Digital design laboratory 8
Greatest common divisor (GCD)

• The GCD of two numbers can be determined using the Euclidean algorithm.
• The basis of the algorithm is the % operation (remainder after division).
• The steps are the following in the calculation of GCD(A, B):
  • A = B*Q1 + R1, where R1 = A%B
  • B = R1*Q2 + R2, where R2 = B%R1
  • R1 = R2*Q3 + R3, where R3 = R1%R2
  • This is done until B>R1>R2>...>=0.
• The GCD is the last nonzero element in the above series.
GCD

• Example: GCD(15, 6)=? A=15, B=6.
• A=B*2+3 (Q1=2, R1=3)
• B=3*2+0 (Q2=2, R2=0)
• R1 is the last nonzero remainder after division => GCD(15, 6) = 3.
• Problem: the % operation can not be synthetized in Verilog.
• A subtraction based algorithm will be presented.
GCD

• Properties of the GCD:
  • GCD(A, B) = GCD(A-B, B) if A>B => A = A-B
  • GCD(A, B) = GCD(A, B-A) if B>A => B = B-A

• Repeating the above steps until A=B leads to the value of the GCD.

• Example: GCD(15, 6) A=15, B=6
  • 1. step: A>B => A = A-B = 15-6 = 9
  • 2. step: A>B => A = A-B = 9-6 = 3
  • 3. step: B>A => B = B-A = 6-3 = 3 => A=B=3 => GCD(15, 6) = 3.
GCD

• We are going to implement the previous algorithm.
• The controller state machine is the following:
GCD

- The state machine will be implemented in the Topmodule. It controls the circuit on the right:
GCD

• Datapath components:
  • Ao_reg and Bo_reg: registers with load and update input. When load=1, the value on the input pins are loaded into the registers. When update=1, the value on the output of the SUB modules are loaded. Otherwise the registers maintains its value.
  • SUB: subtractor circuit that subtract Y from X (X-Y).
  • MUX EXCH: this module provides the inputs of the SUB circuit. If rsub = 0, X=A and Y=B, thus the A-B operation is performed. If rsub=1, X=B and Y=A and B-A is calculated.
  • COMP: magnitude comparator.
GCD

• Create new project: DigLab8 on drive D:\
GCD

- **Next**
GCD

- Finish
GCD

- Add the Topmodule to the project
GCD

- Add the inputs and outputs
- Next

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Direction</th>
<th>Bus</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rst</td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bt</td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dig_n</td>
<td>output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>seg_n</td>
<td>output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>col_n</td>
<td>output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GCD

- Finish
GCD

- Register implementation
- Add new Verilog module
- Name: Reg8
module Reg8(
    input clk,
    input rst,
    input load,
    input upd,
    input [7:0] load_in,
    input [7:0] upd_in,
    output reg [7:0] q
);

always @ (posedge clk)
    if (rst) q <= 8'd0;
    else if (load) q <= load_in;
    else if (upd) q <= upd_in;

endmodule
GCD

- Magnitude comparator implementation
- Name: Comp
GCD

• Implementation

```verilog
module Comp(
    input [7:0] A,
    input [7:0] B,
    output agtb,
    output bgta,
    output aeqb
);

assign agtb = (A>B);
assign bgta = (B>A);
assign aeqb = (A==B);
endmodule
```
GCD

- MUX EXCH and SUB modules will be implemented together
- Add new Verilog module
- Name: Msub
module Msub(
  input [7:0] X,
  input [7:0] Y,
  input rsub,
  output reg [7:0] diff
);

always @ (*)
  if (rsub) diff <= Y-X;
else diff <= X-Y;
endmodule
GCD

• We want to indicate the value of the GCD
• The 7 segment LED display will be used
• We need to implement a Hexadecimal to 7-segment converter, a multiplexer and a decoder circuit
• We also need a 2 bit counter to generate the select signal for the multiplexer and the decoder
GCD

• Decoder
• Name: dec4
module dec4(
    input [1:0] s,
    output reg [3:0] d
);

always @ (*)
    case (s)
        2'b00: d <= 4'b0001;
        2'b01: d <= 4'b0010;
        2'b10: d <= 4'b0100;
        2'b11: d <= 4'b1000;
    endcase
endmodule
GCD

• cnt2 module:
module cnt2(
    input clk,
    input rst,
    output reg [1:0] cnt
);

always @ (posedge clk)
if (rst) cnt <= 2'd0;
else cnt <= cnt + 2'd1;
endmodule
GCD

- Multiplexer module
- Name: Mux
module Mux(
  input [15:0] num,
  input [1:0] sel,
  output reg [3:0] out
);

always @ (*)
  case (sel)
    2'b00: out <= num[3:0];
    2'b01: out <= num[7:4];
    2'b10: out <= num[11:8];
    2'b11: out <= num[15:12];
  endcase
endmodule
GCD

• Hex7Seg module
always @ (*)
  case (hex)
    4'b0000 : seg <= 8'b00111111; // 0
    4'b0001 : seg <= 8'b00000110; // 1
    4'b0010 : seg <= 8'b01011011; // 2
    4'b0011 : seg <= 8'b01001111; // 3
    4'b0100 : seg <= 8'b01100110; // 4
    4'b0101 : seg <= 8'b01101101; // 5
    4'b0110 : seg <= 8'b01111101; // 6
    4'b0111 : seg <= 8'b00000111; // 7
    4'b1000 : seg <= 8'b01111111; // 8
    4'b1001 : seg <= 8'b01101111; // 9
    4'b1010 : seg <= 8'b01110111; // A
    4'b1011 : seg <= 8'b01111100; // b
    4'b1100 : seg <= 8'b00111001; // C
    4'b1101 : seg <= 8'b01011110; // d
    4'b1110 : seg <= 8'b01111001; // E
    4'b1111 : seg <= 8'b01110001; // F
  default : seg <= 8'b00000000; // 0
  endcase
GCD

• The state machine on slide 5 shows that we leave the IDLE state is the start signal is set to 1
• The start signal is connected to the bt inputs (OR relation)
• We need a rising edge sensor circuit, that is able to provide a 1 clk long high output after the button is pressed.
GCD

- Rising edge sensor module
- Name: R_edge_sens
- Inputs: clk, rst, bt [3:0]
- Output: start
module R_edge_sens(
    input clk,
    input rst,
    input [3:0] bt,
    output start
);

reg [1:0] bt0_reg, bt1_reg, bt2_reg, bt3_reg;
always @(posedge clk)
if (rst)
    begin
        bt0_reg <= 2'b11;
        bt1_reg <= 2'b11;
        bt2_reg <= 2'b11;
        bt3_reg <= 2'b11;
    end
else
    begin
        bt0_reg <= {bt0_reg[0], bt[0]}; // Concatenation, use braces
        bt1_reg <= {bt1_reg[0], bt[1]}; // Concatenation, use braces
        bt2_reg <= {bt2_reg[0], bt[2]}; // Concatenation, use braces
        bt3_reg <= {bt3_reg[0], bt[3]}; // Concatenation, use braces
    end

assign start = (bt0_reg == 2'b01) | (bt1_reg == 2'b01) | (bt2_reg == 2'b01) | (bt3_reg == 2'b01);
endmodule
• Now switch back to the Topmodule
• We are going to implement the state machine first
• After setting an 8 bit number on the switches (A), and a button is pressed, we switch to the INIT1 state and the loada signal will be set to 1.
• Now the second 8 bit input can be set, and after pressing a button again, the FSM goes into the INIT2 state. Loada is set to 0 and loadb is set to 1.
GCD

• First we implement the state and next_state registers. To improve readability, we use parameters to encode the states. This way we can write the names of the states instead of their binary codes.

```verilog
parameter IDLE1 = 3'b000;
parameter IDLE2 = 3'b111;
parameter INIT1 = 3'b001;
parameter INIT2 = 3'b010;
parameter TEST = 3'b011;
parameter SUB = 3'b100;
parameter RSUB = 3'b101;
parameter DONE = 3'b110;

reg [2:0] state, next_state;

always @ (posedge clk)
if (rst) state <= IDLE1;
else state <= next_state;
```
GCD

• Study the graph on slide 5. The state machine has 4 inputs that affect the state changes: start, aeqb, agtb, bgta
• There are four outputs to control the determination of the GCD: loada, loadb, upda, updb, rsub
• We are goind to add a wire for every input and output.
GCD

• Add the following code to the topmodule:

```verilog
wire start, aeqb, agtb, bgta;
wire loada, loadb, upda, updb, rsub;
```
GCD

• The logic for the status and next_state registers still has to be added.
• We start with next_state.
always @ (*)
case (state)
  IDLE1:
    if (start) next_state <= INIT1;
    else next_state <= IDLE1;
  INIT1:
    next_state <= IDLE2;
  IDLE2:
    if (start) next_state <= INIT2;
    else next_state <= IDLE2;
  INIT2:
    next_state <= TEST;
  TEST:
    if (aeqb) next_state <= DONE;
    else if (agtb) next_state <= SUB;
    else next_state <= RSUB;
  SUB:
    next_state <= TEST;
  RSUB:
    next_state <= TEST;
  DONE:
    if (start) next_state <= IDLE1;
    else next_state <= DONE;
endcase
As a last step of the FSM implementation, we have to set the outputs:

```plaintext
assign loada = (state == INIT1);
assign loadb = (state == INIT2);
assign upda = (state == SUB);
assign updb = (state == RSUB);
assign rsub = (state == RSUB);
```
Finally, we have to instantiate the previously implemented modules.
First start with the comparator.
Left click on the Comp.v module
On the left bottom of the screen, click on the + sign on the left of Design Utilities.
Double click on View HDL Instantiation Template
GCD
GCD

- Now you can copy the instantiation template into the Topmodule.

```vhdl
wire [7:0] A, B, diff;

Comp C (  
  .A(A),  
  .B(B),  
  .agtb(agtb),  
  .bgta(bgta),  
  .aeqb(aeqb)  
);
```
GCD

- Next implement the Reg8 module in two instances: RegA, RegB
- Some wires have to be updated in the instantiation

Reg8 RegA (  
.clk(clk),  
.rst(rst),  
.load(loada),  
.upd(upda),  
.load_in(sw),  
.upd_in(diff),  
.q(A)  
);  

Reg8 RegB (  
.clk(clk),  
.rst(rst),  
.load(loadb),  
.upd(updb),  
.load_in(sw),  
.upd_in(diff),  
.q(B)  
);
GCD

- Now implement the Msub module:

```c
Msub M (  
  X(A),  
  Y(B),  
  rsub(rsub),  
  diff(diff)  
);  
```
GCD

• Now add the cnt2 and R_edge_sens modules:

```verilog
R_edge_sens R {
  .clk(clk),
  .rst(rst),
  .bt(bt),
  .start(start)
};

wire [1:0] sel;

cnt2 C {
  .clk(clk),
  .rst(rst),
  .cnt(sel)
};
```
GCD

- Finally, we have to implement dec4, Mux and Hex7Seg:

```verilog
wire [3:0] dig, hex;
wire [7:0] seg;

assign dig_n = ~dig;
assign seg_n = ~seg;
assign col_n = 5'b11111;
```
GCD

- Instantiation of dec4 and Hex7Seg:

```plaintext
dec4 Dec (  
  .s(sel),  
  .d(dig)  
);

Hex7Seg H (  
  .hex(hex),  
  .seg(seg)  
);
```
GCD

- And finally, the MUX:

```verilog
reg [15:0] num_to_show;

always @ (*)
case (state)
  IDLE1: num_to_show <= {8'd1, sw};
  IDLE2: num_to_show <= {8'd2, sw};
  default:
    if (A<B) num_to_show <= {8'h9d, A};
    else num_to_show <= {8'h9d, B};
endcase

Mux mux (  
  .num(num_to_show),
  .sel(sel),
  .out(hex)
);
```
Test the Topmodule with a Test Fixture file: Topmodule_TF

Add the following excitations:

```verbatim
initial begin
    // Initialize Inputs
    clk = 0;
    rst = 0;
    bt = 0;
    sw = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    #100 rst = 1;
    #100 rst = 0; sw = 8'd15;
    #100 bt = 4'd1;
    #100 bt = 4'd0;
    #100 sw = 8'd3;
    #100 bt = 4'd1;
    #100 bt = 4'd0;

    end
always #25 clk = ~clk;
```
GCD

• Optional task: In the ISim simulator, add the following wires to the simulation: A, B, start, loada, loadb

• Relaunch the simulation and verify the Topmodule. Is the behavior correct?
GCD

• Now you have to add the .ucf file
• Download it: link
• Unzip the file to the project directory
• Add it to the project with add source or add copy of source
• Uncomment the following lines: clk, rst, bt (all), sw (all), seg_n (all), dig_n (all), col_n (all)
• Now generate the programming file