

CSC258 Winter 2016

Computer Organization

Lecture 1

Thanks to Andrew Petersen, Myrto Papadopoulou and Steve Engels for previous course materials.

Larry Zhang

Office: DH-3076

Email: ylzhang@cs.toronto.edu

Today's outline

- **Why** CSC258
- **What** is in CSC258
- **How** to do well in CSC258

- Start learning

Why take CSC258?

As a computer science student, it is **embarrassing** to not know how a computer works.

You have to take CSC258, then CSC369 to know how computers work.

More specifically...

- Why are computers built with 1's and 0's and boolean logics?
- How does the computer do everything with just 1's and 0's?
- What is stored in that “minecraft.exe” file, what exactly happens when I double-click on it?
- How does the CPU run an if-statement, or for loop, or recursion?

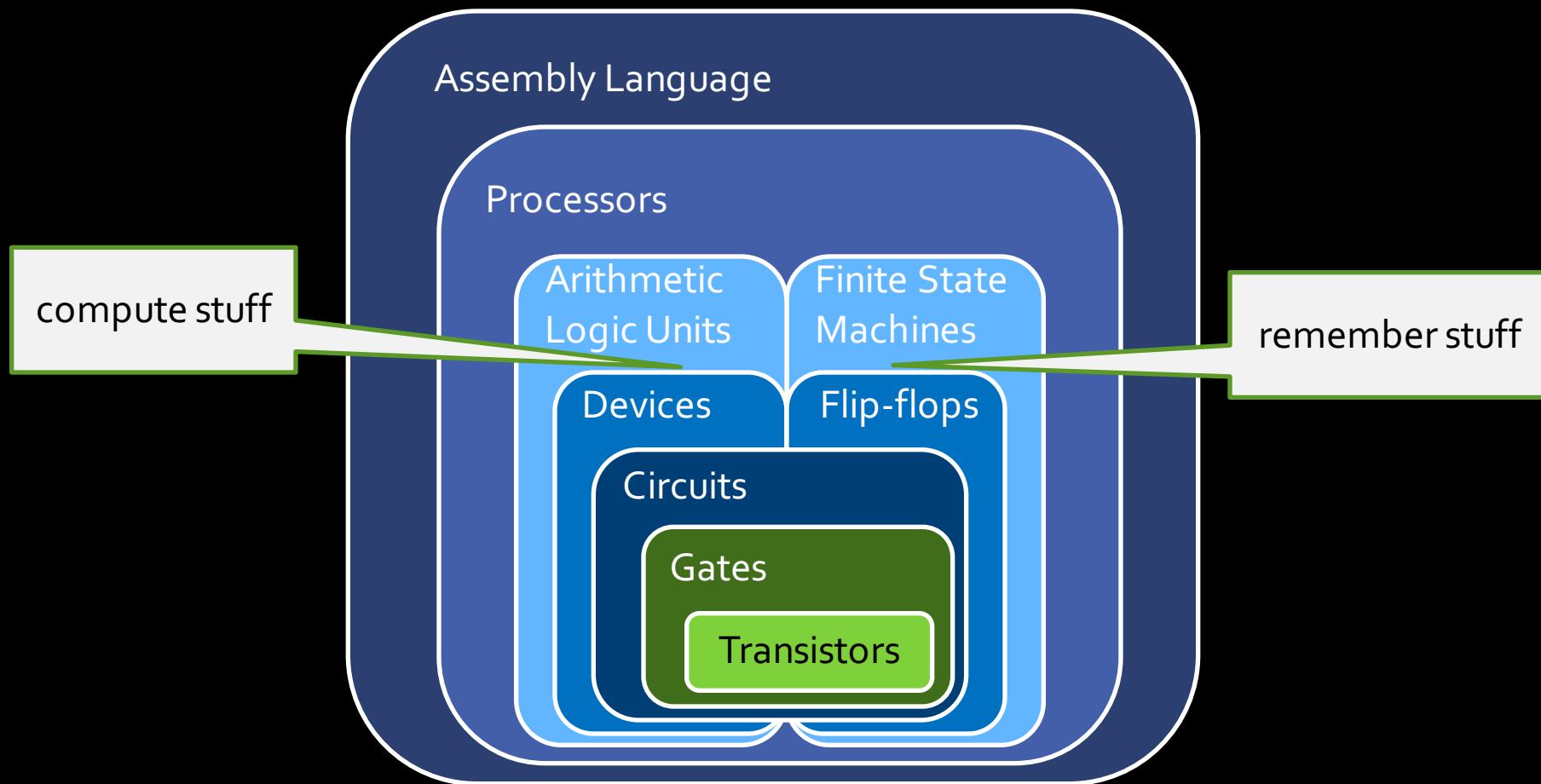
CSC258 has all the answers!

CSC258 Course Goals

- You'll know exactly how a computer is physically built, and you can build one if you want.
- Base on your hardware knowledge, you will be able to engineer the performance of your software like never before.
- Basically, after taking this course, the computer will never look the same to you again.

What's in CSC258?

The architecture of a computer hardware, level by level, bottom-up



We learn the whole real deal

- From atom level to assembly level
- Above the assembly level is the Operating System, whose main job is **virtualization**, i.e., create convenient illusions.
- Everything you learn from every CS course are all **illusions** except for **CSC258**

How to do well in CSC258

First of all ...

Be interested

Course website

<http://www.cs.toronto.edu/~ylzhang/csc258w16/>

All course materials are here.

Lectures (Monday 3-5pm, IB-345)

- Learn the concepts and theories.

A tip for lectures

Get involved in classroom interaction

- Answering a question
- Making a guess / bet / vote
- Back-of-envelope calculations

**Emotional involvement makes the
brain remember better!**

I DON'T INTERACT
IN CLASS

I INTERACT
IN CLASS



Labs! (start from Week 2)

- Hands-on exercises in which you will build real pieces of hardware.
- Work in **pairs**.
- ONLY go to the tutorial section that you are registered to on ROSI/ACORN. If you want to switch lab section, find someone who is willing to change (use Piazza) and let me know.
- Most of the labs involve work **before** the lab.
- **Lab marks are NOT easy free giveaways!**

If you are on the waiting list right now, and still hope to get in, send me an email this week to inform me, so I can make arrangement about labs.

Exams !

- Midterm
 - In class 50 minutes, Feb 22
- Final exam
 - Some time in April

Marking scheme

- Labs: $4\% \times 10 = 40\%$
 - Midterm: 20%
 - Final exam: 40%
- 100%

Must get 40% of final exam to pass

Bonus Mark!

There is a **quiz** at the end of each week's lecture. About what's learned in this week and/or last week. (start from Week 2)

Each quiz earns you some **points**.

At the end of the terms, bonus marks will be awarded according to the **ranking** of quiz points earned.

Top 20% students gets 3% bonus in final grade; top 50% gets 2% bonus in final grade.

The purpose of this is to give you some incentive to keep up with the lecture material, so that you can succeed in exams.

Discussion board (Piazza)

<https://piazza.com/utoronto.ca/winter2016/csc258h5>

- All course announcements will be posted on Piazza.
- **Daily** reading is required.

Office hours

Monday 5-6:30pm, priority for CSC258 students

Tuesday 5-6:30pm, priority for CSC309 students

Friday 5-6:30pm, priority for CSC263 students

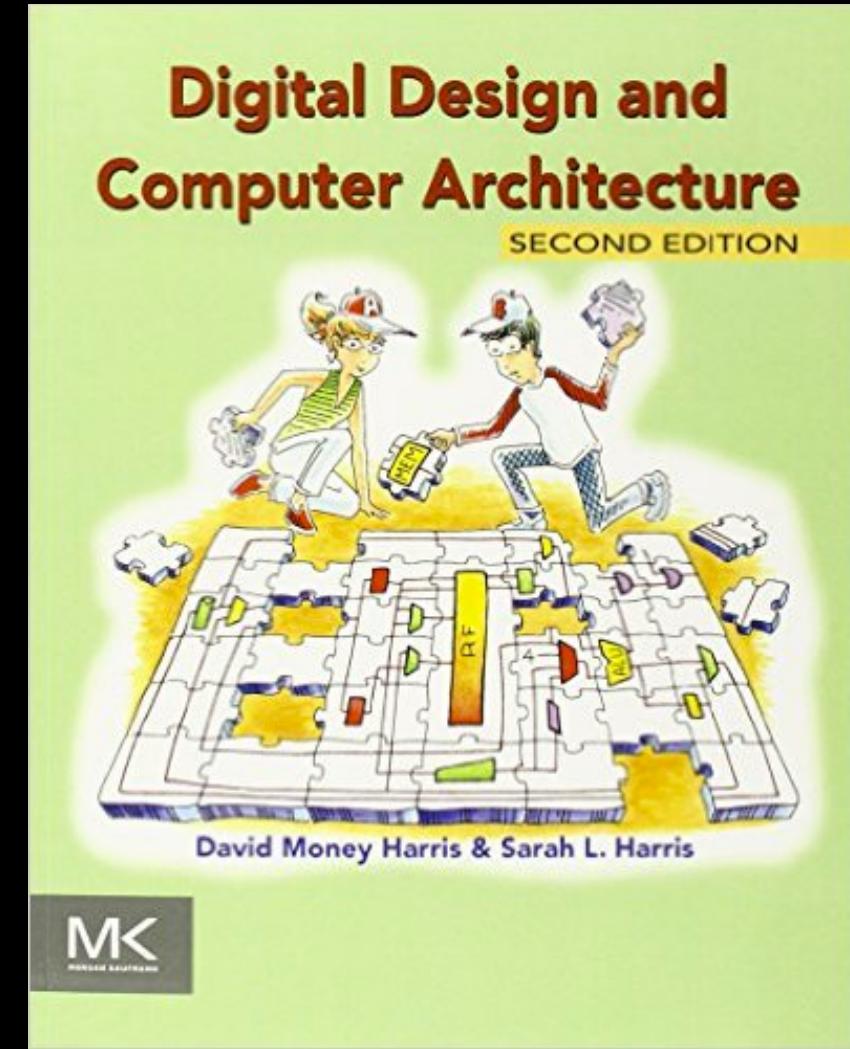
Going to office hours is very helpful, also it
is a good habit to foster in college.



Textbook: DDCA

Digital Design and Computer Architecture, 2nd edition, 2012 by David Harris, Sarah Harris

Available online at UofT library (link in course info sheet)



Weekly feedback form

<http://goo.gl/forms/o248ETWgnS>

Good feedbacks directly improves your learning experience.

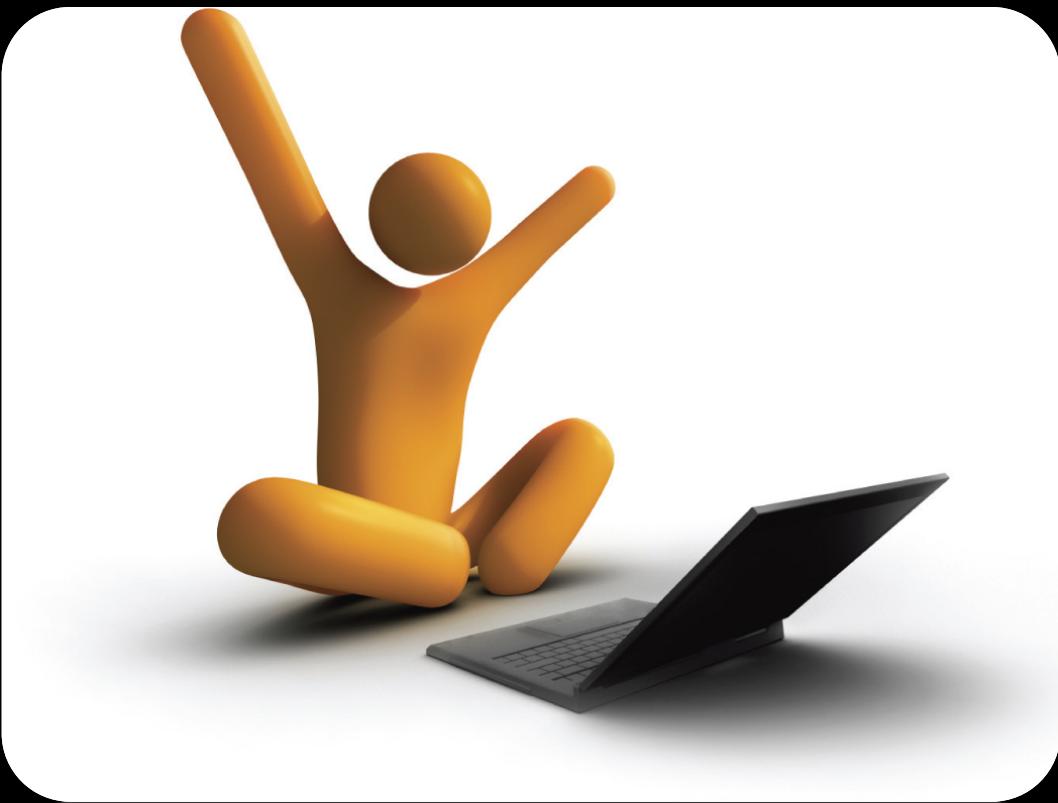
Have your issues addressed weekly rather than termly.

Checklist: How to do well

- Be interested
- Check course website and Piazza regularly
- Go to lectures, interact in class
- Be prepared for the quiz
- Read the slides, read the book
- Work hard on the labs
- Go to office hours
- Discuss on Piazza
- Give weekly feedback
- Do well in midterm and final

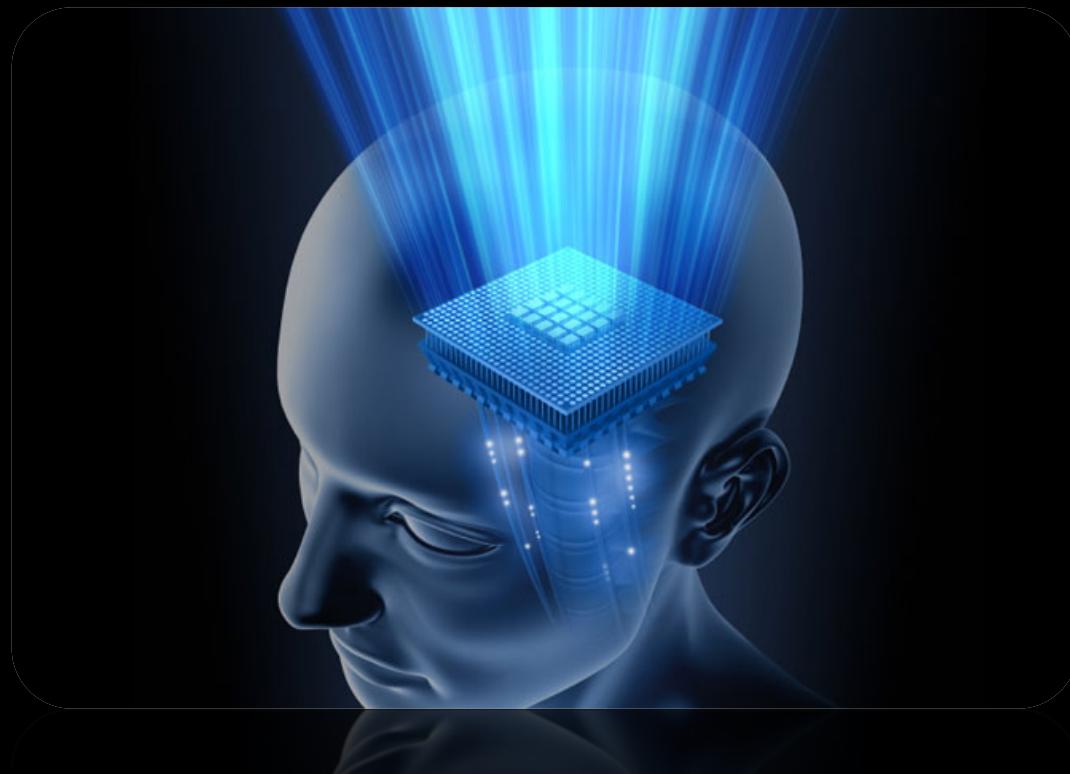
It will be a lot of work, and a lot of fun!

Let the learning begin



Basic Logic Gates

You already know something...

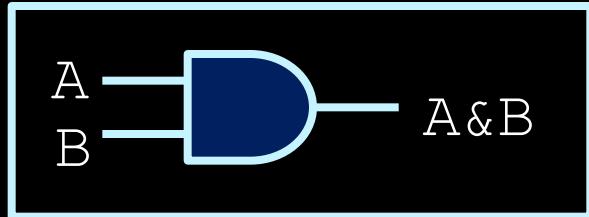


Logic from math course

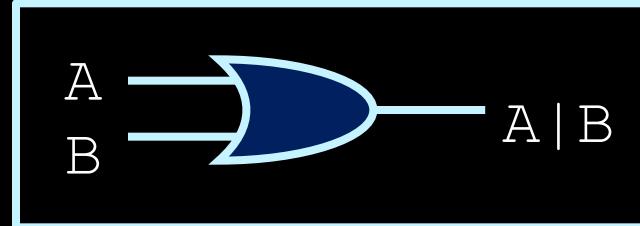
- Create an expression that is true iff the variables A and B are true, or C and D are true.

$$G = (A \ \& \ B) \mid (C \ \& \ D)$$

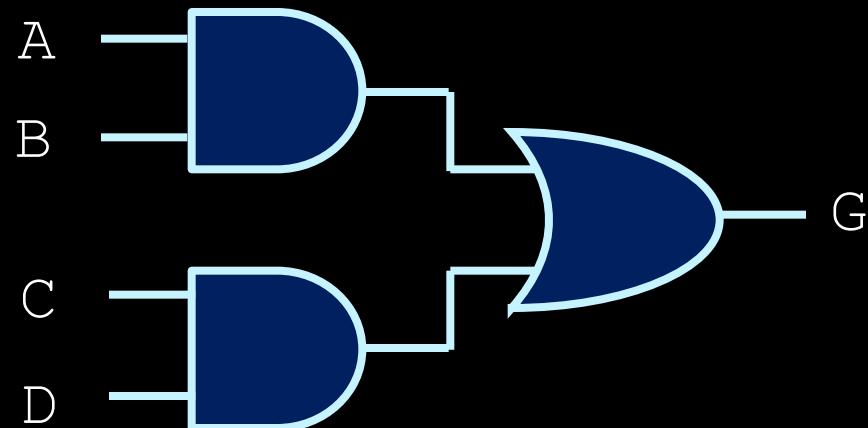
$$G = (A \And B) \Or (C \And D)$$



AND Gate



OR Gate



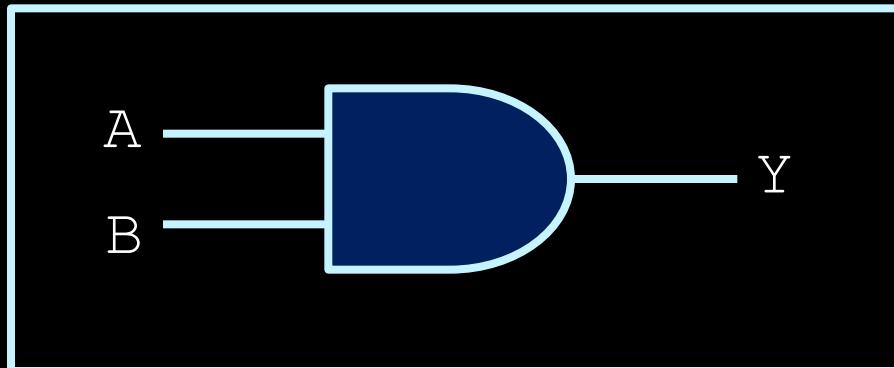
You just designed your first circuit in CSC258!

Gates = Boolean logic

- If we know the logical expression, we already know how to put logic gates together to form a circuit.
- Just need to know which logic operations are represented by which gate!

Let's meet all the gates.

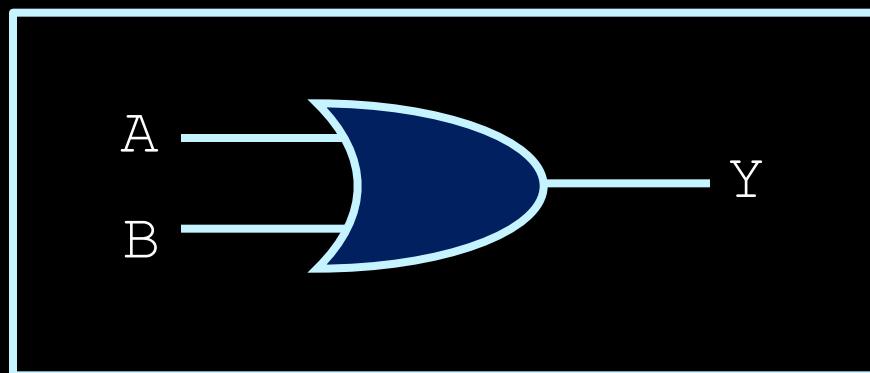
AND Gates



Truth table

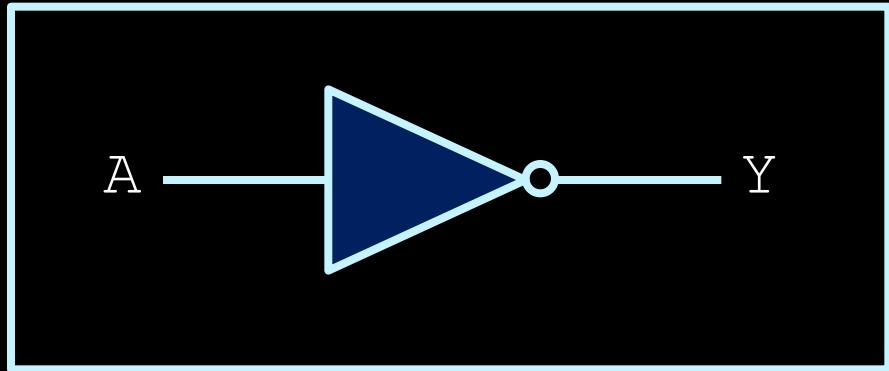
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Gates



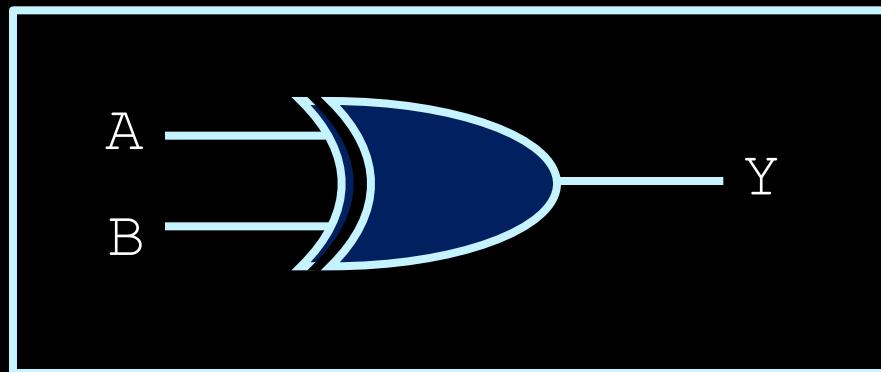
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gates



A	Y
0	1
1	0

XOR Gates

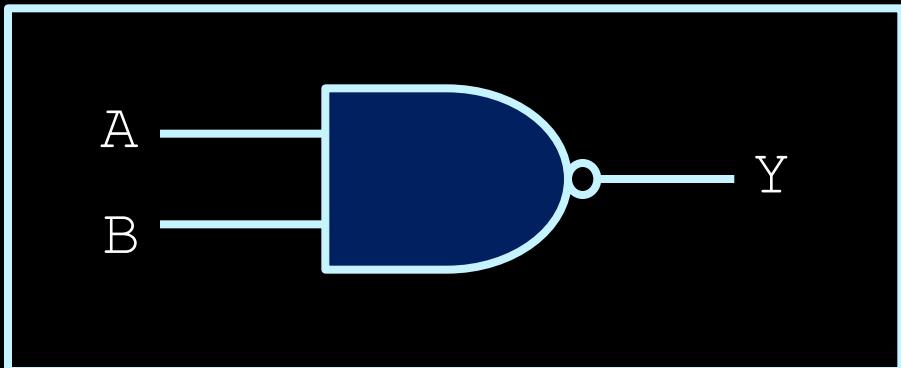


A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Bill Gates

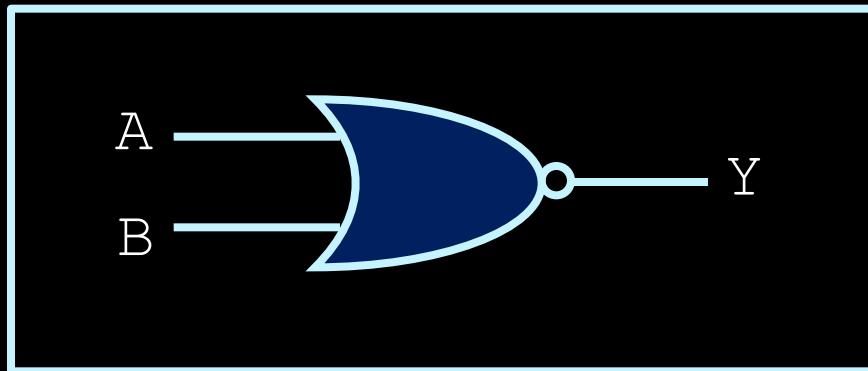


NAND Gates



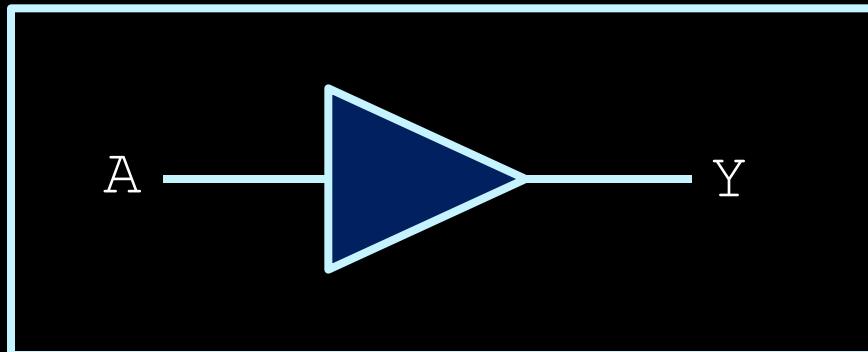
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gates



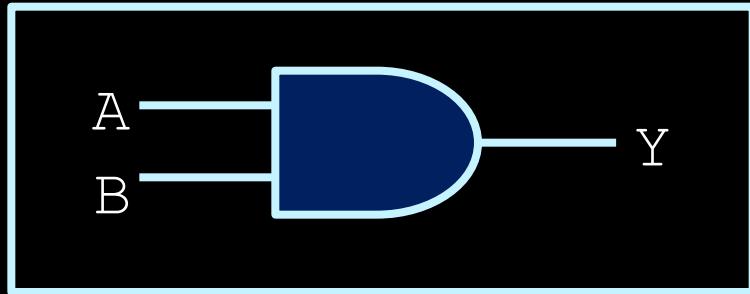
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Buffer



This is not as silly as
you might think now,
as we'll see later...

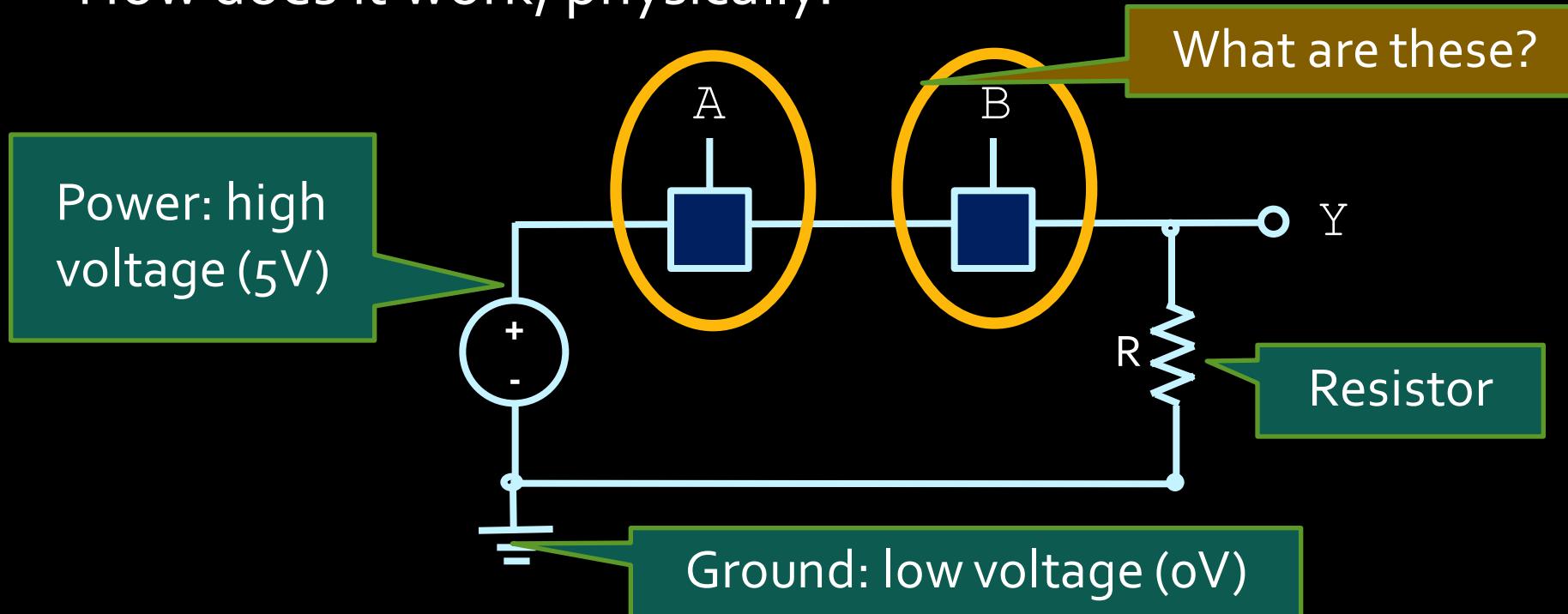
A	Y
0	0
1	1

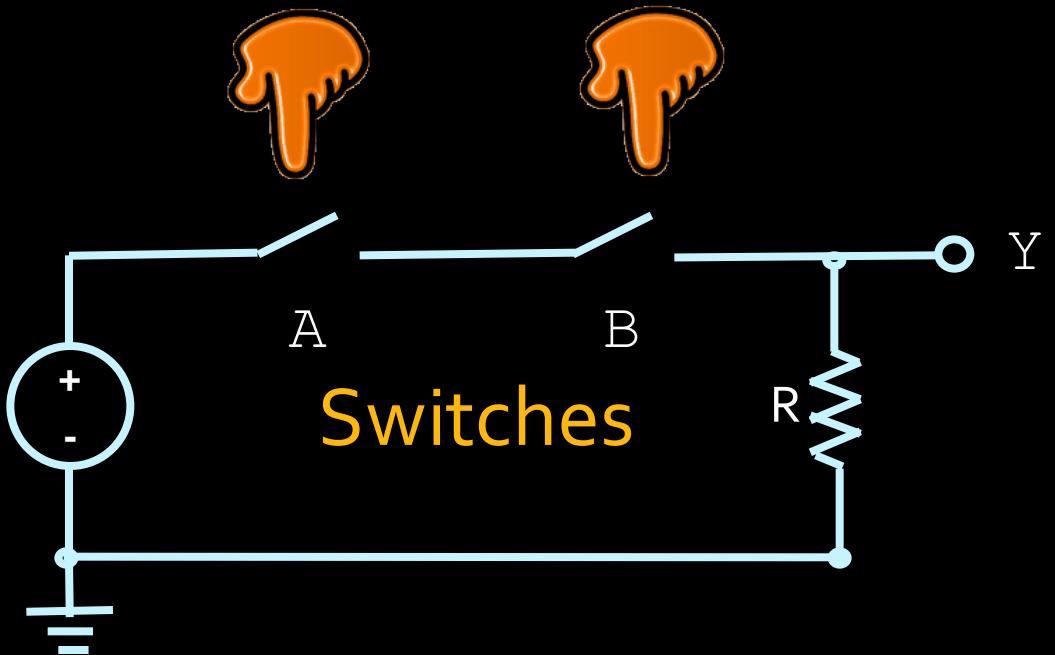


AND Gate

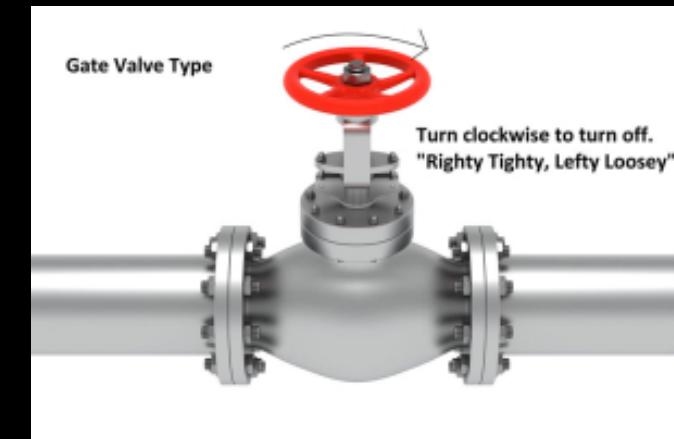
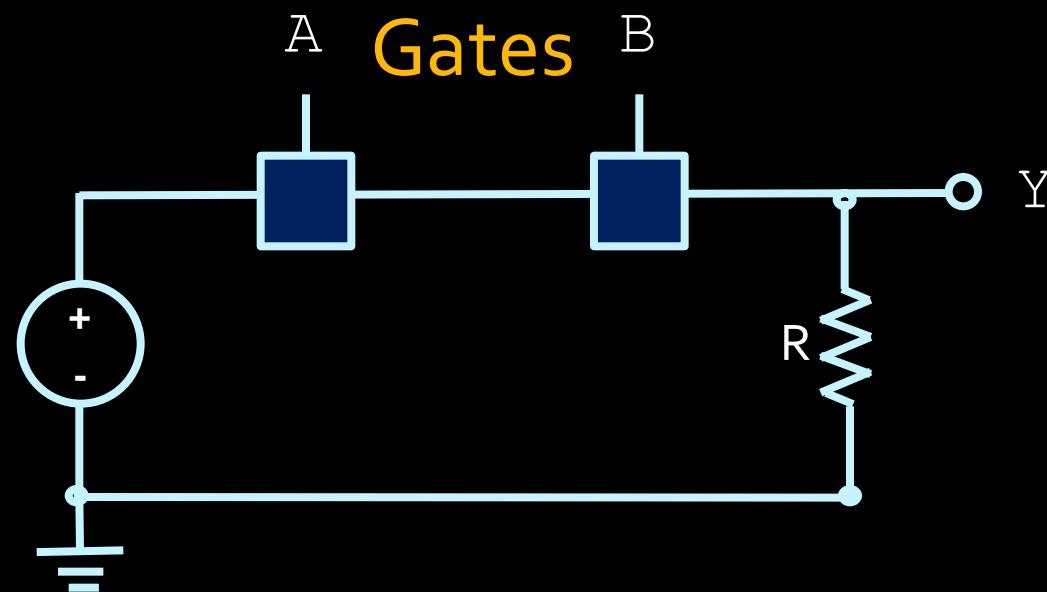


This is just a symbol...
What does it really look like, inside?
How does it work, physically?

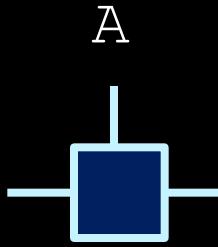




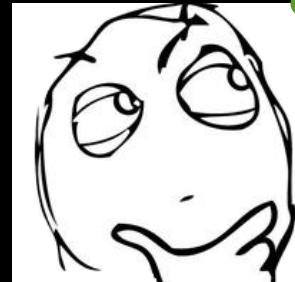
When and only when both A and B are switched **ON**, Y has **high voltage**.



- Gate is like a switch, but controlled by the voltage of the input signal, instead of by a finger.
- Gate A is switched **ON** when signal A is of **high** voltage.
- When and only when **both** A and B have **high** voltage, Y has **high** voltage.
- High voltage is **1** (True), low voltage is **0** (False).
- **Y is True iff both A and B are True ($Y = A \& B$)**.



Gate is switched ON
when signal A is of
high voltage ...



Why?

How?

What does the inside of a gate look like?

Answer: There are **transistors**.

Transistors

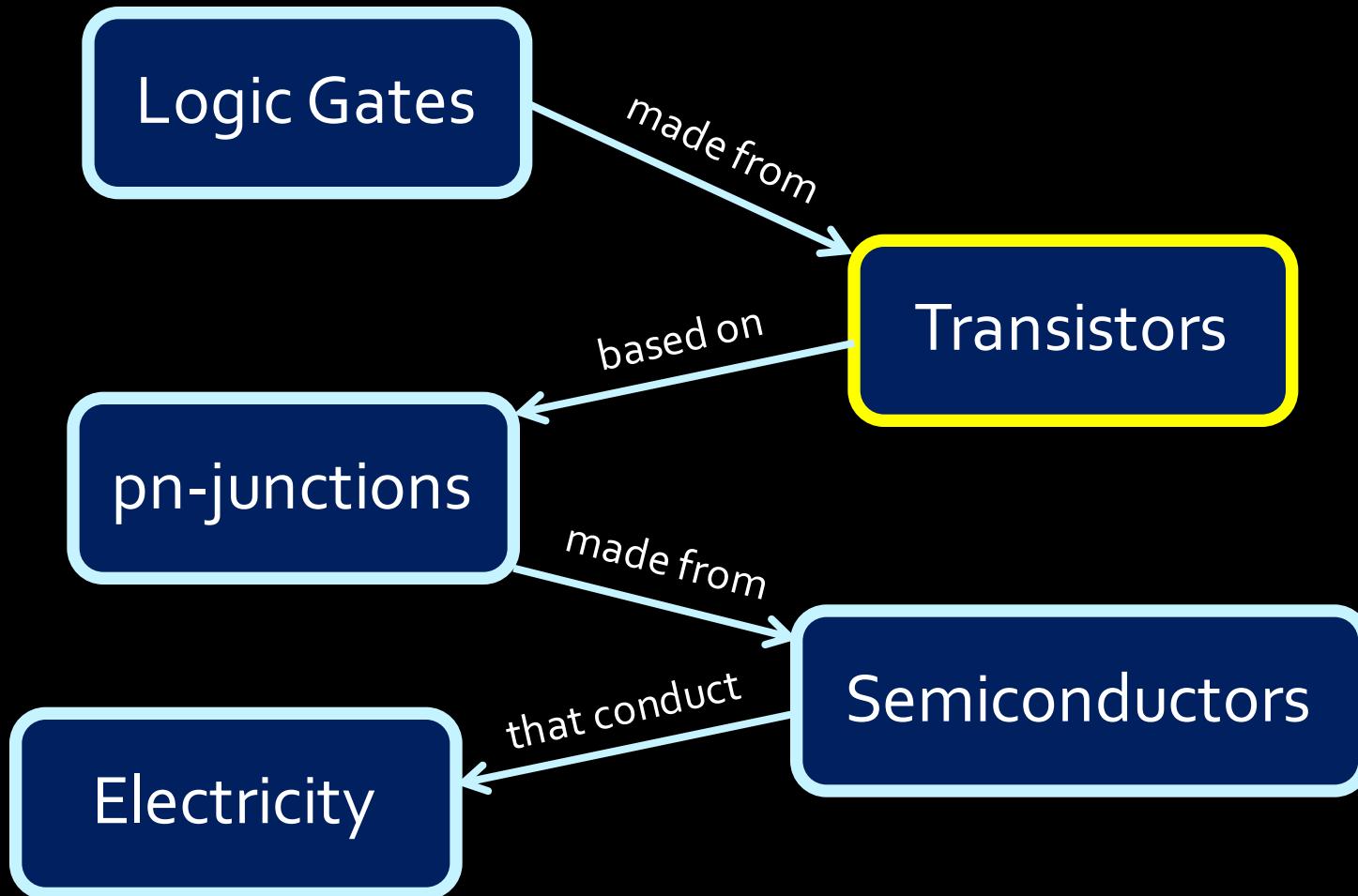
One of the greatest inventions of the 20th century

- Invented by William Shockley, John Bardeen and Walter Brattain in 1947, replacing previous vacuum-tube technology.
 - Nobel Prize for Physics in 1956.



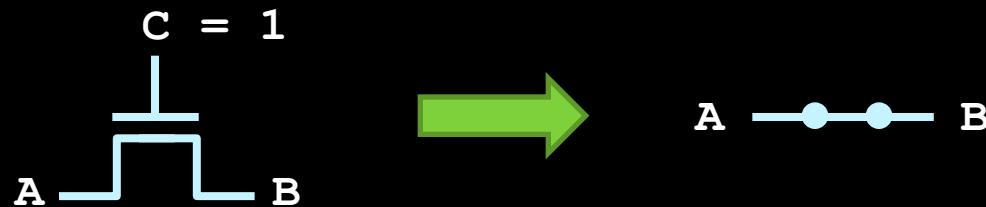
Building block for the hardware of all your computers and electronic devices.

Where do transistors fit?

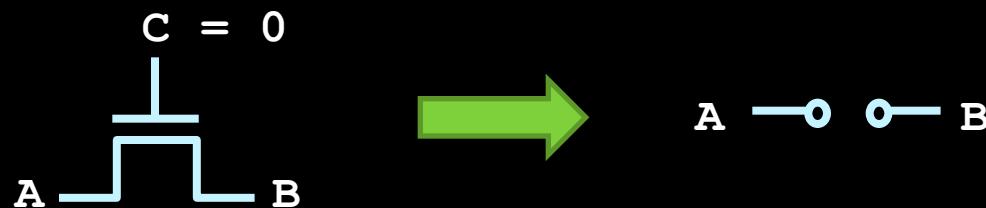


What do transistors do?

- Transistors connect Point A to Point B, based on the value at Point C.
 - If the value at Point C is high, A and B are connected.



- And if the value at Point C is low, A and B are not.



- Need to know a little about electricity now....

BRACE YOURSELVES

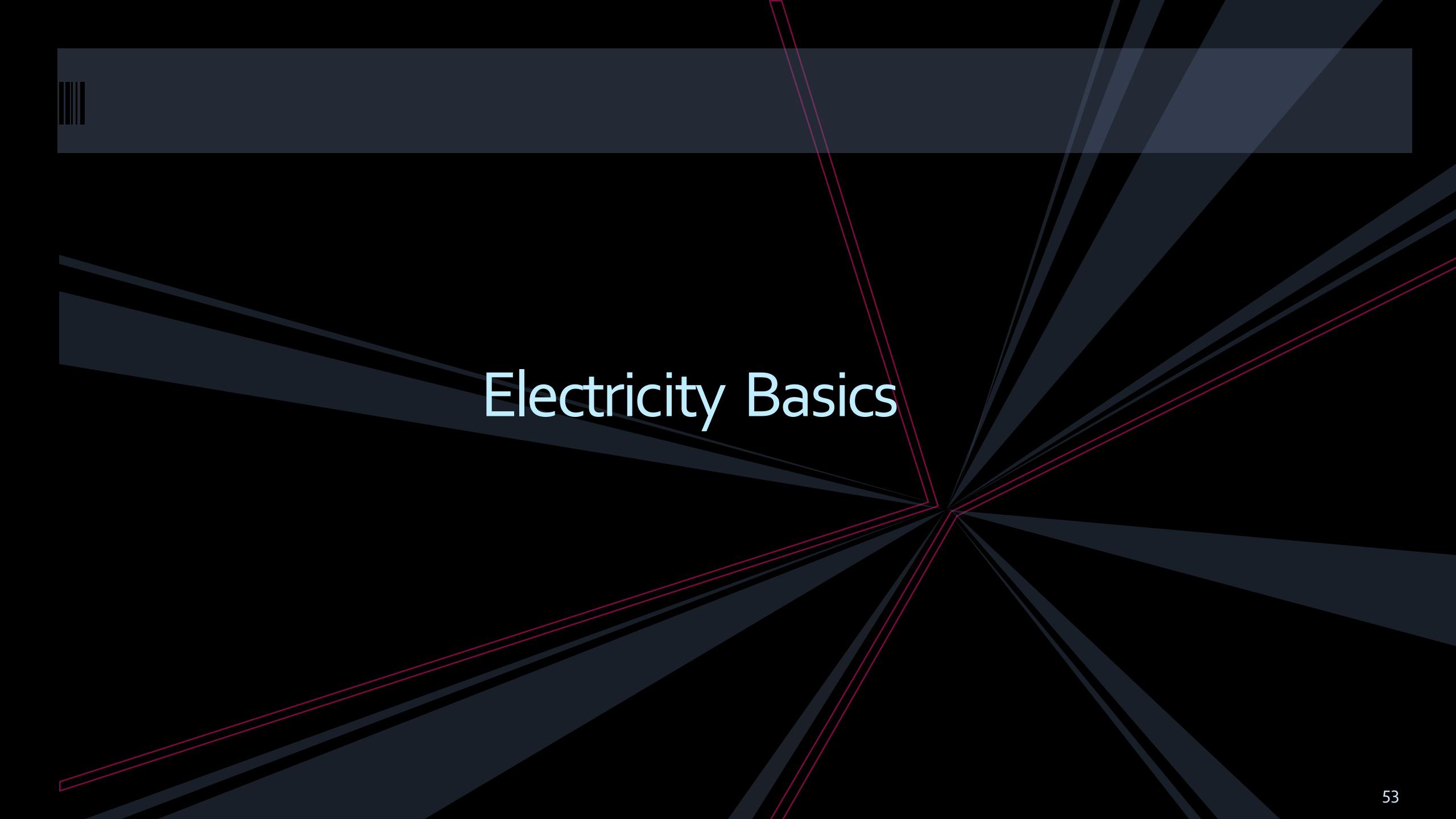


PHYSICS AND CHEMISTRY ARE COMING

imgflip.com

Outline of the story

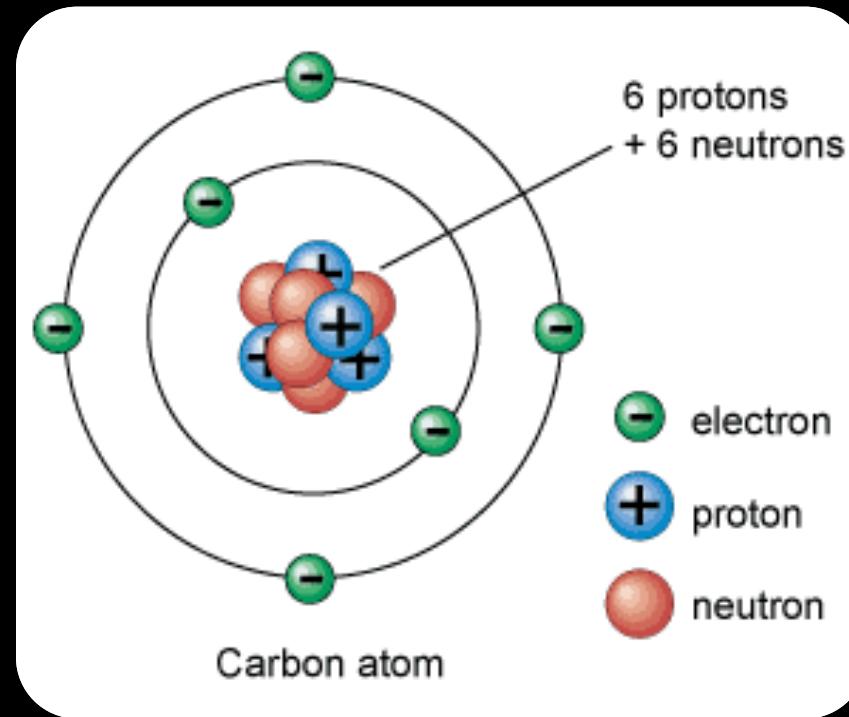
- Electricity, basic concepts
- Insulators, conductors, in between ..., Semiconductors
- Impure semiconductors, p-type / n-type
- Put p-type and n-type together -- pn-junction
- Apply voltage to a pn-junction – principle of transistors
- A real-world manufacturing of transistor -- MOSFET



Electricity Basics

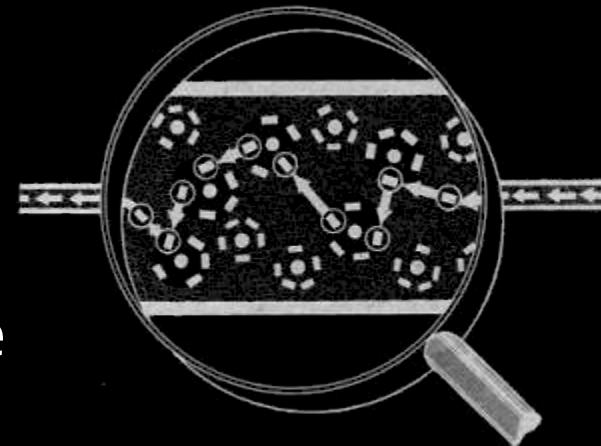
Everything is made out of atoms ...

- Protons are big (hardly move) and positively charged.
- Electrons are small (easily move) and negatively charged.
- Neutrons are big and of course, neutral.
- Overall, an atom is neutral.



What is Electricity?

- Electricity is the **flow** of charged particles (usually electrons) through a material.
 - Electricity could be caused by the flow of protons as well, but since they're so much bigger than electrons, we usually assume that it's the electrons flowing.



How do electrons flow?

They flow ...



Keep this analogy in mind...



Water flowing in a pipe

How do electrons flow?

- Electrons want to flow from regions of **high electrical potential** (many electrons) to regions of **low electrical potential** (fewer electrons).
 - Like water flows from high to low.
- This potential is referred to as **voltage (V)**.
- The rate of this flow is called the **current (I)**.
- **Resistance ($I = V / R$)** is like how narrow the pipe is.
 - Narrower water pipe has higher resistance.

Note

The direction of the current is **opposite** to the direction of the electron movement, because electrons are **negatively** charged.

More on Resistance

- Electrical resistance indicates how well a material allows electricity to flow through it:
 - High resistance (aka insulators) don't conduct electricity at all.
 - Low resistance (aka conductors) conduct electricity well, and are generally used for wires.
- Semiconductors are somewhere in between conductors and insulators, which makes it interesting...

Outline of the story

- Electricity, basic concepts
 - Insulators, conductors, in between ...,
-  Semiconductors
- Impure semiconductors, p-type / n-type
 - Put p-type and n-type together -- pn-junction
 - Apply voltage to a pn-junction – principle of transistors
 - A real-world manufacturing of transistor -- MOSFET

Semiconductors

Here comes the chemistry

Periodic Table of Elements

The Periodic Table of Elements displays the following information:

- Atomic Number (1-118):** The element number is shown in the top-left corner of each cell.
- Name:** The element name is written below the symbol.
- Symbol:** The element symbol is located above the atomic number.
- Atomic Mass:** The element's atomic mass is provided in parentheses below the symbol.
- Phase:** A legend indicates phase states: Solid (C), Liquid (L), Gas (G), and Unknown (U).
- Classification:** Elements are categorized into Metals (yellow), Nonmetals (green), and Noble gases (light blue).
- Groups:** Groups are numbered 1 through 18 across the top of the table.
- Periods:** Periods are numbered 1 through 7 down the left side of the table.
- Isotopes:** For elements with no stable isotopes, the mass number of the isotope with the longest half-life is given in parentheses.
- Design and Interface:** Copyright © 1997 Michael Dayah (michael@dayah.com). <http://www.ptable.com/>

Ptable.com

57 La	58 Ce	59 Pr	60 Nd	61 Pm	62 Sm	63 Eu	64 Gd	65 Tb	66 Dy	67 Ho	68 Er	69 Tm	70 Yb	71 Lu
Lanthanum (138.90547)	Cerium (140.116)	Praseodymium (140.90785)	Neodymium (144.242)	Promethium (147.915)	Samarium (150.38)	Europium (151.964)	Gadolinium (157.25)	Terbium (159.52535)	Dysprosium (162.500)	Holmium (164.93032)	Erbium (167.259)	Thulium (169.93421)	Ytterbium (173.054)	Lutetium (174.9668)
89 Ac	90 Th	91 Pa	92 U	93 Np	94 Pu	95 Am	96 Cm	97 Bk	98 Cf	99 Es	100 Fm	101 Md	102 No	103 Lr
Actinium (227)	Thorium (232.03806)	Postrachium (231.03588)	Uranium (238.02891)	Neptunium (237)	Plutonium (244)	Americium (243)	Curium (247)	Berkelium (247)	Californium (251)	Einsteinium (252)	Fermium (257)	Mendelevium (258)	Nobelium (259)	Lawrencium (202)

Periodic Table of Elements

silicon

Germanium

1	2	3	4	5	6	7	8	9	10	11	16	17	18																																																				
1 H Hydrogen 1.00794	2 He Helium 4.002602	3 Li Lithium 6.941	4 Be Beryllium 9.012182	5 C Solid	6 Hg Liquid	7 H Gas	8 Rf Unknown	9	10	11	16 N Nitrogen 14.0067	17 O Oxygen 15.9994	18 F Fluorine 18.9984032																																																				
11 Na Sodium 22.98976928	12 Mg Magnesium 24.3050	13 Al Aluminum 26.9815395	14 Si Silicon 28.0855	15 P Phosphorus 30.973762	16 S Sulfur 32.065	17 Cl Chlorine 35.453	18 Ar Argon 39.948	K	K	K	K	K	K																																																				
19 K Potassium 39.0983	20 Ca Calcium 40.078	21 Sc Scandium 44.955912	22 Ti Titanium 47.867	23 V Vanadium 50.9415	24 Cr Chromium 51.9981	25 Mn Manganese 54.938045	26 Fe Iron 55.845	27 Co Cobalt 58.933195	28 Ni Nickel 58.6934	29 Cu Copper 63.546	30 Zn Zinc 65.38	31 Ga Gallium 69.723	32 Ge Germanium 71.976	33 As Arsenic 75.99	34 Se Selenium 78.904	35 Br Bromine 80.904	36 Kr Krypton 83.798	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K	K																																
37 Rb Rubidium 85.4678	38 Sr Strontium 87.62	39 Y Yttrium 88.90585	40 Zr Zirconium 91.224	41 Nb Niobium 91.90638	42 Mo Molybdenum 95.98	43 Tc Technetium (97.9072)	44 Ru Ruthenium 101.07	45 Rh Rhodium 102.90550	46 Pd Palladium 106.42	47 Ag Silver 107.8882	48 Cd Cadmium 112.411	49 In Indium 114.76	50 Sb Antimony 121.780	51 Te Tellurium 127.8	52 I Iodine 131.90447	53 Xe Xenon 131.293	54 Cs Cesium 132.9054519	56 Ba Barium 137.327	57 Hf Hafnium 178.49	58 Ta Tantalum 180.94788	59 W Tungsten 183.84	60 Re Rhenium 186.207	61 Os Osmium 190.23	62 Ir Iridium 192.2	63 Pt Iridium (192.2)	64 Dy Dysprosium 162.500	65 Tb Terbium 158.92535	66 Ho Holmium 168.93032	67 Er Erbium 167.259	68 Tm Thulium 168.93421	69 Yb Ytterbium 173.054	70 Lu Lutetium 174.9668	71 Fr Francium (223)	72 Ra Radium (226)	73 Rf Rutherfordium (261)	74 Db Dubnium (262)	75 Sg Seaborgium (256)	76 Bh Bohrium (264)	77 Hs Hassium (277)	78 Mt Meitnerium (288)	79 Ds Darmstadtium (271)	80 Rg Roentgenium (272)	81 Unb Ununbium (285)	82 Uus Ununquadium (289)	83 Uuh Ununpentium (288)	84 Po Polonium (208.9824)	85 At Astatine (210.9824)	86 Rn Radon (222.0178)	87 Fr Francium (223)	88 Ra Radium (226)	89 Rf Rutherfordium (261)	90 Th Thorium 232.03806	91 Pa Protactinium 231.03588	92 U Uranium 238.02891	93 Np Neptunium (237)	94 Pu Plutonium (244)	95 Am Americium (243)	96 Cm Curium (247)	97 Bk Berkellium (247)	98 Cf Californium (251)	99 Es Einsteinium (252)	100 Fm Fermium (257)	101 Md Mendelevium (258)	102 No Nobelium (259)	103 Lr Lawrencium (262)
For elements with no stable isotopes, the mass number of the isotope with the longest half-life is in parentheses.																																																																	
Design and Interface Copyright © 1997 Michael Dayah (michael@dayah.com). http://www.ptable.com/																																																																	
57 La Lanthanum 138.90547	58 Ce Cerium 140.116	59 Pr Praseodymium 140.90765	60 Nd Neodymium 144.242	61 Pm Promethium 145	62 Sm Samarium 150.38	63 Eu Europium 151.964	64 Gd Gadolinium 157.25	65 Tb Terbium 158.92535	66 Dy Dysprosium 162.500	67 Ho Holmium 168.93032	68 Er Erbium 167.259	69 Tm Thulium 168.93421	70 Yb Ytterbium 173.054	71 Lu Lutetium 174.9668																																																			
89 Ac Actinium (227)	90 Th Thorium 232.03806	91 Pa Protactinium 231.03588	92 U Uranium 238.02891	93 Np Neptunium (237)	94 Pu Plutonium (244)	95 Am Americium (243)	96 Cm Curium (247)	97 Bk Berkellium (247)	98 Cf Californium (251)	99 Es Einsteinium (252)	100 Fm Fermium (257)	101 Md Mendelevium (258)	102 No Nobelium (259)	103 Lr Lawrencium (262)																																																			

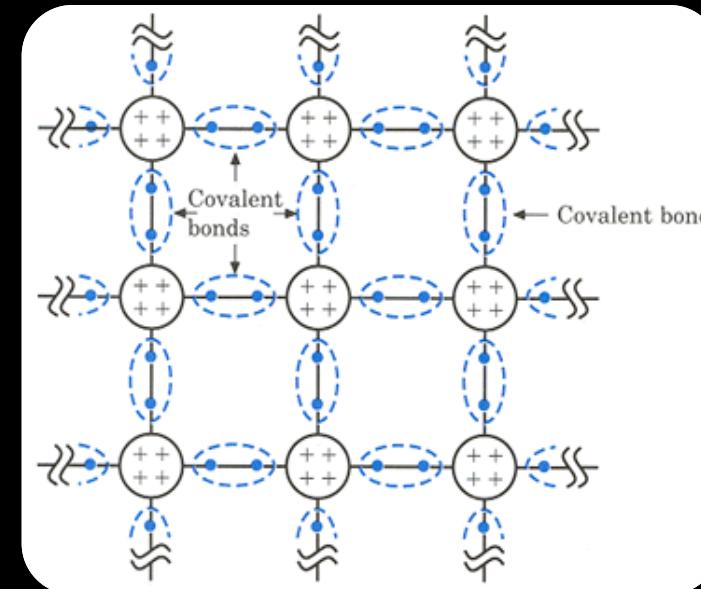
Conductivity of Semiconductors

- Semiconductor materials (e.g., silicon and germanium) straddle the boundary between **conductors** and **insulators**, behaving like one or the other, depending on factors like temperature and **impurities** in the material.

Impurity

Pure semiconductor is pretty stable

- Each atom has **4 electrons**, forming **bonds** with other atoms, and the structure is pretty stable.
- At room temperature, a weak current will flow through the material, much less than that of a conductor.

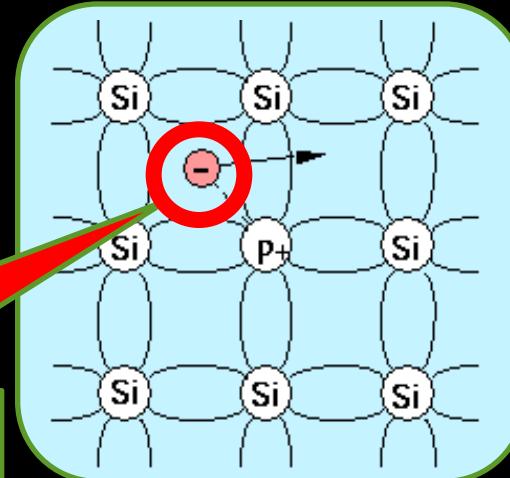


Encourage semiconductor's conductivity

N-type:

Add some atoms with **5** valence electrons, such as Phosphorus.

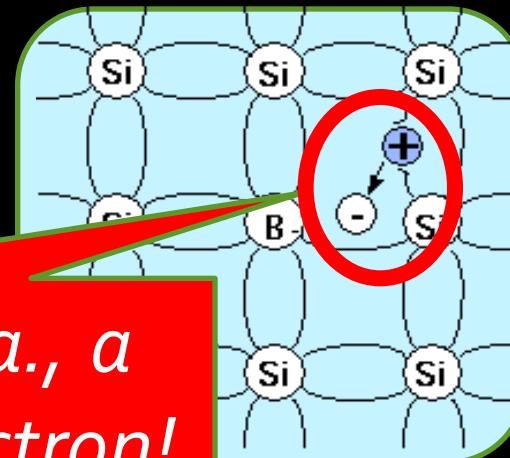
An extra electron!



P-type:

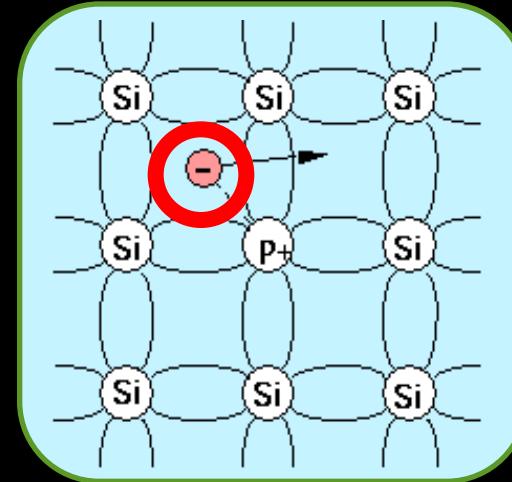
Add some atoms with **3** valence electrons, such as Boron.

A missing electron, a.k.a., a "hole", like a positive electron!



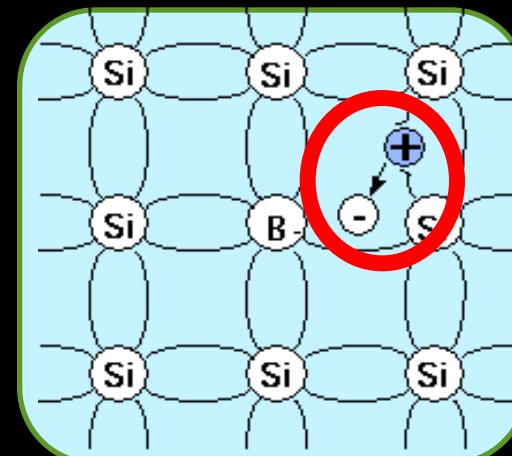
Encourage semiconductor's conductivity

The extra electrons and the holes are **charge carriers**, which can move **freely** through the materials.



Thus the conductivity is encouraged.

This process of adding stuff is called **doping**, (n or p type).



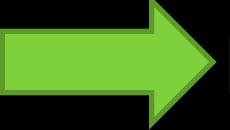
Free electrons
move like



Free holes
move like



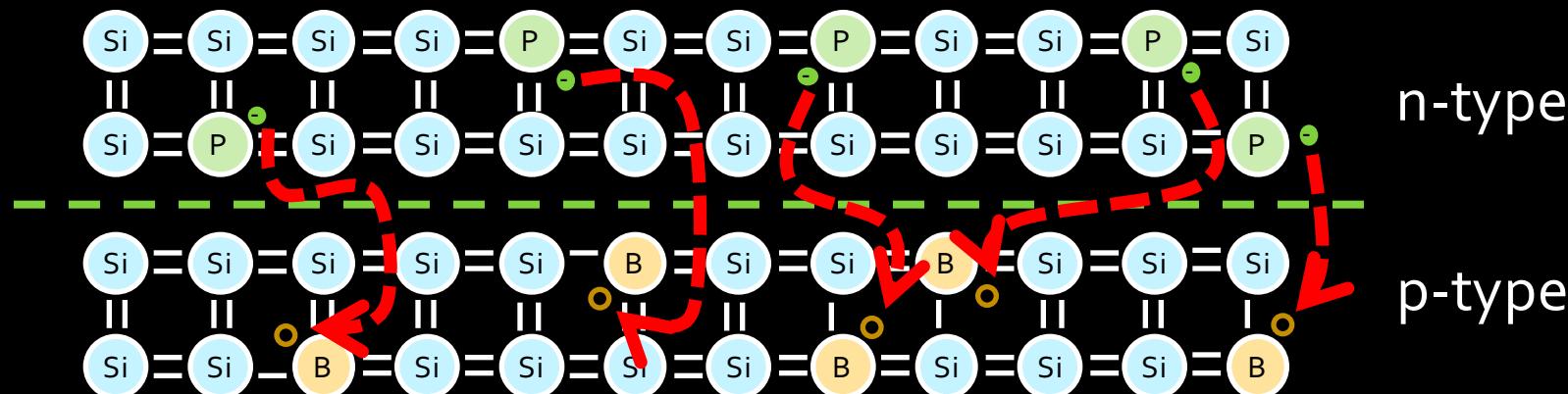
Outline of the story

- Electricity, basic concepts
- Insulators, conductors, in between ...,
Semiconductors
- Impure semiconductors, **p-type / n-type**
 Put p-type and n-type together -- **pn-junction**
- Apply voltage to a pn-junction – **principle of transistors**
- A real-world manufacturing of transistor -- **MOSFET**

PN-junctions

Bringing p and n together

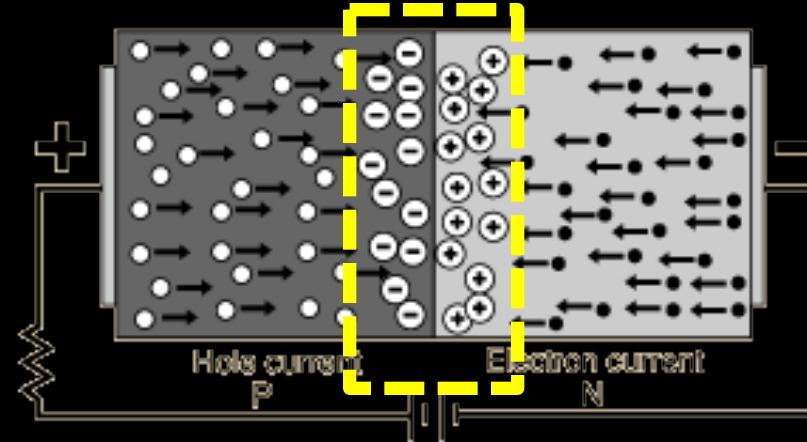
- What happens if you brought some p-type material into contact with some n-type material?



- The **electrons** at the surface of the n-type material are drawn to the **holes** in the p-type.

p-n Junctions

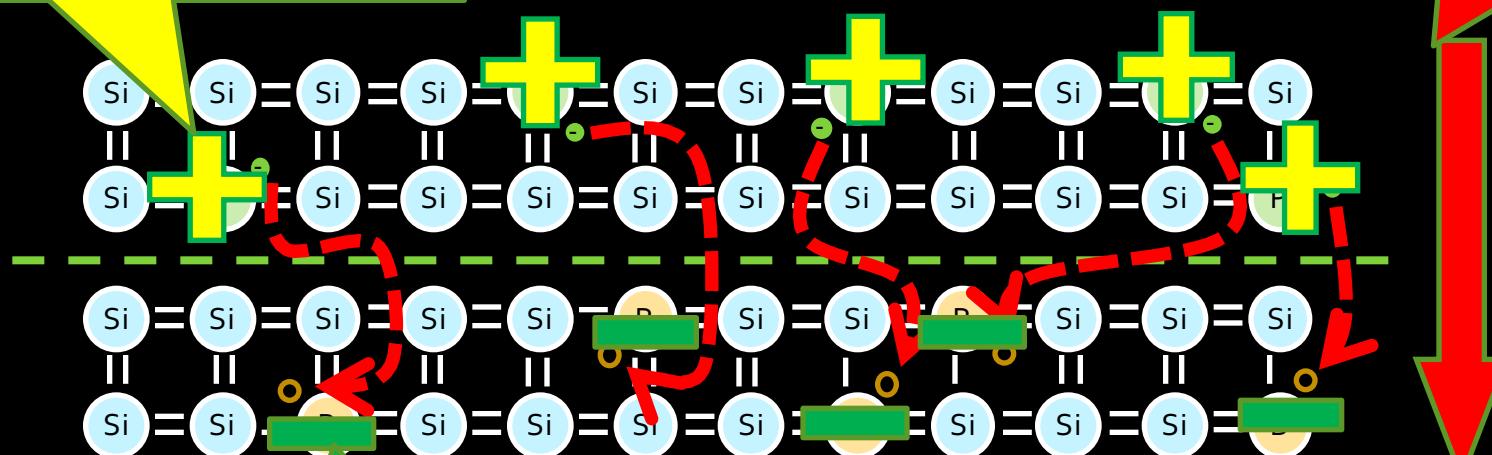
- When left alone, the electrons from the n section of the junction will fill the holes of the p section, cancelling each other and create a section with no free carriers called the depletion layer.
- Once this depletion layer is wide enough, the doping atoms that remain will create an electric field in that region.



Because lost
electron

n-type

Electrons' initial movement
(attracted by holes)



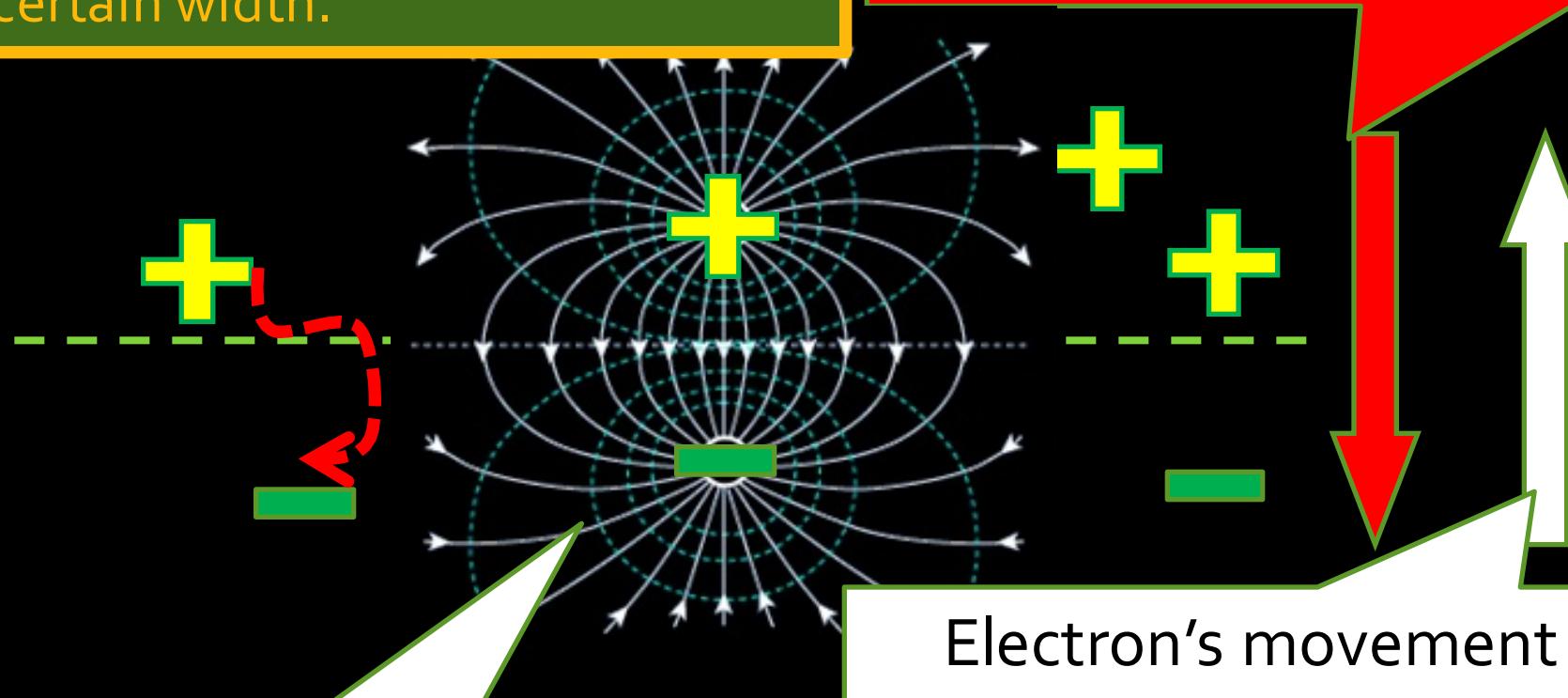
Because gained
electron

p-type

Diffusion increases the width of depletion layer, and drift draws it back. An **equilibrium** is reached, when the depletion layer is of a certain width.

“Diffusion”

Electrons' initial movement
(attracted by holes)

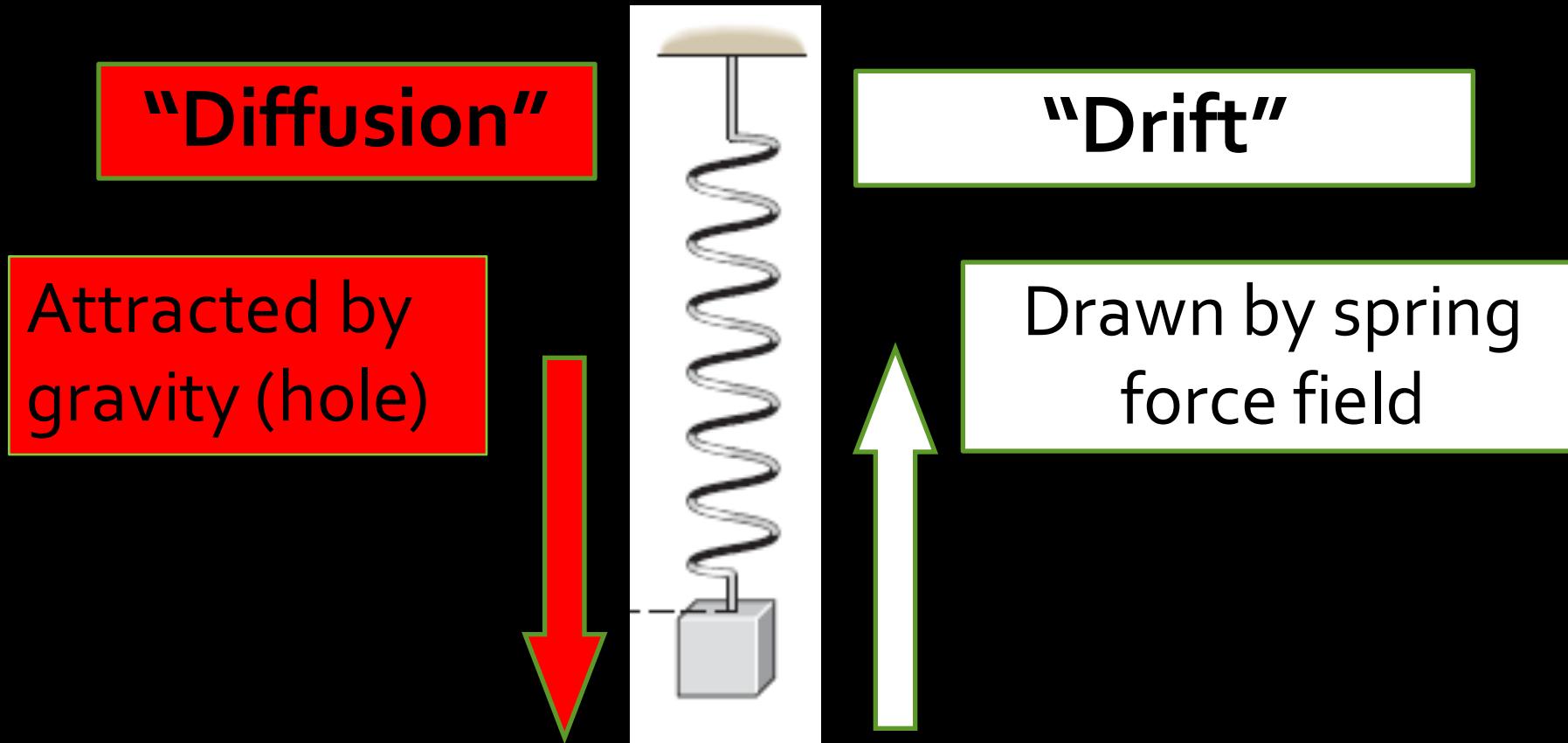


Electric field

Electron's movement
drawn by the electric field

“Drift”

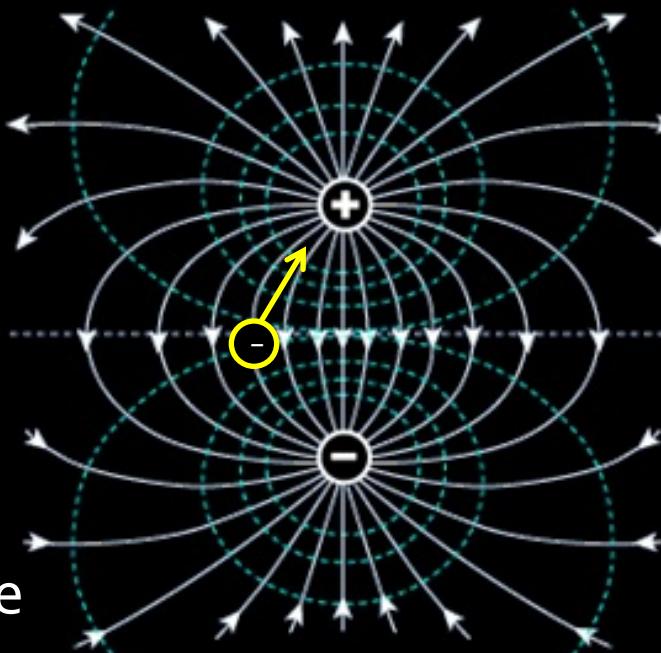
Analogy: Spring with weight



An equilibrium is reached when the spring is stretched by a certain length.

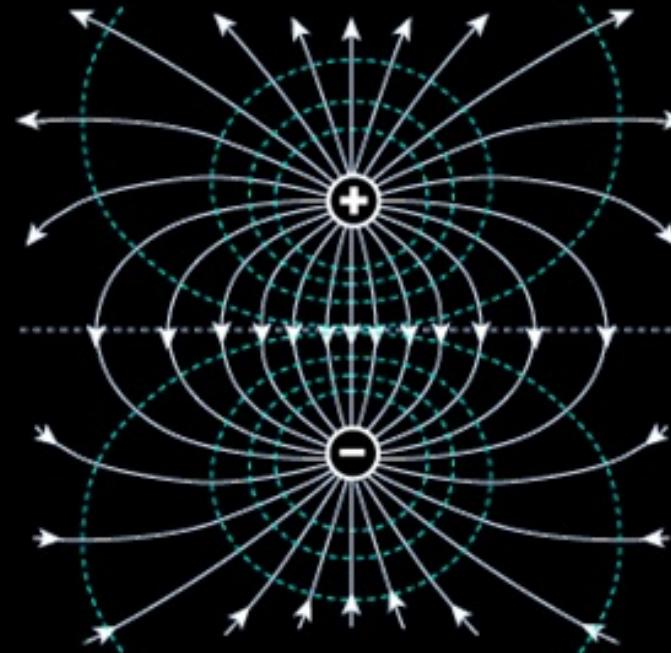
Electric fields (after class reading)

- What is an electric field?
 - When a phosphorus atom loses its electron, the atom develops an overall positive charge.
 - Similarly, when a boron atom takes on an extra electron, that atom develops an overall negative charge.
 - If an electron was dropped between the two, it would be attracted to the phosphorus atom, and repelled by the boron atom.
 - This effect is called an **electric field**.

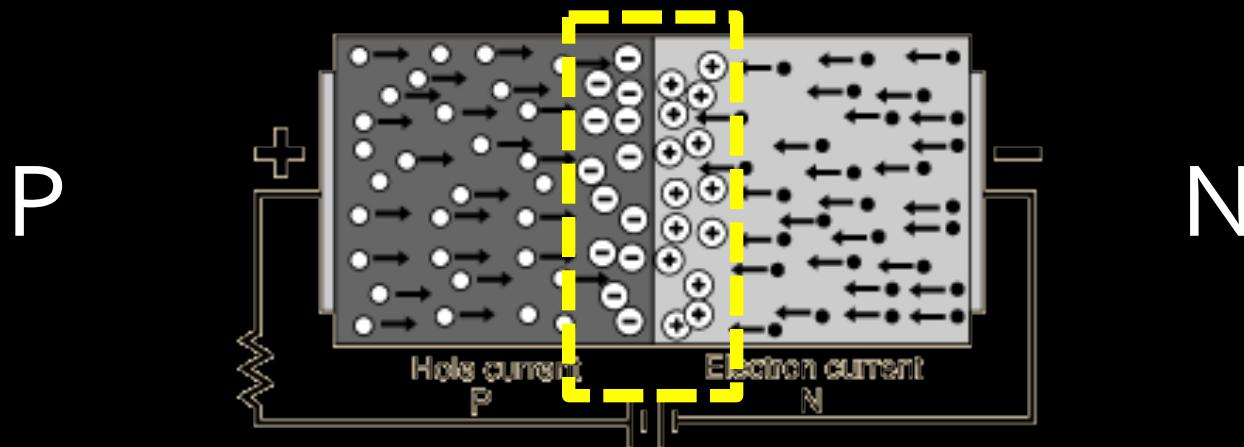


Electric fields (after class reading)

- A depletion layer is made up of many of these electrically imbalanced phosphorus and boron atoms.
- The electric field caused by these atoms will cause holes to flow back to the p section, and electrons to flow back to the n section.
 - The current caused by this electric field is called **drift**.
 - The current caused by the initial electron/hole recombination is called **diffusion**.
- At rest, these two currents reach **equilibrium**.



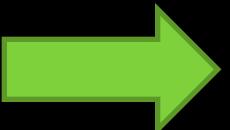
Summary of pn-junction



When we put **p** and **n** together, they will form a depletion layer with electric field in it.

The depletion layer grows up to a certain **width**, until equilibrium is reached.

Outline of the story

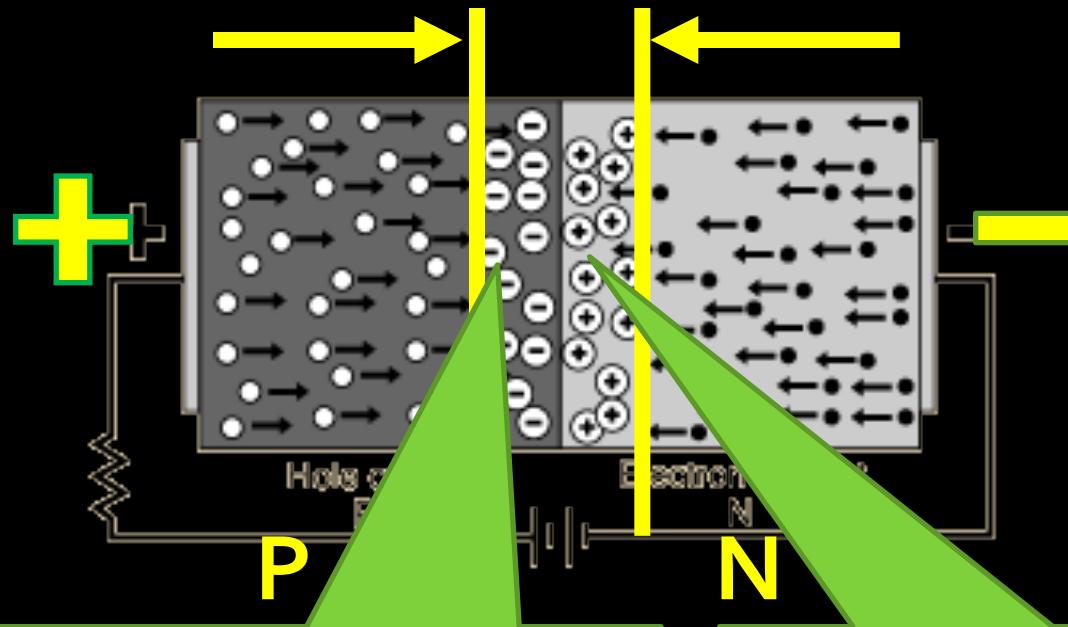
- Electricity, basic concepts
 - Insulators, conductors, in between ...,
Semiconductors
 - Impure semiconductors, **p-type / n-type**
 - Put p-type and n-type together -- **pn-junction**
-  Apply voltage to a pn-junction – **principle of transistors**
- A real-world manufacturing of transistor -- **MOSFET**

Apply voltage to a PN-junction

It could be applied in two possible **directions**

- **Positive voltage to the P side**
- **Positive voltage to the N side**

Forward Bias (Positive voltage to P)

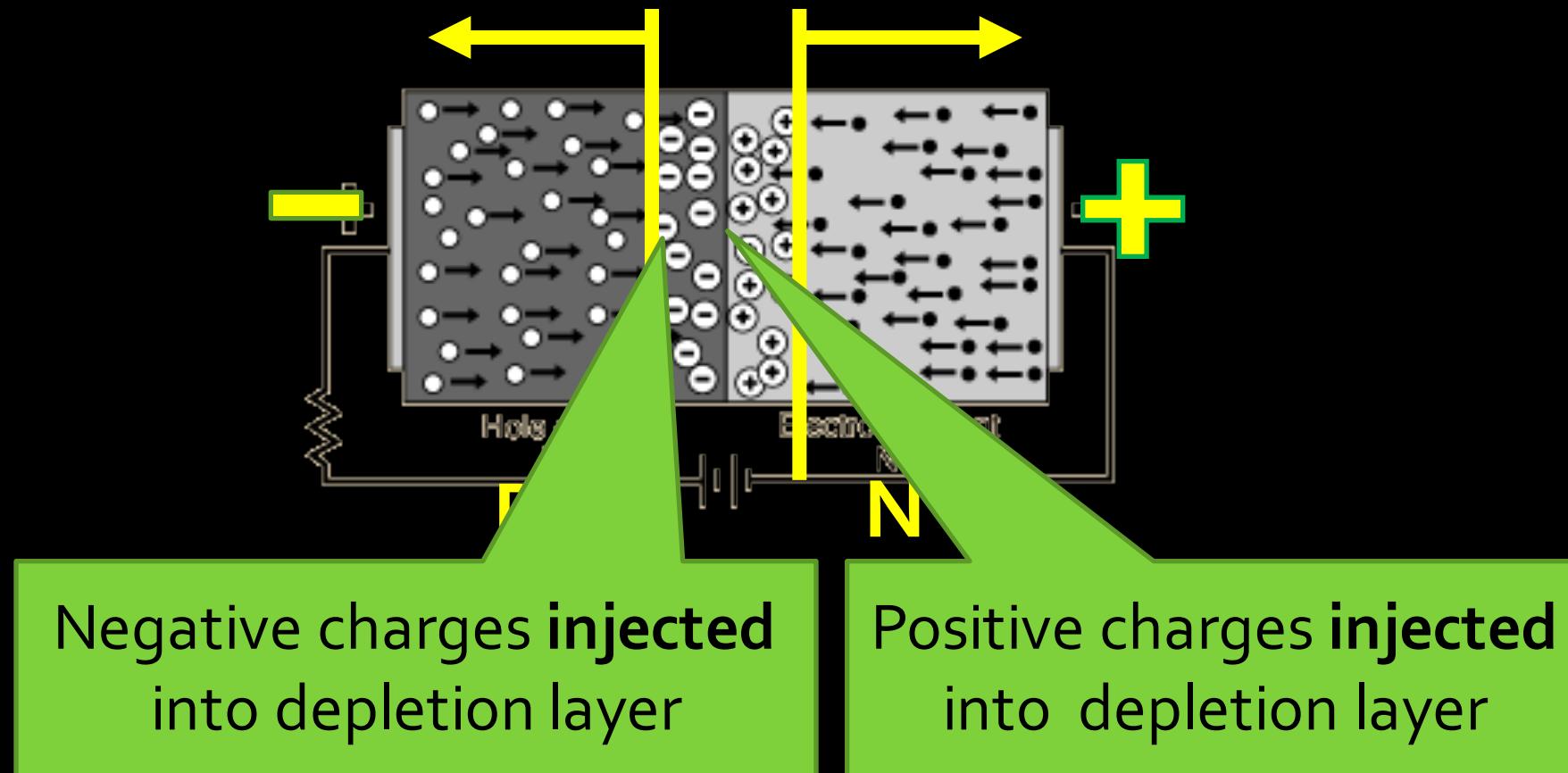


Negative charges sucked
out of depletion layer

Positive charges sucked
out of depletion layer

Depletion layer becomes **narrower**.

Reverse Bias (Positive voltage to N)



Depletion layer becomes **wider**.

Apply **forward bias**

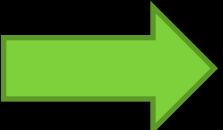
- Depletion layer narrower
- Easier to travel through
- Better conductivity
- Like switch **connected**

Apply **reverse bias**

- Depletion layer wider
- Harder to travel through
- Worse conductivity
- Like switch **disconnected**

That's how transistors work!

Outline of the story

- Electricity, basic concepts
 - Insulators, conductors, in between ...,
Semiconductors
 - Impure semiconductors, **p-type / n-type**
 - Put p-type and n-type together -- **pn-junction**
 - Apply voltage to a pn-junction – **principle of transistors**
-  A real-world manufacturing of transistor -- **MOSFET**

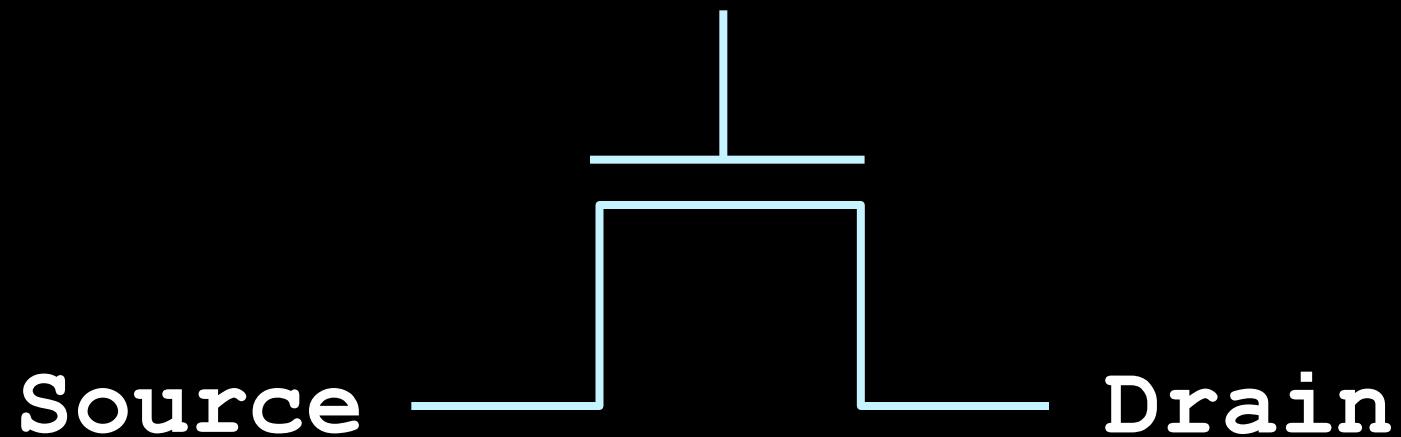
Creating transistors

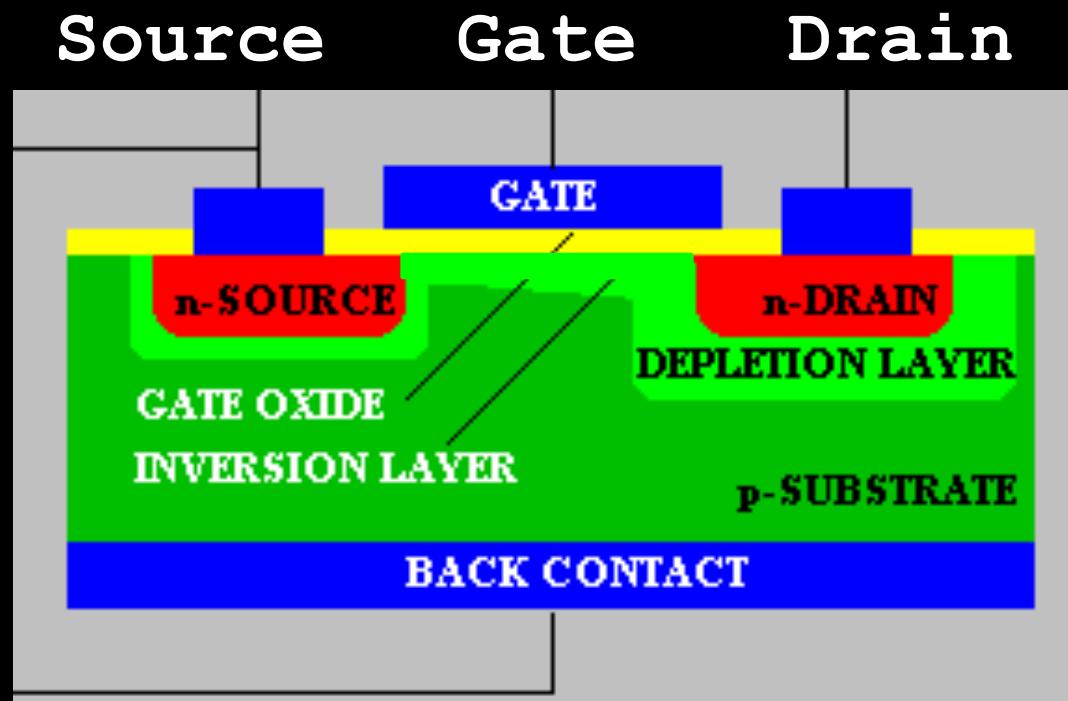
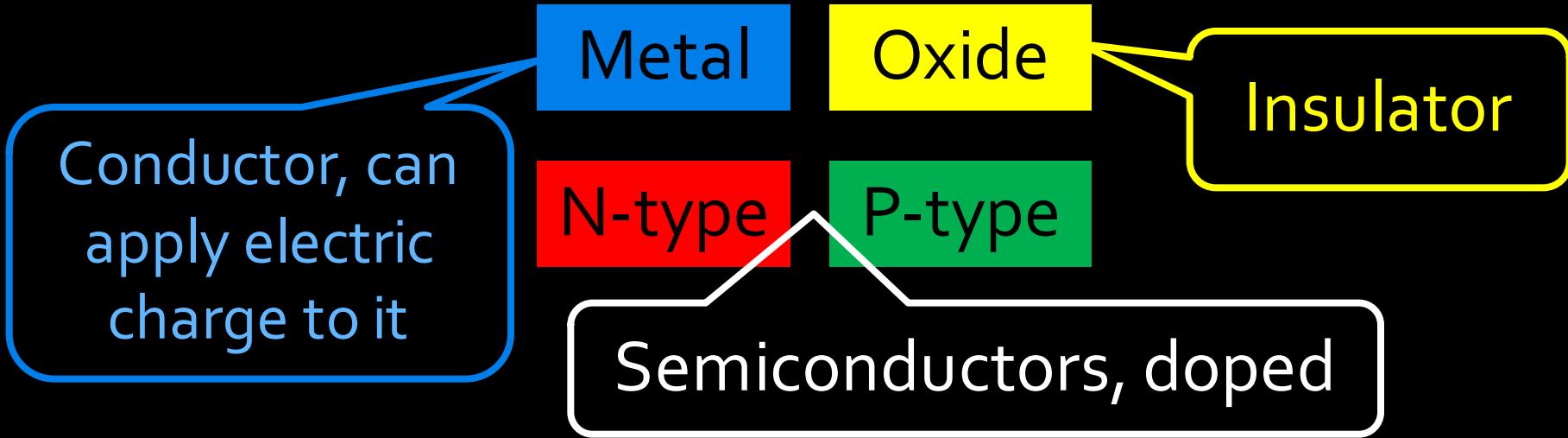
- Transistors use the characteristics of p-n junctions to create more interesting behaviour.
- Three main types:
 - Bipolar Junction Transistors (BJTs)
 - Metal Oxide Semiconductor Field Effect Transistor (MOSFET)
 - Junction Field Effect Transistor (JFET)
- The last two are part of the same family, but we'll only look at the MOSFET for now.

Metal Oxide Semiconductor Field Effect Transistor

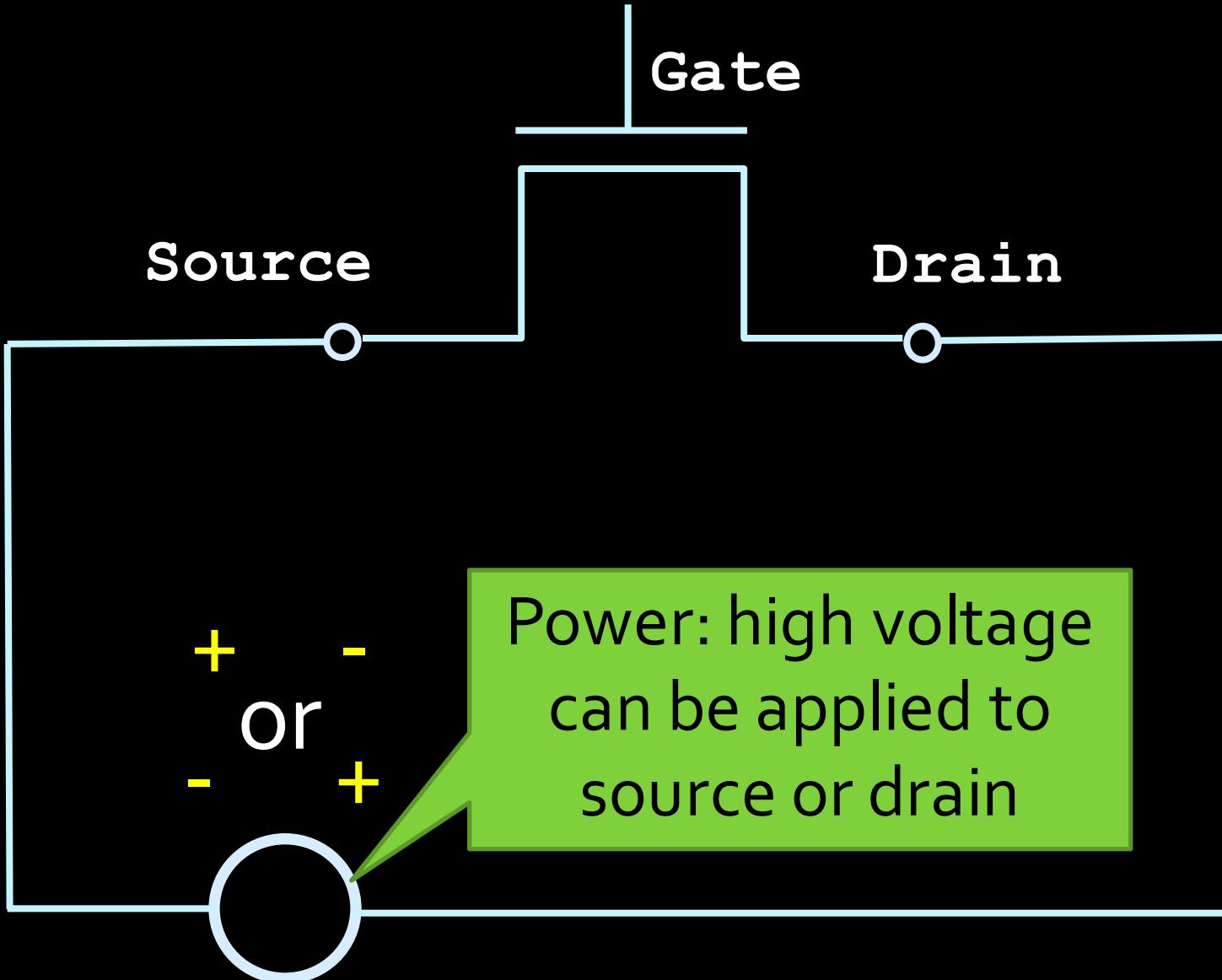


Gate

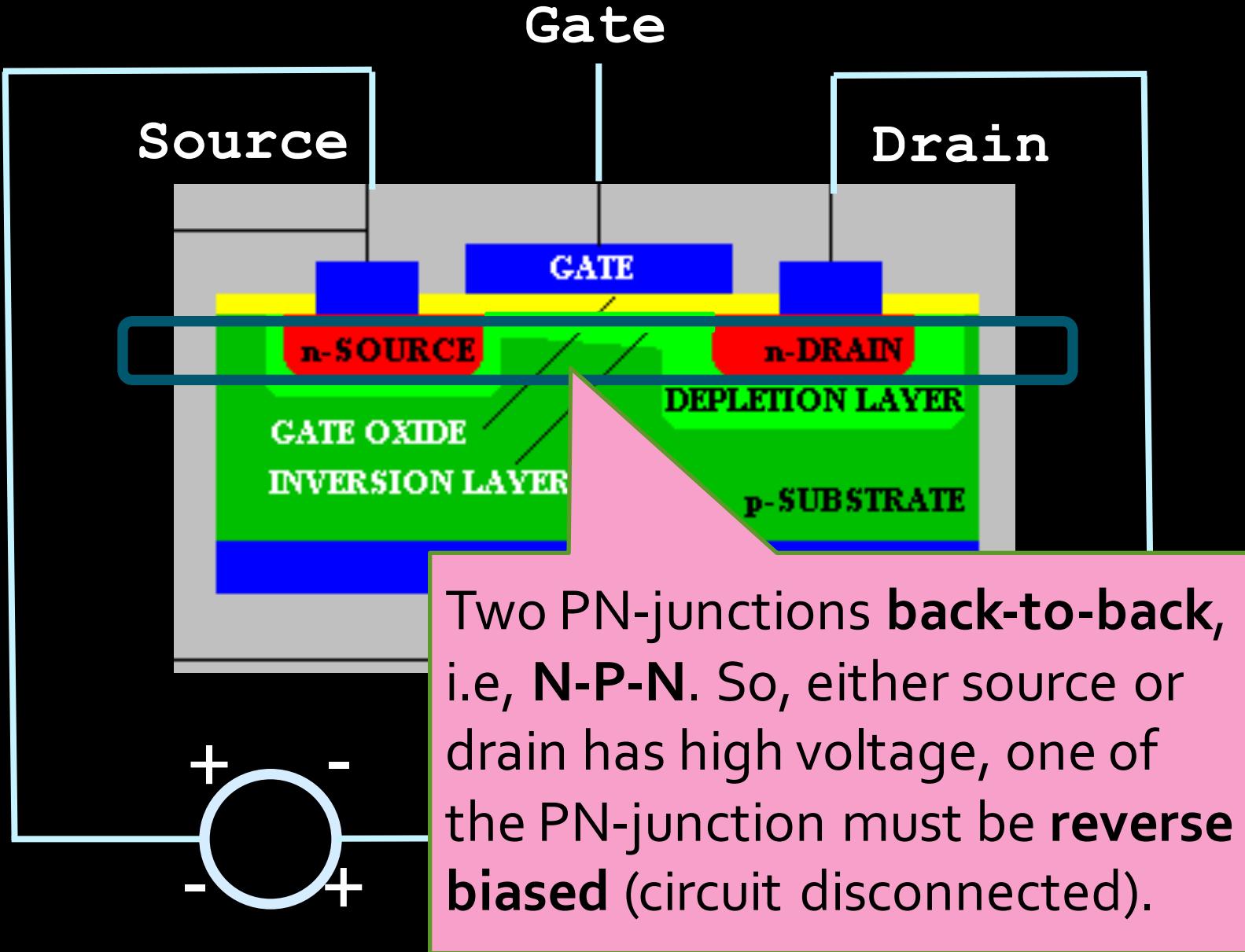




Put a MOSFET into a circuit



Put it into a circuit



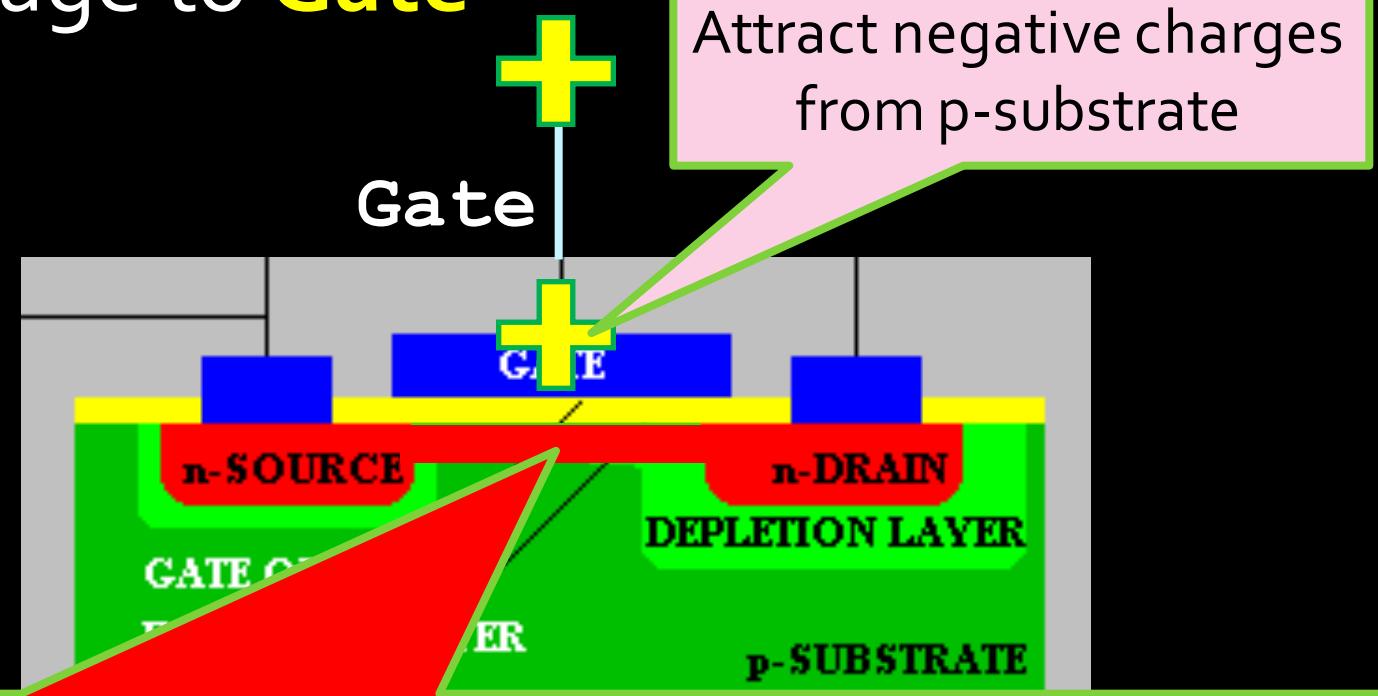
Metal

Oxide

N-type

P-type

But things change if we apply high voltage to **Gate**

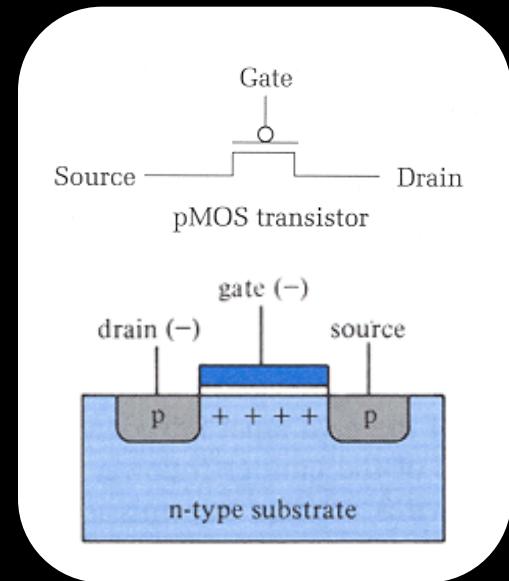
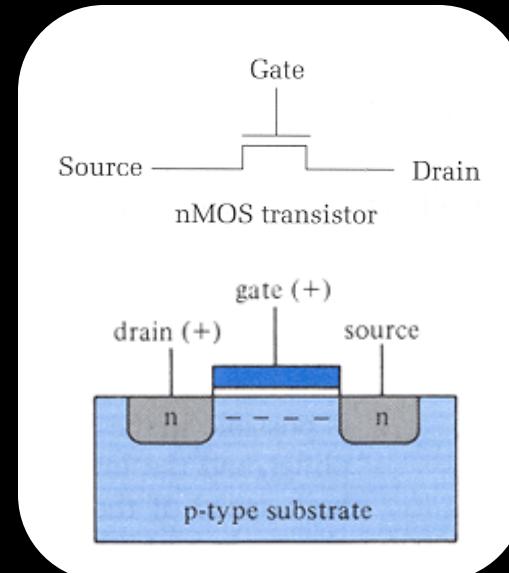


Create n-type **channel** between source and drain, **CIRCUIT CONNECTED**

The wider the channel, the higher the current

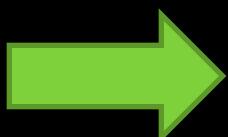
Two types of MOSFET

- **nMOS** (what we just describe)
 - N-P-N
 - Gate high, connected
 - Gate low, disconnected
- **pMOS** (opposite to nMOS)
 - P-N-P
 - Gate low, connected
 - Gate high, disconnected



Outline of the story

- Electricity, basic concepts
- Insulators, conductors, in between ...,
Semiconductors
- Impure semiconductors, p-type / n-type
- Put p-type and n-type together -- pn-junction
- Apply voltage to a pn-junction – principle of transistors
- A real-world manufacturing of transistor – MOSFET



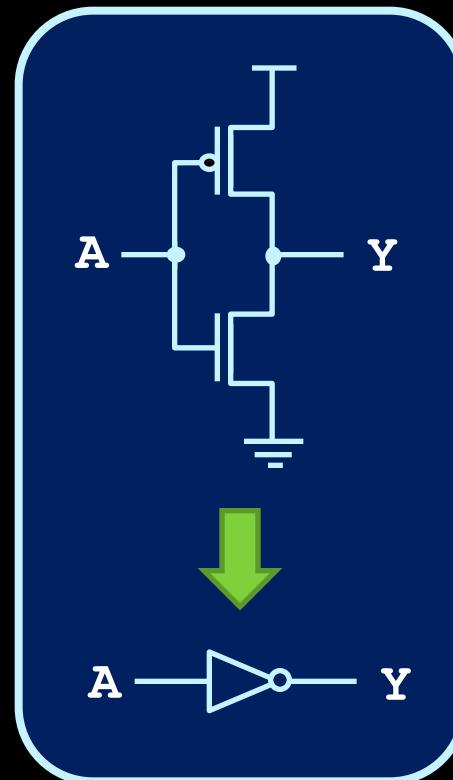
Use transistors build Logic Gates

Transistors to Logic Gates

Create gates using a combination of transistors

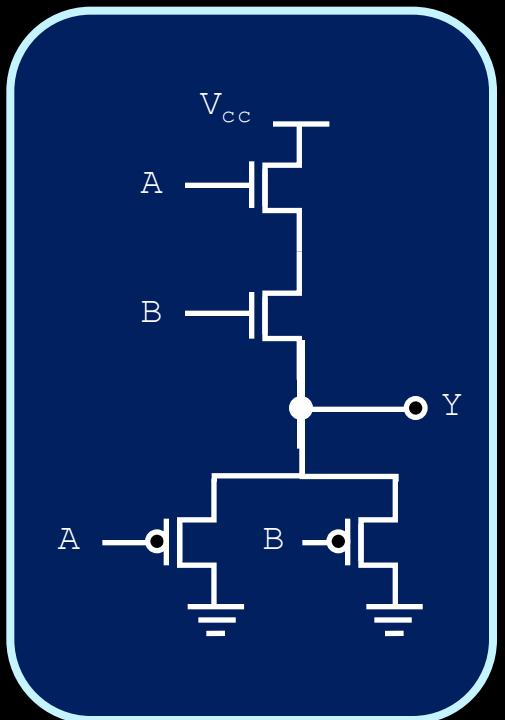
Physical data:

- “High” input = 5V
- “Low” input = 0V
- Switching time ≈ 120 picoseconds
- Switching interval ≈ 10 ns

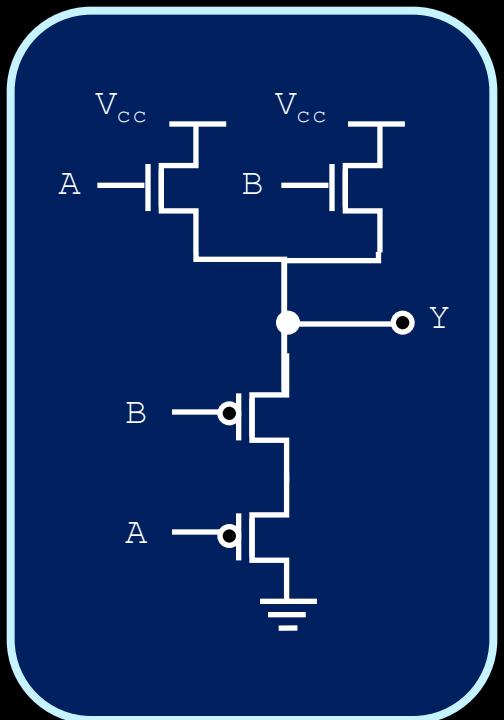


NOT Gate

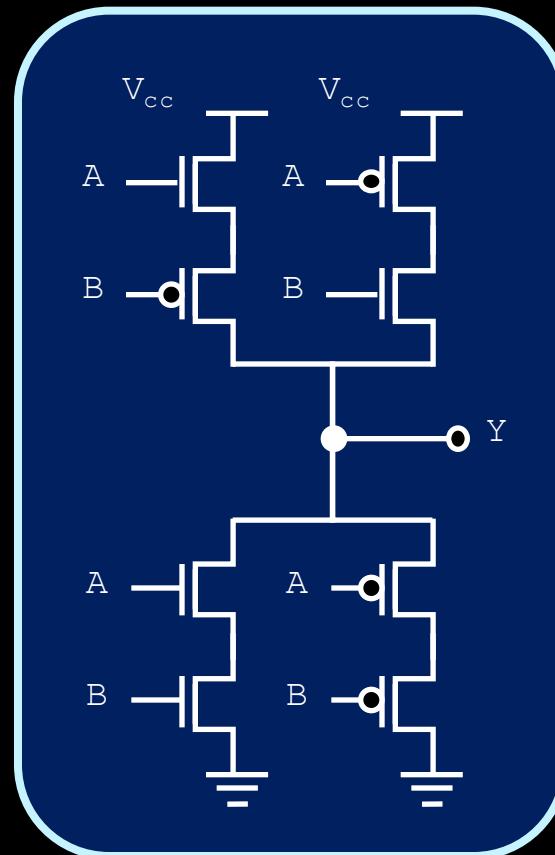
Transistors into gates



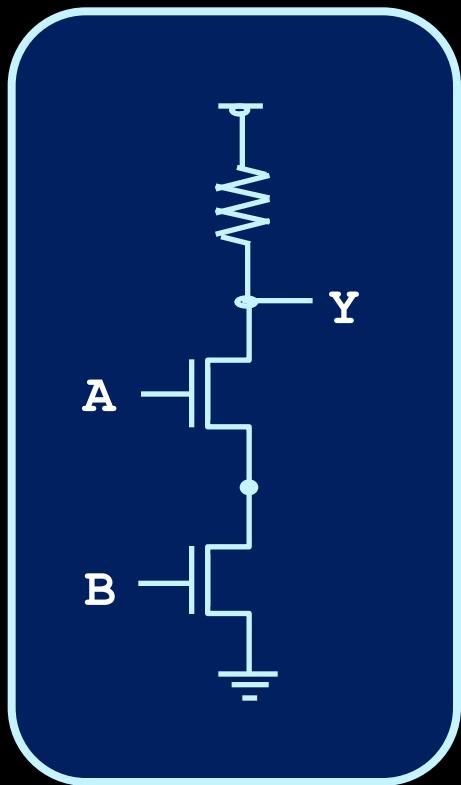
AND



OR

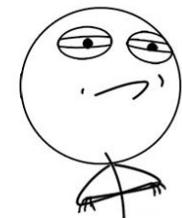


XOR



NAND is the most awesome (and common) logic gate

- It uses **fewer** transistors than other gates
- **All** other logic functions (AND, OR, ...) can be implemented using **only** NAND.



Challenge for home: implement AND, OR, NOT, XOR using only NAND.

Next week:

- Circuit creation

CSC258 Winter 2016

Lecture 2

Lab 1 prep

- You should be registered in a lab section by now; if not, get in now, there is a new section PRA0105 added.
- Lab 1 is about familiarize with everything
 - Create a circuit using Quartus
 - Simulate the circuit using Quartus
 - Burn the circuit into the FPGA board, and test the hardware for real.
- Work in groups of 2 students.

Lab tips

- Read the “Summary of TODOs” part of the handout to quickly know what to do.

6 Summary of TODOs

Below is the summary of the steps to be completed for this lab:

1. Before the lab, read through the Quartus tutorial and/or install the tools on your own computer.
2. Find a partner in the lab to work together.
3. Predict the behaviour of the mystery circuit before implementing it.
4. Implement the mystery circuit using Quartus.
5. Simulate the circuit, explain the waveforms and show them to your TA.
6. Load your the circuit to the DE-2 board, make sure it's working as expected, and show it to your TA.

Evaluation (4 marks in total): 2 marks for attending and making an honest effort; 1 mark for showing and explaining the simulation waveforms; 1 mark for having the circuit working on the DE-2 board.

Lab tips

- Make sure to finish the work that needs to be done **before** the lab.
- Be ready to explain your work to the TA in order to get full mark.
- If your partner can answer but you can't, your partner gets more marks than you.
- Teach your partner! That makes you learn better, too!

Lab 1 tips

- Read the Quartus Tutorial (posted on course website) patiently.
- Follow the tutorial and you'll mostly be mostly fine.
- The DE2 pin assignment posted on course website is useful for loading to DE2 boards, you'll use in future labs over and over again.
- Sometime Quartus crashes unreasonably, just restart it.

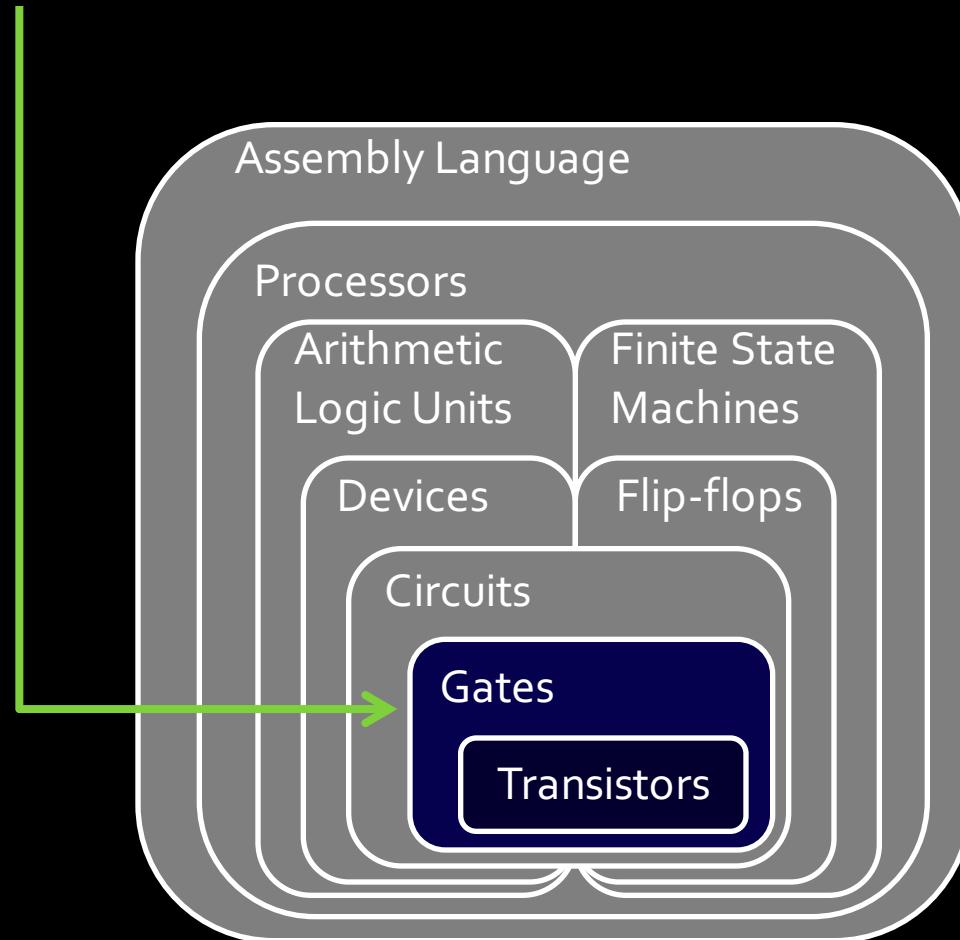
More lab tips

- Back up your work (copy to USB, upload to Dropbox, etc), you may need it for future labs.
- Send feedback using the Weekly Feedback Form after the lab.
 - <http://goo.gl/forms/o248ETWgnS>
 - Or just drop by my office to chat about how it went, while eating candies.

More lab tips

- Consider switching to the new lab section PRAo105, where you can get more help from the TA.

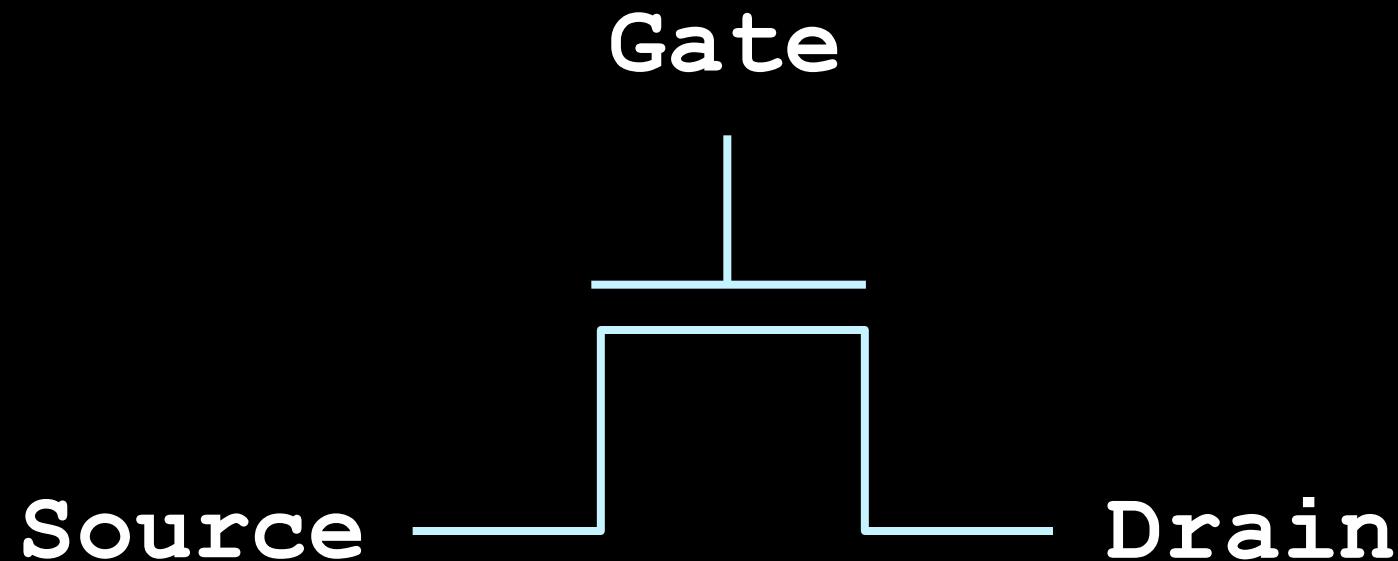
We are here



Recap: Transistors

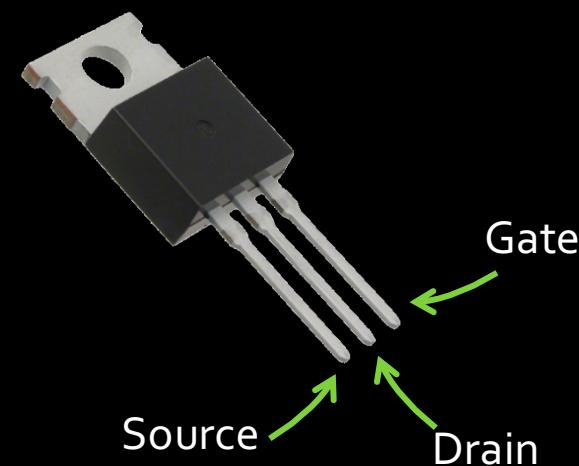
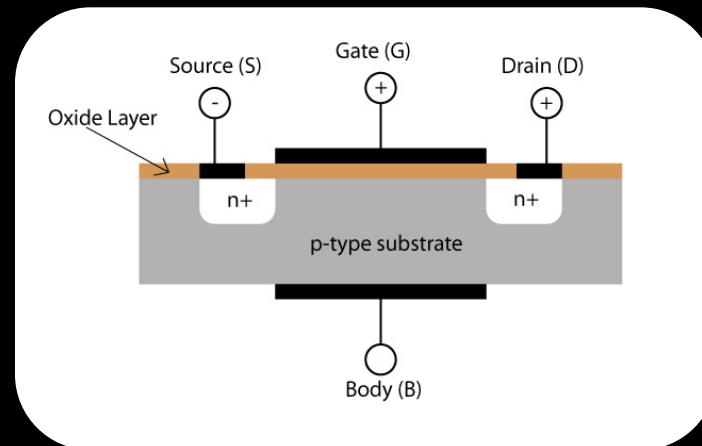
- **Transistors**, made of doped semiconductors put together (PN-Junctions), is like a **resistor** but can **change its resistance**.
- It has two state: connected (switched ON) or disconnected (switched OFF)
 - This is the origin of the 1's and 0's that all current computers are built with. (Quantum computer may do it differently)
- The ON/OFF state of a transistor is control by an electrical signal, like in the MOSFET.

MOSFET



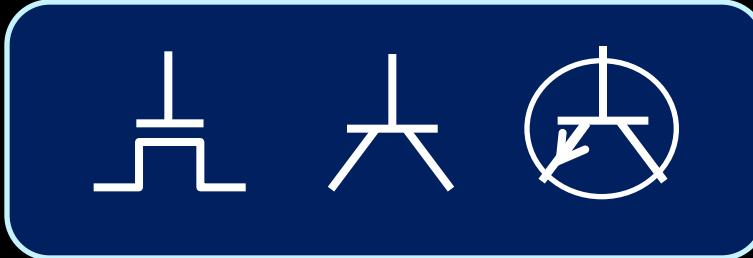
From transistors to gates

- Transistors are semiconductor circuits that can connect the **source** and the **drain** together, depending on the voltage value at the **gate**.
 - For **NPN** MOSFETs (**nMOS**), they are connected when the gate value is high.
 - For **PNP** MOSFETs (**pMOS**), they are connected when the gate value is low.
- These are then used to make **digital logic gates**.

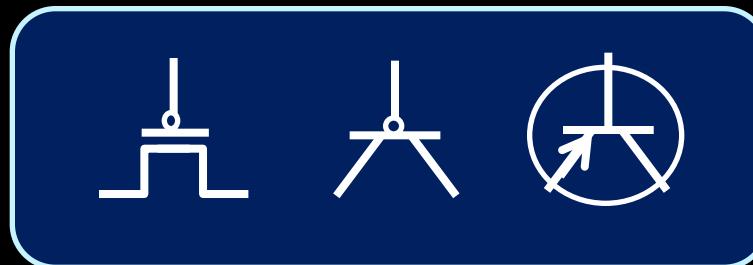


Transistor notation

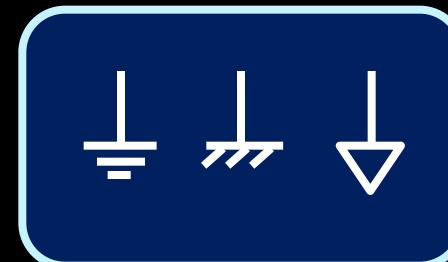
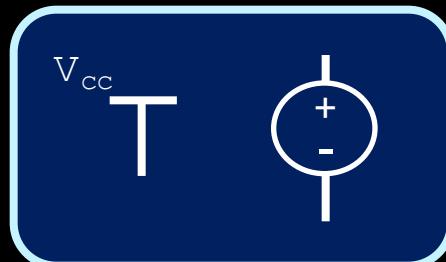
- NPN transistor:



- PNP transistor:

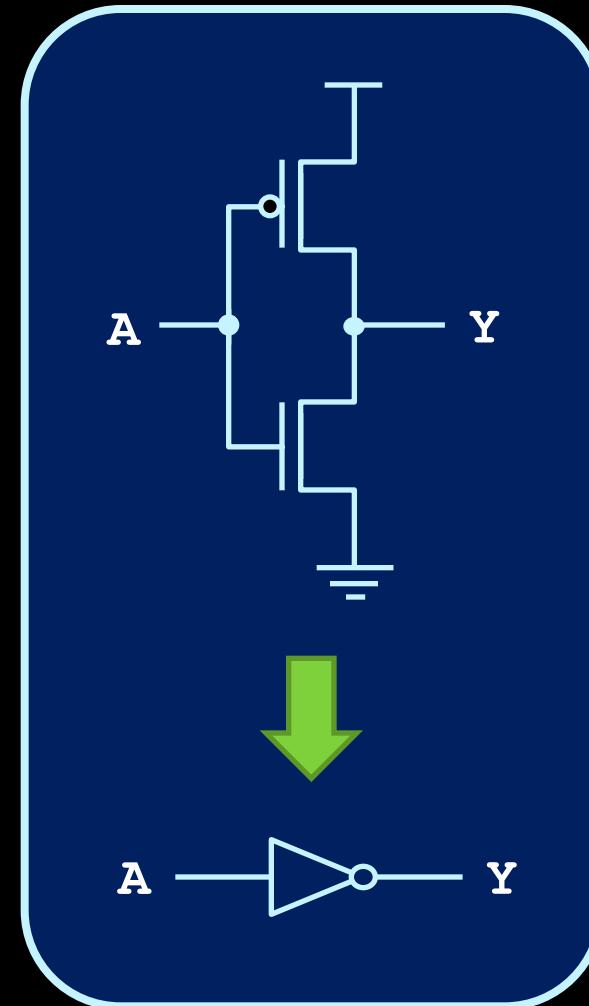


- Voltage values:

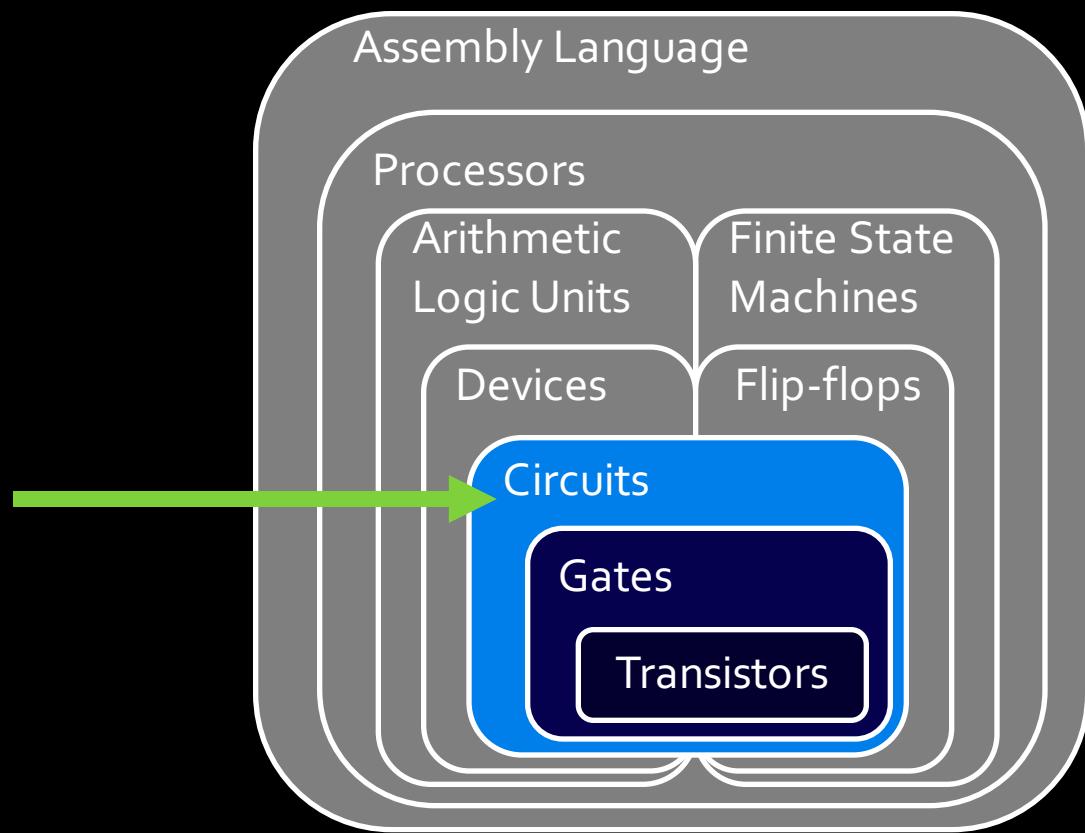


How gates are made

- To create logic gates:
 - Remember that transistors act like faucets for electricity.
 - The inputs to the logic gates determine if the outputs will be connected to high or low voltage.
 - Example: NOT gates:



From gates to circuits



Making logic with gates

- Logic gates like the following allow us to create an output value, based on one or more input values.
 - Each corresponds to Boolean logic that we've seen before in CSC108 and CSC148:



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



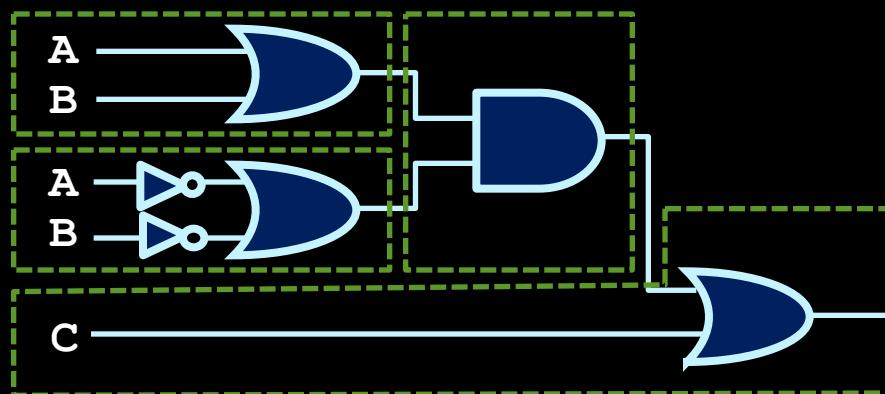
A	Y
0	1
1	0

Making boolean expressions

- So how would you represent Boolean expressions using logic gates?

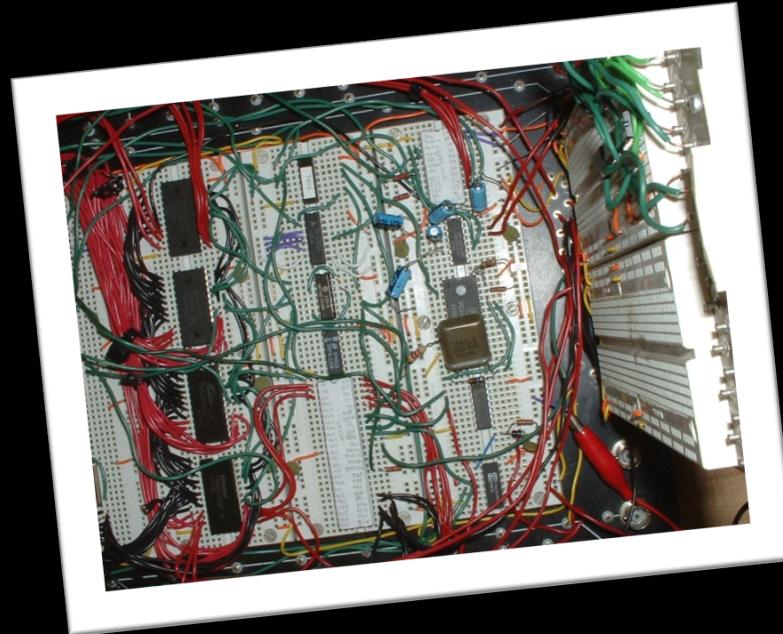
$$Y = (A \text{ or } B) \text{ and } (\text{not } A \text{ or } \text{not } B) \text{ or } C$$

- Like so:



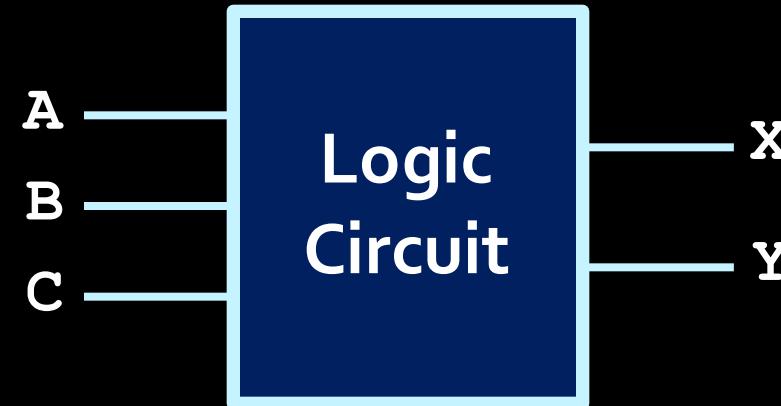
Creating complex circuits

- What do we do in the case of more complex circuits, with several inputs and more than one output?
 - If you're lucky, a **truth table** is provided to express the circuit.
 - Usually the behaviour of the circuit is expressed in words, and the first step involves creating a truth table that represents the described behaviour.



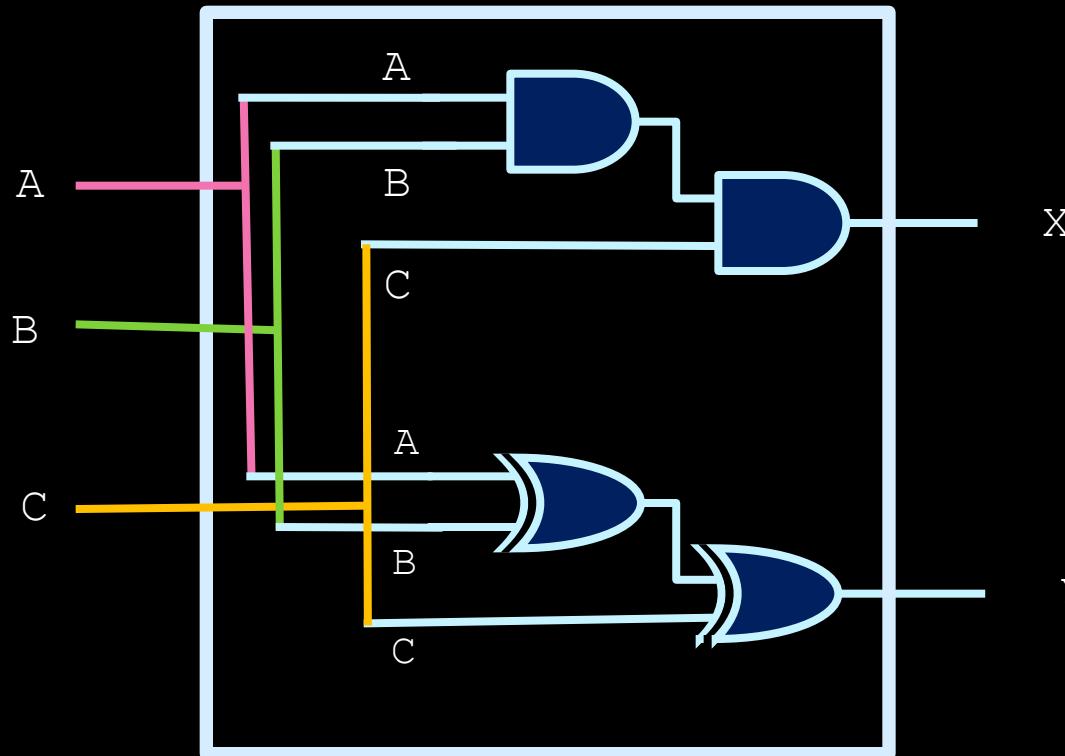
Circuit example

- The circuit on the right has three inputs (A, B and C) and two outputs (X and Y).
- What logic is needed to set X high when all three inputs are high?
- What logic is needed to set Y high when the number of high inputs is odd?



Combinational circuits

- Small problems can be solved easily.



X high when all
three inputs are
high

Y high when
number of high
is odd

For more complicated circuits,
we need a systematical approach

Creating complex logic

- The general approach
- Basic steps:
 1. Create **truth tables** based on the desired behaviour of the circuit.
 2. Come up with a “**good**” **Boolean expression** that has exactly that truth table.
 3. Convert Boolean expression to **gates**.
- The key to an efficient design?
 - Spending extra time on **Step #2**.

First,
a better way to represent
truth tables

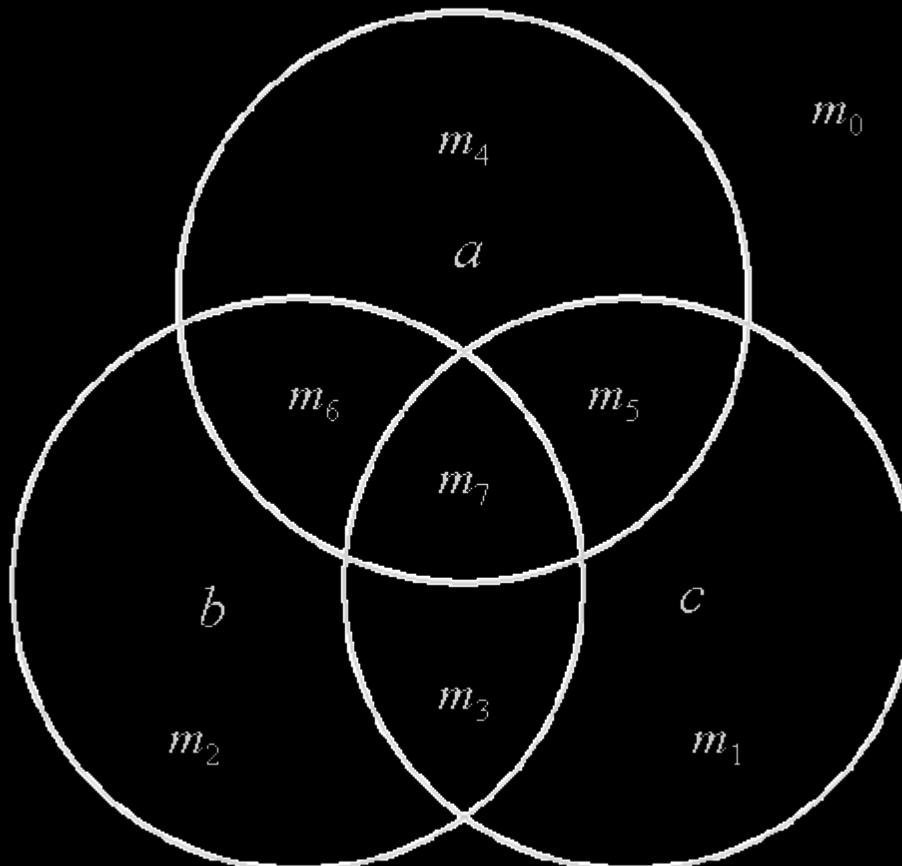
Example truth table

- Consider the following example:
 - "*Y is high only when B and C are both high*"
- This leads to the truth table on the right.
 - Do we always have to draw the whole table?
 - Is there a better way to describe the truth table?

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

This is all we needed to express!

Yes, use “Minterms” and “Maxterms”



Minterms, informally

- First, sort the rows according to the value of the number “ABC” represents
- Then for each row, we name the inputs as $m_{\{row\ number\}}$
- m_0, m_1, m_2, \dots , are called **minterms**

The diagram illustrates the conversion between a truth table and a minterm table. On the left, a green downward-pointing arrow labeled "Sorted" points to a truth table with columns A, B, C and row Y. The rows represent binary values from 000 to 111. On the right, a double-headed green horizontal arrow points between the truth table and a minterm table. The minterm table has columns "Minterm" and "Y". It lists eight minterms: $m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7$, corresponding to the rows of the truth table.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Minterm	Y
m_0	0
m_1	1
m_2	1
m_3	1
m_4	1
m_5	0
m_6	1
m_7	0

Minterm, a more formal description

Minterm: an AND expression with every input present in true or complemented form.

$$m_0: \overline{A} \cdot \overline{B} \cdot \overline{C}$$

$$m_1: \overline{A} \cdot \overline{B} \cdot C$$

$$m_2: \overline{A} \cdot B \cdot \overline{C}$$

$$m_3: \overline{A} \cdot B \cdot C$$

$$m_7: A \cdot B \cdot C$$

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Minterm	Y
m_0	0
m_1	1
m_2	1
m_3	1
m_4	1
m_5	0
m_6	1
m_7	0

Minterm (m) and Maxterm (M)

Minterm: an **AND** expression with **every** input present in **true or complemented** form.

Maxterm: an **OR** expression with **every** input present in **true or complemented** form.

$$M_0: A+B+C$$

$$M_1: A+B+\bar{C}$$

$$M_6: \bar{A}+\bar{B}+C$$

$$M_7: \bar{A}+\bar{B}+\bar{C}$$

Feel something fishy?

Naming!

m_0 is $\overline{A} \cdot \overline{B} \cdot \overline{C}$

Minterm is about
when the output is 1

$\overline{A} \cdot \overline{B} \cdot \overline{C}$ is **1** only when A, B, C are 0, 0, 0

M_0 is $A+B+C$

$A+B+C$ is **0** only when A, B, C are 0, 0, 0

Maxterm is about
when the output is 0

Exercise: Minterm or Maxterm?

given four inputs: (A, B, C, D)

$$A \cdot B \cdot C$$

Neither! **Every** input needs to be there, D is missing!

$$A+B+D$$

Neither! Same reason

$$A \cdot B + C \cdot \bar{D}$$

Neither! It has to be **only AND** or **only OR**, cannot mix

$$\bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \bar{D}$$

Maxterm, M_5

$$A \cdot \bar{B} \cdot C \cdot \bar{D}$$

Minterm, M_{10}

Quick fact

- Given n inputs, how many possible minterms and maxterms are there?
 - 2^n minterms and 2^n maxterms possible (same as the number of rows in a truth table).

Quick note about notations

- AND operations are denoted in these expressions by the multiplication symbol.
 - e.g. $A \cdot B \cdot C$ or $A^*B^*C \approx A \wedge B \wedge C$
- OR operations are denoted by the addition symbol.
 - e.g. $A+B+C \approx A \vee B \vee C$
- NOT is denoted by multiple symbols.
 - e.g. $\neg A$ or A' or \bar{A}
- XOR occurs rarely in circuit expressions.
 - e.g. $A \oplus B$

Use minterms and maxterms
to go from truth table to logic expression

Using minterms

- What are minterms used for?
 - A single minterm indicates a set of inputs that will make the output go high.
 - Example: Describe the truth table on the right using minterm:
 - $m_2 \quad A'B'CD'$

A	B	C	D	m_2
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Using minterms

- What happens when you OR two minterms?
 - Result is output that goes high in both minterm cases.
 - Describe the truth table with the right-most column of outputs
 - $m_2 + m_8$

A	B	C	D	m_2	m_8	$m_2 + m_8$
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	1	0	1
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	1	1
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	0	1	1	0	0	0
1	1	0	0	0	0	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

We came up with a logic expression that has the desired truth table, easily: $A'B'CD' + AB'C'D'$

Creating boolean expressions

- Two canonical forms of boolean expressions:
 - Sum-of-Minterms (SOM): $AB + A'B + AB'$
 - Each minterm corresponds to a single high output in the truth table.
 - Also known as: Sum-of-Products.
 - Product-of-Maxterms (POM): $(A+B)(A' + B)(A+B')$
 - Each maxterm corresponds to a single low output in the truth table.
 - Also known as Product-of-Sums.

Every logic expression can be converted to a SOM, also to a POM.

$$Y = m_2 + m_6 + m_7 + m_{10} \text{ (SOM)}$$

A	B	C	D	m_2	m_6	m_7	m_{10}	Y
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	1	0	0	0	1
0	0	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0	1
0	1	1	1	0	0	1	0	1
1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0

$$Y = M_3 \cdot M_5 \cdot M_7 \cdot M_{10} \cdot M_{14} \text{ (POM)}$$

A	B	C	D	M ₃	M ₅	M ₇	M ₁₀	M ₁₄	Y
0	0	0	0	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	1	0	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	0
0	1	1	0	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	0
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1
1	0	1	0	1	1	1	0	1	0
1	0	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1

Sum-of-Minterms vs Product-of-Maxterm

- SOM expresses which inputs cause the output to go high.
- POM expresses which inputs cause the output to go low
- SOMs are useful in cases with very few input combinations that produce high output.
- POMs are useful when expressing truth tables that have very few low output cases...

What if we do this using POM?

$$Y = m_2 + m_6 + m_7 + m_{10} \text{ (SOM)}$$

A	B	C	D	m_2	m_6	m_7	m_{10}	Y
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	1	0	0	0	1
0	0	1	1	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0
0	1	1	0	0	1	0	0	1
0	1	1	1	0	0	1	0	1
1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	1	0	0	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0

What if we do this using SOM?

$$Y = M_3 \cdot M_5 \cdot M_7 \cdot M_{10} \cdot M_{14} \text{ (POM)}$$

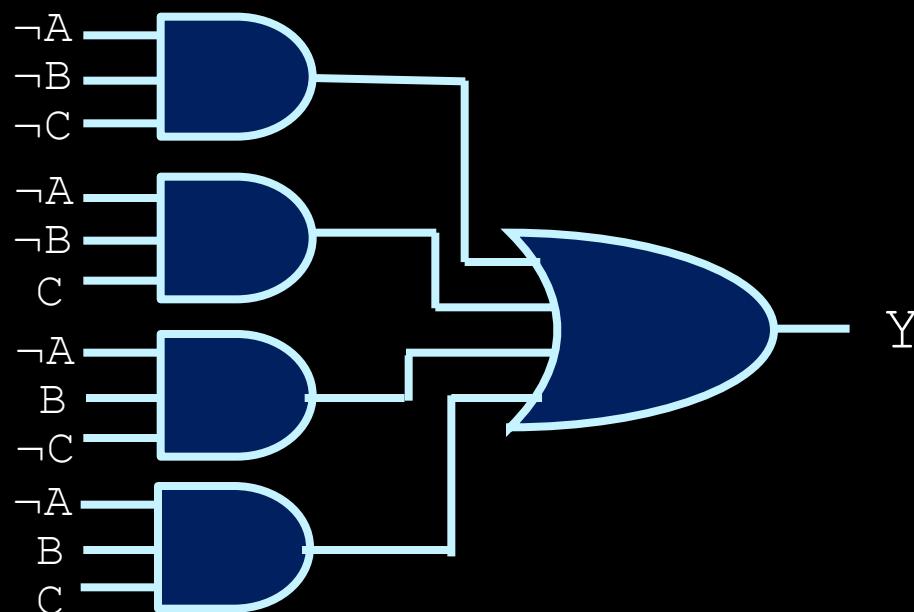
A	B	C	D	M ₃	M ₅	M ₇	M ₁₀	M ₁₄	Y
0	0	0	0	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	1	0	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	0
0	1	1	0	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	0
1	0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1
1	0	1	0	1	1	1	0	1	0
1	0	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1

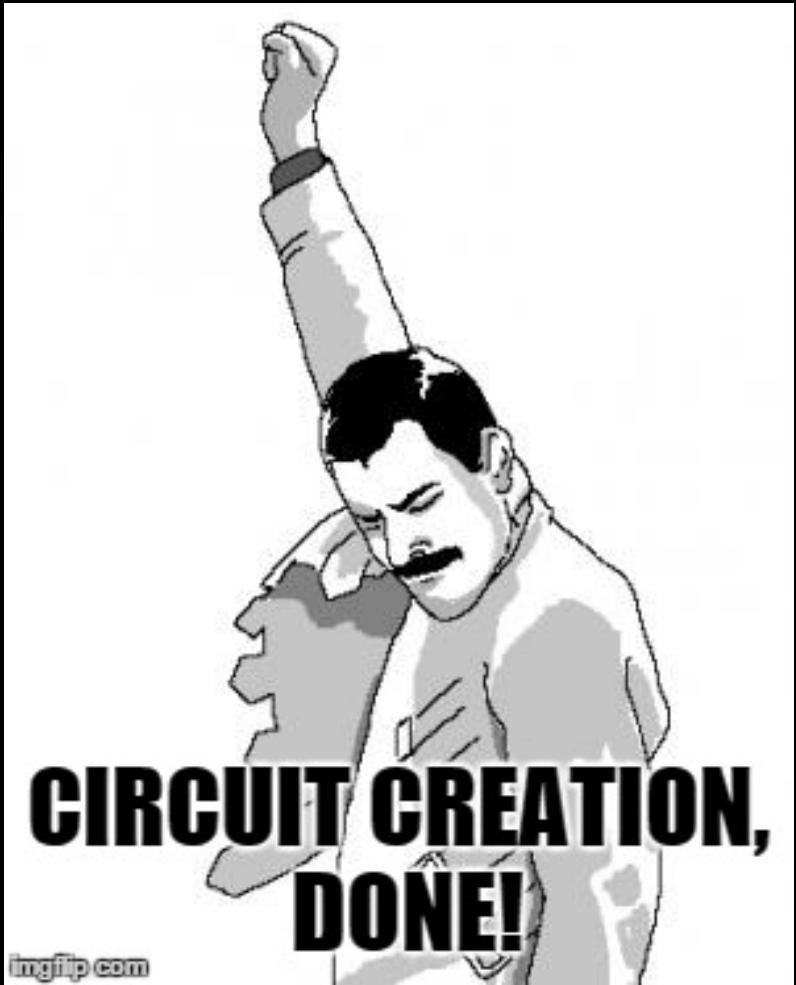
Converting SOM to gates

- Once you have a Sum-of-Minterms expression, it is easy to convert this to the equivalent combination of gates:

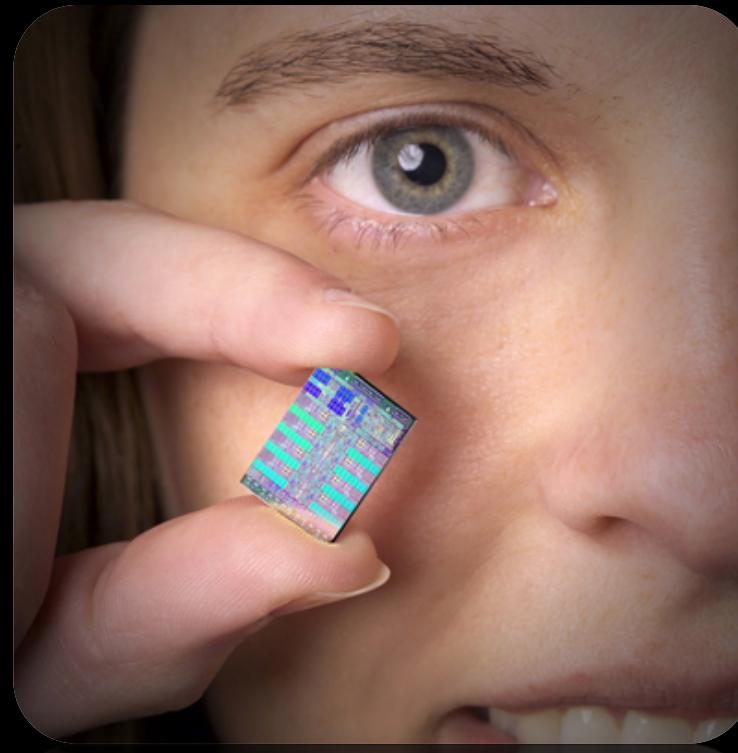
$$m_0 + m_1 + m_2 + m_3 =$$

$$\bar{A} \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + \\ \bar{A} \cdot B \cdot \bar{C} + \bar{A} \cdot B \cdot C$$

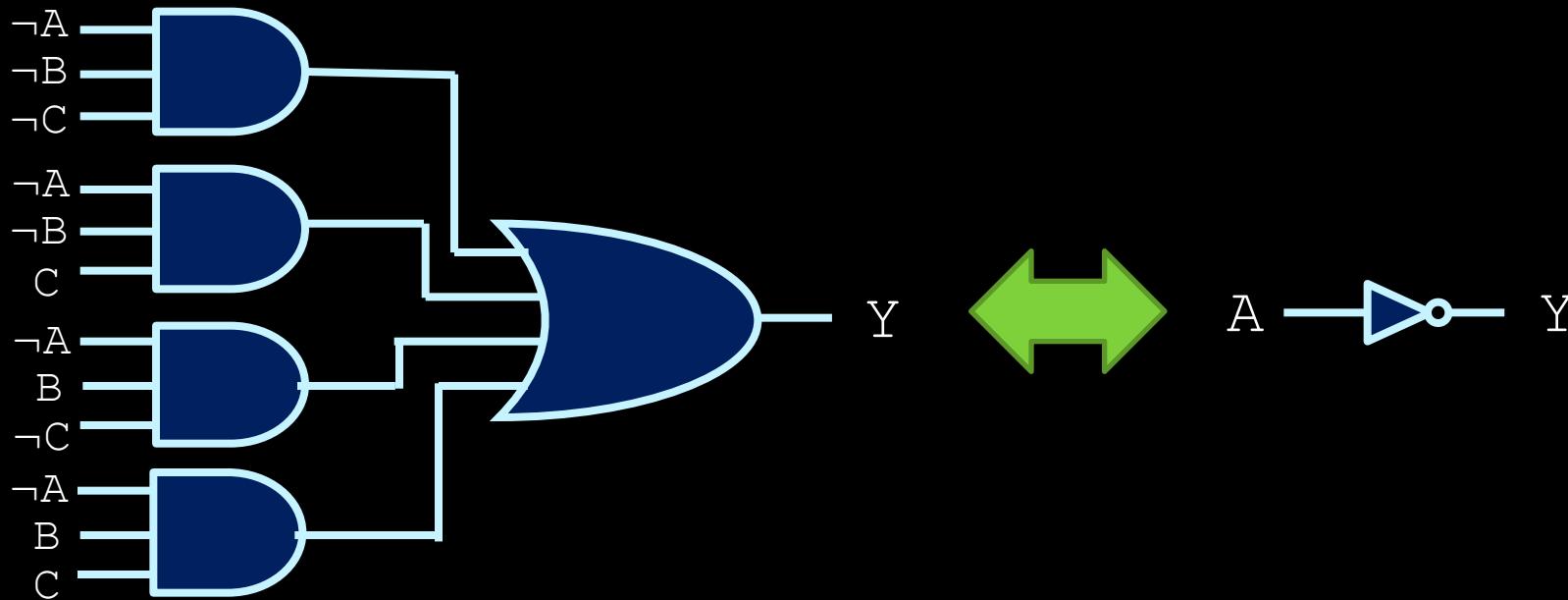




Reducing circuits



Reasons for reducing circuits



- To minimize the number of gates, we want to reduce the Boolean expression as much as possible from a collection of minterms to something smaller.
- This is where math skills come in handy ☺

Boolean algebra review

- Axioms:

$$\begin{array}{ll} 0 \cdot 0 = 0 & 0 \cdot 1 = 1 \cdot 0 = 0 \\ 1 \cdot 1 = 1 & \text{if } x = 1, \overline{x} = 0 \end{array}$$

- From this, we can extrapolate:

$$\begin{array}{ll} x \cdot 0 = 0 & x+1 = 1 \\ x \cdot 1 = x & x+0 = x \\ x \cdot x = x & x+x = x \\ x \cdot \overline{x} = 0 & x+\overline{x} = 1 \\ \overline{\overline{x}} = x & \end{array}$$

Other boolean identities

- Commutative Law:

$$x \cdot y = y \cdot x$$

$$x+y = y+x$$

- Associative Law:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

$$x + (y+z) = (x+y) + z$$

- Distributive Law:

$$x \cdot (y+z) = x \cdot y + x \cdot z$$

$$x + (y \cdot z) = (x+y) \cdot (x+z)$$

Other boolean identities

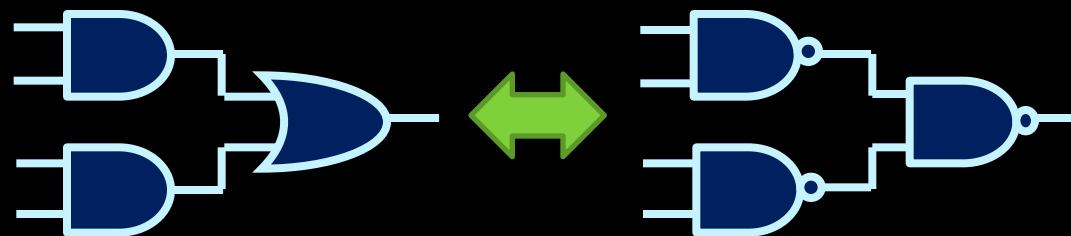
- De Morgan's Laws:

$$\begin{aligned}\overline{x \cdot y} &= \overline{x+y} \\ \overline{x+y} &= \overline{x \cdot y}\end{aligned}$$

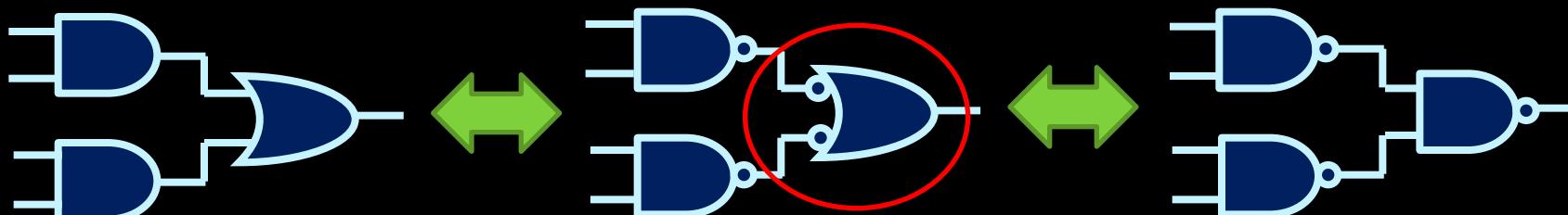


De Morgan and NAND gates

- De Morgan's Law is important because out of all the gates, NANDs are the cheapest to fabricate.
 - a Sum-of-Products circuit could be converted into an equivalent circuit of NAND gates:



- This is all based on de Morgan's Law:



Reducing boolean expressions

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- Assuming logic specs at left, we get the following:

$$m_3 + m_4 + m_6 + m_7$$

$$Y = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$$

Warming up...

$$A \cdot B + A \cdot \bar{B} = A$$

Reduce by combining two terms that differ by a single literal.

Let's reduce this

$$Y = \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C} + A \cdot B \cdot C$$

Combine the last two terms...

$$Y = \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot B$$

Combine the middle two and the end two ...

$$Y = B \cdot C + A \cdot \overline{C}$$

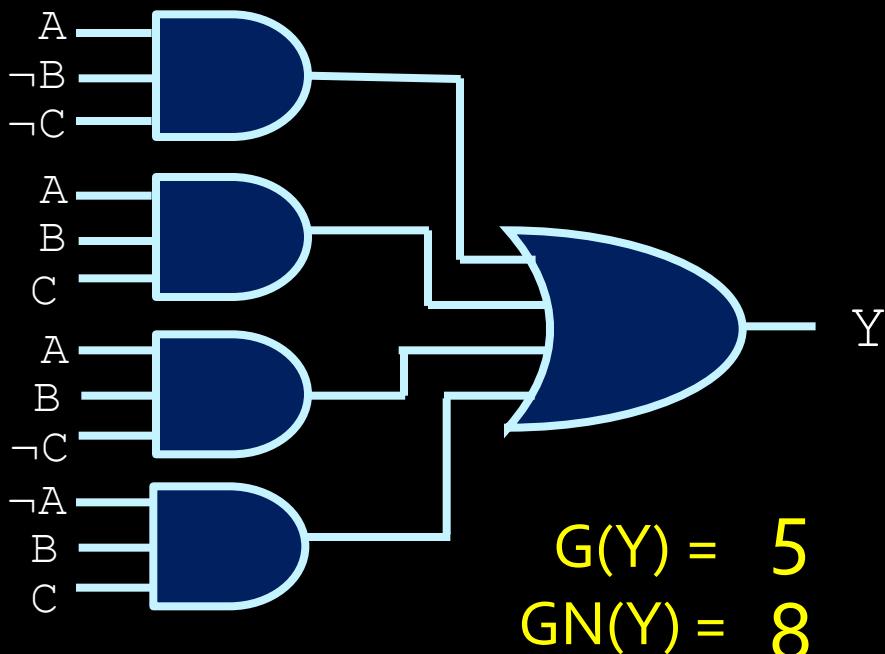
There could be different ways of combining,
some are **simpler** than others.

How to get to the simplest expression?

Wait ... What does “simplest” mean?

What is “simplest”?

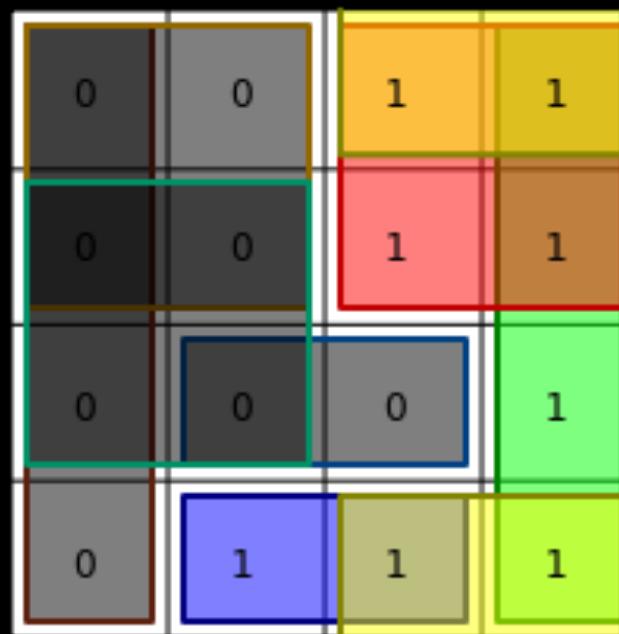
- In this case, “simple” denotes the lowest gate cost (G) or the lowest gate cost with NOTs (GN).
- To calculate the gate cost, simply add all the gates together (as well as the cost of the NOT gates, in the case of the GN cost).



Don't count $\neg C$ twice!

Karnaugh maps

Find the simplest expression, systematically.



Reducing boolean expressions

- How do we find the “simplest” expression for a circuit?
 - Technique called **Karnaugh maps** (or K-maps).
 - Karnaugh maps are a 2D grid of minterms, where adjacent minterm locations in the grid **differ by a single literal**.
 - Values of the grid are the output for that minterm.

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1	1

Compare these...

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1	1

$$Y = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$$

Karnaugh maps

- Karnaugh maps can be of any size, and have any number of inputs.
- Since adjacent minterms only differ by a single literal, they can be combined into a single term that omits that value.

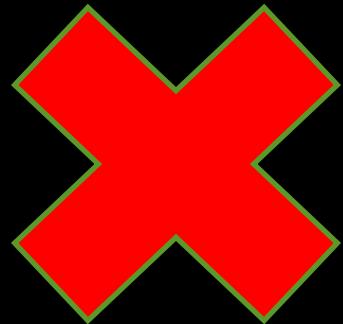
	$\bar{C} \cdot \bar{D}$	$\bar{C} \cdot D$	$C \cdot \bar{D}$	$C \cdot D$
$\bar{A} \cdot \bar{B}$	m_0	m_1	m_3	m_2
$\bar{A} \cdot B$	m_4	m_5	m_7	m_6
$A \cdot \bar{B}$	m_{12}	m_{13}	m_{15}	m_{14}
$A \cdot B$	m_8	m_9	m_{11}	m_{10}

Using Karnaugh maps

- Once Karnaugh maps are created, draw boxes over **groups of high output values**.
 - Boxes must be rectangular, and aligned with map.
 - Number of values contained within each box must be a power of 2.
 - Boxes may overlap with each other.
 - Boxes may wrap across edges of map.

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1 1	1

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	1	1	0
A	0	0	1	0



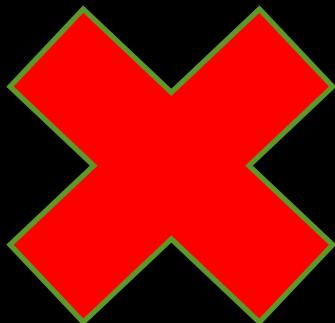
Must be rectangle!

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	1	1	0
A	0	0	1	0



Two boxes
overlapping each
other is fine.

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	1	1	1
A	0	0	0	0



Number of value contained must be power of 2.

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	1	1	1
A	0	0	0	0

1 is a power of 2

$$1 = 2^0$$

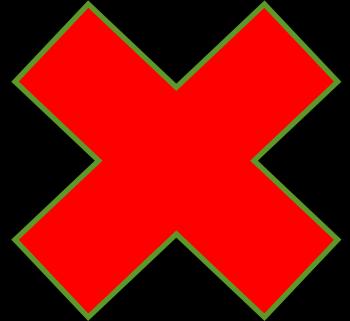


	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	1	1	0
A	0	1	1	0

Rectangle, with
power of 2 entries



	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	1	0	0
A	0	0	1	0



Must be aligned
with map.

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	0	0
A	1	0	0	1

Wrapping across
edge is fine.



So... how to find smallest expression

Minterms in one box can be combined
into one term

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	0	0	1	0

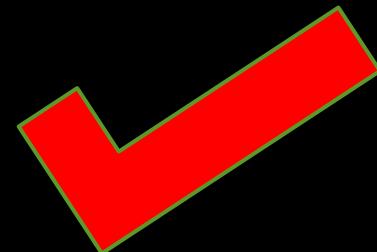
$$\bar{A} \cdot B \cdot C + A \cdot B \cdot C = B \cdot C$$

So... how to find smallest expression

The simplest expression corresponds to the smallest number of **boxes** that cover all the high values (1's).

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1	1

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1	1



$$Y = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot B \cdot \bar{C} + A \cdot B \cdot C$$

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1	1

$$Y = \bar{A} \cdot B \cdot C + A \cdot \bar{B} \cdot \bar{C} + A \cdot B$$

	$\bar{B} \cdot \bar{C}$	$\bar{B} \cdot C$	$B \cdot C$	$B \cdot \bar{C}$
\bar{A}	0	0	1	0
A	1	0	1	1

$$Y = B \cdot C + A \cdot \bar{C}$$

K-map: the steps

Given a complicated expression

1. Convert it to Sum-Of-Minterms
2. Draw the 2D grid
3. Mark all the high values (1's), according to which minterms are in the SOM.
4. Draw boxes that cover 1's.
5. Find the smallest set of boxes that cover all 1's.
6. Write out the simplified result according to the boxes found.

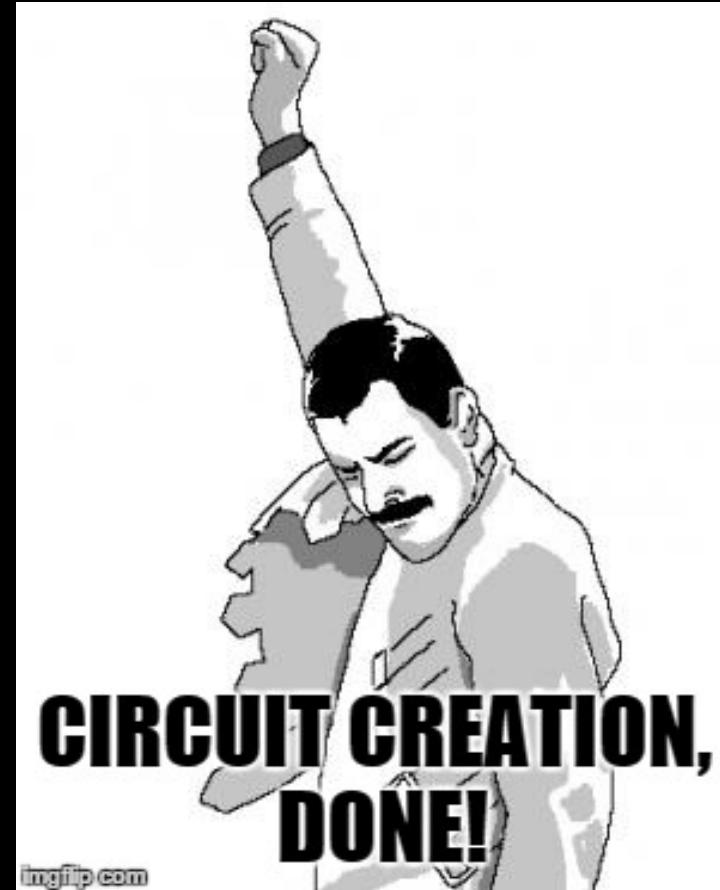
Everything can be done using Maxterms, too

- Can also use this technique to group maxterms together as well.
- Karnaugh maps with maxterms involves grouping the **zero** entries together, instead of grouping the entries with one values.

	$C+D$	$C+\bar{D}$	$\bar{C}+\bar{D}$	$\bar{C}+D$
$A+B$	M_0	M_1	M_3	M_2
$A+\bar{B}$	M_4	M_5	M_7	M_6
$\bar{A}+\bar{B}$	M_{12}	M_{13}	M_{15}	M_{14}
$\bar{A}+B$	M_8	M_9	M_{11}	M_{10}

Circuit creation – the whole flow

1. Understand desired behaviour
2. Write the truth table based on the behaviour
3. Write the SOM (or POM) of that truth table
4. Simplify the SOM using K-map
5. Translate the simplified logic expression into circuit with gates.



Today we learned

- How to create a logic circuit from scratch, given a desired digital behaviour.
- Minterm & Maxterm
- K-Map use to reduce the circuit

Next Week:

- Logical Devices

Quiz Time!

Question 1

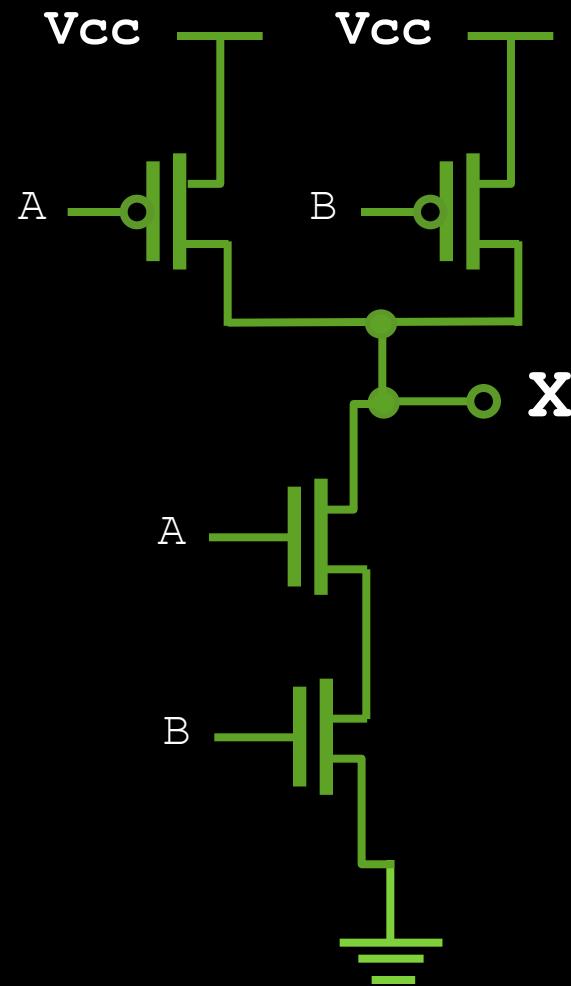
After doping a piece of silicon with n-type impurity (phosphorus), the overall electric charge of the material is

- A. Positive
- B. Negative
- C. **Neutral**
- D. None of above

Question 2

- What gate is this?
- A and B are inputs, X is output.

NAND



Question 3

- Write the following truth table as Sum-of-Minterms.

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Write in this form
 $A'B'C + A'BC'$

Flip your quiz face-down and pass it to your right.

CSC258 Winter 2016

Lecture 3

Announcements

- Check your lab marks and quiz marks on MarkUs
- If you cannot login, email me your UTORID and name
- About borrowing DE2 boards outside the labs: we are a bit short on the number of boards available, so we don't really have spare boards to be circulated outside the labs. So to make things simple and fair, no board borrowing outside the lab.
- If you attend the same lab multiple times, your lowest mark will be recorded.

Feedback 1

About the quiz: “Putting in questions which has to do with the current lecture is a bad move, ..., it favours people who can understand the lecture on the dot and/or are in the right mood for it, which is unfair for people who would put in the work after the lecture...”

- Valid point, so from this week on I will only quiz on content from previous weeks.

Feedback 2

“Could you post the slides before the lecture?”

- I probably won't do this because
 - The slides are typically not ready until a very short time before the lecture. I'm reluctant to post something that is partially done.
 - Some parts of the slides are supposed to “emotional moments” which would make you remember certain things better. Seeing the slides before would spoil it.
 - So I estimated that the loss is bigger than the gain if I post them beforehand.

Feedback 3

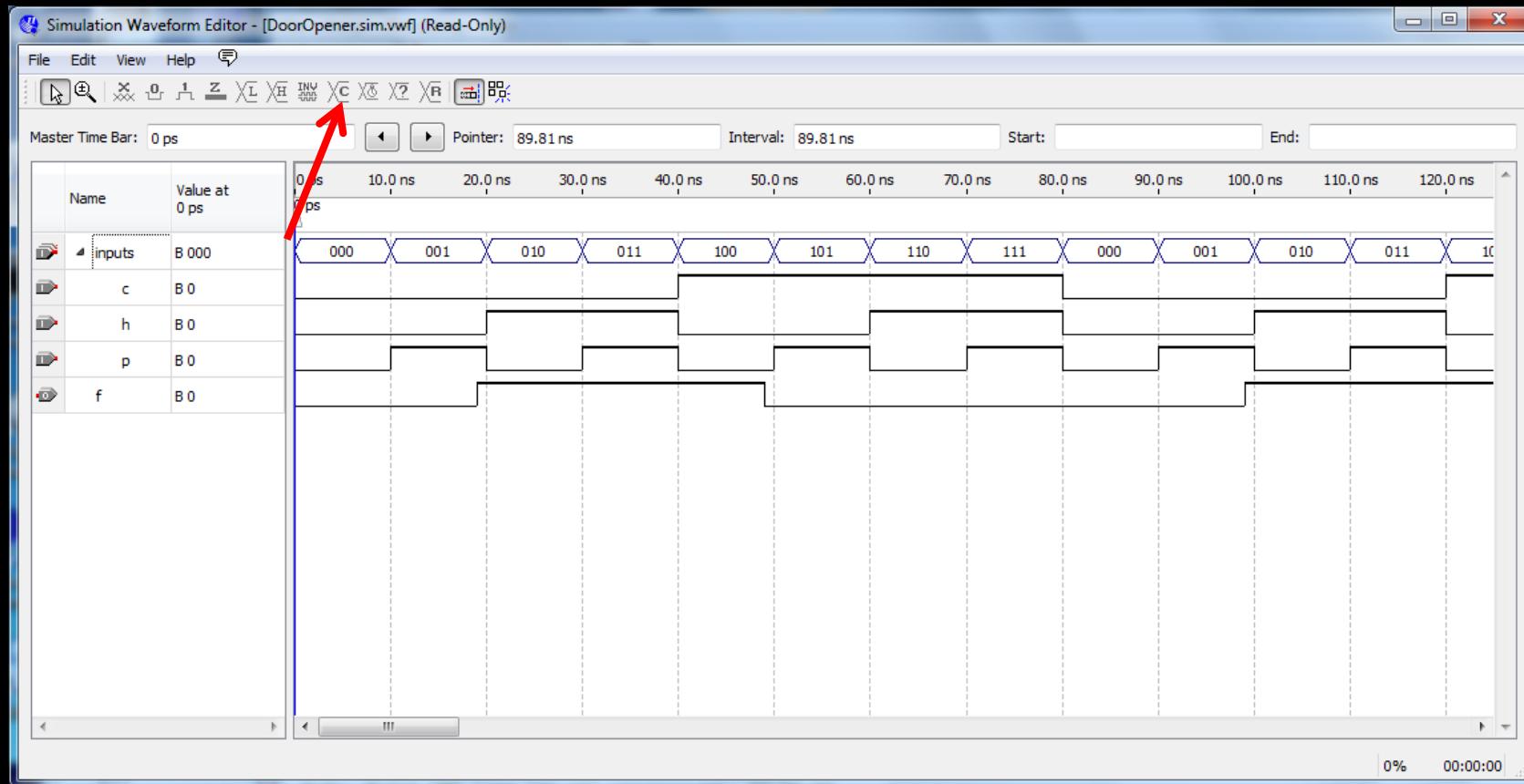
“Go slower!”

“I really like the pace of the lecture!”

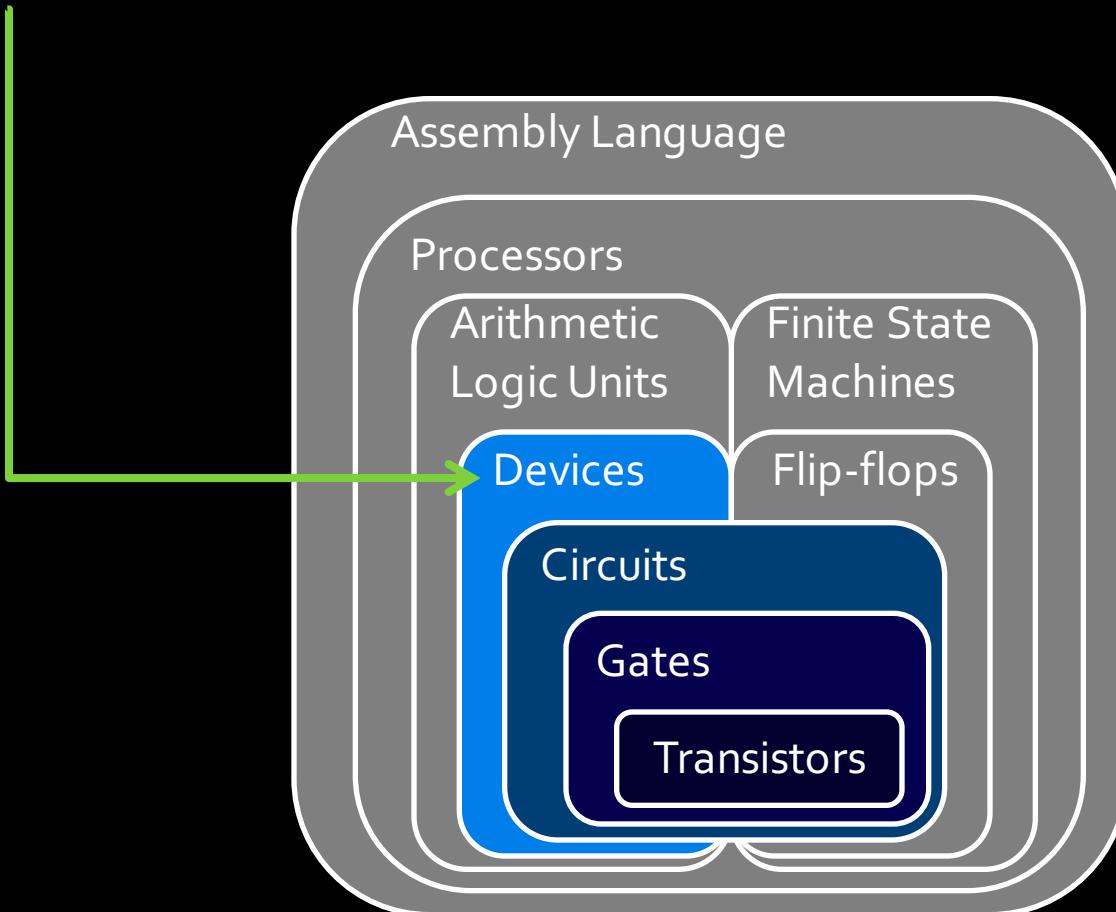
- I'll try to keep it slow given that I can still teach all the important concepts.
- I need to hear more from you to correctly understand whether it is too fast or too slow.

A Quartus Trick

In the waveform editor, use the “**counter**” feature to generate all possible input combinations, i.e., for grouped inputs ABC, generate 0, 1, 2, 3, ..., 7, which is basically 000, 001, 010, 011, ..., 111, all possible combinations.



We are here

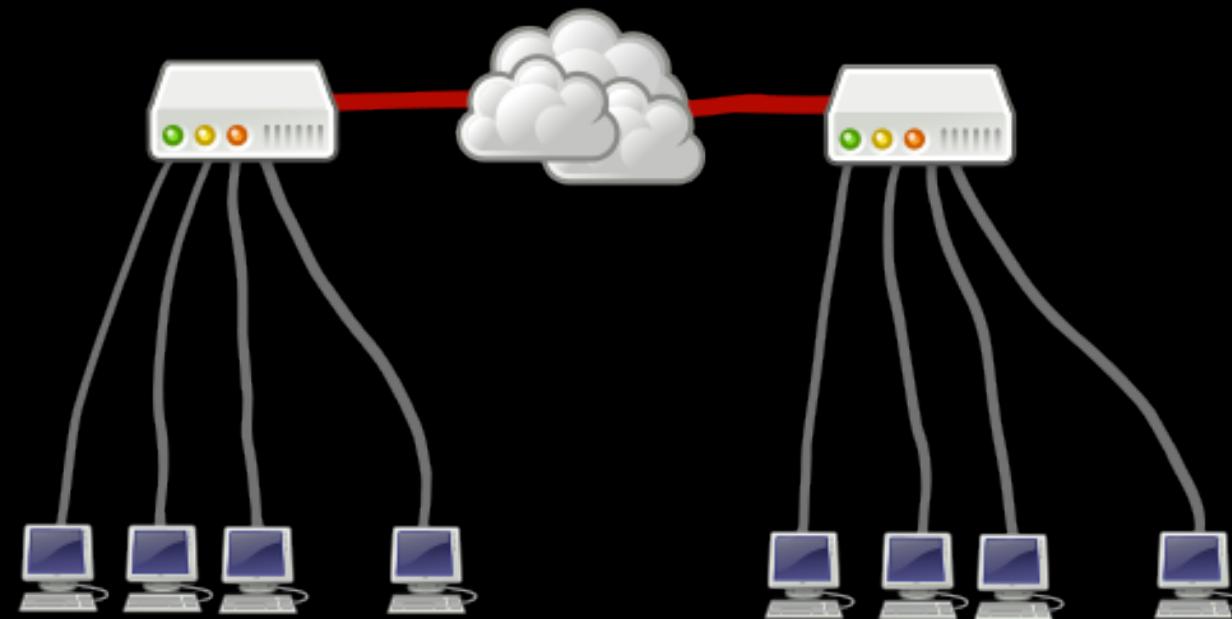


Logical Devices

Building up from gates...

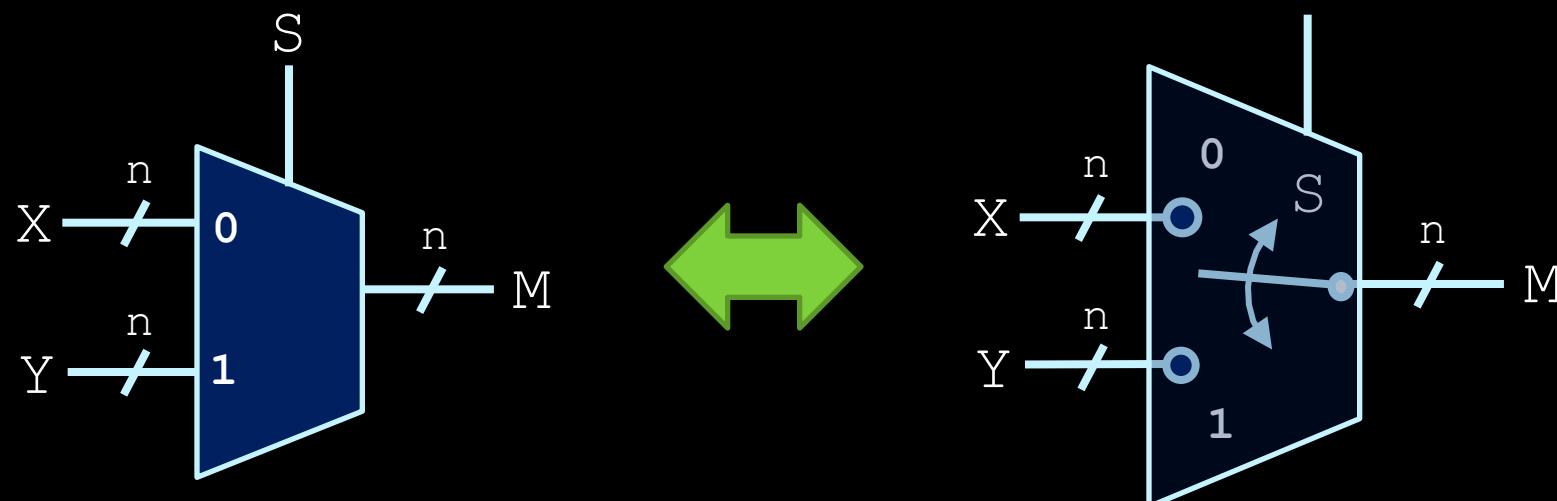
- Some common and more complex structures:
 - Multiplexers (MUX)
 - Adders (half and full)
 - Subtractors
 - Decoders
 - Seven-segment decoders
 - Comparators

Multiplexers



Logical devices

- Certain structures are common to many circuits, and have block elements of their own.
 - e.g. Multiplexers (short form: **mux**)
 - Behaviour: Output is X if S is 0, and Y if S is 1, i.e., S selects which input can go through

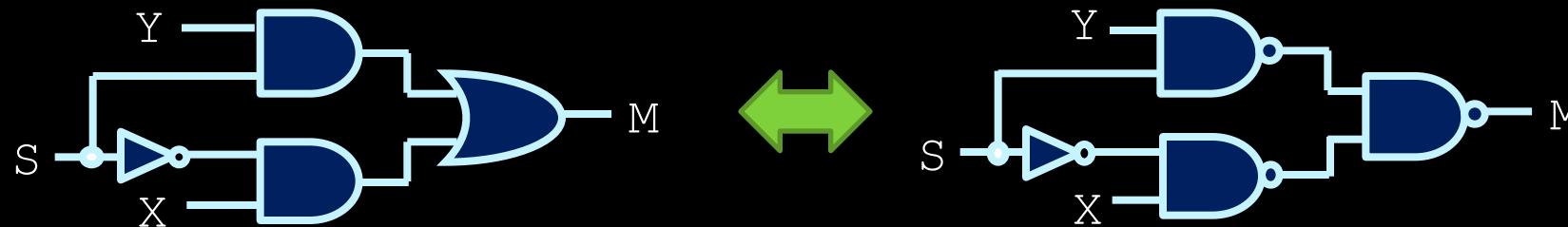


Multiplexer design

x	y	s	m
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

	$\bar{Y} \cdot \bar{S}$	$\bar{Y} \cdot S$	$Y \cdot S$	$Y \cdot \bar{S}$
\bar{x}	0	0	1	0
x	1	0	1	1

$$M = Y \cdot S + X \cdot \bar{S}$$



Multiplexer uses

- Muxes are very useful whenever you need to select from multiple input values.
 - Example:
 - Surveillance video monitors,
 - Digital cable boxes,
 - routers.



Adder circuits



Adders

- Also known as binary adders.
 - Small circuit devices that add two **1-bit** number.
 - Combined together to create **iterative combinational circuits** – add **multiple-bit** numbers
- Types of adders:
 - Half adders (HA)
 - Full adders (FA)
 - Ripple Carry Adder
 - Carry-Look-Ahead Adder (CLA)

Review of Binary Math

Review of Binary Math

- Each digit of a decimal number represents a power of 10:

$$258 = 2 \times 10^2 + 5 \times 10^1 + 8 \times 10^0$$

- Each digit of a binary number represents a power of 2:

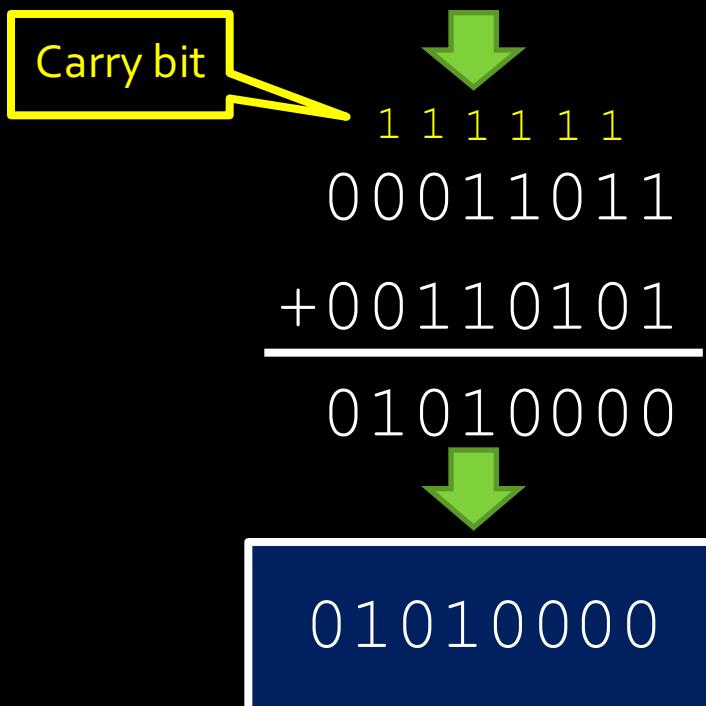
$$\begin{aligned}01101_2 &= 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\&= 13_{10}\end{aligned}$$

Unsigned binary addition

- $27 + 53$

$$27 = 00011011$$

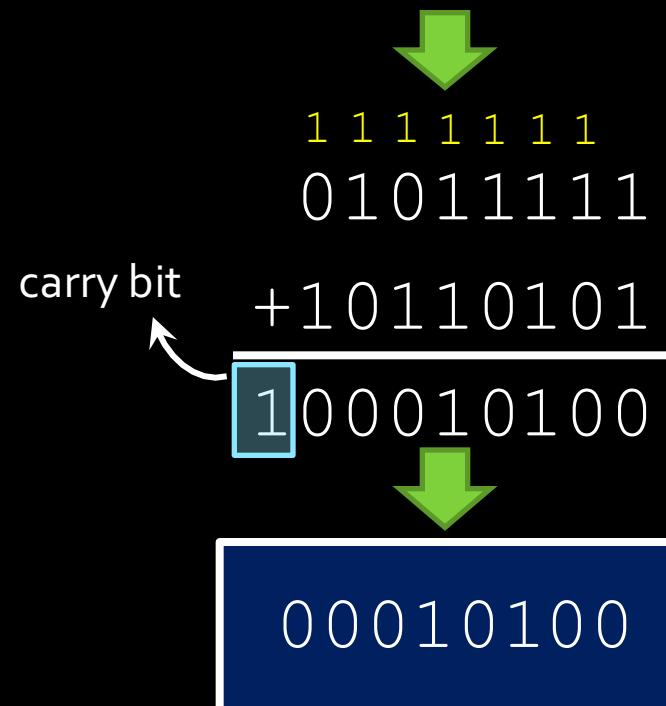
$$53 = 00110101$$



- $95 + 181$

$$01011111$$

$$+10110101$$



Half Adder

Input: two 1-bit numbers

Output: 1-bit sum and 1-bit carry

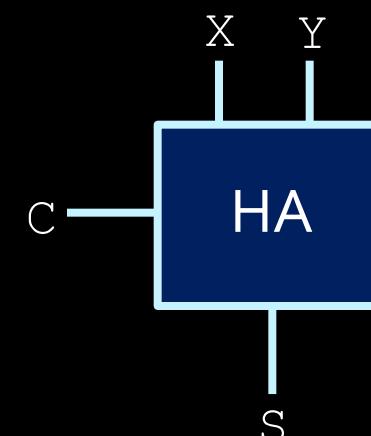
Half Adders

- A 2-input, 1-bit width binary adder that performs the following computations:

X	0	0	1	1
+ Y	+0	+1	+0	+1
CS	00	01	01	10

$C = X?Y$
 $S = X?Y$

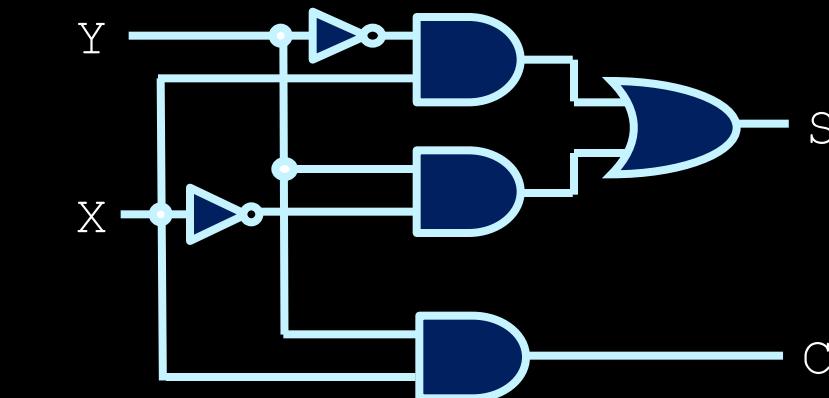
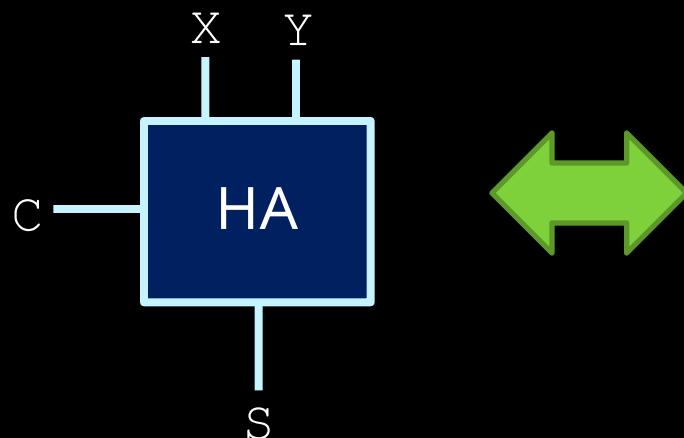
- A half adder adds two bits to produce a two-bit sum.
- The sum is expressed as a sum bit S and a carry bit C .



Half Adder Implementation

- Equations and circuits for half adder units are easy to define (even without Karnaugh maps)

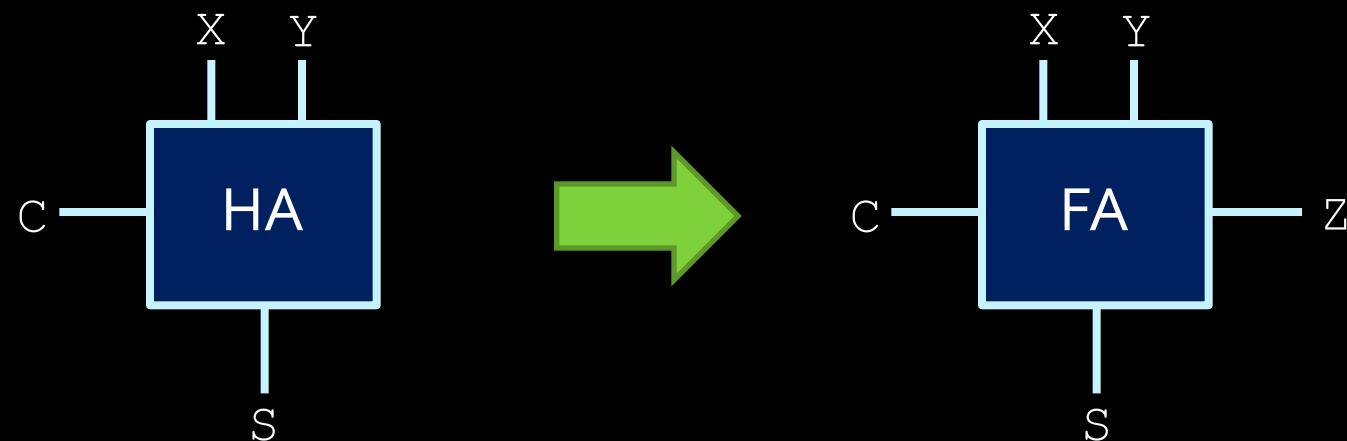
$$\begin{aligned} C &= X \cdot Y \\ S &= X \cdot \bar{Y} + \bar{X} \cdot Y \\ &= X \oplus Y \end{aligned}$$



A half adder **outputs** a carry-bit,
but does not take a carry-bit as **input**.

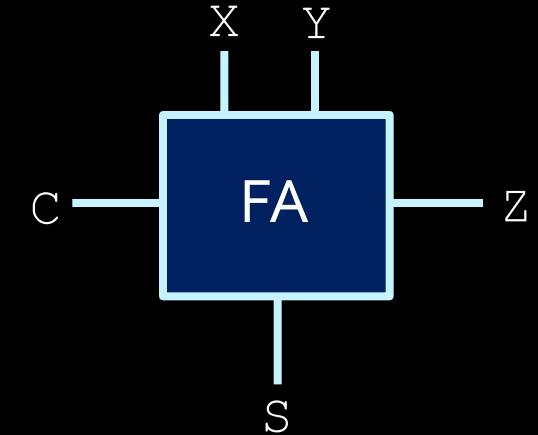
Full Adder

takes a carry bit as **input**



Full Adders

- Similar to half-adders, but with another input **Z**, which represents a **carry-in bit**.
 - C and Z are sometimes labeled as C_{out} and C_{in} .
- When Z is 0, the unit behaves exactly like...
 - a half adder.
- When Z is 1:



X	0	0	1	1
+Y	+0	+1	+0	+1
+Z	+1	+1	+1	+1
CS	01	10	10	11

Full Adder Design

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

c	$\bar{y} \cdot \bar{z}$	$\bar{y} \cdot z$	$y \cdot z$	$y \cdot \bar{z}$
\bar{x}	0	0	1	0
x	0	1	1	1

s	$\bar{y} \cdot \bar{z}$	$\bar{y} \cdot z$	$y \cdot z$	$y \cdot \bar{z}$
\bar{x}	0	1	0	1
x	1	0	1	0

$$C = X \cdot Y + X \cdot Z + Y \cdot Z$$

$$C = X \cdot Y + (X \oplus Y) \cdot Z$$

$$S = X \oplus Y \oplus Z$$

For gate reuse ($X \oplus Y$)
considering both C and S

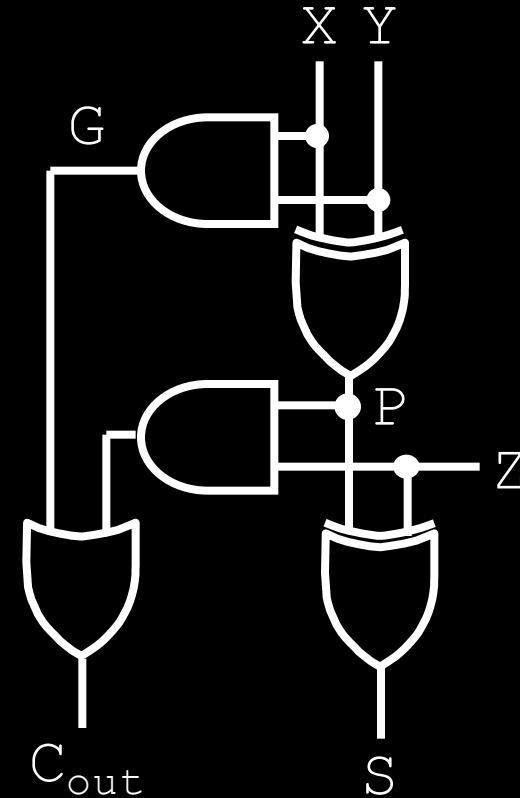
Full Adder Design

$$S = X \oplus Y \oplus Z$$

- The C term can also be rewritten as:

$$C = X \cdot Y + (X \oplus Y) \cdot Z$$

- Two terms come from this:
 - $X \cdot Y$ = carry generate (G).
 - Whether X and Y generate a carry bit
 - $X \oplus Y$ = carry propagate (P).
 - Whether Z will be propagated to Cout
- Results in this circuit →

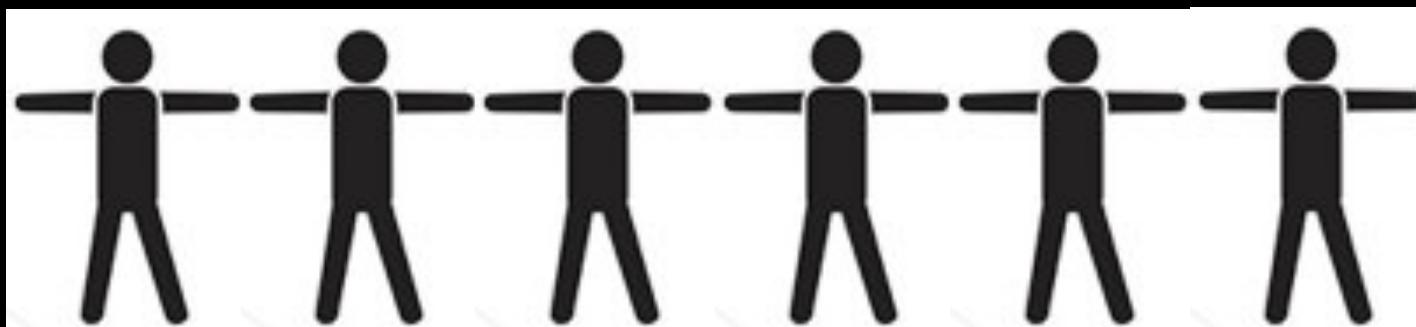
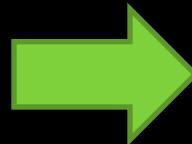
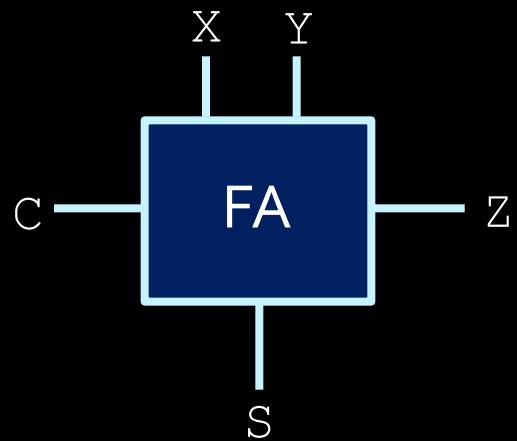
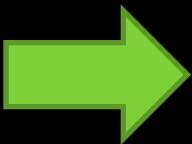
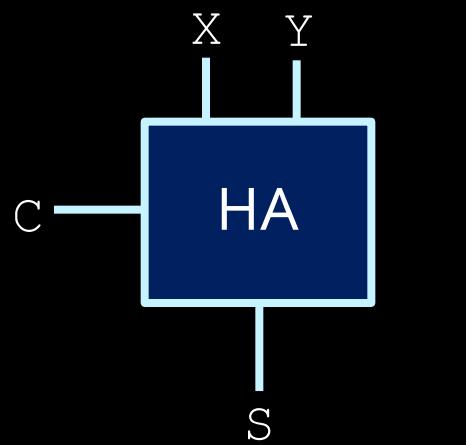


Now we can add one bit properly, but most of the numbers we use have more than one bits.

- int, unsigned int: 32 bits (architecture-dependent)
- short int, unsigned short int: 16 bits
- long long int, unsigned long long int: 64 bit
- char, unsigned char: 8 bits



How do we add multiple-bit numbers?



Each full adder takes in a carry bit and outputs a carry bit.

Each full adder can take in a carry bit which is output by another full adder.

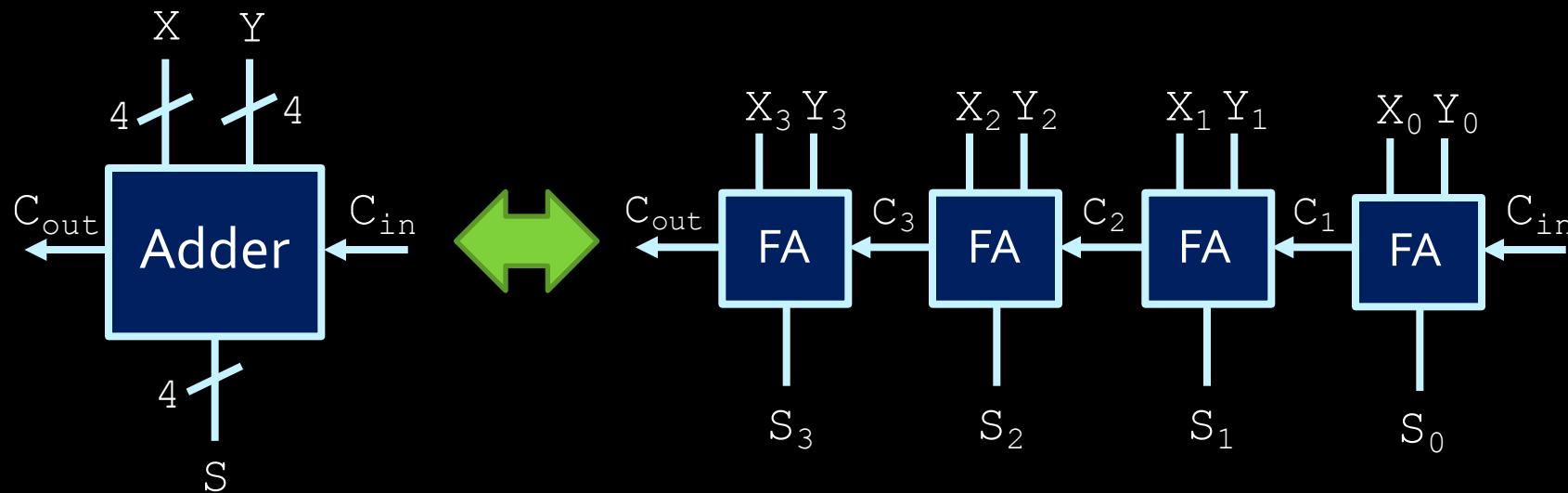
That is, they can be **chained** up.

Ripple-Carry Binary Adder

Full adders chained up,
for **multiple-bit** addition

Ripple-Carry Binary Adder

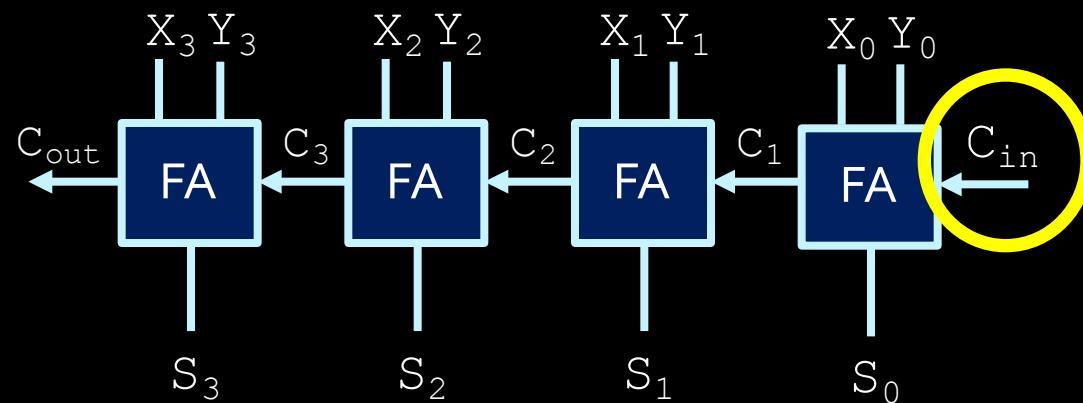
- Full adder units are chained together in order to perform operations on signal **vectors**.



$S_3S_2S_1S_0$ is the sum of $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$

The role of C_{in}

- Why can't we just have a half-adder for the smallest (right-most) bit?
- Because if we can use it to do **subtraction!**



Let's play a game...

- 
1. Find a partner
 2. Each of you pick a number between 0 and 31
 3. Convert both numbers to binary
 4. Invert each digit of the smaller number
 5. Add up the big binary number and the inverted small binary number
 6. Add 1 to the result, keep the lowest 5 digits
 7. Convert the result to a decimal number

What do you get?

You just did subtraction
without doing subtraction!

Subtractors

- Subtractors are an extension of adders.
 - Basically, perform addition on a negative number.
- Before we can do subtraction, need to understand negative binary numbers.
- Two types:
 - Unsigned = a separate bit exists for the sign; data bits store the positive version of the number.
 - Signed = all bits are used to store a 2's complement negative number.

Two's complement

- Need to know how to get 1's complement:
 - Given number X with n bits, take $(2^n - 1) - X$
 - Negates each individual bit (bitwise NOT).

01001101	→	10110010
11111111	→	00000000

- 2's complement = (1's complement + 1)

01001101	→	10110011
11111111	→	00000001

Know
this!

- Note: Adding a 2's complement number to the original number produces a result of zero.

(2's complement of A) + A = 0.

The 2's complement of A is like $-A$

Unsigned subtraction

- General algorithm for **A - B**:
 1. Get the **2's complement** of **B** (-B)
 2. Add that value to **A**
 3. If there is an end carry (C_{out} is high), the final result is positive and does not change.
 4. If there is no end carry (C_{out} is low), get the **2's complement** of the result ($B-A$) and add a **negative sign** to it, or set the sign bit high ($-(B-A) = A-B$).

Unsigned subtraction example

- $53 - 27$

$$\begin{array}{r} 00110101 \\ -00011011 \\ \hline \end{array}$$



$$00110101$$

carry bit

$$+11100101$$

$$\begin{array}{r} \\ 1 \\ \hline 00011010 \end{array}$$



sign bit
is low

$$00011010$$

- $27 - 53$

$$\begin{array}{r} 00011011 \\ -00110101 \\ \hline \end{array}$$



$$00011011$$

no carry bit

$$+11001011$$

$$\begin{array}{r} \\ 0 \\ \hline 11100110 \end{array}$$



sign bit
is high

$$-00011010$$

26

-26

Signed subtraction (easier)

- Store negative numbers in 2's complement notation.
 - Subtraction can then be performed by using the binary **adder** circuit with negative numbers.
 - To compute $A - B$, just do $A + (-B)$
 - Need to get $-B$ first (the 2's complement of B)

Signed representations

Why 2's complement is better than 1's complement?

Decimal	Signed 1's	Signed 2's
3	011	011
2	010	010
1	001	001
0	000	000
-0	111	---
-1	110	111
-2	101	110
-3	100	101
-4	---	100

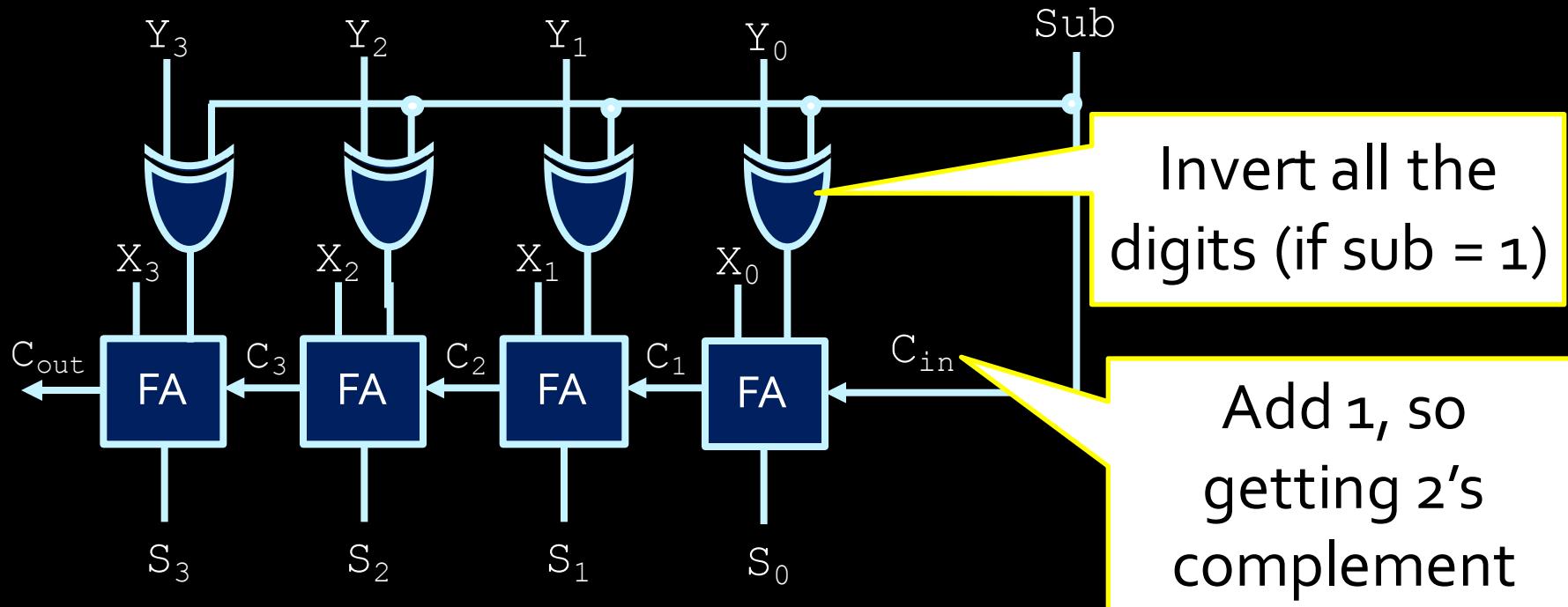
Trivia about sign numbers

- The largest positive 8-bit signed integer?
 - $01111111 = 127$ (0 followed by all 1)
- The smallest negative 8-bit signed integer?
 - $10000000 = -128$ (1 followed by all 0)
- The binary form 8-bit signed integer -1?
 - 11111111 (all one)
- For n-bit signed number there are 2^n possible values
 - 2^{n-1} are negative numbers (e.g. 8 bit, -1 to -128)
 - $2^{n-1}-1$ are positive number (e.g. 8 bit, 1 to 127)
 - and a zero



-128: 10000000 (signed)

Subtraction circuit



- If $sub = 0$, $S = X + Y$
- If $sub = 1$, $S = X - Y$

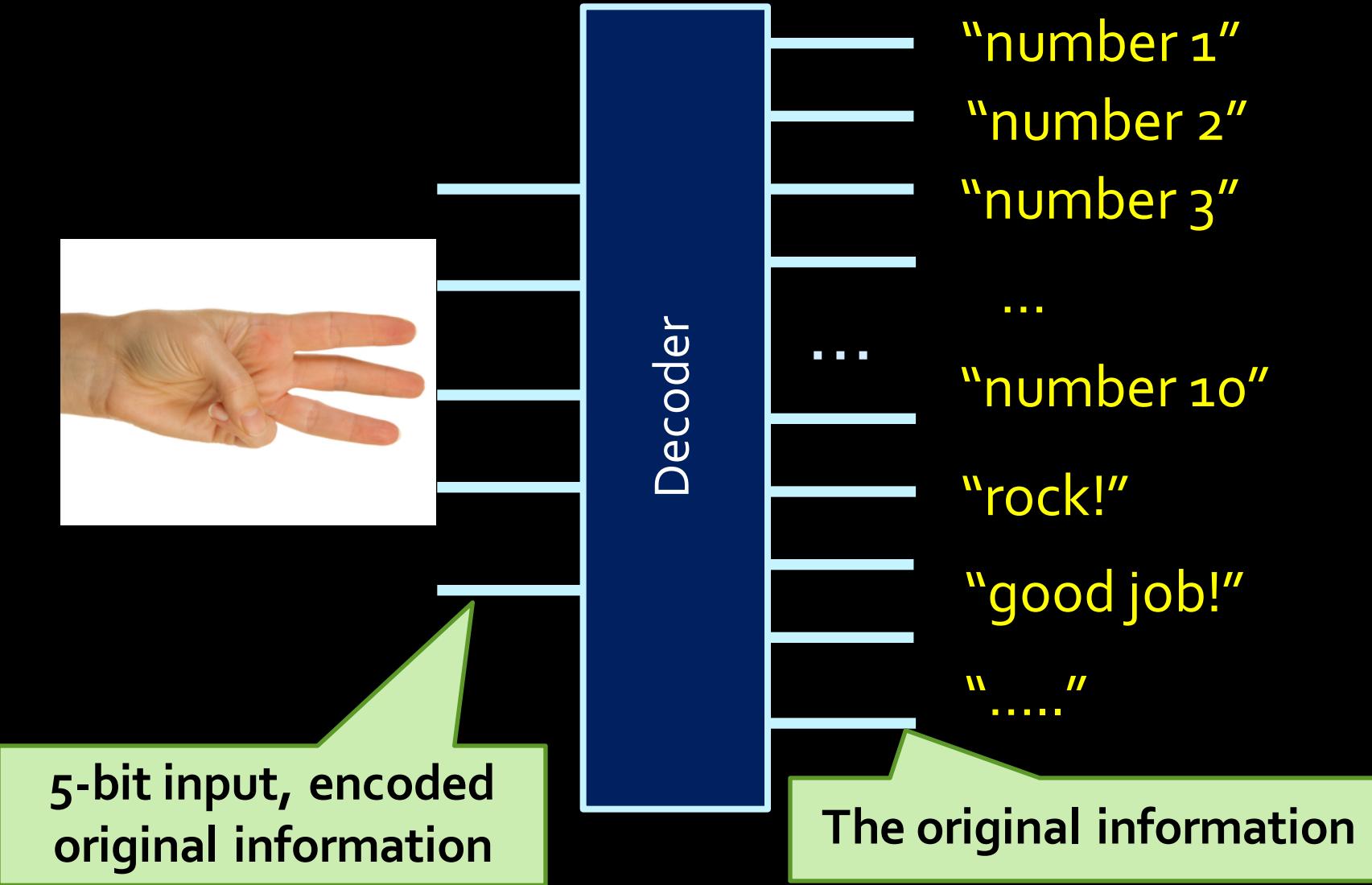
One circuit, both adder or subtractor



Decoders

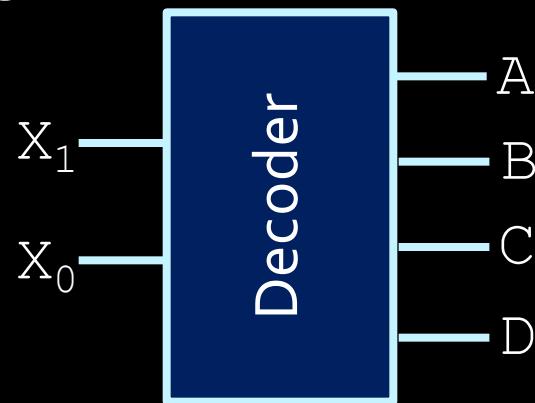


What is a decoder?



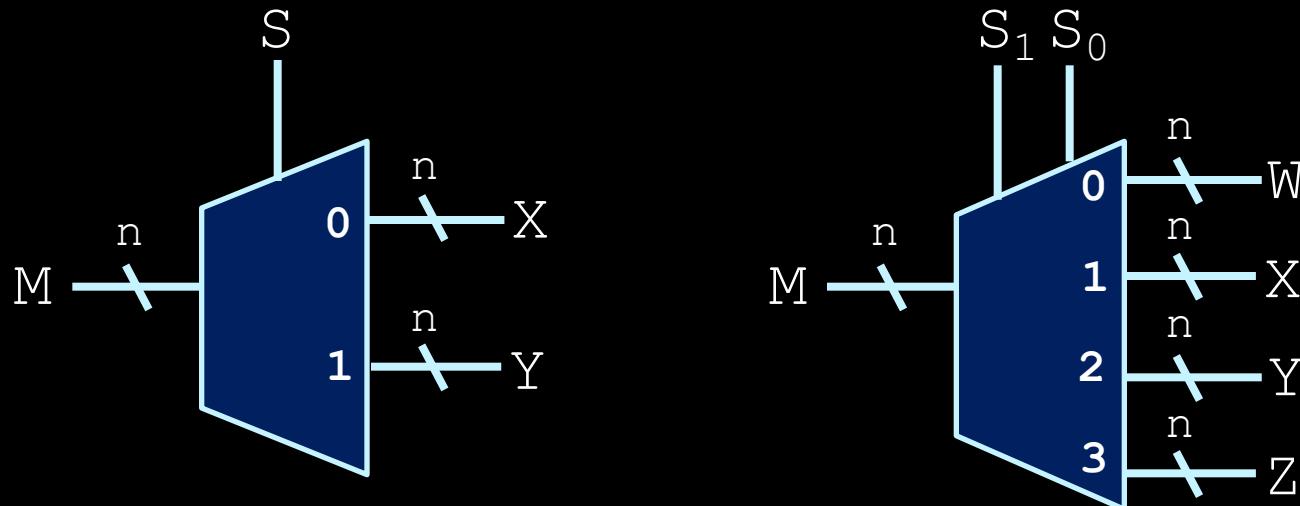
Decoders

- Decoders are essentially translators.
 - Translate from the output of one circuit to the input of another.
- Example: Binary signal splitter
 - Activates one of four output lines, based on a two-digit binary number.



Demultiplexers

- Related to decoders: demultiplexers.
 - Does multiplexer operation, in reverse.



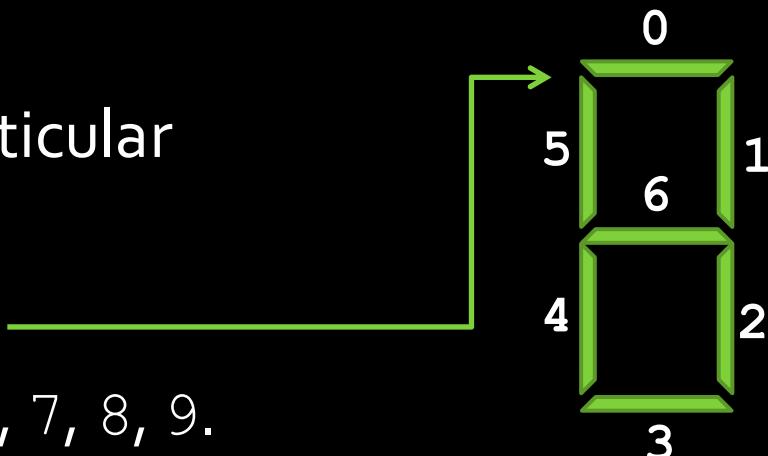
Multiplexer:
Choose one from multiple inputs as output

Demultiplexer:
One input chooses from multiple outputs

7-segment decoder



- Common and useful decoder application.
 - Translate from a 4-digit binary number to the seven segments of a digital display.
 - Each output segment has a particular logic that defines it.
 - Example: Segment 0
 - Activate for values: 0, 2, 3, 5, 6, 7, 8, 9.
 - In binary: 0000, 0010, 0011, 0101, 0110, 0111, 1000, 1001 .
 - First step: Build the truth table and K-map.



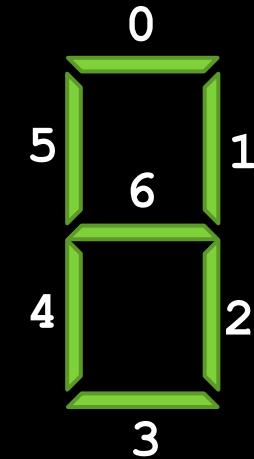
Note

What we talk about here is NOT the same as what we do in Lab 2

- In labs we translate numbers 0, 1, 3, 4, 5, 6 to displayed letters such as (K, E, L, V, I, N)
 - This is specially defined for the lab
- Here we are talking about translating 0, 1, 2, 3, 4, ..., to displayed 0, 1, 2, 3, 4, ...
 - This is more common use

7-segment decoder

- For 7-seg decoders, turning a segment on involves driving it low.
 - i.e. Assuming a 4-digit binary number, segment 0 is low whenever input number is 0000, 0010, 0011, 0101, 0110, 0111, 1000 or 1001, and high whenever input number is 0001 or 0100 .
 - This create a truth table and map like the following...



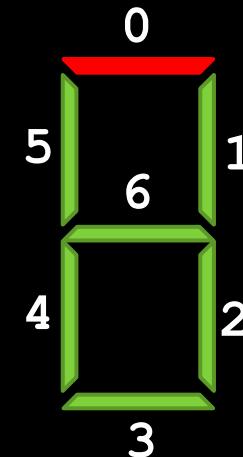
7-segment decoder

\mathbf{x}_3	\mathbf{x}_2	\mathbf{x}_1	\mathbf{x}_0	HEX _o
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0

6 rows missing!
1010 ~ 1111

	$\bar{\mathbf{x}}_1 \cdot \bar{\mathbf{x}}_0$	$\bar{\mathbf{x}}_1 \cdot \mathbf{x}_0$	$\mathbf{x}_1 \cdot \mathbf{x}_0$	$\mathbf{x}_1 \cdot \bar{\mathbf{x}}_0$
$\bar{\mathbf{x}}_3 \cdot \bar{\mathbf{x}}_2$	0	1	0	0
$\bar{\mathbf{x}}_3 \cdot \mathbf{x}_2$	1	0	0	0
$\mathbf{x}_3 \cdot \mathbf{x}_2$	\mathbf{x}	\mathbf{x}	\mathbf{x}	\mathbf{x}
$\mathbf{x}_3 \cdot \bar{\mathbf{x}}_2$	0	0	\mathbf{x}	\mathbf{x}

- $\text{HEX}0 = \bar{\mathbf{x}}_3 \cdot \bar{\mathbf{x}}_2 \cdot \bar{\mathbf{x}}_1 \cdot \mathbf{x}_0$
 $+ \bar{\mathbf{x}}_3 \cdot \mathbf{x}_2 \cdot \bar{\mathbf{x}}_1 \cdot \bar{\mathbf{x}}_0$
- But what about input values from 1010 to 1111?



“Don’t care” values

- Some input values will never happen, so their output values do not have to be defined.
 - Recorded as ‘X’ in the Karnaugh map.
- These values can be assigned to whatever values you want, when constructing the final circuit.

$$\text{HEX0} = \overline{x}_3 \cdot \overline{x}_2 \cdot \overline{x}_1 \cdot x_0 + x_2 \cdot \overline{x}_1 \cdot \overline{x}_0$$

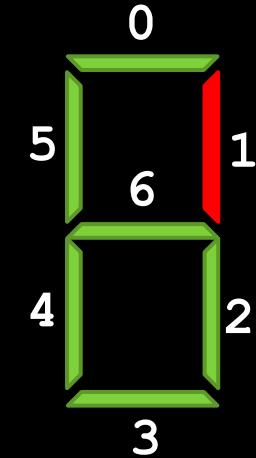
	$\overline{x}_1 \cdot \overline{x}_0$	$\overline{x}_1 \cdot x_0$	$x_1 \cdot x_0$	$x_1 \cdot \overline{x}_0$
$\overline{x}_3 \cdot \overline{x}_2$	0	1	0	0
$\overline{x}_3 \cdot x_2$	1	0	0	0
$x_3 \cdot x_2$	x	x	x	x
$x_3 \cdot \overline{x}_2$	0	0	x	x

Boxes can cover “x”’s, or not, whichever you like.

Again for segment 1

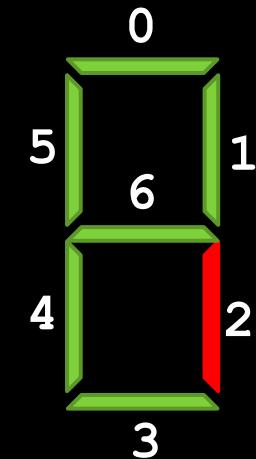
\mathbf{x}_3	\mathbf{x}_2	\mathbf{x}_1	\mathbf{x}_0	HEX_1
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0

	$\bar{\mathbf{x}}_1 \cdot \bar{\mathbf{x}}_0$	$\bar{\mathbf{x}}_1 \cdot \mathbf{x}_0$	$\mathbf{x}_1 \cdot \mathbf{x}_0$	$\mathbf{x}_1 \cdot \bar{\mathbf{x}}_0$
$\bar{\mathbf{x}}_3 \cdot \bar{\mathbf{x}}_2$	0	0	0	0
$\bar{\mathbf{x}}_3 \cdot \mathbf{x}_2$	0	1	0	1
$\mathbf{x}_3 \cdot \bar{\mathbf{x}}_2$	\mathbf{x}	\mathbf{x}	\mathbf{x}	\mathbf{x}
$\mathbf{x}_3 \cdot \bar{\mathbf{x}}_2$	0	0	\mathbf{x}	\mathbf{x}



$$\text{HEX1} = \mathbf{x}_2 \cdot \bar{\mathbf{x}}_1 \cdot \mathbf{x}_0 + \mathbf{x}_2 \cdot \mathbf{x}_1 \cdot \bar{\mathbf{x}}_0$$

Again for segment 2



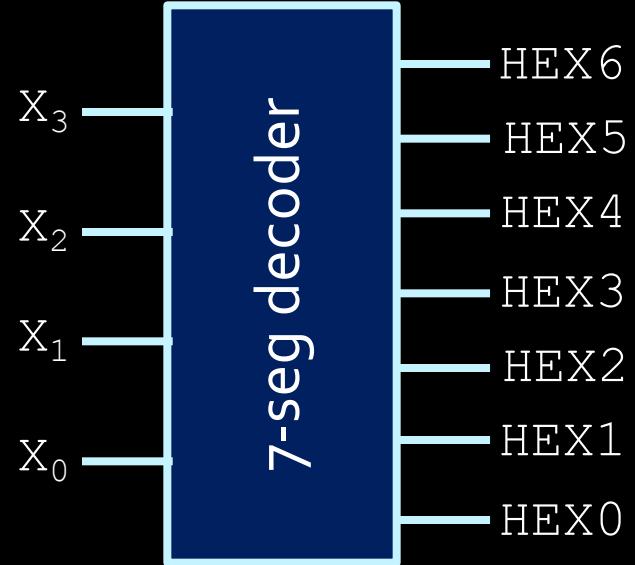
\mathbf{x}_3	\mathbf{x}_2	\mathbf{x}_1	\mathbf{x}_0	HEX_2
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0

	$\bar{\mathbf{x}}_1 \cdot \bar{\mathbf{x}}_0$	$\bar{\mathbf{x}}_1 \cdot \mathbf{x}_0$	$\mathbf{x}_1 \cdot \mathbf{x}_0$	$\mathbf{x}_1 \cdot \bar{\mathbf{x}}_0$
$\bar{\mathbf{x}}_3 \cdot \bar{\mathbf{x}}_2$	0	0	0	1
$\bar{\mathbf{x}}_3 \cdot \mathbf{x}_2$	0	0	0	0
$\mathbf{x}_3 \cdot \bar{\mathbf{x}}_2$	\mathbf{x}	\mathbf{x}	\mathbf{x}	\mathbf{x}
$\mathbf{x}_3 \cdot \bar{\mathbf{x}}_2$	0	0	\mathbf{x}	\mathbf{x}

$$\text{HEX}_2 = \bar{\mathbf{x}}_2 \cdot \mathbf{x}_1 \cdot \bar{\mathbf{x}}_0$$

The final 7-seg decoder

- Decoders all look the same, except for the inputs and outputs.
- Unlike other devices, the implementation differs from decoder to decoder.



Comparators



Comparators

- A circuit that takes in two input vectors, and determines if the first is greater than, less than or equal to the second.
- How does one make that in a circuit?



Basic Comparators

- Consider two binary numbers A and B, where A and B are one bit long.

- The circuits for this would be:

- $A == B$:

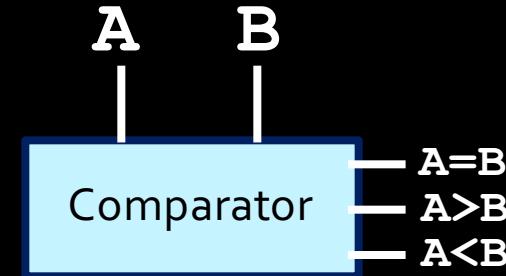
$$A \cdot B + \bar{A} \cdot \bar{B}$$

- $A > B$:

$$A \cdot \bar{B}$$

- $A < B$:

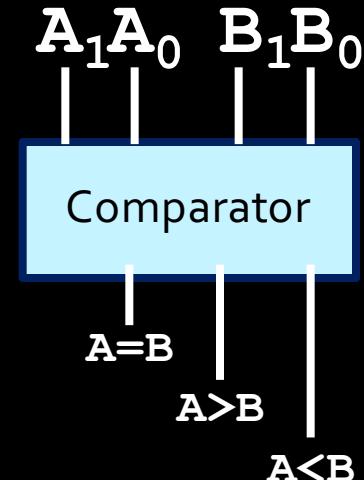
$$\bar{A} \cdot B$$



A	B
0	0
0	1
1	0
1	1

Basic Comparators

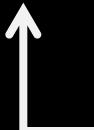
- What if A and B are two bits long?
- The terms for this circuit for have to expand to reflect the second signal.
- For example:



▫ $A==B$:

$$(A_1 \cdot B_1 + \bar{A}_1 \cdot \bar{B}_1) \cdot (A_0 \cdot B_0 + \bar{A}_0 \cdot \bar{B}_0)$$

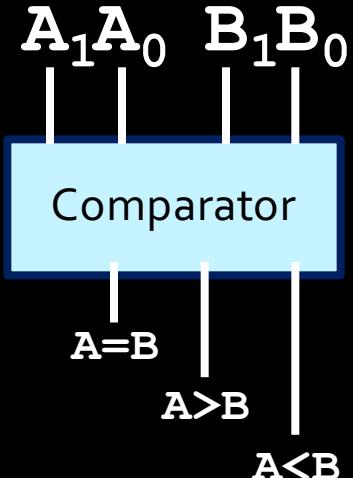
Make sure that the values
of bit 1 are the same



Make sure that the values
of bit 0 are the same

Basic Comparators

- What about checking if A is greater or less than B?



- $A>B$:

$$A_1 \cdot \bar{B}_1 + (A_1 \cdot B_1 + \bar{A}_1 \cdot \bar{B}_1) \cdot (A_0 \cdot \bar{B}_0)$$

Check if first bit satisfies condition



If not, check that the first bits are equal...



...and then do the 1-bit comparison

- $A<B$:

$$\bar{A}_1 \cdot B_1 + (A_1 \cdot B_1 + \bar{A}_1 \cdot \bar{B}_1) \cdot (\bar{A}_0 \cdot B_0)$$

$A > B$ if and only if $A_1 > B_1$ or ($A_1 = B_1$ and $A_0 > B_0$)

General Comparators

- The general circuit for comparators requires you to define equations for each case.
- Case #1: Equality
 - If inputs A and B are equal, then all bits must be the same.
 - Define X_i for any digit i :
 - (equality for digit i)
 - Equality between A and B is defined as:

$$X_i = A_i \cdot B_i + \bar{A}_i \cdot \bar{B}_i$$

$$A == B : X_0 \cdot X_1 \cdot \dots \cdot X_n$$

Comparators

- Case #2: $A > B$

- The first non-matching bits occur at bit i , where $A_i = 1$ and $B_i = 0$. All higher bits match.
- Using the definition for X_i from before:

$$A > B = A_n \cdot \bar{B}_n + X_n \cdot A_{n-1} \cdot \bar{B}_{n-1} + \dots + A_0 \cdot \bar{B}_0 \cdot \prod_{k=1}^n X_k$$

- Case #3: $A < B$

- The first non-matching bits occur at bit i , where $A_i = 0$ and $B_i = 1$. Again, all higher bits match.

$$A < B = \bar{A}_n \cdot B_n + X_n \cdot \bar{A}_{n-1} \cdot B_{n-1} + \dots + \bar{A}_0 \cdot B_0 \cdot \prod_{k=1}^n X_k$$

Comparator truth table

- Given two input vectors of size $n=2$, output of circuit is shown at right.

Inputs				Outputs		
A_1	A_0	B_1	B_0	$A < B$	$A = B$	$A > B$
0	0	0	0	0	1	0
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	0	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	1	0	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	0	1	0

Comparator example (cont'd)

$A < B$:

	$\bar{B}_0 \cdot \bar{B}_1$	$B_0 \cdot \bar{B}_1$	$B_0 \cdot B_1$	$\bar{B}_0 \cdot B_1$
$\bar{A}_0 \cdot \bar{A}_1$	0	1	1	1
$A_0 \cdot \bar{A}_1$	0	0	1	1
$A_0 \cdot A_1$	0	0	0	0
$\bar{A}_0 \cdot A_1$	0	0	1	0

$$LT = B_1 \cdot \bar{A}_1 + B_0 \cdot B_1 \cdot \bar{A}_0 + B_0 \cdot \bar{A}_0 \cdot \bar{A}_1$$

Comparator example (cont'd)

$A=B:$

	$\bar{B}_0 \cdot \bar{B}_1$	$B_0 \cdot \bar{B}_1$	$B_0 \cdot B_1$	$\bar{B}_0 \cdot B_1$
$\bar{A}_0 \cdot \bar{A}_1$	1	0	0	0
$A_0 \cdot \bar{A}_1$	0	1	0	0
$A_0 \cdot A_1$	0	0	1	0
$\bar{A}_0 \cdot A_1$	0	0	0	1

$$EQ = \bar{B}_0 \cdot \bar{B}_1 \cdot \bar{A}_0 \cdot \bar{A}_1 + B_0 \cdot \bar{B}_1 \cdot A_0 \cdot \bar{A}_1 + \\ B_0 \cdot B_1 \cdot A_0 \cdot \bar{A}_1 + \bar{B}_0 \cdot B_1 \cdot \bar{A}_0 \cdot A_1$$

Comparator example (cont'd)

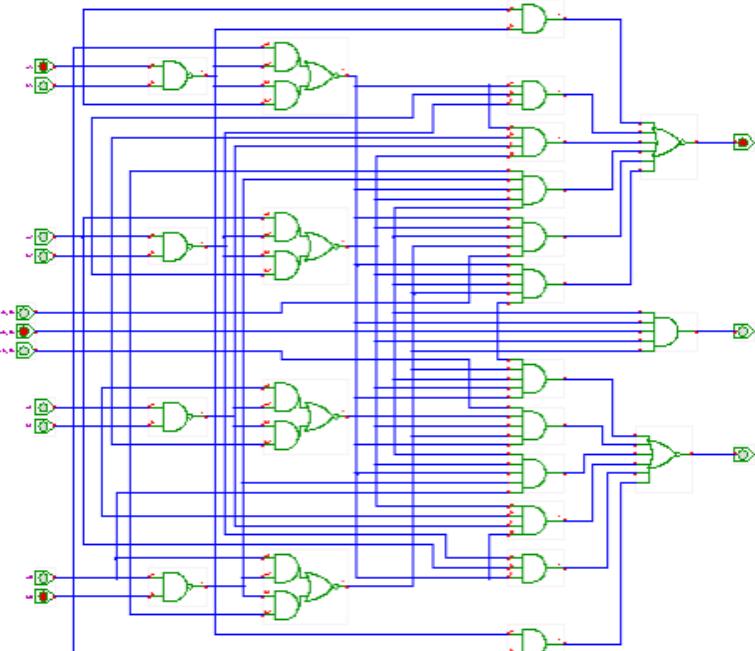
A>B :

	$\bar{B}_0 \cdot \bar{B}_1$	$B_0 \cdot \bar{B}_1$	$B_0 \cdot B_1$	$\bar{B}_0 \cdot B_1$
$\bar{A}_0 \cdot \bar{A}_1$	0	0	0	0
$A_0 \cdot \bar{A}_1$	1	0	0	0
$A_0 \cdot A_1$	1	1	0	1
$\bar{A}_0 \cdot A_1$	1	1	0	0

$$GT = \bar{B}_1 \cdot A_1 + \bar{B}_0 \cdot \bar{B}_1 \cdot A_1 + \bar{B}_0 \cdot A_0 \cdot A_1$$

Comparing larger numbers

- As numbers get larger, the comparator circuit gets more complex.
- At a certain level, it can be easier sometimes to just process the result of a subtraction operation instead.
 - Easier, less circuitry, just not faster.



Today we learned

Nothing new really. Used the circuit creation procedure to create some commonly useful circuits

- Mux and demux
- Adder and subtractor
- Decoder
- Comparator

Next week:

- Sequential circuits: circuits that have **memories**.

**QUIZ
TIME!**

Question 1

Given that minterm $A'B'CD'$ is m_2 (0010)

$AB'C'D$ is m_9 (1001)

Question 2

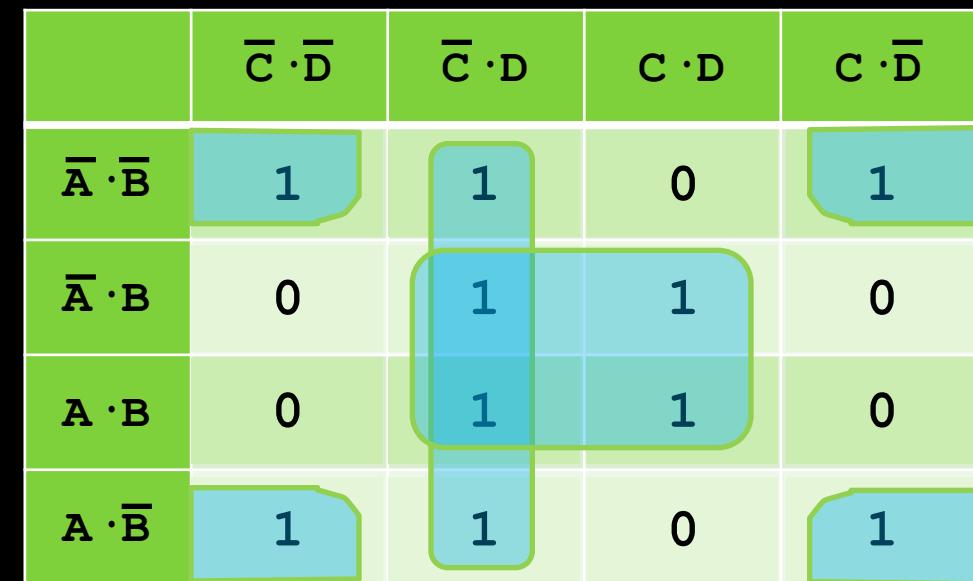
Consider the **optimal** set of boxes that result in the most reduced circuit expression of the following K-map.

How many boxes are needed?

3

What is the size of the smallest box?

4



Question 3

Convert the following Sum-Of-Minterms to the **equivalent** Product-Of-Maxterms expression.

$$AB + A'B$$

Ans: $(A + B)(A' + B)$

Draw the truth table, the minterms corresponds to the rows with output 1, to get POM, look at the rows with output 0.

In this question, rows with output 1 are 11 and 01, rows with output 0 are 00 (maxterm $A+B$) and 10 (maxterm $A' + B$)

CSC258 Winter 2016

Lecture 4

Learning

- Labs
 - You're ready to start preparing for the lab when the handout is posted, no need to wait until Monday's lecture.
 - You will be kept (reasonably) busy with all the pre-lab and in-lab works throughout the term. Need to be persistent to keep on track.
 - Labs are very important learning components for this course. The knowledge involved in the labs will be the part you learn the best in this course.
 - In exams, if certain questions require deeper understanding than others; they are more likely to be related to what you did in the labs.
 - If you have trouble preparing for labs, come to office hours.

Learning

Quizzes

- It also serves as a “weekly reality check” of whether you have really understood last week’s content.
- It may be for bonus marks, but it’s content is not “extra”.
- Solutions (with explanations) are included when slides are posted. Make sure you know the right answers; if have question, ask on Piazza or come to office hours.
- Cheating in Quiz?!
 - Caught once, no bonus for the term.

Comparators (leftover from last week)



Comparators

- A circuit that takes in two input vectors, and determines if the first is greater than, less than or equal to the second.
- How does one make that in a circuit?



Basic Comparators

- Consider two binary numbers A and B, where A and B are one bit long.

- The circuits for this would be:

- $A == B$:

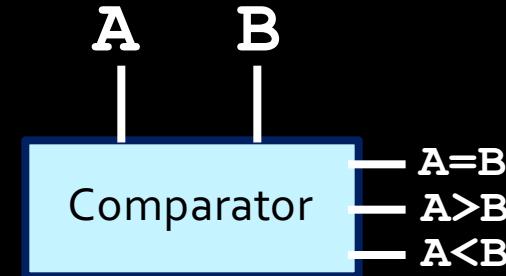
$$A \cdot B + \bar{A} \cdot \bar{B}$$

- $A > B$:

$$A \cdot \bar{B}$$

- $A < B$:

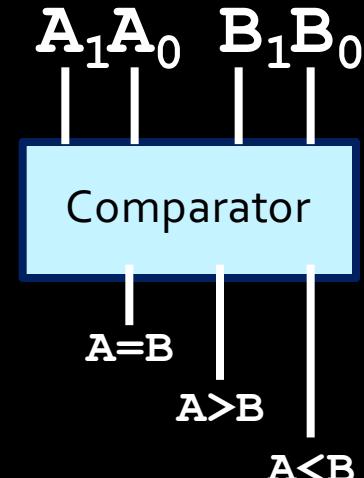
$$\bar{A} \cdot B$$



A	B
0	0
0	1
1	0
1	1

Basic Comparators

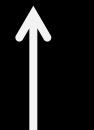
- What if A and B are two bits long?
- The terms for this circuit for have to expand to reflect the second signal.
- For example:



▫ $A==B$:

$$(A_1 \cdot B_1 + \bar{A}_1 \cdot \bar{B}_1) \cdot (A_0 \cdot B_0 + \bar{A}_0 \cdot \bar{B}_0)$$

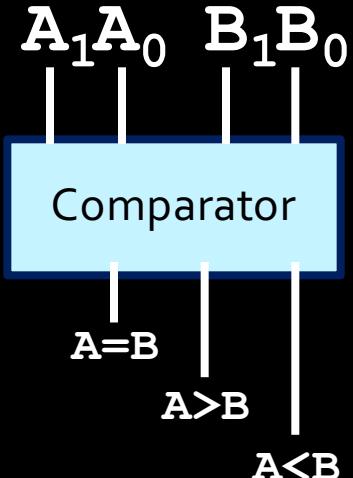
Make sure that the values
of bit 1 are the same



Make sure that the values
of bit 0 are the same

Basic Comparators

- What about checking if A is greater or less than B?



- $A>B$:

$$A_1 \cdot \bar{B}_1 + (A_1 \cdot B_1 + \bar{A}_1 \cdot \bar{B}_1) \cdot (A_0 \cdot \bar{B}_0)$$

Check if first bit satisfies condition



If not, check that the first bits are equal...



...and then do the 1-bit comparison

- $A<B$:

$$\bar{A}_1 \cdot B_1 + (A_1 \cdot B_1 + \bar{A}_1 \cdot \bar{B}_1) \cdot (\bar{A}_0 \cdot B_0)$$

$A > B$ if and only if $A_1 > B_1$ or ($A_1 = B_1$ and $A_0 > B_0$)

Comparing large numbers

- The circuit complexity of comparators increases quickly as the input size increases.
- For comparing large number, it may make more sense to just use a subtractor.
 - Subtract and then check the sign bit.

New Topic:
Sequential Circuit

Something we can't do yet

- How does the Tickle Me Elmo work?

<https://www.youtube.com/watch?v=zG62dirxRgc>

Same input, different outputs.

Two kinds of circuits

- So far, we've dealt with **combinational circuits**:
 - Circuits where the output values are entirely dependent and predictable from the input values.
- Another class of circuits: **sequential circuits**
 - Circuits that also depend on both the inputs and the previous state of the circuit.

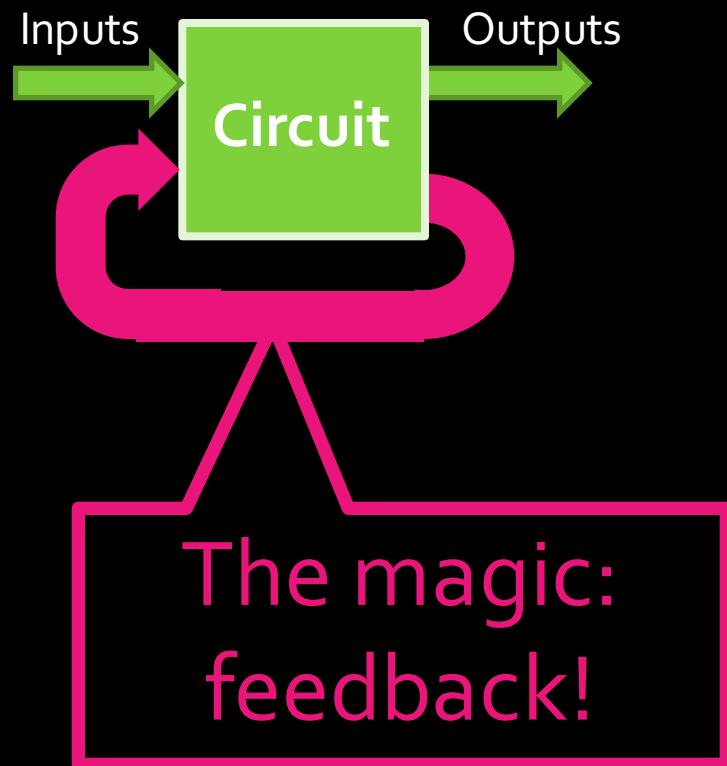
In other words...

- Combination circuits does NOT have memory
 - Everything is based on current inputu value
- Sequential Circuits have **memory**.
 - They **remember** something from the past (the previous state of the circuit), and its output depends on that
 - It is why computers have memory (e.g., RAM)

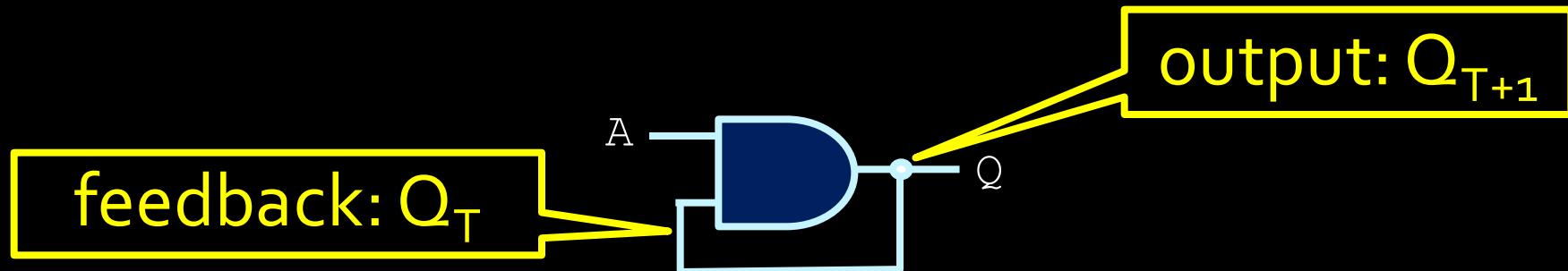
Sequential circuits

- This creates circuits whose **internal state** can change over time, where the same input values can result in different outputs.
- Why would we need circuits like this?
 - Memory values
 - To remember stuff
 - Reacting to changing inputs
 - “Output X = 1 when input A changes”

How can a circuit have memory?



Truth table of a sequential circuit

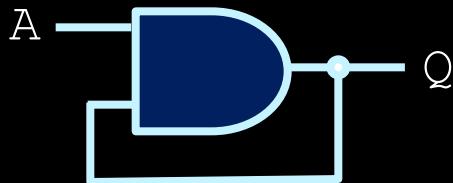


A	Q_T	$Q_{T+1} = A \cdot Q_T$
0	0	0
0	1	0
1	0	0
1	1	1

In these truth tables, Q_T and Q_{T+1} represent the values of Q at a time T , and a point in time immediately after $(T+1)$

The output not only depends on the input A , but also depends on the previous state of the circuit.

AND with feedback

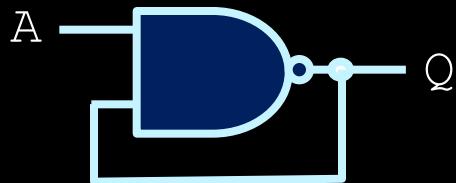


A	Q_T	$Q_{T+1} = A \cdot Q_T$
0	0	0
0	1	0
1	0	0
1	1	1

(0, 1, 0) is a transient state since it will become (0, 0, 0) immediately

Once the output is 0, it will be stuck at 0, no matter how you change A.
It has a memory that cannot be changed.

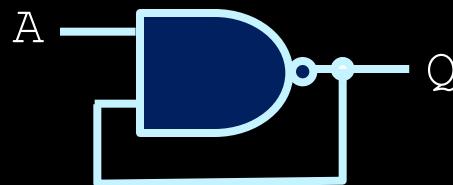
NAND with feedback, more interesting



A	Q_T	Q_{T+1}	
0	0	1	Transient (-> (0, 1, 1))
0	1	1	
1	0	1	Oscillates between each other
1	1	0	

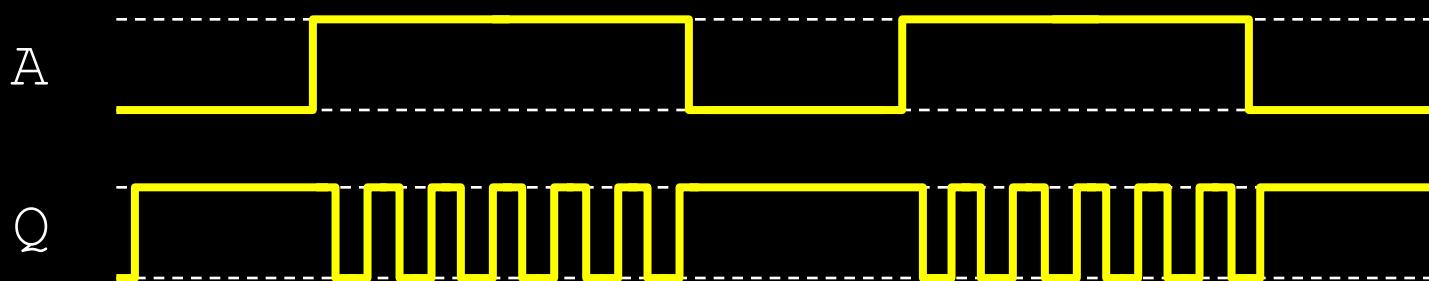
Output Q_{T+1} can be changed by changing A

NAND waveform behaviour



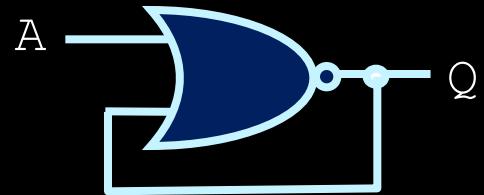
A	Q_T	Q_{T+1}
0	0	1
0	1	1
1	0	1
1	1	0

Oscillates



Input A can control output Q, but the behavior of Q is not very stable.

NOR with feedback



A	Q_T	Q_{T+1}	
0	0	1	Oscillates between each other
0	1	0	
1	0	0	
1	1	0	Transient (-> (1, 0, 0))

Output Q_{T+1} can be changed by changing A

Feedback behaviour

- NAND behaviour

A	Q_T	Q_{T+1}
0	0	1
0	1	1
1	0	1
1	1	0

- NOR behaviour

A	Q_T	Q_{T+1}
0	0	1
0	1	0
1	0	0
1	1	0

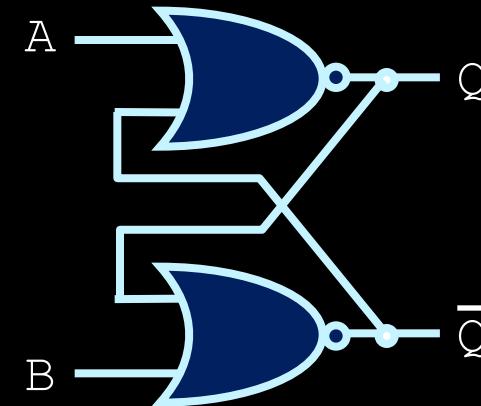
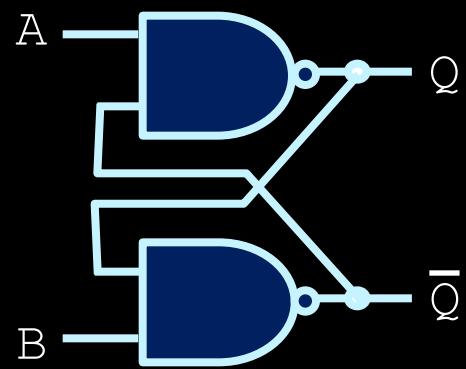
- What makes NAND and NOR feedback circuits different?
 - Unlike the AND and OR gate circuits (which get stuck), the output Q_{T+1} can be changed, based on A.
- However, gates like these that feed back on themselves could enter an unsteady state.

Latch

A feedback circuit with (sort-of) stable behaviour

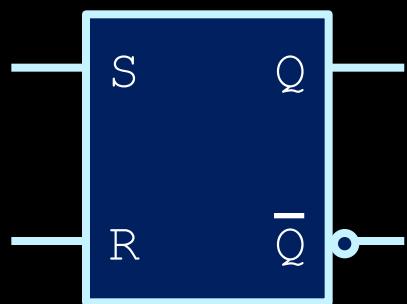
Latches

- If multiple gates of these types are combined, you can get more steady behaviour.

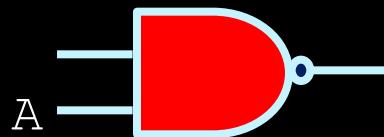


- These circuits are called **latches**.

S'R' Latch



Warm up

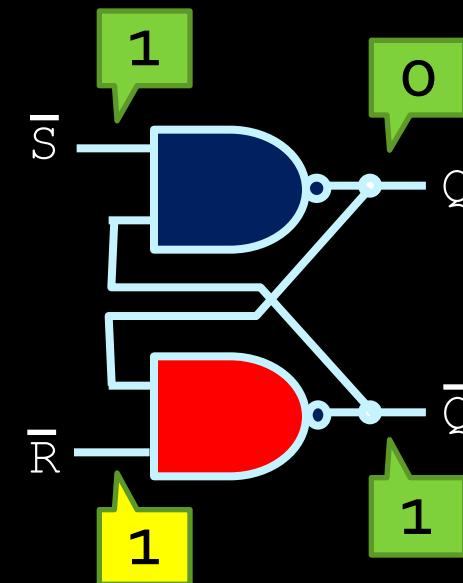
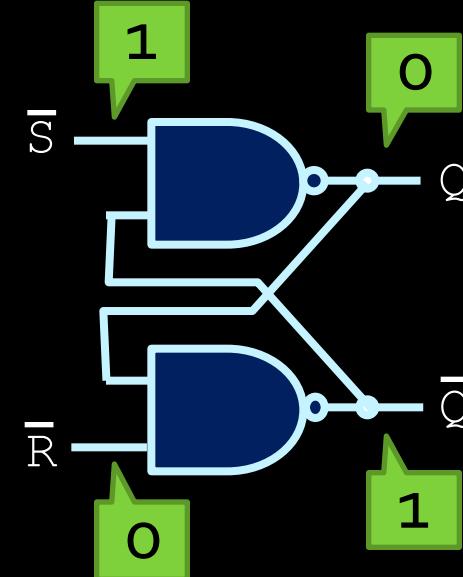


Knowing $A = 0$, what can you say about the output of the NAND gate?

- Must be 1, regardless of the other input
- i.e., a zero input “**locks**” the NAND gate

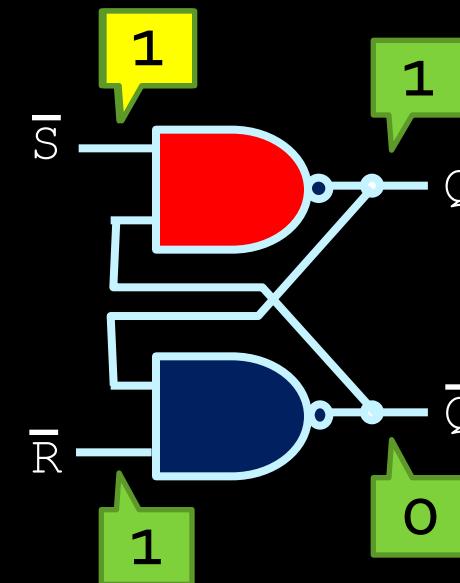
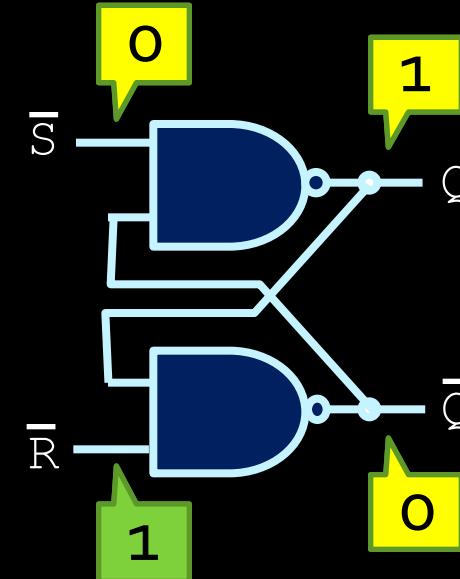
$S'R'$ latch: Case 1

- Let's see what happens when the input values are changed...
 - Assume that S' and R' are set to 1 and 0 to start.
 - The R' input sets the output Q' to 1, which sets the output Q to 0.
 - Setting R' to 1 keeps the output value Q' at 1, which maintains both output values.

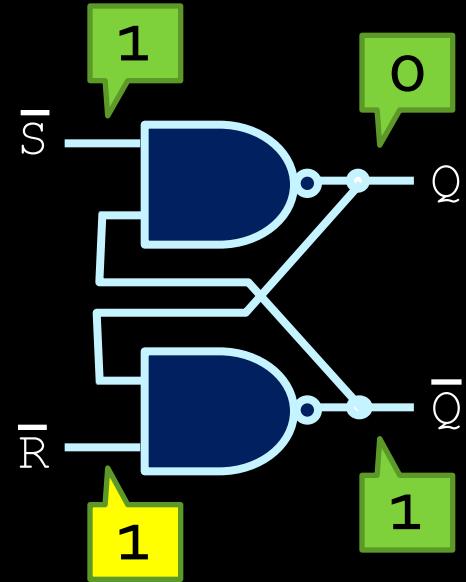


$S'R'$ latch: Case 2

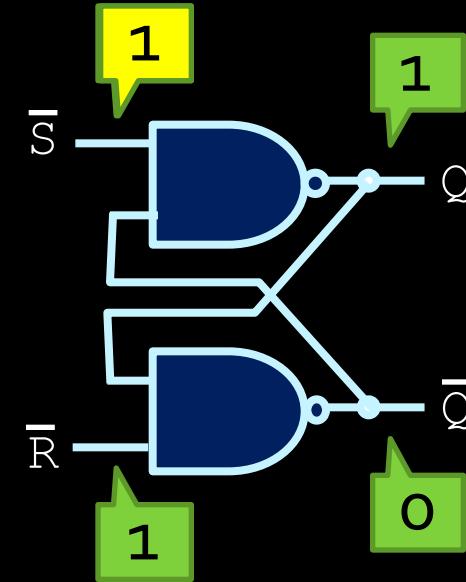
- (continuing from previous)
 - S' and R' start with values of 1, when S' is set to 0.
 - This sets output Q to 1, which sets the output Q' to 0.
 - Setting S' back to 1 keeps the output value Q at 0, which maintains both output values.



Whaaaaat?!

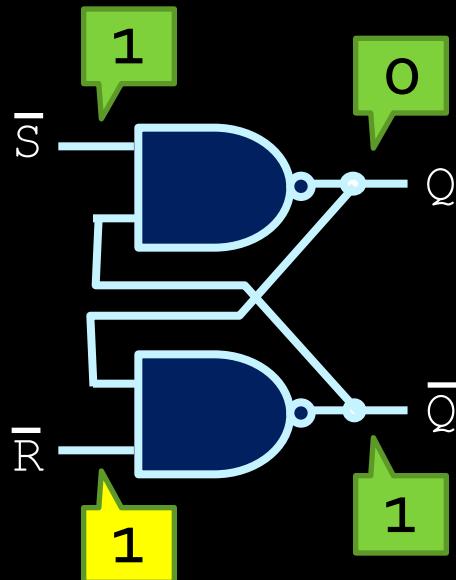
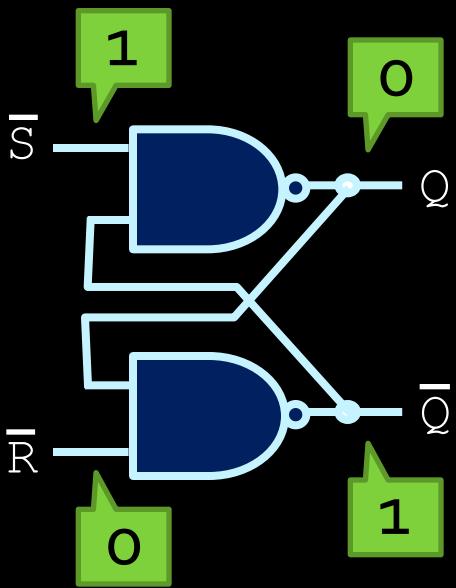


Case 1



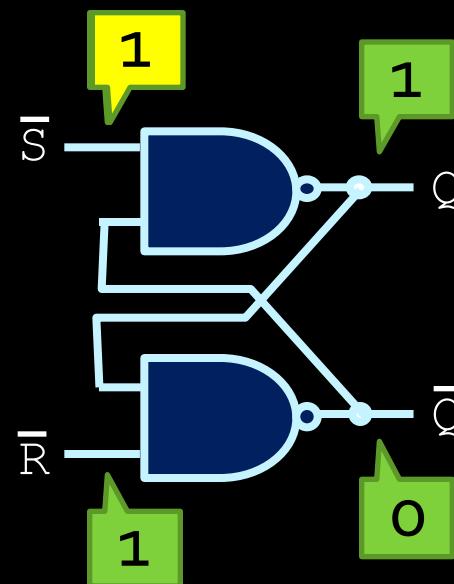
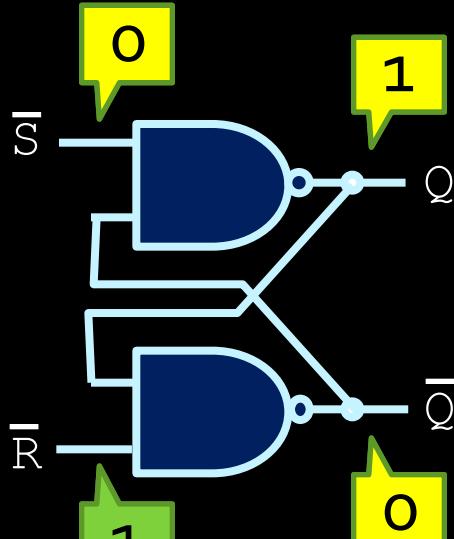
Case 2

Same input, different outputs

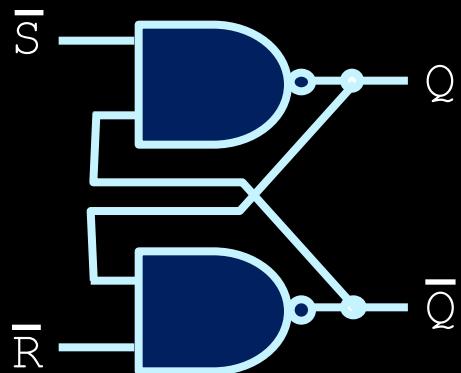


Output of
inputs 11 :

maintain
the previous
output!



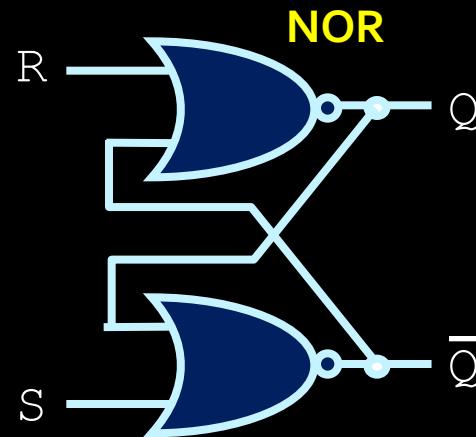
$S'R'$ latch



\bar{S}	\bar{R}	Q_T	\bar{Q}_T	Q_{T+1}	\bar{Q}_{T+1}
0	0	X	X	1	1
0	1	X	X	1	0
1	0	X	X	0	1
1	1	0	1	0	1
1	1	1	0	1	0

- S and R are called “set” and “reset” respectively.
- When $S' = 0$, $R' = 1$, Q is 1
- When $S' = 1$, $R' = 0$, Q is 0
- When $S'R' = 11$, same as previous state (01 or 10)
- How about going from 00 to 11
 - Depends on whether it changes from 00 to 01 to 11, or from 00 to 10 to 11
 - **unstable behaviour**

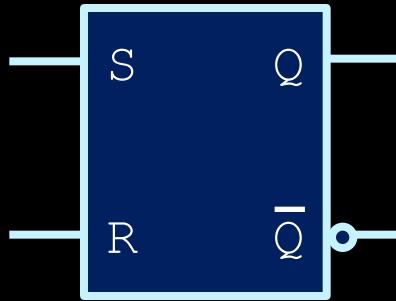
SR latch, with NOR gates



S	R	Q_T	\bar{Q}_T	Q_{T+1}	\bar{Q}_{T+1}
0	0	0	1	0	1
0	0	1	0	1	0
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	0	0

- In this case, S and R are “set” and “reset”.
- In this case, the circuit “remembers” previous output when going from 10 or 01 to 00.
- As with $\overline{S}\overline{R}$ latch, unstable behaviour is possible, but this time when inputs go from 11 to 00.

Summary of S'R' / SR Latch behaviour



S: “set”; R: “reset”

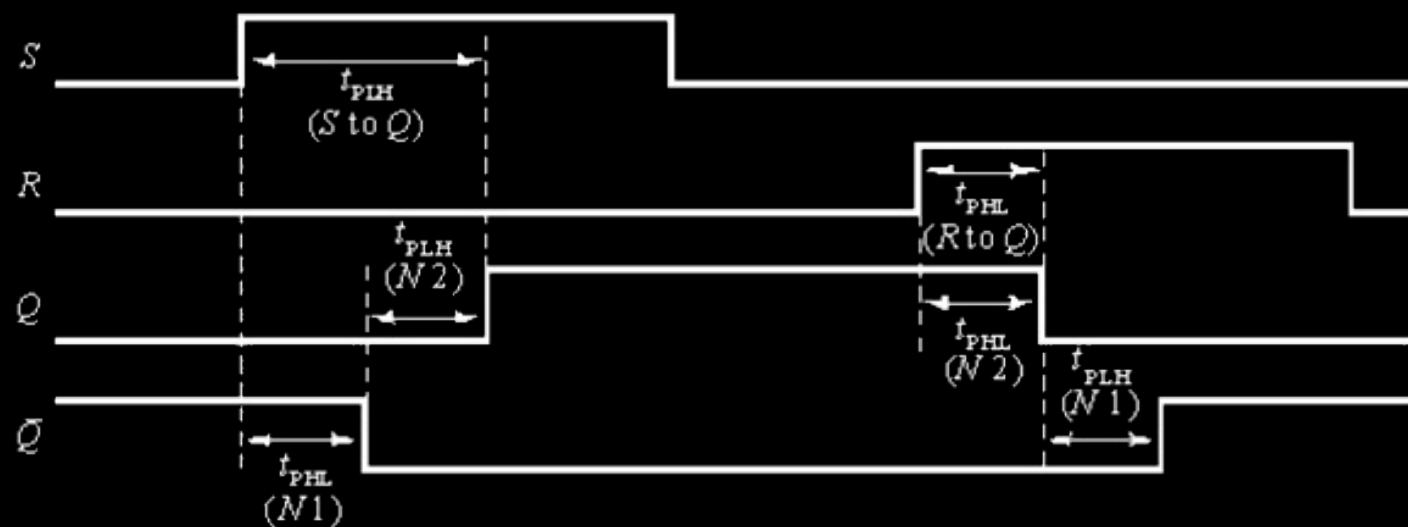
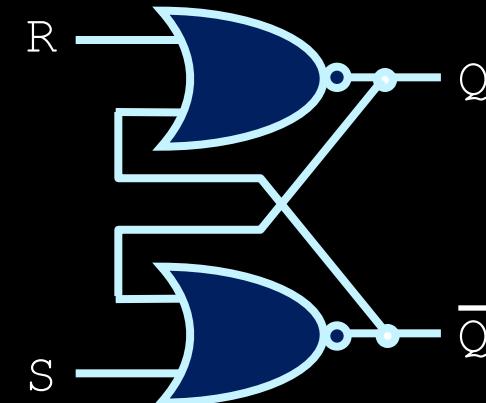
$S = 1, R = 0: Q = 1$

$S = 0, R = 1: Q = 0$

$S = 0, R = 0:$ “keep”

Be aware of delays

- Important to note that, in reality, the output signals don't change instantaneously, but with a certain delay.



More on instability

- Unstable behaviour occurs when a S'R' latch goes from 00 to 11, or a SR latch goes from 11 to 00.
 - The signals don't change simultaneously, so the outcome depends on which signal changes first.
- Because of the unstable behaviour, 00 is considered a **forbidden state** in NAND-based S'R' latches, and 11 is considered a **forbidden state** in NOR-based SR latches.

More on instability

\bar{S}	\bar{R}	Q_T	\bar{Q}_T	Q_{T+1}	\bar{Q}_{T+1}
0	0	X	X	1	1
0	1	X	X	1	0
1	0	X	X	0	1
1	1	0	1	0	1
1	1	1	0	1	0

Forbidden state

$\bar{S}\bar{R}$ -latch

“set” and “reset”
cannot be both 1

S	R	Q _T	\bar{Q}_T	Q _{T+1}	\bar{Q}_{T+1}
0	0	0	1	0	1
0	0	1	0	1	0
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	0	0

SR-latch

Forbidden state

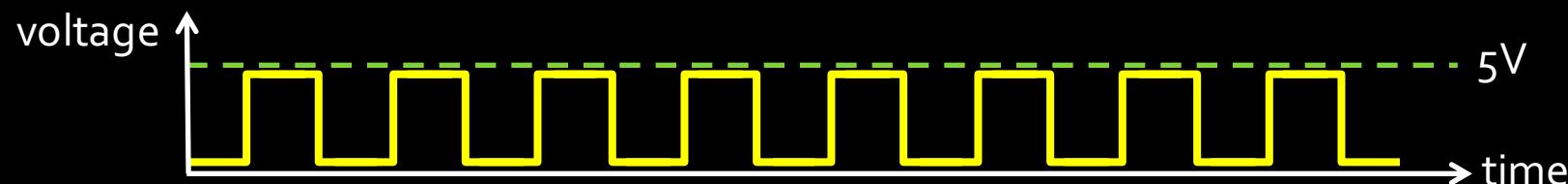
Reading from latches

- Now we have circuit units that can **store** high or low values.
- How do we distinguish
 - “5 highs in a row”
 - “10 highs in a row”
- We want to **sample** the signal with certain frequency.
- Need to use some sort of timing signal, to let the circuit know when the output may be sampled.

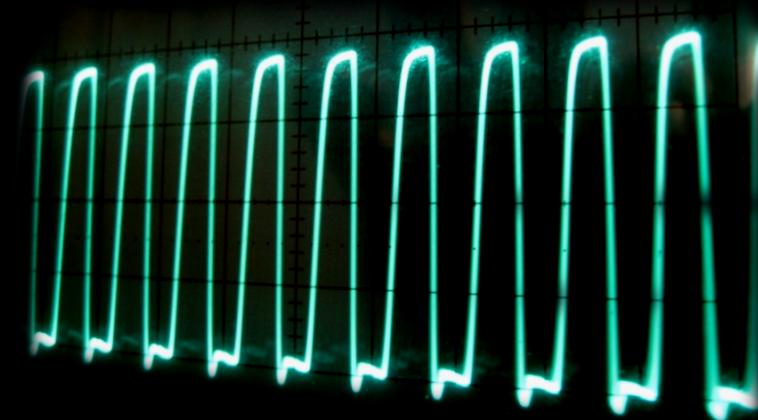
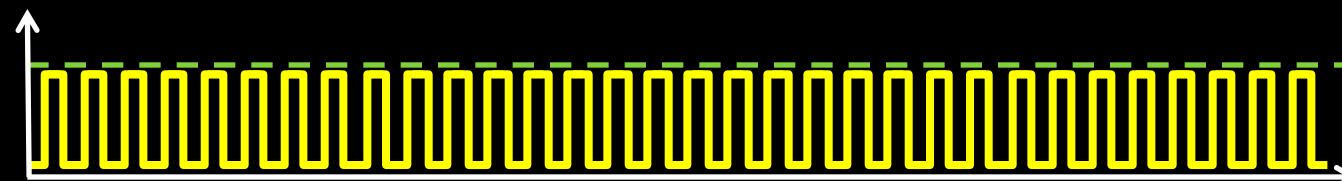
→ **clock signals.**

Clock signals

- “Clocks” are a regular pulse signal, where the high value indicates that the output of the latch may be sampled.
- Usually drawn as:

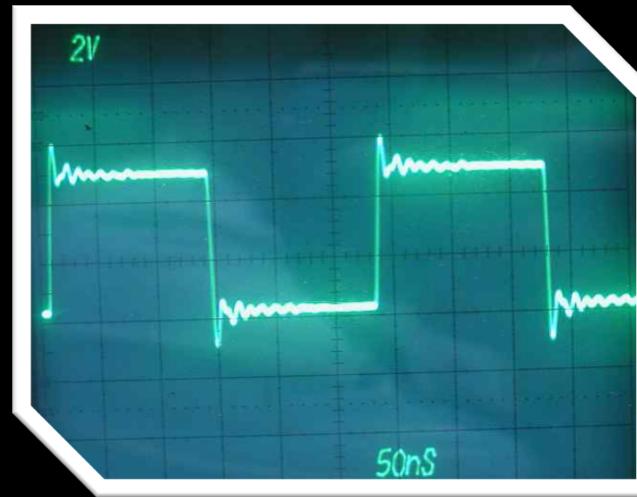


- But looks more like (frequency is usually high):



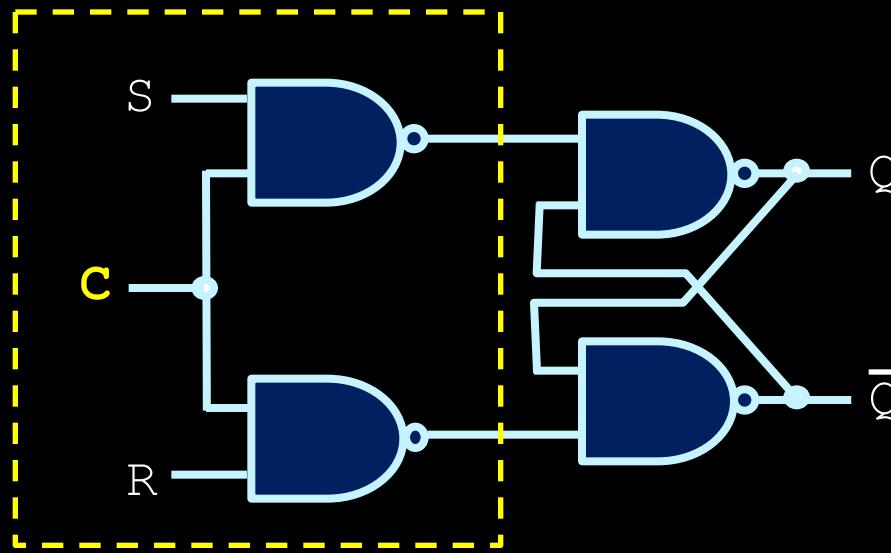
Signal restrictions

- What's the limit to how fast the latch circuit can be sampled?
- Determined by:
 - latency time of transistors
 - Setup and hold time
 - setup time for clock signal
 - Jitter
 - Gibbs phenomenon
- **Frequency** = how many pulses occur per second, measured in Hertz (or Hz).
- What Intel and AMD try to increase every year.



Clocked SR Latch

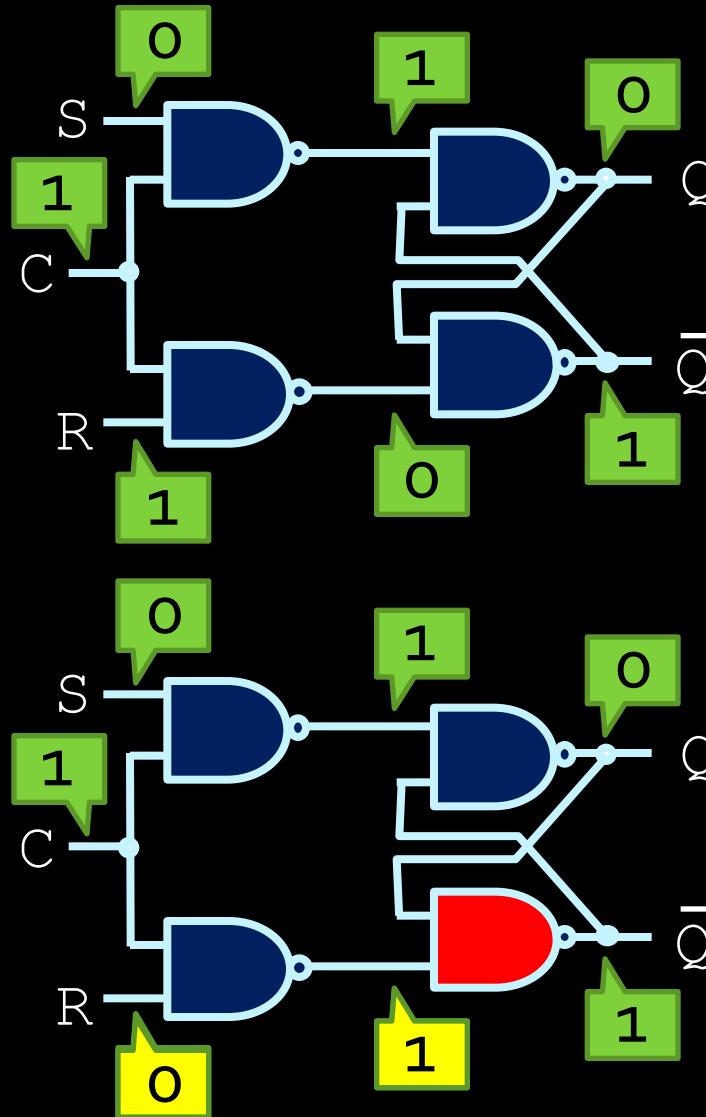
Clocked SR latch



- By adding another layer of NAND gates to the $\bar{S}\bar{R}$ latch, we end up with a **clocked SR latch** circuit.
- The clock is often connected to a pulse signal that alternates regularly between 0 and 1.

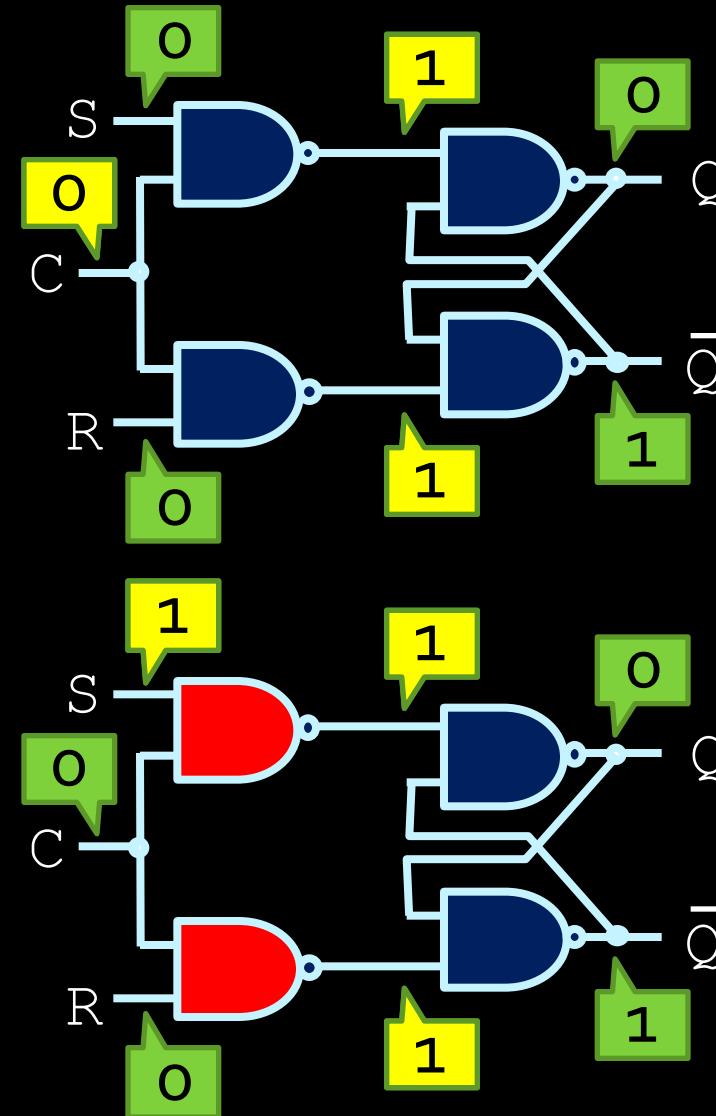
Clocked SR latch behaviour

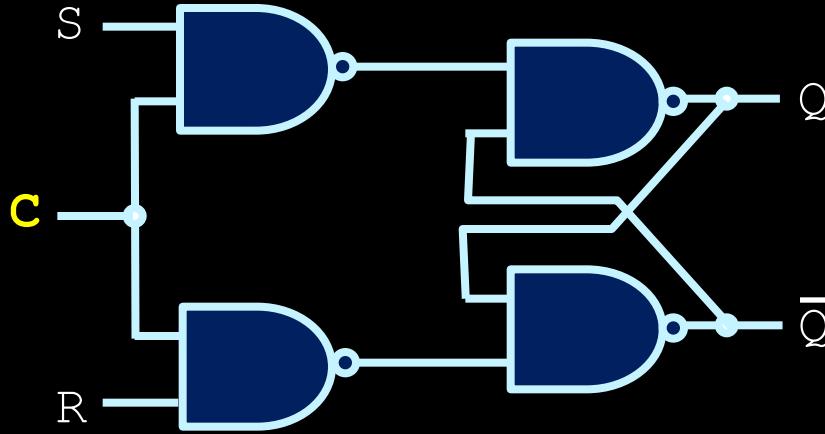
- Same behaviour as SR latch, but with timing:
 - Start off with S=0 and R=1, like earlier example.
 - If **clock is high**, the first NAND gates invert those values, which get inverted again in the output.
 - Setting both inputs to 0 maintains the output values.



Clocked SR latch behaviour

- Continued from previous:
 - Now set the **clock low**.
 - Even if the inputs change, the low clock input prevents the change from reaching the second stage of NAND gates.
 - Result: the clock needs to be high in order for the inputs to have any effect.

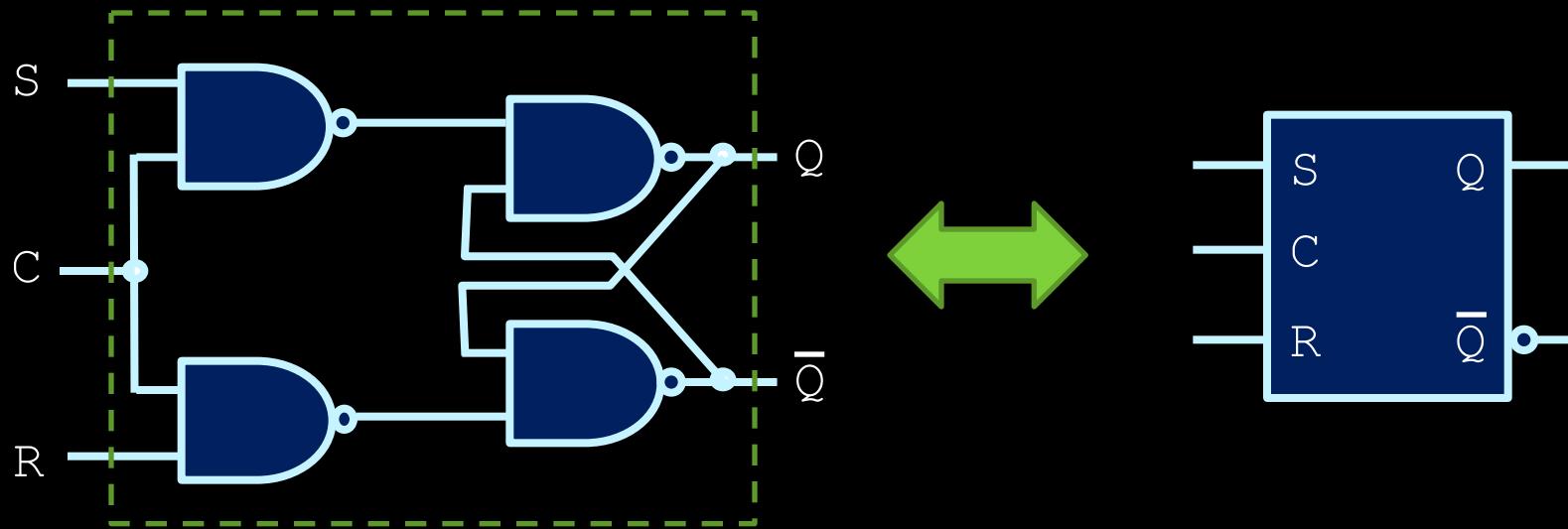




When clock is high, behave like a SR latch.

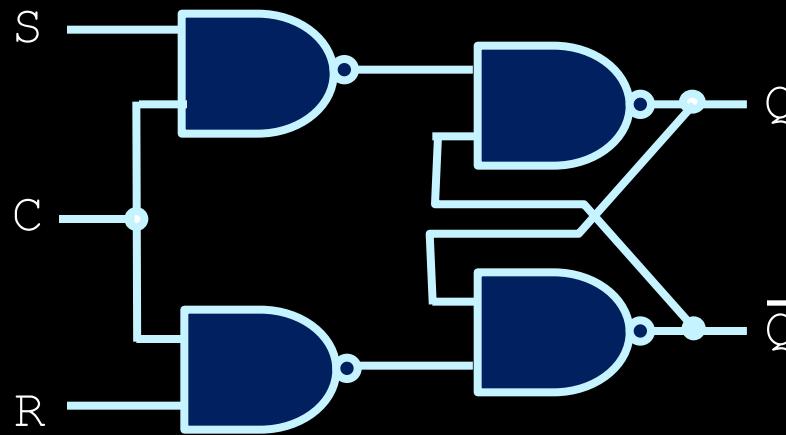
When clock is low, S and R are blocked and there is no way to change the output.

Clocked SR latch



- This is the typical symbol for a clocked SR latch.
- This only allows the S and R signals to affect the circuit when the clock input (C) is high.
- Note: the small NOT circle after the \bar{Q} output is simply the notation to use to denote the inverted output value. It's not an extra NOT gate.

Clocked SR latch behaviour



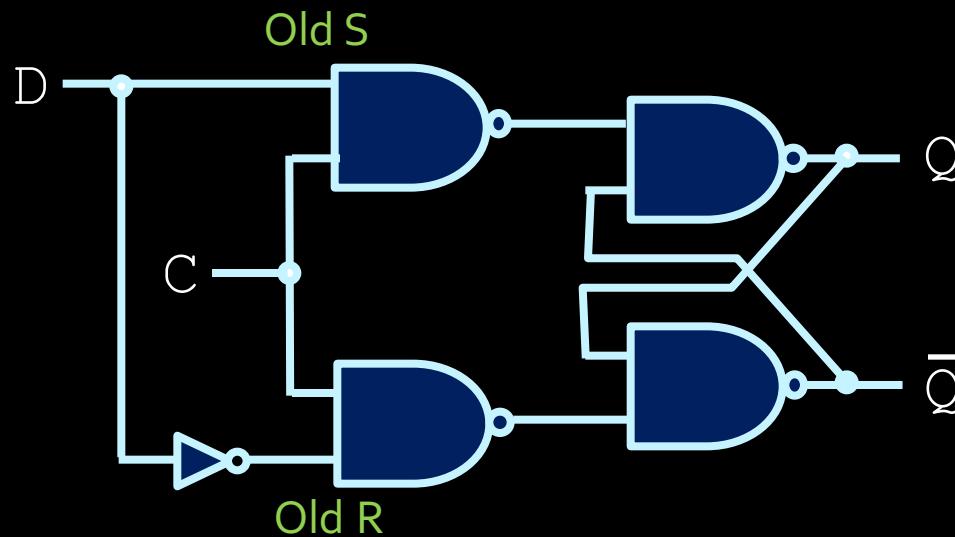
Q_T	S	R	Q_{T+1}	Result
0	0	0	0	no change
0	0	1	0	reset
0	1	0	1	set
0	1	1	?	???
1	0	0	1	no change
1	0	1	0	reset
1	1	0	1	set
1	1	1	?	???

- Assuming the clock is 1, we still have a problem when S and R are both 1, since it is the **forbidden state**.
 - Better design: prevent S and R from both going high.

D latch

prevent S and R from both going high

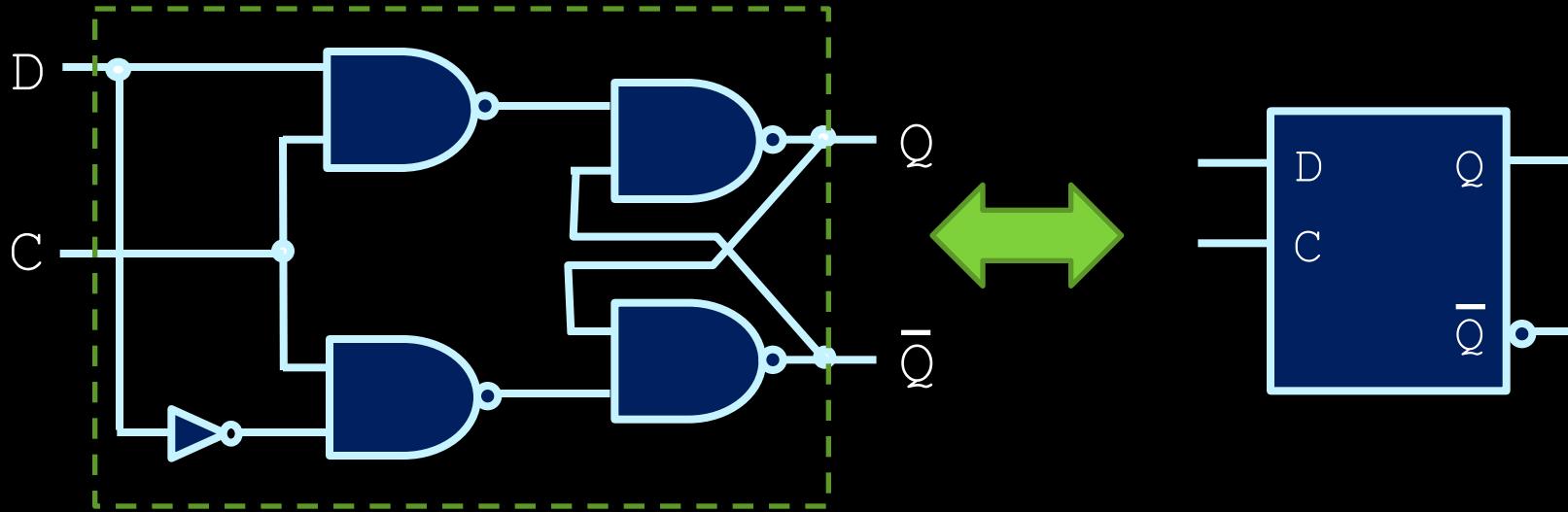
D latch



Q_T	D	Q_{T+1}
0	0	0
0	1	1
1	0	0
1	1	1

- By making the inputs to R and S dependent on a single signal D, you avoid the indeterminate state problem.
- The value of D now sets output Q low or high.

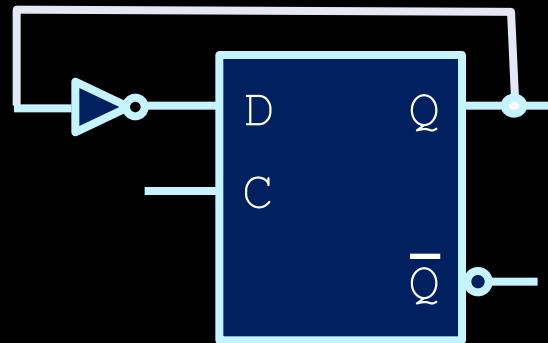
D latch



- This design is good, but still has problems.
 - i.e. timing issues.

Latch timing issues

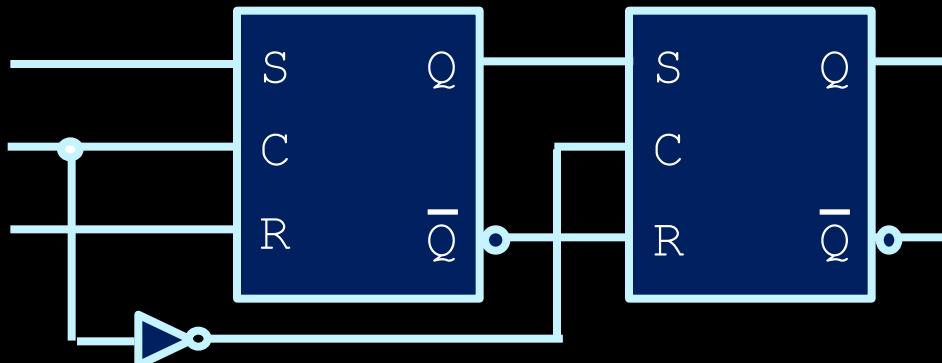
- Consider the circuit on the right:
- When the clock signal is high, the output looks like the waveform below:
 - Output keeps toggling back and forth.



Would be nice if Q changes only once within one clock cycle

Latch timing issues

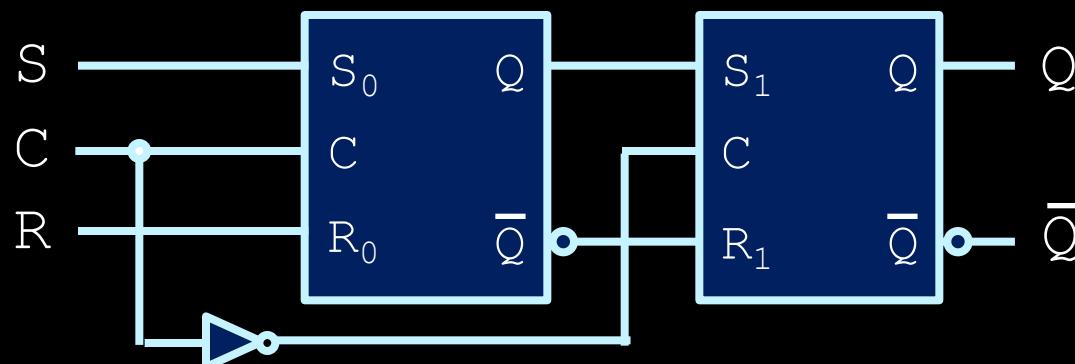
- Preferable behaviour:
 - Have output change only once when the clock pulse changes. 
 - Solution: create disconnect between circuit output and circuit input, to prevent unwanted feedback and changes to output.



Only one latch is active at one time, so in order for a change to propagate from input to output, you need to wait at least for one turn (a clock cycle) of both being active.

SR master-slave flip-flop

- A **flip-flop** is a latched circuit whose output is triggered with the **rising edge** or **falling edge** of a clock pulse.
- Example: The SR master-slave flip-flop



Demo: Human flesh flip-flop



1. Need two volunteers, A and B
2. Clock signal: “flip” and “flop”
3. Person A open eyes when hearing “flip”,
and close eyes when hearing “flop”
4. Person B does the opposite
5. Person A gestures the number (Latch 1
output) upon seeing it from Larry’s input.
6. When Person B sees the number
gestured by A, shout it out. (final output)

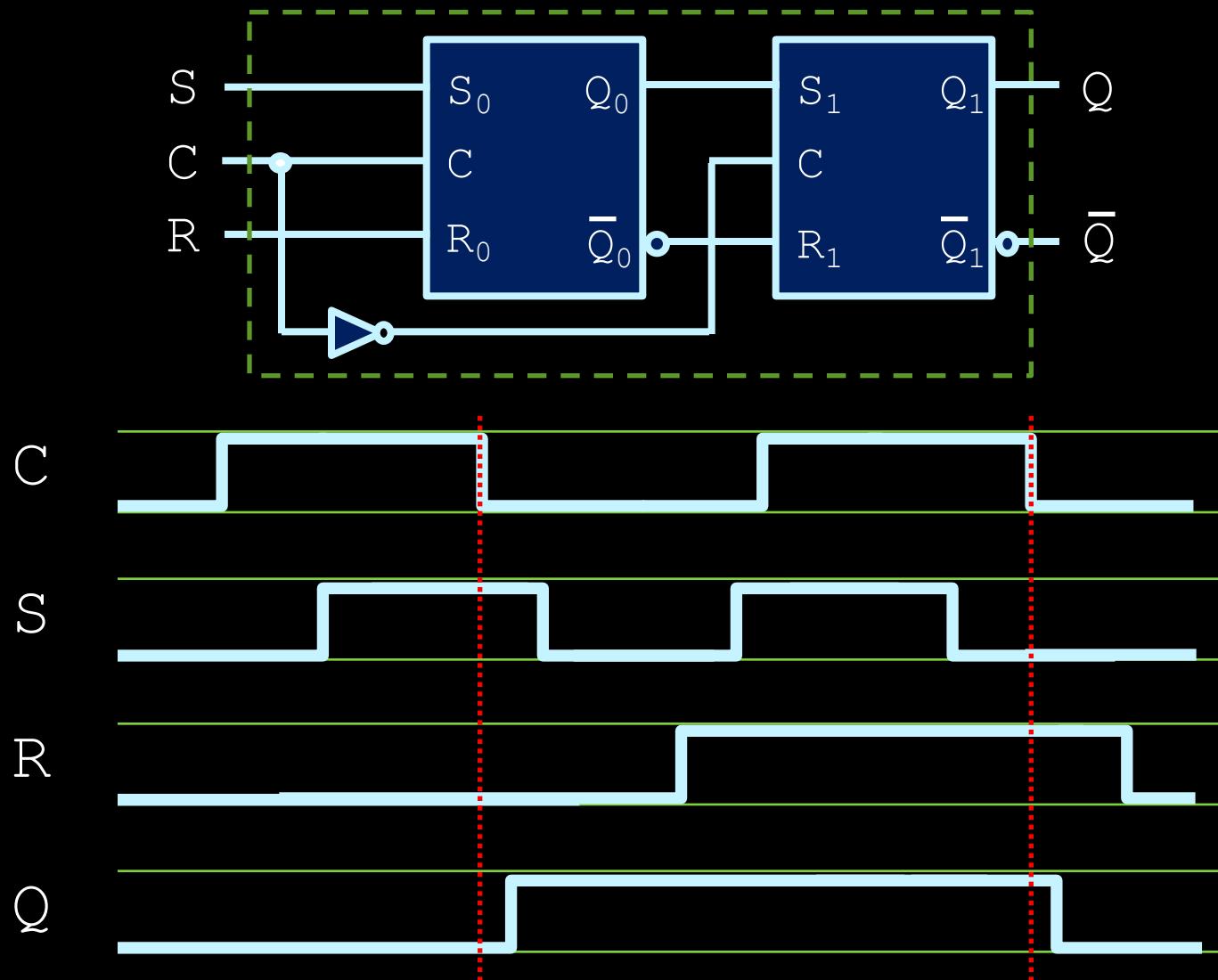
Demo: Human flesh flip-flop



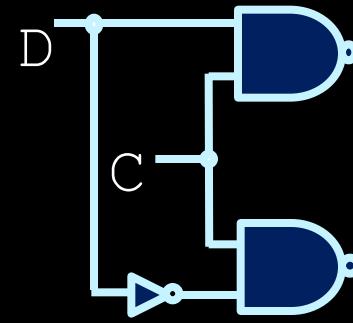
Summary:

1. For input to propagate to output, it takes each of the latches to be active once.
2. Output can only change upon “**flop**”, which is basically the falling edge of the clock signal ↘
3. At most one change per clock cycle

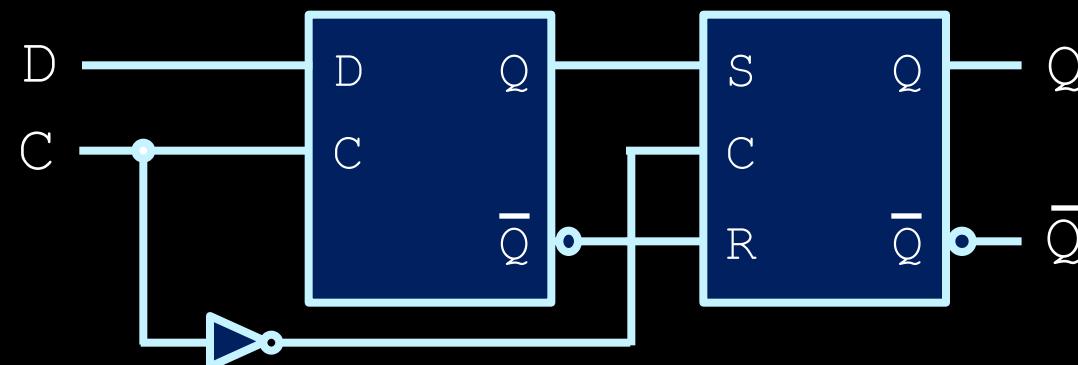
Change of Q triggered by falling clock edges



Edge-triggered D flip-flop

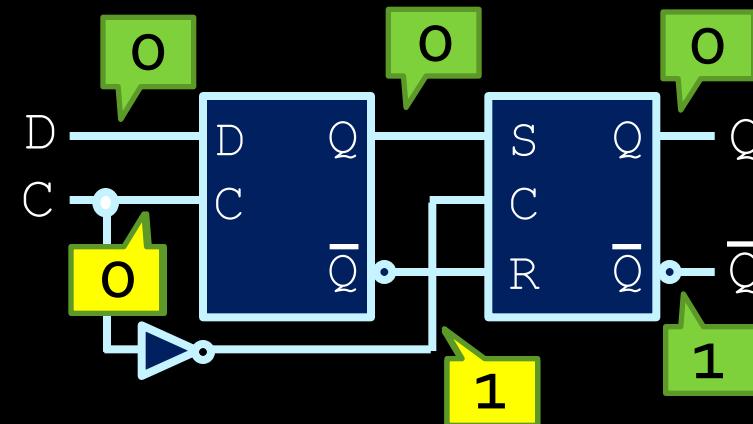
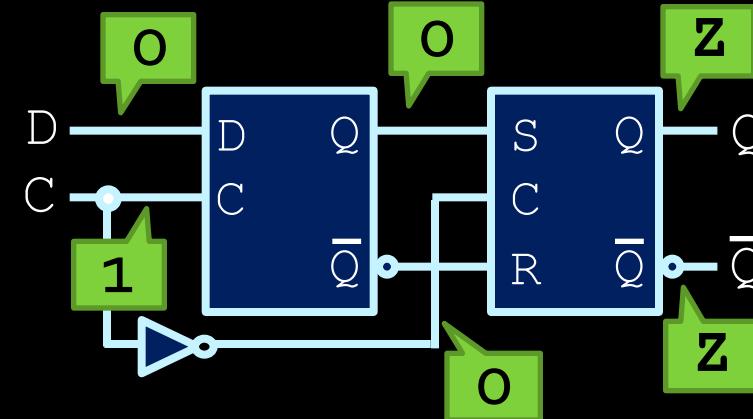


- SR flip-flops still have issues of unstable behavior (**forbidden state**)
- Solution: **D flip-flop**
 - Connect D latch to the input of a SR latch.
 - Negative-edge triggered flip-flop (like the SR)



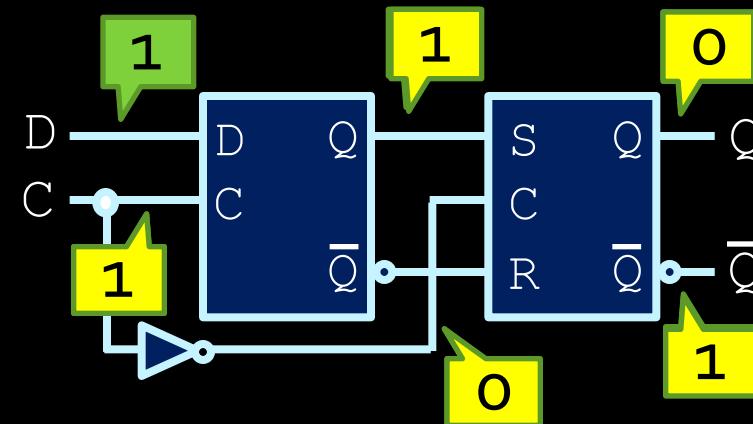
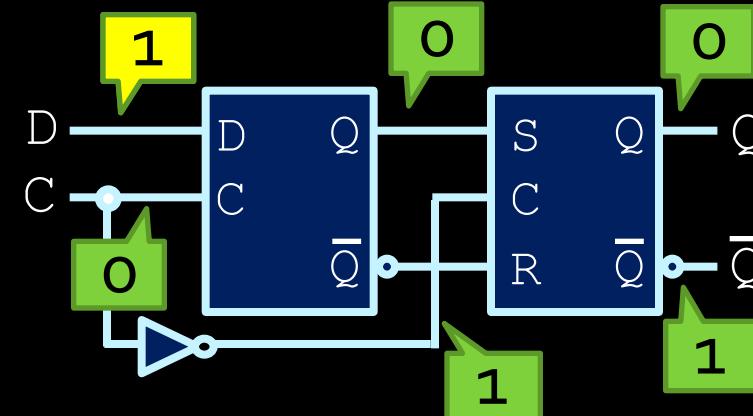
Flip-flop behaviour

- Observe the behaviour:
 - If the clock signal is high, the input to the first flip-flop is sent out to the second.
 - The second flip-flop doesn't do anything until the clock signal goes down again.
 - When it clock goes from high to low, the first flip-flop stops transmitting a signal, and the second one starts.



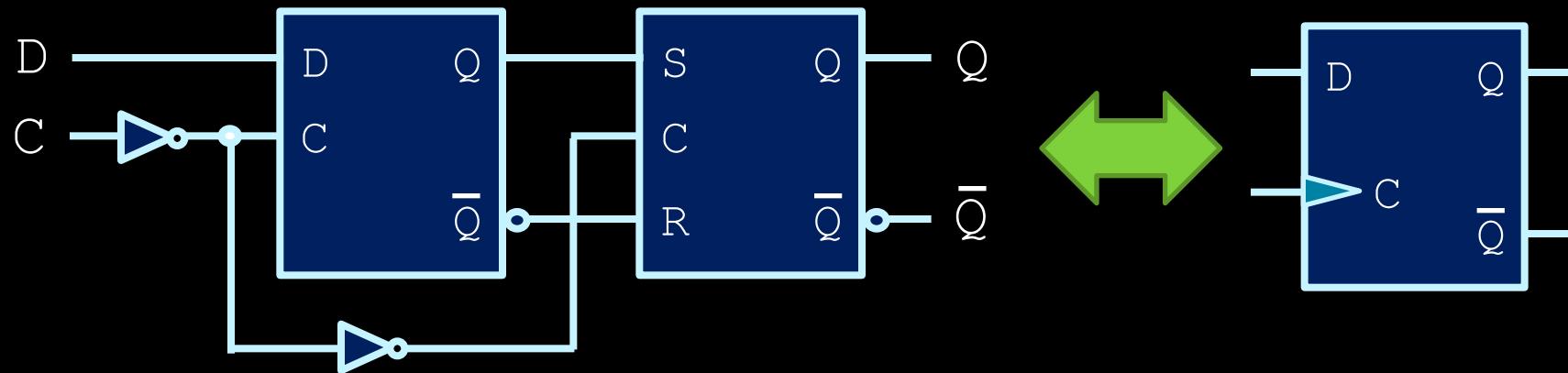
Flip-flop behaviour

- Continued from previous:
 - If the input to D changes, the change isn't transmitted to the second flip-flop until the clock goes high again.
 - Once the clock goes high, the first flip-flop starts transmitting at the same time as the second flip-flop stops.



Edge-triggered flip-flop

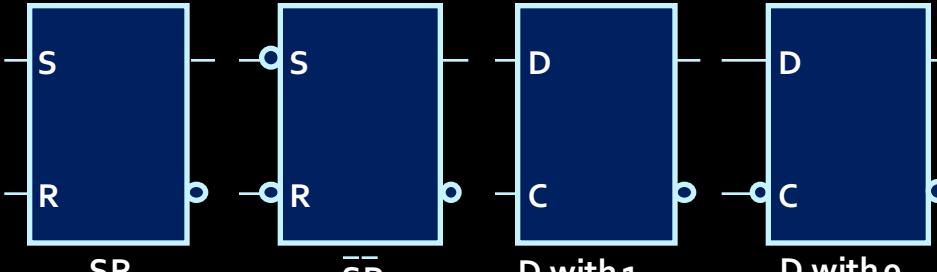
- Alternative: positive-edge triggered flip-flops



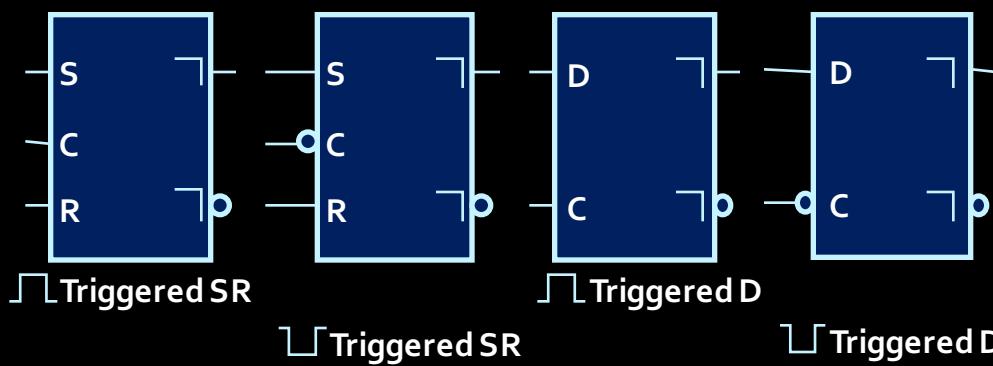
- These are the most commonly-used flip-flop circuits (and our choice for the course).

Notation

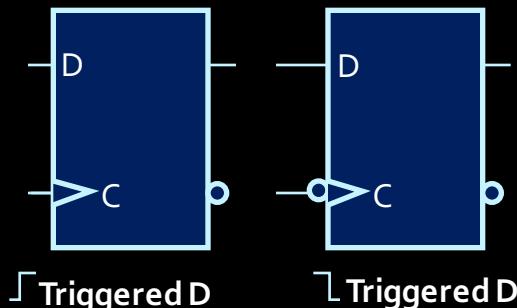
- Latches



- Master-slave flip-flops



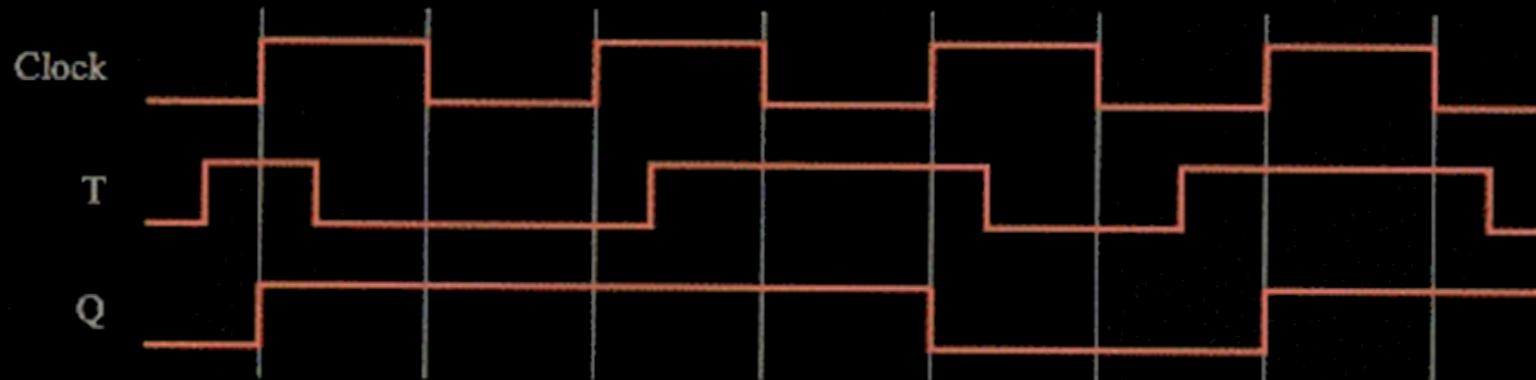
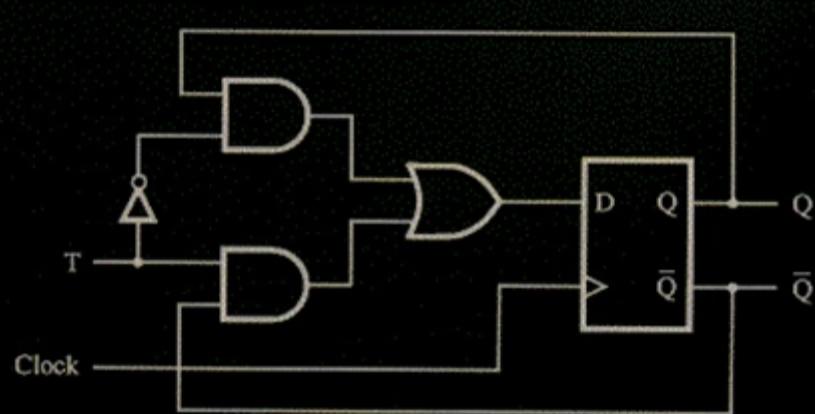
- Edge-triggered flip-flops



Note: While all these are possible, we mainly use edge-triggered D flip-flops in our designs.

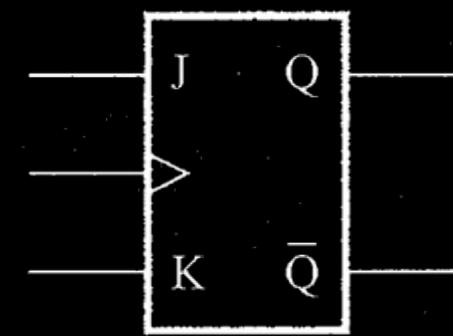
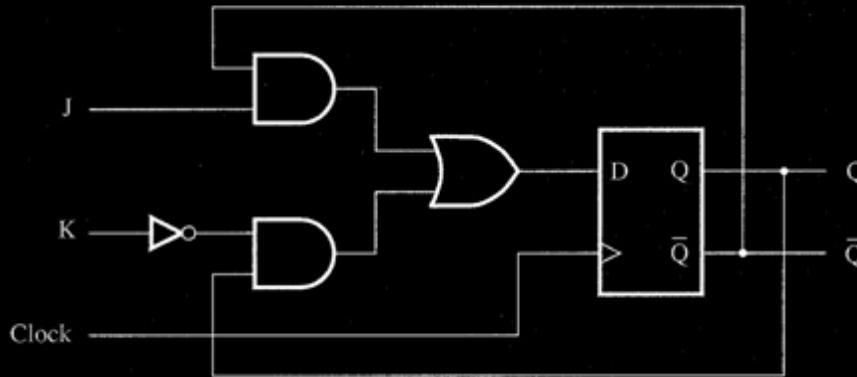
Other Flip-Flops

- The T flip-flop:
 - Like the D flip-flop, except that it toggles its value whenever the input to T is high.



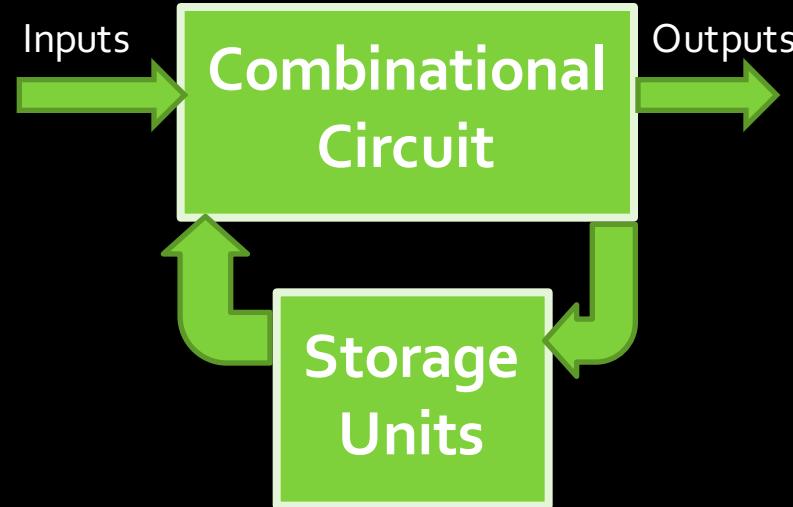
Other Flip-Flops

- The JK Flip-Flop:
 - Takes advantage of all combinations of two inputs (J & K) to produce four different behaviours:
 - if J and K are 0, maintain output.
 - if J is 0 and K is 1, set output to 0.
 - if J is 1 and K is 0, set output to 1.
 - if J and K are 1, toggle output value.



Sequential circuit design

- Similar to creating combinational circuits, with extra considerations:
 - The flip-flops now provide extra inputs to the circuit
 - Extra circuitry needs to be designed for the flip-flop inputs.
 - ...which is next week's lecture ☺

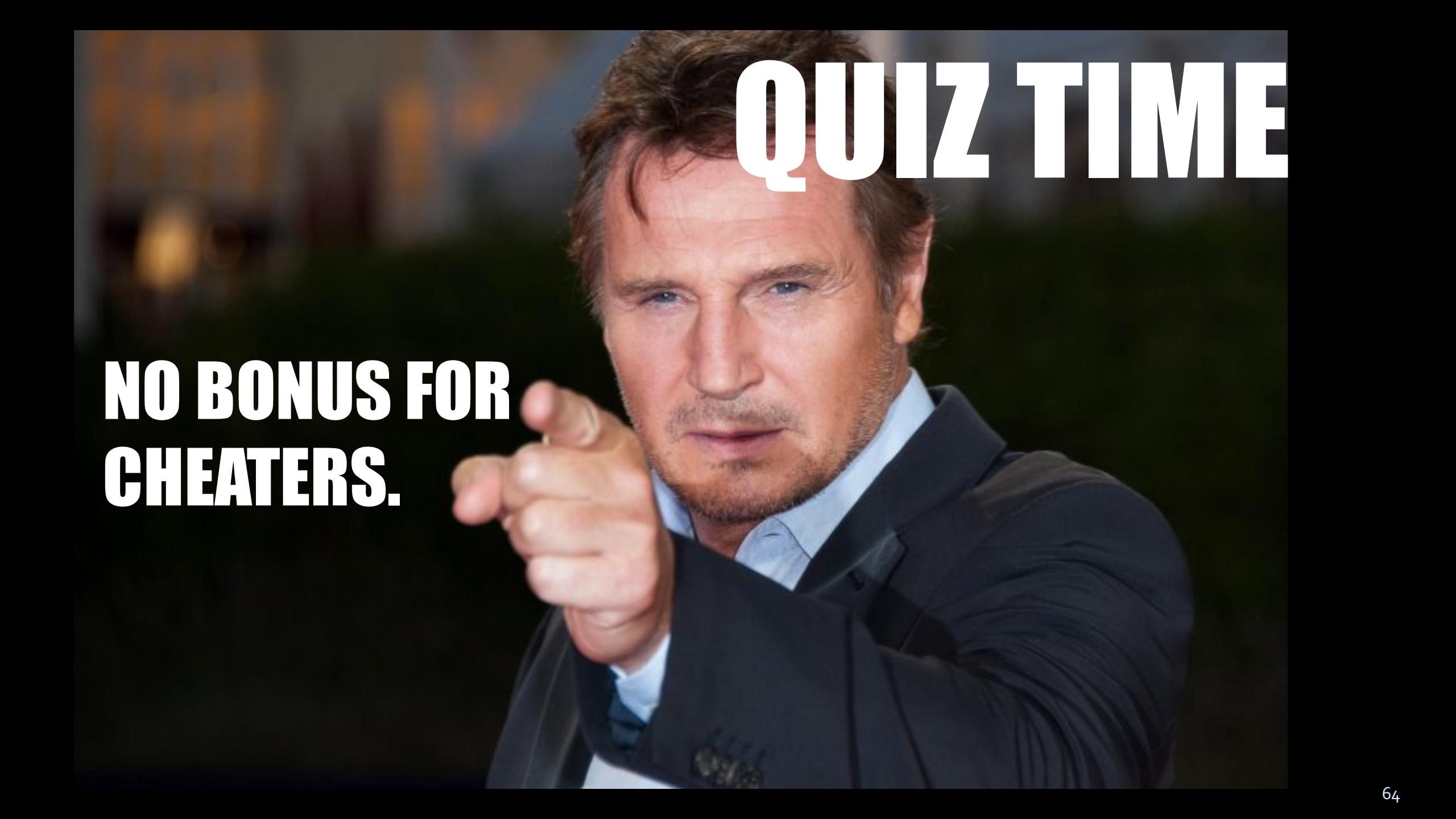


Today we learned

- Sequential circuits – circuits with memory
- Latch (SR, D)
- Flip-flop (SR, D)

Next week

- Registers, Counters
- Finite State Machines
- Sequential circuit design

A close-up photograph of Liam Neeson's face. He has light brown hair, blue eyes, and a goatee. He is wearing a dark suit jacket over a light blue shirt. His right index finger is pointing directly at the camera with a stern, commanding expression.

QUIZ TIME

**NO BONUS FOR
CHEATERS.**

Q1: Write -5 as signed 4-bit binary number.

Answer: 1011

5 is 0101, it's 2's-complement is 1011

Q2: Add two signed 4-bit integers $6 + 7$, what result (also a 4-bit signed integer) will be produced?

- A. 13
- B. -2
- C. -3
- D. None of above

Answer: C

6 is 0110, 7 is 0111, adding them up we get 1101, which is -3 because 1101's 2's-complement is 0011 (3).

This situation is called an **overflow**, since the expected result 13 is exceeding the range of value that a 4-bit signed integer can represent (-8 ~ 7).

Try this C code

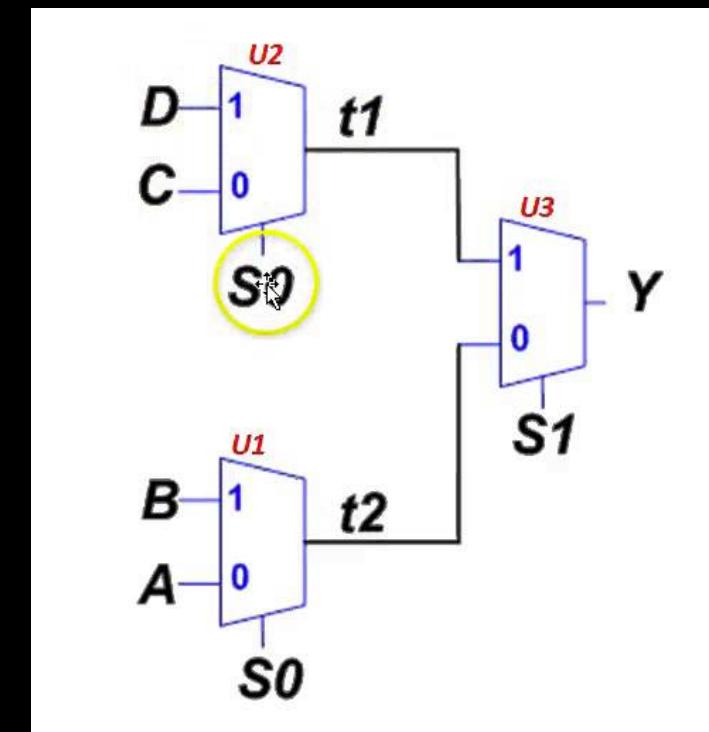
```
#include <stdio.h>

int main()
{
    /* char is 8-bit integer */
    signed char a = 100;
    signed char b = 120;
    signed char s = a + b;
    printf("%d\n", s);
}
```

Q3: What is output Y when $S0 = 1$
And $S1 = 0$? Answer A, B, C or D.

Answer: B

$S1 = 0$ selects either A or B,
 $S0 = 1$ selects B rather than A



CSC258 Winter 2016

Lecture 5

Q2: Add two signed 4-bit integers $6 + 7$, what result (also a 4-bit signed integer) will be produced?

- A. 13
- B. -2
- C. -3
- D. None of above

Answer: C

6 is 0110, 7 is 0111, adding them up we get 1101, which is -3 because 1101's 2's-complement is 0011 (3).

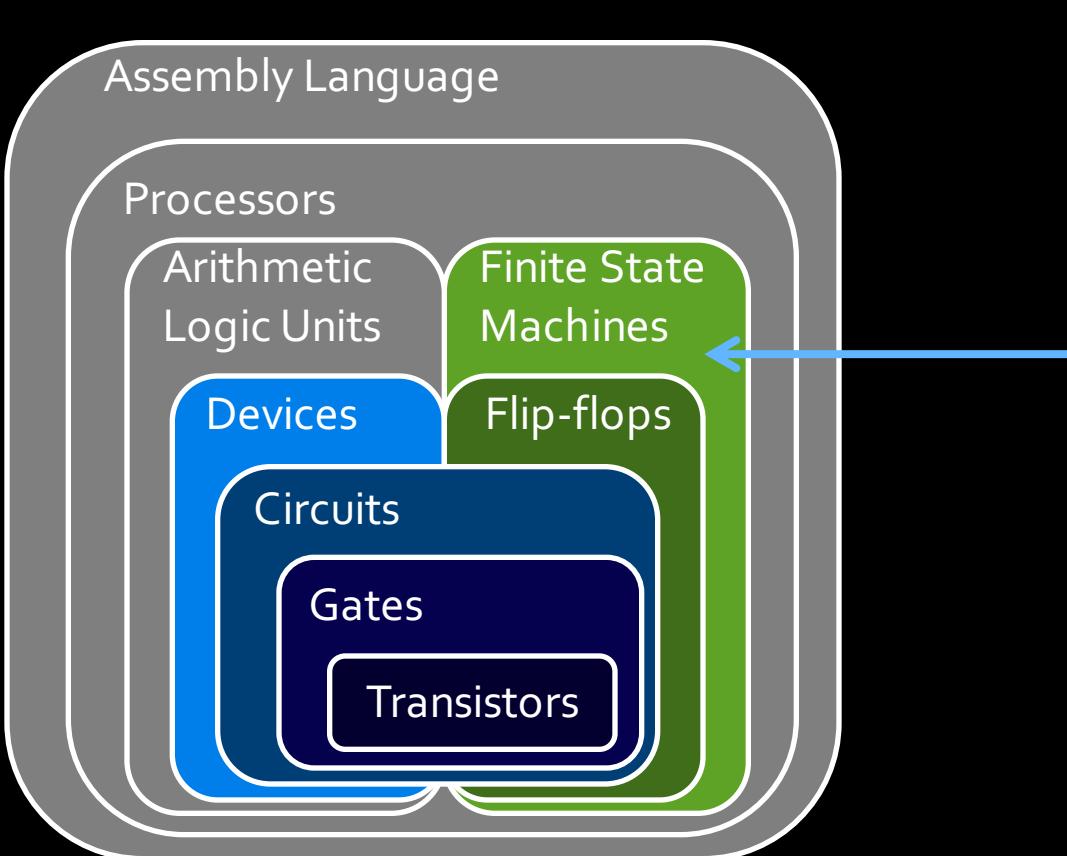
This situation is called an **overflow**, since the expected result 13 is exceeding the range of value that a 4-bit signed integer can represent (-8 ~ 7).

Try this C code

```
#include <stdio.h>

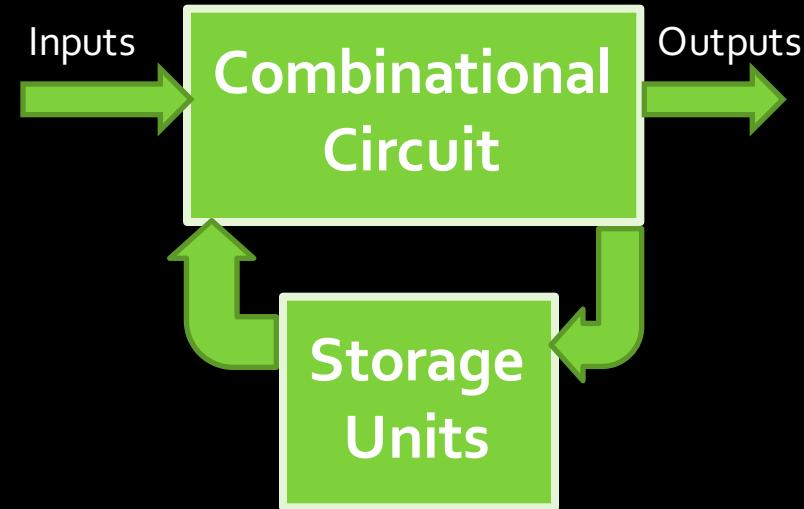
int main()
{
    /* char is 8-bit integer */
    signed char a = 100;
    signed char b = 120;
    signed char s = a + b;
    printf("%d\n", s);
}
```

We are here



Circuits using flip-flops

- Now that we know about flip-flops and what they do, how do we use them in circuit design?
- What's the benefit in using flip-flops in a circuit at all?



We can design much cooler circuits using flip-flops.

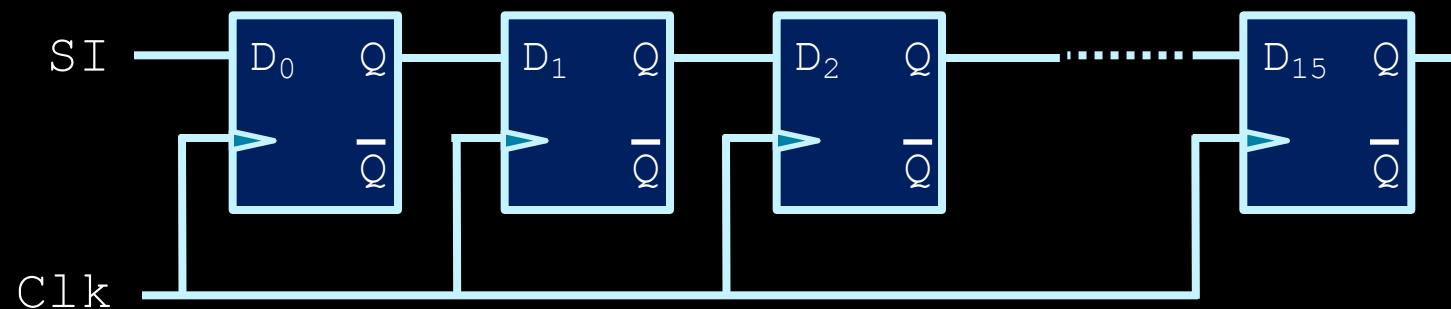
Example #1: Registers

For storing values



Shift registers

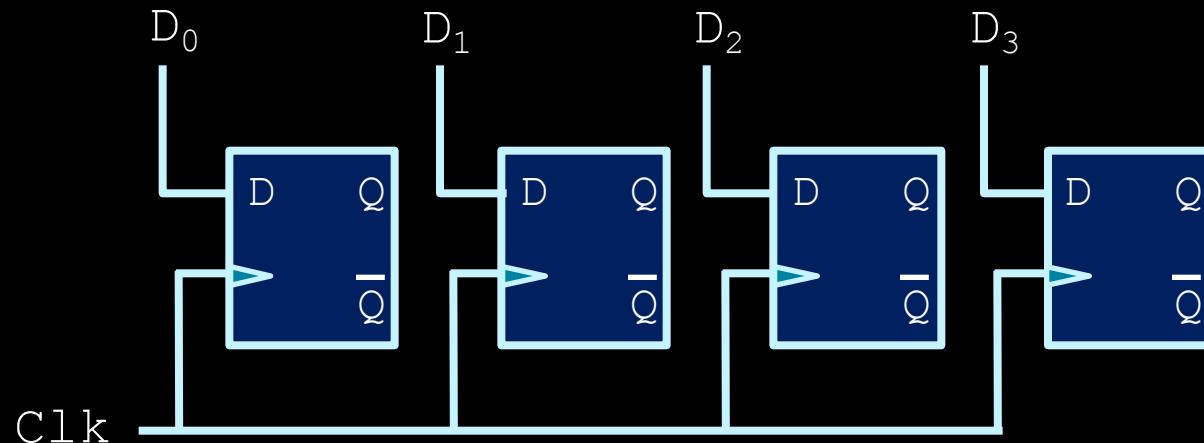
- A series of D flip-flops can store a multi-bit value (such as a 16-bit integer, for example).



- Data can be shifted into this register **one bit at a time**, over 16 clock cycles.
 - Known as a **shift register**.

Load registers

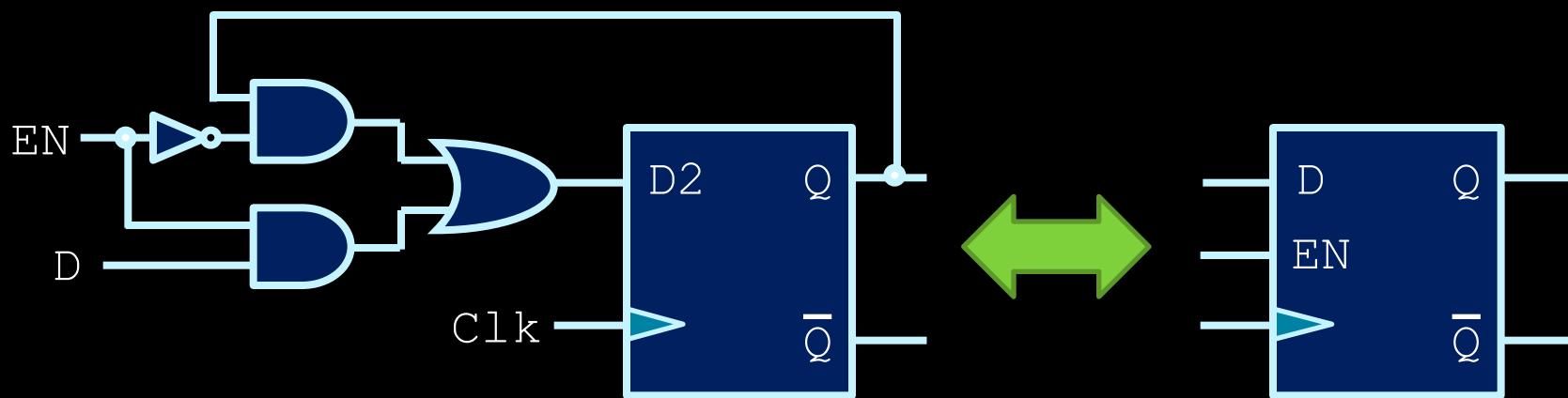
- One can also load a register's values all at once, by feeding signals into each flip-flop:
 - In this example: a 4-bit **load register**.



One clock pulse, 4 bits stored.

Load registers

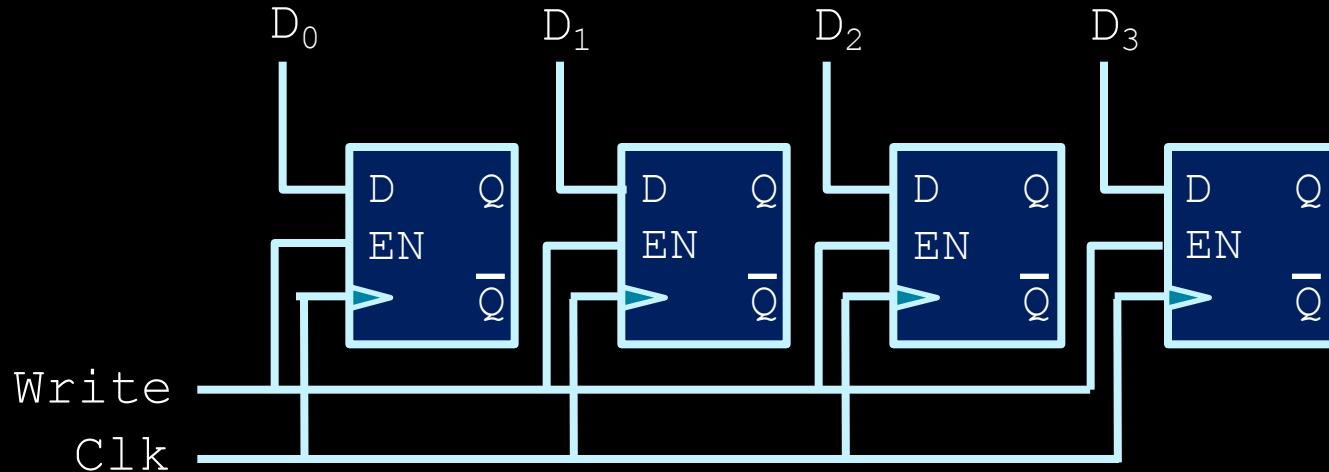
- To control when this register is allowed to load its values, we introduce the D flip-flop with enable:



When $EN = 1$, D_2 is whatever D is, load D

When $EN = 0$, D_2 is whatever Q is, maintain Q

Load registers



- Implementing the register with these special D flip-flops will now maintain values in the register until overwritten by setting EN high.

In computer architecture, registers are the CPU's **most local storages** (30+ of them on-chip), i.e., they are the lowest level of the **memory hierarchy**. They are the memory units that the CPU directly interact with.

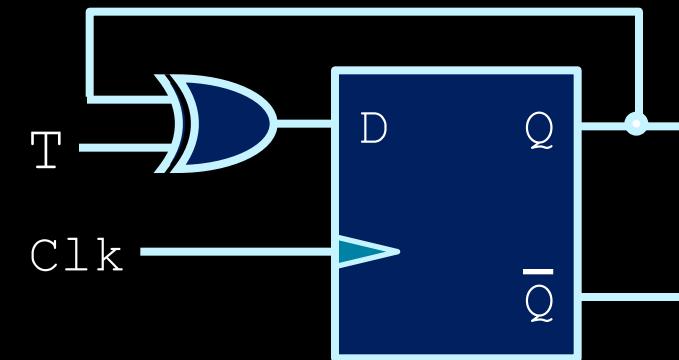
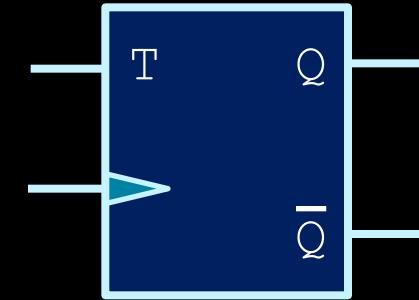
Higher level of memory include cache, RAM, hard disc, etc.

Example #2: Counters



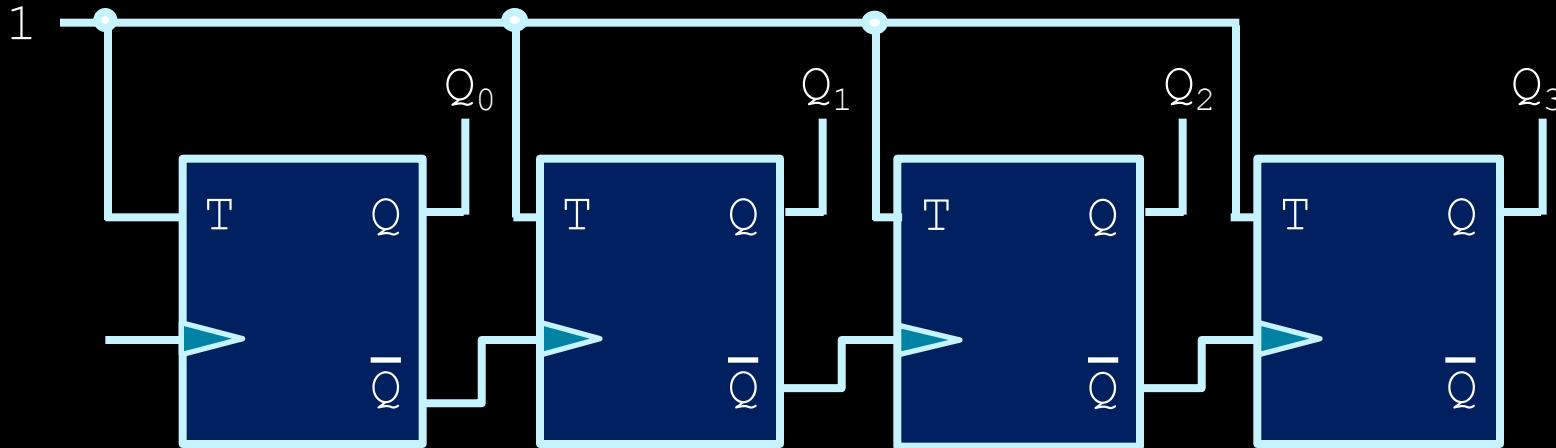
Counters

- Consider the T flip-flop:
 - Output is inverted when input T is high.
- What happens when a series of T flip-flops are connected together in sequence?
- More interesting:
 - Connect the *output* of one flip-flop to the *clock* input of the next!



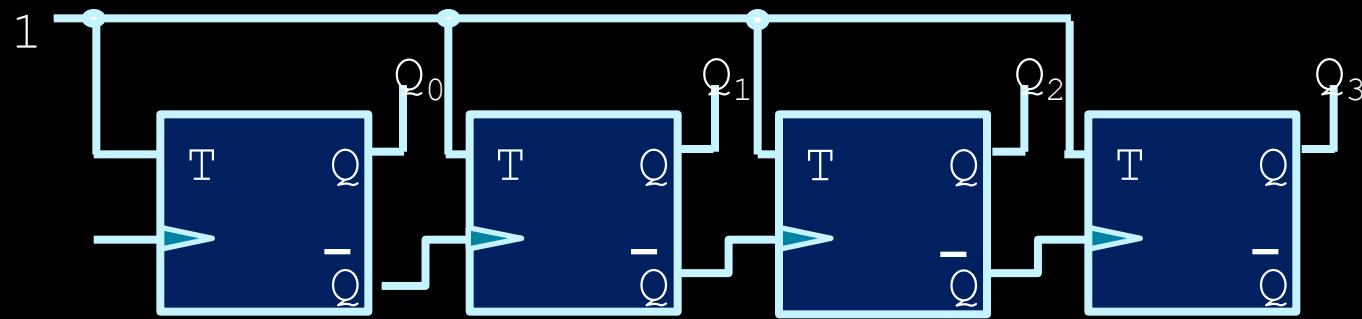
Counters

Asynchronous means the four outputs do not change upon the same clock signal.



- This is a 4-bit **ripple counter**, which is an example of an **asynchronous** circuit.
 - Timing isn't quite synchronized with the rising clock pulse.
 - Cheap to implement, but unreliable for timing.

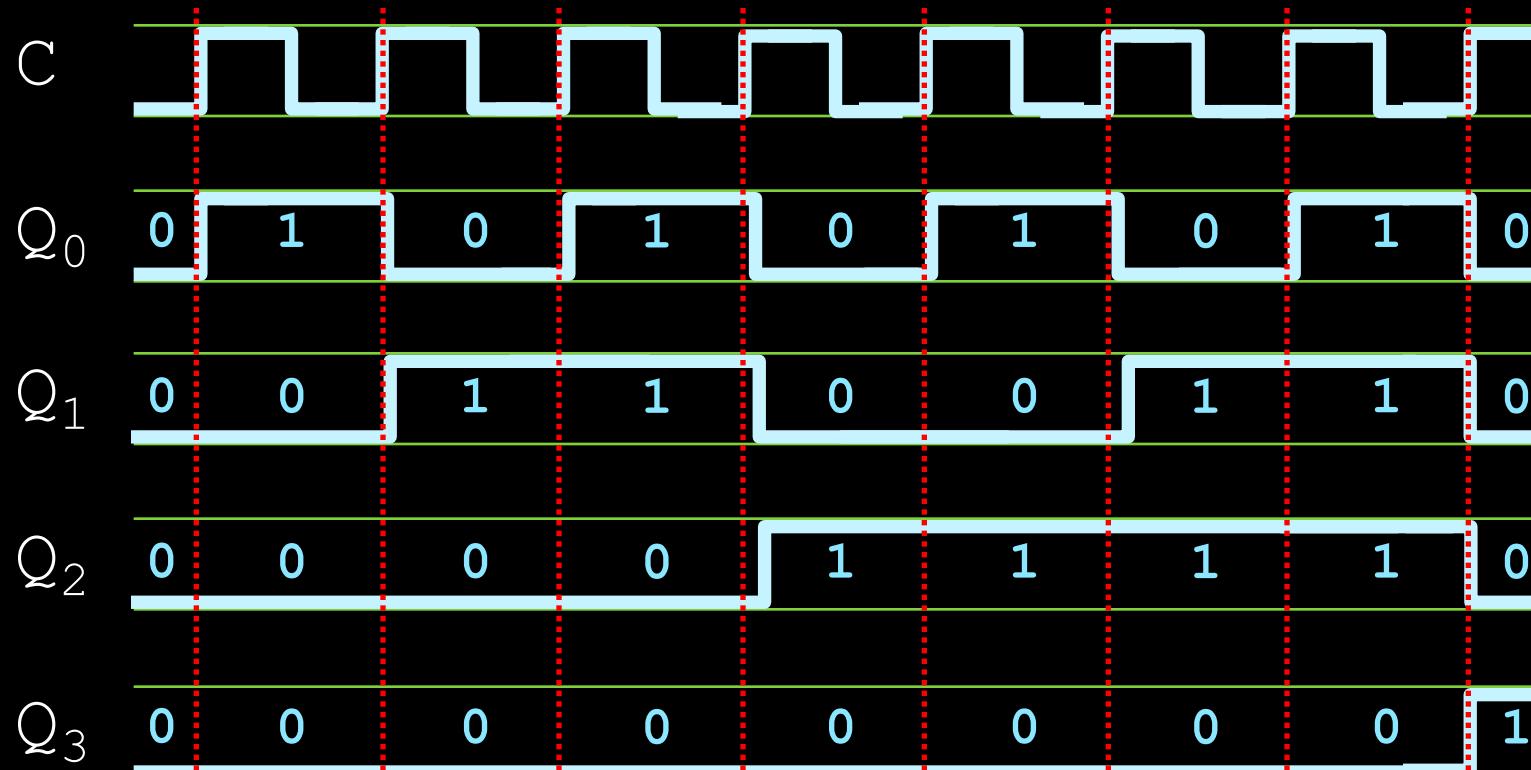
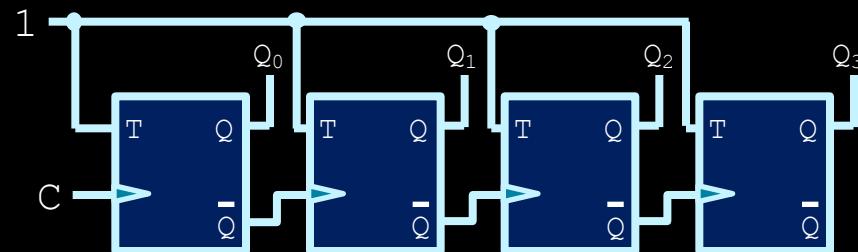
Demo: asynchronous human counter



1. Need 4 volunteers, Q₀, Q₁, Q₂, Q₃
2. Q₀ toggles when receives clock signal (tap on shoulder)
3. Q₁ toggles when Q₀ goes from 1 to 0
4. Q₂ toggles when Q₁ goes from 1 to 0
5. Q₃ toggles when Q₂ goes from 1 to 0
6. Audience read the number Q₃Q₂Q₁Q₀

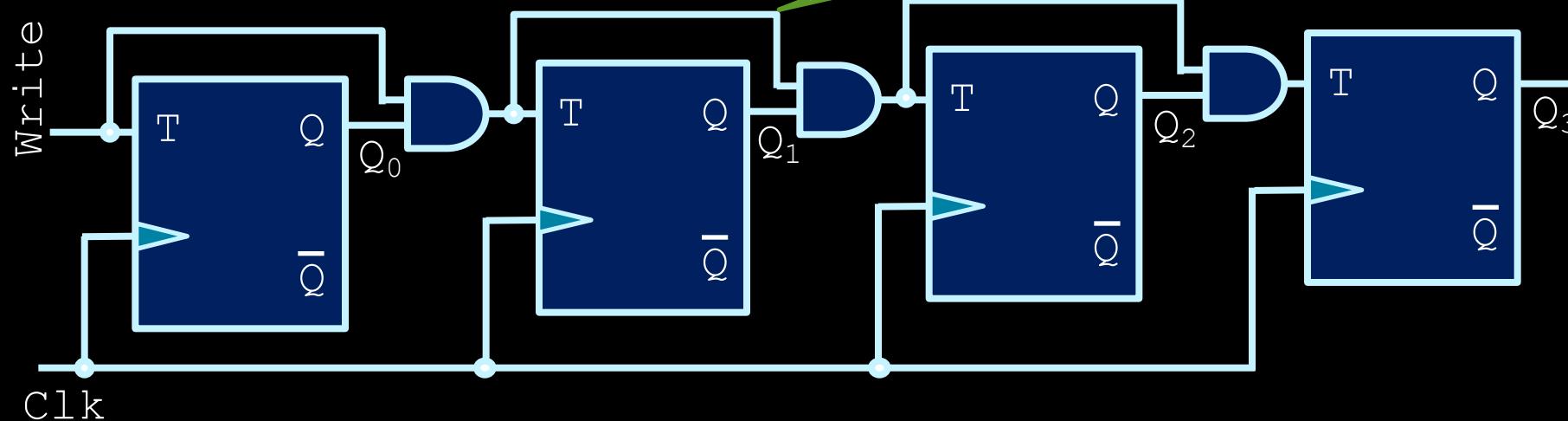
Counters

- Timing diagram



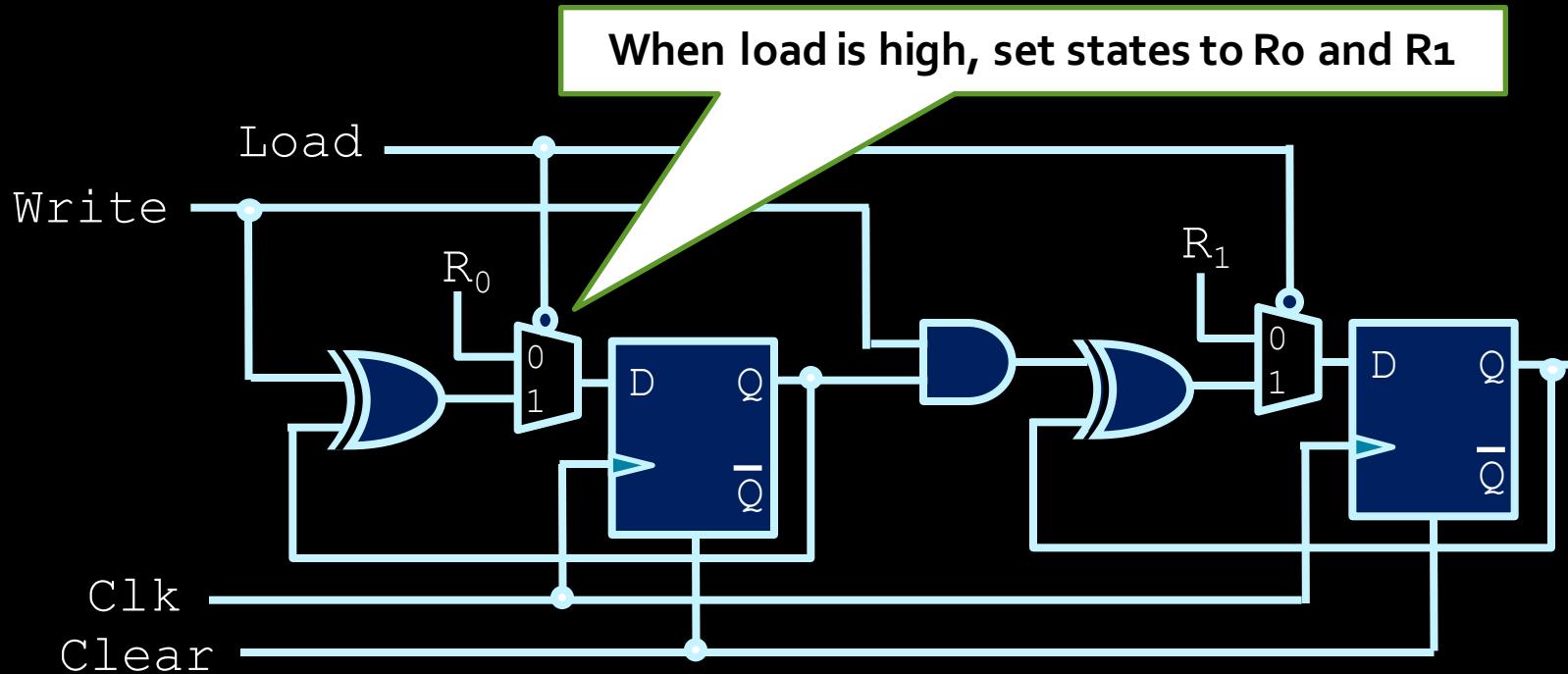
Synchronous Counter

Toggle only when the lower bit is 1
and is toggling to 0 (produce a carry)



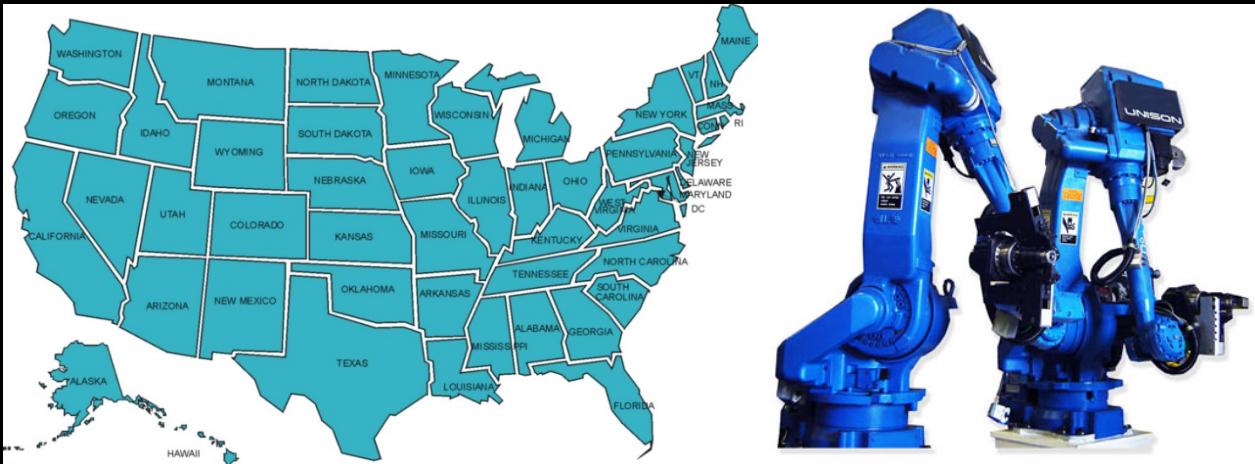
- This is a **synchronous** counter, because all output Q's change upon the same clock edge.

Counter with parallel load



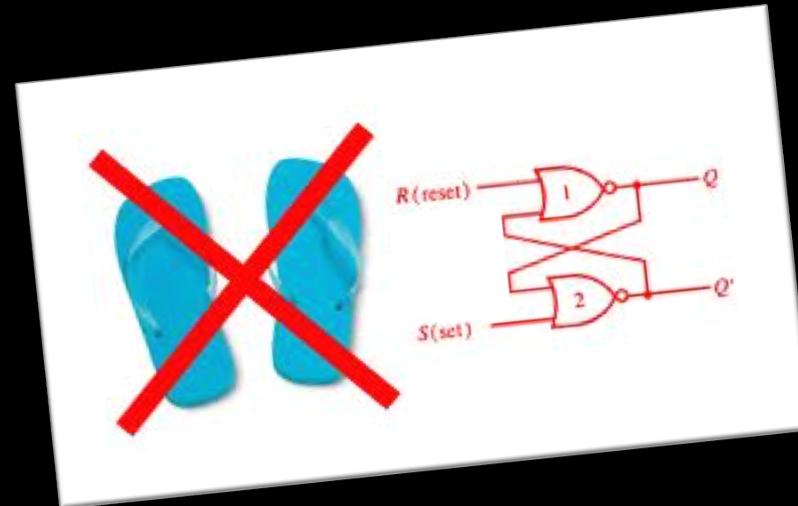
- Counters are often implemented with a **parallel load** and **clear** inputs.
 - Can set the counter to whatever value needed.

State Machines



Designing with flip-flops

- Counters and registers are examples of how flip-flops can implement useful circuits that store values.
 - How do you design these circuits?
 - What would you design with these circuits?

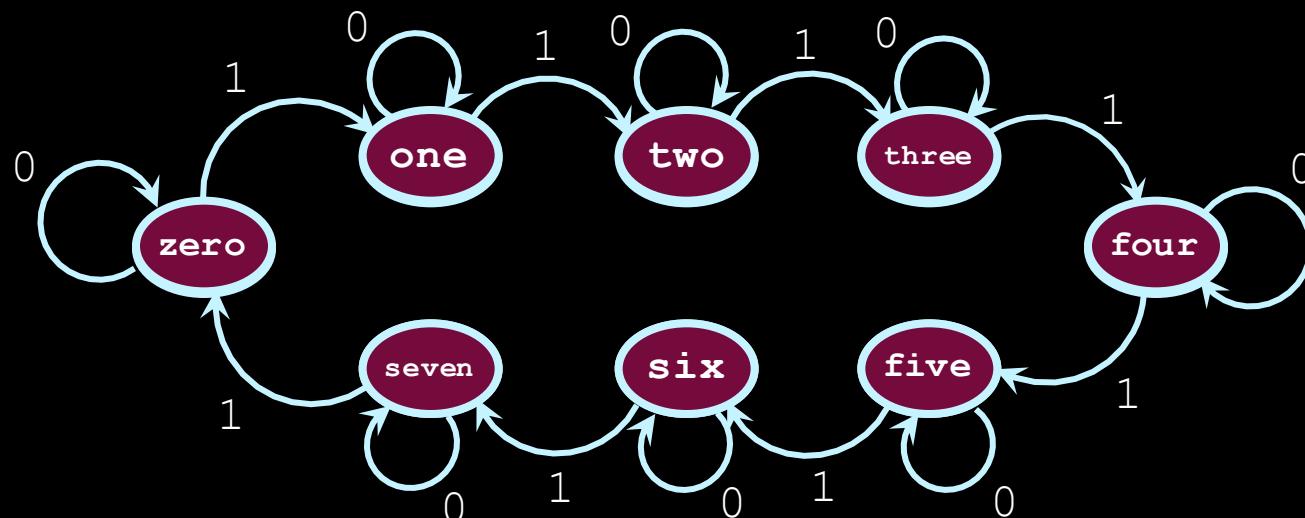


Designing with flip-flops

- Sequential circuits are the basis for memory, instruction processing, and any other operation that requires the circuit to remember **past data values**.
- These past data values are also called the **states** of the circuit.
- Need to describe the relation between the **current state** and the **next state**: use **combinational circuits**

State example: Counters

- With counters, each state is the current number that is stored in the counter.



Each state does not need to correspond to a binary number, they are just different states

- On each clock tick, the circuit **transitions** from one state to the next, based on the inputs.

State Tables

- State tables help to illustrate how the states of the circuit change with various input values.
 - Transitions are understood to take place on the clock ticks.

State	Write	State
zero	0	zero
zero	1	one
one	0	one
one	1	two
two	0	two
two	1	three
three	0	three
three	1	four
four	0	four
four	1	five
five	0	five
five	1	six
six	0	six
six	1	seven
seven	0	seven
seven	1	zero

State Tables

- Same table as on the previous slide, but with the actual flip-flop values instead of state labels.
- Note: Flip-flop values are both inputs and outputs of the circuit here.

F_1	F_2	F_3	Write	F_1	F_2	F_3
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	1	0
0	1	0	1	0	1	1
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	1	0	0
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	1	0
1	1	0	0	1	1	0
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	0	0	0

and this brings us to...

Finite State Machines

As seen in other courses...

- You may have seen finite state machines before, but in a different context.
 - Used mainly to describe the grammars of a language, or to model sequence data.
- In CSC258, finite state machines are models for an actual circuit design.
 - The states represent internal states of the circuit, which are stored in the flip-flop values.

Finite State Machines (FSMs)

- From theory courses...
 - A **Finite State Machine** is an abstract model that captures the operation of a sequential circuit.
- A FSM is defined (in general) by:
 - A finite set of **states**,
 - A finite set of **transitions** between states, **triggered by inputs** to the state machine,
 - Output values that are associated with each state or each transition (depending on the machine),
 - Start and end states for the state machine.

Design procedures comparison (roughly)

Combinational circuits

1. Desired behaviour
2. Truth table
3. Logic expression
4. Circuit

Sequential circuits

1. Desired behaviour
(cooler behaviour)
2. Finite state machine
3. Circuit with flip-flops

Example #1: Tickle Me Elmo

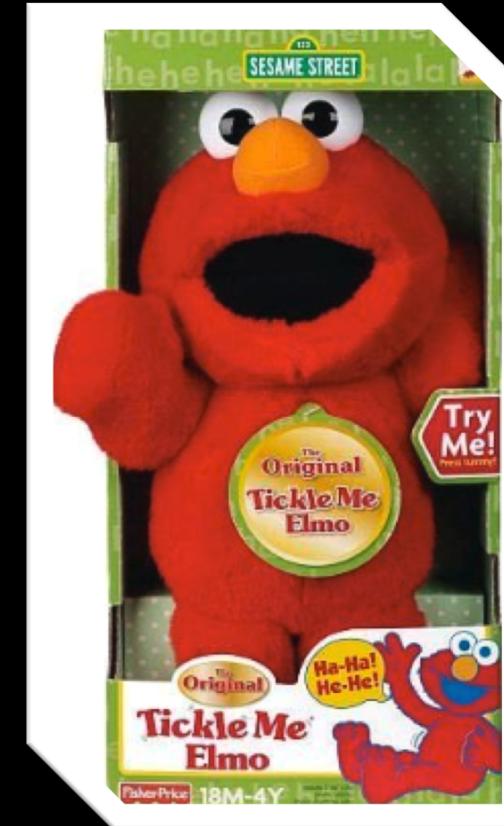
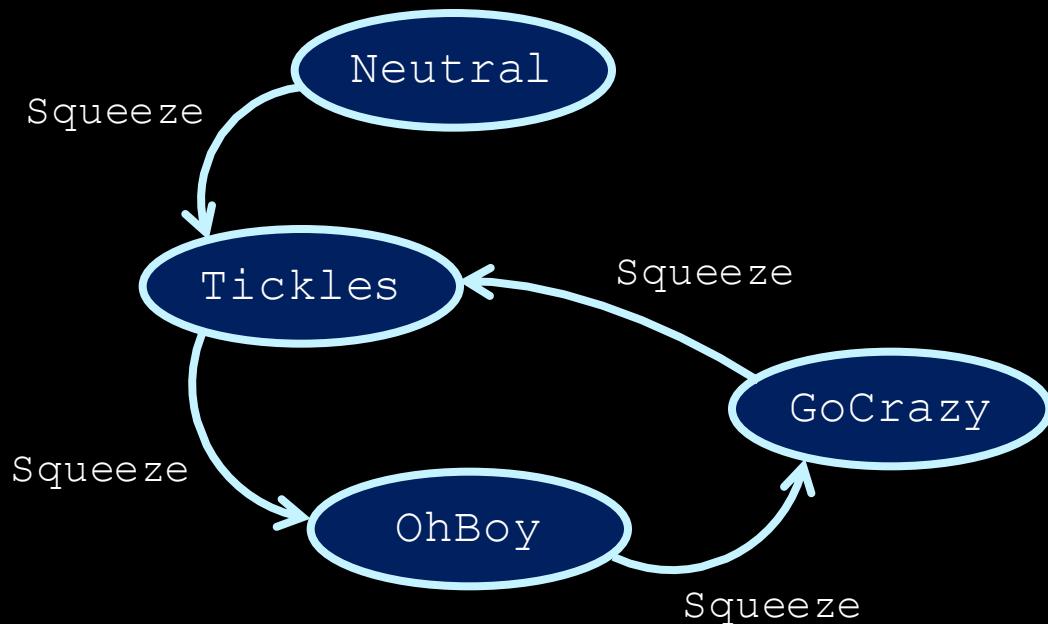
- Remember how the Tickle Me Elmo works!



Example #1: Tickle Me Elmo

- Toy reacts differently each time it is squeezed:
 - First squeeze → "Ha ha ha...that tickles."
 - Second squeeze → "Ha ha ha...oh boy."
 - Third squeeze → "HA HA HA HA...", go crazy
- Questions to ask:
 - What are the inputs?
 - What are the states of this machine?
 - How do you change from one state to the next?

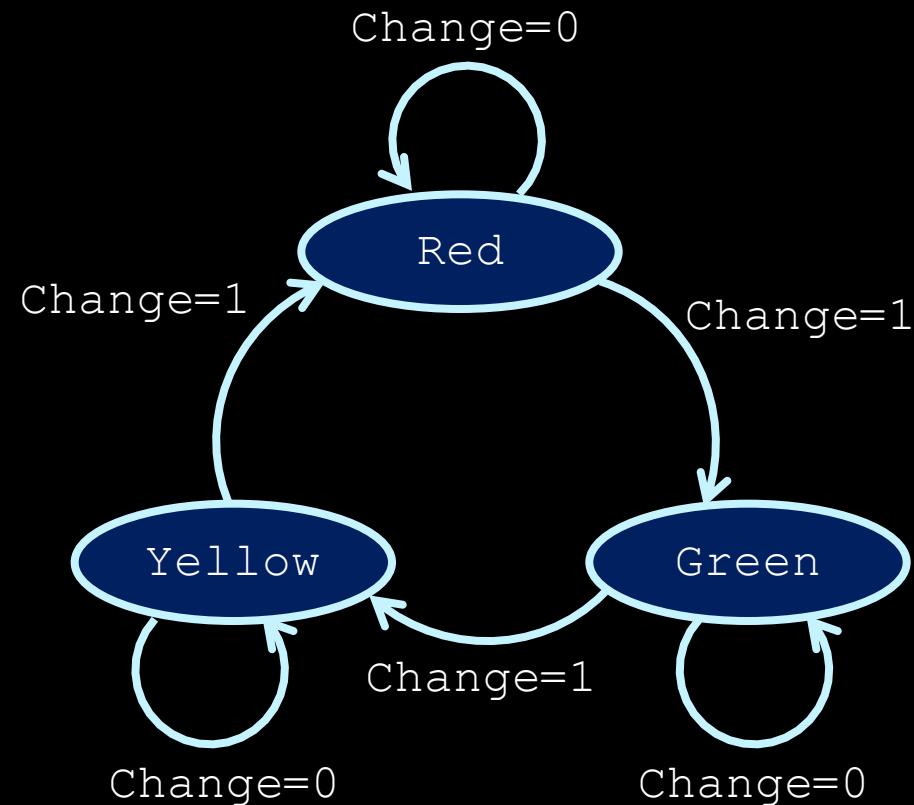
Example #1: Tickle Me Elmo



More elaborate FSMs

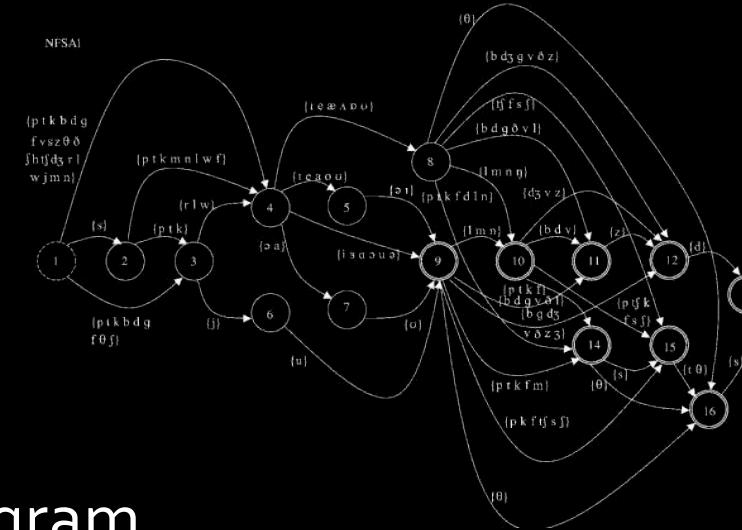
- Usually our FSM has **more than one input**, and will trigger a transition based on certain input values but not others.
- Also might have input values that don't cause a transition, but keep the circuit in the same state (**transitioning to itself**).

Example #2: Traffic Light



FSM design

- Design steps for FSM:
 1. Draw **state diagram**
 2. Derive **state table** from state diagram
 3. Assign **flip-flop configuration** to each state
 - Number of flip-flops needed is:
 - $\lceil \log_2(\# \text{ of states}) \rceil$
 4. Redraw **state table** with flip-flop values
 5. Derive **combinational circuit** for output and for each flip-flop input.



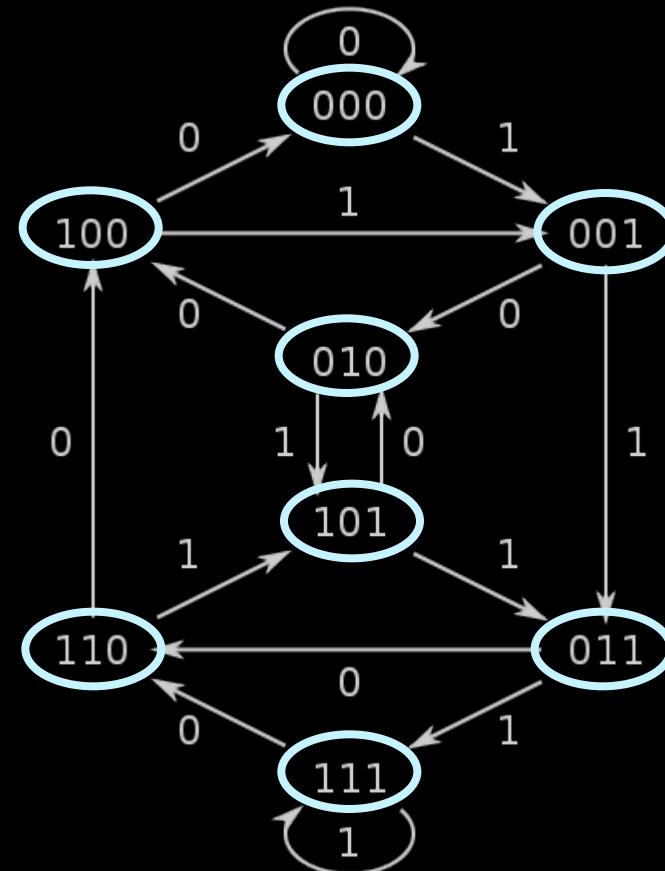
Example: Sequence Recognizer

- Recognize a sequence of input values, and raise a signal if that input has been seen.
- Example: Three high values in a row
 - Detect that the input has been high for three rising clock edges.
 - Assumes a single input X and a single output Z .

What are the states?

Step 1: State diagram

- In this case, the states are labeled with the **most recent three input values**.
- Transitions between states are indicated by the values on the transition arrows.



Step 2: State table

- Make sure that the state table lists **all the states** in the state diagram, and **all the possible inputs** that can occur at that state.

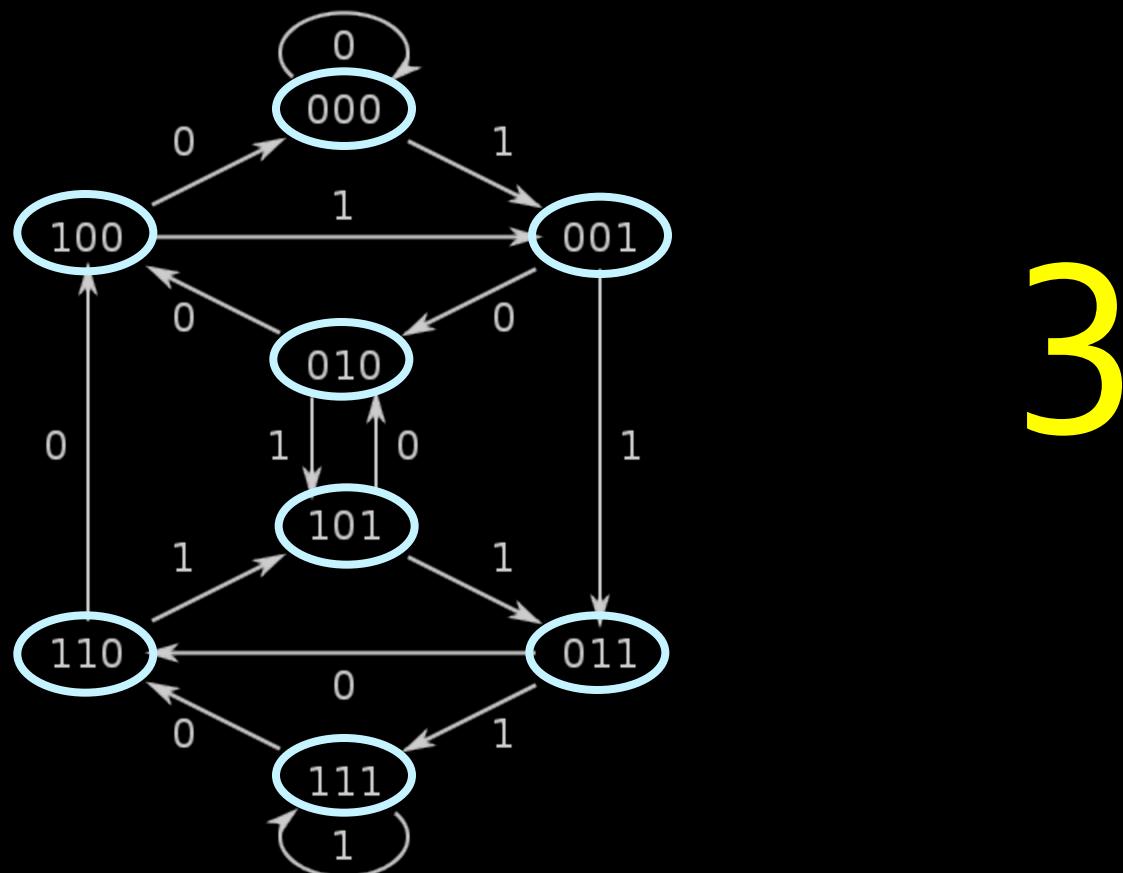
Previous State	EN	Next State
000	0	000
000	1	001
001	0	010
001	1	011
010	0	100
010	1	101
011	0	110
011	1	111
100	0	000
100	1	001
101	0	010
101	1	011
110	0	100
110	1	101
111	0	110
111	1	111

Step 3: Assign flip-flops

- Assign flip-flops for storing states.
- A single flip-flop can store two values (0 and 1), and thus two states.
- How many states can be stored with each additional flip-flop?
 - One flip-flop → 2 states
 - Two flip-flops → 4 states
 - Three flip-flops → 8 states
 - ...
 - Eight flip-flops? → $2^8 = 256$ states

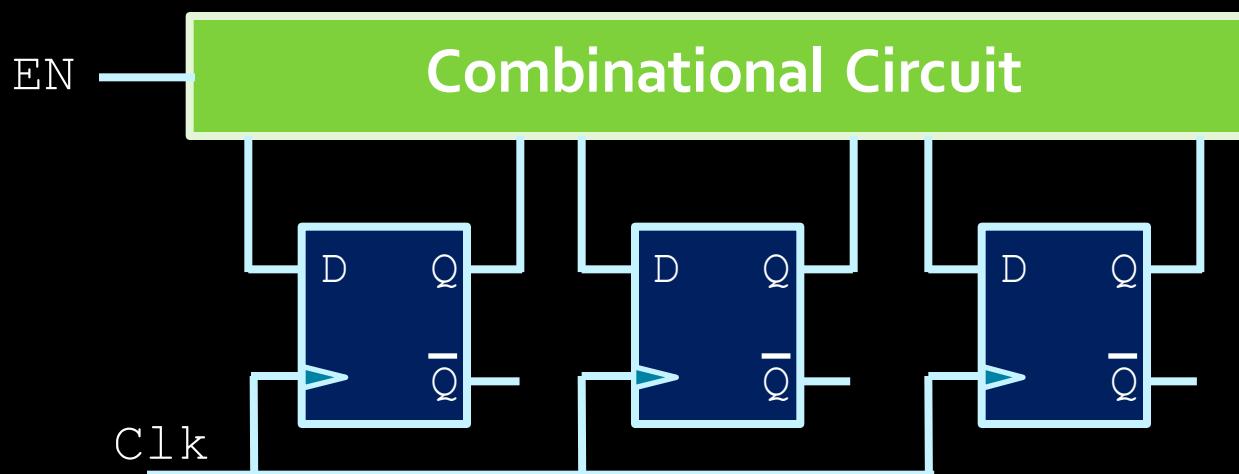
**n states need:
[$\log_2 n$] flip-flops**

How many flip-flops for this one?



Step 3: Assign flip-flops

- In this case, we need to store 8 states.
 - 8 states = 3 flip-flops ($3 = \log_2 8$)
- For now, assign a flip-flop to each digit of the state names in the FSM & state table.



Step 4: State table

- Mapping states to flip-flop values
- This is **NOT** the only way of mapping from state to flip flop values, in fact it is not even a good way, as we will see later.

Prev. State	EN	Next State
000	0	000
000	1	001
001	0	001
001	1	010
010	0	010
010	1	011
011	0	011
011	1	100
100	0	100
100	1	101
101	0	101
101	1	110
110	0	110
110	1	111
111	0	111
111	1	000

Step 4: State table

- Mapping states to flip-flop values
- This is **NOT** the only way of mapping from state to flip flop values, in fact it is not even a good way, as we will see later.

F₂	F₁	F₀	EN	F₂	F₁	F₀
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	1	0	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1
1	1	0	0	1	0	0
1	1	0	1	1	0	1
1	1	1	0	1	1	0
1	1	1	1	1	1	1

Step 5: Circuit design

- Karnaugh map for F_2 :

	$\bar{F}_0 \cdot \bar{E}N$	$\bar{F}_0 \cdot E\bar{N}$	$F_0 \cdot E\bar{N}$	$F_0 \cdot \bar{E}N$
$\bar{F}_2 \cdot \bar{F}_1$	0	0	0	0
$\bar{F}_2 \cdot F_1$	1	1	1	1
$F_2 \cdot F_1$	1	1	1	1
$F_2 \cdot \bar{F}_1$	0	0	0	0

Next state

$$F_2 = F_1$$

Current state

Step 5: Circuit design

- Karnaugh map for F_1 :

	$\bar{F}_0 \cdot \bar{E}N$	$\bar{F}_0 \cdot E\bar{N}$	$F_0 \cdot E\bar{N}$	$F_0 \cdot \bar{E}N$
$\bar{F}_2 \cdot \bar{F}_1$	0	0	1	1
$\bar{F}_2 \cdot F_1$	0	0	1	1
$F_2 \cdot \bar{F}_1$	0	0	1	1
$F_2 \cdot F_1$	0	0	1	1

Next state

$$F_1 = F_0$$

Current state

Step 5: Circuit design

- Karnaugh map for F_0 :

	$\bar{F}_0 \cdot \bar{EN}$	$\bar{F}_0 \cdot EN$	$F_0 \cdot EN$	$F_0 \cdot \bar{EN}$
$\bar{F}_2 \cdot \bar{F}_1$	0	1 1	0	
$\bar{F}_2 \cdot F_1$	0	1 1	0	
$F_2 \cdot \bar{F}_1$	0	1 1	0	
$F_2 \cdot F_1$	0	1 1	0	

Next state

$$F_0 = EN$$

Current state

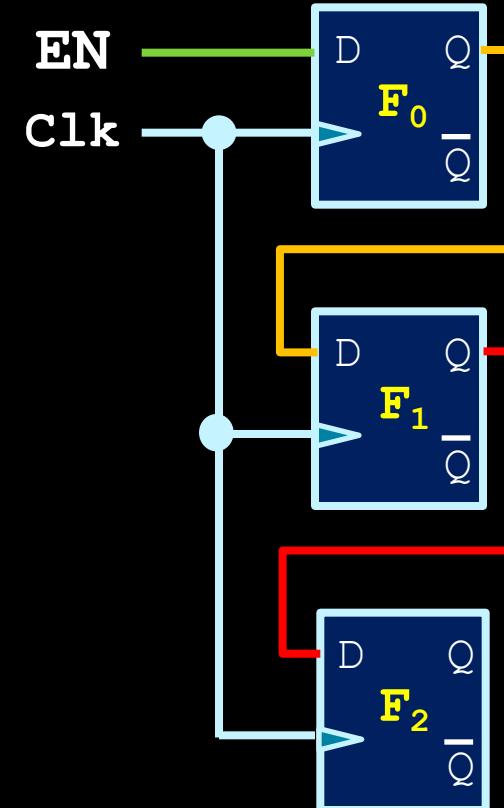
$$F_2 = F_1$$

$$F_1 = F_0$$

$$F_0 = EN$$

Step 5: Circuit design

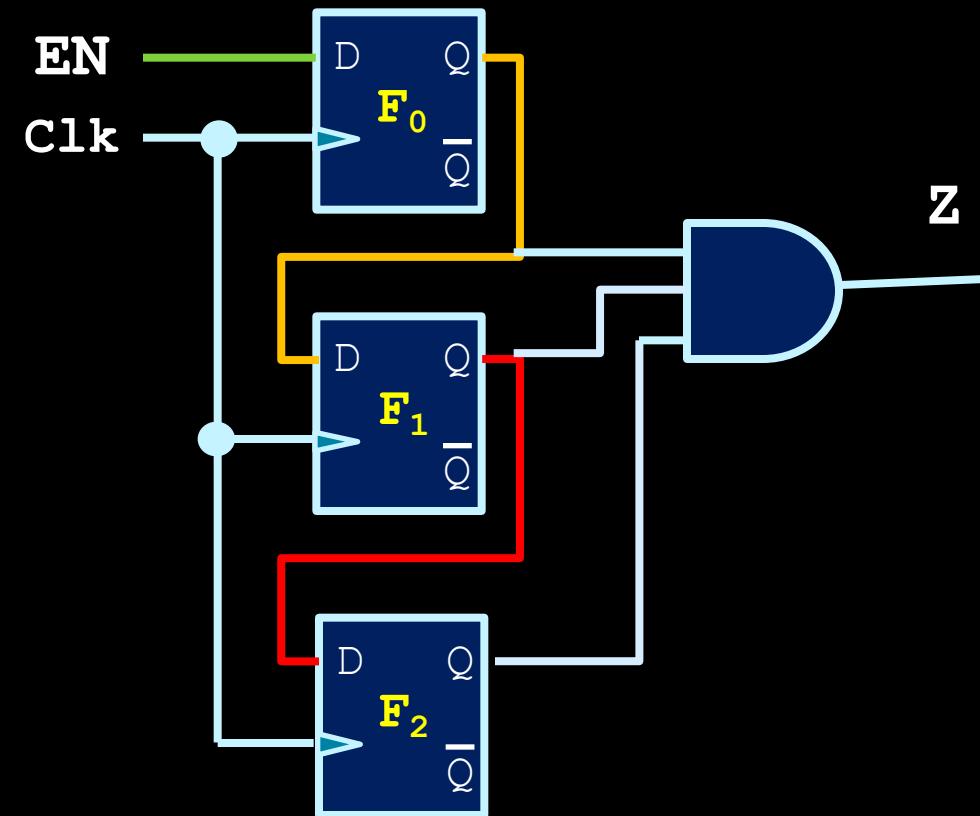
- Resulting circuit looks like the diagram on the right.
- This will record the states and make the state transitions happen based on the input,
- What about the **output value Z** which should go high when we **have three highs in a row**.



Step 5: Circuit design

- Boolean equation for Z:

$$Z = F_0 \cdot F_1 \cdot F_2$$

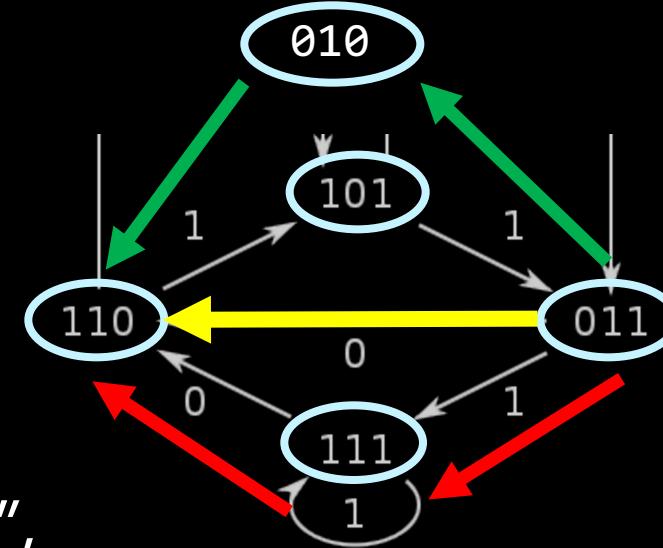


Moore machine vs Mealy machine

- Two ways to derive the circuitry needed for the output values of the state machine:
 - Moore machine:
 - The output for the FSM depends solely on the **current state** (based on entry actions).
 - Mealy machine:
 - The output for the FSM depends on **the state and the input** (based on input actions).
 - Being in state X can result in different output, depending on the **input that caused that state**.

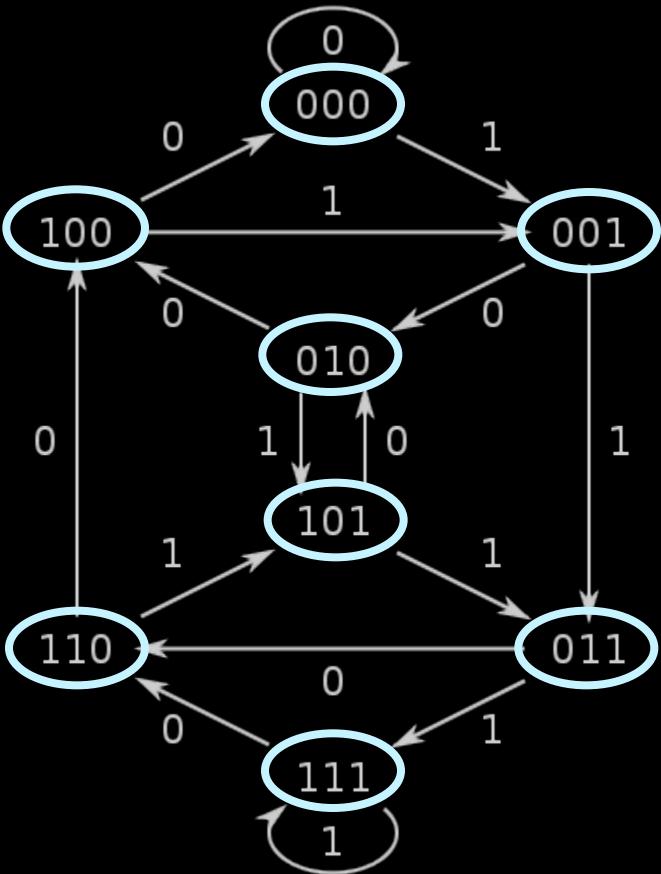
An issue: timing and state assignments

- Example: if recognizer circuit is in state **011** and gets a **0** as an input, it moves to state **110**.
 - The first and last digits should change “at the same time”, but they can’t.
 - If the first flip-flop changes first, the state will change to **111**, and the output **Z** would go **high** for an instant, which is **unexpected behaviour**.
 - If the second flip-flop changes first, it’s fine since the output would not go wrong.



An issue: timing and state assignments

- So how do you solve this?
- Two possible solutions:
 1. Whenever possible, make flip-flop assignments such that neighbouring states differ by **at most one** flip-flop value.
 - Intermediate states can be **allowed** if the output generated by those states is consistent with the output of the starting or destination states.
 2. If the intermediate states are unused in the state diagram, you can set the output for these states to provide the output that you need.
 - Might need to **add more flip-flops** to create these states.



State **ooo** does not have to have flip-flop values **ooo**, it can be anything you want to assign.

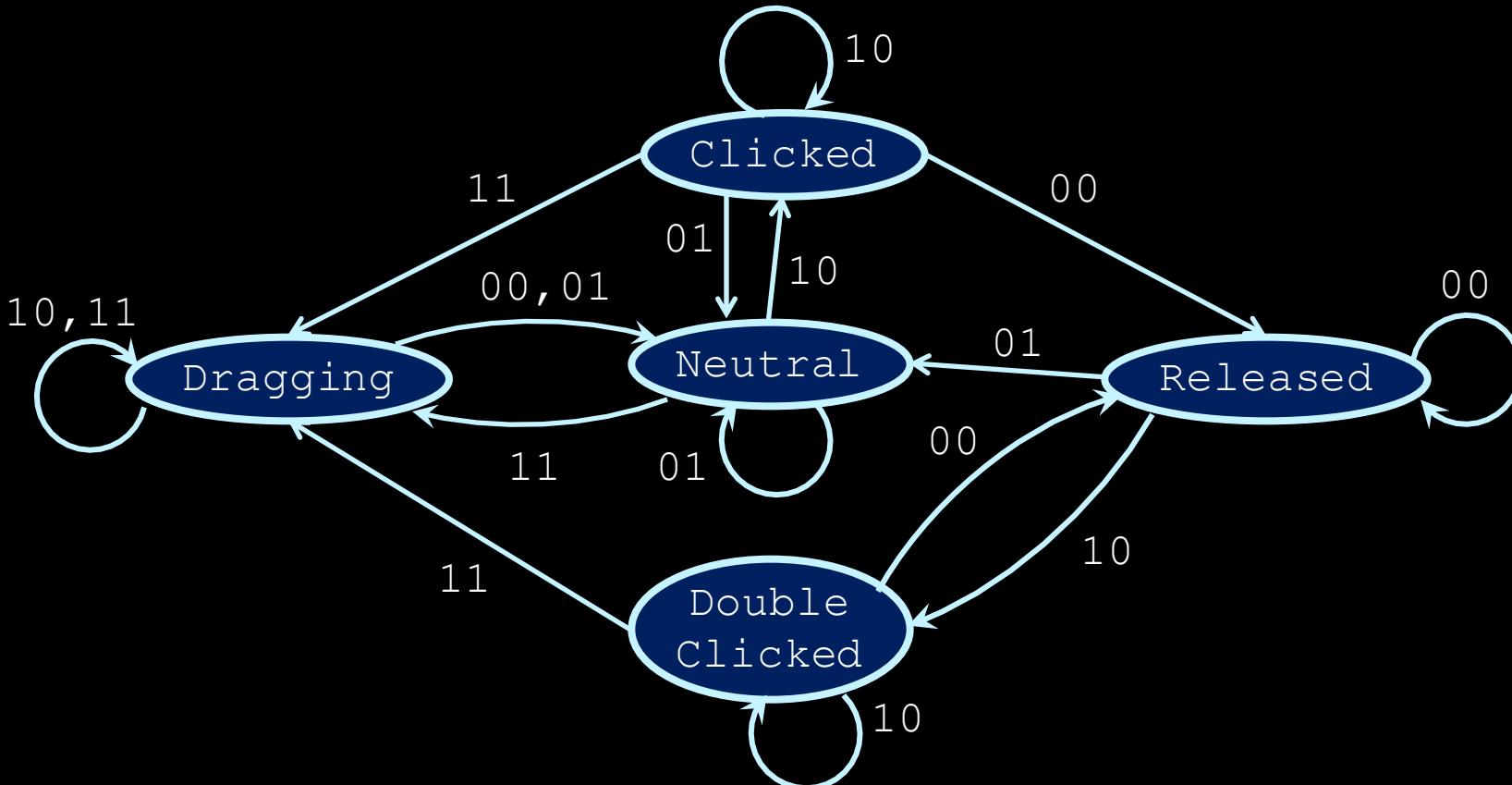
Previous State	EN	Next State
000	0	000
000	1	001
001	0	010
001	1	011
010	0	100
010	1	101
011	0	110
011	1	111
100	0	000
100	1	001
101	0	010
101	1	011
110	0	100
110	1	101
111	0	110
111	1	111

Home exercise: re-assign the states so the time issue doesn't exist

Another example: Mouse clicks

- Design a circuit that takes in two signals:
 - A signal P, which is high if the user is pressing the mouse button,
 - A signal M, which is high if the mouse is being moved.
- Based on the inputs, indicate whether the user is clicking, double-clicking, or dragging the mouse on the screen.





- Transitions indicate the values of P&M.
- Outputs depend on the state (Moore machine)

Next week

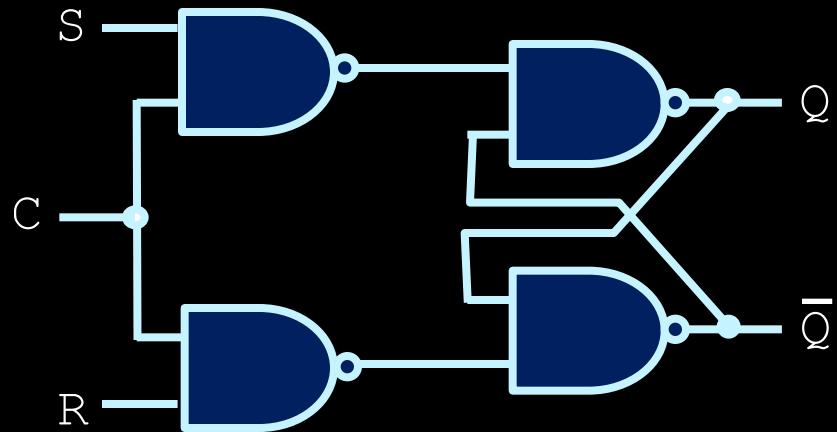
processor architecture



**KEEP
CALM
IT'S
QUIZ
TIME**

Q1: What is the improvement made by **D latch** compare to **SR latch**?

- A. avoid oscillation
- B. avoid forbidden state**
- C. become edge-triggered
- D. none of above



Q2: What is the Q and \bar{Q} when $S = 0$, $R = 1$, $C = 1$

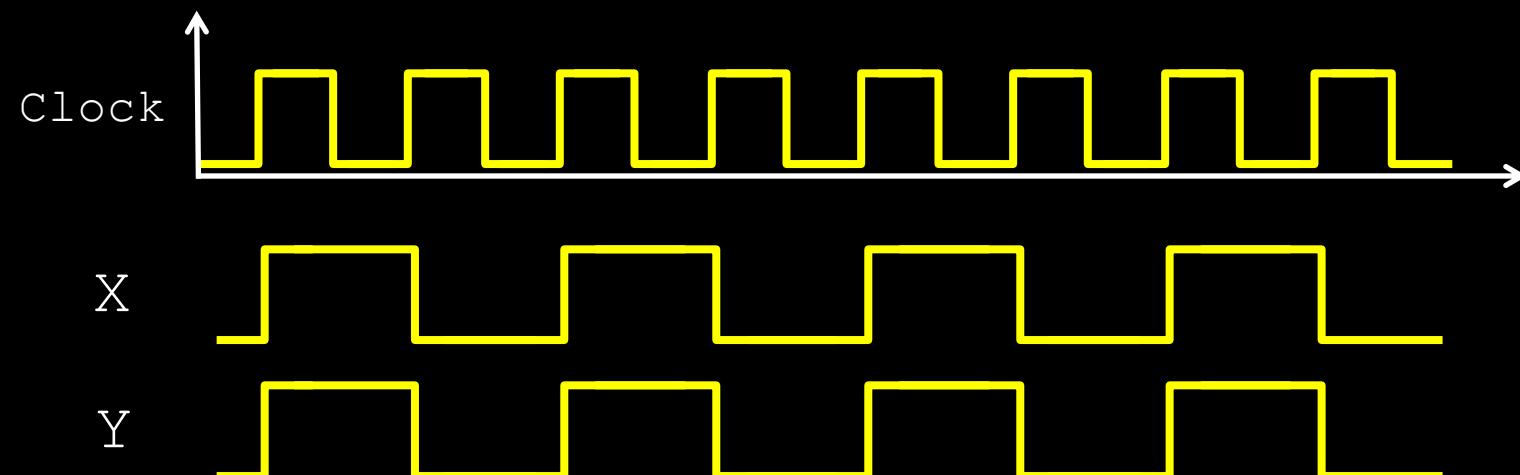
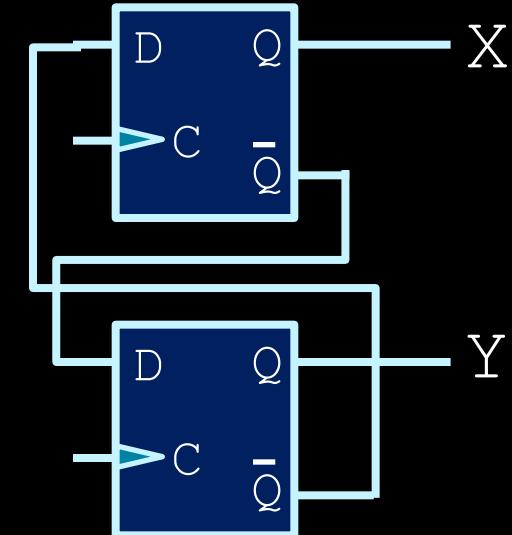
- A. $Q = 1, \bar{Q} = 0$
- B. $Q = 0, \bar{Q} = 1$**
- C. $Q = 1, \bar{Q} = 1$
- D. indeterminate
- E. none of above

Q3

Assuming the Q outputs of both flip-flops start off **low**, and assume **positive edge trigger**.

Draw the waveforms of X and Y.

(assume that signals change **without delay**)



CSC258 Winter 2016

Lecture 6

Today's outline

- Processor components: ALU
- Review for midterm test

Lab 5 Tip

- In simulation, I'm getting indeterminate output values all the time. What can I do?
- This is because in the handout circuit the input is never explicitly set. You can:
 - Add a RESET signal which ANDs with the inputs of the flip-flops, so when RESET is 0, the inputs are forced set to 0.
 - Take a leap of faith and try the DE-2 board directly. (Risky)

A Peek of Lab 6

- Implement a James Bond gadget.

<https://www.youtube.com/watch?v=Vi4LmILZUog>

- This involves the most pre-lab design work so far, so make sure to spend some time in your reading week working on it.

Quiz Time!



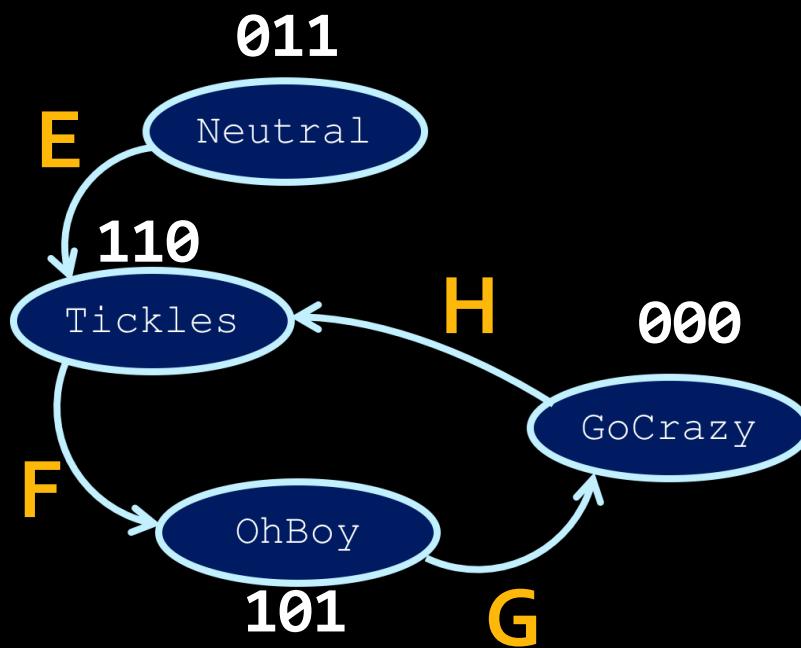
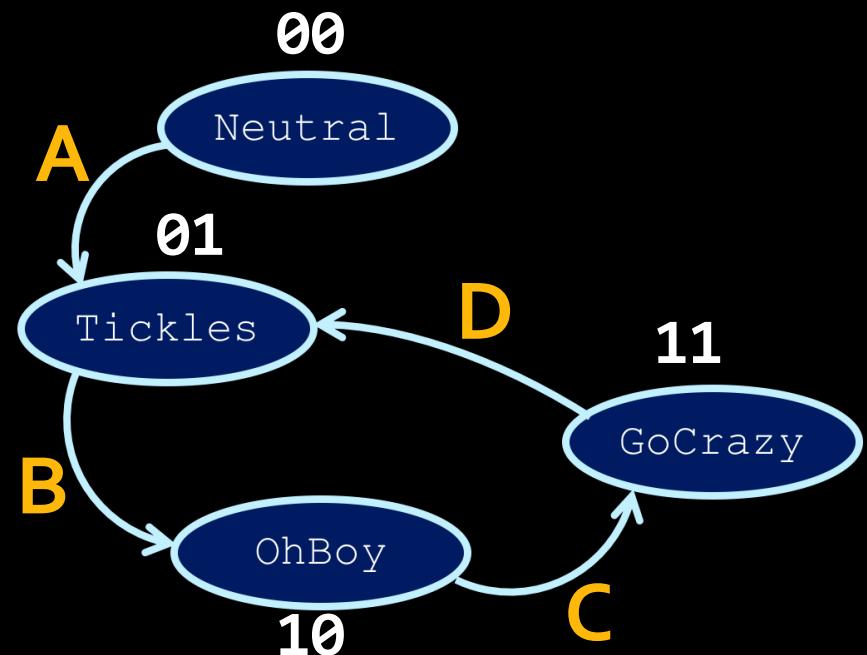
In this quiz, we will go through the steps of designing the Tickle-Me-Elmo circuit ...



Question 1

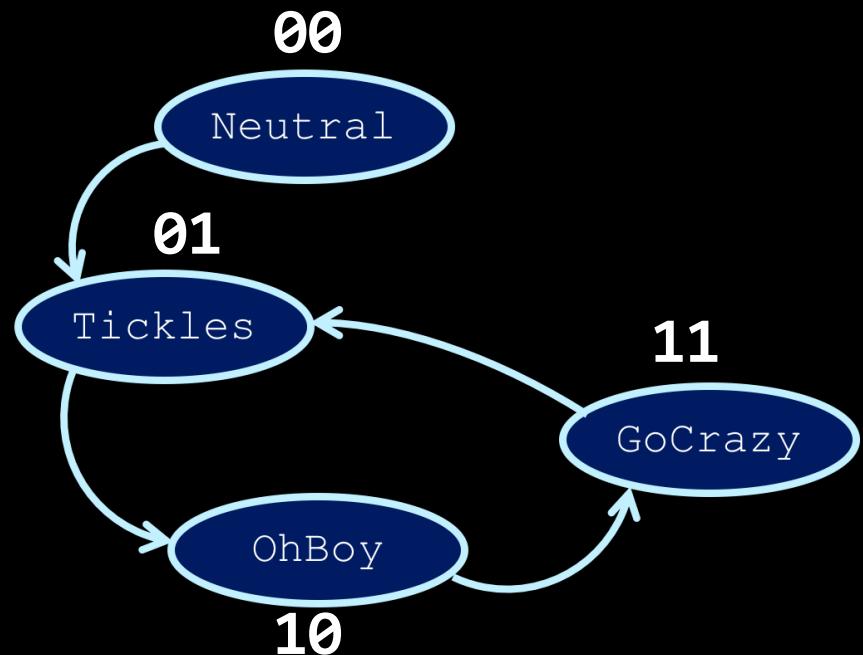
Below is the state transition diagram of Elmo, with two different ways of assigning flip-flop values.

Which one of the transitions (A-H) will cause **unexpected behaviour** that cannot be avoided?



Question 2

Suppose we decided to go with the following flip flop assignment.
And on the right side is the generated State Table.
Which row of the State Table is **wrong**?



	F ₁ F ₀	F ₁ F ₀
A	0 0	0 1
B	0 1	1 0
C	1 0	1 1
D	1 1	0 0

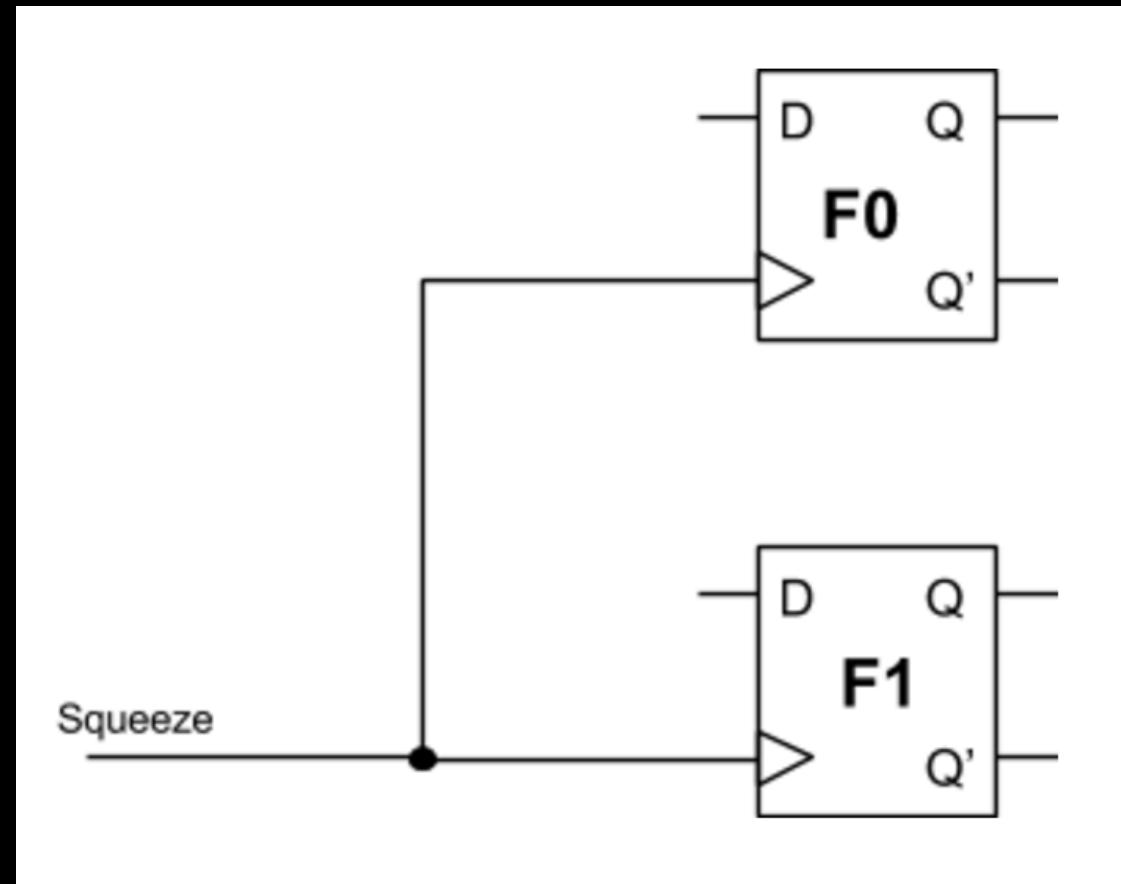
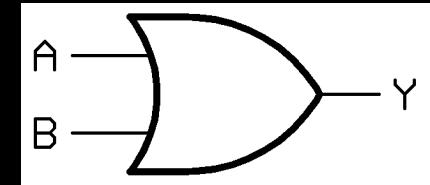
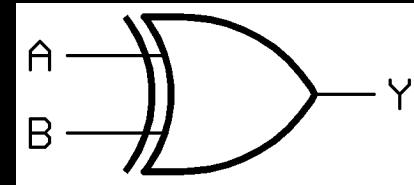
Question 3

Suppose the circuit expression we get from the State Table is the following:

$$F_1 = F_0 \oplus F_1$$

$$F_0 = F_1 + F_0'$$

Complete the circuit.



Done!

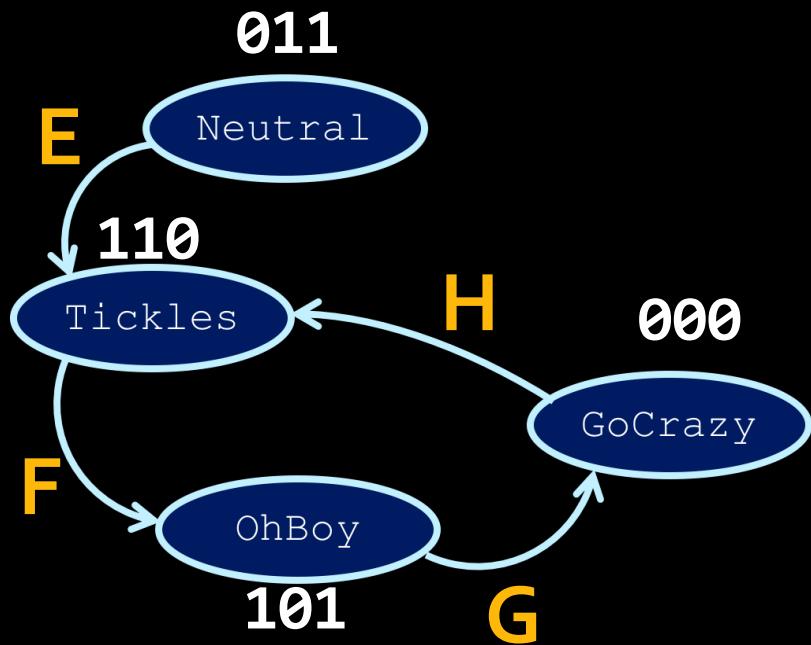
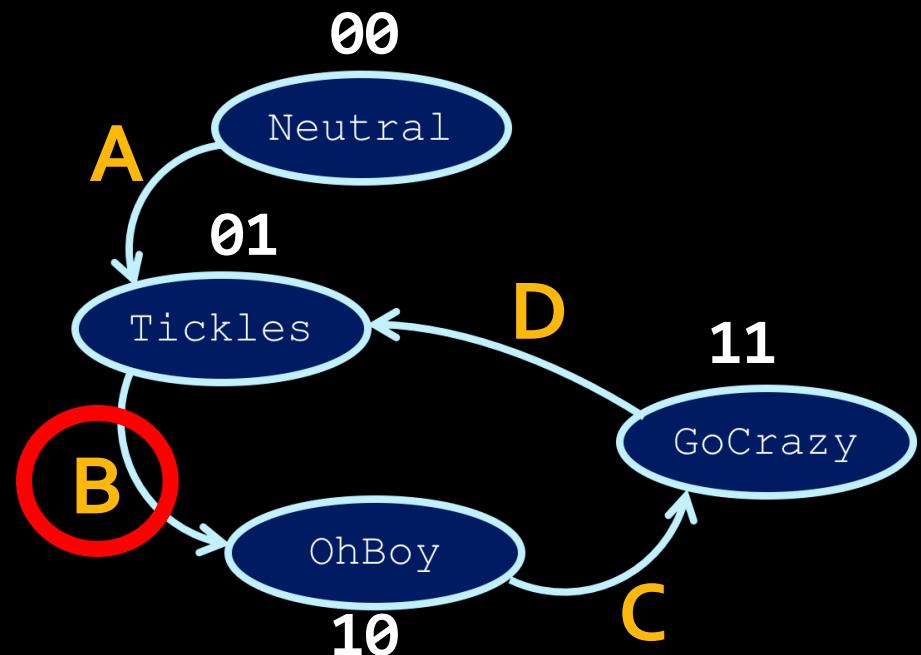
Taking it up...

Question 1

Transition B can enter **11** temporarily and cause Elmo to go crazy

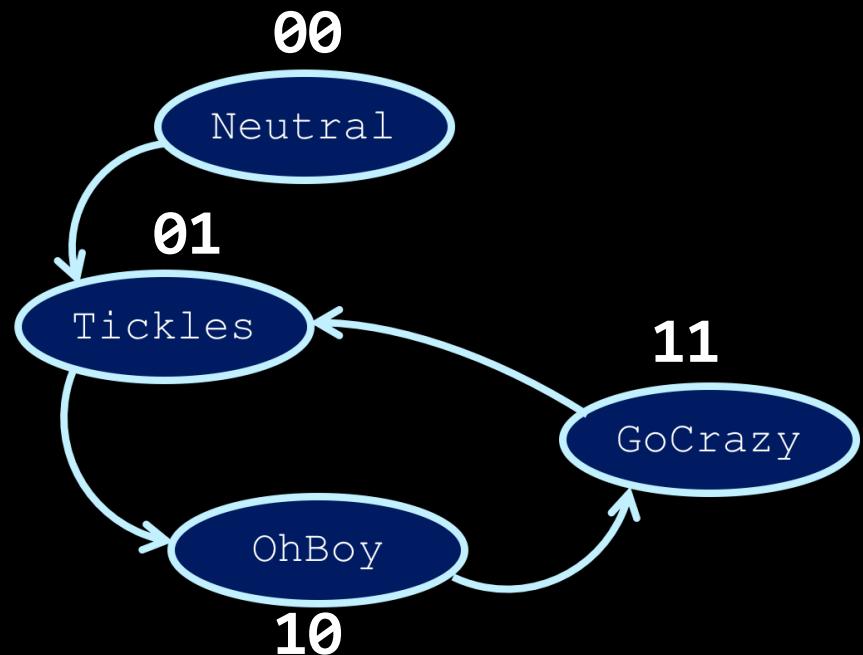
Transition E can enter **111** or **010**, both are **unused** state, so unexpected behaviours can be avoided.

Same argument for F, G, H.



Question 2

Suppose we decided to go with the following flip flop assignment.
And on the right side is the generated State Table.
Which row of the State Table is **wrong**?



	F1Fo	F1Fo
A	0 0	0 1
B	0 1	1 0
C	1 0	1 1
D	1 1	0 1

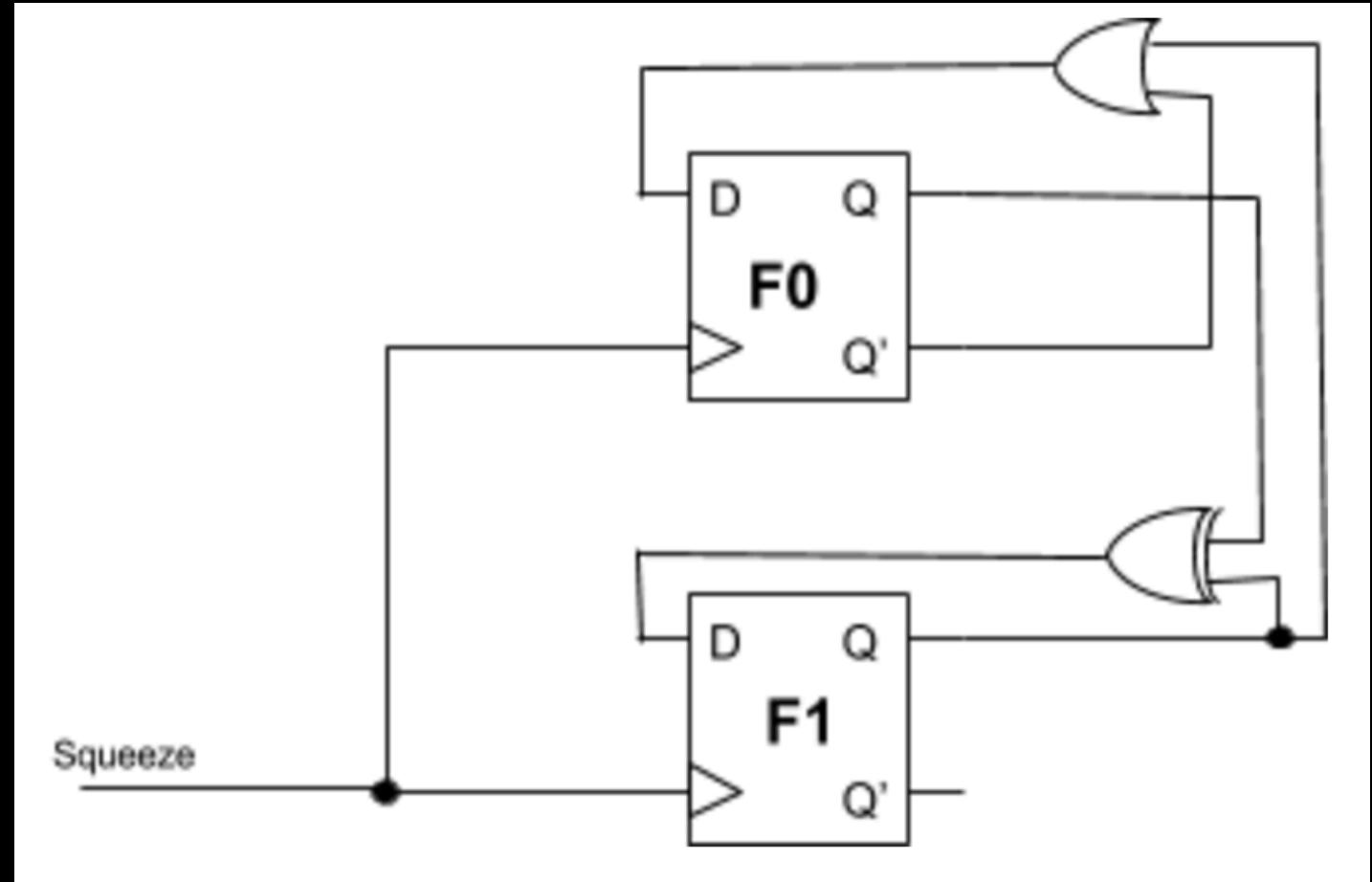
Question 3

Suppose the circuit expression we get from the State Table is the following:

$$F_1 = F_0 \oplus F_1$$

$$F_0 = F_1 + F_0'$$

Complete the circuit.

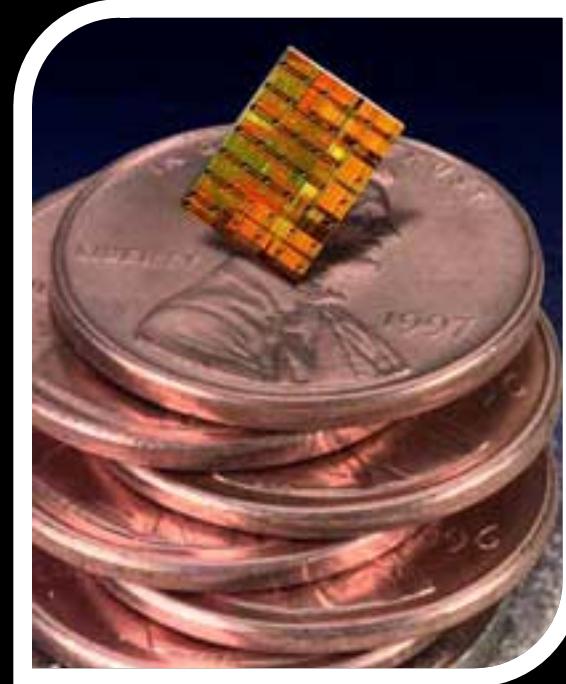


New Topic: Processor Components

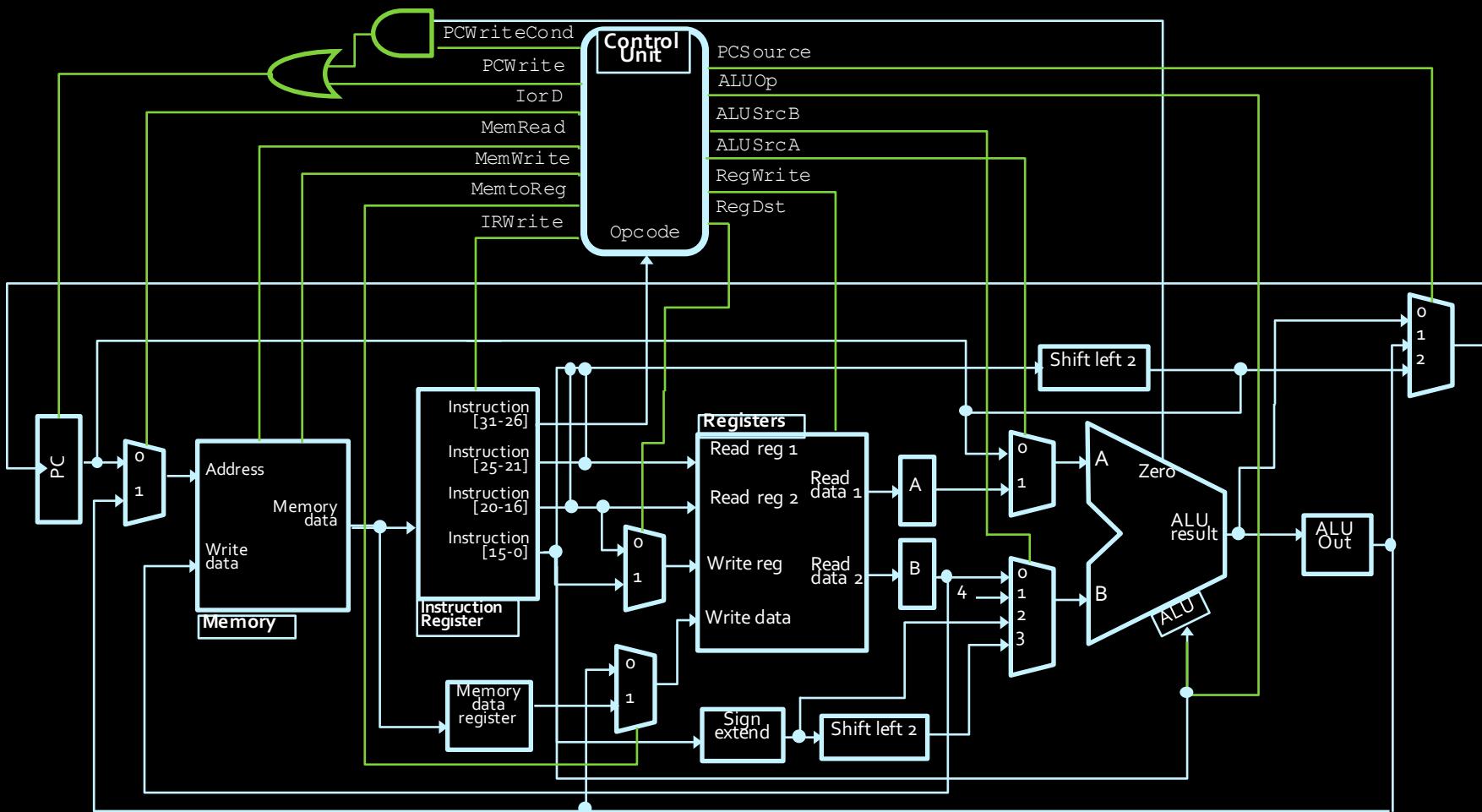
Using what we have learned so far
(combinational logic, devices, sequential
circuits, FSMs), how do we build a processor?

Microprocessors

- So far, we've been talking about making devices, such as adders, counters and registers.
- The ultimate goal is to make a **microprocessor**, which is a digital device that processes input, can store values and produces output, according to a set of on-board instructions.

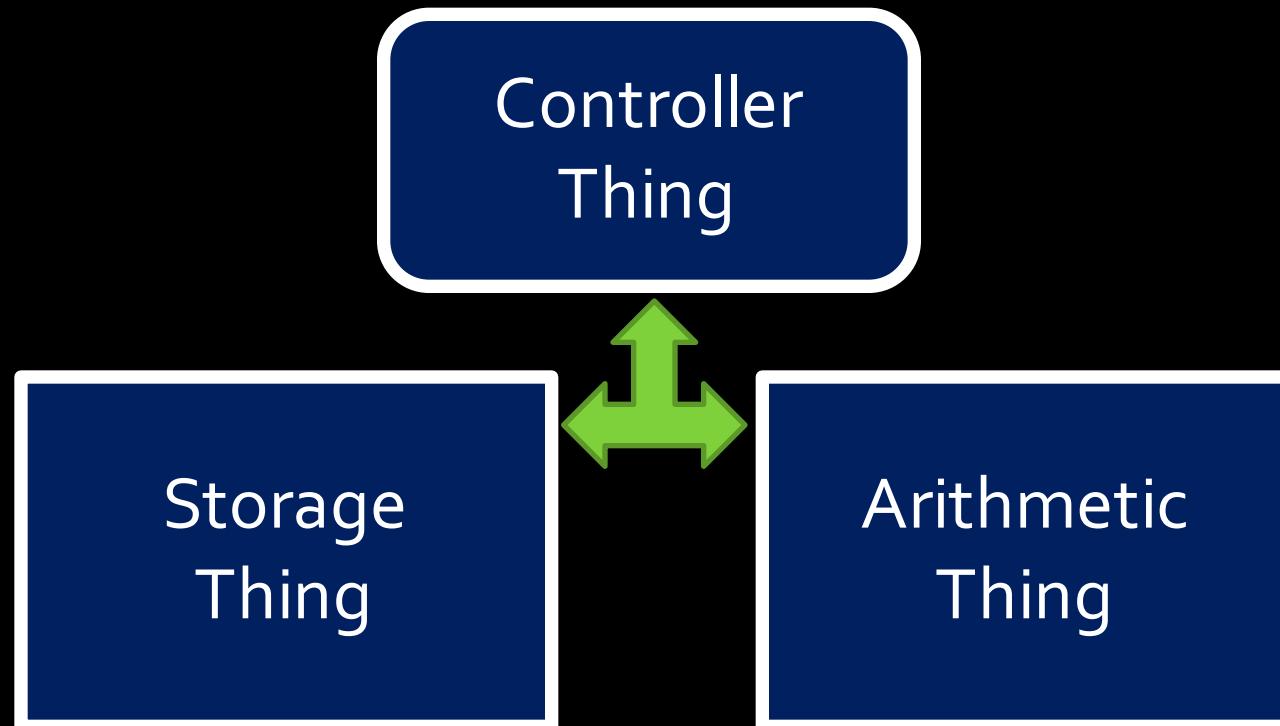


The Final Destination



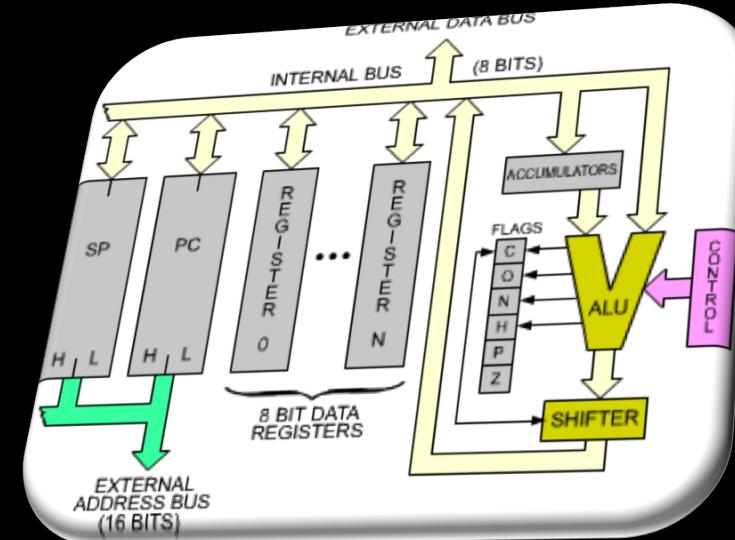
Deconstructing processors

- Processors aren't so bad when you consider them piece by piece:



Microprocessors

- These devices are a combination of the units that we've discussed so far:
 - Registers to store values.
 - Adders and shifters to process data.
 - Finite state machines to control the process.
- Microprocessors are the basis of all computing since the 1970's, and can be found in nearly every sort of electronics.

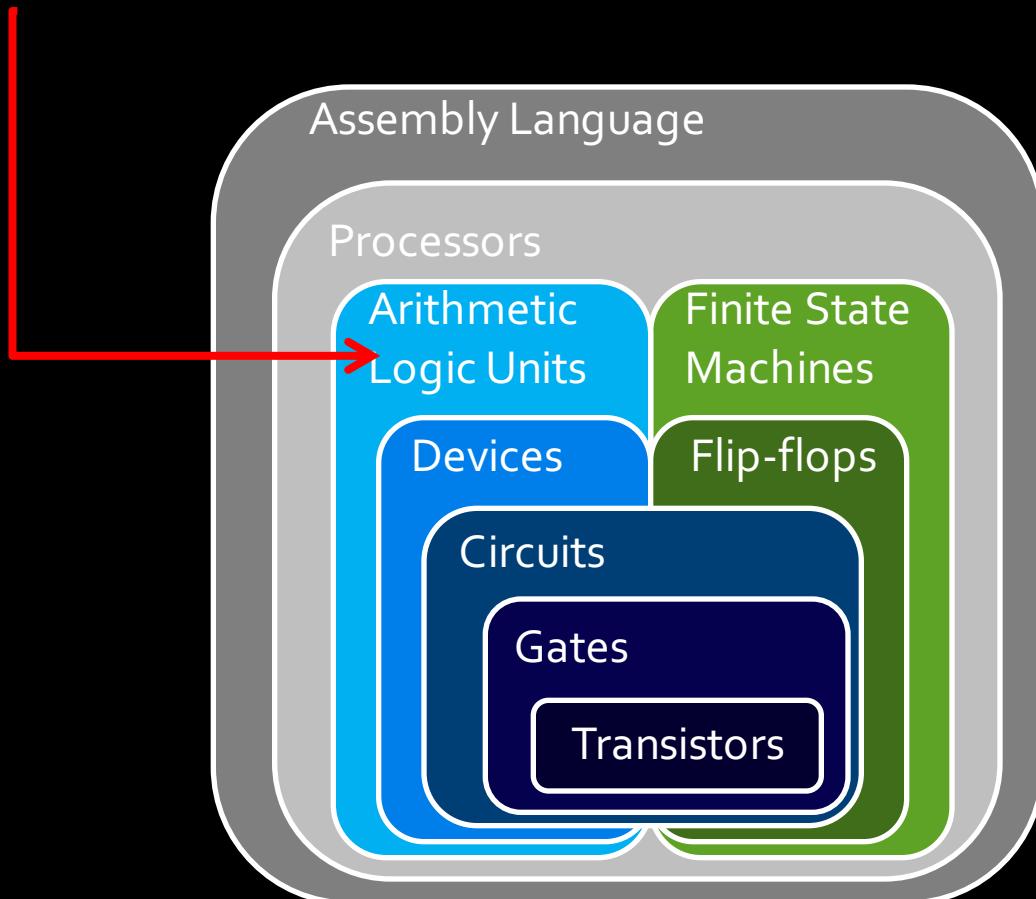


The “Arithmetic Thing”

aka: the Arithmetic Logic Unit (ALU)

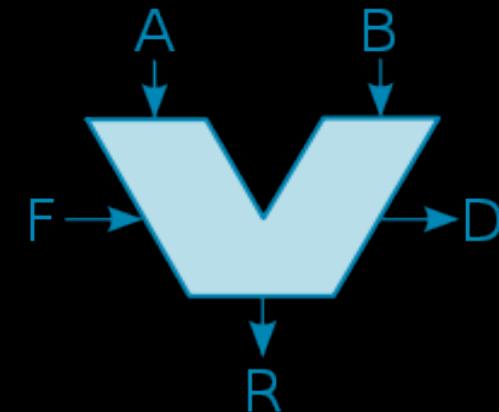


We are here



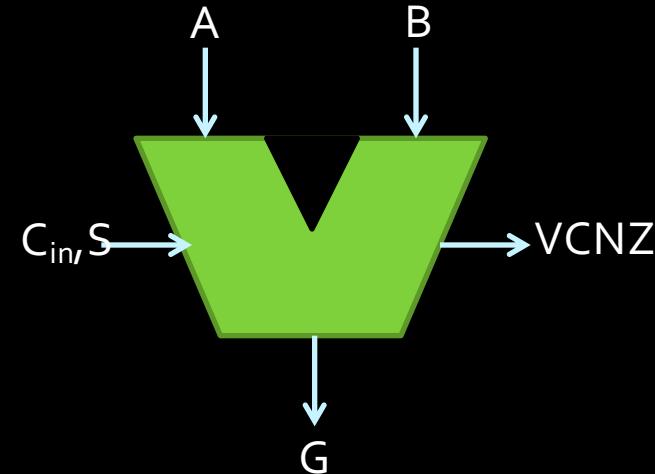
Arithmetic Logic Unit

- The first microprocessor applications were calculators.
 - Recall the unit on adders and subtractors.
 - These are part of a larger structure called the **arithmetic logic unit** (ALU).
- This larger structure is responsible for the processing of all data values in a basic CPU.



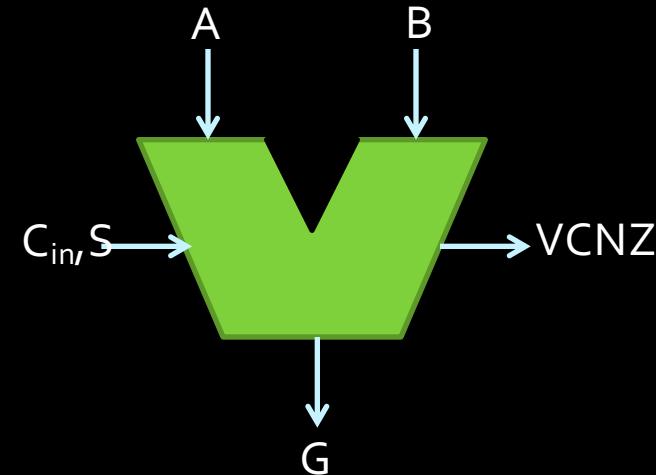
ALU inputs

- The ALU performs all of the arithmetic operations covered in this course so far, and logical operations as well (AND, OR, NOT, etc.)
 - A and B are the operands
 - The select bits (S) indicate which operation is being performed (S_2 is a mode select bit, indicating whether the ALU is in arithmetic or logic mode).
 - The carry bit C_{in} is used in operations such as incrementing an input value or the overall result.



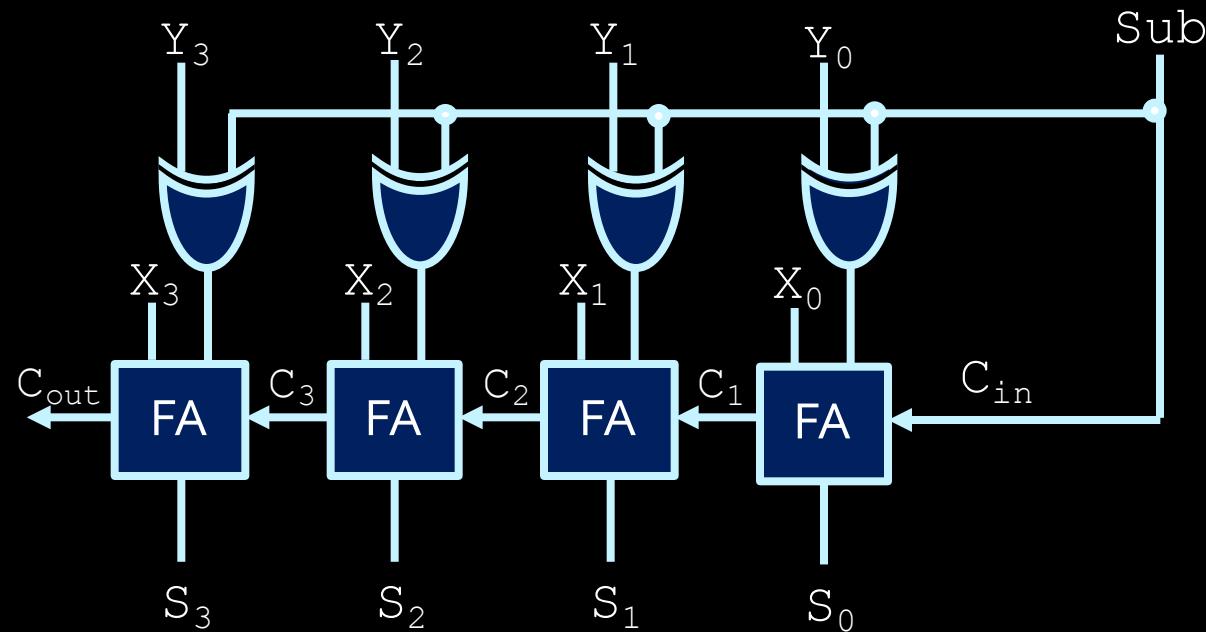
ALU outputs

- In addition to the input signals, there are output signals V, C, N & Z which indicate special conditions in the arithmetic result:
 - **V**: overflow condition
 - The result of the operation could not be stored in the n bits of G, meaning that the result is incorrect.
 - **C**: carry-out bit
 - **N**: Negative indicator
 - **Z**: Zero-condition indicator



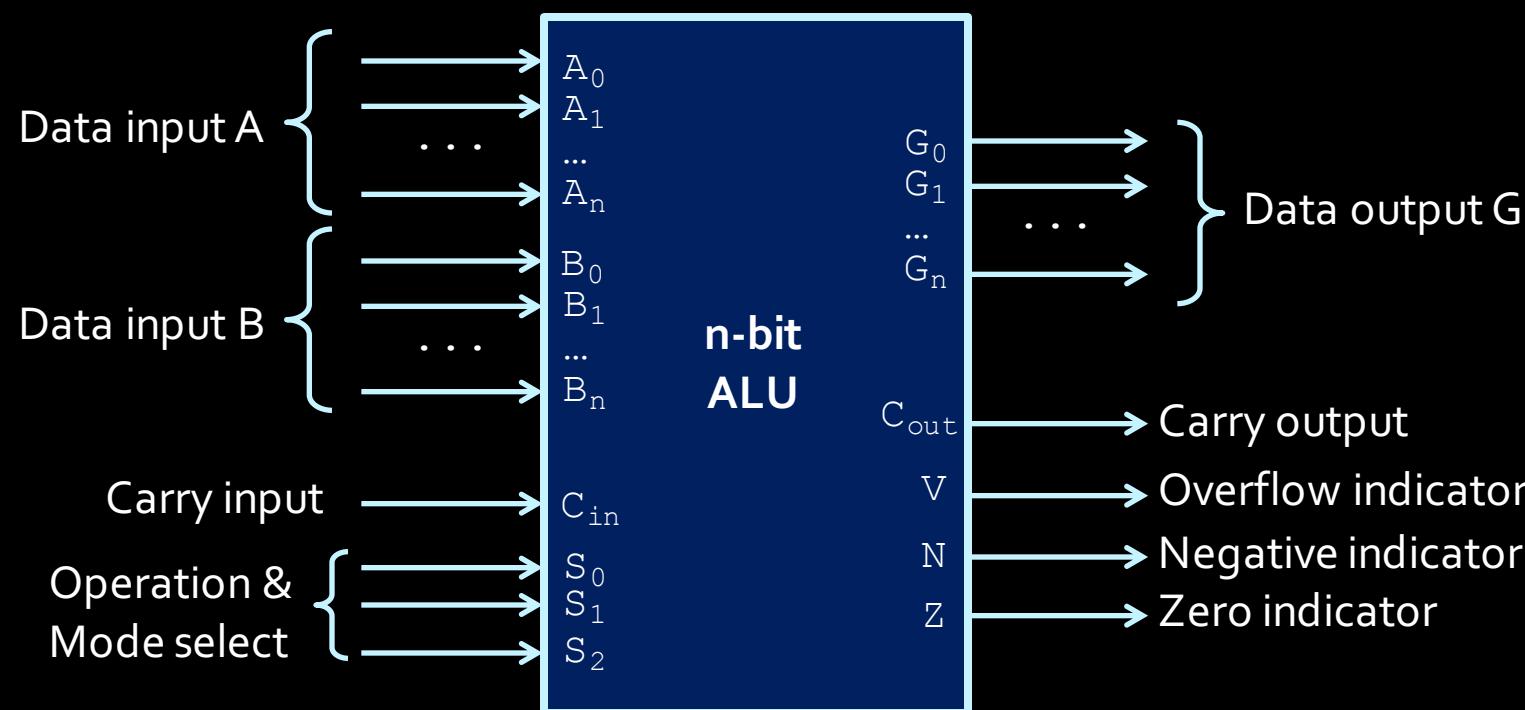
The “A” of ALU

- To understand how the ALU does all of these operations, let's start with the arithmetic side.
- Fundamentally, this side is made of an adder / subtractor unit, which we've seen already:

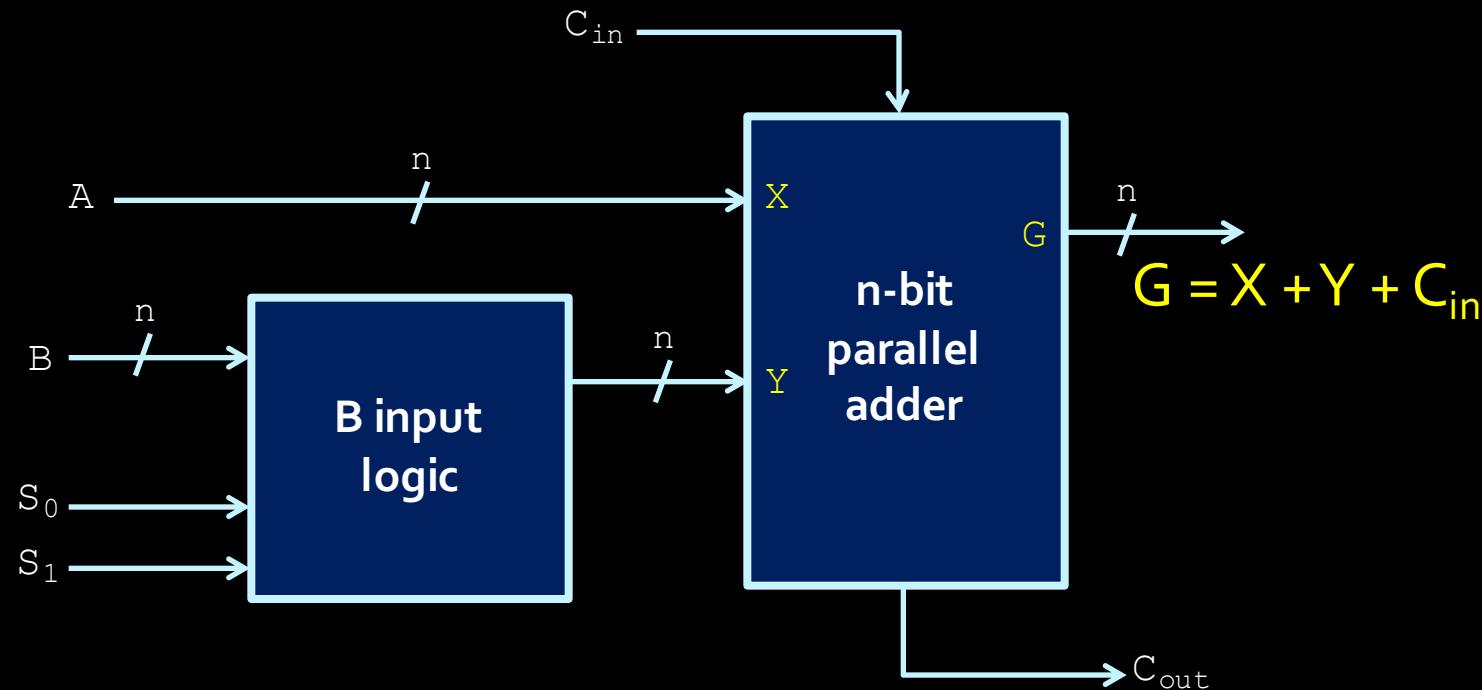


ALU block diagram

- In addition to data inputs and outputs, this circuit also has:
 - outputs indicating the different conditions,
 - inputs specifying the operation to perform (similar to Sub).



Arithmetic components



- In addition to addition and subtraction, many more operations can be performed by manipulating what is added to input A , as shown in the diagram above.

Arithmetic operations

$$1+1=3$$



- If the input logic circuit on the left sends B straight through to the adder, result is $G = A+B$
- What if B was replaced by all ones instead?
 - Result of addition operation: $G = A-1$
- What if B was replaced by \bar{B} ?
 - Result of addition operation: $G = A-B-1$
- And what if B was replaced by all zeroes?
 - Result is: $G = A$. (Not interesting, but useful!)

→ Instead of a Sub signal, the operation you want is signaled using the select bits S_0 & S_1 .

Operation selection

Select bits		Y input	Result	Operation
S_1	S_0			
0	0	All 0s	$G = A$	Transfer
0	1	B	$G = A+B$	Addition
1	0	\bar{B}	$G = A+\bar{B}$	Subtraction - 1
1	1	All 1s	$G = A-1$	Decrement

- This is a good start! But something is missing...
- Wait, what about the carry bit?

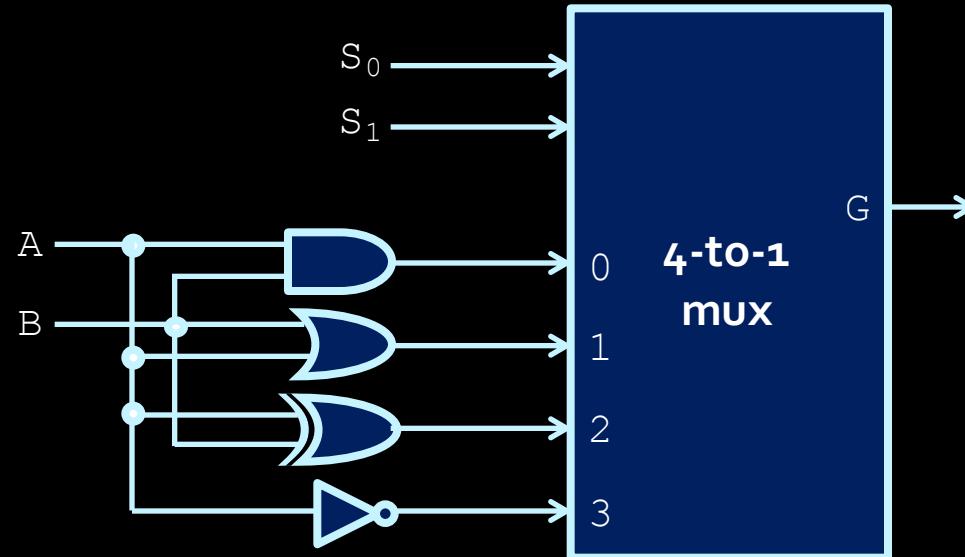
Full operation selection

Select			Input		Operation	
S_1	S_0	Y	$C_{in}=0$		$C_{in}=1$	
0	0	All 0s	$G = A$ (transfer)		$G = A+1$ (increment)	
0	1	B	$G = A+B$ (add)		$G = A+B+1$	
1	0	\bar{B}	$G = A+\bar{B}$		$G = A+\bar{B}+1$ (subtract)	
1	1	All 1s	$G = A-1$ (decrement)		$G = A$ (transfer)	

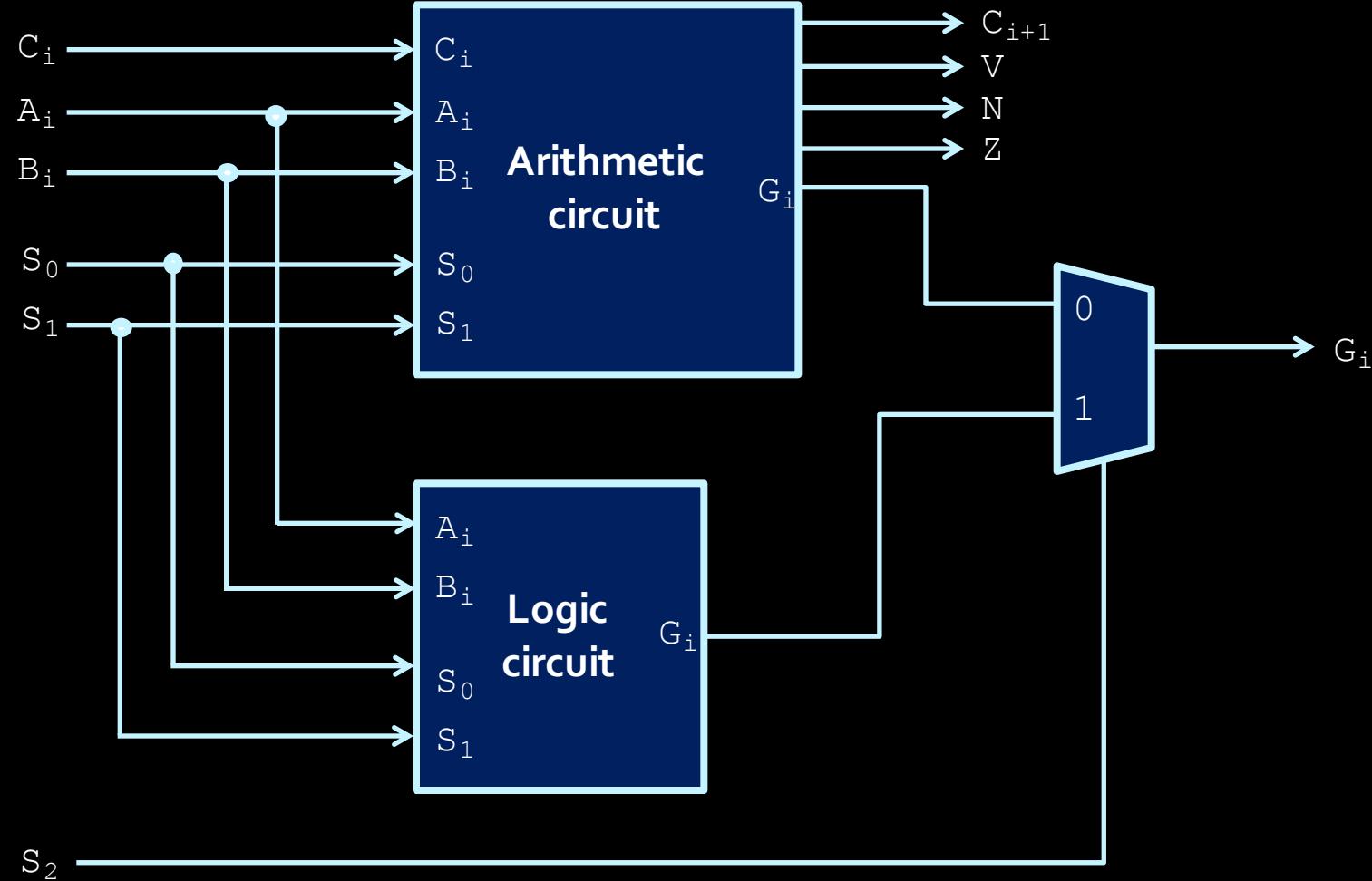
- Based on the values on the select bits and the carry bit, we can perform any number of basic arithmetic operations by manipulating what value is added to A.

The “L” of ALU

- We also want a circuit that can perform **logical operations**, in addition to arithmetic ones.
- How do we tell which operation to perform?
 - Another select bit!
- If $S_2 = 1$, then logic circuit block is activated.
- Multiplexer is used to determine which block (logical or arithmetic) goes to the output.



Single ALU Stage



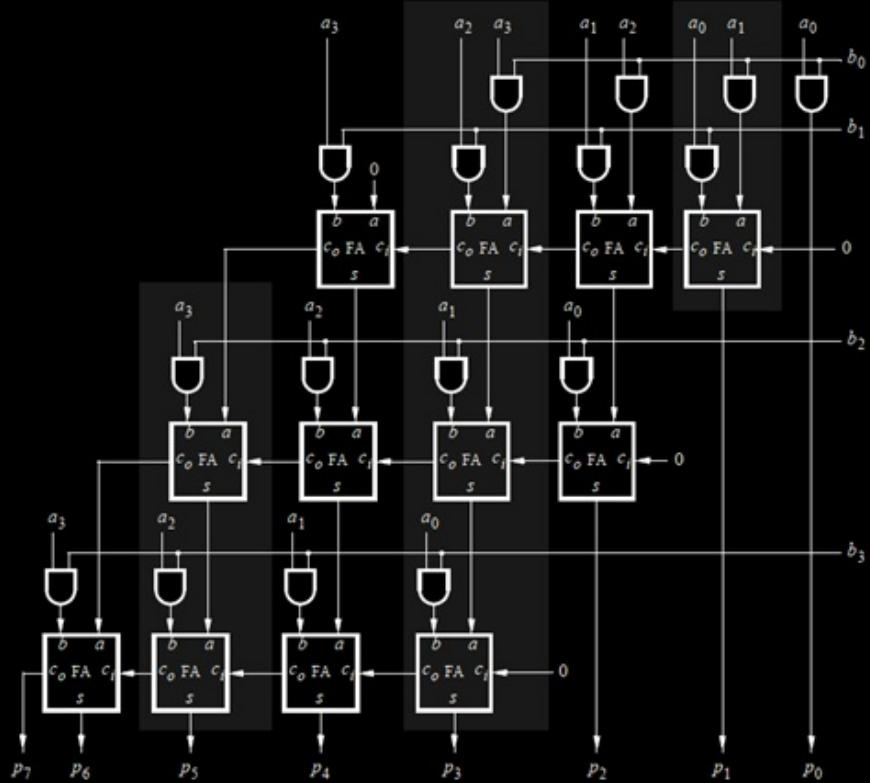
Multiplication

What about multiplication?

- Multiplication (and division) operations are always more complicated than other arithmetic (plus, minus) or logical (AND, OR) operations.
- Three major ways that multiplication can be implemented in circuitry:
 - Layered rows of adder units.
 - An adder/shifter circuit
 - Booth's Algorithm

Multiplication

- Multiplier circuits can be constructed as an array of adder circuits.
- This can get a little expensive as the size of the operands grows.
- Is there an alternative to this circuit?



Multiplication

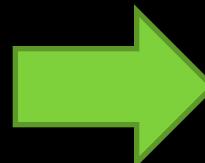
- Revisiting grade 3 math...

$$\begin{array}{r} 123 \\ \times 456 \\ \hline \end{array}$$

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \end{array}$$

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \end{array}$$

$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \\ 456 \end{array}$$



$$\begin{array}{r} 123 \\ \times 456 \\ \hline 1368 \\ 912 \\ 456 \\ \hline 56088 \end{array}$$

Multiplication

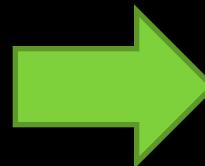
- And now, in binary...

$$\begin{array}{r} 101 \\ \times 110 \\ \hline \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 110 \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 110 \\ 000 \end{array}$$

$$\begin{array}{r} 101 \\ \times 110 \\ \hline 110 \\ 000 \\ 110 \end{array}$$



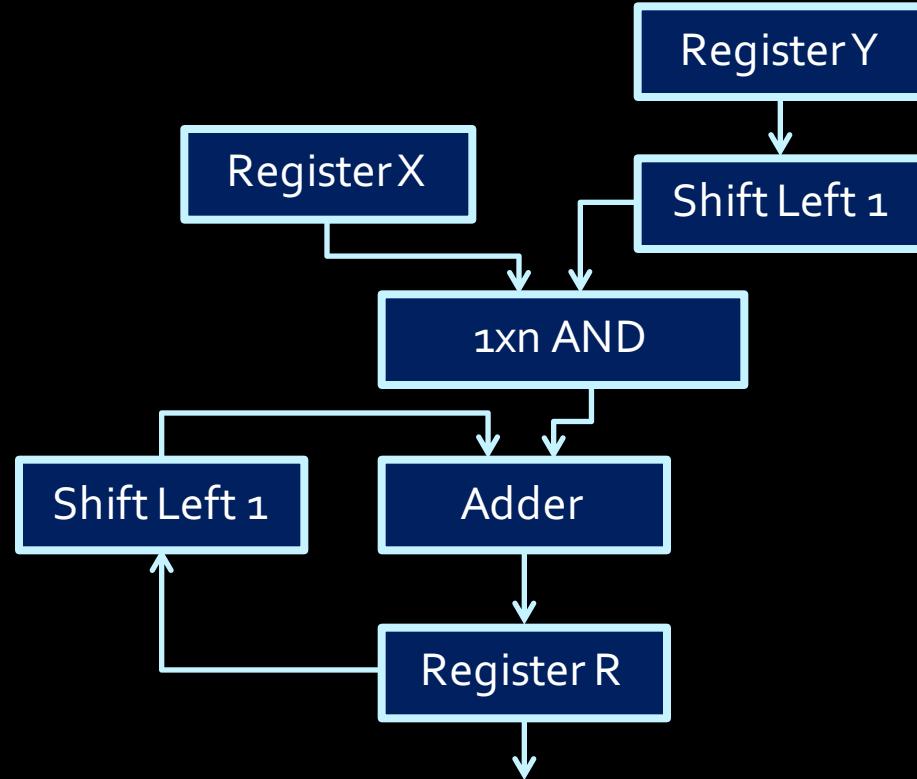
$$\begin{array}{r} 101 \\ \times 110 \\ \hline 110 \\ 000 \\ 110 \\ \hline 11110 \end{array}$$

Observations

- Calculation flow
 - Multiply by 1 bit of multiplier
 - Add to sum and shift sum
 - Shift multiplier by 1 bit
 - Repeat the above
- What is “multiply by 1 bit of binary”?
 - $10101 \times 1 ?$
 - $10101 \times 0 ?$
 - It's an **AND**!

Accumulator circuits

- What if you could perform each stage of the multiplication operation, one after the other?
 - This circuit would only need a single row of adders and a couple of shift registers.



Make it more efficient

Think about 258×9999

- Multiply by 9, add to sum, shift, multiply by 9, add to sum, shift, multiple by 9, add to sum, shift, multiply by 9, add to sum.
- $258 \times 9999 = 258 \times (10000 - 1) = 258 \times 10000 - 258$
- Just shift 258, becomes 2580000, then do $2580000 - 258$
- More efficient!

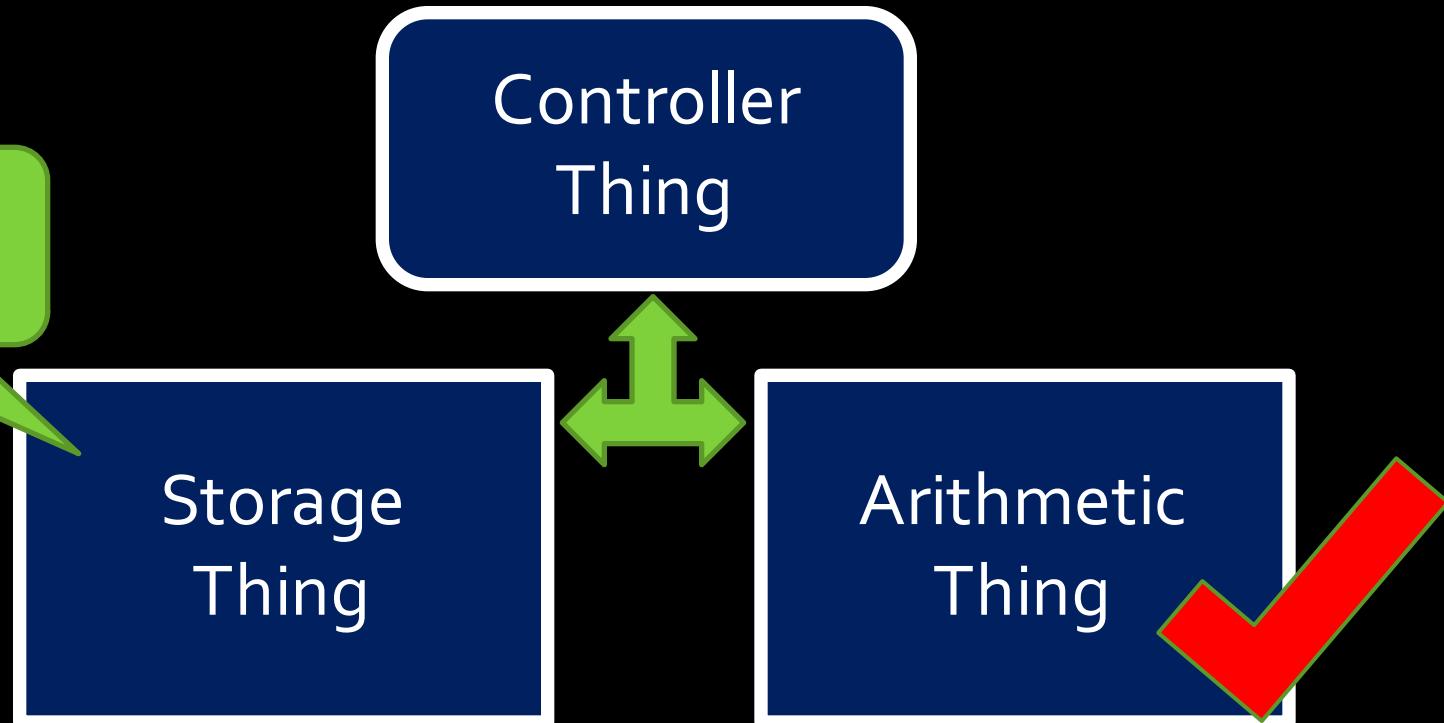
More efficient: Booth's Algorithm

- Take advantage of circuits where **shifting is cheaper** than adding, or where space is at a premium.
 - when multiplying by certain values (e.g. 99), it can be easier to think of this operation as a difference between two products.
- Consider the shortcut method when multiplying a given decimal value X by 9999:
 - $X * 9999 = X * 10000 - X * 1$
- Now consider the equivalent problem in binary:
 - $X * 001111 = X * 010000 - X * 1$
- More details: https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

Reflections on multiplication

- Multiplication isn't as common an operation as addition or subtraction, but occurs enough that its implementation is handled in the hardware.
- Most common multiplication and division operations are powers of 2. For this, the shift register is used instead of the multiplier circuit.

Next



Midterm Review

Time & Location

Monday, Feb 22, 3:10pm to 4:00pm

No aid.

Bring your TCard

Types of questions

- Short answer: basic understanding
- Circuit analysis: given a circuit understand it
- Circuit design: given a requirement, design a circuit

How to study for midterm

1. Review slides
2. Review what you did for labs
3. Review quizzes
4. Practice with past test
 - Posted on course web page with solutions
 - Ignore Verilog questions
5. Whenever confused, ask on Piazza or go to office hours

Office hours in Reading Week:

No office hour on Monday (Family Day)

Tuesday as usual: 5pm ~ 6:30pm

Friday from 3pm ~ 6:30pm

CSC258 Winter 2016

Lecture 7



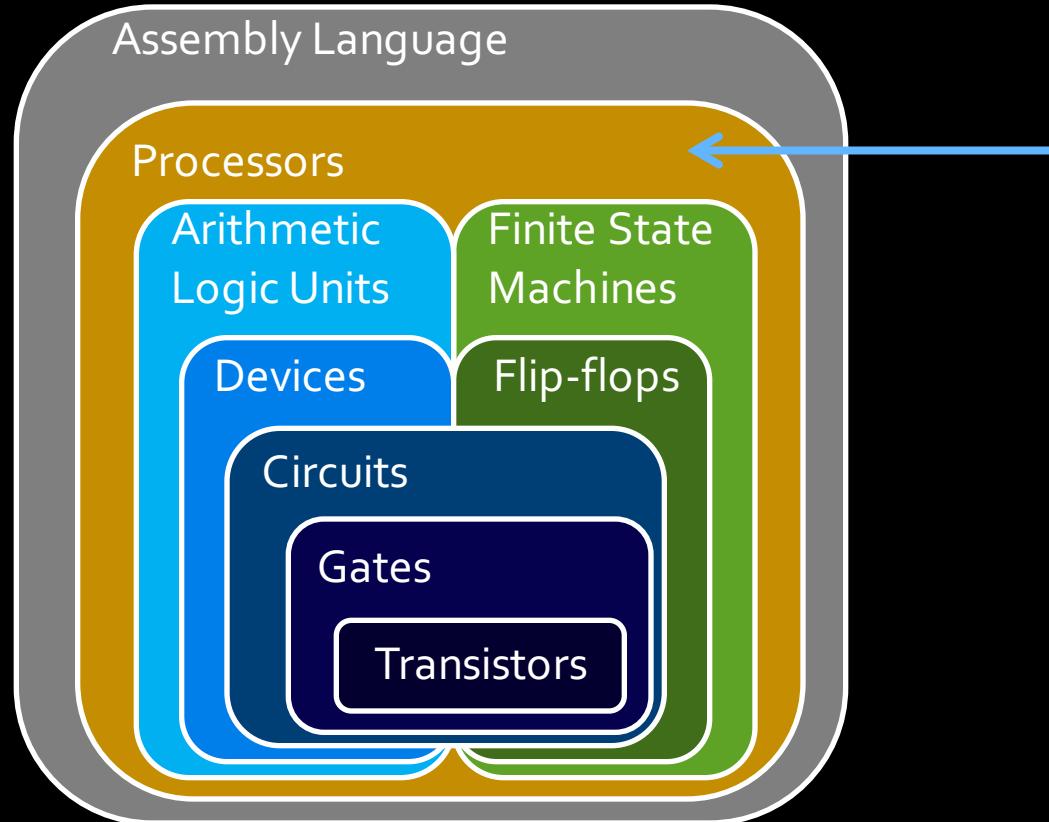
**KEEP
CALM
AND
NEVER
GIVE UP**



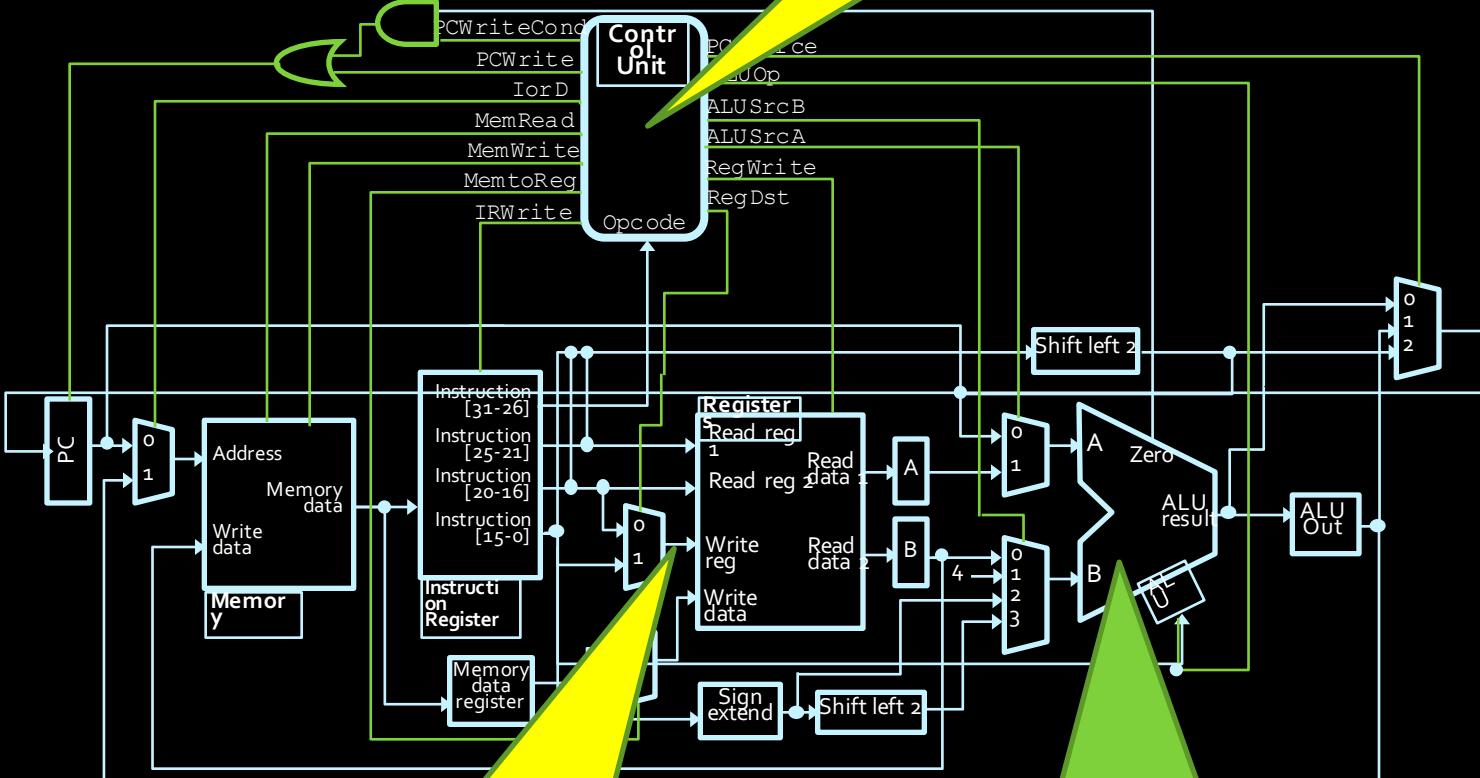
Drop date: Mar 6

JK, don't do it yet, you will get your midterm marks back before that, then you can decide.

Recap: We are here



The Blueprint of a microprocessor

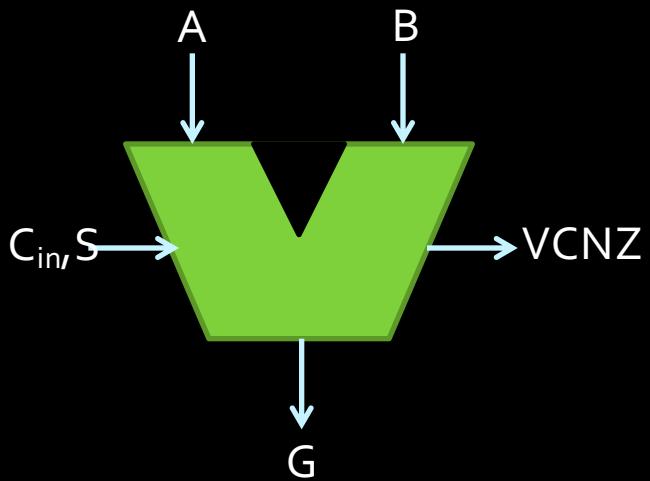


The Storage Thing

The Arithmetic Thing

We've learned the arithmetic thing

ALU



With ALU, we can do addition, subtraction, logic operations, etc.

Multiplication was a bit trickier

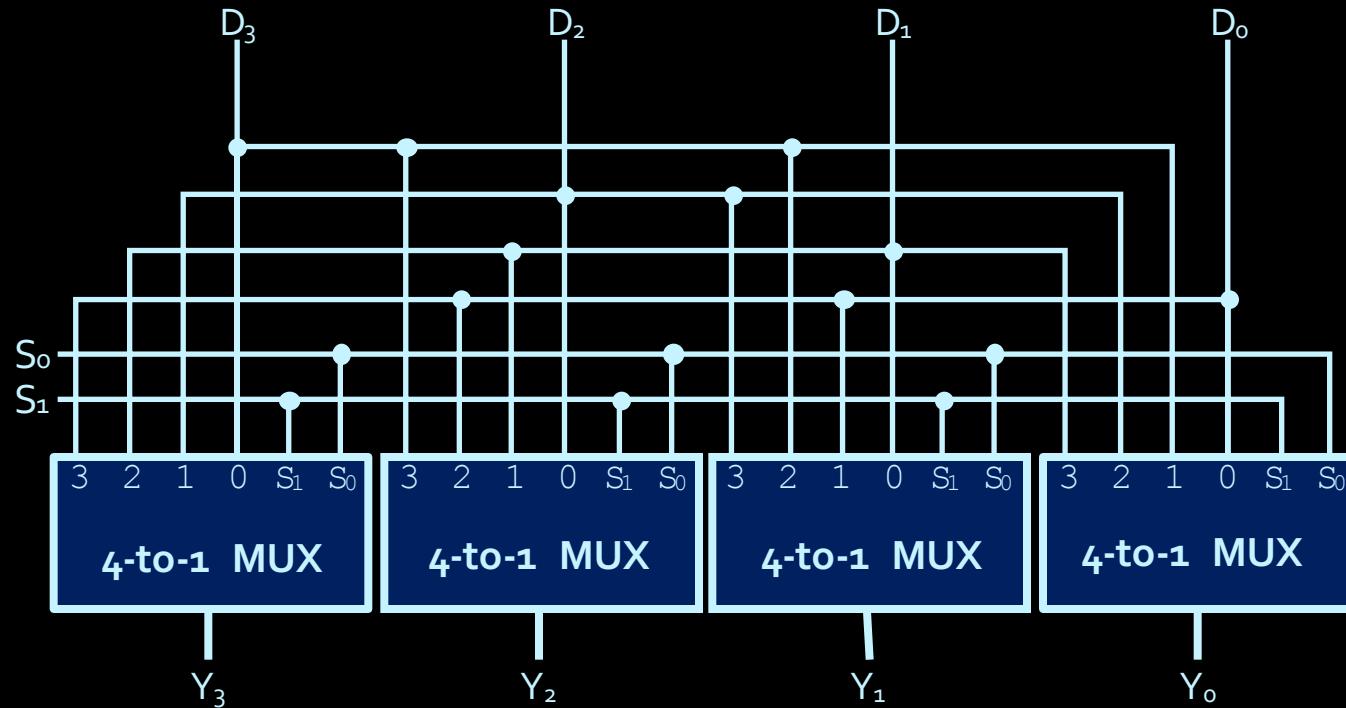
- Booth algorithm

To implement multiplication using Booth algorithm, we basically repeatedly do three things

- Addition
- Subtraction (kind-of an addition)
- Shift

We learned how to do addition and subtraction.
How to do shift?

Shifter unit



- **Barrel shifter** acts alongside ALU to shift data elements according to S_0 and S_1 .

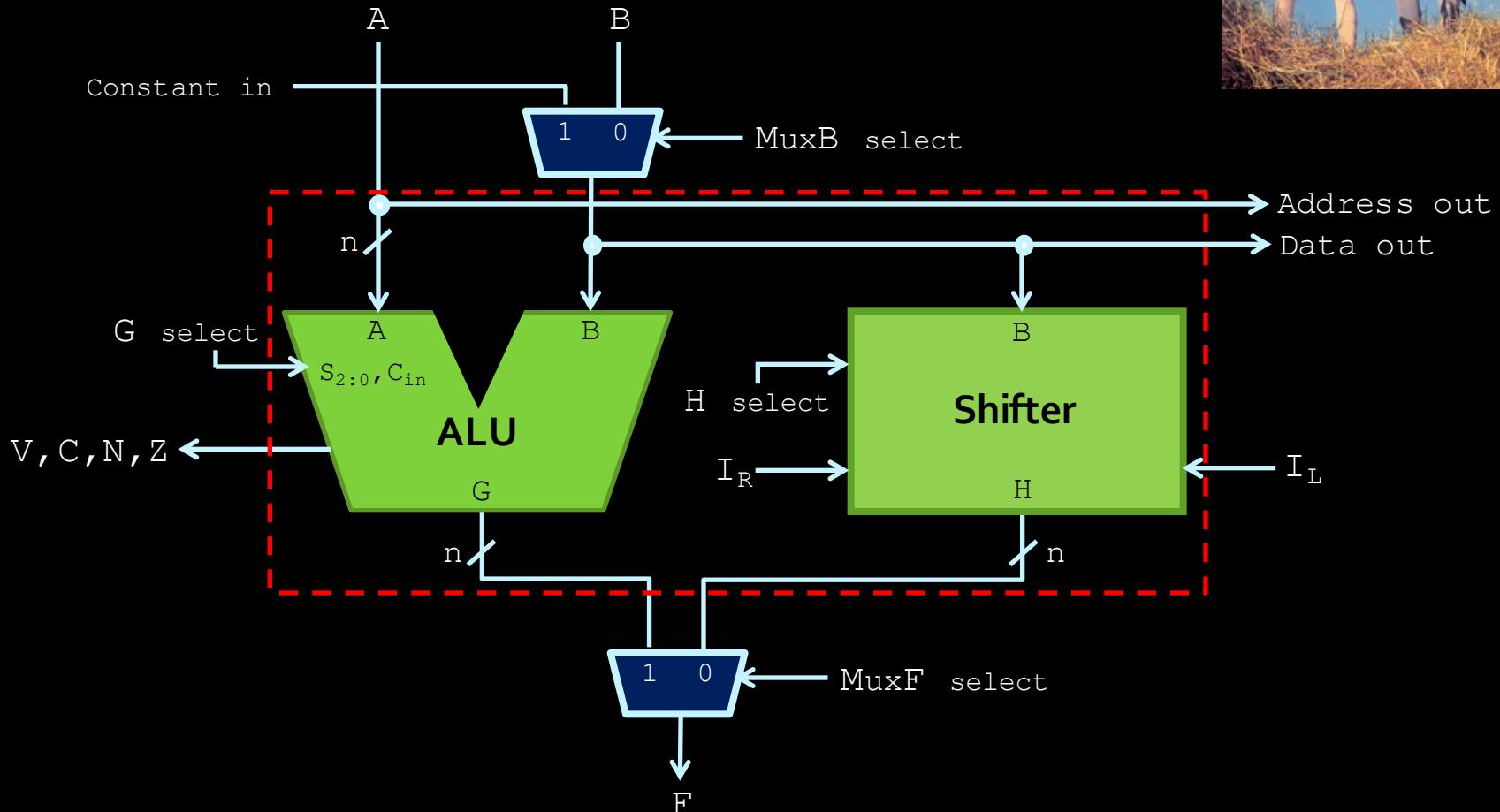
If $S_1S_0 = 00$: $Y_3Y_2Y_1Y_0 = D_3D_2D_1D_0$
If $S_1S_0 = 01$: $Y_3Y_2Y_1Y_0 = D_2D_1D_0D_3$

ALU partnered with Shifter

- We can implement all kinds of arithmetic functions, like...
- $(A+B)*(A-B) - 3*A*B$



Function Unit



So where do A and B come from?

The “Storage Thing”

aka: the register file and main memory



Computer memory hierarchy

In terms of data (food) access speed

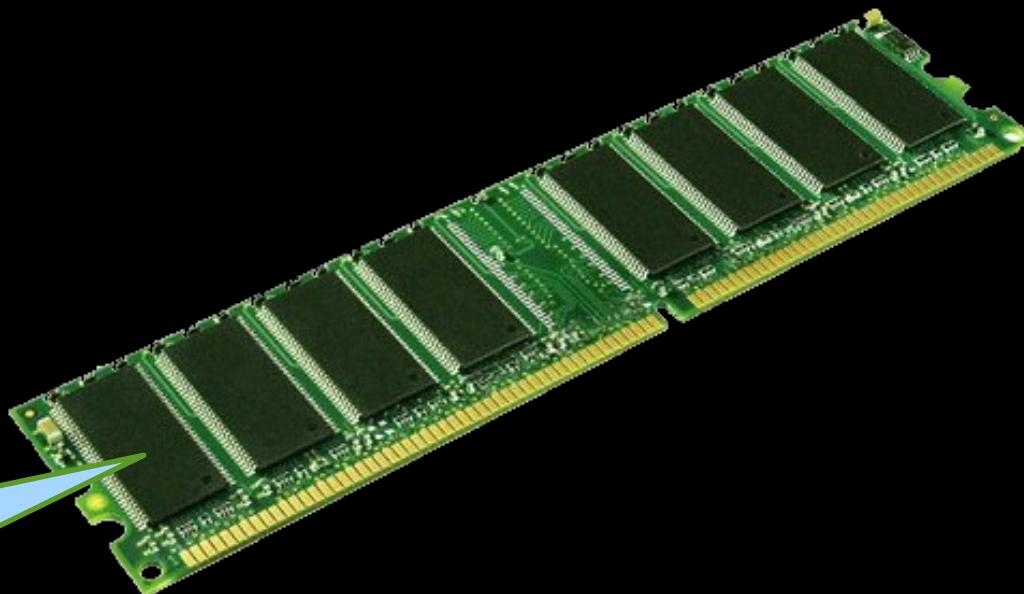
- Register: that plate in front of you
- Cache: the fridge in the kitchen
- Memory: the grocery store downstairs
- Hard disk: the farm in the suburb
- Network: the farm in a different country

Memory and registers

- There are units in the CPU that store multiple data values for use by the CPU:
 - Registers: Small number of fast memory units that allow multiple values to be read and written simultaneously.
 - Main memory: Larger grid of memory cells that are used to store the main information to be processed by the CPU.

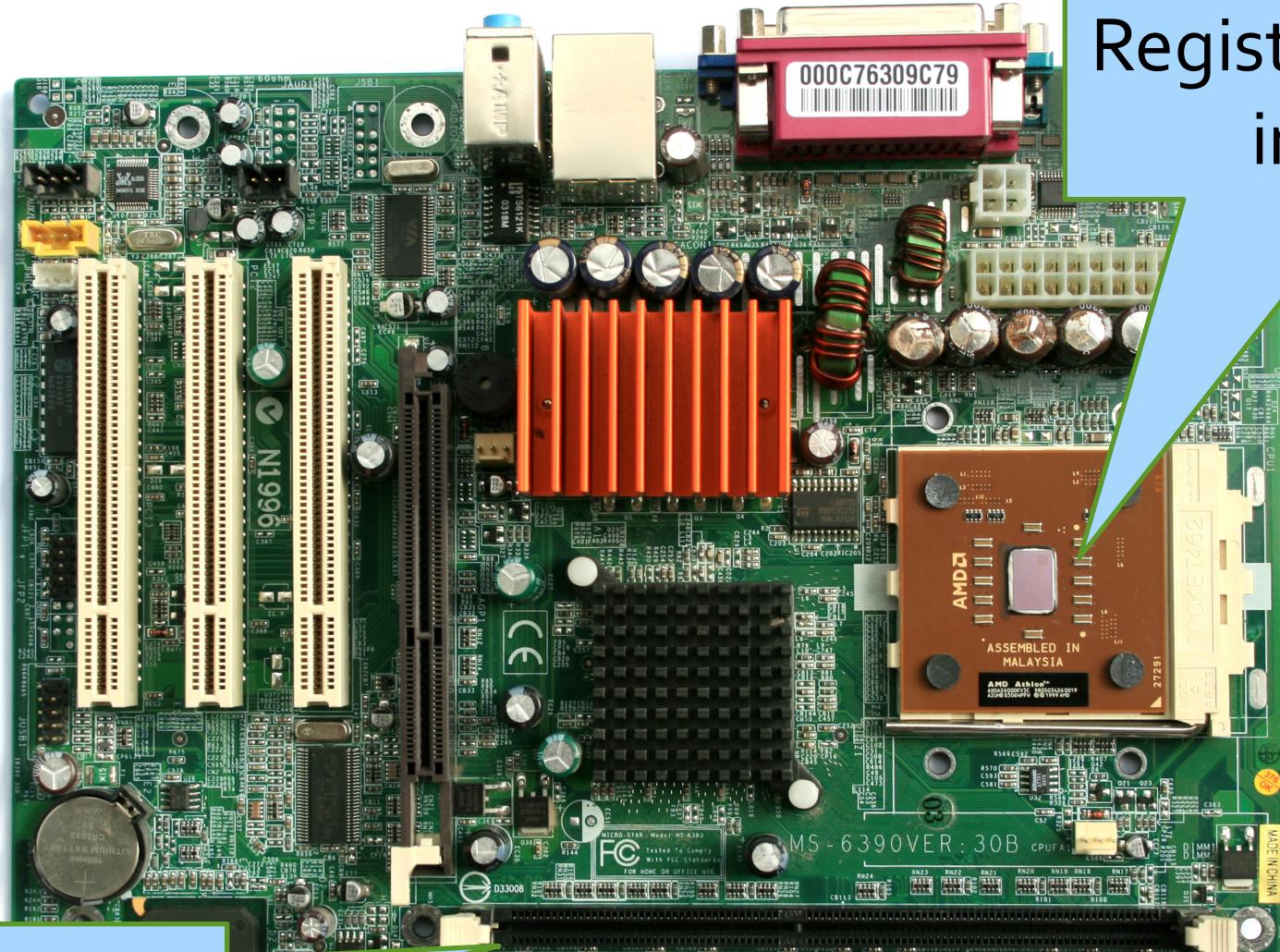


Registers are in here

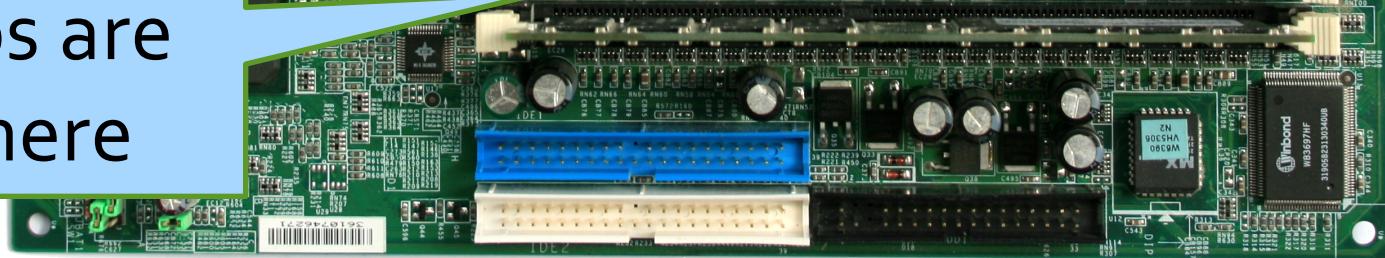


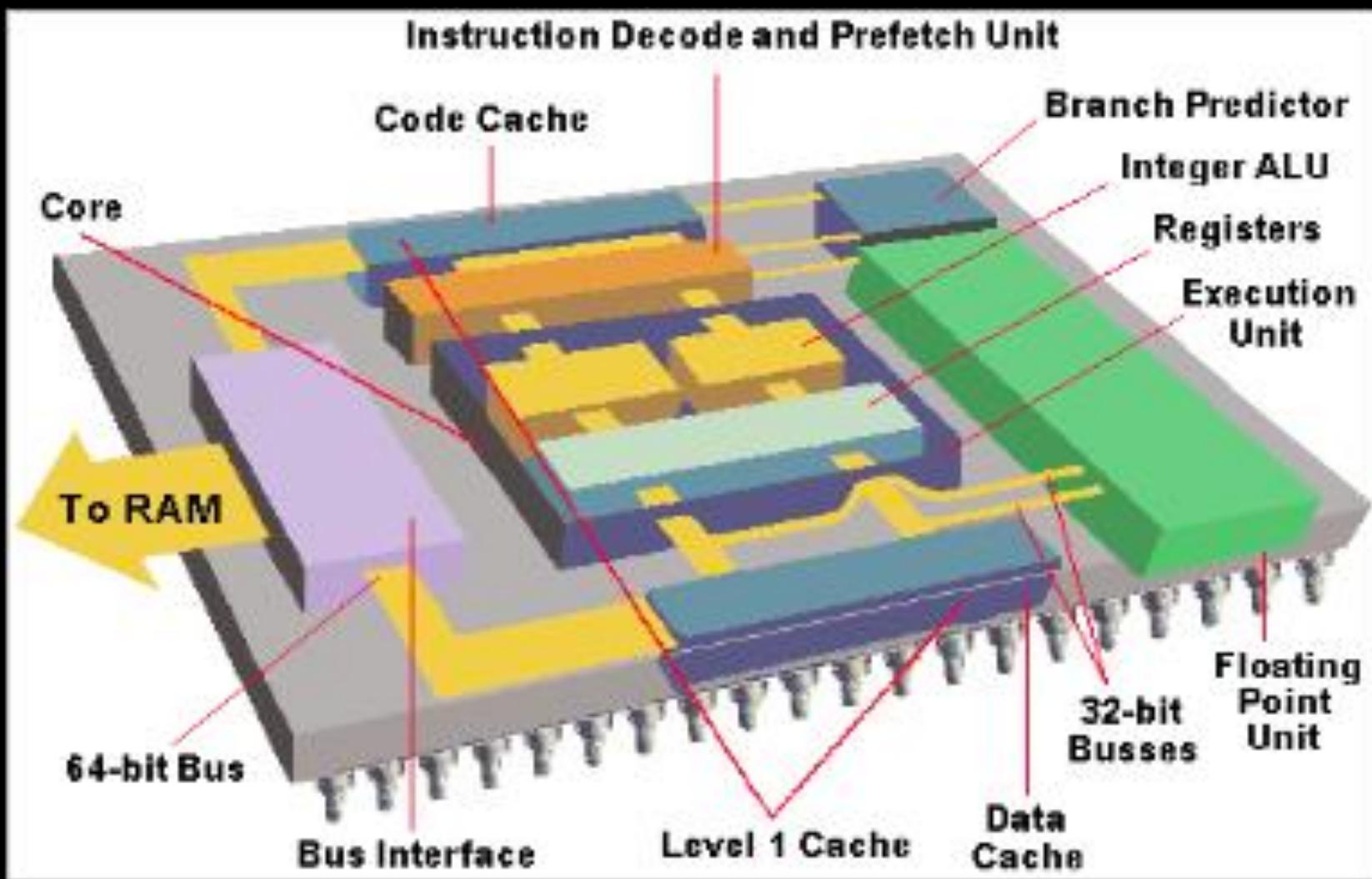
Memory is in here

Registers are in here
in the CPU



Memory chips are
plugged in here

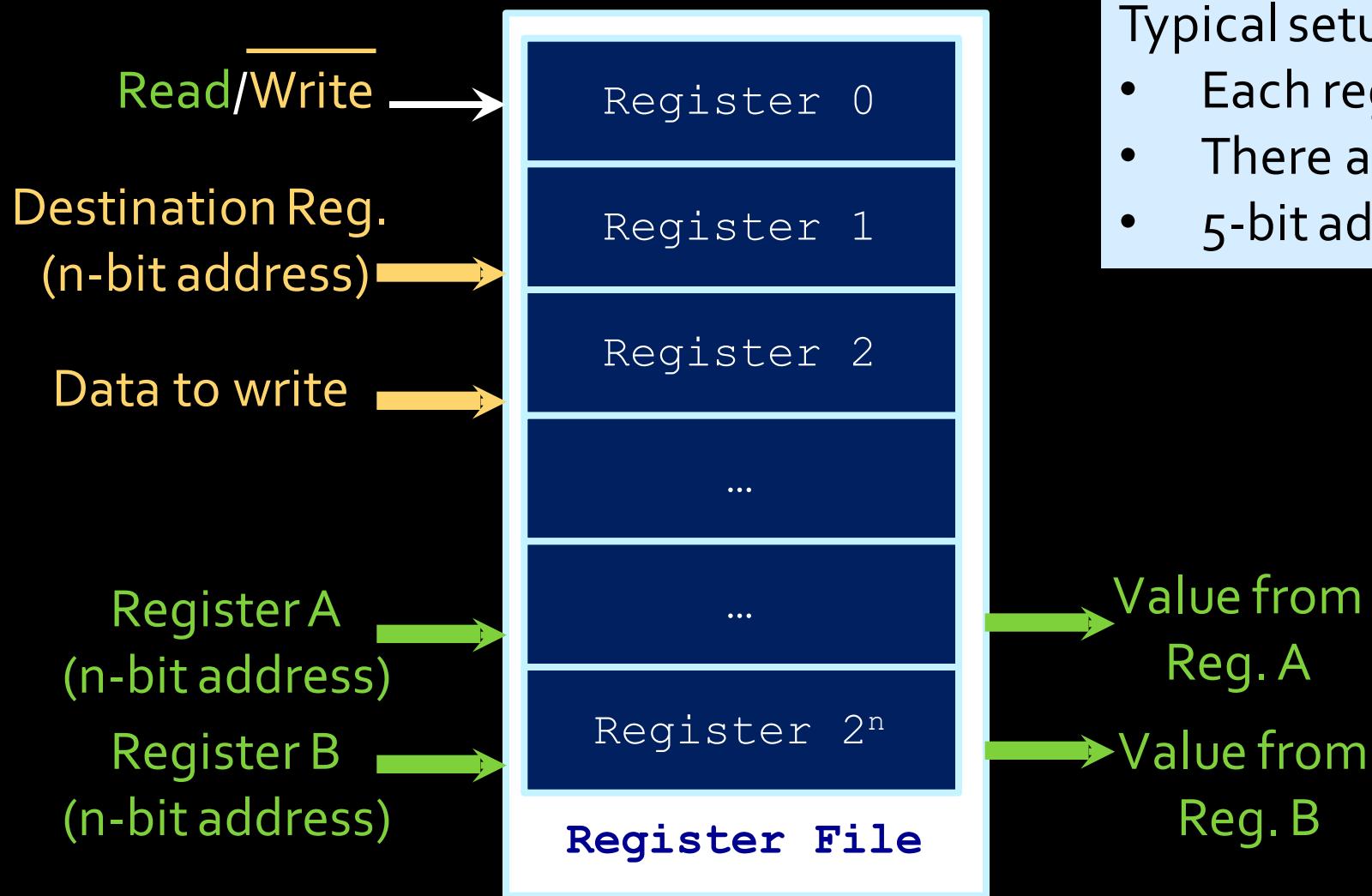




Register file

An array of registers in the CPU

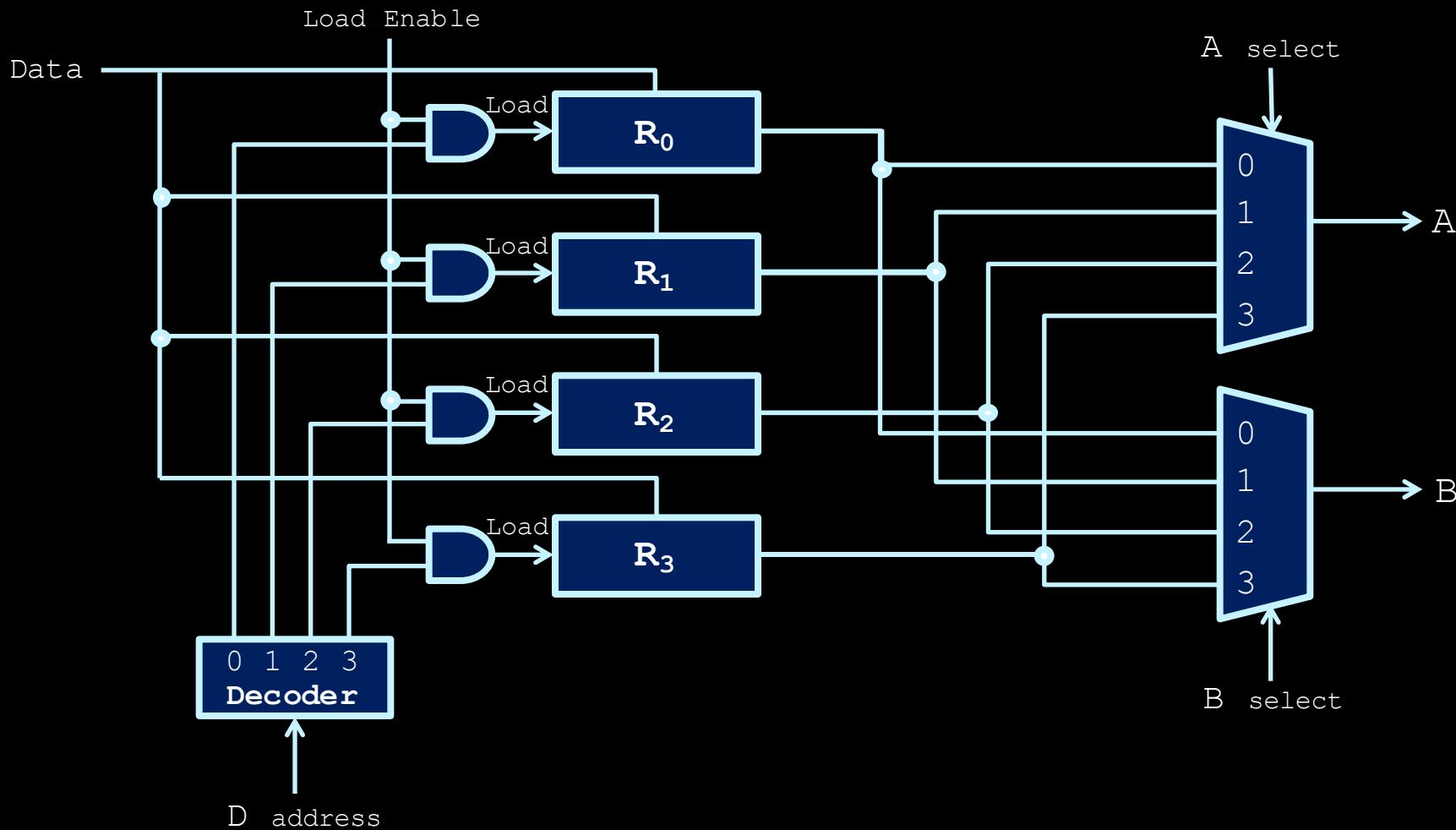
Register File Functionality



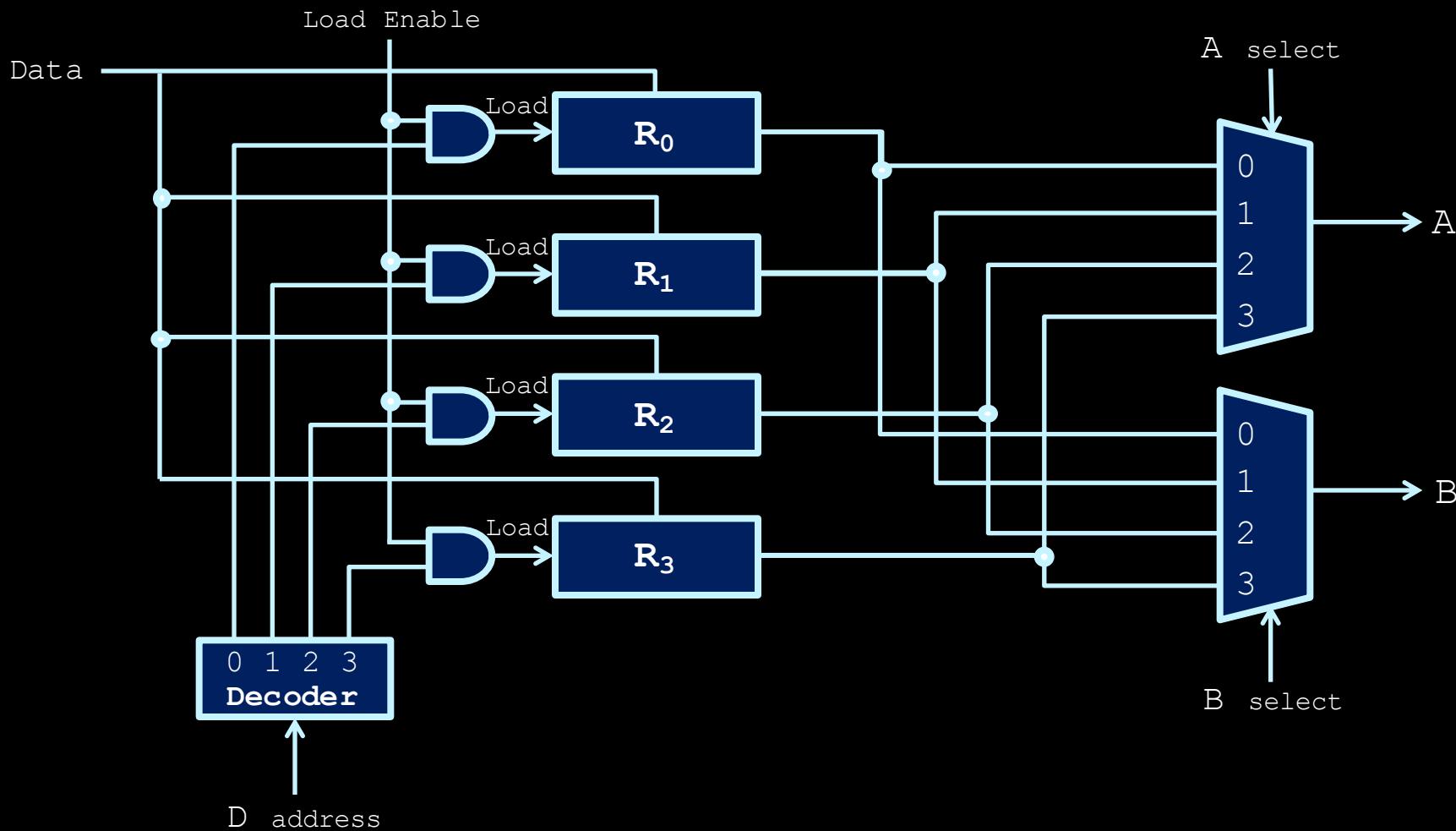
Typical setup (MIPS):

- Each register is 32-bit
- There are 32 registers.
- 5-bit address

Register File - Write Operation



Register File - Read Operation

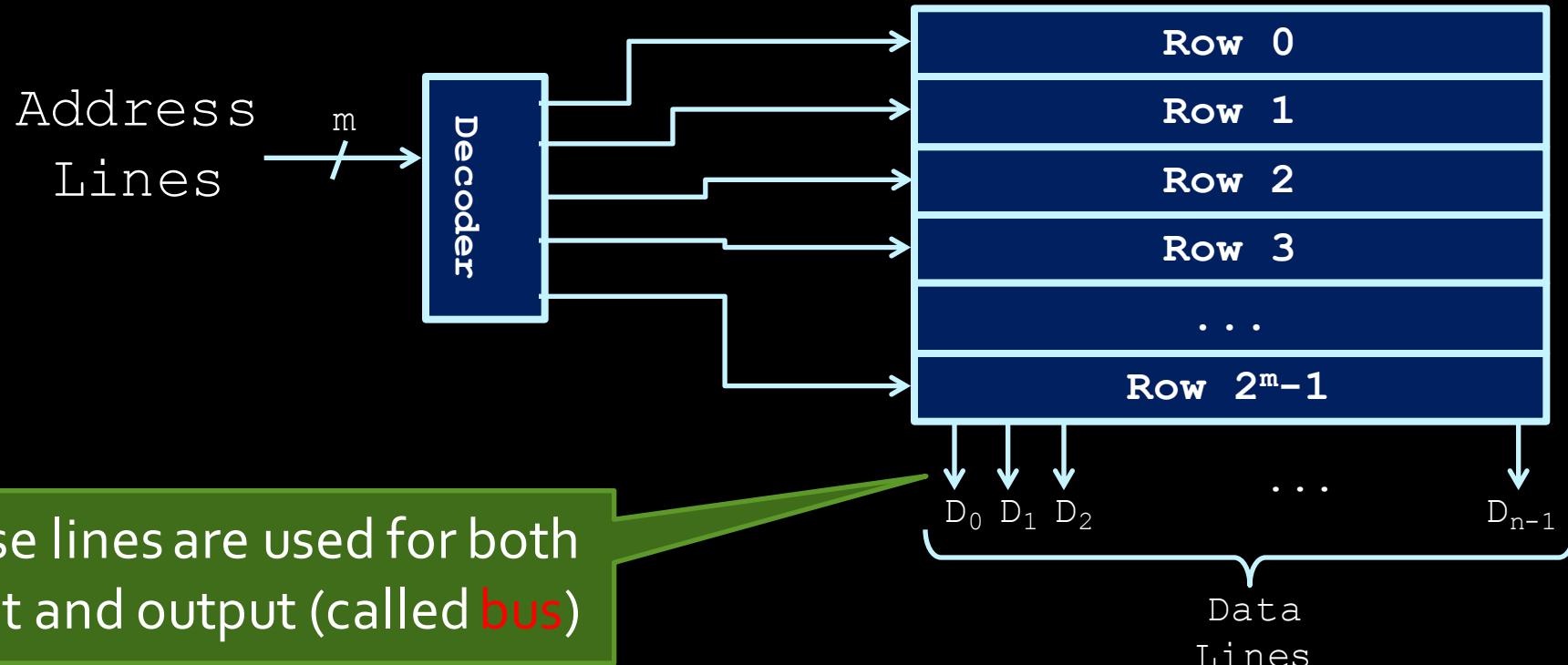


The main memory

An array of memory units

Electronic Memory

- Like register files, main memory is made up of a decoder and rows of memory units.



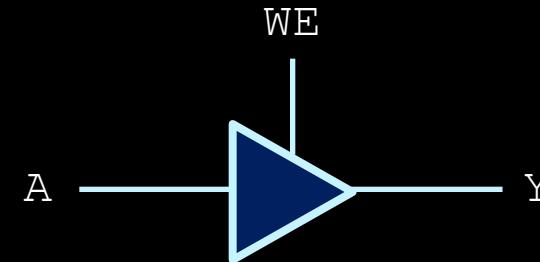
One-hot decoder

- The decoder takes in the m-bit binary address, and activates a single row in the memory array.

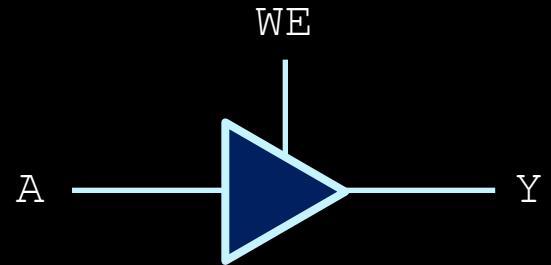
A ₂	A ₁	A ₀	O ₇	O ₆	O ₅	O ₄	O ₃	O ₂	O ₁	O ₀
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
...										
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Controlling the flow

- Since some lines (buses) will now be used for both input and output, we introduce a (sort of) new gate called the **tri-state buffer**.
- When WE (write enable) signal is low, buffer output is a **high impedance** signal.
 - The output is neither connected to high voltage or to the ground.



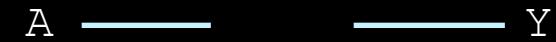
WE	A	Y
0	x	z
1	0	0
1	1	1



$WE = 1$

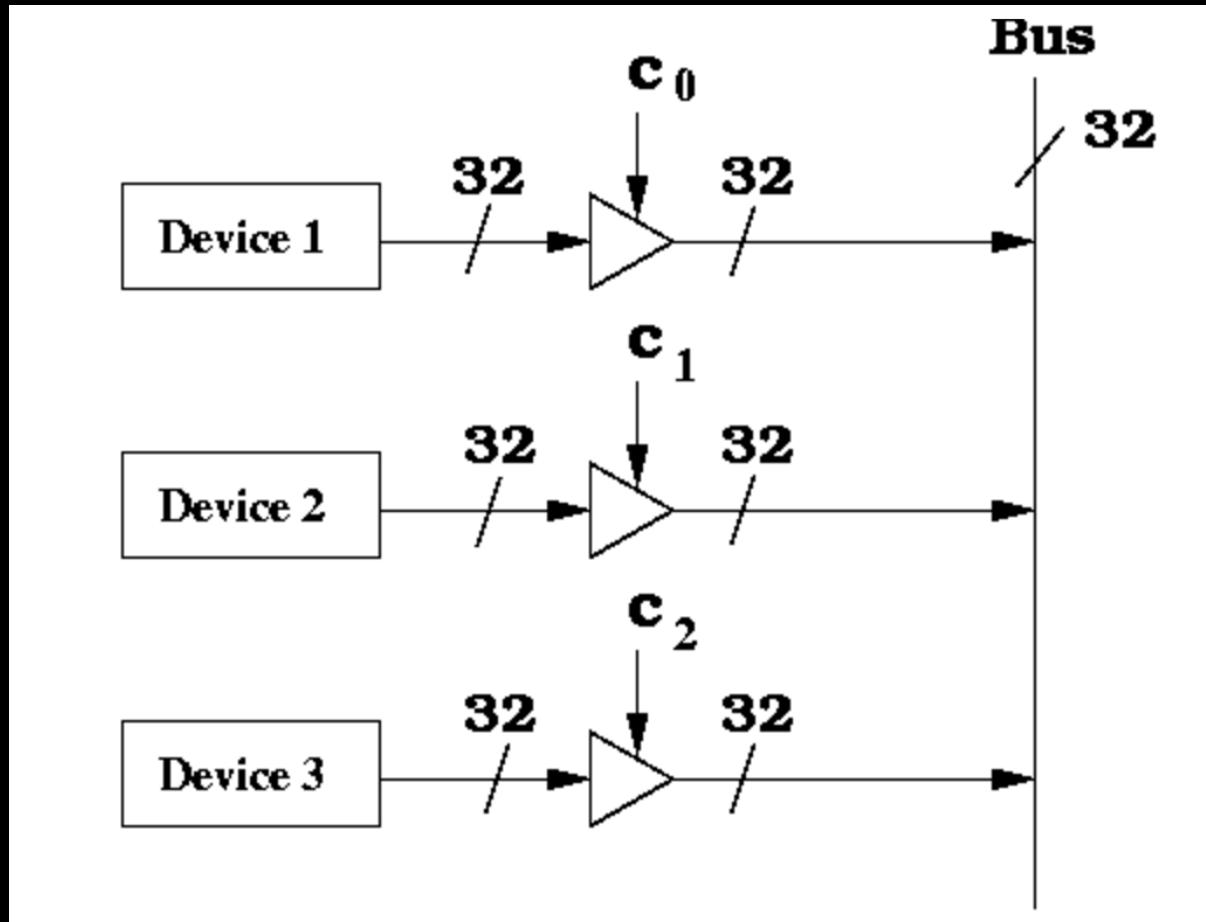


$WE = 0$



WE	A	Y
0	x	z
1	0	0
1	1	1

Control the flow using tri-state buffer



Control c_0 c_1 and c_2 so that only one of the devices output is written to the bus.

In general, the bus can be read by multiple devices at the same time but can only be written by one device at a time.

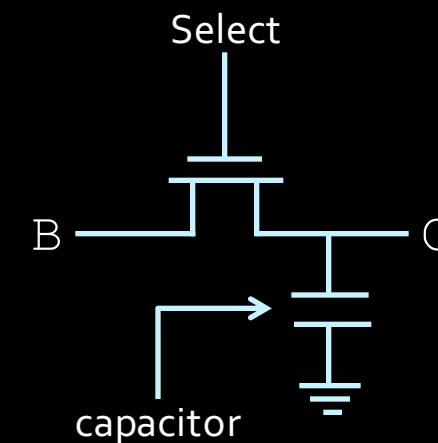
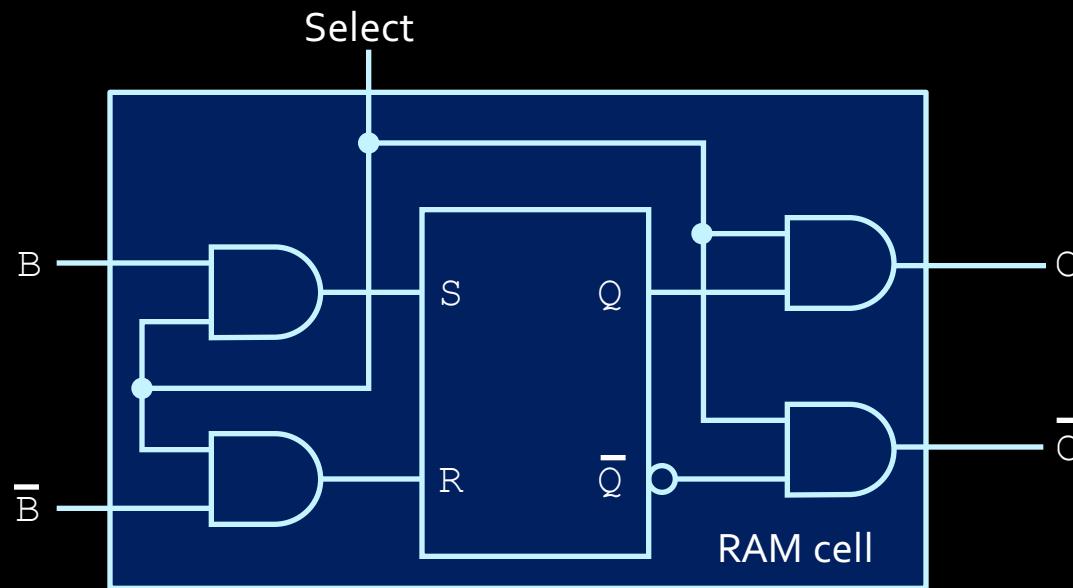
RAM Storage Cell

what stores each bit of the memory

Storage cells

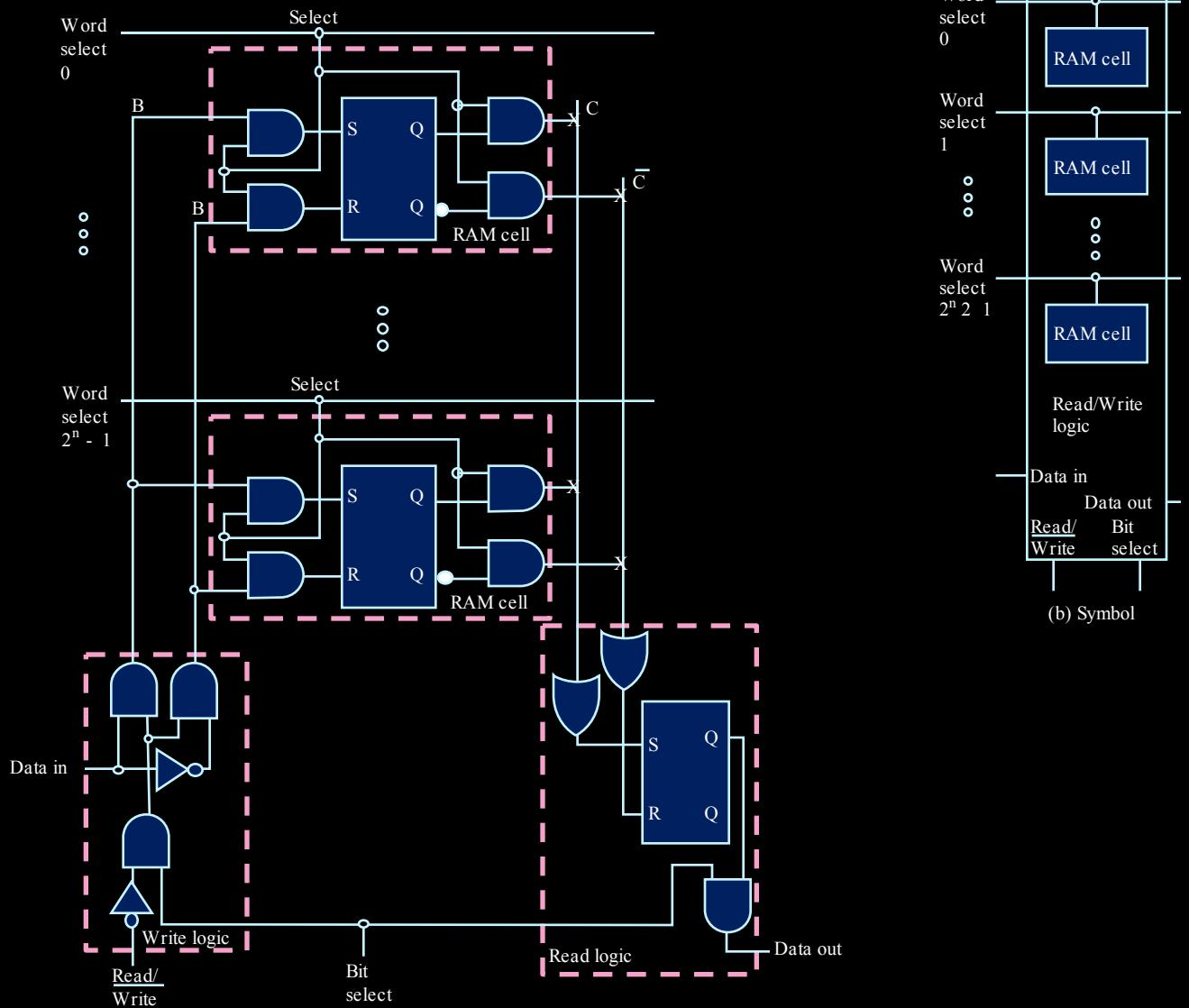
- For storing a single bit
- Each row is made of an array of storage cells.
- Multiple ways of representing these cells.
 - e.g. RAM cell (basically a D-latch):

DRAM IC cell:



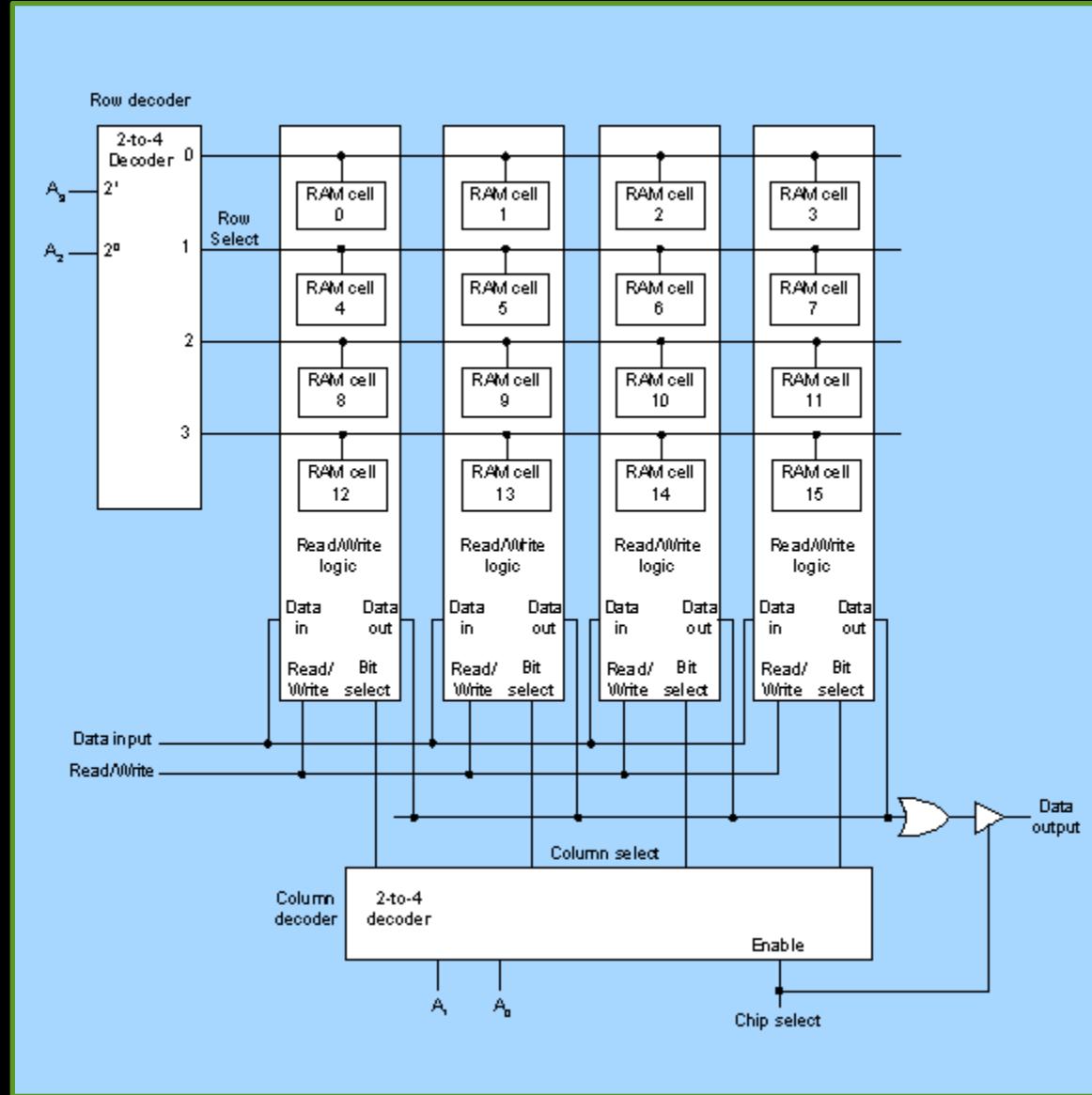
RAM slice model

- Word select signals (via a one-hot decoder) determine which row to send out on the C lines.



RAM slice model

- Or, use word select to choose row, and use bit-select to choose column.



Data Bus

- Communication between components takes place through groups of wires called a **bus** (or **data bus**).
 - Multiple components can read from a bus, but only one can write to a bus at a time.
 - Also called a **bus driver**.
 - Each component has a tri-state buffer that feeds into the bus. When not reading or writing, the tri-state buffer drives high impedance onto the bus.



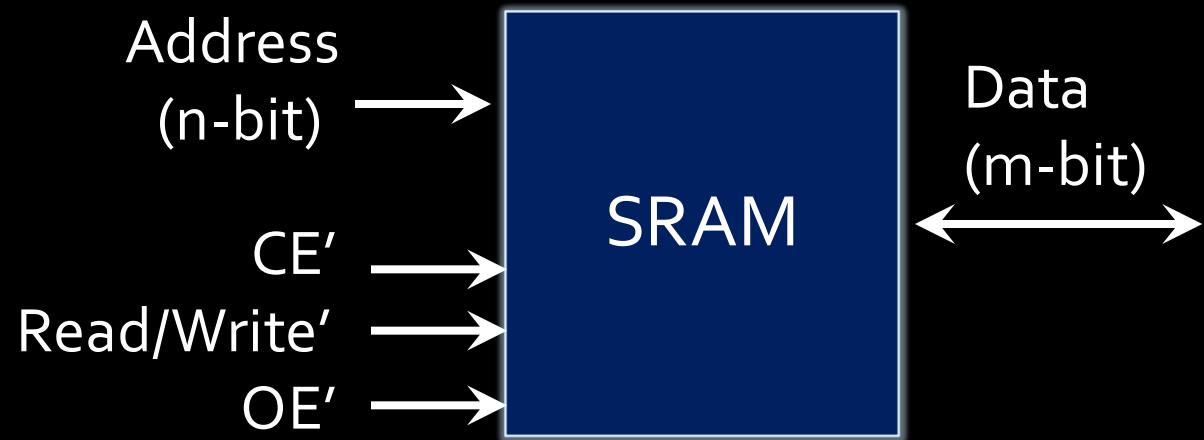
Example:

SRAM

Static Random Access Memory

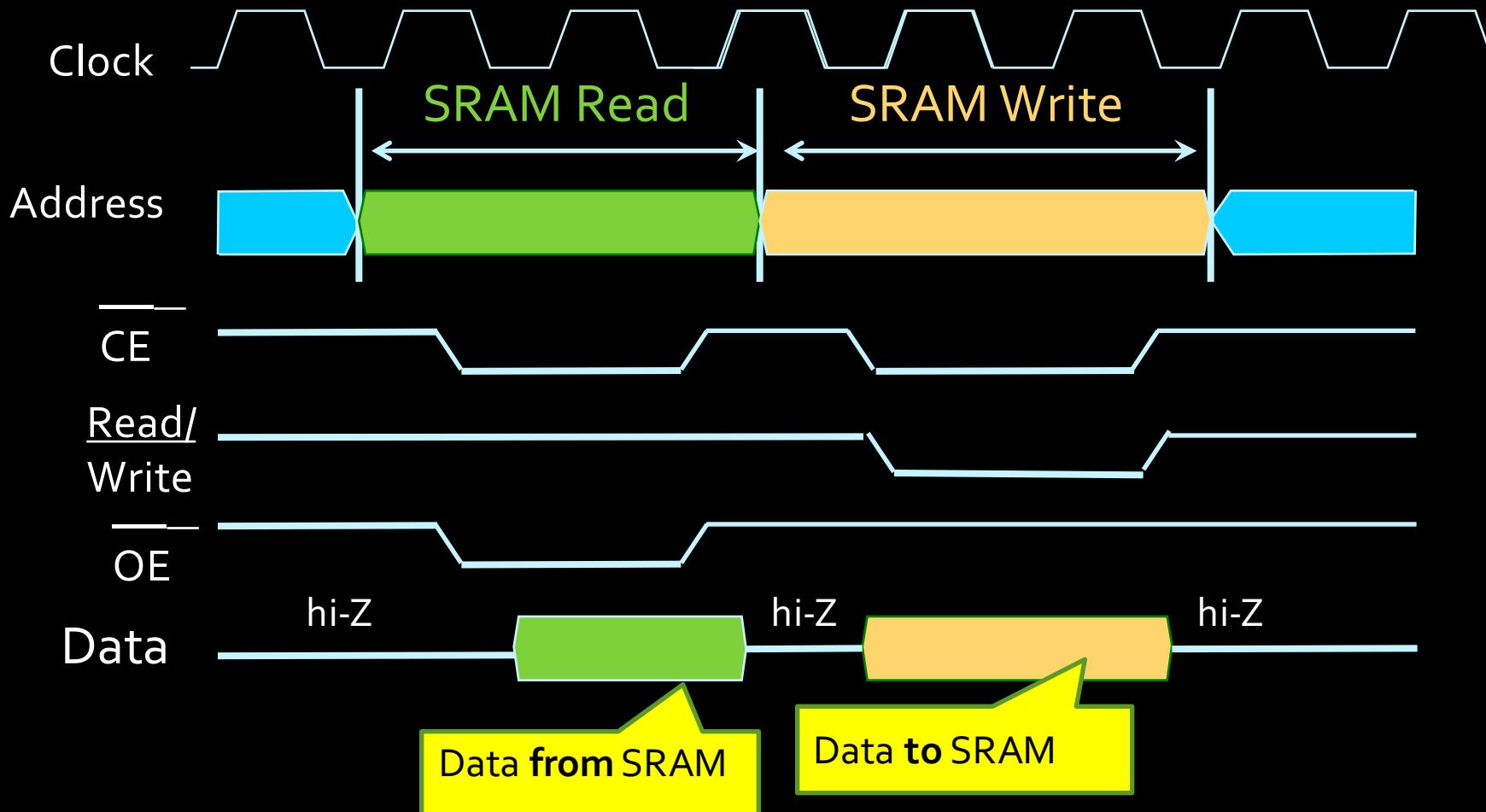
There are other types of RAMs such as DRAM, SDRAM, DDR SDRSM, RDRAM, VRAM, etc.

Asynchronous SRAM Interface - An example



Chip Enable' (CE')	Read/Write'	Output Enable' (OE')	Access Type
0	0	X	SRAM Write
0	1	0	SRAM Read
1	X	X	SRAM not enabled

Read/Write SRAM - Timing waveforms



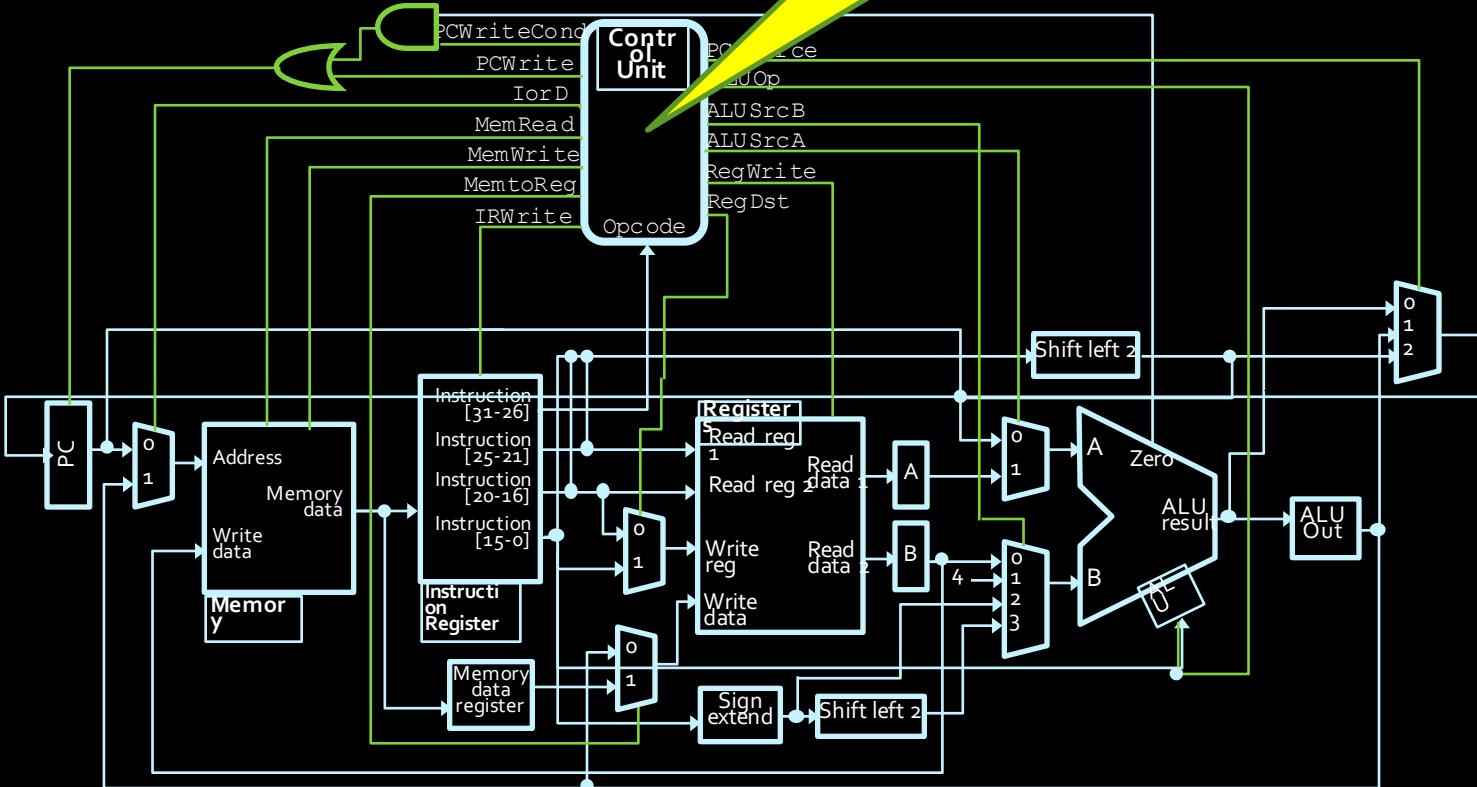
- Reading and writing of signals takes time.

Memory vs registers

- Memory houses most of the data values being used by a program.
- Registers are more local data stores, meant to be used to execute an instruction.
 - Registers are not meant to host memory between instructions (like scrap paper for a calculation).
 - Exception is the stack pointer register, which is sometimes in the same register file as the others.

Next...

The Controller Thing



CSC258 Winter 2016

Lecture 8

Midterm

- Class average 51%, highest mark 43/50
- Class Average including Lab 1-5 and midterm: 73%
- Make sure your midterm mark is correct on MarkUs
- Midterm solution
 - <http://www.cs.toronto.edu/~ylzhang/csc258w16/files/midterm-solution.pdf>
- Remarking request form
 - <http://www.cs.toronto.edu/~ylzhang/csc258w16/files/csc258-midterm-remarking.pdf>

Reflections

What're the reasons why I didn't do as well as I expected?

- Did I do well in the questions that are directly related to labs and quizzes?
- Is there a question that I could have answered better if I had a few minutes more time?
- Is there a question that I could not answer even if I had more time?
- Does the midterm test on a reasonable expectation of how much should be learned from this course?
- Provide feedback, using the feedback form, or just talk to me.
- If you got < 40% in the midterm you should arrange a meeting with me to talk about it how to do better in this course.

- Drop date: March 6

Quiz standings

- Threshold for top 20%: 8 points
- Threshold for top 50%: 5 points

Top Quizzers	
Chris C.	12
Elijah M.	11
Michael Z.	10
Frank Y.	10
Jailani D.	10
Farjad A.	9
Nitharsan K.	9
Alex K.	9
Jay G.	9
Eric C.	9

QUIZ TIME!

Question 1:

A word-addressable RAM unit has 10 address bits going into it. How many bytes is the RAM unit able to store?

A word is 4 bytes

Word-addressable means each word has a unique address.

Question 2:

When reading from RAM, what are the values for CE' and OE'?

CE' = _____

OE' = _____

Question 1:

A word-addressable RAM unit has 10 address bits going into it. How many bytes is the RAM unit able to store?

A word is 4 bytes

Word-addressable means each word has a unique address.

Solution:

Each 4 bytes has a unique address

There are $2^{10} = 1K$ unique addresses

So total size 4 byte \times 1K = 4KB

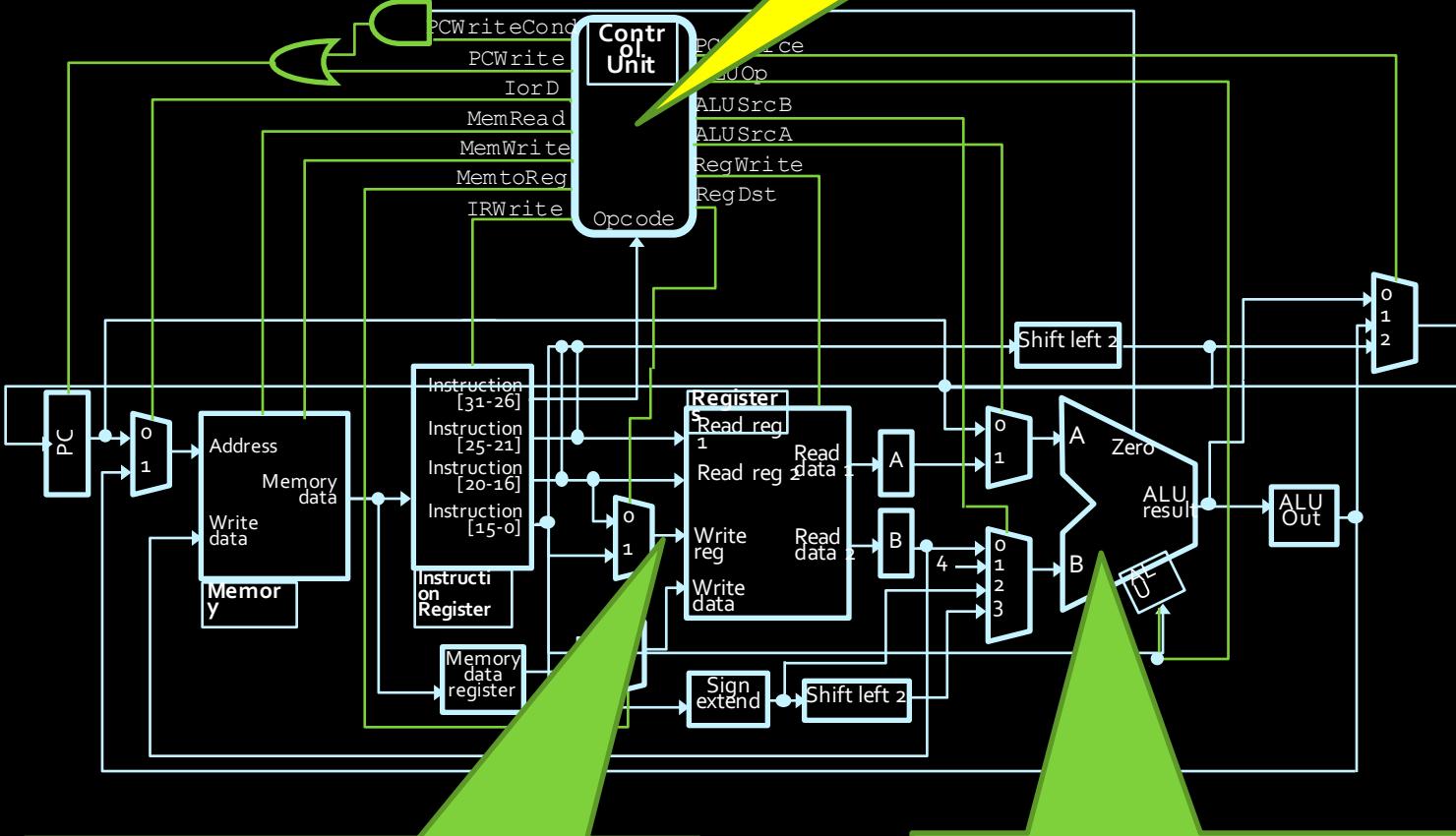
Question 2:

When reading from RAM, what are the values for CE' and OE' ?

$$CE' = \theta \quad OE' = \theta$$

The Blueprint of a microprocessor

The Controller Thing



The Storage Thing

The Arithmetic Thing

The “Controller Thing”

aka: the Control Unit

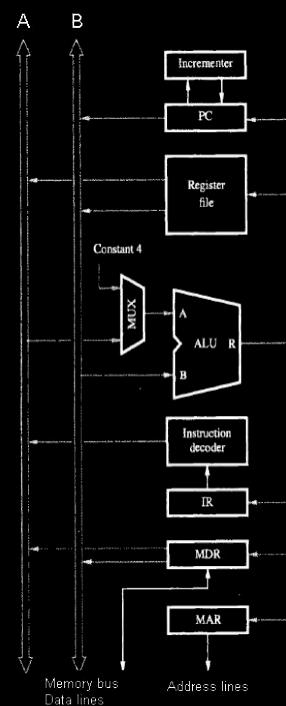
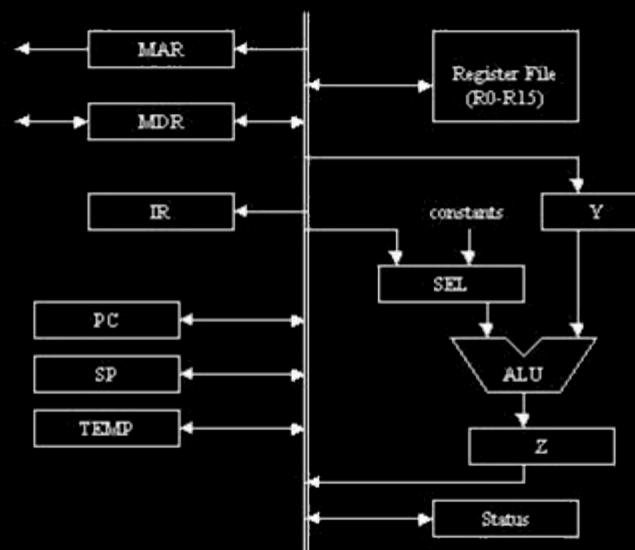


The Control Unit controls how data flows
in the **datapath**.

Different executions have different flows.

Processor Datapath

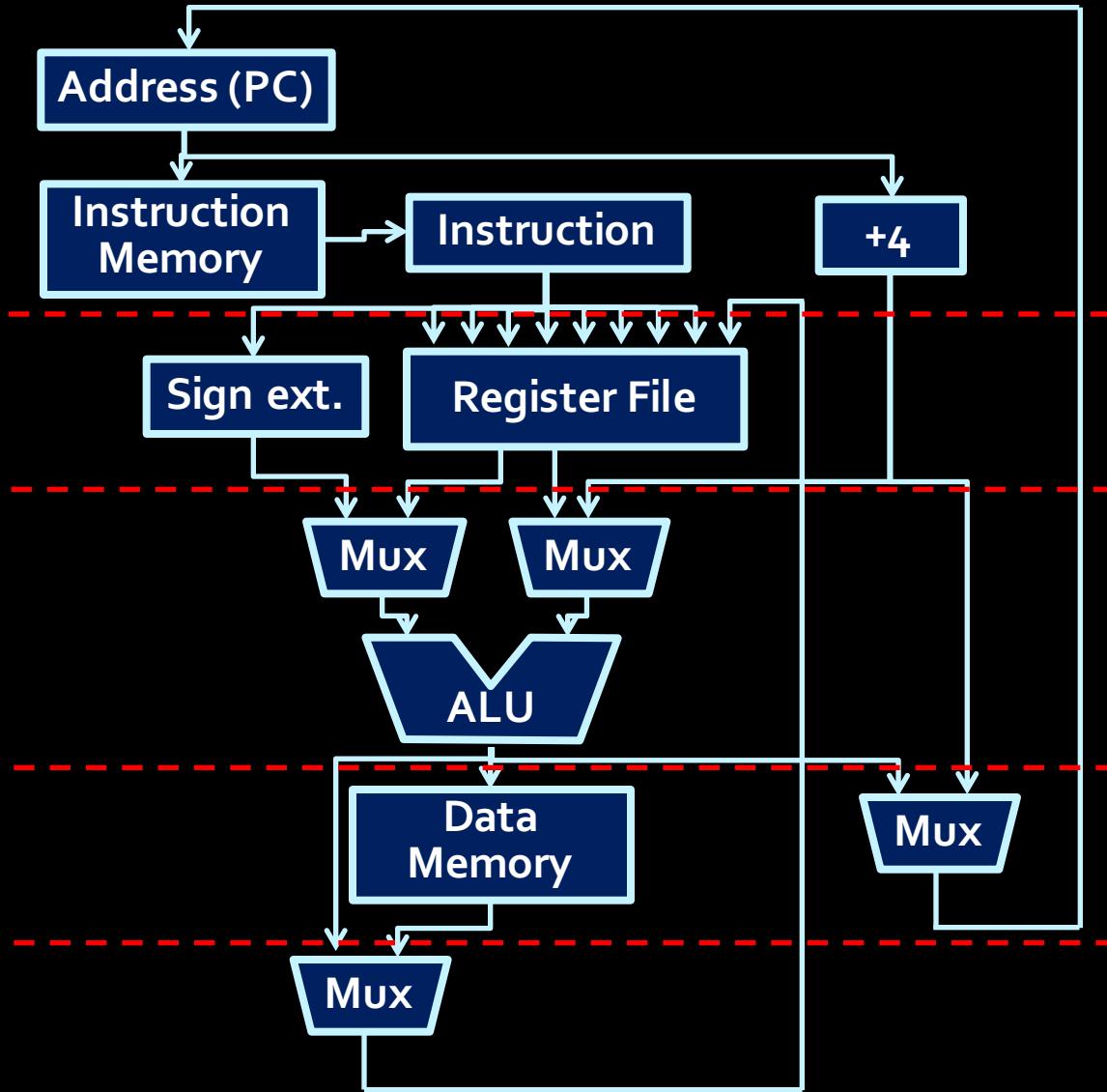
- The **datapath** of a processor is a description/illustration of how the data flows between processor components during the execution of an operation.
 - Examples:



Datapath example

Note: this is just
an abstraction ☺

- The simplified datapath for most processor operations has stages as shown in the diagram:
 - Instruction fetch
 - Instruction decode & register fetch
 - Address/data calculation
 - Memory access
 - Write back.



What happens when you run an executable on your computer?

“Quartus.exe”, “ls”, “minecraft”, ...

1. The OS loads a bunch of instructions into the memory at certain location.
2. CPU finds that location and executes the instructions stored there one by one.

What does an **instruction** look like?

```
00000000 00000001 00111000 00100011
```

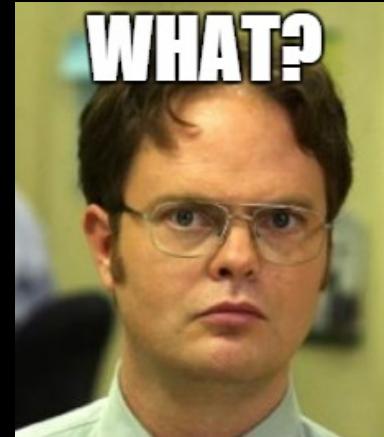
It's a 32-bit (4-byte) binary string.

How do we remember the location of the current instruction?

- The **program counter** (PC) stores the location of the current instruction.
 - Each instruction is 4 bytes long, thus we do +4 to increments the current PC location.
 - PC values can also be loaded from the result of an ALU operation (e.g. jumps to a memory address).

So, here is the instruction. Do it!

```
00000000 00000001 00111000 00100011
```



What does the instruction mean?
What operations do I do?
Where do I get the inputs and put the output?

We need to **decode** the instruction.

Instruction decoding

- The instructions themselves can be broken down into sections that contain all the information needed to execute the operation.
 - Also known as a **control word**.
- Example: unsigned subtraction

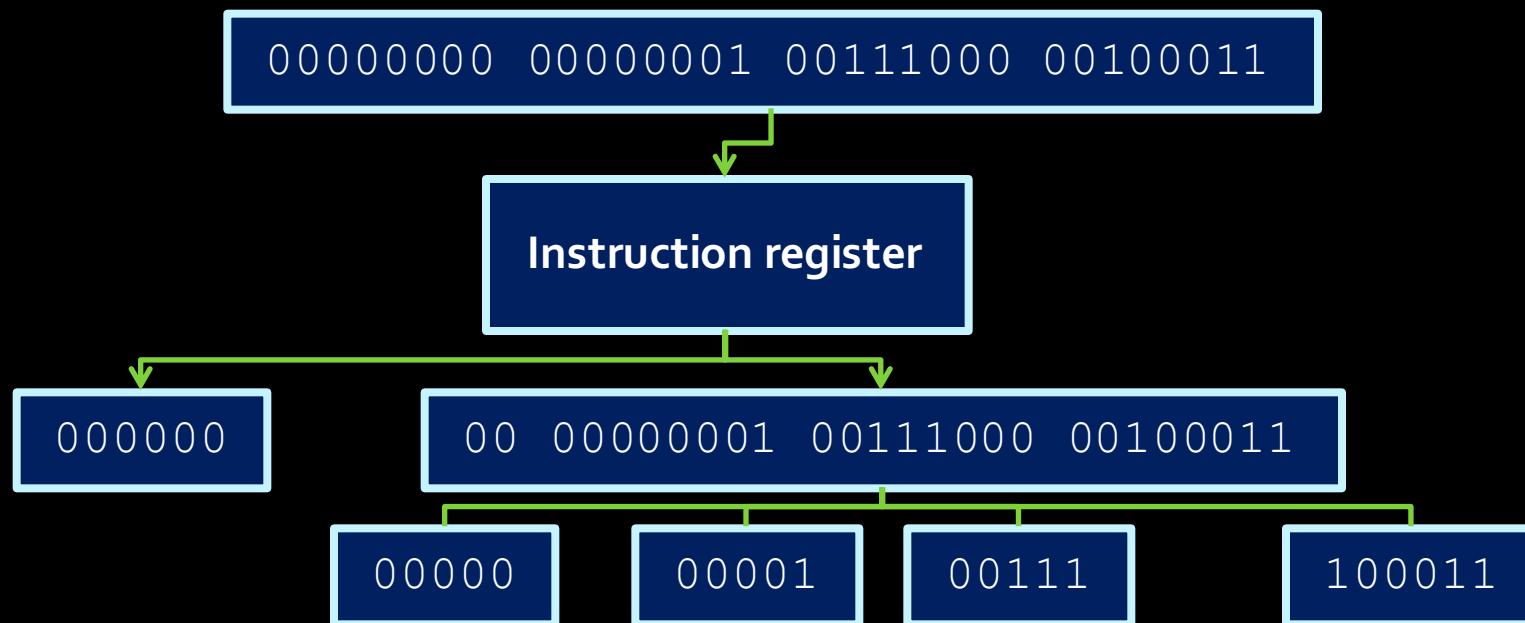
0000000 0000001 00111000 00100011

000000ss sssttttt dddddd000 00100011

Register 7 = Register 0 - Register 1

Instruction registers

- The **instruction register** takes in the 32-bit instruction fetched from memory, and reads the first 6 bits (known as the **opcode**) to determine what operation to perform.



Opcodes

- The first six digits of the instruction (the opcode) will determine the instruction type.
 - Except for “R-type” instructions (marked in yellow)
 - For these, opcode is 000000, and last six digits denote the function.

Instruction	Op/Func	Instruction	Op/Func
add	100000	srav	000111
addu	100001	srl	000010
addi	001000	srlv	000110
addiu	001001	beq	000100
div	011010	bgtz	000111
divu	011011	blez	000110
mult	011000	bne	000101
multu	011001	j	000010
sub	100010	jal	000011
subu	100011	jalr	001001
and	100100	jr	001000
andi	001100	lb	100000
nor	100111	lbu	100100
or	100101	lh	100001
ori	001101	lhu	100101
xor	100110	lw	100011
xori	001110	sb	101000
sll	000000	sh	101001
sllv	000100	sw	101011
sra	000011	mflo	010010

MIPS instruction types

- R-type:



- I-type:



- J-type:



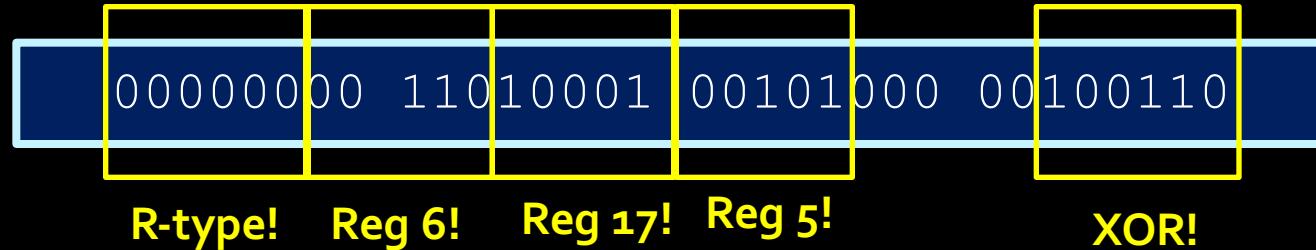
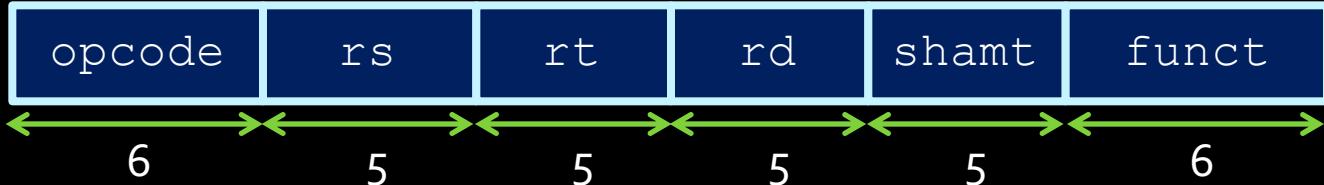
Read the first 6 bits first, then you know how to break it down.

R-type instructions



- Short for “register-type” instructions.
 - Because they operate on the registers, naturally.
- These instructions have fields for specifying up to three registers and a shift amount.
 - Three registers: two source registers (`rs` & `rt`) and one destination register (`rd`).
 - A field is usually coded with all 0 bits when not being used.
- The opcode for all R-type instructions is **000000**.
- The function field specifies the type of operation being performed (add, sub, and, etc).

Examples



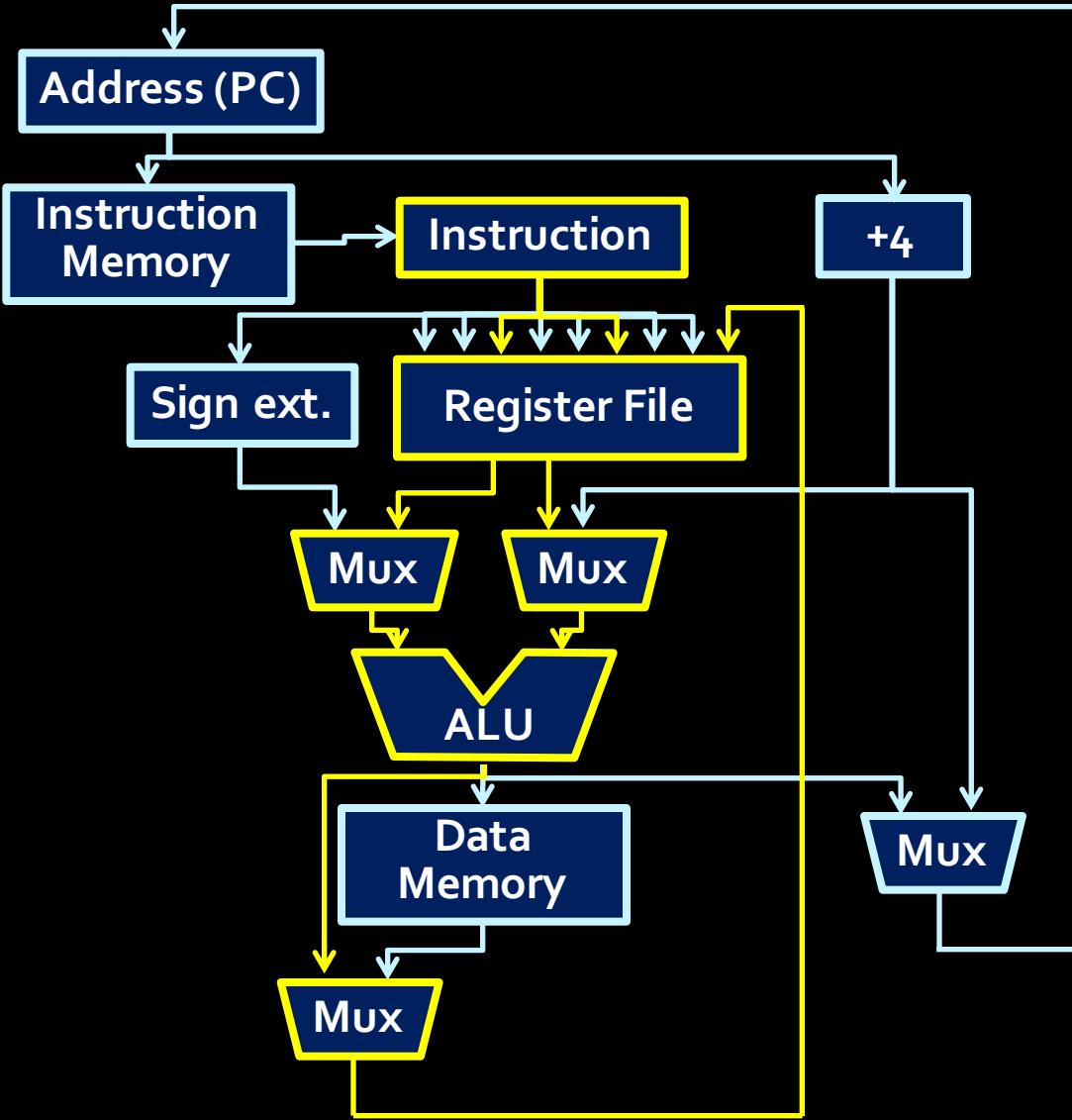
Reg_5 = Reg_6 XOR Reg_17



Left shift what's in Reg_17 by 12 bits
and store result in Reg_5

R-type instruction datapath

- For the most part, the funct field tells the ALU what operation to perform.
- rs and rt are sent to the register file, to specify the ALU operands.
 - Register \$0 and \$1 are usually held in reserve.
- rd is also sent to the register file, to specify the location of the result.



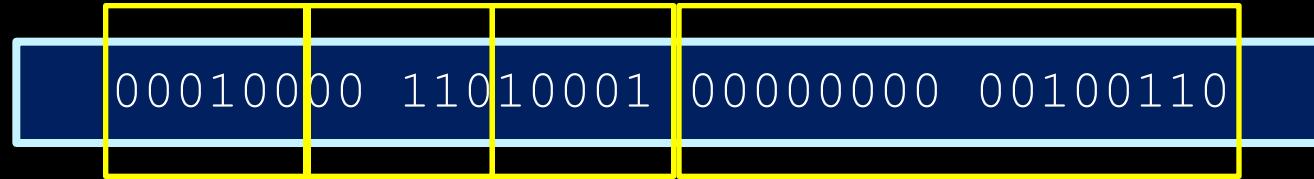
I-type instructions



- These instructions have a 16-bit **immediate** field.
- This field has a constant value, which is used for:
 - an immediate operand,
 - a branch target offset (e.g., in **branch if equal** op), or
 - a displacement for a memory operand (e.g., in **load** op).
- For **branch** target offset operations, the immediate field contains the signed difference between the current address stored in the PC and the address of the target instruction.
 - This offset is stored with the two low order bits dropped. The dropped bits are always 0 since instructions are **word-aligned**.

Word-aligned: Offsets increment by 4, like 0 (0000), 4 (0100), 8 (1000), 12 (1100), note that the two lowest bits are always 00.

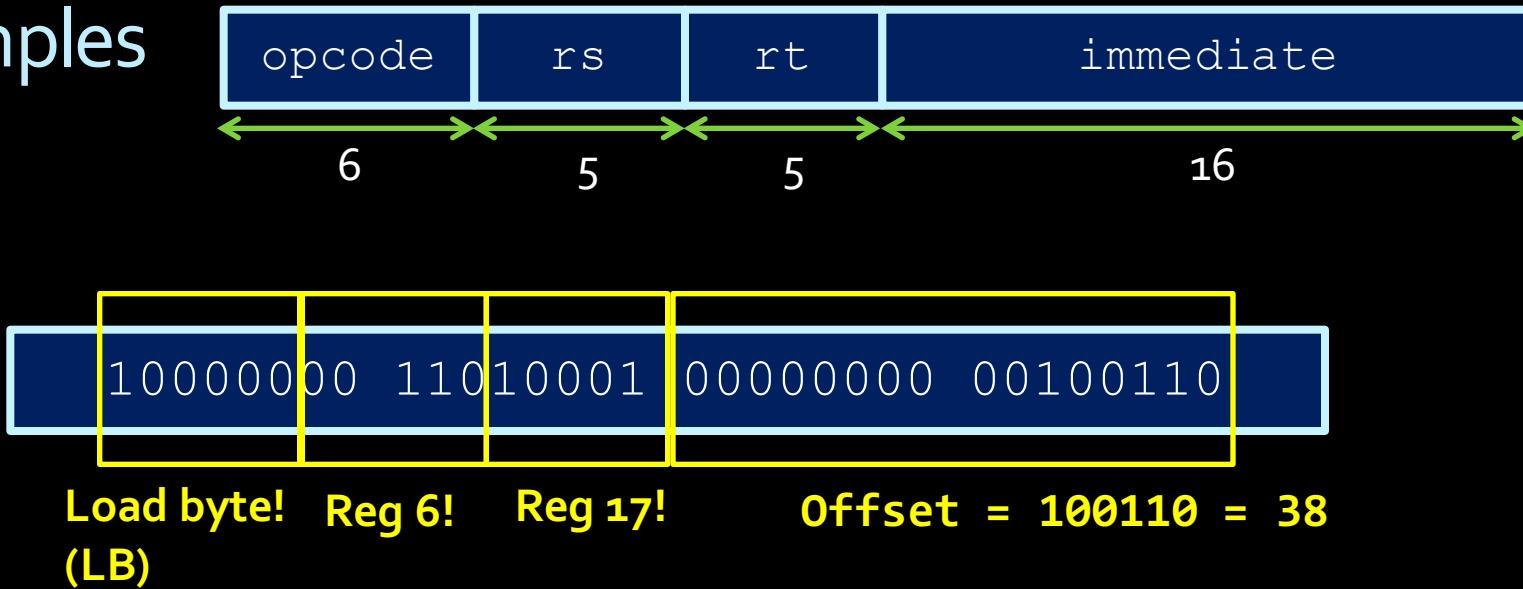
Examples



Branch on equal **Reg 6!** **Reg 17!** **Offset = 10011000 = 152**
(BEQ)

```
If Reg_6 == Reg_17:  
    PC += 152  
Else:  
    PC += 4
```

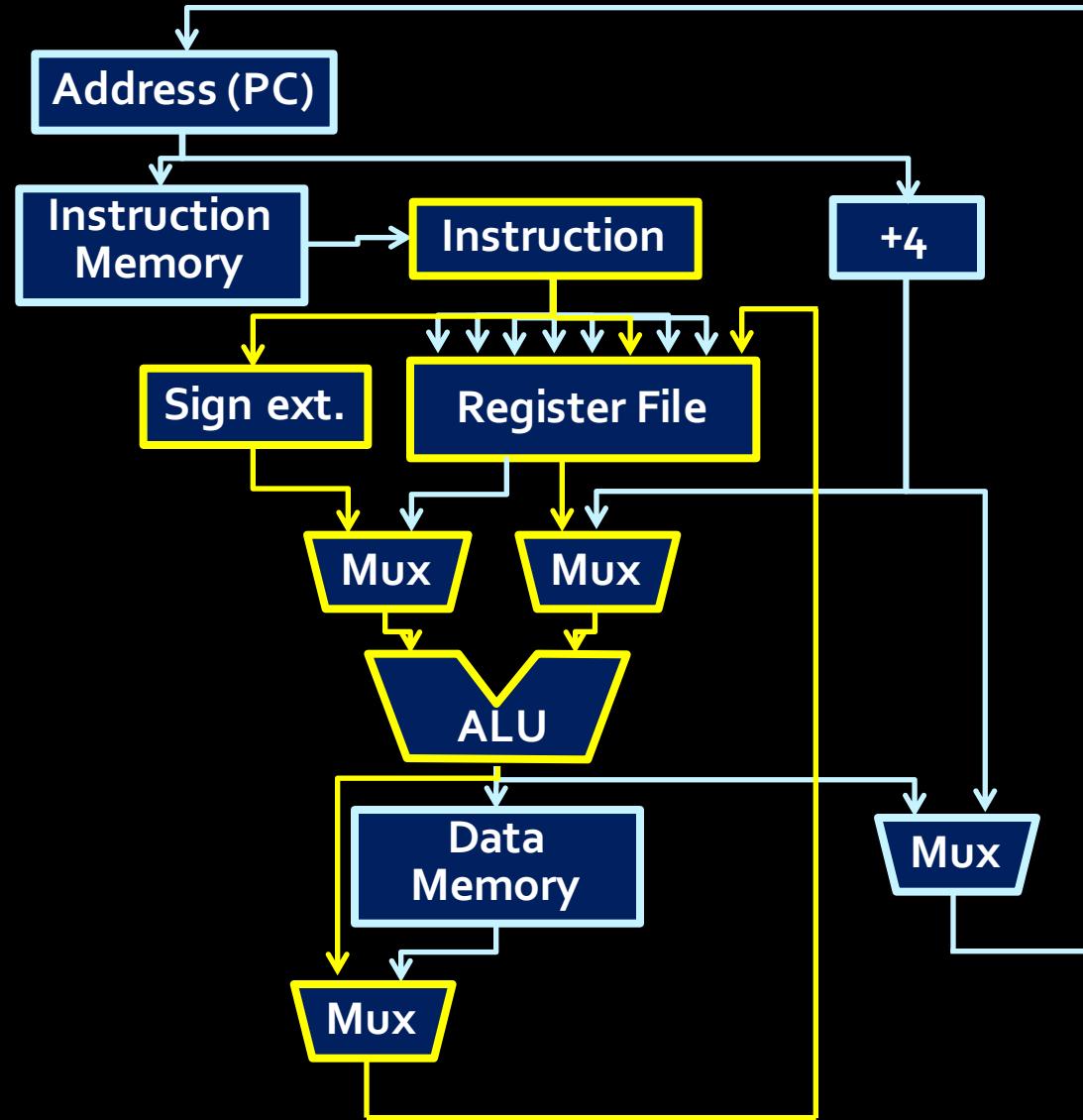
Examples



Load one byte from $\text{MEM}[\text{Reg_6}+38]$ to Reg_17

I-type instruction datapath

- Example #1:
Immediate
arithmetic
operations,
with result
stored in
registers.



Interlude: Sign extension

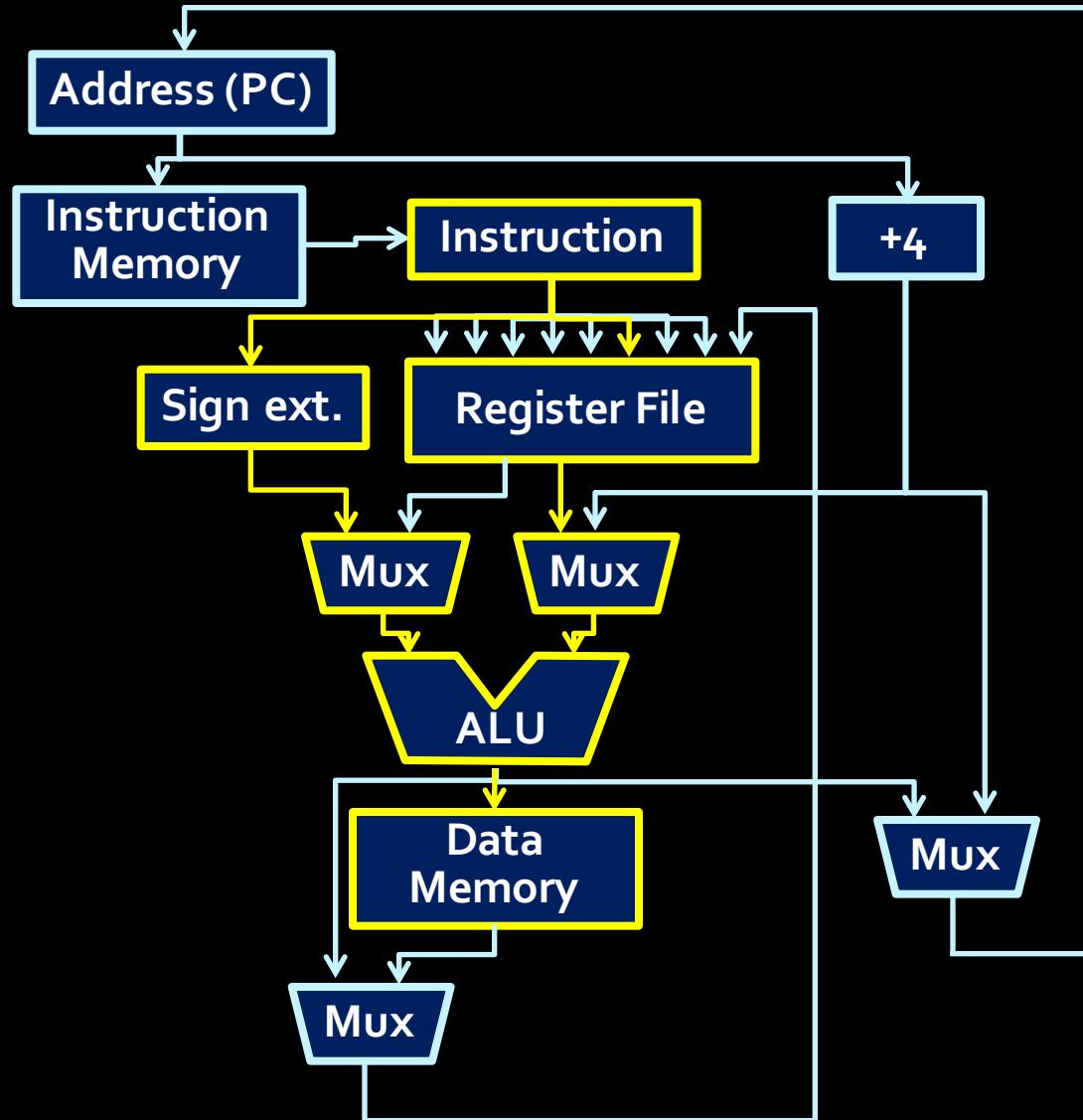
Sign ext.

- The immediate value we get from an I-type instruction is 16-bit long.
- But all operands of ALU are supposed to be 32-bit long.
- So fill the upper 16 bits of the number with the **sign-bit**

- E.g., 1100 1000 1000 1000 becomes
- 1111 1111 1111 1111 1100 1000 1000 1000

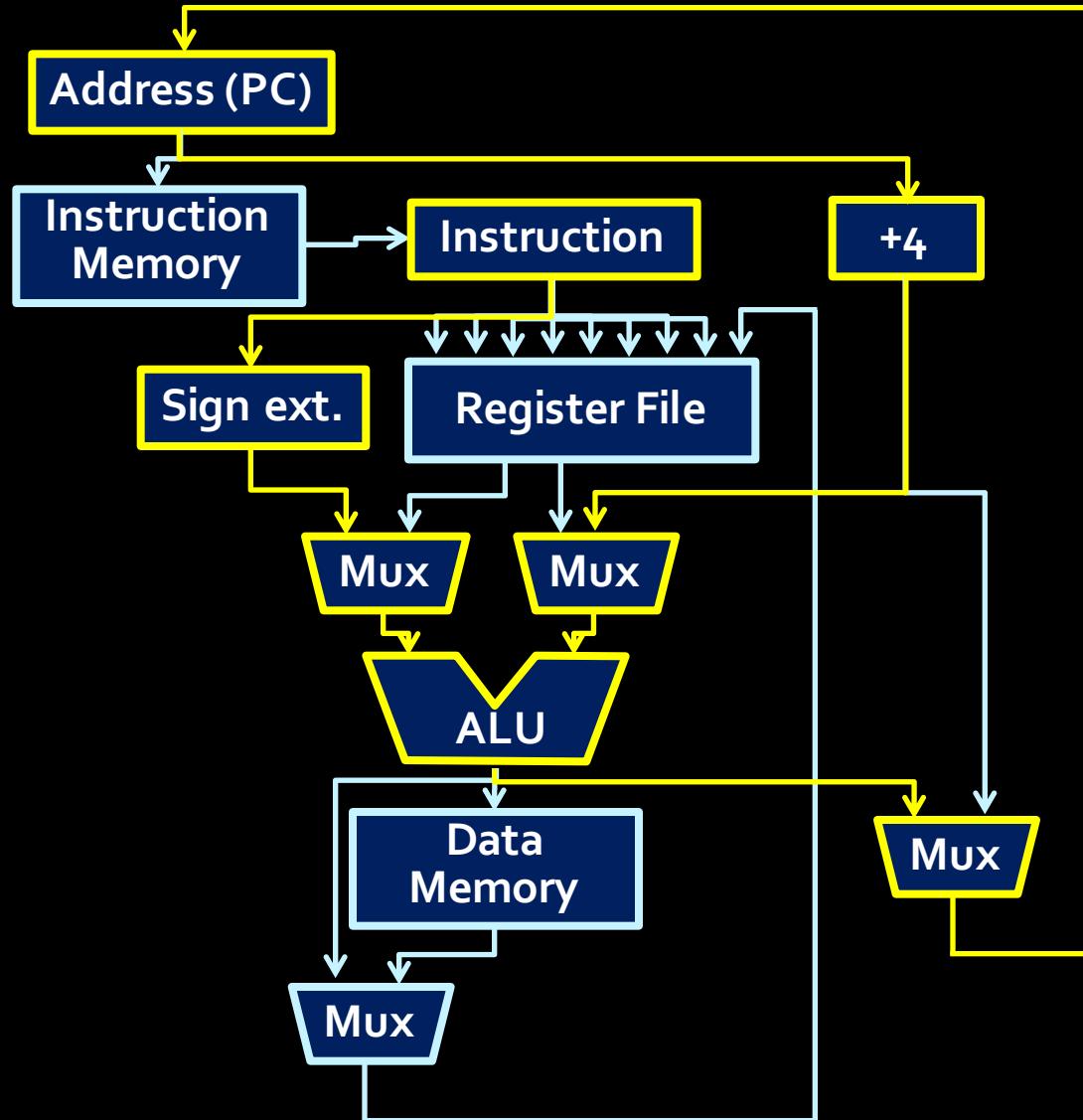
I-type instruction datapath

- Example #2:
Immediate
arithmetic
operations,
with result
stored in
memory.



I-type instruction datapath

- Example #3:
Branch
instructions.
 - Output is written to PC, which looks to that location for the next instruction.

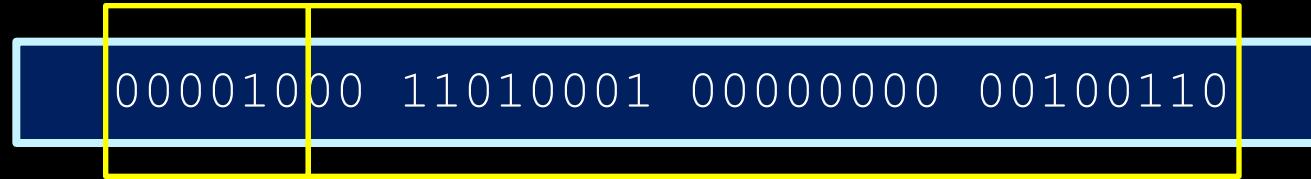


J-type instructions



- Only two J-type instructions:
 - jump (`j`)
 - jump and link (`jal`)
- These instructions use the 26-bit coded address field to specify the target of the jump.
 - The first four bits of the destination address are the same as the current bits in the program counter.
 - The bits in positions 27 to 2 in the address are the 26 bits provided in the instruction.
 - The bits at positions 1 and 0 are always 0 since instructions are word-aligned.

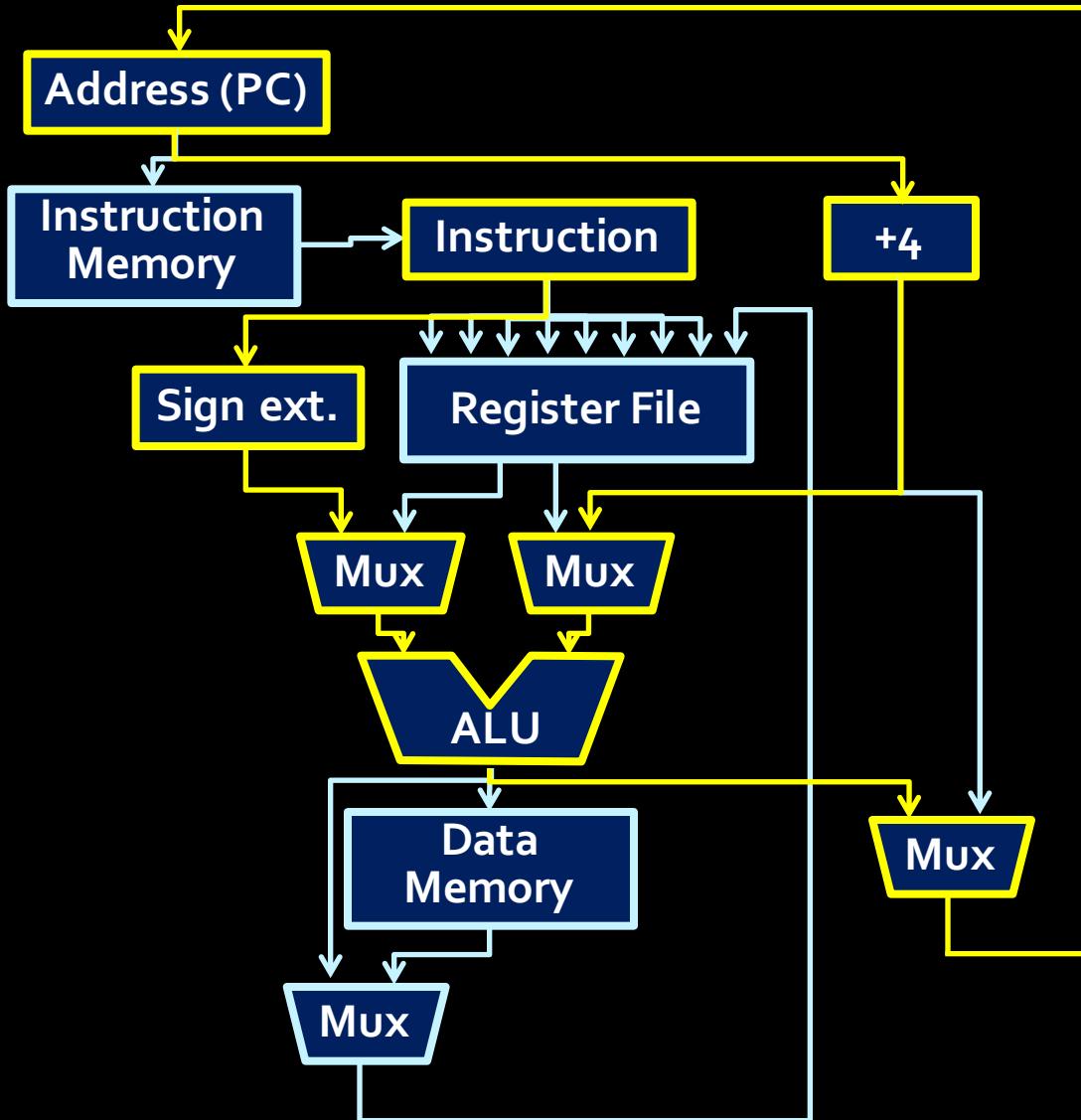
Examples



PC jumps to address
{PC[31:28] (4 bits), what's above (26 bits), 00 (2 bits)}
(32 bits total)

J-type instruction datapath

- Jump and branch use the datapath in similar but different ways:
 - Branch calculates new PC value as old PC value + offset. (relative)
 - Jump loads an immediate value over top of the old PC value. (absolute)



Takeaway

Different instructions flow in different datapaths.

In other words, if we can **control the paths of flow**, then we can control what instruction to execute.

Datapath control

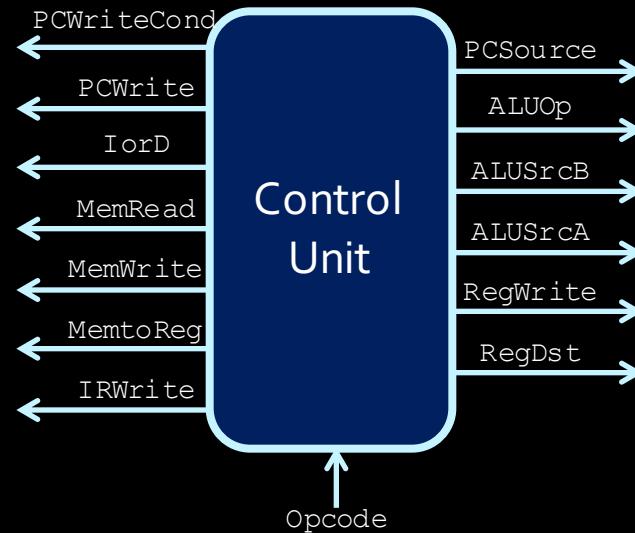
Datapath control

- These instructions are executed by turning various parts of the datapath on and off, to direct the flow of data from the correct source to the correct destination.
- What tells the processor to turn on these various components at the correct times?

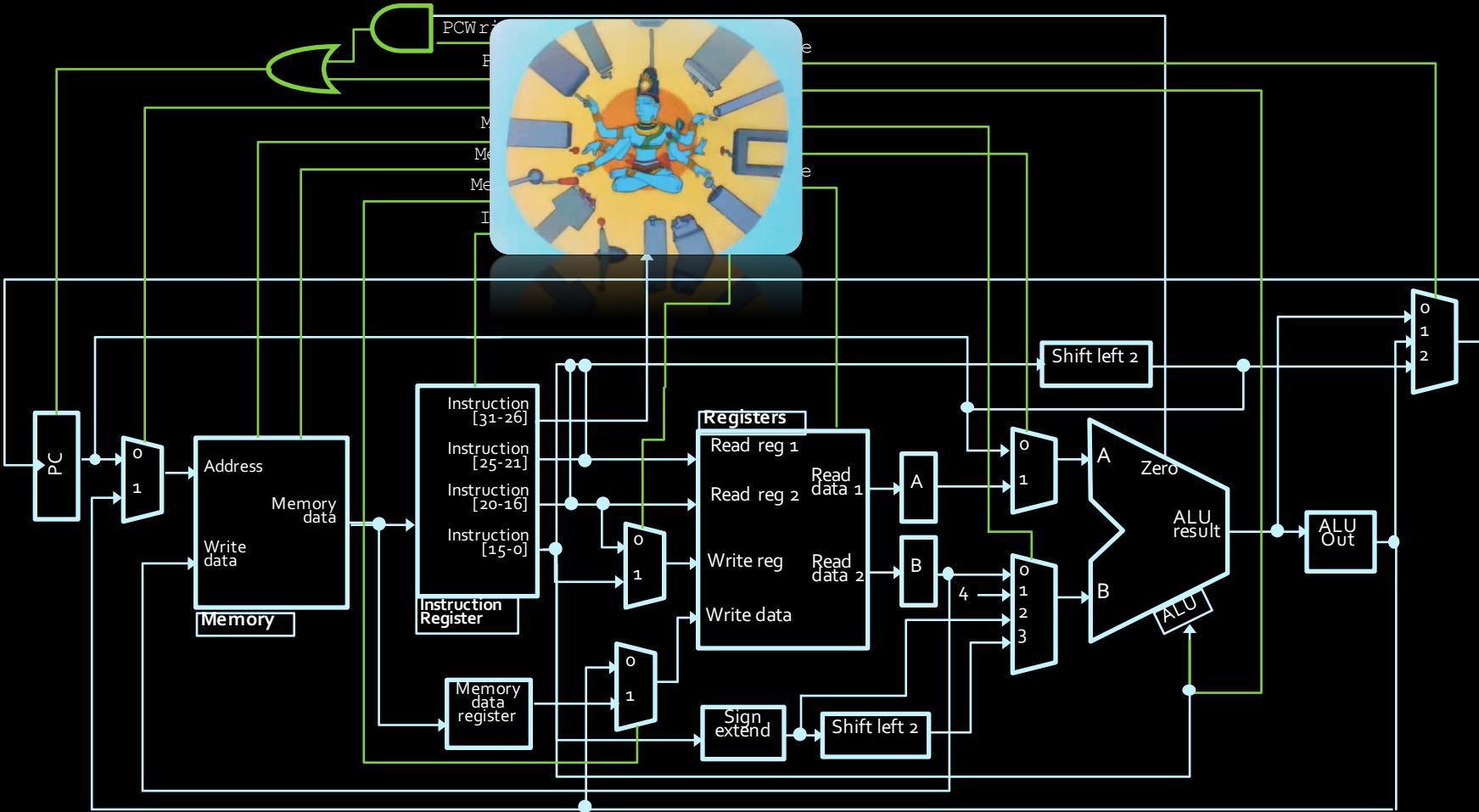


Control unit

- The control unit takes in the **opcode** from the current instruction, and sends **signals** to the rest of the processor.
- Within the control unit is a **finite state machine** that can occupy multiple clock cycles for a single instruction.
 - The control unit send out different signals on each clock cycle, to make the overall operation happen.

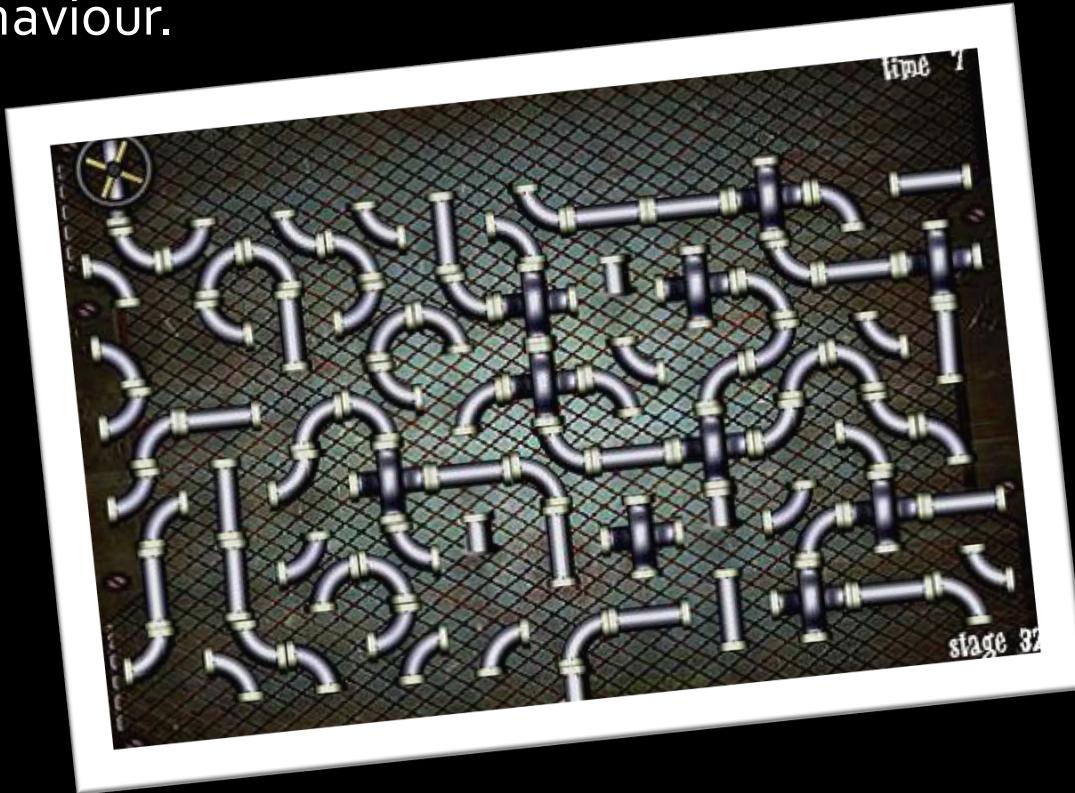


- The control unit sends signals (green lines) to various processor components to enact all possible operations.



Signals → instructions

- A certain combination of signals will make data flow from some source to some destination.
 - Just need to figure out what signals produce what behaviour.



Control unit signals

- **PCWrite**: Write the ALU output to the PC.
- **PCWriteCond**: Write the ALU output to the PC, only if the Zero condition has been met.
- **IorD**: For memory access; short for “Instruction or Data”. Signals whether the memory address is being provided by the PC (for instructions) or an ALU operation (for data).
- **MemRead**: The processor is reading from memory.
- **MemWrite**: The processor is writing to memory.
- **MemToReg**: The register file is receiving data from memory, not from the ALU output.
- **IRWrite**: The instruction register is being filled with a new instruction from memory.

More control unit signals

- **PCSource**: Signals whether the value of the PC resulting from a jump, or an ALU operation.
- **ALUOp** (3 wires): Signals the execution of an ALU operation.
- **ALUSrcA**: Input A into the ALU is coming from the PC (value=0) or the register file (value=1).
- **ALUSrcB** (2 wires): Input B into the ALU is coming from the register file (value=0), a constant value of 4 (value=1), the instruction register (value=2), or the shifted instruction register (value=3).
- **RegWrite**: The processor is writing to the register file.
- **RegDst**: Which part of the instruction is providing the destination address for a register write (`rt` versus `rd`).

Example instruction

- **addi \$t7, \$t0, 42**



- PCWrite = 0
- PCWriteCond = 0
- IorD = X
- MemWrite = 0
- MemRead = 0
- MemToReg = 0
- IRWrite = 0
- PCSource = X
- ALUOp = 001 (add)
- ALUSrcA = 1
- ALUSrcB = 10
- RegWrite = 1
- RegDst = 0

Setting these signals will result in adding register \$t0 by 42 and storing result in register \$t7

- addi \$t7, \$t0, 42

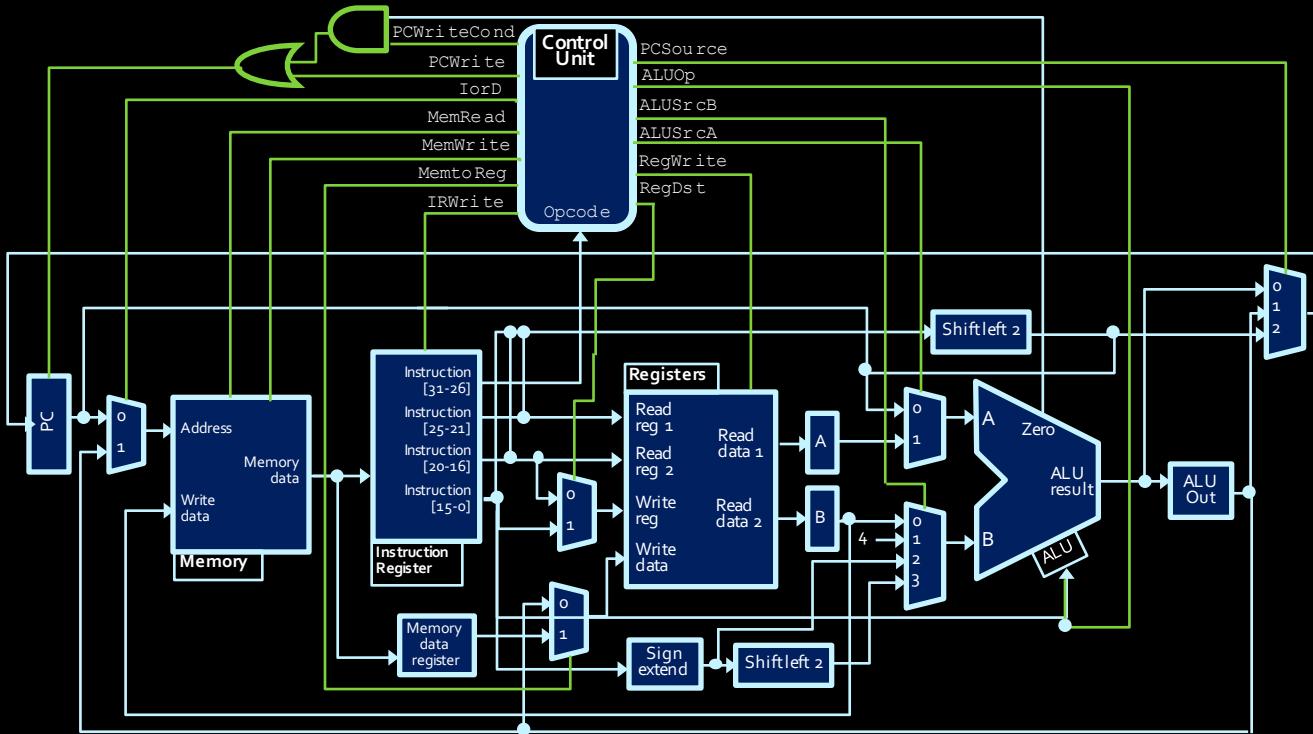


This is a line of
assembly language

Controlling the Datapath



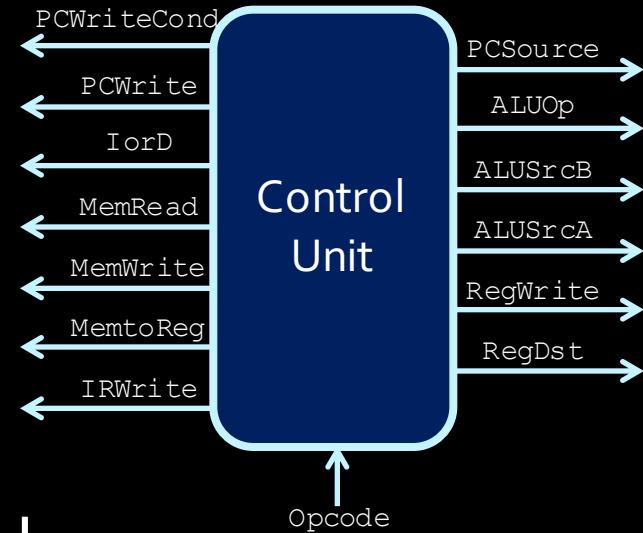
MIPS Datapath



- So, how do we do the following?
 - Increment the PC to the next instruction position.
 - Store $\$t_1 + 12$ into the PC.
 - Assuming that register $\$t_3$ is storing a valid memory address, fetch the data from that location in memory and store it in $\$t_5$.

Controlling the signals

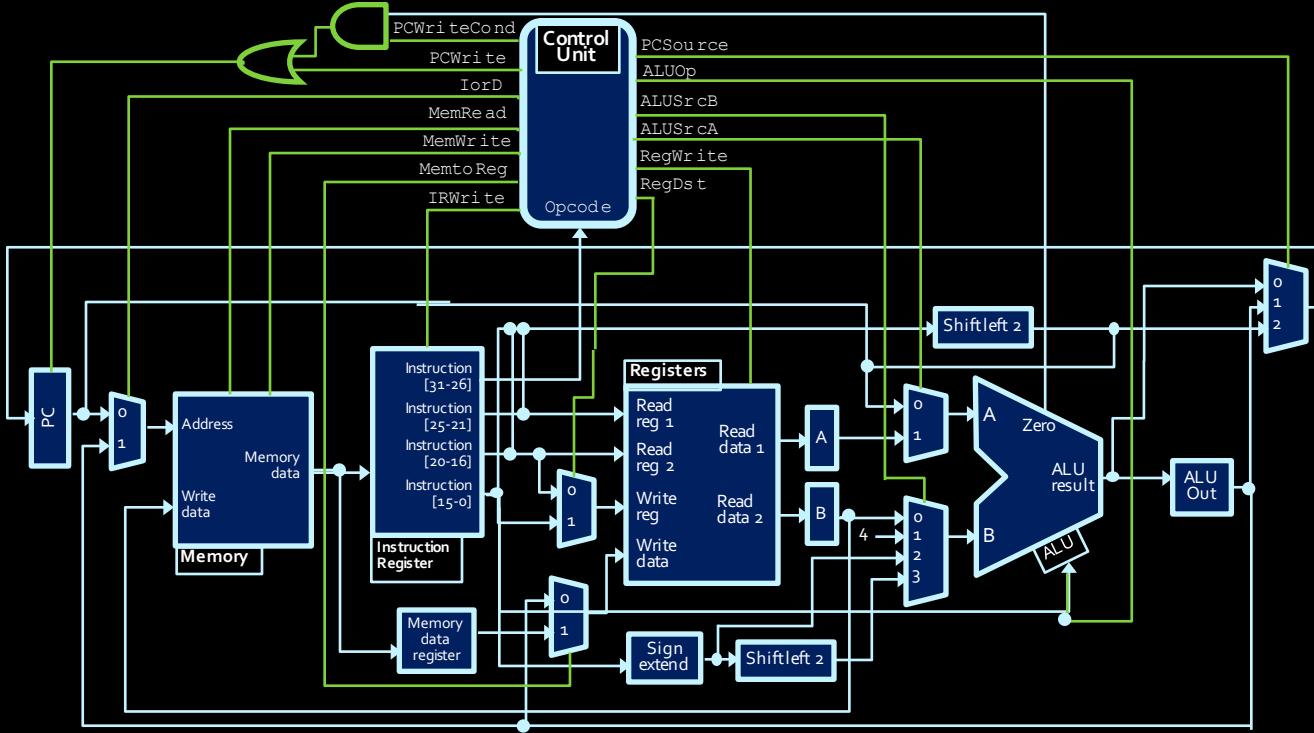
- Need to understand the role of each signal, and what value they need to have in order to perform the given operation.
- So, what's the best approach to make this happen?



Basic approach to datapath

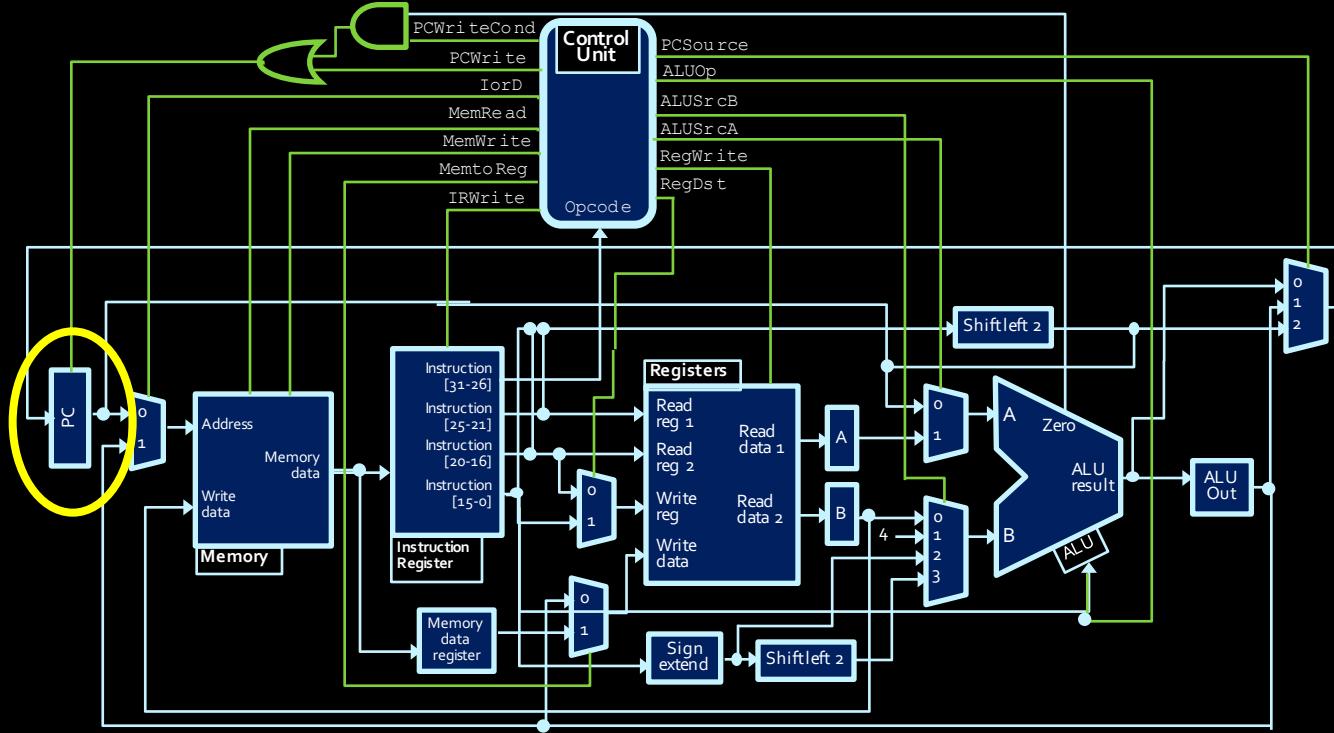
1. Figure out the data source(s) and destination.
2. Determine the path of the data.
3. Deduce the signal values that cause this path:
 - a) Start with Read & Write signals (at most one can be high at a time).
 - b) Then, mux signals along the data path.
 - c) Non-essential signals get an X value.

Example #1: Incrementing PC



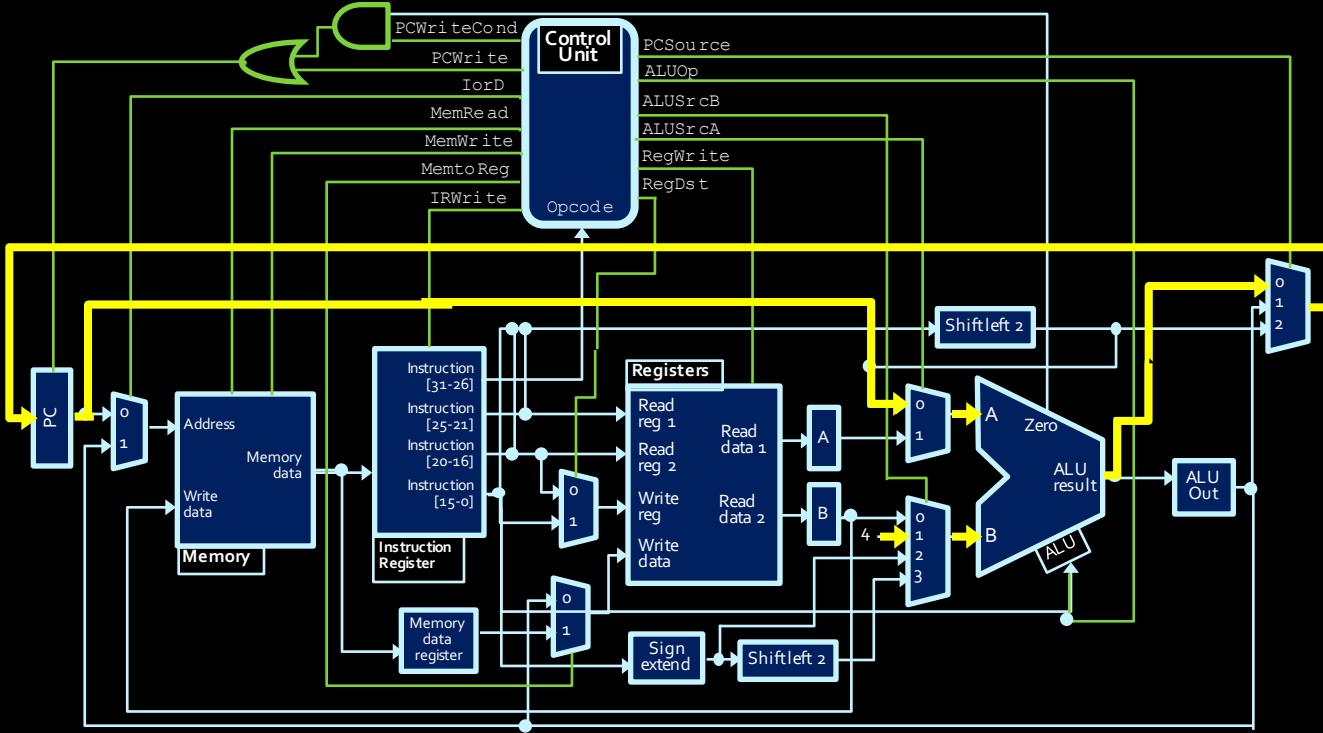
- Given the datapath above, what signals would the control unit turn on and off to increment the program counter by 4?

Example #1: Incrementing PC



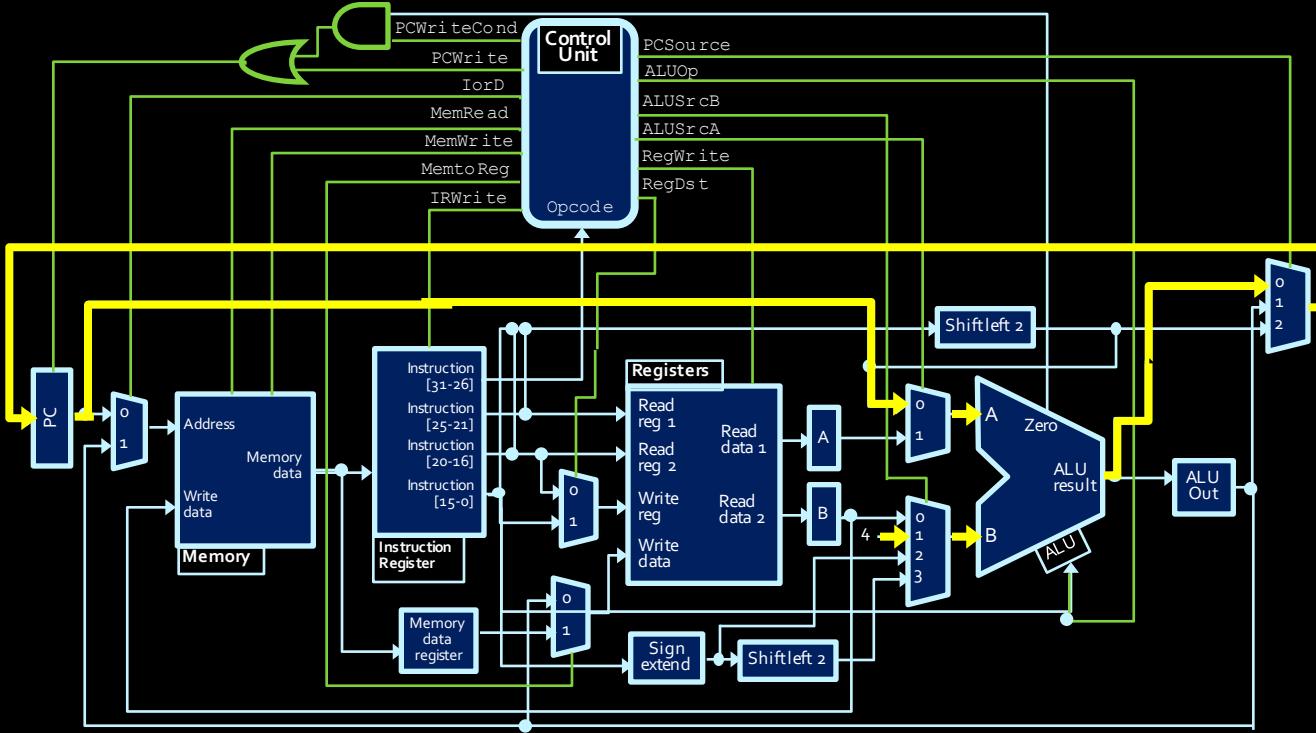
- Step #1: Determine data source and destination.
 - Program counter provides source,
 - Program counter is also destination.

Example #1: Incrementing PC



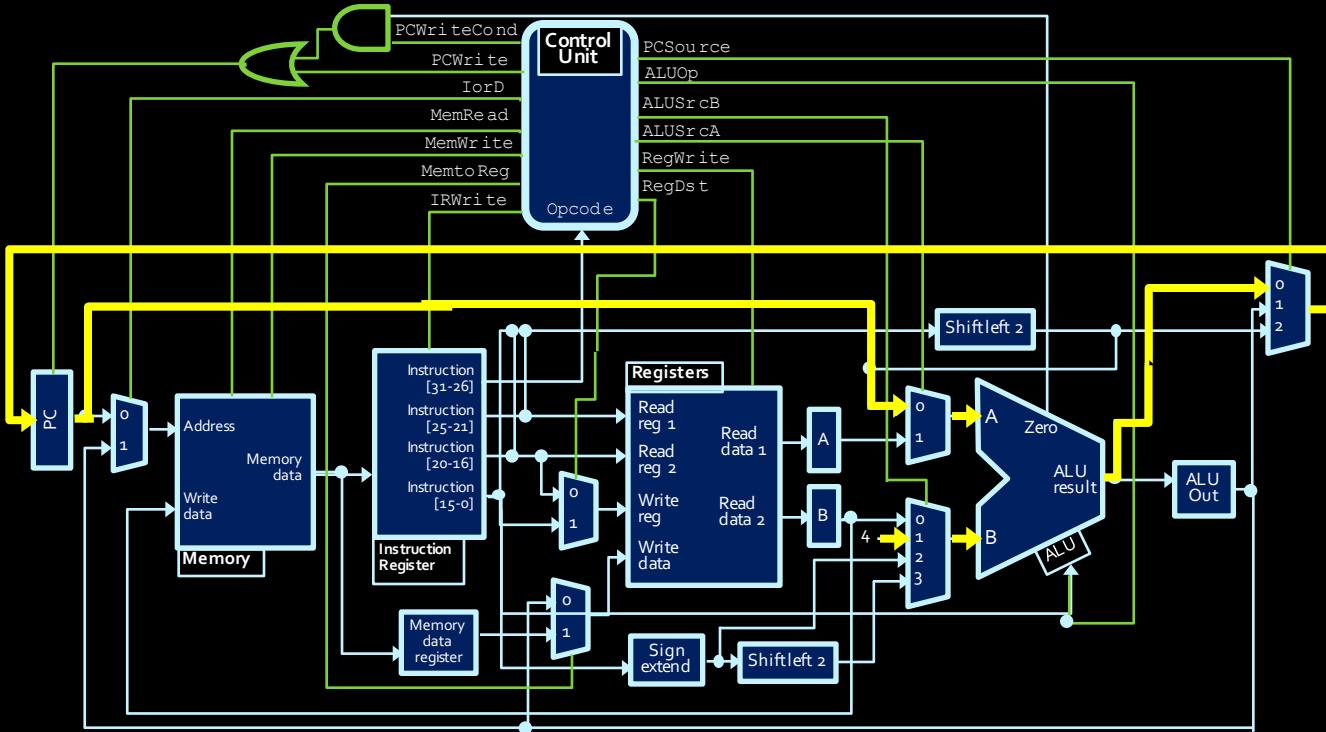
- Step #2: Determine path for data
 - Operand A for ALU: Program counter
 - Operand B for ALU: Literal value 4
 - Destination path: Through mux, back to PC

Example #1: Incrementing PC



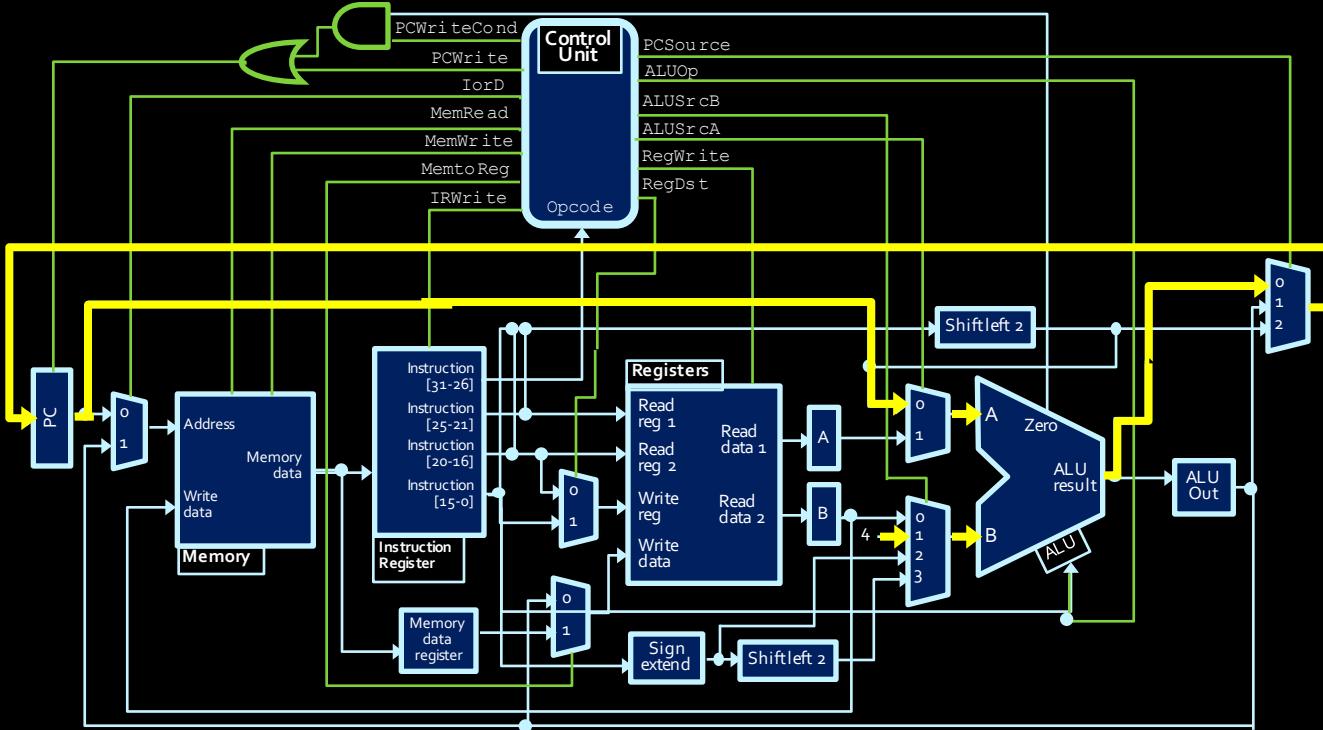
- Setting signals for this datapath:
 1. Read & Write signals:
 - PCWrite is high, all others are low.

Example #1: Incrementing PC



- Setting signals for this datapath:
 2. Mux signals:
 - PCSource is 0, ALUSrcA is 0, ALUSrcB is 1
 - all others are “don’t cares”.

Example #1: Incrementing PC



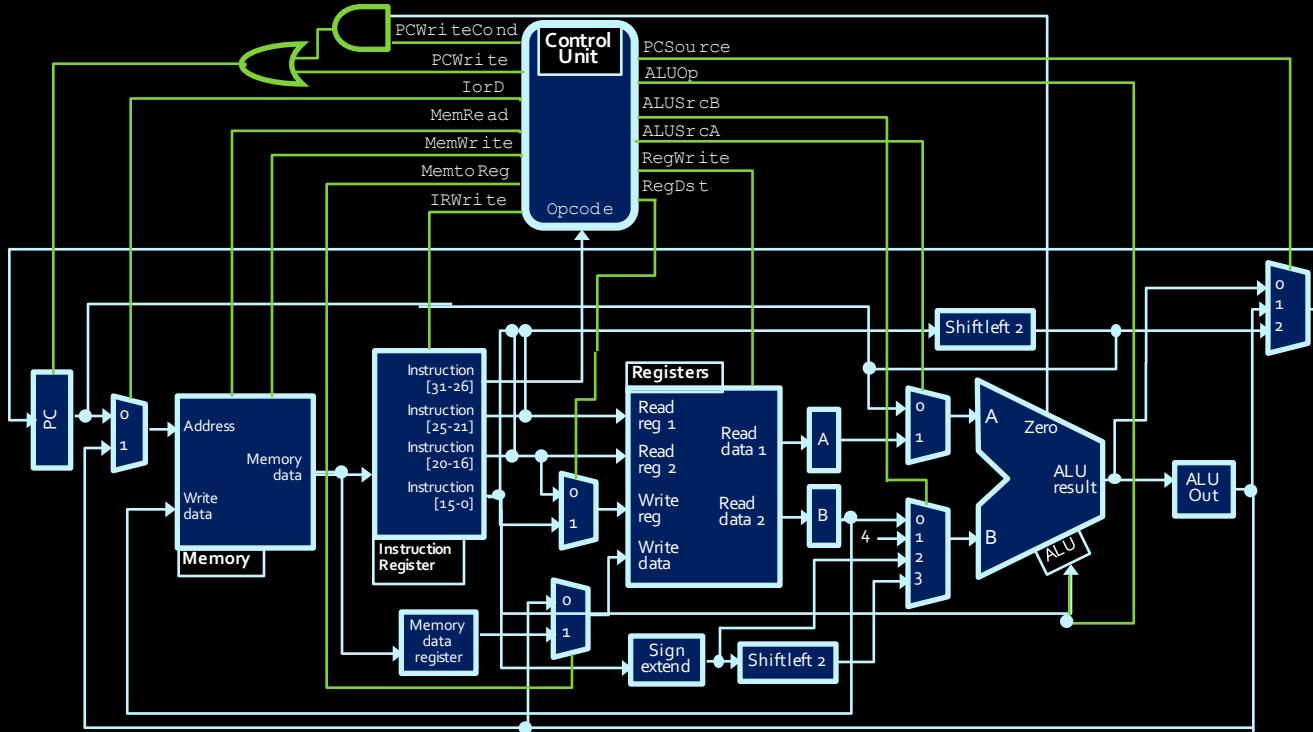
- Other signals for this datapath:
 - ALUOp is 001 ($Cin = 0$, $S1 = 0$, $So = 1$: $A + B$)
 - PCWriteCond is X when PCWrite is 1
 - Otherwise it is 1 except when branching.

Example #1 (final signals)

- PCWrite = 1
- PCWriteCond = X
- IorD = X
- MemRead = 0
- MemWrite = 0
- MemToReg = X
- IRWrite = 0
- PCSource = 0
- ALUOp = 001
- ALUSrcA = 0
- ALUSrcB = 01
- RegWrite = 0
- RegDst = X

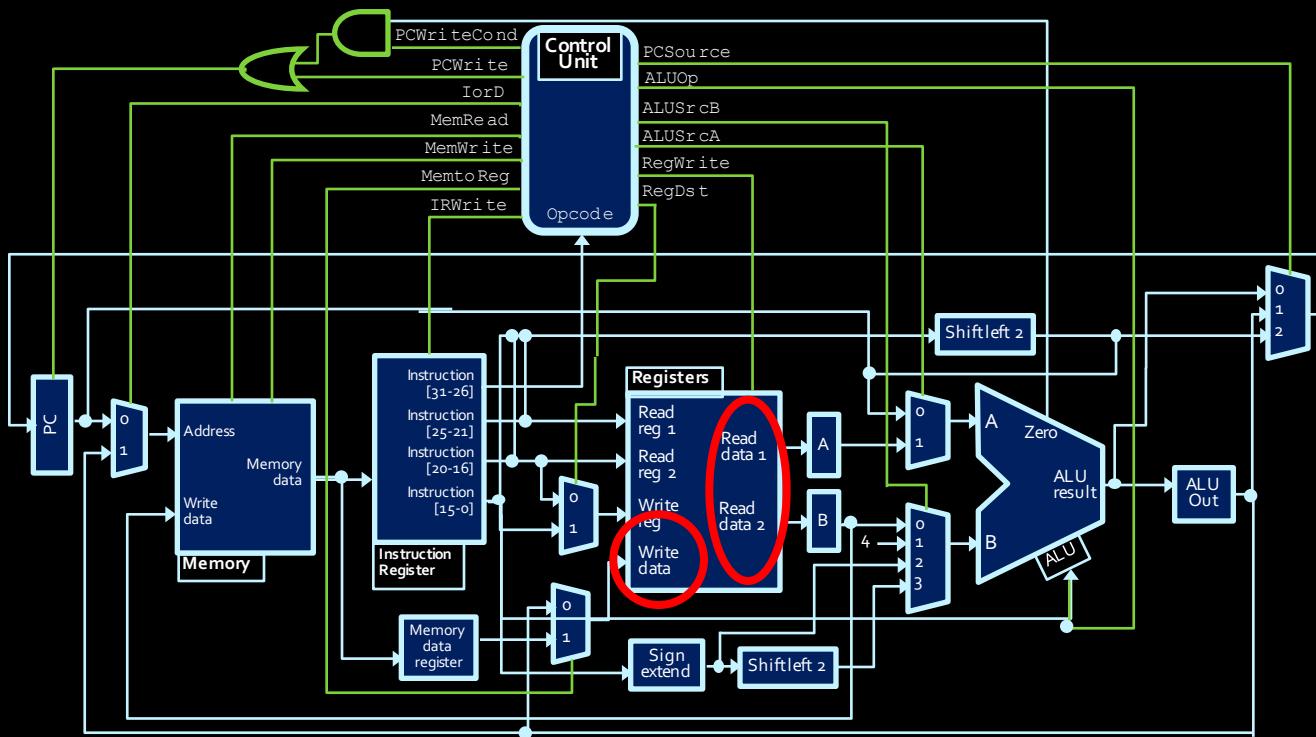
Another example

add \$r7, \$r1, \$r2



- Given the datapath above, what signals would the control unit turn on and off in order to add `$r1` to `$r2` and store the result in `$r7`?

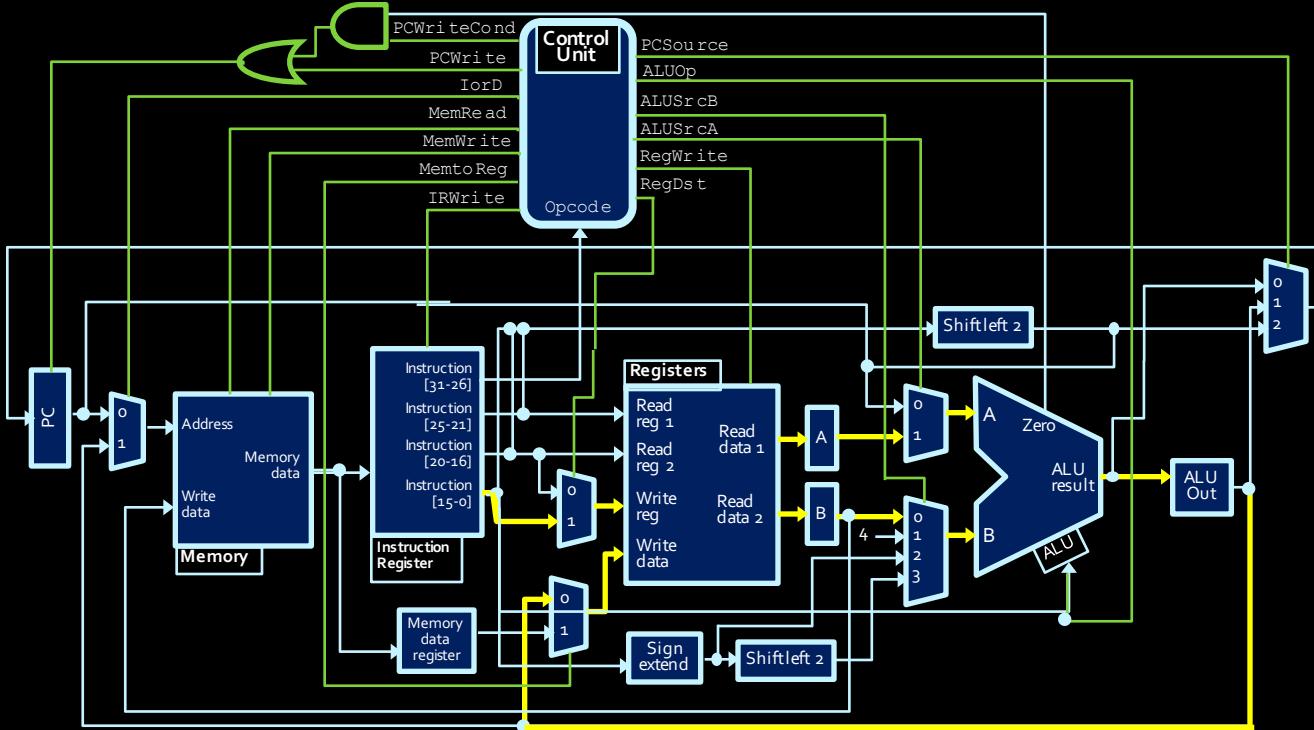
add \$r7, \$r1, \$r2



- Step #1: Data source and destination
 - Data starts in register block.
 - Data goes to register block.

Question #1 (cont'd)

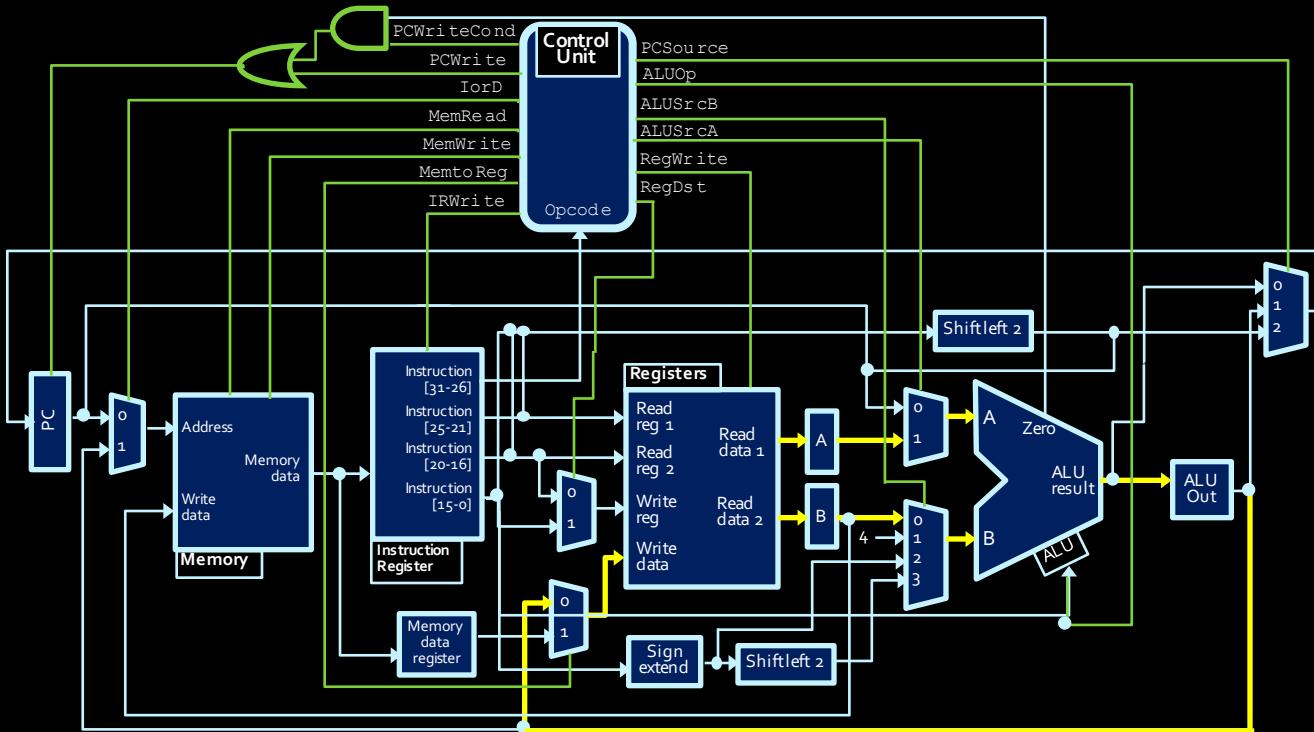
add \$r7, \$r1, \$r2



- Step #2: Determine the path of the data
 - Data needs to go through the ALU before heading back into the register file.

Question #1 (cont'd)

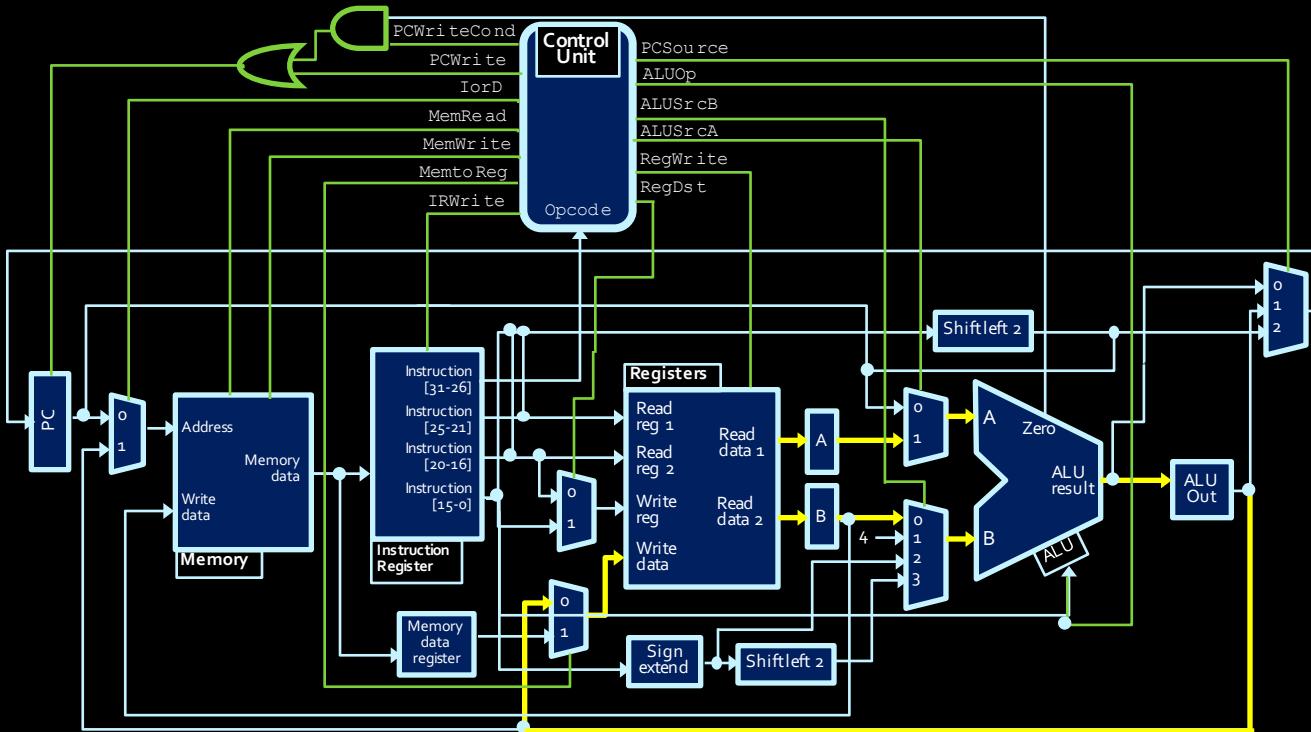
add \$r7, \$r1, \$r2



- Step #3a: Read & Write signals
 - Only `RegWrite` needs to be high.
 - `PCWrite`, `PCWriteCond`, `MemRead`, `MemWrite`, `IRWrite` would be low.

Question #1 (cont'd)

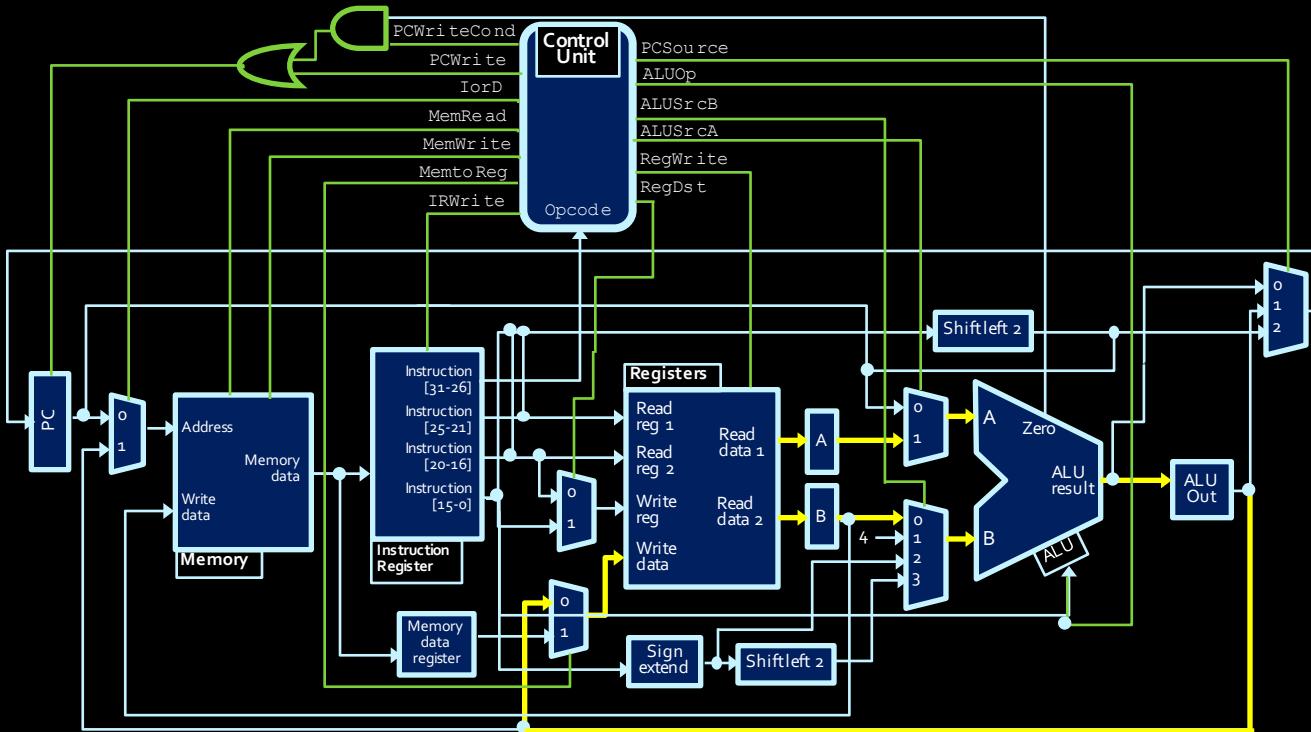
add \$r7, \$r1, \$r2



- Step #3b: Relevant mux signals
 - Muxes before ALU: `ALUSrcA` → 1, `ALUSrcB` → 00.
 - `ALUOp` → 001 (Add)
 - Mux before registers: `MemToReg` → 0

Question #1 (cont'd)

add \$r7, \$r1, \$r2



- Step #3c: Irrelevant mux signals
 - No writing to PC: `PCSource` → X.
 - No reading from memory: `IorD` → X.

Question #1 (cont'd)

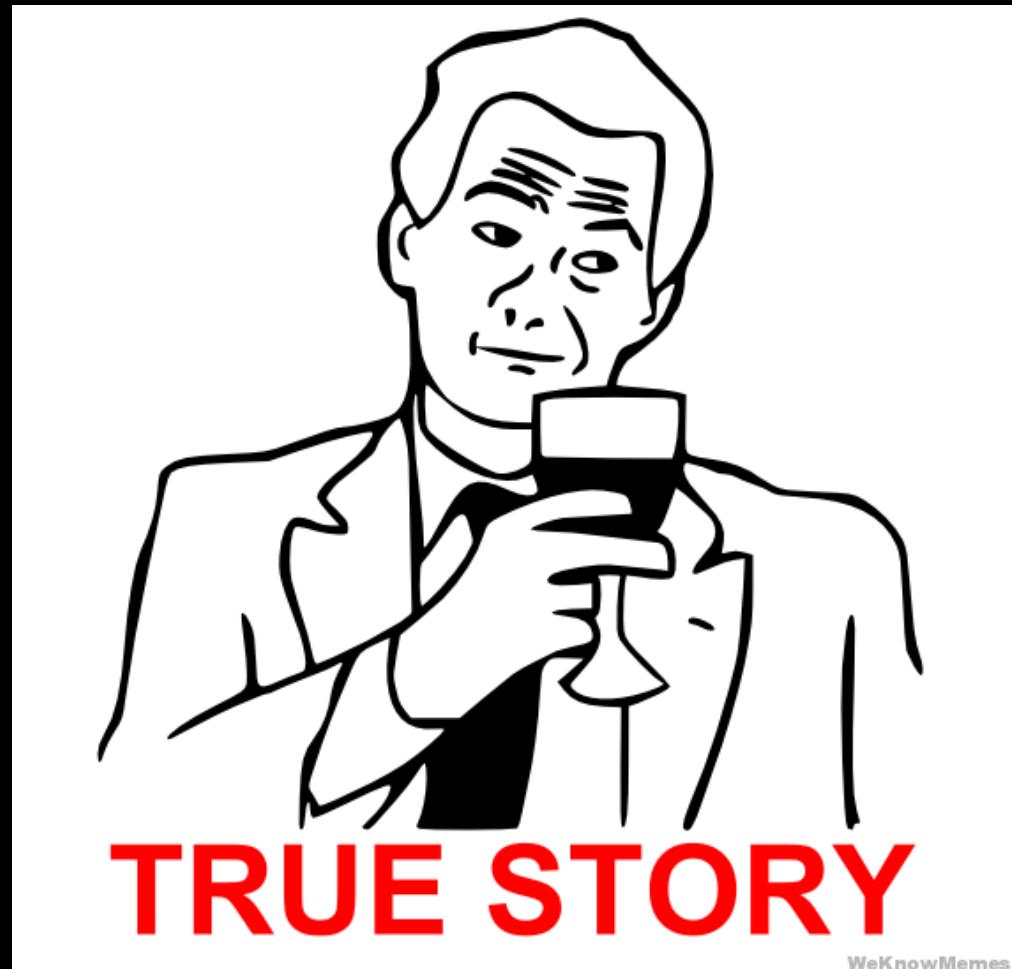
- PCWrite = 0
- PCWriteCond = 0
- IorD = X
- MemRead = 0
- MemWrite = 0
- MemToReg = 0
- IRWrite = 0
- PCSource = X
- ALUOp = 001
- ALUSrcA = 1
- ALUSrcB = 00
- RegWrite = 1
- RegDst = 1

Note: RegDst rule
high for 3-register operations
low for 2-register operations
X if not using register file

The Tale of “Hello world”

1. You, the programmer, write a piece of **code** called `hello.c/java/whatever`
2. You **compile** the code, which translate the code into machine **instructions** and save the in an **executable** file (e.g., `hello`, `hello.exe`)
3. You **run** the executable, OS load the executable (the instructions) into memory, set **PC**.
4. CPU loads the instruction pointed by PC into **instruction register**.
5. Control Unit checks the **opcode** (first 6 bits), decode the rest of the instruction and send **signals** to setup the **datapath** (billions of MOSFETs switching ON/OFF).
6. Data flow through the datapath, electrons move around...
7. And BOOM!, “**Hello world**” shows up on your screen

The Tale of “Hello world”



CSC258 Winter 2016

Lecture 9

Announcements

- If you'd like to borrow a DE-2 board for your own project, talk to Andrew Wang.
- For assembly labs you can work individually or in pairs. How matter how you do it, the important thing is that each individual of you learns from the lab.
- Midterm remarking requests have been processed, you may check you updated marks on MarkUs and pick up your midterms.



Today's quiz has
4 marks in total.

Tear off the
reference sheet,
and keep it.

Question 1

Below is the content of an executable file “mystery.exe”, what does this program do?

```
1000 1110 0000 1000 0101 1010 1111 0001  
1000 1110 0010 1001 1101 0010 0011 0010  
0000 0001 0000 1001 0101 0000 0010 0000  
1000 1110 0100 1011 1111 0011 0011 0111  
0000 0000 0000 1100 0011 0001 0000 0000  
0000 0010 0110 1010 1010 0000 0010 0010  
1010 1101 1101 0100 0000 1111 0101 1010
```

OK, this one is too long for a quiz,
but it is a legit question,
which you should be able to answer.
Do it at home for practice.





Today's quiz has
4 marks in total.

Tear off the
reference sheet,
and keep it.

Question 1

Below is a R-type instruction

0000 0000 0110 0101 0100 0000 0010 0111

Which type operation does it do? _____

Which register is the result stored in? _____

Question 2

Below is an I-type instruction (BNE, branch on not equal)

0001 0100 1010 1001 1111 1111 1110 1111

When the NE condition is satisfied, what is the change of the PC value?

$$\text{PC} = \text{PC} + \underline{\hspace{2cm}}$$

Question 3

In a **Jump (J)** instruction, PC changes by an offset, i.e.,

$$PC = PC + \text{offset}$$

How big is the range of possible offset values (the difference between the most positive possible offset and the most negative possible offset)?

Write your answer in terms of a power of 2.

2^(???)

Solutions



Question 1

Below is a R-type instruction

0000 0000 0110 0101 0100 0000 00**10** 0111

Which type operation does it do? _____ **NOR** _____

Which register is the result stored in? _____ **8** _____

Question 2



Below is an I-type instruction (BNE, branch on not equal)

0001 0100 1010 1001 1111 1111 1110 1111

When the NE condition is satisfied, what is the change of the PC value?

Two's complement: 0000 0000 0001 0001 = 17

so offset is: $-17 \ll 2$, i.e., -17×4

$$PC = PC + \underline{\hspace{2cm}} -68 \underline{\hspace{2cm}}$$

Question 3



destination address = {PC[31:28], whats-above, 00}

In a **Jump (J)** instruction, PC changes by an offset, i.e.,

$$PC = PC + \text{offset}$$

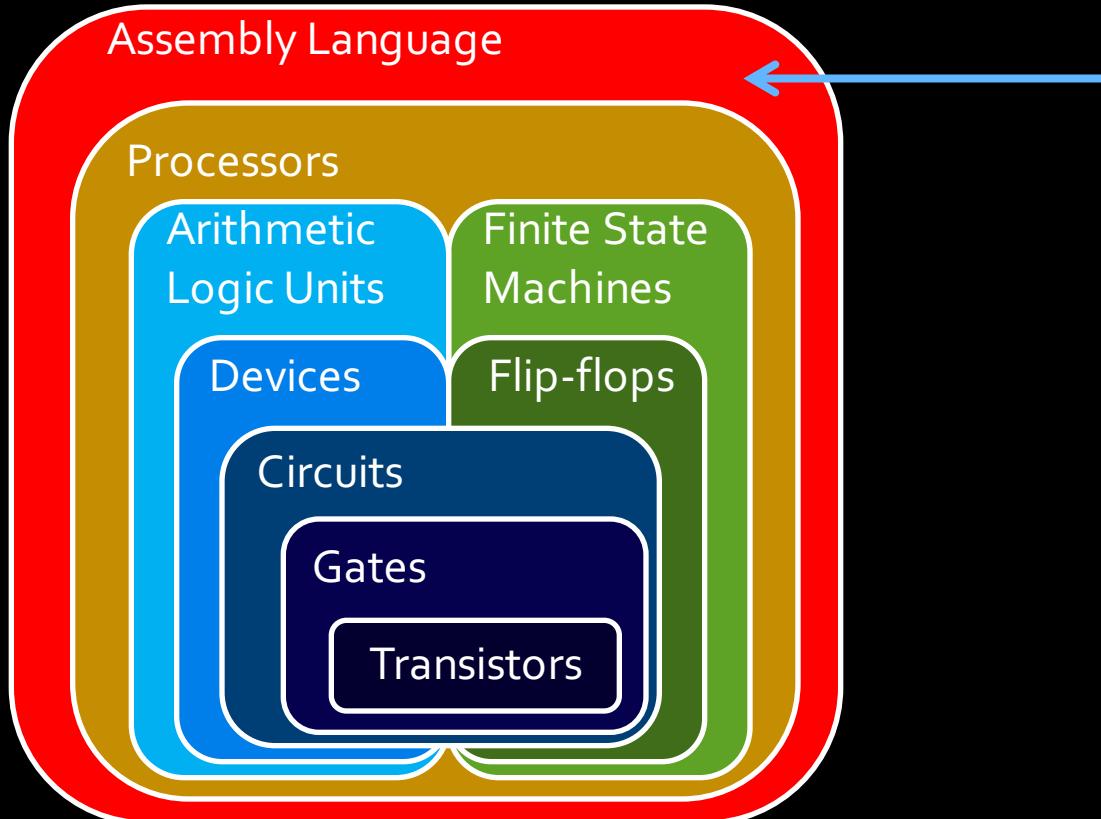
How big is the range of possible offset values (the difference between the most positive offset and the most negative offset)?

Write your answer in terms of a power of 2.

$$2^{28} = 256\text{MB}$$

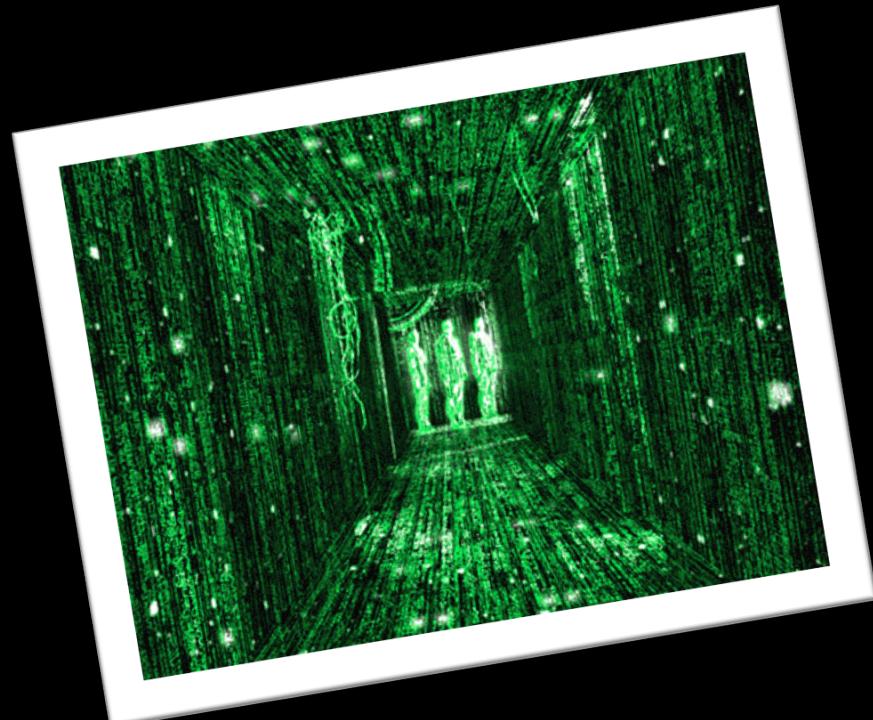
Because the highest 4 bits of the new PC must be the same as the current PC.

We are here



Programming the processor

- Things to learn:
 - Control unit signals to the datapath
 - Machine code instructions
 - Assembly language instructions
 - Programming in assembly language



Machine Code Instructions

00000000	01 00 FF FF 00 00 00 00	00 00 00 00 00 40 00 CC 80@.
00000010	0C 00 00 00 00 26 01	8F 00 00 00 00 00 53 00&....S.
00000020	65 00 6C 00 65 00 63 00	74 00 20 00 52 00 75 00	e.l.e.c.t. R.u.
00000030	6C 00 65 00 00 00 08 00	00 00 00 01 4D 00 53 00	l.e.....M.S.
00000040	20 00 53 00 68 00 65 00	6C 00 6C 00 20 00 44 00	S.h.e.l.l. D.
00000050	6C 00 67 00 00 00 00 00	00 00 00 00 00 02 00 00	l.g.....
00000060	03 01 A1 50 53 00 3A 00	C3 00 36 00 32 25 00 00	...PS.....6.2%
00000070	FF FF 83 00 00 00 00 00	00 00 00 00 00 00 00 00P.V.A. J&.
00000080	03 00 01 50 0E 00 56 00	41 00 0A 00 4A 26 00 00&A.p.p.l.y.
00000090	FF FF 80 00 26 00 41 00	70 00 70 00 6C 00 79 00	t.o. a.l.l.
000000a0	20 00 74 00 6F 00 20 00	61 00 6C 00 6C 00 00 00P
000000b0	00 00 00 00 00 00 00 00	00 00 00 00 01 00 01 50	~}.2.....
000000c0	7E 00 7D 00 32 00 0E 00	01 00 00 00 FF FF 80 00	O.K.....
000000d0	4F 00 4B 00 00 00 00 00	00 00 00 00 00 00 00 00	...P.}.2.....
000000e0	00 00 01 50 B4 00 7D 00	32 00 0E 00 02 00 00 00	...C.a.n.c.e.l.
000000f0	FF FF 80 00 43 00 61 00	6E 00 63 00 65 00 6C 00P
00000100	00 00 00 00 00 00 00 00	00 00 00 00 00 01 50P
00000110	EA 00 7D 00 32 00 0E 00	09 00 00 00 FF FF 80 00	}.2.....
00000120	26 00 48 00 65 00 6C 00	70 00 00 00 00 00 00 00	&H.e.l.p.....
00000130	00 00 00 00 00 00 00 00	80 08 81 50 0E 00 3A 00P..
00000140	3B 00 0E 00 2F 25 00 00	FF FF 81 00 00 00 00 00%.....
00000150	00 00 00 00 00 00 00 00	00 00 02 50 0E 00 30 00P.0
00000160	1E 00 08 00 EE 25 00 00	FF FF 82 00 46 00 69 00%.F.i.
00000170	6C 00 65 00 20 00 54 00	79 00 70 00 65 00 00 00	l.e. T.y.p.e..
00000180	00 00 00 00 00 00 00 00	00 00 00 00 00 00 02 50P
00000190	54 00 30 00 2C 00 08 00	EF 25 00 00 FF FF 82 00	T.O.%.....
000001a0	50 00 61 00 72 00 73 00	69 00 6E 00 67 00 20 00	F.a.r.s.i.n.g..
000001b0	52 00 75 00 6C 00 65 00	73 00 00 00 00 00 00 00	R.u.l.e.s.....
000001c0	00 00 00 00 00 00 00 00	07 00 00 50 06 00 07 00P...
000001d0	1A 01 71 00 ED 25 00 00	FF FF 80 00 00 00 00 00	.q. %.....
000001e0	00 00 00 00 00 00 00 00	00 00 02 50 0E 00 11 00P...
000001f0	3E 00 08 00 EC 25 00 00	FF FF 82 00 53 00 65 00	>....%.S.e.
00000200	6C 00 65 00 63 00 74 00	20 00 52 00 75 00 6C 00	l.e.c.t. R.u.l.
00000210	65 00 20 00 46 00 6F 00	72 00 20 00 46 00 69 00	e. F.o.r. F.i.
00000220	6C 00 65 00 00 00 00 00	00 00 00 00 00 00 00 00	l.e.....
00000230	80 08 81 50 0E 00 1B 00	08 01 0E 00 EB 25 00 00	...P.....%
00000240	FF FF 81 00 00 00 00 00	00 00 00 00 00 00 00 00	...P.a.7...k&..
00000250	00 00 02 50 19 00 61 00	37 00 08 00 6B 26 00 00
00000260	FF FF 82 00 00 00 00 00	

00000520	EE EE 85 00 00 00 00 00	I
00000540	00 00 05 20 1d 00 ET 00	31 00 08 00 0B 5e 00 00	Б.в.1.М?
00000540	EE EE 81 00 00 00 00 00	00 00 00 00 00 00 00 00
00000530	80 08 81 20 0E 00 1B 00	08 07 0E 00 EB 52 00 00	Б.....X
00000550	EC 00 P2 00 00 00 00 00	00 00 00 00 00 00 00 00
00000570	E2 00 50 00 4E 00 FE 00	15 00 50 00 4E 00 FE 00	Б.в.о.г.Б.т
00000590	EC 00 P2 00 E3 00 14 00	50 00 25 00 12 00 PC 00	Г.е.с.ф.Б.п.1
000007E0	3E 00 08 00 EC 52 00 00	EE EE 85 00 23 00 E2 00	Г.х.з.2.е
000007E0	00 00 00 00 00 00 00 00	00 00 03 20 0E 00 II 00

Intro to Machine Code

- Machine code are the 32-bit binary instructions which the processor can understand (you can now understand, too)
- All programs (C, Java, Python) are eventually translated into machine code (by a compiler or interpreter).
- While executing, the instructions of the program are loaded into the **instruction register** one by one
- For each instruction loaded, the **Control Unit** reads the **opcode** and sets the signals to control the datapath, so that the processor works as instructed.

Assembly language

- Each line of assembly code corresponds to one line of 32-bit long machine code.
- Basically, assembly is a user-friendly way to write machine code.
- Example: C = A + B
 - Store A in \$t1, B in \$t2, C in \$t3
 - Assembly language instruction:
`add $t3, $t1, $t2`
 - Machine code instruction:
`000000 01001 01010 01011 XXXXX 100000`

Note: There is a 1-to-1 mapping for all assembly code and machine code instructions!

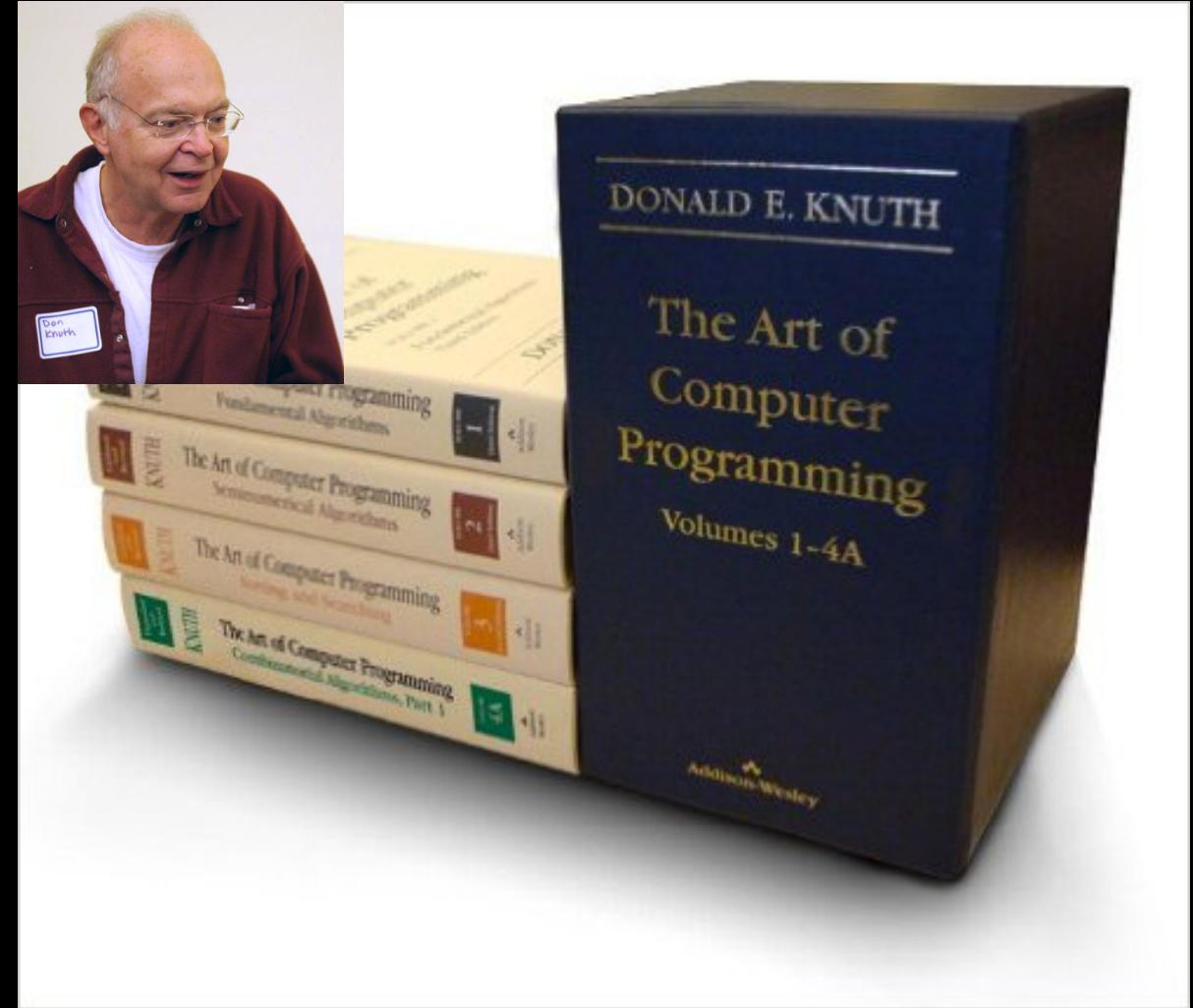
Why learn assembly?

- You'll understand how your program **really** works.
- You'll understand your program's performance better by knowing its real “runtime”.
- You'll understand how control flows (if / else / for / while) are implemented.
- You'll understand why eliminating if statements makes your code faster.
- You'll understand why pointer is such a natural concept for programming.
- You'll understand the cost of making function calls.
- You'll understand why stack can overflow
- You'll understand there is no “recursion” in the hardware, and how it's actually done.
- You'll understand why memory need to be managed.
- You'll understand why people spend so much time creating operating systems.
- You'll appreciate more the constructs in high-level programming languages.
- And much more...

And, you'll be able to read
this book.

Donald Knuth "The Art of
Computer Programming"

"All algorithms in this book
are written in assembly for
clarity."



About register names

- In machine code we have register 0 to register 31, specified by 5 bits of the instruction.
- In assembly we have names like \$t1, \$t2, \$s1, \$v0, etc.
- What's the relation between these two?

Machine code + registers

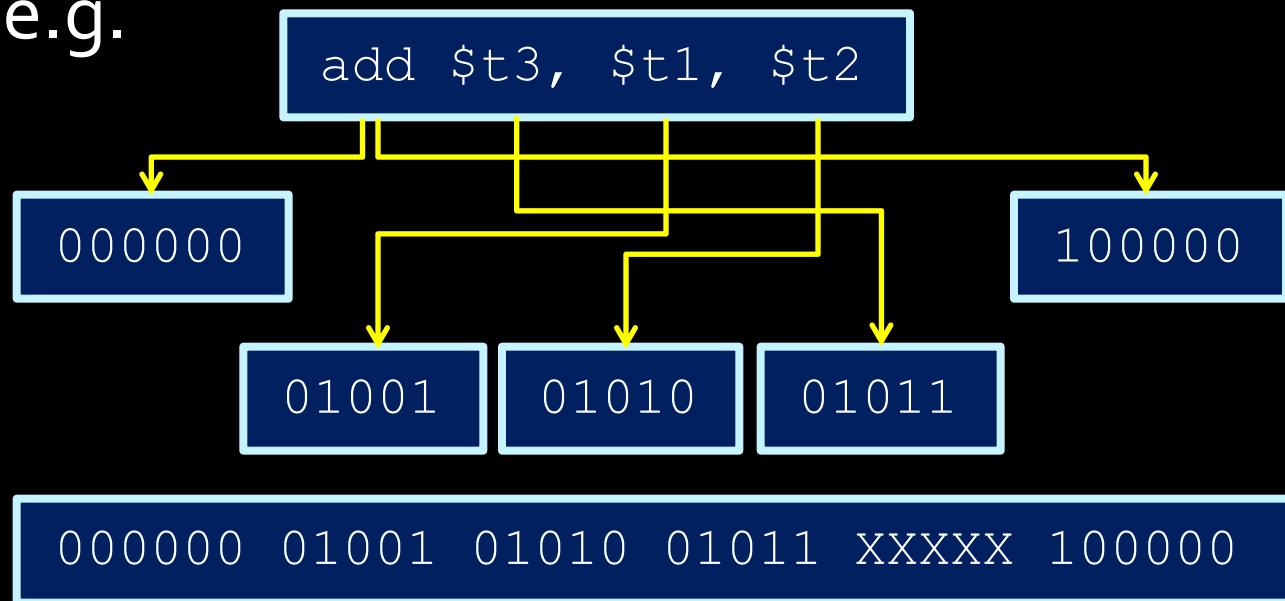
- MIPS is register-to-register.
 - Every operation operates on data in registers.
- MIPS provides 32 registers.
 - Several have special values:
 - Register 0 (\$zero): value 0 -- always.
 - Register 1 (\$at): reserved for the assembler.
 - Registers 2–3 (\$v0, \$v1): return values
 - Registers 4–7 (\$a0-\$a3): function arguments
 - Registers 8–15, 24–25 (\$t0-\$t9): temporaries
 - Registers 16–23 (\$s0-\$s7): saved temporaries
 - Registers 28–31 (\$gp, \$sp, \$fp, \$ra): memory and function support
 - Registers 26–27: reserved for OS kernel
 - Also three special registers (PC, HI, LO) that are not directly accessible.
 - HI and LO are used in multiplication and division, and have special instructions for accessing them.

\$v0, \$t2, \$a3,
etc are the
registers'
nicknames in
assembly

Technically
you can use
any register
for anything,
but this is the
convention

Translate assembly to machine code

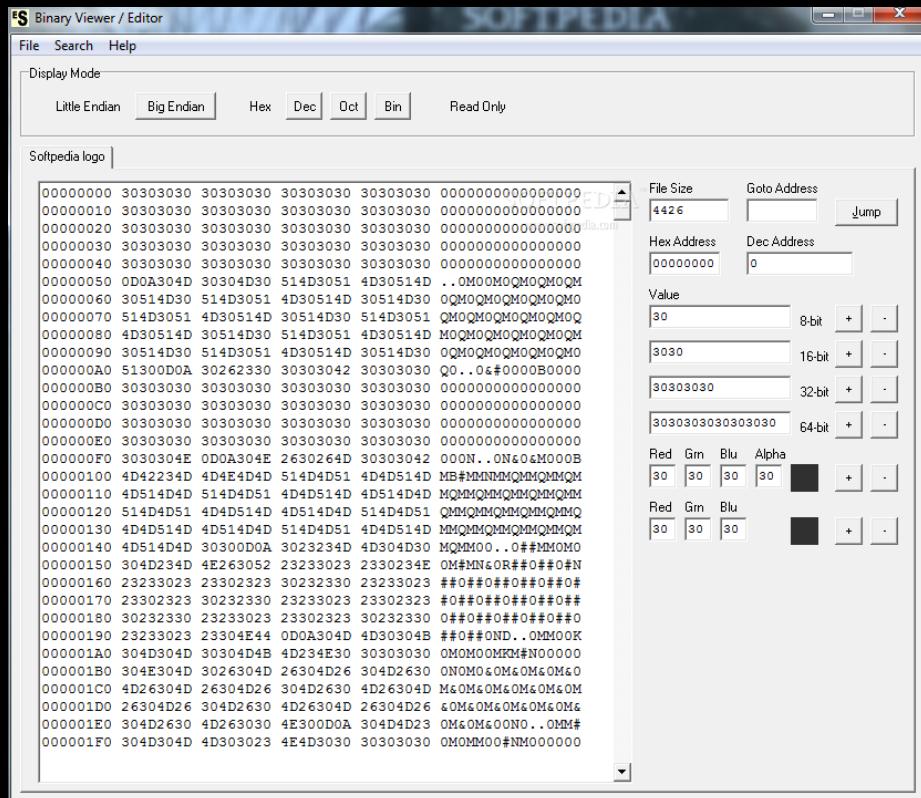
- When writing machine code instructions (or interpreting them), we need to know which register values to encode (or decode).
- e.g.



Machine code details

- Things to note about machine code:
 - R-type instructions have an opcode of 000000, with a 6-bit function listed at the end.
 - Although we specify “**don’t care**” bits as X values, the assembly language interpreter always assigns them to some value (like 0)
 - In exams, we want you to write X instead of 0, to show that you know we don’t care those bits

Now you can totally program an executable like this
(don't even need a compiler).



Assembly Language Instructions

```
loop: lw    $t3, 0($t0)
      lw    $t4, 4($t0)
      add  $t2, $t3, $t4
      sw    $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```

Assembler

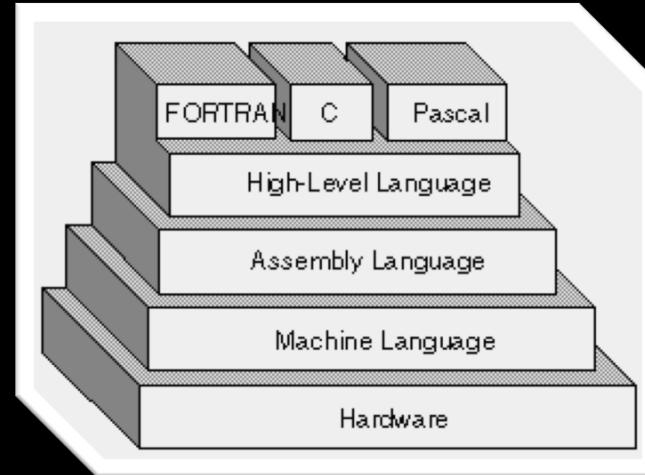
```
0x8d0b0000
0x8d0c0004
0x016c5020
0xad0a0008
0x21080004
0x2129ffff
0x1d20ffff9
```

pðfs 2fJ' Joob
SECRET ACTIVATED

0xJPS0EE6
0XSECRET

Assembly language

- Assembly language is the lowest-level language that you'll ever program in.
- Many compilers translate their high-level program commands into assembly commands, which are then converted into machine code and used by the processor.
- Note: There are multiple types of assembly language, especially for different architectures!



Trivia

The thing that converts assembly code to executable is **NOT** called a **compiler**.

It's called an **assembler**, because there is no fancy complication needed, it just **assembles** the lines!



A little about MIPS

- MIPS
 - Short for Microprocessor without Interlocked Pipeline Stages
 - A type of RISC (Reduced Instruction Set Computer) architecture.
 - Provides a set of simple and fast instructions
 - Compiler translates instructions into 32-bit instructions for instruction memory.
 - Complex instructions are built out of simple ones by the compiler and assembler.

The layout of assembly code

Code sectioning syntax: example

```
.data
A:    .space    400      # array of 100 integers
B:    .space    400      # array of 100 integers

.text
main:   add $t0, $zero, $zero      # load "0" into $t0
        addi $t1, $zero, 400      # load "400" into $t1
        addi $t9, $zero, B        # store address of B
        addi $t8, $zero, A        # store address of A

loop:    add $t4, $t8, $t0      # $t4 = addr(A) + i
        add $t3, $t9, $t0      # $t3 = addr(B) + i
        lw $s4, 0($t3)        # $s4 = B[i]
        addi $t6, $s4, 1        # $t6 = B[i] + 1
        sw $t6, 0($t4)        # A[i] = $t6
        addi $t0, $t0, 4        # $t0 = $t0++
        bne $t0, $t1, loop     # branch back if $t0<400

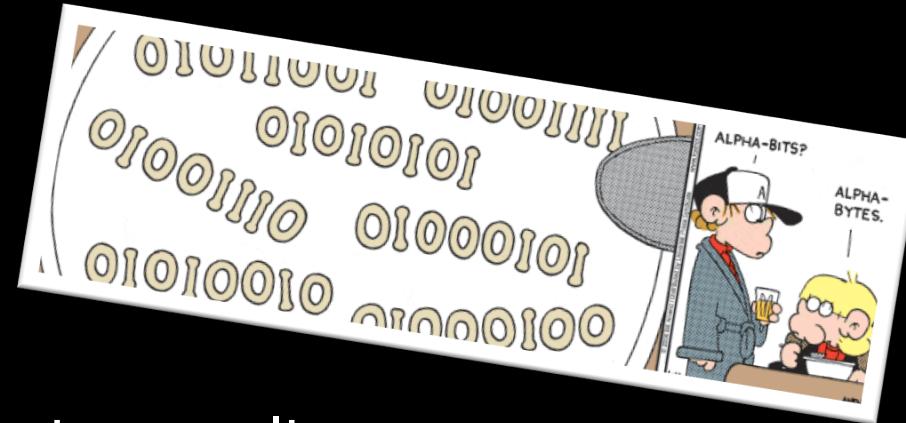
end:
```

Code sectioning syntax

- `.data`
 - Indicates the start of the data declarations.
- `.text`
 - Indicates the start of the program instructions.
- `main:`
 - The initial line to run when executing the program.
- You can create other labels as needed.

MIPS Instructions

- Things to note about MIPS instructions:
 - Instruction are written as: <instr> <parameters>
 - Each instruction is written on its own line
 - All instructions are 32 bits (4 bytes) long
 - Instruction addresses are measured in **bytes**, starting from the instruction at address 0.
- The following tables show the most common MIPS instructions, the syntax for their parameters, and what operation they perform.



Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
add	100000	\$d, \$s, \$t	\$d = \$s + \$t
addu	100001	\$d, \$s, \$t	\$d = \$s + \$t
addi	001000	\$t, \$s, i	\$t = \$s + SE(i)
addiu	001001	\$t, \$s, i	\$t = \$s + SE(i)
div	011010	\$s, \$t	lo = \$s / \$t; hi = \$s % \$t
divu	011011	\$s, \$t	lo = \$s / \$t; hi = \$s % \$t
mult	011000	\$s, \$t	hi:lo = \$s * \$t
multu	011001	\$s, \$t	hi:lo = \$s * \$t
sub	100010	\$d, \$s, \$t	\$d = \$s - \$t
subu	100011	\$d, \$s, \$t	\$d = \$s - \$t

Note: “hi” and “lo” refer to the high and low bits referred to in the register slide.
“SE” = “sign extend”.

ALU instructions

- Note that for ALU instruction, most are R-type instructions.
 - The six-digit codes in the tables are therefore the function codes (pcodes are 000000).
 - Exceptions are the I-type instructions (addi, andi, ori, etc.)
- Not all R-type instructions have an I-type equivalent.
 - RISC architectures dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations.
 - Example: `divi $t0, 42` can be done by
 - `addi $t1, $zero, 42`
 - `div $t0 t1`

Logical instructions

Instruction	Opcode/Function	Syntax	Operation
and	100100	\$d, \$s, \$t	\$d = \$s & \$t
andi	001100	\$t, \$s, i	\$t = \$s & ZE(i)
nor	100111	\$d, \$s, \$t	\$d = ~(\$s \$t)
or	100101	\$d, \$s, \$t	\$d = \$s \$t
ori	001101	\$t, \$s, i	\$t = \$s ZE(i)
xor	100110	\$d, \$s, \$t	\$d = \$s ^ \$t
xori	001110	\$t, \$s, i	\$t = \$s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

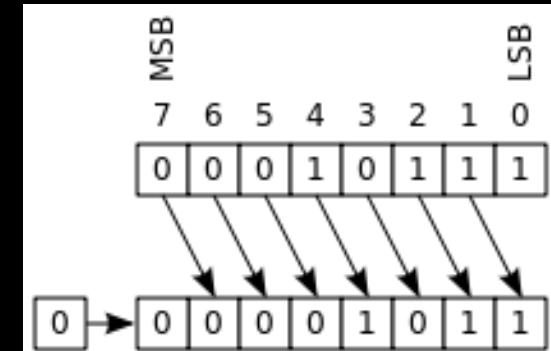
Shift instructions

Instruction	Opcode/Function	Syntax	Operation
sll	000000	\$d, \$t, a	$\$d = \$t \ll a$
sllv	000100	\$d, \$t, \$s	$\$d = \$t \ll \$s$
sra	000011	\$d, \$t, a	$\$d = \$t \gg a$
sraw	000111	\$d, \$t, \$s	$\$d = \$t \gg \$s$
srl	000010	\$d, \$t, a	$\$d = \$t \ggg a$
srlv	000110	\$d, \$t, \$s	$\$d = \$t \ggg \$s$

Note: srl = “shift right logical”, and sra = “shift right arithmetic”.
The “v” denotes a variable number of bits, specified by \$s.

Logic shift vs Arithmetic shift

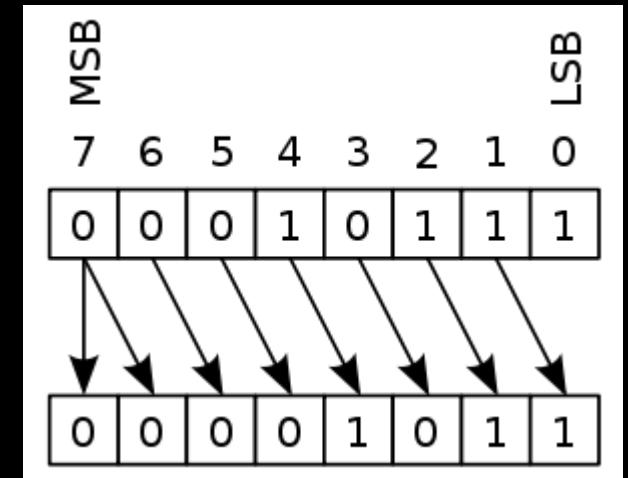
Left shift: same, fill empty spot with zero
(that's why we have **sll** but no **sla**)



Logic

Right shift: different

- Logic shift fills empty spot with zero
- Arithmetic shift fills empty spot with the MSB of the original number.



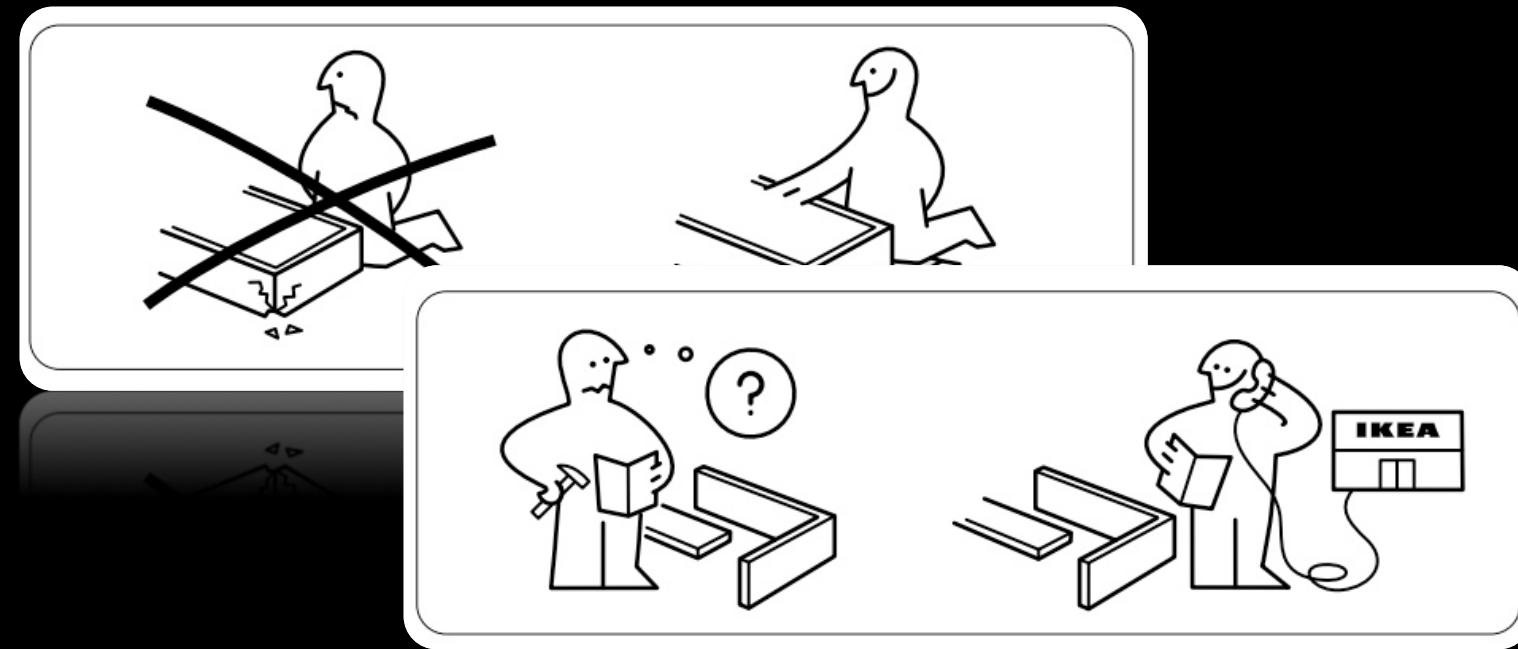
Arithmetic

Data movement instructions

Instruction	Opcode/Function	Syntax	Operation
mfhi	010000	\$d	\$d = hi
mflo	010010	\$d	\$d = lo
mthi	010001	\$s	hi = \$s
mtlo	010011	\$s	lo = \$s

- These are instructions for operating on the HI and LO registers described earlier.

Time for more instructions!



Flow control: Branch and loop

Control flow in assembly

- Not all programs follow a linear set of instructions.
 - Some operations require the code to branch to one section of code or another (if/else).
 - Some require the code to jump back and repeat a section of code again (for/while).
- For this, we have **labels** on the left-hand side that indicate the points that the program flow might need to jump to.
 - References to these points in the assembly code are resolved at compile time to offset values for the program counter.

Branch instructions

Instruction	Opcode/Function	Syntax	Operation
beq	000100	\$s, \$t, label	if (\$s == \$t) pc += i << 2
bgtz	000111	\$s, label	if (\$s > 0) pc += i << 2
blez	000110	\$s, label	if (\$s <= 0) pc += i << 2
bne	000101	\$s, \$t, label	if (\$s != \$t) pc += i << 2

- Branch operations are key when implementing if statements and while loops.
- The labels are memory locations, assigned to each label at compile time.
 - Note: i is calculated as (label - (current PC + 4)) >> 2

Note: Real vs Pseudo instructions

What we list in the slides are all **real instructions**, i.e., each one has an **opcode** corresponding to it.

There are some **pseudo-instructions**, which don't have their own opcode, but is implemented using real instructions; they are provided for coding convenience.

For example:

- `bge $t0,$t1,Label` is actually
- `slt $t2,$t0,$t1; beq $t2,$zero,Label`

Jump instructions

Instruction	Opcode/Function	Syntax	Operation
j	000010	label	$pc += i \ll 2$
jal	000011	label	$\$31 = pc; pc += i \ll 2$
jalr	001001	$\$s$	$\$31 = pc; pc = \s
jr	001000	$\$s$	$pc = \$s$

- jal = “jump and link”.
 - Register $\$31$ (aka $\$ra$) stores the address that’s used when returning from a subroutine.
- Note: jr and jalr are not j-type instructions.

Comparison instructions

Instruction	Opcode/Function	Syntax	Operation
slt	101010	\$d, \$s, \$t	\$d = (\$s < \$t)
sltu	101001	\$d, \$s, \$t	\$d = (\$s < \$t)
slti	001010	\$t, \$s, i	\$t = (\$s < SE(i))
sltiu	001001	\$t, \$s, i	\$t = (\$s < SE(i))

Note: Comparison operation stores a one in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.

If/Else statements in MIPS

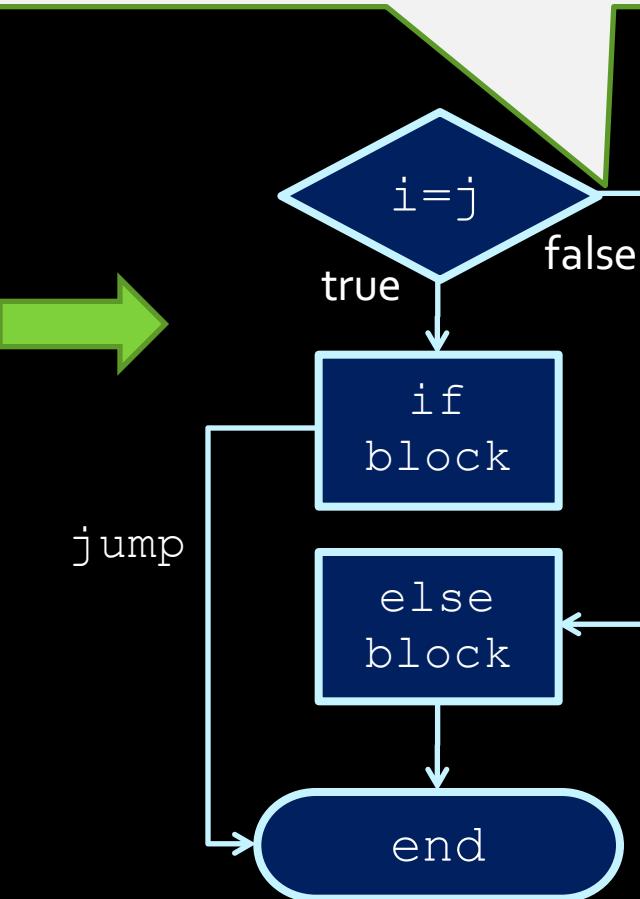
```
if ( i == j )
    i++;
else
    j--;
j += i;
```

- Strategy for if/else statements:
 - Test condition, and jump to if logic block whenever condition is true.
 - Otherwise, perform else logic block, and jump to first line after if logic block.
- A **flowchart** can be helpful here

If statement

Only problem: branch instructions jump on TRUE instead of FALSE, so negate the checked condition to $i \neq j$

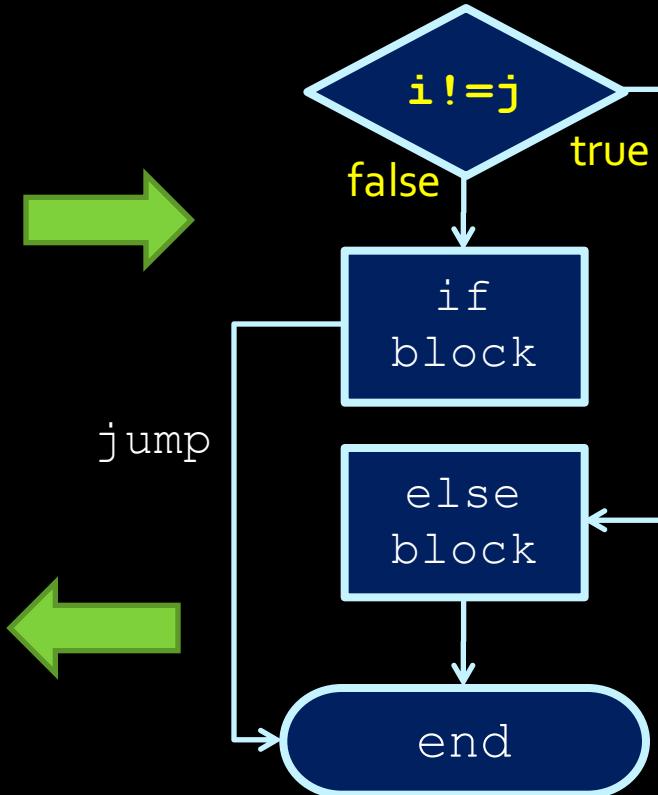
```
if ( i == j )
    i++;
else
    j--;
j += i;
```



If statement flowcharts

```
if ( i == j )
    i++;
else
    j--;
j += i;
```

```
# $t1 = i, $t2 = j
main:   bne $t1, $t2, ELSE
        addi $t1, $t1, 1
        j END
ELSE:   addi $t2, $t2, -1
END:    add $t2, $t2, $t1
```



Translated if/else statements

```
# $t1 = i, $t2 = j
main:    bne  $t1, $t2, ELSE      # branch if ! ( i == j )
          addi $t1, $t1, 1        # i++
          j END                  # jump over ELSE
ELSE:    addi $t2, $t2, -1        # j--
END:     add $t2, $t2, $t1        # j += i
```

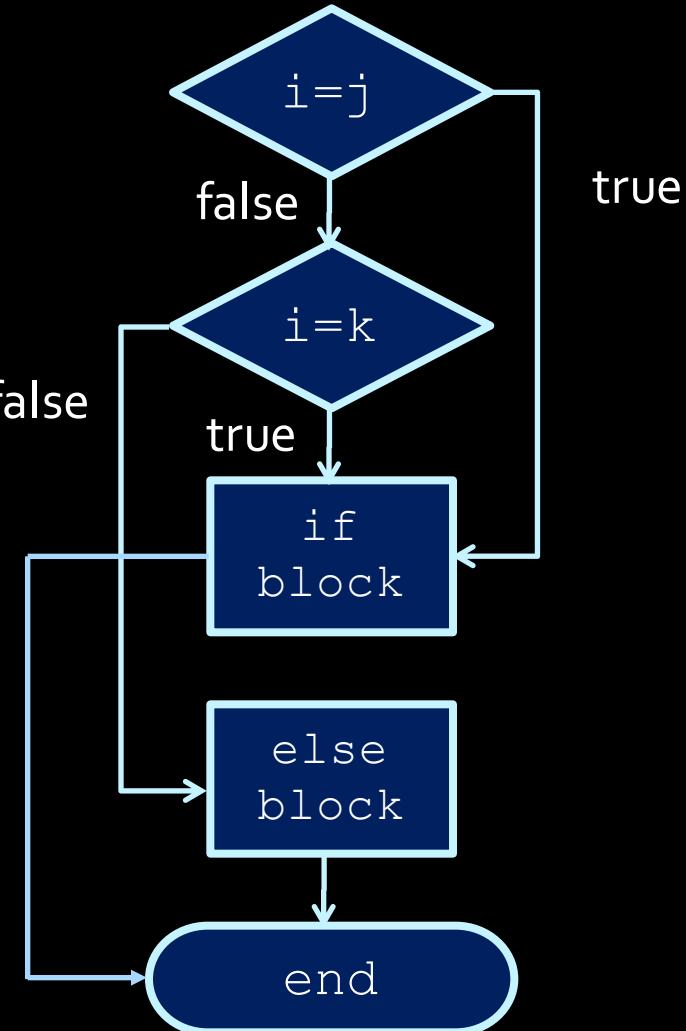
- If we change BNE to BEQ, then we also need to swap the IF and ELSE blocks

```
# $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF       # branch if ( i == j )
          addi $t2, $t2, -1        # j--
          j END                  # jump over IF
IF:      addi $t1, $t1, 1        # i++
END:     add $t2, $t2, $t1        # j += i
```

Multiple if conditions

```
if ( i == j || i == k )
    i++ ; // if-body
else
    j-- ; // else-body
j = i + k ;
```

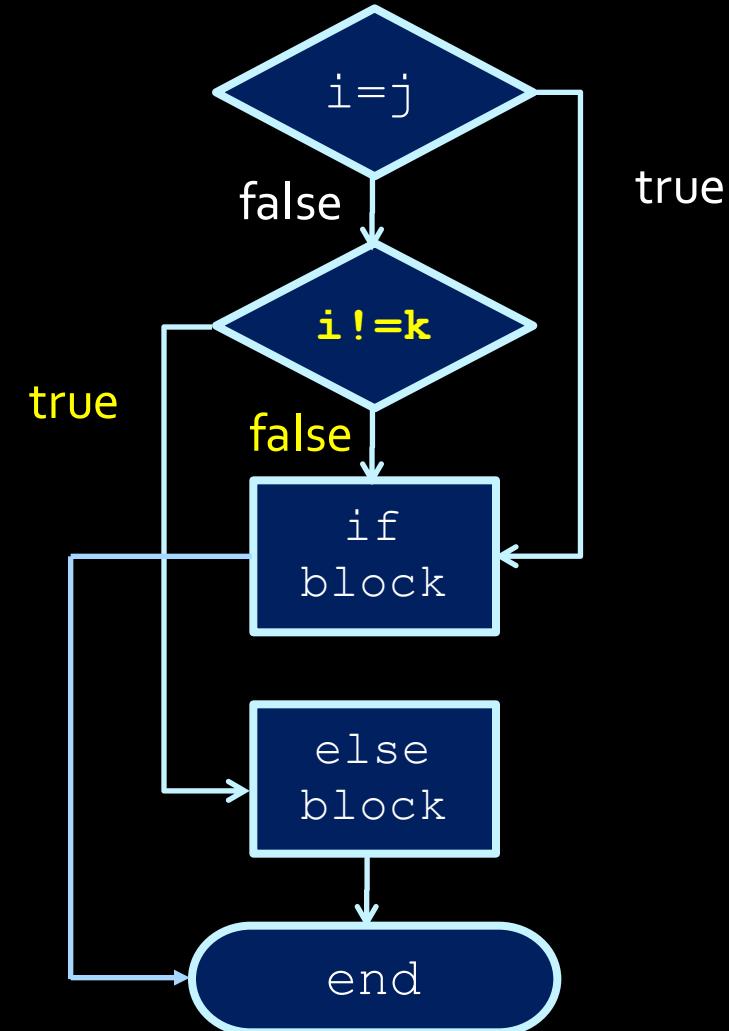
Branch on FALSE!



Multiple if conditions

```
if ( i == j || i == k )
    i++ ; // if-body
else
    j-- ; // else-body
j = i + k ;
```

```
# $t1 = i, $t2 = j, $t3 = k
main: beq $t1, $t2, IF
      bne $t1, $t3, ELSE
IF:   addi $t1, $t1, 1
      j END
ELSE: addi $t2, $t2, -1
END:  add $t2, $t1, $t3
```



Loops

Loops in MIPS (while loop)

- Example of a simple loop, in assembly:

```
# $t0 = i, $t1 = n
main:    add $t0, $zero, $zero      # i = 0
          addi $t1, $zero, 100       # n = 100
START:   beq $t0, $t1, END        # if i == n, END
          addi $t0, $t0, 1          # i++
          j START
END:
```

- ...which is the same as saying (in C):

```
while (i < 100) {
    i++;
}
```

For loop

```
for ( <init> ; <cond> ; <update> ) {  
    <for body>  
}
```

- For loops (such as above) are usually implemented with the following structure:

```
main:    <init>  
START:   if (!<cond>) branch to END  
        <for-body>  
UPDATE:  <update>  
        jump to START  
END:
```

Exercise:

```
j = 0
for ( _____ ; _____ ; _____ )
{
    j = j + i;
}
```

```
# $t0 = i, $t1 = j
main:   add $t1, $zero, $zero          # set j = 0
        add $t0, $zero, $zero          # set i = 0
        addi $t9, $zero, 100          # set $t9 to 100
START:  beq $t0, $t9, EXIT            # branch if i==100
        add $t1, $t1, $t0            # j = j + i
UPDATE: addi $t0, $t0, 1              # i++
        j START
EXIT:
```

Answer

```
j = 0
for ( i=0 ; i!=100 ; i++ )
{
    j = j + i;
}
```

- This translates to:

```
# $t0 = i, $t1 = j
main:    add $t1, $zero, $zero          # set j = 0
          add $t0, $zero, $zero          # set i = 0
          addi $t9, $zero, 100          # set $t9 to 100
START:   beq $t0, $t9, EXIT            # branch if i==100
          add $t1, $t1, $t0            # j = j + i
UPDATE:  addi $t0, $t0, 1              # i++
          j START
EXIT:
```

- while loops are the same, without the initialization and update sections.

Another exercise

- Fibonacci sequence:
 - How would you convert this into assembly?

```
int fib(void) {  
    int n = 10;  
    int f1 = 1, f2 = -1;  
  
    while (n != 0) {  
        f1 = f1 + f2;  
        f2 = f1 - f2;  
        n = n - 1;  
    }  
    return f1;  
}
```

Assembly code example

- Fibonacci sequence in assembly code:

```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
# RES refers to memory address of result
FIB: addi $t3, $zero, 10      # initialize n=10
      addi $t4, $zero, 1       # initialize f1=1
      addi $t5, $zero, -1      # initialize f2=-1
LOOP: beq $t3, $zero, END     # done loop if n==0
      add $t4, $t4, $t5        # f1 = f1 + f2
      sub $t5, $t4, $t5        # f2 = f1 - f2
      addi $t3, $t3, -1        # n = n - 1
      j LOOP                  # repeat until done
END:  sb $t4, RES             # store result
```

```
int fib(void) {
    int n = 10;
    int f1 = 1, f2 = -1;

    while (n != 0) {
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

Making an assembly program

- Assembly language programs typically have structure similar to simple Python or C programs:
 - They set aside registers to store data.
 - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose!
 - More on this later ☺

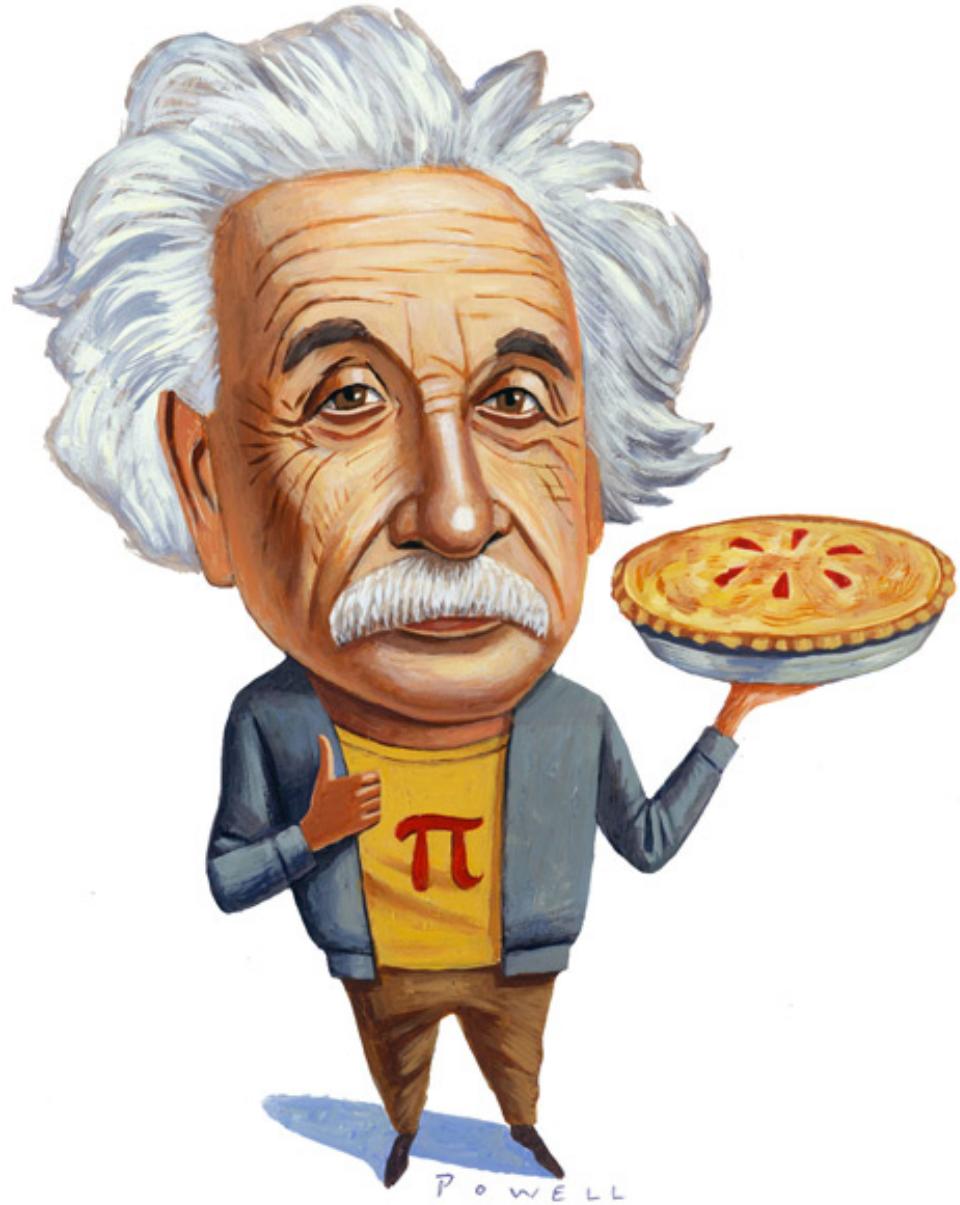
CSC258 Winter 2016

Lecture 10

Announcements

Midterm requests all processed. You can pick them up after the lecture.

Final exam schedule: Monday April 11, 9am~12pm, in IB110



Quiz Time!

Tear off the
reference sheet.

This quiz has 6
marks

Question 1

Complete the following assembly code, which **multiplies** the values in **\$t1** and **\$t2**, and stores the result in **\$t3**. Assume that the result is small enough to be represented by 32 bits.

```
mult $t1, $t2
```



Question 2

blt is a pseudo-instruction, the following line

```
blt $t1, $t2, Label
```

is equivalent to the following two lines combined. Complete them

```
____ $t3, $t1, $t2
```

```
bne $t3, _____, Label
```

Question 3

Translate the C program on the right by completing the following assembly code.

```
if (a > b && c > b) {  
    b++;  
}  
else {  
    b--;  
}  
c = a + b;
```

```
# $t1 = a, $t2 = b, $t3 = c  
IF:  
    ble $t1, $t2, _____  
    bge $t2, $t3, _____  
THEN:  
    addi $t2, $t2, 1  
  
ELSE: _____  
    addi $t2, $t2, -1  
END:  
    add $t3, $t1, $t2
```

Solutions

Question 1

Complete the following assembly code, which **multiplies** the values in **\$t1** and **\$t2**, and stores the result in **\$t3**. Assume that the result is small enough to be represented by 32 bits.

```
mult $t1, $t2
```

```
mflo $t3
```

Question 2

blt is a pseudo-instruction, the following line

```
blt $t1, $t2, Label
```

is equivalent to the following two lines, complete them

```
slt $t3, $t1, $t2
```

```
bne $t3, $zero, Label
```



Just “0” is not right

Question 3

Translate the C program on the right into assembly by completing the following code.

```
if (a > b && c > b) {  
    b++;  
}  
else {  
    b--;  
}  
c = a + b;
```

```
# $t1 = a, $t2 = b, $t3 = c  
IF:  
    ble $t1, $t2, ELSE  
    bge $t2, $t3, ELSE  
THEN:  
    addi $t2, $t2, 1  
    j END  
ELSE:  
    addi $t2, $t2, -1  
END:  
    add $t3, $t1, $t2
```

Loops

are easy once you understand how if-else (branch) works

Loops in MIPS (while loop)

- Example of a simple loop, in assembly:

```
# $t0 = i, $t1 = n
main:    add $t0, $zero, $zero      # i = 0
          addi $t1, $zero, 100       # n = 100
START:   beq $t0, $t1, END        # if i == n, END
          addi $t0, $t0, 1           # i++
          j START
END:
```

- ...which is the same as saying (in C):

```
int i = 0;
while (i != 100) {
    i++;
}
```

For loop

```
for ( <init> ; <cond> ; <update> ) {  
    <for body>  
}
```

- For loops (such as above) are usually implemented with the following structure:

```
main:    <init>  
START:   if (!<cond>) branch to END  
        <for-body>  
UPDATE:  <update>  
        jump to START  
END:
```

Exercise:

```
j = 0
for ( _____ ; _____ ; _____ )
{
    j = j + i;
}
```

```
# $t0 = i, $t1 = j
main:   add $t1, $zero, $zero          # set j = 0
        add $t0, $zero, $zero          # set i = 0
        addi $t9, $zero, 100          # set $t9 to 100
START:  beq $t0, $t9, EXIT            # branch if i==100
        add $t1, $t1, $t0            # j = j + i
UPDATE: addi $t0, $t0, 1              # i++
        j START
EXIT:
```

Answer

```
j = 0
for ( i=0 ; i!=100 ; i++ )
{
    j = j + i;
}
```

- This translates to:

```
# $t0 = i, $t1 = j
main:    add $t1, $zero, $zero          # set j = 0
          add $t0, $zero, $zero          # set i = 0
          addi $t9, $zero, 100          # set $t9 to 100
START:   beq $t0, $t9, EXIT            # branch if i==100
          add $t1, $t1, $t0            # j = j + i
UPDATE:  addi $t0, $t0, 1              # i++
          j START
EXIT:
```

- while loops are the same, without the initialization and update sections.

Another exercise

- Fibonacci sequence:
 - How would you convert this into assembly?

```
int fib(void) {  
    int n = 10;  
    int f1 = 1, f2 = -1;  
  
    while (n != 0) {  
        f1 = f1 + f2;  
        f2 = f1 - f2;  
        n = n - 1;  
    }  
    return f1;  
}
```

Assembly code example

- Fibonacci sequence in assembly code:

```
# fib.s
# register usage: $t3=n, $t4=f1, $t5=f2
# RES refers to memory address of result
FIB: addi $t3, $zero, 10      # initialize n=10
      addi $t4, $zero, 1       # initialize f1=1
      addi $t5, $zero, -1      # initialize f2=-1
LOOP: beq $t3, $zero, END     # done loop if n==0
      add $t4, $t4, $t5        # f1 = f1 + f2
      sub $t5, $t4, $t5        # f2 = f1 - f2
      addi $t3, $t3, -1        # n = n - 1
      j LOOP                  # repeat until done
END:  sw $t4, RES             # store result
```

```
int fib(void) {
    int n = 10;
    int f1 = 1, f2 = -1;

    while (n != 0) {
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

Making an assembly program

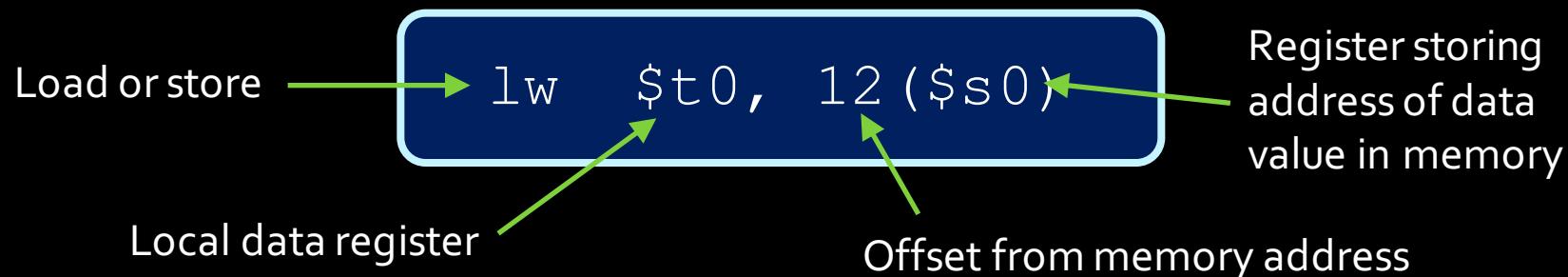
- Assembly language programs typically have structure similar to simple Python or C programs:
 - They set aside registers to store data.
 - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose

Only a few more
instructions left!

- Memory access
- System calls

Interacting with memory

- All of the previous instructions perform operations on **registers** and **immediate values**.
 - What about **memory**?
- All programs must fetch values from memory into registers, operate on them, and then store the values back into memory.
- Memory operations are l-type, with the form:



Quick reminder

Word: 4-byte

Half-word: 2-byte

Byte: 1-byte

Load & store instructions

Instruction	Opcode/Function	Syntax	Operation
lb	100000	\$t, i (\$s)	\$t = SE (MEM [\$s + i]:1)
lbu	100100	\$t, i (\$s)	\$t = ZE (MEM [\$s + i]:1)
lh	100001	\$t, i (\$s)	\$t = SE (MEM [\$s + i]:2)
lhu	100101	\$t, i (\$s)	\$t = ZE (MEM [\$s + i]:2)
lw	100011	\$t, i (\$s)	\$t = MEM [\$s + i]:4
sb	101000	\$t, i (\$s)	MEM [\$s + i]:1 = LB (\$t)
sh	101001	\$t, i (\$s)	MEM [\$s + i]:2 = LH (\$t)
sw	101011	\$t, i (\$s)	MEM [\$s + i]:4 = \$t

- “b”, “h” and “w” correspond to “byte”, “half word” and “word”, indicating the length of the data.
- LB: lowest byte; LH: lowest half word

Examples

```
lh    $t0, 12($s0)
```

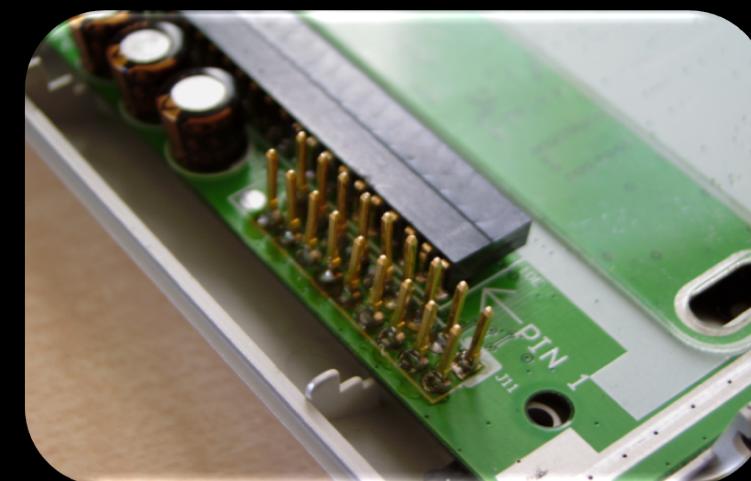
Load a **half-word** (2 bytes) starting from $\text{MEM}(\$s + 12)$,
sign-extend it to 4 bytes, and store in $\$t0$

```
sb    $t0, 12($s0)
```

Take the **lowest byte** of the word stored in $\$t0$,
store it to memory starting from address $\$s0 + 12$

A bit more about memory

- The offset value is useful for arrays or stack parameters, when multiple values are needed from a given memory location.
- Memory is also used to communicate with outside devices, such as keyboards and monitors.
 - Known as **memory-mapped IO**.
 - Invoked with a **trap** or function.





It's a Trap!

Instruction	Function	Syntax
trap	011010	i

- Trap instructions send system calls to the operating system
 - e.g. interacting with the user, and exiting the program, raise exceptions.
- Similar to the syscall command.
 - use registers \$ao and \$vo

\$4 is \$a0, \$2 is \$v0

Service	Trap Code	Input/Output
print_int	1	\$4 is int to print
print_float	2	\$f12 is float to print
print_double	3	\$f12 (with \$f13) is double to print
print_string	4	\$4 is address of ASCII string to print
read_int	5	\$2 is int read
read_float	6	\$f12 is float read
read_double	7	\$f12 (with \$f13) is doubleread
read_string	8	\$4 is address of buffer, \$5 is buffer size in bytes
sbrk	9	\$4 is number of bytes required, \$2 is address of allocated memory
exit	10	
print_byte	101	\$4 contains byte to print
read_byte	102	\$2 contains byte read
set_print_inst_on	103	
set_print_inst_off	104	
get_print_inst	105	\$2 contains current status of printing instructions

syscall example

```
li $v0, 4          # $v0 stores syscal number, 4 is print_string  
la $a0, promptA # $a0 stores the address of the string to print  
syscall           # check $v0 and $a0 and act accordingly
```

Arrays and Structs

Data storage

- At beginning of program, create labels for memory locations that are used to store values.
- Always in form: **label type value**

```
.data
# create a single integer variable with initial value 3
var1:      .word      3
# create a 4-element integer array
array0:     .word      3, 7, 5, 42
# create a 2-element character array with elements
# initialized to a and b
array1:     .byte      'a', 'b'
# allocate 40 consecutive bytes, with uninitialized
# storage. Could be used as a 40-element character array,
# or a 10-element integer array.
array2:     .space     40
```

Integer type (int): 4 byte

Character type (char): 1 byte

Arrays and Structs



Arrays!

```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

- Arrays in assembly language:
 - The address of the **first element** of the array is used to store and access the elements of the array.
 - To access an element of the array, get the address of that element by adding an **offset** distance to the address of the first element.
 - **offset = array index * the size of a single element**
 - Arrays are stored in memory. For examples, fetch the array values and store them in registers. Operate on them, then store them back into memory.

```
int A[100];
```



Offset = 4×4 bytes = 16 bytes

Address of A[4] = Address of A[0] + 16

```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

Making sense of assembly code

- The key to reading and designing assembly code is recognizing portions of code that represent higher-level operations that you're familiar with.

```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

```
.data
A:      .space 400
B:      .space 400

.text
main:   add $t0, $zero, $zero
        addi $t1, $zero, 400
        la $t8, A
        la $t9, B

loop:   add $t4, $t8, $t0
        add $t3, $t9, $t0
        lw $s4, 0($t3)
        addi $t6, $s4, 1
        sw $t6, 0($t4)
        addi $t0, $t0, 4
        bne $t0, $t1, loop

end:
```

Initialization:

- Allocate space
- Initial value **i=0** (**offset**), put into a register
- Put value **size (400)** in register
- Put addresses of A, B into register

The loop:

- Put **addrs** of A[i] and B[i] into registers (addr(A)+offset).
- Load B[i] from mem, then **+1**, keep result in a register
- Store result into mem A[i]
- Update **i++**
- Check loop condition and jump

Code with comments

```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

```
.data
A:      .space 400      # array of 400 bytes (100 ints)
B:      .space 400      # array of 400 bytes (100 ints)

.text
main:   add $t0, $zero, $zero      # load "0" into $t0
        addi $t1, $zero, 400      # load "400" into $t1
        la $t9, B                # store address of B
        la $t8, A                # store address of A

loop:   add $t4, $t8, $t0      # $t4 = addr(A) + i
        add $t3, $t9, $t0      # $t3 = addr(B) + i
        lw $s4, 0($t3)        # $s4 = B[i]
        addi $t6, $s4, 1       # $t6 = B[i] + 1
        sw $t6, 0($t4)        # A[i] = $t6
        addi $t0, $t0, 4       # $t0 = $t0++
        bne $t0, $t1, loop    # branch back if $t0<400

end:
```

Struct

Example: A struct program

- How can we figure out the main purpose of this code?
- The `sw` and `sb` lines indicate that values in `$t1` are being stored at `$t0`, `$t0+4` and `$t0+5`.
 - Each previous line sets the value of `$t1` to store.
- Therefore, this code stores the values 5, 'B' (ascii 66) and 12 into the struct at location `a1`.

```
.data
a1:    .space    9

.text
main:   la      $t0, a1
        addi   $t1, $zero, 5
        sw     $t1, 0($t0)
        addi   $t1, $zero, 'B'
        sb     $t1, 4($t0)
        addi   $t1, $zero, 12
        sw     $t1, 5($t0)
```

Example: A struct program

```
struct foo {  
    int a;  
    char b;  
    int c;  
};  
  
struct foo x;  
x.a= 5;  
x.b = 'B';  
x.c = 12;
```

```
.data  
a1: .space 9  
  
.text  
main: la $t0, a1  
      addi $t1, $zero, 5  
      sw $t1, 0($t0)  
      addi $t1, $zero, 'B'  
      sb $t1, 4($t0)  
      addi $t1, $zero, 12  
      sw $t1, 5($t0)
```

Struct program with comments

```
.data
a1:    .space   9          # declare 9 bytes
                           # of storage to hold
                           # struct of 2 ints and
                           # 1 char

.text
main:   la      $t0, a1      # load base address
                           # of struct into
                           # register $t0
       addi   $t1, $zero, 5    # $t1 = 5
       sw     $t1, 0($t0)      # first struct
                           # element set to 5;
                           # indirect addressing
       addi   $t1, $zero, 'B'   # $t1 = 'B', i.e., 66
       sb     $t1, 4($t0)      # second struct
                           # element set to 'B'
       addi   $t1, $zero, 12    # $t1 = 12
       sw     $t1, 5($t0)      # third struct
                           # element set to 12
```

```
struct foo {
    int a;
    char b;
    int c;
};

struct foo x;
x.a= 5;
x.b = 'B';
x.c = 12;
```

Function calls

Another example:

A function!

Function
arguments!

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
int x, r;  
x = 42;  
r = sign(x);  
r = r + 1;  
...
```

Return!

Function arguments

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
r = res + 1;  
...
```

Where are the function arguments stored?

They are stored at a certain location in the **memory**, which is call the **stack**.

Other conventions are also possible, i.e., store first 4 arguments in \$ao~\$a3, the rest in the stack

Memory model: a quick look



High address

Stack grows this way (going low)

If they collide



Heap grows this way (going high)

Low address

Note: stack grows **backwards**, i.e., when stack pointer (top) decreases, stack becomes bigger; when stack pointer increase, stack becomes smaller.

Function arguments

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
r = res + 1;  
...
```

Why keep the arguments
in **memory** instead of
registers?

Because there aren't
enough registers for this

- One function may have many arguments
- If function calls subroutines, all subroutines' arguments need to be remembered.
(can't forget until function returns)

Note

Of course, you can use the **registers** to store function arguments if you know you have enough registers to do so (e.g., one single-argument function with no subroutines).

An **assembly** programmer makes this type of design decisions and can do whatever they want.

For high-level language programmers, the **complier** makes this type of decisions for them.

How to access stack?

The address of the “top” of the stack is stored in this register -- **\$sp**

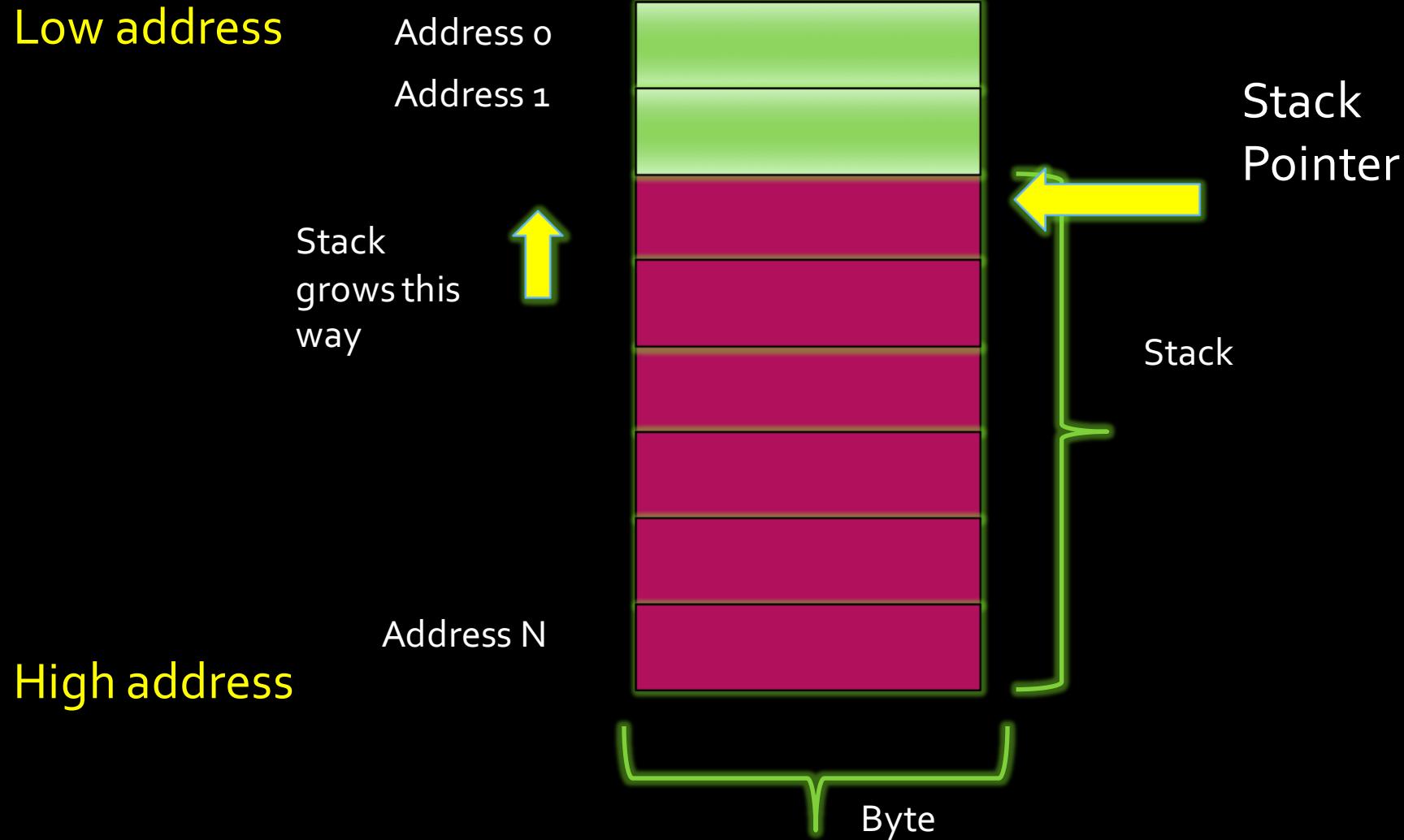
PUSH value in \$t0 into stack

```
addi    $sp, $sp, -4 # move stack pointer to make space  
sw      $t0, 0($sp) # push a word onto the stack
```

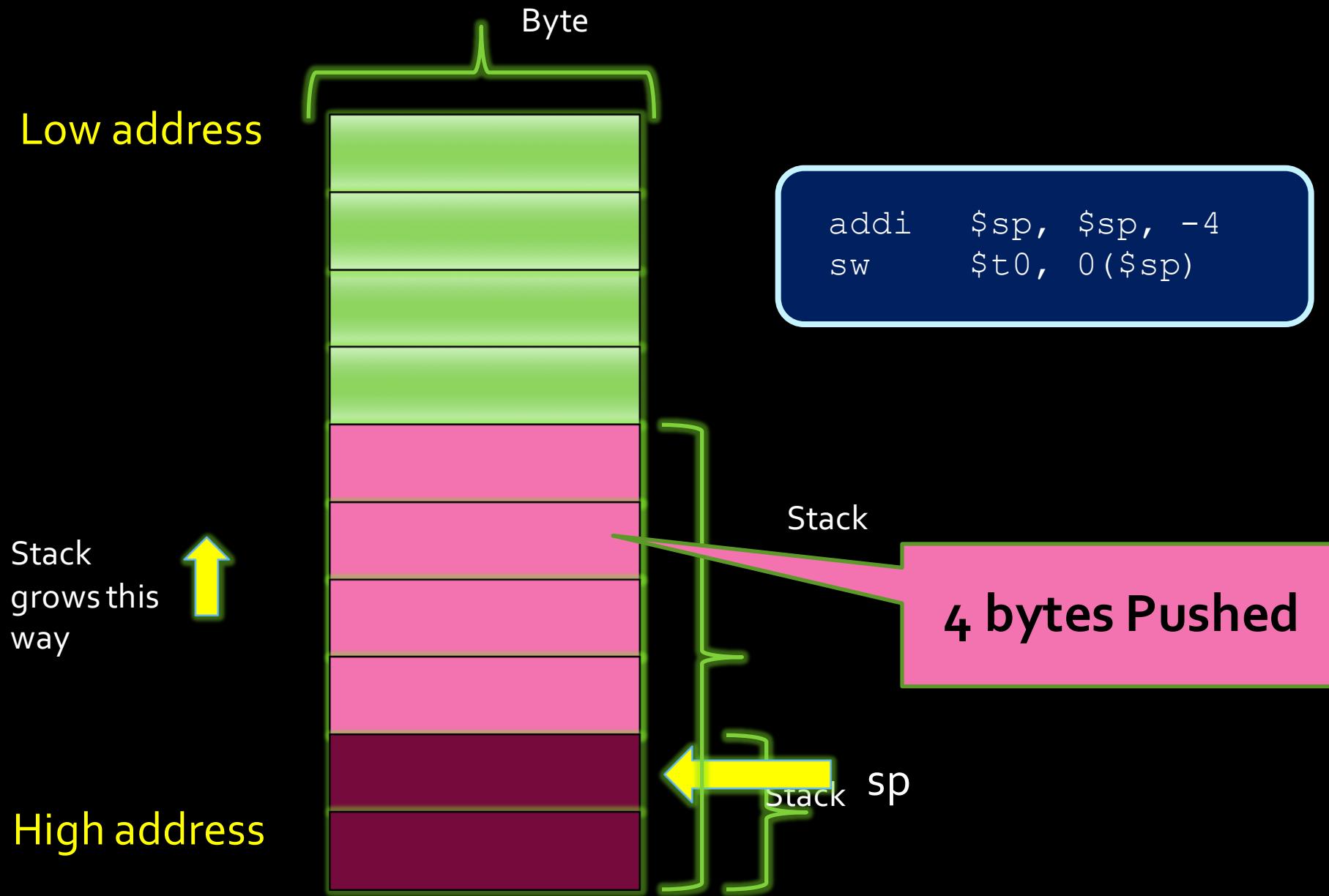
POP a value from stack and store in \$t0

```
lw      $t0, 0($sp) # pop a word from the stack  
addi    $sp, $sp, 4 # update stack pointer, stack size smaller
```

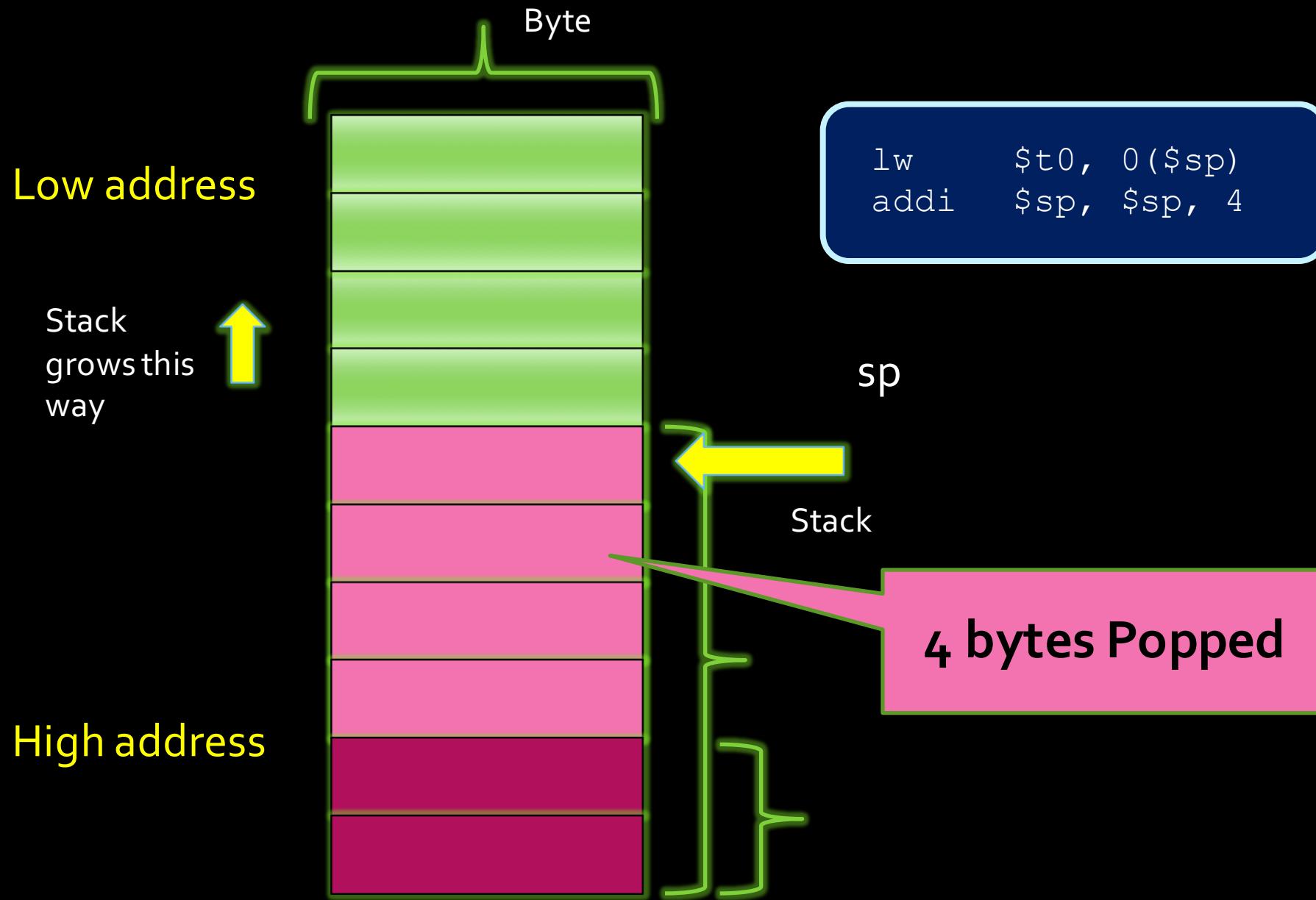
The Stack



Pushing Values to the stack



Popping Values off the stack



Return address



```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
  
    int x, r;  
    x = 42;  
    r = sign(x);  
    r = res + 1;  
    ...
```

How do we pass the **return value** to the caller?

Answer: let's use the **stack**.

Where do we keep the **return address**?

Answer: let's use **\$ra register**.
To return: **jr \$ra**

This is a design choice, NOT the only way to do it

The whole story: “when **Caller** calls **Callee**”

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
r = res + 1;  
...
```

1. **Caller** pushes arguments to the stack
2. **Caller** stores return address to **\$ra**
3. **Callee** invoked, pop arguments from stack
4. **Callee** computes the return value
5. **Callee** pushes the return value into the stack
6. Jump to return addressed stored in **\$ra**
7. **Caller** pops return value from the stack.
8. Move on to next line...

Now, ready to translate the code

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
.text  
sign: lw $t0, 0($sp)  
      addi $sp, $sp, 4  
      bgtz $t0, gt  
      beq $t0, $zero, eq  
      addi $t1, $zero, -1  
      j end  
gt:   addi $t1, $zero, 1  
      j end  
eq:   add $t1, $zero, $zero  
end:  addi $sp, $sp, -4  
      sw $t1, 0($sp)  
      jr $ra
```

1. Callee invoked, pop arguments from stack
2. Callee computes the return value
3. Callee pushes the return value into the stack
4. Jump to return addressed stored in \$ra
5. Caller get return value from the stack.

Code with comments

```
.text
sign: lw $t0, 0($sp)          # pop arg i from
      addi $sp, $sp, 4        # the stack

      bgtz $t0, gt            # if ( i > 0)
      beq $t0, $zero, eq       # if ( i == 0)
      addi $t1, $zero, -1      # i < 0, return value = -1
      j end                   # jump to return
gt:   addi $t1, $zero, 1       # i > 0, return value = 1
      j end                   # jump to return
eq:   add $t1, $zero, $zero  # i == 0, return value = 0
end:  addi $sp, $sp, -4       # push return value to
      sw $t1, 0($sp)         # the stack
      jr $ra                  # return
```

Takeaway

What we did is based on one **function call convention** that we defined, there could be other conventions.

Function calls don't happen for free, it involves manipulating the values of several registers, and accessing memory.

All of these have performance implications.

Why “**inline functions**” are faster? Because the the callee assembly code is **inline** with the the caller code (callee code is copied to everywhere its called, rather than at a different location), so no need to jump, i.e., no stack and \$ra manipulations needed.

Now you really understand when to use inline, and when not to.

Practice for home: String function

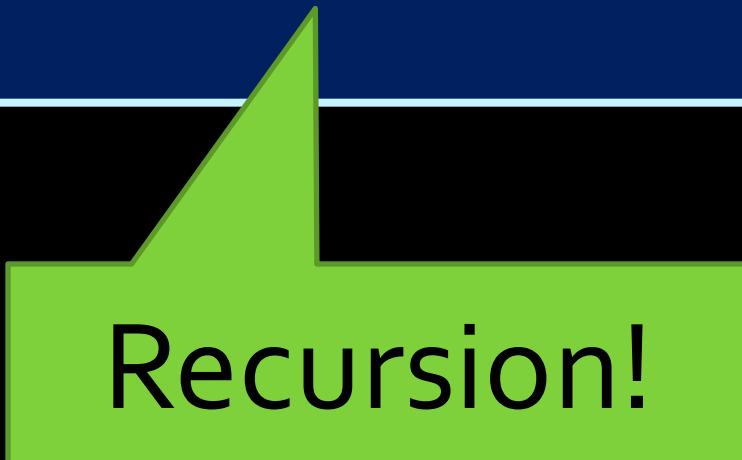
```
int strcpy (char x[ ], char y[ ]) {  
    int i;  
    i=0;  
    while ((x[i] = y[i]) != '\0')  
        i += 1;  
    return i;  
}
```

Translated string program

```
strcpy:      lw    $a0, 0($sp)          # pop x address
             addi $sp, $sp, 4           # off the stack
             initialization {        lw    $a1, 0($sp)          # pop y address
                                         addi $sp, $sp, 4           # off the stack
                                         add  $s0, $zero, $zero   # $s0 = offset i
                                         add  $t1, $s0, $a0       # $t1 = x + i
                                         lb   $t2, 0($t1)         # $t2 = x[i]
                                         add  $t3, $s0, $a1       # $t3 = y + i
                                         sb   $t2, 0($t3)         # y[i] = $t2
                                         beq $t2, $zero, L2       # y[i] = '\0'?
                                         addi $s0, $s0, 1          # i++
                                         j    L1                  # loop
L1:          end {                    addi $sp, $sp, -4      # push i onto
                                         sw   $s0, 0($sp)          # top of stack
                                         jr   $ra                  # return
L2:
```

Next one

```
int factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```



Recursion!

CSC258 Winter 2016

Lecture 11

Announcements

QUIZ

FINALS

Tear off the reference sheet

Question 1

Complete the assembly translation of the C code.

```
int i;
int j = 258;
for (i = 42; j > i; i = i*2) {
    j++;
}
i = j;
```

```
# t1 = i, t2 = j
START:
    li $t2, 258
    li $t1, 42
LOOP:
    ble $t2, $t1, END
    addi $t2, $t2, 1
    

---

---


END:
    move $t1, $t2
```

Question 2

What is the content of array after executing the following code?

```
.data
array:      .word      2, 5, 8
.text
main:
    la $s0, array
    lw $t0, 4($s0)
    addi $t0, $t0, 1
    sw $t0, 4($s0)
    lw $t0, 0($s0)
    addi $t0, $t0, 1
    addi $s0, $s0, 8
    sw $t0, 0($s0)
```

Question 3

Complete the assembly code (3 numbers to be filled) so that it is equivalent to the C code with struct.

```
struct foo {  
    char A;      # 1-byte  
    short int B; # 2-byte  
    int C;       # 4-byte  
};  
  
struct foo x;  
x.A = 55;  
x.B = 66;  
x.C = 77;
```

```
.data  
s1: .space _____  
  
.text  
main: la    $t0, s1  
      li    $t1, 55  
      sb    $t1, 0($t0)  
      li    $t1, 66  
      sh    $t1, _____($t0)  
      li    $t1, 77  
      sw    $t1, _____($t0)
```

Solution

Question 1

Complete the assembly translation of the C code.

```
int i;
int j = 258;
for (i = 42; j > i; i = i*2) {
    j++;
}
i = j;
```

```
# t1 = i, t2 = j
START:
    li $t2, 258
    li $t1, 42
LOOP:
    ble $t2, $t1, END
    addi $t2, $t2, 1
    sll $t1, $t1, 1
    j LOOP
END:
    move $t1, $t2
```

Question 2

What is the content of array after executing the following code?

```
.data
array:      .word    2, 5, 8

.text
main:
    la $s0, array      # load addr of A
    lw $t0, 4($s0)      # load A[1]: 5
    addi $t0, $t0, 1     # 5 + 1 = 6
    sw $t0, 4($s0)      # store A[1] = 6
    lw $t0, 0($s0)      # load A[0]: 2
    addi $t0, $t0, 1     # 2 + 1 = 3
    addi $s0, $s0, 8      # $s0 changed to A[2] addr
    sw $t0, 0($s0)      # store A[2] = 3
```

2, 6, 3

Question 3

Complete the assembly code so that it is equivalent to the C code with struct.

```
struct foo {  
    char A;      # 1-byte  
    short int B; # 2-byte  
    int C;       # 4-byte  
};  
  
struct foo x;  
x.A = 55;  
x.B = 66;  
x.C = 77;
```

```
.data  
s1: .space 7  
  
.text  
main: la    $t0, s1  
      li    $t1, 55  
      sb    $t1, 0($t0)  
      li    $t1, 66  
      sh    $t1, 1($t0)  
      li    $t1, 77  
      sw    $t1, 3($t0)
```

Function calls

Another example:

A function!

Function
arguments!

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
int x, r;  
x = 42;  
r = sign(x);  
r = r + 1;  
...
```

Return!

Function arguments

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
r = res + 1;  
...
```

Where are the function arguments stored?

They are stored at a certain location in the **memory**, which is call the **stack**.

Other conventions are also possible, i.e., store first 4 arguments in \$ao~\$a3, the rest in the stack

Note

- Because assembly programmers have so much control over how things are done at the low level, there are always **multiple** ways of implementing a feature.
- We need to define a **convention** of how function arguments and return values are passed between functions, etc, so all programmers working on the same project are on the same page.
- There can be many different version of the conventions.

Memory model: a quick look



High address

Stack grows this way (going low)

If they collide



Heap grows this way (going high)

Low address

Note: stack grows **backwards**, i.e., when stack pointer (top) decreases, stack becomes bigger; when stack pointer increase, stack becomes smaller.

Function arguments

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
r = res + 1;  
...
```

Why keep the arguments
in **memory** instead of
registers?

Because there aren't
enough registers for this

- One function may have many arguments
- If function calls subroutines, all subroutines' arguments need to be remembered.
(can't forget until function returns)

Note

You can use the **registers** to store function arguments if you know you have enough registers to do so (e.g., one single-argument function with no subroutines).

An **assembly** programmer makes this type of design decisions and can do whatever they want.

For high-level language programmers, the **complier** makes this type of decisions for them.

How to access stack?

The address of the “top” of the stack is stored in this register -- **\$sp**

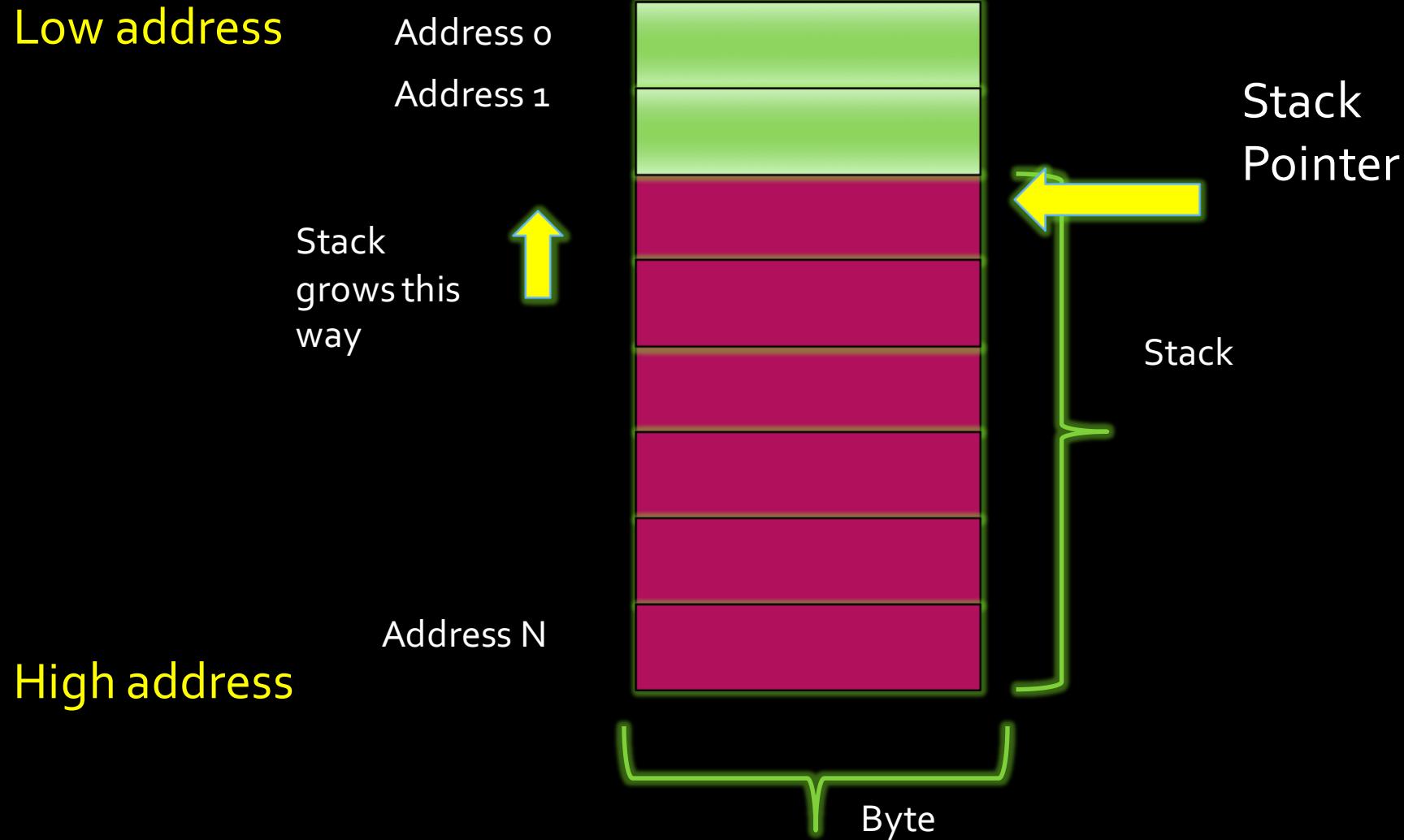
PUSH value in \$t0 into stack

```
addi    $sp, $sp, -4 # move stack pointer to make space  
sw      $t0, 0($sp) # push a word onto the stack
```

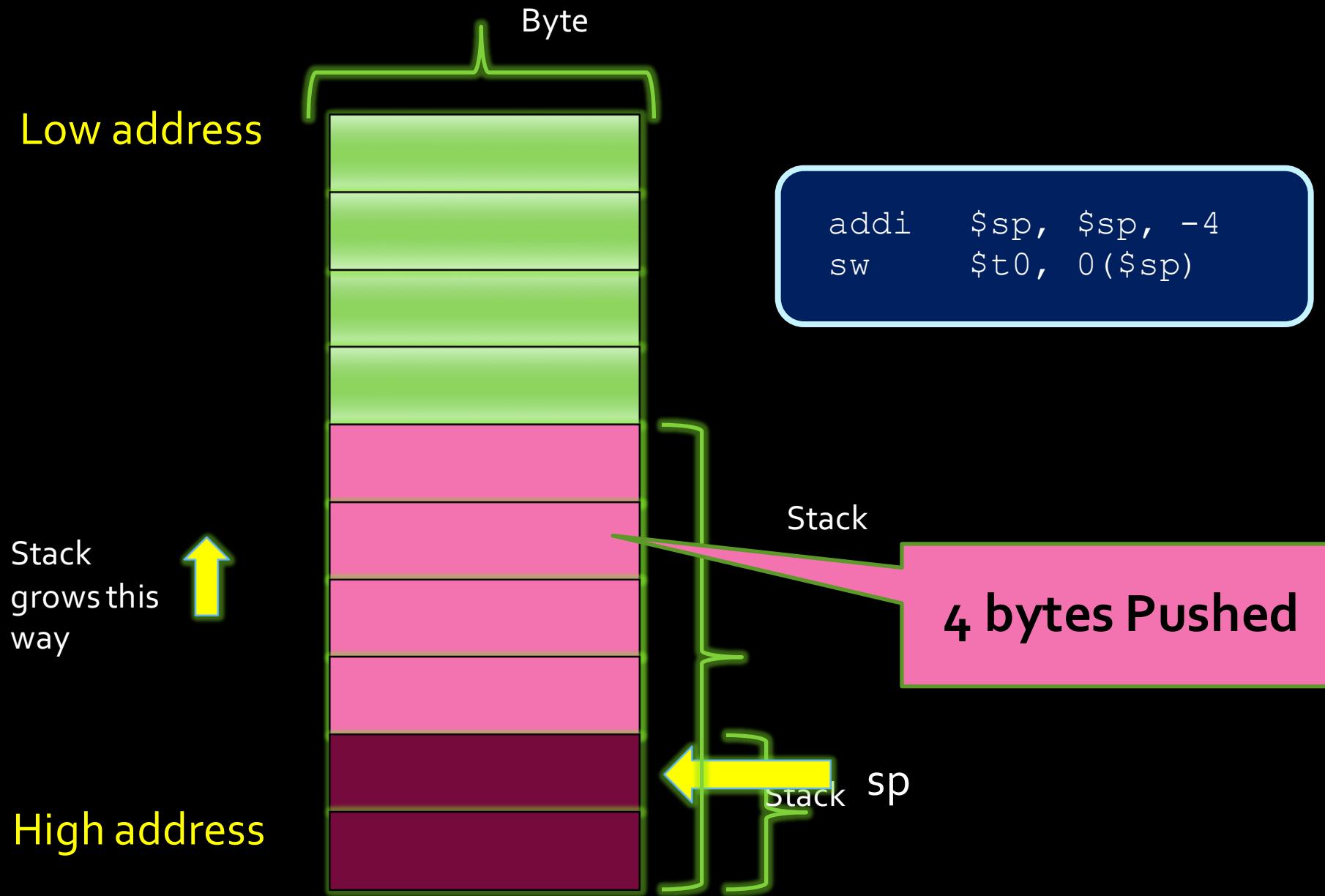
POP a value from stack and store in \$t0

```
lw      $t0, 0($sp) # pop a word from the stack  
addi    $sp, $sp, 4 # update stack pointer, stack size smaller
```

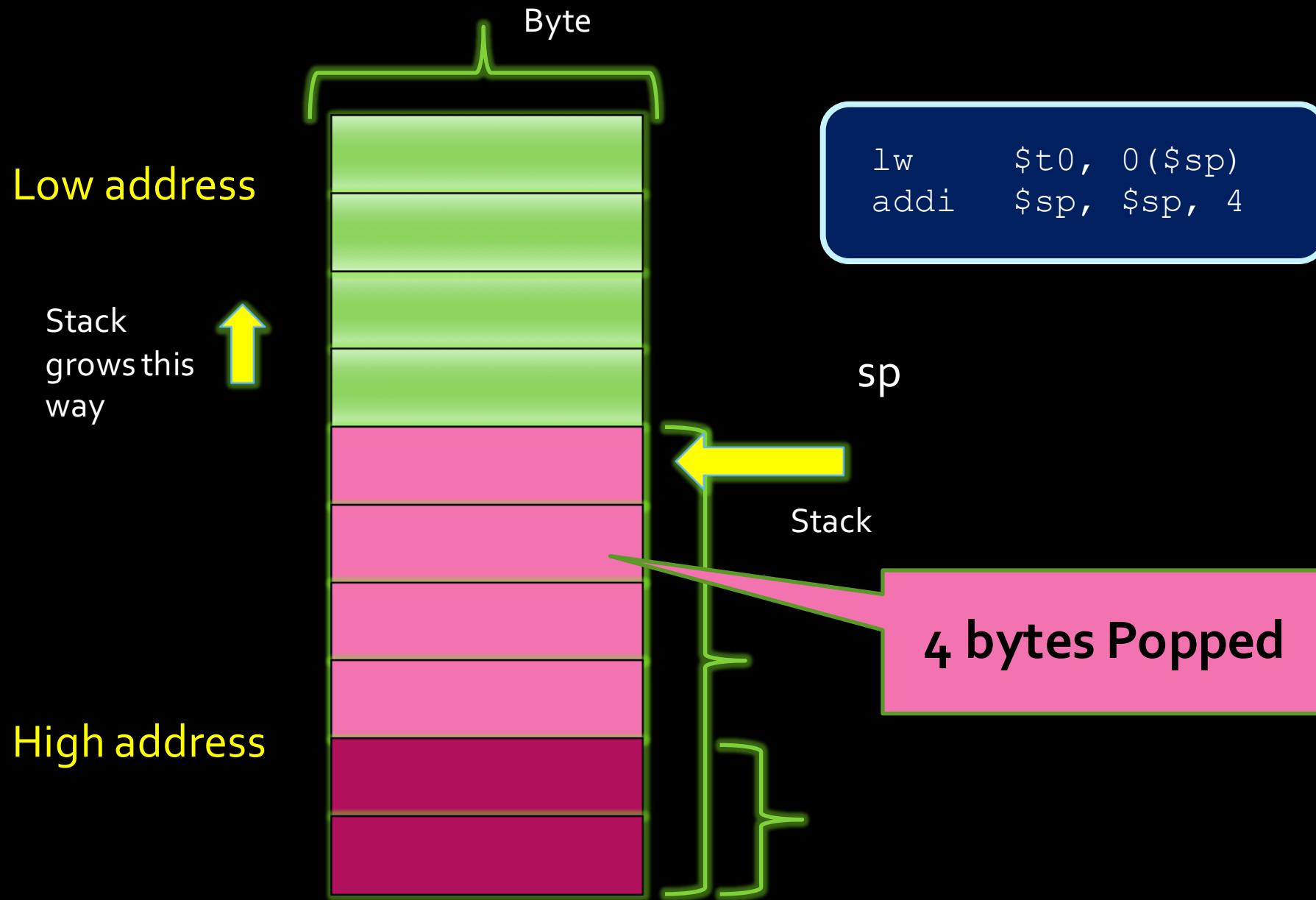
The Stack



Pushing Values to the stack



Popping Values off the stack



Return value/address



```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
  
    int x, r;  
    x = 42;  
    r = sign(x);  
    r = res + 1;  
    ...
```

How do we pass the **return value** to the caller?

Answer: let's use the **stack**.

Where do we keep the **return address**?

Answer: let's use **\$ra register**.

To return: **jr \$ra**

This is a design choice, NOT the only way to do it

The whole story: “when **Caller** calls **Callee**”

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
r = res + 1;  
...
```

1. **Caller** pushes arguments to the stack
2. **Caller** stores return address to **\$ra**
3. **Callee** invoked, pop arguments from stack
4. **Callee** computes the return value
5. **Callee** pushes the return value into the stack
6. Jump to return addressed stored in **\$ra**
7. **Caller** pops return value from the stack.
8. Move on to next line...

Now, ready to translate the code

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
.text  
sign: lw $t0, 0($sp)  
      addi $sp, $sp, 4  
      bgtz $t0, gt  
      beq $t0, $zero, eq  
      addi $t1, $zero, -1  
      j end  
gt:   addi $t1, $zero, 1  
      j end  
eq:   add $t1, $zero, $zero  
end:  addi $sp, $sp, -4  
      sw $t1, 0($sp)  
      jr $ra
```

1. Callee invoked, pop arguments from stack
2. Callee computes the return value
3. Callee pushes the return value into the stack
4. Jump to return addressed stored in \$ra
5. Caller get return value from the stack.

Code with comments

```
.text
sign: lw $t0, 0($sp)          # pop arg i from
      addi $sp, $sp, 4        # the stack

      bgtz $t0, gt            # if ( i > 0)
      beq $t0, $zero, eq       # if ( i == 0)
      addi $t1, $zero, -1      # i < 0, return value = -1
      j end                   # jump to return
gt:   addi $t1, $zero, 1       # i > 0, return value = 1
      j end                   # jump to return
eq:   add $t1, $zero, $zero  # i == 0, return value = 0
end:  addi $sp, $sp, -4       # push return value to
      sw $t1, 0($sp)         # the stack
      jr $ra                  # return
```

Note

In Lab 10, you will implement a different convention, so don't just imitate the code in the slides for the Lab.

Takeaway

What we did is based on one **function call convention** that we defined, there could be other conventions.

Function calls don't happen for free, it involves manipulating the values of several registers, and accessing memory.

All of these have performance implications.

Why “**inline functions**” are faster? Because the the callee assembly code is **inline** with the the caller code (callee code is copied to everywhere its called, rather than at a different location), so no need to jump, i.e., no stack and \$ra manipulations needed.

Now you really understand when to use inline, and when not to.

More takeaway

When we make multiple levels of function calls, the **return address** also need to be stored on stack, since the deeper level function call will overwrite the **\$ra** registers. You will experience this in Lab 10.

Before calling a function all temporary register values need to be pushed to the stack, too. After returning from the called function, you restored the register values from the stack and continue using them.

```
int foo() {  
    int i, j;  
    i=5  
    j=6+i;  
    # save temps to stack  
    bar();  
    # restore from stack  
    i++;  
    printf("%d %d", i, j);  
}
```

Practice for home: String function

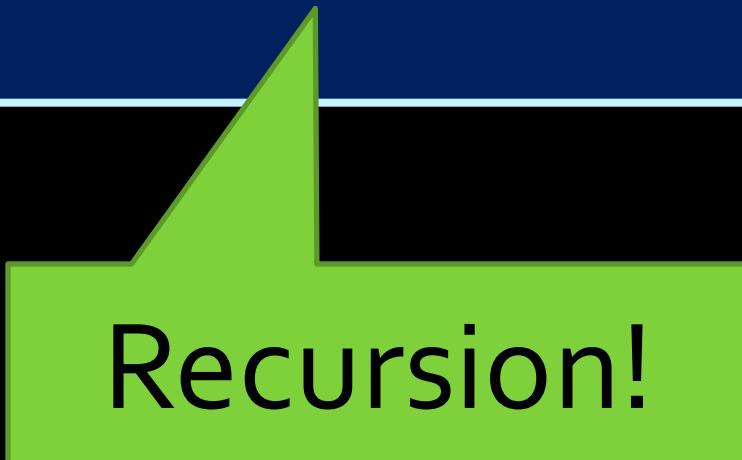
```
int strcpy (char x[ ], char y[ ]) {  
    int i;  
    i=0;  
    while ((x[i] = y[i]) != '\0')  
        i += 1;  
    return i;  
}
```

Translated string program

```
strcpy:    lw      $a0, 0($sp)          # pop x address
            addi   $sp, $sp, 4           # off the stack
            initialization {          |
                lw      $a1, 0($sp)          # pop y address
                addi   $sp, $sp, 4           # off the stack
                add    $s0, $zero, $zero     # $s0 = offset i
                add    $t1, $s0, $a0         # $t1 = x + i
                lb      $t2, 0($t1)          # $t2 = x[i]
                add    $t3, $s0, $a1         # $t3 = y + i
                sb      $t2, 0($t3)          # y[i] = $t2
                beq   $t2, $zero, L2        # y[i] = '\0'?
                addi   $s0, $s0, 1           # i++
                L1:   j       L1              # loop
                addi   $sp, $sp, -4          # push i onto
                sw      $s0, 0($sp)          # top of stack
                jr      $ra              # return
            main algorithm {          |
                L2:   end {               |
                    j       L1              # loop
                    addi   $sp, $sp, -4          # push i onto
                    sw      $s0, 0($sp)          # top of stack
                    jr      $ra              # return
                }
            }
```

Next one

```
int factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```



Recursion!

Recursion in Assembly

what recursion really is in hardware



Example: factorial(int n)

- Basic pseudocode for recursive factorial:

- Base Case ($n == 0$)
 - return 1
- Get $\text{factorial}(n-1)$
 - Store result in “product”
- Multiply product by n
 - Store in “result”
- Return result



```
factorial(3)
p = 3 * factorial(2)
```

```
factorial(2)
p = 2 * factorial(1)
```

```
factorial(1)
p = 1*factorial(0)
```

```
factorial (0)
p = 1 # Base!
return p
```

```
return p
```

```
return p
```

```
return p
```

```
int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

Before writing assembly, we need to know explicitly where to store values

```
int factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

Need to store ...

- the value of n
- the value of $n - 1$
- the value $\text{factorial}(n-1)$
- the return value: 1 or $n * \text{factorial}(n-1)$

Design decision #1: store values in registers

```
int factorial(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*fact(n-1);  
}
```

Does it work?

- store **n** in **\$t0**
- store **n-1** in **\$t1**
- store **factorial(n-1)** in **\$t2**
- store return value in **\$t3**

No, it doesn't work.

Store **n=3** in \$to

Store **n=2** in \$to,
the stored 3 is
overwritten, lost!

Same problem for
\$t₁, t₂, t₃

- store **n** in **\$to**
- store **n-1** in **\$t₁**
- store **factorial(n-1)** in **\$t₂**
- store return value in **\$t₃**

```
factorial (3)
p = 3 * factorial (2)
```

```
factorial (2)
p = 2 * factorial (1)
```

```
factorial (1)
p = 1 * factorial (0)
```

```
factorial (0)
p = 1 # Base!
return p
```

```
return p
```

```
return p
```

```
return p
```



A register is like a laundry basket -- you put your stuff there, but when you call another function (person), that person will use the **same** basket and take / mess up your stuff.



And yes, the other person will guarantee to use the **same** basket because ...
the other person is **YOU!**
(because recursion)

So the correct design decision is to use **Stack**.

Each recursive call has its own space for storing the values

Stores $n=2$ for factorial (2)

Stores $n=3$ for factorial (3)



Two useful things about stack

1. It has a lot of space
2. Its **LIFO** order (last in first out)
is suitable for implementing
recursions

LIFO order & recursive calls

Note: Everybody is getting the **correct** basket because of LIFO!

```
factorial(2)  
p = 2 * factorial(1)
```

```
factorial(1)  
p = 1 * factorial(0)
```

```
factorial(0)  
p = 1 # Base!  
return p
```

```
return p
```

```
return p
```



Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Design decisions made,
now let's actually write the
assembly code

LIFO order & recursive calls

```
factorial(n=2)
r = factorial(1)
```

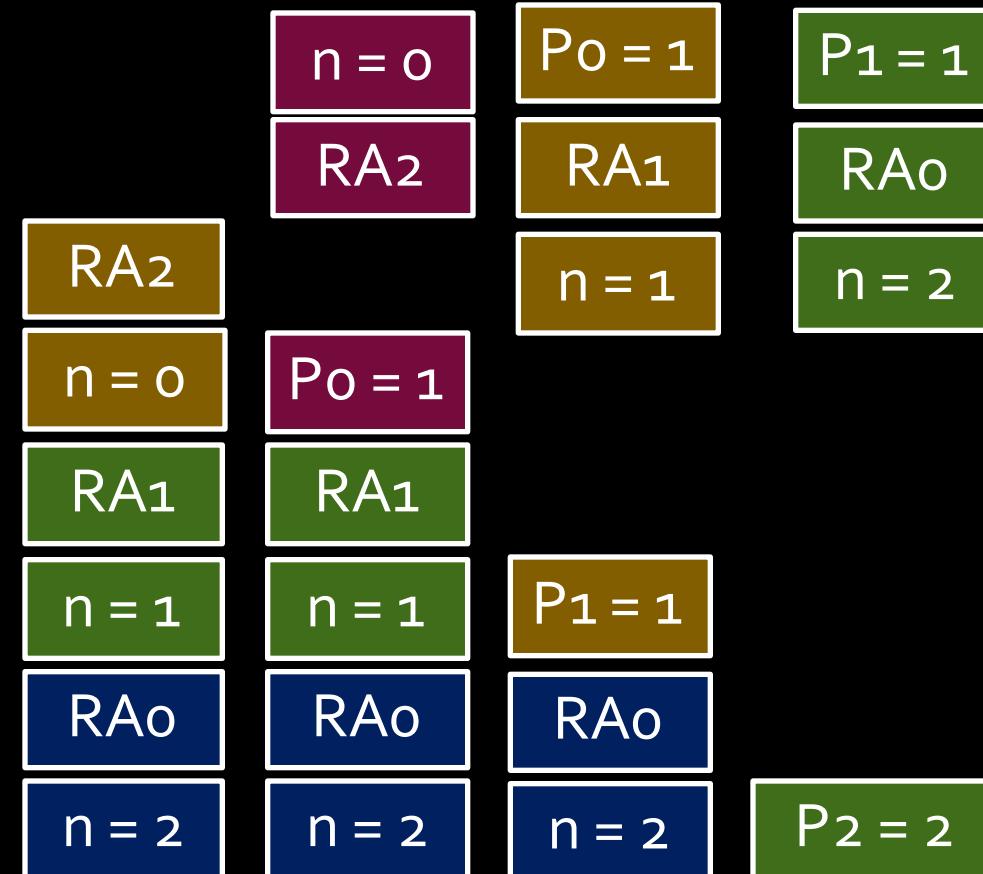
```
factorial(n=1)
r = factorial(0)
```

```
factorial(n=0)
p = 1 # Base!
return p #P0
```

```
p = n * r; # RA2
return p #P1
```

```
p = n * r; # RA1
return p # P2
```

```
int x = 2;
int y = factorial(x)
print(y) # RA0
```



Actions in factorial (n)

Before making the recursive call

- pop argument n
- push argument n-1 (arg for recursive call)
- push return address (remember where to return)
- make the recursive call

After finishing the recursive call

- pop return value from recursive call
- pop return address
- compute return value
- push return value (so the upper call can get it)
- jump to return address

factorial(int n)

$n \rightarrow \$to$
 $n-1 \rightarrow \$t_1$
 $\text{fact}(n-1) \rightarrow \t_2

- Pop n off the stack
 - Store in \$to
- If \$to == 0,
 - Push return value 1 onto stack
 - Return to calling program
- If \$to != 0,
 - Push \$to and \$ra onto stack
 - Calculate n-1
 - Push n-1 onto stack
 - Call factorial
 - ...time passes...
 - Pop the result of factorial(n-1) from stack, store in \$t2
 - Restore \$ra and \$to from stack
 - Multiply factorial(n-1) and n
 - Push result onto stack
 - Return to calling program

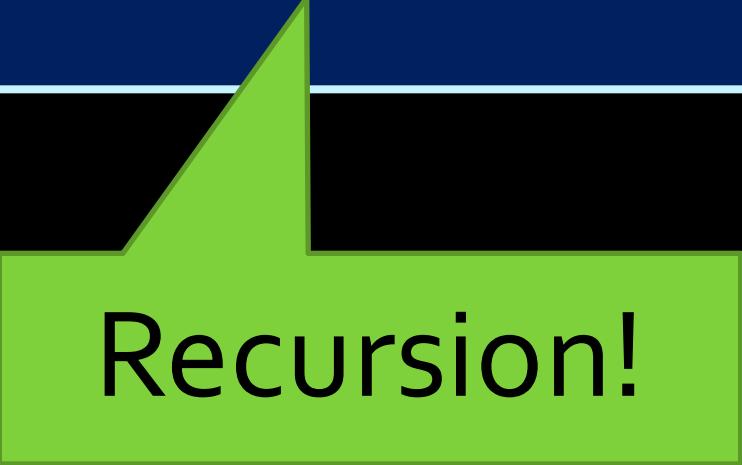
CSC258 Winter 2016

Lecture 12

Announcements

- Missed Lab 10 last Thursday will be made up this Thursday
- If for a valid reason you cannot make it to Thursday, let me know.

```
int factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```



Recursion!

Recursion in Assembly

what recursion really is in hardware



Example: factorial(int n)

- Basic pseudocode for recursive factorial:

- Base Case ($n == 0$)
 - return 1
- Get $\text{factorial}(n-1)$
 - Store result in “product”
- Multiply product by n
 - Store in “result”
- Return result



```
factorial(3)
p = 3 * factorial(2)
```

```
factorial(2)
p = 2 * factorial(1)
```

```
factorial(1)
p = 1*factorial(0)
```

```
factorial (0)
p = 1 # Base!
return p
```

```
return p
```

```
return p
```

```
return p
```

```
int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n*fact(n-1);
}
```

Before writing assembly, we need to know explicitly where to store values

```
int factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

Need to store ...

- the value of n
- the value of $n - 1$
- the value $\text{factorial}(n-1)$
- the return value: 1 or $n * \text{factorial}(n-1)$

Design decision #1: store values in registers

```
int factorial(int n) {  
    if (n==0)  
        return 1;  
    else  
        return n*fact(n-1);  
}
```

Does it work?

- store **n** in **\$t0**
- store **n-1** in **\$t1**
- store **factorial(n-1)** in **\$t2**
- store return value in **\$t3**

No, it doesn't work.

Store **n=3** in \$to

Store **n=2** in \$to,
the stored 3 is
overwritten, lost!

Same problem for
\$t1, t2, t3

- store **n** in **\$to**
- store **n-1** in **\$t1**
- store **factorial(n-1)** in **\$t2**
- store return value in **\$t3**

```
factorial (3)
p = 3 * factorial (2)
```

```
factorial (2)
p = 2 * factorial (1)
```

```
factorial (1)
p = 1*factorial (0)
```

```
factorial (0)
p = 1 # Base!
return p
```

```
return p
```

```
return p
```

```
return p
```



A register is like a laundry basket -- you put your stuff there, but when you call another function (person), that person will use the **same** basket and take / mess up your stuff.



And yes, the other person will guarantee to use the **same** basket because ...
the other person is **YOU!**
(because recursion)

So the correct design decision is to use **Stack**.

Each recursive call has its own space for storing the values

Stores $n=2$ for factorial (2)

Stores $n=3$ for factorial (3)



Two useful things about stack

1. It has a lot of space
2. Its **LIFO** order (last in first out)
is suitable for implementing
recursions

LIFO order & recursive calls

Note: Everybody is getting the **correct** basket because of LIFO!

```
factorial(2)
p = 2 * factorial(1)
```

```
    factorial(1)
    p = 1 * factorial(0)
```

```
        factorial(0)
        p = 1 # Base!
        return p
```

```
    return p
```

```
return p
```



Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Design decisions made,
now let's actually write the
assembly code

LIFO order & recursive calls

```
factorial(n=2)
r = factorial(1)
```

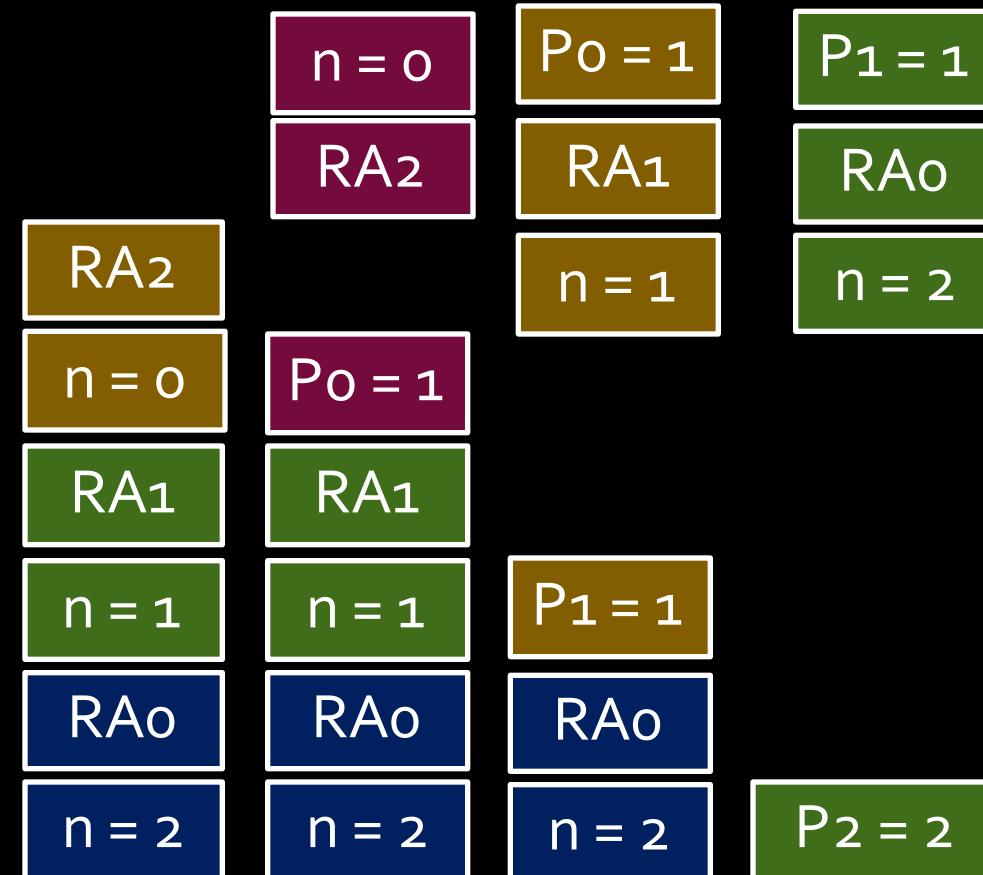
```
factorial(n=1)
r = factorial(0)
```

```
factorial(n=0)
p = 1 # Base!
return p #P0
```

```
p = n * r; # RA2
return p #P1
```

```
p = n * r; # RA1
return p # P2
```

```
int x = 2;
int y = factorial(x)
print(y) # RA0
```



Actions in factorial (n)

Before making the recursive call

- pop argument n
- push argument n-1 (arg for recursive call)
- push return address (remember where to return)
- make the recursive call

After finishing the recursive call

- pop return value from recursive call
- pop return address
- compute return value
- push return value (so the upper call can get it)
- jump to return address

factorial(int n)

$n \rightarrow \$to$
 $n-1 \rightarrow \$t_1$
 $\text{fact}(n-1) \rightarrow \t_2

- Pop n off the stack
 - Store in \$to
- If \$to == 0,
 - Push return value 1 onto stack
 - Return to calling program
- If \$to != 0,
 - Push \$to and \$ra onto stack
 - Calculate n-1
 - Push n-1 onto stack
 - Call factorial
 - ...time passes...
 - Pop the result of factorial(n-1) from stack, store in \$t2
 - Restore \$ra and \$to from stack
 - Multiply factorial(n-1) and n
 - Push result onto stack
 - Return to calling program

factorial(int n)

fact:

```
lw $t0, 0($sp)
addi $sp, $sp, 4
bne $t0, $zero, not_base
addi $t0, $zero, 1
addi $sp, $sp, -4
sw $t0, 0($sp)
jr $ra
```

not_base:

```
addi $sp, $sp, -4
sw $t0, 0($sp)
addi $sp, $sp, -4
sw $ra, 0($sp)
addi $t1, $t0, -1
addi $sp, $sp, -4
sw $t1, 0($sp)
jal fact
```

$n \rightarrow \$to$

$n-1 \rightarrow \$t1$

$\text{fact}(n-1) \rightarrow \$t2$

- Pop n off the stack
 - Store in $\$to$
- If $\$to == 0$,
 - Push return value 1 onto stack
 - Return to calling program
- If $\$to != 0$,
 - Push $\$to$ and $\$ra$ onto stack
 - Calculate $n-1$
 - Push $n-1$ onto stack
 - Call factorial
 - Pop the result of factorial ($n-1$) from stack, store in $\$t2$
 - Restore $\$ra$ and $\$to$ from stack
 - Multiply factorial ($n-1$) and n
 - Push result onto stack
 - Return to calling program

factorial(int n)

```
lw $t2, 0($sp)
addi $sp, $sp, 4
lw $ra, 0($sp)
addi $sp, $sp, 4
lw $t0, 0($sp)
addi $sp, $sp, 4
mult $t0, $t2
mflo $t3
addi $sp, $sp, -4
sw $t3, 0($sp)
jr $ra
```

$n \rightarrow \$to$
 $n-1 \rightarrow \$t_1$
 $\text{fact}(n-1) \rightarrow \t_2

- Pop n off the stack
 - Store in $\$to$
- If $\$to == 0$,
 - Push return value 1 onto stack
 - Return to calling program
- If $\$to != 0$,
 - Push $\$to$ and $\$ra$ onto stack
 - Calculate $n-1$
 - Push $n-1$ onto stack
 - Call factorial
 - Pop the result of factorial ($n-1$) from stack, store in $\$t_2$
 - Restore $\$ra$ and $\$to$ from stack
 - Multiply factorial ($n-1$) and n
 - Push result onto stack
 - Return to calling program

Recursive programs

- How do we handle recursive programs?
 - Still needs base case and recursive step, as with other languages.
 - Main difference: Maintaining register values.
 - When a recursive function calls itself in assembly, it calls `jal` back to the beginning of the program.
 - Between different level of recursive calls, things are passed through the **stack**
 - Function arguments, return addresses, return values

```
int factorial (int x) {  
    if (x==0)  
        return 1;  
    else  
        return x*fact (x-1);  
}
```

Recursive programs

- Use of stack
 - Before recursive call, store the register values that you use onto the stack, and restore them when you come back to that point.
 - Store \$ra as one of those values, to remember where each recursive call should return.

```
int factorial (int x) {  
    if (x==0)  
        return 1;  
    else  
        return x*fact (x-1);  
}
```

There is no recursion in hardware

- It's just a linear sequence of assembly instructions, where you jump to the beginning of the program over and over again.
- While sensibly store and retrieve remembered values from the stack

Translated recursive program (part 1)

```
main:      addi    $t0, $zero, 10      # call fact(10)
           addi    $sp, $sp, -4       # by putting 10
           sw     $t0, 0($sp)        # onto stack
           jal    factorial         # result will be
           ...                         # on the stack

factorial:   lw     $a0, 4($sp)      # get x from stack
              bne   $a0, $zero, rec      # base case?
base:       addi   $t0, $zero, 1       # put return value
           sw     $t0, 4($sp)        # onto stack
           jr    $ra                # return to caller
rec:        addi   $sp, $sp, -4      # store return
           sw     $ra, 0($sp)        # addr on stack
           addi   $a0, $a0, -1       # x--
           addi   $sp, $sp, -4       # push x on stack
           sw     $a0, 4($sp)        # for rec call
           jal    factorial         # recursive call
```

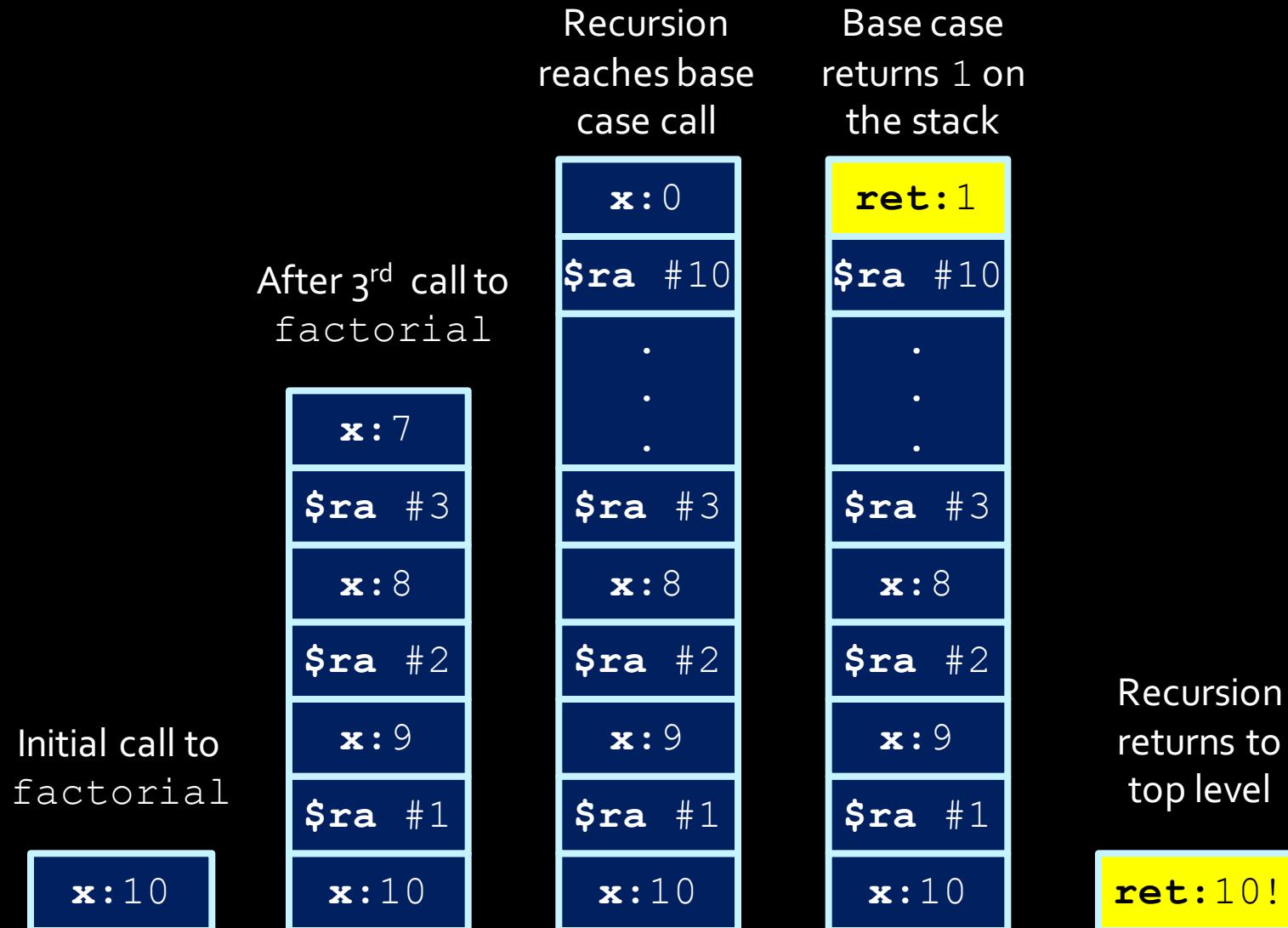
Translated recursive program (part 2)

(continued from part 1)

```
lw    $v0, 0($sp)          # get return value
addi $sp, $sp, 4           #   from stack
lw    $ra, 0($sp)          # restore return
addi $sp, $sp, 4           #   address value
lw    $a0, 0($sp)          # restore x value
addi $sp, $sp, 4           #   for this call
mult $a0, $v0               # x*fact(x-1)
mflo $t0                   # fetch product
addi $sp, $sp, -4          # push product
sw    $t0, 0($sp)          #   onto stack
jr    $ra                   # return to caller
```

- Note: `jal` always stores the next address location into `$ra`, and `jr` returns to that address.

Factorial stack view



You can't recur too much

The stack is NOT of infinite size, so there is always a **limit** on the number of recursive calls that you can make.

When exceeds that limit, you get a **stack overflow**, all content of the stack will be dumped.

```
state = deepcopy(state, memo)
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 238, in _deepcopy_dict
    y[deepcopy(key, memo)] = deepcopy(value, memo)
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 211, in _deepcopy_list
    y.append(deepcopy(a, memo))
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 211, in _deepcopy_list
    y.append(deepcopy(a, memo))
File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 143, in deepcopy
    cls = type(x)
RuntimeError: maximum recursion depth exceeded while calling a Python object
Mac-Pro-van-Mathias:Desktop Mathias$
```



Interrupts and Exception

A note on interrupts

- **Interrupts** take place when an external event requires a change in execution.
 - Example: arithmetic overflow, system calls (`syscall`), `Ctrl-C`, undefined instructions.
 - Usually signaled by an external input wire, which is checked at the end of each instruction.
 - High priority, override other actions



A note on interrupts

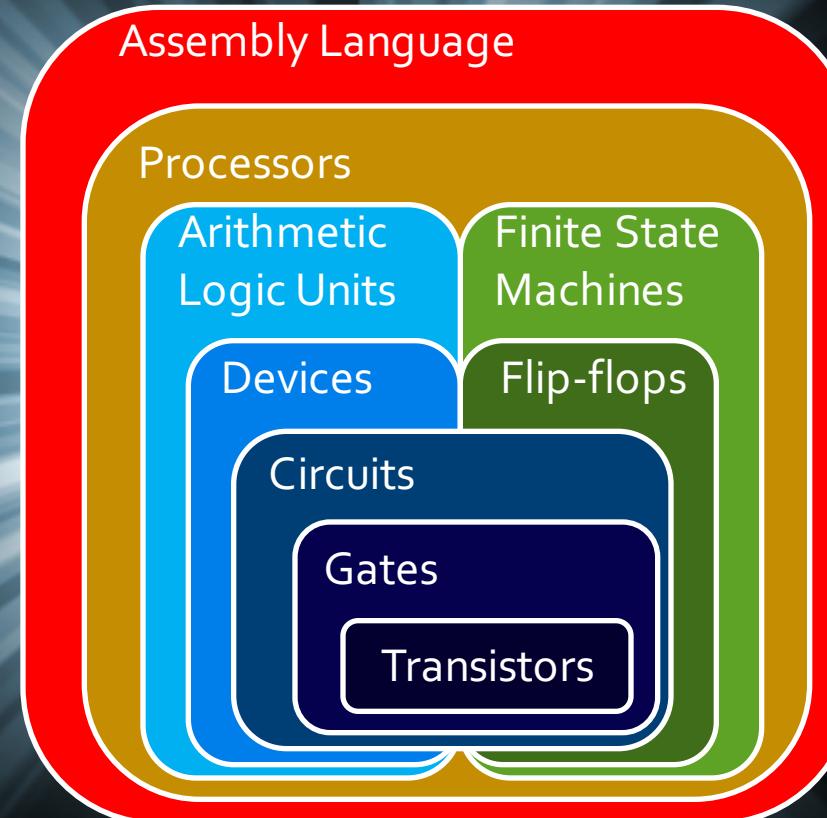
- Interrupts can be handled in two general ways:
 - **Polled handling:** The processor branches to the address of interrupt handling code (interruption handler), which begins a sequence of instructions that check the cause of the exception, i.e., need to ask around to figure out what type of exception.
→ This is what MIPS uses.
 - **Vectored handling:** The processor can branch to a different address for each type of exception. Each exception address is separated by only one word. A jump instruction is placed at each of these addresses for the handler code for that exception. So no need to ask around.

Interrupt handling

- In the case of polled interrupt handling, the processor jumps to exception handler code, based on the value in the **cause register** (see table).
 - If the original program can resume afterwards, this interrupt handler returns to program by calling `rfe` instruction.
 - Otherwise, the stack contents are dumped and execution will continue elsewhere.

0 (INT)	external interrupt.
4 (ADDRL)	address error exception (load or fetch)
5 (ADDRS)	address error exception (store).
6 (IBUS)	bus error on instruction fetch.
7 (DBUS)	bus error on data fetch
8 (Syscall)	Syscall exception
9 (BKPT)	Breakpoint exception
10 (RI)	Reserved Instruction exception
12 (OVF)	Arithmetic overflow exception

We are done!





Given enough silicon,
phosphorus and
boron, you are now
able to build a
computer!

Final Exam Review

Time & Location

Time: April 11, 9am-12pm

Location: IB-110

No aid.

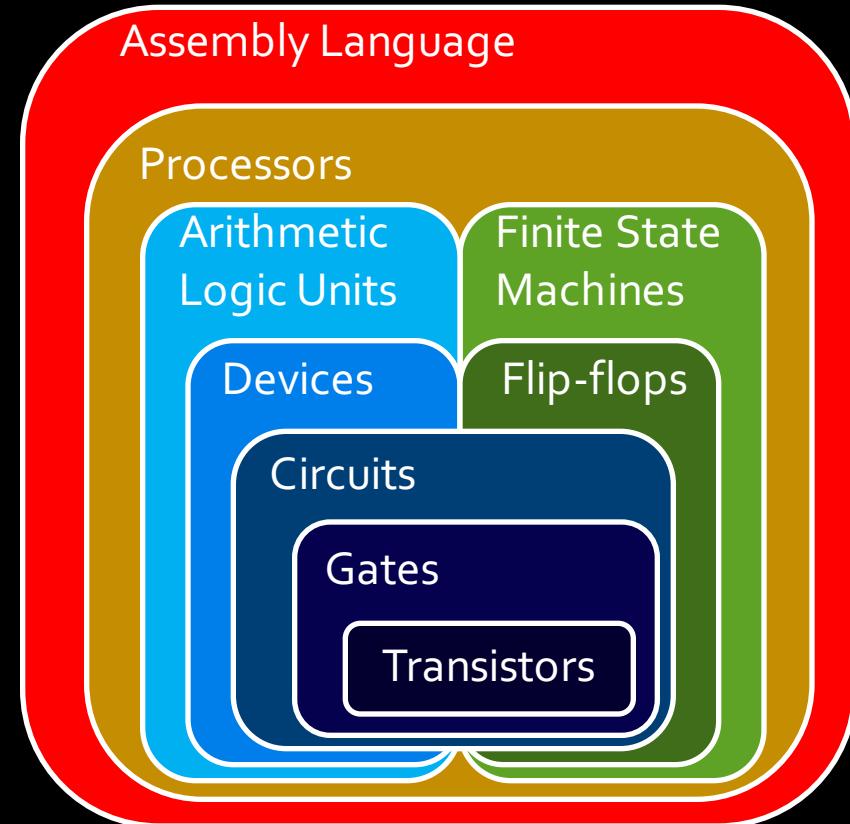
Bring your student card

Pre-exam office hours

- March 28~April 1
 - Monday, Tuesday, Friday
 - 4-6pm
- April 4~April 8
 - Monday, Tuesday, Thursday, Friday
 - starting 3pm

Coverage

- Everything
- Slightly more weight on the second half (after midterm)
- No Verilog questions
- No Booth algorithm



Types of questions

- Short answer: conceptual questions
- Read combination circuit
- Design combinational circuit
- Read sequential circuit, draw waveform
- Design sequential circuit, FSM
- Set Datapath signals
- Write assembly code according to requirement
- Read assembly code, see what it does
- Translate between assembly and machine code
- Translate between assembly and C code.

How to study for final exam

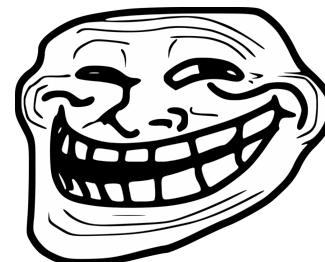
1. Review lecture / tutorial slides
2. Review what you did for labs
3. Review quizzes
4. Practice with past exams
 - One with solution posted on course page
 - More in old exam repository
5. Whenever confused, ask on Piazza, or come to **office hours**

Important Lessons

- Pay attention to **details**
 - Don't just skim over the slides and feel "got the idea".
- Understand very well what you did in the **labs**
- Understand very well the **quiz** questions

Sample questions

1. Who messed up Agent Smith's laundry basket when only one basket is used? (2 marks)
 2. Name the two students who demoed in class the "human flip flop". (2 marks)



Quiz Bonus

3 bonus marks if you get ≥ 18 points

2 bonus marks if you get ≥ 10 points

1 bonus mark if you get ≥ 1 point

Name	Points
Chris C.	29
Elijah M.	29
Alexei F.	23
Jay G.	23
Shayan G.	22
Brandon A. M.	21
Eric C.	21
Gang Z.	21
James T.	21
Joseph C.	21
Raj S.	21
Alexander K.	20
Frank Y.	20
Jailani D.	20
Michael Z.	20
Ramy E.	20

See you in office hours!