [255:0] data1; // Little endian notation
    // LSB ended in lowest addrs
  - reg [0:255] data2; // Big endian notation
    // MSB ended in lowest addrs

Parameter




```
module Mux_2_32_CA (mux_out, data_1, data_0, select);
  parameter                    word_size = 32;
  output        [word_size-1:0] mux_out;
  input         [word_size-1:0] data_1, data_0;
  input                        select;

  assign mux_out = select?data_1:data_0;
endmodule
```

- Parameter is used to size the word_size
- Extendable for future

Time Scales (1/2)

- Reference time unit
  - `timescale <reference_time_unit>/<time_precision>
    - <reference_time_unit> : unit of measurement for times and delays
    - <time_precision>: the precision to which the delays are round off during simulation
  - Only 1, 10, 100 are valid integers
- Examples
  - `timescale 1ns/10ps
  - `timescale 100ns/1ns

Time Scales (2/2)

```
`timescale 1ns/10ps
module mux2to1_tb;
  reg S, I0, I1;    //inputs
  wire Y; //outputs

  mux2to1 m0 (.Y(Y), .S(S), .I0(I0), .I1(I1));

  initial $monitor($time, " S=%d, I0=%d, I1=%d, Y=%d", S, I0, I1, Y);
  initial begin
        S=0;  I0=0; I1=0;
    #10       I0=0; I1=1;
    #10       I0=1; I1=0;
    #10       I0=1; I1=1;
    #10 S=1;  I0=0; I1=0;
    #10       I0=0; I1=1;
    #10       I0=1; I1=0;
    #10       I0=1; I1=1;
  end

  initial begin
  $dumpfile("mux2to1.vcd");
  $dumpvars;
    #200 $finish;
  end
```

Text Substitution (1/4)

- The `define compiler directive provides a simple text-substitution facility.
  - `define <name> <macro_text>

  `<*name*> will substitute <*macro_text*> at compile time.

- Typically used to make the description more readable
  - `define not_delay #1
  - `define and_delay #2
  - `define or_delay #1

Definition of not_delay

Use of not_delay

```
module MUX2_1 (out, a, b, sel);
output out;
input a, b, sel;
                not `not_delay not1(sel_, sel);
                and `and_delay and1(a1, a, sel_);
                and `and_delay and2(b1,b,sel);
                or  `or_delay  or1(out, a1, b1);
endmodule
```

Text Substitution (2/4)


```
//*****************************************************
`timescale 1ns/10ps
`define PERIOD 20
`include "input_memory.v"
module test_input_memory;
reg clk;
reg rst;
reg enable;
reg  [4:0] ADDRInput;
wire [31:0] OutData;

input_memory u1(.clk(clk), .rst(rst), .enable(enable),
                .ADDRInput(ADDRInput), .OutData(OutData));
always
  #((`PERIOD)/2)  clk=~clk;

initial begin
  clk = 0; rst=0; enable=0; ADDRInput=6'd0;
  #(`PERIOD) rst=1;
  #(`PERIOD*2) rst=0; enable=1;
  #(`PERIOD) ADDRInput=6'd1;
  #(`PERIOD) ADDRInput=6'd2;
  #(`PERIOD) ADDRInput=6'd3;
  #(`PERIOD) ADDRInput=6'd5;
  #(`PERIOD) ADDRInput=6'd6;
  #(`PERIOD) ADDRInput=6'd7;
  #(`PERIOD) ADDRInput=6'd31;
  #(`PERIOD) ADDRInput=6'd30;
```

Text Substitution (3/4)


```
// DEFINES
`define ADD  4'b0011
`define SUB  4'b0100

module ALU (opcode, src1, src2, enable_execute, alu_result, alu_overflow);

...
...
always @(opcode or enable_execute)
begin
      if(enable_execute)
      begin
      case(`OPCODE)
      `ADD: begin {alu_overflow, alu_result}=addsub_result;
         end
      `SUB:  begin {alu_overflow, alu_result}=addsub_result;
         end
.....
```

Text Substitution (4/4)


```
module ALU (opcode, src1, src2, enable_execute, alu_result,
   alu_overflow);


...
...
always @(opcode or enable_execute)
begin
      if(enable_execute)
      begin
      case(`OPCODE)
      4'b0011: begin {alu_overflow, alu_result}=addsub_result;
         end
      4'b0100:  begin {alu_overflow, alu_result}=addsub_result;
         end
.....
```

Text Inclusion (1/2)

- Use the `include compiler directive to insert the contents of an entire file.
  - `include "global.v"
  - `include "parts/count.v"
  - `include "../../library/mux.v"
- Use the +incdir command-line option to specify the search directories for the file to be included.
  - +incdir +<directory1>+<directory2>+...<directoryN>
- You can use `include to
  - include global or commonly used definitions.
  - include tasks without encapsulating repeated code within module boundaries.

Text Inclusion (2/2)


```
`timescale 1ns/10ps
`include "one_bit_fulladder.v"
module test_fulladder;
        reg     A, B, cin;
        wire    S, cout;
        one_bit_fulladder u_one(.S(S), .cout(cout), .A(A), .B(B), .cin(cin));
        initial $monitor($time," A=%d, B=%d, cin=%d, S=%d, cout=%d",A, B, cin, S, cout);
        initial
        begin
                A = 1; B = 0; cin = 0;
                #10 cin = 1;
                #10 A = 0;
                #10 B = 1;
                #10 cin = 0;
                #10 A = 1;
        end
        ...
        ...
        ...
endmodule
```

Special Language Tokens

System Tasks and Functions

$<identifier>

- **'$' sign denotes Verilog system tasks and functions**
- **A number of system tasks and functions are available to perform different operations such as**
  - **Finding the current simulation time ($time)**
  - **Displaying/monitoring the values of the signals ($display, $monitor)**
  - **Stopping the simulation ($stop)**
  - **Finishing the simulation ($finish)**

```
$monitor($time, "a= %b , b= %h", a, b);
```

Built-in Functions(1/2)

- $display : Displaying a message one time
  - Display statements in text window.
  - Similar to the *printf* function in C except it automatically generates a newline at the end of a message.
  - The letter after the percent sign "%" tells the $display how to represent the number to be inserted.
    - d - decimal notation; h - hexadecimal notation
    - o - octal notation; b - binary notation
- Example:
  - if (flag) $display("flag is now %b at time = %d",flag,$time);

Built-in Functions(2/2)

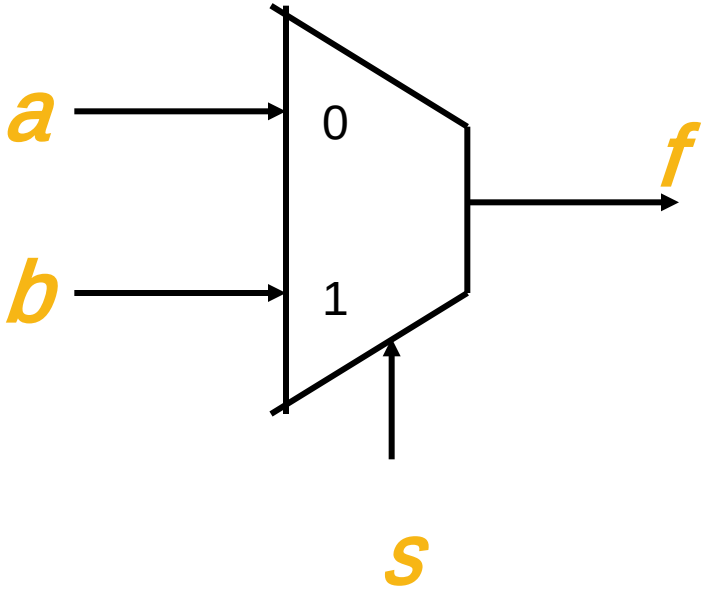
- $monitor: Displaying messages continuously,
  - Displays messages continuously, i.e., it will display statements in text window whenever there is a change with one of the variables in the parameter list.
  - Should be called at the beginning of the simulation
  - The letter after the percent sign "%" tells the $monitor how to represent the number to be inserted.
    - d - decimal notation; h - hexadecimal notation
    - o - octal notation; b - binary notation
  - Example:
    - initial begin $monitor("time = %d num = %h",$time, num); end

Supplement

- How does one get the Boolean function of a Mux using 3 NAND gates?

Functionality of a 2x1 Multiplexier

A multiplexier: use signal *s* to select data from one of two inputs, *a* or *b* port, and output to *f* port



s	a	b	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Boolean Eq. of a MUX

$$f = \bar{s}ab + \bar{s}a\bar{b} + s\bar{a}b + sab$$

$$= \bar{s}a(b + \bar{b}) + s(\bar{a} + a)b$$

$$= \bar{s}a + sb$$

Boolean Eq. of a MUX using NAND Gates only

$$f = \overline{\overline{sa} + \overline{sb}} = \overline{(\overline{sa})(\overline{sb})}$$