

Behavioural HDL coding

Finite State Machines (FSMs):

- Most often used for *specifying sequential logic*.
- What defines a FSM?
- **FSMs are very useful for verification!**
 - 💡 Specify design intent!
 - Can check for state transitions - legal and illegal ones.
 - ⇒ **Need to know how to read/specify FSMs in HDL!**
- FSMs can (even) be synthesized provided rules are followed.

FSMs in Verilog - Synthesis rules

STATES are coded and declared as parameters.

- (• allows use of symbolic names for states)

NEXT STATE LOGIC

- Must be contained in **combinatorial block**, specified in **combinatorial always block** or a function.
- Contains case statement with one branch for every state.
- Each branch defines next state depending on input signals.

STATE REGISTERS

- Must be specified in separate **clocked always block**.
- Only statement in this block is assignment of next state on active edge of clock. 💡 Use next state logic block for this!

OUTPUT LOGIC is (ideally) specified in separate block.

++ Increased readability (and synthesis)!

- (• But can also be specified together with next state logic.)

FSM example

Simple vending machine:

- Accepts 5p, 10p and 20p coins - one at a time.
- No other coins are accepted.
- Releases a chocolate bar if total value of coins is 35p (or greater).
- No change is given. (Simplification!)

Develop an abstract FSM model for this machine.

Verilog FSM model: Ports

Specify the interface of the vending machine:

```
module VM (PutChocBar,Coin5,Coin10,Coin20,Clk,Rst);  
  
output PutChocBar;  reg PutChocBar;  
input  Coin5, Coin10, Coin20;  // Alternatives?  
input  Clk, Rst;
```

Output signals have to be **registered** - so that they can be assigned values inside behavioural blocks.

Verilog FSM model: Internal Signals

Two internal signals have to be specified:

- **next state**
- **actual state**

 Internally declared as registers!

Why?

What size should they be?

Verilog FSM model: Internal Signals

Two internal signals have to be specified:

- **next state**
- **actual state**

 Internally declared as registers! Why?

What size should they be?

Depends on number of **state variables** used for state encoding!

How many states (state variables) do we need for the vending machine?

Verilog FSM model: Internal Signals

Two internal signals have to be specified:

- **next state**
- **actual state**

 Internally declared as registers! Why?

What size should they be?

Depends on number of **state variables** used for state encoding!

How many states (state variables) do we need for the vending machine?

```
reg [2:0] ActState;  
reg [2:0] NextState;
```

Verilog FSM model: State Codes

State encoding: Specify a unique code for each state variable.

```
parameter Paid0    = 3'b000,  
          Paid5    = 3'b001,  
          Paid10   = 3'b010,  
          ...  
          Paid35   = 3'b111;
```

 **parameter** defines (default values) for constants.

NOTE: These can be changed in module instances!

Verilog FSM model: Next State Logic

Usually the **largest block** in the code.

`always` block is sensitive to changes in control inputs and state.

- Determine next state depending on
 - actual state and
 - value of input signals.
- Don't change (actual) state here. Use state registers!

 Remember to add a **default branch!**

■ Allow to go to the initial state when something is wrong.

Next State Logic - Example

```
always @ (ActState or Coin5 or Coin10 or Coin20)
begin
    case (ActState)
        Paid0: ...;
        Paid5: ...;
        Paid10: begin
            if (Coin5) NextState = Paid15;
            else if (Coin10) NextState = Paid20;
            else if (Coin20) NextState = Paid30;
            else NextState = Paid10;
        end
        ...
        default: NextState = Paid0;
    endcase
end
```

Next State Logic - Example: Function



Next state logic can also be defined in a **function**.

```
function [2:0] next_state;
    input [1:0] Coin;  // Alternative!
    input [2:0] CurrentState;
    begin
        case (CurrentState)
            ...
            Paid10: begin
                case (Coin)
                    2'b00: next_state = Paid10; // no coin
                    2'b01: next_state = Paid15; // Coin5
                    2'b10: next_state = Paid20; // Coin10
                    2'b11: next_state = Paid30; // Coin20
                endcase
            end
            ...
        endcase
    end
endfunction
```

Next State Logic - Example: Function

Use **continuous assignment** to re-evaluate combinational logic each time a coin is inserted or the actual state changes.

```
assign NextState = next_state(Coin, ActState);
```

 Need alternative declaration: Why?

```
wire [2:0] NextState;
```

Verilog FSM model: State Registers

Store the actual state.

Change according to spec in **next state logic block**.

State is changed on active edge of the clock.

- Positive clock edge for our VM example.

Reset: `always` block may also contain a **command to reset state registers to initial state**.

State Registers - Example

```
always @ (posedge Clk)
begin
    ActState = NextState;
end
```

With reset option:

```
always @ (posedge Clk or negedge Rst)
begin
    if (!Rst)
        ActState = Paid0;
    else
        ActState = NextState;
end
```

Verilog FSM model: Output Logic

Can be specified dependent on the actual state (or on state and inputs).

Ideally specified in **separate combinational block**.

- Sensitive to actual state (and inputs).
- Sets or resets the output signals.

```
always @ (ActState)
begin
    if (ActState == Paid35) PutChocBar=1'b1;
    else PutChocBar = 1'b0;
end
```

Link to Formal Verification

Remember: **Model/Property Checking**

Characteristics of a design are formally proven or disproved.

Use to check for generic problems or violations of user-defined properties of the behaviour of the design.

- Usually employed at **higher levels** of design process.
- **Properties** are derived from specification.
- **Properties** are expressed in some (temporal) logic.

Checking often involves a (finite) state machine model.



This model ideally is derived from the RTL.

Why?

Link to Formal Verification

Remember: **Model/Property Checking**

Characteristics of a design are formally proven or disproved.

Use to check for generic problems or violations of user-defined properties of the behaviour of the design.

- Usually employed at **higher levels** of design process.
- **Properties** are derived from specification.
- **Properties** are expressed in some (temporal) logic.

Checking often involves a (finite) state machine model.

 This model ideally is derived from the RTL. Why?

++ Automatic model extraction ensures FSM corresponds to design.

- Provided the extraction tool works correctly - of course. ;)
- (• FSMs might need simplification!) Why?

What is left to do is to define/check *interesting* properties.

Summary

FSMs:

- Most often used for *specifying sequential logic*.
- **Very useful for verification!**
 - 💡 Can serve as **behavioural reference** of a design.
 - ++ expressive**
 - ++ high-level**
- Can be synthesized too - provided coding sticks to rules.
- **Link to formal verification:**
 - FSMs of design serve as basis for **model/property checking!**