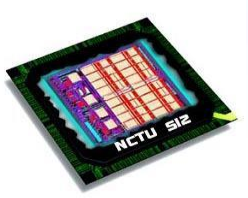
Coding Style & Experience Shared

Author: Lin-Hung, Lai

System Integration and Silicon Implementation (Si2) Lab
Institute of Electronics
National Yang Ming Chiao Tung University, Hsinchu, Taiwan



Learnt from TA Experience

- ✓ **01 Pass, But 03 Fail …**
  - 01 Simulation: will compile RTL to determine "current" value
  - 02 DC compile: will convert your RTL into real circuit
  - 03 Simulation: will compile real circuit to verify
    - Circuit is successfully synthesized to reach RTL function
    - Timing check
  - If I fail 03 …
    - First check where might be the problem occurs
      - ◆ Normally, you can open 2 waveform together to check the 01/03 consistence
        - » Ex: state works correctly, data register store the desired data
      - ◆ Observer the first inconsistent value occur place, and back-trace the related signal
        - » Ex: state expects to be "S_CAL" , but 03 shows "S_IDEL" or "X" at specific time step
      - ◆ Compare your RTL code to examine if
        - » Coding style: synthesis tool can not "understand" your code, or "Misunderstand" your code
          - » Avoid one always block contain many variable
          - » Avoid compiler dependent description (too software)
          - » Use basic circuit description to reach a complicated goal (hardware thought)
        - » Forget to reset, Use input directly: unknown propagation
        - » Non-blocking, blocking mixed use: weird result

99% problems are listed here!

Learnt from TA Experience

- ✓ **01 Pass, But 03 Fail …**
  - 01 Simulation: will compile RTL to determine "current" value
  - 02 DC compile: will convert your RTL into real circuit
  - 03 Simulation: will compile real circuit to verify
    - Circuit is successfully synthesized to reach RTL function
    - Timing check
  - If I fail 03 …
    - Second check if timing encounter problem
      - ◆ Check 02 timing report, examine latch or VIOLATE in your report
      - ◆ Check timing violation
        - » Check if synthesis cycle time is consistent with simulation (02 / 03)
        - » Check if XXX_SYN.sdf timing file is included in TESTBED
        - » Check if critical path encountered problem in timing report
      - ◆ Compare your RTL code

About always block

- ✓ **One always block do one thing!!** Better for synthesis, easier for debug!
  - What's the purpose of this always block?
    - FSM curr_state, Counter, Input/Output register, Data Register
    - FSM next_state, Decision flag, Case Selection, … etc
  - What's type should I describe this circuit?
    - Flip-Flop inferred: always@(posedge clk) --> non-blocking
    - Combinational only: always@(*) --> blocking
    - Decision Flag or simple calculation: assign a = (flag)? D1:D2
  - What's the always block might be inferred?
    - Can you explain this always block?
    - MUX? Cascade MUX? Huge MUX? MUX with DFF?
    - Data[index] --> inferred a huge mux to select data --> use it carefully!!
    - Always@(*) begin

```
A = default;
If(flag1)  A = result1;
If(flag2)  A = result2;
```

      Compiler depended expression method
      Sometimes it works, but really not recommend!!
    - end --> can not be explained, might infer latch, hard to debug

Common Error

- **Unknown Propagation**
  - Do not use input directly in your circuit
  - Recommend: Use a register to store the data when in valid is high
    - Always@(posedge clk) begin

      If(in_valid)

      Data_reg <= data;

      Else

      Data_reg <= Data_reg;
    - end
    - Then use Data_reg in your circuit design instead of using data

```
always @(posedge clk) begin
    if(!rst_n)
        A <= 8'b0;
    else
        case(current_state)
            s_input: if(IN_VALID) A <= B;
            default:              A <= A;
        endcase
end
```

Common Error

✓ Compiler dependent description

```
always@(*) begin
    next_state = state;
    temp_x = 0;
    temp_y = 0;
    current_x = row_x[move];
    current_y = col_y[move];
    move_w = move;
    current_num_w = current_num;
    flag_w = flag;
    out_valid_w = out_valid;
    out_x_w = out_x;
    out_y_w = out_y;
    move_out_w = move_out;
    counter_1 = counter - 1;
    move_1 = move - 1;

    for(k=0;k<25;k=k+1) begin    //25->24
        total_num_w[k] = total_num[k];
    end
    for(i=0;i<5;i=i+1) begin
        chess_w[i] = chess[i];
    end
    for(j=0;j<25;j=j+1) begin
            row_x_w[j] = row_x[j];
            col_y_w[j] = col_y[j];

    end
    case(state)

    IDLE: begin
        for(i=0;i<5;i=i+1) begin
            chess_w[i] = 0;
        end
        for(j=0;j<25;j=j+1) begin
            row_x_w[j] = 0;
            col_y_w[j] = 0;
        end
        flag_w = 0;
        move_w = 0;
```

```
//masked_possible_dir
always@(*) begin
    masked_possible_dir = possible_dir[cur_x][cur_y];
    case (priority_reg)
        3'd0: masked_possible_dir = masked_possible_dir;
        3'd1: masked_possible_dir = {masked_possible_dir[0],masked_possible_dir[7:1]};
        3'd2: masked_possible_dir = {masked_possible_dir[1:0],masked_possible_dir[7:2]};
        3'd3: masked_possible_dir = {masked_possible_dir[2:0],masked_possible_dir[7:3]};
        3'd4: masked_possible_dir = {masked_possible_dir[3:0],masked_possible_dir[7:4]};
        3'd5: masked_possible_dir = {masked_possible_dir[4:0],masked_possible_dir[7:5]};
        3'd6: masked_possible_dir = {masked_possible_dir[5:0],masked_possible_dir[7:6]};
        3'd7: masked_possible_dir = {masked_possible_dir[6:0],masked_possible_dir[7:7]};
        default: masked_possible_dir = masked_possible_dir;
    endcase
    if (state!=INPUT | move_cnt!=1)
        masked_possible_dir = masked_possible_dir & ( (~8'b0) << (used_dir[1] - priority_reg) );
end
```

Compiler depended expression method
Sometimes it works, but really not recommend!!

Circuit Awareness Description: Practice

FSM

```
always @(*) begin
    case(current_state)
        s_idle: if(IN_VALID) next_state = s_input;
                else         next_state = s_idle;
        s_input: if(cnt==2'd2)
                    case(MODE_r)
                    2'd0: next_state = s_add;
                    2'd1: next_state = s_sub;
                    2'd2: next_state = s_mul;
                    default: next_state = s_input;
                    endcase
                 else
                    next_state = s_input;
        s_add:   if(cnt==2'd1) next_state = s_output;
                 else next_state = s_add;
        s_sub:   if(cnt==2'd1) next_state = s_output;
                 else next_state = s_sub;
        s_mul:   if(cnt==2'd3) next_state = s_output;
                 else next_state = s_mul;
        s_output: if(cnt==2'd1) next_state = s_idle;
                 else next_state = s_output;
        default: next_state = current_state;
    endcase
end
```

Counter

```
always @(posedge clk) begin
    if(!rst_n)
        cnt <= 2'b0;
    else
        case(current_state)
            s_idle: cnt <= 2'b0;
            s_input: if(cnt==2'd2) cnt <= 2'b0;
                     else cnt <= cnt+1'b1;
            s_add:   if(cnt==2'd1) cnt <= 2'b0;
                     else cnt <= cnt+1'b1;
            s_sub:   if(cnt==2'd1) cnt <= 2'b0;
                     else cnt <= cnt+1'b1;
            s_mul:   if(cnt==2'd3) cnt <= 2'b0;
                     else cnt <= cnt+1'b1;
            s_output: if(cnt==2'd1) cnt <= 2'b0;
                     else cnt <= cnt+1'b1;
            default: cnt <= cnt;
        endcase
end
```

Combinational

```
always @(*) begin
    case(current_state)
        s_add: if(cnt==2'b0)       ACC_C = A;   else ACC_C = B;
        s_sub: if(cnt==2'b0)       ACC_C = A;   else ACC_C = B;
        s_mul:begin
                if(cnt==2'b0)      ACC_C = 0;
                else if(cnt==2'b1) ACC_C = E;
                else if(cnt==2'd2) ACC_C = 0;
                else               ACC_C = F;
        end
        default: ACC_C = 8'b0;
    endcase
end
```

Sequential

```
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)
        OUT <= 17'b0;
    else
        case(current_state)
        s_output: if(cnt==2'b00) OUT <= E; else OUT <= F;
        default: OUT <= 17'b0;
        endcase
end
```