

---

# EECS 150 -- Digital Design

## Lecture 4 – Synchronous Digital Systems Review (Part II)

---

**2010-1-28**

**John Wawrzynek**

**Today's lecture by John Lazzaro**

---

**[www-inst.eecs.berkeley.edu/~cs150](http://www-inst.eecs.berkeley.edu/~cs150)**

---



EECS150 – Digital Design  
Lecture 4 – Synchronous  
Digital Systems Review Part 2

January 28, 2010

John Wawrzynek

Electrical Engineering and Computer Sciences  
University of California, Berkeley

<http://www-inst.eecs.berkeley.edu/~cs150>

# Outline

- Topics in the review, you have already seen in CS61C, and possibly EE40:
  1. Digital Signals.
  2. General model for synchronous systems.
  3. Combinational logic circuits
  4. Flip-flops, clocking

# Today's Lecture

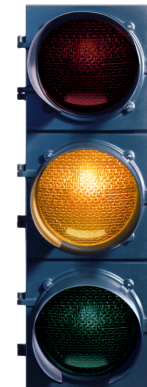
---

- \* **Flip-flop-based state machines**  
*Operates on Boolean (single-bit) values.*
- \* **Register-based state machines**  
*Operates on multi-bit values (integers, CPU instruction, ...)*
- \* **Registers and Pipelining**  
*Adding state to speed up the clock.*
- \* **Flip-flop details ...**  
*(Reset, set, etc ...)*

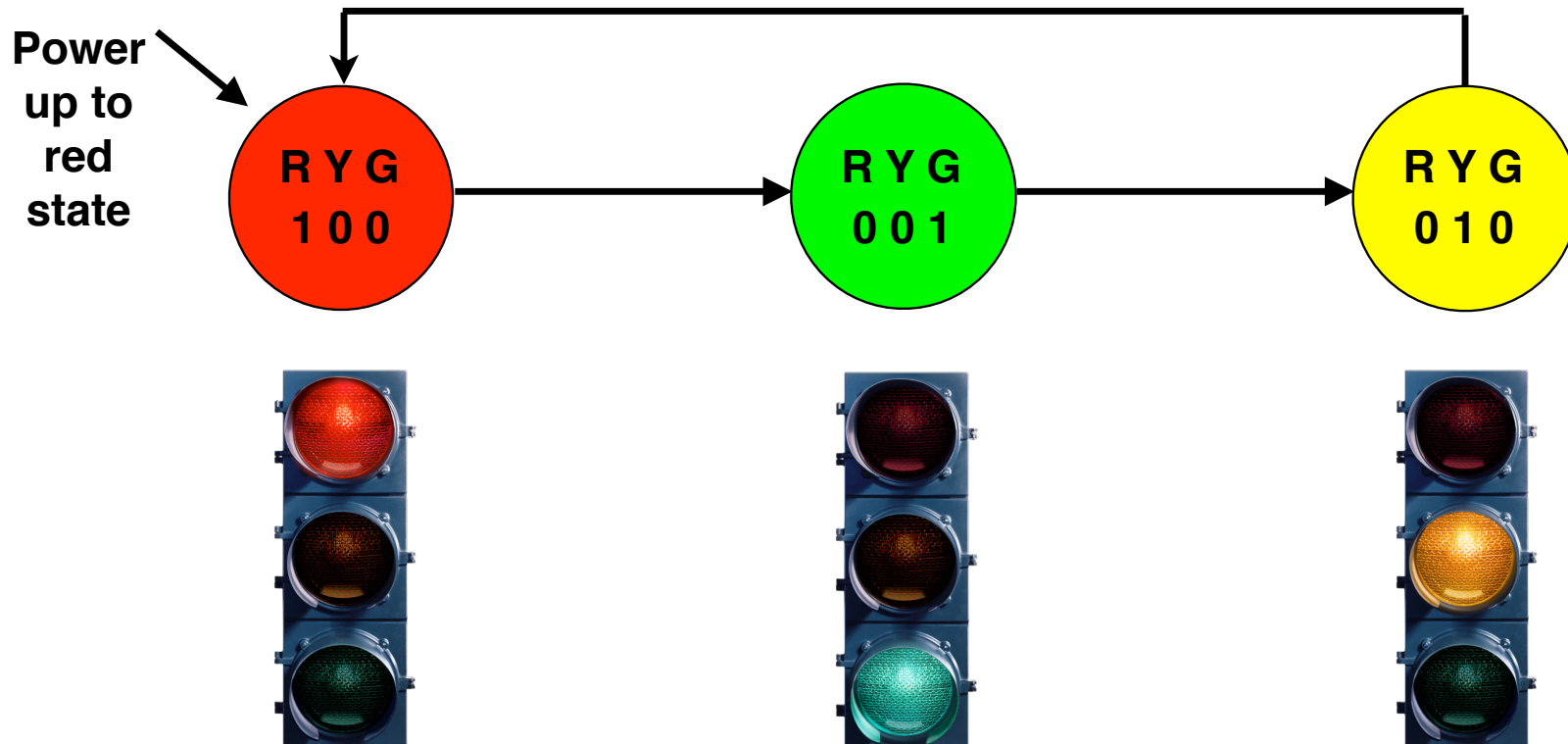


# Flip-Flop State Machines

---



# A Simple System: Traffic Light Controller



\* Show each light for 1 second.







\* "Loop" forever.

# 'C' program for traffic light controller

```
int main() {  
  
    int r = 1, y = 0, g = 0;        /* light off/on */  
  
    while (1)  
    {  
  
        printf("r=%i\ny=%i\ng=%i\n\n", r, y, g);  
        sleep(1);  
  
    }  
}
```

```
% ./TRAFFIC  
R=1  
Y=0  
G=0  
  
R=1  
Y=0  
G=0  
  
R=1  
Y=0  
G=0  
  
R=1  
Y=0  
G=0
```

# 'C' program for traffic light controller

```
int main() {  
  
    int r = 1, y = 0, g = 0;      /* light off/on */  
    int next_r, next_y, next_g; /* extra state */  
  
    while (1)  
    {  
        next_r = y;    
        next_y = g;    
        next_g = r;    
  
        printf("r=%i\ny=%i\ng=%i\n\n", r, y, g);  
        sleep(1);  
  
        r = next_r;    
        y = next_y;    
        g = next_g;    
    }  
}
```

```
% ./TRAFFIC  
R=1  
Y=0  
G=0  
  
R=0  
Y=0  
G=1  
  
R=0  
Y=1  
G=0  
  
R=1  
Y=0  
G=0  
  
R=0  
Y=0  
G=1
```



# A few observations ...

```
int main() {
```

```
int r = 1, y = 0, g = 0; /* light off/on */  
int next_r, next_y, next_g; /* extra state */
```

Wouldn't it be great if we could group "current" and "next" variables with an abstraction?

```
while (1)
```

```
{
```

```
next_r = y;  
next_y = g;  
next_g = r;
```

Code would still work if these statements executed simultaneously.

```
printf("r=%i\ny=%i\ng=%i\n\n", r, y, g);  
sleep(1);
```

Sleep(1) sets "time constant", not C instruction execution rate.

```
r = next_r;  
y = next_y;  
g = next_g;
```

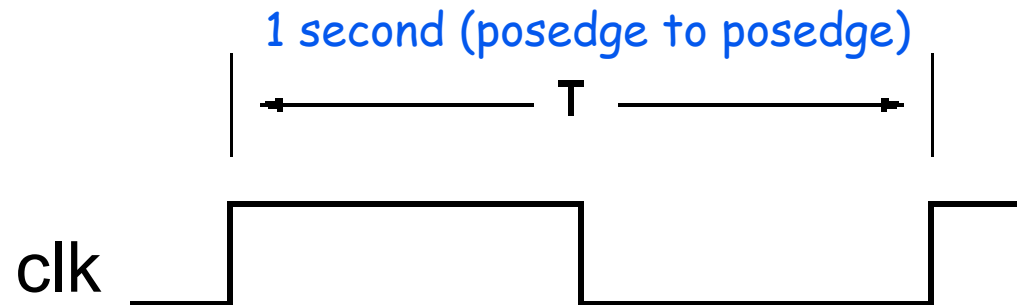
Code would still work if these statements executed simultaneously.

```
}
```

```
}
```

**A direct digital hardware implementation addresses all of these issues!**

# Clock waveform takes the role of sleep(1)

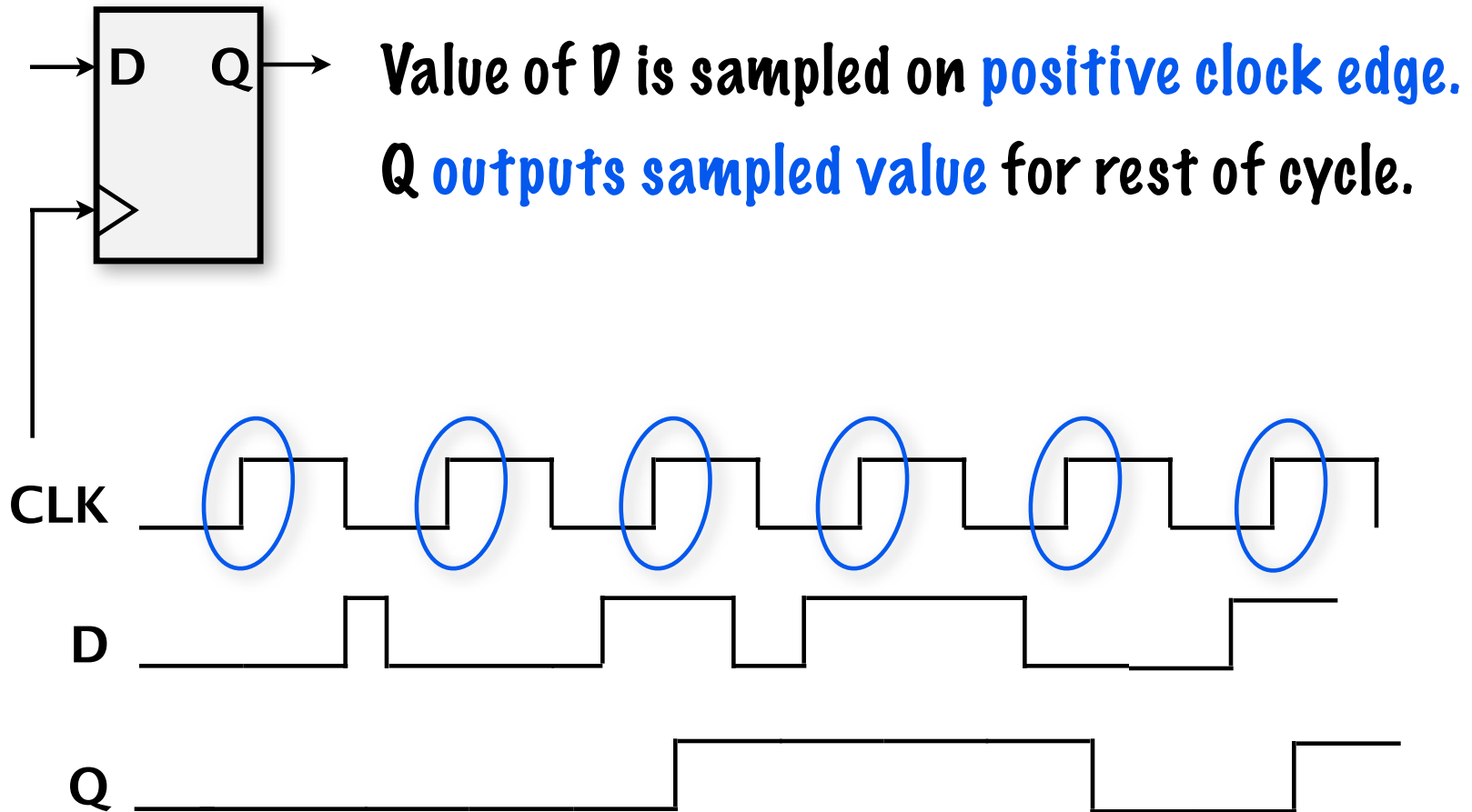


All state changes happen on **leading** edge of our **1 Hz** clock ... thus, lights will switch once per second.

f	T
1 Hz	1 s
1 MHz	1 $\mu$ s
1 GHz	1 ns



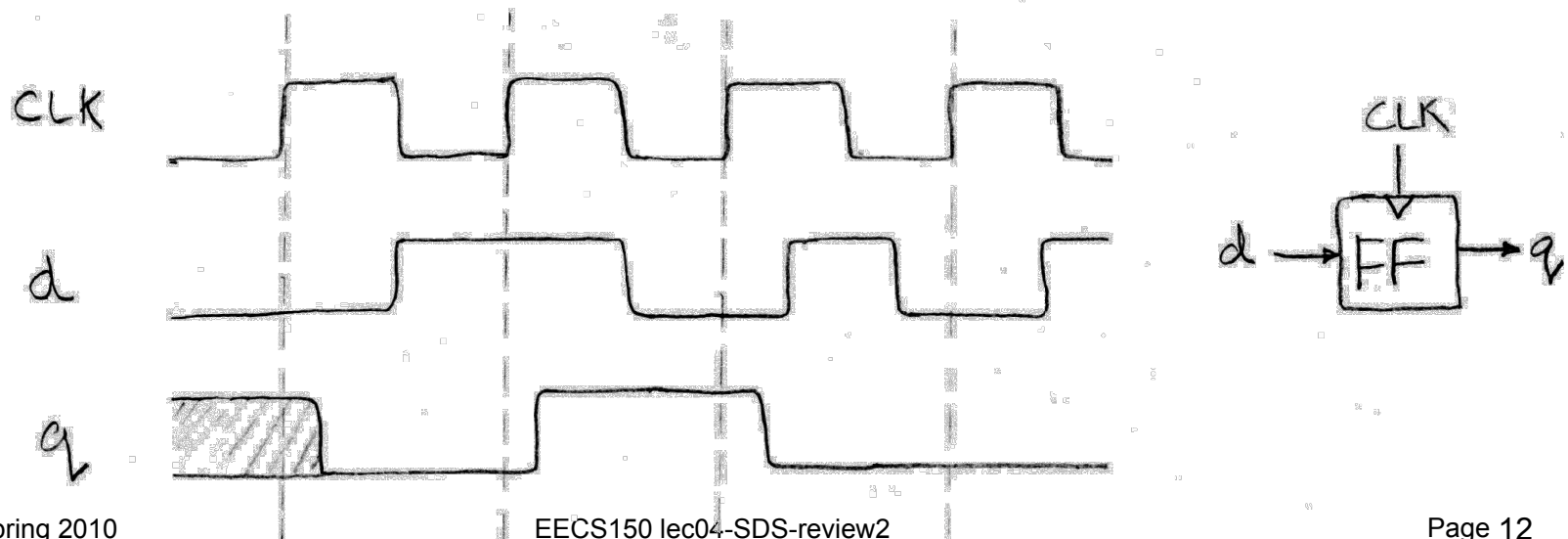
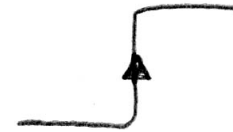
# r & next\_r? One edge-triggered D flip-flop



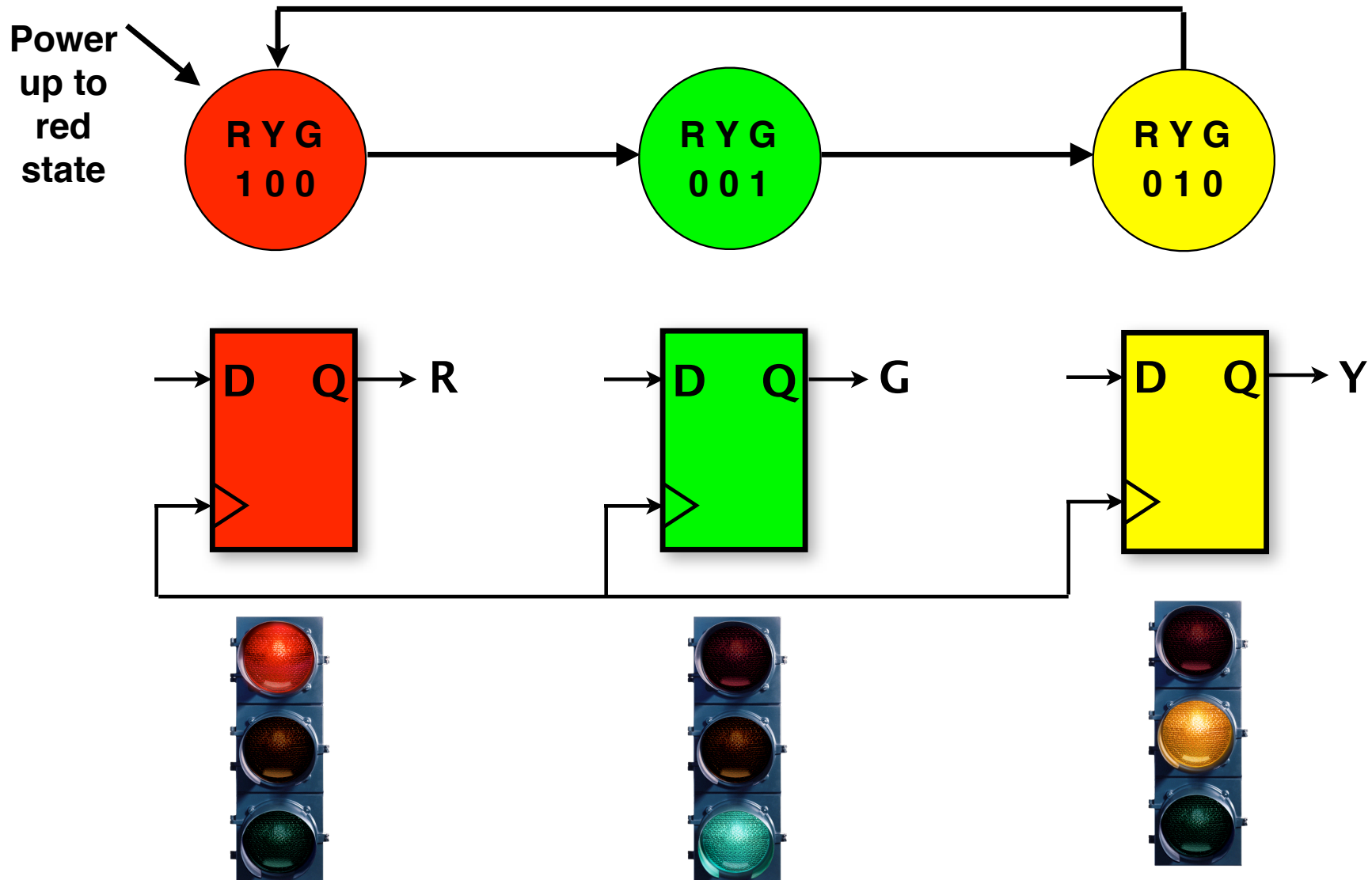
**Positive-edge sampling makes it easy to think about state.**

# Flip-flop Timing Waveforms?

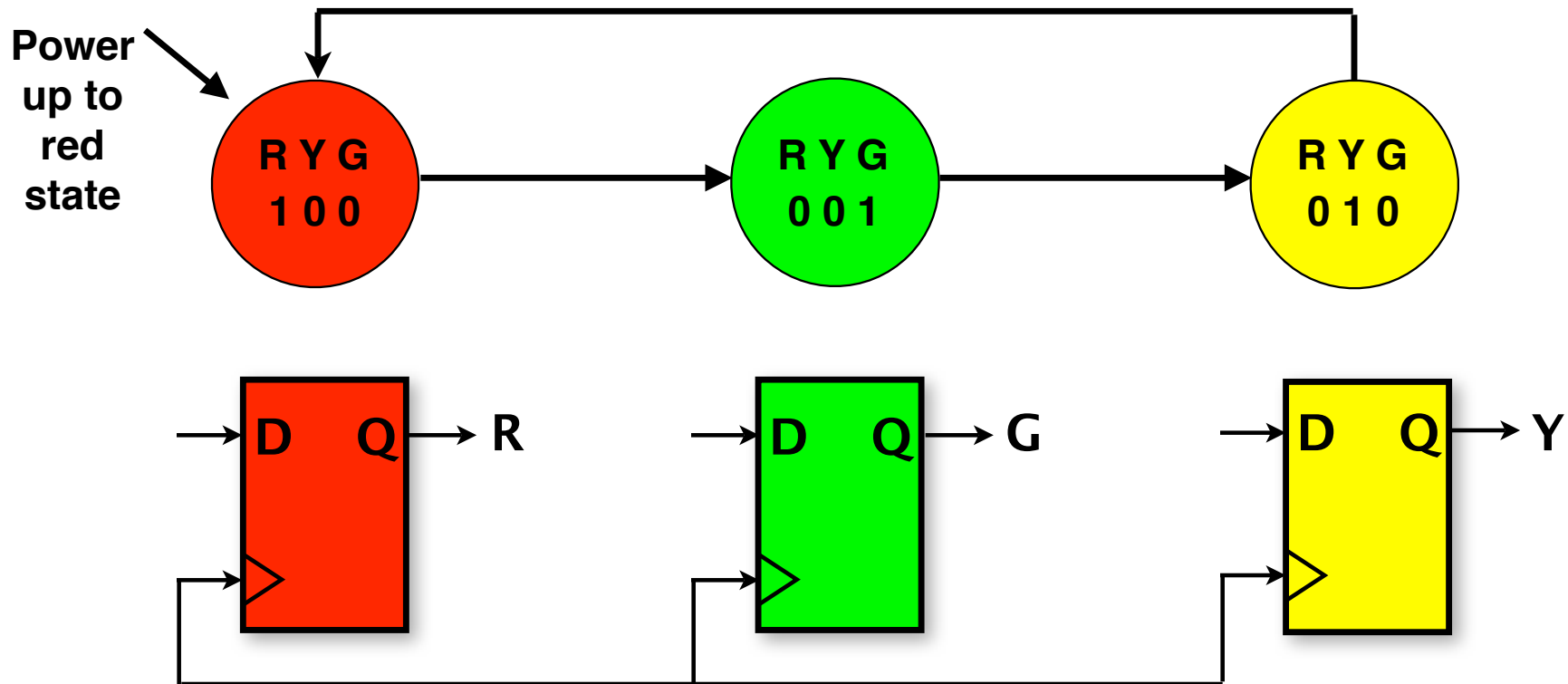
- Edge-triggered d-type flip-flop
  - This one is “positive edge-triggered”
- “On the rising edge of the clock, the input d is sampled and transferred to the output. At all other times, the input d is ignored.”
- Example waveforms:



# Use 3 Flip-Flops to represent state ...

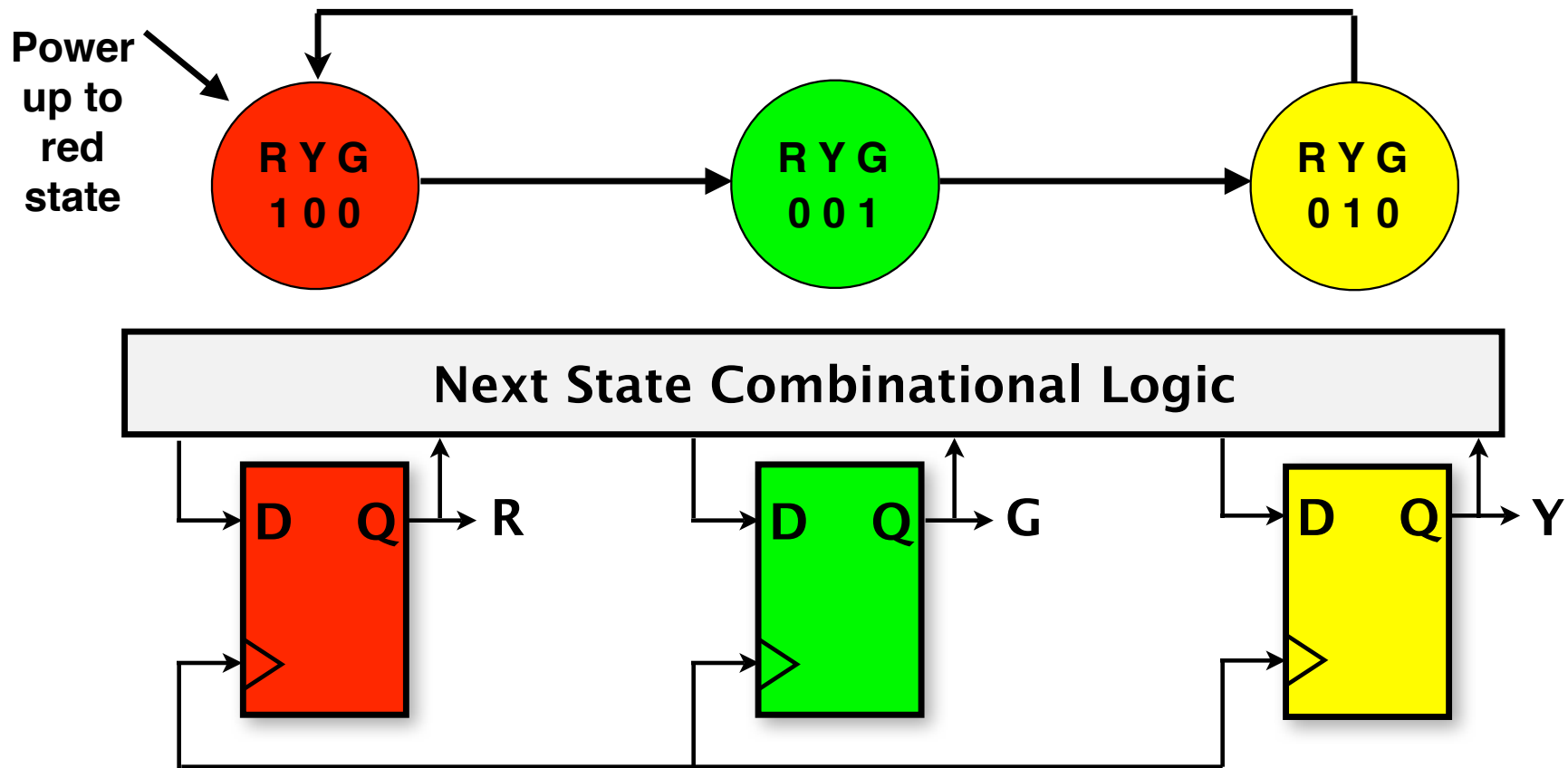


# Use 3 Flip-Flops to represent state ...



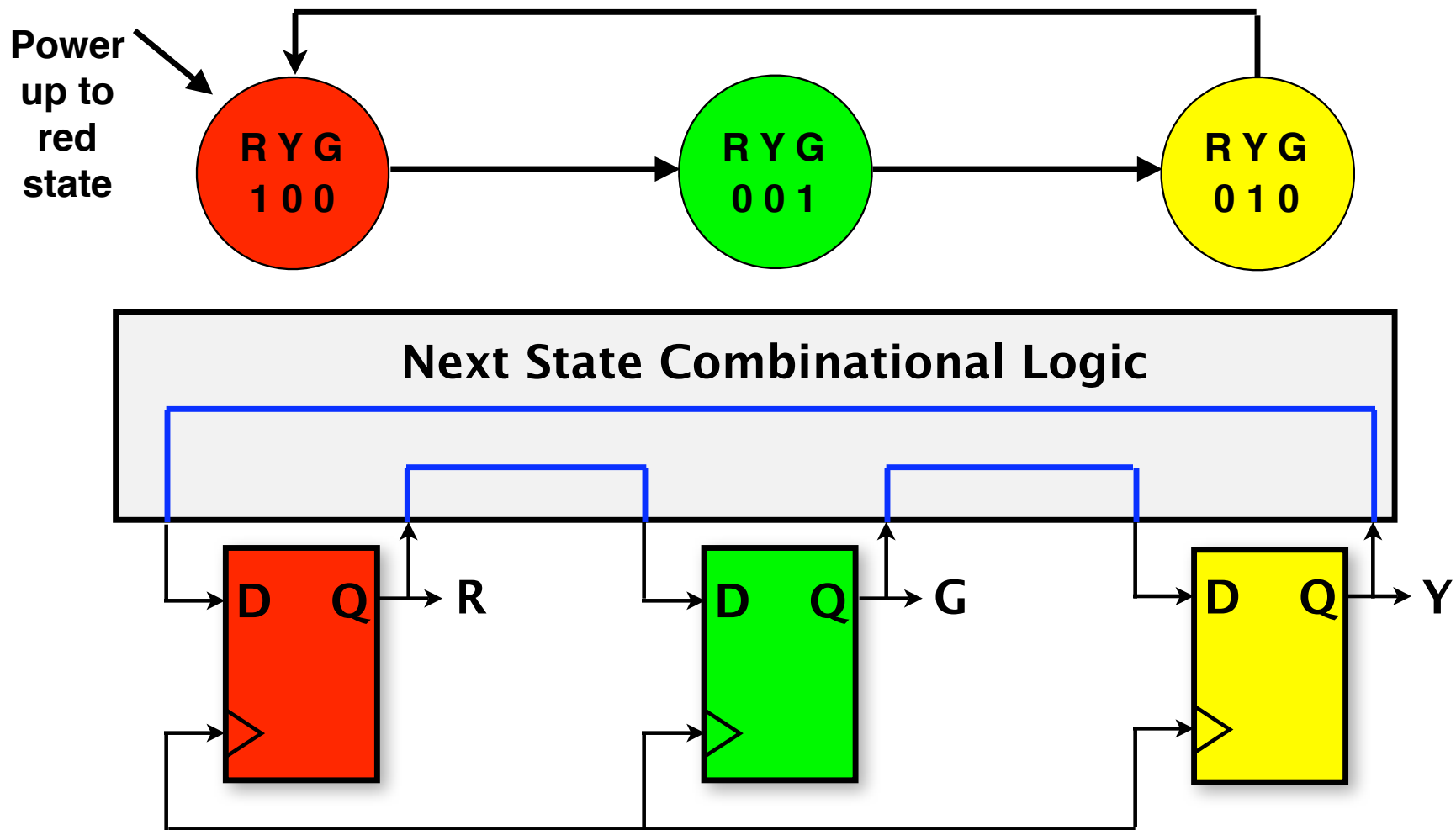
**“One-Hot Encoding”**: State machines where exactly one D flip-flop is in the “1” state at a time (forbidden states: RYG = 000, 011, 101, 110, 111).

# “Simplified” traffic light controller



**“Simplified???”**: We assume the state at the beginning of time is  $RYG == 100$ . A “complete” implementation would include “power up” logic.

# Inside the combinational logic box ...



Let's revisit our original C code ...



# Recall: A few observations ...

---

```
int main() {
```

```
int r = 1, y = 0, g = 0;    /* light off/on */
```

```
int next_r, next_y, next_g; /* extra state */
```

6 C variables, but  
only 3 flip-flops.  
How does that  
work?

```
while (1)
```

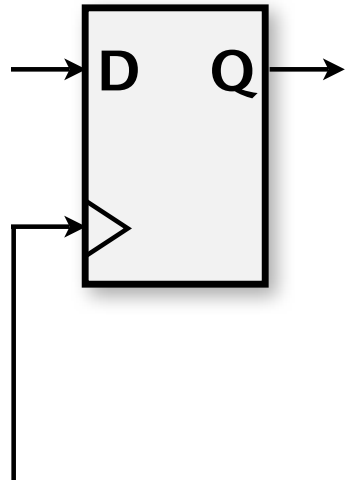
```
{  
  next_r = y;  
  next_y = g;  
  next_g = r;
```

```
  printf("r=%i\ny=%i\ng=%i\n\n", r, y, g);  
  sleep(1);
```

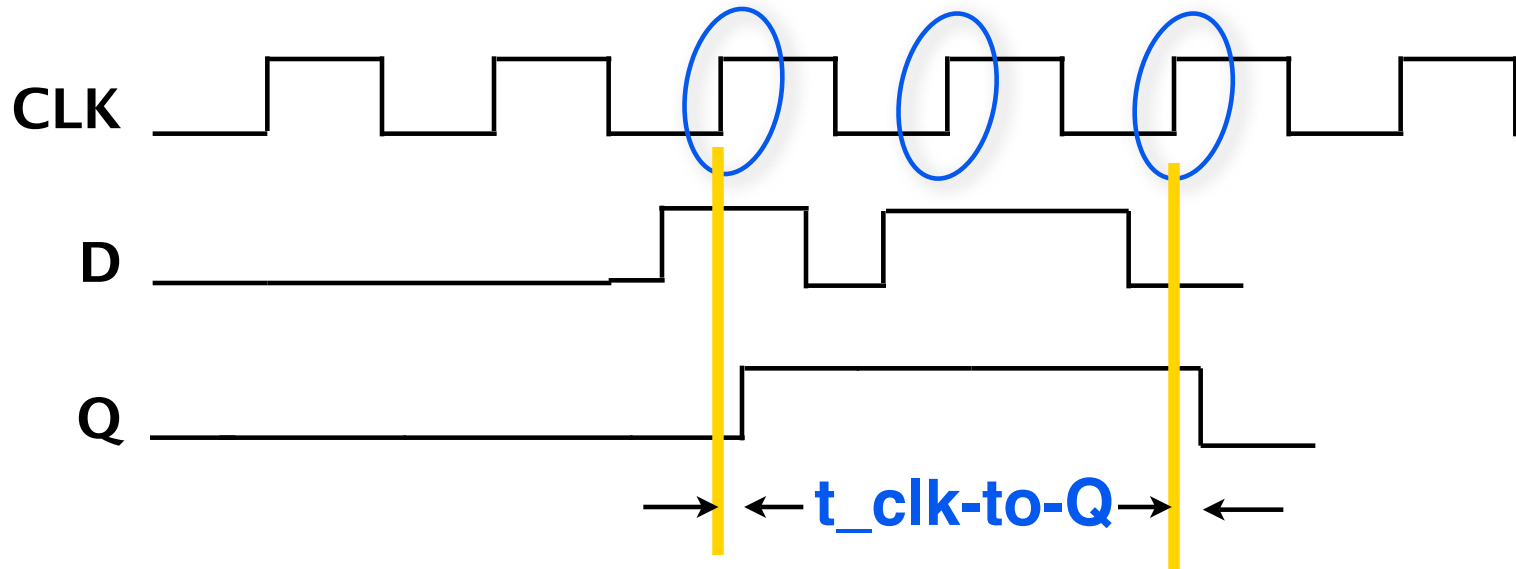
```
  r = next_r;  
  y = next_y;  
  g = next_g;  
}
```

```
}
```

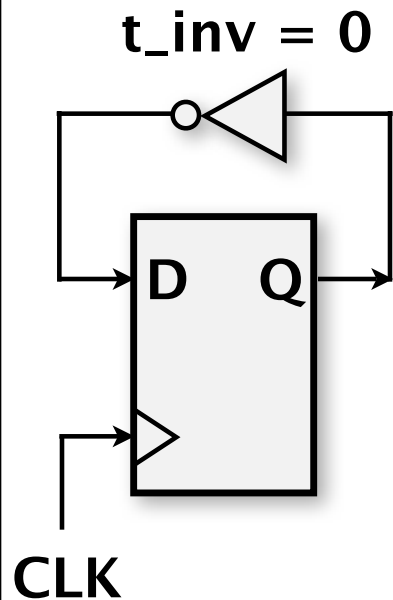
# Flip-flops have an internal output delay ...



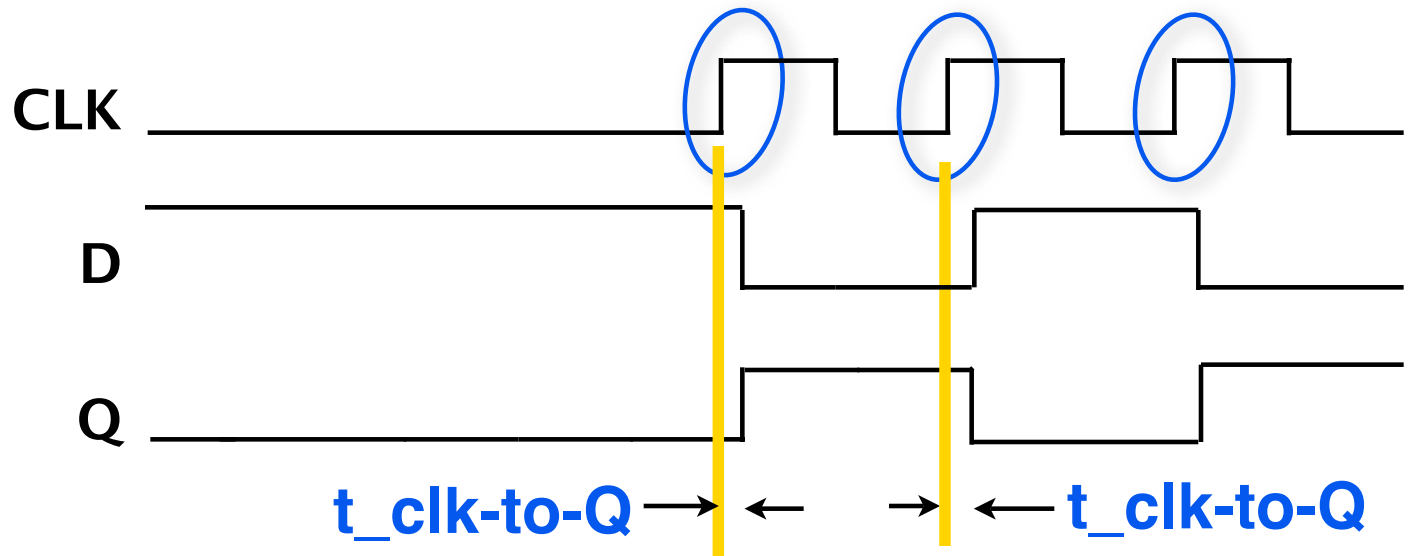
Value of  $D$  is sampled on **positive clock edge**.  
 $Q$  changes  $t_{\text{clk\_to\_Q}}$  seconds after the positive edge happens ( $t_{\text{clk\_to\_Q}} > 0$ ).



# And so, even this circuit “works” ...



Circuit “goal”: 0- $\rightarrow$ 1- $\rightarrow$ 0- $\rightarrow$ 1- $\rightarrow$  ... toggle on posedge.  
Value of  $D$  is sampled on **positive clock edge**.  
Flip-flip  $t_{clk\_to\_Q}$  delay is positive.  
Assume inverter has **no delay**! (real-world inverters always have delay).









Q can't “**race**” back to  $D$  in time to “catch” the positive edge that caused it ... and so the “next” variables **are not needed!**

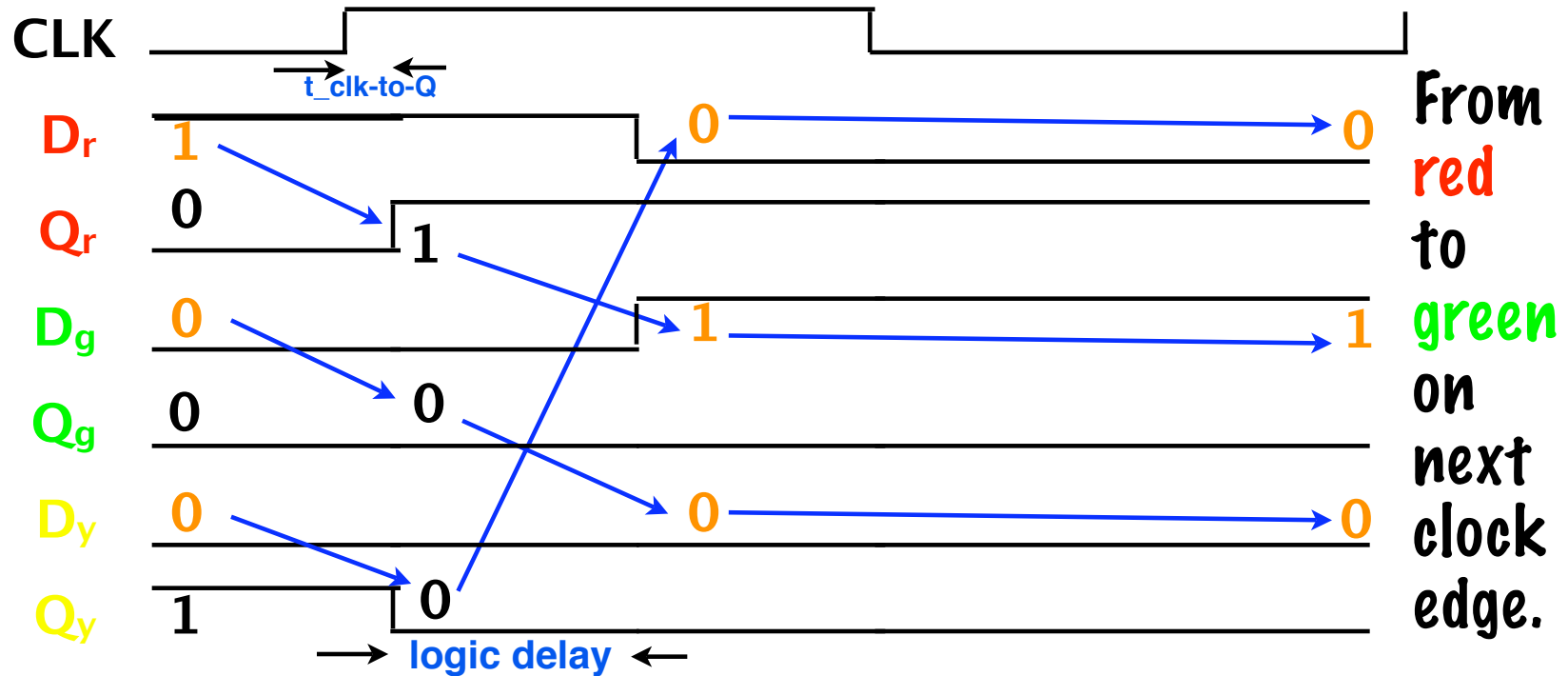
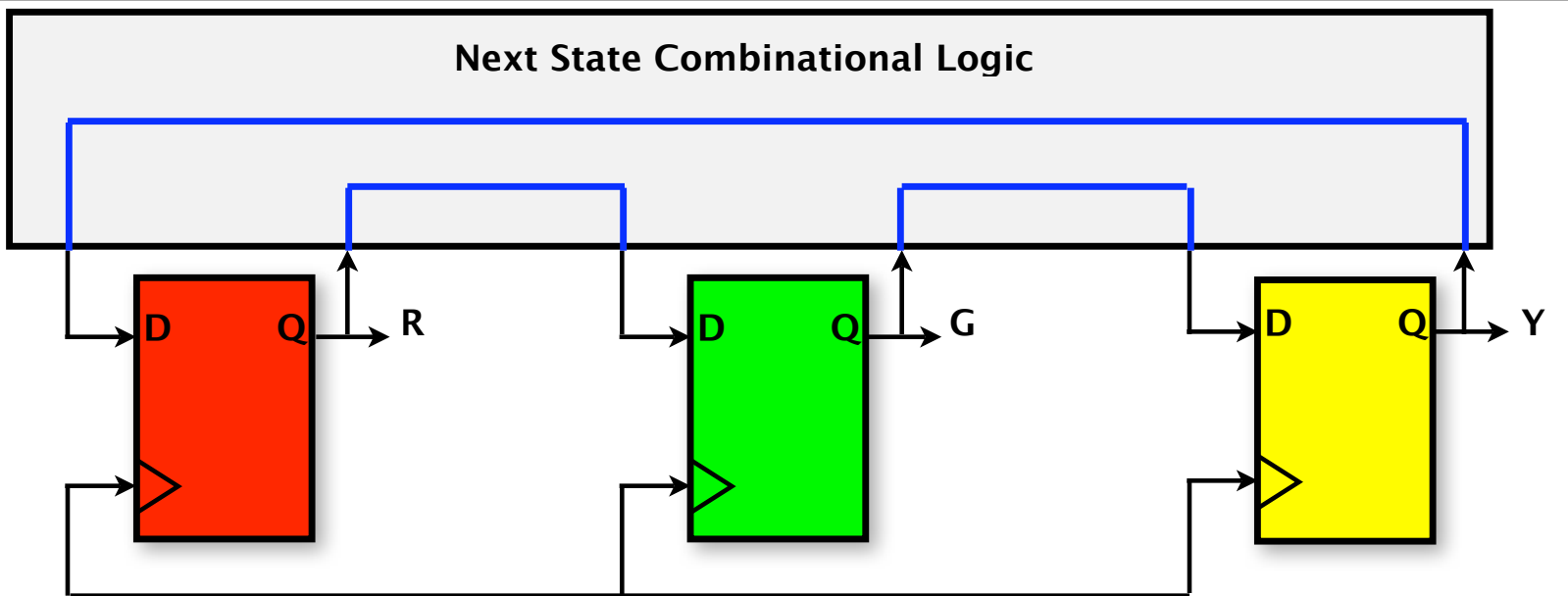


# Recall: A few observations ...

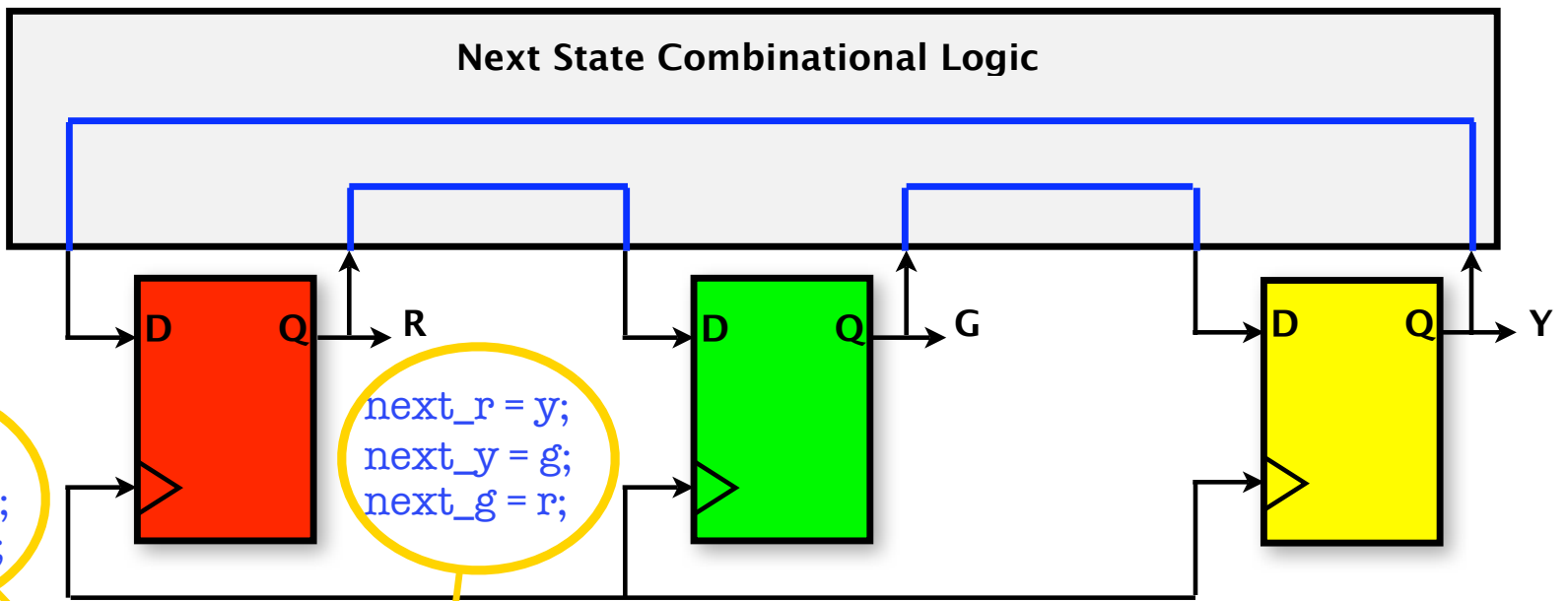
---

```
int main() {  
  
    int r = 1, y = 0, g = 0;      /* light off/on */  
    int next_r, next_y, next_g; /* extra state */  
  
    while (1)  
    {  
        next_r = y;   
        next_y = g;   
        next_g = r;   
  
        printf("r=%i\ny=%i\ng=%i\n\n", r, y, g);  
        sleep(1);  
  
        r = next_r;   
        y = next_y;   
        g = next_g;   
    }  
}
```

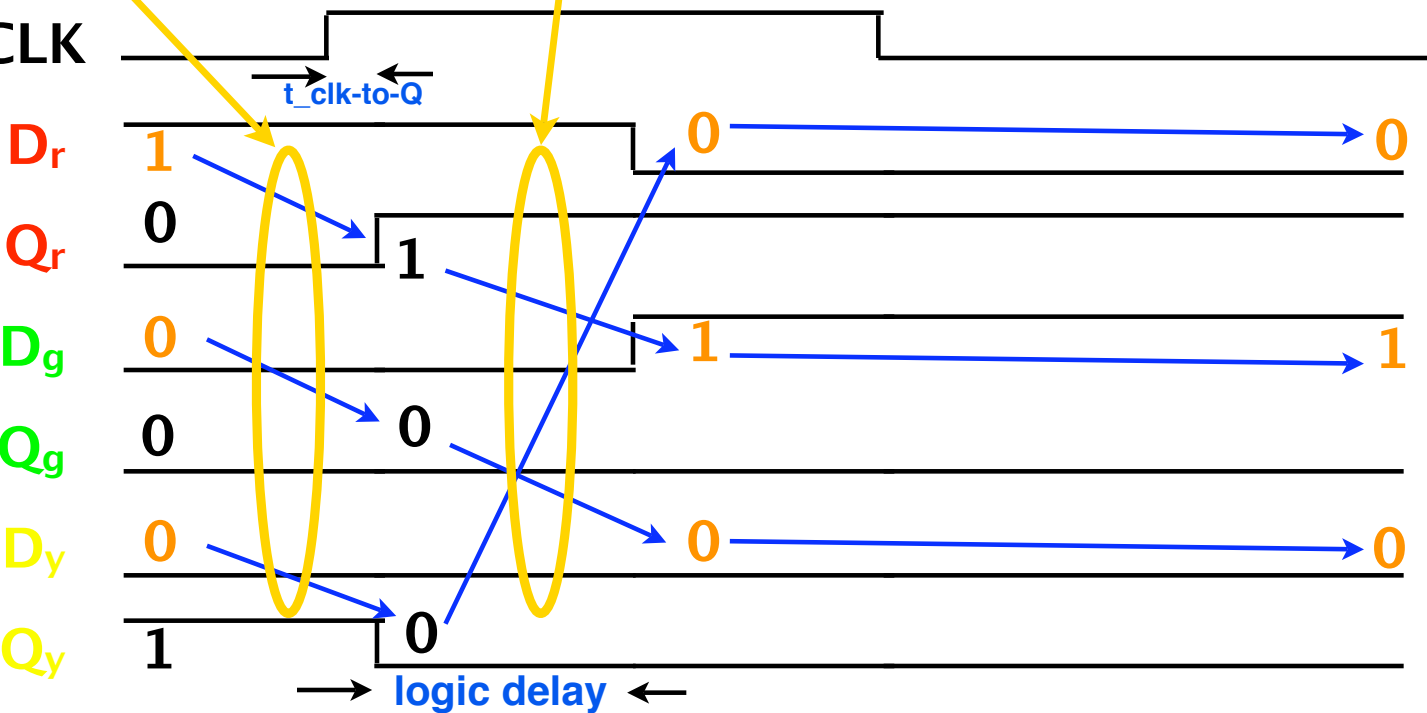
From  
yellow  
to  
red  
...



From  
yellow  
to  
red  
...



CLK

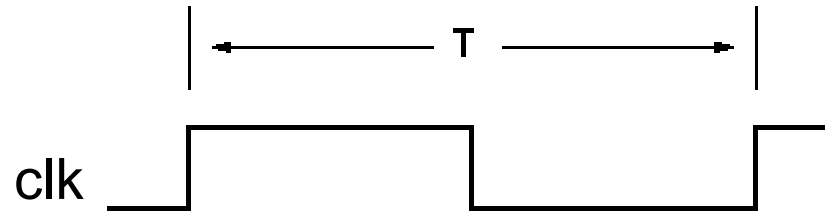


From  
red  
to  
green  
on  
next  
clock  
edge.

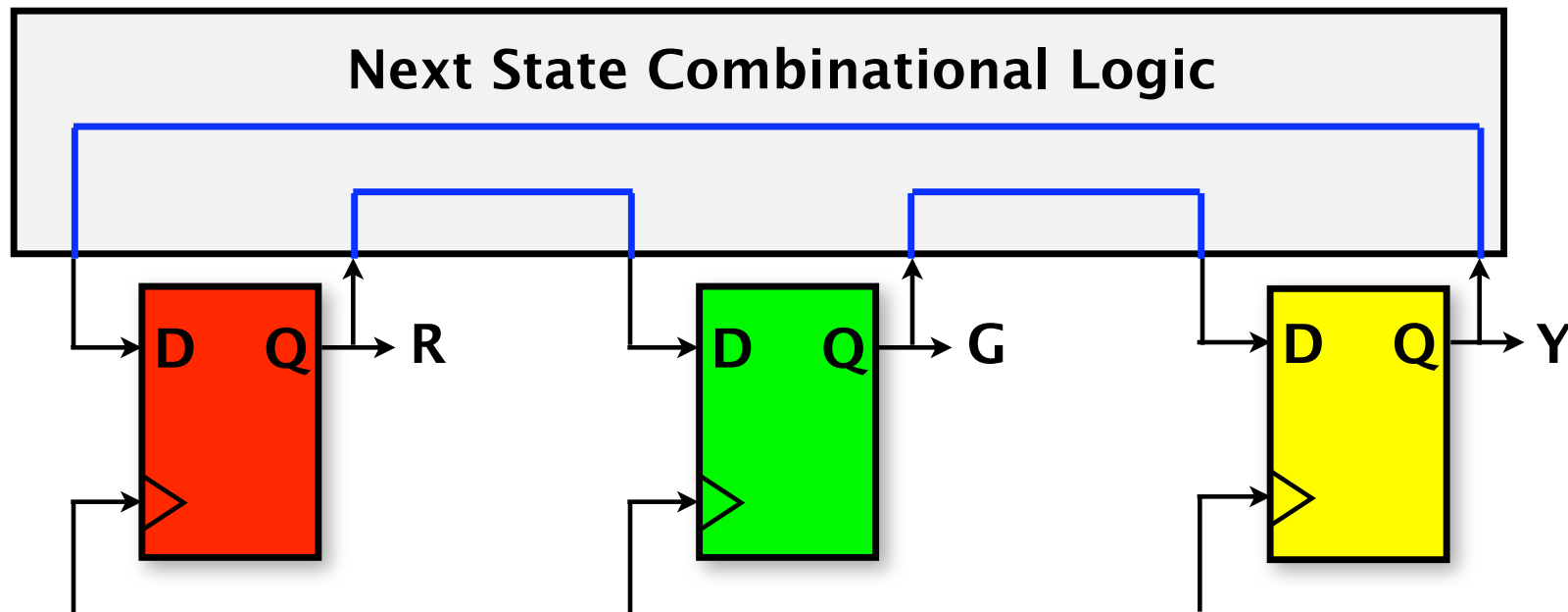
# How fast can we run the clock?

## Timing Analysis

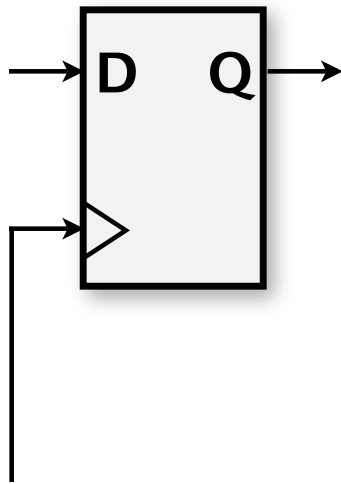
What is the smallest  $T$  that produces correct operation?



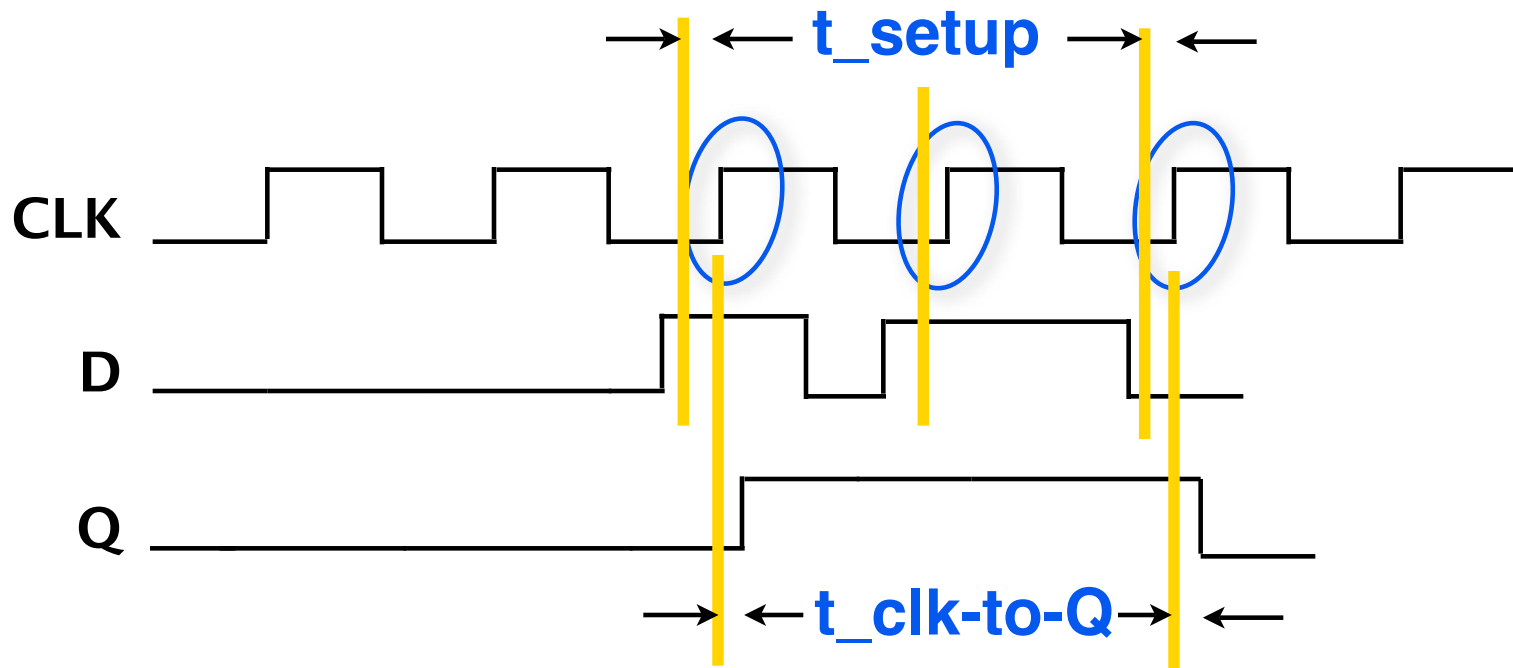
An entire lecture (later in semester), here is the short version ...



# D must stabilize ahead of positive edge

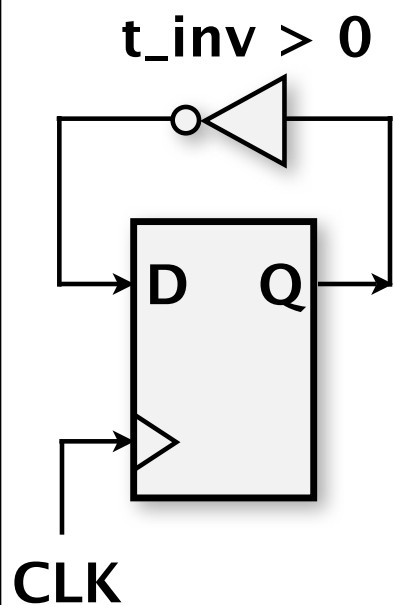


Value of  $D$  is sampled on **positive clock edge**.  
 $D$  must be a stable "0" or "1" by  **$t_{\text{setup}}$**  seconds **before** the positive edge happens.  
 $Q$  changes  **$t_{\text{clk\_to\_Q}}$**  seconds **after** the positive edge happens.

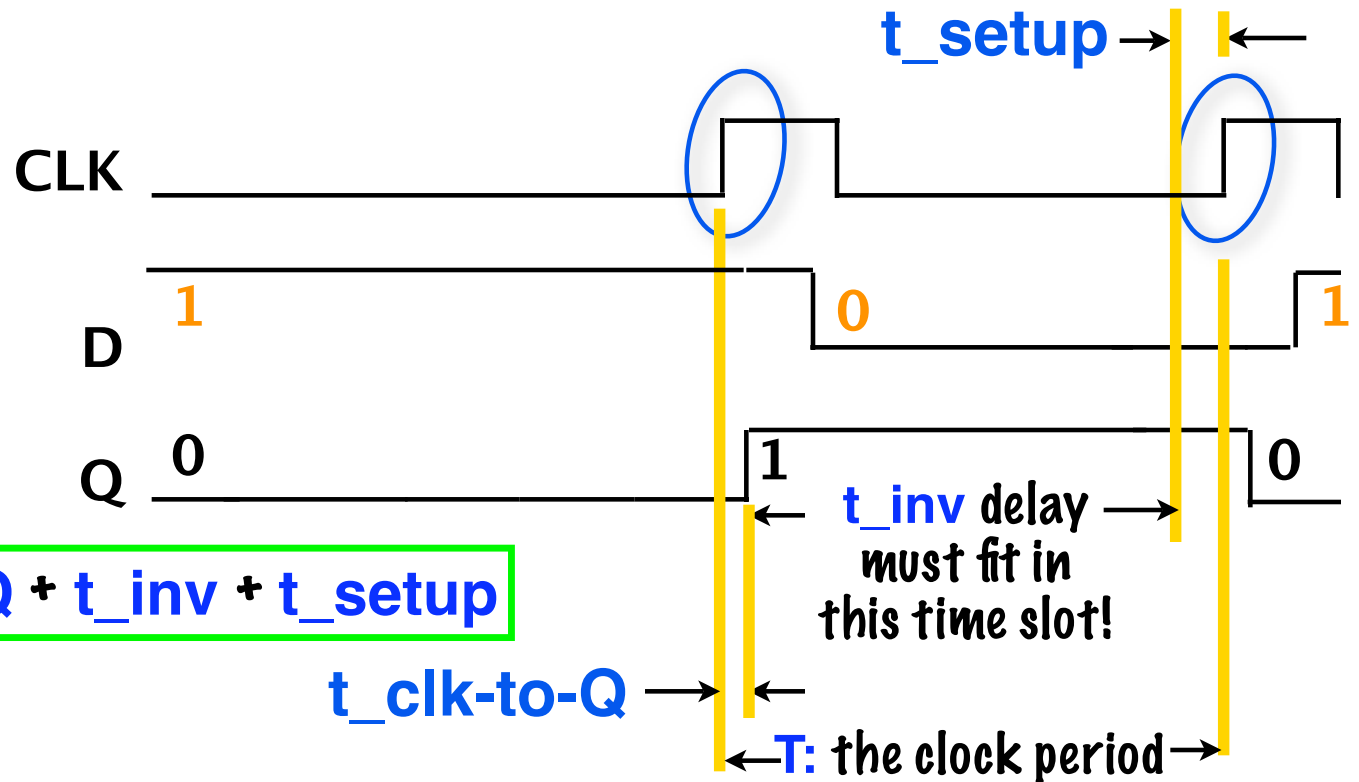




# Revisiting the toggle circuit ...

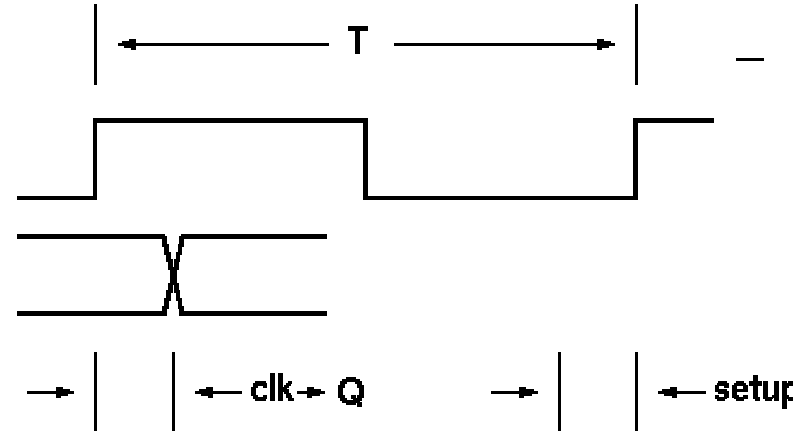
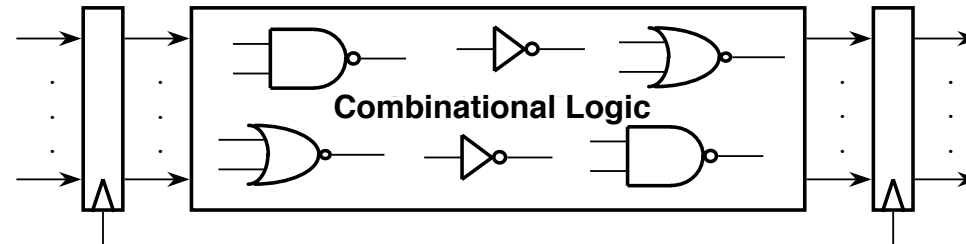


Circuit "goal": 0→1→0→1→... toggle on posedge.  
 Value of  $D$  is sampled on **positive clock edge**.  
 Flip-flip  $t_{\text{clk\_to\_Q}}$  delay is positive.  
 Inverter  $t_{\text{inv}}$  delay is positive.



$$T > t_{\text{clk\_to\_Q}} + t_{\text{inv}} + t_{\text{setup}}$$

# Or, more generally ...



**Combinational Logic (CL) “time budget”**

$$T \geq \tau_{\text{clk} \rightarrow Q} + \tau_{\text{CL}} + \tau_{\text{setup}}$$

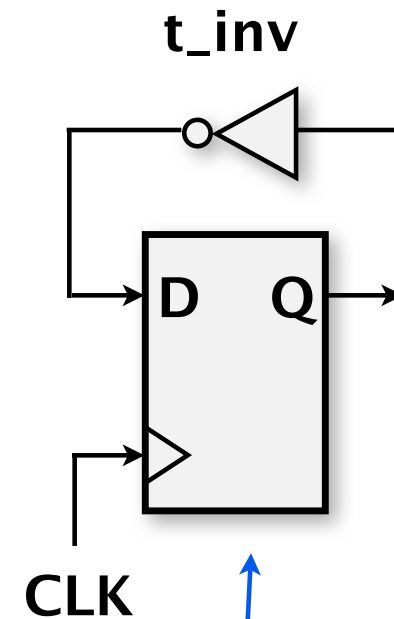
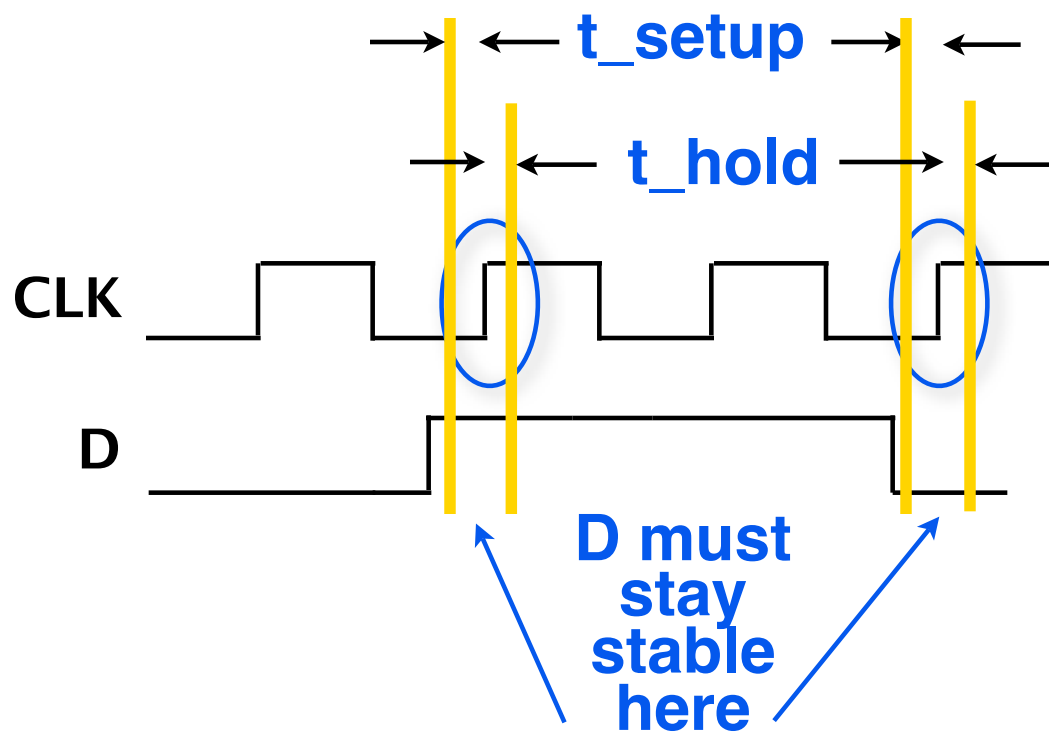


**One part of a long story, to be told later in semester ...**

EECS 150 - L4: Synch Systems II

UC Regents Spr 2010 © UCB

# Some Flip Flops have “hold” time ...

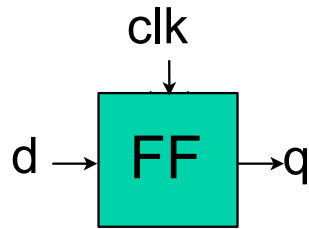


If  $t_{\text{hold}} > 0$ , this circuit may fail even if  $t_{\text{inv}} > 0$  and  $t_{\text{clk\_to\_Q}} > 0$  !!!

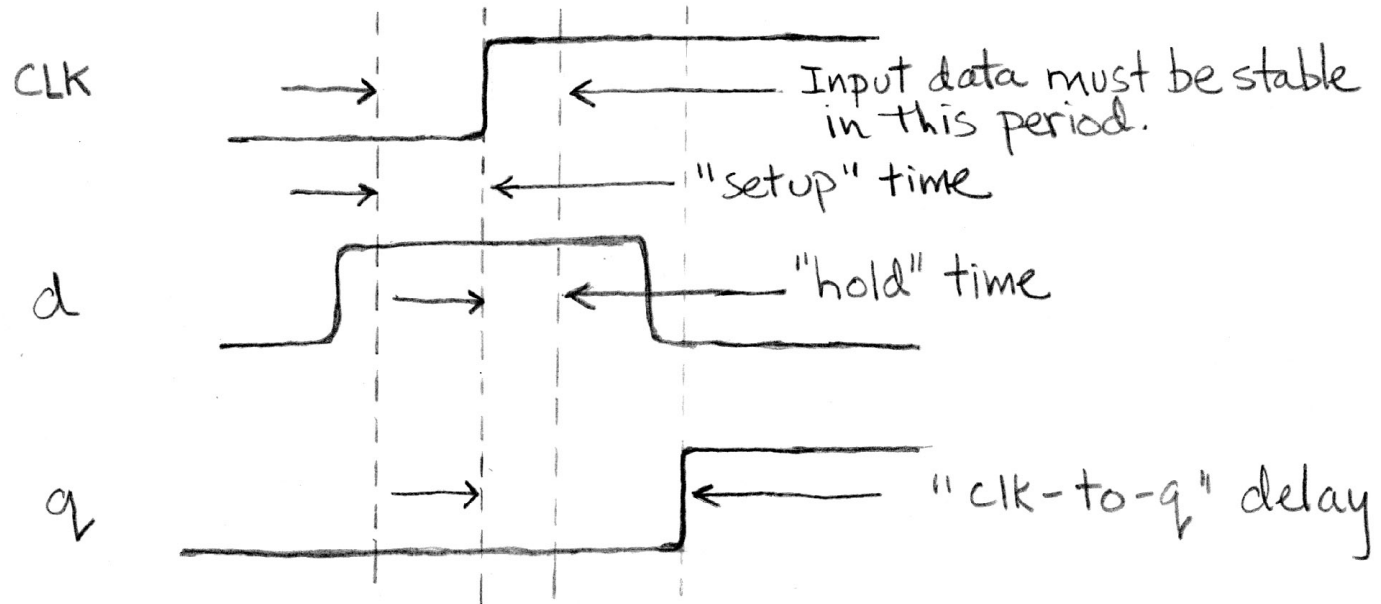
$$t_{\text{clk-to-Q}} + t_{\text{inv}} > t_{\text{hold}}$$

For correct operation.





# Flip-Flop Timing Details



Three important times associated with flip-flops:

setup time

hold time

clock-to-q delay.

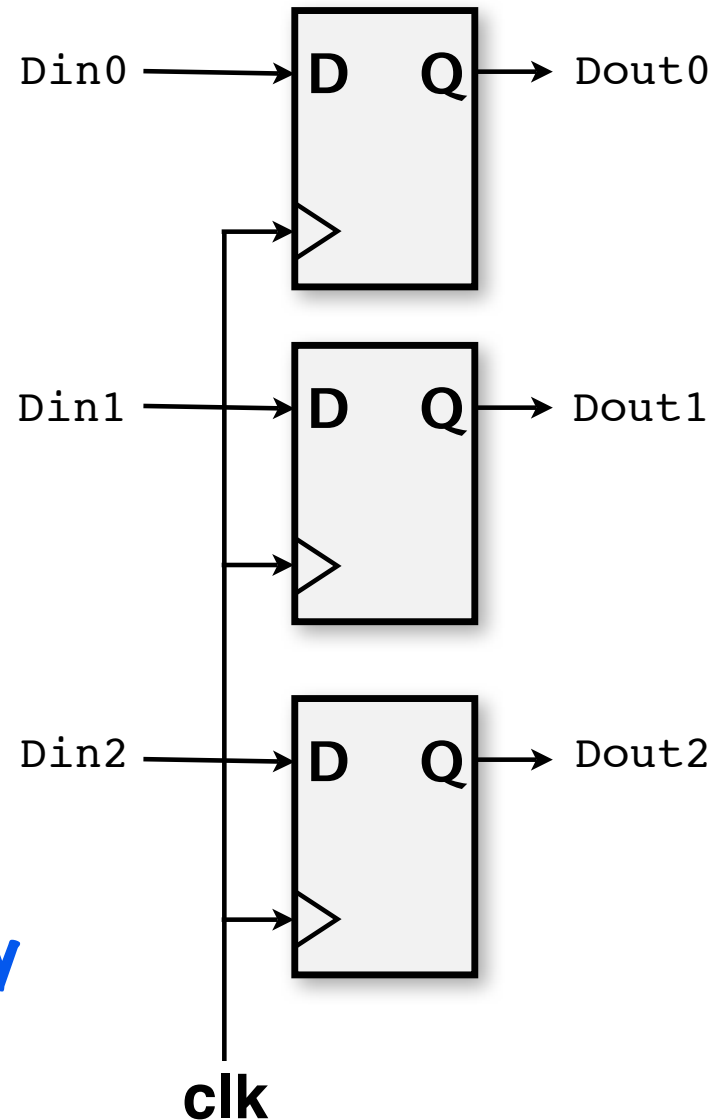
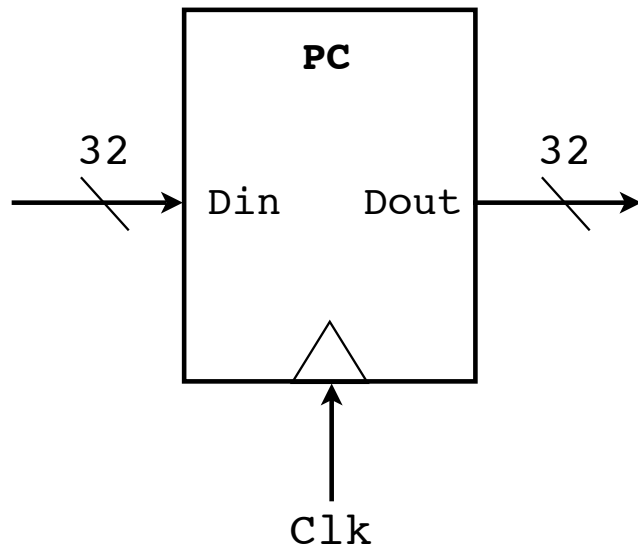
# Register State Machines

---



# Register: Holds an ordered set of bits

Built out of  
an array of  
flip-flops



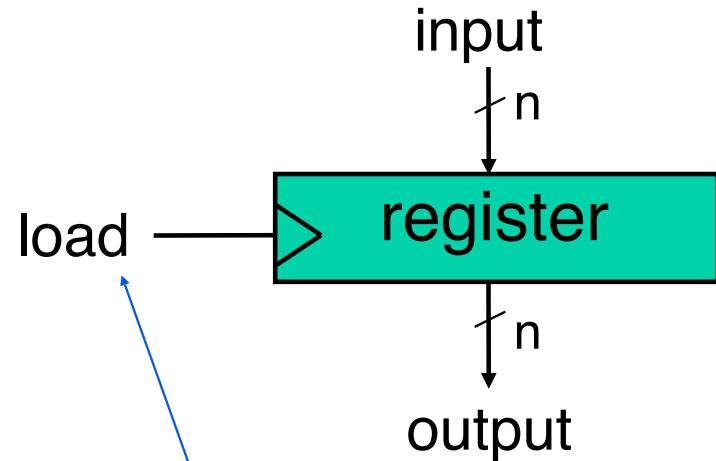
Sometimes, we will add an "enable" input: clock edge updates state only if enable is high.



# State Elements: circuits that store info

Examples: registers, memories

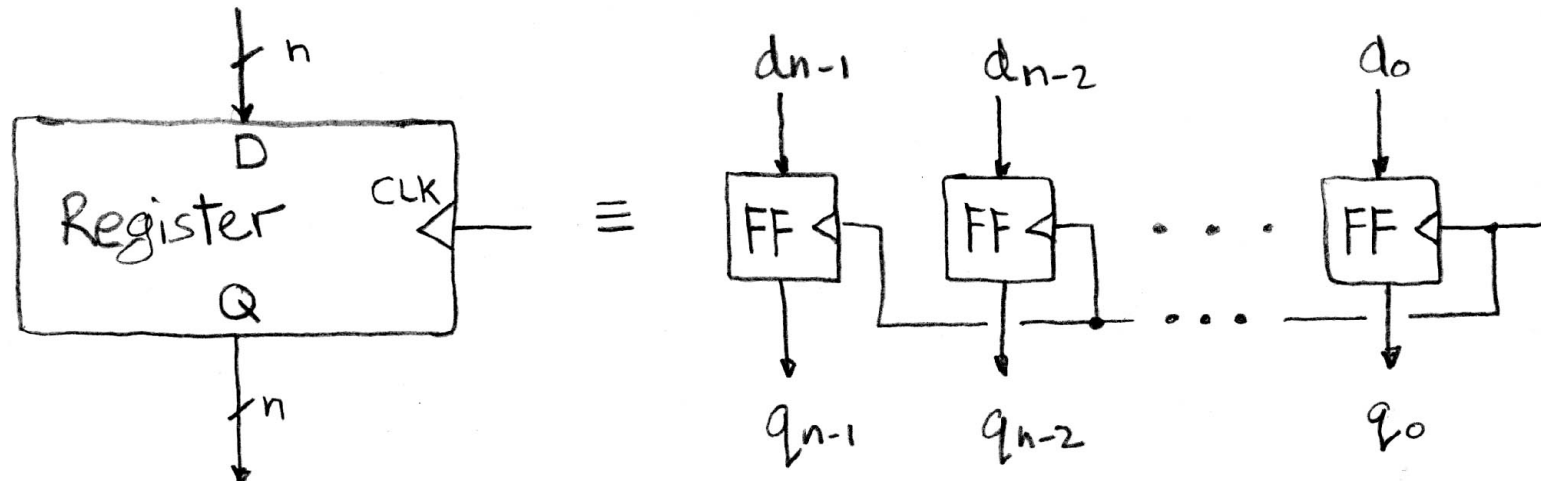
Register: Under the control of the "load" signal, the register captures the input value and stores it indefinitely.



often replace by clock signal (clk)

- The value stored by the register appears on the output (after a small delay).
- Until the next load, changes on the data input are ignored (unlike CL, where input changes change output).
- These get used for short term storage (ex: register file), and to help move data around the processor.

# Register Details...What's inside?

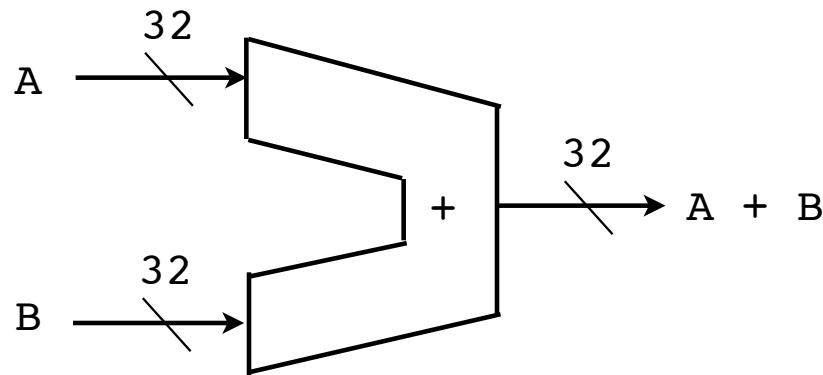


- $n$  instances of a "Flip-Flop"
- Flip-flop name because the output flips and flops between 0,1
- $D$  is "data",  $Q$  is "output"
- Also called "d-type Flip-Flop"



# Multi-bit adder: Doing logic on an integer

---

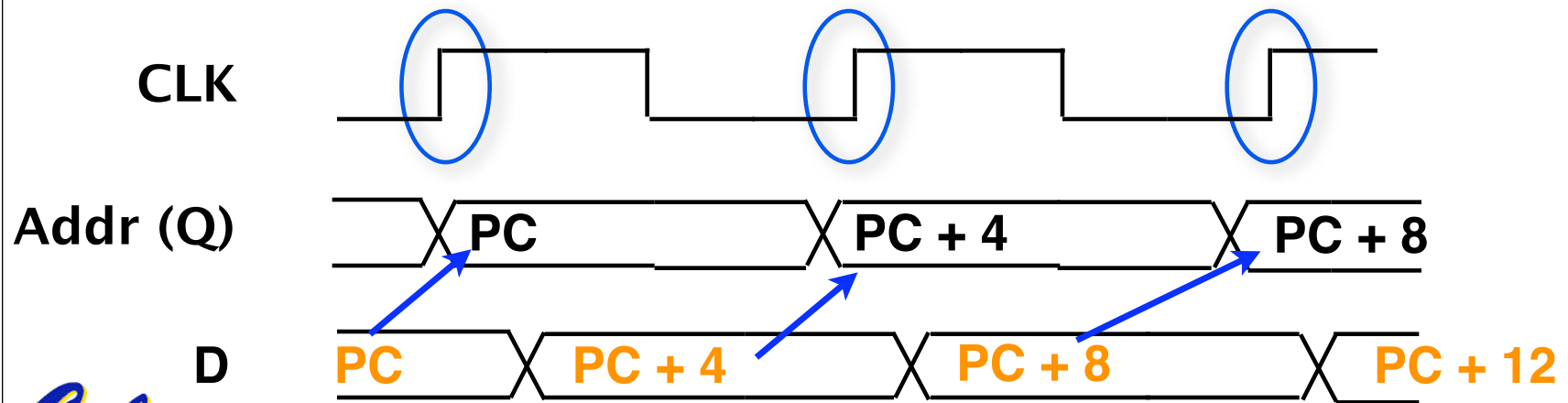
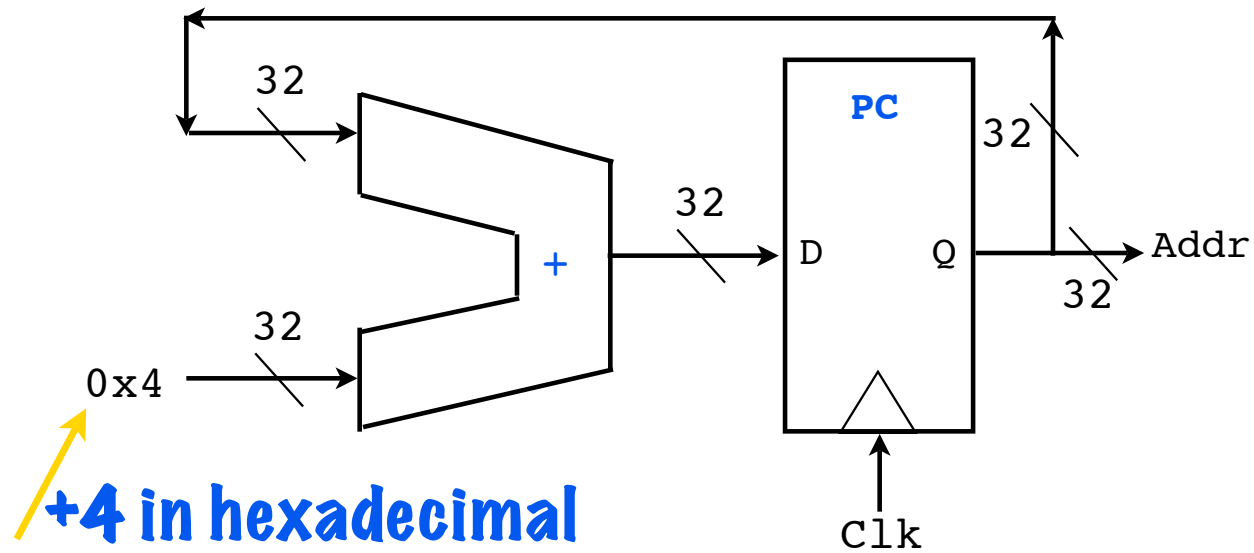


**Combinational:** Put a **A** and **B** values on inputs, a short time later **A + B** appears on output.

Just like we use gates to operate on **Q** output of a flip-flop, we use components like multi-bit adders to operate on all output bits of a register.

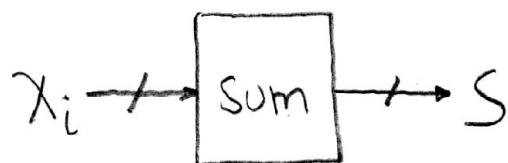


# A simple register-based state machine



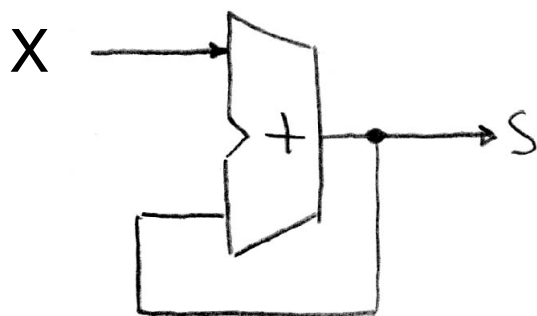
# Accumulator Circuit Example

Assume  $X$  is a vector of  $N$  integers, presented to the input of our accumulator circuit one at a time (one per clock cycle), so that after  $N$  clock cycles,  $S$  hold the sum of all  $N$  numbers.



**S=0; Repeat N times**  
**S = S + X;**

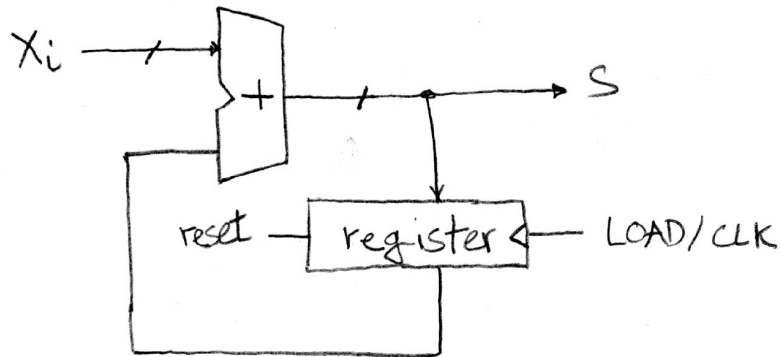
- We need something like this:



But not quite.

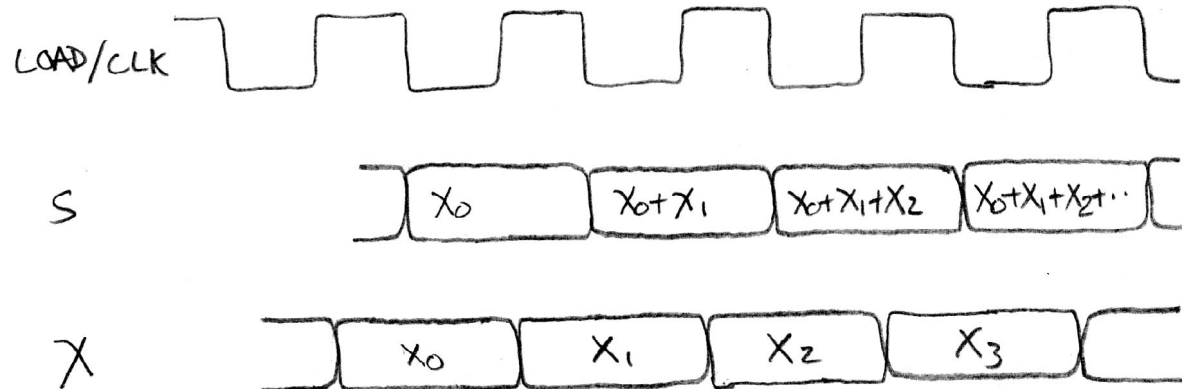
Need to use the clock signal to hold up the feedback to match up with the input signal.

# Accumulator Circuit



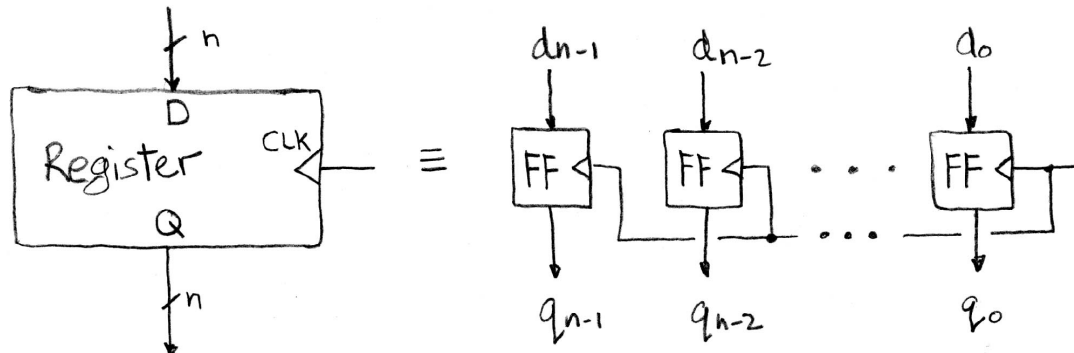
- Put register, with clock signal controlling its load, in feedback path.
- On each clock cycle the register prevents the new value from reaching the input to the adder prematurely. (The new value just waits at the input of the register).

## Timing:

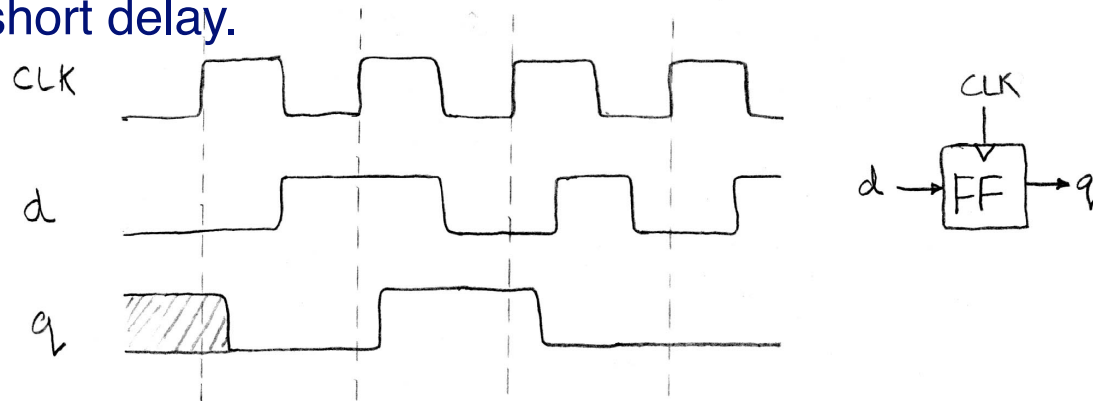


# Register Details (again)

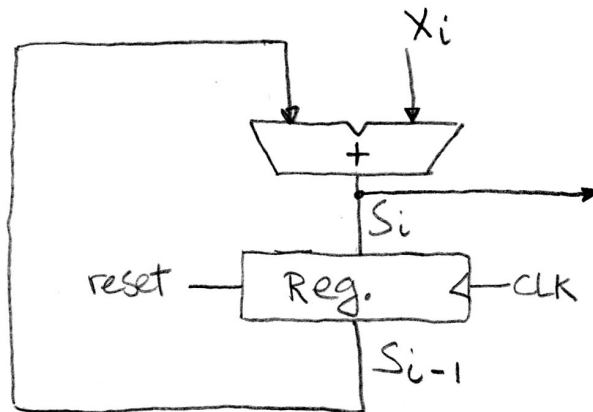
- A n-bit wide register is nothing but a set of flip-flops (1-bit wide registers) with a common load/clock signal.



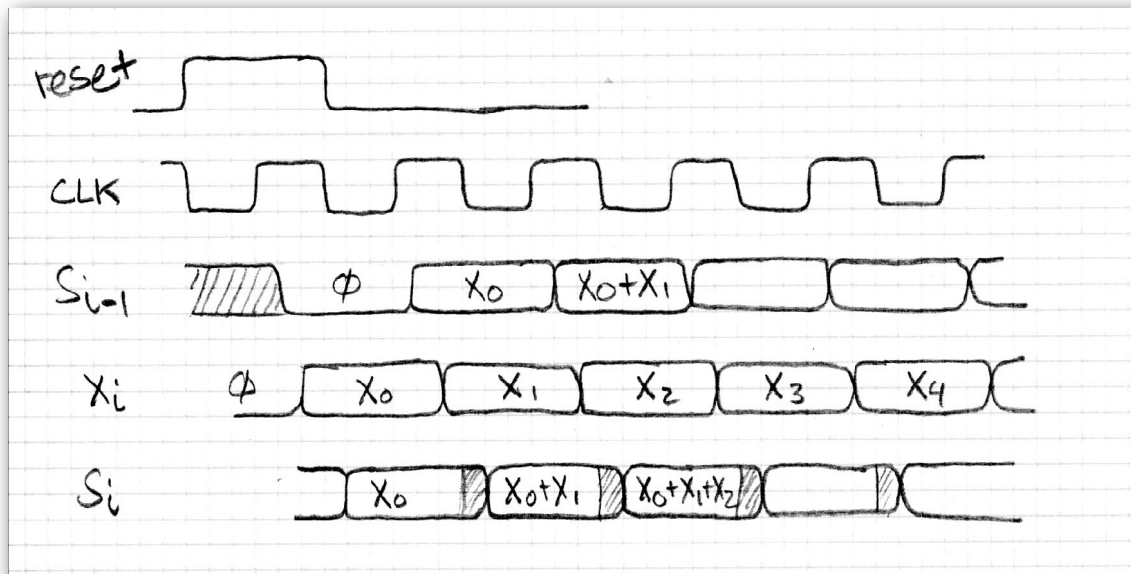
- A flip-flop captures its input on the edge of the clock (rising edge in this case - positive edge flip-flop). The new input appears at the output after a short delay.



# Accumulator Revisited

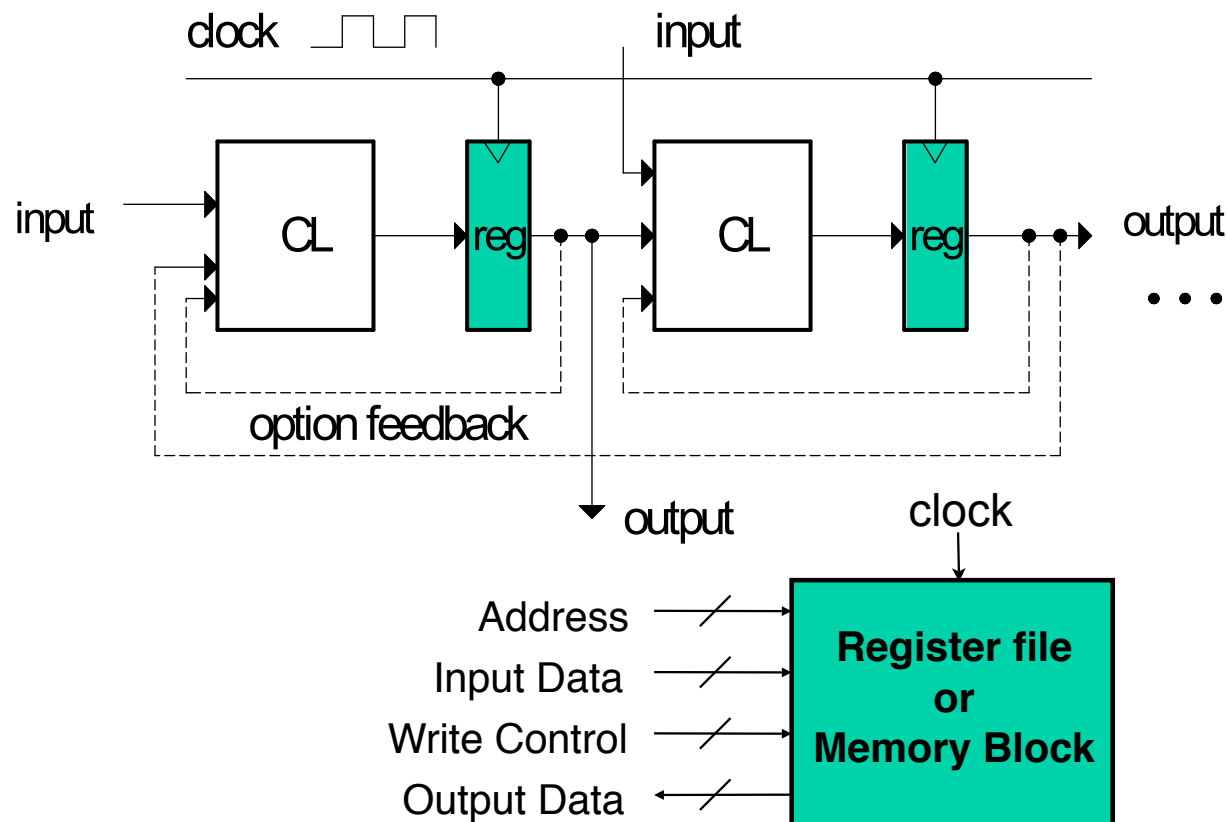


- Note:
  - Reset signal (synchronous)
  - Timing of X signal is not known without investigating the circuit that supplies X. Here we assume it comes just after  $S_{i-1}$ .
- Observe transient behavior of  $S_i$ .



# Only Two Types of Circuits Exist

- Combinational Logic Blocks (CL)
- State Elements (registers)

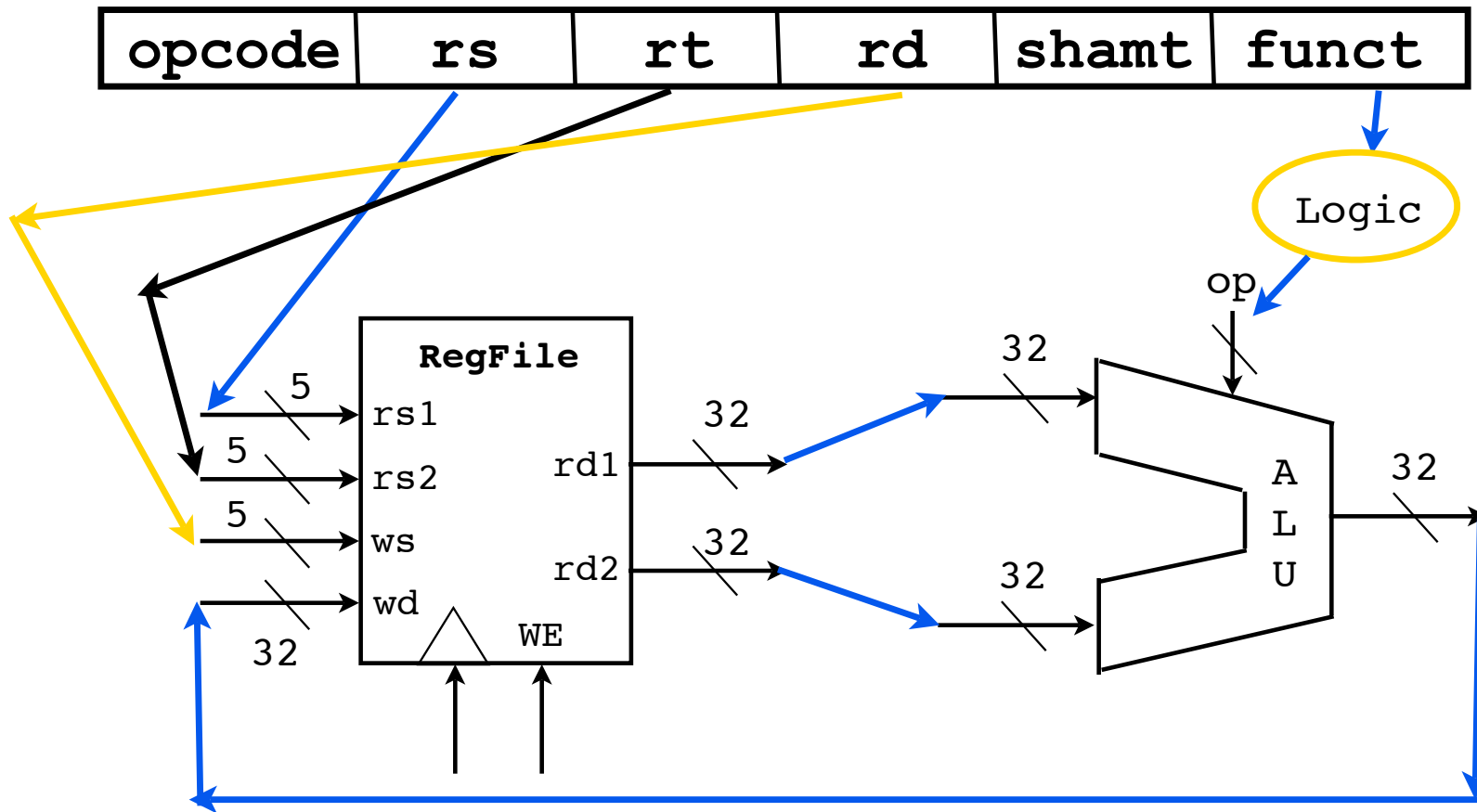


State elements are mixed in with CL blocks to control the flow of data.

Sometimes used in large groups by themselves for "long-term" data storage.

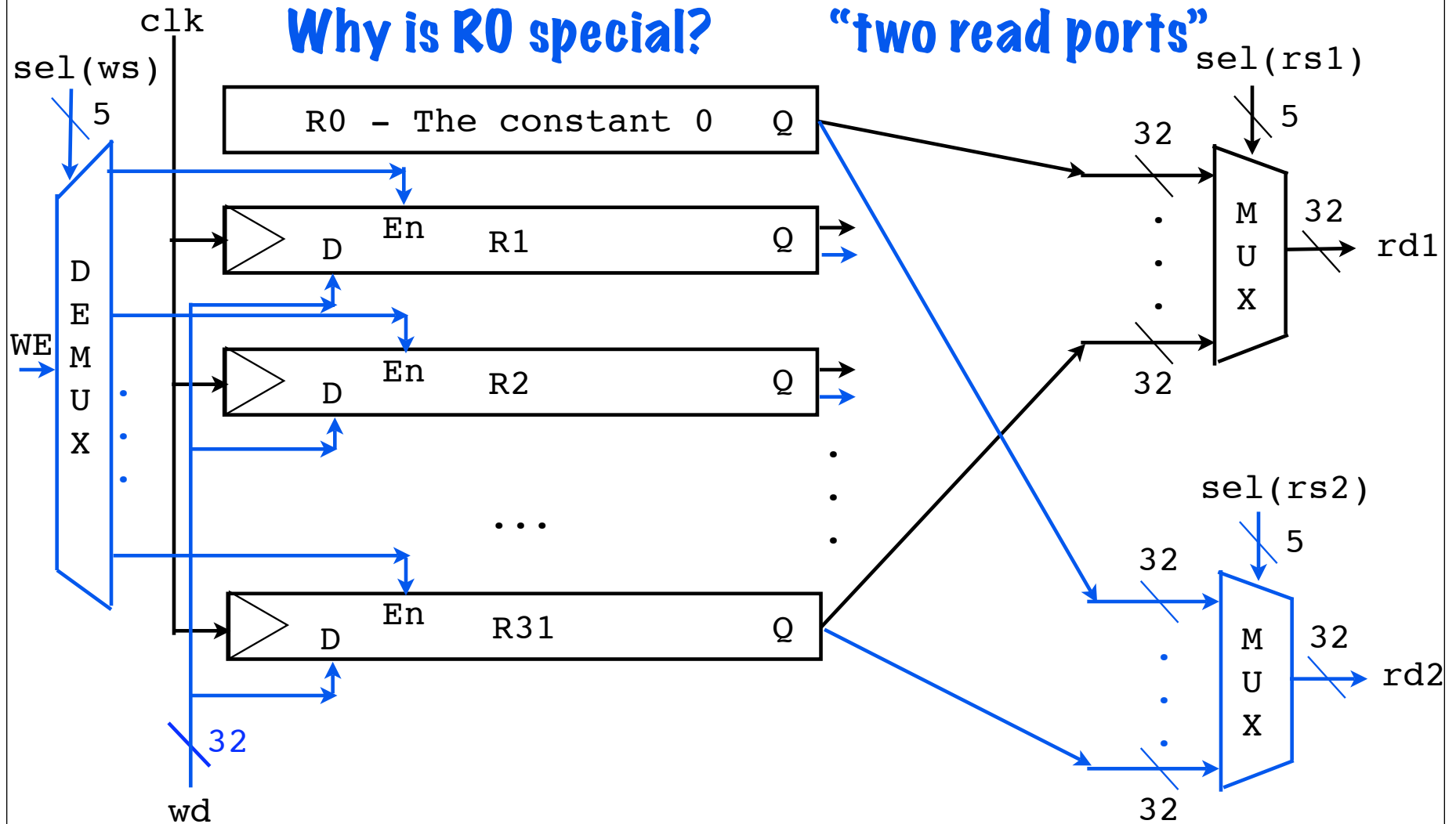
# Register File example: MIPS R-format CPU

Decode fields to get : ADD \$8 \$9 \$10





# MIPS Register file: From the top down



# Uses for State Elements

1) As a place to store values for some indeterminate amount of time:

- Register files (like \$1-\$31 on the MIPS)
- Memory (caches, and main memory)

2) Help control the flow of information between combinational logic blocks.

- State elements are used to hold up the movement of information at the inputs to combinational logic blocks and allow for orderly passage.

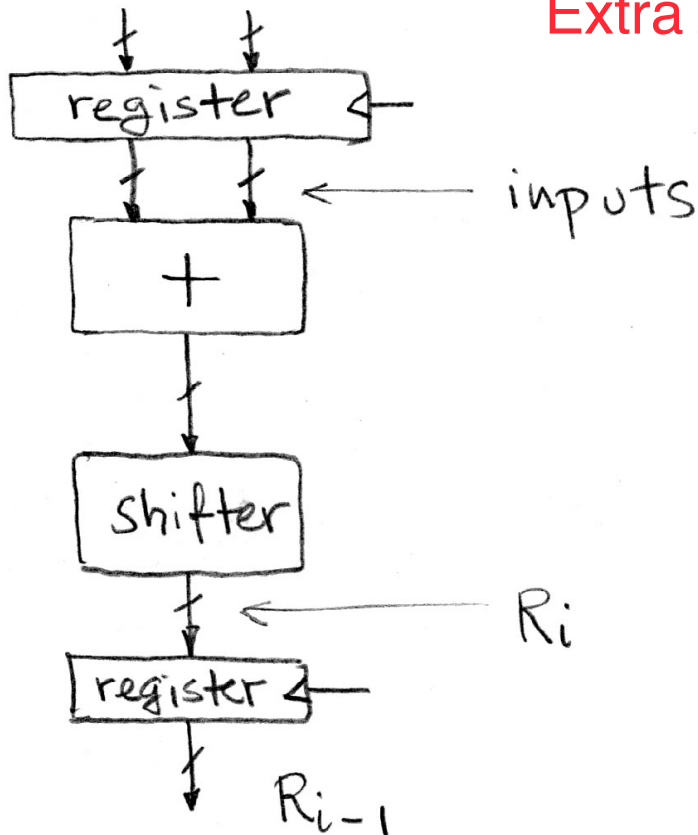
# Pipelining and Registers

---

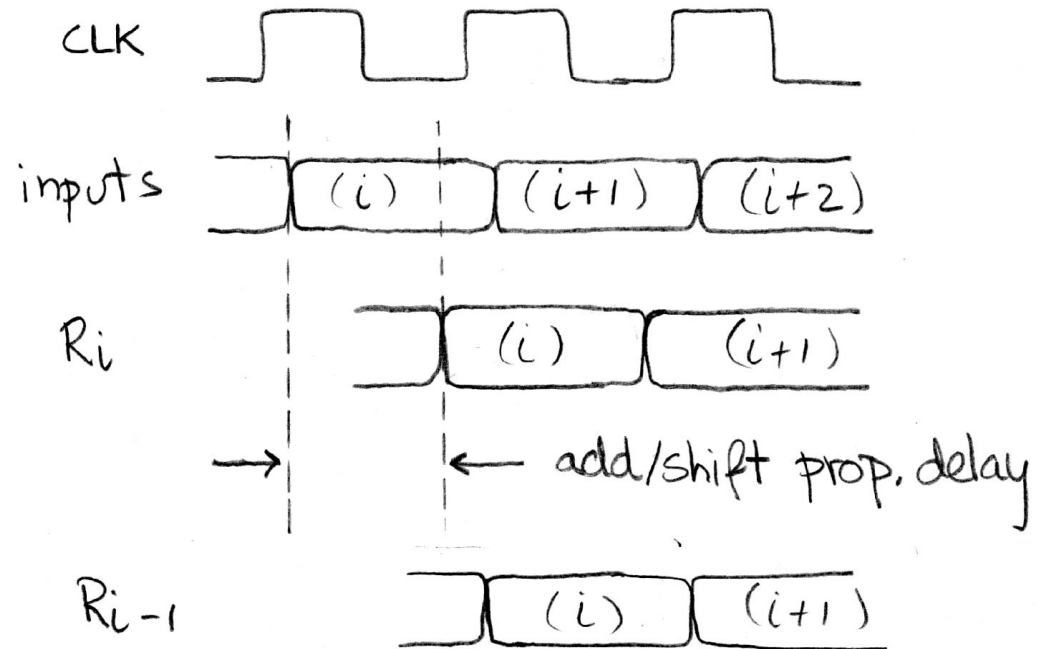


# Pipelining to improve performance (1/2)

Extra Register are often added to help speed up the clock rate.



Timing...



Note: delay of 1 clock cycle from input to output.

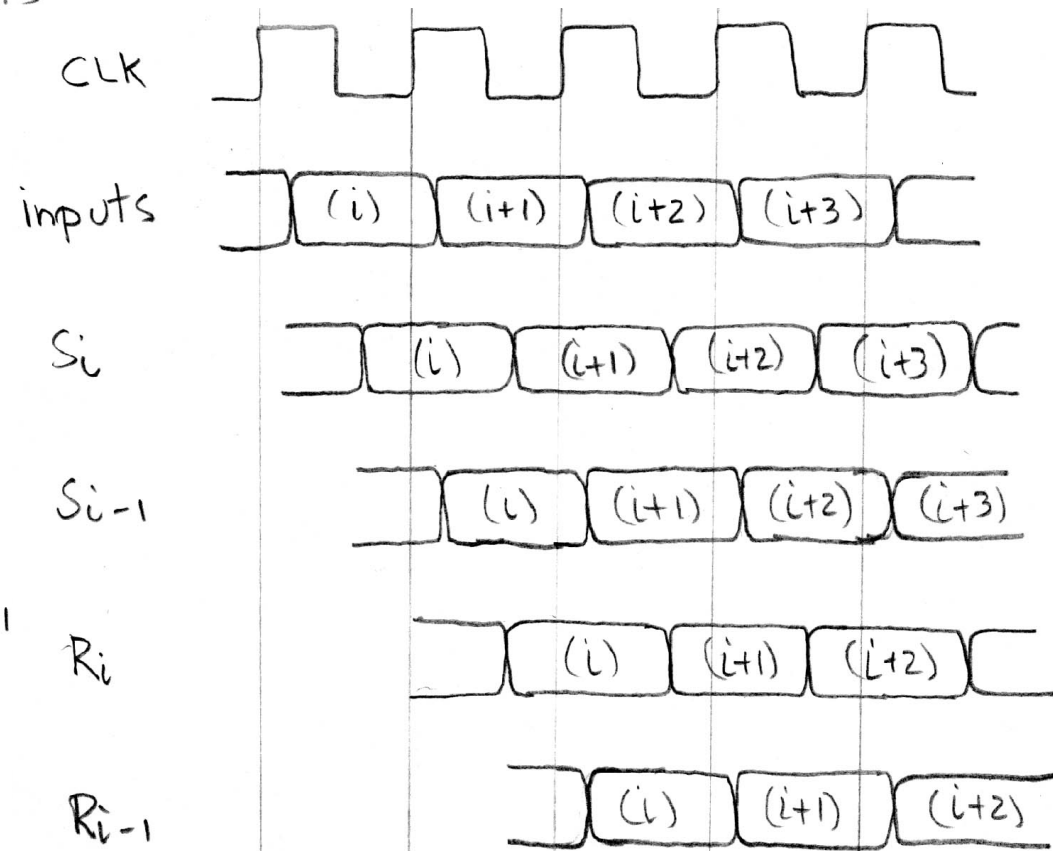
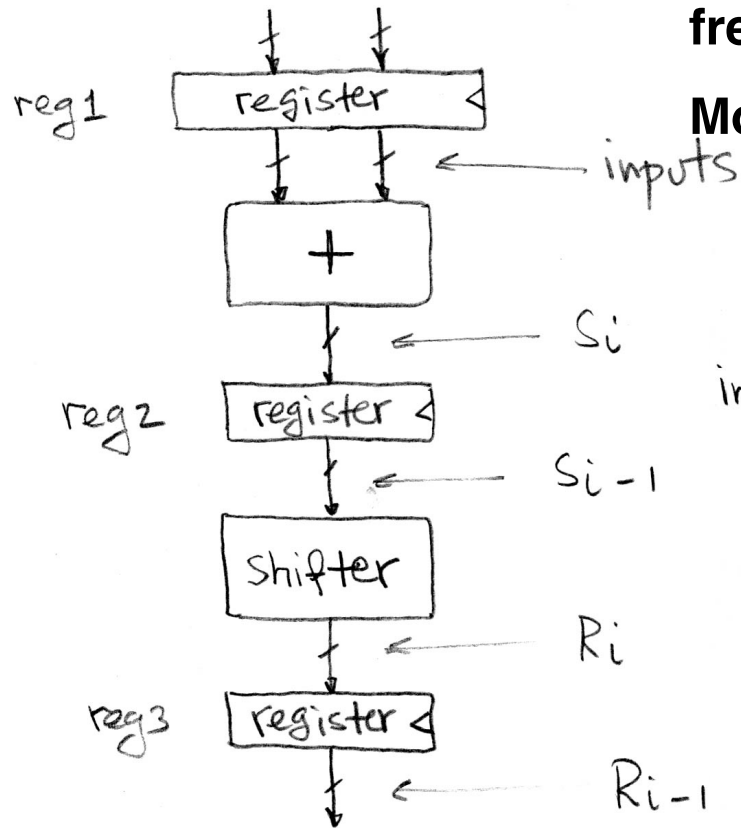
Clock period limited by propagation delay of adder/shifter.

# Pipelining to improve performance (2/2)

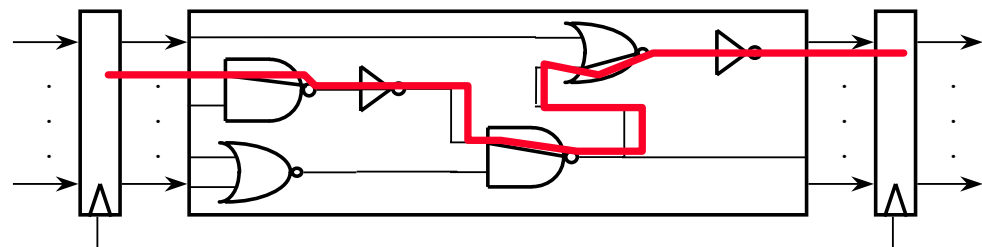
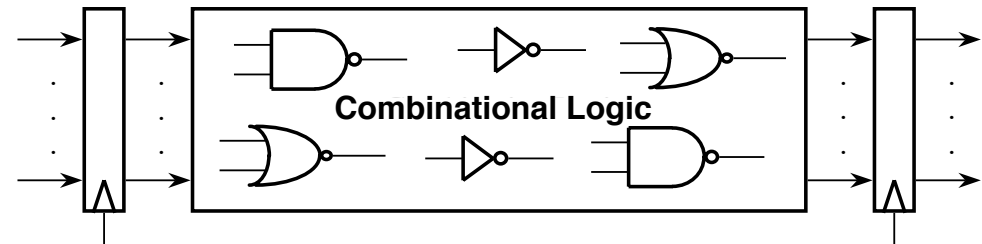
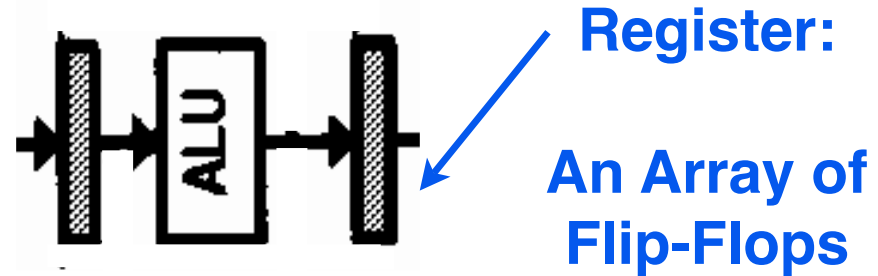
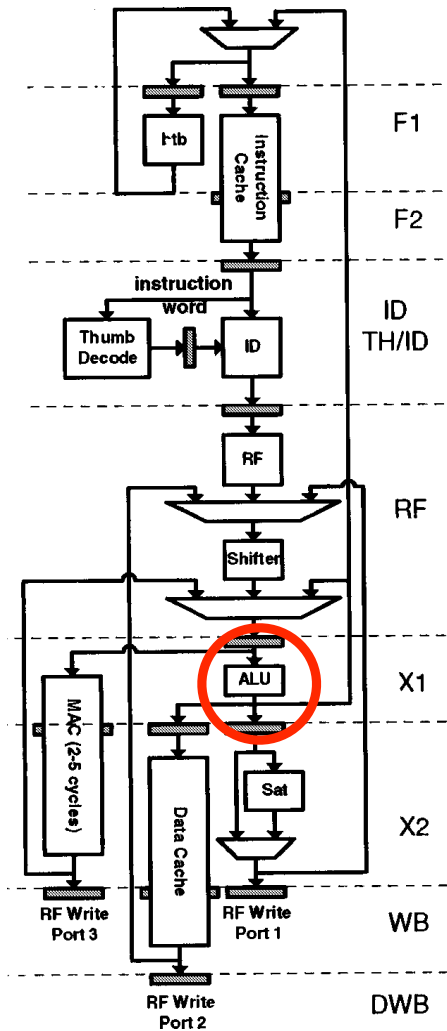
Insertion of register allows higher clock frequency.

More outputs per second.

Timing...



# Pipelining in a real CPU design ...



**Goal: Increase clock frequency by reducing delay between registers.**



# Inspiration: Automobile assembly line

Assembly line moves on a steady clock.  
Each station does the same task on each car.

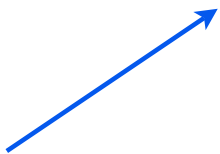
The clock

Car body shell

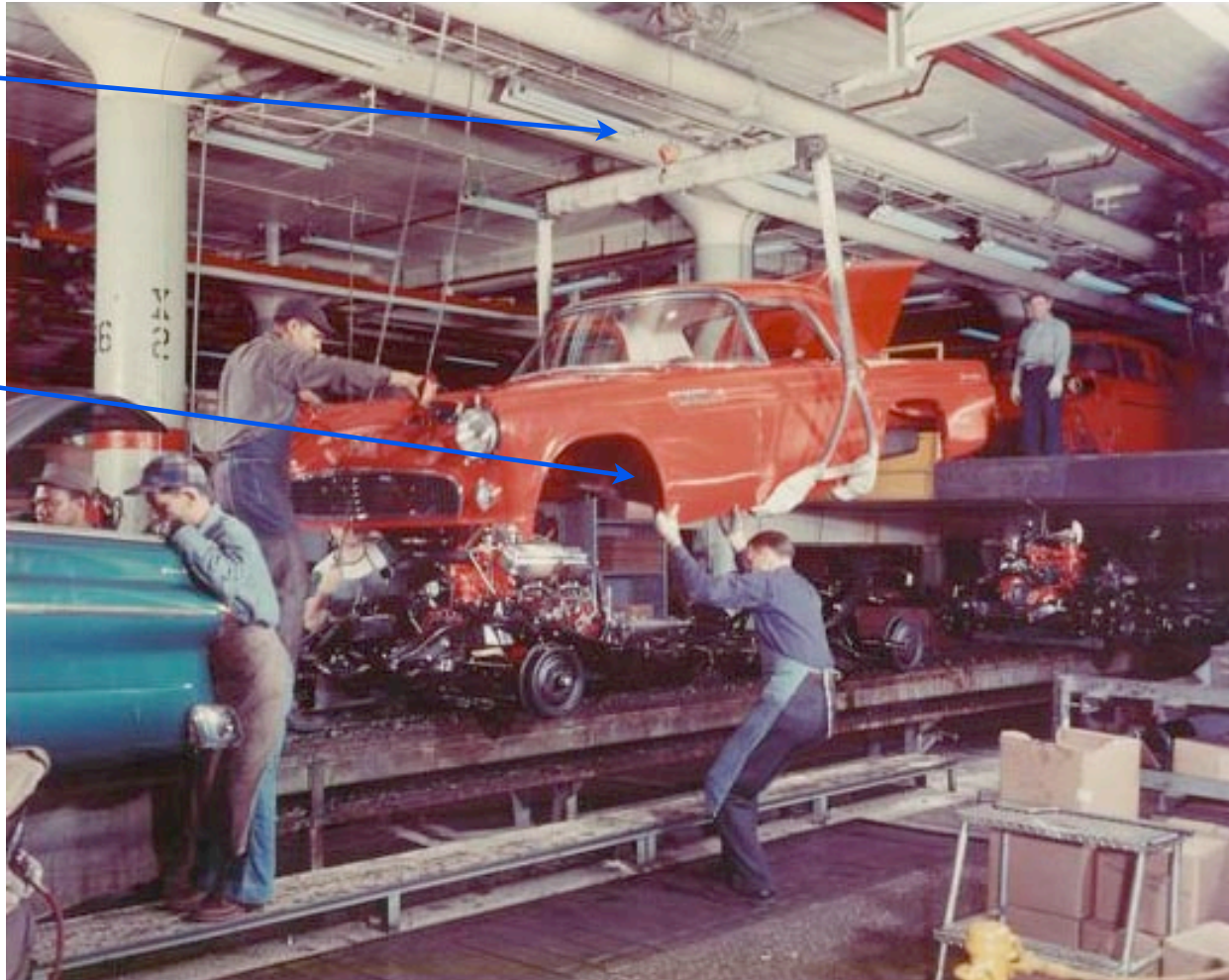
Merge station



Bolting station



Car chassis



# Inspiration: Automobile assembly line

---

**Simpler station tasks → more cars per hour.  
Simple tasks take less time, clock is faster.**





# Inspiration: Automobile assembly line

---

**Line speed limited by slowest task.  
Most efficient if all tasks take same time to do**



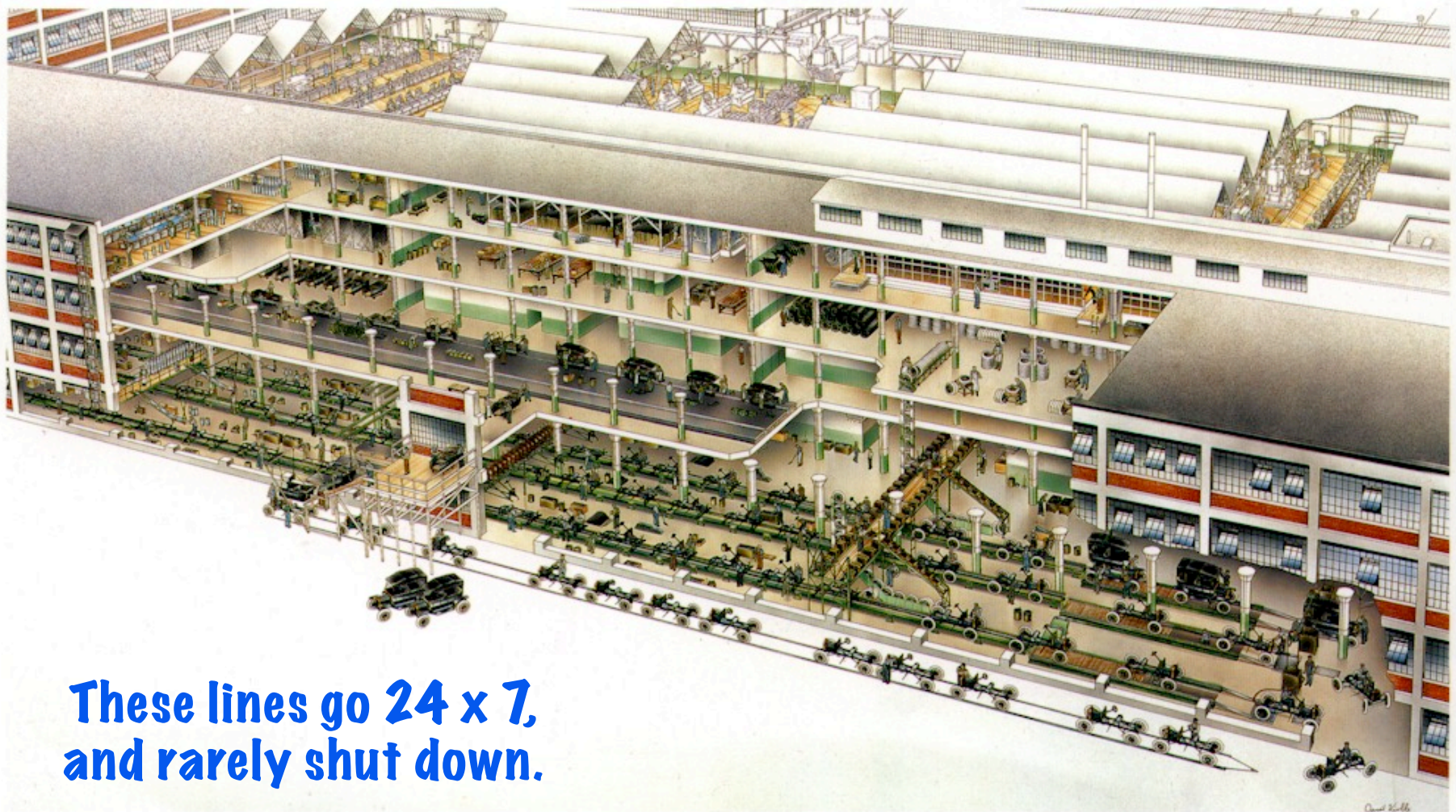
EECS 150 - L4: Synch Systems II

UC Regents Spr 2010 © UCB

# Inspiration: Automobile assembly line

---

**Simpler tasks, complex car → long line!**



# Lessons from car assembly lines

---



**Faster line movement yields more cars per hour off the line.**



**Faster line movement requires more stages, each doing simpler tasks.**



**To maximize efficiency, all stages should take same amount of time (if not, workers in fast stages are idle)**



**“Filling”, “flushing”, and “stalling” assembly line are all bad news.**

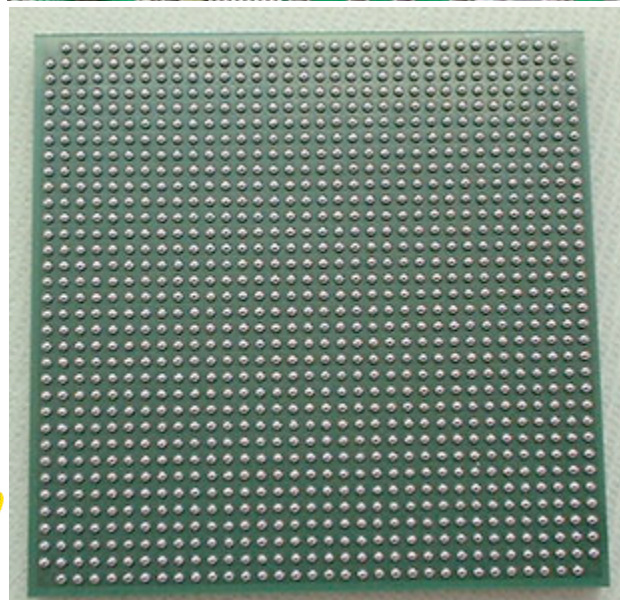


# Flip-Flop Details

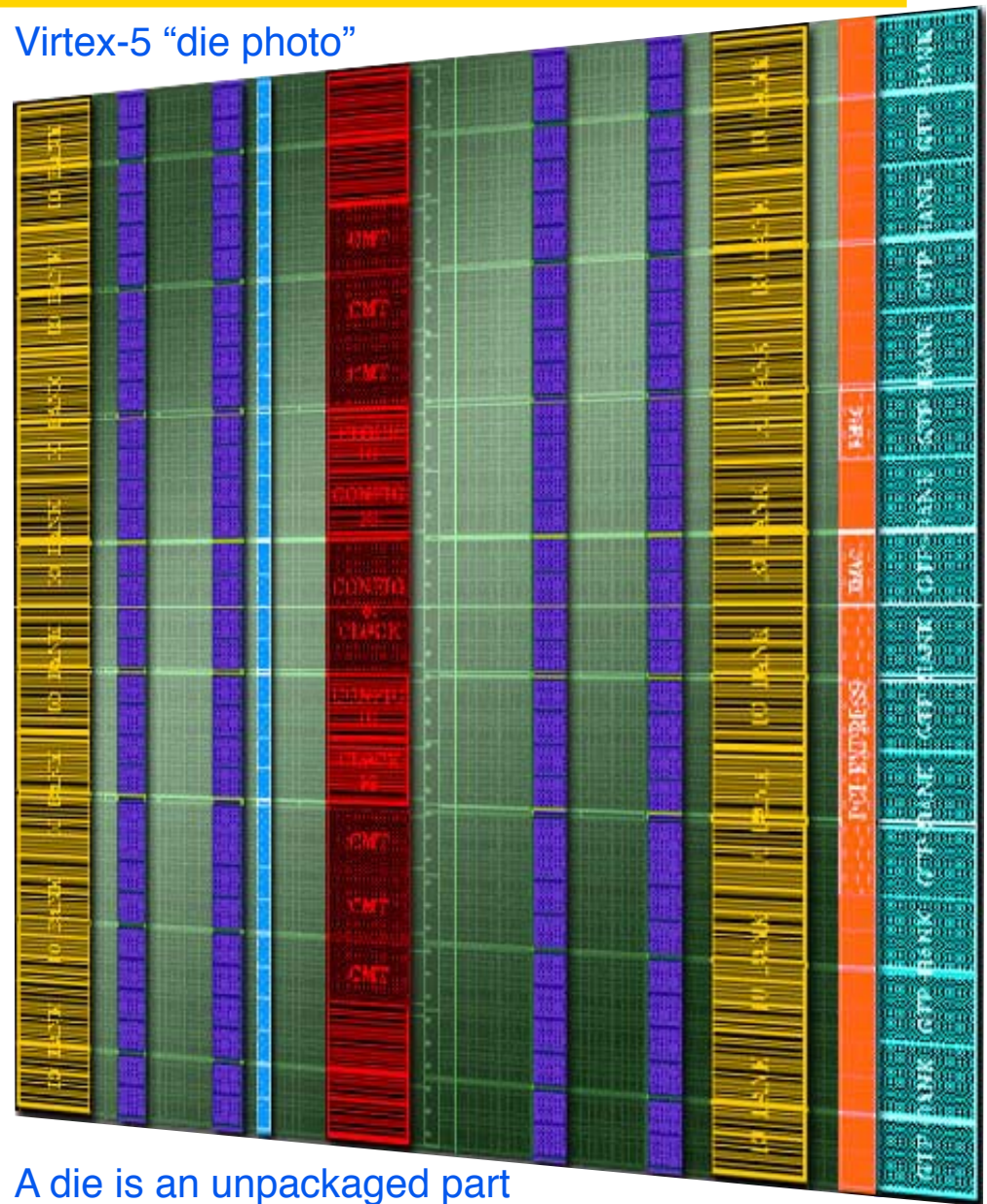
---



# FPGA: Xilinx Virtex-5 XC5VLX110T



Virtex-5 "die photo"



A die is an unpackaged part



Colors represent different types of resources:

Logic

Block RAM

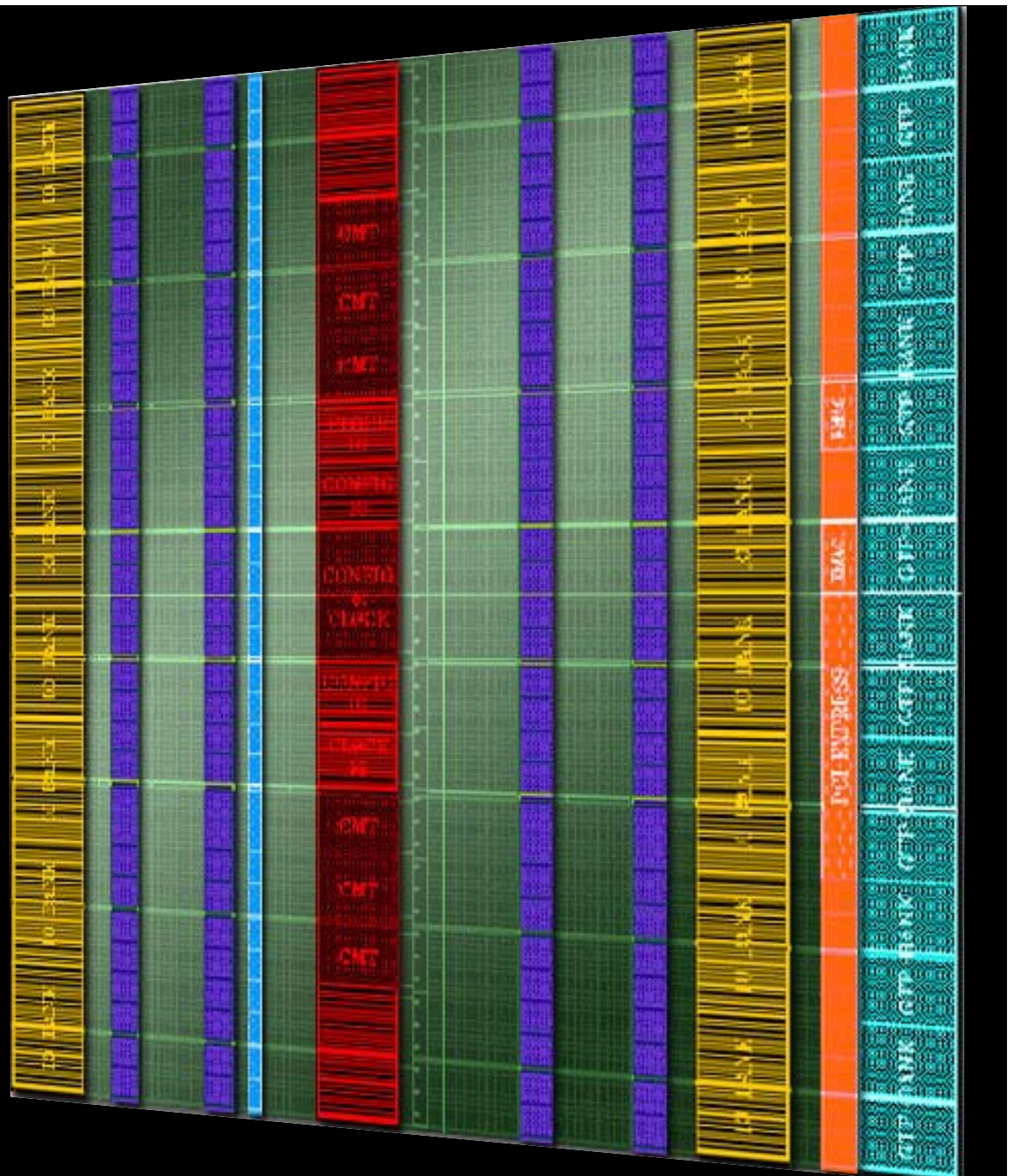
DSP (ALUs)

Clocking

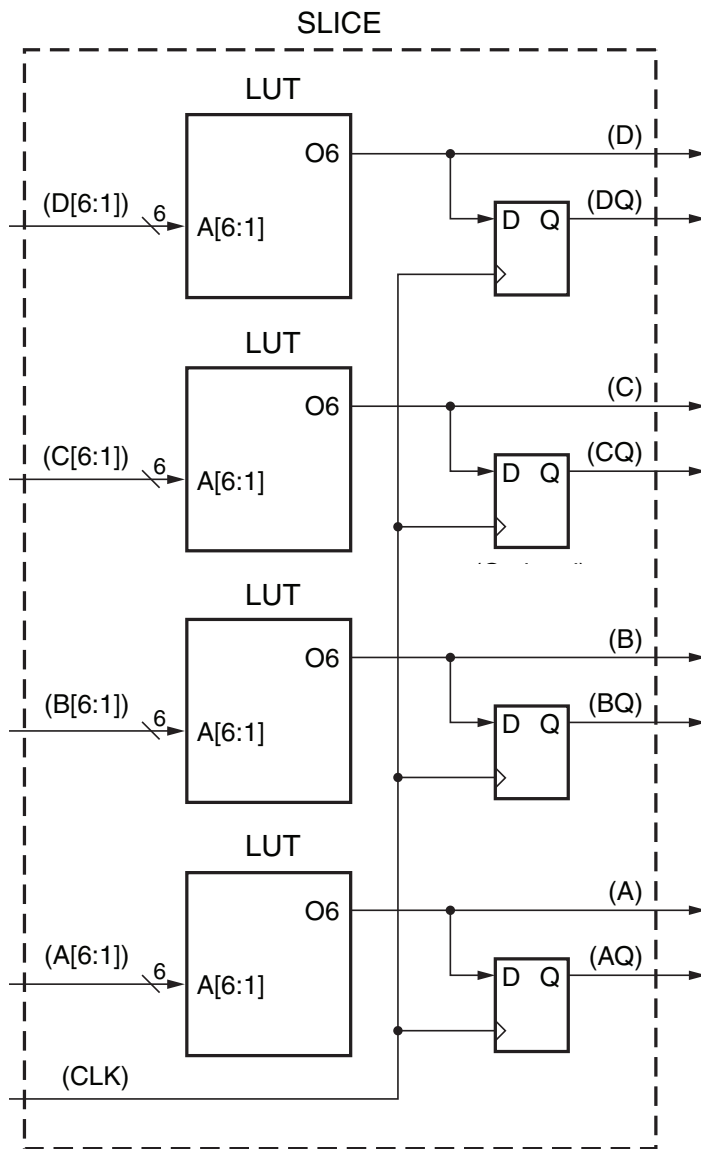
I/O

Serial I/O + PCI

A routing fabric runs throughout the chip to wire everything together.



# The simplest view of a slice

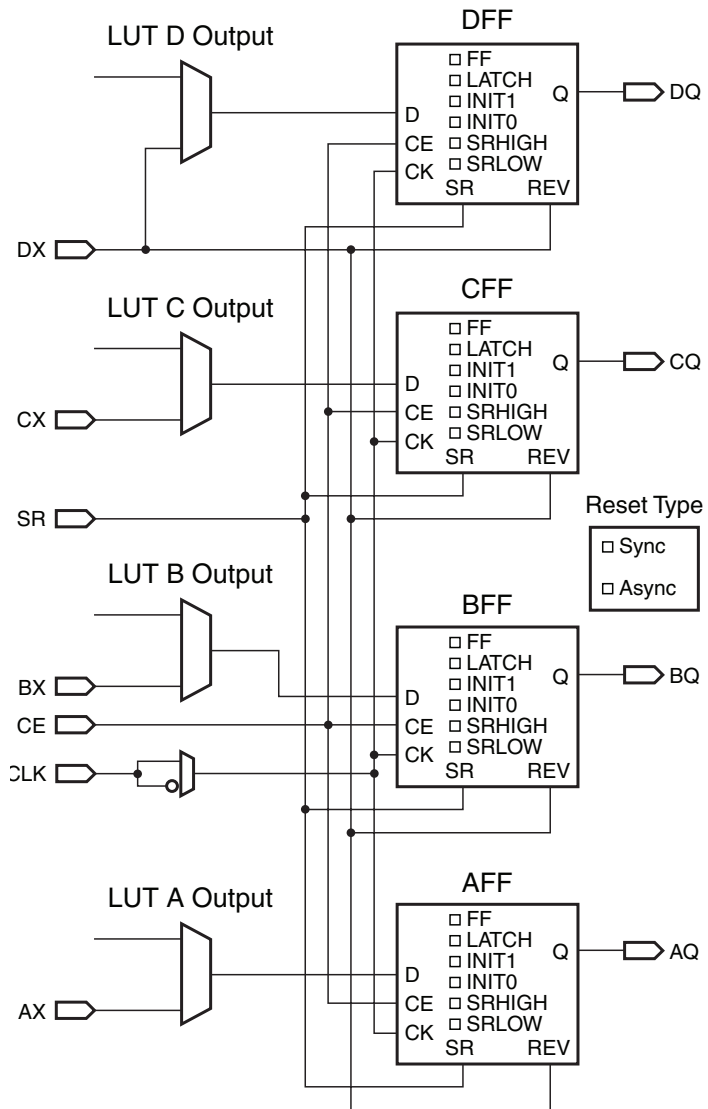


**Four 6-LUTs**

**Four Flip-Flops**

**Switching fabric may see  
combinational and  
registered outputs.**

# Slice flip-flop properties ...



Each state element may be edge-triggered or latching.

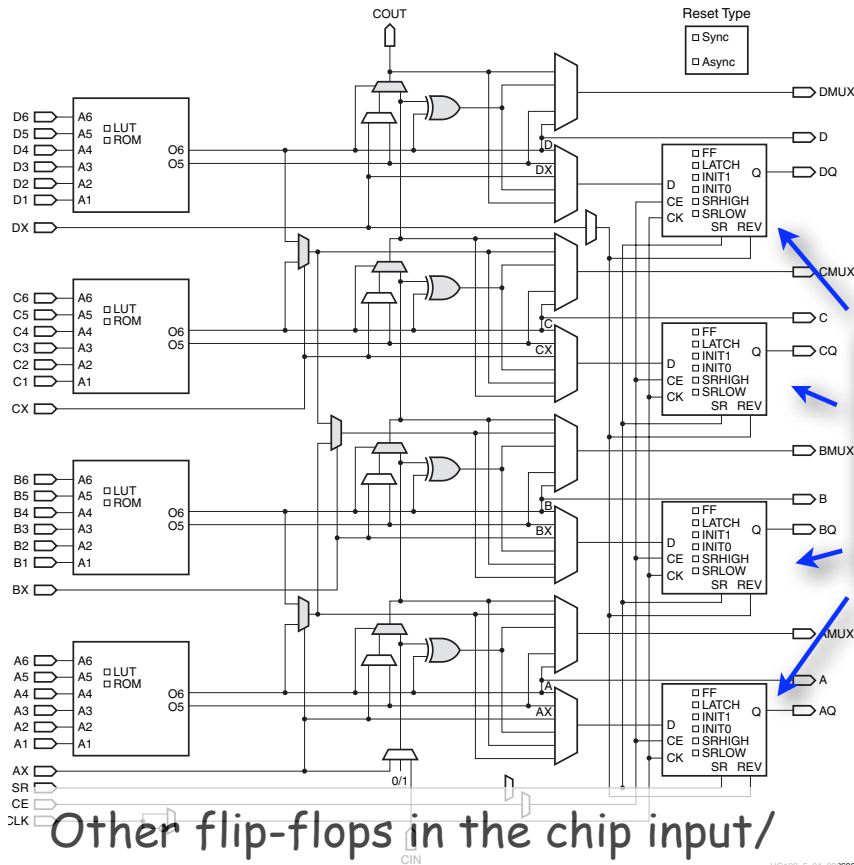
Clock enable, clock polarity, and set/reset lines in a slice are shared.

Each state element may respond differently to set/reset signal.



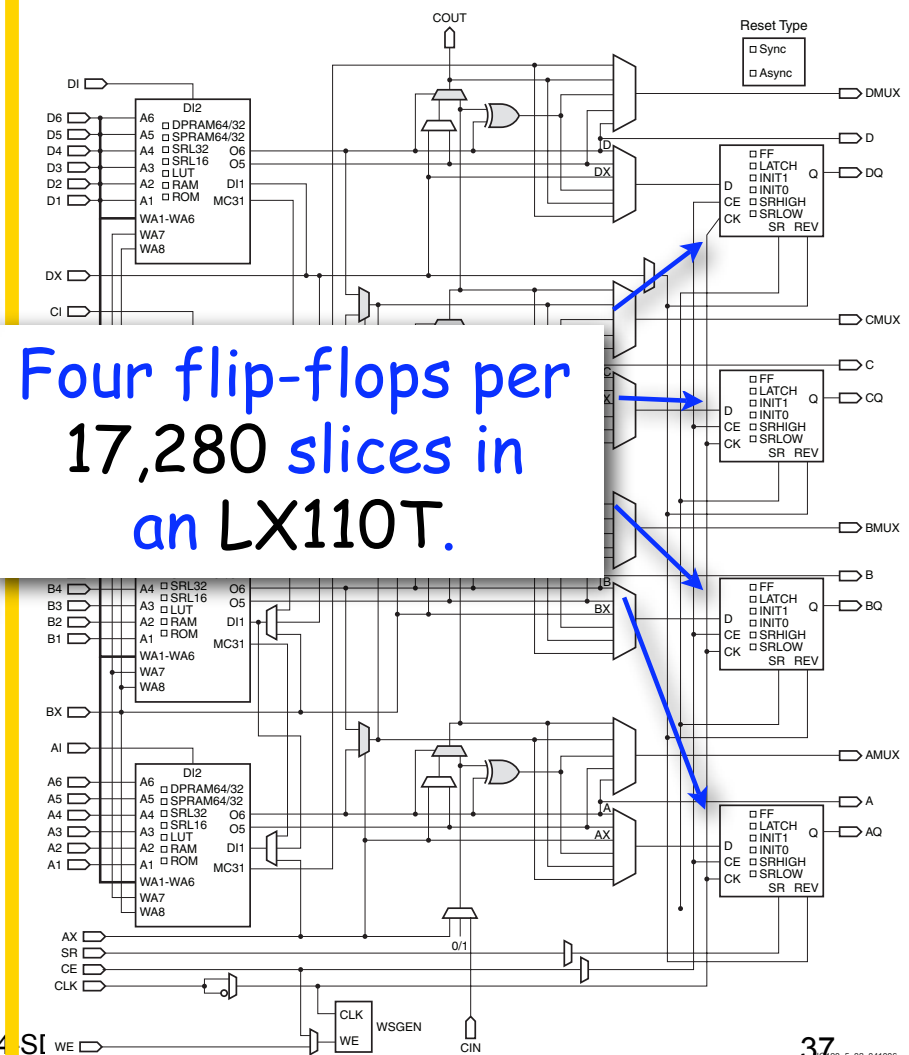
# Flip-flops on Virtex5 FPGA

## SLICEL



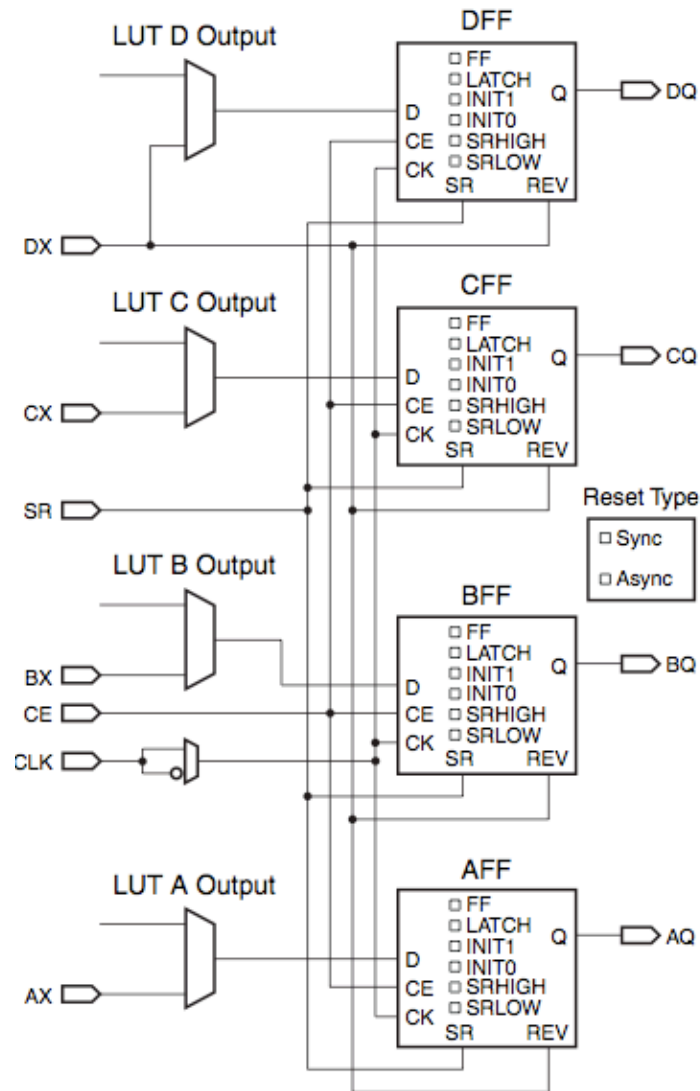
Other flip-flops in the chip input/output cells, and in the form of registers in the DSP slices and memory block interfaces.

## SLICEM



Four flip-flops per 17,280 slices in an LX110T.

# Virtex5 Slice Flip-flops



4 flip-flops / slice (corresponding to the 4 6-LUTs)

Each takes input from LUT output or primary slice input.

Edge-triggered FF vs. level-sensitive latch.  
Clock-enable input (can be set to 1 to disable) (shared).

Positive versus negative clock-edge.

Synchronous vs. asynchronous reset.

SRHIGH/SRLOW select reset (SR) set.

REV forces opposite state.

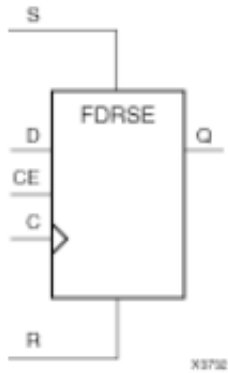
INIT0/INIT1 used for global reset (not shown - usually just after power-on and configuration).

UG190\_5\_05\_071207

150 lec04-SDS-review2

Page 58

# Virtex5 Flip-flops “Primitives”

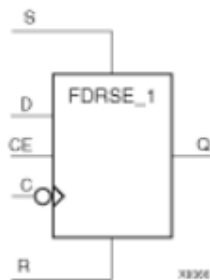


D Flip-Flop with Synchronous Reset and Set and Clock Enable

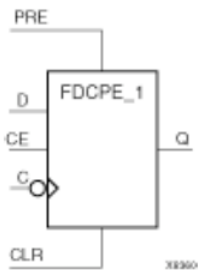
Provided by the CAD tools. This maps to single slice flip-flop.

## Logic Table

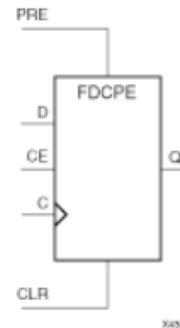
Inputs					Output
R	S	CE	D	C	Q
1	-	-	-	↑	0
0	1	-	-	↑	1
0	0	0	-	-	No Change
0	0	1	1	↑	1
0	0	1	0	↑	0



Negative-Clock Edge, Synchronous Reset and Set, and Clock Enable



Negative-Edge Clock, Clock Enable, and Asynchronous Preset and Clear



Clock Enable and Asynchronous Preset and Clear

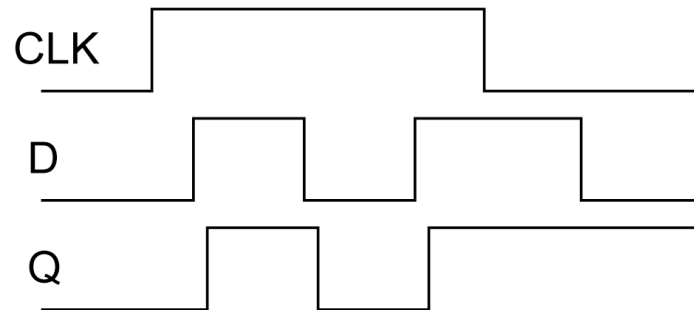
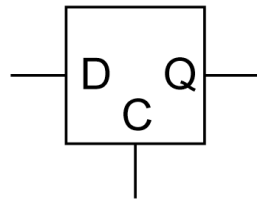
# Inside a Flip-Flop

---



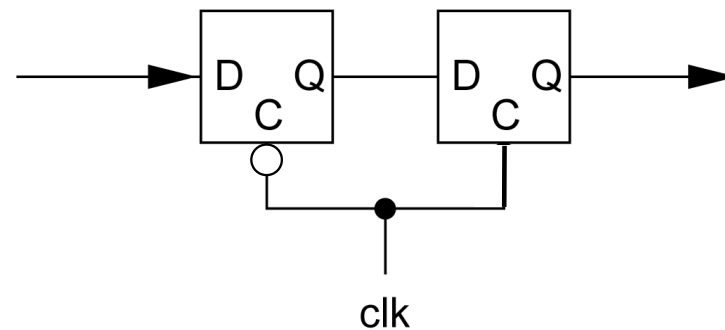
# Level-sensitive Latch

Positive Level-sensitive latch:



When CLK is high, latch is transparent, when clk is low, latch retains previous value.

Positive Edge-triggered flip-flop built from two level-sensitive latches:



# Upcoming events:

Tue 1/26	Lec #3: FPGA Architecture Introduction: <a href="#">[PDF]</a> Reading: Chapter 5 of the <a href="#">Virtex-5 User's Guide</a> (PreLab reading)	HW #1: <a href="#">[PDF]</a> (Due Fri, Jan 29 @ 14:10)	Lab #1: FPGA Physical Layout <a href="#">[ZIP]</a> <a href="#">[PDF]</a>
Thr 1/28	Lec #4: Synchronous Digital Systems Review (2):	Solution:	Lab Lec #2:
Tue 2/2	Lec #5: Verilog Primer: Reading: DDCA: Chapter 4	HW #2: (Due Fri, Feb 5 @ 14:10)	Lab #2: Structure Verilog FPG
Thr 2/4	Lec #6: CAD Tools (Synthesis):	Solution: Quiz:	Lab Lec #3:

Have a good weekend!

