Course Topics - Outline

- Lecture 1 - Introduction
- Lecture 2 - Lexical conventions
- Lecture 3 - Data types
- Lecture 4 - Operators
- Lecture 5 - Behavioral modeling A
- Lecture 6 – Behavioral modeling B
- Lecture 7 – Behavioral modeling C
- Lecture 8 – Data flow modeling
- Lecture 9 – Gate Level modeling
- Lecture 10 – Tasks and Functions
- Lecture 11 – Advanced Modeling Techniques
- Lecture 12 - Coding Styles and Test Benches
- Lecture 13 - Switch Level modeling
Lecture 12 - Coding Styles and Test Benches

- Processor Stack
- FIFO
- Pipeline
- Finite State Machines
  - Moore and Mealy
  - State Diagram
  - Coding styles
- Procedural Continuous Assignment
- Self-checking Test Bench
- Exercise 12
Processor Stack

- Processor stack is a Last-In-First-Out (LIFO) Register Array. Last value pushed on the stack is the first one popped off it.
- A stack can be controlled by a single a stack pointer register – the location used to push data into stack is the same location from which data would be popped.
- Stack has data I/O port, clock, reset, push, pop control signals and error flag (read from empty or write to full)
- Stack Pointer (current read/write location) is internal.
Guidelines for Stack Coding

// Initialize stack_array registers to zero using for loop or $readmeh
if (rst) begin sp=0; full=0; empty=1; err=0; dout=0; end
if (push) begin
  if (empty) begin stack_array[sp] = din; empty = 0; end
  else if (push && full) err=1;
  else begin stack_array[sp]=din; sp=sp+1; end
end /* increment stack pointer (sp) */
else if (pop) begin
  if (sp==0 && empty==0) begin
dout=stack_array[sp]; empty=1; end
  else if (sp!=0) sp = sp-1; // decrement stack pointer
  if (full==1) full=0;
end
FIFO Register File

- First-In-First-Out is more complicated than stack. Like pipeline, it has 2 ends to be controlled.
- FIFO has distinct input and output and often used to balance between 2 clock domains with different data rates.
- There are 2 main conventions in representing FIFO read and write pointers: a. point to the next valid (reg) address. b. point to the most currently used one.
- In this course, we shall adopt the former convention (a).
- The read pointer must point to valid next data to be read.
- The write pointer must point to the unused register into which the next data will be written.
FIFO Schematics

- **data_in**
- **write**
- **clk_i**
- **nRST**

**fifo_array[0]**

- **write pointer**
- **read and write pointers are internal**

**fifo_array[7]**

- **read pointer**
- **data_out**
- **read**
- **clk_o**
Guidelines for behavioral FIFO Coding

// Initialize all FIFO register file to zero
reg[width-1:0] fifo[0:depth-1]
If (rst) begin wp=0;full=0;empty=1; end //wp points to 1st empty slot
assign dout = fifo[0] ; // first stage of fifo is output
always @(posedge clk) begin
if (rd & ~wr) begin  // shift-out (assumes: at least 1 valid value in fifo)
    for (i = 1 ; i < wp ; i = i + 1) begin  // shift all valid entries
        fifo[i-1] <= fifo[i] ; wp <= wp-1 ; full <= 0 ; end
        if (wp==0) begin empty <= 1 ; else empty <= 0 ; end
    end
If (~rd & wr) begin  // shift-in (assumes: at least 1 entry free)
    fifo[wp] <= din ; wp <= wp + 1 ; empty <= 0 ;
    if (wp==depth) full <= 1 ; else full <= 0 ; end
If(rd & wr) begin  //simult shift-in/shift-out. At least 1 valid entry
    for (i = 1 ; i < wp ; i = i + 1) fifo[i-1] <= fifo [i] ; fifo[wp-1] <= din ; end
end
Pipelines, queues, and FIFOs are common logic structures which are all related, in the sense that data moves from one storage location to another synchronously, based on a strobe signal, usually a clock.

```vhdl
always @(posedge clock)
begin
    // use non-blocking assignments
    out <= pipe[3];
    pipe[3] <= pipe[2];
    pipe[2] <= pipe[1];
    pipe[1] <= in;
end
```

// assignments order is irrelevant
Pipe Stage as Separate Module

- It is common to make a single pipe stage module and use it repetitively, as follows:

```verilog
module pipeline(out, in, clock);
    input clock, in;
    output wire out;
    wire s1out, s2out;
    pipestage s1(s1out, in, clock),
               s2(s2out, s1out, clock),
               s3(out, s2out, clock);
endmodule

module pipestage(out, in, clock);
    input clock, in;
    output reg out;
    always @(posedge clock)
        out <= in;
endmodule
```
It is more interesting if there is some combinational logic associated with each pipe stage. Suppose each stage has some logic represented by a function $f_1, f_2, f_3$ which is applied to the input.

```
module pipeline(out, in, clock) ;
    input clock, in :
    output out ;
    wire s1out, s2out, s1in, s2in, s3in ;
    assign s1in = f1(in), s2in = f2(s1out), s3in = f3(s2out) ;
    pipelinestage s1(s1out, s1in, clock),
        s2(s2out, s2in, clock),
        s3(out, s3in, clock) ;
endmodule
```
Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation.

- At each clock edge, combinational logic computes outputs and next state as a function of inputs and present state.
Two types of Finite State Machines

Moore State Machine

Mealy State Machine
There are two common variations of state machines, Mealy and Moore machines.

Mealy machines produce outputs based on both current state and inputs.

Moore machines produce outputs based only on the current state.

Typically, the clock is used to change the state based on the inputs which have been seen up to that point. It is often convenient to think of all the activity of the state machine as taking place on the clock edge:

- sample inputs
- compute next state
- compute outputs
- change state
- produce outputs
State Machine Guidelines

- Draw a state diagram.
- Label the states.
- Allocate state encoding.
- Label the transition conditions.
- Label the output values.
- Use parameters for the state variables.
- Use two procedures (one clocked for the state register and one combinational for the next state logic and the output decode logic).
- Use a case statement in the combinational procedure.
- Have a reset strategy (asynchronous or synchronous).
- Use default assignments and then corner cases.
- Keep state machine code separate from other code (i.e. don’t mix other logic in with the state machine clocked and combinational procedures).
module fsm(clk, nrst, start, transmit, wait, stop, ack, offline, online) ;

// state labels and state encoding
parameter IDLE=2’b10, RUN=2’b00, PAUSE=2’b01, FINISH=2’b11 ;

input clk, nrst, start, transmit, wait, stop ;
output ack, offline, online ;
reg [1:0] state, next_state ; // internal variables declaration
reg ack, offline, online ;

always @(posedge clk, negedge nrst) // sequential procedure
begin
  if (!nrst) // reset strategy
    state <= IDLE ;
  else
    state <= next_state;
end
// combinational procedure with case statement and output logic

always @ (start or transmit or stop or wait or state)
begin
    // default assignment to state and output variables
    next_state = state ; ack = 1’b0 ; offline = 1’b0 ; online = 1’b1 ;

    case (state)
        IDLE: begin
            offline = 1’b1 ; online = 1’b0 ;
            if (start) next_state = RUN ;
        end
        RUN: begin
            if (wait) next_state = PAUSE ;
            if (stop) // this has priority over the wait transition
                next_state = FINISH ;
        end
        PAUSE: begin
            ack = 1’b1 ;
            if (transmit) next_state = RUN ;
            if (stop) // event triggered
                next_state = FINISH ;
        end
        FINISH: ack = 1’b1 ;
    endcase
end
endmodule
<table>
<thead>
<tr>
<th>Coding styles of FSMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• One always block</td>
</tr>
<tr>
<td>• Two always blocks</td>
</tr>
<tr>
<td>• Three always blocks</td>
</tr>
<tr>
<td>• One-hot state decoding</td>
</tr>
</tbody>
</table>

**Drawbacks of Combinational outputs:**
- Can glitch between states.
- Consume part of overall clock cycle time otherwise available to the logic block driven by FSM outputs.
- There is less time for the receiving module to pass signals through inputs and additional combinational logic before they must be clocked.

**Conclusion:** Register FSM outputs!
FSM Output Glitching

- FSM state bits may not transition at precisely the same time
- Combinational logic for outputs may contain hazards
- Result: your FSM outputs may glitch!
- Registered FSM Outputs are Glitch-Free
  - Move output generation into the sequential always block
  - Calculate outputs based on next state
Two Always Block FSM with Registered Outputs

```verilog
reg [n-1:0] current_state, next_state;       // internal registers
always @ (current_state, list of inputs ....) // Combinational Section
    begin
        // use blocking assignments
        if (!nrst) next_state = idle;  // initialize all output registers
        else
            case (current_state)
                .. decide for each current_state, next_state.
            endcase
    end
always @ (posedge clk) // use non-blocking assignments
begin
    current_state <= next_state;
    case (current_state)
        ... decide for each current_state, all output control registers;
    endcase
end
```
Procedural Continuous Assignment

- Keywords **force** and **release** are used to express a procedural continuous assignment.
- It can be used to override assignments on both **regs & nets**.
- **force** - **release** statements are used in interactive debugging process, where certain registers or nets are forced to a value and the effect on other registers and nets is noted.
- It is recommended that **force** - **release** statements will not be used inside design blocks. They should appear only in stimulus or as debug statements. (insert bit errors...).
- Force on registers: **force** on a register overrides any procedural assignments or continuous assignments on the **reg** until the **reg** is released. Register variables continue to store the forced value after being **released** until changed by a future procedural assignment.
Example: force – release on register

```verilog
module dff(q, qbar, d, clk, nrst) // D flip-flop
input d, clk, nrst;
output reg q, qbar;
always @(posedge clk or negedge nrst)
begin
  if (!nrst) begin
    q = 1'b0; qbar = 1'b1; end
  else
    q = d; qbar = ~d;
end
endmodule

module stimulus;

..........
initial begin
  dff UUT(q, qbar, d, clk, nrst); // dff instantiation
  #50 force dff.q = 1'b1; // force value of 1 @ time 50
  #50 release dff.q; // release the value of q @ time 100
end
..........
endmodule
```
force – release on nets

● Force on nets overrides any continuous assignments until the net is released.
● The net will *immediately* return to its normal driven value when it released.
● A net can be forced to an expression or a value.

Example:
```verilog
module top ;

..........
assign out = a & b & c ; // continuous assignment
..........

initial
  begin
    #50 force out = a | b & c ; // a new expression is forced
    #50 release out ;
  end
..........
endmodule
```
Self-checking Test Bench

- **Stimuli SRAM**
  - Data
  - Initialization File
  - Address

- **Unit Under Test**
  - Inputs
  - Outputs
  - clk
  - nrst

- **Comparator**
  - Inputs
  - OK Flag

- **Address Counter**
  - clk
  - nrst

- **Monitor SRAM Initialization File**
  - Data
  - Address

Flow:
- Data from Stimuli SRAM to Unit Under Test
- Address from Stimuli SRAM to Unit Under Test
- clk and nrst from Stimuli SRAM to Unit Under Test
- Data and Address from Unit Under Test to Comparator
- OK Flag from Comparator

Connections:
- Stimuli SRAM to Unit Under Test
- Unit Under Test to Comparator
- Comparator to OK Flag
Exercise 12

- **Part 1:**
  Design parametric 8-level byte-wide micro Controller Stack. Use push/pop signals for write/read and full/empty indicators to report programmer’s error.

- **Part 2:**
  Design 16 bytes, single clock, Synchronous FIFO. Data output is un-registered.

- **Part 3:**
  Design Sequence Detector, implemented with Mealy FSM. Detects a sequence of 1011 in serial input. This is an overlapping sequence. So, if 1011011 come, sequence is repeated twice.