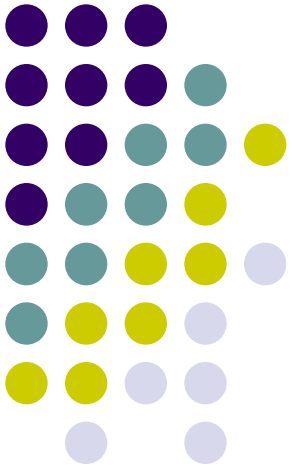
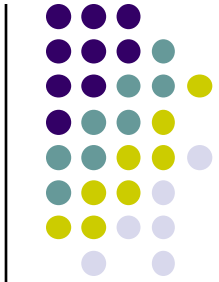
Lab 02
GCD Engine
Due on Oct 16th 2016 11:59pm

黃稚存

Department of Computer Science
National Tsing Hua University



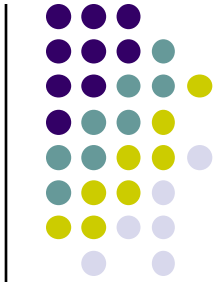
Description



- 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

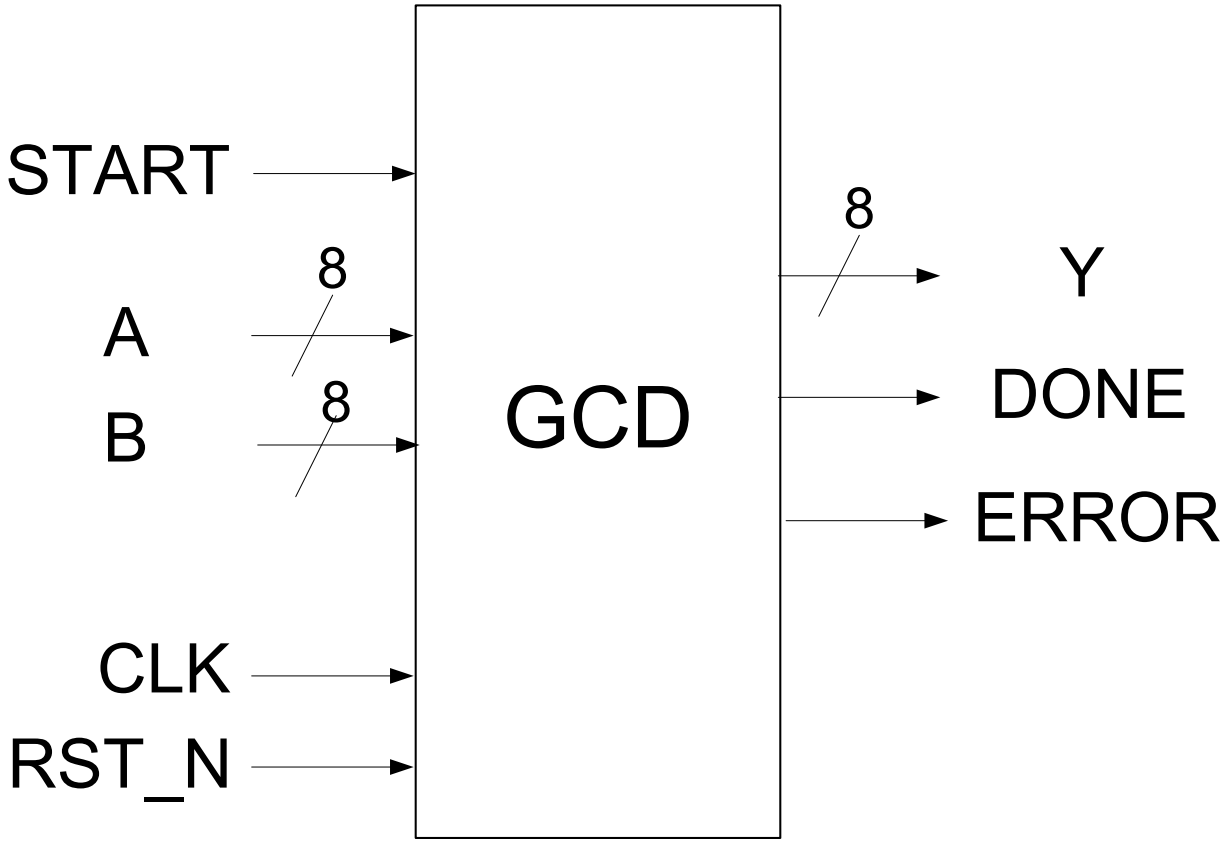
- Note: this is not a full-featured design
  - Only a simple example
  - Do not copy & paste it into your research work

IO Specification

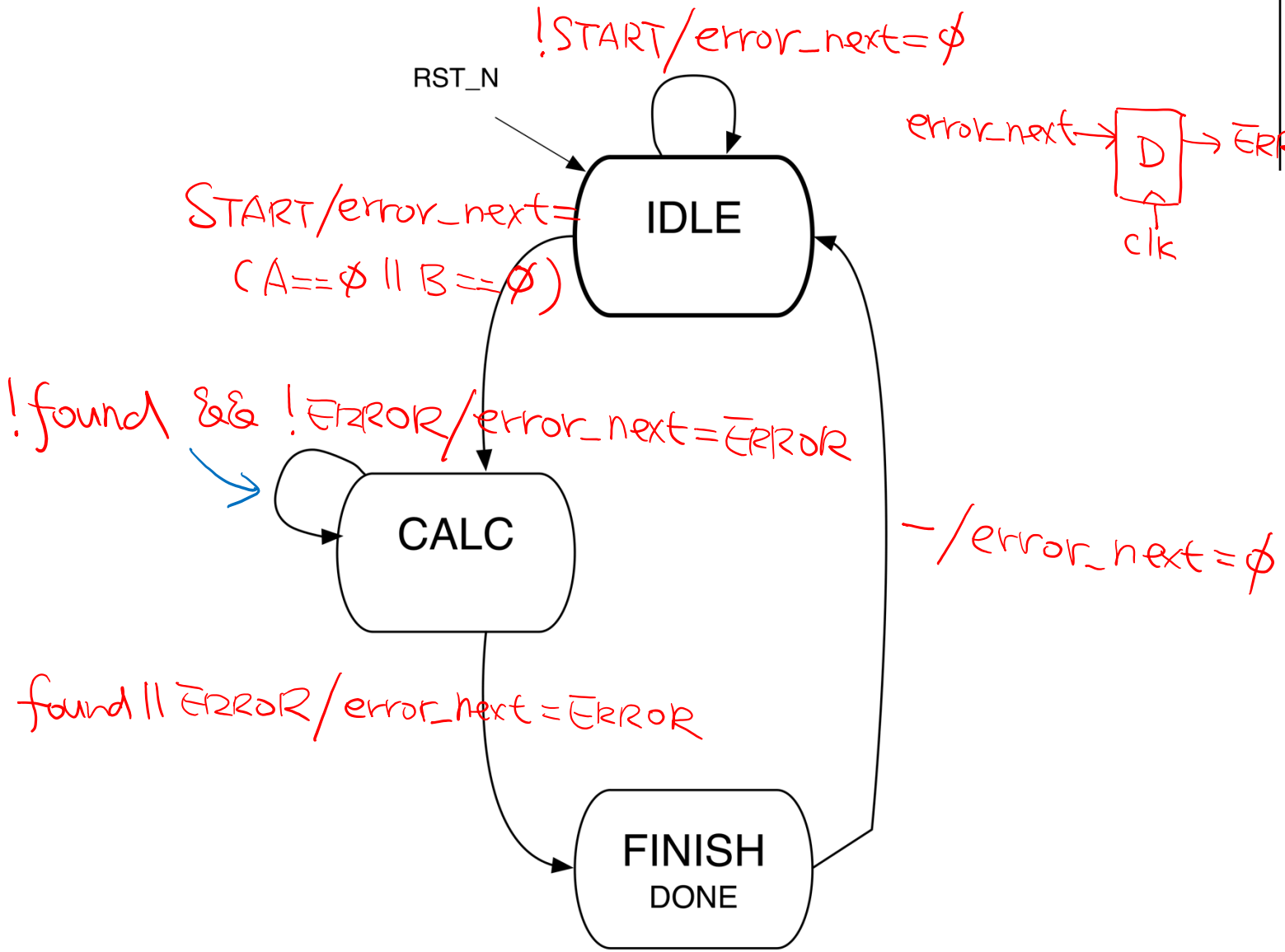


- Inputs:
  - *CLK*: clock source
  - *RST_N*: reset (low active)
  - *START*: to trigger the calculation
    - A one-cycle pulse to indicate the valid input numbers
  - *A, B*: two 8-bit input numbers
- Outputs:
  - *Y*: the result
  - *DONE*: to indicate the valid output with one-cycle pulse
  - *ERROR*: to indicate an error
    - 0: normal result
    - 1: invalid when A = 0 or B = 0

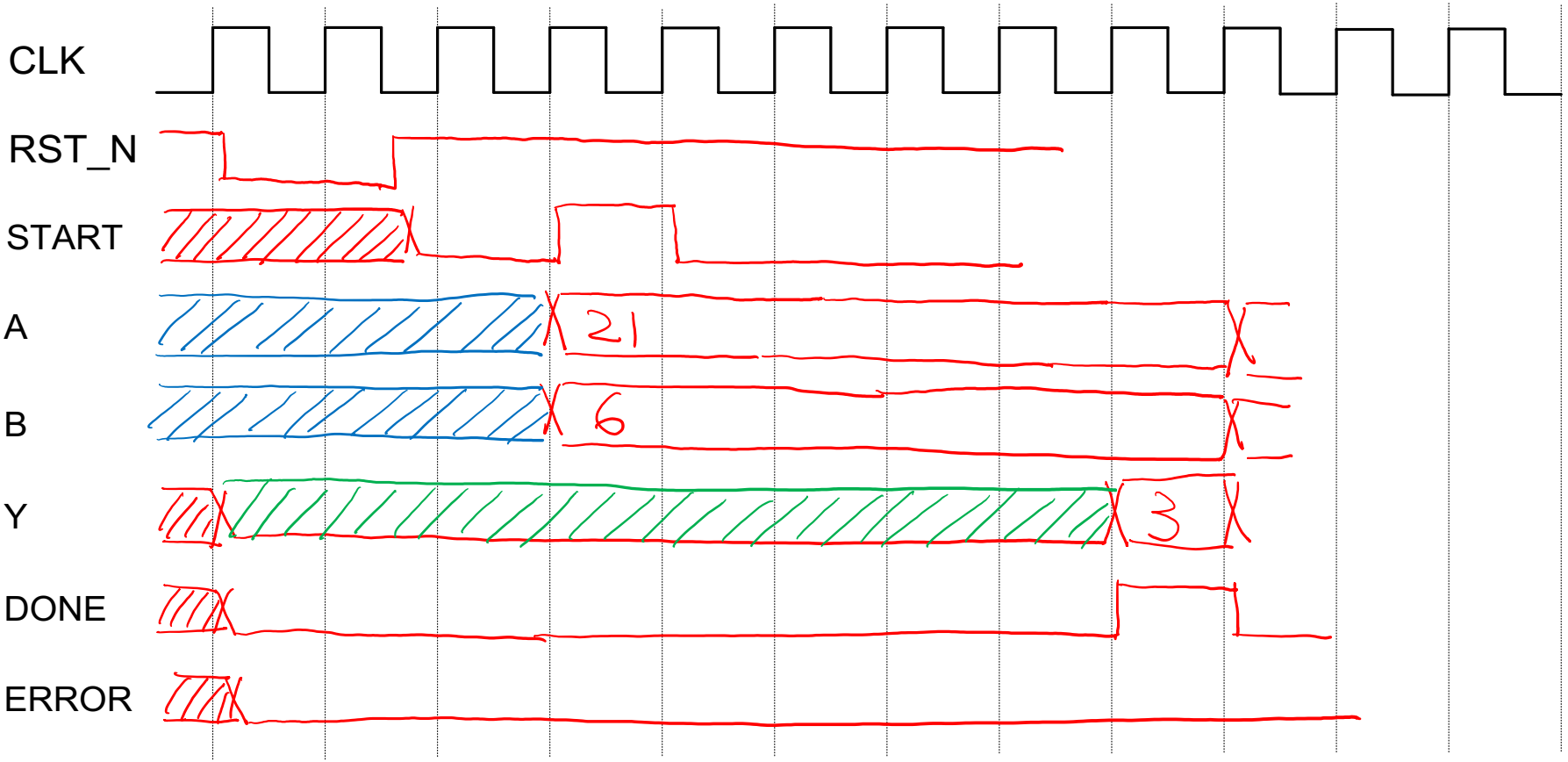
Primary IOs and Block Diagram



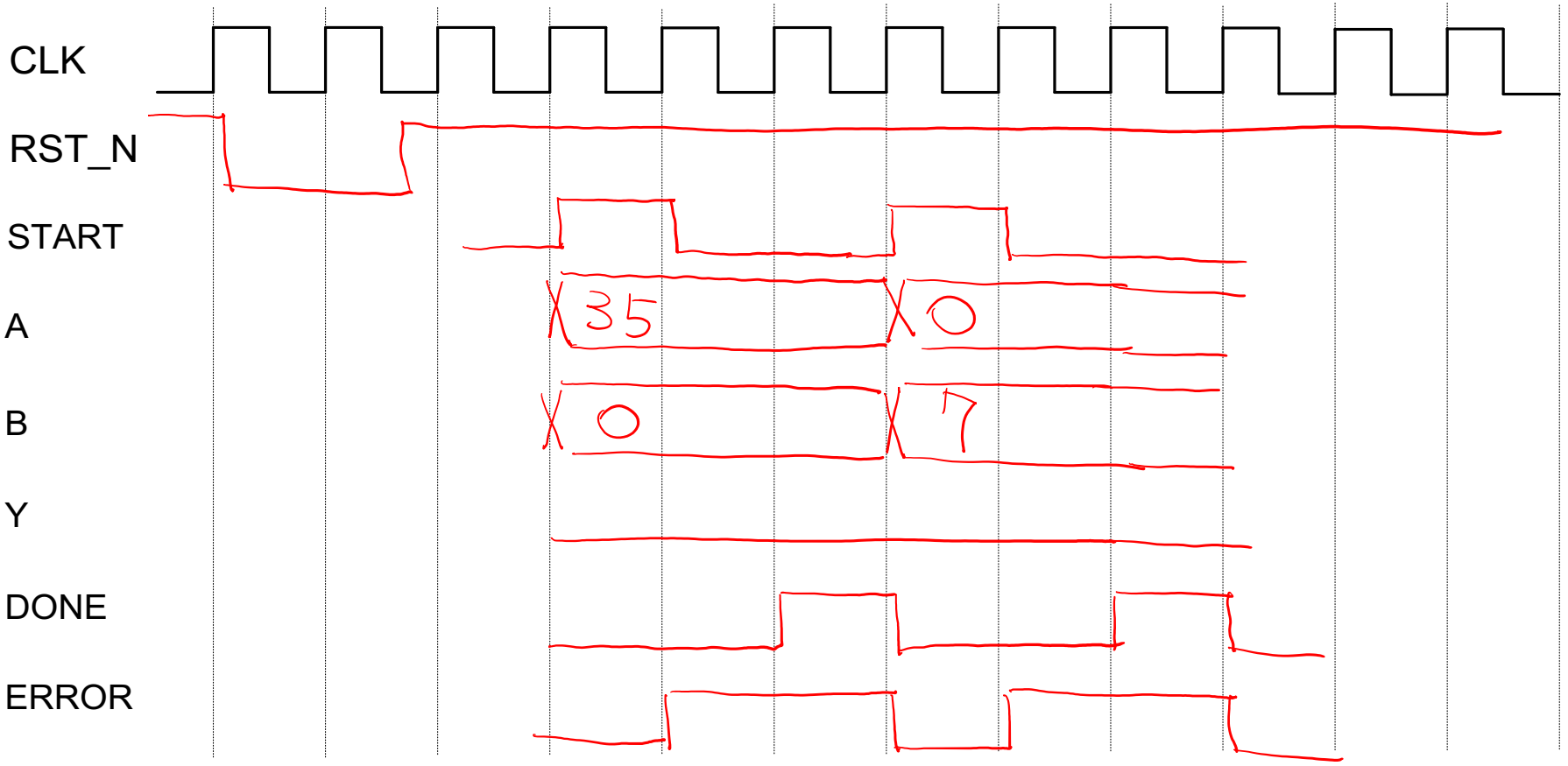
Finite State Machine (Control)



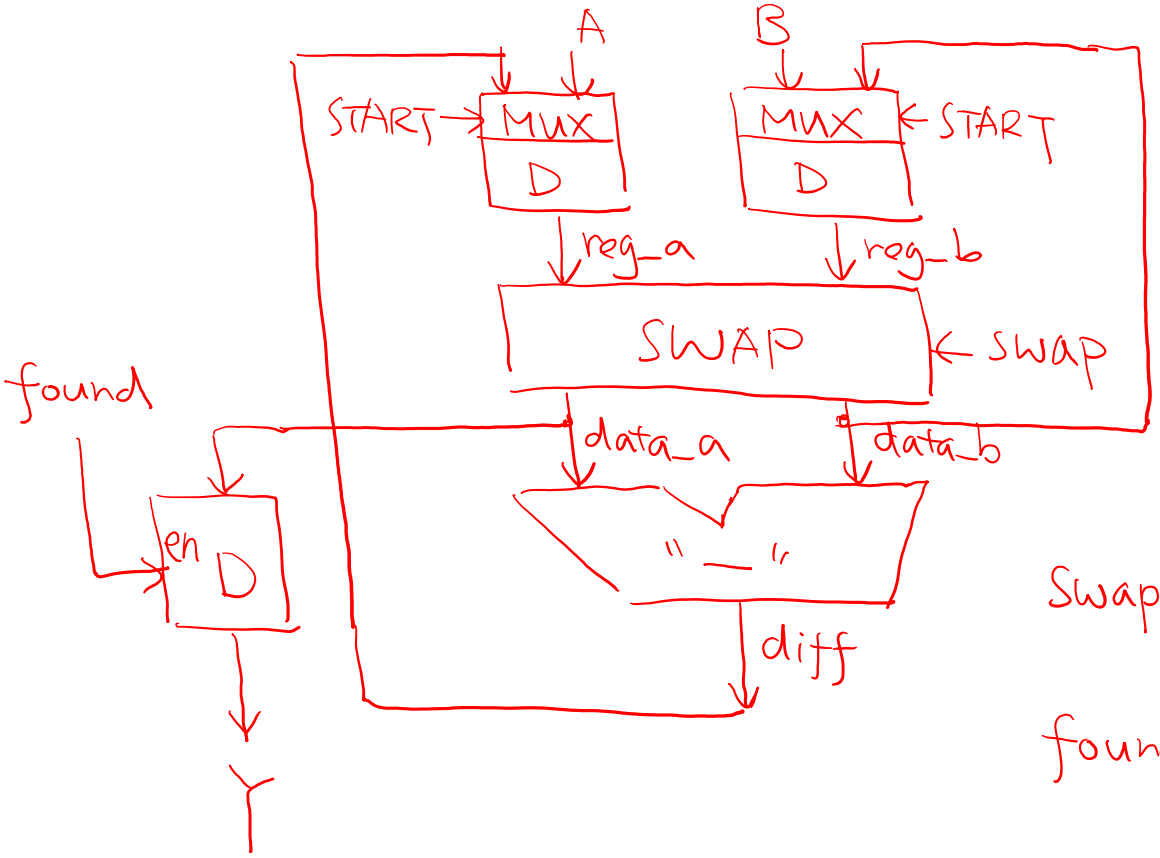
Timing Diagram – Normal Operation



Timing Diagram – Error

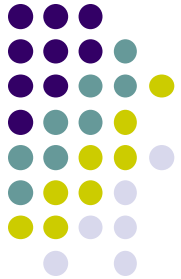


Overall Architecture (Block Diagram)



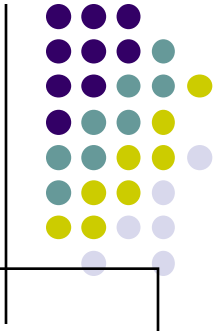

```
Swap = (reg_b > reg_a) ? 1'b1 : 0;

found = (reg_a == reg_b || A == B) ?
        1'b1 : 0;
```

Design Procedure



- Start with spec, primary IOs, and block diagram
- Design the overall architecture
- Specify finite state machine(s), data path, and internal signals
- List the timing diagrams of major operations
- Verilog coding
- Prepare comprehensive test environment for efficient debugging

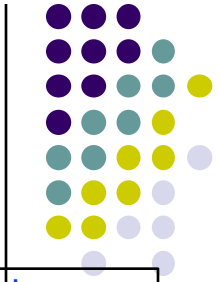
Lab Template: Makefile



```
VLOG        = ncverilog
SRC         = gcd_t.v \
              gcd.v
VLOGARG     = +access+r

TMPFILE     = *.log \
              ncverilog.key \
              nWaveLog \
              INCA_libs

DBFILE      = *.fsdb *.vcd *.bak

RM          = -rm -rf
```

```
all :: sim

sim :
        $(VLOG) $(SRC) $(VLOGARG)

clean :
        $(RM) $(TMPFILE)

veryclean :
        $(RM) $(TMPFILE) $(DBFILE)
```

Lab Template: gcd.v (1/3)

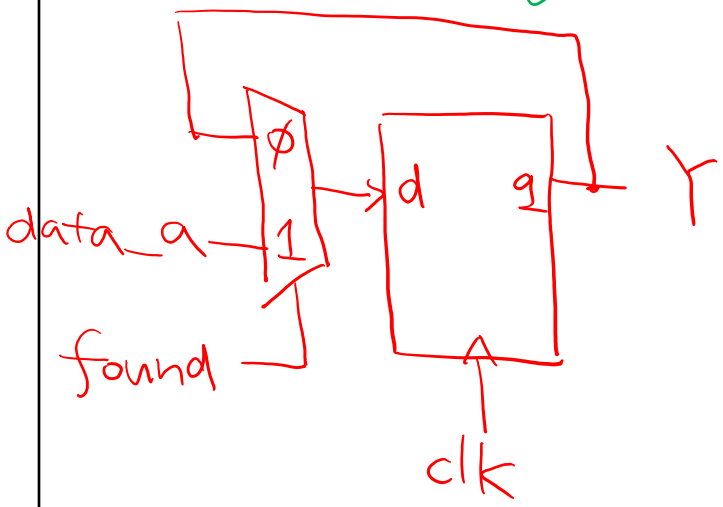


```
module GCD (
  input wire CLK,
  input wire RST_N,
  input wire [7:0] A,
  input wire [7:0] B,
  input wire START,
  output reg [7:0] Y,
  output reg DONE,
  output reg ERROR
);

wire found, err, 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;
```

```
// [lab] define the signal found here
//    assign found =
// [lab] define the signal swap here
//    assign swap =

always @* begin
  if (swap) begin
    data_a = reg_b;
// [lab] define the signal data_b here
//    data_b =
  end else begin
// [lab] finish this block
//    data_a =
//    data_b =
  end
end

always @* begin
// [lab] finish this block
//  diff =
end
```

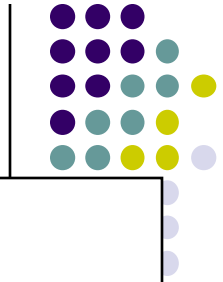
Lab Template: gcd.v (2/3)

```
always @(posedge CLK or negedge RST_N)
begin
// [lab] finish this sequential block
//       to describe Y
//
end
```




```
always @(posedge CLK or negedge RST_N)
begin
  if (~RST_N) begin
    reg_a = 0;
    reg_b = 0;
  end else if (START) begin
// [lab] finish this sequential block
//       to describe reg_a and reg_b
//
  end
end
```

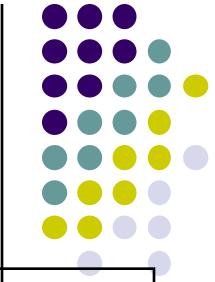
Lab Template: gcd.v (3/3)



```
always @(posedge CLK or negedge RST_N) begin
  if (RST_N == 0) begin
    state <= IDLE;
    ERROR <= 0;
  end else begin
    state <= state_next;
    ERROR <= error_next;
  end
end
// You may revise the coding style
always @* begin
  case (state)
    IDLE: begin
      DONE = 0;
      if (START) begin
        state_next = CALC;
        error_next =
          (A == 0 || B == 0) ? 1'b1 : 0;
      end else begin
        state_next = IDLE;
        error_next = 0;
      end
    end
```

```
    CALC: begin
// [lab] finish this block
//
    end
    FINISH: begin
// [lab] finish this block
//      DONE =
//      error_next =
//      state_next =
    end
    default: begin
      DONE = 0;
      state_next = IDLE;
      error_next = 0;
    end
  endcase
end
endmodule
```

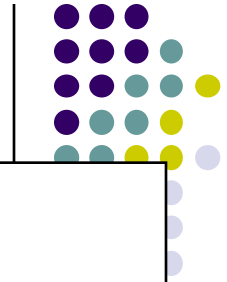
Lab Template: gcd_t.v (1/2)



```
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 (
// [lab] complete the port connections
//    .CLK(...
//    .RST_N(
//    ...
//    ...
  );
```

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

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

    $monitor($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
```

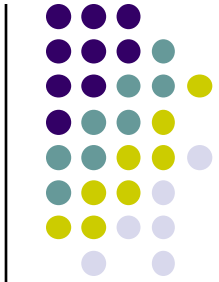
Lab Template: gcd_t.v (2/2)



```
 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'd21, 8'd6);
    #(cyc) nop;
    @(posedge done);

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

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

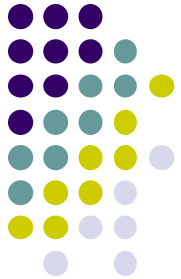
```
  // take a careful look at
  // the usage of task here
  task nop;
    begin
      start = 0;
    end
  endtask
  task load;
    begin
      start = 1;
    end
  endtask
  task data_in;
    input [7:0] data1, data2;
    begin
      a = data1;
      b = data2;
    end
  endtask
endmodule
```

Simulation



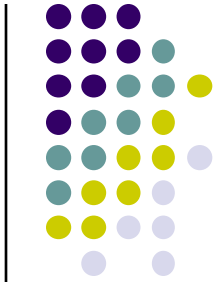
- Syntax checking
  `ncverilog -c gcd.v`
- Verilog simulation
  `ncverilog gcd_t.v gcd.v`
- Using Makefile
  `make`
  `make clean`
  `make veryclean`
- Debugging using waveform viewer
  `nWave`

Guideline of Design Procedure



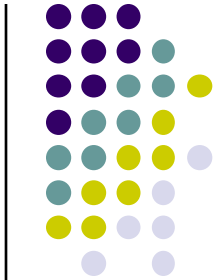
- Prepare the block diagram and FSM carefully
- Try to symbolize every condition for
  - State transitions
  - Mode selection for datapath
- FSM: the simpler the better
  - One single state can take multiple cycles
  - States can be nested
- Design should be done before Verilog coding
- There is no one BEST style for every design
  - State arrangement
  - Naming convention

Lab Requirement (1/2)



- Part A
  - Complete the Verilog RTL design
  - Simulation using Makefile
  - Try to cover as many input conditions as you can to verify the functionality
    - Explain your selection of test patterns
  - Debugging the timing using waveform viewer `nWave`
  - [Optional] Is there any room of improvement regarding to the timing?

Lab Requirement (2/2)



- Part B
  - 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]
    - Now *START* and *DONE* are two-cycle signals
      - Assume they are asserted for two consecutive cycles
      - [Optional] Can you support the control with two separate active cycles?
  - [Optional] Is there any room of improvement regarding to the timing?
- Write a summary report