N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2010

Verilog (III)

Outline

- Behavioral Modeling Examples of Combinational Logic and Sequential Logic
- Tasks and Functions

Behavioral Modeling Examples

Multiplexor, Decoder, Priority Encoder

Adder, Parity Checker

Counter, Shifter

SSRAM

FIFO

Multiplexor

Net-list (gate-level)

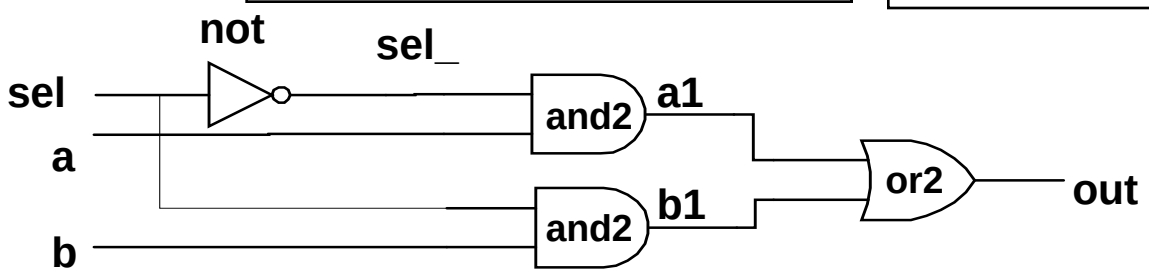
```
module
mux2_1(out,a,b,sel)
      output out;
      input a,b,sel;
      not(sel_,sel);

      and(a1,a,sel_);
      and(b1,b,sel);
      or(out,a1,b1);
endmodule
```

Continuous assignment

```
module
mux2_1(out,a,b,sel)
      output out;
      input a,b,sel;
      assign out=
       (a&~sel)|(b&sel);
endmodule
```

RTL modeling

```
module
mux2_1(out,a,b,sel)
      output out;
      input a,b,sel;
always @(a or b or sel)
if (sel)
   out = b;
else
   out = a;
endmodule
```



4-to-1 multiplexor

Continuous Assignment

- 4-to-1 multiplexor

```
module
mux4_1(out,in0,in1,in2,in3,sel) ;
     output out ;
     input in0,in1,in2,in3 ;
     input[1:0]sel ;
assign out =
        (sel ==2'b00)? in0 :
        (sel ==2'b01)? in1 :
        (sel ==2'b10)? in2 :
        (sel ==2'b11)? in3 :
                       1'bx;
endmodule
```

RTL

- 4-to-1 multiplexor

```
module mux4_1(out,in,sel) ;
     output out ;
     input[3:0] in ;
     input[1:0] sel ;
     reg out ;
  always @(sel or in)
      begin
         case(sel)
            2'd0: out = in[0];
            2'd1: out = in[1];
            2'd2: out = in[2];
            2'd3: out = in[3];
            default: 1'bx;
         endcase
       end
endmodule
```

```
always @(sel or in) out = in[sel];
```

Conditional Statements

- Case Statement (in procedural block)

```
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
```

Decoder

- **3-to-8 decoder with enable control**

```
module decoder(out,enb_,sel) ;
output [7:0] out ;
input enb_ ;
input [2:0] sel ;
reg [7:0] out ;
always @ (enb_ or sel)
   if(enb_)
     out = 8'b1111_1111 ;
   else
     case(sel)
         3'b000:out=8'b1111_1110 ;
         3'b001:out=8'b1111_1101 ;
         3'b010:out=8'b1111_1011 ;
         3'b011:out=8'b1111_0111 ;
         3'b100:out=8'b1110_1111 ;
         3'b101:out=8'b1101_1111 ;
         3'b110:out=8'b1011_1111 ;
         3'b111:out=8'b0111_1111 ;
         default:out=8'bx;
    endcase
endmodule
```

Priority Encoder I


```
always @ (d0 or d1 or d2 or d3)
   if(d3 == 1)
       {x,y,v} = 3'b111 ;
   else
     if(d2 == 1)
        {x,y,v} = 3'b101 ;
   else
     if(d1 == 1)
        {x,y,v} = 3'b011 ;
   else
     if(d0 == 1)
        {x,y,v} = 3'b001 ;
   else
       {x,y,v} = 3'bxx0 ;
```

inputs				outputs		
D_0	D_1	D_2	D_3	x	y	v
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Priority Encoder II (1/2)


```
module priority (Code, valid_data, Data);
  output       [2: 0]  Code;
  output       valid_data;
  input        [7: 0]  Data;
  reg          [2: 0]  Code;

  assign       valid_data = |Data; // "reduction or" operator
  always @  (Data)
    begin
      if (Data[7]) Code = 7; else
      if (Data[6]) Code = 6; else
      if (Data[5]) Code = 5; else
      if (Data[4]) Code = 4; else
      if (Data[3]) Code = 3; else
      if (Data[2]) Code = 2; else
      if (Data[1]) Code = 1; else
      if (Data[0]) Code = 0; else
                   Code = 3'bx;
    end
```

Priority Encoder II (2/2)


```
/*// Alternative description is given below
always @  (Data)
  casex (Data)
       8'b1xxxxxxx  : Code = 7;
       8'b01xxxxxx  : Code = 6;
       8'b001xxxxx  : Code = 5;
       8'b0001xxxx  : Code = 4;
       8'b00001xxx  : Code = 3;
       8'b000001xx  : Code = 2;
       8'b0000001x  : Code = 1;
       8'b00000001  : Code = 0;
       default  : Code = 3'bx;
  endcase
*/
endmodule
```

4-bit Adder


```
module fouradd(A,B,Ci,S,Co);
       input     [3:0]  A,  B;
       input     Ci;
       output    [3:0]  S;
       output    Co;
       reg       [3:0]  S;
       reg       Co;
       integer   i;

       always begin
              i=A+B+Ci;
              S = i;
              Co = i[4];

       end
endmodule
```

Or

```
assign {Co, S} = A + B + Ci;
```

Parity Checker


```
module parity_chk(data,parity);
input[0:7] data ;
output parity ;
reg parity ;
always @(data)
  begin:check_parity
    reg partial;
    integer n;
    partial=data[0];
    for(n=1;n<8;n=n+1)
      begin
        partial=partial^data[n];
      end
     parity<=partial;
  end
endmodule
```

Parity checker : a simple error detection method by adding an extra parity bit to a word of n bits.

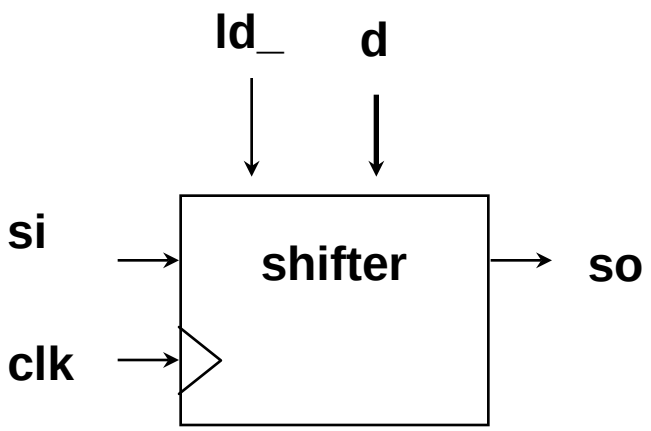
Even parity: If there are odd number of 1s, then the parity =1. If there are already even number of 1s, then the parity =0.

Counter


```
module bcd_count(count,ripple_out,clr,clk); //0-9 Counter
output [3:0] count;
output ripple_out;
reg [3:0] count;
input clr,clk;
wire ripple_out=(count==4'b1001)?0:1; //combinational
always @(posedge clk or posedge clr) //combi.+sequential
   if(clr);
     count=0;
   else if(count==4'b1001)
     count=0;
   else
     count=count+1;
endmodule
```

Shifter


```
module shifter(so,si,d,clk,ld_,clr_);
output so;
input [7:0] d;
input si,clk,ld_,clr_; //asynchronous clear & synchronous load
reg [7:0] q;
assign so = q[7];
always @ (posedge clk or negedge clr_)
  if(~clr_)
      q=0;
  else if (~ld_)
      q=d;
  else
      q[7:0]={q[6:0],si};
endmodule
```

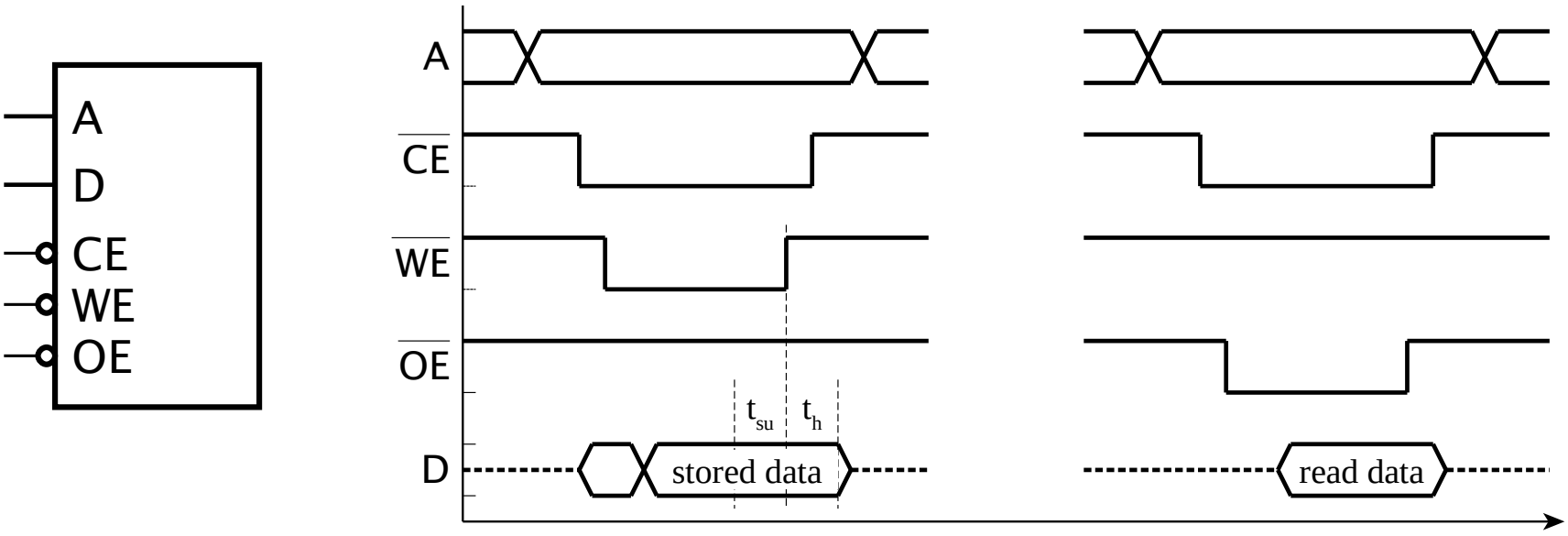


Memory Types

- Random-Access Memory (RAM)
  - Can read and write
  - Static RAM (SRAM)
    - Stores data so long as power is supplied
    - Asynchronous SRAM: not clocked
    - Synchronous SRAM (SSRAM): clocked
  - Dynamic RAM (DRAM)
    - Needs to be periodically refreshed
- Read-Only Memory (ROM)
  - Combinational
  - Programmable and Flash rewritable
- Volatile and non-volatile

Asynchronous SRAM

- Data stored in 1-bit latch cells
  - Address decoded to enable a given cell
- Usually use active-low control inputs
- Not available as components in ASICs or FPGAs



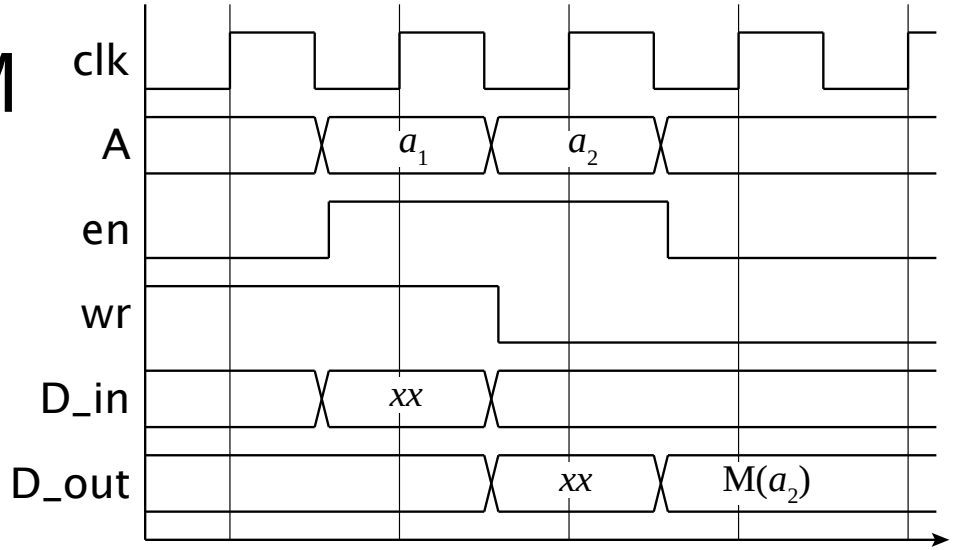
Asynch SRAM Timing

- Timing parameters published in data sheets
- Access time
  - From address/enable valid to data-out valid
- Cycle time
  - From start to end of access
- Data setup and hold
  - Before/after end of WE pulse
  - Makes asynch SRAMs hard to use in clocked synchronous designs

Synchronous SRAM (SSRAM)

- Clocked storage registers for inputs
  - address, data and control inputs
  - stored on a clock edge
  - held for read/write cycle

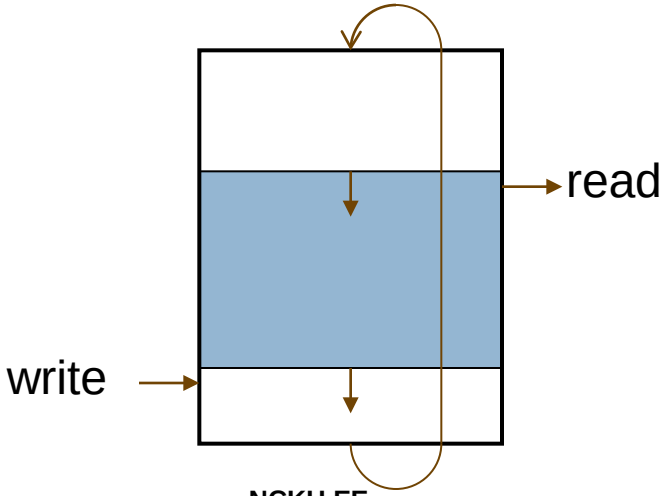
- Flow-through SSRAM
  - no register on data output



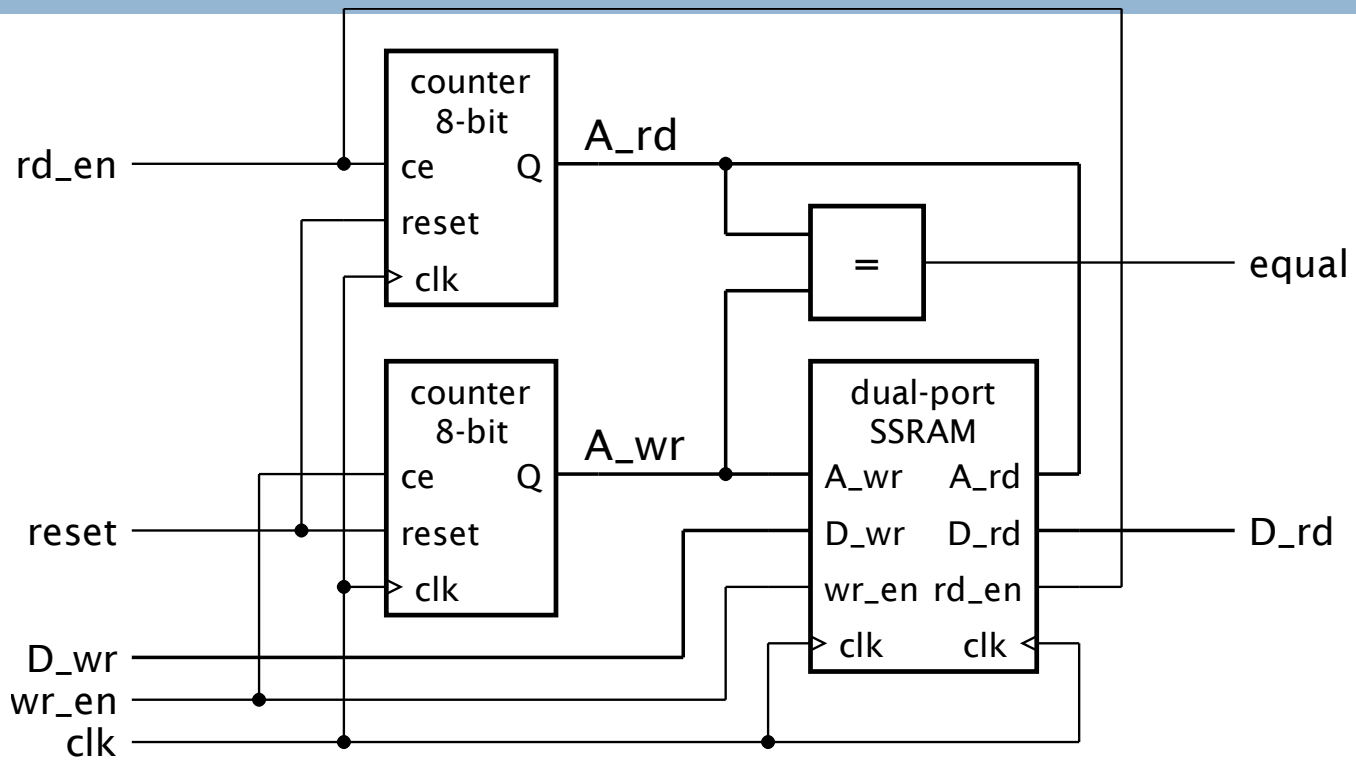
FIFO Memories

- First-In/First-Out buffer
  - Connecting producer and consumer
  - Decouples rates of production/consumption

- Implementation using dual-port RAM
  - Circular buffer
  - Full: write-addr = read-addr
  - Empty: write-addr = read-addr



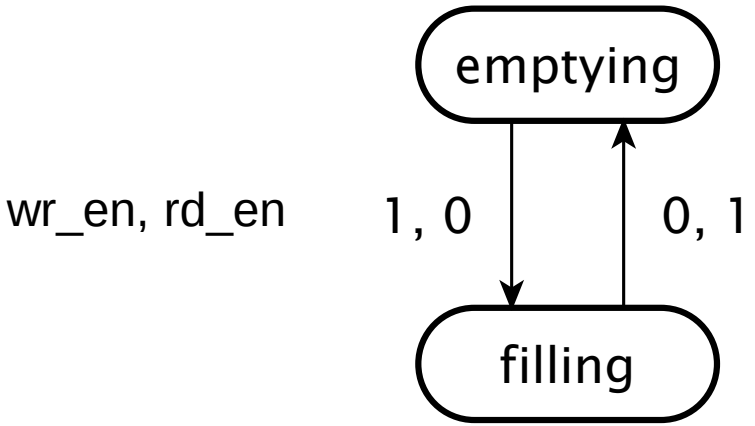
Example: FIFO Datapath



- Equal = full or empty
  - Need to distinguish between these states — How?

Example: FIFO Control

- Control FSM
  - → filling when write without concurrent read
  - → emptying when without concurrent write
  - Unchanged when concurrent write and read



full = filling and equal

empty = emptying and equal

Multiple Clock Domains

- Need to resynchronize data that traverses clock domains
  - Use resynchronizing registers
- May overrun if sender's clock is faster than receiver's clock
- FIFO smooths out differences in data flow rates
  - Latch cells inside FIFO RAM written with sender's clock, read with receiver's clock

Assignment to be Completed by Yourself

- Design a FIFO that would use two cycles or less for each of its operations
  - Reset: set write pointer and read pointer to zero
  - Insert (wr_en): write to memory; then increment the write pointer
  - Remove (rd_en): read from memory; then increment the read pointer
- wordsize =16 bit; capacity = 16

Task and Function

Overview

Difference and Similarity

Examples

Overview

- Verilog provides *tasks* and *functions* to break up large behavioral designs into smaller pieces.
- Just like procedures or subroutines in most programming languages
- Tasks
  - consists of input, output and inout arguments
- functions
  - have input arguments

Differences and Similarity

Functions	Tasks
Can enable another function but not another task	Can enable other tasks and functions
Always execute in 0 simulation time	May execute in non-zero simulation time
Contain no delay, event or timing control statements	May contain delay, ….
At least one input argument	Have zero or more arguments
Return a single value only	No return, but pass through ouput and inout

Differences and Similarity

Functions	Tasks
In an expression, e.g., i=func(a, b, c)	A statement e.g., task(out, in);
Combinational	Break long procedural blocks
Synthesizable	Not synthesizable
Define within a module and local to the module	Define within a module and local to the module
NO wires, always or initial inside	NO wires, always or initial inside

Example I


```
task tsk;
    input i1, i2;
    output o1, o2;
    $display("Task tsk, i1=%0b,
            i2=%0b",i1,i2);
    #1 o1 = i1 & i2;
    #1 o2 = i1 | i2;
endtask
```

```
function [7:0] func;
    input i1;
    integer i;
    reg [7:0] rg;
    begin
      rg = 1;
      for (i=1; i<=i1; i=i+1)
         rg = rg+1;
      func = rg;
    end
endfunction
```

Example II


```
task Idle;
    input [7:0] i;
    begin
       repeat (i)
        @(posedge clk);
    end
endtask
```

Example III


```
task Release_bus;
  begin
    write =1;
    trans = 2'b00;
     addr = 0;
     burst = 0;
     size = 0;
     w_index = 0;
      w_data = 32'h0;
      write_en =0;
     read_en =0;
  end
endtask
```

Example IV


```
module parity;
....
reg [31:0] addr;
reg parity;
always @(addr) begin
  parity = calc_parity(addr);
  $display("Parity calculated = %b", calc_parity(addr))
 end
...
function calc_parity;
input [31:0] address;
  begin
     calc_parity = ^address; // reduction xor
  end
endfunction
endmodule
```