EE 531: ADVANCED VLSI DESIGN

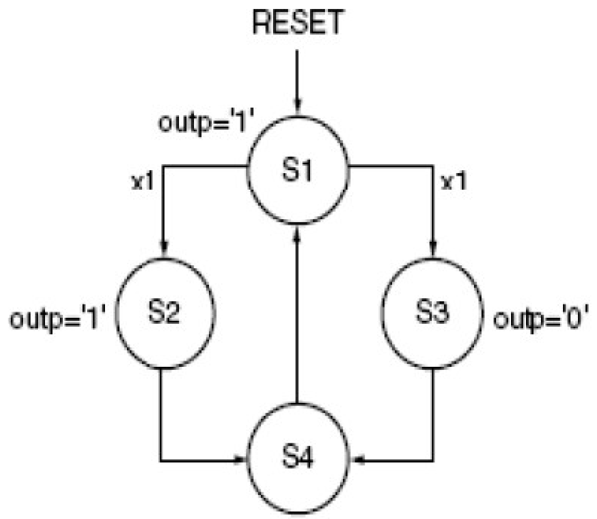
State Machine Techniques

Nishith N. Chakraborty

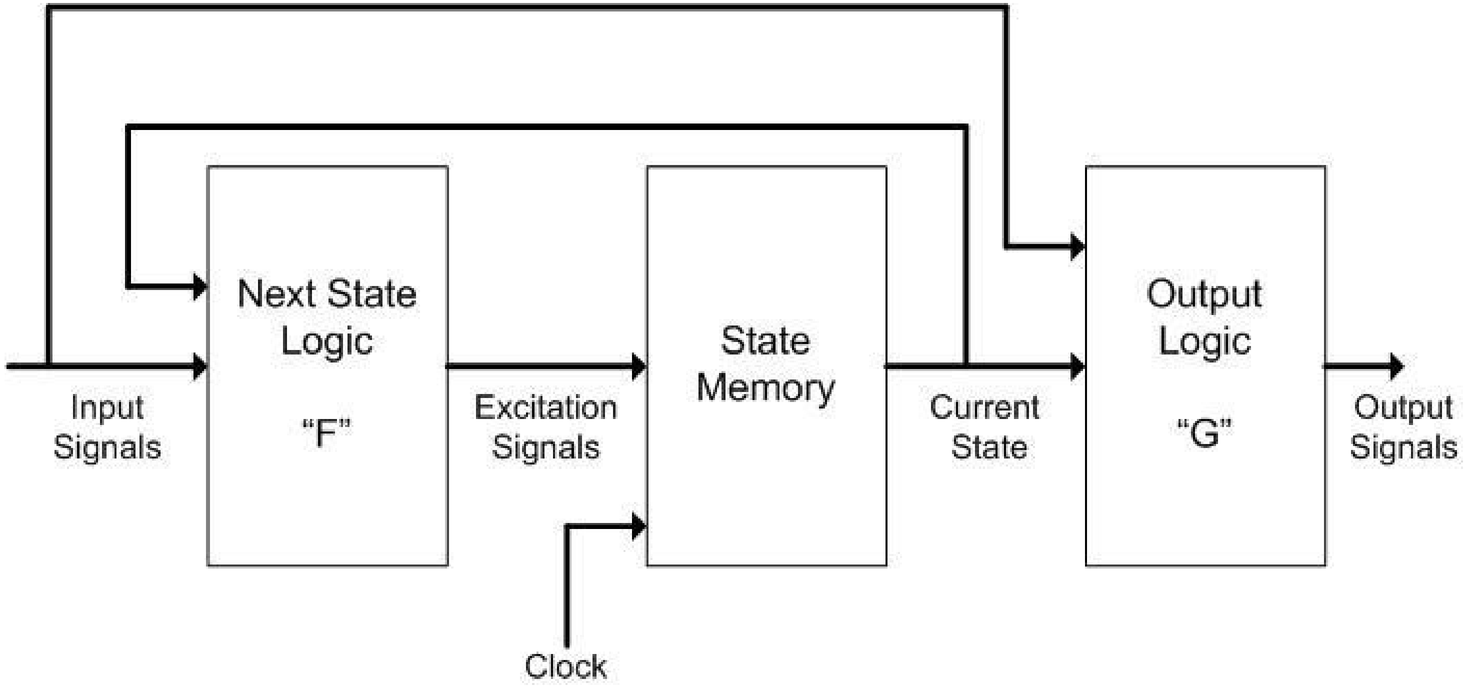
January, 2025

STATE MACHINES

- Finite state machines (FSM) are key elements in logic design
- A synthesizer can perform some state optimization on FSMs to minimize circuit area & delay
- This optimization is often only available if the FSM model fits pre-specified templates

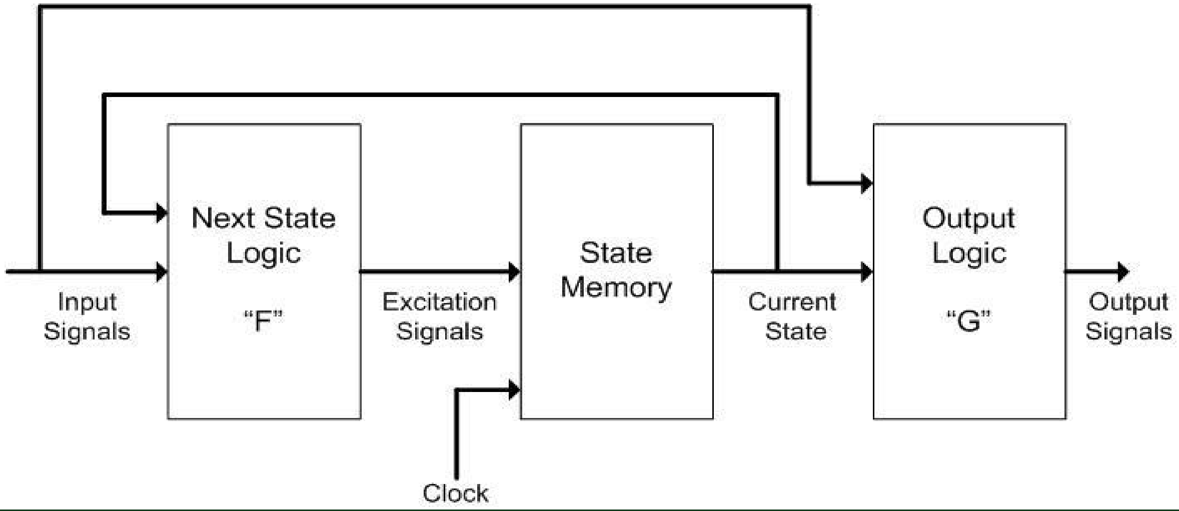


FSM REPRESENTATION



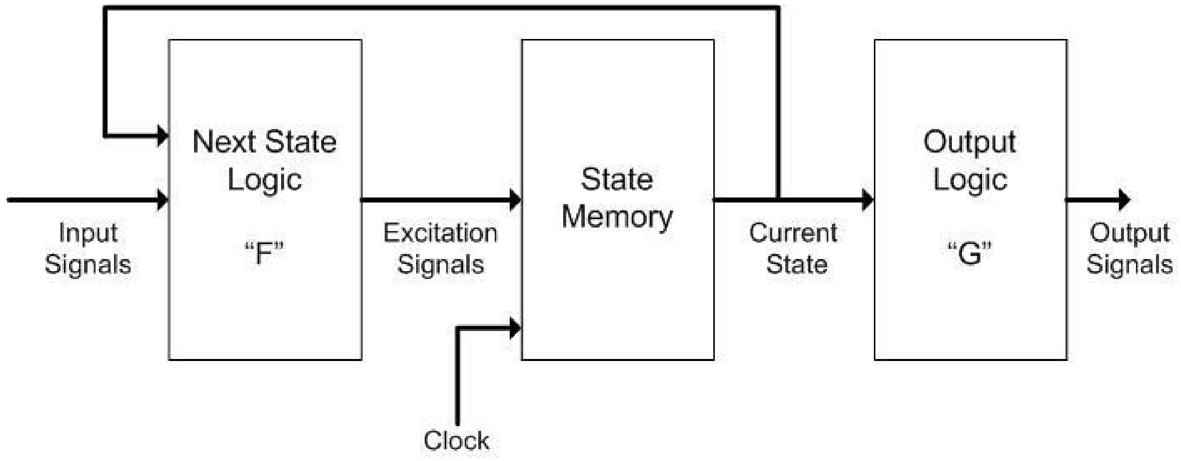
MEALY MACHINE

- Outputs depend on previous (state) AND present inputs
- Input change causes an immediate output change
  - <u>Asynchronous signals</u>



MOORE MACHINE

- Outputs depend ONLY on previous inputs (i.e. current states)
- Outputs change synchronously with state changes

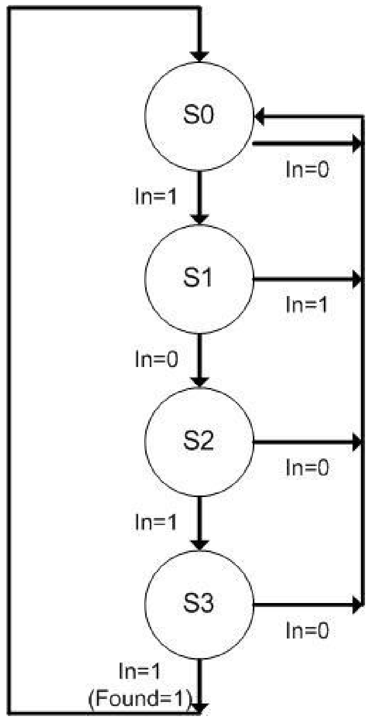


STATE MACHINES IN HDL

1. State memory
   - ➢ Use an `always_ff` procedural block (sensitive to clock) to update the states
   - ➢ State memory based on “user-enumerated” or “pre-defined” data types
   - ➢ Synchronous and asynchronous reset options
2. Next state logic “F”
3. Output logic “G”

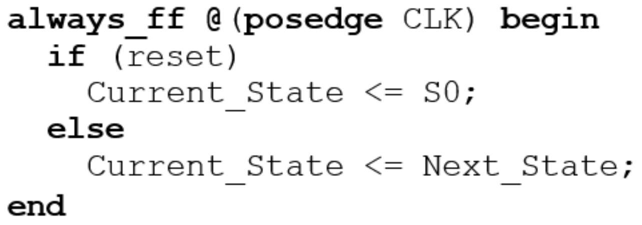
EXAMPLE: SEQUENCE DETECTOR

- Design a machine by hand that takes in a serial bit stream and looks for the pattern “1011”
    - Similar machine useful for detecting malicious network traffic
    - When pattern found, signal called “Found” asserted
    - Must store and recall historic system state
- State diagram
    - If pattern invalidated, restart



STATE MEMORY

- Use process that updates “Current_State” with the “Next_State”
- Describe D flip-flop's using `always_ff`
- This will make assignment on the rising edge of CLK



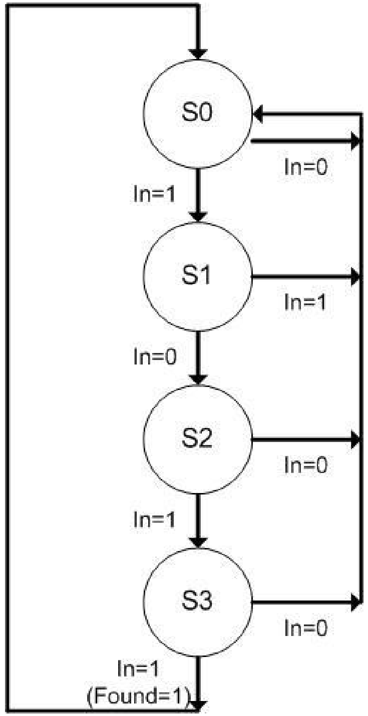
```
always_ff @(posedge CLK) begin
  if (reset)
    Current_State <= S0;
  else
    Current_State <= Next_State;
end
```

PROCEDURAL BLOCK FOR NEXT STATE LOGIC

- Use another always block to construct next state logic "F"

```
always_comb
  case (Current_State)
    S0: if (In==1'b0) Next_State = S0;
        else          Next_State = S1;
    S1: if (In==1'b0) Next_State = S2;
        else          Next_State = S0;
    S2: if (In==1'b0) Next_State = S0;
        else          Next_State = S3;
    S3: if (In==1'b0) Next_State = S0;
        else          Next_State = S0;
  endcase
```

"F" always_comb updates Next_State;
does not update Current_State



PROCESS FOR OUTPUT LOGIC

- Use a third block to construct output "G"
- Inputs to expressions dictate whether state machine is Mealy or Moore type
- For starters, we'll use combinational logic for G

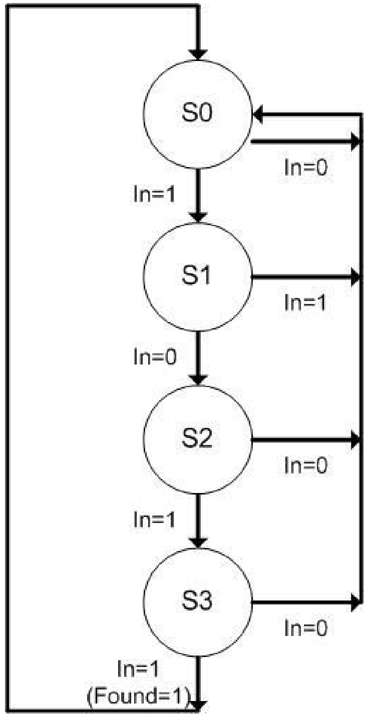
CAL POLY
SAN LUIS OBISPO

MEALY OUTPUT LOGIC

- Use a third process to construct output "G"

```
always_comb
  case (Current_State)
    S0: if      (In==1'b0) Found = 0;
        else if (In==1'b1) Found = 0;
    S1: if      (In==1'b0) Found = 0;
        else if (In==1'b1) Found = 0;
    S2: if      (In==1'b0) Found = 0;
        else if (In==1'b1) Found = 0;
    S3: if      (In==1'b0) Found = 0;
        else if (In==1'b1) Found = 1;
  endcase
```

Executes always_comb whenever inputs In or Current_State changes



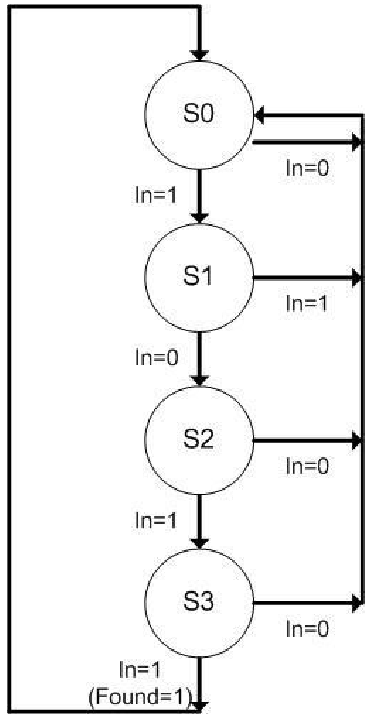
CAL POLY
SAN LUIS OBISPO

MOORE OUTPUT LOGIC

- Use a third process to construct output "G"

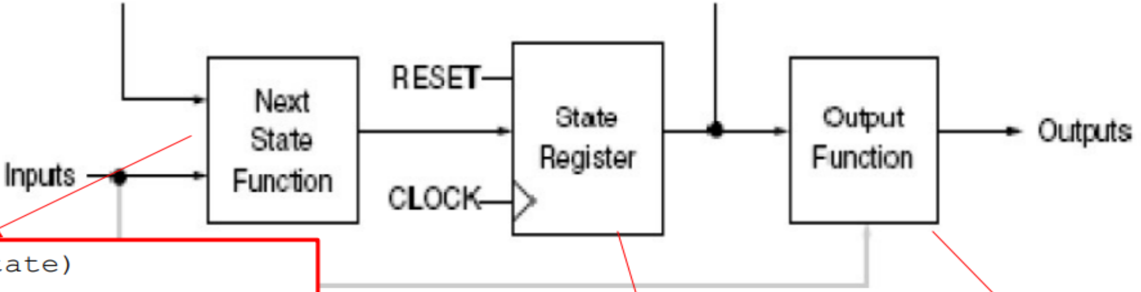
```
always_comb
  case (Current_State)
    S0: Found = 0;
    S1: Found = 0;
    S2: Found = 0;
    S3: Found = 1;
  endcase
```

Only change in current state executes & updates output Found



CAL POLY
SAN LUIS OBISPO

GENERAL VIEW OF STATE MACHINE LOGIC



Process 1

```
case (Current_State)
  Idle:
    Next_State = Grab_Data;
  Grab_Data:
    Next_State = Wait_Ack;
  Wait_Ack:
    if (go_pulse==1'b1)
      Next_State = Send_Data;
    else
      Next_State = Abort;
  ...
```

Process 2

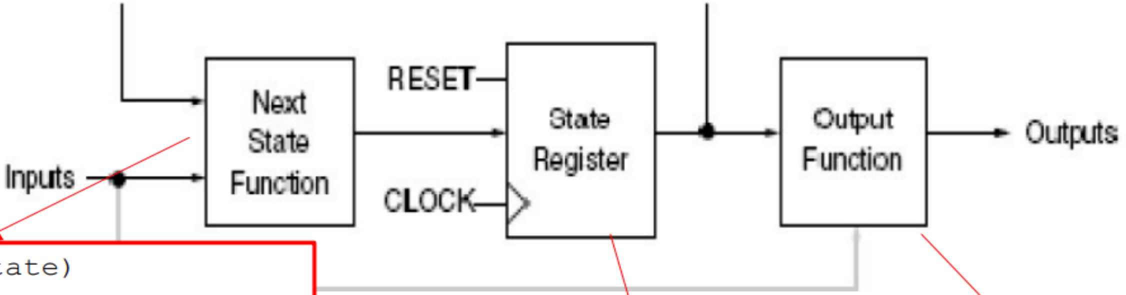
```
if (Reset==1'b1)
  Current_State <= Idle;
else
  Current_State <= Next_State;
...
```

Process 3

```
assign Signal_Sending = (Current_State==Send_Data) ? 1'b1 : 1'b0;
assign Machine_Ready = (Current_State==Idle) ? 1'b1 : 1'b0;
```

CAL POLY
SAN LUIS OBISPO

GENERAL VIEW OF STATE MACHINE LOGIC



Process 1

```
case (Current_State)
  Idle:
    Next_State = Grab_Data;
  Grab_Data:
    Next_State = Wait_Ack;
  Wait_Ack:
    if (go_pulse==1'b1)
      Next_State = Send_Data;
    else
      Next_State = Abort;
  ...
```

Process 2

```
if (Reset==1'b1)
  Current_State <= Idle;
else
  Current_State <= Next_State;
...
```

May not need in own process, depends on complexity

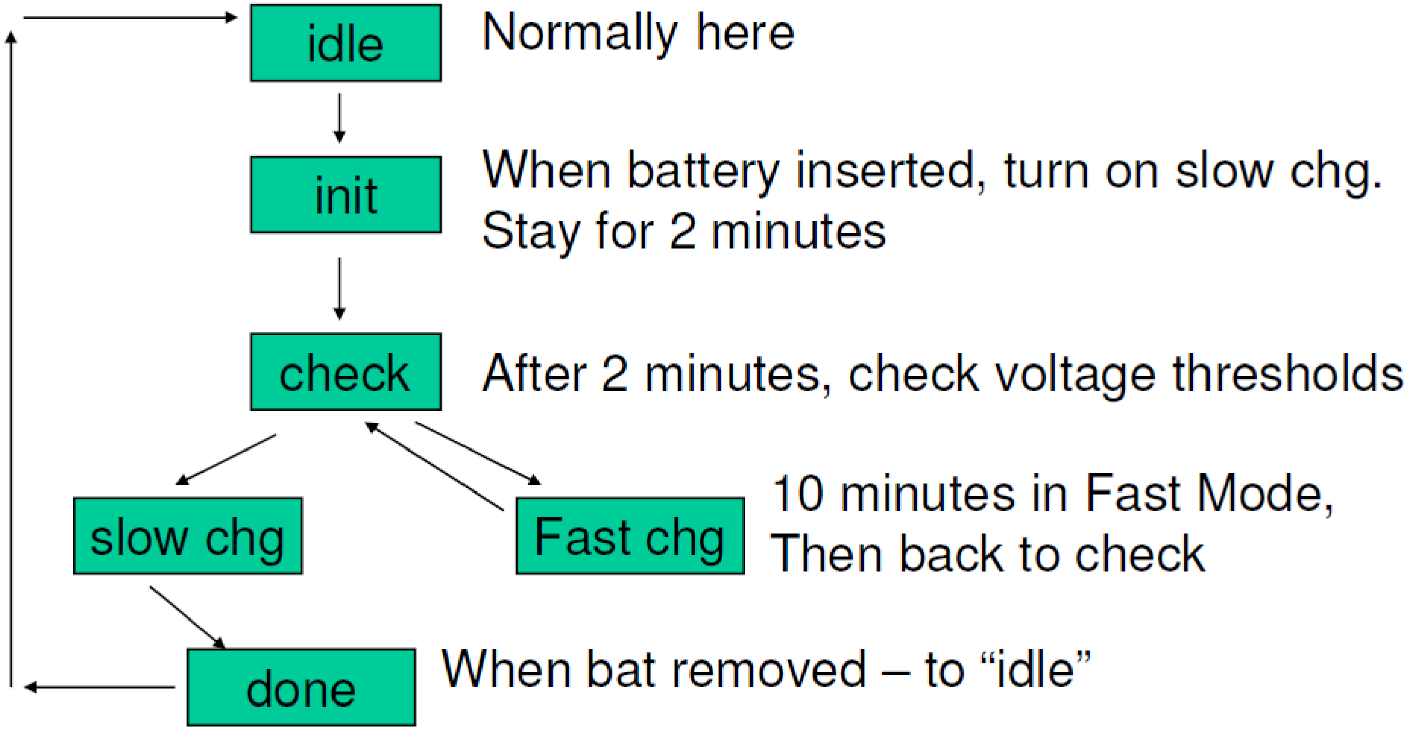
Process 3

```
assign Signal_Sending = (Current_State==Send_Data) ? 1'b1 : 1'b0;
assign Machine_Ready = (Current_State==Idle) ? 1'b1 : 1'b0;
```

COMBINING PROCEDURAL BLOCKS

- Can combine output decoding into always block for exactly same behavior
- Can combine register and state decoding into one procedural block for same behavior (example to come)
- Can combine all into one synchronous always block for slightly modified behavior

SMART BATTERY CHARGER



Normally here

When battery inserted, turn on slow chg. Stay for 2 minutes

After 2 minutes, check voltage thresholds

10 minutes in Fast Mode, Then back to check

When bat removed – to "idle"

COMBINING PROCESSES EXAMPLE

```
always_ff @(CLK, Reset)
  if (Reset==1'b1)
    CurState <= Idle;
  else
    case (CurState)
      Idle:
        if (batt_attached=1'b1)
          CurState <= Init;
      Init:
        if (timer >= two_min)
          CurState <= Check;
      Check:
        if (high_thresh==1'b1)
          CurState <= FastChg;
        else
          CurState <= SlowChg;
      FastChg:
        if (timer >= ten_min)
          CurState <= Check;
      SlowChg:
        if (charge_thresh == 1'b1)
          CurState <= DoneChg;
      DoneChg:
        if (batt_attached==1'b0)
          CurState <= Idle;
    endcase
```

PROCESS AFTER COMBINATION

- Outputs now registered (assigned within clocked always block)
- *Flops made for each output*
    - May be expensive
    - Synchronizes outputs
- Extra flexibility available for setting outputs low and high
- Care should be taken to specify these signals, recognizing when they actually take affect

```
always_ff @(CLK, Reset) begin
  if (Reset==1'b1)
    CurState <= Idle;
  else
    case (CurState)
      Idle:
        if (batt_attached==1'b1)
          CurState <= Init;
          reset_timer <= 1'b1;
      Init:
        if (timer >= two_min)
          CurState <= Check;
          source_low <= 1'b1;
      Check:
  ...
```

STATE ENCODING

```
enum {red, yellow, green, fl_yellow, f_red, turn_arrow}
      Current_State, Next_State;
```

- Sequential states: encodes the states as binary numbers:
  - 000, 001, 010, 011, 100, 101
  - Is this the only way to encode?
- Some encoding options:
  - One-hot encoding
  - Compact encoding
  - Gray encoding
  - Others listed in Xilinx user guides

ONE-HOT ENCODING

- Ensures that individual state register is dedicated to one state
  - Only one flip-flop is active, or hot, at any one time
- Very appropriate with most FPGA targets where a large number of flip-flops are available
- Good alternative when optimizing for speed low power dissipation

```
enum {red, yellow, green, fl_yellow, f_red, turn_arrow}
     Current_State, Next_State;
```

- 100000, 010000, 001000, 000100, 000010, 000001

COMPACT ENCODING

- Option minimizes the number of state variables and flip-flops
  - ➢ Technique is based on hypercube immersion
- Appropriate when trying to optimize for reduced area

```
enum {red, yellow, green, fl_yellow, f_red, turn_arrow}
     Current_State, Next_State;
```

- 000, 001, 010, 011, 100, 101

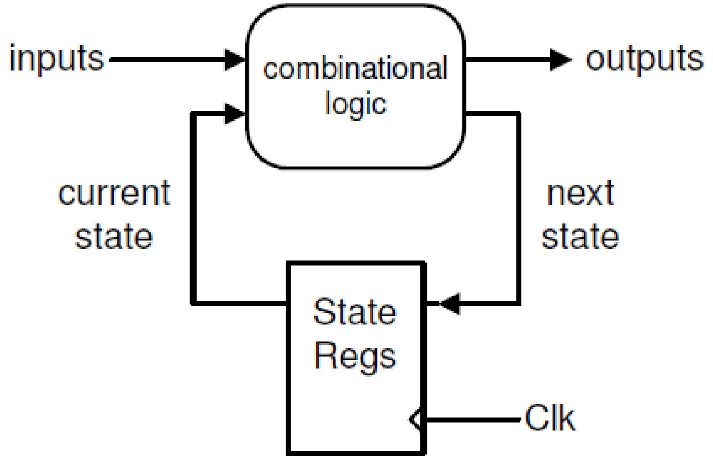
GRAY ENCODING

- Guarantee that only one state variable switches between two consecutive states
- Appropriate for controllers exhibiting long paths without branching
- Minimizes hazards and glitches
- Very good results can be obtained when implementing state register(s) with T or JK flip-flops

- You can explicitly specify your FSM encoding methods for synthesis

WHY STATE ENCODING MATTERS

- Speed: decoding state variable to determine outputs and next state can be simpler via one-hot, for instance
- State transitions: if combinatorial decode of output with no glitches is desired, encoding makes a difference
- Size: how many flops are required?



ILLEGAL STATES

- Given our states: (red, yellow, green, fl_yellow, f_red, turn_arrow)
  - 000, 001, 010, 011, 100, 101
- If encoded as above, two illegal states exist
  - Undefined states in the FSM
- What will logic do when those states are encountered?

```
case (Current_State)
  red:        Next_State <= turn_arrow;
  turn_arrow: Next_State <= green;
  green:      Next_State <= yellow;
  ...
```

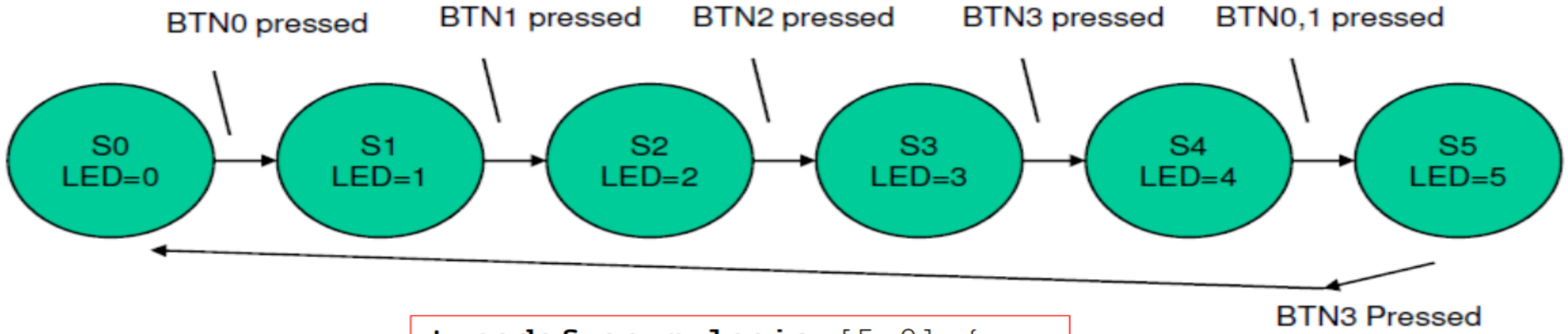
  - It is a mystery...
  - Could present security concerns: trigger Hardware Trojans

MORE ON ILLEGAL STATES

- Consequences of entering illegal state unknown
  - Frequently results in “wedged” machine, doesn't recover
  - System can get stuck once entering illegal state
- Faulty reset circuitry (or none) could lead to power-up into illegal state
  - The initial state would be random!
- Single event upsets in radiation environments can cause a flop to toggle, leaving machine in an illegal state
- Synchronization errors on inputs
  - Setup/hold violations
  - Metastability

MORE ON ILLEGAL STATES

- Suppose we use one-hot encoding for the following:

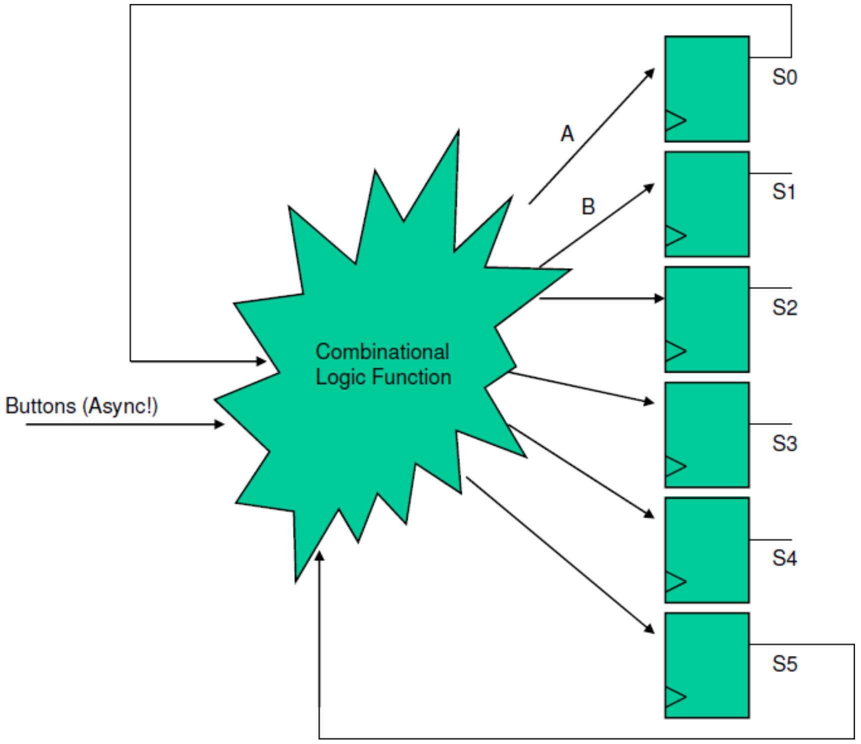


```
typedef enum logic [5:0] {
  s0=6'b100000,
  s1=6'b010000,
  s2=6'b001000,
  s3=6'b000100,
  s4=6'b000010,
  s5=6'b000001} statetype;
```

FSM IMPLEMENTATION: ONE-HOT

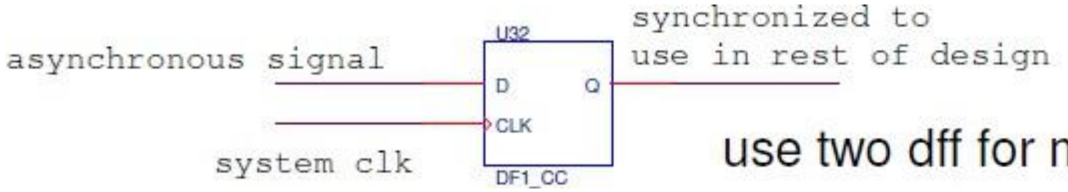
Imagine:

- Current_State = “100000”
- Button pressed near clock edge
- Line A to go from 1 – 0
- Line B to go from 0 – 1
- Possible outcome:
  - S0 → 0
  - S1 stays 0
- What now???



DEALING WITH ILLEGAL STATES

- Problems not specific to one-hot encoding – they are simply magnified by the scheme as there are more illegal states
- Solutions:
  - ➢ Always have a reset state – somewhat obvious
  - ➢ Carefully synchronize all inputs



use two dff for metastability issues

- If design in inaccessible place and cannot be restarted, etc.
  - ➢ Make a "safe" state machine...

SAFE STATE MACHINES

- The “default” clause is typically not implemented by FSM extractor
    - There are no others, everything in enumerated type is covered
    - Might call this the result of “reachability” analysis
    - Likely up to you to generate reset logic which places machine in a known state
- Some synthesis tools provide attributes for encoding machine that include: safe, one-hot; safe, grey … etc.

SUMMARY

- More issues in FSM design and implementation
  - ➢ Combining processes in FSM design
  - ➢ FSM implementation
- State encoding
  - ➢ One-hot encoding
  - ➢ Compact encoding
  - ➢ Gray encoding
- Dealing with illegal states
  - ➢ Safe state machines

Thank you!