PUFacademy

A PUFsecurity Alliance

Digital Logic Design
- Lecture 6
- Design Guideline

2025 Spring

PUFacademy

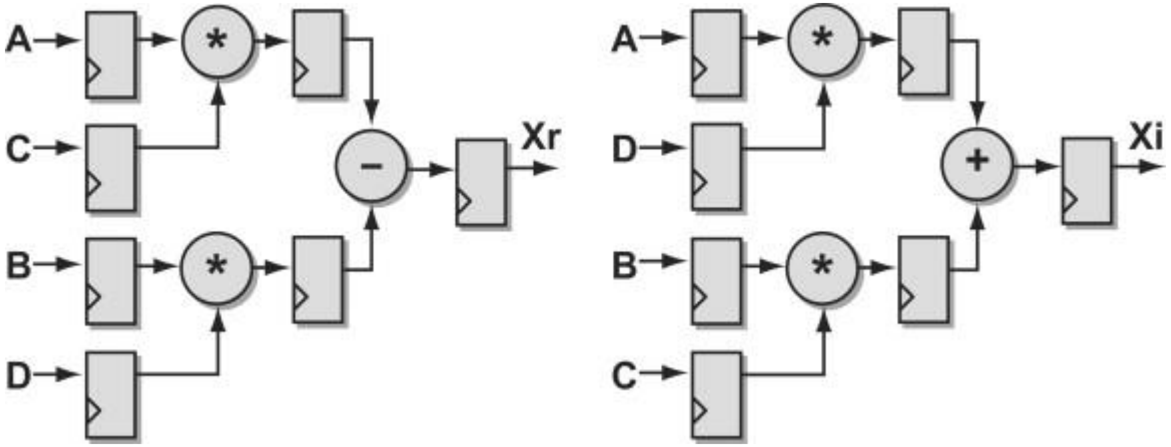
A PUFsecurity Alliance

Agenda

1. From Code to Chip
2. Architectural Planning
3. Module Planning
4. Finite State Machine
5. Coding Guideline

What is RTL?

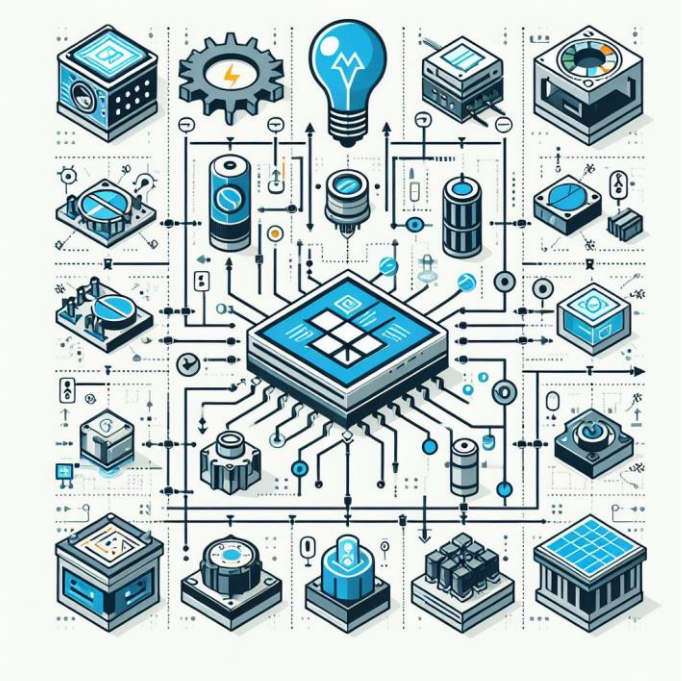
- Register-transfer level (RTL) is a design abstraction
  which models a synchronous digital circuit
  in terms of the flow of digital signals between **registers** (flip-flops).
- Ex: The product of two complex numbers
  - (A+Bi) * (C+Di) = $X_r + X_i$ i



  - https://www.sciencedirect.com/topics/computer-science/register-transfer-level

RTL Design Life Cycle

- Top-down planning (by paper-pencil) (80%)
  - Top-module macro architecture
    - sub-module block diagram
  - IO and sub-module interconnection
    - Handshake protocol
  - Sub-modules micro architecture
    - State machine
    - Register
    - Logic

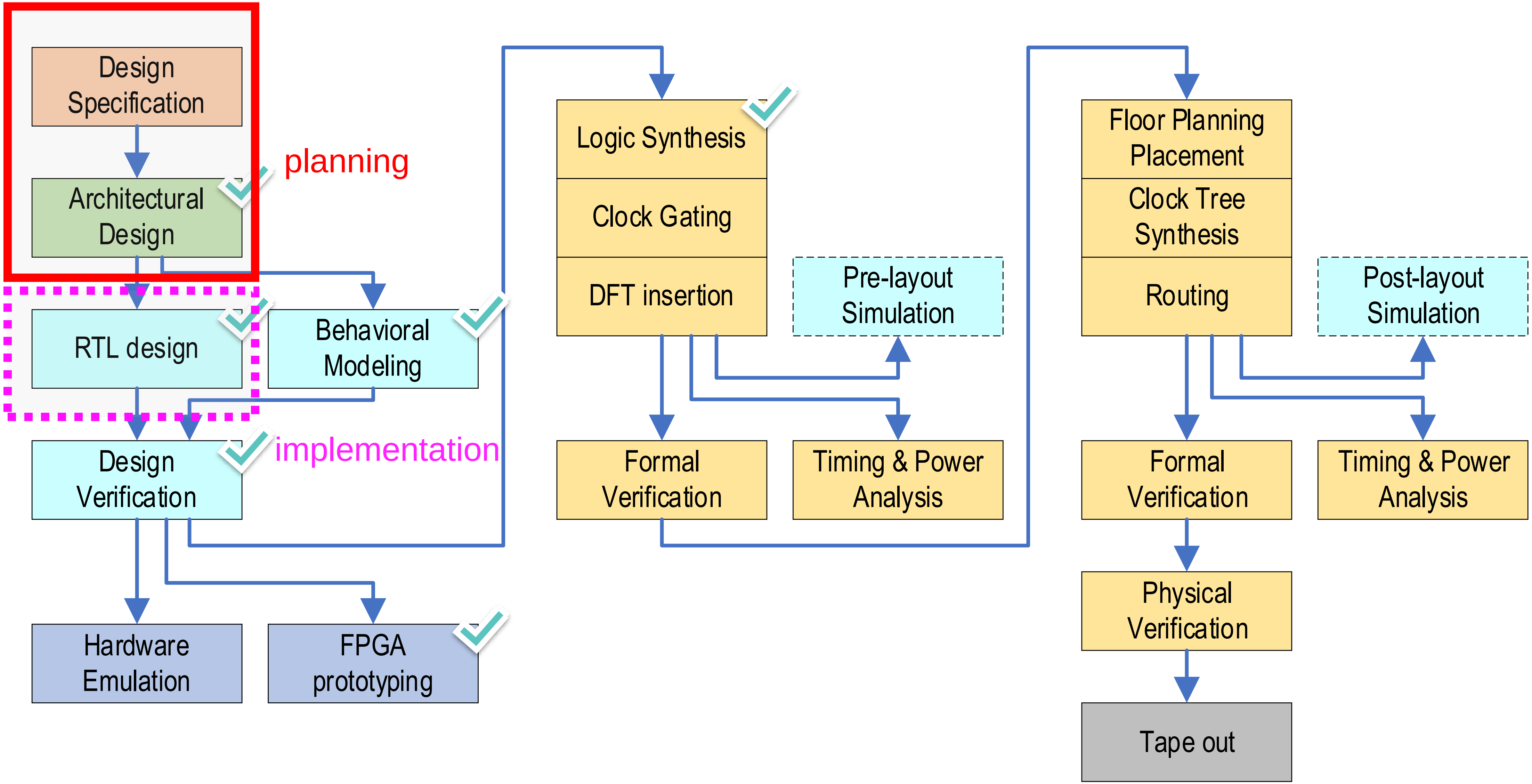


- Bottom-up implementation (write code) (20%)
  - Sub-module (with simple testbench)
  - Top-module (with complex testbench)

design:testbench=1:3

- Planning and verification take up a large part of the design cycle.

From Code to Chip

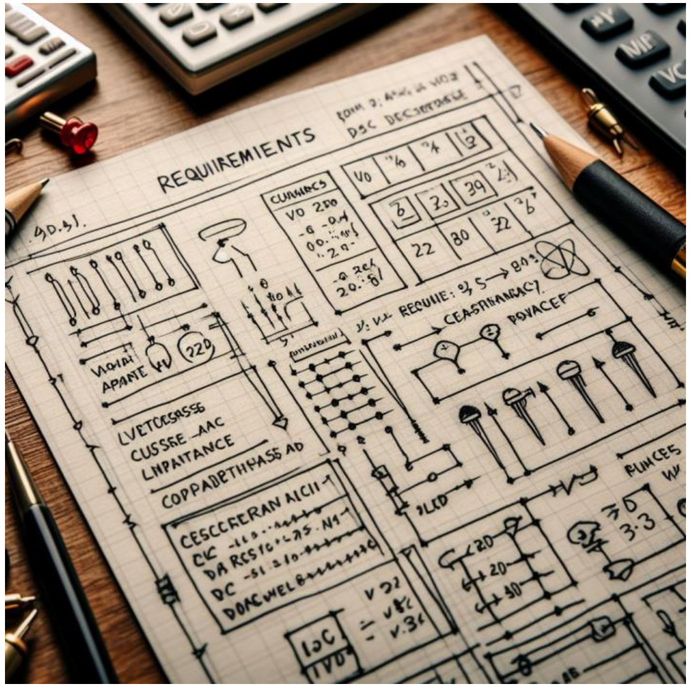


Agenda

1. From Code to Chip
2. Architectural Planning
3. Module Planning
4. Finite State Machine
5. Coding Guideline

Design Specification

- ***Requirements***
  - Purpose
    - Algorithm
    - Performance
  - Hardware specification
    - System platform
      - IO protocol connect to the system
    - Technology
      - Timing constraint, clock rate
    - Performance vs Area
      - Latency (time)
      - Throughput (data size/time)

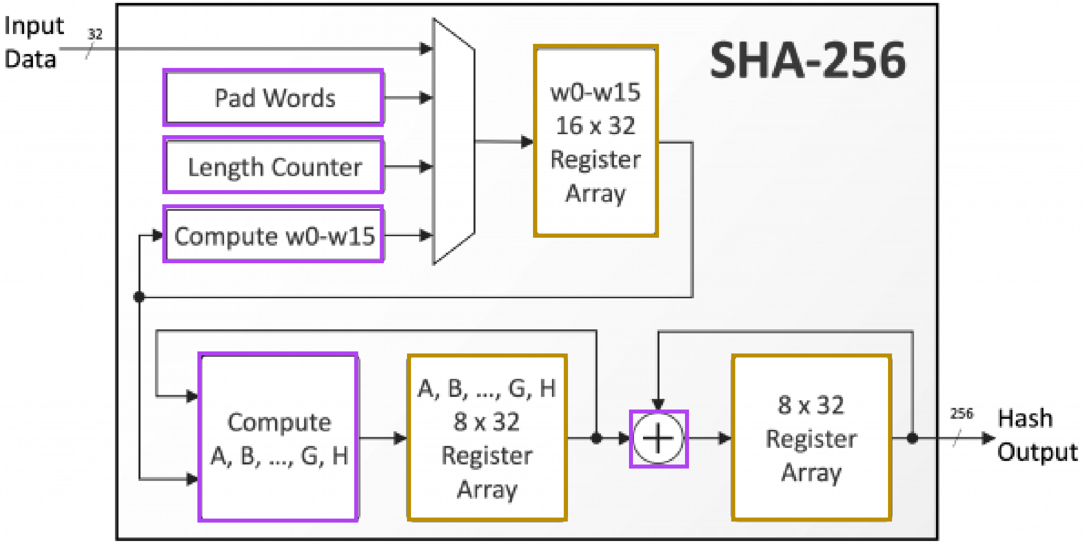


Architectural Planning (1/3)

- ***Profiling*** *– specification to architecture*
  - Algorithm (by paper-pencil)
    - What kinds of operations and storages are needed
      - Operation → logics
      - Storage → registers, SRAM
    - Is operation too complicated?
      - Pipe-line
        - » break-down timing path
      - Mathematical transformation
        - » ex: division to many subtractions

Architectual Planning (2/3)

- ***Profiling** – specification to architecture*
  - Design partition (by paper-pencil)
    - Decide the number of operation and storage.
      - Shared operation → low area
      - Multiple sets → high performance
    - Planning sub-modules
      - Control blocks
      - Operation blocks
      - Storage blocks
      - Interconnections

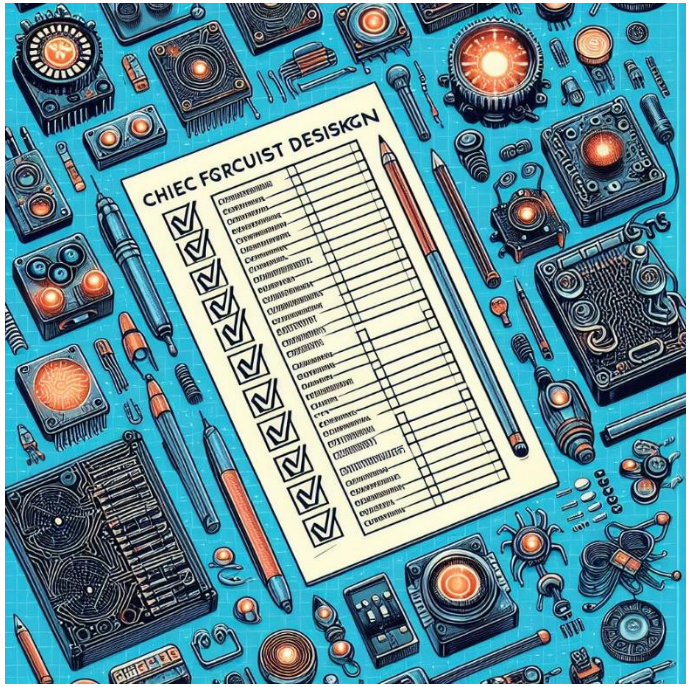


https://www.cast-inc.com/security/encryption-primitives/sha-256

- ***Now you should have a block diagram of the design !!***

Architectural Planning (3/3)

- ***Profiling*** *– specification to architecture*
  - – Constraint (by experience or EDA tool)
    - Critical path vs clock rate in this technology
    - Write a circuit to test synthesis
  - – Performance vs Area (by paper-pencil)
    - Calculate the latency
    - Calculate the throughput



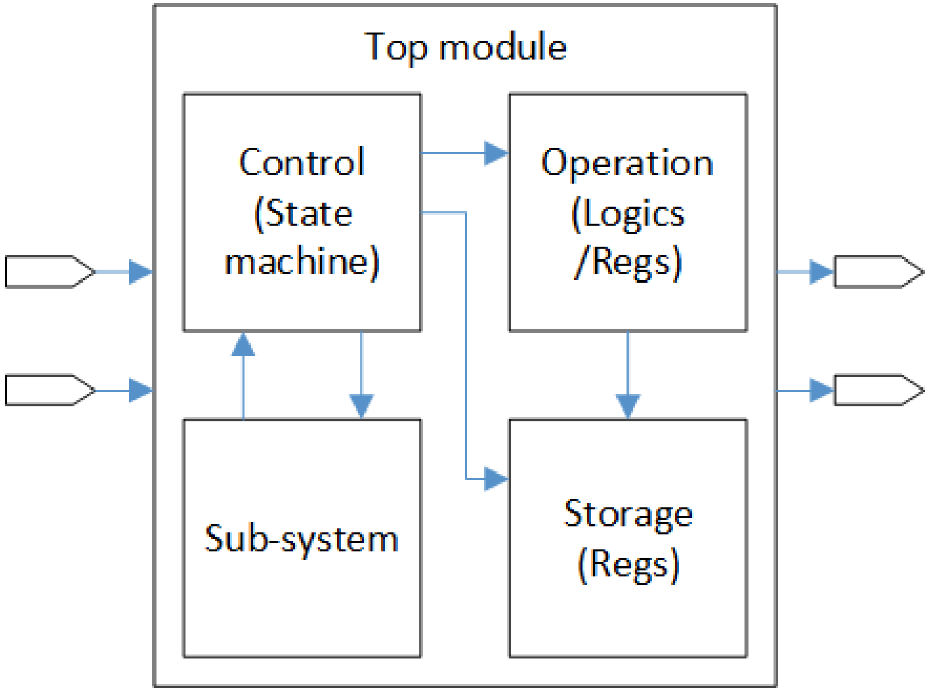
- ***If it does not meet the requirements, the plan must be revised !!***

Agenda

1. From Code to Chip
2. Architectural Planning
3. Module Planning
4. Finite State Machine
5. Coding Guideline

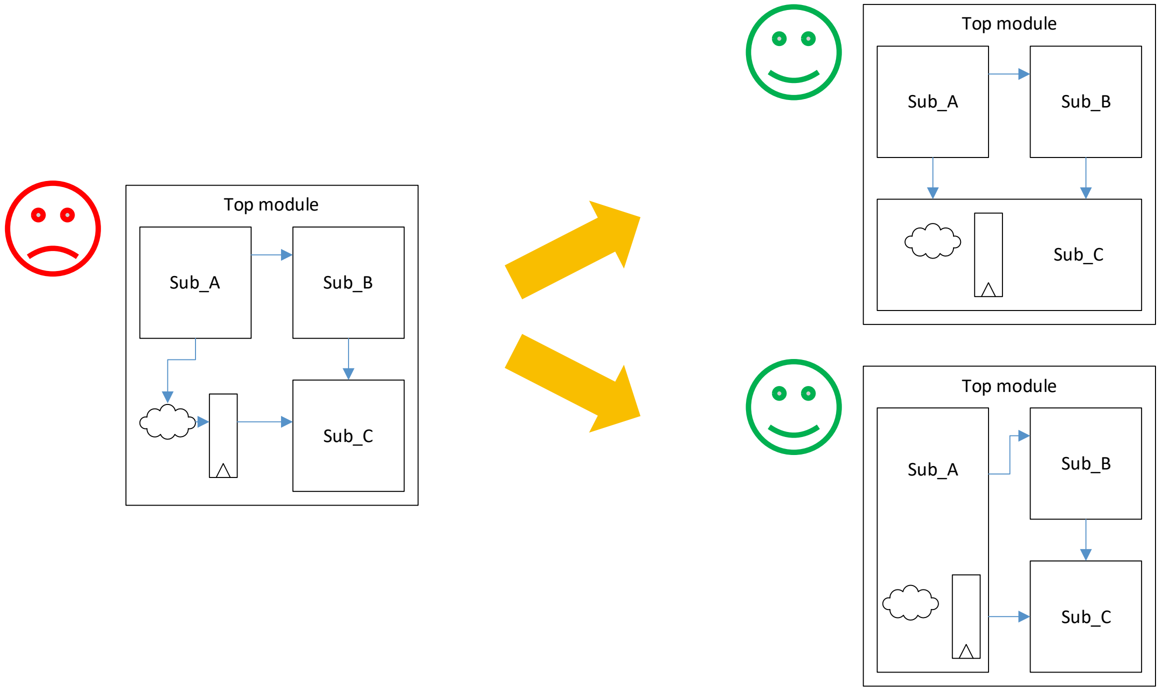
Module Planning

- ***Type of sub-modules** – according to the purpose*
  - Control module
    - State machine for the top module
    - Control all other sub-modules
  - Data operation module
    - Complex logics (w/ or w/o data register)
    - For reuse or multiple instance
  - Pure Storage
    - Register, FIFO or SRAM
    - Without other logic operation
  - Sub-system
    - It has own state machine and operations
    - Reducing the complexity of the main state machine



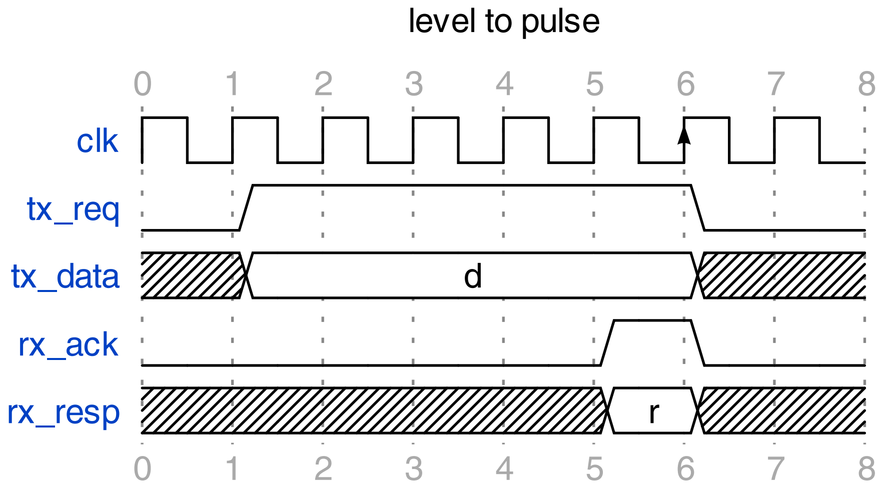
Design Partition

- Keep connections between sub-modules as simple as possible.
  - don’t write complex circuit outside the sub-modules
  - If there are too many circuits in the top-level module, you should try to categorize them into sub-modules.



Input And Output Design

- ***Interconnection** – planning the IO of sub-module*
  - Control signal
    - Level
      - Suitable for indicating status
      - Holds the value until the status is gone.
    - Pulse
      - Suitable for immediate event
      - Only exists for one clock cycle
  - Data signal
    - Bus width
      - Choose the suitable width for each purpose

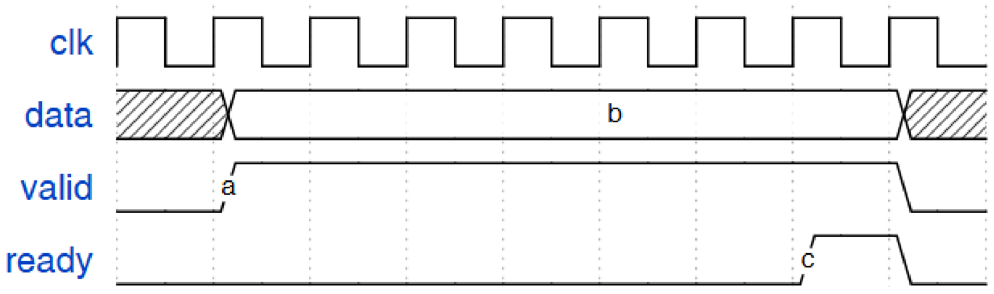


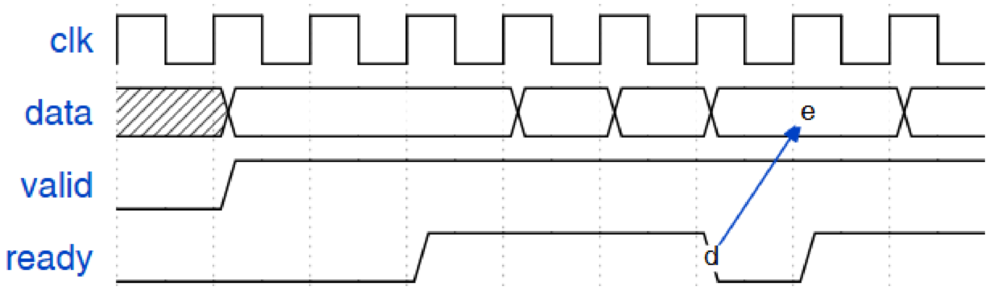
- protocol can be 1-way (without response) or 2-way

Valid-Ready Handshake

- When the Sender has data to transmit, it pulls the valid signal high (a) and keeps it stable (b) until the Receiver pulls the ready signal high (c)

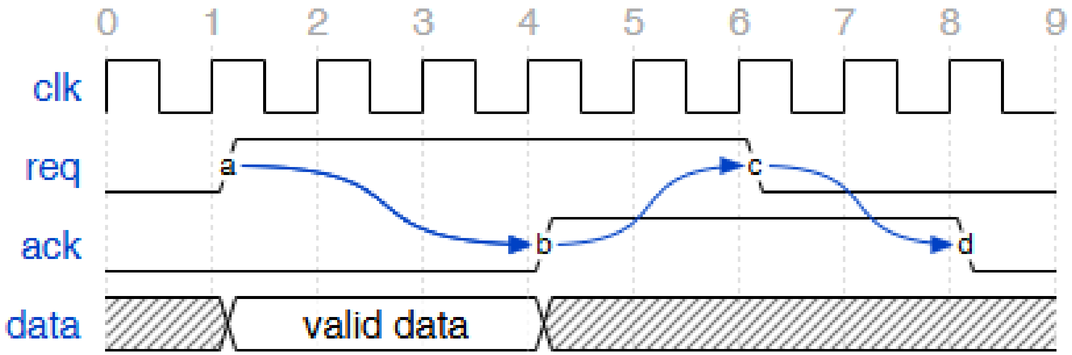
- When the Receiver pulls the rea dy signal low (d), the Sender kee ps data stable again (e)





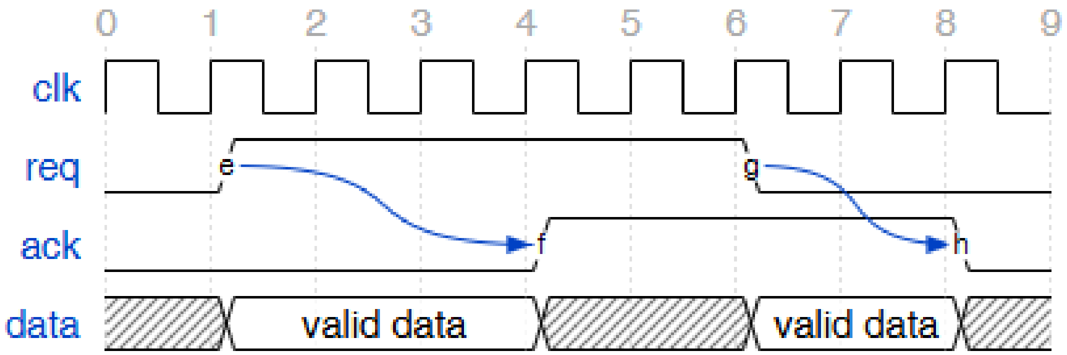
- For the Sender, it cannot decide whether to raise the valid level based on the ready level
- For the Receiver, it cannot decide whether to raise the ready level based on the valid level

4-Phase Handshake



- Phase 1: Cycle 0
  - The Sender and Receiver are both 0
- Phase 2: Cycle 1~3
  - The Sender pulls up req (a) while keeping the data, waiting for the Receiver’s ack to be high
- Phase 3: Cycle 4~5
  - The Receiver pulls up ack (b) indicating that the Receiver has received the data
- Phase 4: Cycle 6~7
  - The Sender sees the ack and pulls down the req (c)
  - The Receiver sees the req is pulled low, so it also pulls down ack (d)

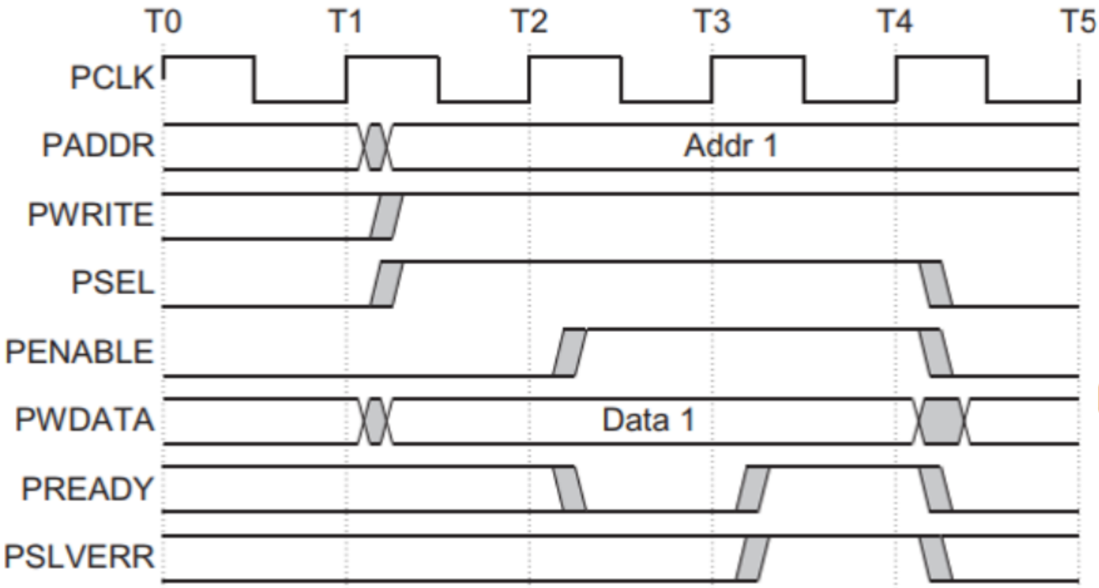
2-Phase Handshake

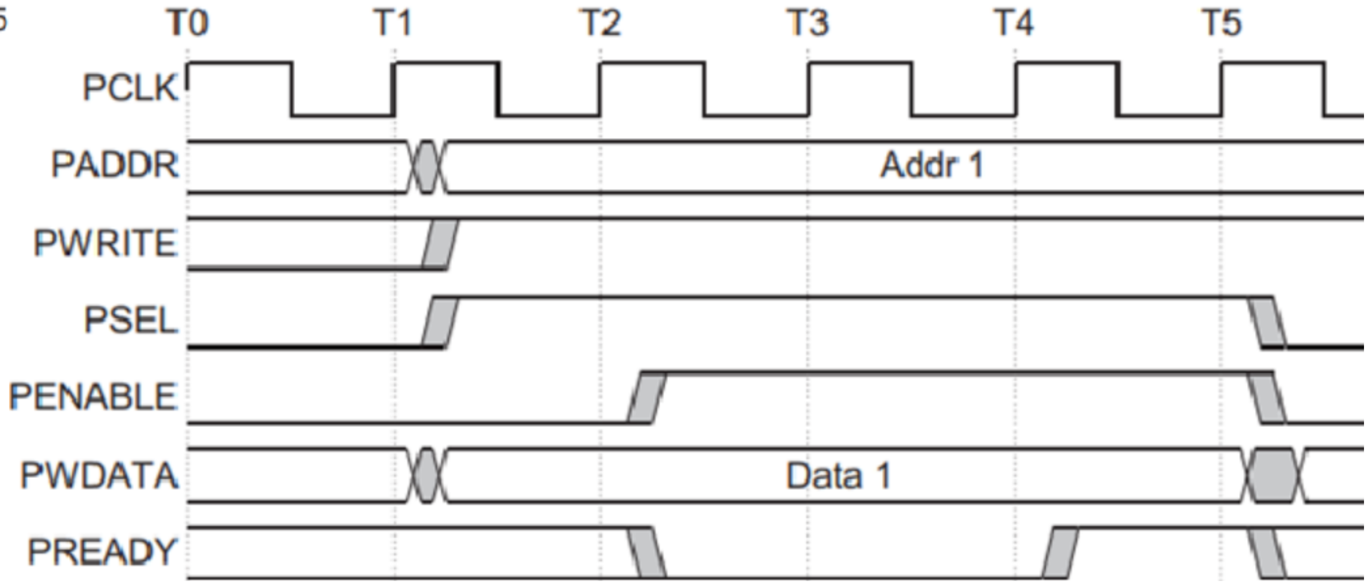


- Phase 1: Cycle 0
  - The Sender and Receiver are both 0
- Phase 2: Cycle 1~3
  - The Sender pulls up req (e) while keeping the data, waiting for the Receiver’s ack to be high
  - The Receiver pulls up ack (f) indicating that the Receiver has received the data
  - The Sender and Receiver are both 1, as phase 1

APB Protocol

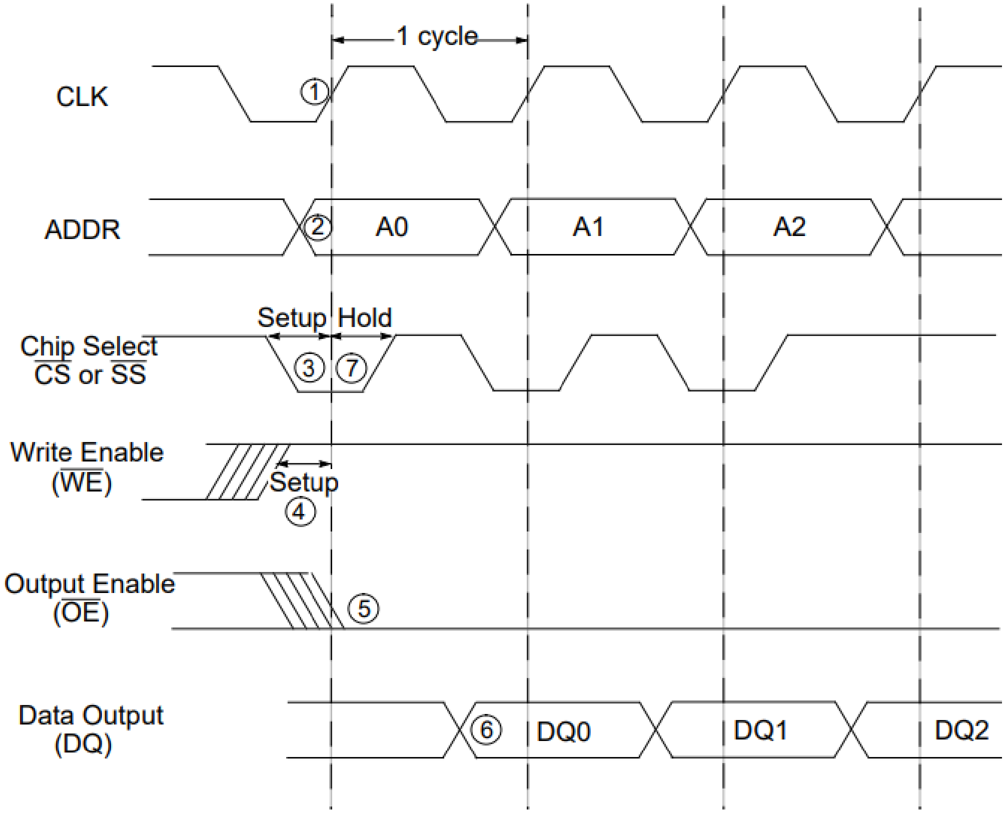
- ARM AMBA APB(Advanced Peripheral Bus) protocol specification
- Read transfer with wait status
- Write transfer with wait status



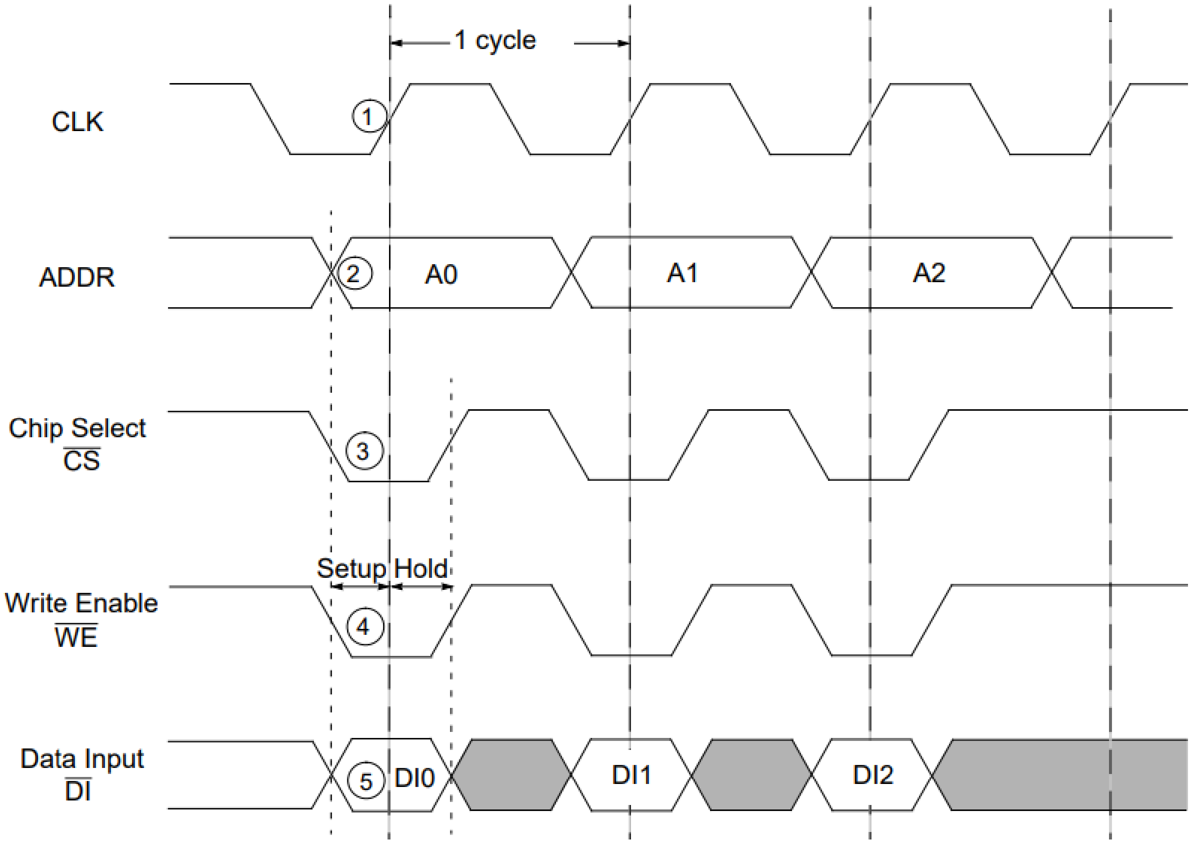


SRAM Protocol

- IBM Static RAM operation
- Read from SRAM



- Write from SRAM



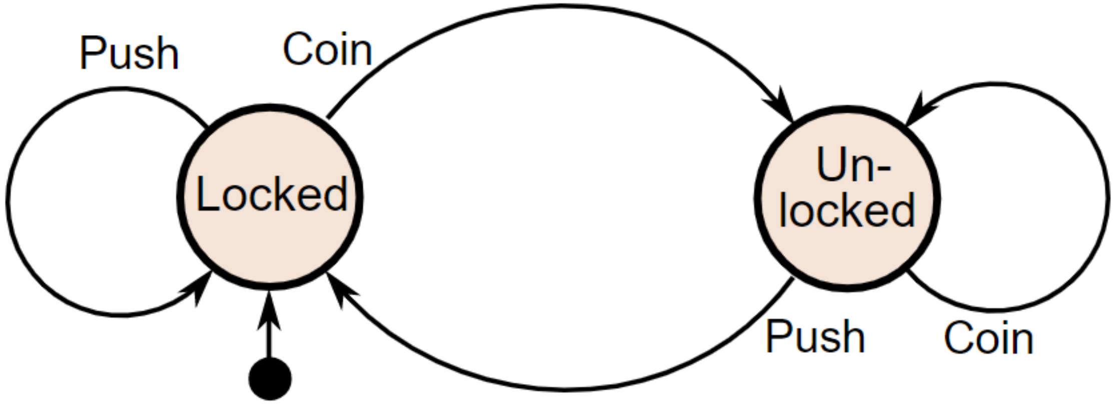
Agenda

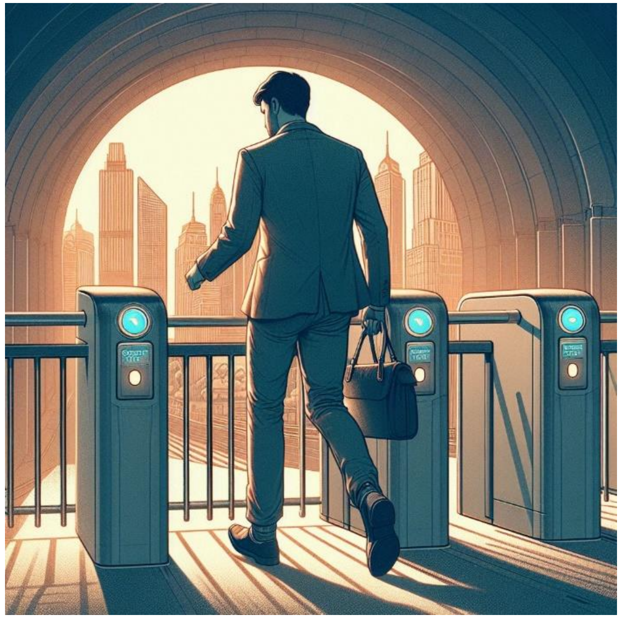
1. From Code to Chip
2. Architectural Planning
3. Module Planning
4. Finite State Machine
5. Coding Guideline

What Is A Finite State Machine?

- Finite state machine is a model of computation
  - It is not a physical circuit; it is a system composed of sequential circuits
  - It can be in **exactly one** of a finite number of states at any given time
  - The FSM can change from one state to another in response to some inputs

- Example: coin-operated turnstile
  - Two states: **Locked** and **Unlocked**





Why We Use Finite State Machine?

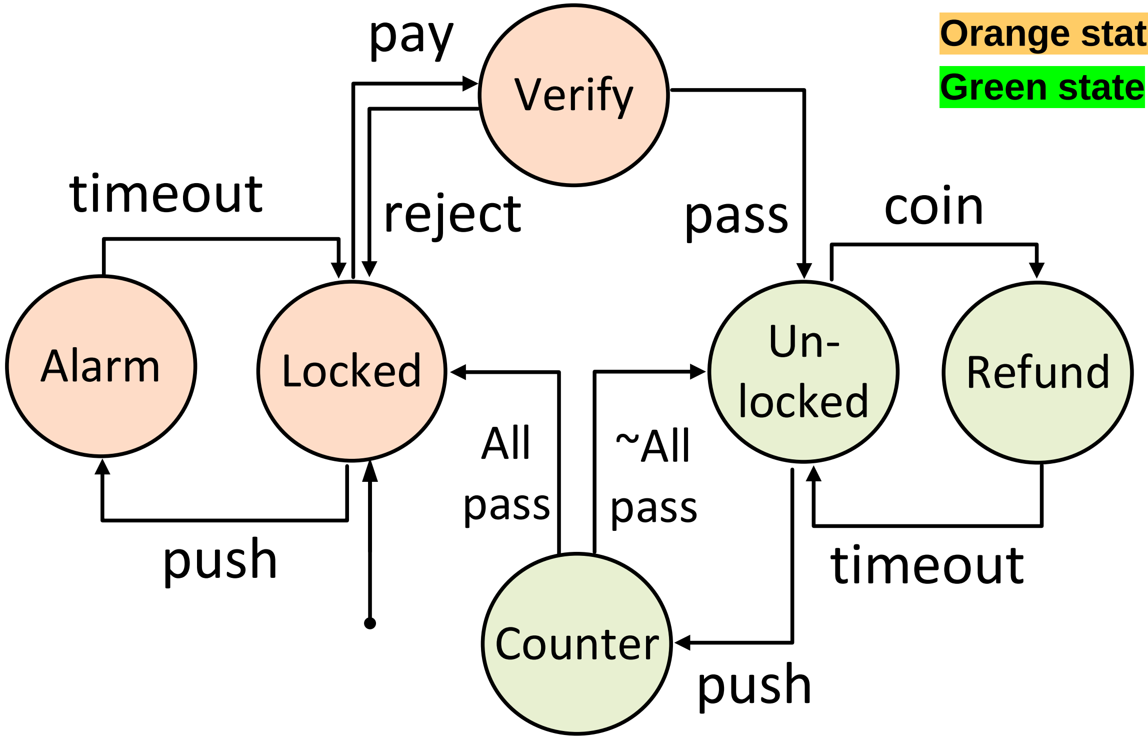
- A state machine includes **current state**, **inputs**, **next state**, and **output**

Current State	Input	Next State	Output
Locked	coin	unlocked	unlocks the turnstile so that the customer can push through
	push	locked	none
Unlocked	coin	unlocked	none
	push	locked	When the customer has pushed through, locks the turnstile

- Consider more situations?
  - Refund if someone inserts coins while unlocked
  - The alarm sounds if someone pushes the locked turnstile
  - Someone use fake coins
  - Group ticket
  - …

A Finite State Machine

- If you use the FSM everything will be clear!!



Orange state: Lock the gate

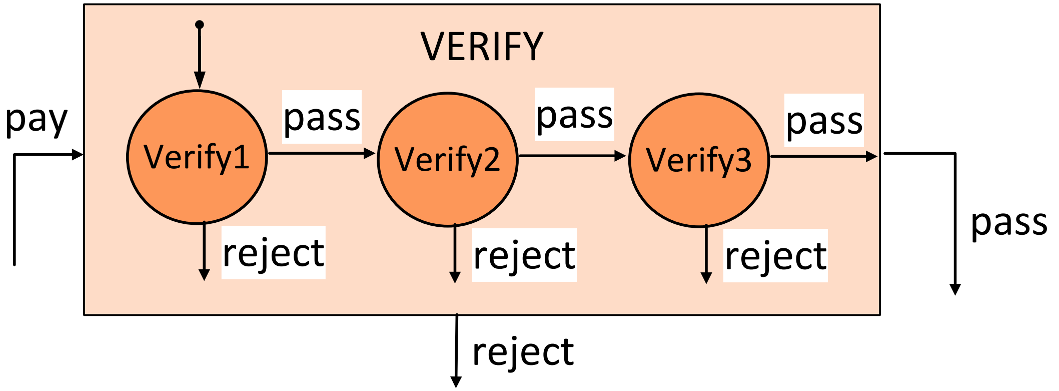
Green state: Unlock the gate

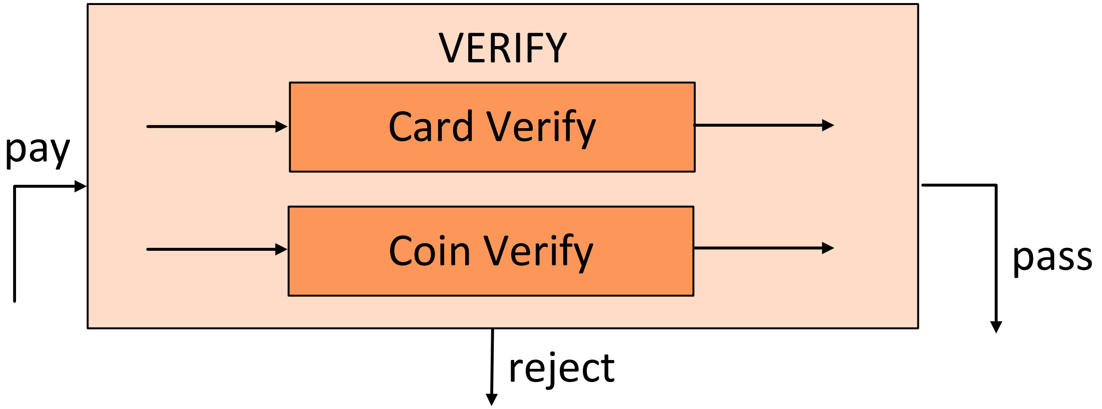
Extended question:

If you need a mode to always keep the turnstile open during maintenance, how would you do?

A Finite State Machine

- FSM can be used to implement, extend, and debug a **complex decision-making** algorithms
- Use an FSM in your circuit if possible
- An FSM contains an appropriate number of states
- Sometimes we may use **Hierarchical (Nested) State Machine** or **Parallel State Machines**

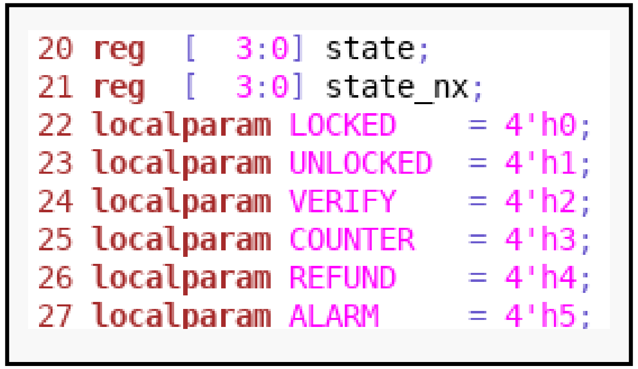




Declare States

- Three steps of implement an FSM
  1. Declare states
  2. Decide next states of every possible inputs
  3. Decide outputs

- Declare states
  – Short and easy-to-understand
  – Pick an appropriate encode format

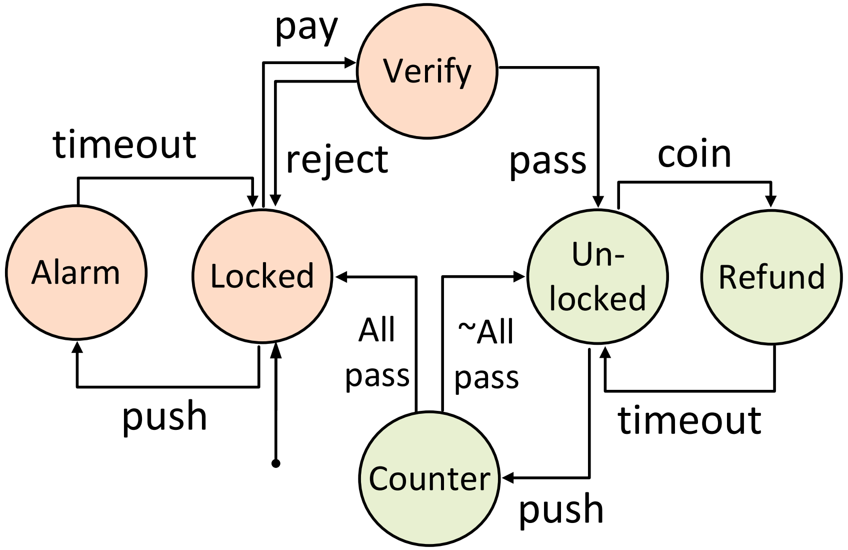


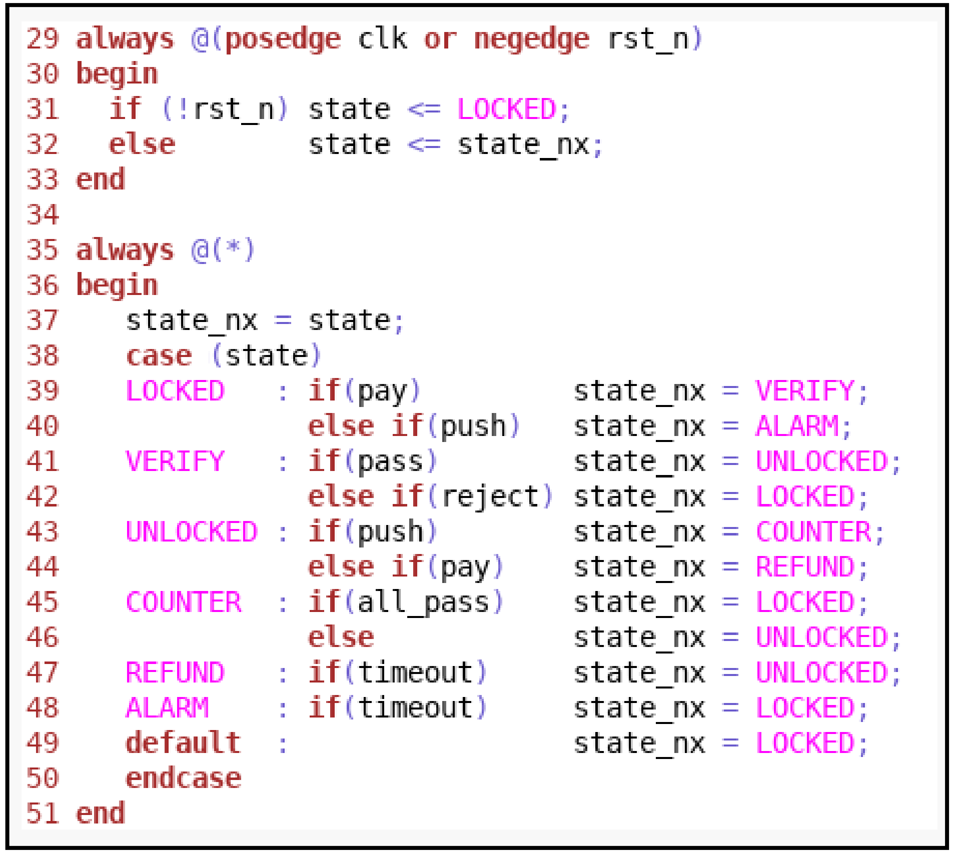
```verilog
reg   [  3:0] state;
reg   [  3:0] state_nx;
localparam LOCKED     = 4'h0;
localparam UNLOCKED   = 4'h1;
localparam VERIFY     = 4'h2;
localparam COUNTER    = 4'h3;
localparam REFUND     = 4'h4;
localparam ALARM      = 4'h5;
```

State	Binary	Gray	Johnson	One-hot
0	0000	0000	00000000	0000000000000001
1	0001	0001	00000001	0000000000000010
2	0010	0011	00000011	0000000000000100
3	0011	0010	00000111	0000000000001000
4	0100	0110	00001111	0000000000010000
5	0101	0111	00011111	0000000000100000
6	0110	0101	00111111	0000000001000000
7	0111	0100	01111111	0000000010000000
8	1000	1100	11111111	0000000100000000
9	1001	1101	11111110	0000001000000000
10	1010	1111	11111100	0000010000000000
…	…	…	…	…
15	1111	1000	10000000	1000000000000000

Decide Next States

- Decide next states of every possible inputs
  - Using **TWO** always blocks
  - Decide initial states
  - To think through all inputs completely
  - Choose a next state for undeclared states





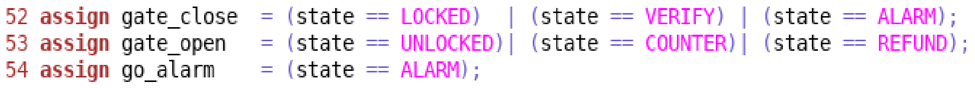
```
always @(posedge clk or negedge rst_n)
begin
  if (!rst_n) state <= LOCKED;
  else        state <= state_nx;
end

always @(*)
begin
   state_nx = state;
   case (state)
   LOCKED   : if(pay)         state_nx = VERIFY;
              else if(push)   state_nx = ALARM;
   VERIFY   : if(pass)        state_nx = UNLOCKED;
              else if(reject) state_nx = LOCKED;
   UNLOCKED : if(push)        state_nx = COUNTER;
              else if(pay)    state_nx = REFUND;
   COUNTER  : if(all_pass)    state_nx = LOCKED;
              else            state_nx = UNLOCKED;
   REFUND   : if(timeout)     state_nx = UNLOCKED;
   ALARM    : if(timeout)     state_nx = LOCKED;
   default  :                 state_nx = LOCKED;
   endcase
end
```

Decide Outputs

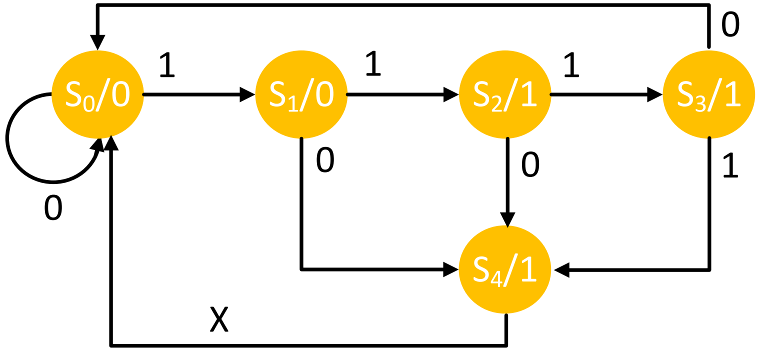
- Two types of output logic: **Moore Machine** and **Mealy Machine**

Moore Machine

- The output depends on the **current state** only
- May have **more states** than Mealy machine
- React **one cycle later** than Mealy machine

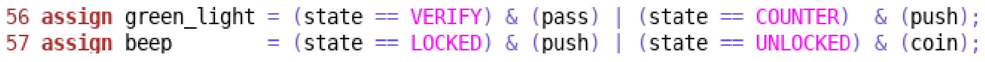


```
assign gate_close  = (state == LOCKED)  | (state == VERIFY) | (state == ALARM);
assign gate_open   = (state == UNLOCKED)| (state == COUNTER)| (state == REFUND);
assign go_alarm    = (state == ALARM);
```

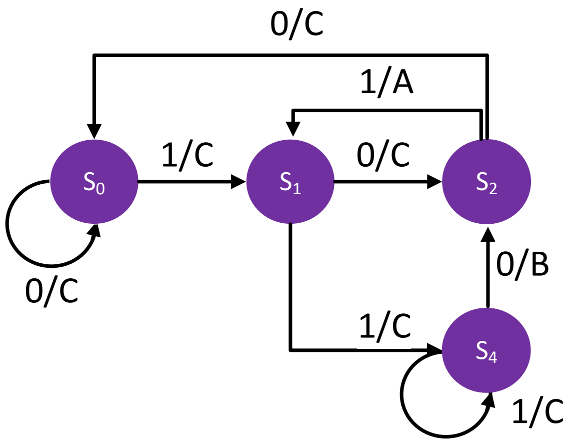


Mealy Machine

- The output follows **the inputs** and **the current state**
- May have **less states** than Moore machine
- React **one cycle earlier** than Moore machine



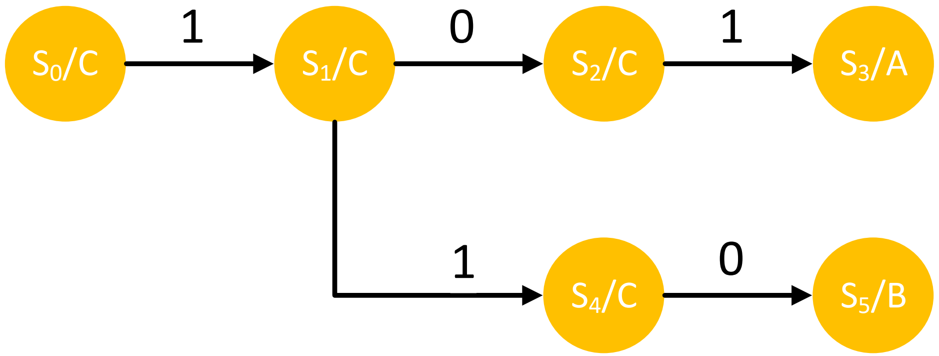
```
assign green_light = (state == VERIFY) & (pass) | (state == COUNTER)  & (push);
assign beep        = (state == LOCKED) & (push) | (state == UNLOCKED) & (coin);
```



Moore Machine

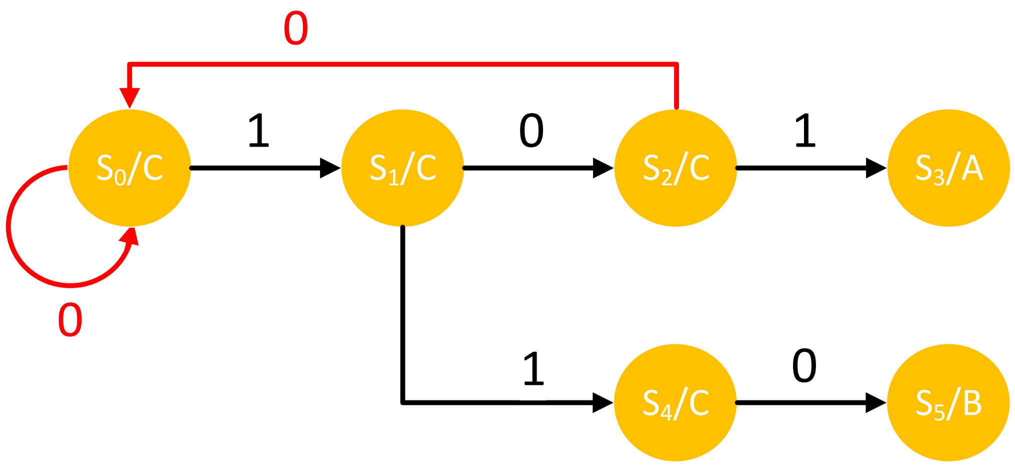
- Design a state machine for reading a binary input sequence
  - If it has a substring “101”, it outputs A
  - If the input has substring “110”, it outputs B
  - Otherwise, it outputs C.

- Implement it by Moore machine
  - Define states and main arcs

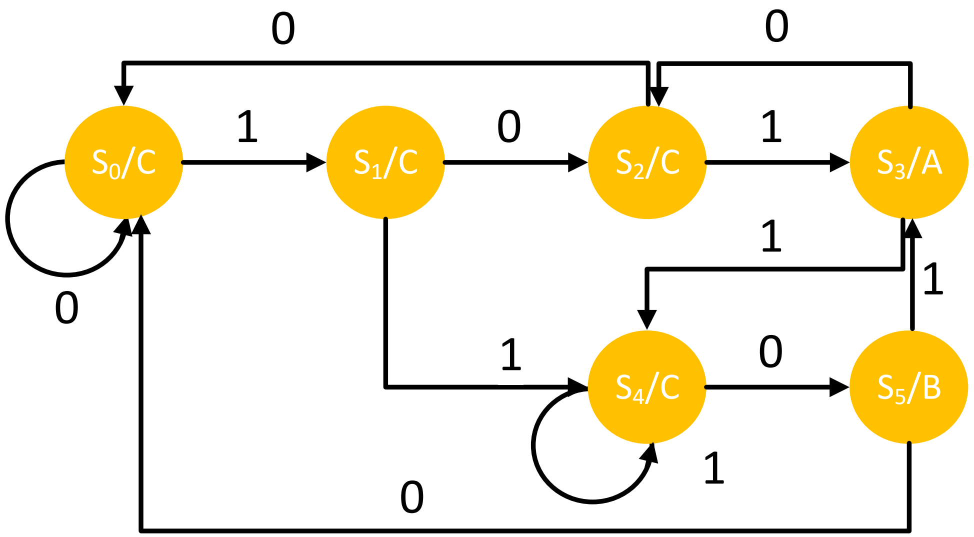


Moore Machine

- Implement it by Moore machine
  - Fill in other arcs and make sure every state has both “0” and “1” arcs

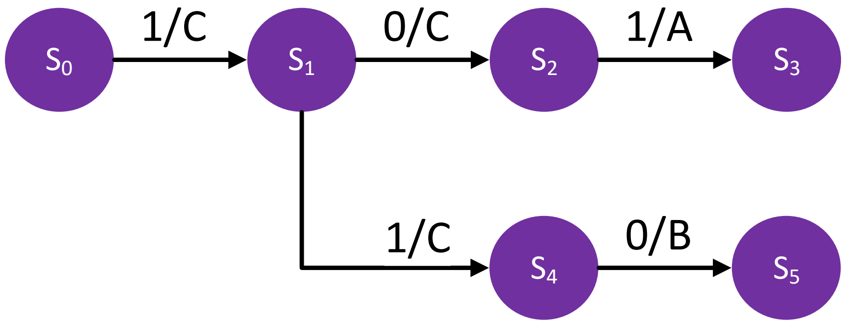


- Implement it by Moore machine
  - Check if there is no redundant state

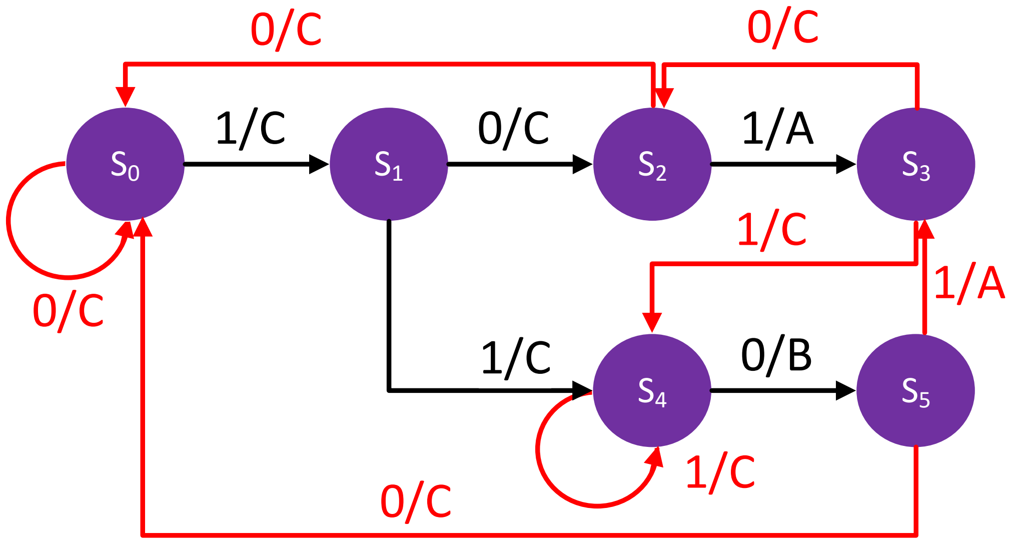


Mealy Machine

- Implement it by Mealy machine
  - Define states and main arcs

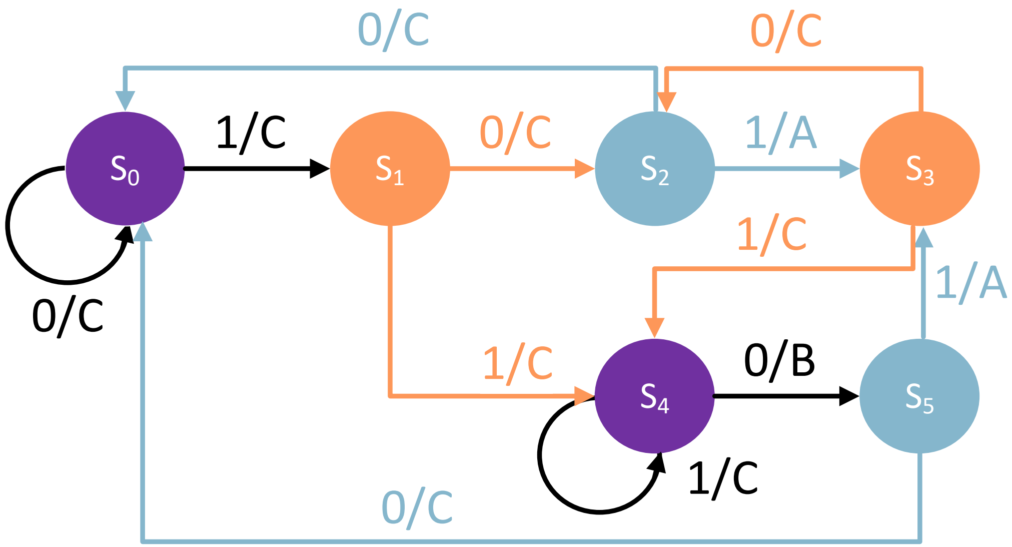


- Implement it by Mealy machine
  - Fill in other arcs and make sure every state has both “0” and “1” arcs

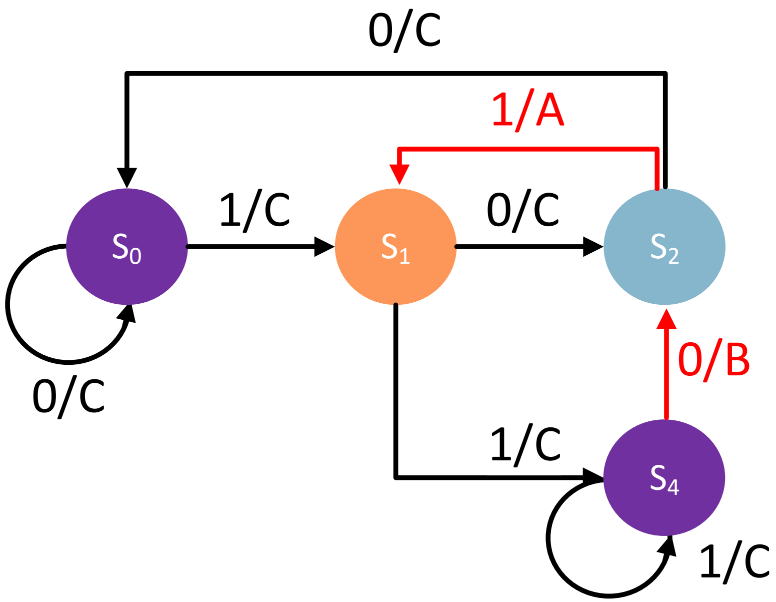


Mealy Machine

- Implement it by Mealy machine
  - Check if there is no redundant state

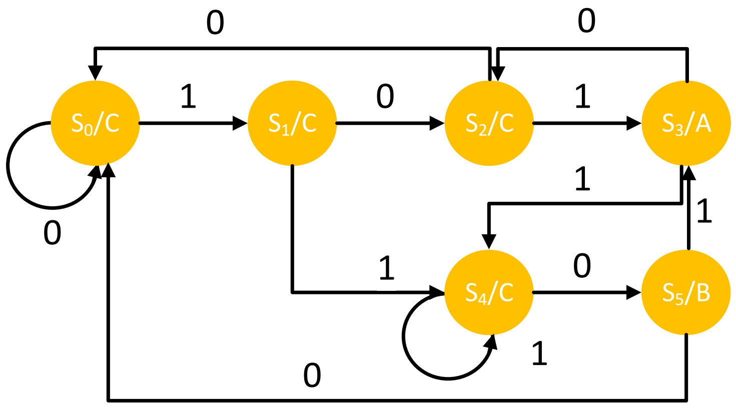


- Implement it by Mealy machine

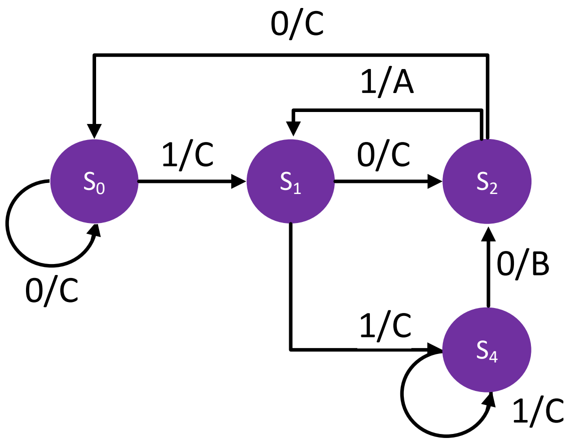


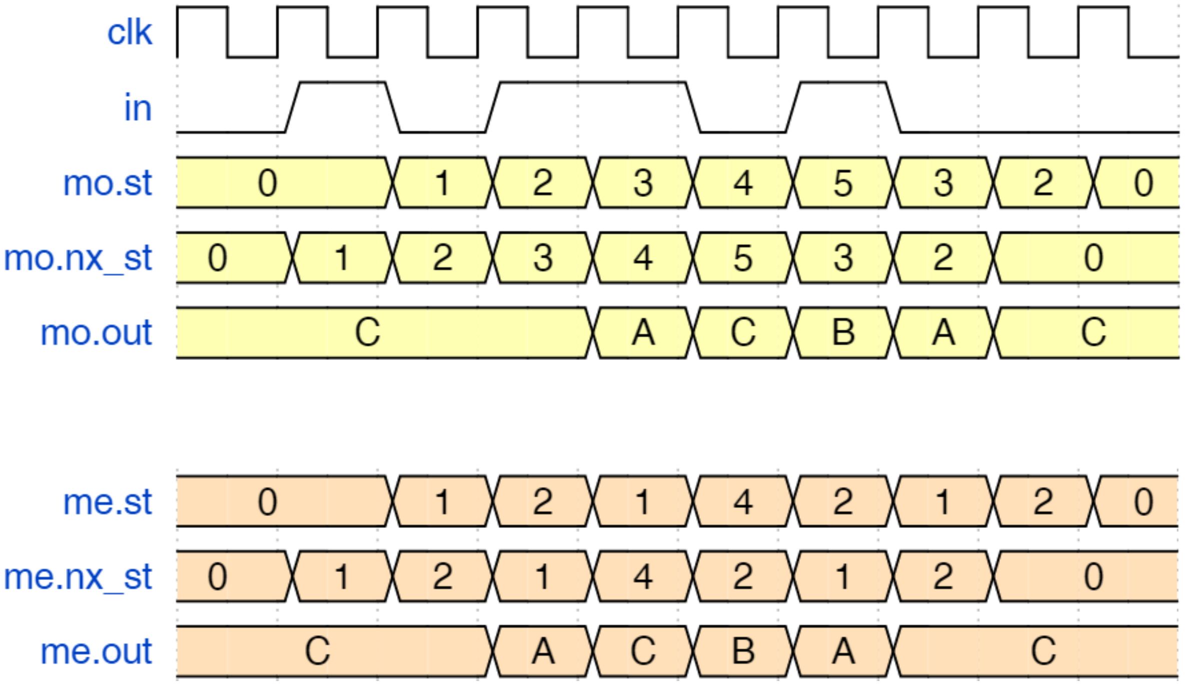
Waveform

- Moore Machine

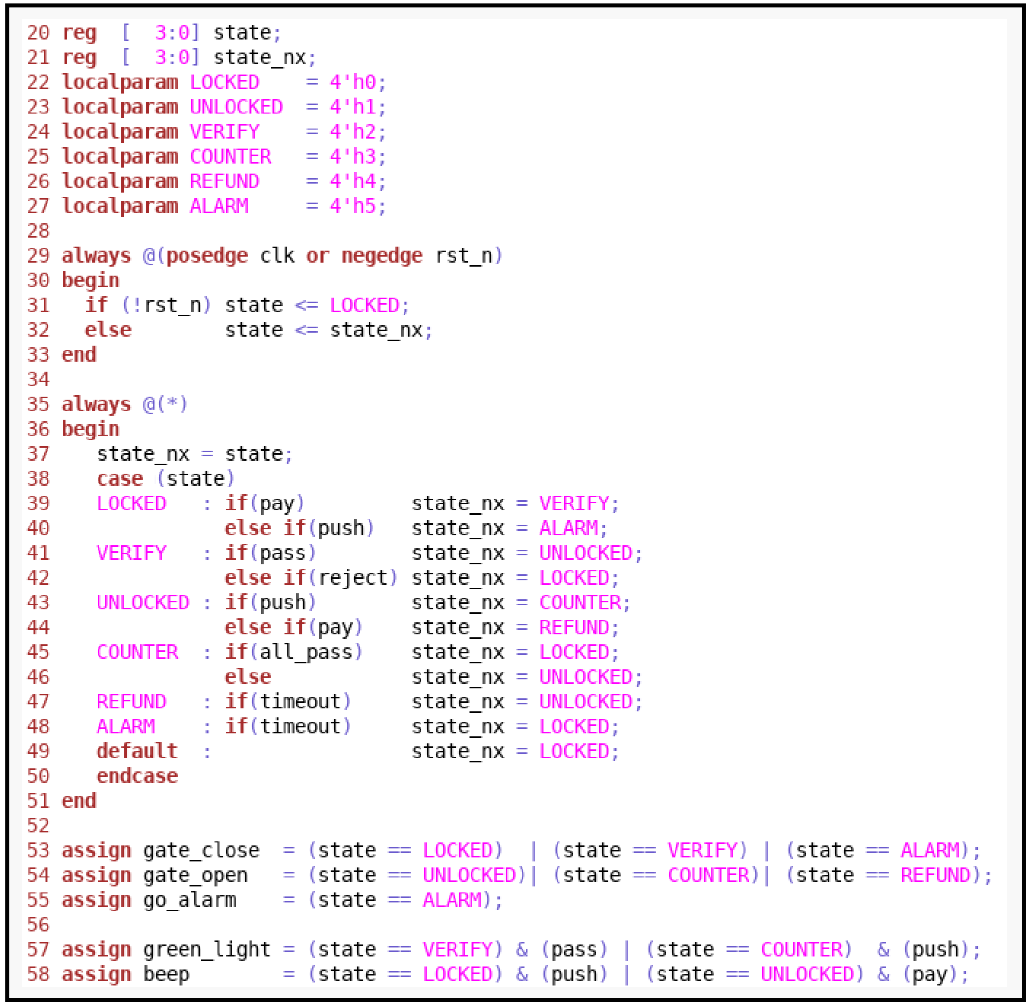


- Mealy Machine





Quick Tips



```
reg  [  3:0] state;
reg  [  3:0] state_nx;
localparam LOCKED     = 4'h0;
localparam UNLOCKED   = 4'h1;
localparam VERIFY     = 4'h2;
localparam COUNTER    = 4'h3;
localparam REFUND     = 4'h4;
localparam ALARM      = 4'h5;

always @(posedge clk or negedge rst_n)
begin
  if (!rst_n) state <= LOCKED;
  else        state <= state_nx;
end

always @(*)
begin
   state_nx = state;
   case (state)
   LOCKED   : if(pay)         state_nx = VERIFY;
              else if(push)   state_nx = ALARM;
   VERIFY   : if(pass)        state_nx = UNLOCKED;
              else if(reject) state_nx = LOCKED;
   UNLOCKED : if(push)        state_nx = COUNTER;
              else if(pay)    state_nx = REFUND;
   COUNTER  : if(all_pass)    state_nx = LOCKED;
              else            state_nx = UNLOCKED;
   REFUND   : if(timeout)     state_nx = UNLOCKED;
   ALARM    : if(timeout)     state_nx = LOCKED;
   default  :                 state_nx = LOCKED;
   endcase
end

assign gate_close  = (state == LOCKED)  | (state == VERIFY) | (state == ALARM);
assign gate_open   = (state == UNLOCKED)| (state == COUNTER)| (state == REFUND);
assign go_alarm    = (state == ALARM);

assign green_light = (state == VERIFY) & (pass) | (state == COUNTER)  & (push);
assign beep        = (state == LOCKED) & (push) | (state == UNLOCKED) & (pay);
```

- **Draw block diagrams before you write codes !!!**
- Declare states, decide next states of every possible inputs, and decide outputs
- Choose names that are short and easy-to-understand
- Decide an initial state
- To think through all inputs completely
- Decide the next state of undeclared states
- Write outputs from Moore machine or Mealy machine
- Check if there is no redundant state

Agenda

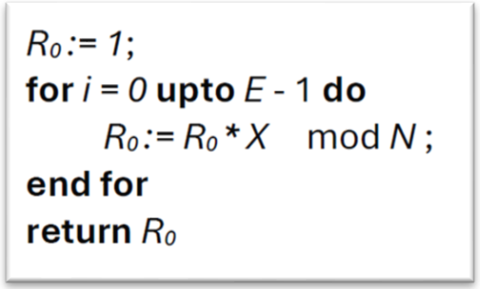
1. From Code to Chip
2. Architectural Planning
3. Module Planning
4. Finite State Machine
5. Coding Guideline

Turning Loops into Designs (1/2)

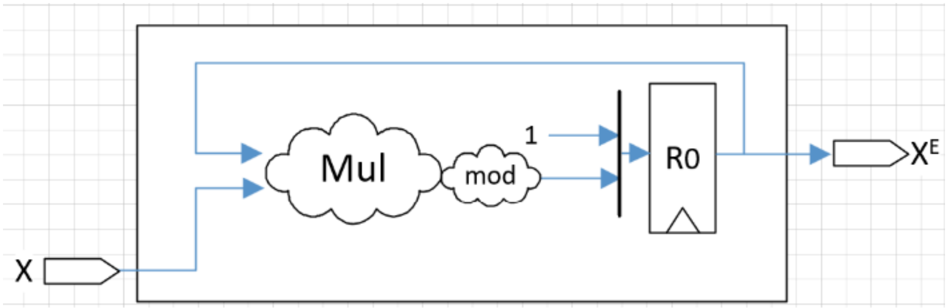
- Design Specification

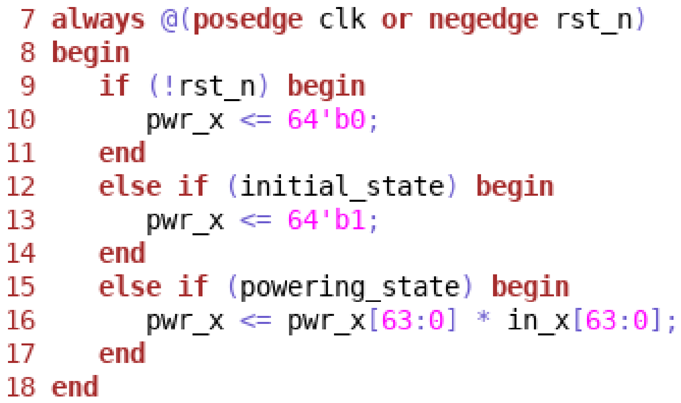
Input	$X, E=(e_{k-1},\dots, e_1, e_0)_2$
Output	X^E mod N $(N=2^{64})$

- Turning specification into loops
- Turning loops into designs
- Turning designs into code



```
$R_0 := 1$;
for $i = 0$ upto $E - 1$ do
    $R_0 := R_0 * X$  mod $N$;
end for
return $R_0$
```



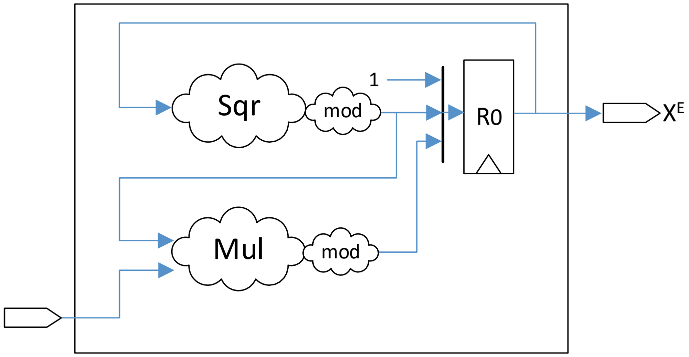


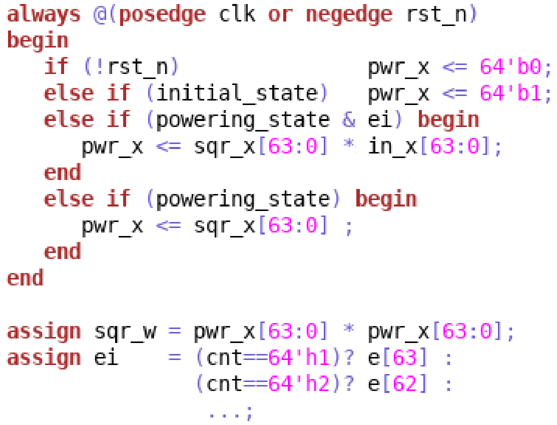
```
always @(posedge clk or negedge rst_n)
begin
    if (!rst_n) begin
        pwr_x <= 64'b0;
    end
    else if (initial_state) begin
        pwr_x <= 64'b1;
    end
    else if (powering_state) begin
        pwr_x <= pwr_x[63:0] * in_x[63:0];
    end
end
```

Turning Loops into Designs (2/2)

- Reduce the number of loops

```
$R_0$ := 1;
for $i$ = $k$-1 downto 0 do
    $R_0$ := $R_0 * R_0$   mod $N$ ;
  if $e_i$ = 1 then
    $R_0$ := $R_0 * X$    mod $N$ ;
  end if
end for
return $R_0$
```

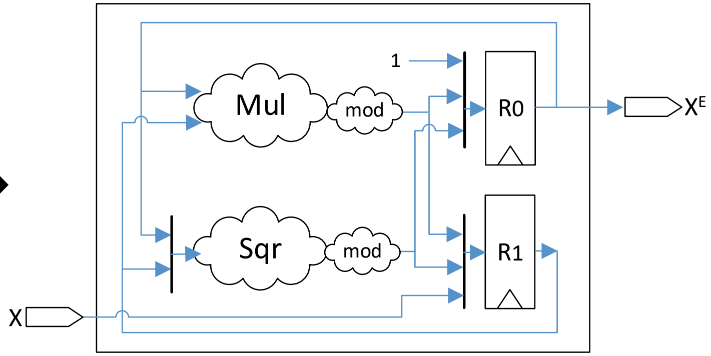


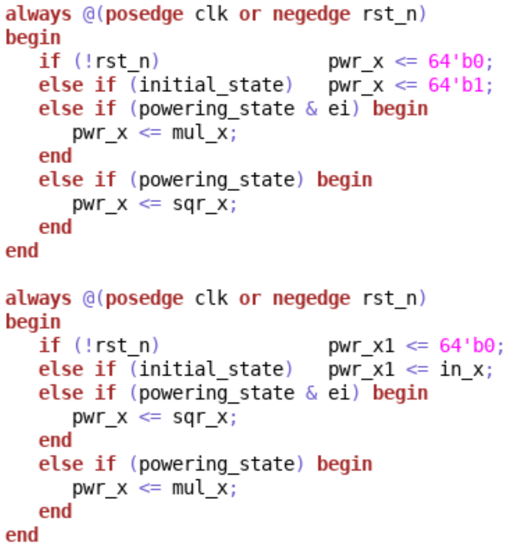


```
always @(posedge clk or negedge rst_n)
begin
   if (!rst_n)                   pwr_x <= 64'b0;
   else if (initial_state)       pwr_x <= 64'b1;
   else if (powering_state & ei) begin
      pwr_x <= sqr_x[63:0] * in_x[63:0];
   end
   else if (powering_state) begin
      pwr_x <= sqr_x[63:0] ;
   end
end

assign sqr_w = pwr_x[63:0] * pwr_x[63:0];
assign ei    = (cnt==64'h1)? e[63] :
               (cnt==64'h2)? e[62] :
                ...;
```

- Shorten the critical path

```
$R_0$ := 1;    $R_1$ := $X$ ;
for $i$ = $k$-1 downto 0 do
  if $e_i$ = 1 then
    $R_0$ := $R_0 * R_1$   mod $N$ ;
    $R_1$ := $R_1 * R_1$   mod $N$ ;
  else
    $R_1$ := $R_0 * R_1$   mod $N$ ;
    $R_0$ := $R_0 * R_0$   mod $N$ ;
  end if
end for
return $R_0$
```

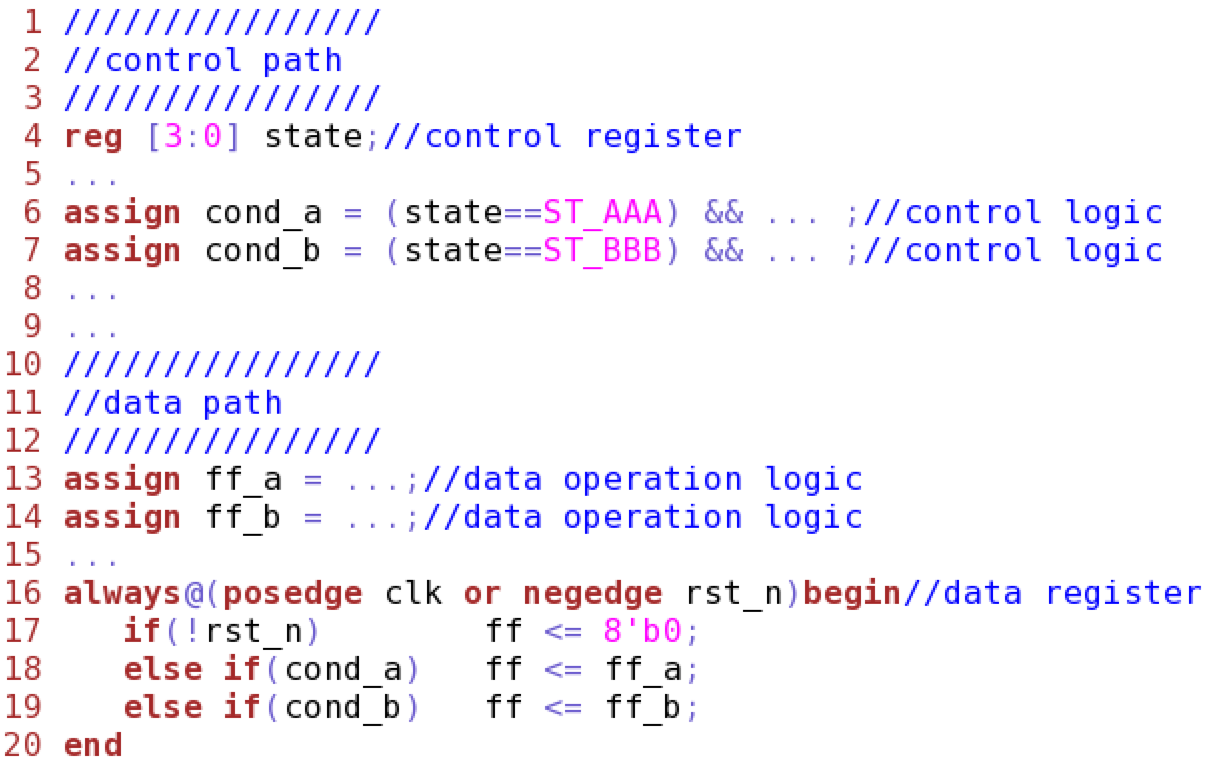




```
always @(posedge clk or negedge rst_n)
begin
   if (!rst_n)                   pwr_x <= 64'b0;
   else if (initial_state)       pwr_x <= 64'b1;
   else if (powering_state & ei) begin
      pwr_x <= mul_x;
   end
   else if (powering_state) begin
      pwr_x <= sqr_x;
   end
end

always @(posedge clk or negedge rst_n)
begin
   if (!rst_n)                   pwr_x1 <= 64'b0;
   else if (initial_state)       pwr_x1 <= in_x;
   else if (powering_state & ei) begin
      pwr_x <= sqr_x;
   end
   else if (powering_state) begin
      pwr_x <= mul_x;
   end
end
```

Control Path And Data Path (1/2)

- ***Internal signal** – planning the sub-module itself*
  - Control path
    - Main control register
      - State machine
    - Minor control register
      - Counter, flag
    - Fine control logic
      - Condition that trigger the register
  - Data path
    - Operation logic
    - Data Register



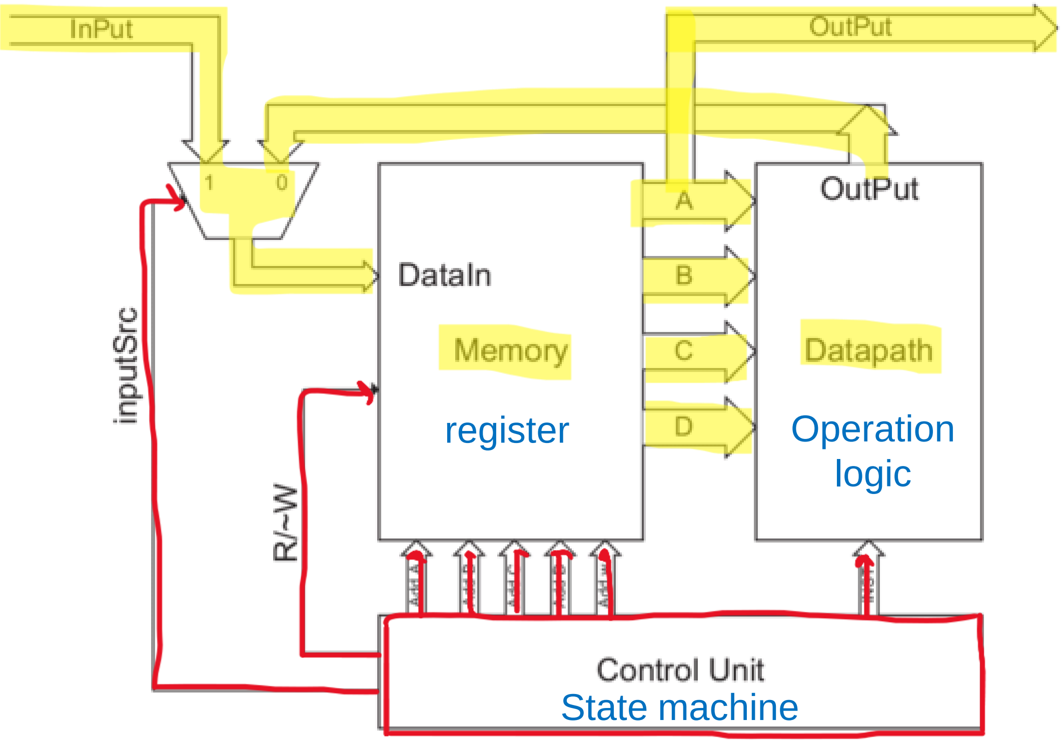
```
////////////////
//control path
////////////////
reg [3:0] state;//control register
...
assign cond_a = (state==ST_AAA) && ... ;//control logic
assign cond_b = (state==ST_BBB) && ... ;//control logic
...
...
////////////////
//data path
////////////////
assign ff_a = ...;//data operation logic
assign ff_b = ...;//data operation logic
...
always@(posedge clk or negedge rst_n)begin//data register
   if(!rst_n)        ff <= 8'b0;
   else if(cond_a)   ff <= ff_a;
   else if(cond_b)   ff <= ff_b;
end
```

Control Path And Data Path (2/2)

- SHA256
  - García, Rommel & Algredo-Badillo, Ignacio & Morales-Sandoval, Miguel & Feregrino, Claudia & Cumplido, René. (2013). A compact FPGA-based processor for the Secure Hash Algorithm SHA-256. Computers & Electrical Engineering. 40. 10.1016/j.compeleceng.2013.11.014.

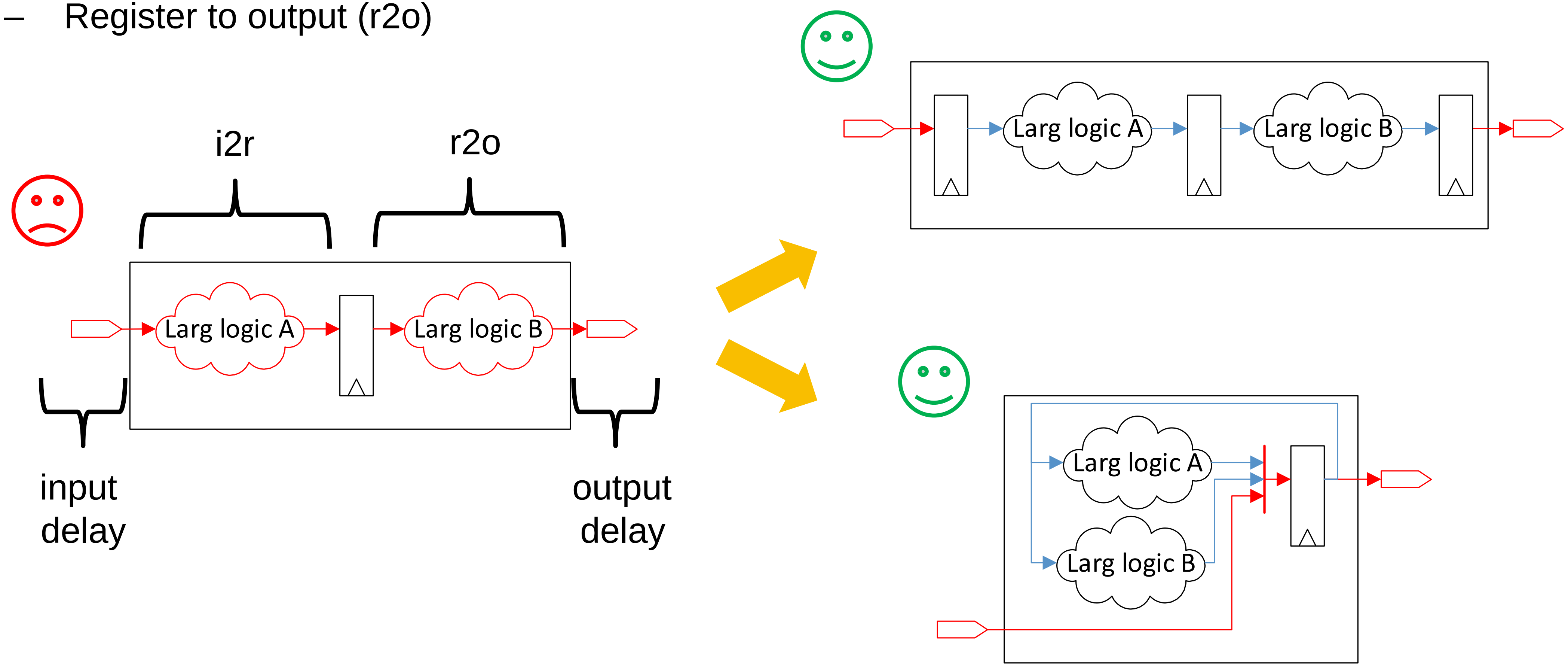
Data path

Control path



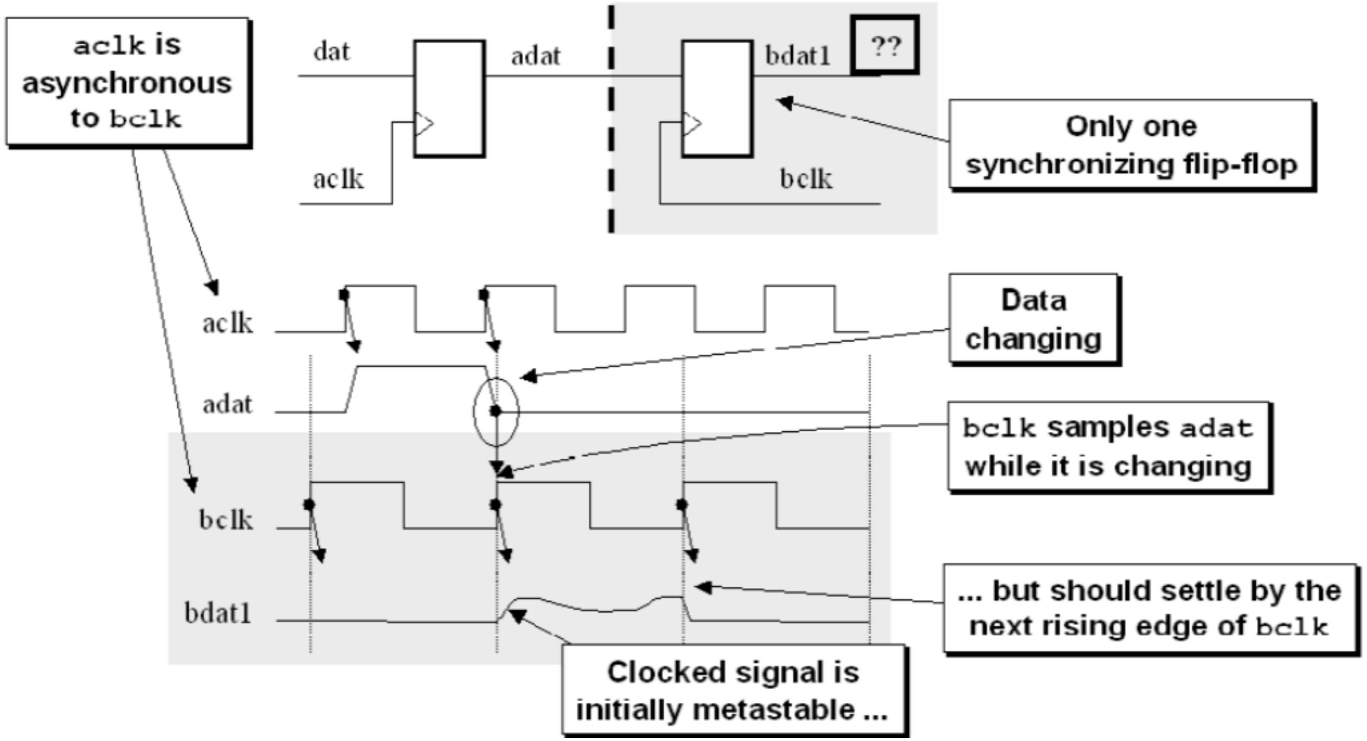
Input Delay And Output Delay

- Minimize the input and output logic path of top-module
  - Input to register (i2r)
  - Register to output (r2o)



Clock Domain Crossing

- Clock Domain Crossing (CDC)
  - The signal goes from the *aclk* domain to the *bclk* domain. There is no fixed phase relationship between *aclk* and *bclk*

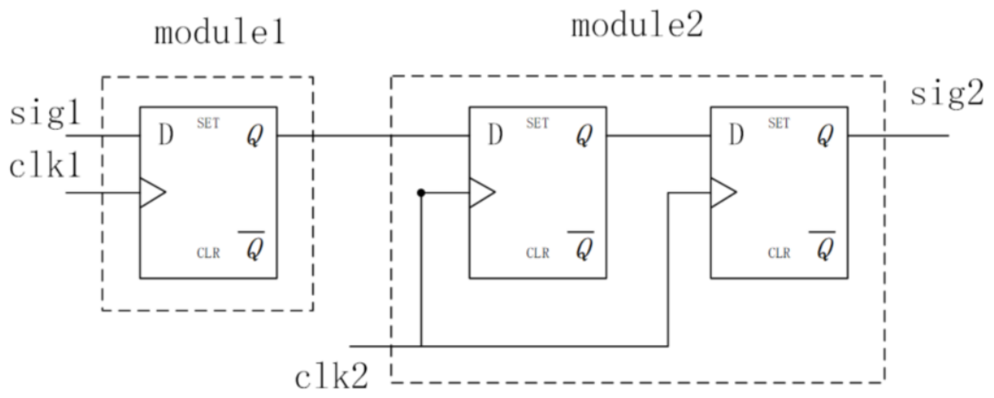


aclk is asynchronous to bclk
dat
adat
bdat1
??
aclk
bclk
Only one synchronizing flip-flop
aclk
Data changing
adat
bclk samples adat while it is changing
bclk
... but should settle by the next rising edge of bclk
bdat1
Clocked signal is initially metastable ...

- Metastability
  - When *adat* changes near the *bclk* sampling point, the setup time and hold time are violation, *bdat* is in an uncertain value for a long time

Clock Domain Crossing

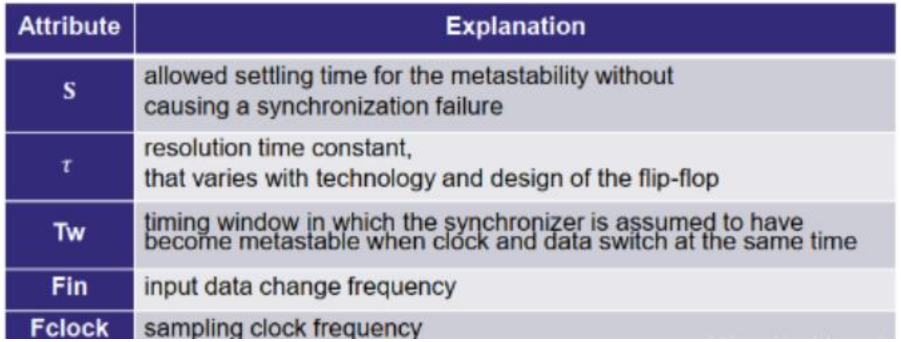
- Synchronizer
  - The common processing method is two-stage flip-flops. The Q end of the 1st FF appears metastable, and the 2nd FF samples at the same frequency, which is generally a stable level



- Module 2 can only have one output “sig2.” Do not use any signal as output from module 2

- The higher frequency, the more l ikely it appears metastable, you may need to use the 3rd FF

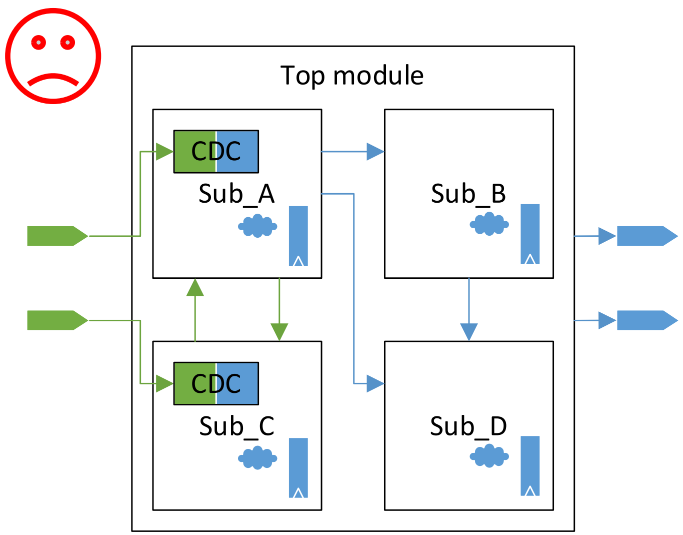
$$MTBF = \frac{e^{(S/\tau)}}{T_W \cdot F_{in} \cdot F_{clock}}$$

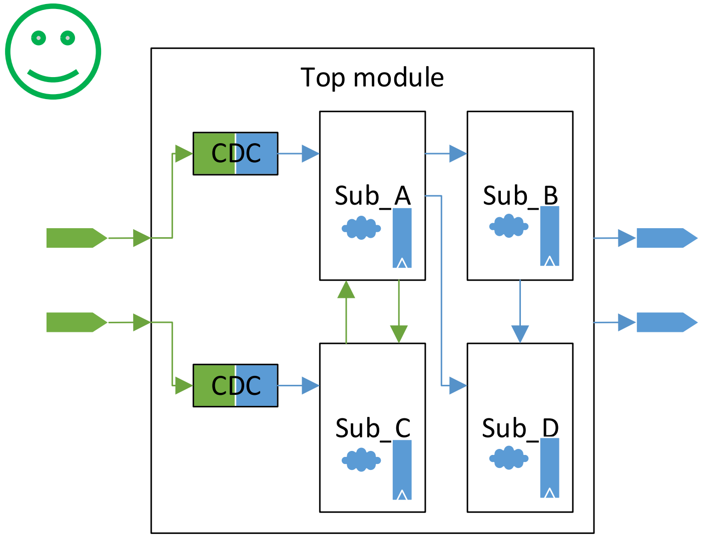


Attribute	Explanation
S	allowed settling time for the metastability without causing a synchronization failure
τ	resolution time constant, that varies with technology and design of the flip-flop
Tw	timing window in which the synchronizer is assumed to have become metastable when clock and data switch at the same time
Fin	input data change frequency
Fclock	sampling clock frequency

Clock Domain Crossing

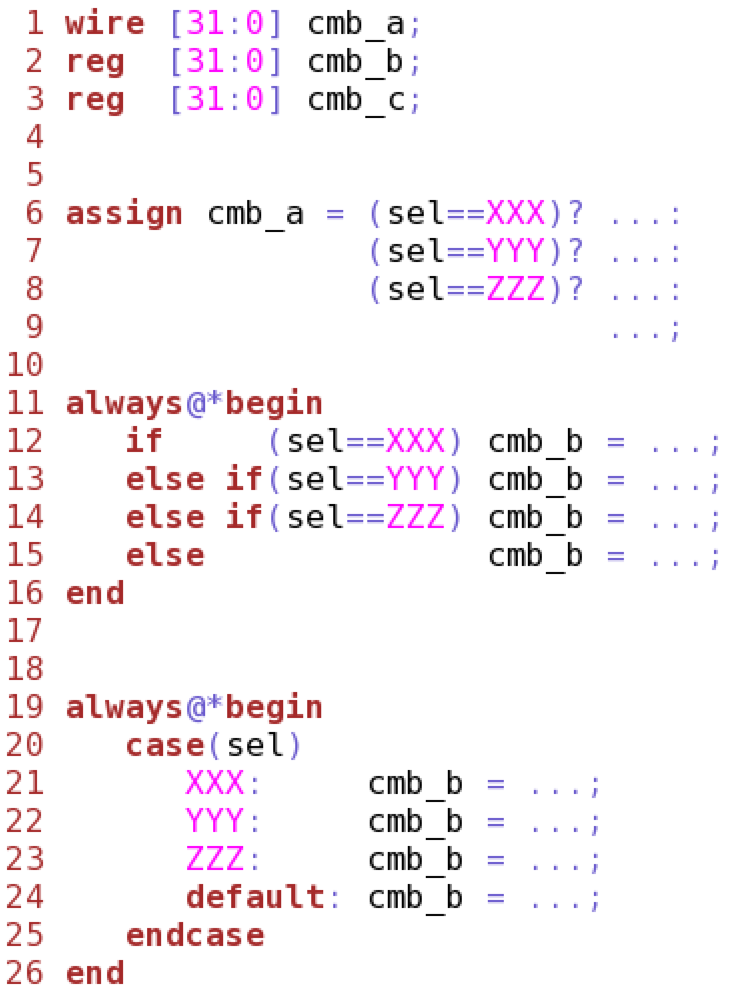
- Planning a Dedicated Module for CDC
  - Data path: Asynchronous FIFO
  - Control path: Synchronizer
- Keep sub-modules in a single clock domain





Coding style

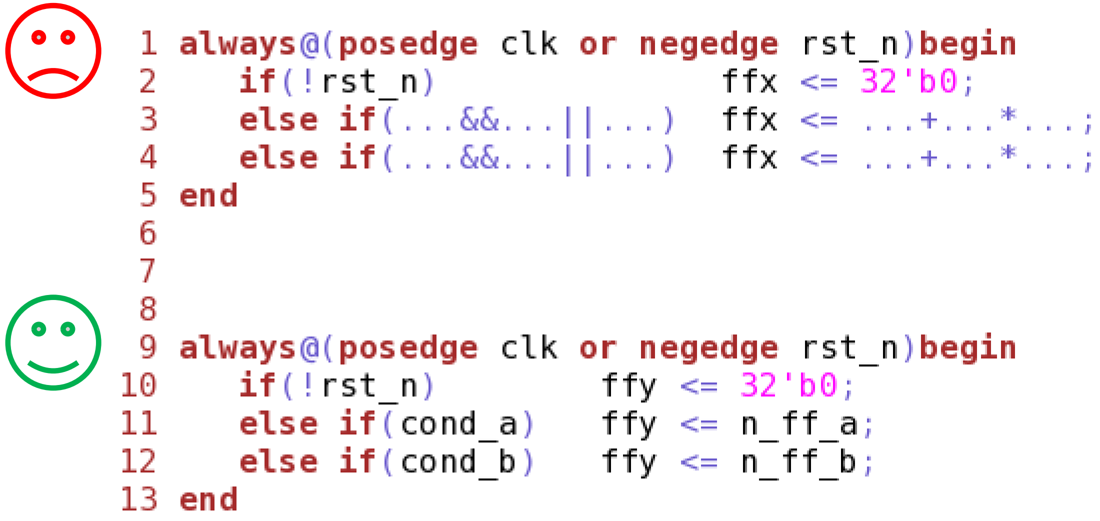
- Combinational logic
  - Use assign
  - Or 1 reg in 1 always(*) block
    - Avoiding topology errors
      which leads to simulation/synthesis mismatch



```verilog
wire [31:0] cmb_a;
reg  [31:0] cmb_b;
reg  [31:0] cmb_c;


assign cmb_a = (sel==XXX)? ...:
               (sel==YYY)? ...:
               (sel==ZZZ)? ...:
                           ...;

always@*begin
   if     (sel==XXX) cmb_b = ...;
   else if(sel==YYY) cmb_b = ...;
   else if(sel==ZZZ) cmb_b = ...;
   else              cmb_b = ...;
end


always@*begin
   case(sel)
      XXX:     cmb_b = ...;
      YYY:     cmb_b = ...;
      ZZZ:     cmb_b = ...;
      default: cmb_b = ...;
   endcase
end
```

Coding style

- Flipflop
  - 1 reg in 1 always(posedge clk) block
  - Separate control logic
  - Separate operation logic
    - Make the code
      look more like a circuit diagram.



☹

```
always@(posedge clk or negedge rst_n)begin
   if(!rst_n)              ffx <= 32'b0;
   else if(...&&...||...)  ffx <= ...+...*...;
   else if(...&&...||...)  ffx <= ...+...*...;
end
```

☺

```
always@(posedge clk or negedge rst_n)begin
   if(!rst_n)        ffy <= 32'b0;
   else if(cond_a)   ffy <= n_ff_a;
   else if(cond_b)   ffy <= n_ff_b;
end
```

Feedback to us

https://forms.office.com/r/DYDu8vLaWN

Thank you!

PUFacademy

A PUFsecurity Alliance

Visit our website: pufacademy.com

Contact us: PUFacademy@pufsecurity.com