N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2010

Verilog (I)

Outline

- History of Verilog
- Logic Values
- Structure Style of Modeling
- Behavioral Style of Modeling
- Number Basics

History of Verilog

Hardware Description Languages (HDL)

- Similarity and Uniqueness
  - Similar to general-purpose languages like C
  - Additional features
    - Modeling and simulation of the functionality of combinational and sequential circuits
    - Parallel vs. sequential behaviors
- Two competitive forces
  - Verilog: C-like language
  - VHDL: ADA-like language

What HDL can do?

- Simulate the behavior of a circuit before it is actually realized.
  - Describe models in common language
  - Execute the models as if they are hardware
  - Be translated into gate-level designs

Development of Verilog

1984	**Gateway Design Automation, Phil Moorby**
1986	**Verilog-XL: an efficient gate-level simulator**
1988	**Verilog logic synthesizer, Synopsys**
1989	**Cadence Data System Inc. acquired Gateway**
1990	**Verilog HDL is released to public domain**
1991	**Open Verilog International (OVI)**
1994	**IEEE 1364 Working group**
1995	**December : Verilog becomes an IEEE standard (IEEE Std. 1364)**
2001	**SystemVerilog**

Verilog HDL and Verilog-XL

Verilog HDL

A hardware description language that allows one to describe circuits at different levels of abstraction.

Verilog-XL software:

A high speed, event-driven logic simulator for Verilog HDL

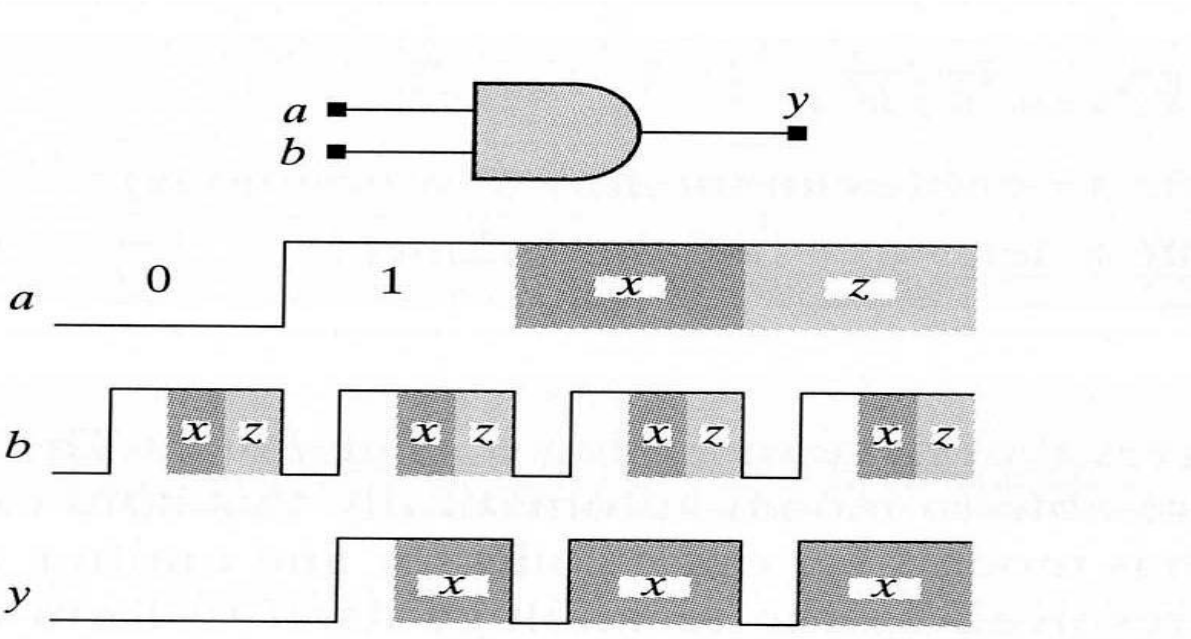
- **Event-driven simulator : Evaluate an element only when its inputs change states. Most practical and widely used simulation algorithm.**

- **Verilog-XL : Incorporates Turbo algorithm, XL algorithm, Switch-XL Algorithm, Caxl algorithm.**

Logic Values

Logic values of the signal

- 4 valued logic: 0, 1, x, and z
  - x: unknow
  - z: high impedance



Four Logic Values

- **1** or High, also **H**, usually representing TRUE.
- **0** or Low, also **L**, usually representing FALSE.
- **X** representing "Unknown", "Don't Know", or "Don't Care".
- **Z** representing "high impedance", or a disconnected input.
- Note
  - X only exists in simulators, not in real hardware

Unknown and High-impedance

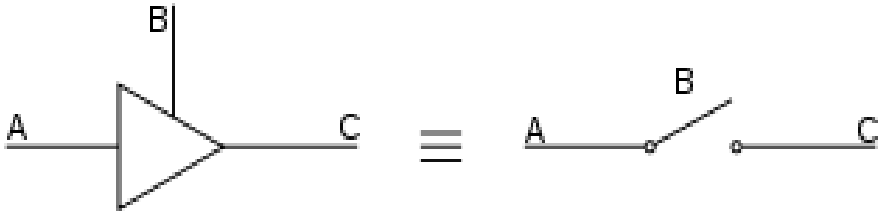
- Unknown (x)
  - Value of signal can not be determined
  - Causes of Value unknown
    - uninitialized
    - Design error → two driving sources to the same signal
    - Design error → unknown state in case selector
    - Design error → MSB of memory address went unknown
- High impedance (z)
  - Floating or tri-stated

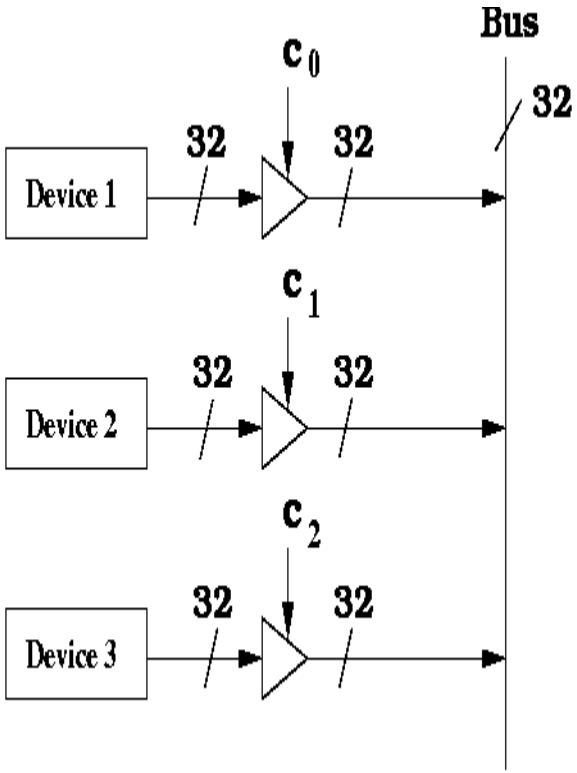
Unknown, X

- Result of a design error
  - Two or more sources driving the same net at the same time
  - Stable output of a flip-flop have not been reached
- Uninitialized memory values or input values before their real values are asserted

High Impedance, Z

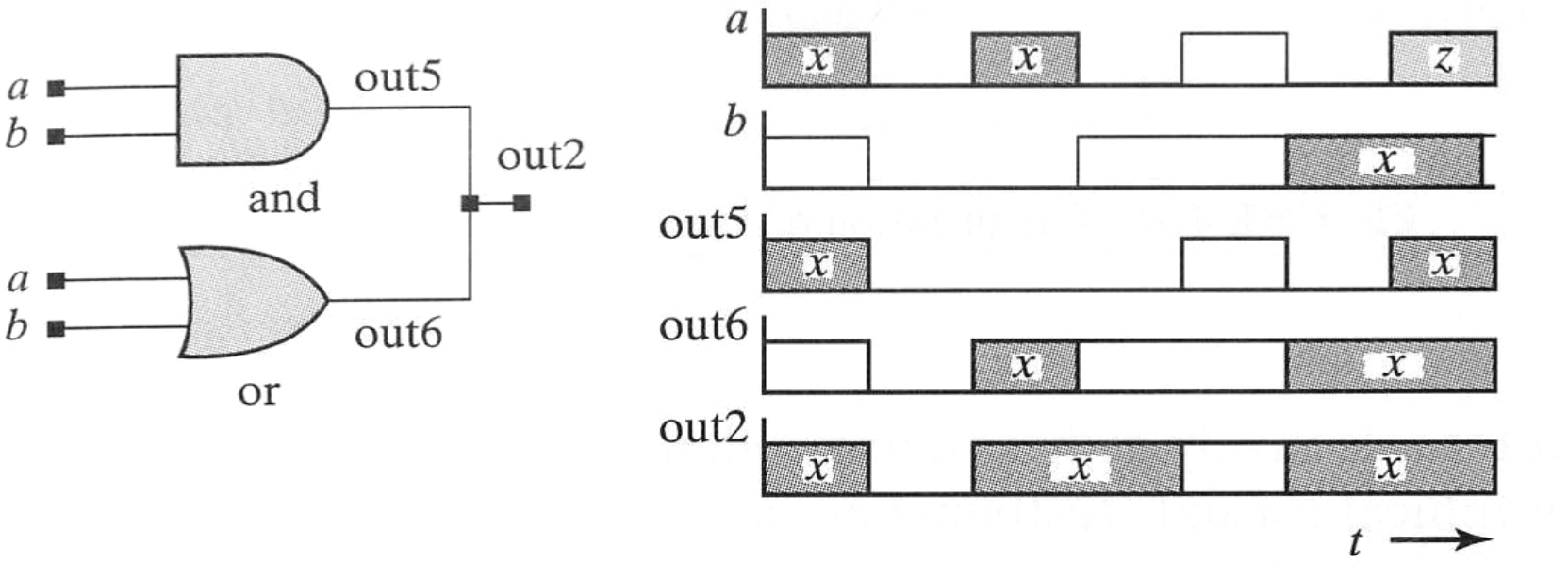
- A disconnected output
- Usually used in bus, a collection of wires in parallel
  - Several devices can communicate one at a time by the same channel.





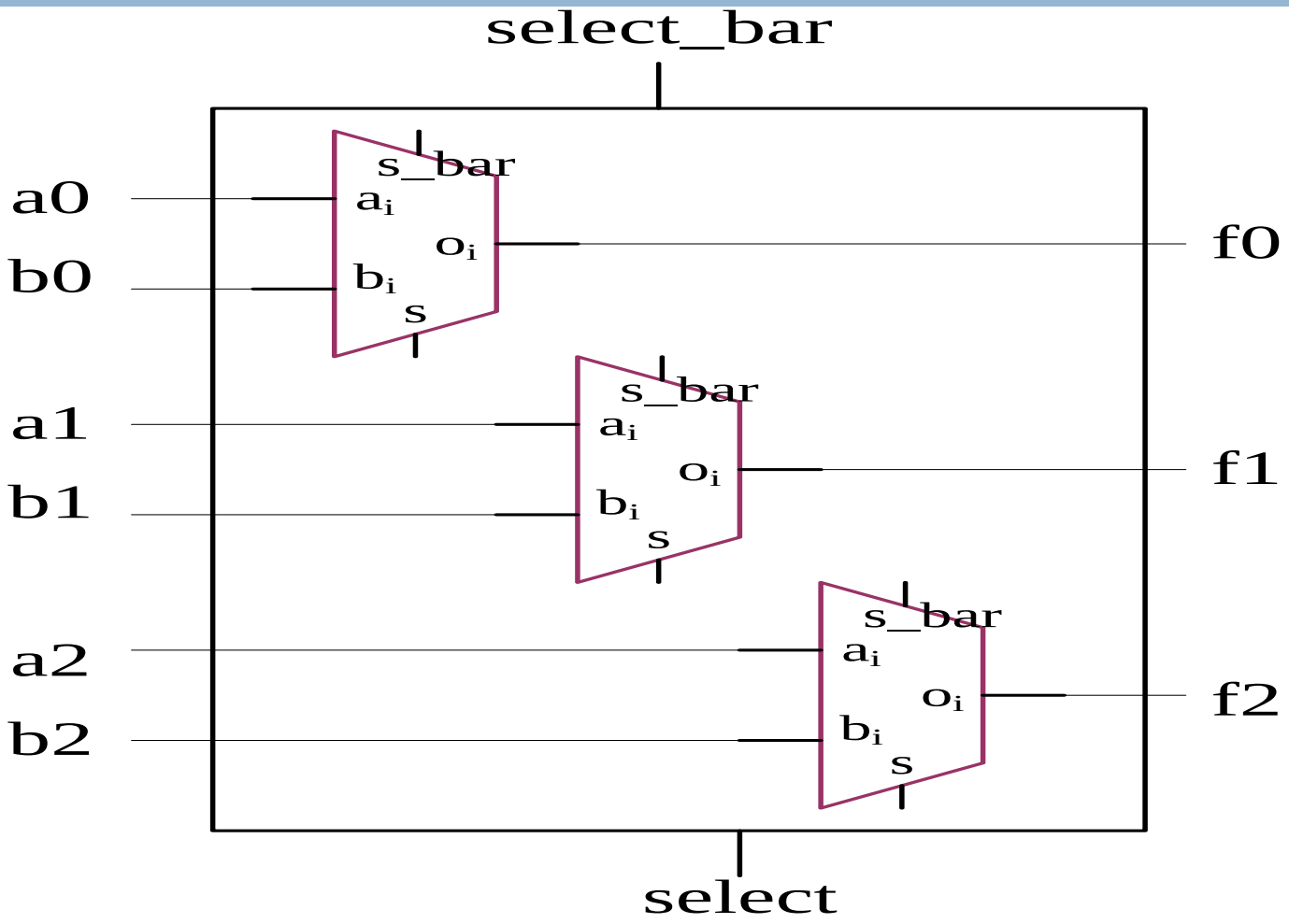
Source: www.wikipedia.org

Signal Contention and Resolution

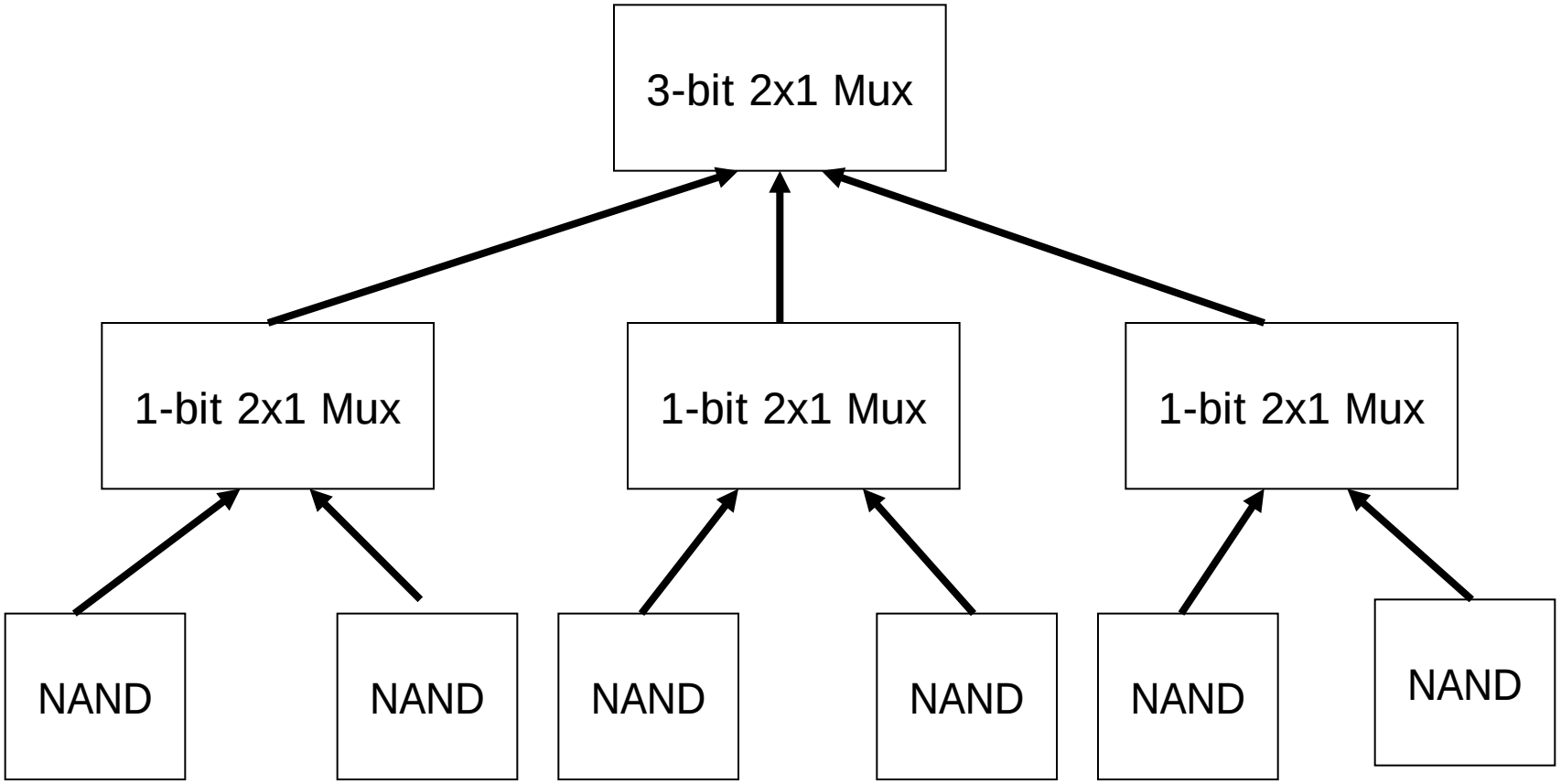


Structure Style of Modeling

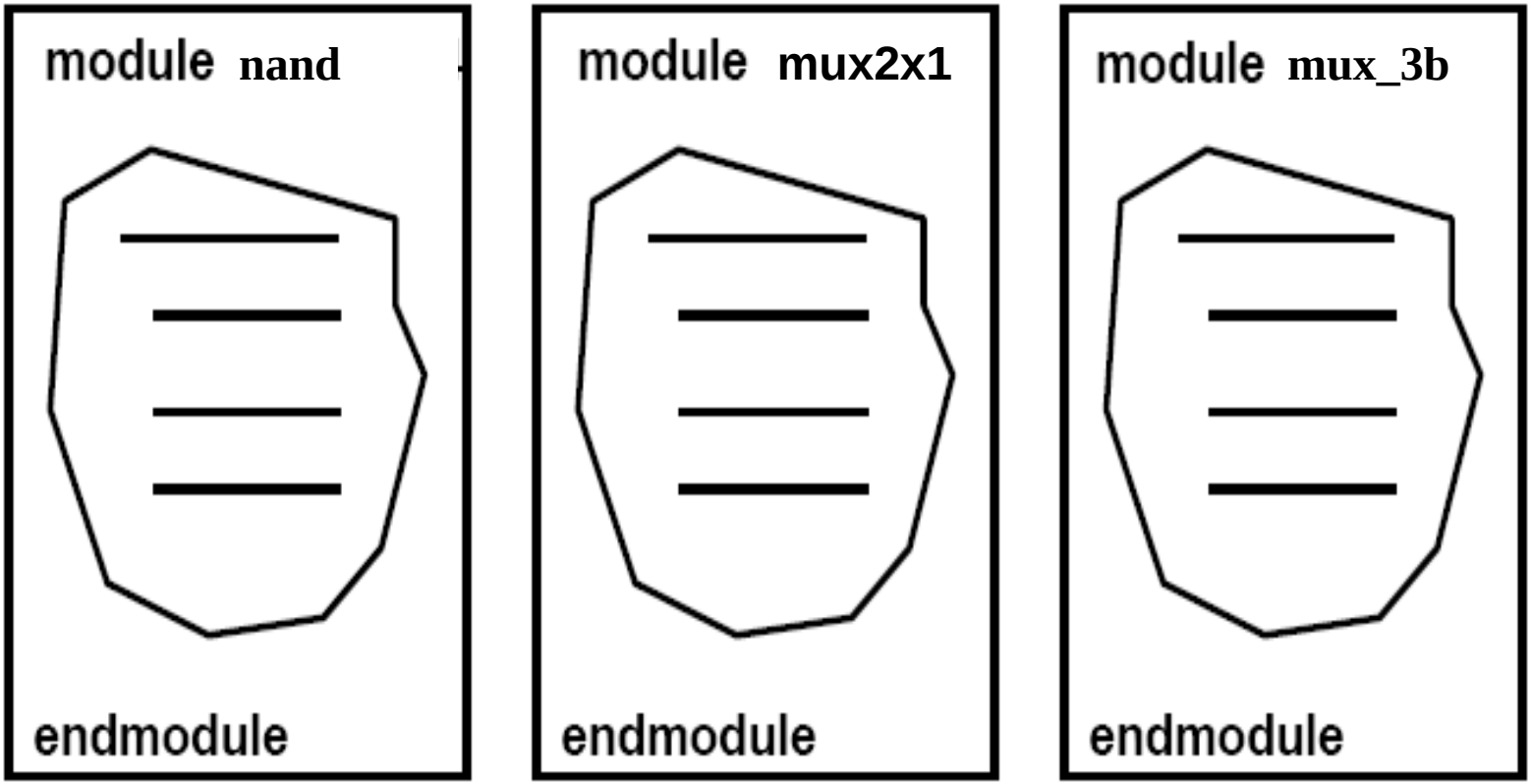
3-bit mux


select_bar
s_bar
a0
ai
oi
f0
b0
bi
s
s_bar
a1
ai
oi
f1
b1
bi
s
s_bar
a2
ai
oi
f2
b2
bi
s
select

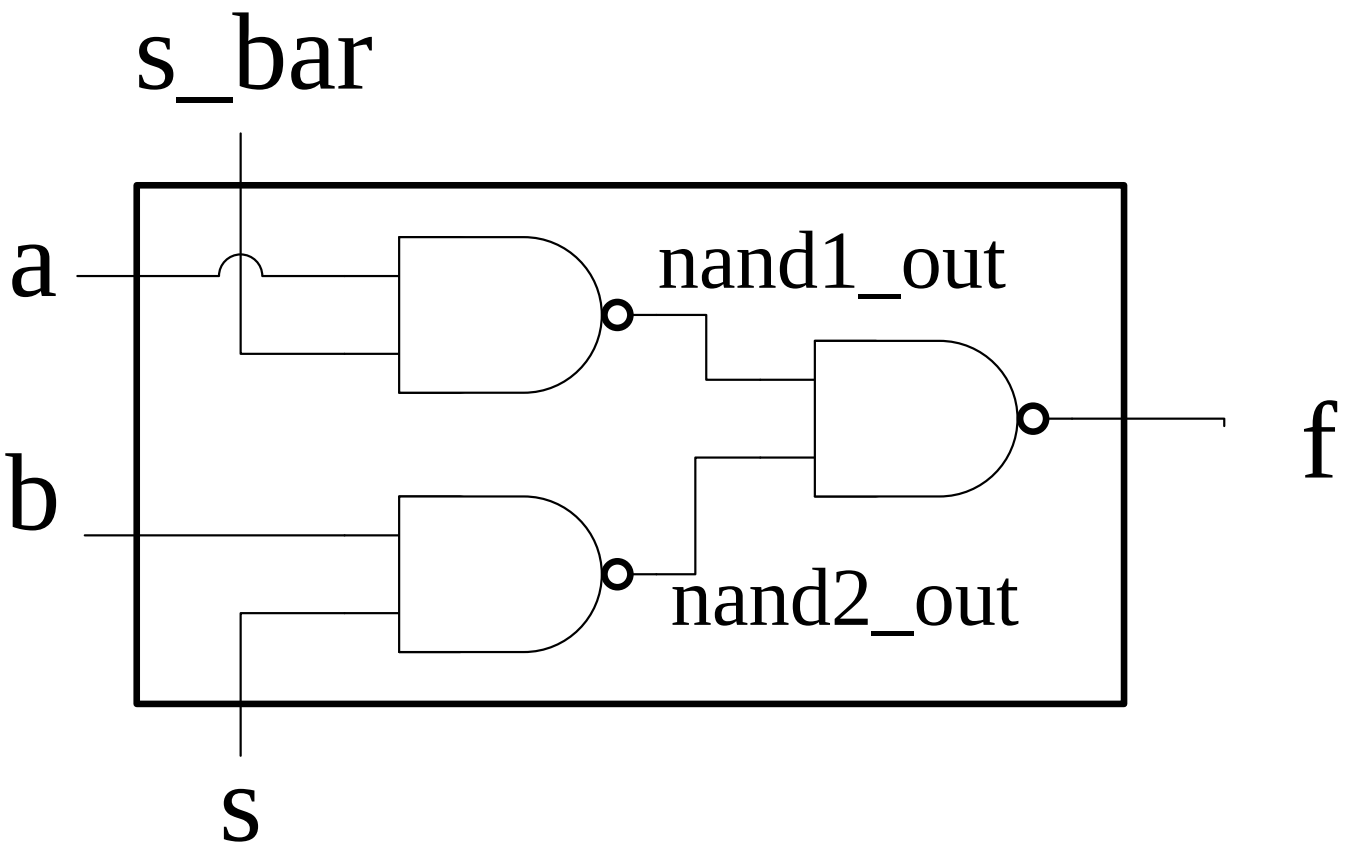
3-bit 2x1 Multiplexier



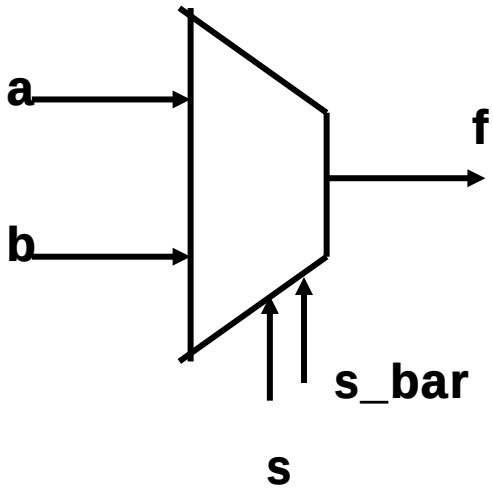
Concepts of Modules



One-bit mux



Mux2x1




```
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
```

Verilog Implementation


```
module mux_3b (f2,f1,f0,a0,b0,a1,b1,a2,b2,
select,select_bar);

   input a0,b0,a1,b1,a2,b2;
   input select, select_bar;
   output f2,f1,f0;

   mux2x1   M0(f0,select, select_bar, a0,b0);
   mux2x1   M1(f1,select, select_bar, a1,b1);
   mux2x1   M2(f2,select, select_bar, a2,b2);

endmodule
```

Behavioral Style of Modeling

Introduction

Data types for Behavioral Modeling

Combinational Logic Blocks

Sequential Logic Blocks

Behavioral Model

- Describe the functionality of a design
  - What the design will do
  - NOT how to build it in hardware
- Specify input-output model of logic circuit
  - Suppress details about internal structure and physical implementation

Why behavioral model?

- Increasing complexity of digital design
  - Need to evaluate the tradeoffs of various architectures
  - Avoid gate-level details by using higher level of abstract
- Encourage
  - Rapid prototyping
  - Fast function verification
  - Leave optimization to a synthesis tool

Example

- Sometime in the design process of CPU
  - For ALU
    - Not care about either ripple-carry adder or carry-lookahead adder
    - ONLY concern whether ADD instruction procesude the sum of two values stored in the register file
- Behavioral model
  - Ignores all timing information
  - Leaves specific timing to the lower level of models

Wire and Register Revisited

- 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

Types of Nets

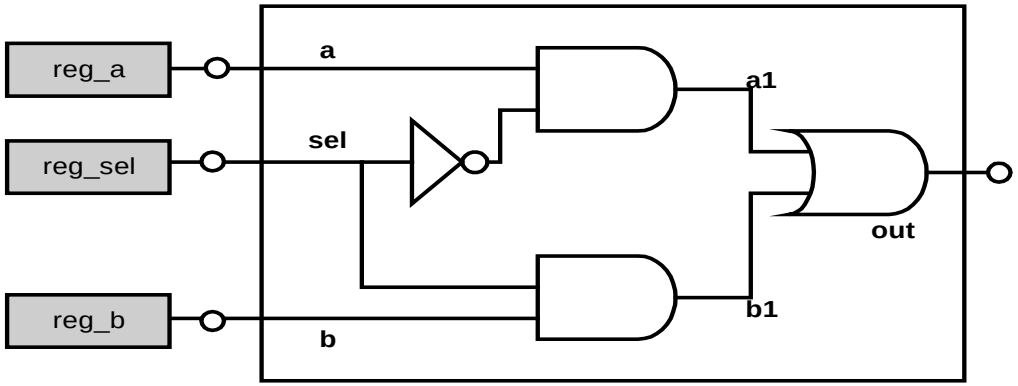
Net Types	Functionality
wire, tri	for standard interconnection wires (default)
wor, trior	for multiple drivers that are Wire-ORed
wand, triand	for multiple drivers that are Wire-ANDed
trireg	for nets with capacitive storage
tri1	for nets which pull up when not driven
tri0	for nets which pull down when not driven
supply1	for power rails
supply0	for ground rails

Nets that are defaulted to single bit nets of type wire. This can be overridden by using the following compiler directive.

‘default_nettype <nettype>

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.
- Registers are used extensively in behavioral modeling and in applying stimuli.
- Values are applied to registers using behavioral constructs.



Behavioral Style of Modeling

Introduction

Data types for Behavioral Modeling

Boolean –Equation Based

Continuous Assignment

Operators

Branching Statements

Utility Constructs

Boolean-Equation-Based Behavioral Models for Combinational Logic


```
module AOI_5 _CA0 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5);
   input       x_in1, x_in2, x_in3, x_in4, x_in5;
   output      y_out;

   assign y_out = ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));

endmodule
```

```
module AOI_5 _CA1 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5, enable);
   // md ciletti
   input            x_in1, x_in2, x_in3, x_in4, x_in5, enable;
   output           y_out;

   assign y_out = enable ? ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5)) : 1'bz;

endmodule
```

Continuous Assignment

- A handy way to model combinational logic
- Operands + operators
- Drive values to a net
  - **wire out, eq;**
    assign out = a&b;
    assign eq = (a==b);
  - **wire #10 inv = ~in;**
  - **wire [7:0] c=a+b;**
- Avoid logic loops
  - **assign a= b+ a;**
  - **asynchronous design**

Operators

{ }	concatenation
+, -, *, /	arithmetic
%	modulus
>, >=, <, <=	relational
!	logical NOT
&&	logical AND
||	logical OR
==	logical equality
!=	logical inequality
?:	conditional

~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
^~, ~^	bit-wise XNOR
&	reduction AND
|	reduction OR
~&	reduction NAND
~|	reduction NOR
^	reduction XOR
~^, ^~	reduction XNOR
<<	shift left
>>	shift right

Reduction Operator

- Logical, bit-wise and unary operators

Example: a=1011, b=0010

logical	bit-wise	unary
a | | b = 1	a | b = 1011	|a = 1
a && b = 1	a & b = 0010	&a = 0

If a = 4'bx000, b=4'bx111,

|a =

&b =

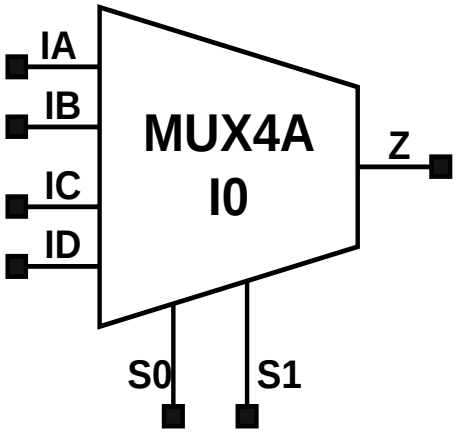
Conditional operator

- Typical conditional operator
  - <condition_expr> ?: <true_expr>:<false_expr>;
  - Acts like a software if-then-else, a switching control
- Nested conditional operator
  - True_expr or false_expr can itself be a conditional operation

Examples

- Conditional operator

```
assign z=({s1, s0} == 2'b00)? IA:
         ({s1, s0} == 2'b01)? IB:
         ({s1, s0} == 2'b10)? IC:
         ({s1, s0} == 2'b11)? ID: 1'bx;
```




```
assign s = ( op == ADD)? a+b: a-b;
```

Equality Operators

- == is the equality operator.

```
a = 2'b1x;
b = 2'b1x;
if(a == b)
    $display("a is equal to b");
else
    $display("a is not equal to 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;
if(a === b)
    $display("a is identical to b");
else
    $display("a is not identical to 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 Operator

- { }
- Target operands may be wires or regs
- wire [3:0] A = 4'b0010;
- reg [7:0] B = 8'b0000_1111;
- wire [7:0] C = {A, B[5:2]};
- wire [15:0] A_ones = 1111_1111_1111_1111;
- wire [15:0] B_ones = {16{1'b1}};
- wire [23:0] = {A,B[5:2],{16{1'b1}}};

Conditional Statements

- If and If-Else Statements

```
initial
if (index > 0) // Beginning of Outer if
    if (rega > regb) // Beginning of the 1st inner if
        result = rega;
    else
        result = 0; // End of the 1st inner if
else
    if (index == 0)
        $display("Note : Index is zero");
    else
        $display("Note : Index is negative");
```

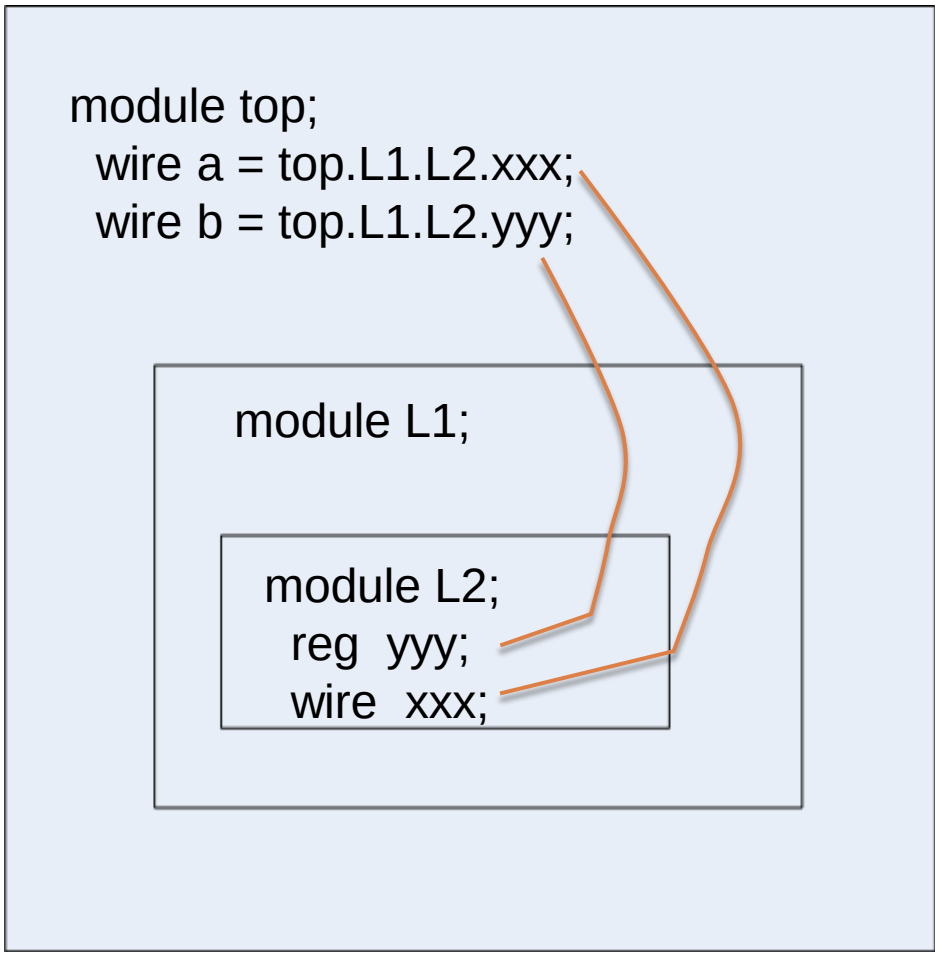
Multiway Branching (Case)


```
reg [2:0]  opcode;

....
        case (opcode)
          3'b000 : result = rega + regb;
          3'b001 : result = rega - regb;
          3'b010 : result = rega * regb;
          3'b100 : result = rega / regb;
          default : begin
             result = 'bx;
             $display ("no match");
             end
        endcase
```

Signal Extraction

- Useful for testing and debugging
- Enable designers to extract signals on the bottom blocks from higher-level blocks



Random Number Generator

- $random(seed) // preferable in test benches

```
integer   k, seed8 = 2;
reg [11:0]  memA [1023:0];
initial i =0;
begin
        repeat (1024) begin
                memA[k] = $random(seed8); k = k +1;
        end
 end;
```

File I/O

- File_ID = $fopen("filedir/filename");
- $fclose(File_ID)

```
integer fid1, fird2, fid3;
initial
begin
   fid1 = $fopen("a.dat");
   fid2 = $fopen("../b.dat");
end
```

File I/O

- $fmonitor(file_id, format_string, variable_list);

```
integer fid1;
initial
begin
    fid1 = $fopen("a.dat");
  $fmonitor(fid1,"%m:  %t in1=%d out1=%h", $time, in1, out1);
end
…
…
```

File I/O

- $fread(temp_variable, file_id, string_format);

```
integer fid1;
reg [31:0] temp;
initial
begin
    fid1 = $fopen("test.dat");
end
always @(posedge clk)
   $fread(temp, fid1, "%h");
end
…
…
```

//test.dat

00 00 00 00 04 03 00 00 c1 df 00 00 e7 87 00 00

54 54 00 00 16 f0 00 00 e7 07 00 00 01 00 00 00

60 f0 00 00 e7 07 00 00 01 00 00 00 62 f0 00 00