
Digital Design: A Systems Approach

Lecture 14: Sequential Logic Review

Readings

- L14: Sequential Logic Review
- L15: 22, 24, & 25

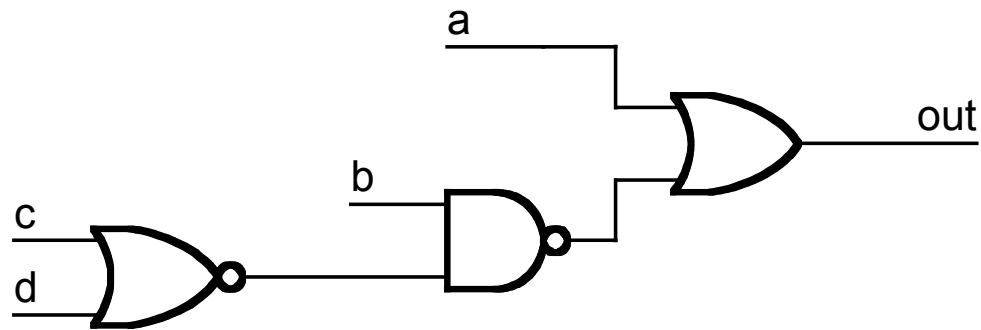
Outline

- Combinational vs. sequential logic
- Classic FSMs
- Datapath FSMs
- Microcode
- System decomposition
- Timing, concurrency and pipelines

Combinational Logic

- We compose gates into combinational logic circuits

Output depends only on present value of inputs

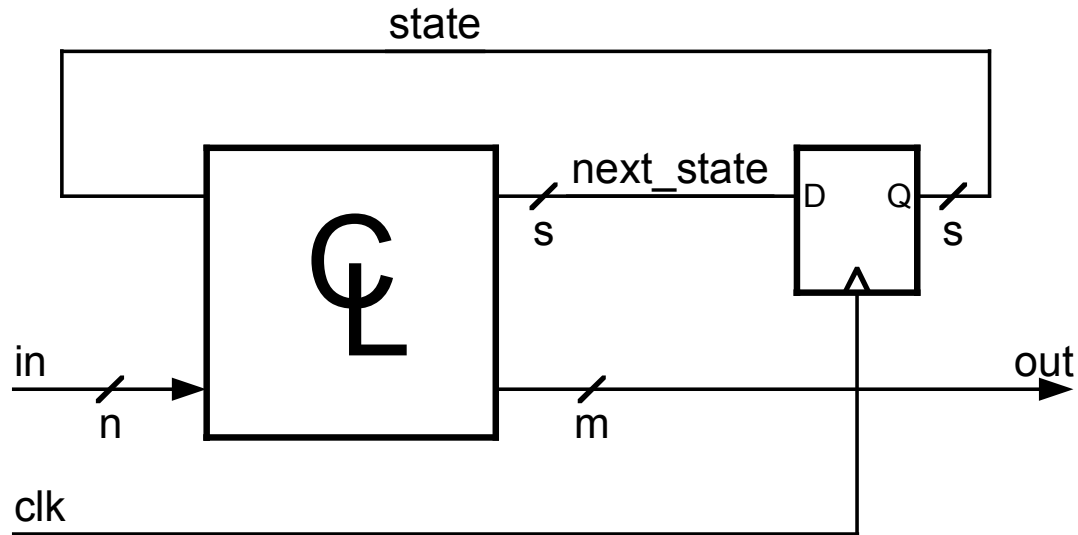


This circuit realizes the function $f =$ _____

Sequential Logic

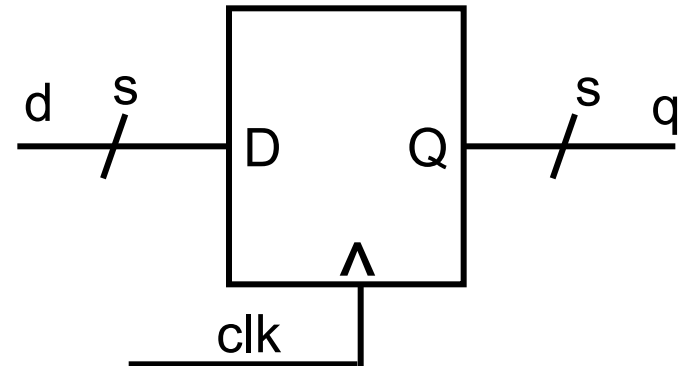
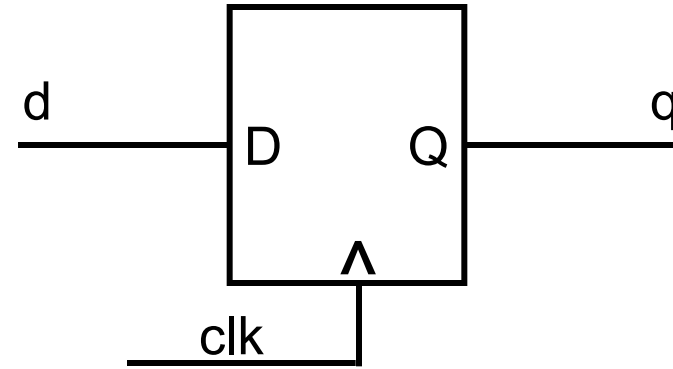
- Sequential logic circuits have *state*.

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

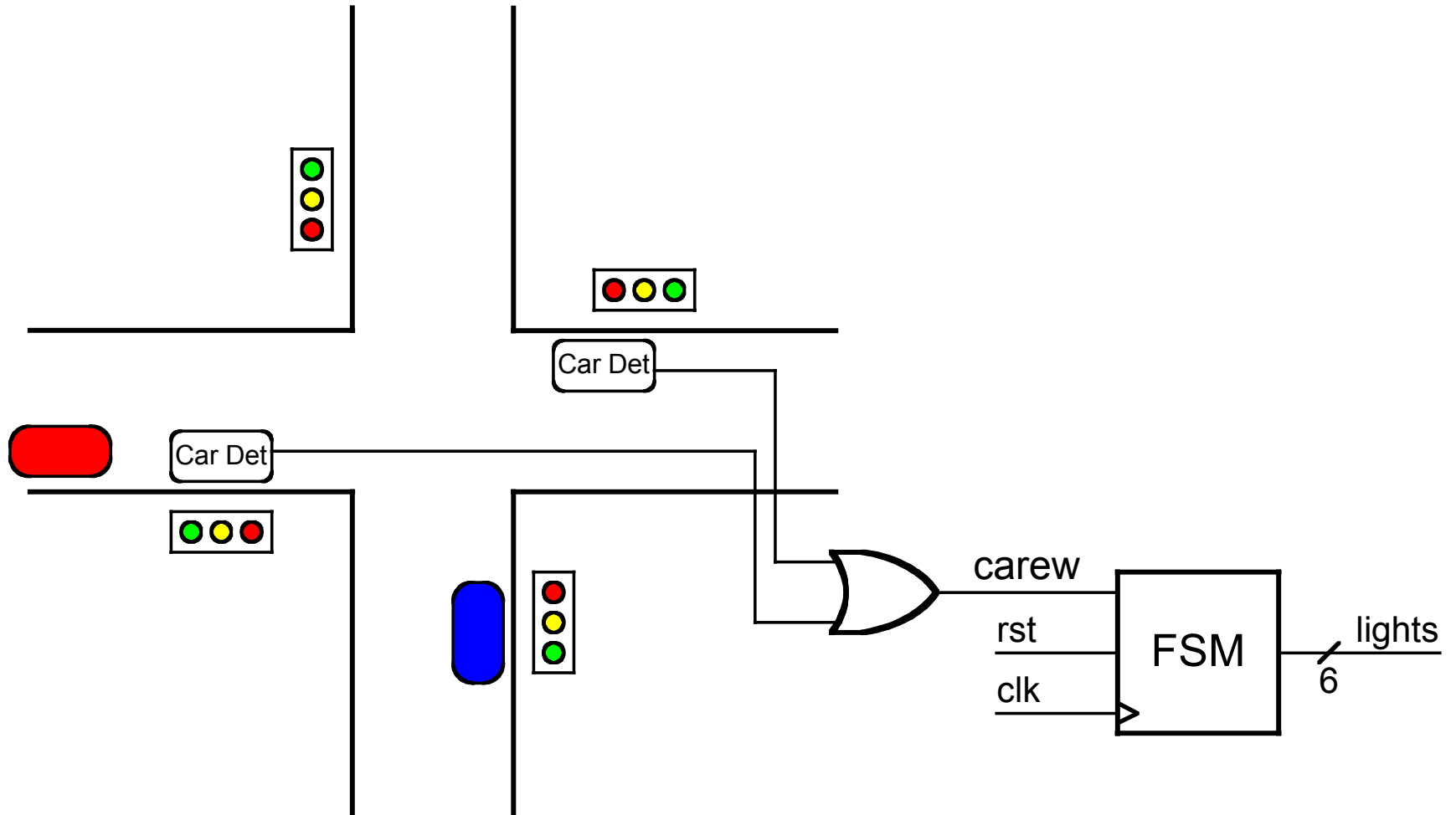


Sequential Building Block: D Flip-Flop

- Input: D
- Output: Q
- Clock: \wedge
- Q outputs a steady value
- Edge on \wedge changes Q to be D
- Flip-flop stores state



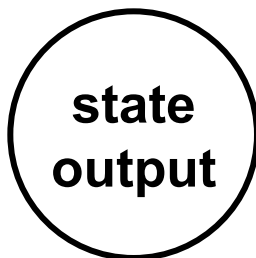
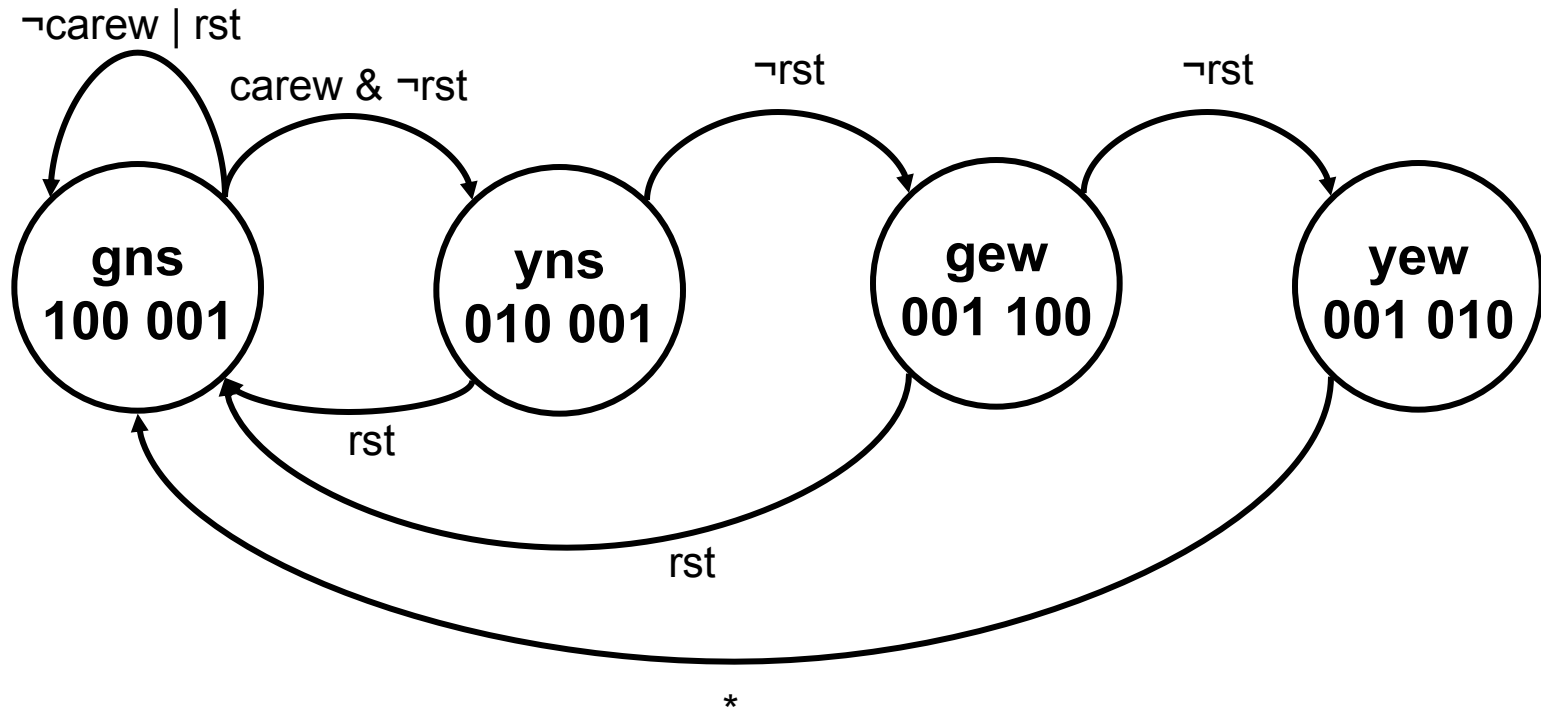
Example: A Traffic-Light Controller



Finite State Machines (State Table)

State	Next State		Output
	!carew	carew	
00	00	01	100001
01	11	11	010001
11	10	10	001100
10	00	00	001010

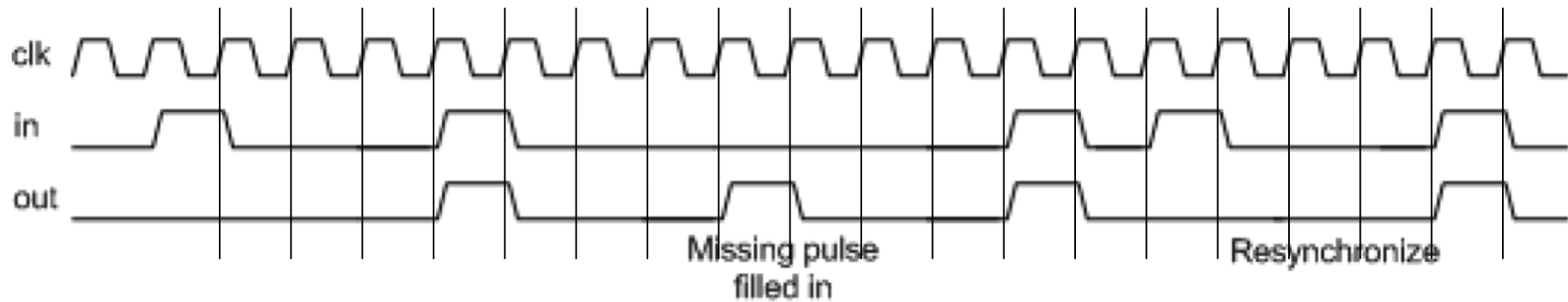
Finite State Machines (State Diagram)



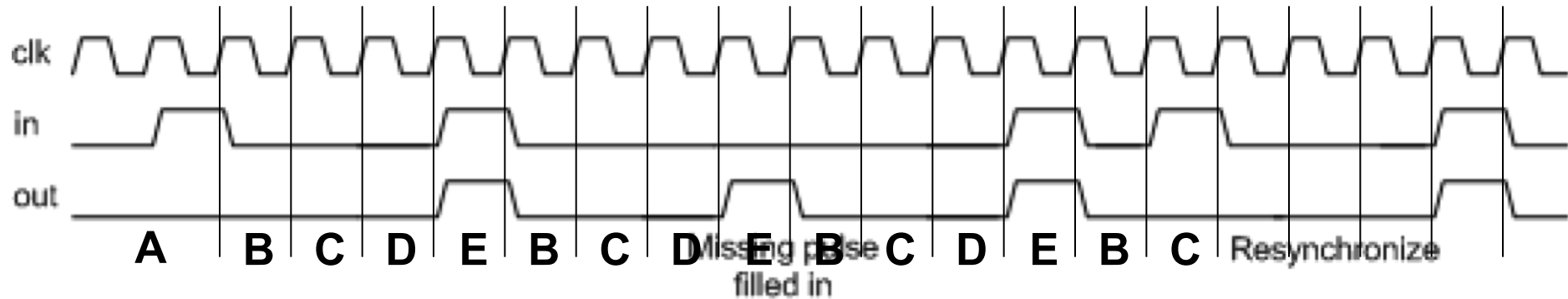
= gyr gyr
ns ew

carew = car east-west sensor active
rst = reset

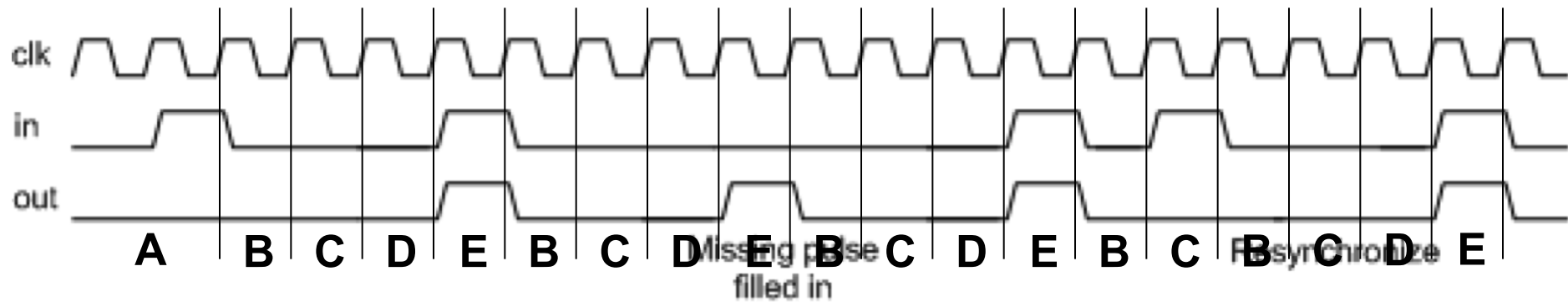
Draw a State Diagram for a Missing-Pulse Filling FSM



Draw a State Diagram for a Missing-Pulse Filling FSM



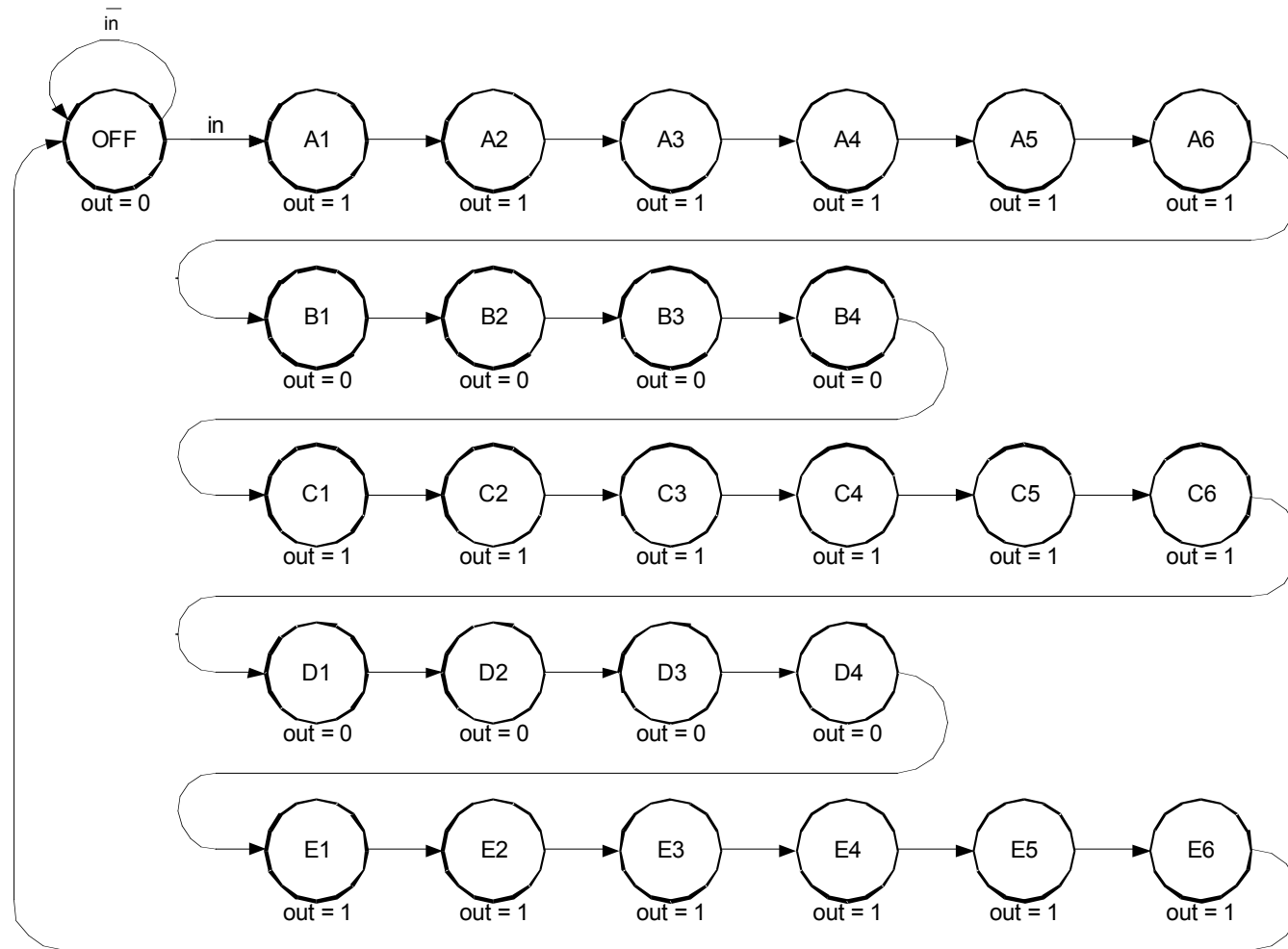
Draw a State Diagram for a Missing-Pulse Filling FSM



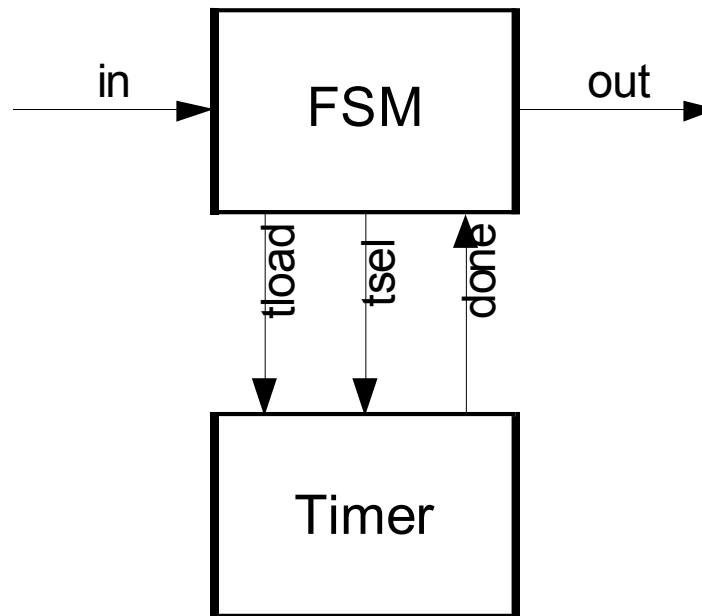
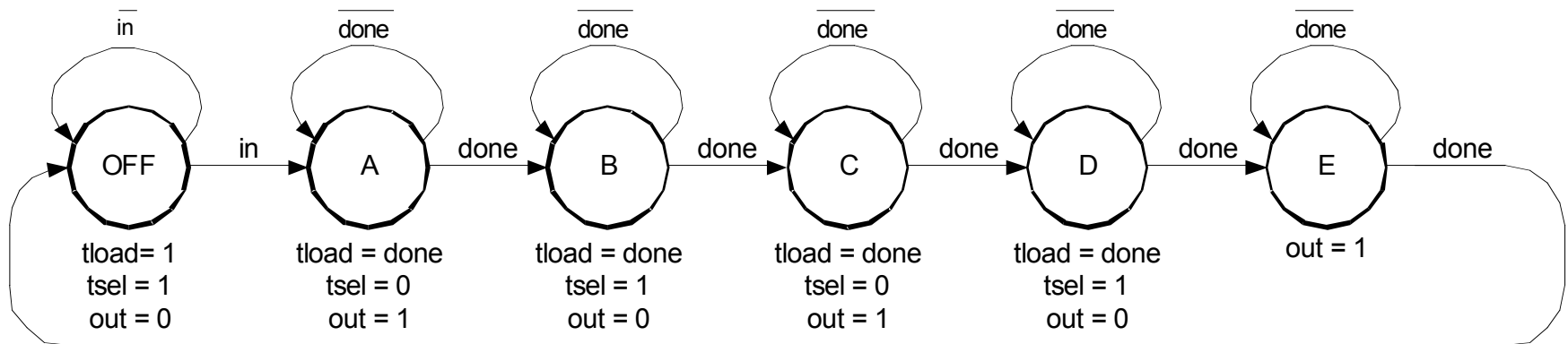
FSM Factoring

- Partition problem into smaller problems
 - Major theme of circuit design
- Separate a multi-dimensional state space into multiple smaller FSMs
 - Control and data
 - State variables

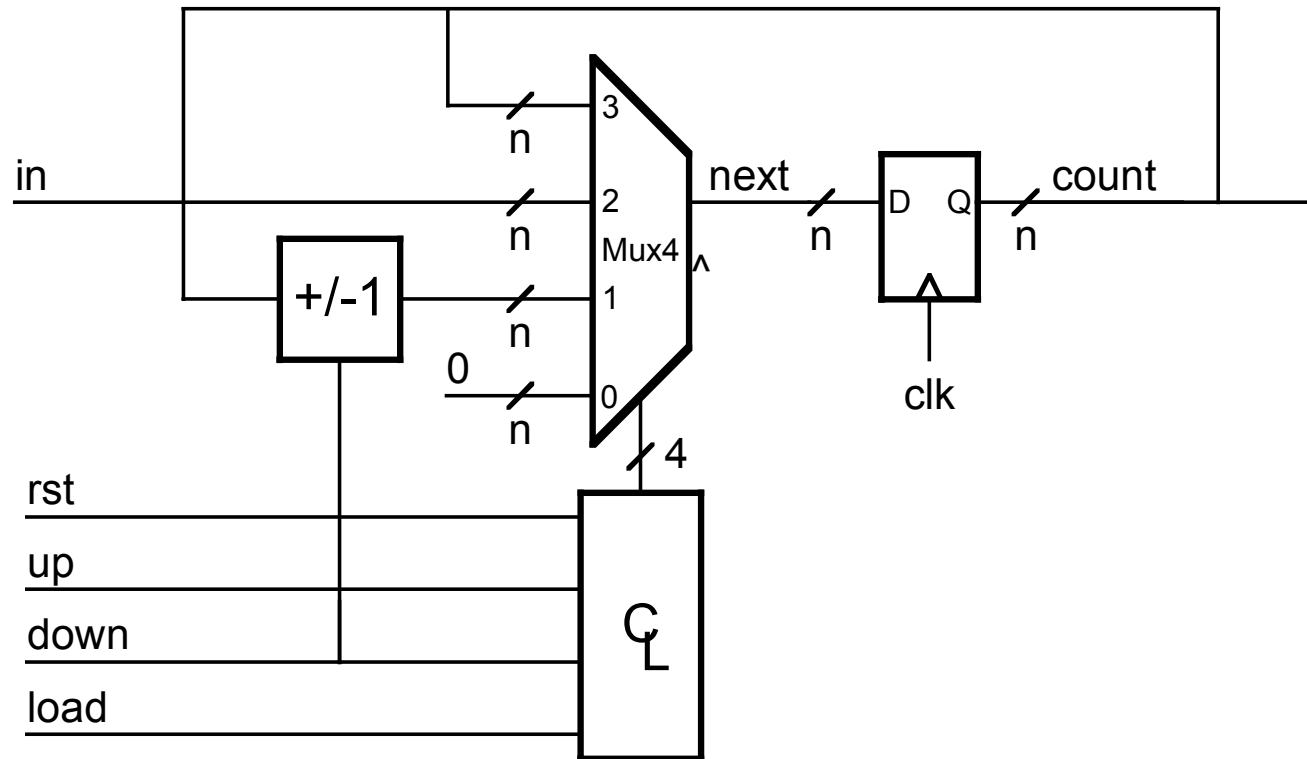
Light Flasher FSM Diagram



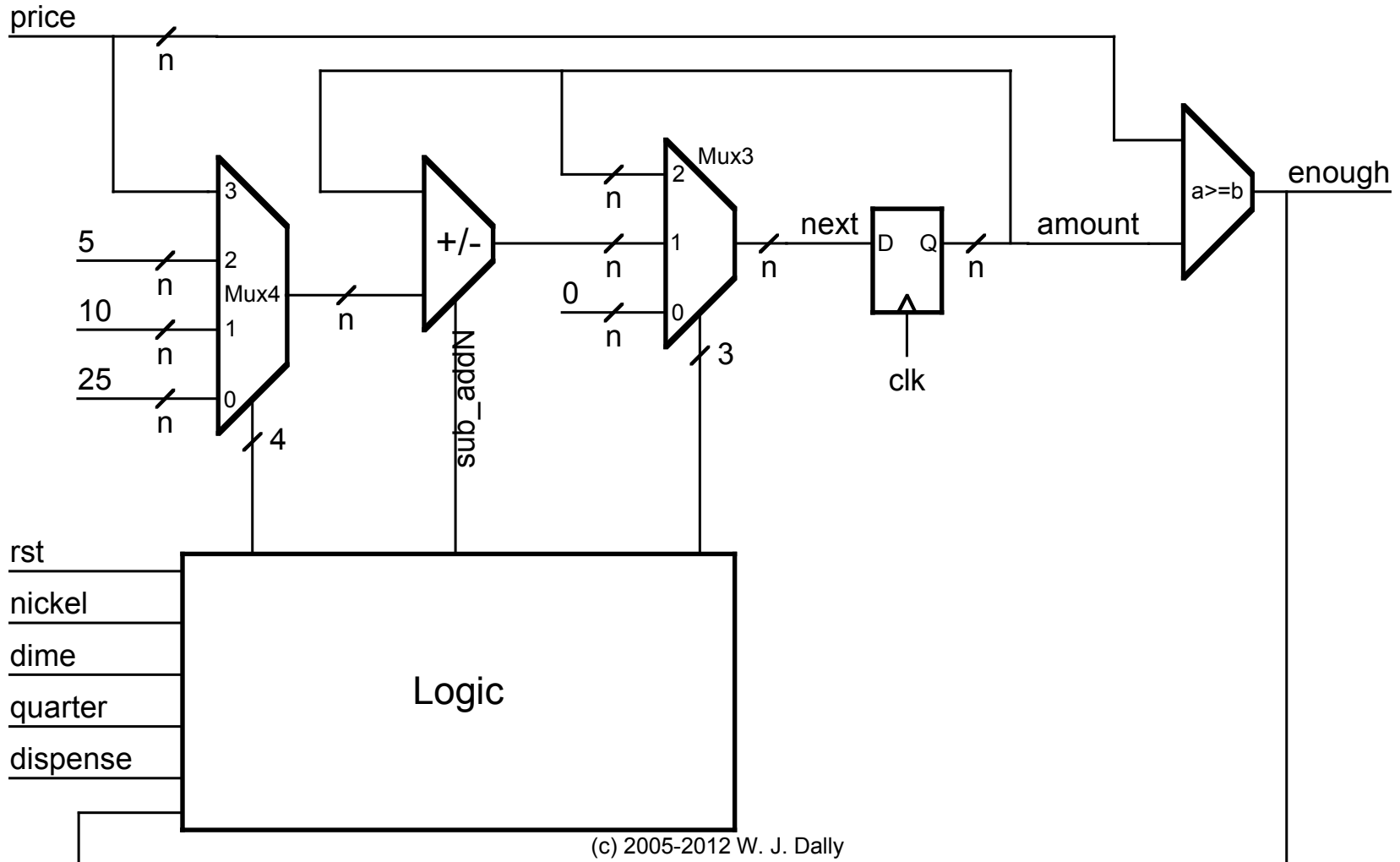
Factored Light Flasher



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



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



Design a DP FSM that computes parity across a block of 16 8-bit words

Design a DP FSM that computes 'horizontal' parity across a block of 16 8-bit words

- Inputs
 - Start – first word
 - In – 8-bit data
- Outputs
 - P – parity – running parity across 16 words then hold until next start
- Two pieces of state
 - P – running parity
 - N – words to go

DP FSM is designed by writing equations for each case

Case	P	N
Start	in	15
~Start & ~Done	$\text{in} \wedge P$	$N-1$
Done	P	$N (=0)$

$P = \text{start} ? \text{in} : (\text{done} ? P : P \wedge \text{in})$

$N = \text{start} ? 15 : (\text{done} ? 0 : N-1)$

$\text{Done} = (N == 0)$

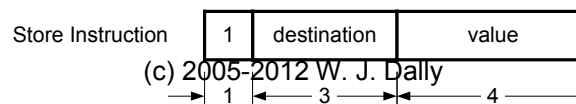
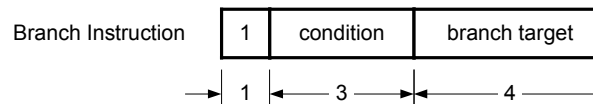
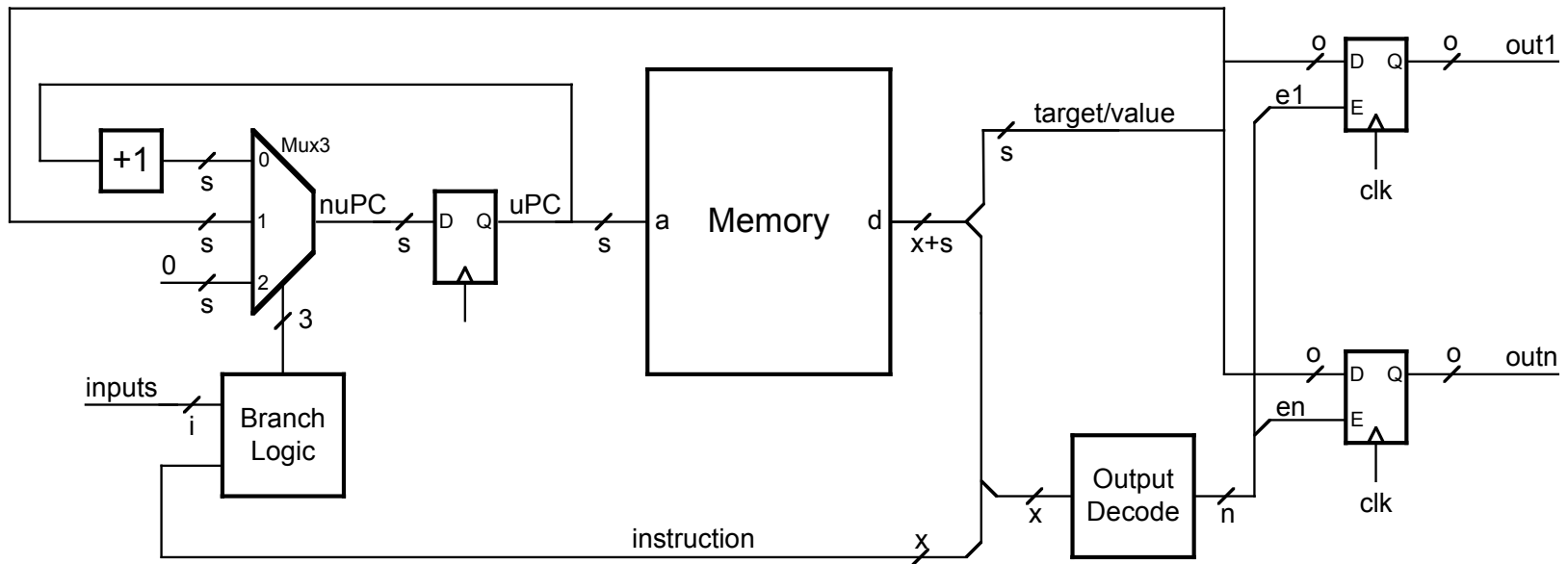
Check timing with table

Cycle	Start	Done	Next P	N
i-1	0	1	P	0
i	1	1	in_i	0
i+1	0	0	$in_i \wedge in_{i+1}$	15
i+j	0	0	$in_i \wedge \dots \wedge in_{i+j}$	16-j
i+15	0	0	$in_i \wedge \dots \wedge in_{i+15}$	1
i+16	0	1	$in_i \wedge \dots \wedge in_{i+15}$	0

Catches off-by-one errors

Microcode

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



Microcode For Traffic-Light Controller With brx & Idx Instructions

	State	Addr	Inst	Value					
NS Green	rst1	00000	ldlt	RED	EW to Red	ew6	10000	ldew	YELLOW
	rst2	00001	ldew	RED		ew7	10001	ltim	T_YELLOW
	ns1	00010	ldns	GREEN		ew8	10010	bntz	ew8
	ns3	00011	ltim	T_GREEN		ew9	10011	ldew	RED
	ns4	00100	bntz	ns4		ew10	10100	br	ns1
Until input	ns5	00101	brnle	ns5	Wait for NS Red, make LT Green	lt1	10101	ltim	T_RED
NS Yellow to Red	ns6	00110	ldns	YELLOW		lt2	10110	bntz	lt2
	ns7	00111	ltim	T_YELLOW		lt3	10111	ldlt	GREEN
	ns8	01000	bntz	ns8		lt4	11000	ltim	T_GREEN
	ns9	01001	ldns	RED		lt5	11001	bntz	lt5
EW or LT?	ns10	01010	blt	lt1	LT to Red	lt6	11010	ldlt	YELLOW
Wait for NS Red, make EW Green	ew1	01011	ltim	T_RED		lt7	11011	ltim	T_YELLOW
	ew2	01100	bntz	ew2		lt8	11100	bntz	lt8
	ew3	01101	ldew	GREEN		lt9	11101	ldlt	RED
	ew4	01110	ltim	T_GREEN		Back to NS	lt10	11110	br
	ew5	01111	bntz	ew5					

Microcode

- Microcode is just FSM implemented with a ROM or RAM
 - One address for each state x input combination
 - Address contains next state and output
- Adding a *sequencer* reduces size of ROM/RAM
 - One entry per state rather than 2^i
 - uPC, incrementer, branch address, and branch control
- Adding instruction types reduces width of ROM/RAM
 - Branch *or* output in each instruction – rather than both
 - Type field specifies which one

Circuit Timing

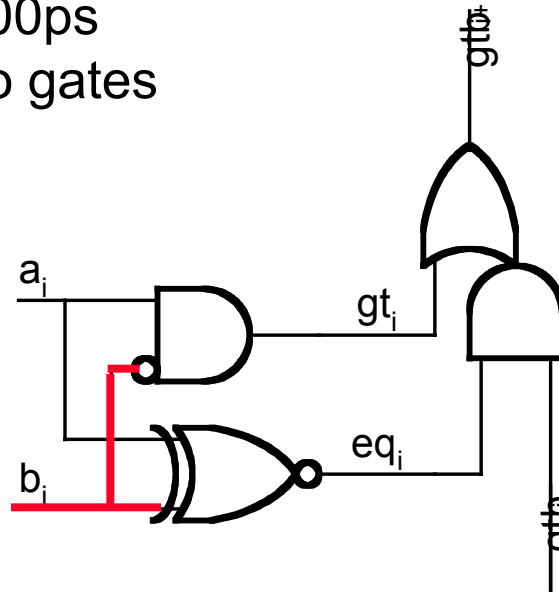
- t_{cab} (contamination delay): minimum time it takes the output B to be contaminated from a contamination of input A
- t_{dab} (propagation delay): maximum time it takes the output B to be stable from when input A is stable
- t_s (setup time): duration signal must be stable before clocking a flip-flop
- t_h (hold time): duration signal must be stable after clocking a flip-flop

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



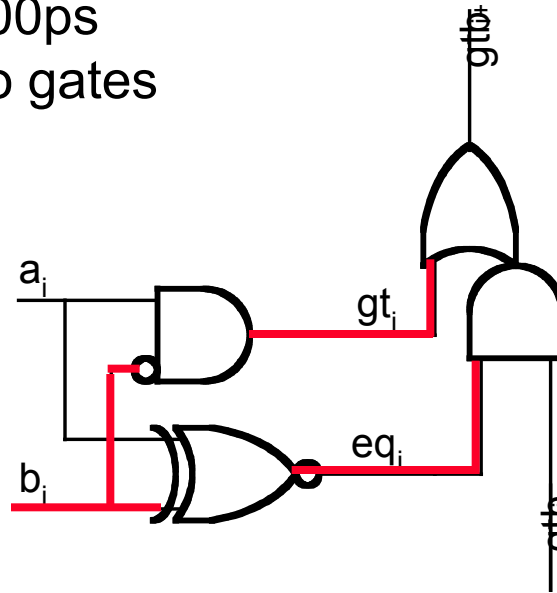
$t = 0$

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



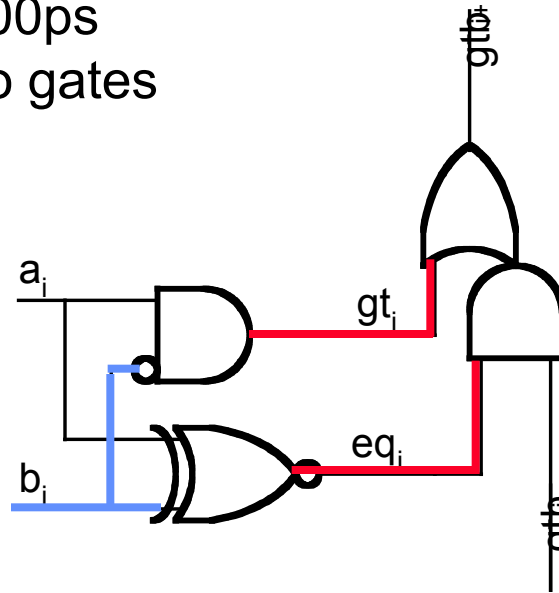
$t = 100$

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



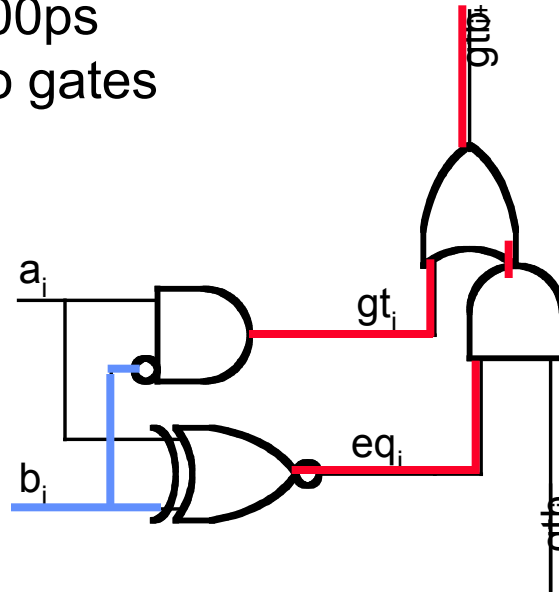
$t = 150$

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



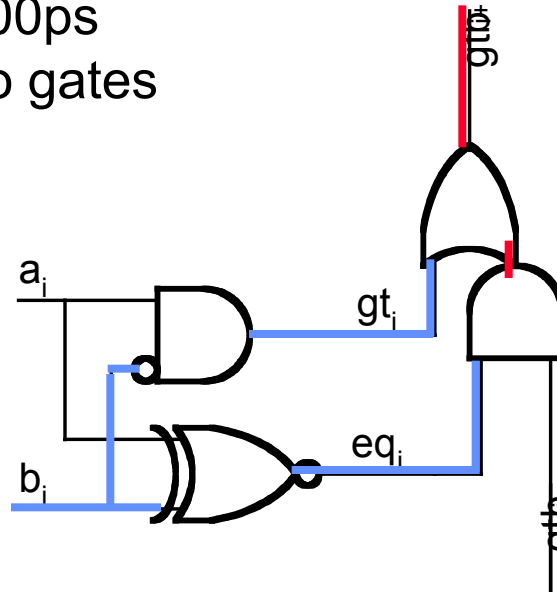
$t = 200$
 $t_{cbg} = 200\text{ps}$

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



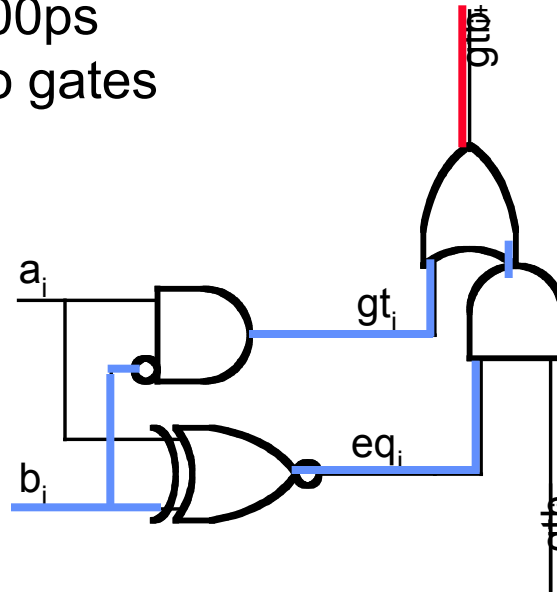
$t = 250$
 $t_{cbg} = 200\text{ps}$

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



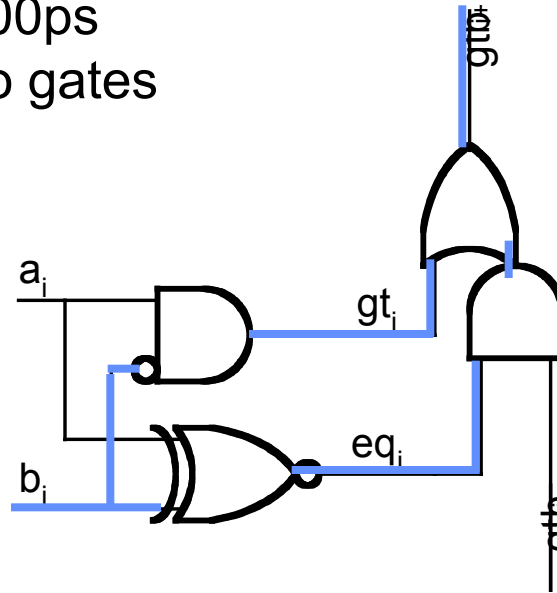
$$t = 350$$
$$t_{cbg} = 200\text{ps}$$

Timing Example

b contaminated at time $t=0$, stable at $t=150$

All gates have delay 100ps

AND-OR counts as two gates



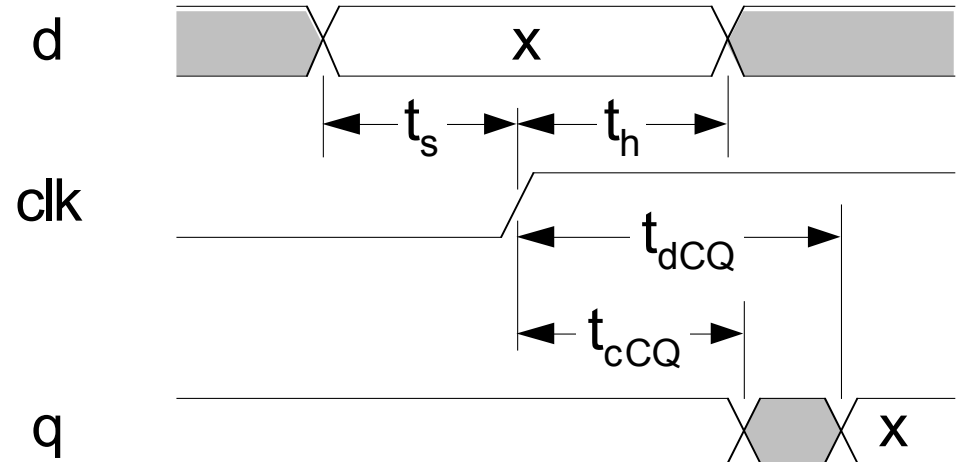
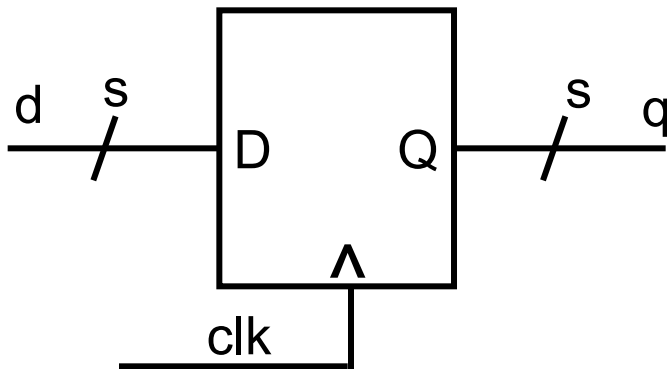
$t = 450$

$t_{cbg} = 200\text{ps}$

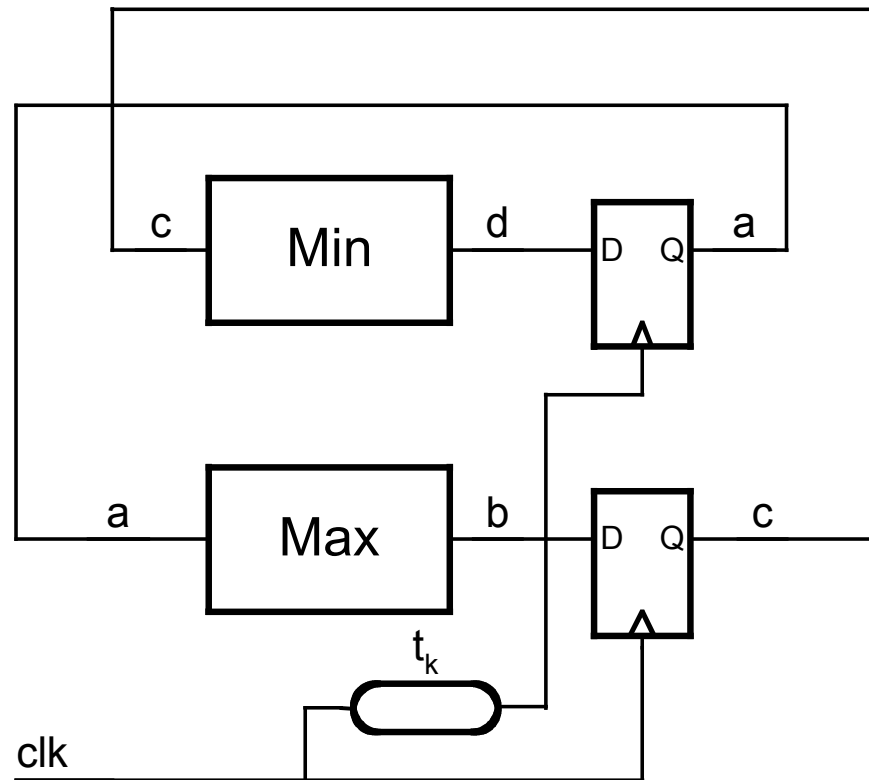
$t_{dbg} = 300\text{ps}$

Setup and Hold Times (back to our friend the DFF)

- t_s (setup time): duration signal must be stable before clocking a flip-flop
- t_h (hold time): duration signal must be stable after clocking a flip-flop
- t_{ccQ} : contamination time of clock to Q
- t_{dcQ} : propagation time of clock to Q



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

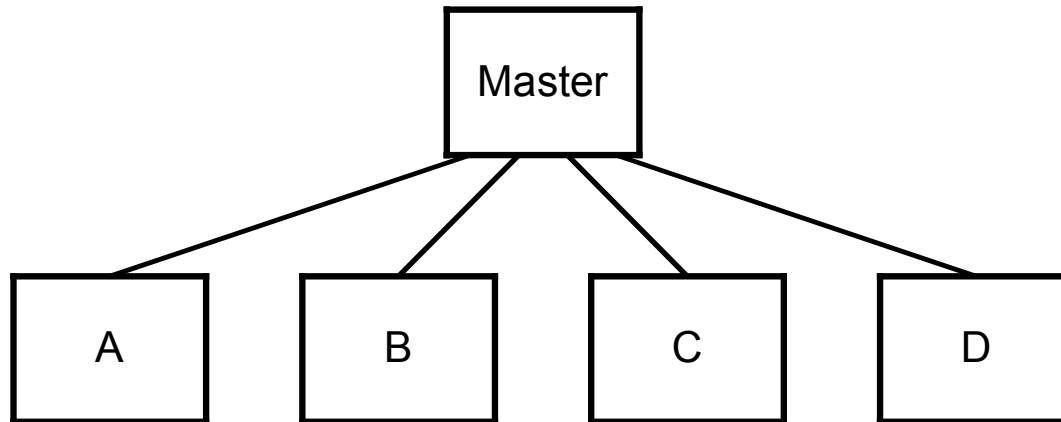
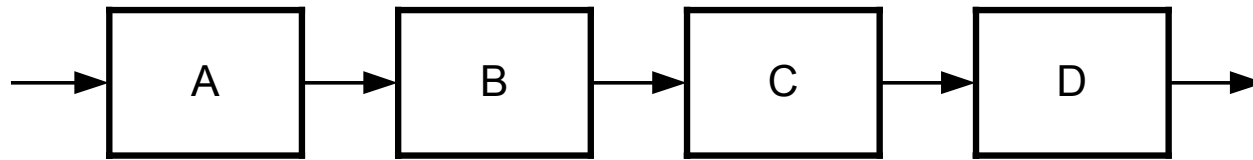


Suppose $t_{dcQ} = t_{ccQ} = t_s = t_h = 100\text{ps}$, $t_k = 200\text{ps}$ or -200ps ,
 $t_{\min} = 50\text{ps}$, $t_{\max} = 2\text{ns}$.

Is hold time met? What is minimum t_{cy} ?

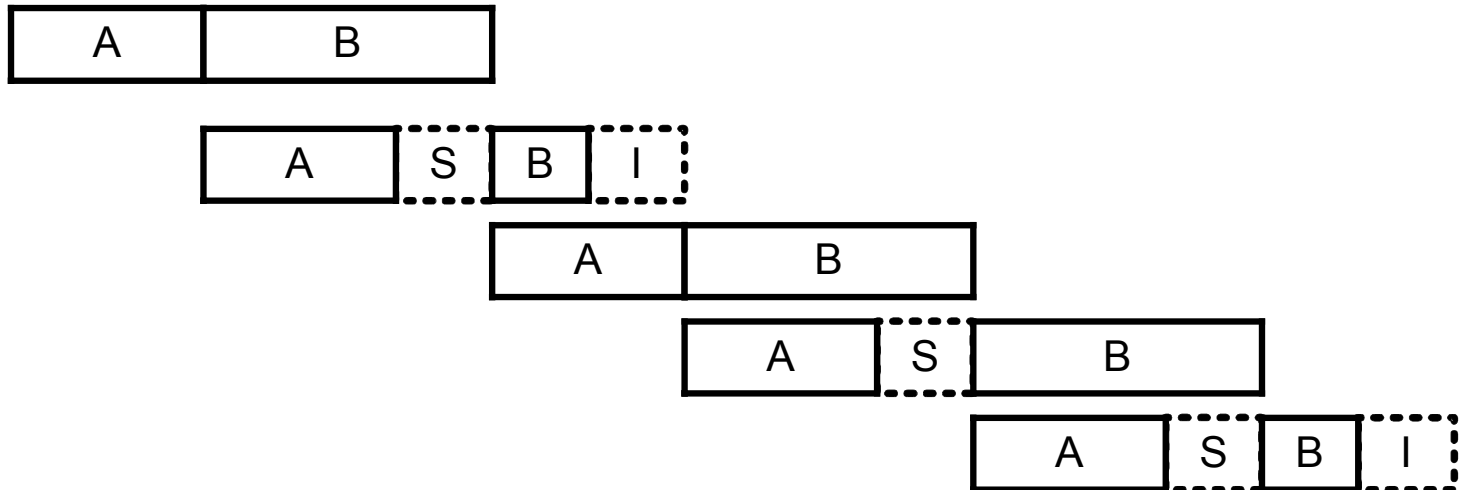
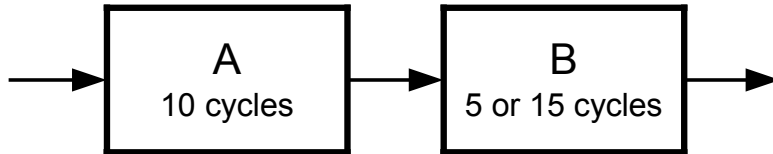
Pipelining

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



Pipelines

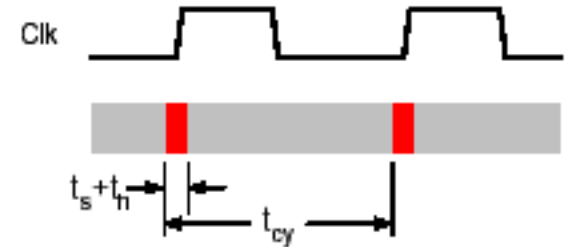
- Pipelines can stall and idle
- When do these happen? How can you prevent them?
- Max latency vs. average latency (absorbing bursts)



Asynchronous Failure

- Dealing with asynchrony
 - Purely asynchronous signals
 - Traversing clock domains
- Probability of entering illegal state is the probability that a clock edge lands during setup or hold time:

$$P_E = (t_s + t_h) / t_{cy}$$



- For reasonable values, error rate is very high:

$$t_s = t_h = 100\text{ps}, t_{cy} = 2\text{ns}, P_E = 0.1$$

Synchronizing

- Sample asynchronous signal a
- Clock a into intermediate (possibly metastable) value a_w
- Clock a_w into synchronized value a_s
- Take advantage of exponential decay of metastability

$$t_s = t_h = t_{dCQ} = \tau_s = 100\text{ps} \text{ and } t_{cy} = 2\text{ns}$$

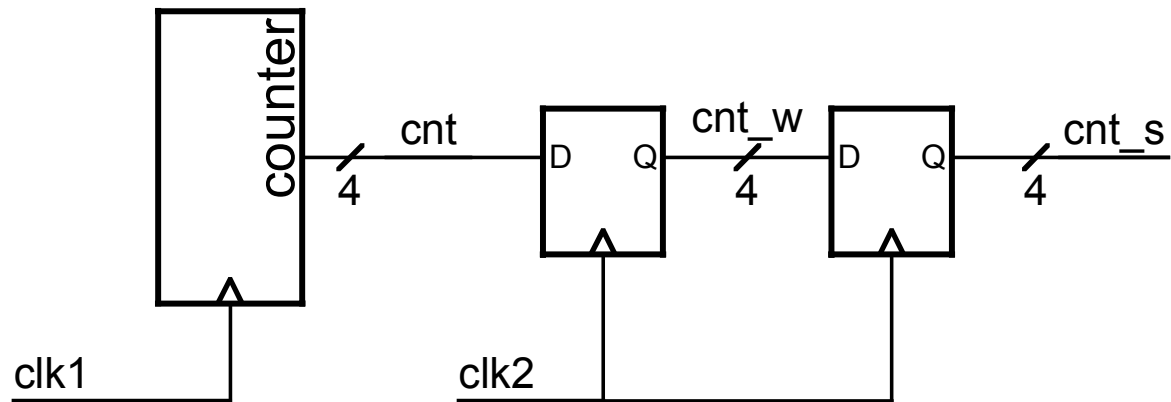
$$P_{ES} = P_E P_S = \left(\frac{t_s + t_h}{t_{cy}} \right) \exp \left(\frac{-t_w}{\tau_s} \right)$$

$$t_w = t_{cy} - t_s - t_{dCQ}$$

$$\begin{aligned} P_{ES} &= \left(\frac{t_s + t_h}{t_{cy}} \right) \exp \left(\frac{-t_w}{\tau_s} \right) \\ &= \left(\frac{100\text{ps} + 100\text{ps}}{2\text{ns}} \right) \exp \left(\frac{-1.8\text{ns}}{100\text{ps}} \right) \\ &= 0.1 \exp(-18) = 1.5 \times 10^{-9} \end{aligned}$$

- Note $\ln(10)=2.3$ so $\exp(-18) = 10^{(-18/2.3)} \sim 10^{(-8)}$

How do you fix this?



Questions
