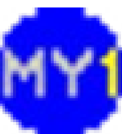
Introduction to Verilog and ModelSim

(Part 5 – Sequential Logic)



Sequential Logic

- It implements storage capabilities of the system
- Data latch structure
  - Requires clock/gate/latch signal input
  - Latch (level triggered)
  - Flip-flop (edge triggered – sequential latch?)
  - Setup-hold delay
- Allows data to be processed in stages

MY1

Concurrency and sequential

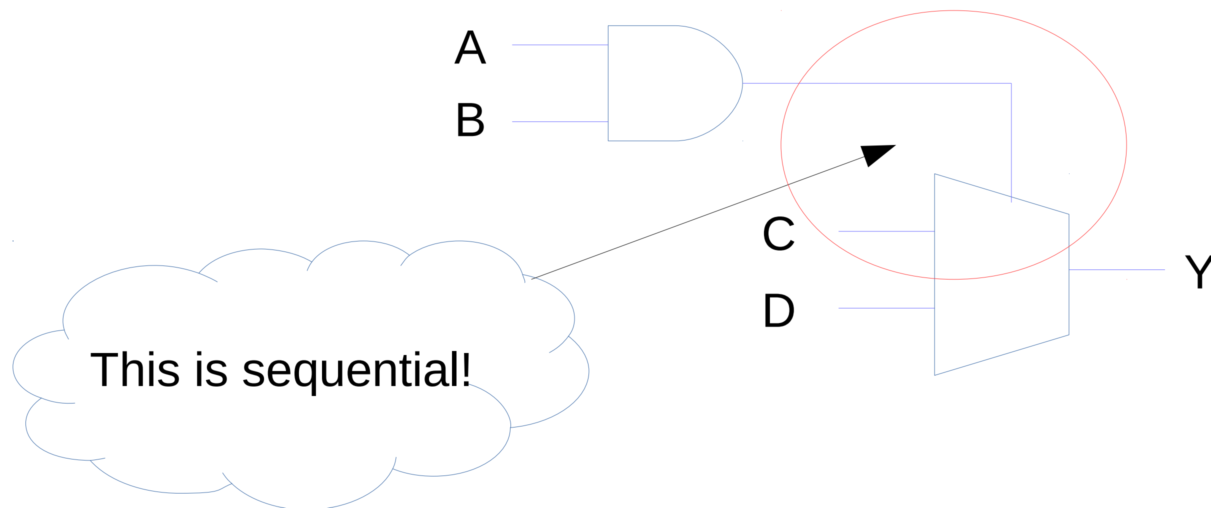
- Verilog coding concept
  - Hardware IS concurrent (continuously charging / discharging)
  - Sequential here refers to signal propagation, NOT sequential logic
- 'assign' statement and 'always' blocks are concurrent
- Codes inside 'always' block can be concurrent or sequential

Can be more than one always block in a module

MY1

Concurrency and sequential (cont.)

- Codes in 'always' block: if-else statements
  - sequential statements to represent propagation of signals (sometimes described as code priority)
  - e.g. to evaluate Y, output from AND gate must be resolved before hand



MY1

Concurrency and sequential (cont.)

- Codes in 'always' block: case/casex statements
  - Similar to switch-case in C (alternative to if-else)
  - Usually used to represent multiplexers
  - casex being used when 'Z' and 'X' needed

MY1

Concurrency and sequential (cont.)

- Maintaining previous value
  - Usually occurs in if-without-else statements
  - Indirectly means we need a storage
  - Known as 'latch inference' (a latch is inferred)
- How to avoid latch inference?
  - Provide an else statement @ specify all possible conditions
  - Avoid using if-else statements (separate combinational logic) → use conditional operator in 'assign' statements

MY1

Suggested Coding Style

- Separate combinational logic from sequential logic
  - Follows 2-process (VHDL) design
- Use 'assign' statements for combinational logic
  - 'wire'-type variable (signal)
  - Bitwise operators only
- Use 'always' block for sequential logic
  - 'reg'-type variable (signal)
  - Use unblocking statements ('<=') instead of ('=')
  - Only clock signal in the sensitivity list

Registers

- An array of flip-flops (usually DFF)
  - Microprocessor internal storage
  - Size usually depends on internal bus size
- Arranged in a group to form a register file
  - More registers usually allow faster complex calculations (no memory access needed)
- Register addressing usually part of instruction
  - Trade-off number of registers ↔ instruction size

MY1

Try this!

NEW! 'parameter'

NEW! 'defparam'

```
module register (clock,enable,
    data_in,data_out);

parameter BITS=8;

input clock, enable;
input [BITS-1:0] data_in;
output [BITS-1:0] data_out;
reg [BITS-1:0] data_out;

always @(posedge clock)
begin
    if (enable==1)
    begin
        data_out <= data_in;
    end
end

endmodule
```

```
module register_tb ();
parameter SIZE=4;
reg clk, enb;
reg[SIZE-1:0] idat;
wire[SIZE-1:0] odat;
// reset block
initial begin
    clk = 1'b0; enb = 1'b0;
end
// generate clock
always begin
    #(5) clk = !clk;
end
//generate stimuli
always begin
    #(10); enb = 1'b0; idat = 4'ha;
    #(10); enb = 1'b1;
    #(10); enb = 1'b0; idat = 4'h5;
    #(10); enb = 1'b1;
    #(10); enb = 1'b0;
end
defparam dut.BITS = SIZE;
register dut (clk,enb,idat,odat);
endmodule
```

MY1

Or… Try this!

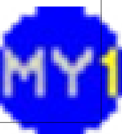
Verilog-2001 style

```
module register
#( // parameter block
    parameter BITS=8
)
( // ports block
    input clock,
    input enable,
    input[BITS-1:0] data_in,
    output reg[BITS-1:0] data_out
);

always @(posedge clock)
begin
    if (enable==1)
    begin
        data_out <= data_in;
    end
end

endmodule
```

```
module register_tb ();
parameter SIZE=4;
reg clk, enb;
reg[SIZE-1:0] idat;
wire[SIZE-1:0] odat;
// reset block
initial begin
    clk = 1'b0; enb = 1'b0;
end
// generate clock
always begin
    #(5) clk = !clk;
end
//generate stimuli
always begin
    #(10); enb = 1'b0; idat = 4'ha;
    #(10); enb = 1'b1;
    #(10); enb = 1'b0; idat = 4'h5;
    #(10); enb = 1'b1;
    #(10); enb = 1'b0;
end
register #(SIZE) dut (clk,enb,idat,odat);
endmodule
```



Addressing a register

- Basically needs to demultiplex input and multiplex output
  - Mux/demux selector signals are addresses
  - Some registers can be customized to have shift/increment/decrement logic
- Common signals for register file block
  - wenb (write enable), renb (read enable)
  - data_in , data_out (data bus size)
  - clock, address (internal address bus size)

MY1

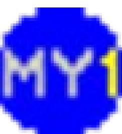
Practical Session 5.1

- Implement 4 x 8-bit register file block
  - Must have all the previously mentioned signals
- Create a suitable testbench for the modules
- Simulate and analyze the results to verify the functionality of the modules

MY1

Self-checking Testbench

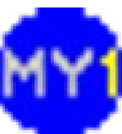
- Monitoring waveforms for verification
  - Not so bad if small number of cases, e.g. 4-bit input
  - Visually verify 256-possibilities? Let's get smart!
- We need self-checking testbench!
  - Verilog system tasks ($display,$time,$stop)
    - $write is like $display, but it does not auto-insert new-line char at the end
    - $finish is like $stop, but also terminates ModelSim
  - Verilog language construct (task, function)



Verilog System Task: Example

- Using $display, $time, $stop
  - $display : equivalent to printf in C
  - $time : provide current simulation time value
  - $stop : breaks the simulation run

```
module demo_tb ();
initial begin
    #10;
    $display("[%d] Begin Simulation.", $time);
    #10;
    $display("[%d] End Simulation.", $time);
    $stop;
end
endmodule
```



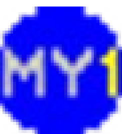
Verilog Task

- Task is a sub-routine definition
  - Specified in a module (can be 'included' elsewhere)
  - Can have both input/output parameters (local) – can also use/drive global variables
  - Can include timing delays (# delay, wait, etc.)
  - Called as statements (not in expressions – does not 'return' value)
  - Can call other tasks/functions

```
task <task-label>;
// parameters here
begin
// code here
end
endtask
```

Verilog Task: Example

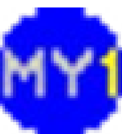
```
module tdemo_tb ();
    task exec_loop;
        input integer size;
        integer loop;
        begin
            $display("[%d] Starting loop...", $time);
            loop = 0;
            while(loop<size)
            begin
              loop = loop + 1;
                $display("[%d] Loop=%d", $time, loop);
                #(10);
            end
        end
    endtask
    // run task!
    initial begin
        #10;
        $display("[%d] Begin Simulation.", $time);
        exec_loop(20);
        $display("[%d] End Simulation.", $time);
        $stop;
    end
endmodule
```



Verilog Function

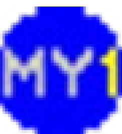
- Function similar to task, but
  - Can have input parameters only and 'returns' one value (can be used in expressions)
  - Cannot have timing delays (combinational logic only)
  - Can call other functions but not tasks
  - Return value assigned to function name

```
function <function-label>;
// parameters here
begin
// code here
// <function-label> = ??
end
endfunction
```



Verilog Function: Example

```
module fdemo_tb ();
    integer loop; reg [1:0] test; reg [3:0] look;
    function[3:0] doDecode;
        input[1:0] inCode;
        reg[3:0] doCode;
        begin
            doCode = 2'bXX;
            case(inCode)
                2'b00: doCode = 4'b0001;
                2'b01: doCode = 4'b0010;
                2'b10: doCode = 4'b0100;
                2'b11: doCode = 4'b1000;
            endcase
            doDecode = doCode;
        end
    endfunction
    // test function!
    initial begin
        for (loop=0;loop<4;loop=loop+1) begin
            test = loop; look = doDecode(test);
            $display("%b => %b",test, look);
        end
        $stop;
    end
endmodule
```



Practical Session 5.2

- Implement 16-bit ALU with 16 x 16-bit registers
  - Arithmetic (ADD,SUB,MUL,DIV)
  - Logic (NOT, AND, OR, XOR)
  - Use tasks to populate registers
- Create a suitable testbench for the module(s)
- Simulate and analyze the results to verify the functionality of the ALU and register access/assignments