Digital logic

\[ F = \text{Sm}(3, 4, 5, 6, 8, 10, 12, 14, 15) \]

\[ F = \Sigma m(3, 4, 5, 6, 11, 12, 13, 14) \]
### Starting a K-map – truth values filled in

<table>
<thead>
<tr>
<th>AB</th>
<th>CD</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>01</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>11</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>10</td>
<td>0 0 1 0</td>
</tr>
</tbody>
</table>
Grouped K-map with red, green, and blue shading

\[ F = B\bar{C} + B\bar{D} + \bar{B}CD \]
NAND implementation

Decompose into ternary NAND of three intermediate functions.

\[ F = B\bar{C} + B\bar{D} + \bar{B}CD \]

\[ F = X + Y + Z \]

\[ F = (\bar{X}\bar{Y}\bar{Z}) \]

\[ F = \text{NAND3}(\bar{X}, \bar{Y}, \bar{Z}) \]
NAND implementation

\[ X = B\bar{C} \implies \bar{X} = \text{NAND2}(B, \bar{C}) \]
\[ Y = B\bar{D} \implies \bar{Y} = \text{NAND2}(B, \bar{D}) \]
\[ Z = \bar{B}CD \implies \bar{Z} = \text{NAND3}(\bar{B}CD) \]
NAND implementation

Putting it all together:
CMOS implementation

Use SOP to draw pull-up (taking care to invert each signal)

$$\bar{F} = \bar{B}\bar{C} + \bar{B}\bar{D} + BCD$$
CMOS implementation – could be better still (but okay)

Swap parallel for serial to draw pull-down with NMOS.

\[ \bar{F} = \bar{B}\bar{C} + \bar{B}\bar{D} + BCD \]
Part a): The purpose of this FSM is to detect the pattern 0110. It does not detect overlapping patterns.
assign out = (current_state == STATE_3) && in;
always @(posedge clk) begin
    if (rst) current_state <= STATE_0;
    else current_state <= next_state;
end
always @(*) begin
    case (current_state)
        STATE_0 : next_state = in ? STATE_0 : STATE_1;
        STATE_1 : next_state = in ? STATE_2 : STATE_1;
        STATE_2 : next_state = in ? STATE_3 : STATE_1;
        default : next_state = STATE_0;
    endcase
end
Logic-level implementation

Create per-bit truth tables for next_state:

<table>
<thead>
<tr>
<th>$cs_1$</th>
<th>$cs_0$</th>
<th>$in$</th>
<th>$ns_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$$ns_0 = cs_1 \cdot \bar{cs}_0 + \bar{cs}_1 \cdot \bar{in}$$
Logic-level implementation

<table>
<thead>
<tr>
<th>$cs_1$</th>
<th>$cs_0$</th>
<th>$in$</th>
<th>$ns_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$$ns_1 = cs_1 \cdot \bar{cs}_0 \cdot in + \bar{cs}_1 \cdot cs_0 \cdot in$$
Logic-level implementation
Moore machine: add output high state
Ready/valid

a.) $4'b0000, 4'b0110, 4'b0110$

b.) 0, 4, 5
Downsampler control
Downsampler control

Very simple datapath:

- An odd number of inputs would mean that the controller would either hang or would combine the first input of the next ‘line’ with the last input of the first ‘line.’
- This could be solved by making DataInValid and DataOutValid multi-bit so that there could be an ‘end-of-line’ value for valid.
Downsampler control

Very simple datapath:

- An odd number of inputs would mean that the controller would either hang or would combine the first input of the next ‘line’ with the last input of the first ‘line.’

- This could be solved by making DataInValid and DataOutValid multi-bit so that there could be an ‘end-of-line’ value for valid.
Memory operations

a.) lui $t0, 0x3000
    addiu $t0, $t0, 0x20
    lw $t0, 40($t0)
    Read – all three masks 4'b0000.

b.) lui $t0, 0x1000
    addiu $t0, $t0, 0x20
    sb $t0, 43($t0)
    Store byte to 0x10000063
    IMEM 4'b0000, DMEM 4'b0001, ISRMEM 4'b0000

c.) lui $t0, 0x3000
    sh $t0, 42($t0)
    Store half to 0x30000042
    IMEM 4'b0011, DMEM 4'b0011, ISRMEM 4'b0000
Given the 5-3-3-8 200MHz SDRAM as implemented in your project, and assuming what is the fastest clock speed that your MIPS processor could run and never stall waiting for a fetch from SDRAM? Assume that you can completely redesign the control logic optimally.

- $t_{CL} = t_{CAS} = 5$ The time between writing a column address and the first bit of data coming out – this is the period of sequential bursts.
- $t_{RCD} = 3$ Time between writing row address and writing column address.
- $t_{RP} = 3$ Time between writing precharge command (‘closing’ old row) and writing row address (‘opening’ new row).
- $t_{RAS} = 8$ Time between writing row address (‘opening’ row) and writing precharge (‘closing’ row).
Memory timings

- Assuming every fetch is completely random in the address space, might have to write precharge, wait $t_{RP}$ to write row, wait $t_{RCD}$ to write column, and wait $t_{CL}$ to get data.
- Also, the time $t_{RCD} + t_{CL}$ we have the row open must be longer than $t_{RAS}$ to close the row.
- This has total latency $t = 3 + 3 + 5 = 11$ in units of 200 MHz clock periods.
- Assuming every fetch is completely random, this latency is a lower bound on the minimum CPU clock period.

$$11 \times 5 \text{ ns} = 55 \text{ ns} \implies f_{\text{max}} = 18.2 \text{ MHz}$$
Given a 5-5-5-12 SDRAM, how fast would its clock need to run to match the read cycle time of the SDRAM in problem 1?

\[
\frac{t_{RP} + t_{RCD} + t_{CL}}{f} = \frac{t'_{RP} + t'_{RCD} + t'_{CL}}{f'}
\]

\[
\frac{3 + 3 + 5}{200 \text{ MHz}} = \frac{5 + 5 + 5}{f'}
\]

\[
f' = \frac{15}{11} \times 200 \text{ MHz}
\]

\[
f' = 273 \text{ MHz}
\]
Memory timings

How long does it take to refresh a single row in the SDRAM in problem 1?

- Time to refresh is simply $t_{RAS}$
- This is why the row must be active for $t_{RAS}$ or longer!