SYNOPSYS®

Digital Design With Verilog

State Machines

December 2023

CONFIDENTIAL INFORMATION

The information contained in this presentation is the confidential and proprietary information of Synopsys. You are not permitted to disseminate or use any of the information provided to you in this presentation outside of Synopsys without prior written authorization.

IMPORTANT NOTICE

In the event information in this presentation reflects Synopsys' future plans, such plans are as of the date of this presentation and are subject to change. Synopsys is not obligated to update this presentation or develop the products with the features and functionality discussed in this presentation. Additionally, Synopsys' services and products may only be offered and purchased pursuant to an authorized quote and purchase order or a mutually agreed upon written contract with Synopsys.

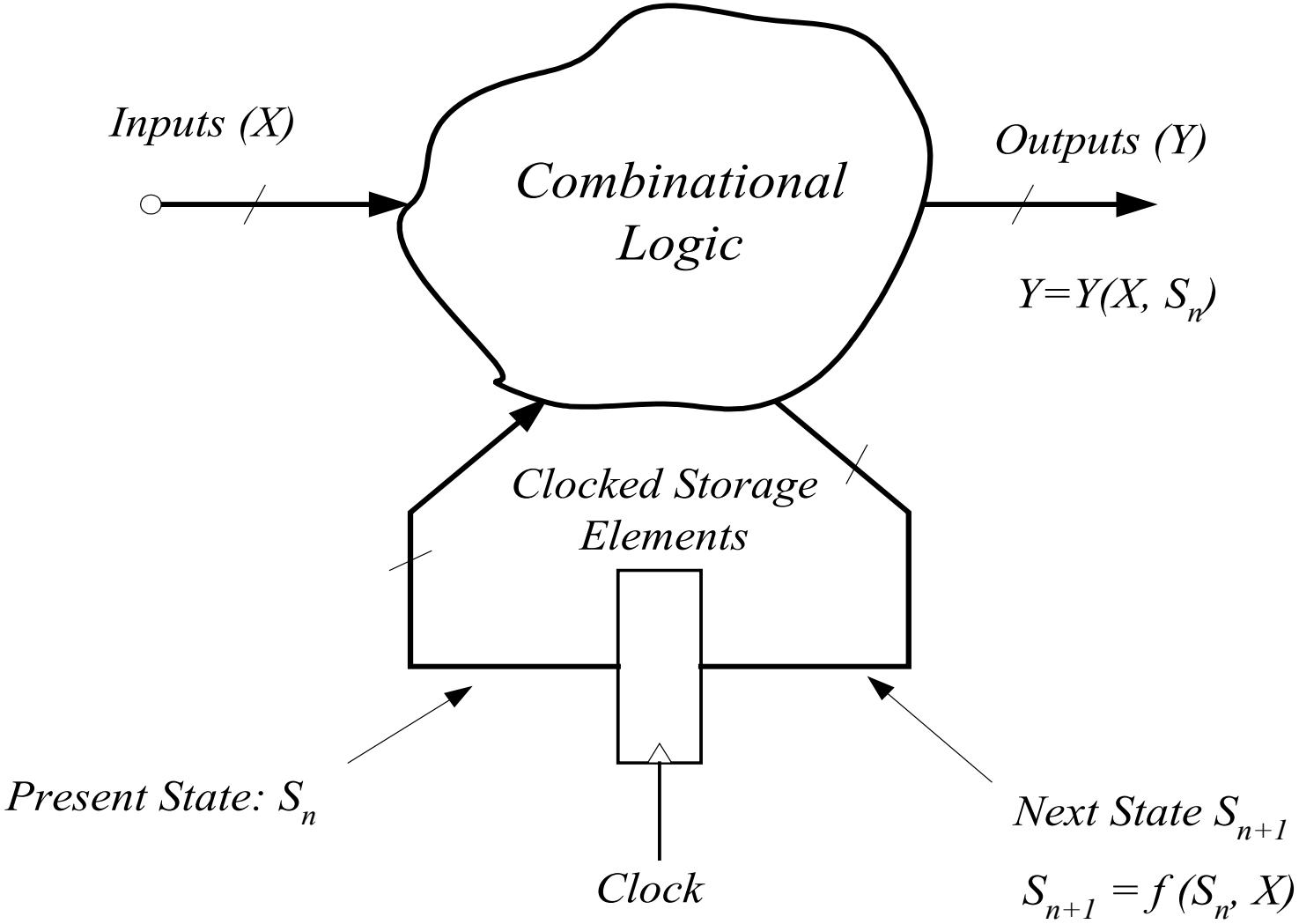
Agenda

- Introduction
- Finite State Machines
- State Machine Charts
- Describing Finite State Machines with Verilog

Finite State Machines Review

Finite-State machine

Abstraction

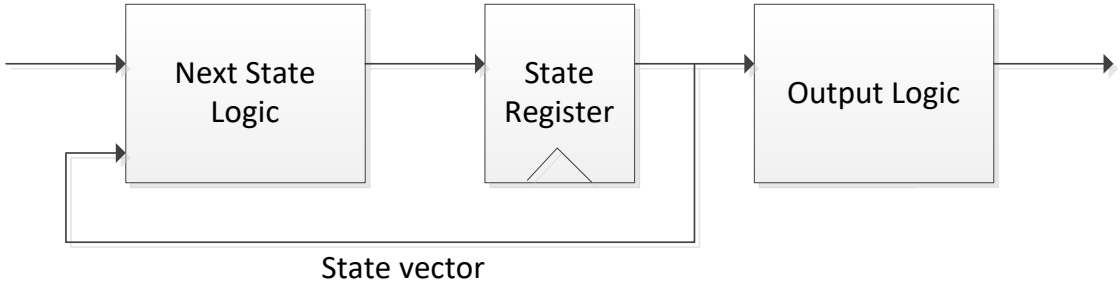


Finite-State machine
Abstraction

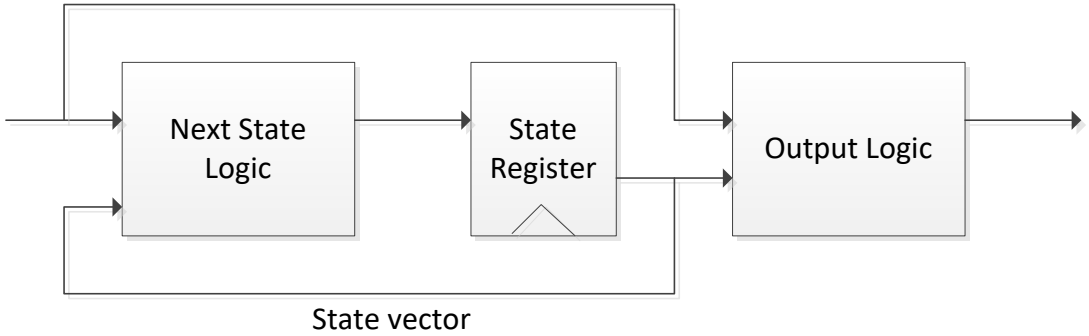
- Clocked Storage Elements: Flip-Flops and Latches should be viewed as **synchronization elements**, not merely as **storage elements** !
- Their main purpose is to **synchronize** fast and slow paths:
  - Prevent the fast path from corrupting the state
- Function of clock signals is to provide a reference point in time when the FSM changes states

FSM - Types

- Moore machine
  - Outputs depend solely on state vector (simplest to design)

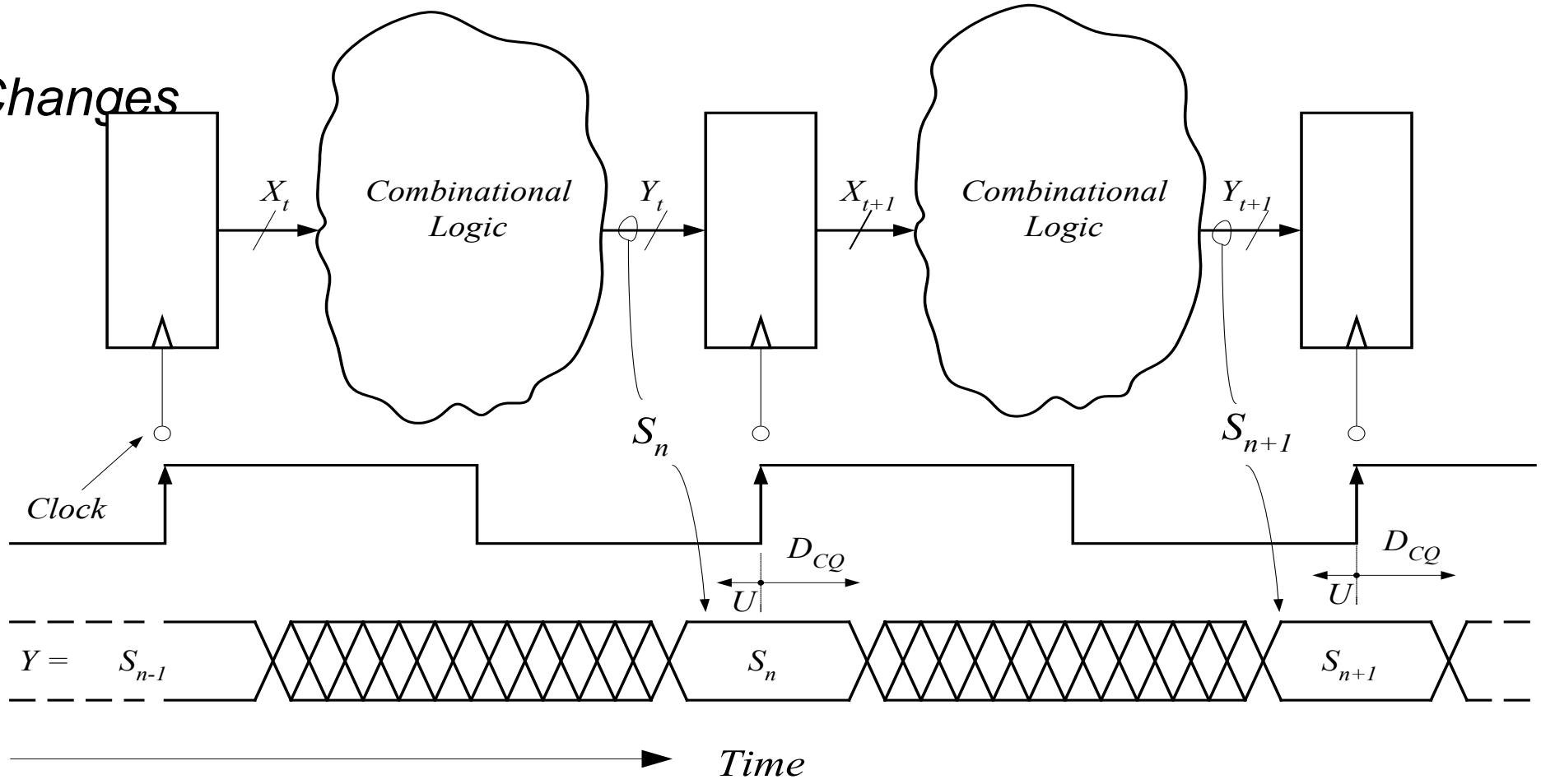


- Mealy machine
  - Outputs depend on inputs and state vector (only use it if smaller or faster machines are needed)



Finite-State machine

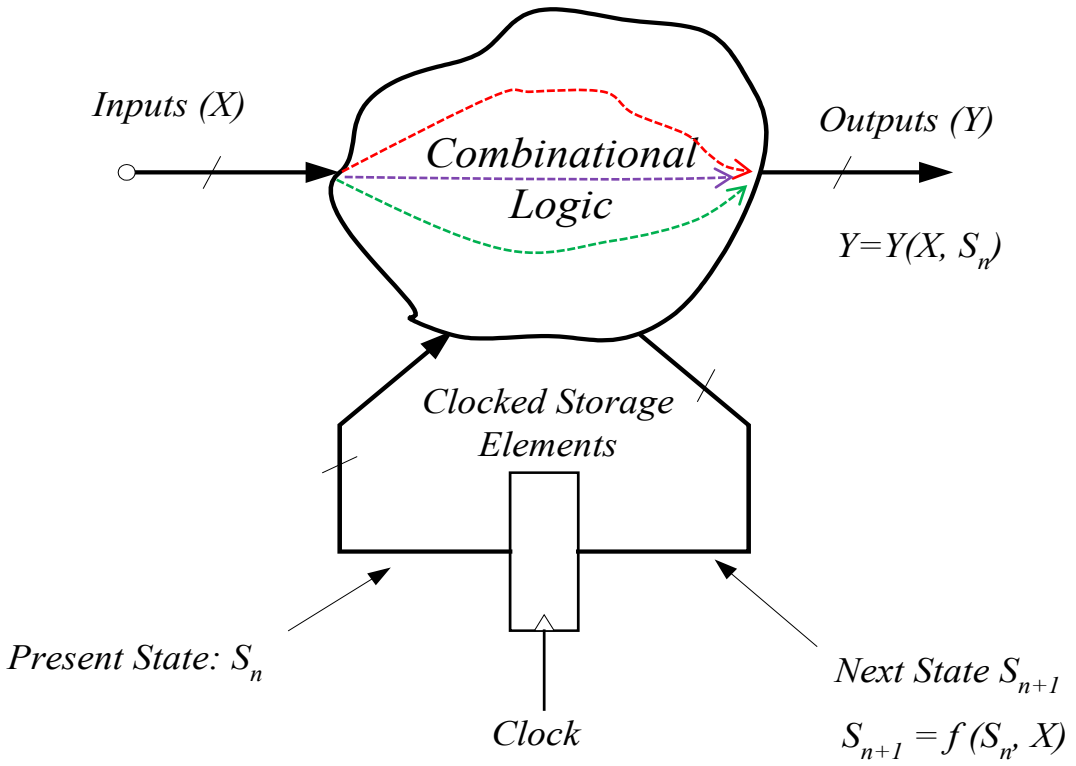
- *State Changes*



Finite-State machine

Critical Path

Critical path is defined as the chain of gates in the longest (slowed) path thought the logic



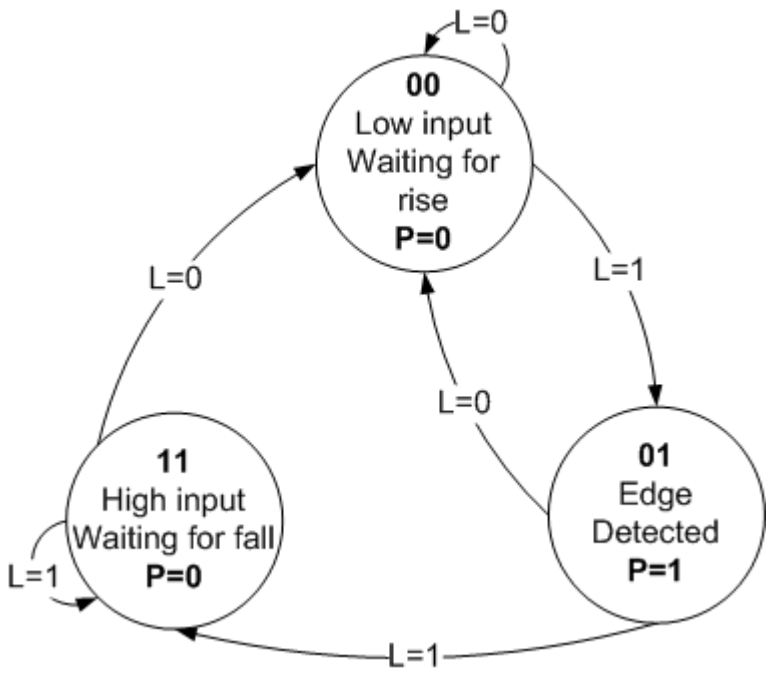
FSM Implementation

Design example

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
- In other words, it's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

FSM Implementation

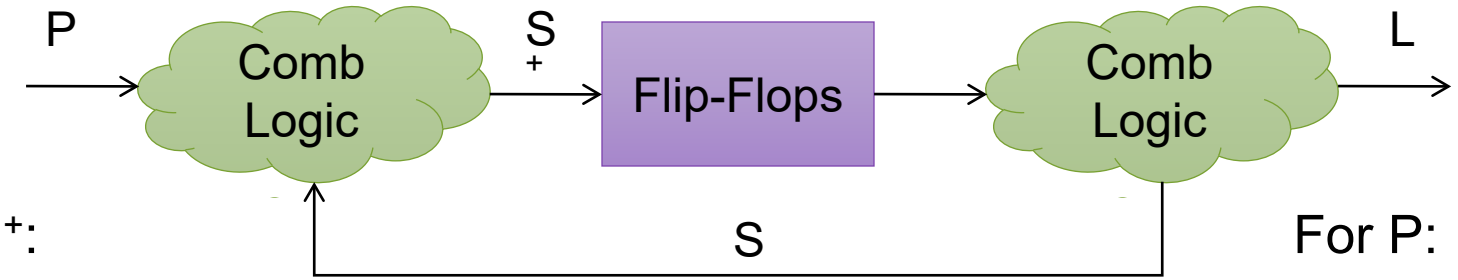
State Diagram *(Moore implementation)*



Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

FSM Implementation

Logic implementation



For S_1^+:

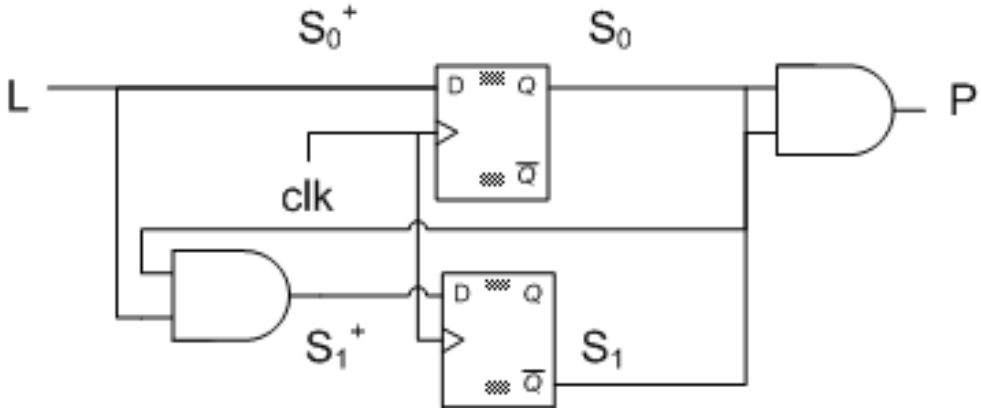
$L\backslash S_1$ / S_0	00	01	11	10
0	0	0	0	X
1	0	1	1	X

For S_0^+:

$L\backslash S_1$ / S_0	00	01	11	10
0	0	0	0	X
1	1	1	1	X

For P:

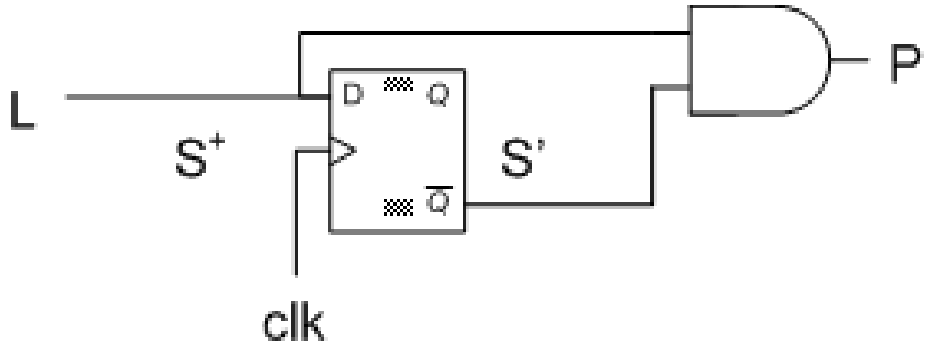
$S_0\backslash S_1$	0	1
0	0	X
1	1	0

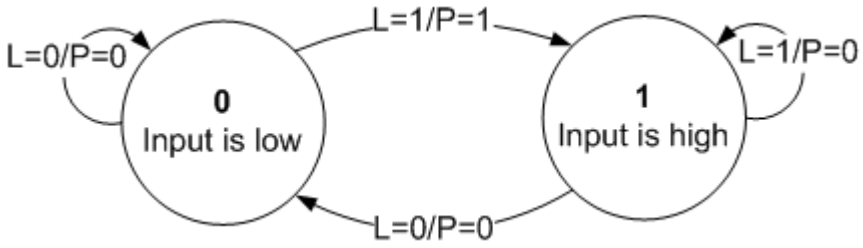


FSM Implementation

Mealy implementation

- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations



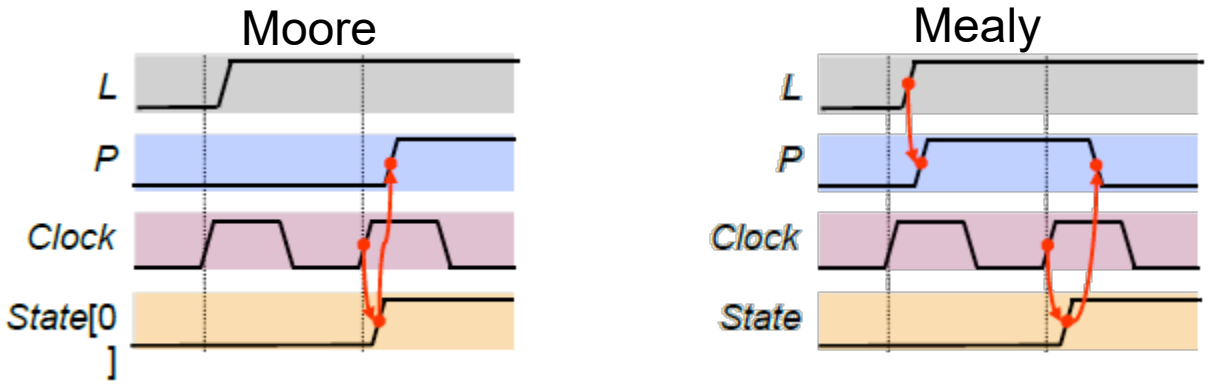


Pre State	In	Next State	Out
S	L	S^+	P
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

FSM Implementation

Moore/Mealy trade-off

- Remember that the difference is in the output:
  - Moore outputs are based on state only
  - Mealy outputs are based on state and input
  - Therefore, Mealy outputs generally occur one cycle earlier than a Moore:

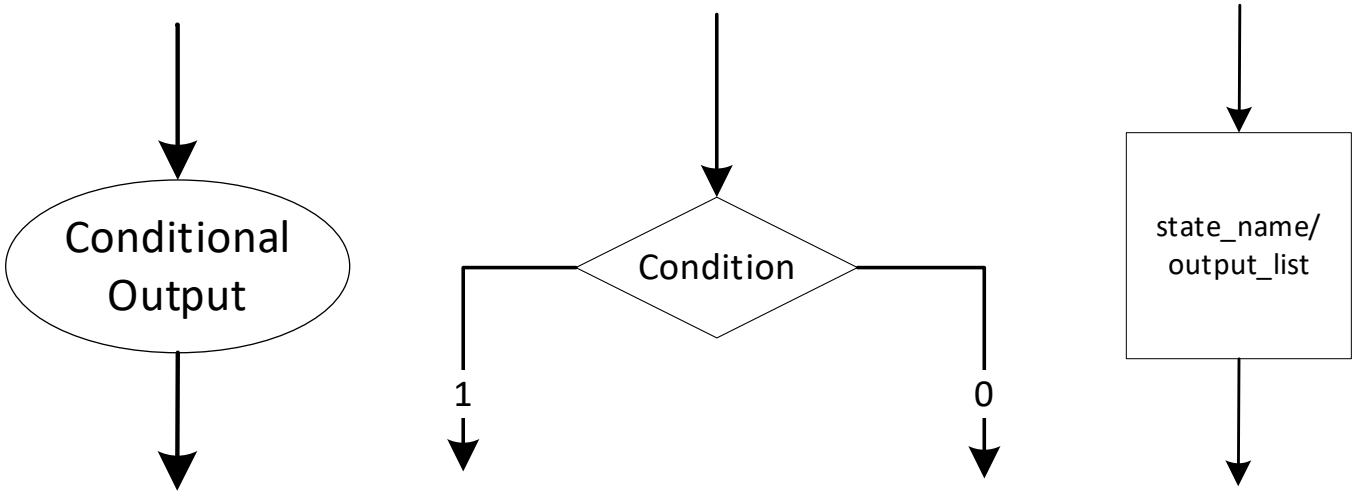


- Compared to a Moore FSM, a Mealy FSM might...
  - Be more difficult to conceptualize and design
  - Have fewer states

State machine Charts

State machine Charts

Basic Elements



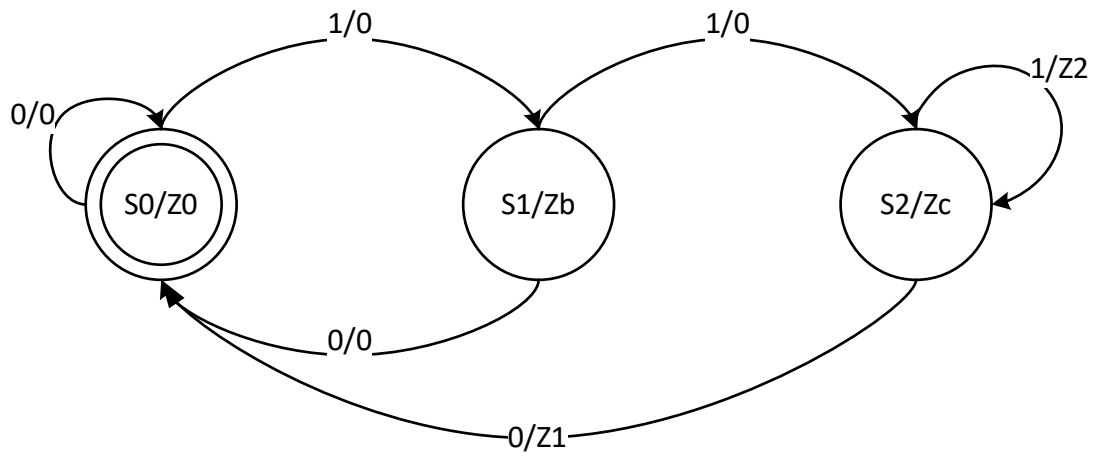
State machine Charts

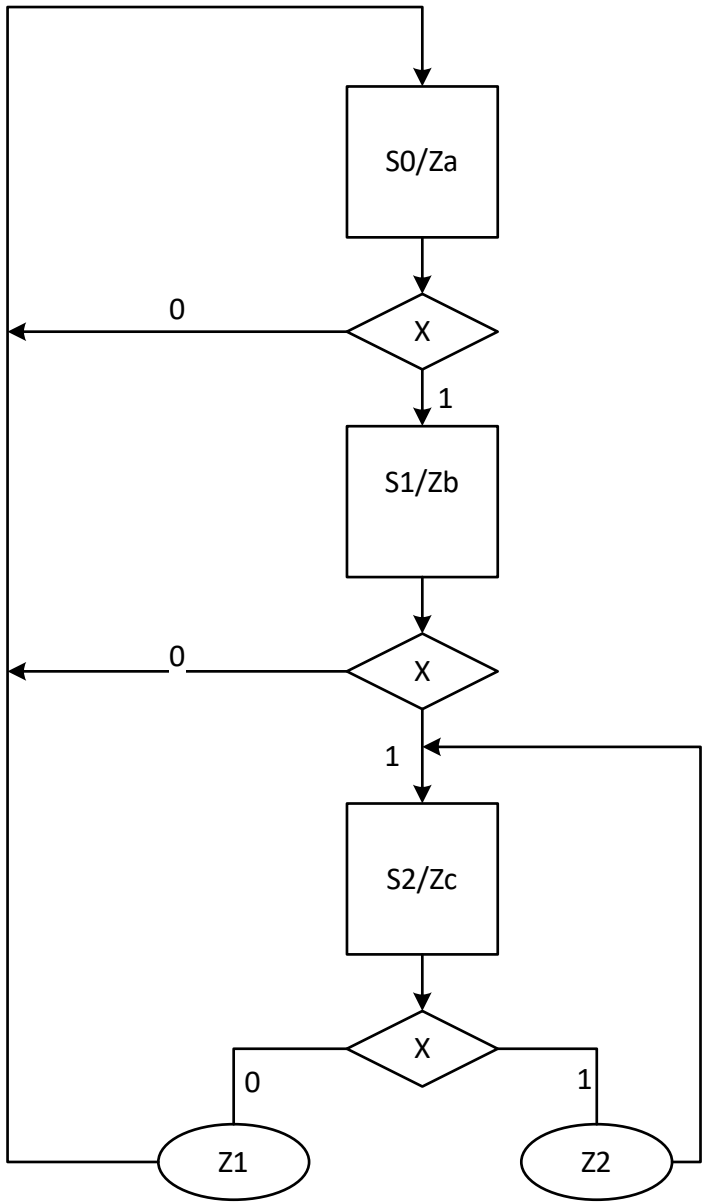
Basic Elements

- Rules
  - Conditionals cannot have feedbacks
  - Conditionals can only have one entrance
  - Conditionals evaluate to 0/1, yes/no
  - One entrance path, one or multiple exit paths (parallelism) in the flow

State machine Charts

Comparison





State machine Charts

Realization

- Resolve incomplete specifications
  - Conditional blocks are not always tested in all inputs
  - Conditional outputs are not explicitly defined in the diagram. Define them in the code.
- Accommodate expressions to result in reductions
  - This creates clean, dichotomic transitions, and enable signals that can be used for datapath blocks

Debugging with Verdi®

Debugging With Verdi

Initial steps

- Compile the simulation executable with a Verdi database

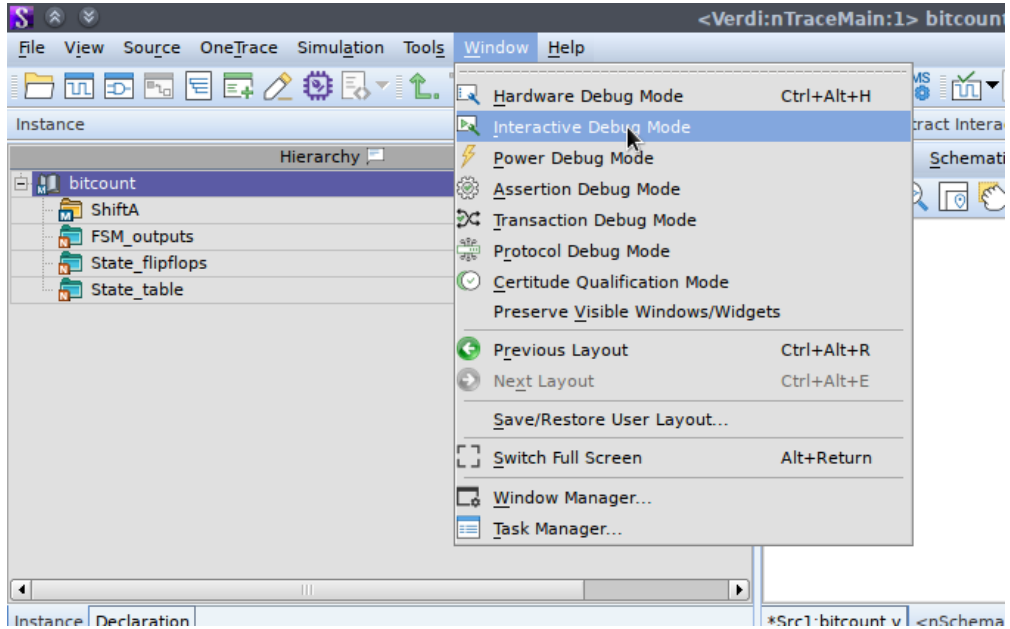
```
vcs –sverilog –debug_access+all -kdb <verilog_files_list>
```

- Launch the simulation with the Verdi GUI opened

```
./simv –verdi <verilog_files_list>
```

Debugging With Verdi

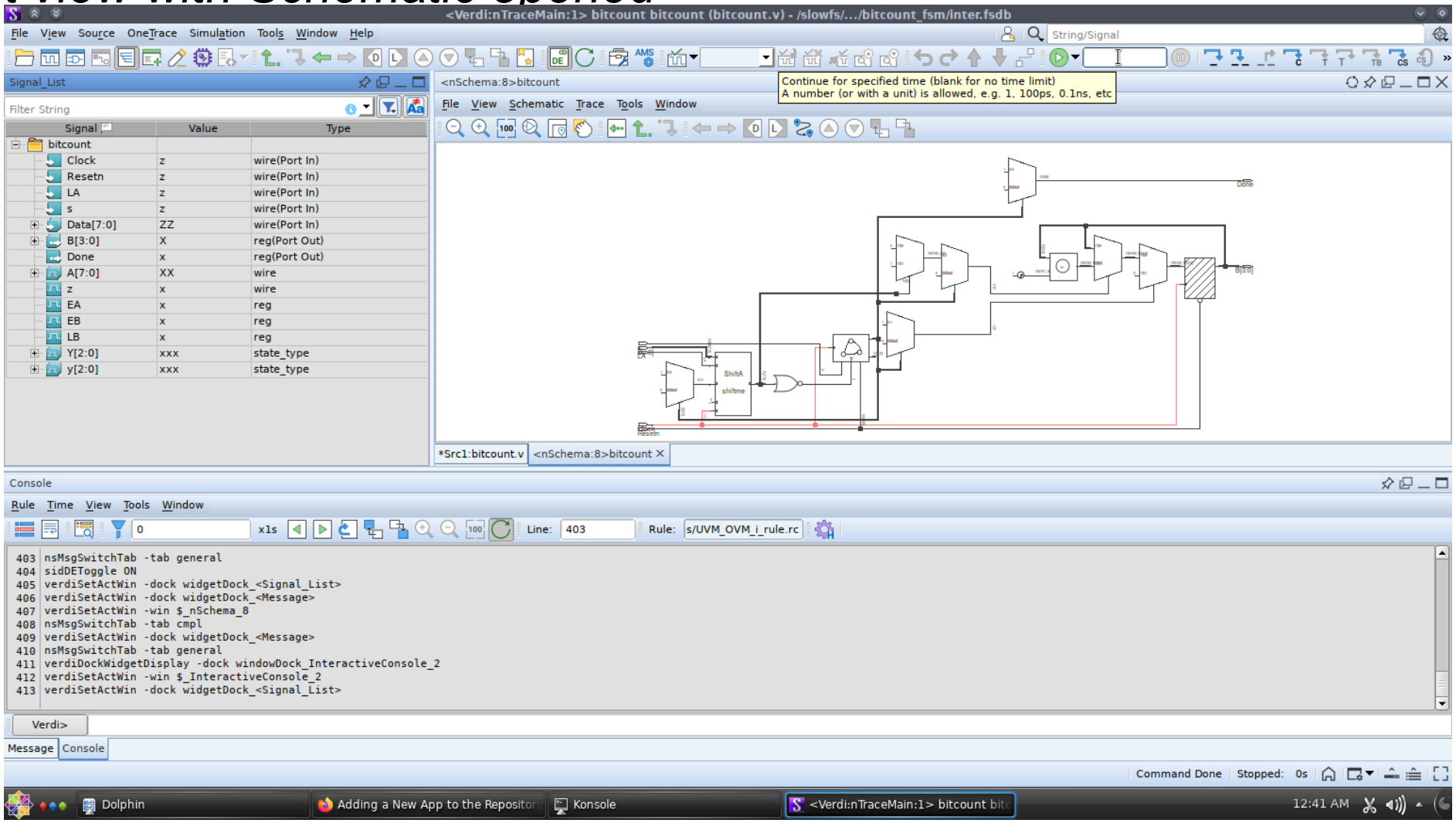
Default Window Layouts



- Pre established window layouts and behaviors for debugging tasks
- Current loaded data remains intact
- For simulation, interactive debug mode serves best.

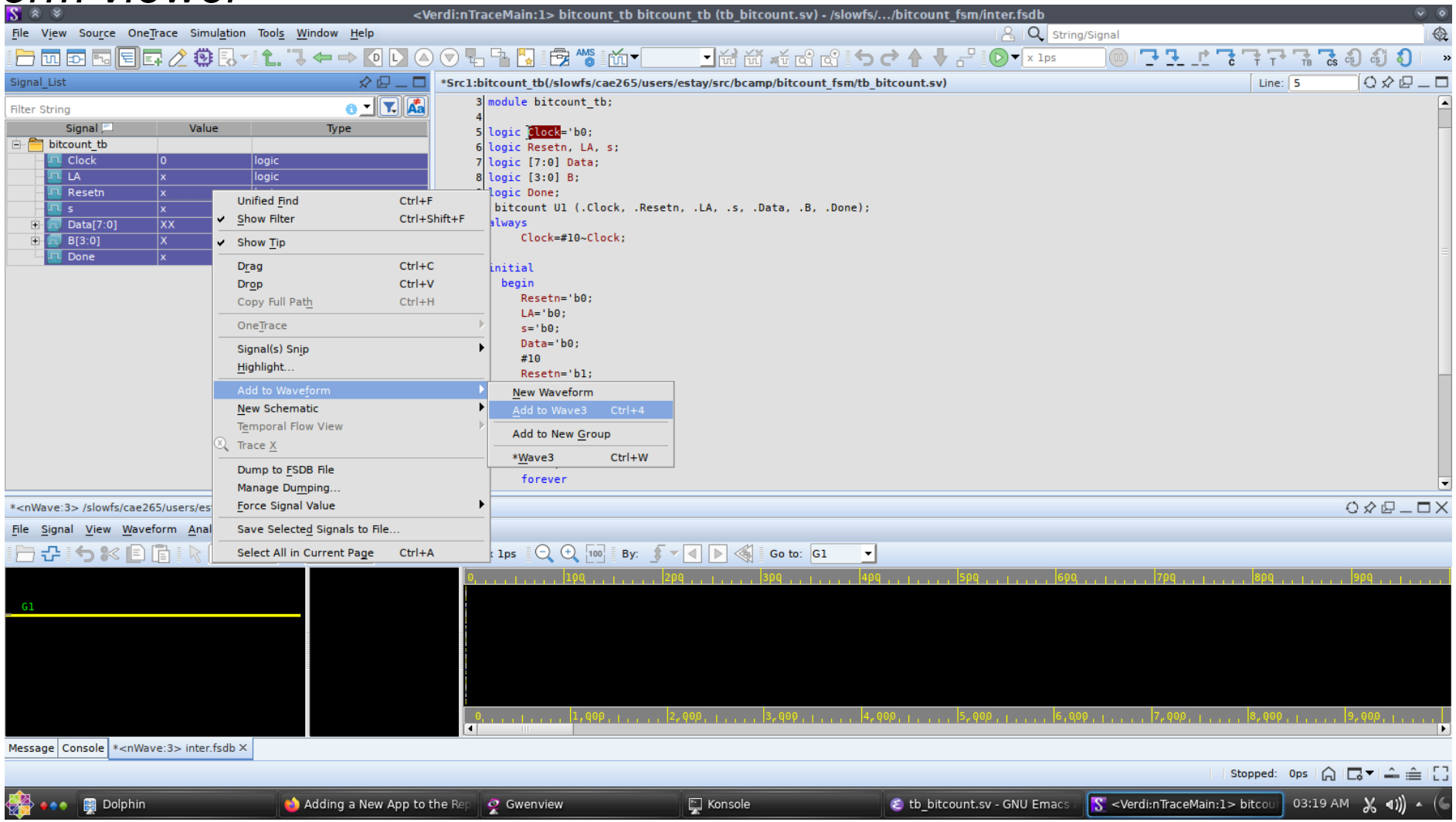
Debugging With Verdi

Default view with Schematic opened



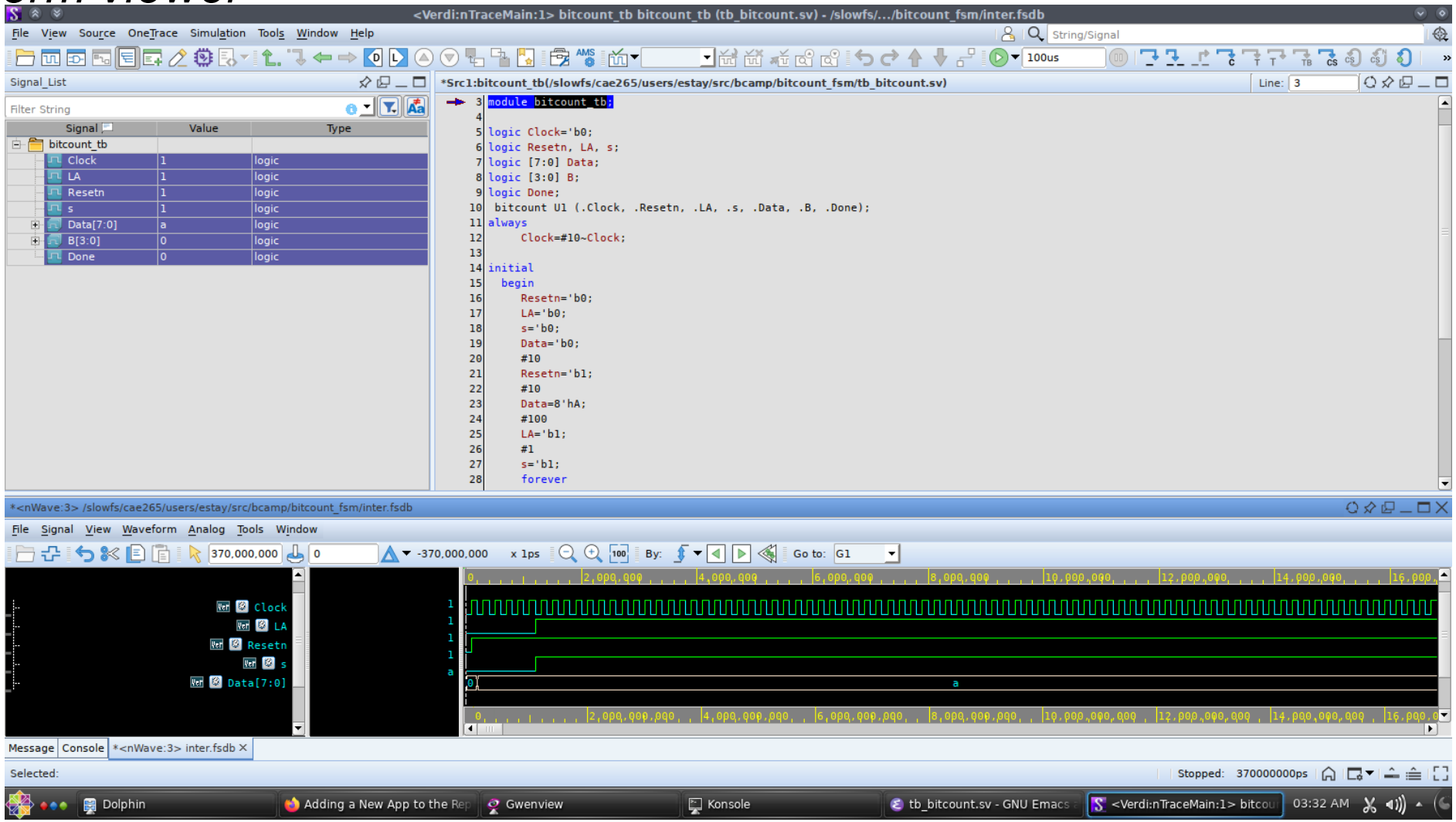
Debugging With Verdi

Waveform viewer



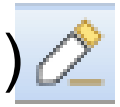
Debugging With Verdi

Waveform viewer



Debugging With Verdi

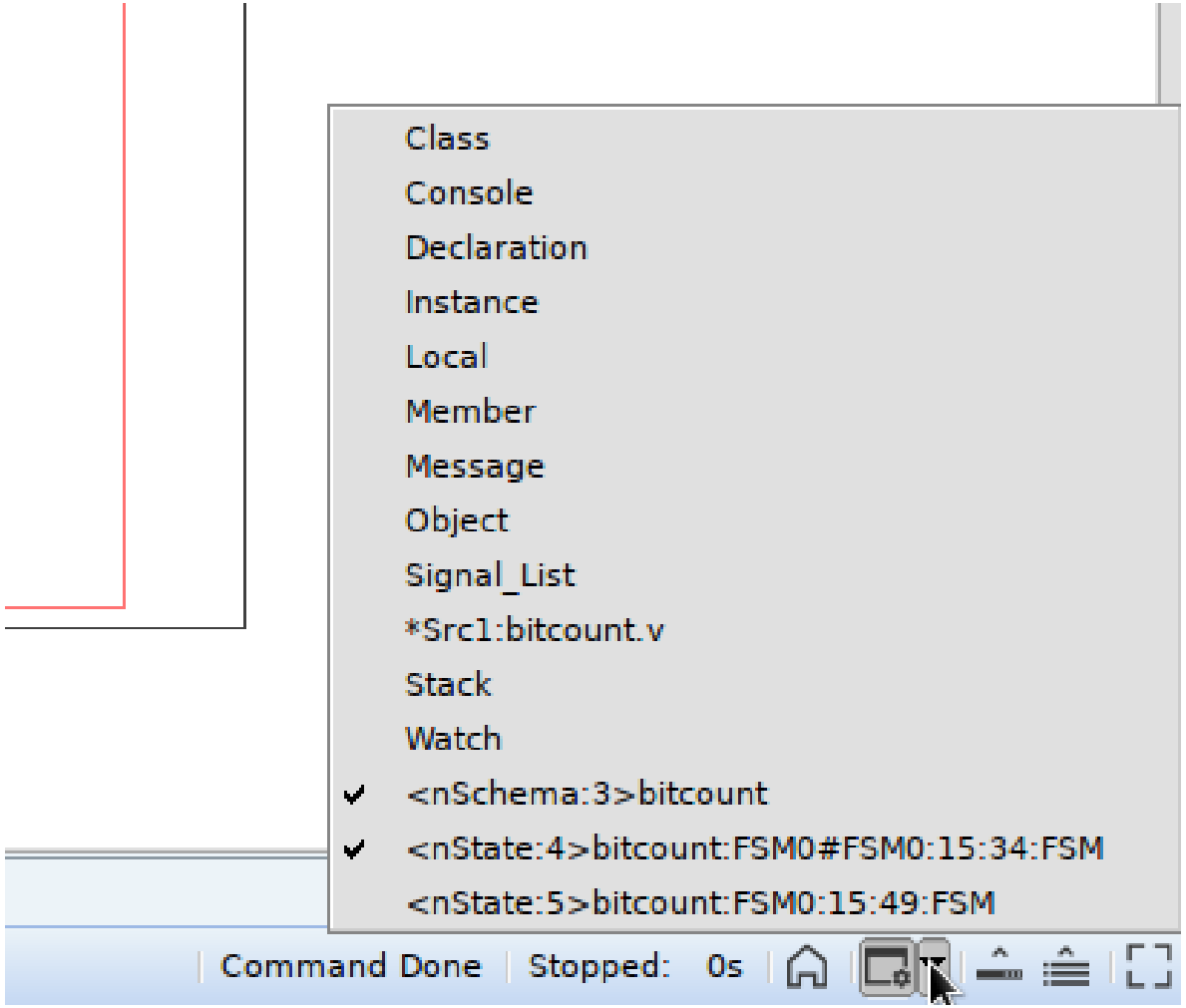
Basic Controls

- New Schematic View 
- Edit Source (from the signal or instance selected) 
- Signal List 
- Driver/Load buttons  
- Simulation time control 

- Pop-up Hierarchy 
- New source Tab/Window 

State machine Charts

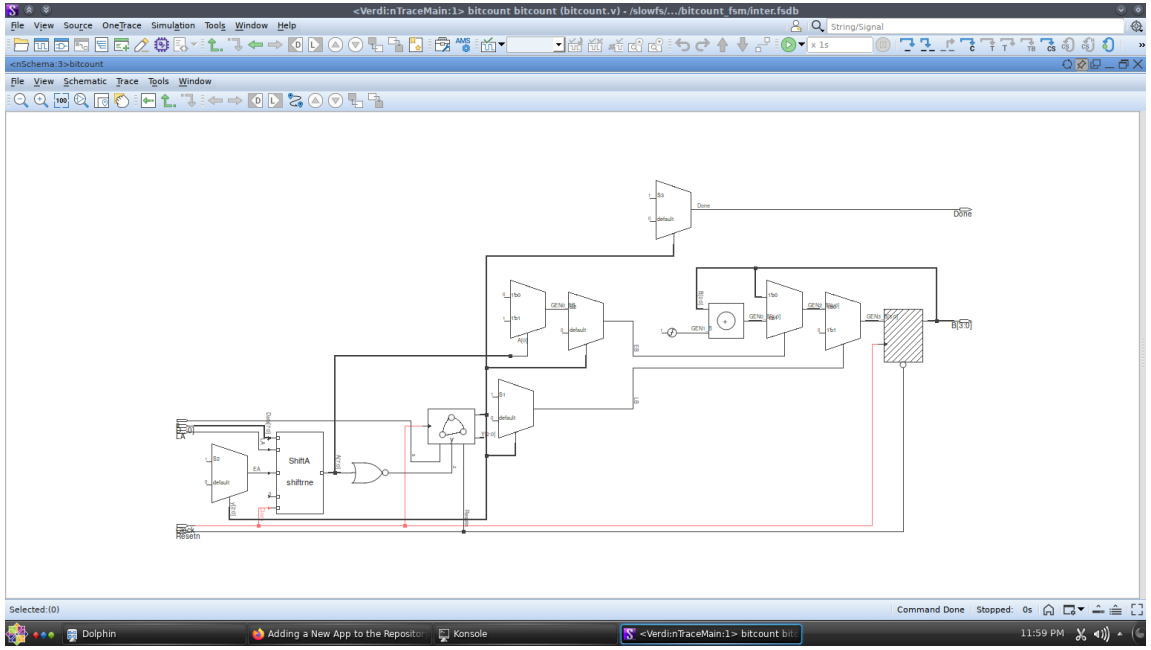
Window manager



- You can select any of the loaded/opened windows from the window manager
- You can go back to the “home” default with the home icon

Debugging With Verdi

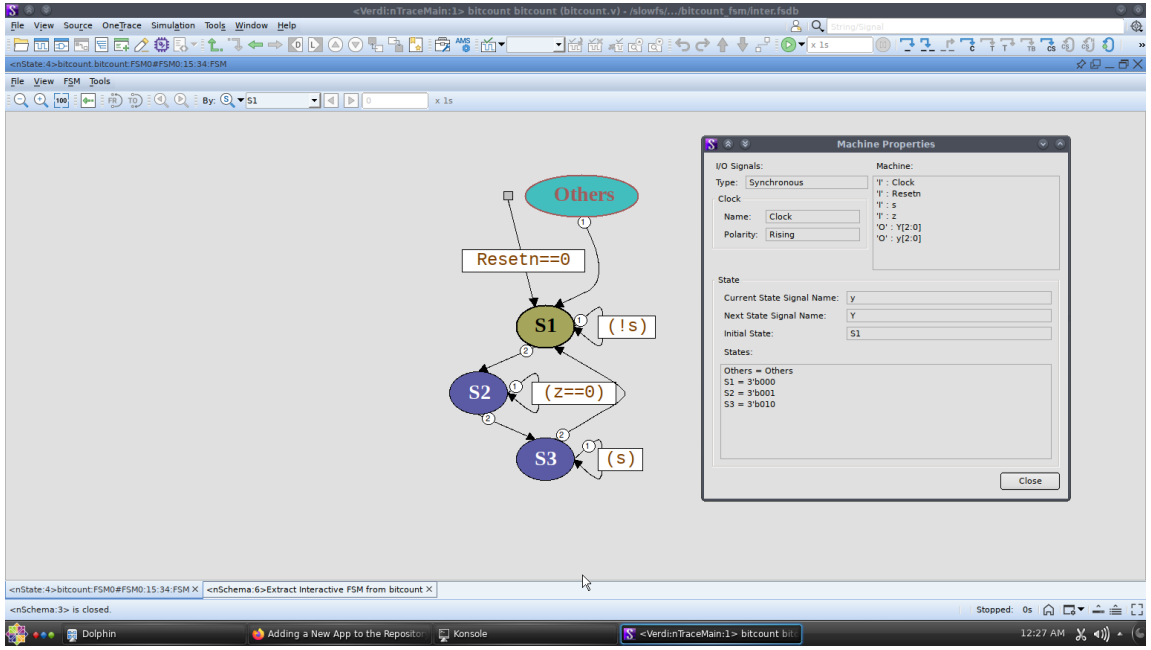
Schematic Viewer



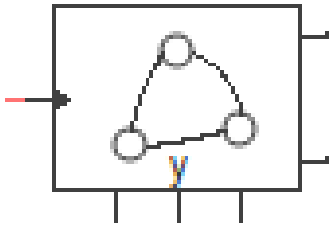
- Shows the schematic coming from the design in simulation
  - Note that the design does not necessarily synthesizes. (E.g. Can have symbolic/software parts).
  - No technology cells/timing
- Extracts state machines, arithmetic circuits and functions

Debugging With Verdi

Schematic Viewer-State machine analyzer (nState)



- State machines extracted from the circuit can be found at blocks with the following icon



- Other extracted information can be found in the Properties window

HDL FSM Implementation Examples

HDL FSM Implementation

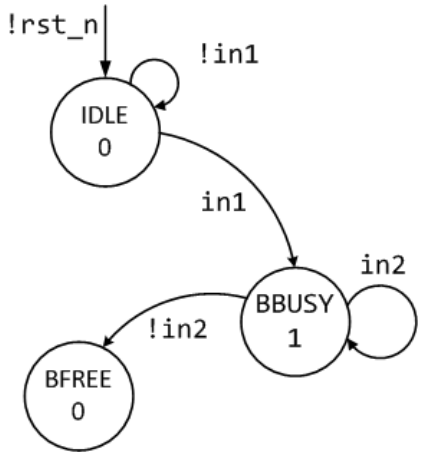
FSM Coding goals

- The FSM coding style should
  - Easily modified to change state encodings and FSM styles
  - Be compact
  - Be easy to code and understand
  - Should facilitate debugging
  - Should yield efficient synthesis results

HDL FSM Implementation

Two Always Block FSM Style (Good Style)

- One of the best Verilog coding styles
- Code the FSM design using two always blocks,
  - One for the sequential state register
  - One for the combinational next-state and combinational output logic.




```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state, next;

always_ff @(posedge clk or negedge rst_n) begin
   if (!rst_n) state <= IDLE;
   else        state <= next;
end

always_comb begin
    next = IDLE;  out1 = 1'b0;
    unique case (state)
      IDLE : if (in1) next = BBUSY;
             else     next = IDLE;
      BBUSY: begin
             out1 = 1'b1;
             if (in2) next = BBUSY;
             else     next = BFREE;
             end
      //...
    endcase
end
```

HDL FSM Implementation

Two Always Block FSM Style (Good Style)

1. Use enumerations to define state names and values. The state variable will only take the values defined in the enum.
2. The sequential always block is coded using nonblocking assignments, in order to module sequential logic (accurately simulate hardware)

```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state, next;

always_ff @(posedge clk or negedge rst_n) begin
   if (!rst_n) state <= IDLE;
   else        state <= next;
end

always_comb begin
    next = IDLE;  out1 = 1'b0;
    unique case (state)
      IDLE : if (in1) next = BBUSY;
             else     next = IDLE;
      BBUSY: begin
             out1 = 1'b1;
             if (in2) next = BBUSY;
             else     next = BFREE;
             end
      //...
    endcase
end
```

1

2

HDL FSM Implementation

Two Always Block FSM Style (Good Style)

3. The combinational always_comb block resolves the sensible variables
4. Assignments within the combinational always_comb block are made using Verilog blocking assignments, in order to module sequential logic (accurately simulate hardware)
5. Unique case forces that only one case branch is used

```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state, next;

always_ff @(posedge clk or negedge rst_n) begin
   if (!rst_n) state <= IDLE;
   else        state <= next;
end

always_comb begin
    next = IDLE;
    out1 = 1'b0;
    unique case (state)
      IDLE : if (in1) next = BBUSY;
             else     next = IDLE;
      BBUSY: begin
             out1 = 1'b1;
             if (in2) next = BBUSY;
             else     next = BFREE;
             end
      //...
    endcase
end
```

3

5

4

HDL FSM Implementation

Two Always Block FSM Style (Good Style)

5. Default output assignments are made before coding the case statement (this eliminates latches and reduces the amount of code required. Also replaces the default: branch of case)
6. Placing a default next state assignment on the line immediately following the always block sensitivity list is a very efficient coding style
7. Force the next state variable to a known value. Avoid X in output or transition variables

6

5

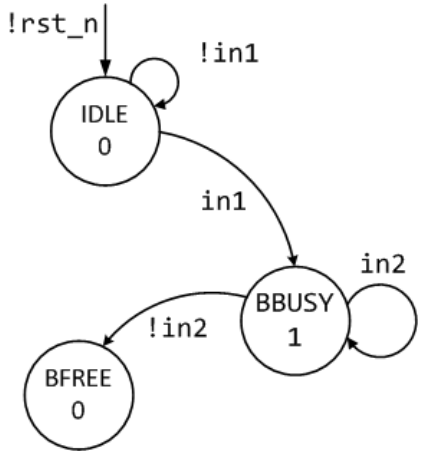
```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state, next;

always_ff @(posedge clk or negedge rst_n) begin
   if (!rst_n) state <= IDLE;
   else        state <= next;
end

always_comb begin
    next = IDLE;  out1 = 1'b0;
    unique case (state)
      IDLE : if (in1) next = BBUSY;
             else     next = IDLE;
      BBUSY: begin
             out1 = 1'b1;
             if (in2) next = BBUSY;
             else     next = BFREE;
             end
      //...
    endcase
end
```

HDL FSM Implementation

One Always Block FSM Style
(Avoid This Style!)

- One of the most common FSM coding styles in use today
  - It is more verbose
  - It is more confusing
  - It is more error prone

(comparable two always block coding style)




```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      state <= IDLE;
      out1  <= 1'b0;
    end
    else begin
      state <= IDLE;  out1 <= 1'b0;
      case (state)
        IDLE : if (in1) begin
                  state <= BBUSY;
                  out1  <= 1'b1;
               end
               else     state <= IDLE;
        BBUSY: if (in2) begin
                  state <= BBUSY;
                  out1 <= 1'b1;
               end
               else     state <= BFREE;
       endcase
    end
end
```

HDL FSM Implementation

One Always Block FSM Style
(Avoid This Style!)

1. A declaration is made for state. Not for next.
2. The state assignments do not correspond to the current state of the case statement, but the state that case statement is transitioning to.
   This is **error prone** (but it does work if coded correctly).

```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      state <= IDLE;
      out1  <= 1'b0;
    end
    else begin
      state <= IDLE;  out1 <= 1'b0;
      case (state)
        IDLE : if (in1) begin
                 state <= BBUSY;
                 out1  <= 1'b1;
               end
               else     state <= IDLE;
        BBUSY: if (in2) begin
                 state <= BBUSY;
                 out1 <= 1'b1;
               end
               else     state <= BFREE;
       endcase
    end
end
```

1

2

HDL FSM Implementation

One Always Block FSM Style (Avoid This Style!)

3. There is just one sequential always block, coded using nonblocking assignments.
4. Difficult to track the changes
5. All outputs will be registered (may generate involuntary registers).
6. No asynchronous Mealy outputs can be generated from a single synchronous always block.

```
type enum logic [1:0] {IDLE=2'b00, BBUSY=2'b01,BFREE=2’b10 }
   state_type;
state_type state;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      state <= IDLE;
      out1  <= 1'b0;
    end
    else begin
      state <= IDLE;  out1 <= 1'b0;
      case (state)
        IDLE : if (in1) begin
                  state <= BBUSY;
                  out1  <= 1'b1;
               end
               else     state <= IDLE;
        BBUSY: if (in2) begin
                  state <= BBUSY;
                  out1 <= 1'b1;
               end
               else     state <= BFREE;
       endcase
    end
end
```

3

4

Thank You