Design of Digital Circuits and Systems

Algorithms to Hardware II, Timing Review

Instructor: Vikram Iyer

Teaching Assistants:

Ariel Kao

Josh Wentzien

Selim Saridede

Jared Yoder

Derek Thorp

Adapted from material by Justin Hisa

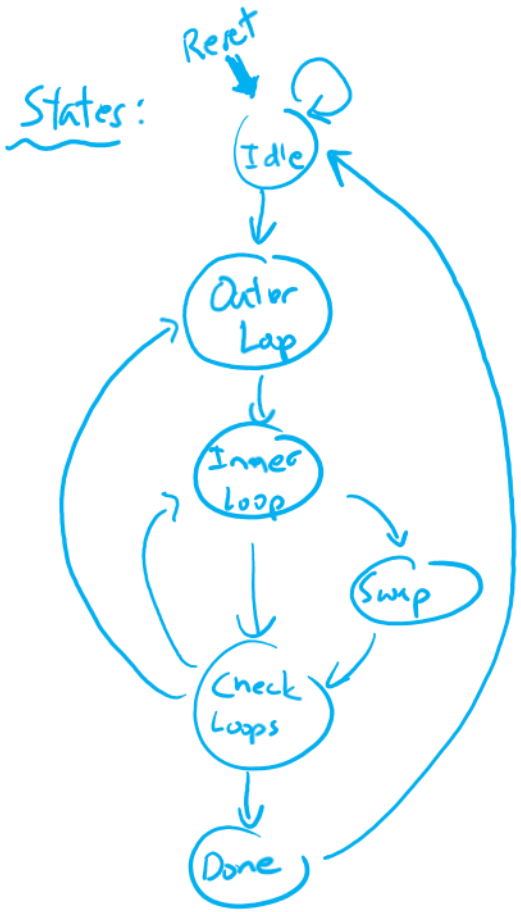
Relevant Course Information

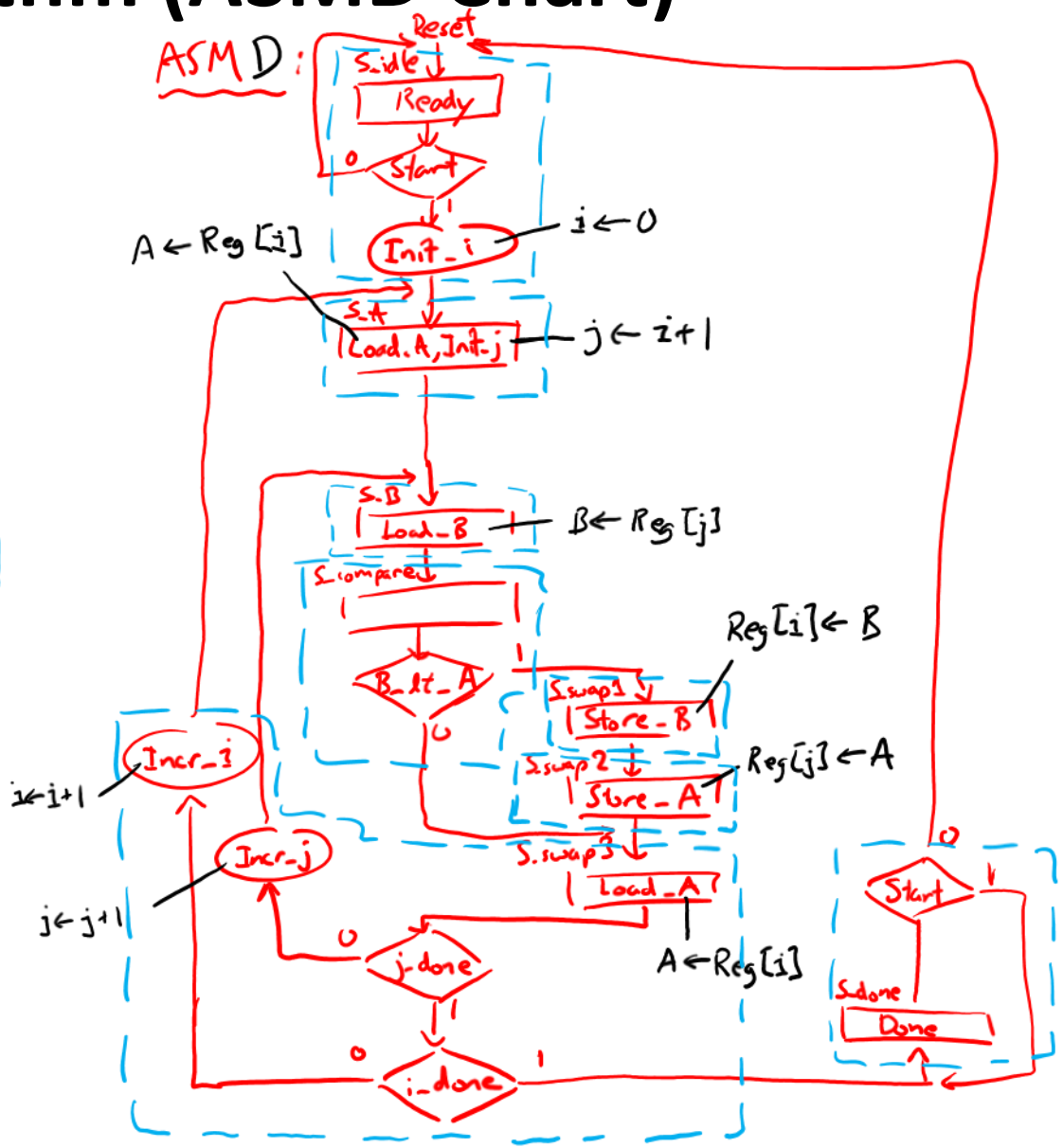
- Anonymous mid-quarter survey on Canvas (due 5/5)
- Homework 4 due on Wednesday (5/7)
- Lab 3 due Friday (5/2)
- Lab 4 due next Friday (5/9)
- Lab 5 will be released next week, due 5/17
- Quiz 3 (ASM, ASMD) next Thursday (5/8)

Outline

- **Algorithm → Hardware Wrap-up**
- Timing Constraints Review

Sorting Algorithm (ASMD Chart)

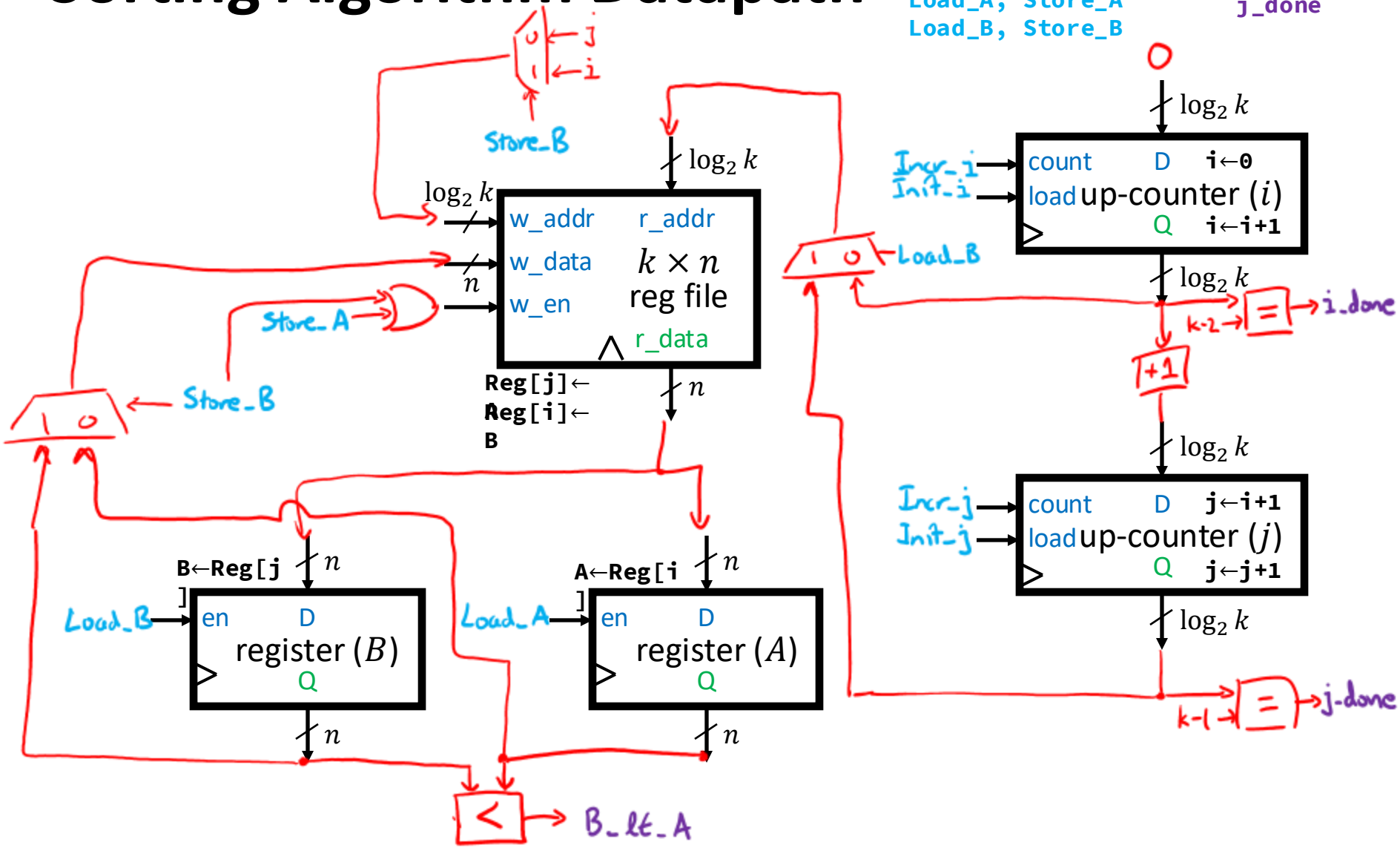




Sorting Algorithm Datapath

Control Signals:
Init_i, Incr_i
Init_j, Incr_j
Load_A, Store_A
Load_B, Store_B

Status Signals:
B_lt_A
i_done
j_done



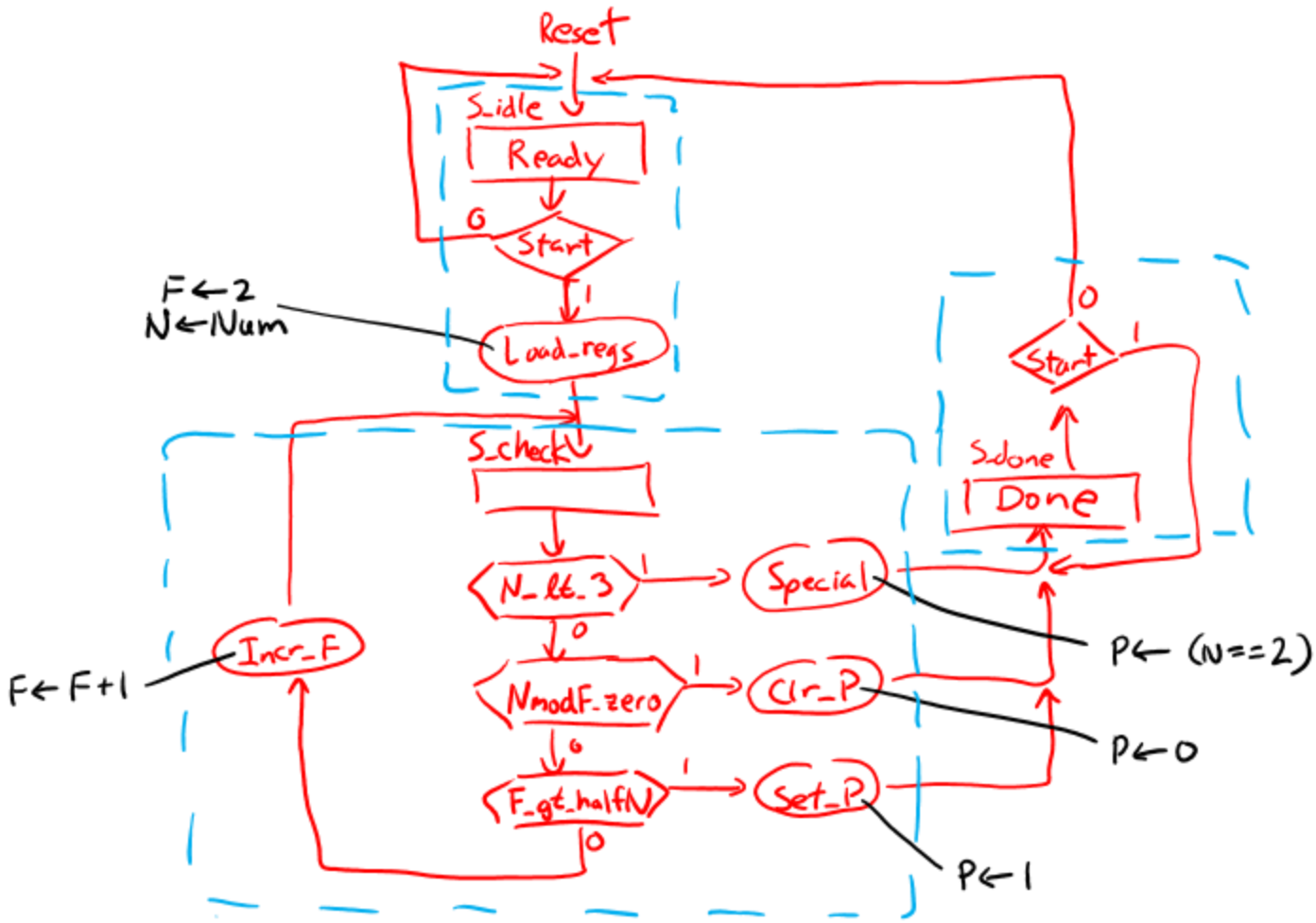
isPrime Algorithm

- Design a sequential circuit that determines if an input Num is a prime number (*e.g.*, 2, 3, 5, 7) or not
  - Num has width W, output P is 1 (prime) or 0 (not prime)
  - Assume the usual Ready, Done, Start, Reset
- Algorithm:

```
N = Num
if N < 3 do              // handle special cases
   return (N == 2)
for F = 2 to N/2 do   // factor to check
   if (N % F == 0) do
      return 0
   endif
endfor
return 1
```

isPrime (ASMD Chart)

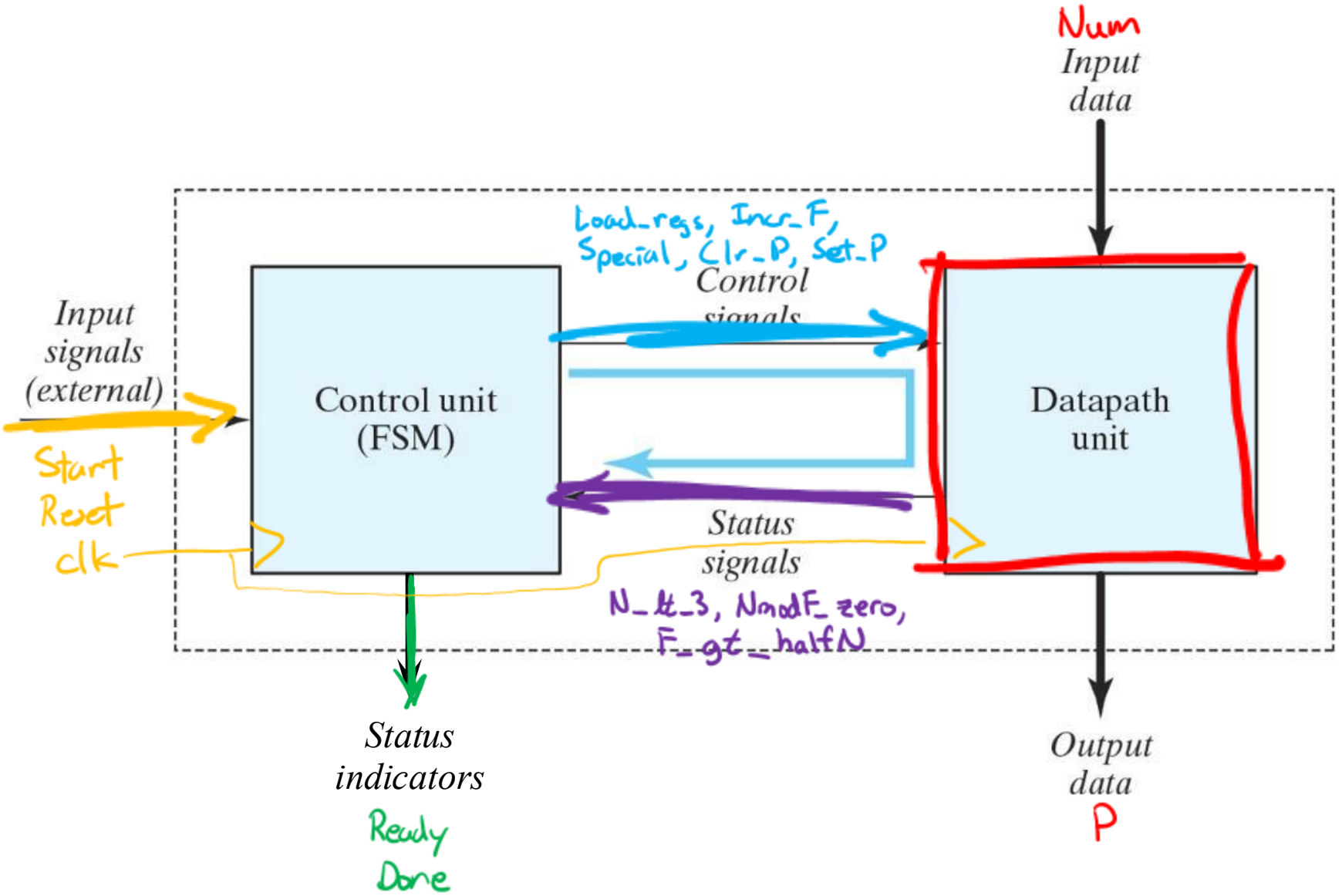
isPrime (ASMD Chart)



Alternatives:

- Incr_F could be Moore output on S_check
- Special could be extra decision box on N=2 that uses Clr_P & Set_P on its outbound paths
- N_lt_3 check could be done in S_idle

isPrime (Block Diagram)



Short Tech Break

isPrime in SystemVerilog

```
isPrime_control:
   // port definitions
   // define state names and variables
   // controller logic w/synchronous reset
   // next state logic
   // output assignments

isPrime_datapath:
   // port definitions
   // internal datapath signals and registers
   // datapath logic
   // output assignments

isPrime:
   // port definitions
   // define status and control signals
   // instantiate control and datapath
```

```
module isPrime_control (
  input  logic clk, Start, Reset,
  output logic Ready, Done,
  input  logic N_lt_3, NmodF_zero, F_gt_halfN,
  output logic Load_regs, Special, Clr_P, Set_P, Incr_F);

  // define state names and variables
  enum {S_idle, S_check, S_done} ps, ns;

  // controller logic w/synchronous reset
  always_ff @(posedge clk)

    if (Reset)
      ps <= S_idle;
    else
      ps <= ns;


  // continued on next slide…
```

```
// CONTROL (continued…)
// next state logic
  always_comb
    case (ps)
      S_idle: ns = Start ? S_check : S_idle;
      S_check: ns = (N_lt_3 | NmodF_zero | F_gt_halfN) ?
                    S_done : S_check;
      S_done: ns = Start ? S_done : S_idle;
    endcase

// output assignments
  assign Ready     = (ps == S_idle);
  assign Done      = (ps == S_done);
  assign Load_regs = (ps == S_idle) & Start;
  assign Special   = (ps == S_check) & N_lt_3;
  assign Clr_P     = (ps == S_check) & ~N_lt_3 &
                     NmodF_zero;
  assign Set_P     = (ps == S_check) & ~N_lt_3 &
                     ~NmodF_zero & F_gt_halfN;
  assign Incr_F    = (ps == S_check) & ~N_lt_3 &
                     ~NmodF_zero & ~F_gt_halfN;

endmodule
```

```
module isPrime_datapath #(parameter W=4)(
  input  logic clk,
  input  logic [W-1:0] Num,
  output logic P,
  output logic N_lt_3, NmodF_zero, F_gt_halfN,
  input  logic Load_regs, Special, Clr_P, Set_P, Incr_F);

  // internal datapath signals and registers
  logic [W-1:0] F;  // current factor
  logic [W-1:0] N;  // copy of Num

  // datapath logic
  always_ff @(posedge clk) begin
    if (Load_regs)  begin
      F <= 2;
      N <= Num;
    end
    if (Incr_F)      F <= F + 1'b1;
    if (Special)     P <= (N == 2);
    if (Clr_P)       P <= 1'b0;
    if (Set_P)       P <= 1'b1;
  end
  // continued on next slide…
```

```
  // DATAPATH (continued…)
  assign N_lt_3             = (N < 3);
  assign NmodF_zero = (N % F == 0);
  assign F_gt_halfN = (F > N/2);
endmodule
```

```
module isPrime #(parameter W=4)(
  input  logic clk,
  input  logic [W-1:0] Num,
  output logic P,
  input  logic Start, Reset,
  output logic Ready, Done);

  // define status and control signals
  logic N_lt_3, NmodF_zero, F_gt_halfN;
  logic Load_regs, Special, Incr_F, Clr_P, Set_P;

  // instantiate control and datapath
  isPrime_control c_unit (.*);
  isPrime_datapath #(W) d_unit (.*);
endmodule  // isPrime
```

Basic Testbench Review

- How to start a testbench:
  1) Create a `<name>_tb` module with no ports
  2) Create variables that match the module's ports
  3) Instantiate an instance of the module can call it `dut` or `uut`
  4) If needed, create a simulated clock
  5) Create an `initial` block to define your test inputs

- Tools we know about:
  - Timing controls: `#<time>;`, `@(posedge <signal>);`
  - Loops:
```
integer i; for (i = 0; i < 2**3; i++)
    repeat (2**4)
```

Testing Your Algorithm

- Create a testbench for our `isPrime` module:

```
module isPrime_tb ();
   // define parameters
   parameter T = 20, W = 4;
   // define module port connections
   logic clk;
   logic [W-1:0] Num;
   logic P;
   logic Start, Reset;
   logic Ready, Done;
   // instantiate module
   isPrime #(W) dut (.*);
   // create simulated clock
   initial begin
     clk <= 0;
     forever #(T/2) clk <= ~clk;
   end  // clock initial
   // define test inputs on next slides…
endmodule
```

Testing Your Algorithm

- Create a testbench for our `isPrime` module:
  - Single `Num` value (arbitrary wait):

```
// define test inputs

initial begin
   Num = 3;
   Reset = 1; Start = 0;  @(posedge clk);
   Reset = 0;             @(posedge clk);
   Start = 1;             @(posedge clk);
   Start = 0;

   repeat (2**4)       // wait 16 clock cycles
      @(posedge clk);

   @(posedge clk);  // extra cycle of output
   $stop();
end
```

Testing Your Algorithm

- Create a testbench for our `isPrime` module:
  - Single Num value (check Ready):

```
// define test inputs

initial begin
    Num = 3;
    Reset = 1; Start = 0;  @(posedge clk);
    Reset = 0;             @(posedge clk);
    Start = 1;             @(posedge clk);
    Start = 0;

    @(posedge Ready);  // wait until module says it's ready


    @(posedge clk);  // extra cycle of output
    $stop();
end
```

Testing Your Algorithm

- Create a testbench for our `isPrime` module:
  - All Num values:

```
// define test inputs
integer i;
initial begin
   Reset = 1; Start = 0;  @(posedge clk);
   Reset = 0;             @(posedge clk);

   for (i = 0; i < 2**W; i++) begin
      Start = 1; Num = i; @(posedge clk);
      Start = 0;          @(posedge Ready);
   end


   @(posedge clk);  // extra cycle of output
   $stop();
end
```

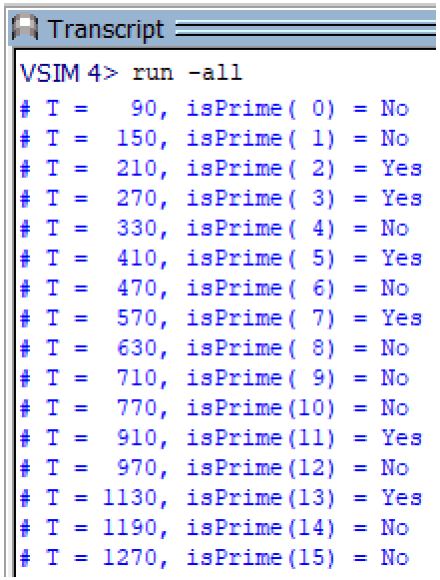
Testing Your Algorithm: $display

- Triggers once when encountered, prints the given format string and adds a new line:

```
// define test inputs
integer i;
initial begin
  Reset = 1; Start = 0;  @(posedge clk);
  Reset = 0;             @(posedge clk);
  for (i = 0; i < 2**W; i++) begin
    Start = 1; Num = i; @(posedge clk);
    Start = 0;          @(posedge Ready);

    $display("T = %4t, isPrime(%2d) = %s",
             $time, Num, P ? "Yes" : "No ");

  end
  @(posedge clk);  // extra cycle of output
  $stop();
end
```



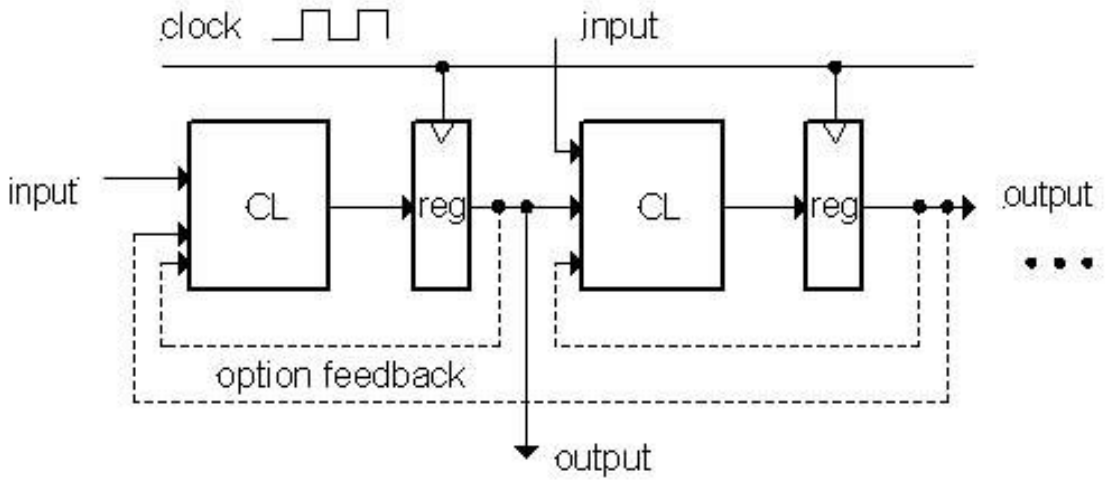
Short Tech Break

Outline

- Algorithm → Hardware Wrap-up
- **Timing Constraints Review**

Why Are Timing Constraints Important?

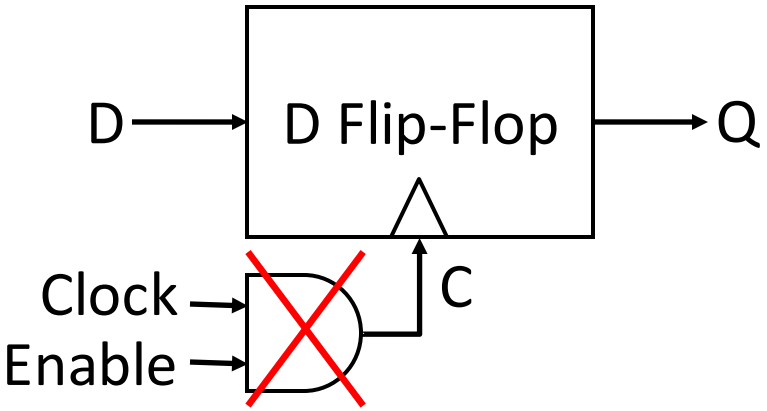
- Our model of synchronous digital systems relies on timing behavior of the clock and logic stages
  - Combinational logic blocks separated by registers



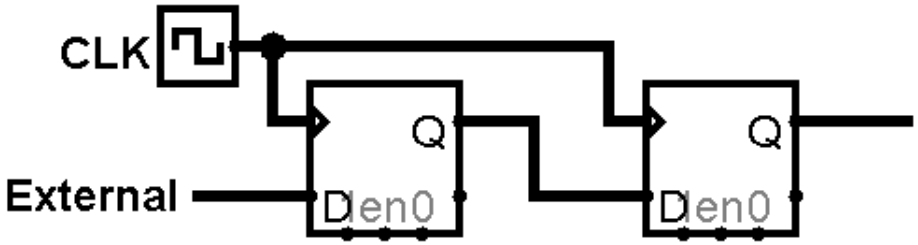
  - Violations can cause incorrect behavior or can cause produced semiconductor circuits to fail!
  - Timing considerations can limit how fast our system/clock can run

Review: Timing Issues

- **NEVER GATE THE CLOCK!!!**
  - Delays can cause part of circuit to get out of sync with rest
    - Adds to *clock skew*
  - Hard to track non-uniform triggers

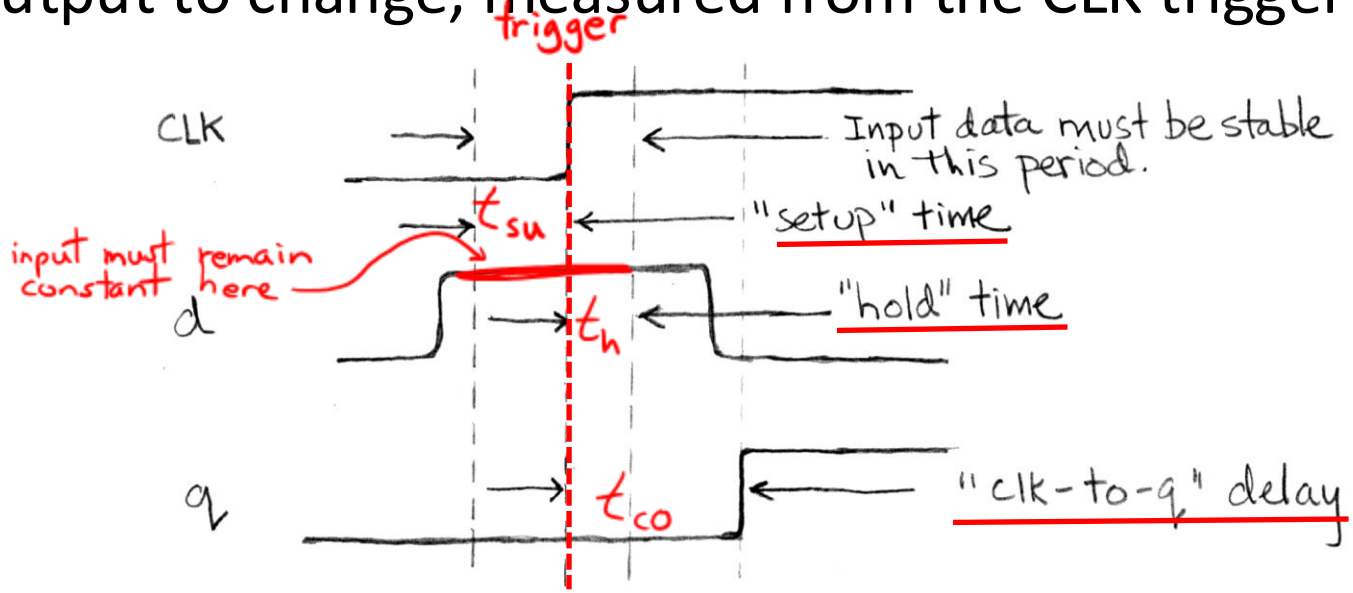


- **Metastability** is the ability of a digital system to persist for an unbounded time in an unstable equilibrium or metastable state
  - Add chains of registers to allow to settle:



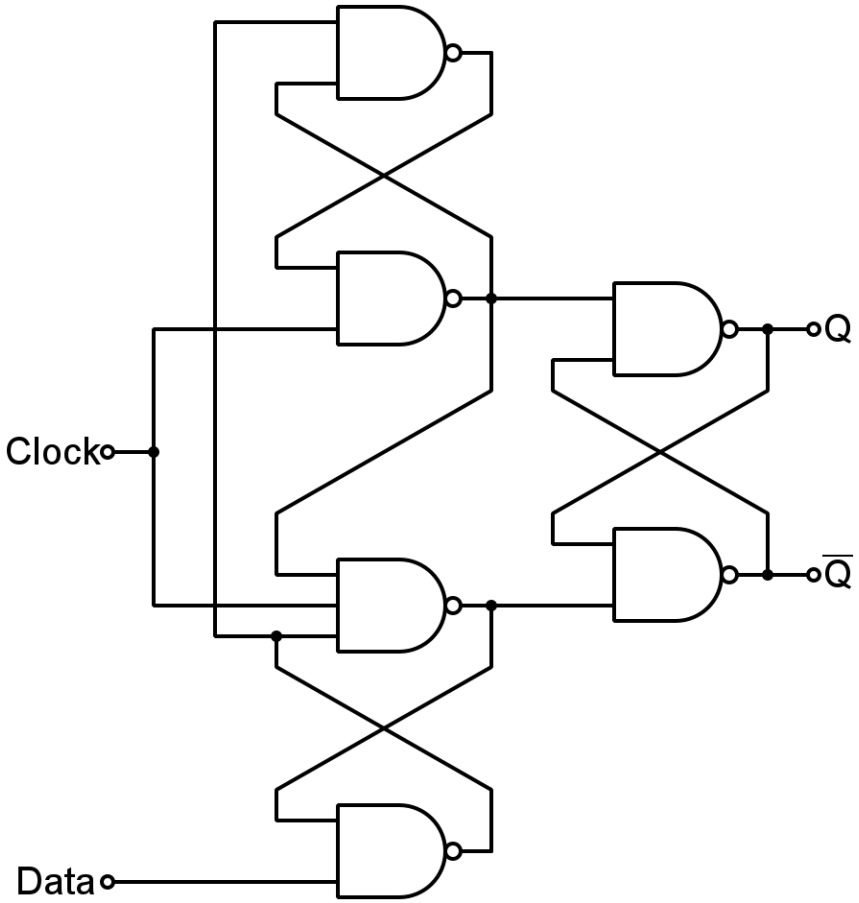
Review: Sequential Timing Constraints

- *Setup Time (t_s or t_{su}):* how long the input must be stable *before* the CLK trigger for proper input read
- *Hold Time (t_h):* how long the input must be stable *after* the CLK trigger for proper input read
- *"CLK-to-Q" Delay (t_{C2Q} or t_{CO}):* how long it takes the output to change, measured from the CLK trigger



Where Do Timing Terms Come From?

Edge-triggered
D flip-flop:



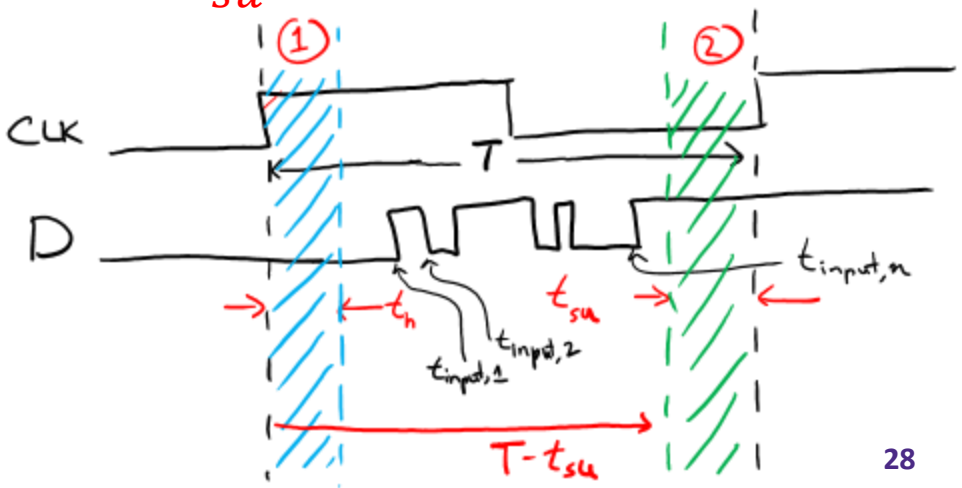
By Nolanjshettle at English Wikipedia, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=40852354

When Can the Input Change?

- When a register input changes shouldn't violate hold time (t_h) or setup time (t_{su}) constraints within a clock period (T)
- Let $t_{input,i}$ be the time it takes for the input of a register to change for the i-th time in a single clock cycle, measured from the CLK trigger:
  - Then we need $t_h \overset{①}{\leq} t_{input,i} \overset{②}{\leq} T - t_{su}$ for all i
  - Two separate constraints!

① $t_{input,1} \geq t_h$

② $t_{input,n} \leq T - t_{su}$

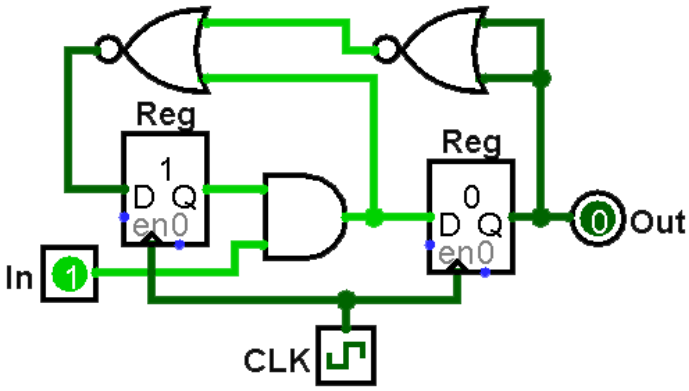


Timing Constraint Considerations

- Use *worst-case analysis*
  - Asynchronous inputs are *really* problematic
  - Use the critical path – the longest delay between *any* two registers in a circuit – for setup time constraint
  - Use the shortest path for the hold time constraint
- A timing violation could be caused by a signal change from the previous clock cycle
  - A really late input change could violate the hold time constraint in the next clock cycle
- Setup time constraint helps determine feasibility of clock frequency: max freq = 1/(min period)

Timing Constraint Example

- $t_{su} = 12$ ns, $t_h = 18$ ns, $t_{CO} = 8$ ns, $t_{AND} = 25$ ns, and $t_{NOR} = 20$ ns



- If In changes t_{CO} after each clock trigger, what is the **minimum clock period** we can use?

- If we use the minimum clock period, when within a clock period can In change without causing a timing violation?