שפת תכנון חומרה
Verilog - ורילוג

Verilog - Extra

Dr. Avihai Aharon

FSM: Encoding Style

- Binary Encoding
- One Hot Encoding
- One Cold Encoding
- Almost One Hot Encoding
- Almost One Cold Encoding
- Grey Encoding

One Hot Encoding

```
parameter  [4:0]  IDLE  = 5'b0_0001;
parameter  [4:0]  GNT0  = 5'b0_0010;
parameter  [4:0]  GNT1  = 5'b0_0100;
parameter  [4:0]  GNT2  = 5'b0_1000;
parameter  [4:0]  GNT3  = 5'b1_0000;
```

Almost One Hot Encoding

```
parameter  [3:0]  IDLE  =  4'b0000;
parameter  [3:0]  GNT0  = 4'b0001;
parameter  [3:0]  GNT1  = 4'b0010;
parameter  [3:0]  GNT2  = 4'b0100;
parameter  [3:0]  GNT3  = 4'b1000;
```

Binary Encoding

```
parameter  [2:0]  IDLE  = 3'b000;
parameter  [2:0]  GNT0  = 3'b001;
parameter  [2:0]  GNT1  = 3'b010;
parameter  [2:0]  GNT2  = 3'b011;
parameter  [2:0]  GNT3  = 3'b100;
```

Grey Encoding

```
parameter  [2:0]  IDLE  = 3'b000;
parameter  [2:0]  GNT0  = 3'b001;
parameter  [2:0]  GNT1  = 3'b011;
parameter  [2:0]  GNT2  = 3'b010;
parameter  [2:0]  GNT3  = 3'b110;
```

FSM: Encoding Style

Which Encoding Is the Best?

This is a tough question, mostly because each encoding has its benefits, it comes down to an optimization problem that depends on a large number of factors.

- **If a very simple system yields very similar results across encodings, then the original encoding is the best choice.**
- **If the FSM cycles through its states in one path (like a counter) then Gray code is a very good choice.**
- **If the FSM has an arbitrary set of state transitions or is expected to run at high frequencies, maybe one-hot encoding is the way to go.**

Binary Encoding

```
parameter  [2:0]  IDLE  = 3'b000;
parameter  [2:0]  GNT0  = 3'b001;
parameter  [2:0]  GNT1  = 3'b010;
parameter  [2:0]  GNT2  = 3'b011;
parameter  [2:0]  GNT3  = 3'b100;
```

Grey Encoding

```
parameter  [2:0]  IDLE  = 3'b000;
parameter  [2:0]  GNT0  = 3'b001;
parameter  [2:0]  GNT1  = 3'b011;
parameter  [2:0]  GNT2  = 3'b010;
parameter  [2:0]  GNT3  = 3'b110;
```

One Hot Encoding

```
parameter  [4:0]  IDLE  = 5'b0_0001;
parameter  [4:0]  GNT0  = 5'b0_0010;
parameter  [4:0]  GNT1  = 5'b0_0100;
parameter  [4:0]  GNT2  = 5'b0_1000;
parameter  [4:0]  GNT3  = 5'b1_0000;
```

TB - Simulation Behaviour

Concurrent processes (initial, always) run until they stop at one of the following

- **#42**
  Schedule process to resume 42 time units from now
- **wait(cf & of)**
  Resume when expression “cf & of” becomes true
- **@(a or b or y)**
  Resume when a, b, or y changes
- **@(posedge clk)**
  Resume when clk changes from 0 to 1

What Is Not Translated

Initial blocks

- Used to set up initial state or describe finite testbench stimuli
- Don’t have obvious hardware component

Delays

- May be in the Verilog source, but are simply ignored

A variety of other obscure language features

- In general, things heavily dependent on discrete-event simulation semantics
- Certain “disable” statements

Register Inference

The main trick

A reg is not always a latch or flip-flop!

Rule:

Combinational if outputs always depend exclusively on sensitivity list

Sequential if outputs may also depend on previous values or Clock

FSM Moore template

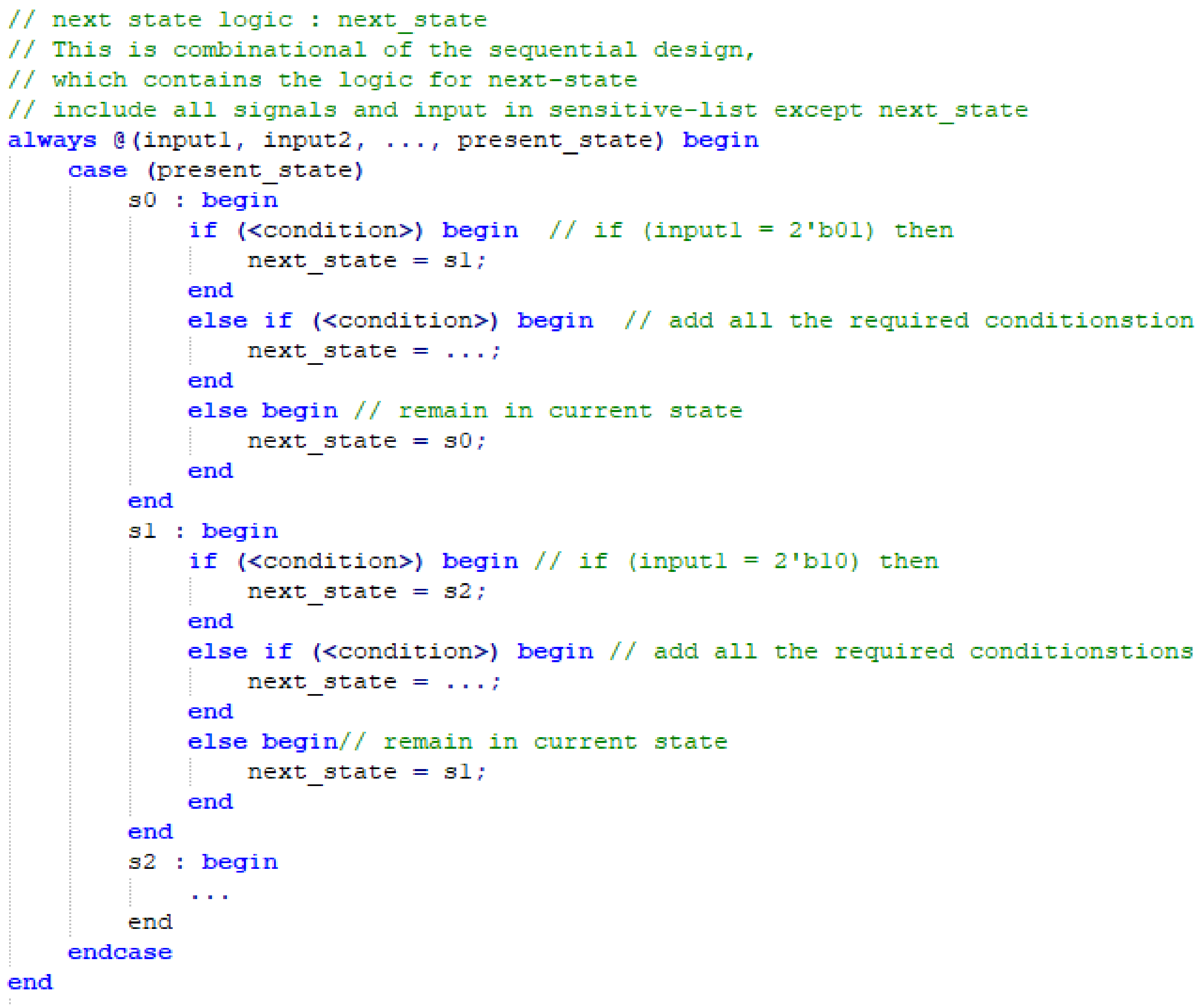
```
module moore_regular_template
(
    input clk, reset,
    input [<size>] input1, input2, ...,
    output reg [<size>] output1, output2, ...,
);

    parameter [<size_state>] // for 4 states : size_state = 1:0
    s0 = 0,
    s1 = 1,
    s2 = 2,
    ... ;

    reg[<size_state>] present_state, next_state;


// state register : present_state
// This process contains sequential part and all the D-FF are
// included in this process. Hence, only 'clk' and 'reset' are
// required for this process.
always @(posedge clk, posedge reset) begin
    if (reset) begin
        present_state <= s0;
    end
    else begin
        present_state <= next_state;
    end
end
```

FSM Moore template



```
// next state logic : next_state
// This is combinational of the sequential design,
// which contains the logic for next-state
// include all signals and input in sensitive-list except next_state
always @(input1, input2, ..., present_state) begin
    case (present_state)
        s0 : begin
            if (<condition>) begin  // if (input1 = 2'b01) then
                next_state = s1;
            end
            else if (<condition>) begin  // add all the required conditionstion
                next_state = ...;
            end
            else begin // remain in current state
                next_state = s0;
            end
        end
        s1 : begin
            if (<condition>) begin // if (input1 = 2'b10) then
                next_state = s2;
            end
            else if (<condition>) begin // add all the required conditionstions
                next_state = ...;
            end
            else begin// remain in current state
                next_state = s1;
            end
        end
        s2 : begin
            ...
        end
    endcase
end
```

FSM Moore template

```
// combination output logic
// This part contains the output of the design
// no if-else statement is used in this part
// include the present_state in sensitive-list in moore FSM
always @(present_state) begin
    // default outputs
    output1 = <value>;
    output2 = <value>;
    ...
    case (present_state)
        s0 : begin
            output1 = <value>;
            output2 = <value>;
            ...
        end
        s1 : begin
            output1 = <value>;
            output2 = <value>;
            ...
        end
        s2 : begin
            ...
        end
    endcase
end
```

```
// optional D-FF to remove glitches
always @(posedge clk, posedge reset)
begin
    if (reset) begin
        new_output1 <= ... ;
        new_output2 <= ... ;
    end
    else begin
        new_output1 <= output1;
        new_output2 <= output2;
    end
end

endmodule
```