3. Finite State Machines

Daniel E. Gisselquist, Ph.D.
Lesson Overview

- What is a Finite State Machine?
- Why do I need it?
- How do I build one?

Objectives

- Learn the concatenation operator
- Be able to explain a shift register
- To get basic understanding of Finite State Machines
- To learn how to build and use Finite State Machines
The concatenation operator

\[
\text{always } @(\text{posedge } i_{\text{clk}}) \\
\quad o_{\text{led}} \leftarrow \{o_{\text{led}}[6:0], o_{\text{led}}[7]\};
\]

Composes a new bit-vector from other pieces
The concatenation operator

```verilog
always @(posedge i_clk)
    o_led <= { o_led[6:0], o_led[7] };
```

Simplifies what otherwise would be quite painful

```verilog
always @(posedge i_clk)
begin
    o_led[0] <= o_led[7];
    o_led[1] <= o_led[0];
    o_led[2] <= o_led[1];
    o_led[3] <= o_led[2];
    o_led[4] <= o_led[3];
    o_led[5] <= o_led[4];
    o_led[6] <= o_led[5];
    o_led[7] <= o_led[6];
end
```
A shift register shifts bits through a register

- Can shift from LSB to MSB

```verilog
class always @(posedge i_clk)
o_led <= { o_led[6:0], i_input };
```

- or from MSB to LSB

```verilog
class always @(posedge i_clk)
o_led <= { i_input, o_led[7:1] };
```
You can use this to create a neat LED display as well

- You just need to mix the shift register

```
initial o_led = 8'h1;
always @(posedge i_clk)
if (stb)
o_led <= { o_led[6:0], o_led[7] };
```

- With a counter to slow it down

```
reg [26:0] counter;
reg stb;
initial { stb, counter } = 0;
always @(posedge i_clk)
{ stb, counter } <= counter + 1'b1;
```

- stb here is a *strobe* signal. A *strobe* signal is true for one clock only, whenever an event takes place
You can use this to create a neat LED display as well

- You just need to mix the shift register

```verilog
initial o_led = 8'h1;
always @(posedge i_clk) if (stb)
    o_led <= { o_led[6:0], o_led[7] };
```

- With a counter to slow it down

```verilog
reg [26:0] counter;
reg stb;
initial { stb, counter } = 0;
always @(posedge i_clk)
    { stb, counter } <= counter + 1'b1;
```

- Note that you can assign to a concatenation as well
If you’ve never seen **Wavedrom**, it is an awesome tool!

Here’s a waveform description of our shift register:

```verbatim
case @(posedge i_clk)
    initial o_led = 8’h1;
    always @ (posedge i_clk)
        o_led <= { o_led[6:0], o_led[7] };
endcase
```

What would it take to make the LED’s go *back and forth*?
Let’s build an LED walker!

- Active LED should walk across valid LED’s and back
  
  We’ll assume 8 LEDs
  
  Shift registers don’t naturally go both ways

- Only one LED should be active at any time
- One LED should always be active at any given time

Most of this project can be done in simulation
If you’ve never seen **Wavedrom**, it is an awesome tool! Here’s a waveform description of what I want this design to do:

- This “goal” diagram can help mitigate complexity
Tikz-timing also works nicely for \LaTeX\ users

Our goal will be to create a design with these outputs

- If successful, you’ll see this in GTKwave
Were we building in C, this would be our program:

```c
while (1) {
    o_led = 0x01;
    o_led = 0x02;
    o_led = 0x04;
    // ...
    o_led = 0x80;
    o_led = 0x40;
    // ...
    o_led = 0x04;
    oLed = 0x02;
}
```

How do we turn this code into Verilog?
Case Statement

We could use a giant cascaded \texttt{if} statement

```verilog
always @(posedge i_clk)
    if (o_led == 8'b0000_0001)
        o_led <= 8'h02;
    else if (o_led == 8'b0000_0010)
        o_led <= 8'h04;
    else if (o_led == 8'b0000_0100)
        o_led <= 8'h08;
    else if (o_led == 8'b0000_1000)
        o_led <= 8'h08;
    // ...
    // Don't forget a final \texttt{else}!
    else // if (o_led == 8'b0000_0010)
        o_led <= 8'h01
```

We could use a giant case statement

```verilog
always @(posedge i_clk)
case (o_led)
  8'b0000_0001: o_led <= 8'h02;
  8'b0000_0010: o_led <= 8'h04;
  // ...
  8'b0010_0000: o_led <= 8'h40;
  8'b0100_0000: o_led <= 8'h80;
  8'b1000_0000: o_led <= 8'h40;
  // ...
  8'b0000_0100: o_led <= 8'h02;
  8'b0000_0010: o_led <= 8'h01;
default: o_led <= 8'h01;
endcase
```

Can anyone see a problem with these two approaches?
A better way: Let’s assign an index to each of these outputs

```cpp
// ... using C++ notation again
o_led = 0x01;  // 1
o_led = 0x02;  // 2
o_led = 0x04;  // 3
// ...
o_led = 0x80;  // 8
o_led = 0x40;  // 9
// ...
o_led = 0x04;  // 13
o_led = 0x02;  // 14
```

In software, you might think of this as an *instruction address*
Here’s what an updated waveform diagram might look like

Our goal will be to create a design with these outputs
If successful, you’ll see this in GTKwave
We can now set the result based upon the *instruction address*

```verilog
always @(posedge i_clk)
  case(led_index)
    4'h0: o_led <= 8'h01;
    4'h1: o_led <= 8'h02;
    4'h2: o_led <= 8'h04;
    // ...
    4'h7: o_led <= 8'h80;
    4'h8: o_led <= 8'h40;
    // ...
    4'hc: o_led <= 8'h02;
    4'hd: o_led <= 8'h01;
  default: o_led <= 8'h01;
  endcase
```

- This is an example of a *finite state machine*
The addresses

All we need now is something to drive the *instruction address*

- This is known as the *state* of our finite state machine

```verilog
initial led_index = 0; // Our "state" variable
always @(posedge i_clk)
  if (led_index >= 4'd13)
    led_index <= 0;
  else
    led_index <= led_index + 1'b1;
```
Simulation

Go ahead and simulate this design

- Does it work as intended?
- Did we miss anything?
A finite state machine consists of…

- Inputs
- State Variable,
  
  *Finite* means there are a limited number of states

- Outputs
A finite state machine consists of…

- Inputs // we didn’t have any
- State Variable, // led_index, or addr

*Finite* means there are a limited number of states

- Outputs // o_led

Keep it just that simple.
- State machines are conceptually very simple
- We’ll ignore the excess math here

Two classical FSM forms
- Mealy
- Moore

Two implementation approaches
- One process
- Two process
Outputs depend upon the current state _plus inputs_

```verilog
always @(*)
    if (!i_display_enable)
        o_led = 0;
    else
        case (led_index)
            4'h1: o_led = 8'h01;
            4'h2: o_led = 8'h02;
            4'h3: o_led = 8'h04;
            4'h4: o_led = 8'h08;
            // ...
        endcase
```
Moore

Lesson Overview
Shift Register
Wavedrom
LED Walker
Wavedrom
The Need
Case Statement
The Need
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
▷ Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbiYosys
Integer Clock Divider
Exercise
Conclusion

Outputs depend upon the *current state only*

```
// Update the state
always @(posedge i_clk)
  enabled <= i_display_enable;

// Create the outputs
always @(*)
  if (!enabled)
    o_led = 0;
  else
    case(led_index)
      4'h1: o_led = 8'h01;
      4'h2: o_led = 8'h02;
      // ...
    endcase
endcase
```

The inputs are then used to determine the next state
A one process state machine

- Created with *synchronous* always block(s)

```verilog
initial led_index = 0; // Our "state" variable
always @(posedge i_clk)
begin
    if (led_index >= 4'h0)
        led_index <= 0;
    else
        led_index <= led_index + 1'b1;
    case(led_index)
    4'h0: o_led <= 8'h01;
    // ...
    endcase
end
```
Two Process FSM uses both synchronous and *combinatorial* logic.

```verbatim
always @(*)&
begin
  if (led_index >= 4'hE)
  begin
    next_led_index = 0;
  end
  else
  begin
    next_led_index = next_led_index + 1'b1;
    case(led_index)
      4'h0: o_led = 8'h01;
      //= ...
    endcase
  end
end

always @(posedge i_clk)
  led_index <= next_led_index;
```
Which to use?

Pick whichever finite state machine form ...  
- ... you are most comfortable with

There is no right answer here
Which to use?

Pick whichever finite state machine form . . .

- . . . you are most comfortable with

There is no right answer here

*but people still argue about it!*

- Tastes great
- Less Filling

I tend to use one process FSM’s
Formal Verification is a process to prove your design "works"

- Fairly easy to use
- Can be faster and easier than simulation
- Most valuable
  - Early in the design process
  - For design components, and not entire designs
Formal Verification

- You specify properties your design must have
- A solver attempts to prove if your design has them
- If the solver fails
  - It will tell you what property failed
    By line number
  - It will generate a trace showing the failure
- These traces tend to be much shorter than simulation failure traces
The free version of Yosys supports immediate assertions

Two types

- Clocked – only checked on clock edges

```verilog
// Remember how we only
// used some of the states?
always @(posedge i_clk)
    assert (led_state <= 4'd13);
```

- Combinational – always checked

```verilog
always (@(*))
    assert (led_state <= 4'd13);
```
To verify this design using SymbiYosys,

- You’ll need a script

```plaintext
[options]
mode  prove

[engines]
smtbmc

[script]
read -f formal ledwalker.v
# ... other files would go here
prep -t top ledwalker

[files]
# List all files and relative paths here
ledwalker.v
```
1. **BMC**

```
[options]
mode bmc
depth 20
```

- Examines the first $N$ steps (20 in this case)
- ...looking for a way to break your assertion(s)
- Can find property (i.e. `assert`) failures
- An `assert` is a safety property
  - Succeeds only if *no trace* can be found that makes any one of your assertions fail
Three Basic FV Modes

1. BMC
2. Cover

```
[options]
mode cover
depth 20
```

- Examines the first $N$ steps (20 in this case)
- ...looking for a way to make any cover statement pass

```
always @(posedge i_clk)
  cover(led_state == 4'h0);
```

- No trace will be generated if no way is found
- `cover` is a *liveness* property
  
  Succeeds if one trace, any trace, can be found to make the statement *true*
Three Basic FV Modes

1. BMC
2. Cover
3. Full proof using $k$-induction

```
[options]
mode prove
depth 20
```

- Examines the first $N$ steps (20 in this case)
- Also examines an arbitrary $N$ steps starting in an arbitrary state

The induction step will ignore your initial statements.
Correct functionality must be guaranteed using assert statements.

- Can prove your properties hold for all time
- This is also a safety property check
Assert the design can only contain one of eight outputs

```verilog
always @(*)
begin
    f_valid_output = 0;
    case(o_led)
        8'h01: f_valid_output = 1'b1;
        8'h02: f_valid_output = 1'b1;
        8'h04: f_valid_output = 1'b1;
        8'h08: f_valid_output = 1'b1;
        8'h10: f_valid_output = 1'b1;
        8'h20: f_valid_output = 1'b1;
        8'h40: f_valid_output = 1'b1;
        8'h80: f_valid_output = 1'b1;
    endcase
    assert(f_valid_output);
end
```
It doesn’t work

If you try implementing this design as it is now,

- You’ll be disappointed
- All the LED’s will light dimly
  
  *The LED’s will toggle so fast you cannot see them change*

We need a way to slow this down.
You may remember the integer clock divider

- Let’s use it here

```verilog
always @(posedge i_clk)
if (wait_counter == 0)
    wait_counter <= CLK_RATE_HZ - 1;
else
    wait_counter <= wait_counter - 1'b1;

always @(posedge i_clk)
begin
    stb <= 1'b0;
    if (wait_counter == 0)
        stb <= 1'b1;
end
```
This `wait_counter` is limited in range

- It will only range from 0 to `CLK_RATE_HZ - 1`
- Don’t forget the assertion that `wait_counter` remains in range!

```verilog
always @(posedge i_clk)
assert (wait_counter <= CLK_RATE_HZ - 1);
```

*If your state variable can only take on some values, always make an assertion to that affect*

- Let’s also make sure the `stb` matches the `wait_counter` too

```verilog
always @(posedge i_clk)
assert (stb == (wait_counter == 0));
```
Now we can use `stb` to tell us when to adjust our state.

```verilog
initial led_index = 0;
always @(posedge i_clk)
  if (stb)
    begin
      // The logic inside is just
      // what it was before
      // Only the if(stb) changed
      if (led_index >= 4'd13)
        led_index <= 0;
      else
        led_index <= led_index + 1'b1;
    end // else nothing changes
  // wait for stb to be true before changing state
end
```
Exercise

Try out the tools

1. Recreate this waveform using Wavedrom
Exercise

Try out the tools

1. Recreate this waveform using **Wavedrom**
2. Simulate this design
   - `printf o_led` anytime it changes
   - Look at the trace in gtkwave
     *Does it match our design goal?*
     *Don't forget to slow it down!*
Exercise

Run the tools

1. Recreate this waveform using Wavedrom
2. Simulate this design
3. Run SymbiYosys

Does this design pass?
If it passes, try \texttt{assert(led\_index <= 4)};
Examine the resulting waveform
Exercise

Run the tools

1. Recreate this waveform using Wavedrom
2. Simulate this design
3. Run SymbiYosys

*Does this design pass?*

If it passes, try `assert(led_index <= 4);`

Examine the resulting waveform

*Let’s do this one together*
% verilator -Wall -cc ledwalker.v
%Error: ledwalker.v:61: Can’t find definition of variable: o_leed
%Error: Exiting due to 1 error(s)
%Error: Command Failed /usr/bin/verilator_bin -Wall -cc ledwalker.v
%

- Oops, we misspelled o_led in our case statement
- We also forgot to start our file with `default_nettype none`
- Once fixed, we pass the Verilator check

% verilator -Wall -cc ledwalker.v
%
Running SymbiYosys

```
% sby -f ledwalker.sby
```

- Another syntax error, mislabeled `led_index` as `led_state`
- Let’s try again
Running SymbiYosys

% sby -f ledwalker.sby

It failed, but how? Need to scroll up for the details
Fail in line 96

- Trace file in ledwalker/engine_0/trace.vcd
- Open this in GTKWave, compare to line 96
See the bug?
Running SymbiYosys

- See the bug? oLed starts at 8’h00
- We never initialized oLed to a valid value
- initial oLed = 8’h01; fixes this
Running SymbiYosys

- Same trace file name
- Assertion failed in line 72
Lesson Overview
Shift Register
Wavedrom
LED Walker
The Need
Case Statement
The Need
The Need
The addresses
Simulation
Finite State Machine
Simple
Mealy
Moore
One Process FSM
Two Process FSM
Which to use?
Formal Verification
Assertion
SymbyYosys
Integer Clock Divider
Exercise
Conclusion

Running SymbiYosys

- if (led_index > 4'd12) in line 39 fixes this
Let’s add a quick cover property

```verilog
always @(*)
    cover((led_index = 0) && (o_led == 4'h2));
```
Your turn! Run the tools

1. Recreate this waveform using **Wavedrom**
2. Simulate this design
3. Run SymbiYosys
4. Run your device’s Synthesis tool
   - Make sure your design . . .
     - Passes a timing check
     - Fits within your device
5. Now repeat with the clock divider
**Bonus:** If you have hardware and more than one LED

- Adjust this design for the number of LEDs you have
- Implement this on your hardware

*Does it work?*
Conclusion

What did we learn this lesson?

- What a Finite State Machine (FSM) is
- Why FSM’s are necessary
- Verilog `case` statement
- Verilog cascaded `if`
- Formal `assert` statement
- How to run SymbiYosys
- How to run slow down an FSM
- Verilog is fun!