Verilog Tutorial-2

COMBINATIONAL CIRCUIT

Vector

• Nets or “reg” type variable can be declared as vectors, of mulCple 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;
```

8 bit adder

```
// An 8-bit adder description
module paralle_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
```

Test bench

```
 module tb_8bit Adder;
  reg [7:0] A, B;
  reg C;
  wire [7:0] SUM;
  wire Cr;
  adder8bit fa8 (.sum( SUM), .cr( Cr),.a( A), .b( B), .c( C));
 initial
   begin
     $dumpfile (“8bitadder.vcd");
     $dumpvars (0,8bitadder);
     $monitor (“$time A=%d B=%d C=%c SUM=%d Cr=%d”, A, B, C, SUM, Cr);
    #0 A=4’b0000; B=4’b0000; C=1’b0;
    #10 A=4’b0100; B=4’b0011; C=1’b1;
    #20 A=4’b0011; B=4’b0111; C=1’b1;
    #30 A=4’b1000; B=4’b0100; C=1’b0;
    #40 A=4’b0101; B=4’b0101; C=1’b1;
  end
endmodule
```

2 bit comparator

```
// A 2-bit comparator
 module compare (A1, A0, B1, B0, lt, gt, eq);
 input A1, A0, B1, B0;
 output reg lt, gt, eq;
 always @ (A1, A0, B1, B0)
    begin
        lt = ({A1,A0) < {B1,B0});
        gt = ({A1,A0) > {B1,B0});
        eq = ({A1,A0) == {B1,B0});
    end
endmodule
```

N-bit comparator

```
// An n-bit comparator
module compare (A, B, lt, gt, eq);
 parameter word_size = 16;
 input [word_size-1:0] A, B;
 output reg lt, gt, eq;
    always @ (*)
      begin
      gt = 0;
      lt = 0; eq = 0;
      if (A > B) gt = 1;
      else if (A < B) lt = 1;
      else eq = 1l
     end
endmodule
```

ALU

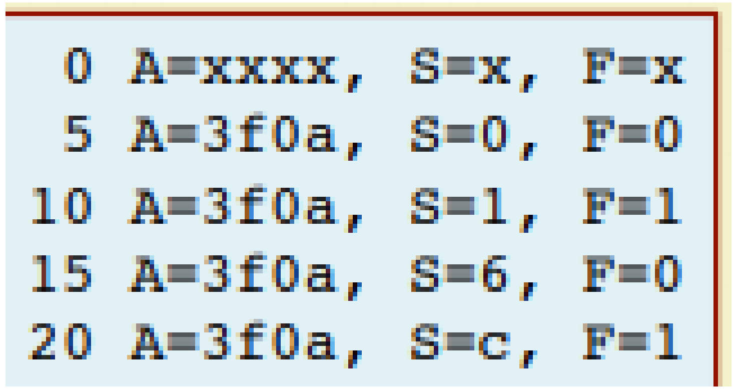
```
// A simple 4-function ALU
module ALU_4bit (f, a, b, op);
 input [1:0] op;
 input [7:0] a, b;
 output reg [7:0] f;
 parameter ADD=2’b00, SUB=2’b01, MUL=2’b10, DIV=2’b11;
 always @(*)
    case (op)
      ADD : f = a + b;
      SUB : f = a – b;
      MUL : f = a * b;
      DIV : f = a / b;
   endcase
endmodule
```

Multiplexer

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

Test bench for mux

```
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
```

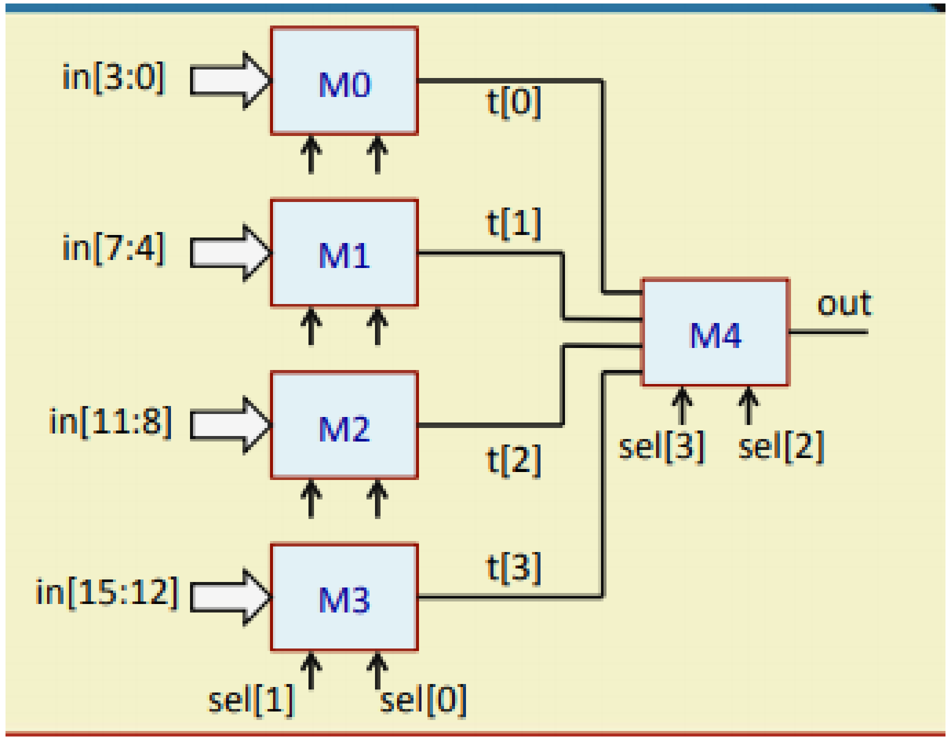


```
 0 A=xxxx, S=x, F=x
 5 A=3f0a, S=0, F=0
10 A=3f0a, S=1, F=1
15 A=3f0a, S=6, F=0
20 A=3f0a, S=c, F=1
```

Another kind of behavioural modelling

```
// A combinational logic example
module mux21 (in1, in0, s, f);
 input in1, in0, s;
 output reg f;
 always @(*)
    if (s)
       f = in1;
    else
      f= in0;
endmodule
```

If we want to do structural modelling



To make a structural modelling of large circuit we use the technique of hierarchical model and break down in such a way that the simplest form is easy to model structurally.

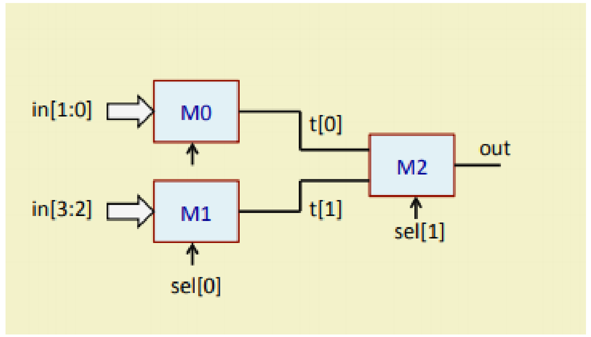
16x1 using 4x1 mux

```
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
```

4x1 mux using 2x1 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
```

Simplest module (2x1 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
```

Simple decoder (try to build larger decoder)

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

Thank You