Discussion Session 6

CS/ECE 552

Ramkumar Ravi

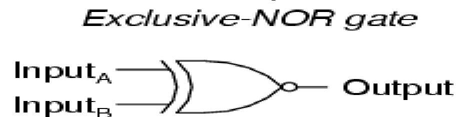
05 Mar 2012

Introduction

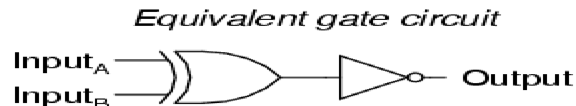
- Rules for HW are up-> Please follow instructions
  - HW3 is due on **03/07**
- EXPLORING FIFO
- MIDTERM REVIEW
  - **03/06, 7:15 to 9:15 PM, Location: CHEM B371**
- WARNING: Codes here are for demonstration purposes only; Not tested and might have bugs as well

Problem 1

- Expect just 3-4 lines of your opinion for each instruction
  - As an example, consider the “Bit Equal” instruction
  - This instruction does a bit-for-bit compare between two registers. For each bit i, if bit i of $rs is equal erwise set bit i of $rd to zero.

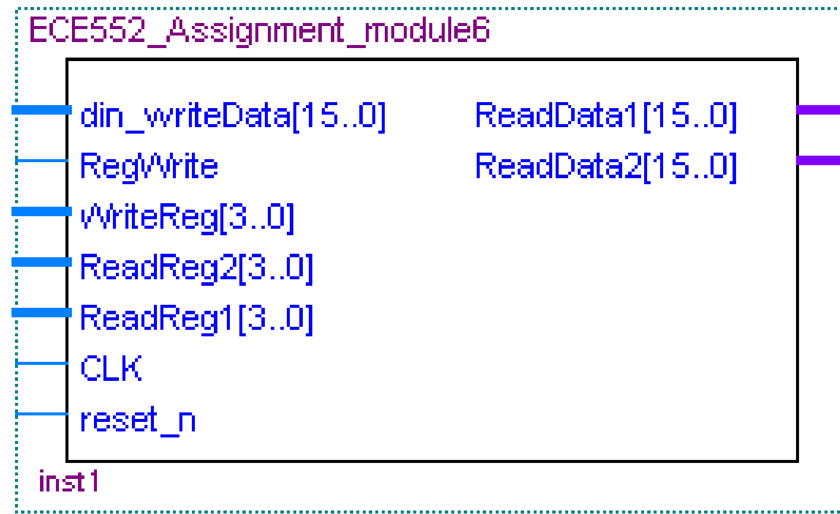


A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1



  - Bit equal is hence equivalent to an XNOR instruction. So to incorporate the XNOR instruction in your data path, what changes/modifications will you need to

Problem 3 - Register file design



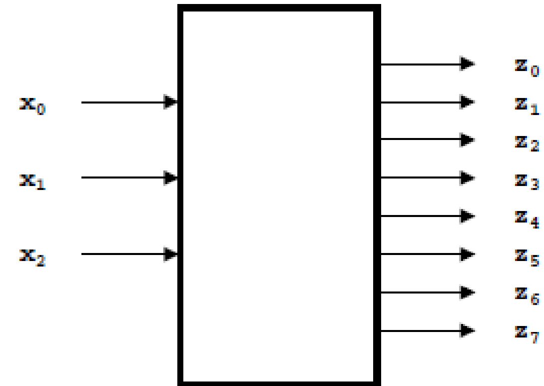
1. din_writeData [15:0] Á writedata [15:0]
2. RegWrite Á write
3. WriteReg [3..0] Á writeregsel [2:0]
4. ReadReg2[3..0] Á read2regsel [2:0]
5. ReadReg1 [3..0] Á read1regsel [2:0]
6. CLK Á clk
7. Reset_n Á rst
8. ReadData1 [15..0] Á read1data [15:0]
9. ReadData2 [15:0] Á read2data [15:0]

Register File Interface

```
parameter WIDTH = 16;
input clk, rst;
input [2:0] read1regsel;
input [2:0] read2regsel;
input [2:0] writeregsel;
input [WIDTH-1:0] writedata;
input       write;
output [WIDTH-1:0] read1data;
output [WIDTH-1:0] read2data;
output       err;
```

Register File design

- Lets start with a 3-8 Decoder



- If X0X1X2 = 3’b000; select z0 and so on..

```
wire [7:0] we, awe;
decode3_8 decoder (.sel(writeregsel), .Out(we));
and2 inst[7:0] (.in1(we), .in2({8{write}}), .out(awe));
```

Register file – The registers

wire [WIDTH-1:0] q0, q1, q2, q3, q4, q5, q6, q7;

register regs7 (.q(q7), .d(writedata), .clk(clk), .rst(rst), .we(awe[7]));

register regs6 (.q(q6), .d(writedata), .clk(clk), .rst(rst), .we(awe[6]));

and so on..

Now wh

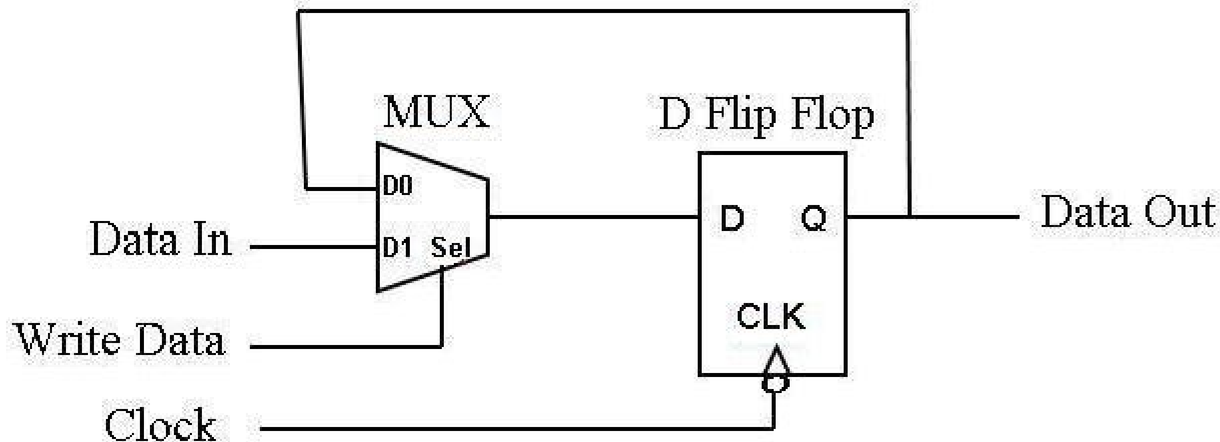


figure below)

REGISTER module

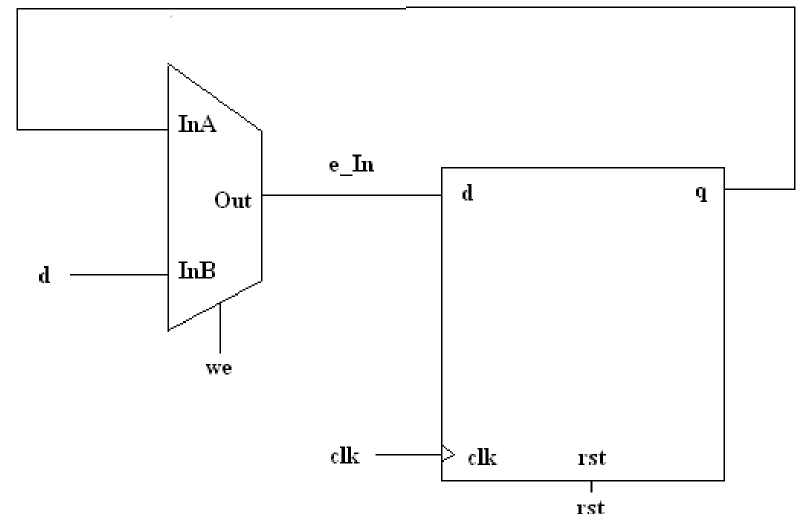
```
module register (input [15:0] d, input clk, rst, we, output [15:0] q);

wire [15:0] e_In;
mux2_1 mux[15:0] (.InA (q), .InB(d), .S({16{we}}), .Out(e_In));
dff inst[15:0] (.q(q), .d(e_In), .clk({16{clk}}), .rst({16{rst}}));
```

2:1 MUX -> If S is 1'b1; InB is selected el

16 copies of the DFF module

NOTE:

You might also need a 8:1 MUX for



Problem 4 - Saturating Counter

```
module sc( clk, rst, ctr_rst, out, err);
    input clk;
    input rst;
    input ctr_rst;
    output [2:0] out;
    output err;
endmodule
```

- **rst**: Synchronous reset that sets output to zero at pos clock edge
- **ctr_rst**: ctr_rst is different from the global rst signal

Code Example – One Possible implementation

```
reg [2:0] nextState;

dff inst [2:0](out, nextState, clk, rst);      /* Out maps to q; nextState maps to d */



always@(out, ctr_rst)

      begin

            case(out)            // Every time out changes, keep evaluating by reading
ctr_rst

                                    // If ctr_rst is 0; keep incrementing; if not nextState is 0
                              // Observe that nextState is feeding the D Flip-Flop

                  3'd0: begin

                        nextState=ctr_rst? 3'd0:3'd1;

                        err=1'd0;

                          end
```

Counter – Sample Output

- Shown at the right are the values of **ctr_rst** and **out** for

  a sample simulation run

1. **Out** is initially **X** (cycle 0)
2. **Out** is 0 on posedge (**rst=1**). definition, **rst** will be held HIGH in first 2 cycles (Cycle 100)
3. **Out** is 0 (**rst=1**) (Cycle 200)
4. **Out** is being incremented from 0->1->2 (see cycles 300,400 ar 500). At 500, **ctr_rst** is 1
5. So, **Out=0** on next posedge (cycle 600)

```
# time:                                 0 Cycle
#  clk 1 ctr_rst 0 Out x
#
# time:                               100 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                               200 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                               300 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                               400 Cycle
#  clk 1 ctr_rst 0 Out 1
#
# time:                               500 Cycle
#  clk 1 ctr_rst 1 Out 2
#
# time:                               600 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                               700 Cycle
#  clk 1 ctr_rst 0 Out 1
#
# time:                               800 Cycle
#  clk 1 ctr_rst 1 Out 2
#
# time:                               900 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                              1000 Cycle
#  clk 1 ctr_rst 0 Out 1
#
# time:                              1100 Cycle
#  clk 1 ctr_rst 0 Out 2
#
# time:                              1200 Cycle
#  clk 1 ctr_rst 1 Out 3
#
# time:                              1300 Cycle
#  clk 1 ctr_rst 1 Out 0
```

Sample Output - Continued

1. **Out** is being continuously increment now from 0->1->2->3->4->5 (see cyc 2000,2100,2200,2300,2400,2500)
2. From 2600 onwards, **Out** retains the final value of 5 (see cycles 2600, 27 and 2800)
3. At cycle 2800, **ctr_rst** goes HIGH
4. At the next clock, **Out** is reset to ZE

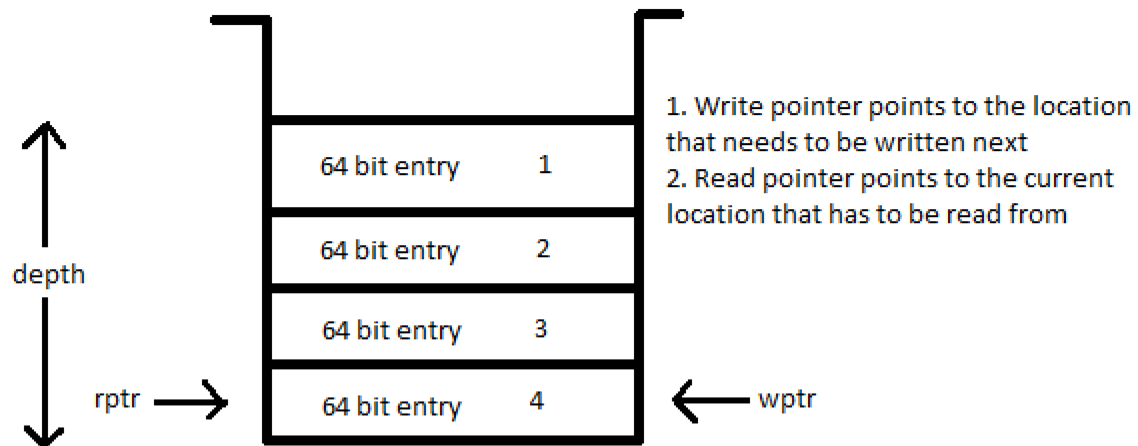
```
# time:                    2000 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                    2100 Cycle
#  clk 1 ctr_rst 0 Out 1
#
# time:                    2200 Cycle
#  clk 1 ctr_rst 0 Out 2
#
# time:                    2300 Cycle
#  clk 1 ctr_rst 0 Out 3
#
# time:                    2400 Cycle
#  clk 1 ctr_rst 0 Out 4
#
# time:                    2500 Cycle
#  clk 1 ctr_rst 0 Out 5
#
# time:                    2600 Cycle
#  clk 1 ctr_rst 0 Out 5
#
# time:                    2700 Cycle
#  clk 1 ctr_rst 0 Out 5
#
# time:                    2800 Cycle
#  clk 1 ctr_rst 1 Out 5
#
# time:                    2900 Cycle
#  clk 1 ctr_rst 0 Out 0
#
```

Sample Output – Special Case (Hold Out at Zero)

1. Over cycles 1000, 1100 and 1200, Cou is incrementing
2. At 1200, it saw **ctr_rst** is 1. So in the r cycle (cycle 1300), **Out** is ZERO
3. At 1300, **ctr_rst** is still1. So in next cy (cycle 1400), **Out** is still ZERO
4. At 1400, **ctr_rst** is still 1. So in next cy (cycle 1500), **Out** is still ZERO
5. At 1500, **ctr_rst** is 0 due to which cou resumes counting (You can see that it incremented from 0 to 1 in cycle 1600)

```
# time:                          1000 Cycle
#  clk 1 ctr_rst 0 Out 1
#
# time:                          1100 Cycle
#  clk 1 ctr_rst 0 Out 2
#
# time:                          1200 Cycle
#  clk 1 ctr_rst 1 Out 3
#
# time:                          1300 Cycle
#  clk 1 ctr_rst 1 Out 0
#
# time:                          1400 Cycle
#  clk 1 ctr_rst 1 Out 0
#
# time:                          1500 Cycle
#  clk 1 ctr_rst 0 Out 0
#
# time:                          1600 Cycle
#  clk 1 ctr_rst 0 Out 1
```

Problem 5 - FIFO



Sample Code - Instantiation

```
wire [63:0] data[3:0], data_flop[3:0];      // Multi-dimensional array

wire [1:0] rd_ptr, wr_ptr;
reg [1:0]  rd_nxt, wr_nxt;
wire [2:0] count;
reg [2:0] nxt_count;

// Instantiate DFF for read pointer; similarly for write and count
dff temp0 [1:0](rd_ptr, rd_nxt, clk, rst);

dff temp3 [63:0](data_flop[0], data[0], clk, rst);
```

FIFO SAMPLE CODE (Contd..)

```
assign data[0] =(data_in_valid & (wr_ptr==2'd0) & (~fifo_full)) ? data_in : data_flop[0];

assign data[1] =(data_in_valid & (wr_ptr==2'd1) & (~fifo_full)) ? data_in : data_flop[1];

// Similarly for data[2] and data[3]



// logic for read and write pointers

always@(*)begin

        case(rd_ptr)

                2'd0:rd_nxt=(pop_fifo & (~fifo_empty)) ? 2'd1:2'd0;
2'd1:rd_nxt=(pop_fifo & (~fifo_empty)) ? 2'd2:2'd1;                    // logic for 2'd2 and 2'd3
comes here

        endcase

end



always@(*)begin
```

FIFO Sample Code (Contd..)

```
always @(*)begin
    case(count)
        3'd0: nxt_count=~(data_in_valid^pop_fifo)?3'd0:data_in_valid?3'd1:3'd0;
        3'd1: nxt_count=~(data_in_valid^pop_fifo)?3'd1:data_in_valid?3'd2:3'd0;
        // What happens for other values ?
    endcase
end

assign data_out= data_flop[rd_ptr];
assign fifo_full=(count==3‘??);     // When count reaches ?
assign fifo_empty=(count==3‘??);        // When count is  ??
```