SystemVerilog Implementation of GCD

- Three modules ("design sources")
  - gcd_core.sv
  - dp.sv
  - fsm.sv

- Testbench (tb.sv) added later as a "simulation" source
- Then clk_def.xdc as a "constraint" source

```
`timescale 1ns / 1ps
///////////////////////////////////////////////////////////
// Company:
// Engineer:

[…]

///////////////////////////////////////////////////////////
module gcd_core(
    input logic clk,
    input logic rst,
    input logic load,
    input logic [7:0] din,
    output logic [7:0] gcd_rslt,
    output logic done
    );

[…]

endmodule
```

Structural Model

- Under the old ISE I followed this sequence:
- Next, created black-box models of the datapath and the controller
- ~~Used "View HDL Instantiation Template" process under the "Design Utilities" category~~
- Instantiated the datapath and controller in the top module
- Added internal wires for connecting

Unfortunately, HDL Instantiation Template is now a TCL app!

```
module gcd_core(  […]    );

// internal signals
logic xsel, xload, ysel, yload;
logic sub_sel, x_eq_y, x_gt_y;

// Instantiate the datapath - could have used .name or .*
dp dp1 (
    .clk(clk),
    .xsel(xsel),
    .xload(xload),
    .ysel(ysel),
    .yload(yload),
    .sub_sel(sub_sel),
    .din(din),
    .x_eq_y(x_eq_y),
    .x_gt_y(x_gt_y),
    .gcd_rslt(gcd_rslt)
    );
```

[…]

[...]

```
// Instantiate the controller
fsm fsm1 (
    .clk(clk),
    .rst(rst),
    .load(load),
    .x_eq_y(x_eq_y),
    .x_gt_y(x_gt_y),
    .xsel(xsel),
    .xload(xload),
    .ysel(ysel),
    .yload(yload),
    .sub_sel(sub_sel),
    .done(done)
    );


endmodule
```

Modeling Logic

- Continuous assignments (with operators)
- Module or primitive instantiations (structural)
- Procedural assignments (~ VHDL process)
  - Blocking assignments (=)
    - Best for combinational logic (~ VHDL variables)
  - Non-Blocking assignments (<=)
    - Best for clocked logic (~ VHDL signals)

Datapath (dp.sv)

- Comparators, muxes, subtrator, registers
- Mix of continuous assignments and procedural assignments
- ~~Continuous assignments require type “wire”~~
- ~~Procedural assignments require a type “reg”~~
- ~~The “reg” identifier does not imply register!~~

Use “logic” for everything!!! ☺

```
module dp( […] );

logic [7:0] diff;
logic  [7:0] x, y;

// comparators

assign x_eq_y = (x == y);
assign x_gt_y = (x >= y);

// subtractor and muxes

assign diff = (sub_sel ? y : x) - (sub_sel ? x : y);
```

[…]

```
// x and y registers with cascaded muxes
always_ff @(posedge clk)

begin

if (xload)
    if (xsel)  // could have used ternary operator
        x <= din;
    else x <= diff;

if (yload)
    if (ysel)
        y <= din;
    else y <= diff;

end

assign gcd_rslt = x;

endmodule
```

Controller (fsm.sv)

- Moore FSM with unregistered outputs
- Two procedures
  - Next state logic and state register (“clocked”)
  - Output logic (combinational)
- Use non-blocking for clocked logic
- Use blocking for combinational

```
module fsm( […] );

// symbolic state names as an enumerated type
typedef enum logic [2:0]
        {idle, loadx, loady, wait1, y_diff, x_diff, fine} statetype;
statetype  state;

// NSL and state register - non-blocking assignment
always_ff @(posedge clk)
begin
if (rst)
        state <= idle;
else
        case (state)
                idle : if (load) state <= loadx;
                loadx : state <= loady;
                wait1 :
                        case ({x_eq_y, x_gt_y})
                                2'b10, 2'b11     : state <= fine;
                                2'b00            : state <= y_diff;
                                2'b01            : state <= x_diff;
                        endcase
                fine : state <= idle;
                default : state <= wait1;
        endcase
end

[…]
```

```
// output logic - blocking assignments
always_comb
begin
       // default outputs
       xload = 0; xsel = 0; yload = 0; ysel = 0;
       sub_sel = 0; done = 0;
              case (state)
                     loadx :
                            begin
                                   xload = 1; xsel = 1;
                            end
                     loady :
                            begin
                                   yload = 1; ysel = 1;
                            end
                     y_diff :
                            begin
                                   yload = 1; sub_sel = 1;
                            end
                     x_diff : xload = 1;
                     fine : done = 1;
              endcase
end

endmodule
```

"Default" outputs avoid "latches"

Test Fixture (Testbench)

- Much more like programming! =:0
- Generate clock and reset
- Apply stimulus
- Check results (optional)
- Terminate simulation (recommended)
- Add test fixture as a “simulation source” under “Add Sources”

```
`timescale 1ns / 1ps
module tb;
        // Inputs
        logic clk;
        logic rst;
        logic load;
        logic [7:0] din;
        // Outputs
        logic [7:0] gcd_rslt;
        logic done;
        // Instantiate the Unit Under Test (UUT)
        gcd_core uut (
                .clk(clk),
                .rst(rst),
                .load(load),
                .din(din),
                .gcd_rslt(gcd_rslt),
                .done(done)
        );
[...]
```

Clock Generation

[...]

```
parameter CLK_PRD = 100; // 10 MHz clock
parameter HOLD_TIME = (CLK_PRD*0.3);

initial begin
        clk <= 0;
        forever #(CLK_PRD/2) clk = ~clk;
end
```

[...]

Reset and Stimulus Alignment

```
initial begin
        // Initialize Inputs
        rst = 0;
        load = 0;
        din = 8'bx;

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

        // Add stimulus here

        @(posedge clk); // align with clock edge

        #HOLD_TIME; // offset a hold time

        repeat(2) #CLK_PRD; // Now only wait integer clock periods

        rst = 1; #CLK_PRD;

        rst = 0;

        repeat(2) #CLK_PRD;            [...]
```

Stimulus and Done Check

```
[...]                   load = 1; #CLK_PRD;

                        load = 0; din = 8'd27; #CLK_PRD;

                        din = 8'd18; #CLK_PRD;

                        din = 8'bx; #CLK_PRD;

                        begin : run_loop
                                forever
                                        begin
                                                @(posedge clk);
                                                if (done) disable run_loop;
                                        end
                        end // run_loop

                        $finish;

          end
endmodule
```

Simulation Termination

```
parameter MAX_SIM_TIME = (100*CLK_PRD);

initial #(MAX_SIM_TIME)  $finish;
```

Allows a maximum simulation time of 100 clock cycles unless “done” occurs first