Sequential Logic Design

CS 64: Computer Organization and Design Logic
Lecture #14

Ziad Matni
Dept. of Computer Science, UCSB

Administrative

- Only 2.5 weeks left!!!!!!!! OMG!!!!!
- Th. 5/24 Sequential Logic
  - *Lab# 7 Combinatorial Logic*
- Tu. 5/29 More Sequential Logic!
- Th. 5/31 Finite State Machines
  - *Lab# 8 Sequential Logic and CPU Design*
- Tu. 6/4 More Finite State Machines!
- Th. 6/8 CS Ethics and Societal Impact of CS
  - *Lab# 9 Finite State Machines*
  - *Lab# 10 Ethics/Societal Impact (short; online)*

Lecture Outline

- Sequential Logic
- S-R Latch
- D-Latch
- D-Flip Flop
- Reviewing what's needed for Lab 8

  (next week's lab)

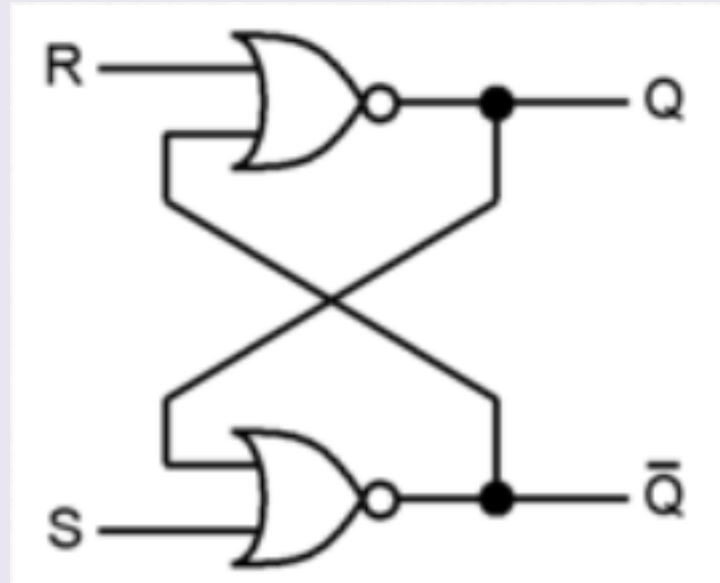
Sequential Logic

- **Combinatorial Logic**
  - Combining multiple logic blocks
  - The output is a function **only** of the present inputs
  - There is no memory of past "states"

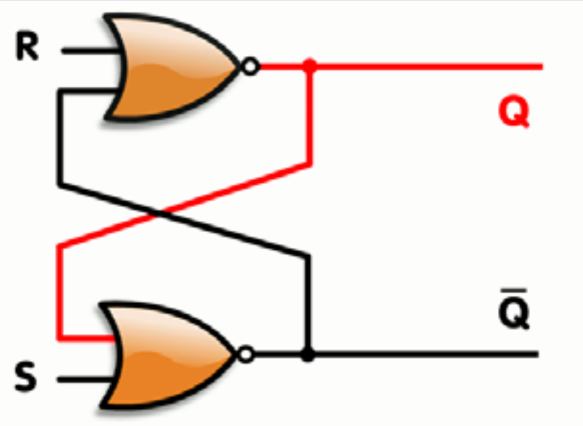
- **Sequential Logic**
  - Combining multiple logic blocks
  - The output is a function of ***both*** present and ***past*** inputs
  - There exists a memory of past "states"

The S-R Latch

- Only involves 2 NORs
- The outputs are fed-back to the inputs
- The result is that the output state (either a 1 or a 0) is maintained even if the input changes!



How a S-R Latch Works



S	R	Q_0	Comment
0	0	Q*	Hold output
0	1	0	Reset output
1	0	1	Set output
1	1	X	Undetermined

- Note that if one NOR input is **0**, the output becomes the inverse of the other input
- So, if output Q already exists and if S = 0, R = 0, then Q will remain at whatever it was before! (**hold output state**)
- If S = 0, R = 1, then Q becomes 0 (**reset output**)
- If S = 1, R = 0, then Q becomes 1 (**set output**)
- Making S = 1, R = 1 is *not allowed* (**undetermined output**)

Consequences?

- As long as **S = 0 and R = 0**,
  the circuit output holds memory of its prior value (state)

S	R	Q_0	Comment
0	0	Q*	Hold output
0	1	0	Reset output
1	0	1	Set output
1	1	X	Undetermined

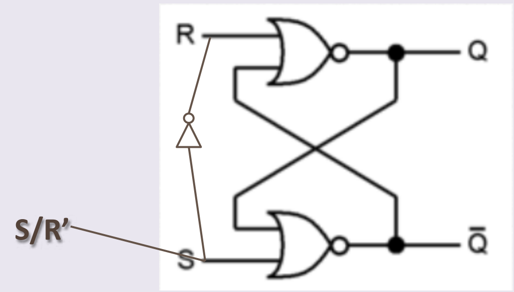
- To change the output, just make
  S = 1 (but also R = 0) to make the output 1 (set) ***OR***
  S = 0 (but also R = 1) to make the output 0 (reset)

- Just avoid S = 1, R = 1…

About that S = 1, R = 1 Case…

S	R	Q_0	Comment
0	0	Q*	Hold output
0	1	0	Reset output
1	0	1	Set output
1	1	X	Undetermined

- What if we avoided it on purpose by making R = NOT (S)?
  - Where's the problem?

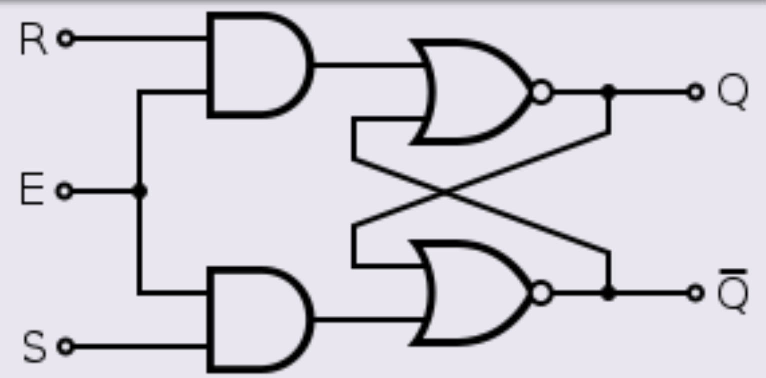


- This, by itself, precludes a case when R = S = 0
  - You'd need that if you want to preserve the previous output state!

- Solution: the ***clocked latch*** and ***the flip-flop***

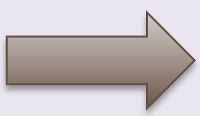
Adding an "Enable" Input: **The Gated S-R Latch**

- Create a way to "gate" the inputs
  - R/S inputs go through *only if* an **"enable input" (E) is 1**
  - If E is 0, then the S-R latch gets SR = 00 and it hold the state of previous outputs



- So, the truth table would look like:

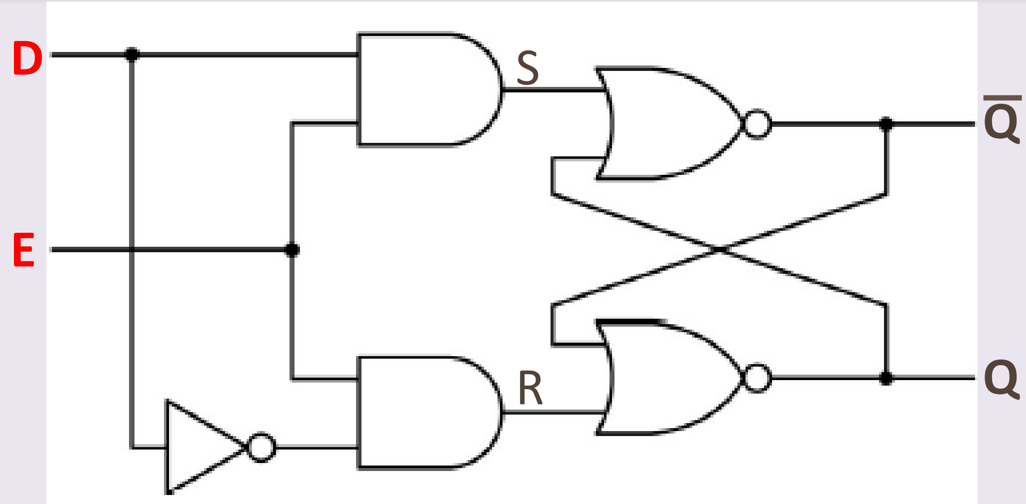
S	R	Q_0	Comment
0	0	Q*	Hold output
0	1	0	Reset output
1	0	1	Set output
1	1	X	Undetermined



S	R	E	Q_0	Comment
X	X	0	Q*	Hold output
0	1	1	0	Reset output
1	0	1	1	Set output

Combining R and S inputs into One: **The Gated D Latch**

- Force S and R inputs to always be opposite of each other
  - Make them the same as an input D, where **D = R** and **!D = S**.

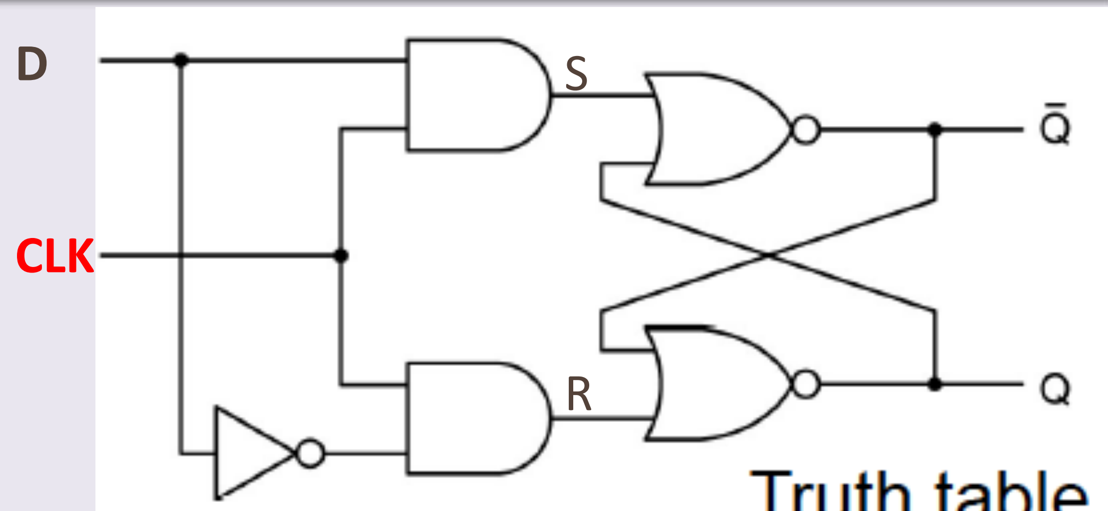


- Create a way to "gate" the D input
  - D input goes through only if an enable input (E) is 1
  - If E is 0, then hold the state of the previous outputs

D	E	Q_0	Comment
X	0	Q*	Hold output
0	1	0	Reset output
1	1	1	Set output

Enabling the Latch Synchronously: **The Clocked D Latch**

- If you apply a clock on input E, you get a **clocked D latch.**



- A clock is an input that goes 1 then 0, then 1 again in a set time period

- When CLK is 0, both S and R inputs to the latch are 0 too, so the Q holds its value ($Q = Q_0$)

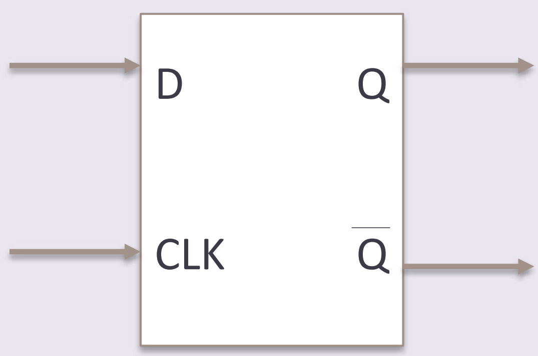
- When CLK is 1, then if D = 1, then Q =1, but if D = 0, then Q = 0

Truth table

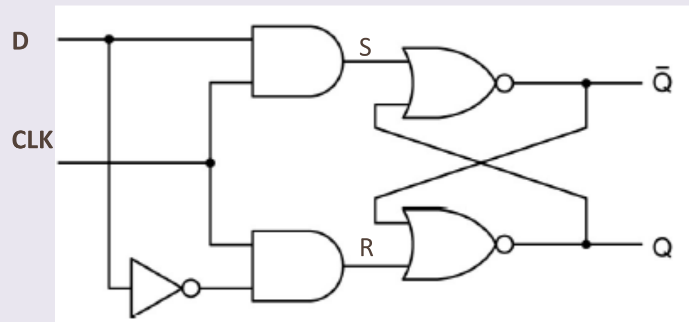
D	CK	Q
0	1	0
1	1	1
X	0	Q_0

Clocked D Latch as Digital Sampler

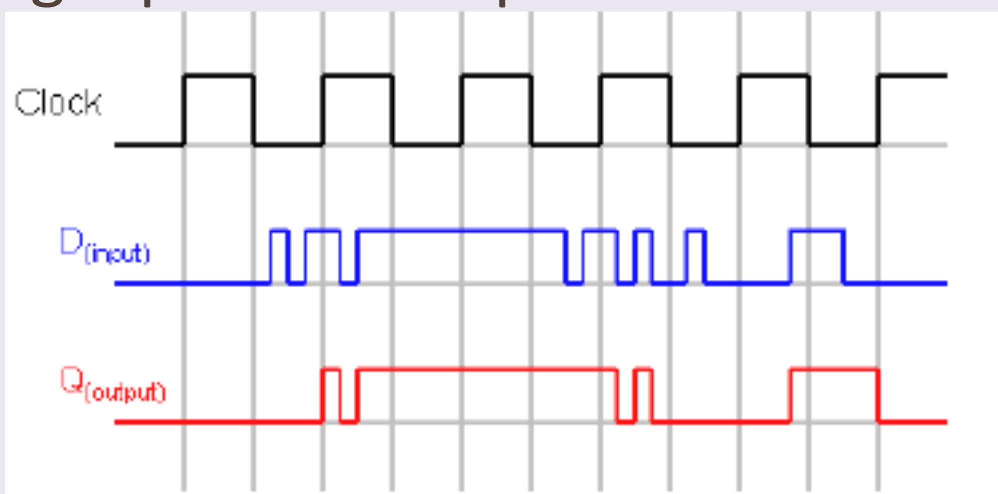
- This clocked latch can be used as a "programmable" memory device that "samples" an input on a regular basis



The Clocked D Latch By Any Other Name…



- Observing input and output “waveforms”



The Joys of Sampling…

- Sampling data in a periodic way is advantageous
  - I can start designing more complex circuits that can help me do *synchronous* logical functions
    - *Synchronous*: in-time

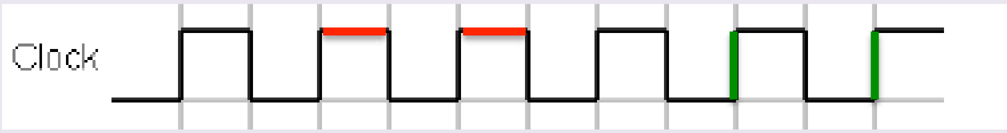
- Very useful in *pipelining* designs used in CPUs
  - Pipelining: a technique that allows CPUs to execute instructions more efficiently – in parallel

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Instruction fetch, decode, execute, memory access, register write

The Most Efficient Way to Sample Inputs

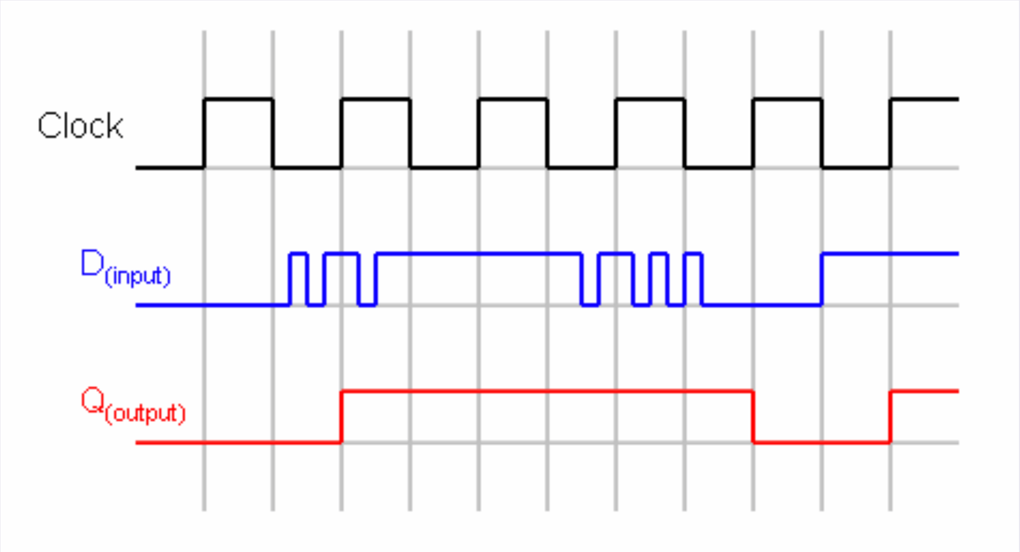
- Instead of sampling the input to the latch using a ***level*** of the clock…
  - That is, when the clock is "1" (or "0")



- … sample the input at the ***edge*** of the clock
  - That is, when the clock is transitioning from 0→1, called a *rising* or *positive* edge
    (or it could be done from 1→0,
    the *falling* edge a.k.a *negative* edge)
  - Why??

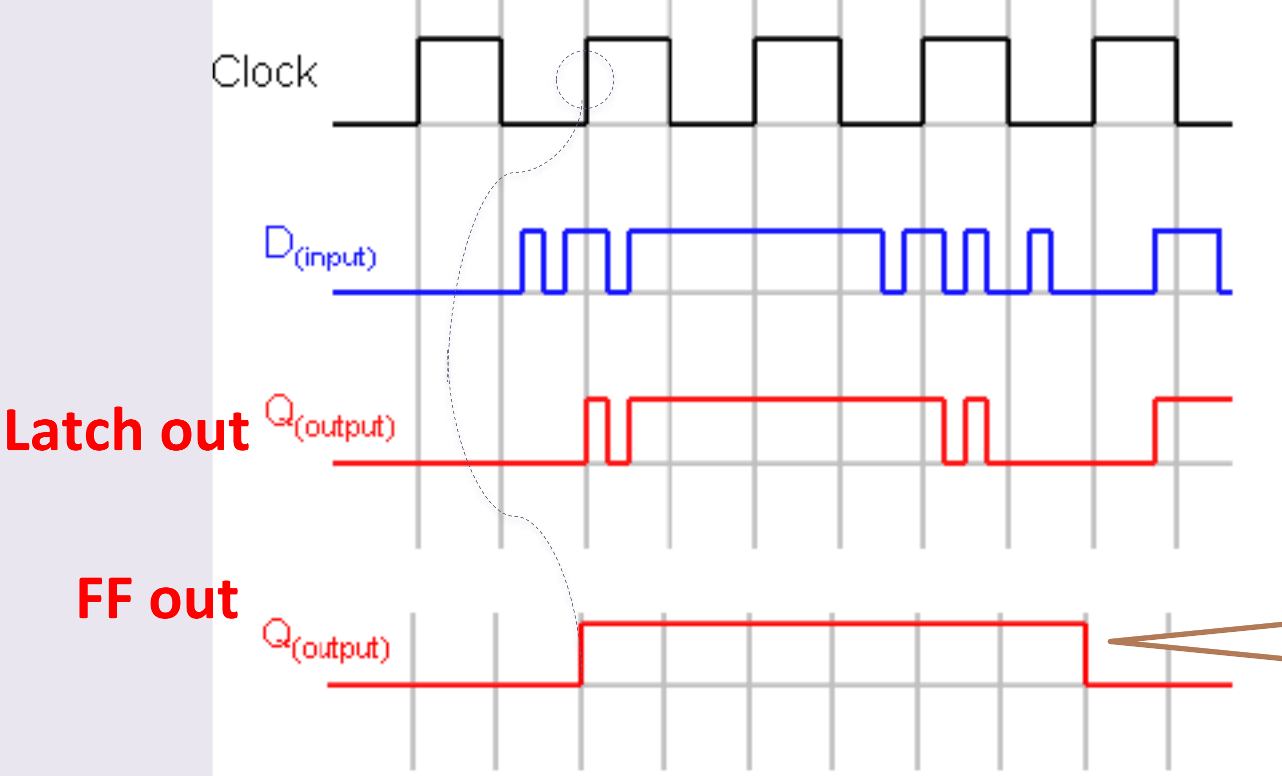
The D-FF

- When the input clock edge is *rising*, the input (D) is *captured* and placed on the output (Q)
  - Rising edge a.k.a positive edge FF
  - Some FF are negative edge FF (capture on the falling edge)



Latches vs. FFs

- Latches capture data on an entire 1 or 0 of the clock
- FFs capture data on the edge of the clock
  - This example shows the positive (0→1) edge used

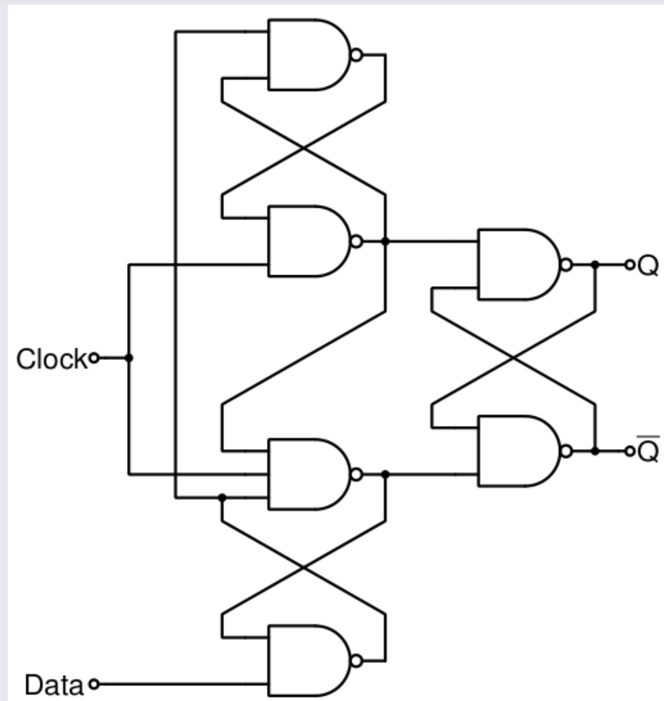


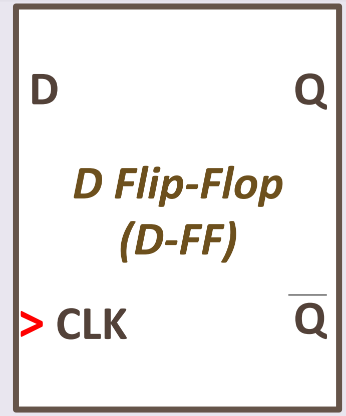
FFs give out less "glitchy" outputs

An Improvement on the Latch: **The D Flip-Flop**

Don’t worry about the circuit implementation details, but understand the use!

The **D Flip-Flop** only changes the output (Q) into the input (D) at the ***positive edge*** (the 0 → 1 transition) of the clock





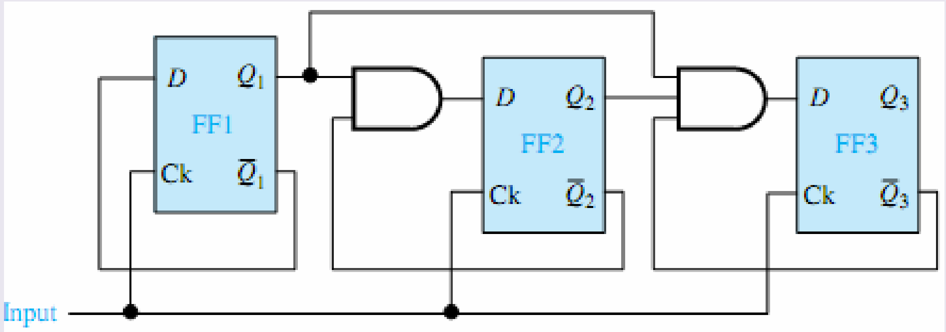
As opposed to:

Note the (slight) difference in the 2 symbols…

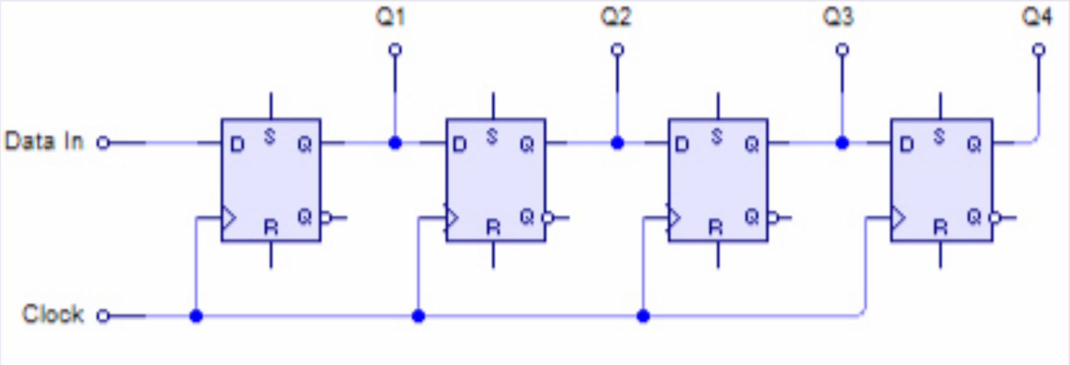
Again, don't worry about the circuit implementation details, but understand the uses!

Popular Uses for D-FFs

- Counter



- Serial-to-Parallel converter



- Digital delay line

Lab 8

Note When Everything is Due…

- Next Monday (5/28) is a University Holiday
- So… no lab on Monday

- BUT! You still have a new Lab (#8) that week!!!
  - Issued to you over the weekend

- Get started on it early – it's long-ish
- Go see TAs during their office hours (We. & Fr.) for help

- Lab#8 will be due on **MONDAY 6/4**
  - Not the usual Friday!

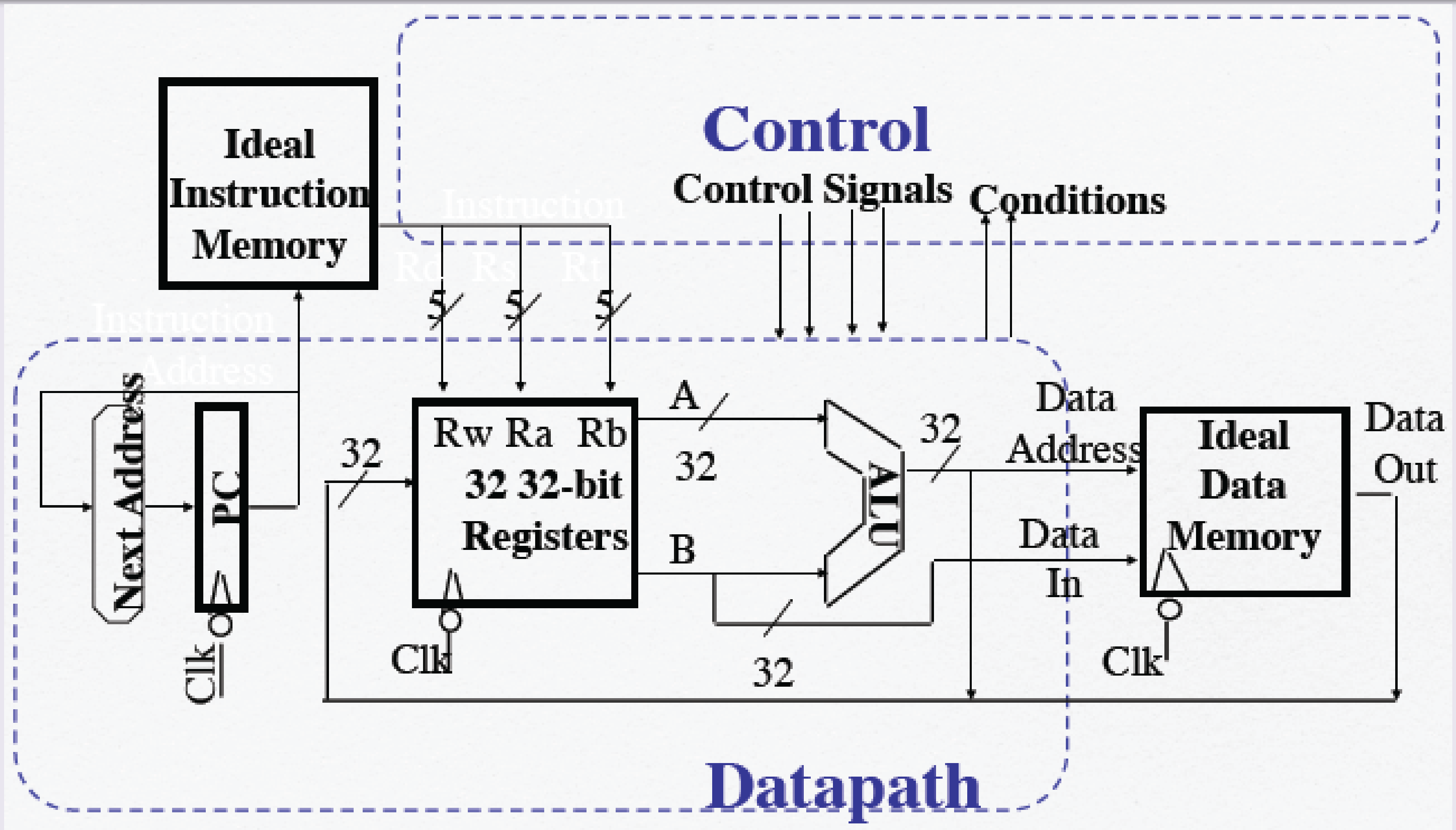
- **Lab#9** will be ISSUED ON MONDAY 6/4 & BE DUE ON FRIDAY 6/8
- **Lab#10** will be ISSUED ON THURSDAY 6/7 & BE DUE ON FRIDAY 6/8
  - Here's Why…

What's Lab8 About?

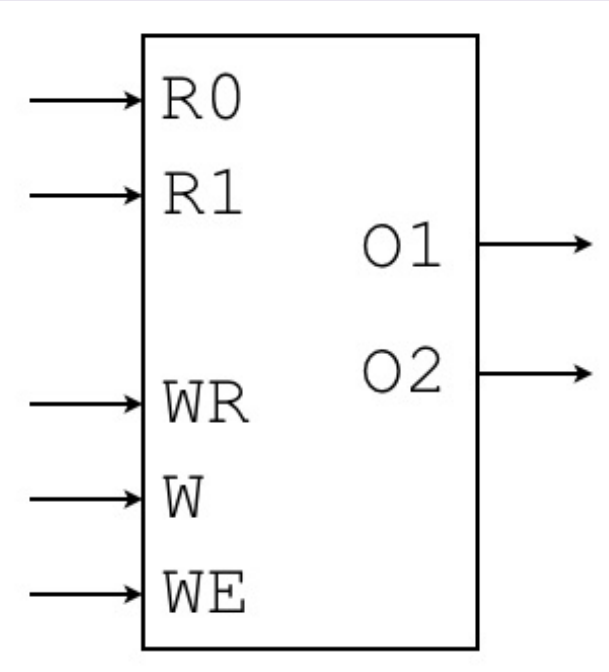
- Design a "simple" ALU (Task 1)
- Design a "simple" register block using D-FFs (Task 2)
- Be given a specification for a "simple" CPU that uses:
  - 1 "Simple" Register Block
  - 1 "Simple" ALU
  - 1 Abstract Computer Memory Interface
  - As many ANDs, ORs, NOTs, XORs, Muxes that you need
- Design this CPU! (Task 3)

- You will draw all of these (BE NEAT!)
  - Take pictures or (better yet) scan them, then turnin

Abstract Schematic of the MIPS CPU



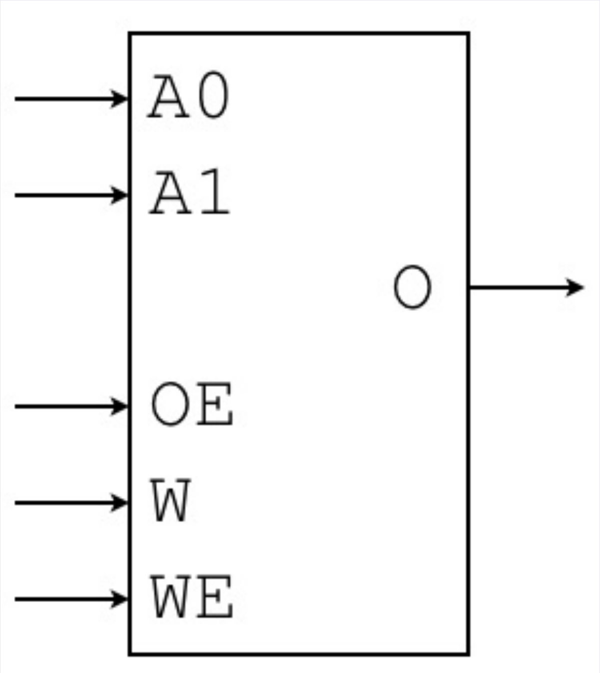
Register Object for Lab 8 (Task 2)



This will be made from D-FFs or D-Latches and Combinatorial Logic

I/O Name	I/O Description
R0	The first register to read, as a single bit. If 0, then reg0 should be read. If 1, then reg1 should be read.
R1	The second register to read, as a single bit. If 0, then reg0 should be read. If 1, then reg1 should be read.
WR	"Write Register". Specifies which register to write to. If 0, then reg0 should be written to. If 1, then reg1 should be written to.
W	The data that should be written to the register specified by WR. This is a single bit.
WE	"Write Enable". If 1, then we will write to a register. If 0, then we will **not** write to a register. Note that if WE = 0, then the inputs to WR and W are effectively ignored.
O1	Value of the first register read. As described previously, this depends on which register was selected to be read, via R0.
O2	Value of the second register read. As described previously, this depends on which register was selected to be read, via R1.

Memory Interface Object for Lab 8



I/O Name	I/O Description
A0	Bit 0 of the address (LSB)
A1	Bit 1 of the address (MSB)
OE	"Output Enable". If 1, then the value at the address specified by A0 and A1 will be read, and sent to the output line O. If 0, then the memory will not be accessed, and the value sent to the output line is unspecified (could be either 0 or 1, in an unpredictable fashion).
W	The value to write to memory.
WE	"Write Enable". If 1, then the value sent into W will be written to memory at the address specified by A0 and A1. If 0, then no memory write occurs (the value sent to W will be ignored).
O	The value read from memory (or unspecified if OE = 0).

Task 3: Build a Mock-CPU!

Actually, just a small instruction decoder and executor…

OP1	OP0	B0	B1	B2	Human-readable Encoding	Description
0	0	0	0	0	xor reg0, reg0, reg0	Compute the XOR of the contents of reg0 with the contents of reg0, storing the result in reg0.
0	1	1	0	1	nor reg1, reg0, reg1	Compute the NOR of the contents of reg0 with the contents of reg1, storing the result in reg1.
1	0	1	0	1	load reg1, 01	Copy the bit stored at address 01 (decimal 1) into register reg1.
1	1	0	1	0	store reg0, 10	Store the contents of reg0 at address 10 (decimal 2)
1	1	1	1	1	store reg1, 11	Store the contents of reg1 at address 11 (decimal 3)

These say something about which **registers** are used

These say something about which **operation** is being done

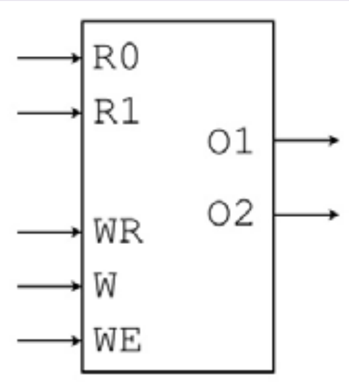
Hints for Task 3

- Design the final circuit in pieces:
  - One piece for each of the 3 types of instruction: load, store, XOR/NOR

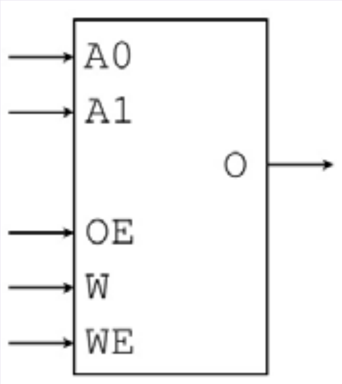
- For example, the store task:
  - If an output isn't used, tie it to a permanent "0" (i.e. ground)
  - If an input isn't used, then you can use "X" (don't care) on it

What registers am I using? Which instruction bits go where?

What controls should I use? What values should they be?



How do I best connect the registers to the memory interface? Again, ask yourself if some of the inputs here should be op-code bits.



Is the output even used in this "store" task?

Tying In All The Pieces (Task 3)

- Now see how they can all fit together
  - You will have 1 register block + 1 memory interface
  - You *won't* need to use any additional latches here
  - You *will* need to use muxes and regular logic (and the simple ALU you designed earlier – see lab instructions for more details)

- REQUIREMENT:
  Use ONLY 1 register block, 1 ALU, and 1 memory interface

Your To Dos

- Lab #7 is due end of day Friday
- Lab #8 will be issued this weekend
  - Due Monday 6/4
- Lab #9 will be issued weekend after next
  - Due Friday 6/8 (last day of classes)
- Lab #10 will be issued in the last week
  - Due Friday 6/8 (last day of classes)

</LECTURE>