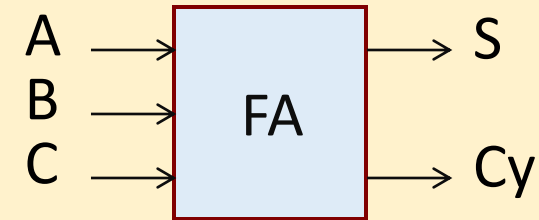


Design Representation

Behavioral Representation :: Example

Full Adder:

- two operand inputs A and B
- a carry input C
- a carry output Cy
- a sum output S



- Express in terms of Boolean expressions:

$$S = A.B'.C' + A'.B'.C + A'.B.C' + A.B.C = A \oplus B \oplus C$$

$$Cy = A.B + A.C + B.C$$

- Express in Verilog in terms of Boolean expressions

```
module carry (S, Cy, A, B, C);  
    input  A, B, C;  
    output S, Cy;  
    assign S  =  A ^ B ^ C;  
    assign Cy =  (A & B) | (B & C) | (C & A);  
endmodule
```

Structural Representation

- Specifies how components are interconnected.
- In general, the description is a list of modules and their interconnection.
 - Called *netlist*.
 - Can be specified at various levels.
 - Gate level
 - Module level

Structural Representation :: Example

```
module xor2 (f, a, b);  
    input a, b;  
    output cy_out;  
    wire t1, t2, t3, t4;  
    not g1 (t1, b);  
    not g2 (t2, a);  
    and g3 (t3, a, t1);  
    and g4 (t4, t2, b);  
    or g5 (f, t3, t4);  
endmodule
```

```
module parity4 (p, x1, x2, x3, x4);  
    input x1, x2, x3, x4;  
    output p;  
    wire t1, t2;  
    xor2 g1 (t1, x1, x2);  
    xor2 g2 (t2, x3, x4);  
    xor2 g3 (p, t1, t2);  
endmodule
```

Verilog Language Features

Concept of Verilog “Module”

- In Verilog, the basic unit of hardware is called a *module*.
 - A module cannot contain definition of other modules.
 - A module can, however, be *instantiated* within another module.
 - Instantiation allows the creation of a *hierarchy* in Verilog description.

```
module module_name (list_of_ports);  
    input/output declarations  
    local net declarations  
    Parallel statements  
endmodule
```

```
// A simple AND function
module simpleand (f, x, y);
    input  x, y;
    output f;
    assign f = x & y;
endmodule
```

This is a behavioral description. The synthesis tool will decide how to realize f:

- a) Using a single AND gate
- b) Using a NAND gate followed by a NOT gate.

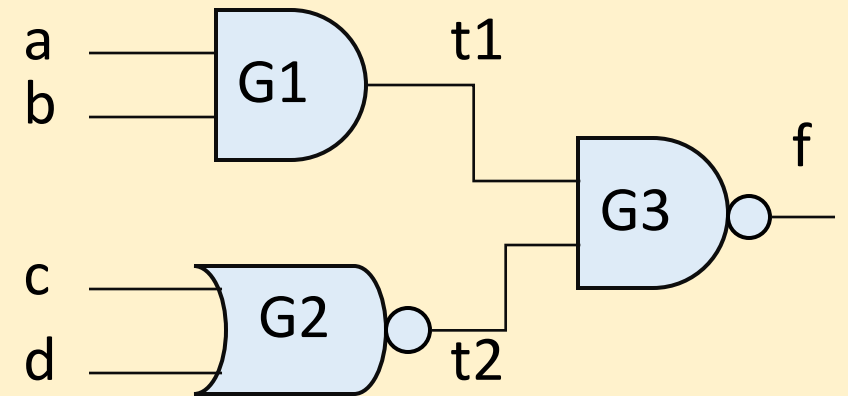

```

/* A 2-level combinational circuit */
module two_level (a, b, c, d, f);
    input  a, b, c, d;
    output f;
    wire  t1, t2;  // Intermediate lines
    assign t1 = a & b;
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule

```

This is also behavioral description.

- One possible gate level realization is shown.
- t1 and t2 are intermediate lines; termed as wire data type.



- Point to note:

- The “assign” statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS changes.

assign variable = expression;

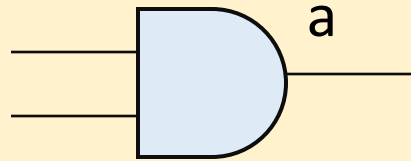
- The LHS must be a “*net*” type variable, typically a “*wire*”.
- The RHS can contain both “*register*” and “*net*” type variables.
- A Verilog module can contain any number of “*assign*” statements; they are typically placed in the beginning after the port declarations.
- The “*assign*” statement models behavioral design style, and is typically used to model combinational circuits.

Data Types in Verilog

- A variable in Verilog belongs to one of two data types:
 - a) Net
 - Must be continuously driven.
 - Cannot be used to store a value.
 - Used to model connections between continuous assignments and instantiations.
 - b) Register
 - Retains the last value assigned to it.
 - Often used to represent storage elements, but sometimes it can translate to combinational circuits also.

(a) Net data type

- Nets represents connection between hardware elements.
- Nets are continuously driven by the outputs of the devices they are connected to.



- Net “a” is continuously driven by the output of the AND gate.
- Nets are 1-bit values by default unless they are declared explicitly as vectors.
 - Default value of a net is “z”.

- Various “*Net*” data types are supported for synthesis in Verilog:
 - wire, wor, wand, tri, supply0, supply1, etc.
- “*wire*” and “*tri*” are equivalent; when there are multiple drivers driving them, the driver outputs are shorted together.
- “*wor*” and “*wand*” inserts an OR and AND gate respectively at the connection.
- “*supply0*” and “*supply1*” model power supply connections.
- The Net data type “*wire*” is most common.

```
module using_supply_wire (A, B, C, f);  
    input    A, B, C;  
    output   f;  
    supply0  gnd;  
    supply1  vdd;  
    nand     G1  (t1, vdd, A, B);  
    xor      G2  (t2, C, gnd);  
    and      G3  (f, t1, t2);  
endmodule
```

(b) Register Data Type

- In Verilog, a “*register*” is a variable that can hold a value.
 - Unlike a “*net*” that is continuously driven and cannot hold any value.
 - Does not necessarily mean that it will map to a hardware register during synthesis.
 - Combinational circuit specifications can also use register type variables.
- Register data types supported by Verilog:
 - `reg` : Most widely used
 - `integer` : Used for loop counting (typical use)
 - `real` : Used to store floating-point numbers
 - `time` : Keeps track of simulation time (not used in synthesis)

- “reg” data type:

- Default value of a “reg” data type is “x”.
- It can be assigned a value in synchronism with a clock or even otherwise.
- The declaration explicitly specifies the size (default is 1-bit):

```
reg x, y;           // Single-bit register variables
```

```
reg [15:0] bus;     // A 16-bit bus
```

- Treated as an unsigned number in arithmetic expressions.
- Must be used when we model actual sequential hardware elements like counters, shift registers, etc.


```

module simple_counter (clk, rst, count);
    input    clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule

```

32-bit counter with synchronous reset.

- Count value increases at the positive edge of the clock.
- If “rst” is high, the counter is reset at the positive edge of the next clock.

```

module simple_counter (clk, rst, count);
    input    clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk or posedge rst)
    begin
        if (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule

```

32-bit counter with asynchronous reset.

- Here reset occurs whenever “rst” goes high.
- Does not synchronize with clock.

- “integer” data type:
 - It is a general-purpose register data type used for manipulating quantities.
 - More convenient to use in situations like loop counting than “reg”.
 - It is treated as a 2’s complement signed integer in arithmetic expressions.
 - Default size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.
 - Example:
 - `wire [15:0] X, Y;`
 - `integer C;`
 - `Z = X + Y;`
 - Size of Z can be deduced to be 17 (16 bits plus a carry).

Vectors

- Nets or “reg” type variable can be declared as vectors, of multiple bit widths.
 - If bit width is not specified, default size is 1-bit.
- Vectors are declared by specifying a range [range1:range2], where range1 is always the most significant bit and range2 is the least significant bit.
- Examples:
 - wire x, y, z; // Single bit variables
 - wire [7:0] sum; // MSB is sum[7], LSB is sum[0]
 - reg [31:0] MDR;
 - reg [1:10] data; // MSB is data[1], LSB is data[10]
 - reg clock;

- Parts of a vector can be addressed and used in an expression.
- Example:
 - A 32-bit instruction register, that contains a 6-bit opcode, three register operands of 5 bits each, and an 11-bit offset.

reg [31:0] IR;

reg [5:0] opcode;

reg [4:0] reg1, reg2, reg3;

reg [10:0] offset;

opcode = IR[31:26];

reg1 = IR[25:21];

reg2 = IR[20:16];

reg3 = IR[15:11];

offset = IR[10:0];

Multi-dimensional Arrays and Memories

- Multi-dimensional arrays of any dimension can be declared in Verilog.
- Example:
 - `reg [31:0] register_bank[15:0]; // 16 32-bit registers`
 - `integer matrix[7:0][15:0];`

Specifying Constant Values

- A constant value may be specified in either the sized or the unsized form.

- Syntax of sized form:

`<size>'<base><number>`

- Examples:

- `4'b0101` `// 4-bit binary number 0101`
 - `1'b0` `// Logic 0 (1-bit)`
 - `12'hB3C` `// 12-bit number 1011 0011 1100`
 - `12'h8xF` `// 12-bit number 1000 xxxx 1111`
 - `25` `// signed number, in 32 bits (size not specified)`

Variables of type integer and real are typically expressed in unsized form.

Parameters

- A parameter is a constant with a given name.
 - We cannot specify the size of a parameter.
 - The size gets decided from the constant value itself; if size is not specified, it is taken to be 32 bits.

- Examples:

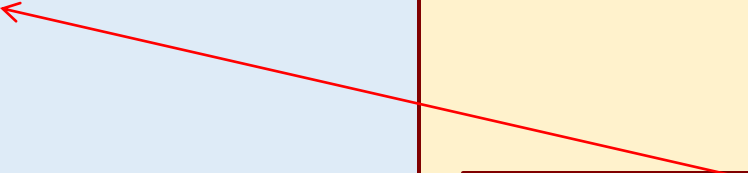
parameter HI = 25, LO = 5;

parameter up = 2b'00, down = 2b'01, steady = 2b'10;

parameter RED = 3b'100, YELLOW = 3b'010, GREEN = 3b'001;


```
// Parameterized design:: an N-bit counter
```

```
module counter (clear, clock, count);  
    parameter N = 7;  
    input clear, clock;  
    output [0:N] count;    reg [0:N] count;  
  
    always @ (negedge clock)  
        if (clear)  
            count <= 0;  
        else  
            count <= count + 1;  
  
endmodule
```



Any variable assigned within the “always” block must be of type “reg”.

List of Primitive Gates

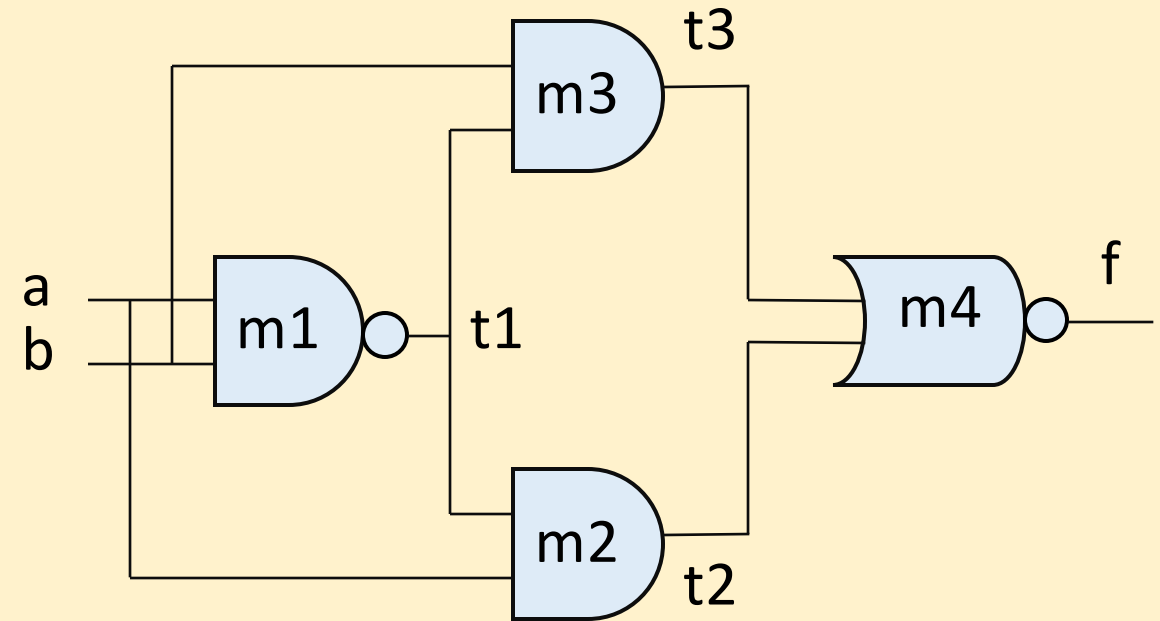
```
and    G (out, in1, in2);
nand   G (out, in1, in2);
or     G (out, in1, in2);
nor    G (out, in1, in2);
xor    G (out, in1, in2);
xnor   G (out, in1, in2);
not    G (out, in);
buf    G (out, in);
```

```
bufif1 G (out, in, ctrl);
bufif0 G (out, in, ctrl);
notif0 G (out, in, ctrl);
notif1 G (out, in, ctrl);
```

These are gates with tristate controls

- Some restriction when instantiating primitive gates:
 - The output port must be connected to a net (e.g. a wire).
 - An “*output*” signal is a wire by default, unless explicitly declared as a register.
 - The input ports may be connected to nets or register type variables.
 - They have a single output but can have any number of inputs (except NOT and BUF).
 - When instantiating a gate, an optional delay may be specified.
 - Used for simulation.
 - Logic synthesis tools ignore the time delays.

```
`timescale 10ns/1ns
module exclusive_or (f, a, b);
    input a, b;
    output f;
    wire t1, t2, t3;
    nand #5 m1 (t1, a, b);
    and #5 m2 (t2, a, t1);
    and #5 m3 (t3, t1, b);
    nor #5 m4 (f, t2, t3);
endmodule
```



- Example:

``timescale 10ns/1ns`

- Reference time unit is 10ns, and simulation precision is 1ns.
- If we specify #5 as delay, it will mean 50ns.
- The time units can be specified in s (second), ms (millisecond), us (microsecond), ps (picosecond), and fs (femtosecond).

Specifying Connectivity during Instantiation

- When a module is instantiated within another module, there are two ways to specify the connectivity of the signal lines between the two modules.
 - a) **Positional association**
 - The parameters of the module being instantiated are listed in the same order as in the original module description.
 - b) **Explicit association**
 - The parameters of the module being instantiated are listed in arbitrary order.
 - Chance of errors is less.

```

module  testbench;
  reg   X1,X2,X3,X4,X5,X6;  wire  OUT;
  example DUT (X1,X2,X3,X4,X5,X6,OUT) ;

  initial
  begin
    $monitor ($time," X1=%b, X2=%b,
      X3=%b, X4=%b, X5=%b, X6=%b,
      OUT=%b", X1,X2,X3,X4,X5,X6,OUT) ;
    #5 X1=1;X2=0; X3=0; X4=1; X5=0; X6=0;
    #5 X1=0; X3=1;
    #5 X1=1; X3=0;
    #5 X6=1;
    #5 $finish;
  end
endmodule

```

Positional Association

```

module  example
      (A,B,C,D,E,F,Y) ;
  wire  t1, t2, t3, Y;

  nand  #1 G1 (t1,A,B) ;
  and   #2 G2 (t2,C,~B,D) ;
  nor   #1 G3 (t3,E,F) ;
  nand  #1 G4 (Y,t1,t2,t3) ;
endmodule

```

```

module testbench;
  reg  X1,X2,X3,X4,X5,X6;  wire OUT;
  example DUT (.OUT(Y) , .X1(A) , .X2(B) , .X3(C) ,
              .X4(D) , .X5(E) , .X6(F) ) ;

  initial
    begin
      $monitor ($time," X1=%b, X2=%b,
        X3=%b, X4=%b, X5=%b, X6=%b,
        OUT=%b" , X1,X2,X3,X4,X5,X6,OUT) ;
      #5 X1=1;X2=0; X3=0; X4=1; X5=0; X6=0;
      #5 X1=0; X3=1;
      #5 X1=1; X3=0;
      #5 X6=1;
      #5 $finish;
    end
endmodule

```

Explicit Association

```

module example
  (A,B,C,D,E,F,Y) ;
  wire t1, t2, t3, Y;
  nand #1 G1 (t1,A,B) ;
  and  #2 G2 (t2,C,~B,D) ;
  nor  #1 G3 (t3,E,F) ;
  nand #1 G4 (Y,t1,t2,t3) ;
endmodule

```


Verilog Operators

Arithmetic Operators:

+	unary (sign) plus
−	unary (sign) minus
+	binary plus (add)
−	binary minus (subtract)
*	multiply
/	divide
%	modulus
**	exponentiation

Examples:

− (b + c)
(a − b) + (c * d)
(a + b) / (a − b)
a % b
a ** 3

Logical Operators:

! logical negation
&& logical AND
|| logical OR

Examples:

(done && ack)

(a || b)

! (a && b)

((a > b) || (c == 0))

((a > b) && ! (b > c))

- The value 0 is treated as logical FALSE while any non-zero value is treated as TRUE.
- Logical operators return either 0 (FALSE) or 1 (TRUE).

Relational Operators:

<code>!=</code>	not equal
<code>==</code>	equal
<code>>=</code>	greater or equal
<code><=</code>	less or equal
<code>></code>	greater
<code><</code>	less

Examples:

```
(a != b)
((a + b) == (c - d))
((a > b) && (c < d))
(count <= 0)
```

Relational operators operate on numbers, and return a Boolean value (true or false).

Bitwise Operators:

~	bitwise NOT
&	bitwise AND
	bitwise OR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Examples:

```
wire a, b, c, d, f1, f2, f3, f4;  
assign f1 = ~a | b;  
assign f2 = (a & b) | (b & c) | (c & a)  
assign f3 = a ^ b ^ c;  
assign f4 = (a & ~b) | (b & c & ~d);
```

Bitwise operators operate on bits, and return a value that is also a bit.

Reduction operators accept a single word operand and produce a single bit as output.

- Operates on all the bits within the word.

Reduction Operators:

&	bitwise AND
	bitwise OR
~&	bitwise NAND
~	bitwise NOR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Examples:

```
wire [3:0] a, b, c; wire f1, f2, f3;
assign a = 4'b0111;
assign b = 4'b1100;
assign c = 4'b0100;
assign f1 = ^a;           // gives a 1
assign f2 = &(a ^ b);     // gives a 0
assign f3 = ^a & ~^b;     // gives a 1
```

Shift Operators:

>> shift right
<< shift left
>>> arithmetic shift right

Conditional Operator:

cond_expr ? true_expr : false_expr;

Examples:

```
wire [15:0] data, target;  
assign target = data >> 3;  
assign target = data >>> 2;
```

Examples:

```
wire a, b, c;  
wire [7:0] x, y, z;  
assign a = (b > c) ? b : c;  
assign z = (x == y) ? x+2 : x-2;
```

Concatenation Operator:

`{..., ..., ...}`

Joins together bits from two or more comma-separated expressions.

Replication Operator:

`{n{m}}`

Joins together n copies of an expression m, where n is a constant.

Examples:

```
assign f = {a, b};
```

```
assign f = {a, 3'b101, b};
```

```
assign f = {x[2], y[0], a};
```

```
assign f = {2'b10, 3{2'b01}, x};
```

```
// An 8-bit adder description
module parallel_adder (sum, cout, in1, in2, cin);
    input    [7:0]  in1, in2;
    input    cin;
    output   [7:0]  sum;
    output   cout;

    assign   #20 {cout,sum} = in1 + in2 + cin;
endmodule
```


Operator Precedence

- Operators on same line have the same precedence.
- All operators associate left to right in an expression, except ?:
- Parentheses can be used to change the precedence.

+	-	!	~	(unary)
**				
* / %				
<< >> >>>				
< <= > >=				
== != === !==				
& ~&				
^ ~^				
~				
&&				
?:				

↑
Precedence increases

Structural and Behavioral Modeling Example

Example 1

- The structural hierarchical description of a 16-to-1 multiplexer.
 - a) Using pure behavioral modeling.
 - b) Structural modeling using 4-to-1 multiplexer specified using behavioral model.
 - c) Make structural modeling of 4-to-1 multiplexer, using behavioral modeling of 2-to-1 multiplexer.
 - d) Make structural gate-level modeling of 2-to-1 multiplexer, to have a complete structural hierarchical description.

Version 1: Using pure behavioral modeling

```
module mux16to1 (in, sel, out);  
    input [15:0] in;  
    input [3:0] sel;  
    output out;  
  
    assign out = in[sel];  
endmodule
```

Selects one of the input bits depending upon the value of “sel”.

```

module muxtest;
    reg [15:0] A;      reg [3:0] S;      wire F;

    mux16to1 M (.in(A), .sel(S), .out(F));

    initial
        begin
            $dumpfile ("mux16to1.vcd");
            $dumpvars (0,muxtest);
            $monitor ($time," A=%h, S=%h, F=%b", A,S,F);
            #5 A=16'h3f0a; S=4'h0;
            #5 S=4'h1;
            #5 S=4'h6;
            #5 S=4'hc;
            #5 $finish;
        end
    endmodule

```

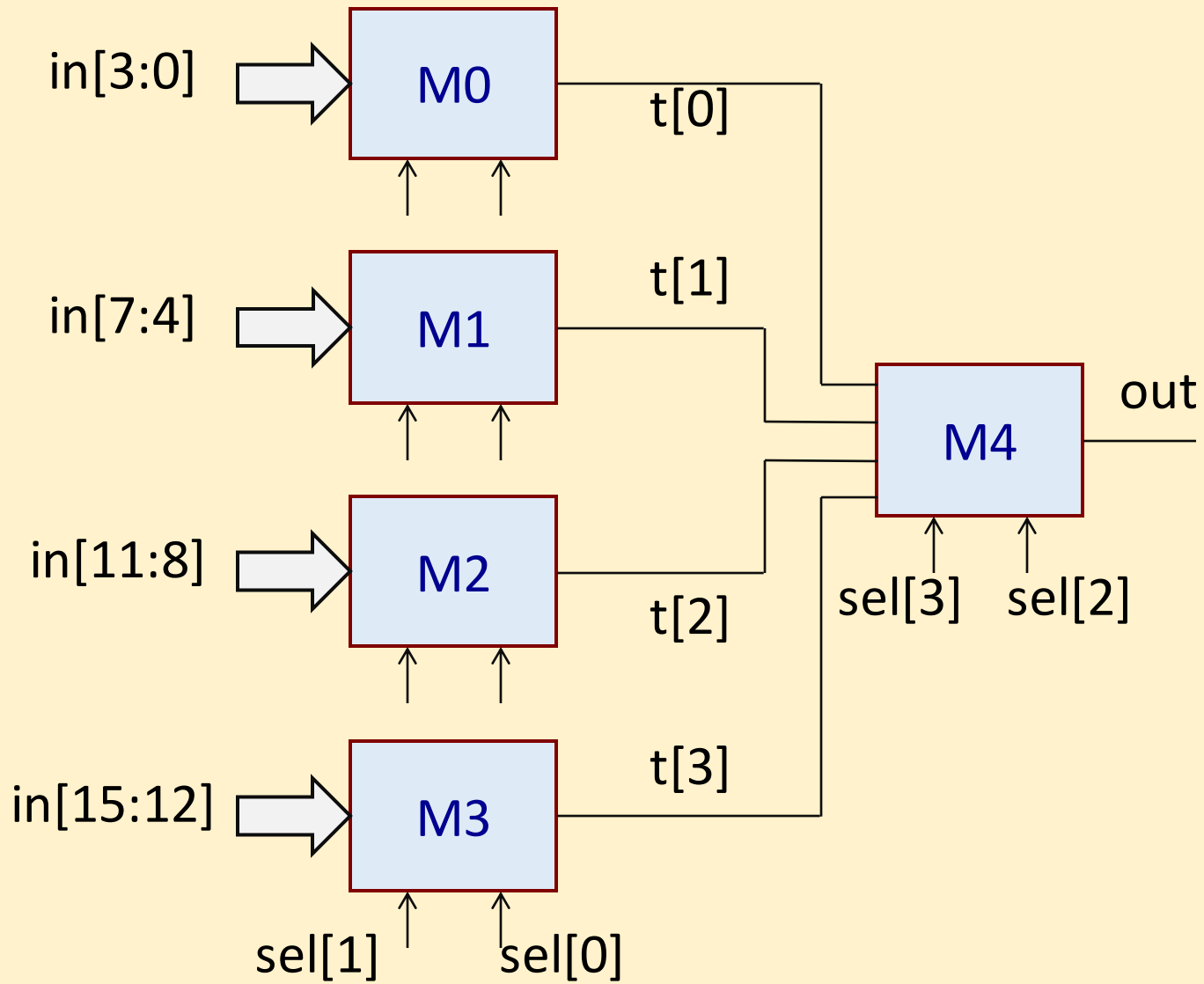
0	A=xxxx,	S=x,	F=x
5	A=3f0a,	S=0,	F=0
10	A=3f0a,	S=1,	F=1
15	A=3f0a,	S=6,	F=0
20	A=3f0a,	S=c,	F=1

Version 2: Behavioral modeling of 4-to-1 MUX

Structural modeling of 16-to-1 MUX

```
module mux4to1 (in, sel, out);  
    input [3:0] in;  
    input [1:0] sel;  
    output out;  
  
    assign out = in[sel];  
endmodule
```

```
module mux16to1 (in, sel, out);  
    input [15:0] in;  
    input [3:0] sel;  
    output out;  
    wire [3:0] t;  
  
    mux4to1 M0 (in[3:0], sel[1:0], t[0]);  
    mux4to1 M1 (in[7:4], sel[1:0], t[1]);  
    mux4to1 M2 (in[11:8], sel[1:0], t[2]);  
    mux4to1 M3 (in[15:12], sel[1:0], t[3]);  
    mux4to1 M4 (t, sel[3:2], out);  
endmodule
```



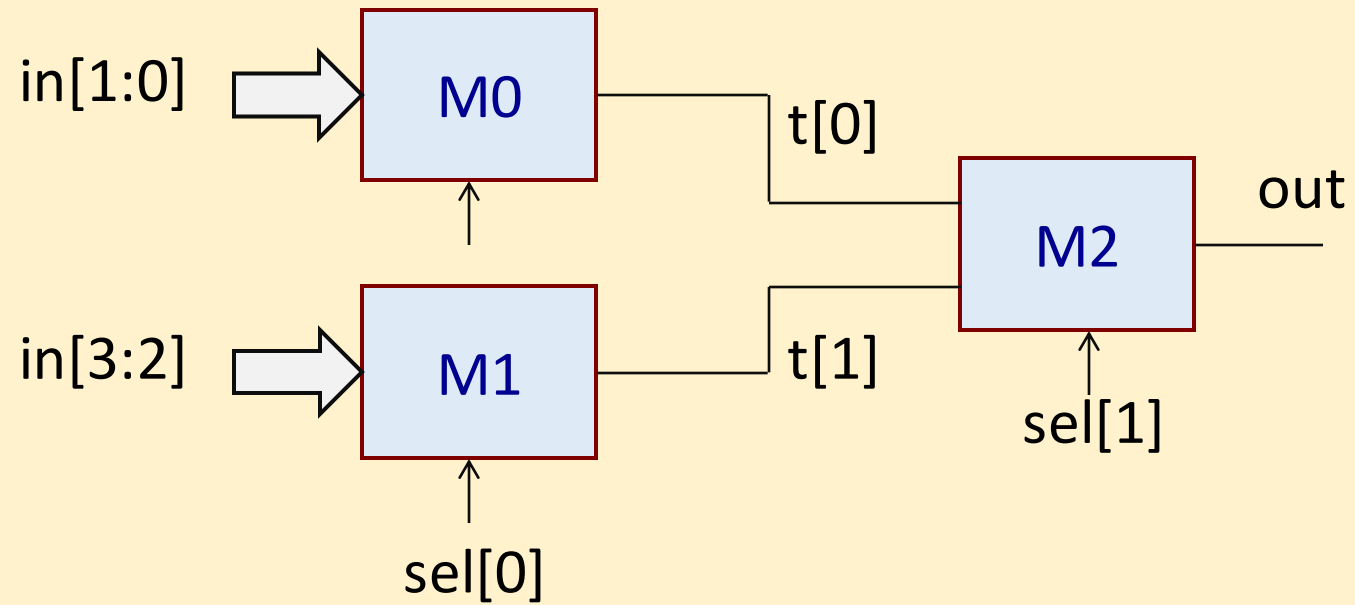
**16-to-1 multiplexer
using 4-to-1
multiplexers**

Version 3: Behavioral modeling of 2-to-1 MUX

Structural modeling of 4-to-1 MUX

```
module mux2to1 (in, sel, out);  
    input [1:0] in;  
    input sel;  
    output out;  
  
    assign out = in[sel];  
endmodule
```

```
module mux4to1 (in, sel, out);  
    input [3:0] in;  
    input [1:0] sel;  
    output out;  
    wire [1:0] t;  
  
    mux2to1 M0 (in[1:0], sel[0], t[0]);  
    mux2to1 M1 (in[3:2], sel[0], t[1]);  
    mux2to1 M2 (t, sel[1], out);  
endmodule
```

**4-to-1 multiplexer
using 2-to-1
multiplexers**

Version 4: Structural modeling of 2-to-1 MUX

```
module mux2to1 (in, sel, out);  
    input [1:0] in;  
    input sel;  
    output out;  
    wire t1, t2, t3;  
  
    NOT G1 (t1, sel);  
    AND G2 (t2, in[0], t1);  
    AND G3 (t3, in[1], sel);  
    OR G4 (out, t2, t3);  
endmodule
```

Point to note:

- Same test bench can be used for all the versions.
- The versions illustrate hierarchical refinement of design.