N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2010

Verilog (II)

Outline

- Behavioral Models of Sequential Logics
  - Latches and Flip-flops
  - Procedural assignment
  - Non-blocking assignment
- Basic Finite State Machine
  - Moore and Mealy
  - State Diagram
  - EXM2 and EXM3
  - Controller FSM

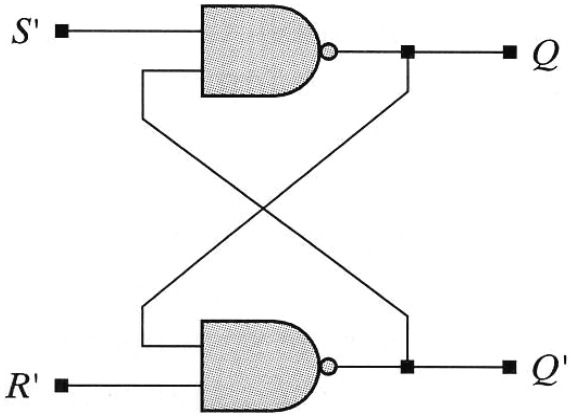
Behavioral Models of Sequential Logic

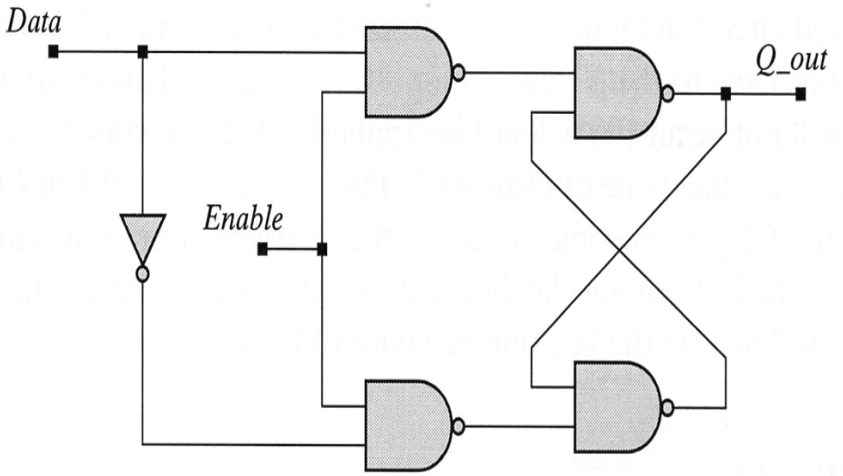
Latches and Flip-flops

Procedural assignment

Non-blocking assignment

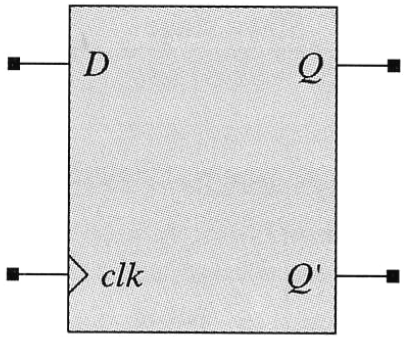
Latches



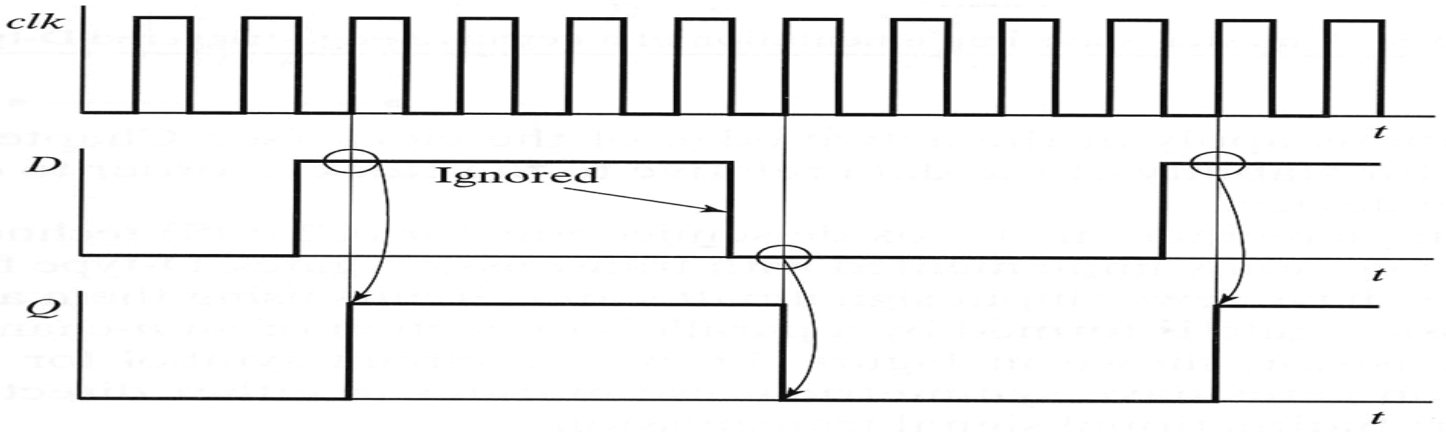


S'	R'	Q_{next}	Q'_{next}	
0	0	1	1	Not allowed
0	1	1	0	Set
1	0	0	1	Reset
1	1	Q	Q'	Hold

Edge-Triggered D Flip-flop



D	Q	Q_{next}
0	0	0
0	1	0
1	0	1
1	1	1

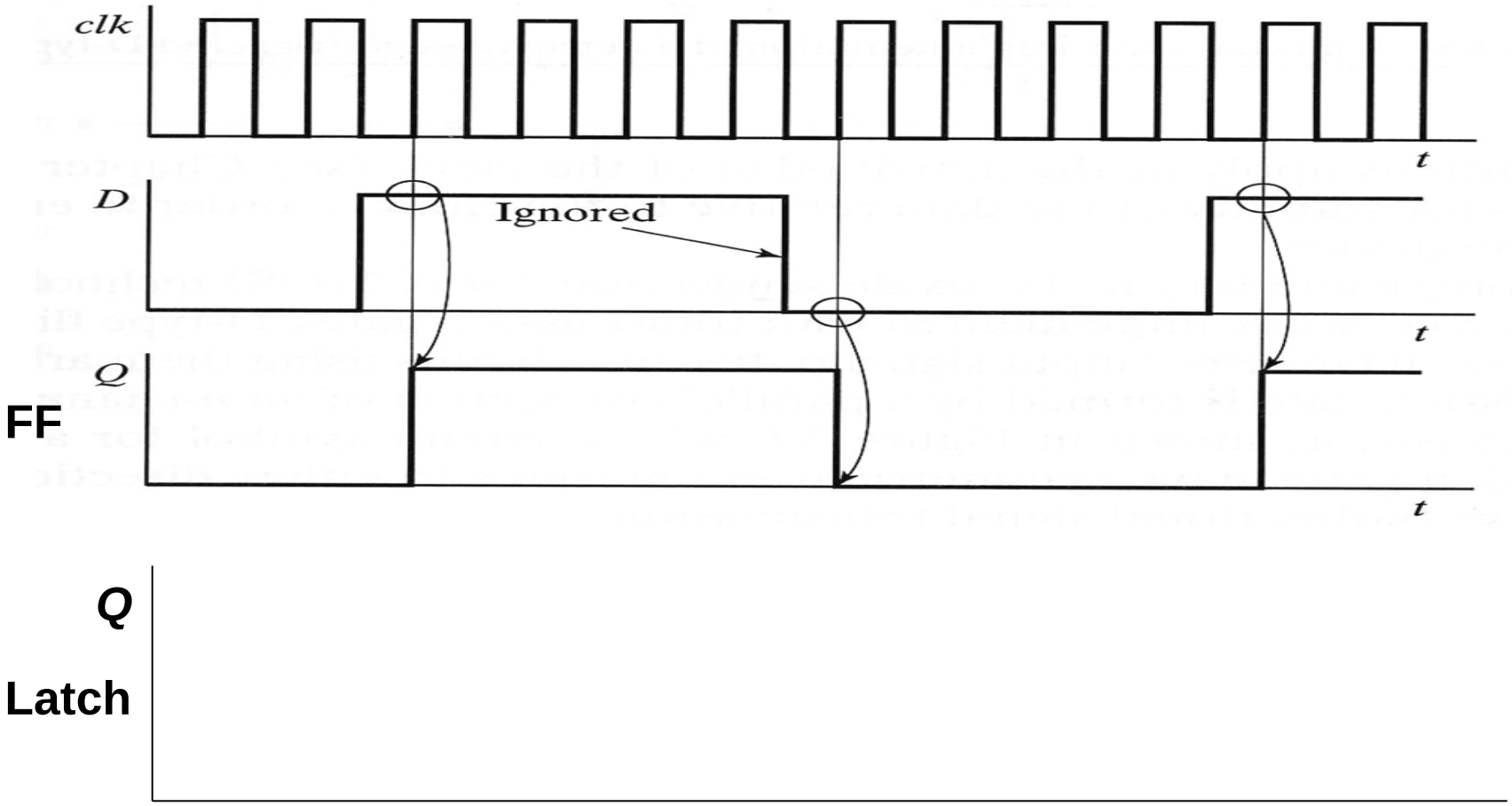


Differences between a latch and a flip-flop(1/2)

- A latch does not have a clock signal
- Latches does not need synchronized with the clock, i.e., they are asynchronous.

  -- the output changes soon after the input changes
- A flip-flop is a synchronous version of the latch
- Most of the digital designs today are synchronous

Differences between a latch and a flip-flop (2/2)

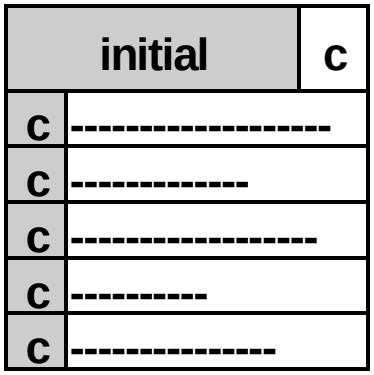


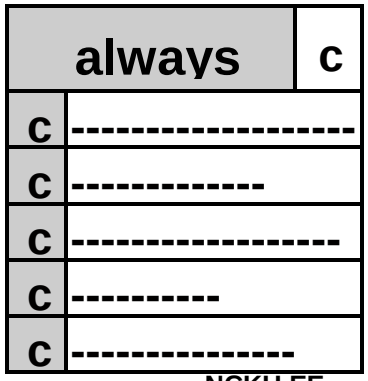
Behavioral DFF


```
module df_behav (q, q_bar, data, set, reset, clk);
input           data, set, clk, reset;
   output             q, q_bar;
   reg             q;

   assign q_bar = ~ q;

   always @   (posedge clk)   // Flip-flop with synchronous set/reset
   begin
      if (reset == 0) q <= 0;
         else if (set ==0) q <= 1;
            else q <= data;
   end
endmodule
```

Procedural Blocks

- Procedural blocks are the basis for behavioral modeling
- Procedural blocks are of two types
  -
  -
- Procedural blocks have the following components
  - **Procedural assignment statements**
  - **Timing controls**
  - **High-level programming language constructs.**





Coding Notes

- Edge-sensitive behavior
  -
  - Use procedural statements, statements of an ordinary procedural language (e.g., C)
- Procedural assignment operator
  - operator(=)
  -
  - Register variables store information during simulation, NOT necessary implying hardware registers

Procedural assignment

- Blocked assignment
- = is the assignment operator. It copies the value of the RHS of the expression to the LHS.
- A statement must complete execution before the next statement in the behavior can execute

Nonblocking (concurrent) assignment

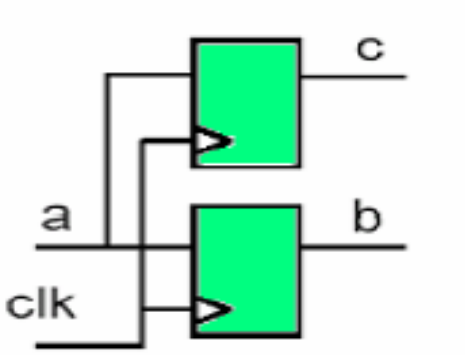
- Operator(<=)
- Statements will be execute concurrently (in parallel), NOT sequentially. So the order in which they are listed has no effect.

```
module swap_vals;
        reg a, b, c;
        initial
        begin
                a = 0;
                b = 1;
                c = 0;
        end
        always #5 c = ~c;
        always @(posedge c)
        begin
                a <= b; //Nonblocking procedural assignment
                b <= a; //swaps the values of a and b.
        end
endmodule
```

Blocking and Non-Blocking Assignments

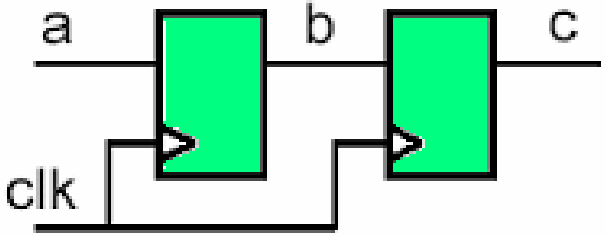
Blocking

```
always @(posedge clk)
 begin
    b = a;
    c =  b;
end
```



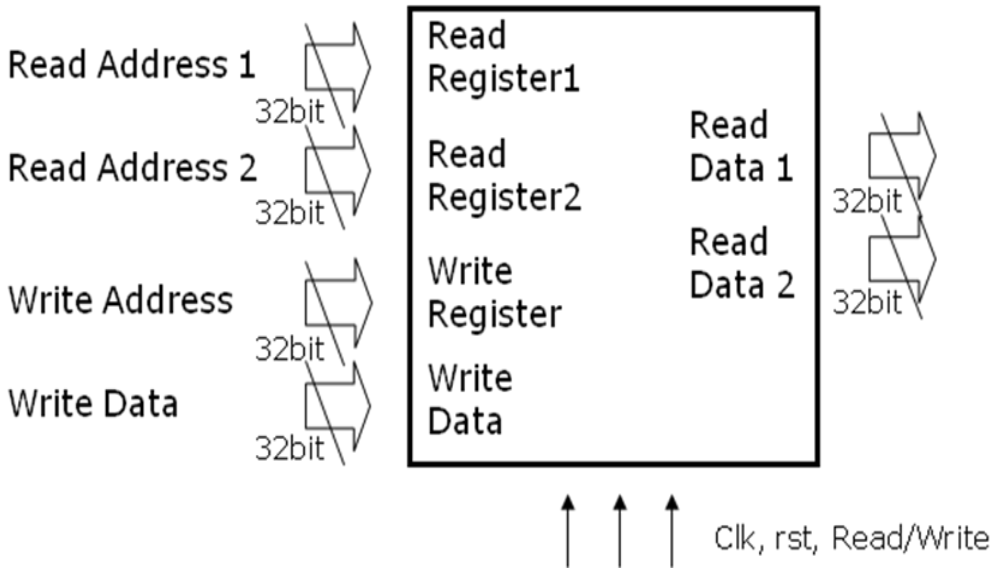
Non-blocking

```
always @(posedge clock)
 begin
    b <= a;
    c <= b;
end
```



Register File (1/2)


```
`timescale 1ns/10ps
module Mreg_4for32(OUT_1, OUT_2, Write, Read, Read_ADDR_1,
Read_ADDR_2, Write_ADDR, DIN, enable, clk, rst);

parameter ADSize = 13;                    //13 bits
parameter REGSize = 32;                   //32
parameter DASize = 32;                    //32bits

input clk;
input rst;
input enable;

input [ADSize-1:0]Read_ADDR_1;
input [ADSize-1:0]Read_ADDR_2;
input [ADSize-1:0]Write_ADDR;

input [DASize-1:0]DIN;

input       Write;
input       Read;

output [DASize-1:0]OUT_1;
output [DASize-1:0]OUT_2;

reg   [DASize-1:0]OUT_1;
reg   [DASize-1:0]OUT_2;

reg   [DASize-1:0]  Mreg[REGSize-1:0];

integer i;
```



Register File (2/2)


```
always@(posedge clk)begin
                    if( rst )begin
                    for(i=0 ; i<REGSize ; i=i+1)
                    Mreg[i] = 0;
                    end

                    else
                    begin
                     if(enable)begin
                        if( Write )
                        Mreg[Write_ADDR] = DIN;
                        else if( Read )begin
                        OUT_1 = Mreg[Read_ADDR_1];
                        OUT_2 = Mreg[Read_ADDR_2];
                        end
                     end
                    end
end

endmodule
```

Parallel and Serial Statements

- Continuous assignment statements

```
assign  A_lt_B = (A < B);
assign  A_gt_B = (A > B);
assign  A_eq_B = (A == B);
```

- Procedural block

```
always @ (reset or enable or data)
  begin
            if(reset)  q_out = 0;
               else if (enable)
                     q_out = data;
  end
```

Parallel and Serial Statements

- Mixed continuous, procedural, gate-level

```
assign mux_out = enable ? mux_int : 32'bz;
always @ (data_3 or data_2 or data_1 or data_0 or select)
    begin
      ....................
    end
xor       mxor (mux_int, a1, b1);
add_full  mfadd (a1, c_in, y, z);
```

Basic Finite State Machine

Moore and Mealy

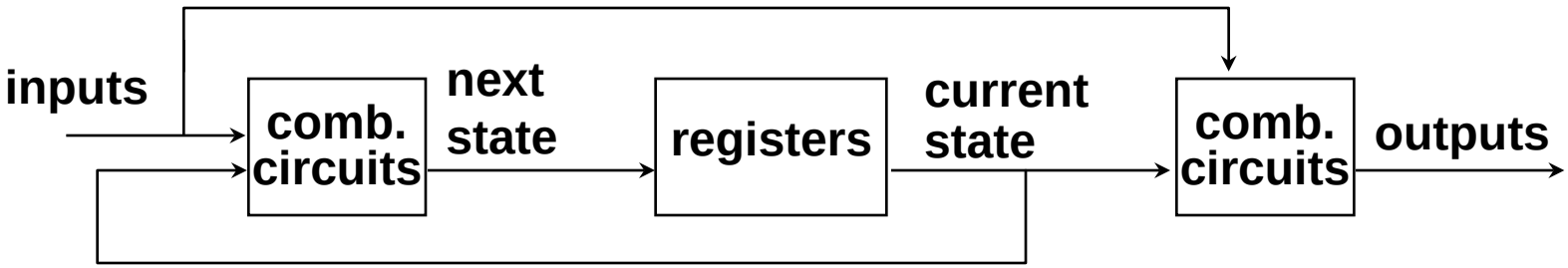
Controller FSM

Read Memory

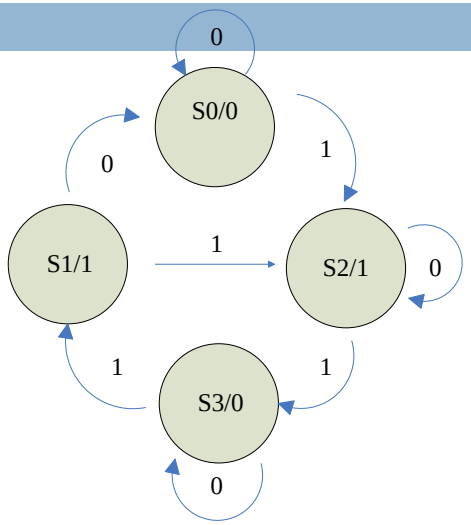
Moore and Mealy Machines

- **Moore model**

- **Mealy model**



Moore Machine



Current State	Next State		Qout
	Din=0	Din=1	
S0=00			0
S1=01			1
S2=10			1
S3=11			0

- Use binary numbers to represent each state

  parameter [1:0] S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
- Reading data input and changes state for every clock cycles
- Upon reset, machine goes back to state S0
- Next state will be determined by current state and input

Verilog Code (1/3)


```
`timescale 1ns/10ps
module moore (Qout, clk, rst, Din);
output Qout;
input clk, rst, Din;
parameter [1:0] S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
reg Qout;
reg [1:0] CS, NS;   // CS: current state; NS: next state

always @ (posedge clk or posedge rst)
begin
if (rst==1'b1) CS=S0;
else CS = NS;
end
```

Verilog Code (2/3)

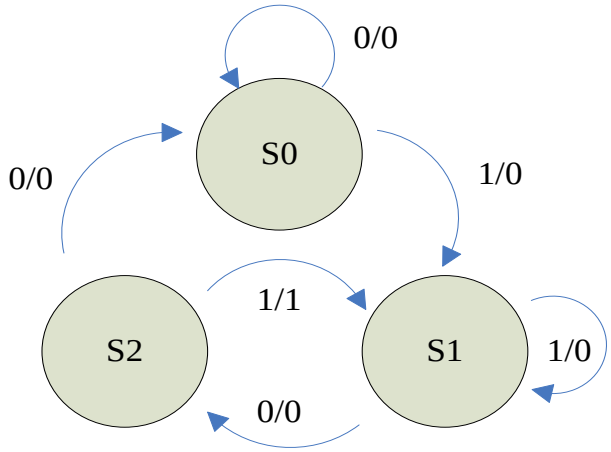

```
always @ (CS or Din)
begin
case (CS)
  S0:   begin
           Qout=1'b0;
           if (Din==1'b0) NS=S0;
           else NS = S2;
         end
  S1:   begin
           Qout=1'b1;
           if (Din==1'b0) NS=S0;
           else NS = S2;
        end
```

Verilog Code (3/3)


```
  S2:  begin
          // fill the rest of the code
       end
  S3:  begin
          // fill the rest of the code
       end
   endcase
end // always(cs or din)
endmodule
```

Mealy Machine

Construct the state diagram and state table as below:

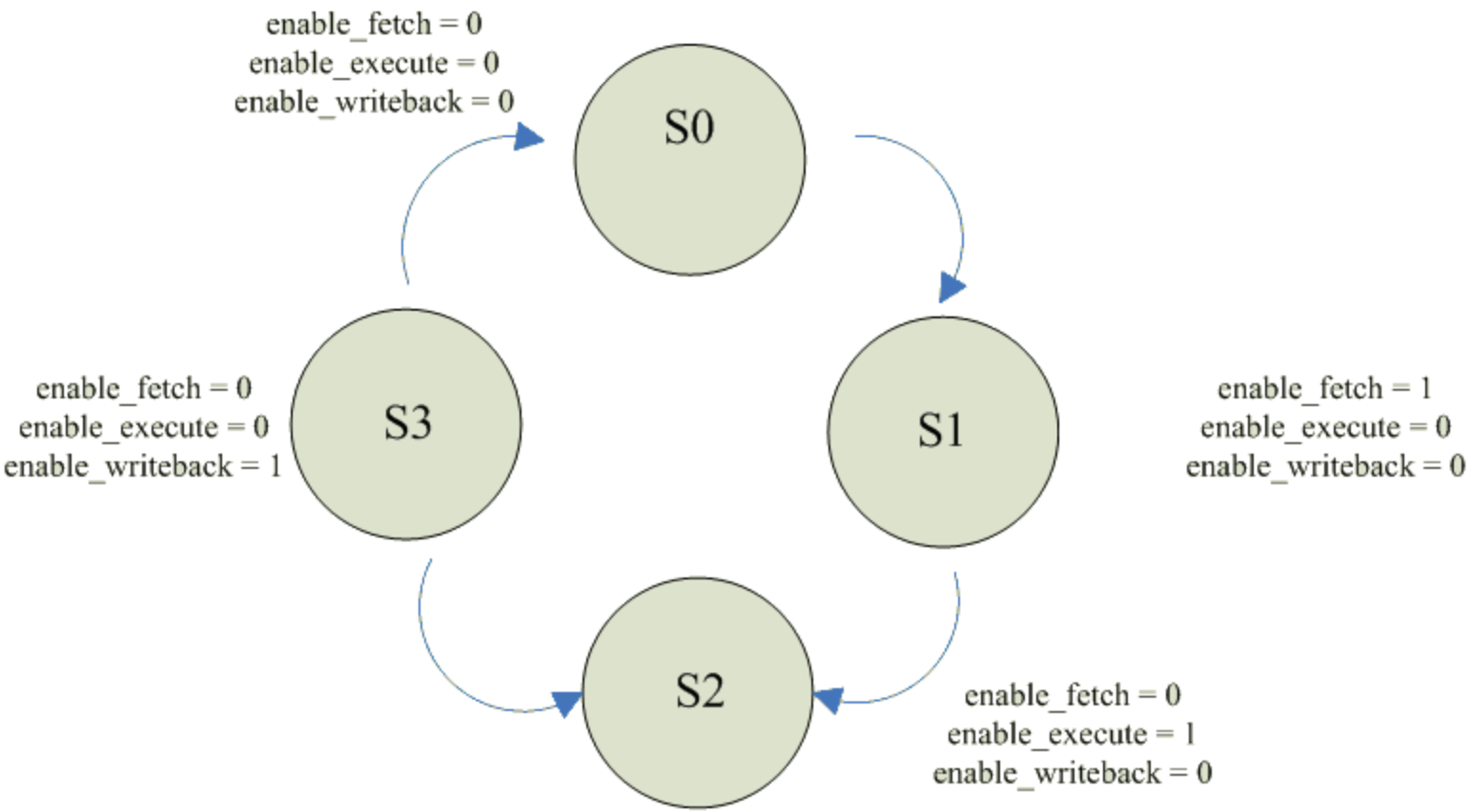


Current State	Next State	
	Din=0	Din=1
S0=00	S0,0	S1,0
S1=01	S2,0	S1,0
S2=11	S0,0	S1,1

Reference Code


```
always@(CS or Din)
begin
   case (CS)
      S0: begin
            if (Din==1'b0)
            begin
               NS=S0;
               Qout=1'b0;
            end
            else begin
               NS = S1;
               Qout=1'b0;
            end
          end
      S1:begin
            ...     //complete the rest of the code
         end
      S2:begin
            ...     //complete the rest of the code
         end
    default:
            begin
            ...
            end
      endcase
end
endmodule
```

Controller FSM (1/3)



Controller FSM (2/3)


```
module ctrl_state(clk, reset, enable_fetch,
enable_execute, enable_writeback, ir, PC_adjust, pc);
`define OPCODE  ir[31:28]
`define SRCTYPE ir[27]    //1= imm; 0 = reg
`define DSTTYPE ir[26]    //1 = Data Mem; 0 = reg
`define SRC     ir[25:13]
`define DST     ir[12:0]
input [31:0] ir;
input [4:0]  pc;
input clk,reset;
output enable_fetch, enable_execute;
output enable_writeback, PC_adjust;
reg  enable_fetch, enable_execute;
reg  enable_writeback, PC_adjust;
reg  [1:0] current_state, next_state;
reg  [31:0] pre_opcode;

parameter S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;

always @(posedge clk)
begin
  if(reset) begin
    current_state=S0;
  end
  else begin
    current_state=next_state;
  end
end
```

Controller FSM (3/3)


```
always @(current_state)
begin
  case(current_state)
    2'b00: begin
          next_state=S1;
          enable_fetch=0;
          enable_execute=0;
          enable_writeback=0;
        end
                .
                .
                .
    default: next_state=S0; //avoid unknown state
  endcase
end

always @(posedge enable_fetch)
begin
  if(pc==0) pre_opcode=0;
  else pre_opcode=ir;
end
```

Loading a Memory Array

- Assigning values to each word of the memory array.
- Call to $readmem system task to initialize memory from file

MUST

```
$readmem<base>("<filename>",<mem_name>,<start>,<finish>);
```

optional

```
parameter memsize = 256;
reg [7:0] mema [memsize-1:0];
...
initial
        for (i = 0; I < memsize; i = i + 1)
              mema[i] = 8'b0;
...

initial
   $readmemb("mem_file.txt", mema);
        ....
```

$readmemb and &readmemh

- $readmemb("mem_file.txt",mem);

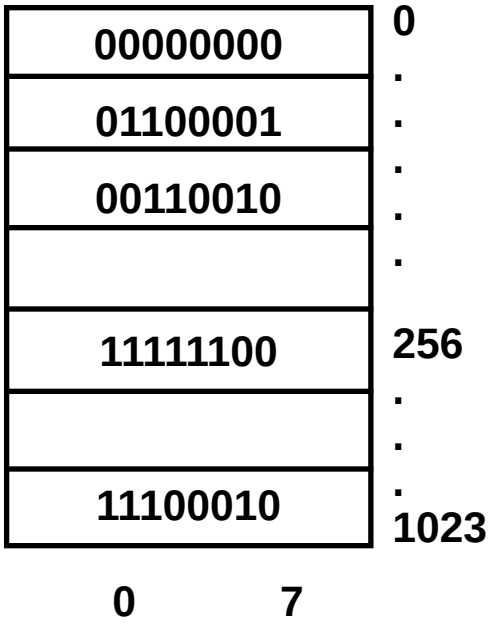
UNIX Text File

mem_file.txt

```
0000_0000
0110_0001   0011_0010
//addresses 3-255 are not
//defined
@100
1111_1100
/*addresses 257-1022 are
not defined */
@3FF
1110_0010
```

Declared Memory Array

reg[0:7] mem [1023:0]



Memory Addressing

- A memory element is addressed by giving the location to the memory array.

```
reg [8:1] mema [0:255]; //declare memory called mema
reg [8:1] mem_word; // temp register called mem_word
...
// Display contents of the 6th memory address
$displayb(mema[5]);
...
//Display the msb of the 6th memory word
mem_word = mema[5];
$displayb(mem_word[8]);
...
```

// It is incorrect to use mema[5][8];

Memory


```
module memory(data,addr,read,write);
input read,write;
input [4:0] addr;
inout [7:0] data;
reg [7:0] data_reg;
reg [7:0] memory [0:8'hff];
parameter load_file = "cputl.txt";
assign data=(read)?memory [addr]:8'hz;
always @(posedge write)
    memory[addr]=data;
initial
    $readmemb(load_file,memory);
endmodule
```

Reading in Verification Vector Files


```
module test;
  parameter N =5;
  ....
  reg [7:0] mult1 [0:N], m1;
  reg [7:0] mult2 [0:N], m2;
  reg [15:0] prod [0:N];
  wire [15:0] p;
  ....
  init begin
     $readmemh("s1_vec.dat", mult1);
     $readmemh("s2_vec.dat", mult2);
      $readmemh("prod_vec.dat", prod);
   end
  init begin
     reset =1, clock = 0, load = 0;
   for (i=0;i<=N;i=i+1)  begin
        m1 = mult1[i];
        m2 = mult2[i];
        #10   load = 1;
        #10   load = 0;
        #120  if (p!=prod[i])
            $display("Error! @ %i\n", i);
    end
    $finish;
  end
always #5 clock = ~ clock;

PSMult u1 (clock, reset, load, m1,m2,p);
endmodule
```

s_vec.dat


```
// vector inputs
73hf
beef
1234
5678
9ABC
BEF0
FFFF
0000
7923
8943
```

Looping Statements

- For Loops (within procedural blocks or generate blocks)

```
for (index = 0; index < size; index = index + 1)
    if (val[index] === 1'bx)
        $display ("found an X");
```

```
for (i = 10; i > index; i = i - 1)
    memory[i] = 0;
```

-- repeat (num_Of_execution) begin …; …; …;…; end

-- while (control_expression) begin …; …; …;…; end

For Loop

- Provide a shorthand way of writing a series of statements
- Loop index variable must be integer type
- Step, start & end value must be constant
- In synthesis, for-loop loops are “unrolled”, and then synthesized

```
always @(a or b) begin
for (k = 0; k <=3; k= k + 1)
   begin
     out[k] =a[k]^b[k];
     c = (a[k]|b[k])&c;
   end
end
```

```
out[0] = a[0] ^ b[0];
out[1] = a[1] ^ b[1];
out[2] = a[2] ^ b[2];
out[3] = a[3] ^ b[3];
c = (a[0]|b[0]) & (a[1]|b[1]) &
      (a[2]|b[2]) & (a[3]|b[3]) & c
```