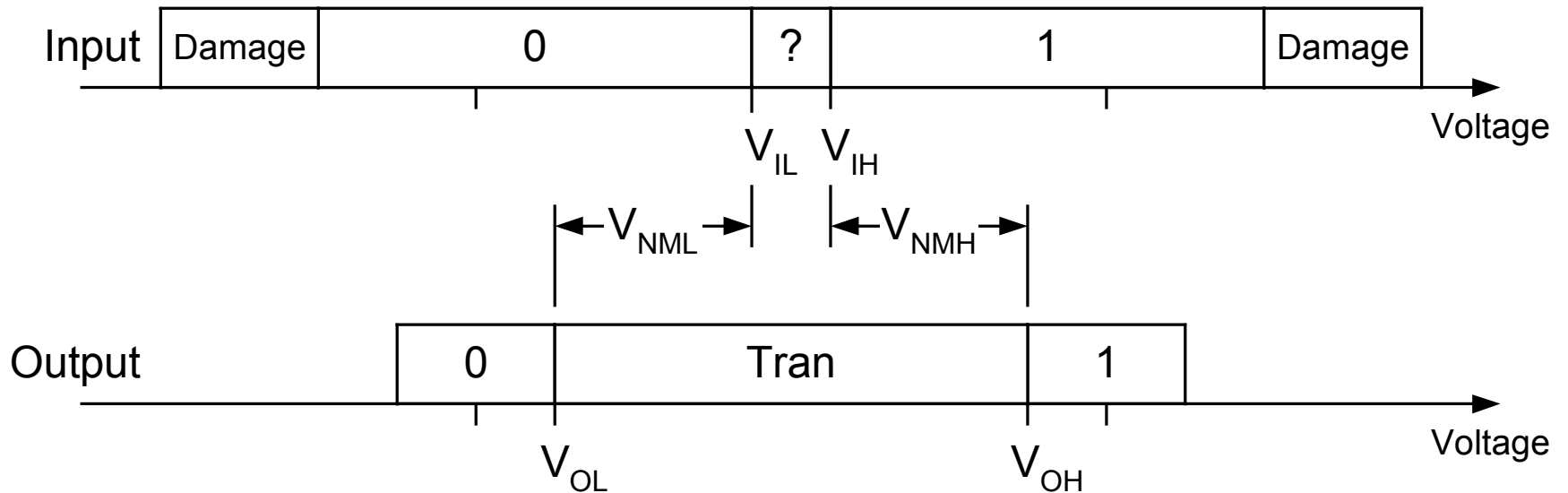

Digital Design: A Systems Approach

Lecture 17: Overview and Wrapup

Restoring logic gives *noise margins*



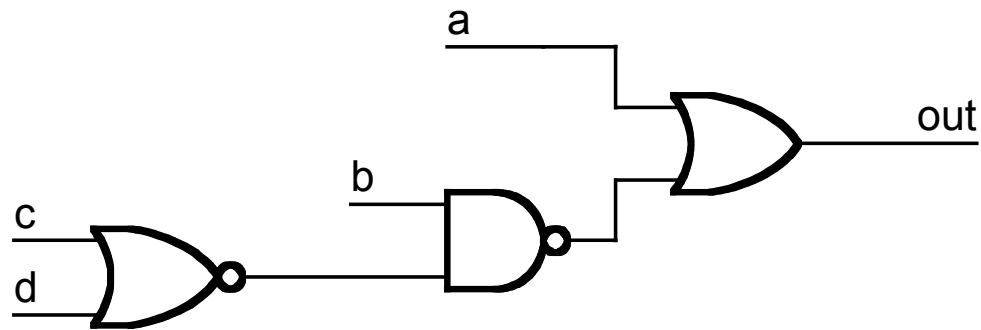
Represent values with bits

- Truth values, e.g., door is open
 - Single bit
- Numbers, temperature (range, precision)
 - Integers, fixed-point, floating-point
- Sets, e.g., colors
 - Encoded one-hot or binary
- Compound
 - Includes several of above

Combinational Logic

- We compose gates into combinational logic circuits

Output depends only on present value of inputs

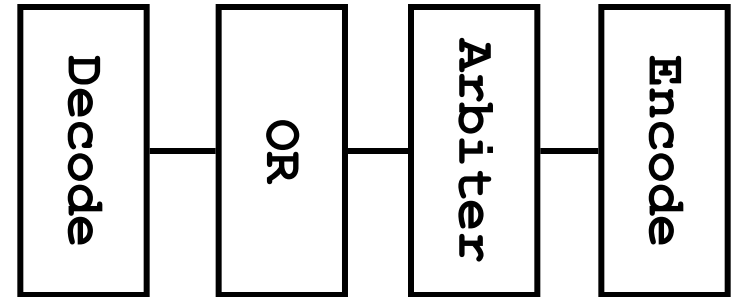


This circuit realizes the function $f =$ _____

Why don't we realize this function with a single gate?

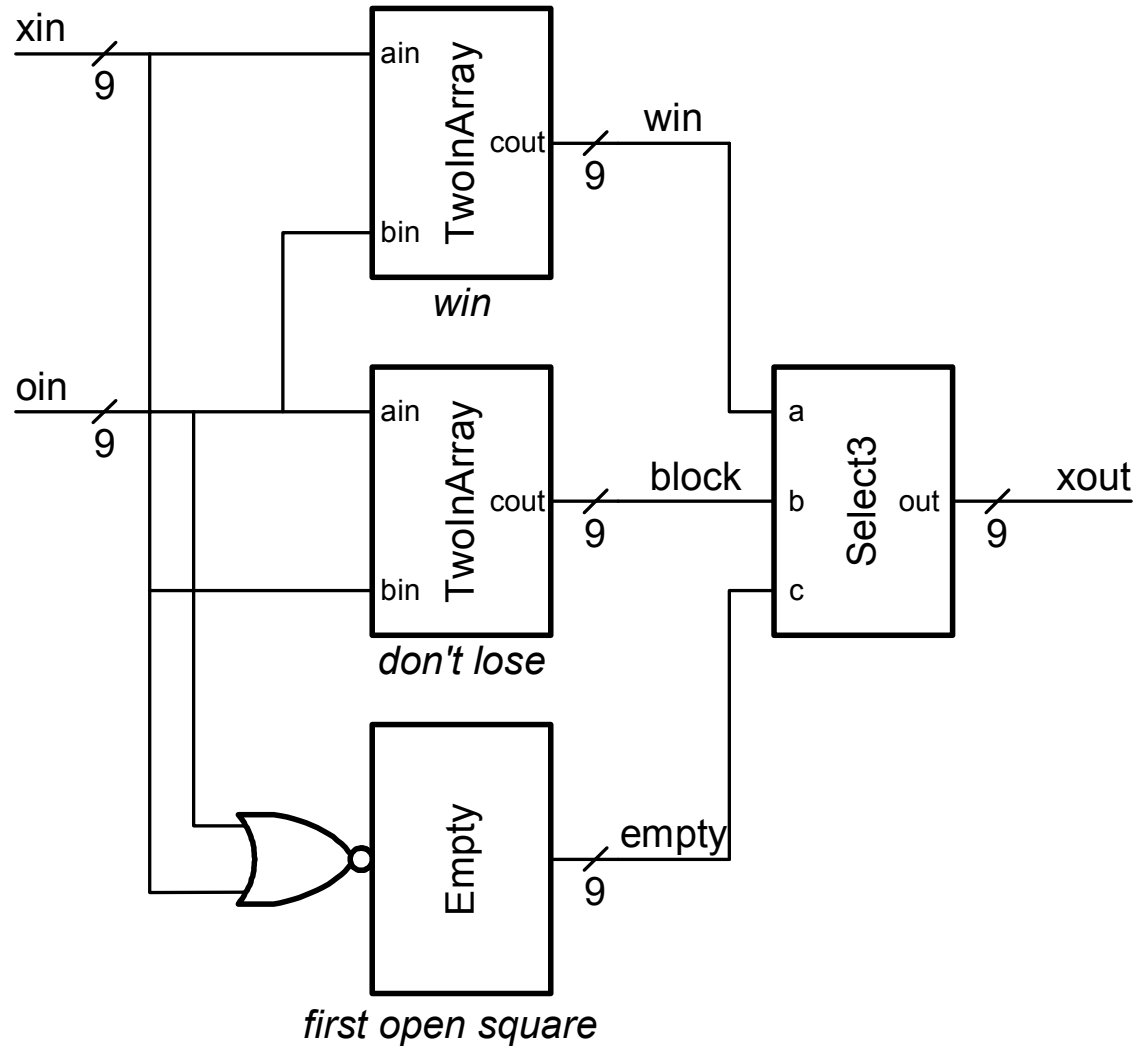
Functions built from decoders and other blocks

- Example – find maximum of 256 small 4-bit numbers
 - Decode each number 4→16 bits (256 times)
 - OR these together w/16 256-bit ORs
 - (each a 4-level tree of 4-input ORs)
 - Use an Arbiter to get highest of 16
 - Encode the 16→4

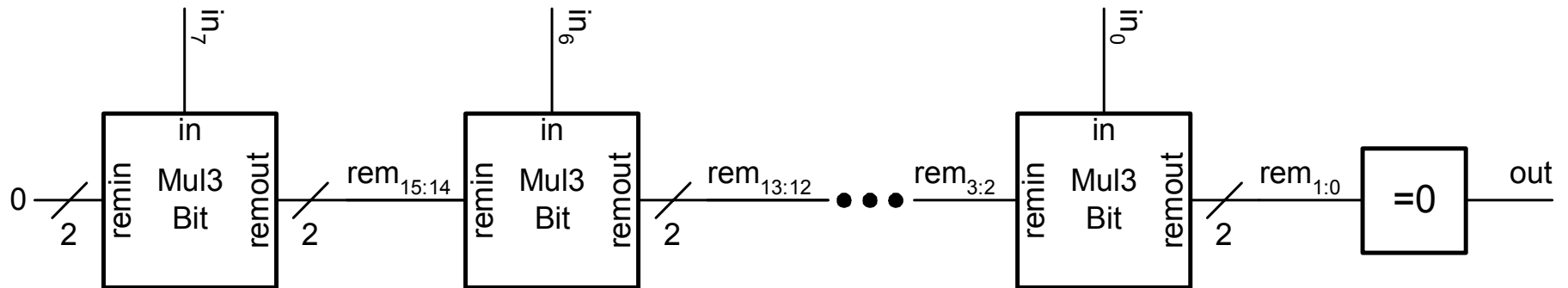


- Need to see if this is more efficient than a tournament
- Can this determine which input has the ‘winning’ number?

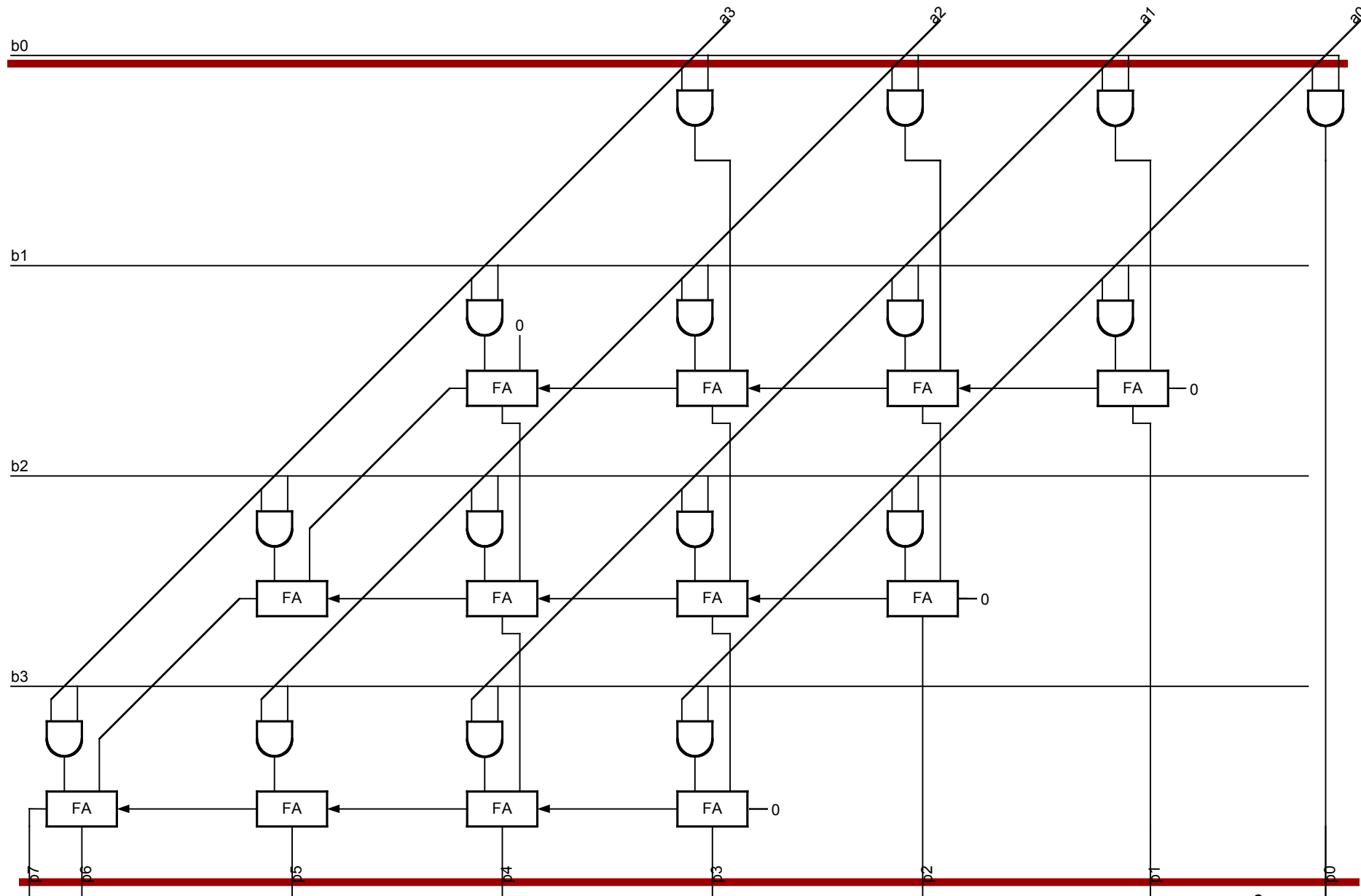
Building Blocks



Iterative Circuits



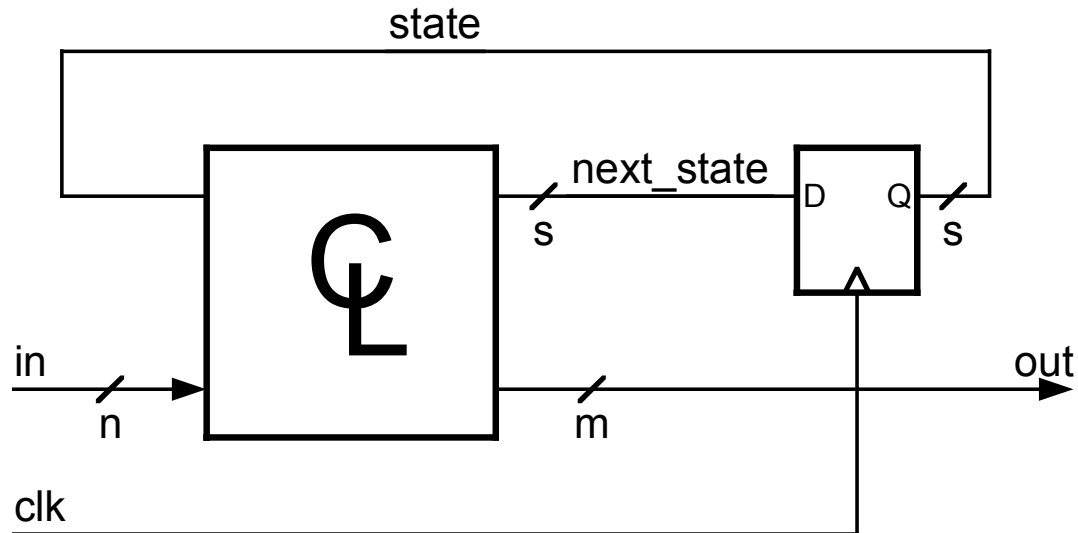
Arithmetic



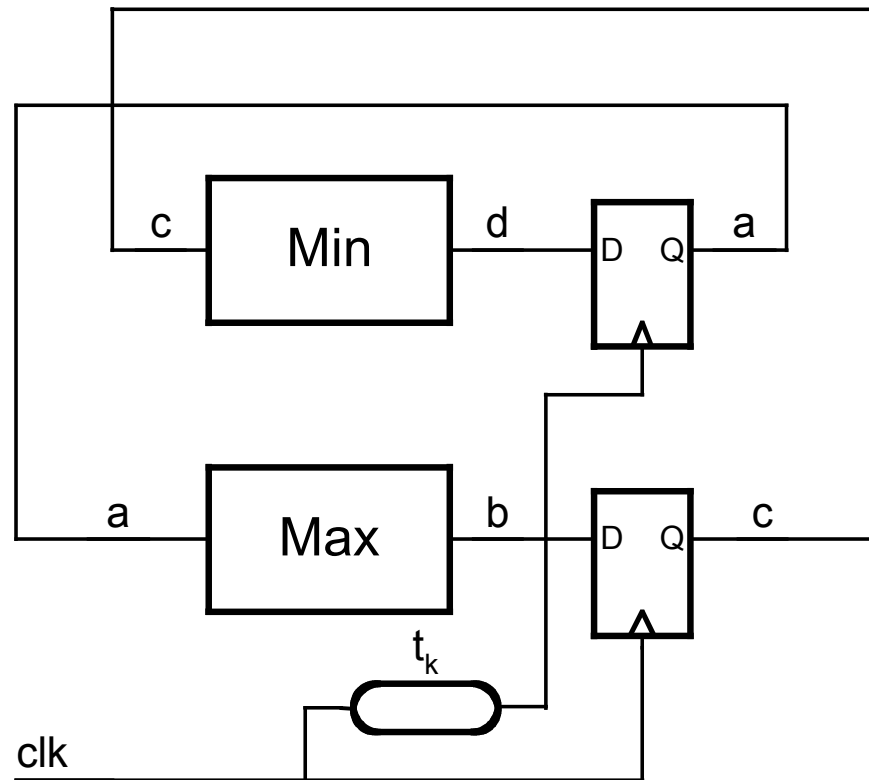
Sequential Logic

- Sequential logic circuits have *state*.

Next state and outputs are a function of inputs and present state.



Sequential circuits work properly if setup and hold time constraints are met

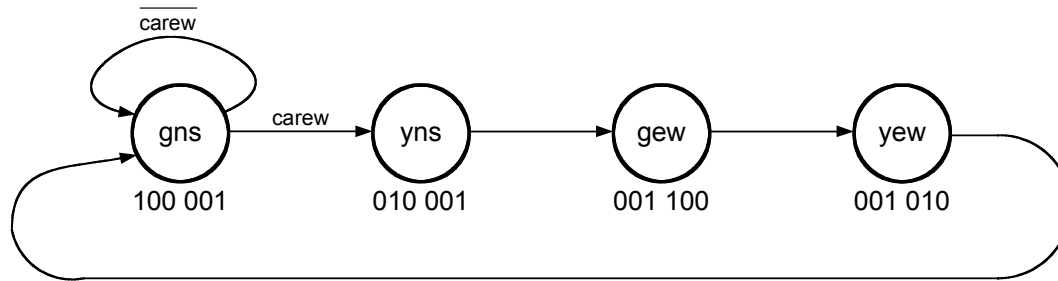


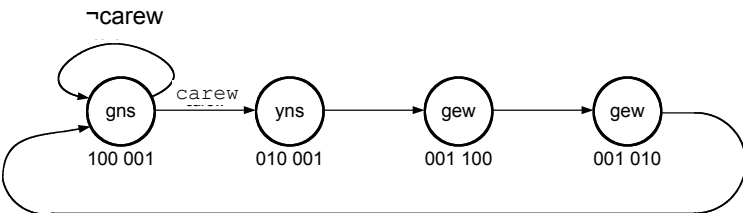
Suppose $t_{dCQ} = t_s = t_h = 100\text{ps}$, $t_k = 200\text{ps}$ or -200ps ,
 $t_{\min} = 50\text{ps}$, $t_{\max} = 2\text{ns}$.
Is hold time met? What is minimum t_{cy} ?

Finite-State Machines

- Finite-state machines are described by a next-state and output function

Can be described with a state diagram or state table.





```

//-----
// FSM Example for Lecture 7
// Bill Dally 1/30/03
//-----
// define state assignment - one hot
//-----
`define SWIDTH 4
`define GNS 4'b1000
`define YNS 4'b0100
`define GEW 4'b0010
`define YEW 4'b0001
//-----
// define output codes
//-----
`define GNSL 6'b100001
`define YNSL 6'b010001
`define GEWL 6'b001100
`define YEWL 6'b001010
//-----
// define flip-flop
//-----
module DFF(clk, in, out) ;
    parameter n = 1; // width
    input clk ;
    input [n-1:0] in ;
    output [n-1:0] out ;
    reg [n-1:0] out ;

    always @(posedge clk)
        out = in ;
endmodule

```

```

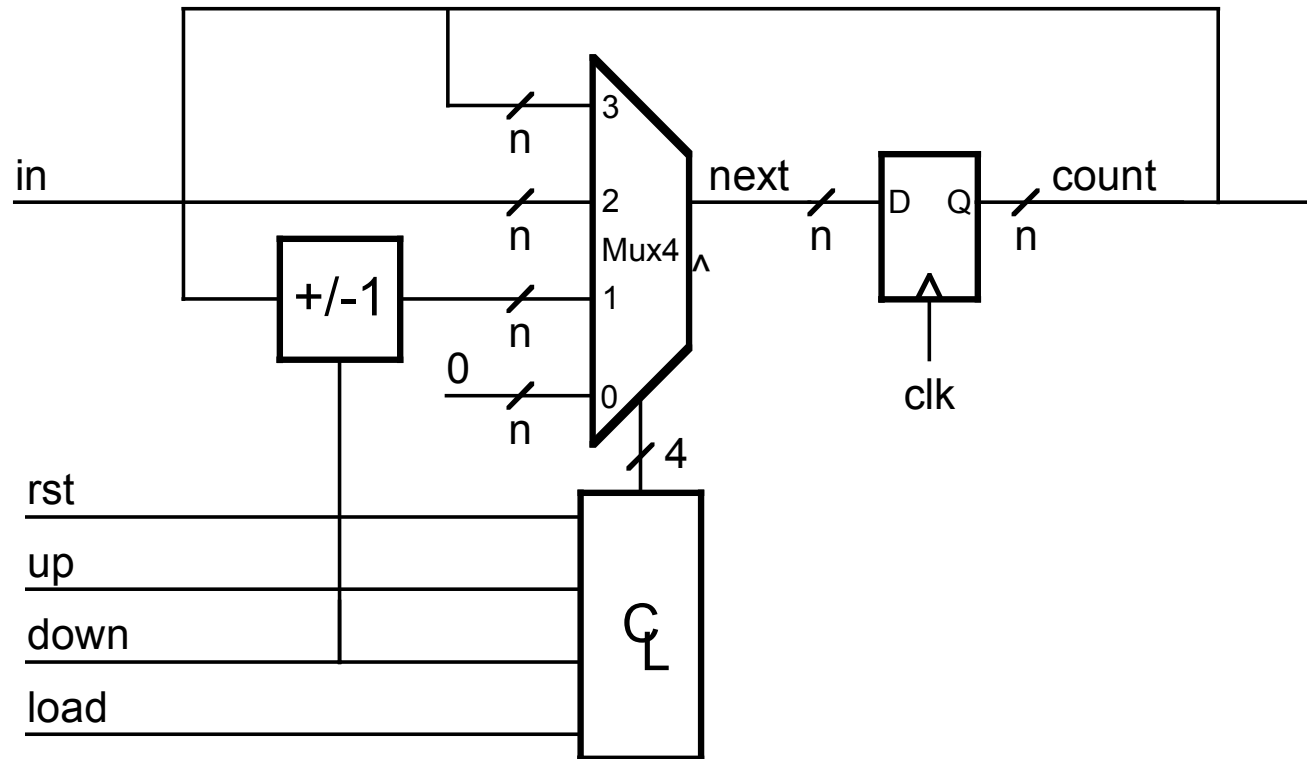
//-----
// Traffic_Light
// Inputs:
//   clk - system clock
//   rst - reset - high true
//   carew - car east/west - true when car is waiting in east-west direction
// Outputs:
//   lights - (6 bits) {gns, yns, rns, gew, yew, rew}
// Waits in state GNS until carew is true, then sequences YNS, GEW, YEW
// and back to GNS.
//-----
module Traffic_Light(clk, rst, carew, lights) ;
    input clk ;
    input rst ; // reset
    input carew ; // car present on east-west road
    output [5:0] lights ; // {gns, yns, rns, gew, yew, rew}
    wire [SWIDTH-1:0] state, next ; // current and next state
    reg [SWIDTH-1:0] next1 ; // next state w/o reset
    reg [5:0] lights ; // output - six lights 1=on

    // instantiate state register
    DFF #(SWIDTH) state_reg(clk, next, state) ;

    // next state and output equations - this is combinational logic
    always @(state or carew) begin
        case(state)
            `GNS: {next1, lights} = {(carew ? `YNS : `GNS), `GNSL} ;
            `YNS: {next1, lights} = `{GEW, `YNSL} ;
            `GEW: {next1, lights} = `{YEW, `GEWL} ;
            `YEW: {next1, lights} = `{GNS, `YEWL} ;
        endcase
    end
    // add reset
    assign next = rst ? `GNS : next1 ;
endmodule

```

Data paths are more easily described by realizing the next state function from *building blocks*

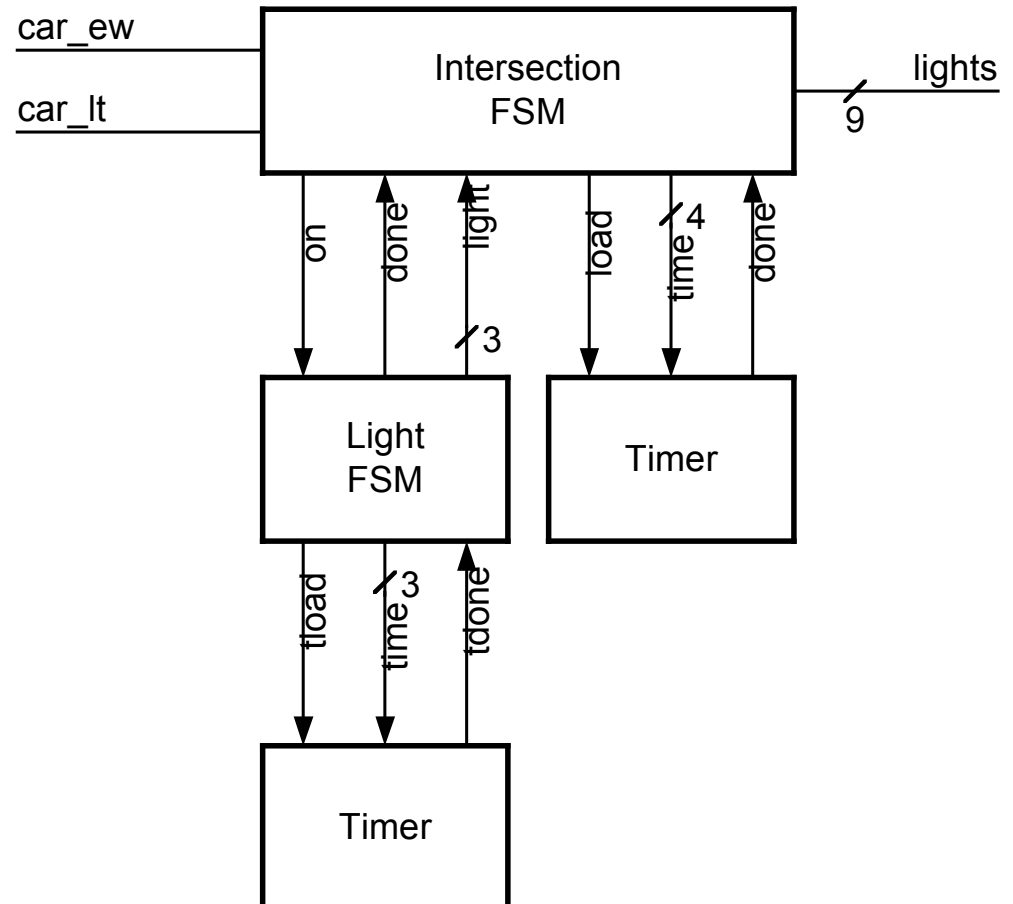


```
module Timer(clk, rst, load, in, done) ;  
    parameter n=4 ;  
    input clk, rst, load ;  
    input [n-1:0] in ;  
    output done ;  
    wire [n-1:0] count, next_count ;  
    wire done ;  
  
    DFF #(n) cnt(clk, next_count, count) ;  
  
    always@(rst, load, in, out) begin  
        casex({rst, load, done})  
            3'b1xx: next_count = 0 ; // reset  
            3'b001: next_count = 0 ; // done  
            3'b01x: next_count = in ; // load  
            default: next_count = count-1'b1; // count down  
        endcase  
    end  
  
    assign done = (count == 0) ;  
endmodule
```

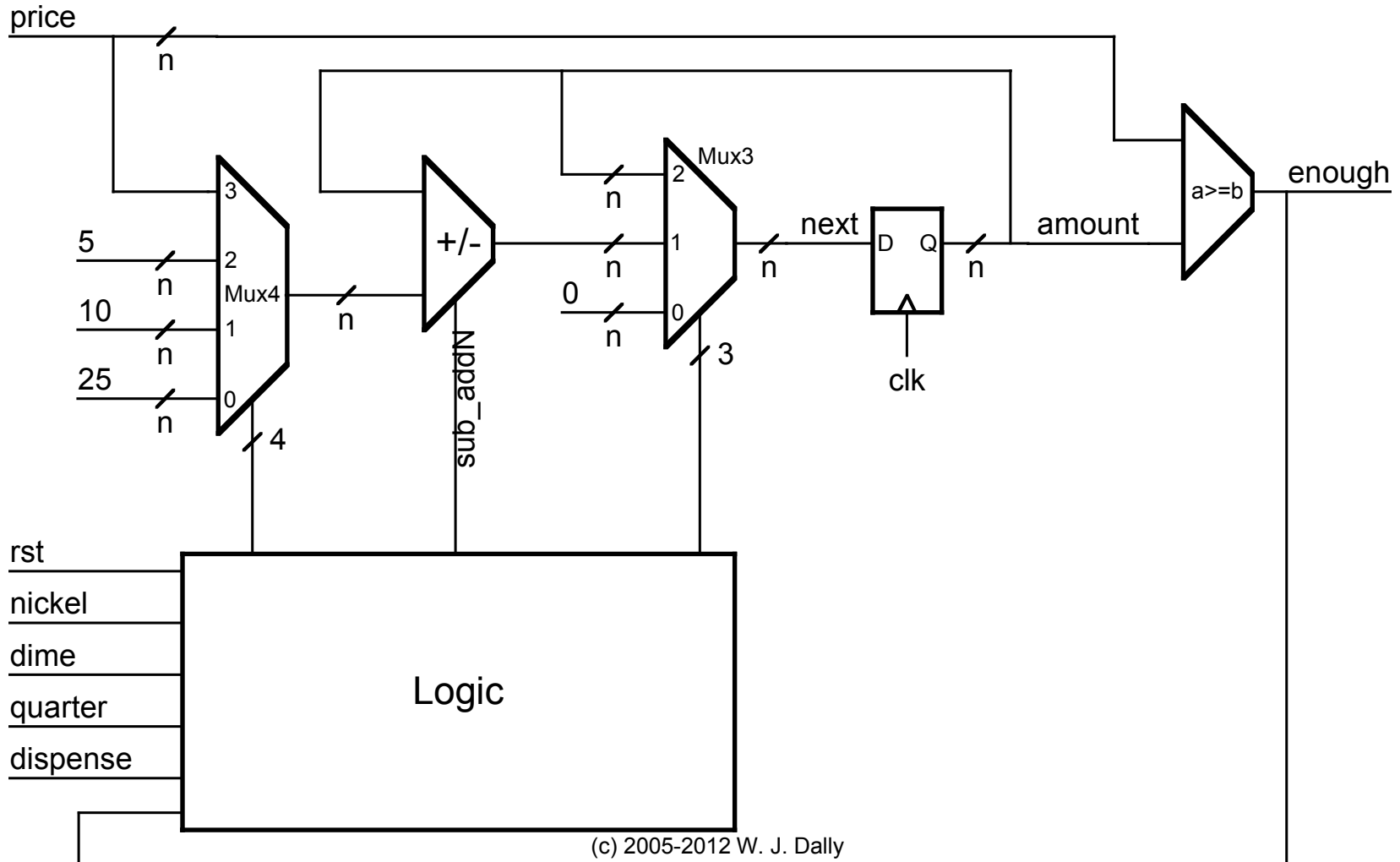
Factoring

- Factor Machines to reduce complexity

Divide and conquer

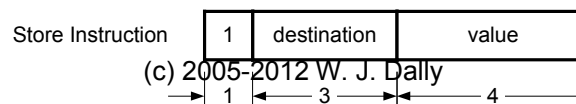
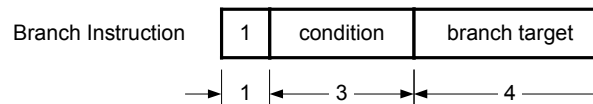
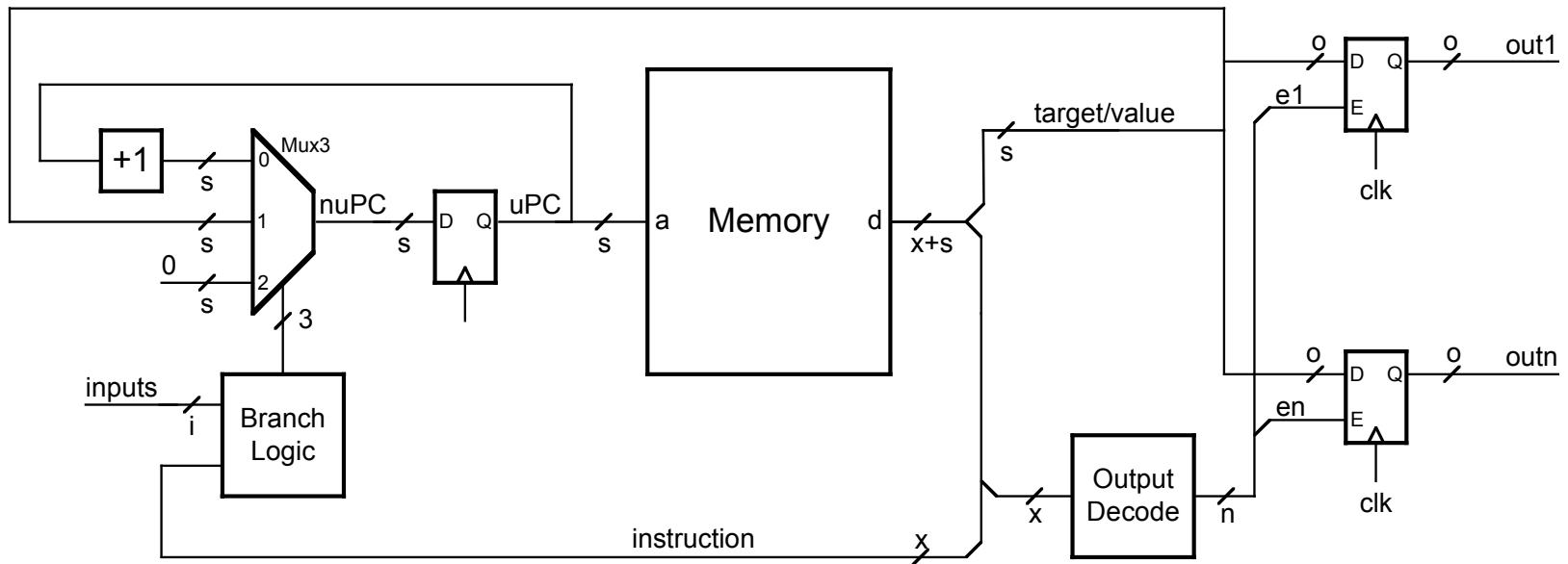


Most FSMs are a combination of a *data path* realized from building blocks, and a *controller* designed from a state diagram.



Microcode

- Microcode, realizing a FSM with a memory - a programmable FSM
- Compress the size of the memory by encoding control and output instructions

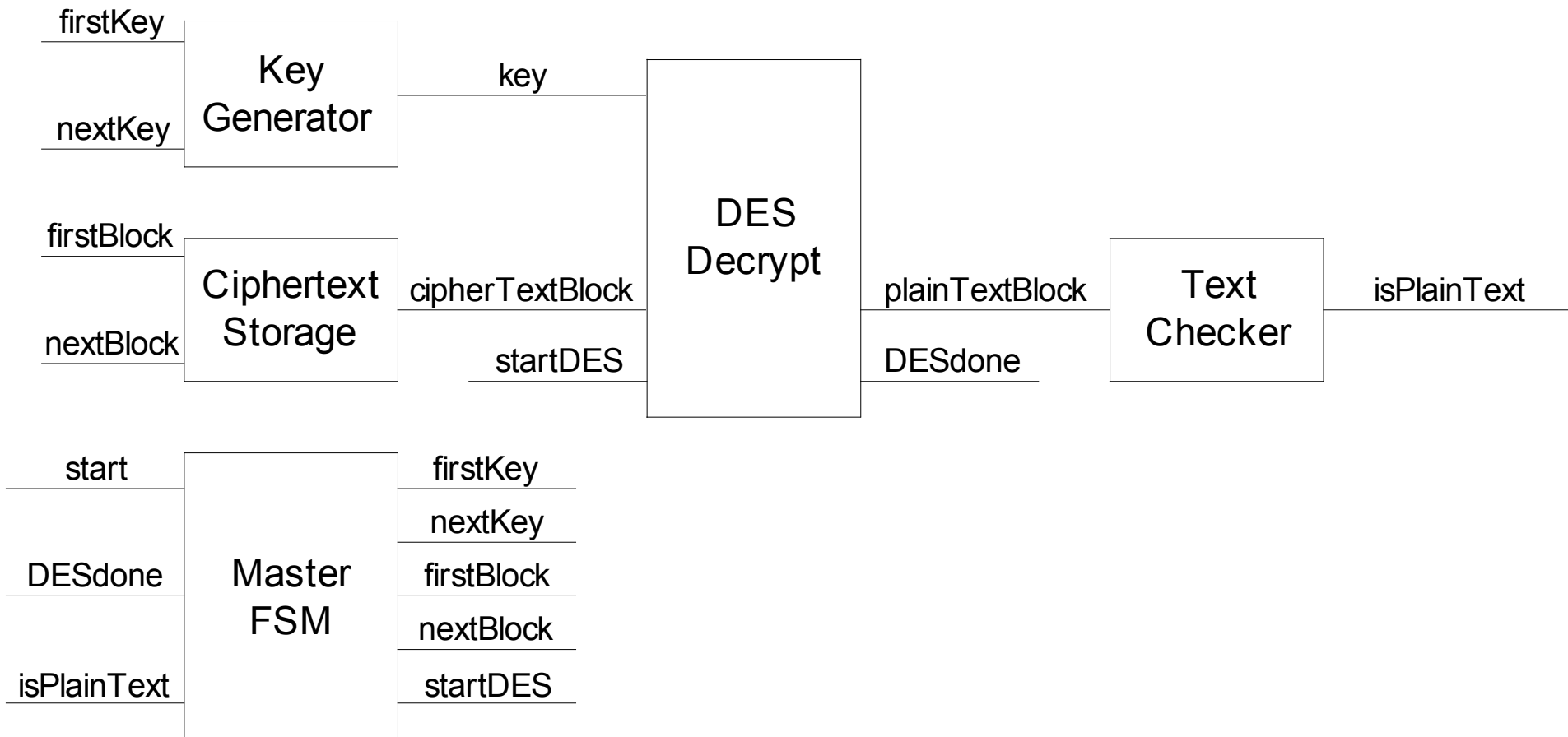


System Design – a process

- Specification
 - Understand what you need to build
- Divide and conquer
 - Break it down into manageable pieces
- Define interfaces
 - Clearly specify every signal between pieces
 - Hide implementation
 - Choose representations
- Timing and sequencing
 - Overall timing – use a table
 - Timing of each interface – use a simple convention (e.g., valid – ready)
- Add parallelism as needed (pipeline or duplicate units)
- Timing and sequencing (of parallel structures)
- Design each module
- Code
- Verify

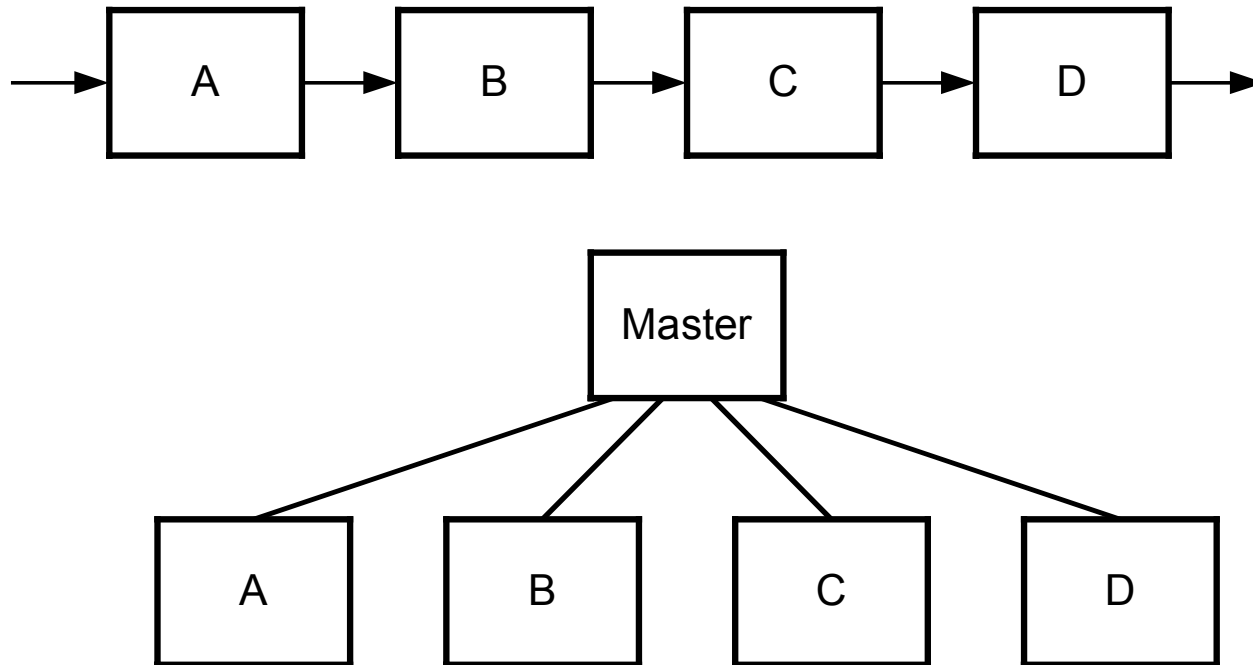
Iterate back to the top at any step as needed.

DES Example



Pipelining

- Modules are composed in *pipelines* and *parallel* configurations
- *Throughput* and *latency*

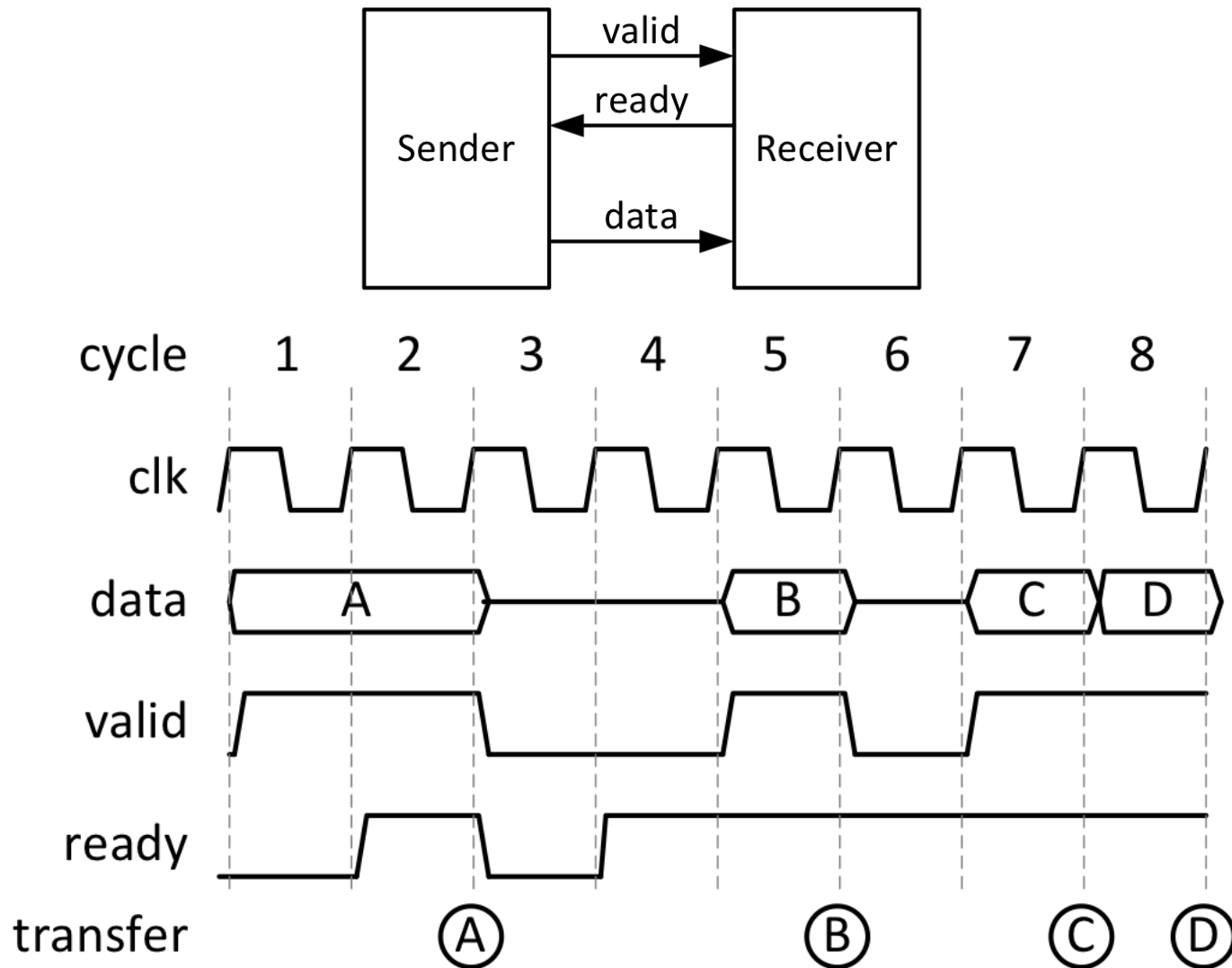


How does a typical *pipeline handshake* work?

What signals are used between stages?

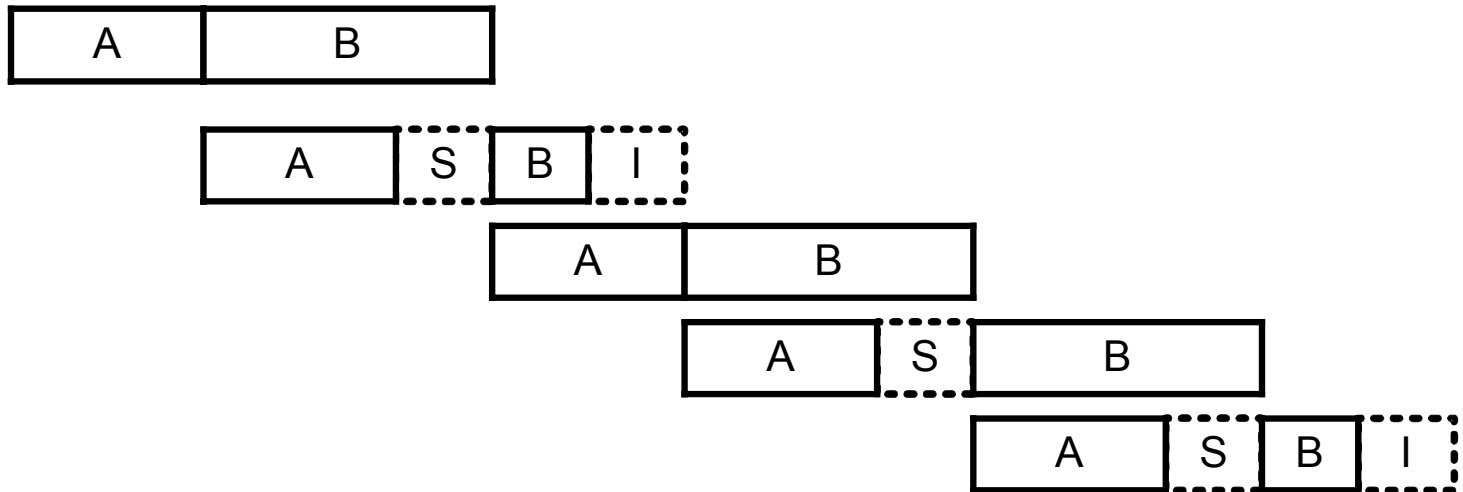
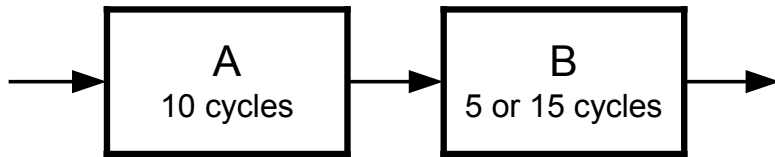
What values must these signals have for data to move from one stage to the next?

Flow Control

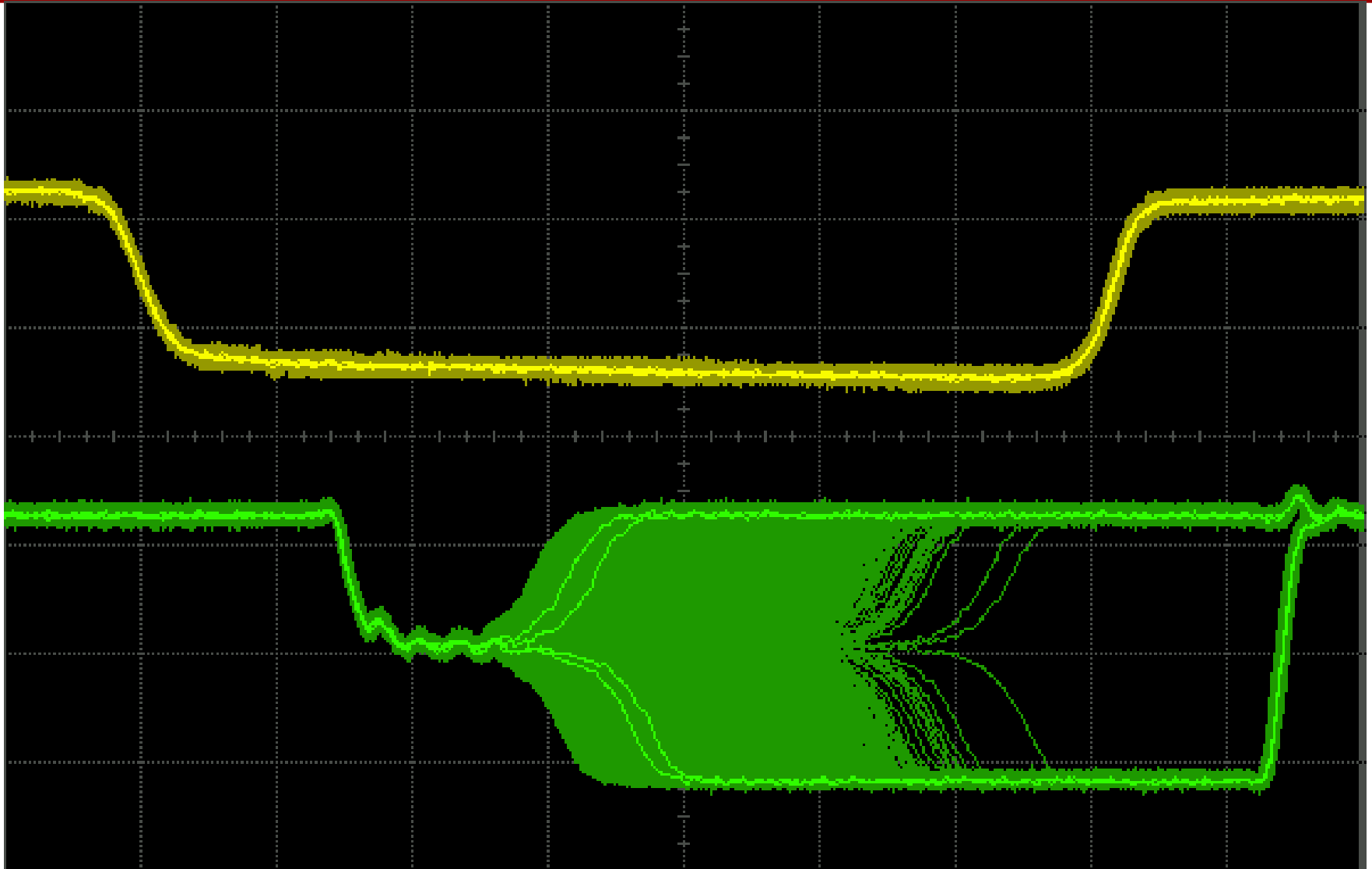


Pipelines

- Pipelines can stall and idle
- When do these happen? How can you prevent them



Synchronization Failure

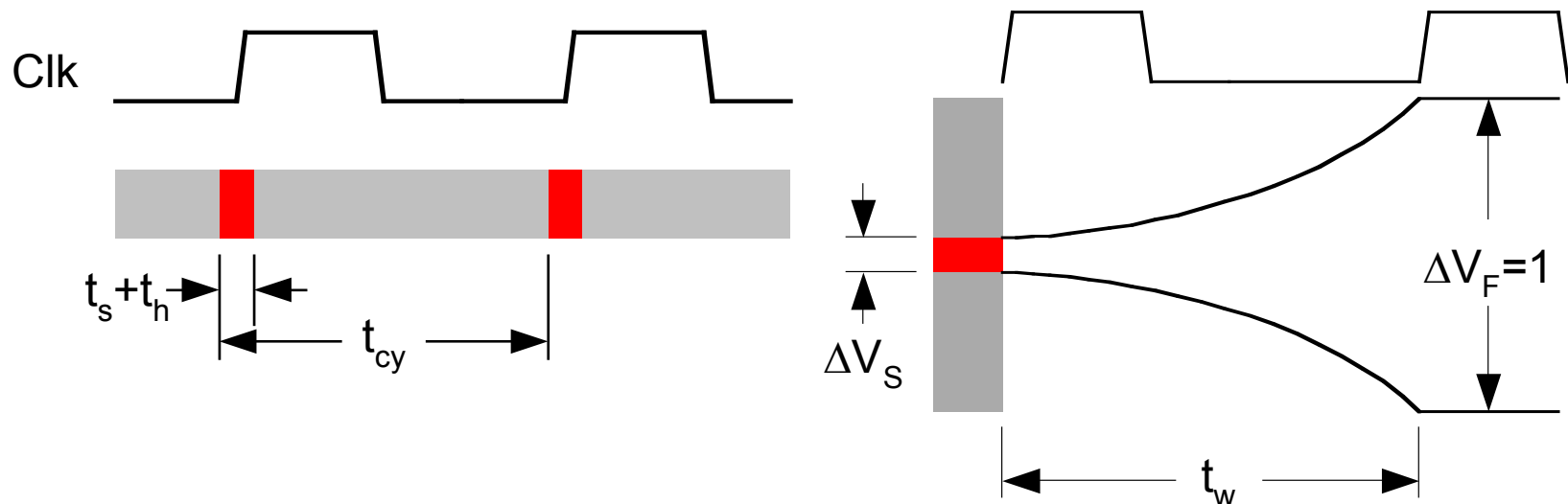


Failure Probability and Error Rate

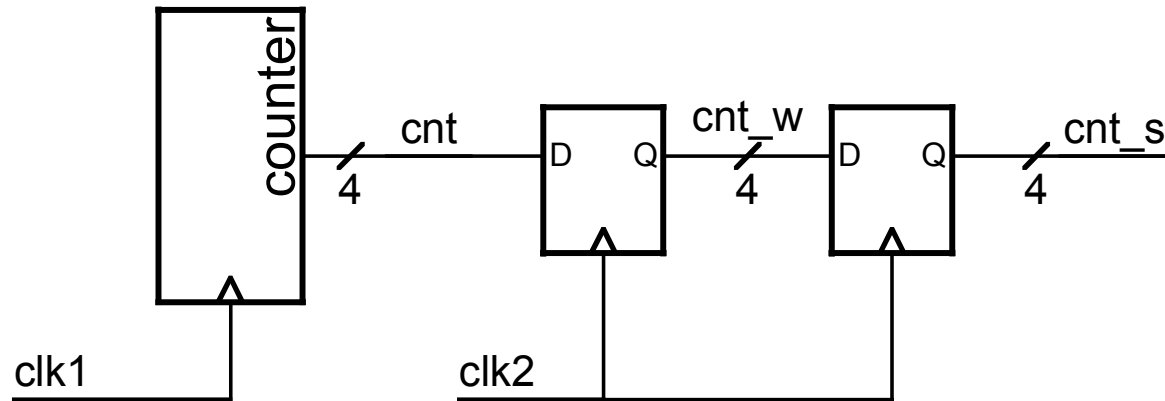
- Each event can potentially fail.
- Failure rate = event rate x failure probability

$$P_F = P_E P_S = (t_s + t_h) f_{cy} \exp\left[-\frac{t_w}{\tau}\right]$$

$$f_F = f_e P_F = (t_s + t_h) f_e f_{cy} \exp\left[-\frac{t_w}{\tau}\right]$$



What is wrong with this picture?

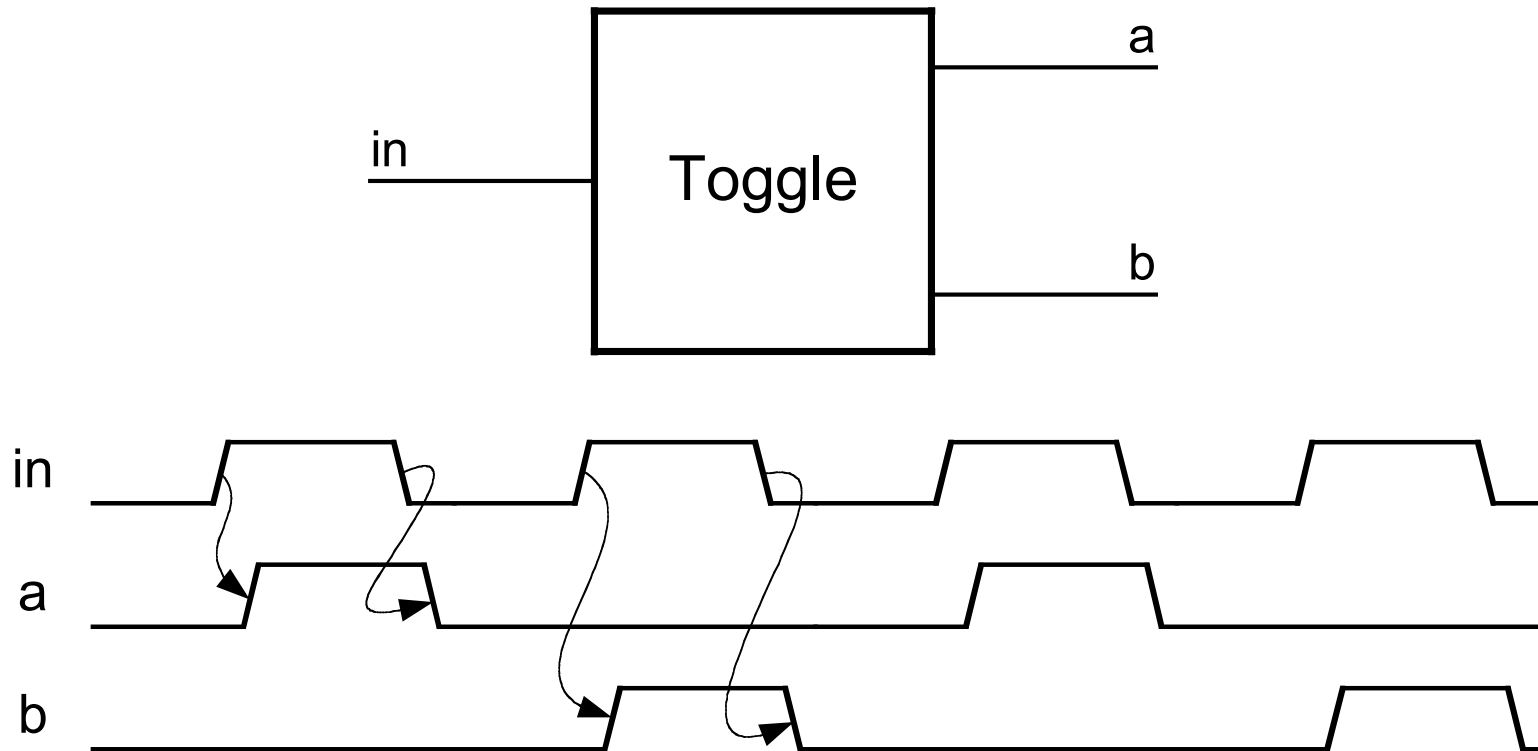


Asynchronous logic design

Continuous state feedback

Affected by races and hazards

Synthesize/Analyze with flow tables



Congratulations! You are now a logic designer.

- To become a better one:
 - Practice, practice, practice...
 - Study other designs
 - Become a student of the art of digital design
 - Stay on top of the latest technology
 - New parts, processes, tools, techniques
 - Read the *trade press*
 - Take more courses
 - Go to conferences
 - Build a “network” (of people)
 - Continue learning
 - What you have now is a “license to learn”
- It's a fun field
 - Design lots of neat things – chips, boards, systems
 - Play with fun tools, processes, chips, lab equipment
 - Meet fun people (mostly)
 - Make \$\$