



CT Verilog Series

Design Example

黃稚存

Chih-Tsun Huang

cthuang@cs.nthu.edu.tw



國立清華大學
資訊工程學系

聲明

- ◎ 本課程之內容 (包括但不限於教材、影片、圖片、檔案資料等)，僅供修課學生個人合理使用，非經授課教師同意，不得以任何形式轉載、重製、散布、公開播送、出版或發行本影片內容 (例如將課程內容放置公開平台上，如 Facebook, Instagram, YouTube, Twitter, Google Drive, Dropbox 等等)。如有侵權行為，需自負法律責任。

Outline

- ① Procedure for Digital Design with Verilog
- ① Control and Data Partitioning
- ① Design Example of GCD Engine
- ① Requirements for GCD Engine
- ① Summary

Procedure for Digital Design with Verilog

Design Procedure

- ◉ Function Description
- ◉ IO Specification
- ◉ Control and Data Partitioning
(top down vs. bottom up)
 - ◆ N -bit Ripple-carry adder partitioned into N 1-bit full adders
 - ▣ Obsolete, way-too-simple example!!
 - ◆ **Finite-State Machine + Datapath**



Block/State Diagram + Timing Diagram

- ◉ Verilog Coding
- ◉ Simulation and Verification

Control and Data Partitioning

- » Sequential circuits can be partitioned into control unit and datapath
- » Datapath: combinational blocks, registers, counters, etc.
- » Control unit: usually FSMs

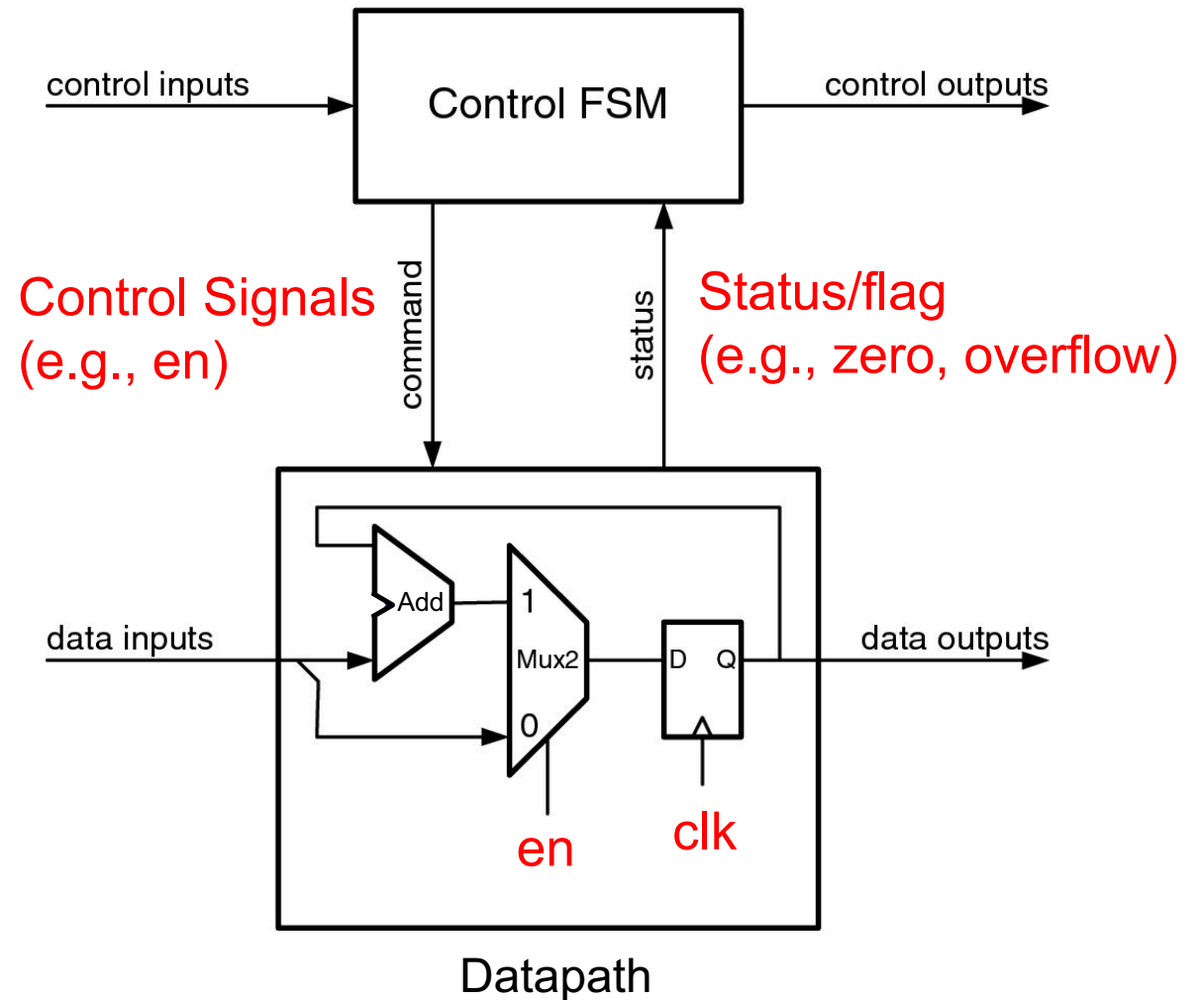
Datapath/Control Partitioning

◉ Datapath

- ◆ Function (arithmetic) units that perform **data processing** (e.g., add, multiply, etc.), **data registering** (e.g., register, buffer, etc.), and **data moving** (e.g., interconnect)
- ◆ Also produce **status (flag) signals**

◉ Control unit

- ◆ Behavior control usually determined by **FSMs**
- ◆ Also generate **control signals (commands)**



Design Example of GCD Engine

Design (Functional) Description

- ◉ Calculate the greatest common divisor (GCD) of two 8-bit positive integers
 - ◉ Using a START signal to load inputs
 - ◉ Generate a DONE signal to indicate the result
 - ◉ Assert an ERROR signal when one or more inputs are zero
-
- ◉ Note: you may come to a different design specification. This is just an example.

IO Specification

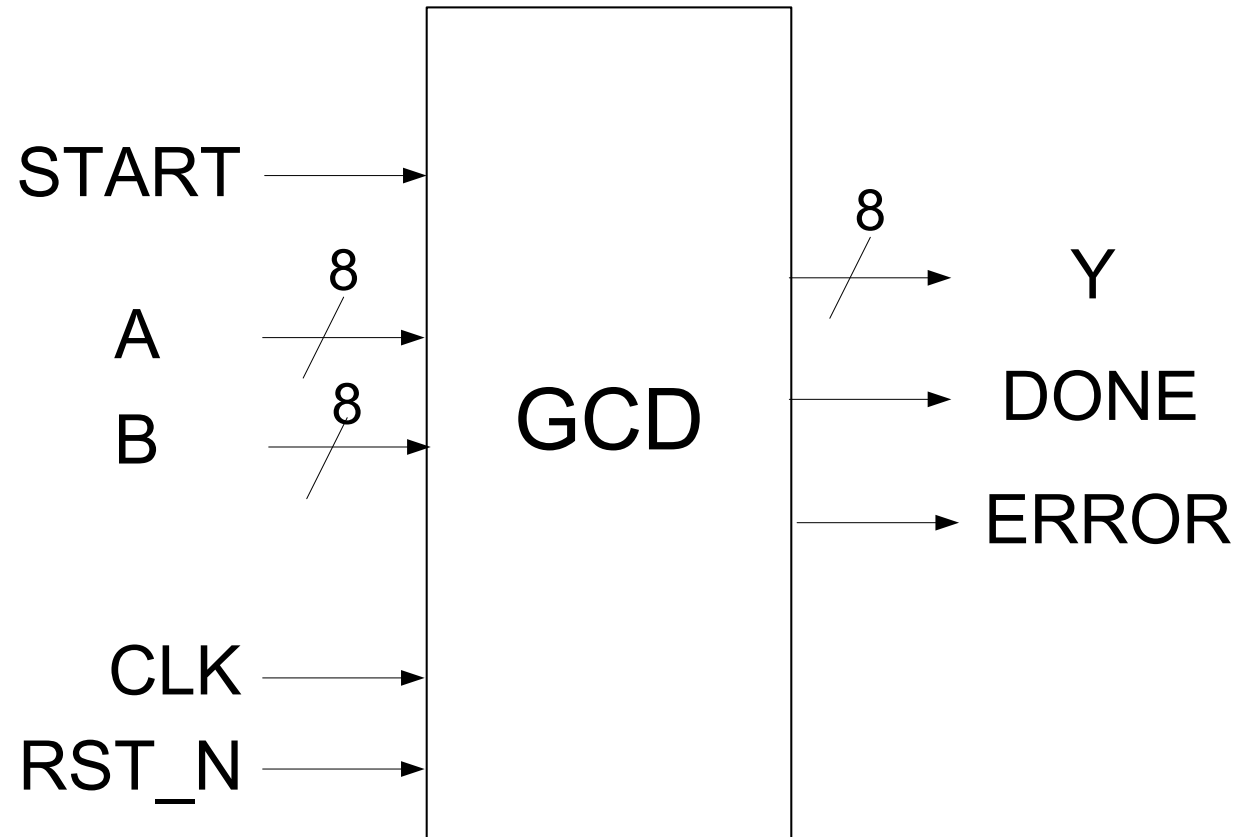
⦿ Primary inputs:

- ◆ *CLK*: clock source
- ◆ *RST_N*: reset (low active)
- ◆ *START*: to trigger the calculation
 - ▣ A **one-cycle pulse** to indicate the valid input numbers
- ◆ *A, B*: two 8-bit input numbers

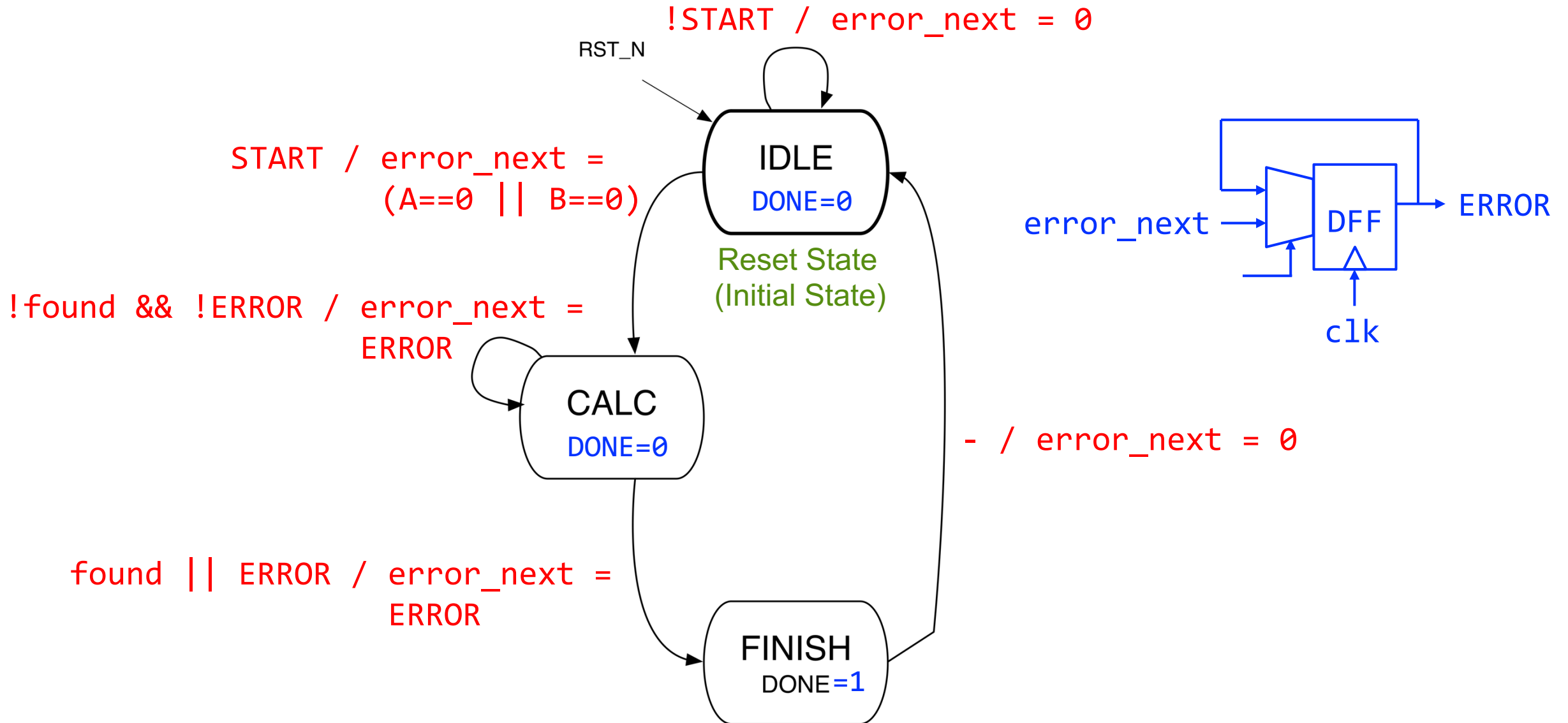
⦿ Primary outputs:

- ◆ *Y*: the result
- ◆ *DONE*: to indicate the valid output with **one-cycle pulse**
- ◆ *ERROR*: to indicate an error
 - ▣ 0: valid result
 - ▣ 1: invalid when $A = 0$ or $B = 0$

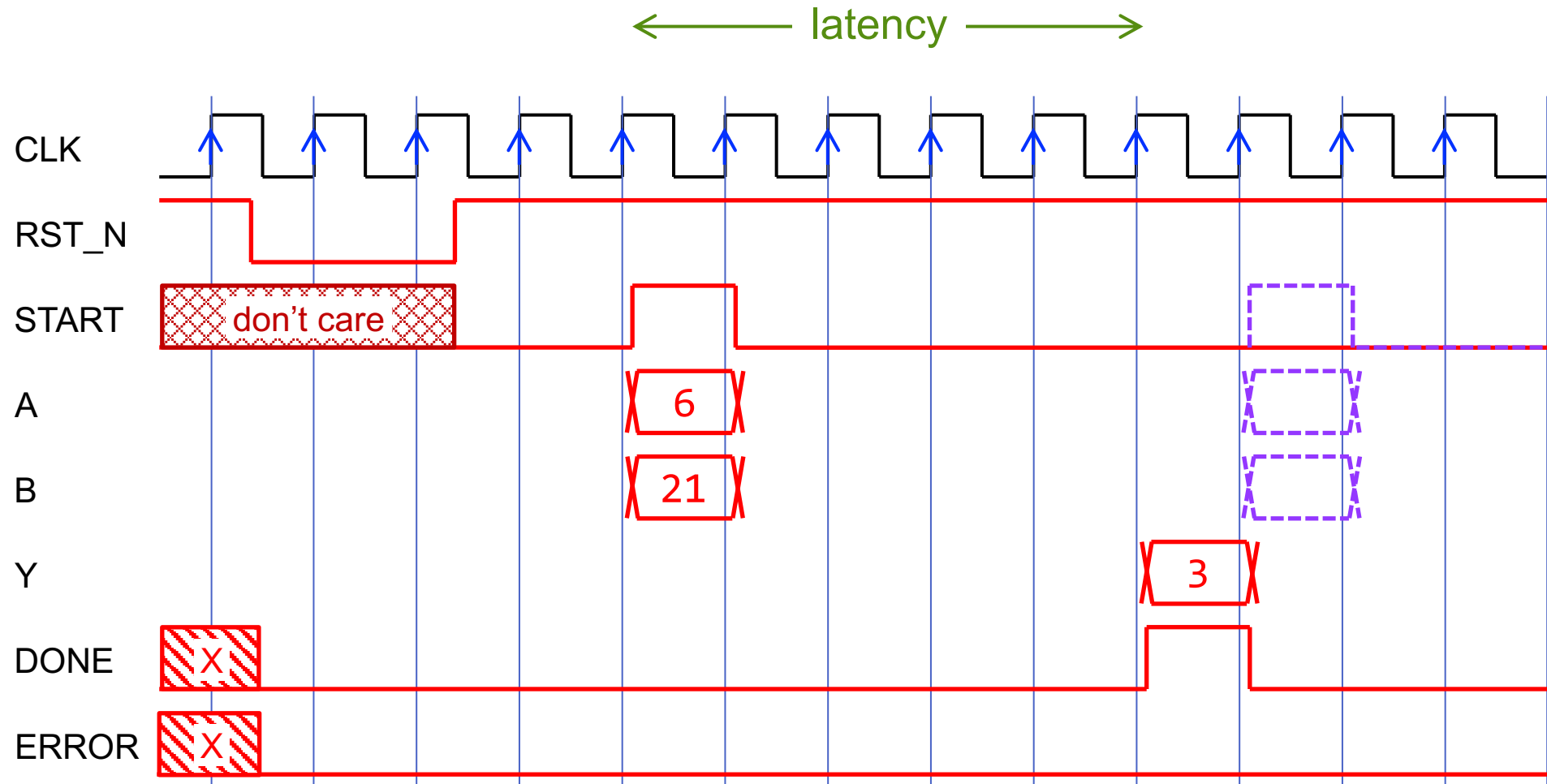
Top-level Block Diagram and Primary IOs



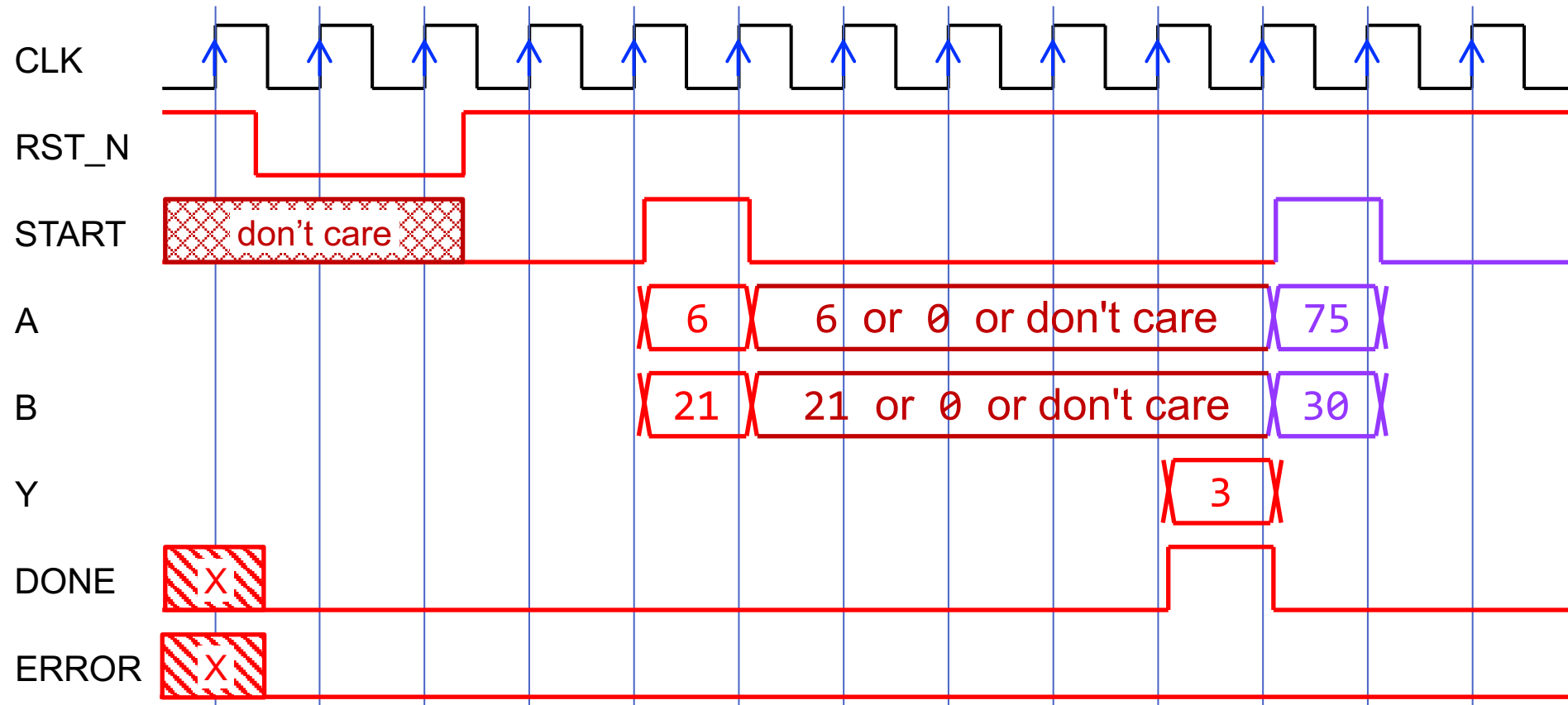
Finite-State Machine (Control Unit)



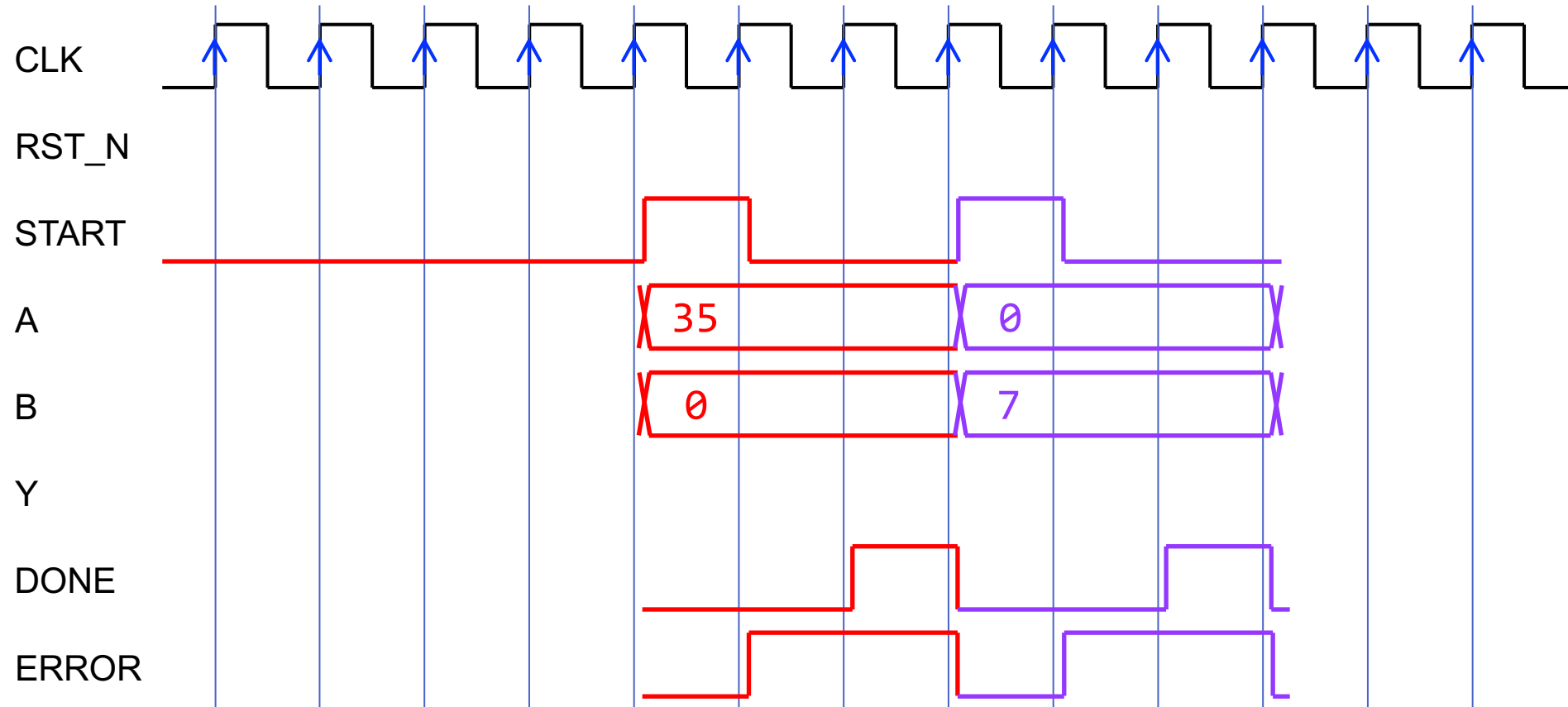
Timing Diagram – Normal Operation



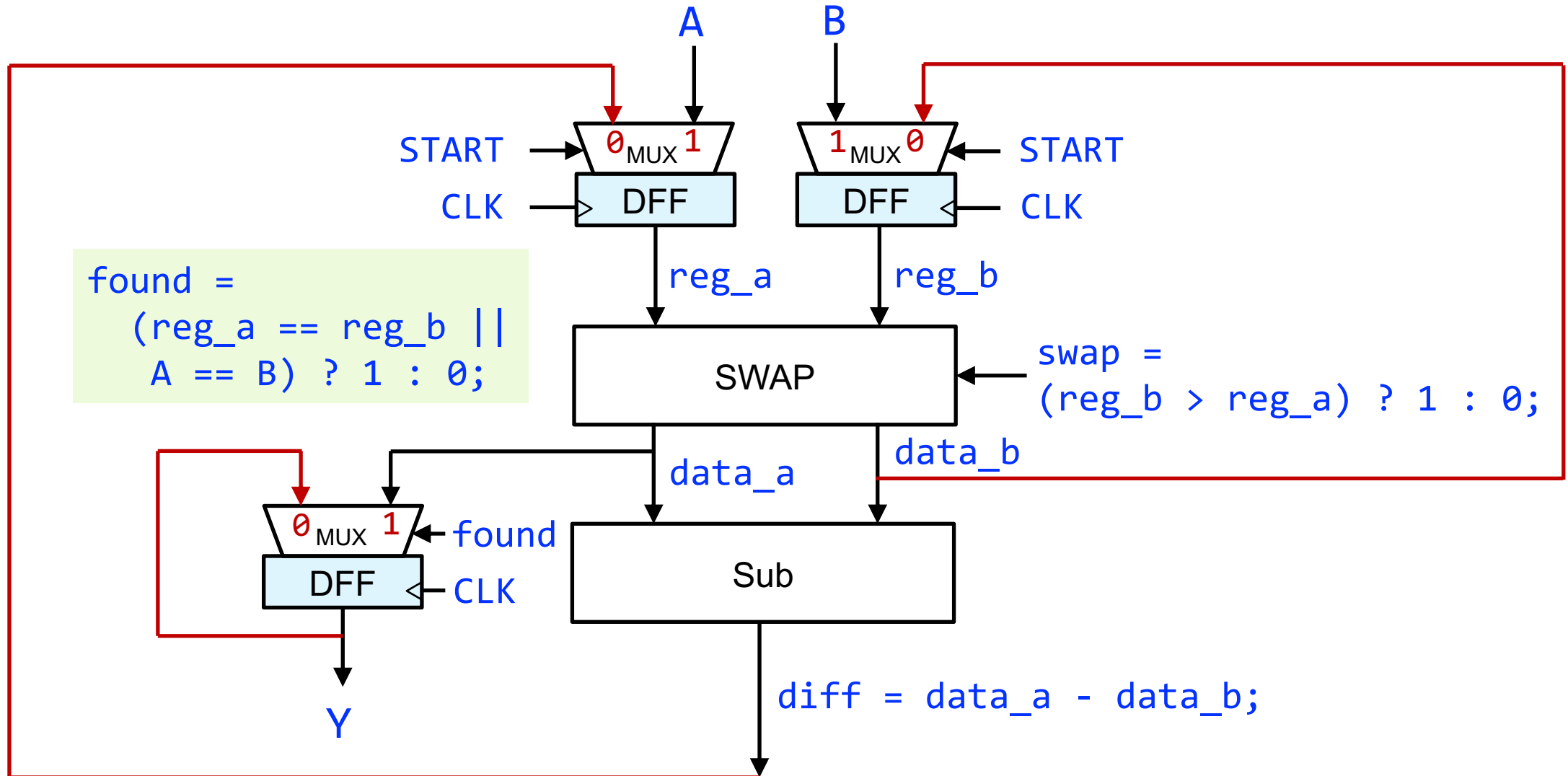
Timing Diagram – Normal Operation



Timing Diagram – Input with Error



Block Diagram of Datapath



Template: gcd.v

```
module GCD (  
    input wire CLK,  
    input wire RST_N,  
    input wire [7:0] A,  
    input wire [7:0] B,  
    input wire START,  
    output reg [7:0] Y,  
    output reg DONE,  
    output reg ERROR  
);  
  
wire found, swap;  
reg [7:0] reg_a, reg_b, data_a, data_b;  
reg [7:0] diff;  
reg error_next;  
reg [1:0] state, state_next;
```

```
parameter [1:0] IDLE = 2'b00;  
parameter [1:0] CALC = 2'b01;  
parameter [1:0] FINISH = 2'b10;  
  
// [HW]  
// Finish this design based on  
// the block diagram:  
//     1. FSM  
//     2. Datapath diagram  
//  
  
endmodule
```

Template: gcd_t.v (1/2)

```
`timescale 1ns/100ps
module stimulus;
    parameter cyc = 10;
    parameter delay = 1;

    reg clk, rst_n, start;
    reg [7:0] a, b;
    wire done, error;
    wire [7:0] y;

    GCD gcd01 (
// [HW] complete the port connections
//     .CLK(...)
//     .RST_N(...)
//     ...
//     ...
    );
```

Instance

```
always #(cyc/2) clk = ~clk; // clock

initial begin
    $fsdbDumpfile("gcd.fsdb");
    $fsdbDumpvars;

    $sdf_annotate ("gcd_syn.sdf", gcd01);

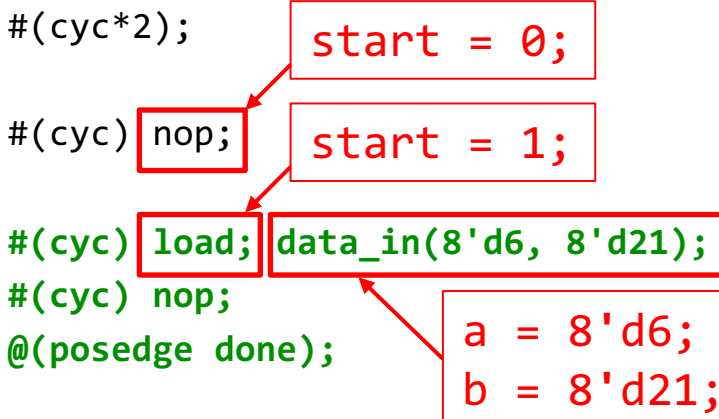
    $monitor($time, " CLK=%b RST_N=%b
START=%b A=%d B=%d | DONE=%b Y=%d
ERROR=%b",
        clk, rst_n, start, a, b, done,
        y, error);
end
```

Template: gcd_t.v (2/2)

```
initial begin
    clk = 1;
    rst_n = 1;
    #(cyc);
    #(delay) rst_n = 0;
    #(cyc*4) rst_n = 1;
    #(cyc*2);
    #(cyc) nop;
    #(cyc) load; data_in(8'd6, 8'd21);
    #(cyc) nop;
    @(posedge done);

    // [HW] apply more patterns to cover
    // different conditions

    #(cyc) nop;
    #(cyc*8);
    $finish;
end
```



```
// using tasks to improve the
// readability
task nop;
    begin
        start = 0;
    end
endtask
task load;
    begin
        start = 1;
    end
endtask
task data_in;
    input [7:0] data1, data2;
    begin
        a = data1;
        b = data2;
    end
endtask

endmodule
```

Simulation

- ◉ Syntax checking

```
$ ncverilog -c gcd.v
```

- ◉ Verilog simulation

```
$ ncverilog gcd_t.v gcd.v
```

- ◉ Debug using waveform viewer

```
$ nWave
```

Requirements for GCD Engine

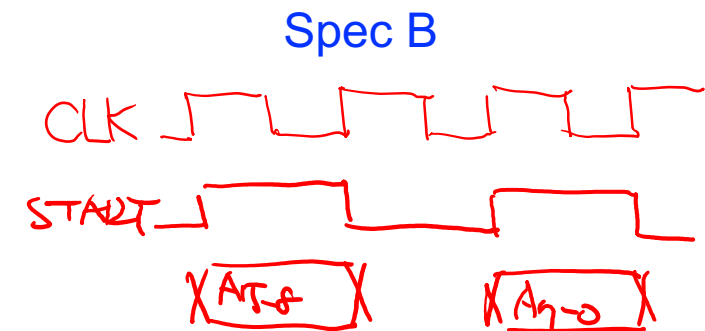
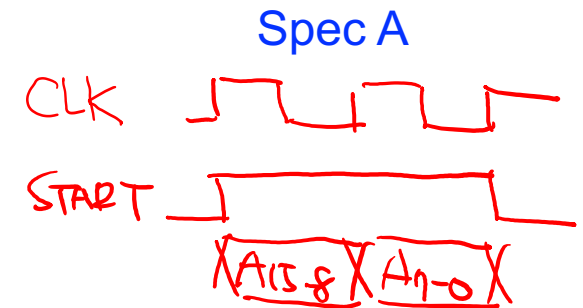
- » Baseline
- » Extensions

Baseline

- ⦿ Complete the GCD engine in Verilog with test stimulus
- ⦿ Try to cover **as many input conditions as you can** to verify the functionality
Most importantly, boundary conditions
 - ◆ Explain your selection of test patterns
- ⦿ Debug the timing using waveform viewer nWave
- ⦿ Discussion
 - ◆ You are encouraged to discuss if there is any room of improvement regarding to the timing
 - ◆ How difficult to achieve it?

Extension 1: Data-Width Extension

- Extend your design to support GCD calculation of two 16-bit positive integers
- The I/O names and widths remain the same
 - ◆ A , B , and Y are still 8-bit signals
 - ◆ You need two cycles to complete the data transfer
 - E.g., the 1st cycle: $A[15:8]$; the 2nd cycle: $A[7:0]$
 - The result also needs two cycles
 - ◆ $START$ and $DONE$ are two-cycle pulses
 - Assume they are asserted for two consecutive cycles (Spec A)
 - Discussion:
 - Can you support the input control which allows two separate active cycles? (Spec B)
 - Which one is easier to design?



Extension 2: Performance Improvement

- ⊙ The example uses repeated subtractions for GCD calculation
- ⊙ It will require a lot of subtractions if $A \gg B$
 - ◆ A lot of cycles to compute the GCD
- ⊙ Can you improve it?
 - ◆ Long division?

→
11 $\overline{) 100000000}$

Summary

Design Procedure in More Details

- ① Start with spec, primary I/Os, and block diagram
- ① Design the overall architecture
 - ◆ Specify finite-state machine(s), datapath, and internal signals (control, status, interconnects)
 - ▣ Partition into subblocks to ease your design
 - ◆ List the timing diagrams of major operations
- ① Verilog coding
- ① Prepare comprehensive test environment for efficient debugging

Guideline for Verilog Design

- ⦿ Prepare the block diagram and FSM carefully
- ⦿ Try to **symbolize (abstract)** each condition for
 - ◆ State transitions
 - ◆ Mode selection for datapath
- ⦿ FSM: the simpler the better
 - ◆ One single state can take multiple cycles
 - ◆ States can be nested

💡 **Design should be done before Verilog coding**

→ Avoid blind debugging!!

💡 **There is no one BEST style for every design**

- ◆ Define your own coding elegance
 - ▣ Naming convention
 - ▣ State arrangement