211 Review: Storing Data in Memory

Byte Address	Value
0	0b00000101 (5)
1	0b00000100 (4)
2	0b11111110 (-2)

How much data can be stored in 1 mem. address?
→ 1 byte ; aka 8 bits
→ The top row is "byte address 0"

What if you want to store an integer?
→ An int is size = 4 bytes, so it will span multiple addresses in memory!
→ 4 bytes → 4 addresses in memory.
→ Ex: 329,421 = 0b0000 0000 0000 0101 0000 0110 1100 1101

What is the Most Significant Byte (MSB)?
→ The leftmost byte of a number, e.g. 0000 0000

What is the Least Significant Byte? (LSB)
→ The rightmost byte, e.g. 1100 1101

What is Little Endian Ordering?
→ Where the LSB of an int goes in the first byte address
→ AKA, fill the table from "right to left";
- the last 2 nibbles of num → 1st address
- nibbles 5 & 6 → 2nd address
- nibbles 1 & 2 → 4th address

Little Endian

Byte Address	Value
0	1100 1101
1	0000 0110
2	0000 0101
3	0000 0000

What is Big Endian Ordering?
→ Where the LSB of an int goes in the largest byte address

Big Endian

Byte Address	Value
0	0000 0000
1	0000 0101
2	0000 0110
3	1100 1101

Which do we use in this class?
→ Little Endian

How do you compute the address of an element in an array?
→ (Index[element] × sizeof(element type)) + base_address
→ Ex: An int array with 10 elements
- Each element requires 4 bytes/addresses
- element 0 → address 0 × 4 = addr 0 (or whatever the base addr. is)
- element 8 → address 8 × 4 = addr 32

Diagrams to visualize arrays?
→ Ex: arr = [5, 90, 100, 40]

1 row = 1 byte

Addr.	Value	Data
0	5	0000 0101
1		0000 0000
2		0000 0000
3		0000 0000
4	90	0101 1010
5		0000 0000
6		0000 0000
7		0000 0000
8	100	0110 0100
9		0000 0000
10		0000 0000
11		0000 0000
12	40	0010 1000
13		0000 0000
14		0000 0000
15		0000 0000

1 row = 4 bytes

Addr.	Value	Data
0x00000000	5	0000 0000 0000 0000 0000 0000 0000 0101
0x00000004	90	0000 0000 0000 0000 0000 0000 0101 1010
0x00000008	100	0000 0000 0000 0000 0000 0000 0110 0100
0x0000000C	40	0000 0000 0000 0000 0000 0000 0010 1000
0x00000010		
0x00000014		
0x00000018		
0x0000001C		
0x00000020		

How do we know how many **bits** are needed to address our memory?

→ It depends on the size of our total memory!

→ bits_per_addr = $\log_2$(size of memory in bytes)

→ Ex:

- Memory = 16 bytes ... each address is $\log_2(16) = \underline{4}$ bits (eg, addr 1 = 0010)
- Memory = 4 GB...
  - 1 GB = 2^{30} bytes
  - each address is $\log_2(4 \cdot 2^{30}) = \underline{32}$ bits
  - eg, addr 1 is 0000 0000 0000 0000 0000 0000 0000 0010.

Digital Logic

What is a logic gate?

→ A device that performs a boolean function to produce a single binary output.

→ Acts as a building block for digital circuits.

→ Several types of logic gates, each of which represent a single logical operator.

What is an "Inverter" or "Not" Gate?

→ Represents NOT logical operator

→ Can be represented by a symbol, truth table, or equation:

Symbol: A —▷o— Y

Equation: $Y = \overline{A}$

Truth Table:

A	Y
0	1
1	0

Gate — Symbol — Equation — Truth Table

What is an AND Gate?

Equation: $Y = AB$ or $Y = A * B$

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

What is an OR Gate?

Equation: $Y = A + B$

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

What is an XOR Gate?

Equation: $Y = A \oplus B$

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

→ Y is true when either A OR B is true, but not both. "XOR" = exclusive OR

What is a NAND Gate?

(the circle indicates negation)

Equation: $Y = \overline{A * B}$ or $Y = \overline{AB}$

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

→ Y is true when (A and B) is false ... just negate the result of AB.

What is a NOR Gate?

Equation: $Y = \overline{A + B}$

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

→ Negate the result of (A OR B)

What is a XNOR Gate?

Equation: $Y = \overline{A \oplus B}$

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

→ Negate the result of (A XOR B)

Example: How do you form an equation from a logic diagram?

Logic Diagram

A 0, B 0, C 0, D 0, E 0 → Y 0

Equation

$$(\overline{AB} \times \overline{C+D}) + \overline{E} = (\overline{AB})(\overline{C+D}) + \overline{E}$$

How do you do the reverse?

→ Equation: $Y = \overline{\overline{(A+B)}\,CD}$

How do you form an equation from a truth table?

→ Go through each line of the table where Y = 1 & create an equation that describes it using AND statements.

→ Then, your final equation is the **addition** (aka OR) of all those equations

- "sum of products" form

Example?

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

Row 1 → $\bar{A}\bar{B}\bar{C} = Y$

Row 3 → $\bar{A}B\bar{C} = Y$

Row 6 → $A\bar{B}C = Y$

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

Transistors

- Basic Circuits -

What is voltage?	→ The force that makes currents flow !
	→ Measured in Volts (V)
What is current ?	→ The rate of flow of electrons.
	→ Measured in Amperes (A)
Analogy to understand voltage & current?	→ A Dam with water @ the top and a reservoir at the bottom.
	→ Voltage ≈ The desire of the water to flow downhill.
	• NOT the actual movement of the water. Voltage itself is static.
	→ Current ≈ The actual flow of the water downstream.
	• The size of the opening affects the amount of water (current) that can flow.
	• The wider the "opening", the higher the current
What is a closed circuit?	→ A circuit that is fully connected & allows electricity to flow uninterrupted.
	→ When a switch (and circuit) is closed, we know the voltage on both ends of the circuit.
	5V —o—o— 5V "closed" = ON
What is an open circuit?	→ A circuit that contains a broken connection.
	→ Electricity stops flowing @ point where connection was lost.
	→ When a switch is open, we don't necessarily know the voltage at the end of the circuit — We'd need to see the full circuit.
	5V —o/ o— ??? X "open" = OFF
What are switches? What is the symbol?	→ Controlled by physical contact. SYMBOL:
	→ •—o/ o—• or •—o—o—•

-Transistors -

What are transistors?	→ Like switches, but controlled by a voltage, rather than physical contact.
What "states" do transistors operate in?	→ Just 2 states - ON or OFF ... its binary.
	→ This is why all machines communicate in binary (1s and 0s)!
	ON —o—o— OFF —o/ o—
Wait... I thought you said transistors are controlled by voltage?	→ They are, but we can use binary terms rather than exact voltage levels for discussing them.
	→ high voltage = "logic high" = logic 1 = 1
	→ low voltage = "logic low" = logic 0 = 0

Question	Answer
What is the symbol for transistors?	→ The terminal labeled with the blue dot determines whether current can flow between the terminals labeled with the red dot
What are the terminals in an **NMOS transistor**?	1. Gate : controls whether transistor is **on** or **off** 2. Source : endpoint 3. Drain : endpoint V_D (drain voltage), V_G (gate voltage), V_S (source voltage)
What does it mean if the **gate voltage is high**?	→ a.k.a. "logic 1" → current **can flow** between the source and the drain... they will be connected & will have the same voltage.
Example?	→ e.g., if V_S = logic 0, V_D also = logic 0. And vice versa. V_D (1), V_G (1), V_S (1) **OR** V_D (0), V_G (1), V_S (0)
What does it mean if the **gate voltage is low**?	→ aka "logic 0" → source & drain are not connected; current cannot flow. → The voltage of the drain will be unknown. V_D – UNKNOWN, V_G (0), V_S (1) **OR** V_D – UNKNOWN, V_G (0), V_S (0)
What is a **PMOS Transistor**?	→ Same 3 terminals as with NMOS transistor. → **The gate voltages are reversed**: • Gate voltage LOW (logic=0) = current can flow • Gate voltage HIGH (logic=1) = current cannot flow
How do we define the behavior of a transistor?	**nMOS** • behaves as an **open switch** when V_G is low. • behaves as a **closed switch** when V_G is high. **pMOS** • behaves as an **open switch** when V_G is high. • behaves as a **closed switch** when V_G is low.

What is Moore's Law?

→ A trend describing the reduction in **transistor size** in machines/computers

→ Moore's Law: The # of transistors in a given area on a chip doubles every 2 years

- ~2,000 transistors in 1970 → > 10 billion today!

→ The trend is coming to an end now, as the industry develops alternative technologies to ntinue improving chip performance.

Building Logic Gates with Transistors & CMOS

Question	Notes
RECALL: EX of a logic gate?	→ Inverter gate: A —▷o— Y ($Y = \bar{A}$) → See pg 3 of notes.
What is power?	→ A component (of a circuit?) that produces a logic high (1) value → Symbol: ⊤ → Always connected to output via a pmos transistor. • RECALL: pmos transistor is closed/ON when input = 0
What is ground?	→ A component that produces a logic low (0) value. → Symbol: ↓ → Always connected to output via an nmos transistor. • RECALL: nmos transistor is closed/ON when input = 1
Visual example of an Inverter with switches?	→ RECALL: switches are a way to physically control the opening & closing of circuits, while transistors operate on logic values.

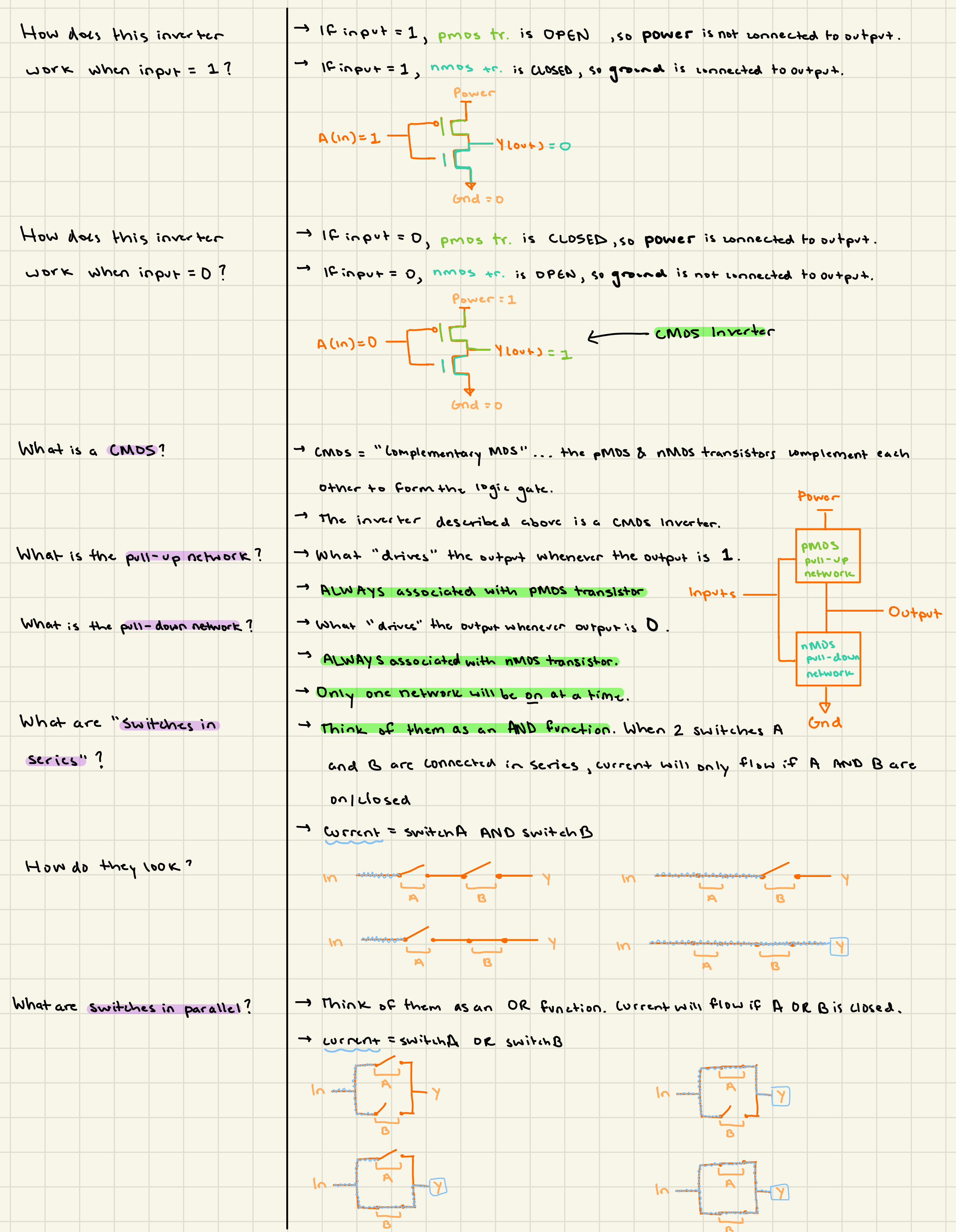
Question	Notes
How does this inverter work when input = 0?	→ When input = 0, we manually close the top/power switch. → When input = 1, we manually close the bottom/ground switch.

→ Since power produces logic = 1 & power is connected to output, output = 1

Question	Notes
How does it work when input = 1?	

→ Since ground produces logic = 0 & ground is connected to output, output = 0.

Question	Notes
How would we build this inverter with transistors?	→ Replace the power switch with a pmos transistor. Replace ground switch w/ an nmos transistor. → Then, invert the output of the power transistor (with the o symbol)



How does this inverter work when input = 1?

→ If input = 1, pmos tr. is OPEN, so power is not connected to output.

→ If input = 1, nmos tr. is CLOSED, so ground is connected to output.

How does this inverter work when input = 0?

→ If input = 0, pmos tr. is CLOSED, so power is connected to output.

→ If input = 0, nmos tr. is OPEN, so ground is not connected to output.

What is a CMOS?

→ CMOS = "Complementary MOS" ... the pMOS & nMOS transistors complement each other to form the logic gate.

→ The inverter described above is a CMOS Inverter.

What is the pull-up network?

→ What "drives" the output whenever the output is 1.

→ ALWAYS associated with pMOS transistor

What is the pull-down network?

→ What "drives" the output whenever output is 0.

→ ALWAYS associated with nMOS transistor.

→ Only one network will be on at a time.

What are "switches in series"?

→ Think of them as an AND function. When 2 switches A and B are connected in series, current will only flow if A AND B are on/closed

→ current = switch A AND switch B

How do they look?

What are switches in parallel?

→ Think of them as an OR function. Current will flow if A OR B is closed.

→ current = switch A OR switch B

How do we build a CMOS for the NAND logic gate?

→ RECALL: NAND $Y = \overline{A \times B}$ / $Y = \overline{AB}$

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

→ When AB is true, we want output = 0, so we can build "switches in series" (aka A AND B) and connect them to GROUND, since GROUND always sends output of 0:

A
B
GND
Y

- This part of the CMOS will result in something (namely, logic = 0) being sent to Y whenever (AB) is true. A = 1, B = 1 ⇒ Ground = 1 ⇒ Y = 0

What do we build for when A and B aren't both logic = 1?

→ When A = 0 OR B = 0, we want output = 1. So we can build "switches in parallel" (aka A OR B), but invert the values being sent to the A & B transistors.

- if A = 0 and B = 0, send 1 and 1 to the 2 switches. Since at least one is 1, output of 1 is received by power. A = 0, B = 0 ⇒ Power = 1 ⇒ Y = 1
- if A = 0 and B = 1, send 1 and 0 to the 2 switches. Since at least one is 1, output of 1 is received by power. A = 0, B = 1 ⇒ Power = 1 ⇒ Y = 1

Vdd
A
B
Y

How do we combine these 2 pieces to form the CMOS NAND?

Vdd
A
B
Y
GND
Fig 1

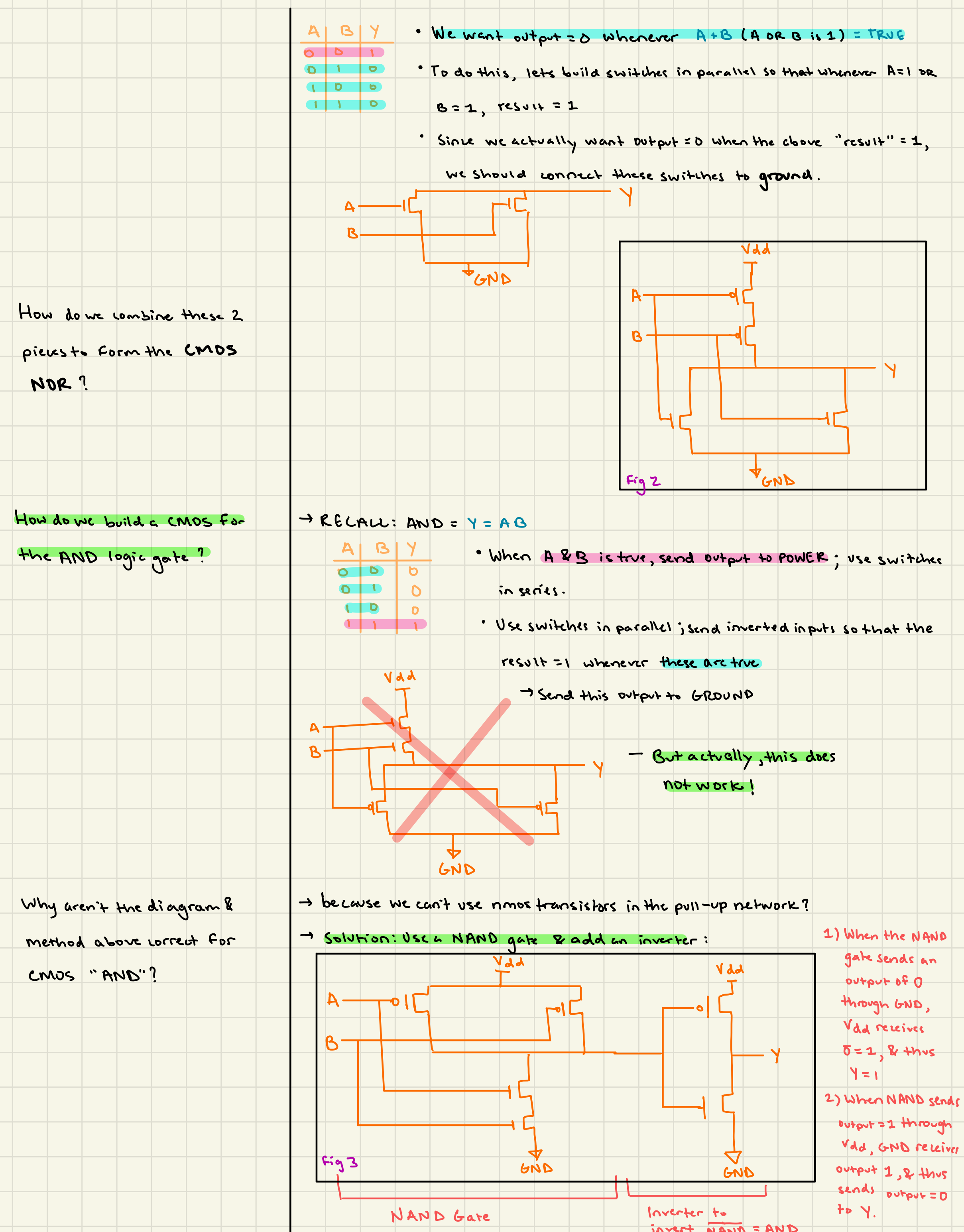
How do we build a CMOS for the NOR logic gate?

→ RECALL: NOR $Y = \overline{A + B}$

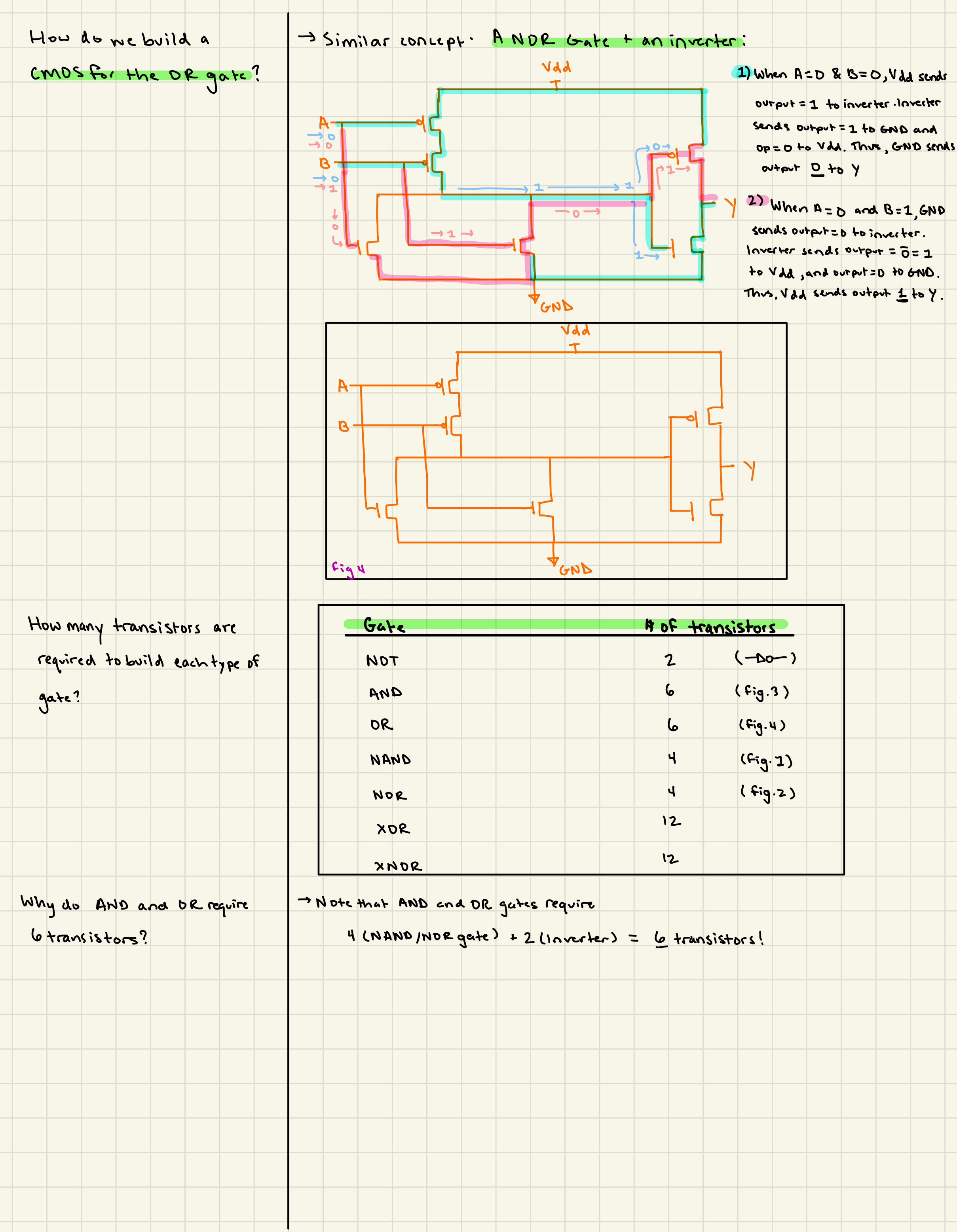
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

- Lets build "switches in series" & invert the inputs so that result = 1 if and only if A = 0 and B = 0
- Since we actually want the output to be 1 (output = result) when A = 0 and B = 0, we should connect these switches to power.

Vdd
A
B
Y



How do we combine these 2 pieces to form the CMOS NOR ?
A | B | Y
0 | 0 | 1
0 | 1 | 0
1 | 0 | 0
1 | 1 | 0
• We want output = 0 whenever A+B (A OR B is 1) = TRUE
• To do this, lets build switches in parallel so that whenever A=1 or B=1, result = 1
• Since we actually want output = 0 when the above "result" = 1, we should connect these switches to ground.
A
B
Y
GND
Vdd
A
B
Y
GND
Fig 2
How do we build a CMOS for the AND logic gate ?
→ RECALL: AND = Y = AB
A | B | Y
0 | 0 | 0
0 | 1 | 0
1 | 0 | 0
1 | 1 | 1
• When A & B is true, send output to POWER ; use switches in series.
• Use switches in parallel ; send inverted inputs so that the result = 1 whenever these are true
→ Send this output to GROUND
Vdd
A
B
Y
GND
— But actually, this does not work !
Why aren't the diagram & method above correct for CMOS "AND"?
→ because we can't use nmos transistors in the pull-up network?
→ Solution: Use a NAND gate & add an inverter :
Vdd
A
B
GND
Vdd
Y
GND
Fig 3
NAND Gate
Inverter to invert NAND = AND
1) When the NAND gate sends an output of 0 through GND, Vdd receives 0 = 1, & thus Y = 1
2) When NAND sends output = 1 through Vdd, GND receives output 1, & thus sends output = 0 to Y.



How do we build a CMOS for the OR gate?

→ Similar concept: A NOR Gate + an inverter:

1) When A=0 & B=0, Vdd sends output = 1 to inverter. Inverter sends output = 1 to GND and op = 0 to Vdd. Thus, GND sends output 0 to Y

2) When A=0 and B=1, GND sends output=0 to inverter. Inverter sends output = $\bar{0}$ = 1 to Vdd, and output=0 to GND. Thus, Vdd sends output 1 to Y.

How many transistors are required to build each type of gate?

Gate	# of transistors	
NOT	2	(—▷o—)
AND	6	(fig.3)
OR	6	(fig.4)
NAND	4	(fig.1)
NOR	4	(fig.2)
XOR	12	
XNOR	12	

Why do AND and OR require 6 transistors?

→ Note that AND and OR gates require

4 (NAND/NOR gate) + 2 (Inverter) = 6 transistors!

Digital Logic Pt. 2

Why do we want to minimize transistor count when building logic gates?

→ Smaller amt. of transistors **Reduces**:
- Delay from input to output
- The area of the circuit taken up; this is good b/c we can pack more logic (i.e. transistors) in a given space.
- Power consumption
- The cost of the circuit

What is a logically equivalent circuit to an inverter?

→ The "bubble" on an AND or OR symbol:

OR + NOT → Y: 6 (OR) + 2 (NOT) = 8 transistors

is equivalent to

NOR → Y: 4 (NOR) transistors

But obviously, one uses much less transistors than the other!

What is a logically equivalent circuit for NAND?

NAND (OR with inverted inputs A, B) → Y is equivalent to (AND with bubble output) → This is the better, conventional way to draw NAND

What about NOR?

NOR (AND with inverted inputs A, B) → Y is equivalent to (OR with bubble output) → This is the better, conventional way to draw NOR

How do we minimize transistor count on logic gate diagrams?

1. Replace AND and OR gates with NAND gates, or NOR gates.
2. Add bubbles to make the circuit logically equivalent.
3. Cancel out the free bubbles
4. Draw the NAND and NOR gates in their conventional form.

Example?

(Circuit: inputs A, B into AND; C into NOT; outputs into OR) — labels: AND, OR

1) NAND

2) Added inverter bubbles to make the circuit equivalent after replacing the AND and OR with NANDs

3) these 2 cancel out; These 2 cancel out

4) Replace (OR with inverted inputs) with (NAND)

- Gates with more than 2 inputs -

What does the symbol for a 3-input AND gate look like?

A, B, C → Y OR (logical equivalent) A, B → · C → Y

Symbol for 3-input OR gate?

A, B, C → Y OR (logical equivalent) A, B → · C → Y

Symbol for 3-input NAND gate?

A, B, C → Y OR (logical equivalent) A, B → · C → Y

Symbol for 3-input NOR gate?

A, B, C → Y OR (logical equivalent) A, B → · C → Y

Symbol for 3-input XOR gate?

A, B, C → Y OR (logical equivalent) A, B → · C → Y

Symbol for 3-input XNOR gate?

A, B, C → Y OR (logical equivalent) A, B → · C → Y

The Laws of Boolean Algebra

What do we use boolean algebra for?

→ To create logically equivalent circuits that use fewer transistors ← (MOTIVATION: Recall notes on Digital Logic pt. 2)

What are boolean algebra laws?

→ Laws about boolean statements where:

- Letters (A, B, C, etc.) represent literals
- "+" = OR
- "·" (aka multiplication) = AND

→ Every Boolean alg law comes in a pair of statements: the AND version and the OR version.

What is the Identity Law?

→ $A + 0 = A$ (Any literal A or'd with 0 will always output A).

→ $A \cdot 1 = A$ (Any literal A and'd with 1 will always output A).

What is the Null Law?

→ $A + 1 = 1$ (A OR 1 always = 1)

→ $A \cdot 0 = 0$ (A AND 0 always = 0)

What is the Idempotent Law?

→ $A + A = A$ (A OR A = A)

→ $A \cdot A = A$ (A AND A = A)

What is the complement Law?

→ $A + \bar{A} = 1$ (A OR NOT A = 1)

→ $A \cdot \bar{A} = 0$ (A AND NOT A = 0)

What is an example of optimizing a circuit?

→ The equation $Y = AA + BC$ produces this circuit:

- it has 6 + 6 + 6 = 18 transistors

A, A, B, C, Y, 6, 6, 6

→ Using the idempotent law $AA = A$, we can reduce the equation to $Y = A + BC$, producing the circuit

- it has 6 + 6 = 12 transistors

A, B, C, Y, 6, 6

→ By converting the gates into NAND gates (RECALL: Digital Logic 2), we can further reduce the circuit to 10 transistors:

A, B, C, Y, 2, 4, 4

What is the Commutative Law?

→ $A + B = B + A$

→ $A \cdot B = B \cdot A$

What is the Associative Law?

→ $(A + B) + C = A + (B + C)$

→ $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

What is the Distributive Law?

→ $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

↳ (A AND (B OR C)) = ((A AND B) OR (A AND C))

→ $A + (B \cdot C) = (A + B) \cdot (A + C)$

↳ (A OR (B AND C)) = ((A OR B) AND (A OR C))

Proof for the distributive law (OR version)?	→ Lets use the other laws to prove the truth of $A + (B \cdot C) = (A+B) \cdot (A+C)$ 1. Distribute out the terms: $A + (BC) = AA + AC + AB + BC$ $= A + AC + AB + BC$ $= A(1 + C + B) + BC$ $= A(1) + BC$ $= A + BC$
What is De Morgan's Theorem?	→ Used to simplify large NOT bars (e.g. $\overline{A+B}$) → $\overline{A+B} = \bar{A} \cdot \bar{B}$ (NOT (A OR B) = NOT(A) AND NOT(B)) → $\overline{AB} = \bar{A} + \bar{B}$ (NOT (A AND B) = NOT(A) OR NOT(B)) → This theorem is actually how we created the "conventional" NAND gate: A, B → Y, $Y = \bar{A} + \bar{B}$ is equivelent to A, B → Y, $Y = \overline{AB}$ → DeMorgan's: flipping the inputs ($\overline{AB}$ → AB), operations (AB → A+B), and the outputs (A+B → $\bar{A} + \bar{B}$)... Thus $\overline{AB}$ → $\bar{A} + \bar{B}$
When would we need to use these laws for simplification?	→ When converting a truth table into a boolean equation! 1. Create an equation in "sum of products" form (RECALL) 2. Use the laws to simplify!

Karnaugh Maps (K Maps)

What are K Maps?

→ A tool used to simplify boolean expressions

- An alternate tool to using boolean algebra laws.

→ K-Maps provide a graphical method to find simplified boolean expressions.

→ Benefit: If used properly, K Maps always derive the most simplified equation!

Unlike boolean alg, where you aren't always 100% sure if you've maximized simplification

How do K Maps work?

→ A diagram / grid that makes it easy to identify combinable terms.

→ Each box in the K-map corresponds to a single row / entry in the truth table.

Example of a K Map for a 3-Input Function?

→ Given this truth table:

A	B	C	Y	
0	0	0	0	0
0	0	1	1	1
0	1	0	0	2
0	1	1	1	3
1	0	0	0	4
1	0	1	0	5
1	1	0	0	6
1	1	1	0	7

A \ BC	00	01	11	10
0	R 0	R 1	R 3	R 2
1	R 4	R 5	R 7	R 6

(10: B, C) Because in Row 2, A=0, B=1, C=0

→ The "0" & "1" on y-axis indicate values of A

→ The "00", "01", "11", "10" are values of B & C, respectively.

B=0 C=0; B=0 C=1; B=1 C=1; B=1 C=0

How do we fill in the K-Map?

→ With the Y (output) of each row!

A \ BC	00	01	11	10
0	0	1	1	0
1	0	0	0	0

What is gray code?

→ The ordering that we used for the B and C values in the ex above.

→ In Gray code, successive values differ by only 1 bit - unlike binary:

Gray Code: 00, 01, 11, 10

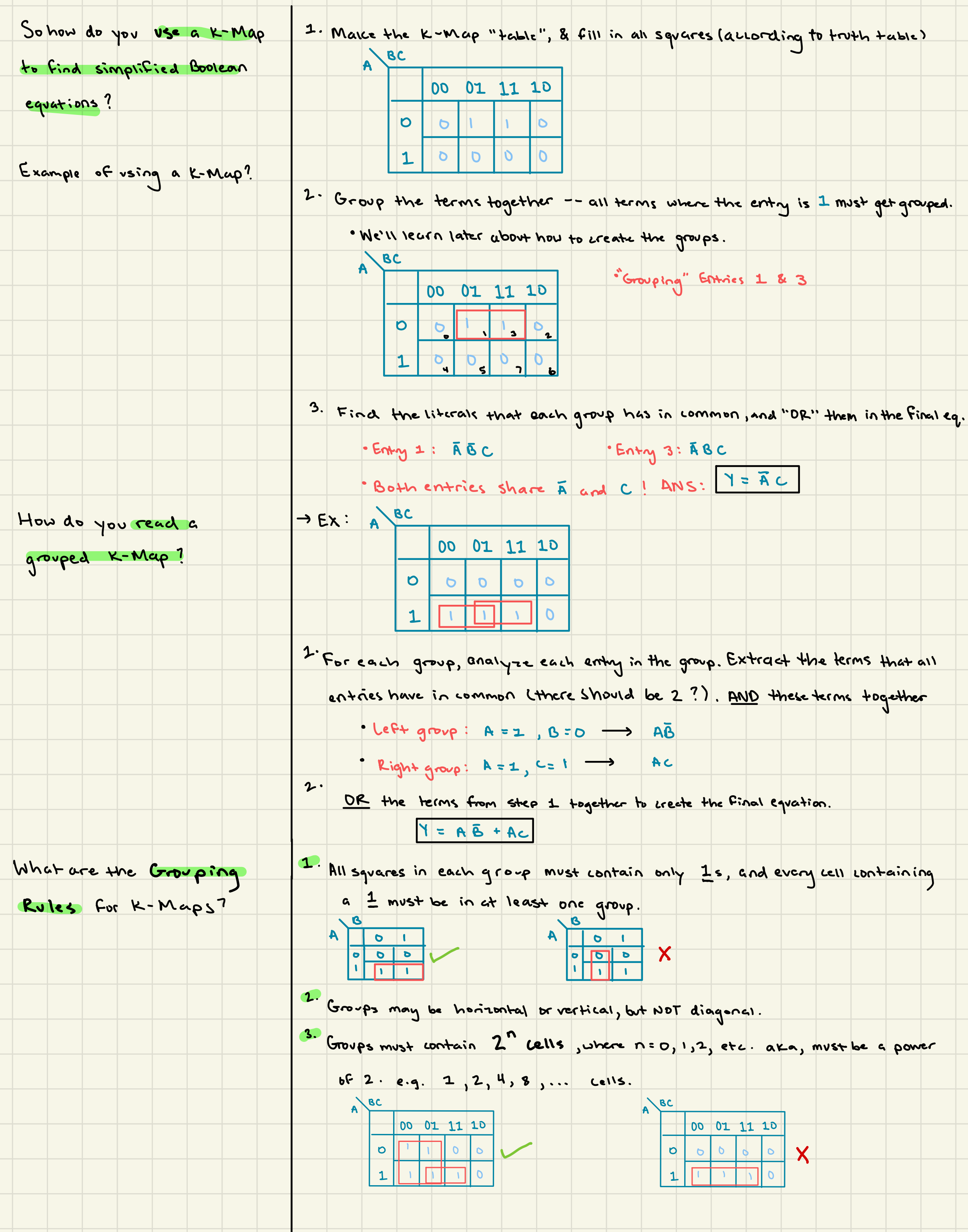
Binary: 00, 01, 10, 11

Why is gray code useful for K-Maps?

→ Each square in the K-Map differs from its adjacent squares by one literal: (eg L, R, above, or below)

A \ BC	00	01	11	10
0	0 (0)	1 (1)	1 (3)	0 (2)
1	0 (4)	0 (5)	0 (7)	0 (6)

- Entry 0 & Entry 1 both share "$\bar{A}\bar{B}$", and only differ by the value of C literal
- Entries 1 & 3 both differ only in their B literal



So how do you use a K-Map to find simplified Boolean equations?

Example of using a K-Map?

1. Make the K-Map "table", & fill in all squares (according to truth table)

A \ BC	00	01	11	10
0	0	1	1	0
1	0	0	0	0

2. Group the terms together -- all terms where the entry is 1 must get grouped.

- We'll learn later about how to create the groups.

A \ BC	00	01	11	10
0	0 (0)	1 (1)	1 (3)	0 (2)
1	0 (4)	0 (5)	0 (7)	0 (6)

"Grouping" Entries 1 & 3

3. Find the literals that each group has in common, and "OR" them in the final eq.

- Entry 1: $\bar{A}\bar{B}C$
- Entry 3: $\bar{A}BC$
- Both entries share $\bar{A}$ and C! ANS: $Y = \bar{A}C$

How do you read a grouped K-Map?

→ Ex:

A \ BC	00	01	11	10
0	0	0	0	0
1	1	1	1	0

1. For each group, analyze each entry in the group. Extract the terms that all entries have in common (there should be 2?). AND these terms together

- Left group: A = 1, B = 0 ⟶ $A\bar{B}$
- Right group: A = 1, C = 1 ⟶ AC

2. OR the terms from step 1 together to create the final equation.

$Y = A\bar{B} + AC$

What are the Grouping Rules for K-Maps?

1. All squares in each group must contain only 1s, and every cell containing a 1 must be in at least one group.

A \ B	0	1
0	0	0
1	1	1

✓

A \ B	0	1
0	0	0
1	1	1

X

2. Groups may be horizontal or vertical, but NOT diagonal.

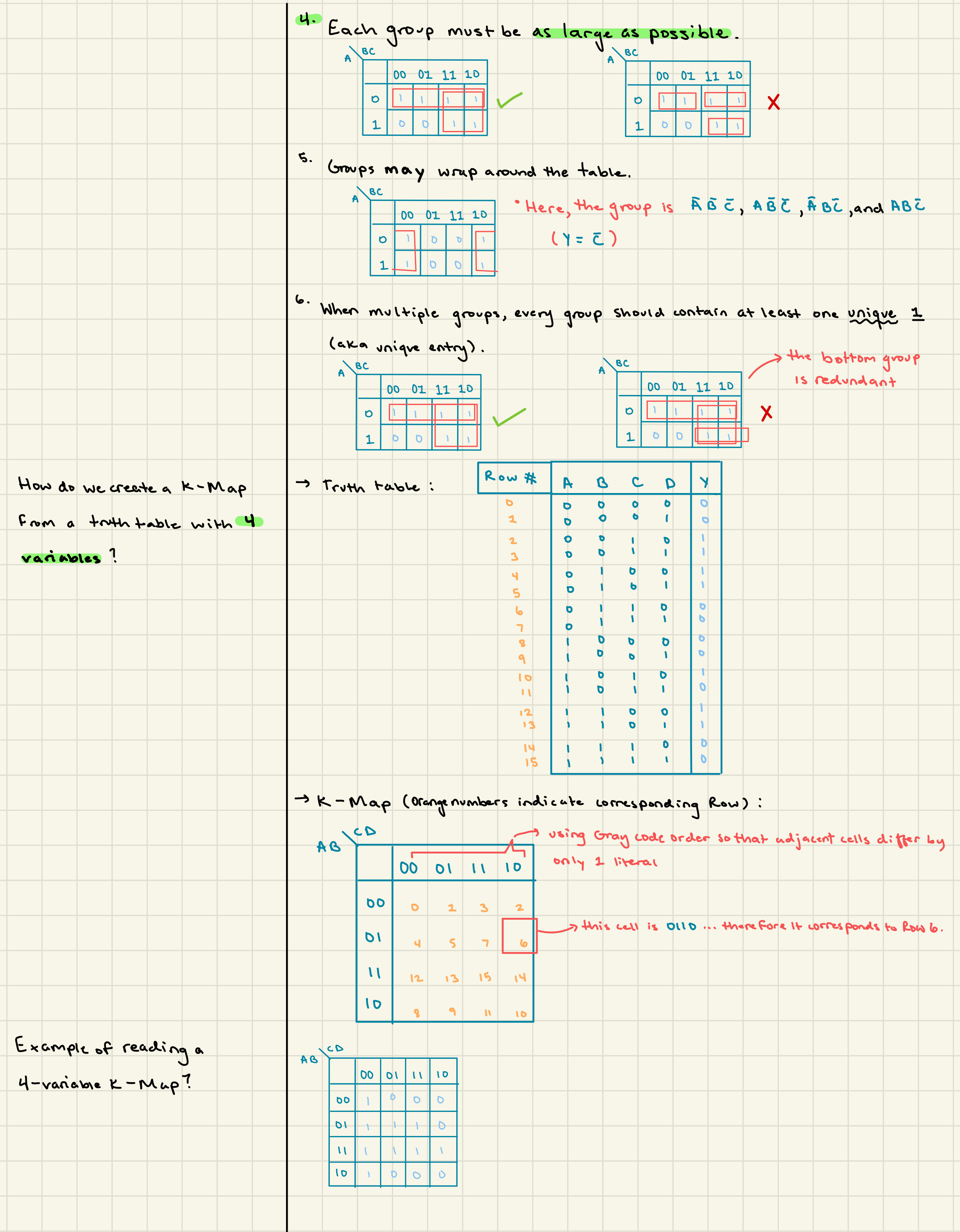
3. Groups must contain 2^n cells, where n = 0, 1, 2, etc. aka, must be a power of 2. e.g. 1, 2, 4, 8, ... cells.

A \ BC	00	01	11	10
0	1	1	0	0
1	1	1	1	0

✓

A \ BC	00	01	11	10
0	0	0	0	0
1	1	1	1	0

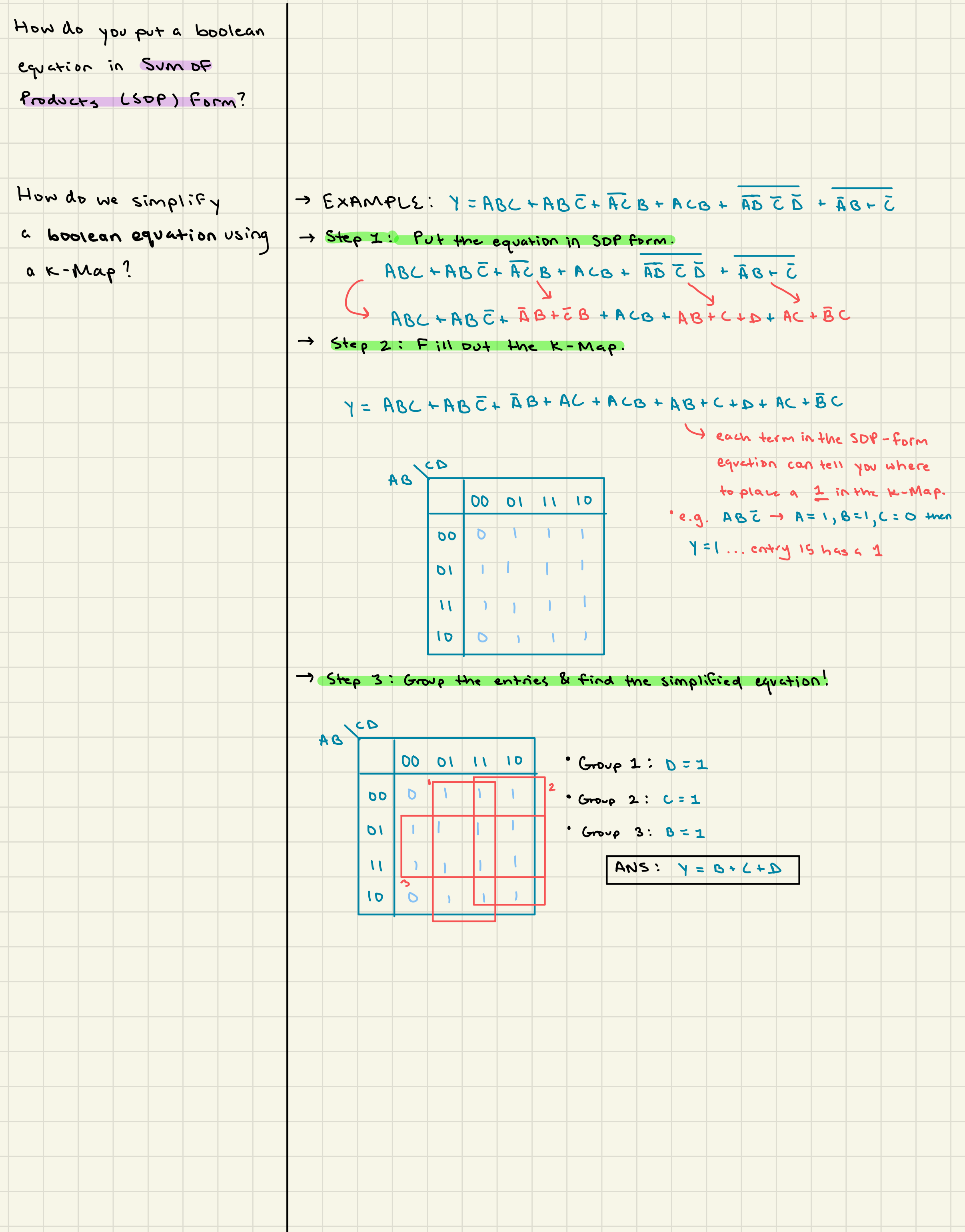
X



Row #	A	B	C	D	Y
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	0

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

AB \ CD	00	01	11	10
00	1	0	0	0
01	1	1	1	0
11	1	1	1	1
10	1	0	0	0



How do you put a boolean equation in Sum of Products (SOP) Form?
How do we simplify a boolean equation using a K-Map?
→ EXAMPLE: Y = ABC + ABC̄ + ĀC̄B + ACB + (ĀD̄ C̄ D̄)‾ + (ĀB + C̄)‾
→ Step 1: Put the equation in SOP form.
ABC + ABC̄ + ĀC̄B + ACB + (ĀD̄ C̄ D̄)‾ + (ĀB + C̄)‾
ABC + ABC̄ + ĀB + C̄B + ACB + AB + C + D + AC + B̄C
→ Step 2: Fill out the K-Map.
Y = ABC + ABC̄ + ĀB + AC + ACB + AB + C + D + AC + B̄C
each term in the SOP-form equation can tell you where to place a 1 in the K-Map.
e.g. ABC̄ → A = 1, B = 1, C = 0 then Y = 1 ... entry 15 has a 1
AB
CD
00
01
11
10
→ Step 3: Group the entries & find the simplified equation!
Group 1: D = 1
Group 2: C = 1
Group 3: B = 1
ANS: Y = B + C + D

Quiz 0 Review

Logic Gates

Name	Equation	Symbol	Transistor Count
AND	$Y = AB$	A, B → Y	6
OR	$Y = A + B$	A, B → Y	6
NOT	$Y = \overline{A}$	A → Y	2
NAND	$Y = \overline{AB}$	A, B → Y OR A, B → Y	4
NOR	$Y = \overline{A + B}$	A, B → Y OR A, B → Y	4
XOR	$Y = A \oplus B$	A, B → Y	12
XNOR	$Y = \overline{A \oplus B}$	A, B → Y	12

211 Review

unit	bits	example
byte	8	00001101
nibble	4	1101
word	16 or 32 ?	

→ With n bits,

- you can represent 2^n distinct values.
- LARGEST NUM: $2^n - 1$
- 2's complement most negative num: $-(2^{n-1})$
- 2's complement most positive num: $(2^{n-1}) - 1$

→ LSB = RIGHTMOST 8 bits. Stored in the 1st mem address (little endian)

→ MSB = LEFTMOST 8 bits. Stored in the last mem address

42,185 = 0b0000 0000 0000 0000 1010 0100 1100 1001

(MSB: 0b0000 0000; LSB: 1100 1001)

Address	Value
0	1100 1001
1	1010 0100
2	0000 0000
3	0000 0000

→ Bits needed to address memory = $\log_2$ (size in bytes of total memory)

Quiz 0 Review

211 Review

Decimal	0	1	2	3	4	5	6	7	8	9	10	11
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011
Hex	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB

Decimal	12	13	14	15	16	17	18	19
Binary	1100	1101	1110	1111	10000	10001	10010	10011
Hex	0xC	0xD	0xE	0xF	0x10	0x11	0x12	0x13

→ Converting decimal to hex:

1. Fill out this chart __ __ __ __ (4096 256 16 1), where every blank can have a max value of F (aka 11)

Ex 50 → 0 0 3 2 because 3(16) + 2(1) = 50

→ Finding address of element in array: (index [i] × sizeof (element type)) + base addr

Ex • Element 10 in int array; base addr = 0x 0000 300C

1. 10 × sizeof(int) = 4 = 40
2. Add decimal num from step 1 to decimal version of last relevant digits of base address:
   - 0C → 12
   - 40 + 12 = 52
3. Convert ans to decimal: 52 → 16(3) + 1(4) → 0x34
4. Add result to base addr (replacing last digits w/ result):

```
  0x 0000 3000
+ 0x 0000 0034
--------------
  0x 0000 3034  ✓✓
```

Transistors

→ nMOS: V_g low = open
V_g high = closed = ON

→ pMOS: V_g low = closed = ON
V_g high = open

nMOS → "normal" MOS → the normal thinking would be that "high" power results in it flowing through. Therefore, in nMOS, V_g high ⇒ circuit closed ⇒ power on.

Multiplexers

What is a 2:1 multiplexer? (2:1 mux)

→ A circuit that chooses an output from among 2 inputs, based on the value of a **select signal**.

→ Multiplexers are useful for implementing if-statements.

What is the 2:1 mux symbol?

Inputs: A → 0, B → 1; output Y; select signal S

These labels tell us how the mux chooses the output based on the value of the select signal

S → select signal: selects which input gets sent to output

Example of a regular schematic for an if-statement?

Statement

if $(S == 0)$: $Y = A$

else if $(S == 1)$: $Y = B$

Truth Table

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

Rows 1–2: Y = A = 0; rows 3–4: Y = A = 1; rows 5, 7: Y = B = 0; rows 6, 8: Y = B = 1

SOP Equation

$$Y = \bar{S}A\bar{B} + \bar{S}AB + S\bar{A}B + SAB$$

$$= A\bar{S}(\bar{B}+B) + BS(\bar{A}+A)$$

$$= \boxed{A\bar{S} + BS}$$

Diagram

Inputs A, B, S → gates → Y

How could we replace this diagram with a 2:1 mux?

→ A → 0, B → 1, select S, output Y

- When $S = 0$, it signals output $= Y = A$
- When $S = 1$, it signals output $= Y = B$

→ This mux diagram replaces/avoids having to draw that bigger more complicated gate diagram.

What is a 4:1 multiplexer? (4:1 mux)

→ A circuit that chooses an output from among 4 inputs, based on the value of a select signal.

→ Since we need 4 signaling options, S must be a 2-bit wire that can send a 2-bit signal ($n = 2$, $2^2 = 4$ "bit combinations")

How does the 4:1 mux diagram look?

→ Ex: For the Truth Table:

S_1	S_0	Y
0	0	A
0	1	B
1	0	C
1	1	D

A → 00, B → 01, C → 10, D → 11; output Y; select $S_{1:0}$

2-bit wire: S[1] S[0]

- When $S = 0b00$, $Y = A$
- When $S = 0b01$, $Y = B$
- When $S = 0b10$, $Y = C$
- When $S = 0b11$, $Y = D$

How can you implement the same truth table with a 4:1 mux made out of 2 2:1 muxes?

A → 0, B → 1 (select S_0) → 0; C → 0, D → 1 (select S_0) → 1; select S_1 → Y

Definitions in Digital Logic

What is a literal?

→ A single variable. May be complemented.

→ eg A, B, $\bar{A}$

What is a product term?

→ An AND of individual literals

→ eg $A\bar{B}C$. But NOT $A\overline{BC}$ (because "$\overline{BC}$" isn't a literal)

What is a minterm?

→ A product term in which all variables appear once.

→ eg $\bar{A}\bar{B}\bar{C}$, $\bar{A}BC$, $\bar{A}B\bar{C}$, or ABC

→ but NOT A, $\bar{A}C$, BC, etc.

How do you derive an equation using minterms?

Truth Table

A	B	C	F	Minterm	Minterm name
0	0	0	1	$\bar{A}\bar{B}\bar{C}$	m_0
0	0	1	0	$\bar{A}\bar{B}C$	m_1
0	1	0	1	$\bar{A}B\bar{C}$	m_2
0	1	1	0	$\bar{A}BC$	m_3
1	0	0	0	$A\bar{B}\bar{C}$	m_4
1	0	1	1	$A\bar{B}C$	m_5
1	1	0	0	$AB\bar{C}$	m_6
1	1	1	1	ABC	m_7

1. Write the minterm for each row.
2. Take the sum of the minterms for rows where output = 1

Equation

$F = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}C + ABC$

OR

$F(A,B,C) = \Sigma(m_0, m_2, m_5, m_7)$

What is sum of products?

→ When 2 or more **product terms** are summed (OR) together.

→ e.g. $Y = AB + A\bar{C}$

→ but NOT $Y = (A+B)(C+A)$

What is Canonical Sum of Products form?

→ An SOP form in which each product contains all literals, e.g.

$F = \bar{A}\bar{B}\bar{C} + \bar{A}BC + A\bar{B}C + ABC$

What is simplified Sum of Products form?

→ When you use boolean algebra to simplify the canonical SOP form, e.g.

$F = \bar{A}\bar{B}\bar{C} + C(\bar{A}B + A(\bar{B}+B)) \Rightarrow \bar{A}\bar{B}\bar{C} + C(\bar{A}B + A)$

$\Rightarrow \bar{B}(\bar{A}\bar{C}) + AC + AB \Rightarrow \boxed{\bar{A}\bar{C} + AC}$

What is a maxterm?

→ A term in which all variables appear once, as literals ORd together

→ eg $\bar{A} + B + \bar{C}$

→ To write the maxterm for a truth table row, sum the **complement** of each literal's value together.

→ Eg:

A	B	C	Y
0	1	0	1

→ $A + \bar{B} + C = Y$

How do you derive an equation using maxterms?

1. Write the maxterm for each row
2. Take the product of the maxterms for rows where output = 0

Truth Table

A	B	C	F	Maxterm	Maxterm name
0	0	0	1	$\bar{A}+\bar{B}+\bar{C}$	M_0
0	0	1	0	$A+B+\bar{C}$	M_1
0	1	0	1	$A+\bar{B}+C$	M_2
0	1	1	0	$A+\bar{B}+\bar{C}$	M_3
1	0	0	0	$\bar{A}+B+C$	M_4
1	0	1	1	$\bar{A}+B+\bar{C}$	M_5
1	1	0	0	$\bar{A}+\bar{B}+C$	M_6
1	1	1	1	$\bar{A}+\bar{B}+\bar{C}$	M_7

Equation

$$F = (A+B+\bar{C})(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+\bar{B}+C)$$

OR

$$F = \Pi(M_1, M_3, M_4, M_6)$$

Why does this derived equation work?

→ Creating the canonical SOP equation for $\bar{F}$

$$\bar{F} = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C}$$

And negating both sides + applying DeMorgans

$$F = \overline{\bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C}}$$

$$F = (A+B+\bar{C})(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+\bar{B}+C)$$

Actually yields the same term!

K-Map Definitions

What is an implicant?

→ Any product term (RECALL: AB, $\bar{B}C$, etc) whose output for a given Boolean equation is 1.

→ aka, in a K-Map, the terms that define any group of 1s.

Example?

A \ BC	00	01	11	10
0	0	0	1	0
1	0	1	1	0

Implicants

$\bar{A}BC$ BC

$A\bar{B}C$ AC

ABC

What is a prime implicant?

→ An implicant that is not a subset of any other implicant.

→ aka, in a K-Map, an implicant that corresponds to a group which can NOT be covered by any other group.

Example?

AB \ CD	00	01	11	10
00	1	1	0	0
01	0	1	1	0
11	1	1	1	0
10	1	1	0	0

- red = implicants
- prime implicants:

  $\bar{C}D$, $\bar{B}\bar{C}$

What is an essential prime implicant?

→ A prime implicant where:

- At least 1 element is not covered by 1 or more other prime implicants.

→ aka, in a K-Map, a group that is necessary to use in the final solution to cover all 1s.

Example?

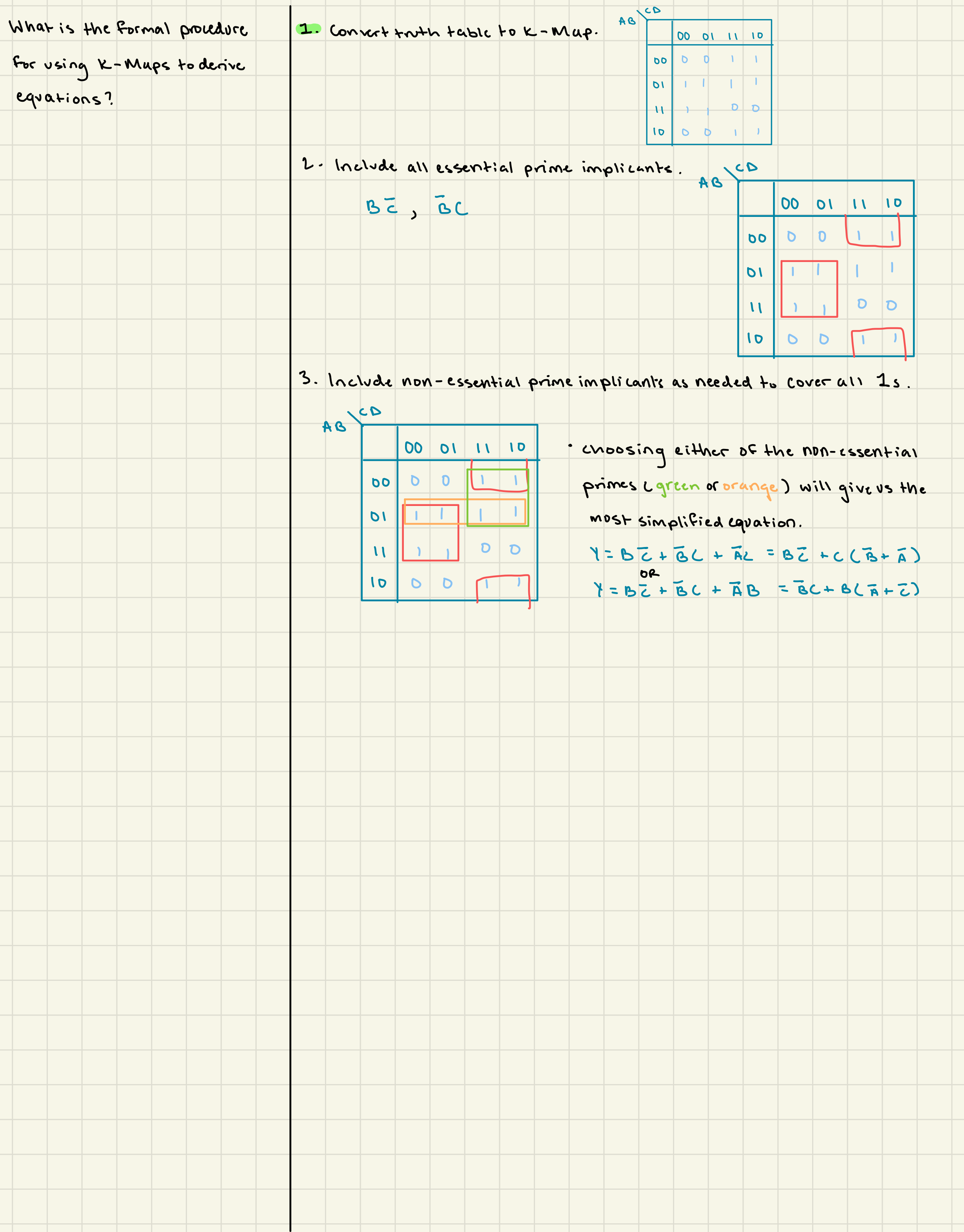
AB \ CD	00	01	11	10
00	1	1	0	0
01	0	1	1	0
11	1	1	1	0
10	1	1	0	0

- red → essential prime implicants, aka the final groups (in this case)
- orange → non-essential prime implicant.

What is an non-essential prime implicant?

→ A prime implicant that:

- contains NO elements which can't be covered by another prime implicant group


What is the formal procedure for using K-Maps to derive equations?
1. Convert truth table to K-Map.
AB \ CD
00 01 11 10
00 0 0 1 1
01 1 1 1 1
11 1 1 0 0
10 0 0 1 1
2. Include all essential prime implicants.
BC̄ , B̄C
AB \ CD
00 01 11 10
00 0 0 1 1
01 1 1 1 1
11 1 1 0 0
10 0 0 1 1
3. Include non-essential prime implicants as needed to cover all 1s.
AB \ CD
00 01 11 10
00 0 0 1 1
01 1 1 1 1
11 1 1 0 0
10 0 0 1 1
• choosing either of the non-essential primes (green or orange) will give us the most simplified equation.
Y = BC̄ + B̄C + ĀC = BC̄ + C(B̄ + Ā)
OR
Y = BC̄ + B̄C + ĀB = B̄C + B(Ā + C̄)

Lab 0

What is a tunnel?	→ A way to draw an "invisible wire" to bind 2 points together
	→ Tunnels are grouped by case sensitive labels
Example?	→ Normal circuit:
	→ With Tunnels:

A 0 — Y_0, Y_1

A 0 — IN

IN — 0 Y_0

IN — 0 Y_1

Adder/Subtractor

How can we label the parts of a binary addition operation?

```
  [1]7 [1]5 [1]3
        1    0  |1|  1
  +     1    1  |1|
  ----------------
  [1]6 [0]4 [0]2
```

1. Column 0
2. Column 0 sum (S_0)
3. Carry-out from column 0 ($C_{o,0}$); Carry-in to column 1 ($C_{in,1}$)
4. Column 1 sum (S_1)
5. Carry-out column 1 ($C_{o,1}$); Carry-in to column 2 ($C_{in,2}$)
6. Column 2 sum (S_2)
7. Carry-out from column 2 ($C_{o,2}$)

What is a half-adder circuit?

→ A circuit that can add together 2 one-bit values

→ It can add the 2 bits in column 0 (A and B) and output 2 things:

- The sum S
- The carry-out bit C_o

Half Adder Circuit: inputs A, B; outputs C_o, S

What is the truth table and equation(s) for half adder?

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = \bar{A}B + A\bar{B} \approx A \oplus B$ (exclusive OR)

$C_o = AB$

What is the circuit diagram?

A, B → XOR gate → S; A, B → AND gate → C_o

What is a full adder circuit?

→ A circuit that can add together **three** one-bit values and output the sum S and the CO-bit C_o.

Full Adder Circuit: inputs A, B, C_i; outputs C_o, S

→ This circuit can be used to add the values of a column of binary addition when there was a CO-bit, eg column 1:

```
    1  1 |1|
       1 |0|  1
  +    1 |1|  1
  -------------
    1  1 |0|  0
```

→ The 3 inputs of the addition in this EX are A, B, and C_i (the CO-bit from previous column).

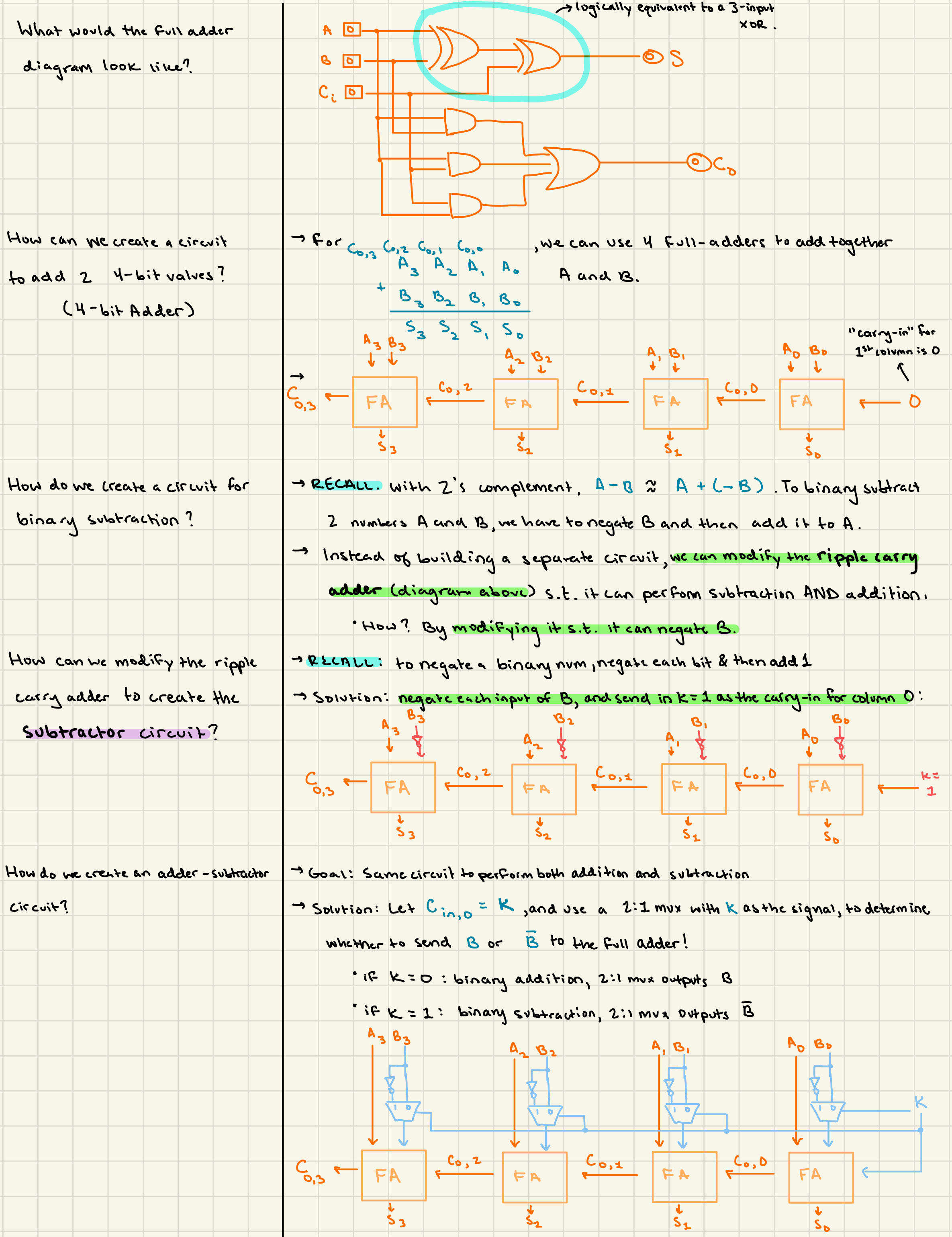
What is the truth table & derived equation?

A	B	C_i	C_o	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

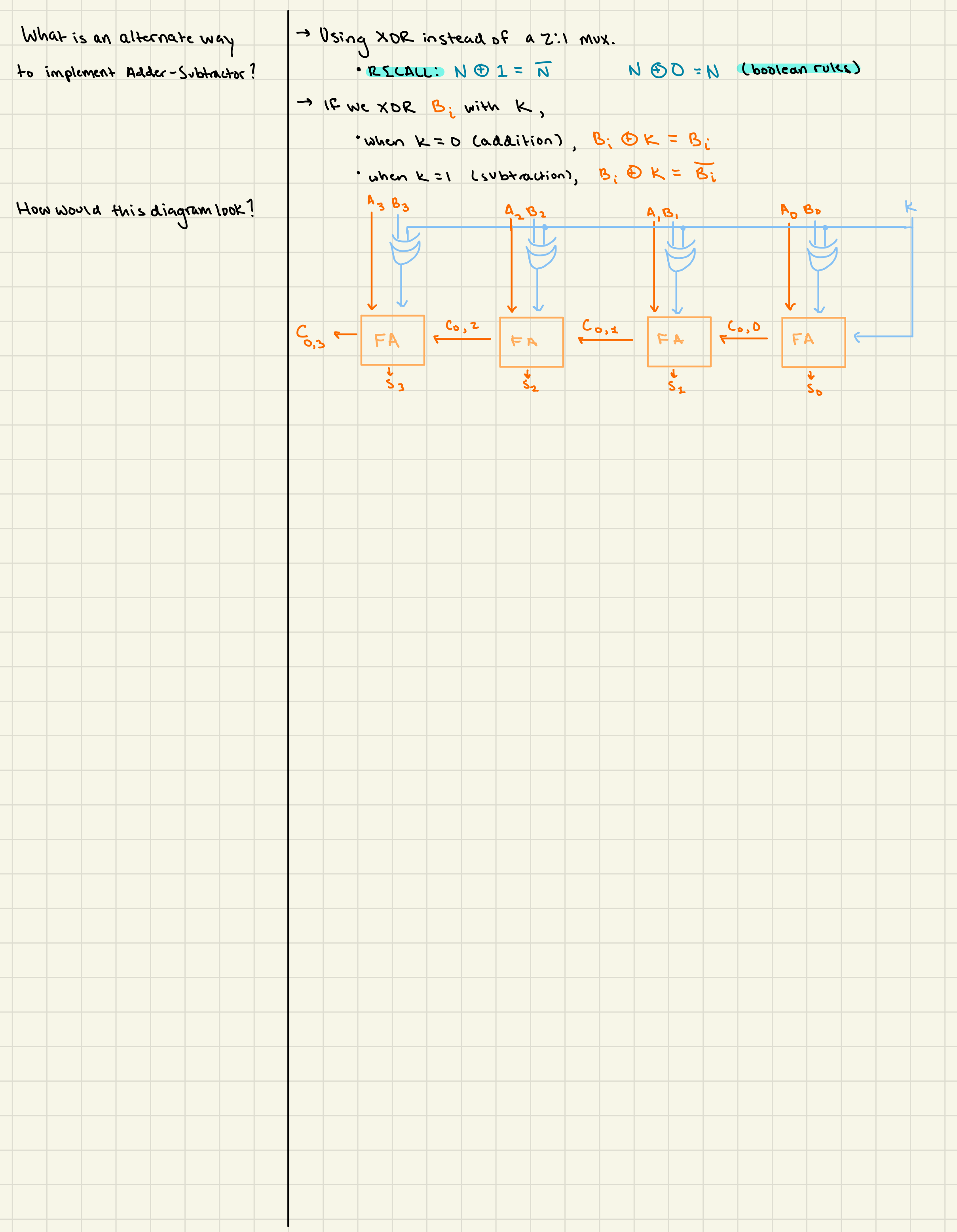
- Notice that when an odd # of inputs = 1 (1 or 3 inputs), $S = 1$. A 3-input XOR can be used to represent this

$S = A \oplus B \oplus C_i$

- $C_o = AC_i + AB + BC_i$ (K-Map used)



What would the full adder diagram look like?
→ logically equivalent to a 3-input XOR.
A
B
C_i
S
C_o
How can we create a circuit to add 2 4-bit values? (4-bit Adder)
→ For C_o,3 C_o,2 C_o,1 C_o,0 / A_3 A_2 A_1 A_0 / + B_3 B_2 B_1 B_0 / S_3 S_2 S_1 S_0, we can use 4 full-adders to add together A and B.
"carry-in" for 1st column is 0
FA
C_o,3
C_o,2
C_o,1
C_o,0
0
S_3 S_2 S_1 S_0
How do we create a circuit for binary subtraction?
→ RECALL: with 2's complement, A − B ≈ A + (−B). To binary subtract 2 numbers A and B, we have to negate B and then add it to A.
→ Instead of building a separate circuit, we can modify the ripple carry adder (diagram above) s.t. it can perform subtraction AND addition.
• How? By modifying it s.t. it can negate B.
How can we modify the ripple carry adder to create the subtractor circuit?
→ RECALL: to negate a binary num, negate each bit & then add 1
→ Solution: negate each input of B, and send in k=1 as the carry-in for column 0:
k=1
How do we create an adder-subtractor circuit?
→ Goal: Same circuit to perform both addition and subtraction
→ Solution: Let C_in,0 = K, and use a 2:1 mux with K as the signal, to determine whether to send B or B̄ to the full adder!
• if K=0: binary addition, 2:1 mux outputs B
• if K=1: binary subtraction, 2:1 mux outputs B̄
K


What is an alternate way to implement Adder-Subtractor?
→ Using XOR instead of a 2:1 mux.
• RECALL: N ⊕ 1 = N̄   N ⊕ 0 = N (boolean rules)
→ If we XOR Bi with K,
• when K = 0 (addition), Bi ⊕ K = Bi
• when K = 1 (subtraction), Bi ⊕ K = B̄i
How would this diagram look?
A3 B3
A2 B2
A1 B1
A0 B0
K
C0,3
FA
C0,2
FA
C0,1
FA
C0,0
FA
S3
S2
S1
S0

Comparator and Shifter

What is an equality comparator circuit?

→ A circuit that determines whether the 2 inputs are equal

A, B (n) → Equality Comparator → Equal?

n = # of bits in the input

no diagonal line = default 1-bit input/output

How would you design a circuit to compare 4-bit inputs and output 1 if they are equal?

→ Notice that when comparing 2 1-bit inputs, $\overline{A \oplus B}$ (XNOR) = 1 when A = B and 0 when A != B:

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

A_0, B_0; A_1, B_1; A_2, B_2; A_3, B_3 → XNOR gates → AND gate → Equal

→ Alternatively, we can save transistors by converting the AND gate to a NOR gate and subsequently turning the XNOR gates into XOR gates.

How do we design a circuit to perform a left-shift on a 4-bit input?

→ Since n = 4, shifting an input A by more than n-1 = 3 bits will always produce 0b0000. So we only need to support the A<<1, A<<2, and A<<3 operations in our circuit.

- Ex: A = 0b1101
  - A<<1 = 1010
  - A<<2 = 0100
  - A<<3 = 1000
  - A<<4 = 0000

→ Solution: Have a 2-bit "shamt" input that acts as a select signal to the output Y. It's 2 bits, so can represent up to 4 values.

number to perform shift on → $A_{3:0}$ (4) → Left Shift Circuit → $Y_{3:0}$ (4) ← result

$shamt_{1:0}$ (2) ← # of bits to shift by

What would be the truth table?

→ Let $A_3 A_2 A_1 A_0$ depict the value of A, and same for Y.

Output Value

Shift Amount	Y_3	Y_2	Y_1	Y_0
0	A_3	A_2	A_1	A_0
1	A_2	A_1	A_0	0
2	A_1	A_0	0	0
3	A_0	0	0	0

The diagram?

A3, A2, A1, A0, shamt → Y3, Y2, Y1, Y0

Arithmetic Logic Unit (ALU)

What is a boolean unit?

→ A circuit/logic unit that can perform 4 boolean operations (AND, OR, XOR, NOR)

→ Encompasses the logic for all 4 operations, and uses a 2-bit control signal (RECALL: multiplexers!) to tell the circuit which operation to perform.

A, B (n bits each) → Boolean Unit ← Bool (2 bits); output Y (n bits)

How does the Boolean unit operate?

→ The value of the bool select signal indicates which operation to perform:

Bool	Operation	Expression
00	AND	AB
01	OR	$A+B$
10	XOR	$A \oplus B$
11	NOR	$\overline{A+B}$

→ The unit will perform all of the operations, and then output the one that was requested based on the value of "bool".

What is the diagram for a boolean unit?

Inputs A (n), B (n) feed AND, OR, XOR, NOR gates → multiplexer inputs 00, 01, 10, 11, selected by Bool (2) → output Y (n)

What is the Bidirectional shifter?

→ A circuit/logic unit that can perform 3 shift operations: left shift, logical right shift, or arithmetic right shift.

→ Encompasses the logic for all 3 operations, and uses a 2-bit control signal to tell the circuit which operation to perform.

RECALL: What is a "logical" vs "arithmetic" right shift?

Operation	Expression	Meaning	Example
logical right shift	A >> B	Shift B to the right by A bits, and pad the left side with 0s	1100 >> 2 = 0011
arithmetic right shift	A >>> B	Shift B to the right by A bits, and pad the left side with the MSB of B! • if MSB(B) = 1, B is neg., so we pad with 1s. • if MSB(B) = 0, then B << A ≈ B <<< A	1100 >>> 2 = 1111

How does the bidirectional shifter operate?

→ The 2-bit select signal "bool" indicates which operation to perform:

Bool	Operation	Expression
0 0	left shift	B << A
0 1	NONE	N/A
1 0	logical R-shift	B >> A
1 1	arithmetic R-shift	B >>> A

The left bit (Bool[1]) tells you the direction of the shift (L or R)

The right bit tells you the type of shift (logic or arithmetic)

we only need to perform 3 diff operations

A
n
B
n
control signal
Bool
2
Bidirectional Shifter
n
result

What is the add/sub?

→ RECALL: The adder/subtractor logic unit!

→ Uses a 1-bit control signal to tell the circuit which operation to perform.

Sub	Operation	Expression
0	Add	A + B
1	Sub	A − B

A
n
B
n
control signal
Sub
Add/Sub
n
Y

What is an Arithmetic Logic Unit?

→ A circuit/logic unit that can perform add/sub, bidirectional shift, AND boolean operations!

→ The value of the Shift and Math select signals are used to indicate which operation to perform.

What are "Shift" and "Math"?

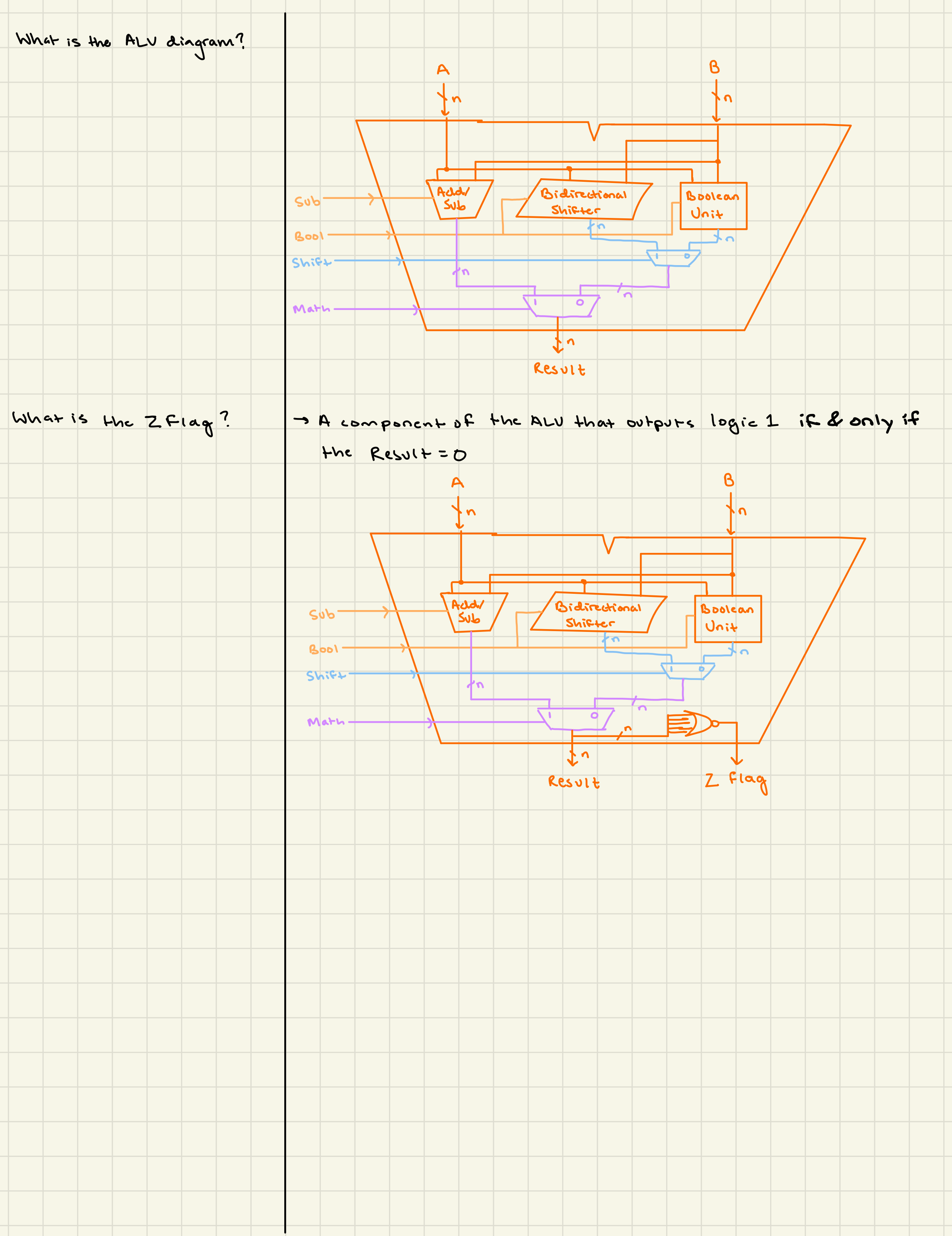
- Shift (1-bit): if Shift = 1, perform a shift operation based on the value of Bool. If Shift = 0, perform a boolean operation based on the value of Bool.
- Math (1-bit): if Math = 1, perform an add/sub op. based on the value of Sub. if Math = 0, perform a boolean or shift op. based on the values of Shift and Bool.

Table of all operations in the ALU?

Math	Shift	Sub	Bool	Operation
1	X	0	XX	A + B
1	X	1	XX	A − B
0	1	X	00	B << A
0	1	X	10	B >> A
0	1	X	11	B >>> A
0	0	X	00	A AND B
0	0	X	01	A OR B
0	0	X	10	A XOR B
0	0	X	11	A NOR B

When Math = 1, we don't care ab the values of Shift and Bool; we just need to know if doing ADD or SUB

When Math = 0, we don't care ab the value of Sub.


What is the ALU diagram?
A
n
B
n
Add/
Sub
Bidirectional
Shifter
Boolean
Unit
Sub
Bool
Shift
Math
n
n
n
n
1
0
1
0
n
Result
What is the Z Flag?
→ A component of the ALU that outputs logic 1 if & only if
the Result = 0
A
n
B
n
Add/
Sub
Bidirectional
Shifter
Boolean
Unit
Sub
Bool
Shift
Math
n
n
n
n
n
1
0
1
0
n
Result
Z Flag

Waveform Diagrams

What are waveform diagrams?

→ A way to represent the values of our signals over time.

→ Each one-bit signal will have its own waveform.

How does a waveform look?

→ Like a square wave, where a low value ≈ logic 0 and a high value ≈ logic 1:

What are rising and falling edges?

→ Rising edge: when a signal "rises" from logic low (0) to logic high (1).

→ Falling edge: when a signal "falls" from logic high (1) to logic low (0).

Example of a waveform diagram?

→ Take the following circuit:

- At time = 0ps, we will set input A = 0 and B = 1
- At time t = 40 ps, we will change input A to A = 1.

What is propogation delay?

→ The time taken for a signal to travel from input to output

- It doesn't happen immediately, as we've been assuming thus far.

→ REASON: Time for electricity to travel through a wire; capacitors charging & discharging, etc.

→ Denoted: t_{pd} = time of propogation delay.

How does propogation delay affect our waveform diagrams?

→ When we change an input (e.g. A or B), the change in the output's waveform diagram won't be reflected immediately - b/c first there is a delay of x picoseconds.

→ The exact value of t_{pd} for different logic gates will be provided by lecc; don't need to calculate.

Example of computing propogation delay?

→ Let t_{pd} (NOT gate) = 10 and t_{pd} (OR gate) = 20.

→ If the value of input A changes at t = n, the value won't be inverted until n + 10ps.

→ If the value of input A changes at t = n, the value of Y won't change until

n + 10ps (NOT gate) + 20ps (OR gate) = n + 30ps

- Therefore, delay from A → Y = 30 ps

→ Delay from B to Y = 20 ps

How do we draw the waveform diagram for this circuit?

→ GIVEN: A and B start at 0. at t = 40ps, we will set A = 1.

→ To visualize the p.d., we also track a "point C" placed right after the inverter.

What is the longest combinatorial path?

→ Given a circuit and the t_{pd} values for each logic gate, the LCP is the path from an input to Y that has the greatest propogation delay.

→ Ex

Gate	t_{pd} (ps)
NOT	15
NAND	20
AND	30
OR	40

→ LCP = A → Y :

15 ps + 20ps + 40ps = 75 ps

→ Shortest path = B → Y : 20 ps + 40ps = 60 ps

Flip-Flops

3.05mm (normal highlighter)

What is a D Flip Flop?

→ A digital electronic circuit that is used to **delay** ("D" = delay) the change of state of its output signal using clock timing.

→ Diagram:

- D = the value we want to store
- Q = the stored value (from D) at any given point.
- $\bar{Q}$ = Inverted stored value
- C = the **clock**; determines <u>when</u> D is stored.

What is a clock?

→ A one-bit signal that oscillates (or "toggles") back and forth between 0 and 1 at a constant/consistent pace (e.g. "clock period = 700 ps")

→ The purpose of the clock is to ensure that our circuits stay in sync.

What is a period?

→ RECALL Calculus: The period of a clock is the time between one <u>rising</u> edge and the next:

Period

→ The period is how we define/quantify the speed of a clock; e.g. "A clock with clock period = 700ps".

→ The period determines how **fast** our circuit runs; the shorter the period, the faster our circuit.

What are the 2 types of D Flip-Flops?

1. **Positive-Edge triggered:**
   - Q stores the value of D when the clock goes from $0 \to 1$
   - Q updates on every rising edge of the clock
2. **Negative-Edge triggered:**
   - Q stores the value of D when the clock goes from $1 \to 0$
   - Q updates on every falling edge of the clock

How does a positive-edge D flip-flop work?

→ Basically, we have 1 or more inputs and some logic circuit that outputs a value D based on the input(s) — e.g., a NAND circuit, a full adder, an ALU, etc.

→ The Flip-Flop adds a variable Q which starts at some specified value (e.g. 0) and then, every time the clock has a rising edge — aka every p seconds, where p = the period length —, the value of Q is updated to equal the value of D at that point in time.

Example of a waveform diagram for a pos. edge-triggered flip flop?

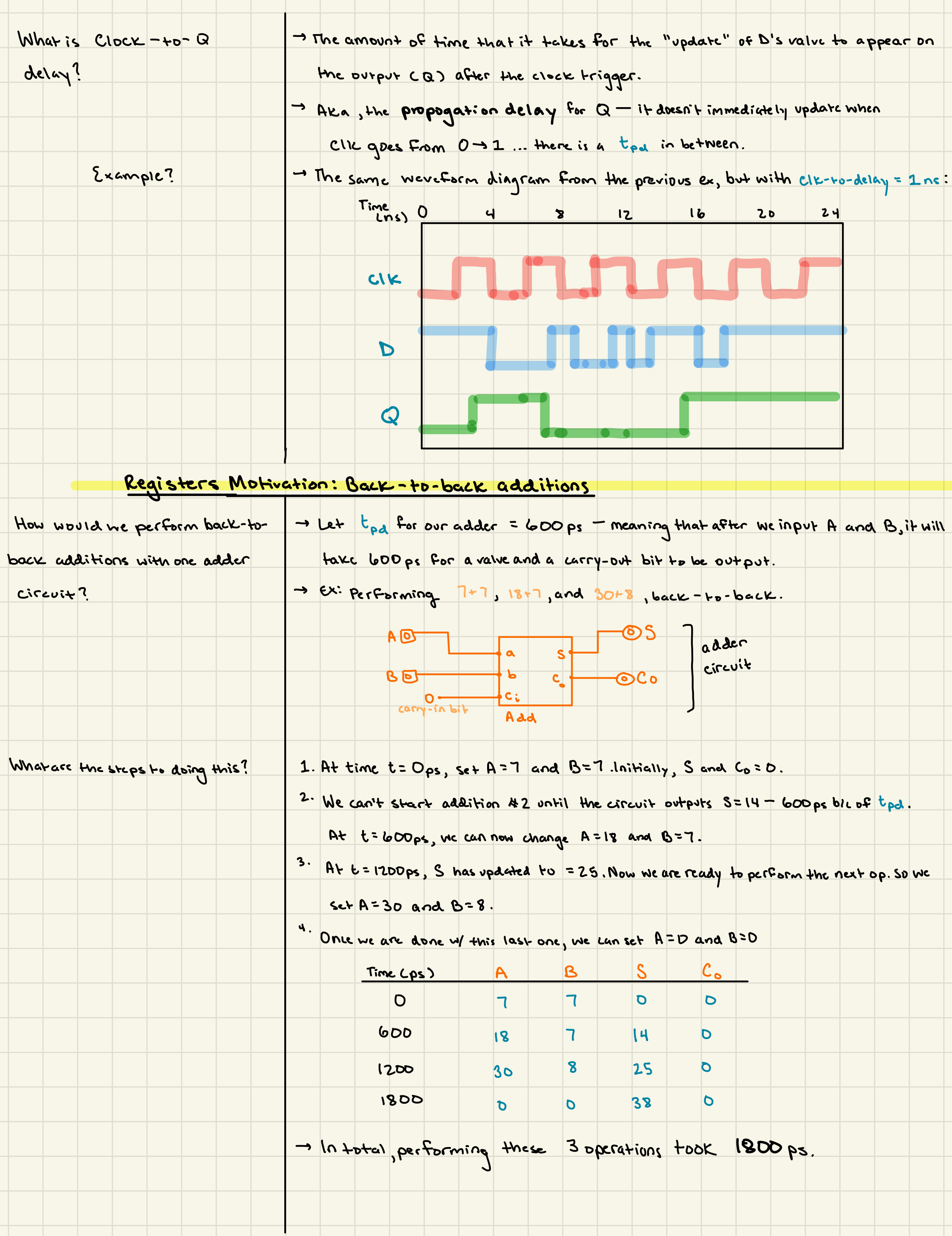
Time (ns): 0, 4, 8, 12, 16, 20, 24

Clk

D

Q

- The clock has a period of 4 ns
- Every time clock has a rising edge, check the value of D and change Q (if needed).
- Notice that D=0 at both indicated clock rising edges



What is Clock-to-Q delay?

→ The amount of time that it takes for the "update" of D's value to appear on the output (Q) after the clock trigger.

→ Aka, the **propogation delay** for Q — it doesn't immediately update when clk goes from 0 → 1 ... there is a t_{pd} in between.

Example?

→ The same waveform diagram from the previous ex, but with clk-to-delay = 1 ns:

Registers Motivation: Back-to-back additions

How would we perform back-to-back additions with one adder circuit?

→ Let t_{pd} for our adder = 600 ps — meaning that after we input A and B, it will take 600 ps for a value and a carry-out bit to be output.

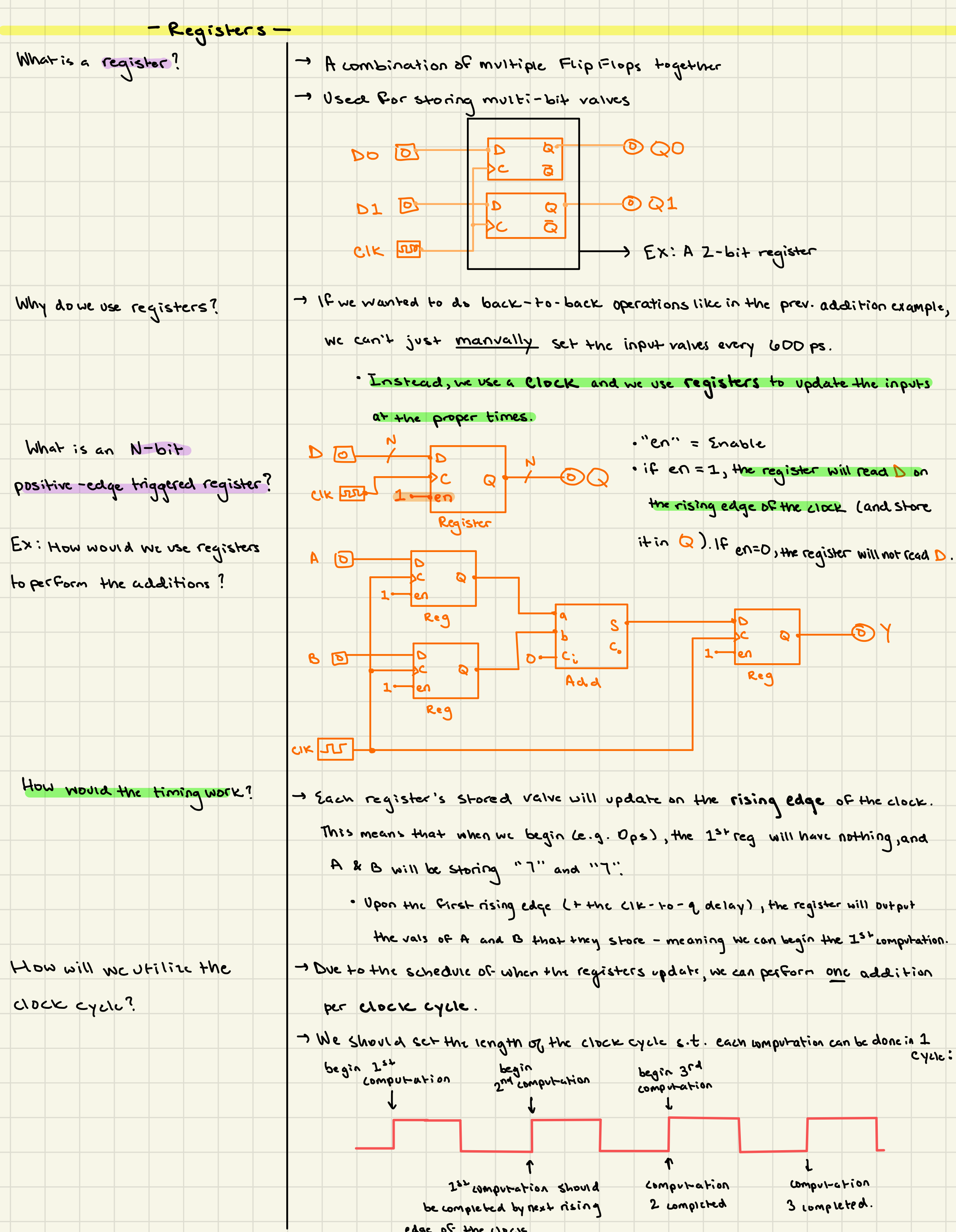
→ Ex: Performing 7+7, 18+7, and 30+8, back-to-back.

What are the steps to doing this?

1. At time t = 0ps, set A = 7 and B = 7. Initially, S and C_o = 0.
2. We can't start addition #2 until the circuit outputs S = 14 — 600 ps b/c of t_{pd}. At t = 600ps, we can now change A = 18 and B = 7.
3. At t = 1200ps, S has updated to = 25. Now we are ready to perform the next op. So we set A = 30 and B = 8.
4. Once we are done w/ this last one, we can set A = 0 and B = 0

Time (ps)	A	B	S	C_o
0	7	7	0	0
600	18	7	14	0
1200	30	8	25	0
1800	0	0	38	0

→ In total, performing these 3 operations took 1800 ps.



– Registers –

What is a register?

→ A combination of multiple Flip Flops together

→ Used for storing multi-bit values

→ Ex: A 2-bit register

Why do we use registers?

→ If we wanted to do back-to-back operations like in the prev. addition example, we can't just manually set the input values every 600 ps.

- Instead, we use a **clock** and we use **registers** to update the inputs at the proper times.

What is an N-bit positive-edge triggered register?

- "en" = Enable
- if en = 1, the register will read D on the rising edge of the clock (and store it in Q). If en=0, the register will not read D.

Ex: How would we use registers to perform the additions?

How would the timing work?

→ Each register's stored value will update on the rising edge of the clock. This means that when we begin (e.g. 0ps), the 1st reg will have nothing, and A & B will be storing "7" and "7".

- Upon the first rising edge (+ the clk-to-q delay), the register will output the vals of A and B that they store – meaning we can begin the 1st computation.

How will we utilize the clock cycle?

→ Due to the schedule of when the registers update, we can perform one addition per clock cycle.

→ We should set the length of the clock cycle s.t. each computation can be done in 1 cycle:

Timing

Example of performing back-to-back additions with registers?

A, next_A, D, C, en, Q, Reg, 1
B, next_B, D, C, en, Q, Reg, 1
curr_A, curr_B, a, b, C_i, S, C_o, 0, Add
curr_sum, D, C, en, Q, Reg, 1, prev_sum, Y
clk

→ In this Ex, we will perform 30+8, 18+7, and 7+7. We will complete **one addition** per clock cycle.

What is the starting state?

→ All values = 0

What happens in Cycle 0?

→ On the **Rising Edge**, the registers will update to hold values for A and B:

- next_A = 30
- next_B = 8

→ After the **clk-to-q** delay, the values stored in registers will get stored in Q:

- next_A = 30
- next_B = 8
- curr_A = 30
- curr_B = 8

→ After the **adder propagation delay** (Add pd), the adder will output the sum of Q(A) and Q(B):

- next_A = 30
- next_B = 8
- curr_A = 30
- curr_B = 8
- curr_sum = 38
- prev_sum = 0

→ The sum will not be sent to output Y until the next rising edge (plus clk-to-q delay) because curr_sum first has to go through that final register.

What happens in Cycle 1?

→ Cycle 1 begins on the next **Rising Edge** of the clock, when the new operands are sent in:

- next_A = 18
- next_B = 7
- curr_A = 30
- curr_B = 8
- curr_sum = 38
- prev_sum = 0

→ After **clk-to-q delay**:

- next_A = 18
- next_B = 18
- curr_A = 18
- curr_B = 7
- curr_sum = 38
- prev_sum = 38

(Now, the value of the first add operation is "available")

→ After **Adder pd**:

- next_A = 18
- next_B = 7
- curr_A = 18
- curr_B = 7
- curr_sum = 25
- prev_sum = 38

(the new sum is output by the adder)

What happens in Cycle 2?

→ On next **Rising Edge**:

- next_A = 7
- next_B = 7
- curr_A = 18
- curr_B = 7
- curr_sum = 25
- prev_sum = 38

→ After **clk-to-q** delay:

- next_A = 7
- next_B = 7
- curr_A = 7
- curr_B = 7
- curr_sum = 25
- prev_sum = 25

→ After **Adder pd**:

- next_A = 7
- next_B = 7
- curr_A = 7
- curr_B = 7
- curr_sum = 14
- prev_sum = 25

What happens in Cycle 3?

→ After the next rising edge & clk-to-q delay, the sum 7+7 = 14 will be sent to Y. i.e., prev_sum = 14.

How long is the clock cycle/period?

→ We can adjust the length of the clock cycle — aka time between one rising edge and the next — based on our needs! Its up to us.

→ There is a **minimum** clock period at which the circuit will work properly, so our clock period can be set to any value ≥ min. clock period.

What is the minimum clock period?

→ The most optimized (i.e. shortest) time within which the circuit can complete one operation.

- We don't include the clk-to-q delay of the final output register when calculating the min. clock period.

How do we calculate the min clock period in this example circuit?

→ Min clock period = clk-to-q delay of the input registers + adder propogation delay

↳ b/c the longest path has 1 adder

How do we calculate the min clock period in general?

→ **Min clock period** = clk-to-q delay + longest combinational delay

What is the "Longest combinational delay"?

→ The length (in time) of the **longest path between 2 registers**

- AKA longest combinational path

→ Basically, the amt of times it takes for each operation unit in a path between 2 registers

→ Does NOT include the clk-to-q delay time of the registers.

What is the max clock frequency?

→ 1 / min clock period

Pipelining

What is a single-cycle implementation?
→ An implementation of a circuit such that it completes one operation in a single clock cycle.

What is pipelining?
→ Breaking our circuit down into multiple stages that can operate simultaneously in order to make more efficient use of our hardware.

- We create "stages" by adding registers.

Example of a single-cycle circuit?

A, B, C, D → Reg → Adder → Adder → Reg → Y (clk)

→ This circuit performs $Y = A + B + C + D$

→ Let clk-to-q delay = 20ps & adder pd = 480 ps

Ex: What is the longest combinational delay?
→ The longest path btwn 2 registers has 2 adders, so $480 + 480 = 960$ps

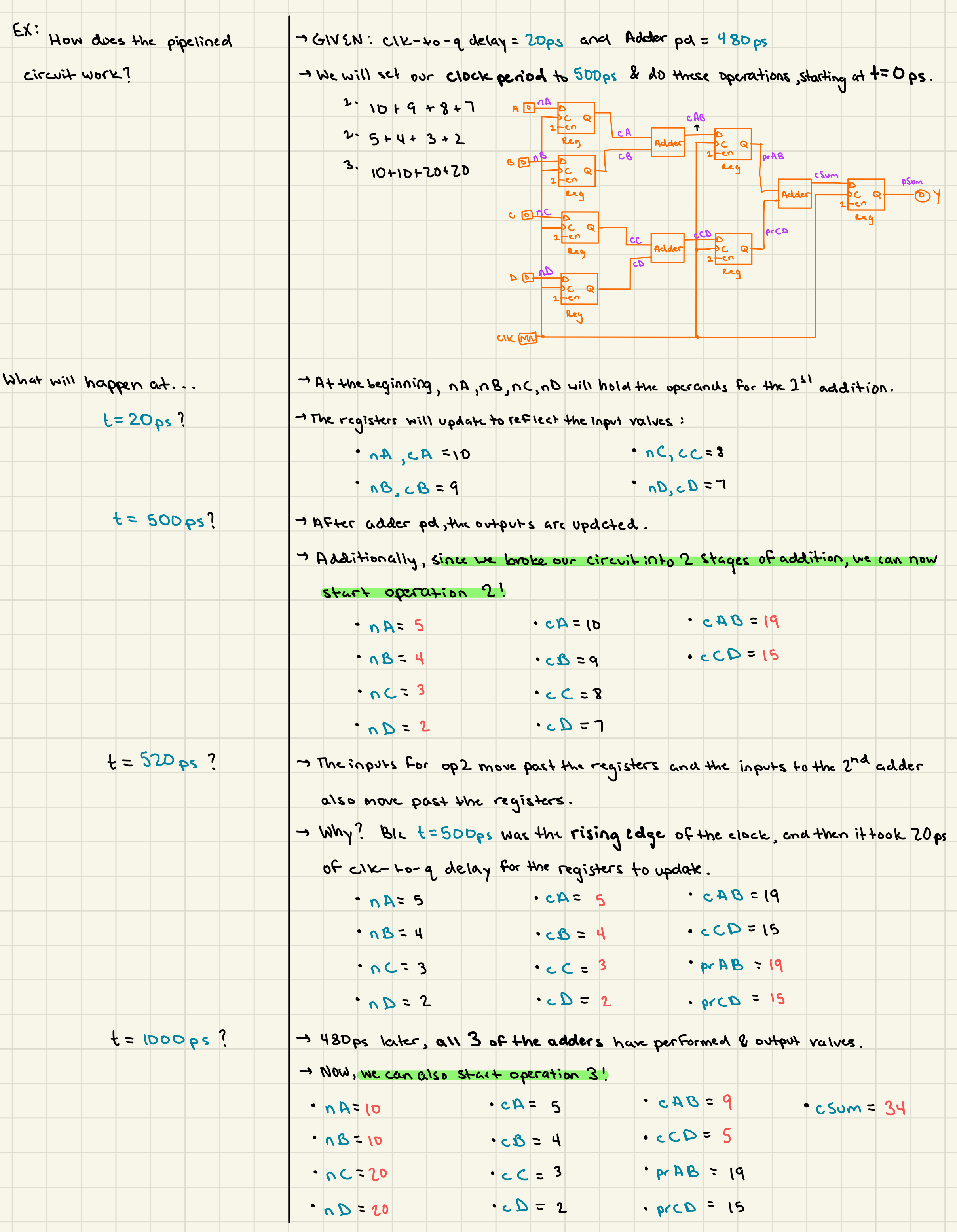
Ex: What is the min clock period?
→ 20ps (clk-to-q delay) + 960ps (LCD) = 980ps

→ If we run this circuit at the min clock period, it will take 980 ps (10) = 9800ps to complete 10 operations.

How do we implement pipelining on this circuit?

A, B, C, D → Reg → Adder → Reg → Adder → Reg → Y (clk)

How are these 2 circuits different?
→ The single-cycle impl. performs A+B and C+D simultaneously, then immediately does the sum of the 2.
→ The pipelined impl. performs A+B and C+D simultaneously, but then sends the outputs into 2 registers.



EX: How does the pipelined circuit work?

→ GIVEN: clk-to-q delay = 20ps and Adder pd = 480ps

→ We will set our clock period to 500ps & do these operations, starting at t = 0 ps.

1. 10 + 9 + 8 + 7
2. 5 + 4 + 3 + 2
3. 10 + 10 + 20 + 20

What will happen at...

t = 20ps ?

→ At the beginning, nA, nB, nC, nD will hold the operands for the 1st addition.

→ The registers will update to reflect the input values:

- nA, cA = 10
- nB, cB = 9
- nC, cC = 8
- nD, cD = 7

t = 500ps ?

→ After adder pd, the outputs are updated.

→ Additionally, since we broke our circuit into 2 stages of addition, we can now start operation 2!

- nA = 5
- nB = 4
- nC = 3
- nD = 2
- cA = 10
- cB = 9
- cC = 8
- cD = 7
- cAB = 19
- cCD = 15

t = 520ps ?

→ The inputs for op2 move past the registers and the inputs to the 2nd adder also move past the registers.

→ Why? B/c t = 500ps was the rising edge of the clock, and then it took 20ps of clk-to-q delay for the registers to update.

- nA = 5
- nB = 4
- nC = 3
- nD = 2
- cA = 5
- cB = 4
- cC = 3
- cD = 2
- cAB = 19
- cCD = 15
- prAB = 19
- prCD = 15

t = 1000ps ?

→ 480ps later, all 3 of the adders have performed & output values.

→ Now, we can also start operation 3!

- nA = 10
- nB = 10
- nC = 20
- nD = 20
- cA = 5
- cB = 4
- cC = 3
- cD = 2
- cAB = 9
- cCD = 5
- prAB = 19
- prCD = 15
- cSum = 34

t = 1020 ps ?

→ After the R.E. at t = 1000 ps, the registers update to hold the values of A, B, C, D for op.3, A+B and C+D for op.2, and A+B+C+D for op.1.

→ Operation 1 is now complete.

- nA = 10
- nB = 10
- nC = 20
- nD = 20
- cA = 10
- cB = 10
- cC = 20
- cD = 20
- cAB = 9
- cCD = 5
- prAB = 9
- prCD = 5
- cSum = 34
- pSum = 34

t = 1500 ps ?

→ All 3 adders perform addition operations & output values.

- nA = 0
- nB = 0
- nC = 0
- nD = 0
- cA = 10
- cB = 10
- cC = 20
- cD = 20
- cAB = 20
- cCD = 40
- prAB = 9
- prCD = 5
- cSum = 14
- pSum = 34

t = 1520 ps ?

→ The values of A+B and C+D for op.3 and A+B+C+D for op.2 get sent through the registers.

→ Operation 2 is now complete.

Ex: How many clock cycles did it take to complete 1 operation?

→ It took **2 clock cycles** (op.1 was done at 1000 ps, and our clock period was 500)

Ex: What is the longest combinational path/delay?

→ The longest circuit between 2 registers has 1 adder, so LCP = 480 ps

Ex: What is the min clock period?

→ 20 ps (clk-to-q delay) + 480 ps (LCD) = 500 ps

Ex: At the MCP, how long will it take to do 10 operations?

→ OP 1 finished at 1000 ps, but operations 2 and beyond will finish at 1500, 2000, 2500 ... etc. ps, so for 10 operations:

500 ps * (10+1) = 5,500 ps

Comparison of the single cycle & pipelined impls of Y = A+B+C+D ?

	Single Cycle	Pipelined
longest comb. path	960 ps	480 ps
min clock period	980 ps	500 ps
# of cycles to complete 1 op.	1	2
Time to complete 10 ops at MCP	9800 ps	5,500 ps

The pipelined circuit is much faster!

What are the execution times for both implementations?

→ To perform n operations:

- Single Cycle: (n)(clock period)
- Pipelined: (n+1)(clock period)

Why is the pipelined impl. usually faster?	→ Because it has a significantly smaller MCP. → n(x) v.s. (n+1)(x÷2) ... the 2nd expression will OFTEN yield a smaller value.

- Metrics -

	→ Let SC-Y denote the single-cycle example circuit performing Y=A+B+C+D and let PL-Y denote the pipelined example.
What is latency?	→ The amount of time it takes to complete a single operation, from beginning to end. • Ex: Latency of SC-Y = 1 clock cycle = 980 ps • Latency of PL-Y = 2 clock cycles = 2*500 = 1000 ps
What is throughput?	→ The number of operations that can be completed in a given amount of time. • PL-Y has a higher throughput than SC-Y.
What is speedup?	→ The measure of how much faster the pipelined impl. is in comparison to the single-cycle impl: SPEEDUP = $\frac{\text{Time to complete x ops on SC impl.}}{\text{Time to complete x ops on Pipelined impl.}}$ • Ex: The speedup of operation Y=A+B+C+D for 10 operations = 9800 ps / 5500 ps = 1.78.

311 Quiz 1 Review

→ TOPICS: K-maps – including "terms", Multiplexers, Digital Logic terms – Canonical SOP, maxterms, minterms, Adder, comparator, shifter, ALU, Waveform diagrams, timings on FF diagrams.

K-Maps

→ Find the literals that each group has in common, and OR them for the final term.

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	0	0	x
11	0	0	1	1
10	1	0	x	1

Grouping Rules:

- Each group must contain only 1s, and all 1s have to get grouped
- Groups must be 2^n cells (e.g. 1, 2, 4, 8, ...)
- Each group must be as large as possible
- If >1 groups, each group must have at least 1 unique cell.
- Ways to wrap around the table:

→ STEPS:

1. Create k-map
2. Include all essential prime implicants
3. Include all non-essential prime implicants as needed to cover all 1s.

Implicants

→ Implicant = any product/minterm in the SOP form of an equation.

- Ex: $Y = MN + MND + ND$ = implicant ... aka every square with a "1" in the k-map.

→ Prime Implicant = All possible groups that can be formed in a k-map. The p.i. itself is the term that defines a given group

→ Essential Prime Implicant = groups that cover at least one minterm that cannot be covered by another prime implicant.

Multiplexers

→ Chooses output based on a select signal:

→ If # of inputs = n, select signal S must be at least $\log_2(n)$ bits.

- aka, $2^{(\# \text{ of bits})} \geq n$

A → 00, B → 01, C → 10, D → 11, output Y, select S (2 bits)

- if S = 00, Y = A. if S = 01, Y = B, if S = 10, Y = C, if S = 11, Y = D.

Intro to MIPS

RECALL: What is COMP 311 about?

Converting:

high-level prog. languages

```
1  int square(int num) {
2      return num * num;
3  }
```

(Easy for humans to read/write)

→ into

Assembly, low-level languages

```
1  square(int):
2          addiu   $sp,$sp,-8
3          sw      $fp,4($sp)
4          move    $fp,$sp
5          sw      $4,8($fp)
6          lw      $2,8($fp)
7          nop
8          mult    $2,$2
9          mflo    $2
10         move    $sp,$fp
11         lw      $fp,4($sp)
12         addiu   $sp,$sp,8
13         jr      $31
14         nop
```

(human readable)

→ into

Machine Code

```
00100111101111011111111111111000
10101111101111100000000000000100
00000011101000001111000000100101
10101111110001000000000000001000
10101111110000100000000000001000
00000000000000000000000000000000
00000000010000100000000000011000
00000000000000000001000000010010
00000011110000000111010000100101
10001111101111100000000000000100
00100111101111010000000000001000
00000011111000000000000000001000
00000000000000000000000000000000
```

(Machine readable)

What is MIPS?

→ "<u>M</u>icroprocessor without <u>I</u>nterlocked <u>P</u>ipeline <u>S</u>tages"

→ A computer architecture & assembly language.

- Historically used in routers, embedded systems, gaming consoles (eg PlayStation)
- A little old/outdated today.

→ All assembly langs are very similar, but MIPS is the easiest to learn. Once you know MIPS, you can pick other langs up very easily.

What are some other computer architectures?

→ x86 : Developed by Intel, commonly used in servers, desktop & laptop computers.

- All Macbooks from 2005-2021 use x86_64 architecture on their 64-bit CPUs.

→ ARM: Used in most smartphones & tablets.

- All Macbooks after 2021 use the arm64 architecture.

What is an Instruction Set Architecture (ISA)?

→ A set of rules that define the interface between the hardware and software, providing a way for the sw to tell the hw what to do.

- A full vocabulary that combines <u>instructions</u> with registers, addressing models, and data types

→ Every ISA is specific to a processor architecture; for ex, the processor archtre assoc. w/ MIPS is RISC.

→ RECALL: COMP 211 notes: "Compilation System overview", "compiler step".

– ALU Unit –

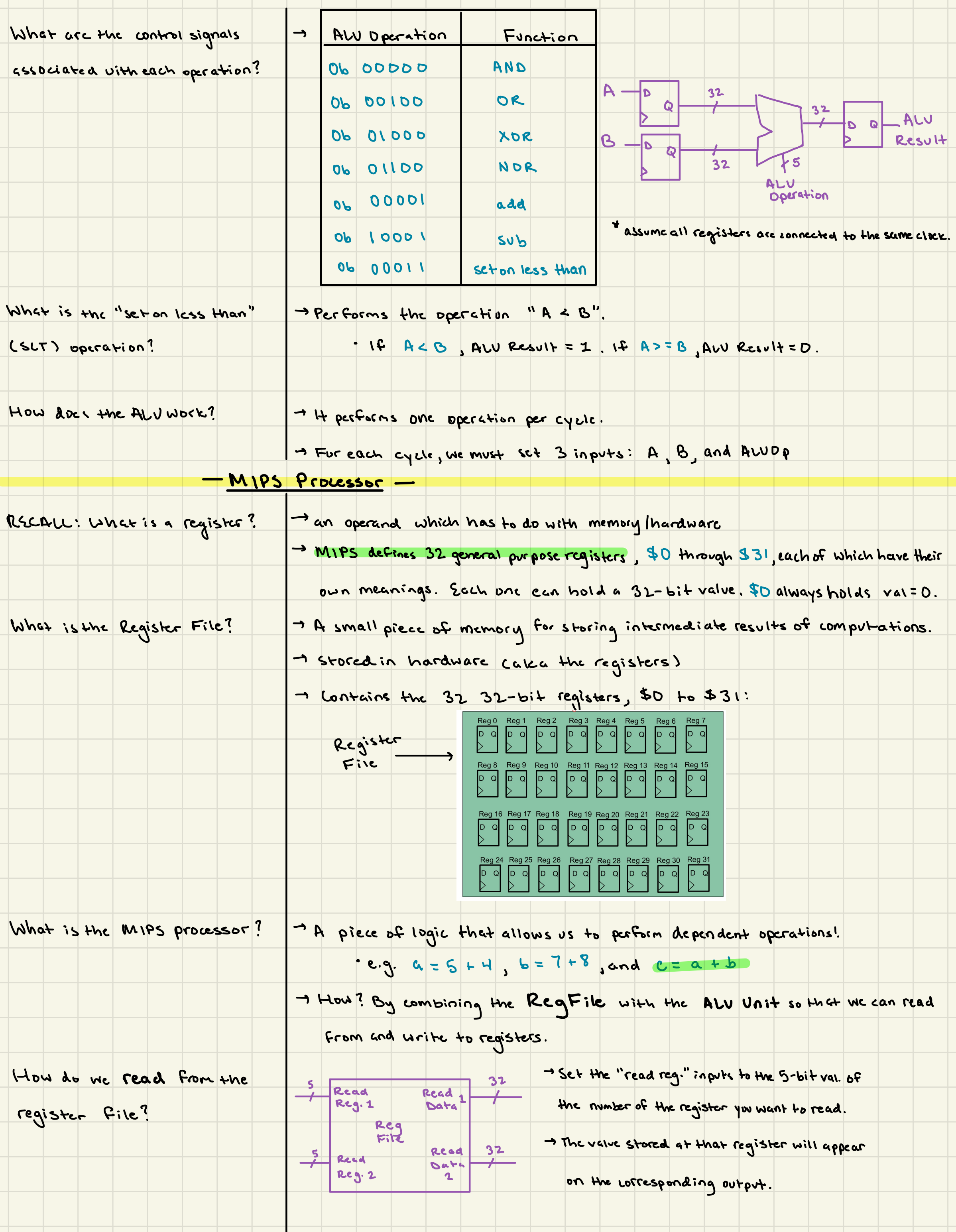
What is the ALU Unit in a MIPS processor?

→ A unit that performs ALL the ALU operations, on a 32-bit input.

- This is the thing that we built in Lab 2!

→ The ALU Unit employs a 5-bit ALU Op Control Signal Input that tells it which operation to perform

A (32), B (32) → ALU → ALU Result (32); ALU Op (5)



What are the control signals associated with each operation?

ALU Operation	Function
0b 00000	AND
0b 00100	OR
0b 01000	XOR
0b 01100	NOR
0b 00001	add
0b 10001	sub
0b 00011	set on less than

* assume all registers are connected to the same clock.

What is the "set on less than" (SLT) operation?

→ Performs the operation "A < B".

- If A < B, ALU Result = 1. If A >= B, ALU Result = 0.

How does the ALU work?

→ It performs one operation per cycle.

→ For each cycle, we must set 3 inputs: A, B, and ALUOp

— MIPS Processor —

RECALL: What is a register?

→ an operand which has to do with memory/hardware

→ MIPS defines 32 general purpose registers, \$0 through \$31, each of which have their own meanings. Each one can hold a 32-bit value. \$0 always holds val = 0.

What is the Register File?

→ A small piece of memory for storing intermediate results of computations.

→ stored in hardware (aka the registers)

→ Contains the 32 32-bit registers, \$0 to \$31:

What is the MIPS processor?

→ A piece of logic that allows us to perform dependent operations!

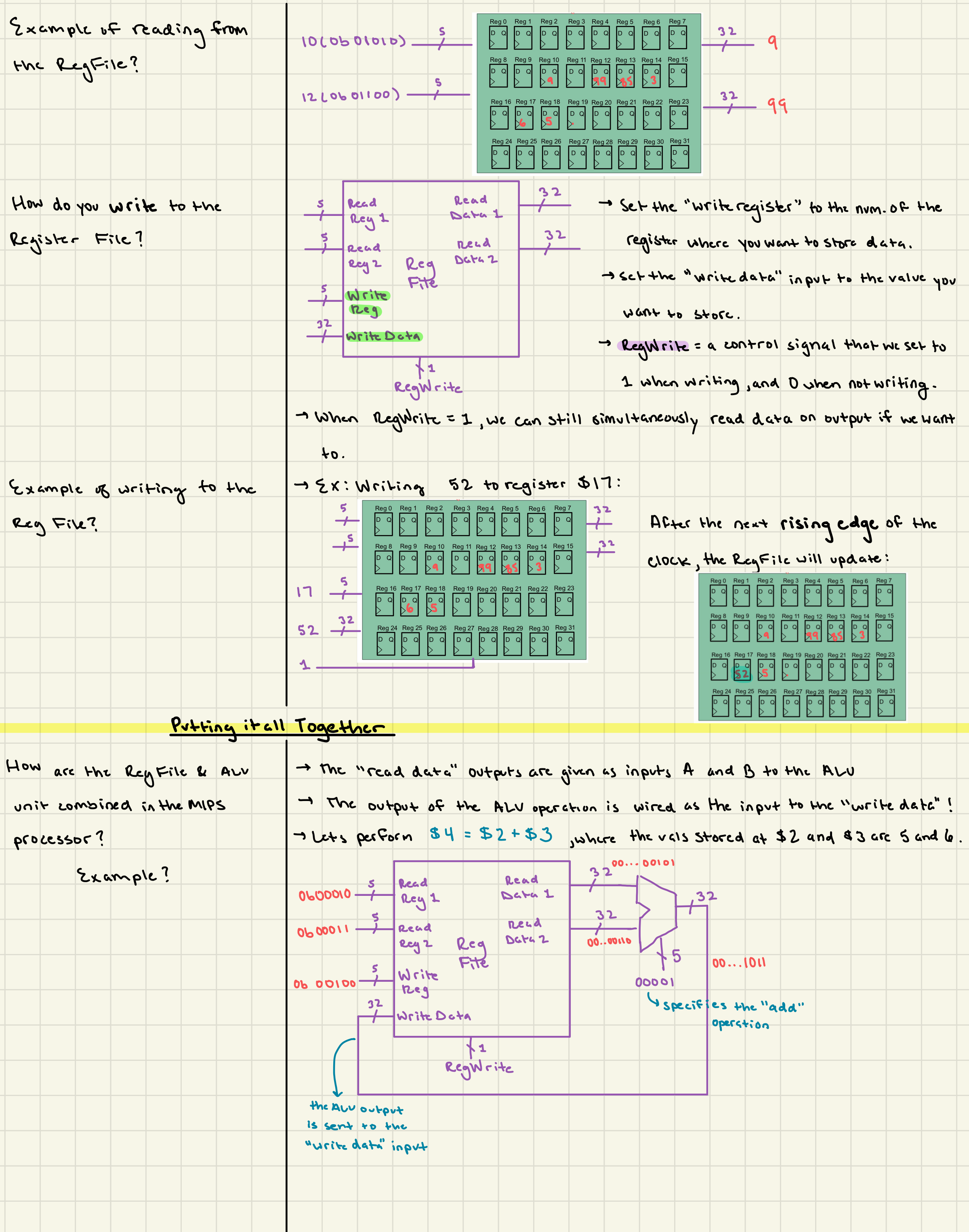
- e.g. a = 5 + 4, b = 7 + 8, and c = a + b

→ How? By combining the RegFile with the ALU Unit so that we can read from and write to registers.

How do we read from the register file?

→ Set the "read reg." inputs to the 5-bit val. of the number of the register you want to read.

→ The value stored at that register will appear on the corresponding output.



Example of reading from the RegFile?
10 (0b 01010)
12 (0b 01100)
5
32
9
99
How do you write to the Register File?
Read Reg 1
Read Reg 2
Write Reg
Write Data
Read Data 1
Read Data 2
Reg File
RegWrite
1
→ Set the "write register" to the num. of the register where you want to store data.
→ set the "write data" input to the value you want to store.
→ RegWrite = a control signal that we set to 1 when writing, and 0 when not writing.
→ When RegWrite = 1, we can still simultaneously read data on output if we want to.
Example of writing to the Reg File?
→ Ex: Writing 52 to register $17:
17
52
After the next rising edge of the clock, the RegFile will update:
Putting it all Together
How are the RegFile & ALU unit combined in the MIPS processor?
→ The "read data" outputs are given as inputs A and B to the ALU
→ The output of the ALU operation is wired as the input to the "write data"!
→ Lets perform $4 = $2 + $3 , where the vals stored at $2 and $3 are 5 and 6.
Example?
0b00010
0b 00011
0b 00100
00...00101
00..00110
00...1011
00001
↳ specifies the "add" operation
the ALU output is sent to the "write data" input

R-Format Instructions

RECAP: What are the inputs & outputs of a MIPS processor?

How is assembly code turned into binary code?

→ An assembly instruction can be translated into a 32-bit binary string that can then be input into the MIPS processor!

→ The 32-bit data is broken up into several "sections"

What are r-format instructions?

→ Assembly instructions to perform an operation where the registers are the operands.

- e.g., not adding a constant val, but rather 2 vals alr stored at 2 registers.

What is the syntax of r-format instructions?

[operation] [$rd], [$rs], [$rt]

1. "operation": the operation being performed (e.g. AND, add, etc.)
2. $rd: the destination register; where the output of the operation will be stored.
3. $rs, $rt: the source & target registers whose stored values will be the inputs to the operation (e.g. "A" and "B")

→ Ex: add $4, $2, $3

- add → the operation
- $4 → the destination operand
- $2, $3 → the source operands

What are the fields of the 32-bit binary instruction?

Field:	opcode	rs	rt	rd	shamt	funct
Bits:	31:26	25:21	20:16	15:11	6:10	5:0
	(6 bits)	(5 bits)	(5 bits)	(5 bits)	(5 bits)	(5 bits)

→ The 5-bit binary nums of the 3 registers in the operation (rs, rt, rd) are stored in the "rs", "rt", and "rd" fields.

What are the "opcode" and "funct" fields?

→ They are specified by the ISA, and they tell the processor which operation to perform.

→ The values at opcode & funct are used to determine the value of the ALUOp control signal.

What are the opcode & funct fields for common operations?

Instruction	Opcode	Funct	ALUOp:
add	000000_2	$10\,0000_2$	00001
sub	000000_2	$10\,0010_2$	10001
and	000000_2	$10\,0100_2$	00000
or	000000_2	$10\,0101_2$	00100
nor	000000_2	$10\,0111_2$	01100
slt	000000_2	$10\,1010_2$	00011

Example converting assembly to binary?

→ EXAMPLE: the instruction add $12, $7, $10

1) interpret the instruction and identify register fields:

- $12 = $7 + $10 (rd, rs, rt)

2) Fill out fields:

opcode	rs	rt	rd	shamt	funct
10 0000	00111	01010	01100	00000	100000

ANS: 0b 1000 0000 1110 1010 0110 0000 0010 0000

How is the 32-bit r-instruction interpreted by the MIPS processor?

→ The 32-bit inst. is taken as an input, and then split into substrings corresponding to the different fields!

field	bits of inst
funct	5:0
shamt	10:6
rd	15:11
rt	20:16
rs	25:21
opcode	31:26

Summary: What are all of the R-format operations?

→ Syntax: OP $rd, $rs, $rt , R[d] = R[rs] OP R[rd]

- R[rd] denotes "RegFile[rd]", aka, the contents stored at register rd.

NAME	OPERATION
add	R[rd] = R[rs] + R[rt]
sub	R[rd] = R[rs] − R[rt]
and	R[rd] = R[rs] AND R[rt]
or	R[rd] = R[rs] OR R[rt]
nor	R[rd] = R[rs] NOR R[rt]
slt	R[rd] = (R[rs] < R[rt]) ? 1 : 0

if R[rs] < R[rt], R[d] = 1. else, R[d] = 0

I-Format Instructions

What are i-format instructions?

→ Where the instruction **operands** are a combination of a register and a 16-bit **constant** (an "immediate" operand)

→ Used when you want to perform an op between a register and an immediate value (i.e. a constant that is not in a register)

- As opposed to r-type instructions, which perform operations between 2 registers.

What is the syntax of an i-format instruction?

(result stored here → $rt)

[operation] [$rt], [$rs], [immediate]

→ In i-format instructions, we need the last 16 bits of the inst. to be designated for storing the constant val, so we don't have an rd field. Instead, we store the result in the reg. specified by $rt. We write to $rt.

→ Ex: $5 = $4 + 12 ⟶ addi $5, $4, 12

→ Syntax: R[rt] = R[rs] OP (immediate)

How do we set a register to a certain constant value?

→ Using i-format instructions with the source operand register $0, since it always holds value = 0 !

→ Ex:

- a = 5 ⟶ addi $10, $0, 5 ($10 now holds var a)
- b = 3 ⟶ addi $11, $0, 3 ($11 holds var b)
- c = a + b ⟶ add $12, $10, $11 ($12 holds var c = a + b)

What are the fields of the 32-bit i-instruction?

Field:	opcode	rs	rt	immediate
Bits:	31:26	25:21	20:16	15:0
	(6 bits)	(5 bits)	(5 bits)	(16 bits)

→ The $rd, shamt, and funct fields are excluded to make room for the immediate value.

Example converting assembly → binary?

→ INST: addi $5, $0, 2

opcode	rs	rt	immediate
001000	00000	00101	0000 0000 0000 1100

- ANS: 06 0010 0000 0000 0101 0000 0000 0000 1100

What are the opcodes for i-format inst. operations?

Instruction	Opcode	ALU op
addi	$00\ 1000_2$	00001
andi	$00\ 1100_2$	00000
ori	$00\ 1101_2$	00100
slti	$00\ 1010_2$	00011

What will be input into the ALU for an i-instruction?

→ The 32-bit value stored at \$rs, aka R[rs], aka "Read Data 1"

→ The 16-bit value specified for the immediate, 15:0 ... EXCEPT, we need to bit-extend it to 32 bits!

- The ALU takes 2 32-bit inputs

→ There are 2 ways to extend the immediate value:

* Zero-extension
* Sign-extension

What is zero-extension and when should we use it?

→ DEFN: pad immediate with 16 zeroes at the beginning.

immediate (16) = $A_{15}\ A_{14}\ A_{13}, \dots\ A_0$

immediate (32) = $0000000000000000\,A_{15}\ A_{14}\ A_{13}, \dots\ A_0$

→ USE: For logical operations, where we do not want to preserve the decimal value of a binary # whose MSB=1 (aka a negative number)

- andi and ori operations will use a zero-extended immediate value.

What is sign extension and when should we use it?

→ DEFN: pad immediate with 16 zeroes if MSB is 0, or 16 ones if MSB = 1

immediate (16) = $A_{15}\ A_{14}\ A_{13}, \dots\ A_0$

immediate (32) =

$A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}A_{15}\ A_{15}\ A_{14}\ A_{13}, \dots\ A_0$

(where A_{15} = MSB)

→ USE: for arithmetic operations, where we want to preserve the decimal value.

- addi and slti will use a sign-extended immediate value.

What are all of the i-format operations?

NAME	OPERATION
addi	R[rt] = R[rs] + sign_ext (immediate)
andi	R[rt] = R[rs] AND zero_ext (immediate)
ori	R[rt] = R[rs] OR zero_ext (immediate)
slti	R[rt] = (R[rs] < sign_ext (immediate)) ? 1 : 0

How does the datapath to the MIPS Processor change for i-instructions?

→ OG Datapath for r-instructions:

→ If executing an i-instruction, there a 3 key differences:

- The 5-bit value denoting which register will store the result of the operation (aka the Write Reg) will be the \$rt instead of the \$rd.
- The 2nd 32-bit input to the ALU will be the immediate instead of R[rt], aka "Read Data 2"
- The 16-bit immediate value will have to be either 0- or sign-extended before being input to the ALU.

How do we modify the MIPS processor Hardware to execute both R and I instructions?

1) Add a 2:1 mux with a control signal that takes as inputs:

- The 5-bit value of $rd (aka bits 15:11, aka the write register for r-instructions), AND
- The 5-bit value of $rt (aka bits 20:16, aka the write register for i-instructions)

And uses a RegDst control signal to determine which value will get sent to the **Write Register** input.

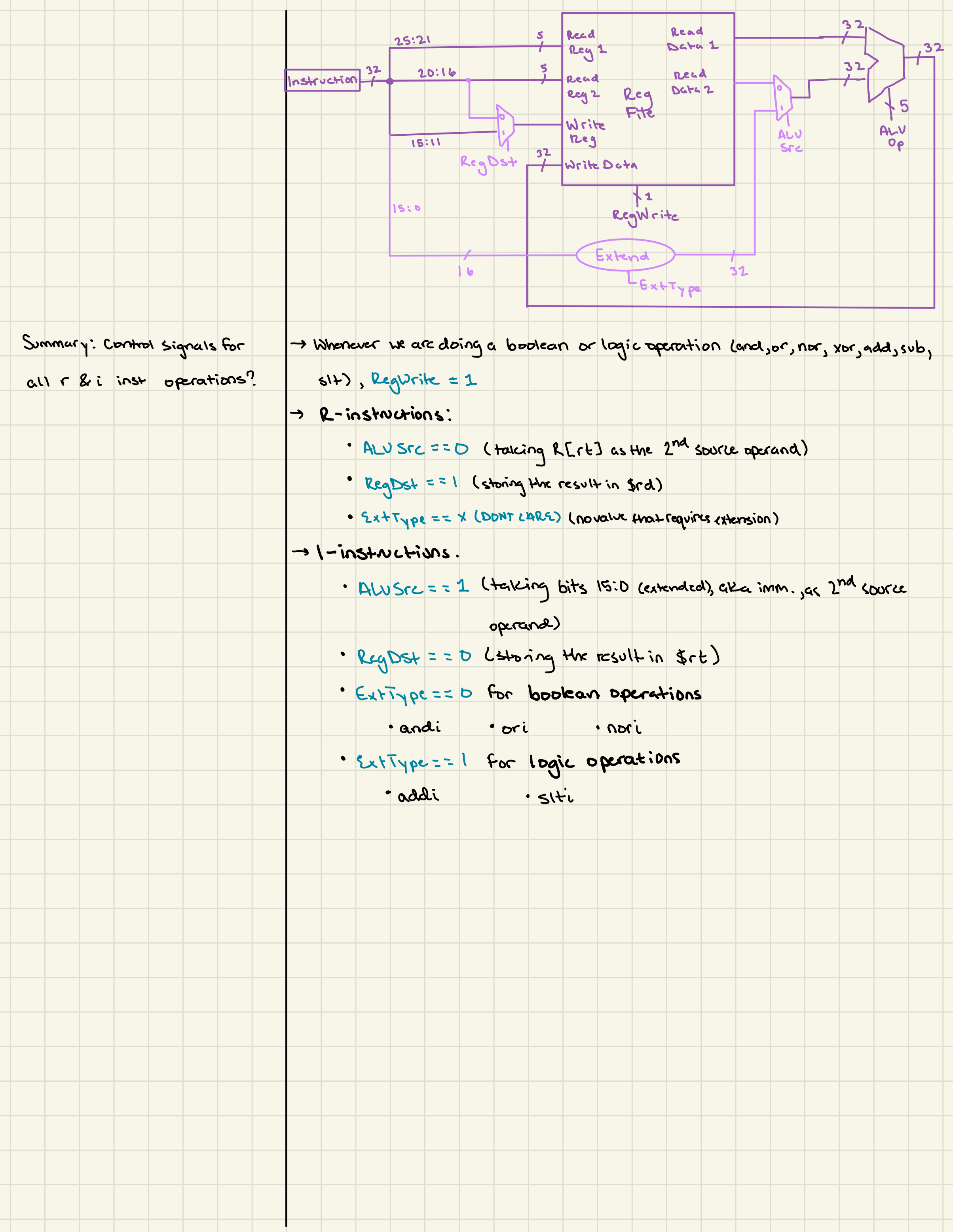
2) Add a 2:1 mux that takes as inputs:

- The 32-bit value R[rt] (aka "Read Data 2" aka the 2nd operand for r-instructions, AND
- The 32-bit value Extended_immediate (aka [extension + bits 15:0] aka the 2nd operand for i-instructions)

And uses an ALUSrc control signal to determine which value will get sent to the second **ALU** input.

3) A piece of logic that takes the immediate value (aka bits 15:0) as an input, and outputs the extended 32-bit imm. value.

- It has an ExtType control signal that specifies whether the value should be zero- or sign- extended.



Summary: Control signals for all r & i inst operations?

→ Whenever we are doing a boolean or logic operation (and, or, nor, xor, add, sub, slt), RegWrite = 1

→ R-instructions:

- ALU Src == 0 (taking R[rt] as the 2nd source operand)
- RegDst == 1 (storing the result in $rd)
- ExtType == x (DONT CARE) (no value that requires extension)

→ I-instructions.

- ALUSrc == 1 (taking bits 15:0 (extended), aka imm., as 2nd source operand)
- RegDst == 0 (storing the result in $rt)
- ExtType == 0 for boolean operations
  - andi
  - ori
  - nori
- ExtType == 1 for logic operations
  - addi
  - slti

MIPS Programming

What are Labels?

→ A way to "mark" sections of assembly code that you may want the program to be able to jump to or repeatedly execute. Like the beginning & end of a loop.
- Ex: For-loops.

Example?

→ loop.c:

```
sum = 0;
for (i=0; i<5; i++){
   sum += 2 }
sum += 10
```

→ loop.o
- We'll store sum and i in $10, $8
- We store the value representing the "terminating point" of i, aka 5, in $9

→ We enclose the instructions in loop.c's for-loop in a **Label**:

```
   addi $10 $0 0      } initialize sum, i, and terminating point
   addi $8 $0 0
   addi $9 $0 5
LoopStart:            → Label "LoopStart"
   addi $10 $10 2     → sum = sum + 2
   addi $8 $8 1       → i++ (i = i + 1)
LoopEnd:              → Label "LoopEnd"
   addi $10 $10 10    → sum = sum + 10
```

How do we actually utilize the labels?

→ With **branch instructions** that check whether the terminating condition (in this case $i \geq = 5$) has been reached!

What are branch instructions?

→ Instructions that evaluate some condition. If the condition is true, they tell the program to "branch" to a different spot in the code. Specifically, they define a **label** that should be branched to if the condition is true.
- Ex: If $i = 5$, end the for-loop & go to the label for the code to be executed after the loop ends.

What are all of the branch instructions?

Instruction	Format	Meaning: Branch to Label if...
Branch on equal	beq $rs $rt Label	$rs == $rt
Branch on not equal	bne $rs $rt Label	$rs != $rt
B.O greater than	bgt $rs $rt Label	$rs > $rt
B.O. greater than or equal	bge $rs $rt Label	$rs ≥ $rt
B.O. less than	blt $rs $rt Label	$rs < $rt
B.O. less than or equal	ble $rs $rt Label	$rs ≤ $rt

→ Note that for Branch instructions, **RegWrite = 0**!

Example loop.o using branch instructions?

```
addi $10 $0 0
addi $8 $0 0        → i = 0
addi $9 $0 5        → terminate pt = 5
LoopStart:
  beq $8 $9 LoopEnd → if R[$8] = R[$9] - aka, if i = 5, immediately "branch to" the LoopEnd label to continue executing inst. If not, continue to next inst. below.
  addi $10 $10 2
  addi $8 $8 1
  j LoopStart       → This is a jump instruction that tells the program to jump to the line labeled LoopStart; aka, jump back to beg. of loop after executing the body.
LoopEnd:
  addi $10 $10 10
```

What are **Native Instructions**?

→ Instructions that are supported by the datapath (the MIPS hardware diagram thing) AKA inst. that have actual hardware support.

- Includes: all the instructions we learned up until now
- Includes: beq, bne

What are **Pseudo Instructions**?

→ Ones that aren't supported by the datapath; they only exist to make programs easier to read & write.

→ When the program is assembled, pseudo instructions are converted to one or more native instructions.

- Includes: bgt, bge, blt, ble

Example of a pseudo instruction being converted to a native one?

```
...
ble $8 $9 LoopEnd
...
```

jump to "LoopEnd" if $8 ≤ $9

is converted to

```
...
slt $1 $9 $8
beq $1 $0 LoopEnd
...
```

slt: if $9 < $8, let $1 = 1. Else, let $1 = 0.

beq: if $1 = 0, aka $9 ≥ $8, branch to LoopEnd. Else, don't branch.

– Register Usage –

What is register 0 ($0) for?

→ ALWAYS holds the value 0. Cannot be written to.

→ Useful for when you need the value 0 (like initializing variables with addi $X $0 0)

What is register 1 ($1) for?

→ Used by the assembler to resolve pseudo instructions.

→ You CAN write to this register, but the assembler may override the value when it needs to resolve a pseudo-inst.

- Shouldn't rely on $1 to store values you might need later.

MIPS Memory Model

RECALL: What is the structure of a computer's main memory?

→ Data stored in a "primary storage" component like RAM, accessible to the CPU via a BUS (communication system between diff pieces of hardware)

- Unlike Register data, which is stored in the register file which is actually located in the CPU.

1. Stack: Used to manage function calls & local variables.
   - STACK grows down
2. Dynamic Data aka Heap: Used for dynamic memory allocation.
   - HEAP grows up
3. Static Data: variables allocated at compile-time.
   - e.g., statically allocated arrays.
4. Text: Programs (aka actual code).
5. Reserved memory for the OS

Stack[1] ↓ ↑ Dynamic Data[2]
Static Data[3]
Text[4]
Reserved[5]

higher memory addresses (top) — lower mem. addresses (bottom)

How does Memory compare to the Register File?

→ Memory:
- large
- takes longer to access
- NOT part of the CPU

→ Register File:
- small
- much, much, much faster to access
- part of the CPU

→ We need MM b/c we can't store all of our data in the RF

→ We store values/data that we are currently working with in the RF.

What is the load word instruction?

→ lw ; used to read 4 bytes (aka 1 word) of data from memory and store it in a register.

→ `lw $rt offset($rs)`

What do $rt, offset, and $rs mean in the lw inst.?

→ $rs : a register that (already) holds a memory address.

→ offset : an immediate value that is **added** to R[rs] to obtain the address

- e.g., lw $rt 4($rs) ≈ loading a word from 4 + [addr stored at $rs]

→ $rt : where we store the 4 byte (32-bit) data that we read from addr [offset + R[rs]]

Formula for the lw instruction?

→ R[$rt] = M[R[$rs] + offset], where "M" means Main Memory.

Example of the lw instruction?

→ Let $10 = 0x00004000, and let A = [5, 90, 100, 40, 11] be an int array stored at base addr. 0x00004000 (aka A[0]). The size of an int is 4 bytes.

→ Ex: `lw $11 8($10)` will store the value 100 in register $11.

- $10 holds 0x00004000 & offset = 8, so we want to read the data at address 0x00004000 + 8 = 0x00004008
- Since sizeof(int) = 4 bytes, 0x00004008 holds the 3rd element of A.

What is the store word instruction?

→ sw: Used to store 4 bytes (1 word) of data from the RF into memory!

→ `sw $rt offset($rs)`

What do $rt, offset, and $rs mean in the sw inst.?

→ $rs: A register that (already) holds a memory address.

→ offset: An immediate value that is added to R[rs] to obtain the address where we want to store the data.

→ $rt: Holds the data/value that we want to store at the MM address.

What is the formula?

→ M[R[$rs] + offset] = R[$rt]

Example using sw?

→ Let $10 = 0x00004000, $7 = 8, and let A = [5, 90, 100, 40, 11] be an int array stored at base addr. 0x00004000 (aka A[0]).

→ Ex: `sw $7 12($10)` will modify array A into:

A = [5, 90, 100, 8, 11]

How do we store arrays in memory?

→ We store statically allocated arrays in the static data segment (below the heap).

→ To store ANY data in the static data segment, we use the .data directive in our assembly code.

Example of using .data to store data in static memory?

```
.data
A: .word 5 90 100 40 11
```

- .data → .data directive
- A → The name of the array
- .word → Indicates the size of each element in the array. In this case, the size is 1 word, aka 4 bytes.

How do we actually get a memory address into a register?

→ E.g., the value stored in $rs for load-word & store-word instructions.

→ ANS: with the load address instruction, la.

What is the syntax of the la inst.?

`la $rt Label`

- Gets the start address of the array/variable specified by Label and stores it in $rt.
- For an array, la gets the start address of the array.

Example using la?

```
.data
A: .word 5 90 100
la $10, A
```

- "A" is the Label
- `la $10, A` → stores the base address of A in $10.

How do we store data in the text segment of memory?

→ Using the .text directive

→ We will store our actual assembly **code** in the text segment, using the .text directive.

Example of using the .text directive?

```
.data
A: .word 5 90 100 40 11      # Stored in the data segment

.text
la $10, A                    # Stored in the text segment; gets the start addr of A
lw $11, 0($10)               # gets the value of A[0]
```

Ex: What is an assembly program to increment every element in an array by 1?

→ Let Arr = [10, 15, 100, 2, 4]

ex1.o

```
.data
arr: .word 10 15 100 2 4
.text
la $10 arr                 # store base addr of arr in $10
addi $8 $10 20             # set $8 to be the "terminating point" of the loop; since there are 5 elements in arr, once we've incremented $10 4 times, we will have looped through all elements & are finished.
LoopBegin:
    beq $10 $8 LoopEnd     # end the loop (by jumping to "LoopEnd") when R[$10] = R[$8]
    lw $6 0($10)           # let $6 store value of element at addr $10
    addi $6 $6 1           # increment the element by 1
    sw $6 0($10)           # store the new value at $6 back in its spot (addr $10)
    addi $10 $10 4         # increment the "curr_address" by 4 to point to the next element in arr.
    j LoopBegin
LoopEnd:
```

What is the static instruction count?

→ The number of **native** instructions that a program has.

→ AKA, if you convert every pseudo-instruction into a native one, & then count up all the lines of assembly code written. Labels don't count.

→ Ex: ex1.o has 9 static instructions.

- every inst. is native except la
- la $6 arr resolves to → lui $1, 4097
  ori $10, $1, 0

What does the static inst. count tell us?

→ How much space the program takes up in memory.

Question	Notes
What is the dynamic instruction count?	→ The # of native instructions that actually get executed. → E.g., if there is a for-loop that has 5 inst. & 3 iterations, the dynamic inst. count (DIC) = 5×3 = 15 → Ex: ex1.o has 3 + 6(5) + 1 = 34 dynamic instructions.
What does the DIC tell us?	→ Gives us an idea of the runtime. → The DIC & SIC are 2 metrics to look at when trying to improve program efficiency.
How can we reduce the instruction count of ex1.o?	See the two versions of ex1.o below.

ex1.o

```
.data
 arr: .word 10 15 100 2 4
.text
 la $10 arr
 addi $8 $10 20
 LoopBegin:
   beq $10 $8 LoopEnd
   lw $6 0($10)
   addi $6 $6 1
   sw $6 0($10)
   addi $10 $10 4
   j LoopBegin
 LoopEnd:
```

beq $rs $rt Label is native

→

ex1.o

```
.data
 arr: .word 10 15 100 2 4
.text
 la $10 arr
 addi $8 $10 20
 LoopBegin:
   lw $8 0($6)
   addi $8 $8 1
   sw $8 0($6)
   addi $6 $6 4
   bne $6 $7 Loop Begin
```

the assembler always uses $1 for storing temp. values when resolving instructions.

Question	Notes
What are some pseudo instructions & their translations?	→ beq $rs imm Label → addi $1 $0 imm / beq $rs $1 Label → bge $8 $9 Label → slt $1 $8 $9 / beq $1 $0 Label ($1 will equal 1 if $8<9$, and 0 if $8 \geq 9$) ble $9 $14 Label → slt $1 $14 $9 / beq $1 $0 Label
What is the load immediate instruction?	→ Sets $rd to the immediate value: li $rd immediate R[rd] = immediate
What is the move instruction?	→ Copies the value of $rs into $rd: move $rd $rs R[rd] = R[rs]
Example using these?	→ addi $8 $0 4 → li $8 4 → addi $7 $6 0 → move $7 $6

Load Word and Store Word: Encoding & Hardware Support

RECALL: What are the "load word" and "store word" instructions?

→ LOAD WORD: lw $rt immediate ($rs)

- Store the data value at mem. address [$rs + imm] in $rt

→ STORE WORD: sw $rt immediate ($rs)

- Store the data inside $rt at mem. address [$rs + imm.]

How are lw and sw instructions encoded in binary?

→ Same as i-format!

Field:	opcode	rs	rt	immediate
Bits:	31:26	25:21	20:16	15:0
	(6 bits)	(5 bits)	(5 bits)	(16 bits)

- lw : opcode = 100011
- sw : opcode = 101011

What "operation" is being done for lw and sw?

→ Computing the address, aka R[rs] + SignExtend(immediate)

- We can use the ALU for this!
- Since it is addition, always sign extend the immediate.

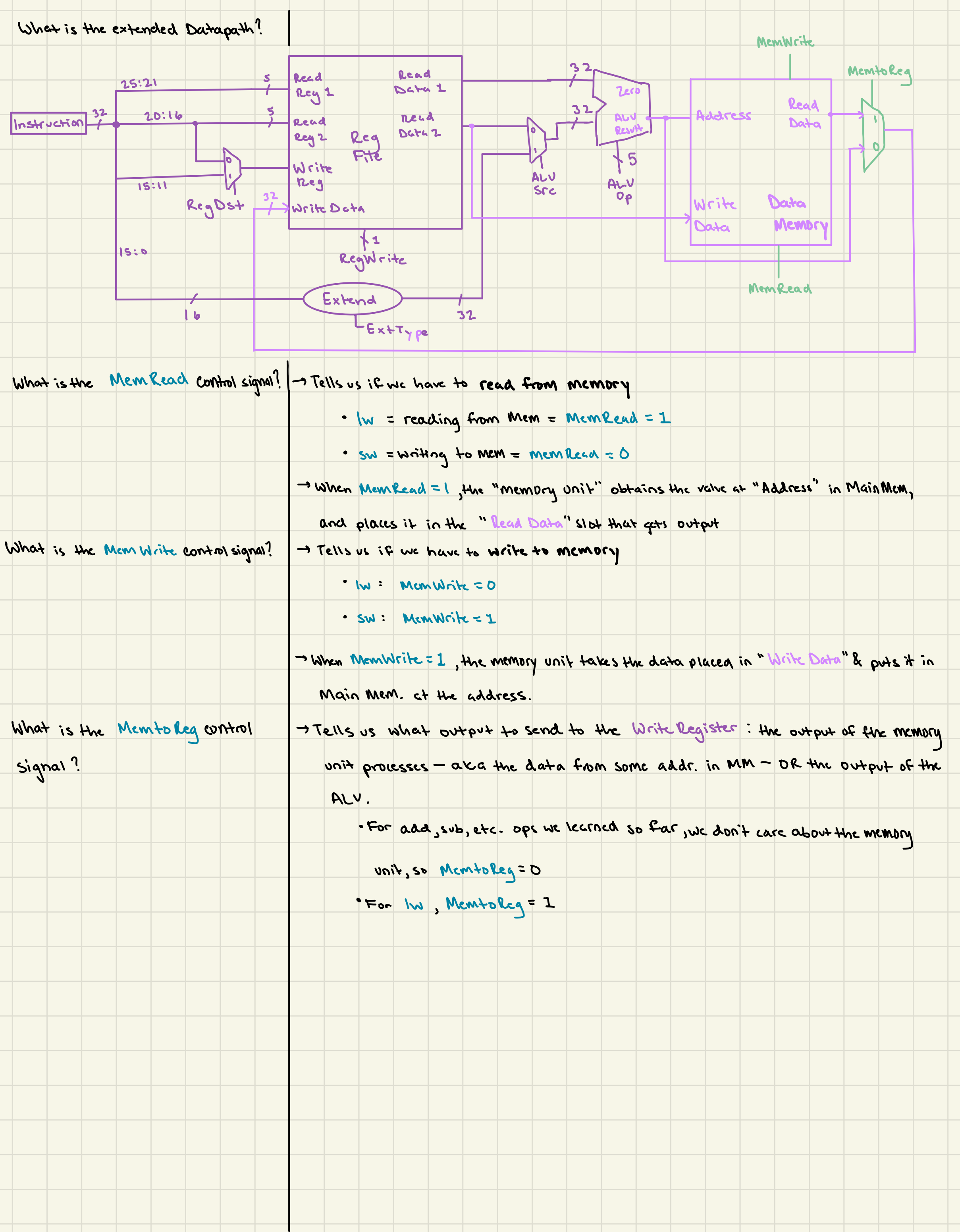
How do we extend the Datapath to support lw and sw?

Instruction
32
25:21
20:16
value of $rt
15:11
15:0
5
5
RegDst
32
Read Reg 1
Read Reg 2
Write Reg
Write Data
Reg File
Read Data 1
Read Data 2
1
RegWrite
16
Extend
ExtType
32
ALU Src
32
32
5
ALU Op
32
Result of R[rs] + imm.
Result of R[rt]

→ From the current DP, we can obtain 3 things:

- R[rt], aka the value to place in memory for sw instructions
- $rt, aka the register to store the mem. data for lw instructions
- R[rs] + imm, aka the memory address in question

→ To perform operations involving memory, we add a new component & new control signals!



What is the extended Datapath?

What is the MemRead control signal?

→ Tells us if we have to read from memory

- lw = reading from Mem = MemRead = 1
- sw = writing to mem = memRead = 0

→ When MemRead = 1, the "memory unit" obtains the value at "Address" in MainMem, and places it in the "Read Data" slot that gets output

What is the MemWrite control signal?

→ Tells us if we have to write to memory

- lw : MemWrite = 0
- sw : MemWrite = 1

→ When MemWrite = 1, the memory unit takes the data placed in "Write Data" & puts it in Main Mem. at the address.

What is the MemtoReg control signal?

→ Tells us what output to send to the Write Register : the output of the memory unit processes — aka the data from some addr. in MM — OR the output of the ALU.

- For add, sub, etc. ops we learned so far, we don't care about the memory unit, so MemtoReg = 0
- For lw, MemtoReg = 1

The Program Counter (PC)

What is the Stored Program Concept?

→ RECALL: Instructions (like add $8 $7 $6) are stored in memory as 32-bit binary numbers.

→ Since inst. are 32 bits, each take up 4 bytes in memory. In the Text section.

→ The instructions for a given program are stored in subsequent memory addresses that increment by 4.

How does the code that we write get "resolved" for execution?

1) Pseudo-instructions are converted into native ones

2) Addresses for labels and offsets get resolved.

Example of the stored program concept?

Address in memory where the inst. is stored → Address
Machine code representation of the inst. → Code
Assembled program (code that is actually executed) → Basic
og program (the code that you wrote) → Source

Address	Code	Basic	Source
0x00003000	0x20060000	addi $6,$0,0	la $6 arr
0x00003004	0x20c70014	addi $7,$6,20	addi $7,$6,20
0x00003008	0x8cc80000	lw $8,0($6)	lw $8,0($6)
0x0000300c	0x21080001	addi $8,$8,1	addi $8,$8,1
0x00003010	0xacc80000	sw $8,0($6)	sw $8,0($6)
0x00003014	0x20c60004	addi $6,$6,4	addi $6,$6,4
0x00003018	0x14c7fffb	bne $6,$7,-5	bne $6 $7 LoopBegin

notice that addrs increment by 4

What is the program counter (PC)?

→ A register that holds the address of the instruction being currently executed!
- Holds the 32-bit address in a 32-bit register

→ The PC is not inside the register file.

→ For ex, when executing addi $8 $8 1 from the prog. above, PC = 0x0000300c

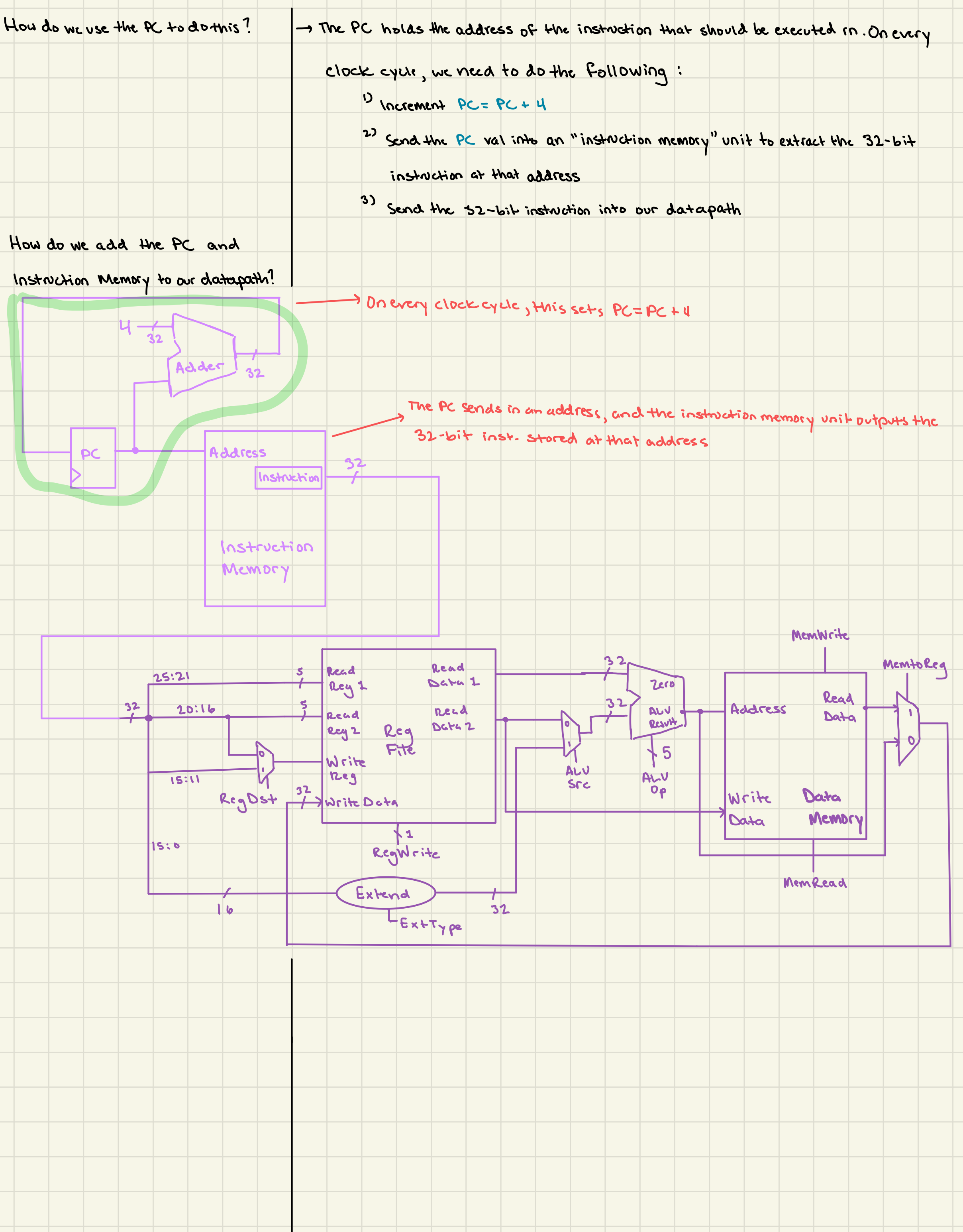
How is the value of the PC set?

→ Our datapath executes one instruction per clock cycle. So on every clock cycle, the datapath will increment the PC value by 4 to fetch the next instruction.

WAIT so... what does this have to do with anything?

→ The PC is what automates the process of sending 32-bit instructions as inputs to our data path!

**


How do we use the PC to do this?
→ The PC holds the address of the instruction that should be executed rn. On every clock cycle, we need to do the following:
1) Increment PC = PC + 4
2) Send the PC val into an "instruction memory" unit to extract the 32-bit instruction at that address
3) Send the 32-bit instruction into our datapath
How do we add the PC and Instruction Memory to our datapath?
On every clock cycle, this sets PC = PC + 4
The PC sends in an address, and the instruction memory unit outputs the 32-bit inst. stored at that address
4
32
Adder
32
PC
Address
Instruction
32
Instruction Memory
25:21
5
Read Reg 1
Read Data 1
32
20:16
5
Read reg 2
Reg File
Read Data 2
Write Reg
15:11
RegDst
32
Write Data
1
RegWrite
15:0
Extend
16
32
ExtType
32
Zero
32
ALU Result
ALU Src
5
ALU Op
MemWrite
Address
Read Data
Write Data
Data Memory
MemRead
MemtoReg

Branch Instruction Encoding

How do we encode branch instructions in binary?

→ In i-format! `beq rs rt Label`

Instruction	Opcode
beq	000100
bne	000101

→ For Ex, beq \$10 \$11 LoopBegin

Field:	opcode	rs	rt	immediate
Bits:	000100	01010	01011	???

What is the target instruction?

→ The instruction that the program will jump to if the **branch** is taken

What is the "byte offset to target instruction"?

→ The distance, in bytes, between the current branch instruction itself, and the target instruction.

→ The byte offset will always be a multiple of 4, which means that the last 2 bits of the byte offset will always be 0. Because 0x0, 4, 8, C, etc. all have 00 at the end.

→ We actually calculate the byte offset relative to the **next instruction**, e.g. PC+4.

How do you generate the value of the immediate to encode a branch instruction?

1. Obtain the value of PC+4 - aka, the next inst. below the branch instruction, b/c the PC is currently on the branch instruction.
2. Compute the **byte offset** between PC+4 and the target instruction (the inst. that we would branch to)
   - AKA, (target addr) − (PC+4)
3. Since each inst. are 4 addresses away from each other, the byte offset is always a multiple of 4, and its last 2 bits are 0. So we remove the last 2 bits.

immediate = (target addr − (PC+4)) >> 2

Ex: beq \$4 \$5 elseIf

```
PC     → beq  $4 $5 elseIf
PC+4   → add  $7 $8 $9      ] 4
         add  $7 $8 $20     ] 4
         j exit             ] 4
elseIf:
target → bne  $4 $6 else
         add  $7 $10 $11
         add  $7 $7 $12
         add  $7 $7 $8
         j exit
else:
         add  $7 $8 $11
exit:
```

ANS = 0000 0000 0000 0100 (remove these — the last 2 bits)

How do we compute the branch target address given the immediate?

→ Ex: The instruction 0x15320007 translates to:

opcode	rs	rt	immediate
000101	01001	10010	0000 0000 0000 0111

→ Assume the current PC = 0x0000301C

1. **Extract the immediate value**: 0b 0000 0000 0000 0111
2. **Sign-extend it to 30 bits**: 0b 00 0000 0000 0000 0000 0000 0000 0111
3. **Append 2 0s to the end of it**: 0b 0000 0000 0000 0000 0000 0000 0001 1100
4. **Add PC + 4 to it**:

```
    0x 0000001C
  + 0x 0000301C
  + 0x 00000004
  -------------
    0x 0000303C   ← The branch target address
```

Branch Target Addr = PC + 4 + {30-bit sign-ext imm., 0b00}

(this syntax means concatenate)

– Datapath Support for Branching –

How do we obtain the branch target address?

→ RECALL: to generate the branch target address:

1. Sign Extend the imm. value to 30 bits
2. Append 2 0s
3. Add it to PC+4

→ We can achieve step 2 by **left-shifting** the 30-bit imm. by 2!

How do we do this on the datapath?

What do we do if we shouldn't branch upon a branch inst.?

→ The branch target addr. = the value we should set PC to **IF we want to branch.**

→ If not, set PC = PC+4, like usual

→ **We will need a mux to make this decision:**

How will we set the control signal for the mux?

→ Using a piece of logic that interprets the beq and bne branch instructions.

→ This logic will output 1 to the Mux if we should branch, and 0 otherwise.

How do we support beq & bne on the datapath?

→ Using 2 new 1-bit control signals to indicate them:

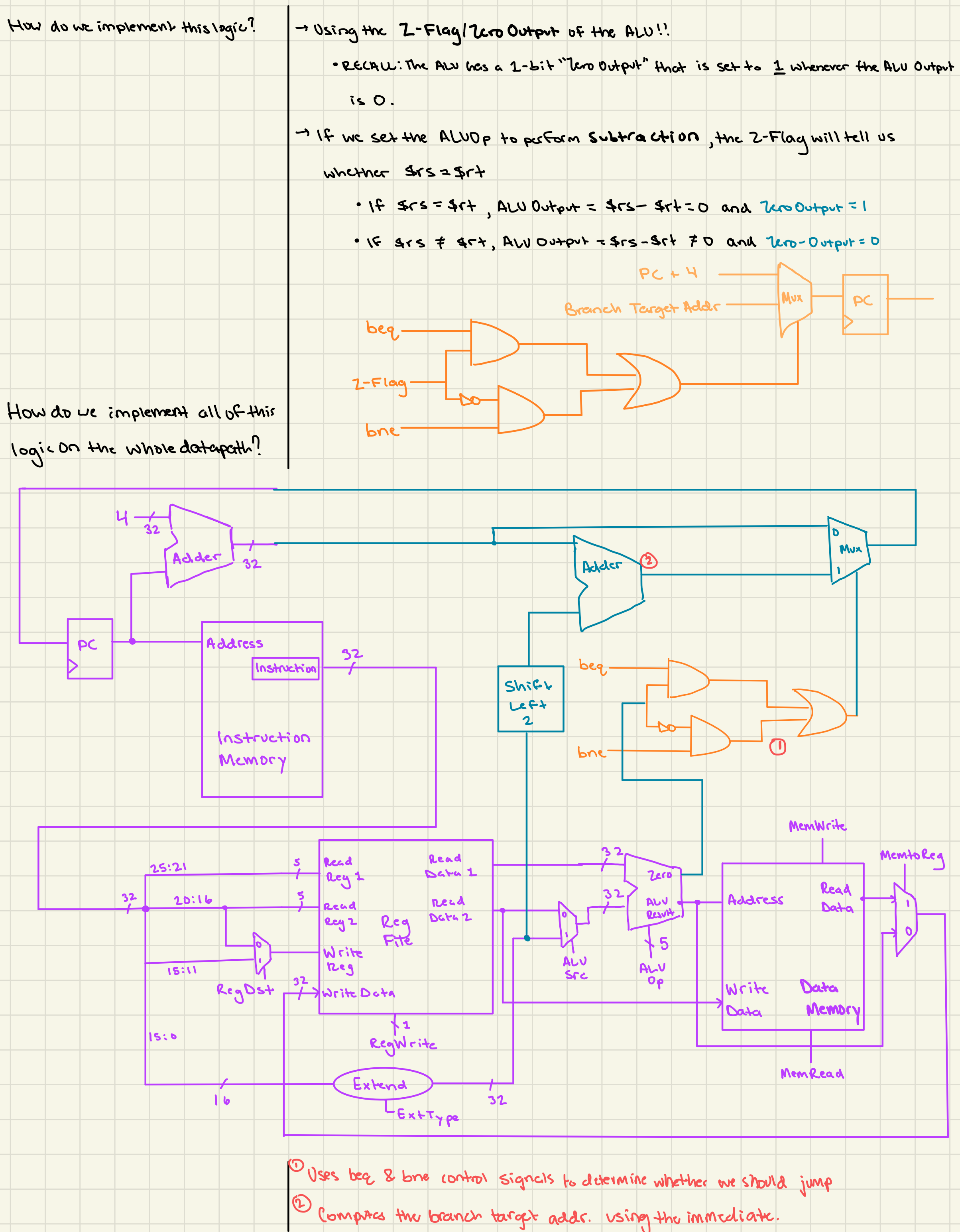
- beq : 1 if inst. is beq, 0 otherwise
- bne : 1 if inst. is bne, 0 otherwise

→ beq \$rs \$rt Label : if \$rs = \$rt, we want to branch.

→ bne \$rs \$rt Label : if \$rs ≠ \$rt, we want to branch.

→ SUMMARY: Output 1 to the Mux control signal IF:

- beq = 1 AND \$rs = \$rt OR
- bne = 1 AND \$rs ≠ \$rt


How do we implement this logic?
→ Using the Z-Flag/Zero Output of the ALU!!
• RECALL: The ALU has a 1-bit "Zero Output" that is set to 1 whenever the ALU Output is 0.
→ If we set the ALUOp to perform subtraction, the Z-Flag will tell us whether $rs = $rt
• If $rs = $rt, ALU Output = $rs − $rt = 0 and Zero Output = 1
• If $rs ≠ $rt, ALU Output = $rs − $rt ≠ 0 and Zero-Output = 0
PC + 4
Branch Target Addr
Mux
PC
beq
Z-Flag
bne
How do we implement all of this logic on the whole datapath?
4
32
Adder
32
PC
Address
Instruction
Instruction Memory
32
Adder
②
Mux
0
1
Shift Left 2
beq
bne
①
MemWrite
MemtoReg
25:21
5
Read Reg 1
Read Data 1
32
20:16
5
Read reg 2
Read Data 2
Reg File
32
Zero
ALU Result
Address
Read Data
Write Reg
15:11
RegDst
32
Write Data
ALU Src
5
ALU Op
Write Data
Data Memory
1
RegWrite
15:0
MemRead
Extend
16
32
ExtType
① Uses beq & bne control signals to determine whether we should jump
② Computes the branch target addr. using the immediate.

Jump Instructions

RECALL: What are jump instructions?

→ An instruction that tells the program to jump to an address (specified by the Label) to continue execution.

`j Label`

• aka, set PC to the address of Label

How are jump instructions encoded in binary?

→ In a new instruction format called **j-Format**:

opcode = 000010

opcode (31–26)	partial address (PA) (25–0)

What is the problem?

→ We can only fit a 26-bit address in the instruction, but the PC needs a 32-bit address.

How can we reduce the size of the address to 30 bits?

→ We can chop off/not store the last 2 bits, since the last 2 bits of any instruction address are always 0 — due to word alignment:

0b0 .... 0000 (0)
0b0 ... 0100 (4)
0b0 ... 1000 (8)
0b0 ... 1100 (12)
0b0 ... 11100 (24)

How do we get rid of the last 4 extra bits?

→ By limiting the range of address values we can jump to

→ When setting the PA field of a jump instruction, we will chop off the top 4 bits of the target address.

→ When constructing the new PC addr given the PA field of a jump instruction, we will set the top 4 bits to be the top 4 bits of (PC + 4).

Why do we do this?

→ Setting the top 4 bits to be those of (PC+4) gives us an address range that is nearby to where we are currently located.

• To jump outside this range, we'd have to use a different instruction.

What is the JumpAddress given a binary j-instruction?

JumpAddr = { (PC+4)[31:28] , partial_address , 0b00 }

(bits 31:28 of PC+4; bits 25:0 of j-inst)

Given a program with a jump inst., do we compute the binary encoding of the partial address?

→ Ex, where the program begins at address 0x00003000:

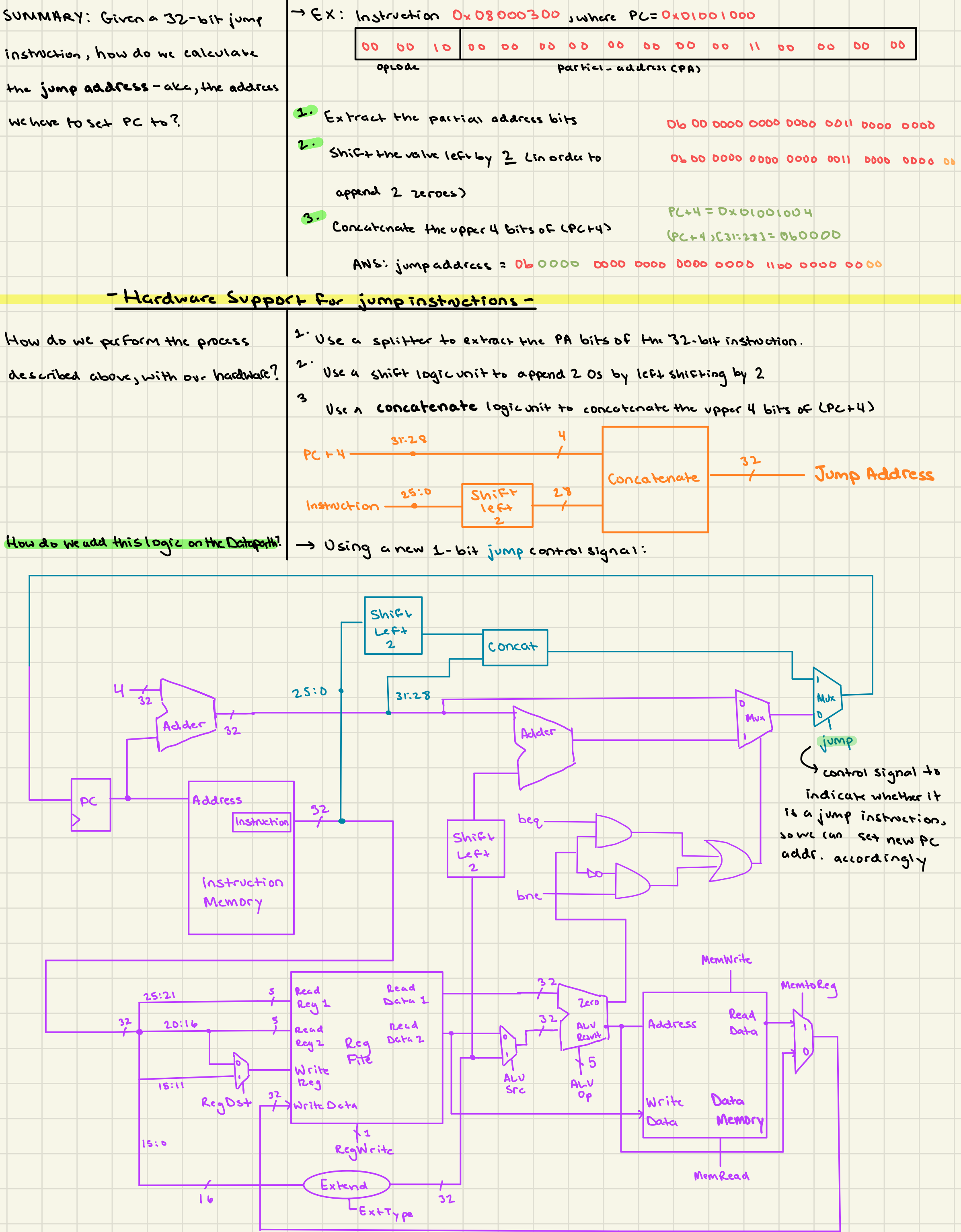
```
1  .data
2
3  A:          .word 8, 9, 14, 15
4
5  .text
6
7          la $5 A       # address of A           0x00003000
8          addi $6 $0 0  # sum                    0x00003004
9          addi $7 $0 4  # size of array          0x00003008
10         addi $8 $0 0  # i: loop counter        0x0000300C
11
12 LoopBegin:
13         beq $8 $7 LoopEnd # exit loop when we reach the end of the array   0x00003010
14         sll $9 $8 2       # compute byte offset          0x00003014
15         add $9 $9 $5      # compute address of A[i]      0x00003018
16         lw $9 0($9)       # get the value of A[i]        0x0000301C
17         add $6 $6 $9      # sum += A[i]                  0x00003020
18         addi $8 $8 1      # i++                          0x00003024
19         j LoopBegin       # jump back to start of array  0x00003028
20 LoopEnd:
```

(Address column header: Address:)

1. Determine the address we want to jump to:
   • The address of the first inst. after the Label ⟶ 0x00003010
2. Convert it to binary: 0b0000 0000 0000 0000 0011 0000 0001 0000
3. Remove top 4 bits & bottom 2 bits: 0b0000 0000 0000 0000 0011 0000 0001 00

ANS:

000010 (31–26)	0000 0000 0000 0011 0000 0001 00 (25–0)



SUMMARY: Given a 32-bit jump instruction, how do we calculate the jump address - aka, the address where to set PC to?

→ EX: Instruction 0x08000300, where PC = 0x01001000

00	00	10	00	00	00	00	00	00	00	00	11	00	00	00	00

opcode | partial-address (PA)

1. Extract the partial address bits — 0b 00 0000 0000 0000 0011 0000 0000
2. Shift the value left by 2 (in order to append 2 zeroes) — 0b 00 0000 0000 0000 0011 0000 0000 00
3. Concatenate the upper 4 bits of (PC+4) — PC+4 = 0x01001004; (PC+4)[31:28] = 0b0000

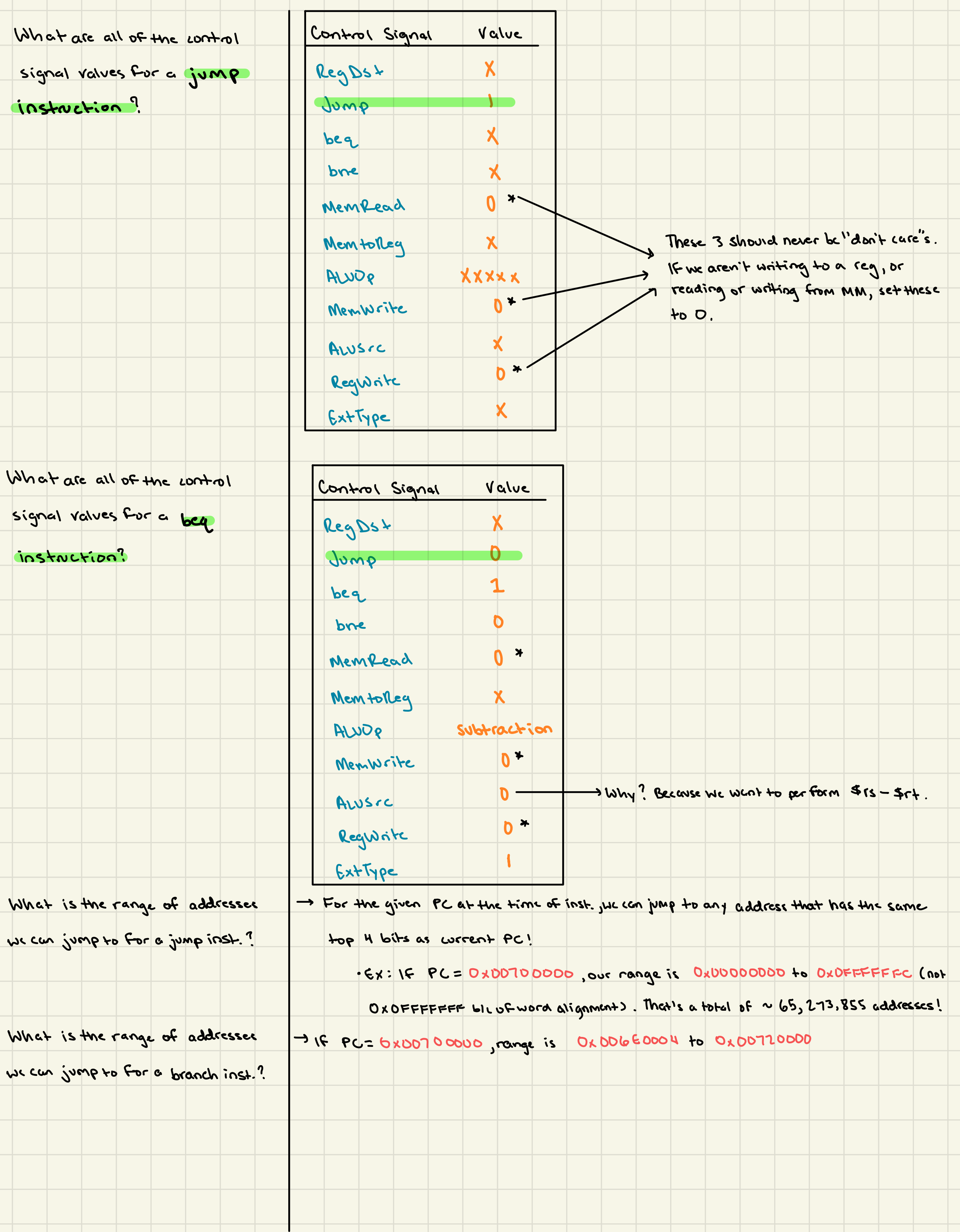
ANS: jump address = 0b 0000 0000 0000 0000 0000 1100 0000 0000

- Hardware Support for jump instructions -

How do we perform the process described above, with our hardware?

1. Use a splitter to extract the PA bits of the 32-bit instruction.
2. Use a shift logic unit to append 2 0s by left shifting by 2
3. Use a **concatenate** logic unit to concatenate the upper 4 bits of (PC+4)

How do we add this logic on the Datapath? → Using a new 1-bit jump control signal:



What are all of the control signal values for a **jump instruction**?

Control Signal	Value
RegDst	X
Jump	1
beq	X
bne	X
MemRead	0 *
MemtoReg	X
ALUOp	XXXX
MemWrite	0 *
ALUSrc	X
RegWrite	0 *
ExtType	X

These 3 should never be "don't care"s. If we aren't writing to a reg, or reading or writing from MM, set these to 0.

What are all of the control signal values for a **beq instruction**?

Control Signal	Value
RegDst	X
Jump	0
beq	1
bne	0
MemRead	0 *
MemtoReg	X
ALUOp	subtraction
MemWrite	0 *
ALUSrc	0
RegWrite	0 *
ExtType	1

ALUSrc 0 → Why? Because we want to perform $rs − $rt.

What is the range of addresses we can jump to for a jump inst.?

→ For the given PC at the time of inst., we can jump to any address that has the same top 4 bits as current PC!

- Ex: If PC = 0x00700000, our range is 0x00000000 to 0x0FFFFFFC (not 0x0FFFFFFF b/c of word alignment). That's a total of ~ 65,273,855 addresses!

What is the range of addresses we can jump to for a branch inst.?

→ If PC = 0x00700000, range is 0x006E0004 to 0x00720000

Function Calls

What is the calling convention?

→ A scheme for how functions receive arguments & return values

→ **Caller** : A function that calls another function

→ **Callee** : A function called by another function

What is the protocol for callers and callees?

→ Before making the function call, the caller places the parameter args in registers $a0 - $a3 (4-7)

→ When finished, the callee places the return values in registers $v0 - $v1 (2-3)

How does the callee know where to return when it finishes execution?

→ Return Address : the address of the inst. where the callee will return.

→ When the caller calls the callee, they place the ret. addr. in a special register ($ra = $31) so the callee knows where to return.

So how does a caller actually call a function?

→ Using the jump-and-link (JAL) instruction! To jump to a function.

→ Calling a function means 2 things on the 'hardware' side:

1. Setting PC to the address of the function being called
2. Setting reg $ra to the address we want the program to resume executing at after the function is over.

What does the JAL instruction do?

→ Syntax: `jal Label`

→ Sets the PC to be the address of "Label" – aka the callee's label

→ AND automatically sets register $ra to be the return address

- return addr = address of the inst. in main right after the JAL instruction

How does the callee return execution to the caller when its done?

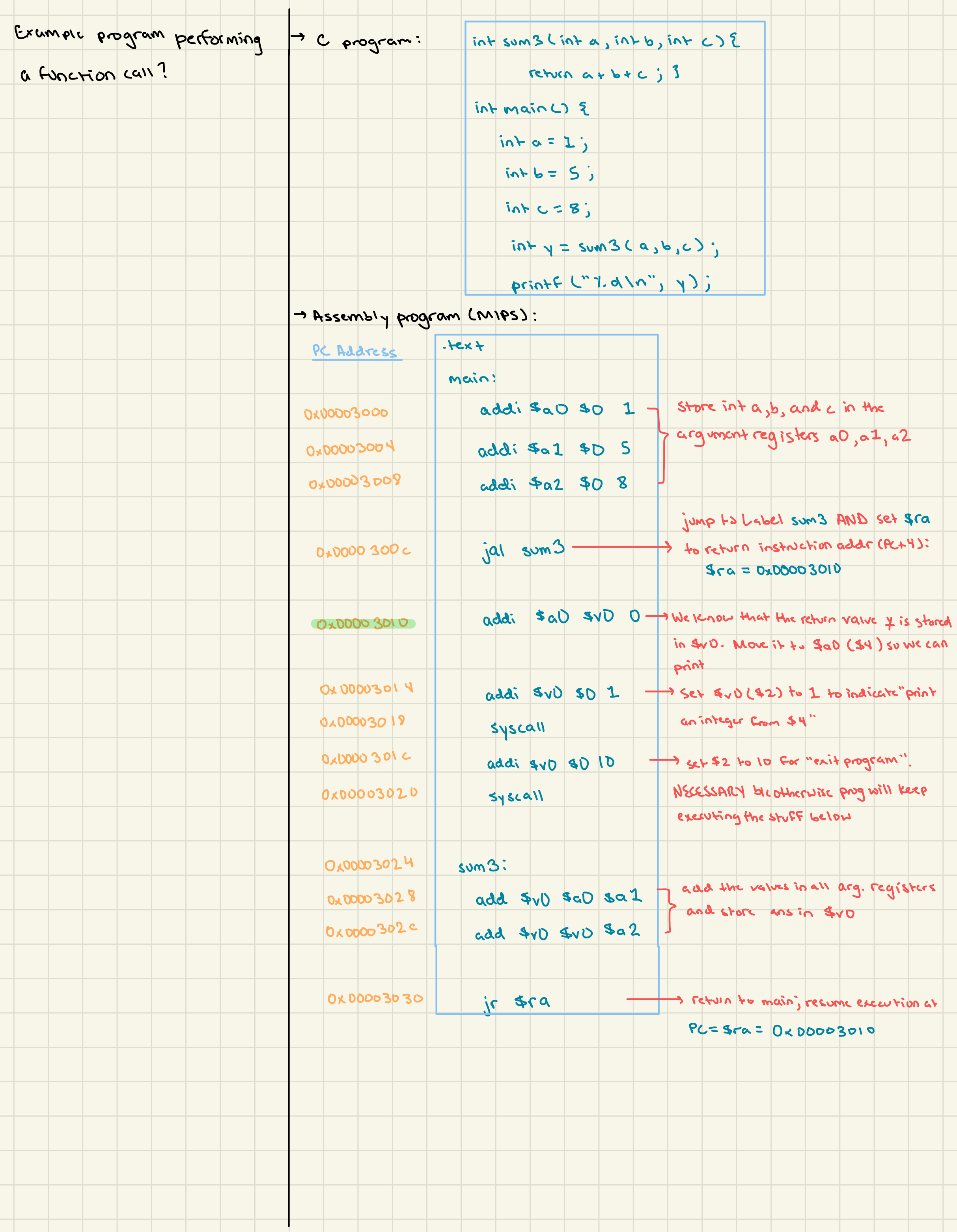
→ Using the jump register instruction!

```
jr $ra
```

Sets the PC to be the address value stored at $ra

Summary: What are some special registers and their uses?

Number	Name	Uses
$0	$zero	• always holding the value 0
$1	$at	• Resolving pseudo instructions
$2	$v0	• Where callee places return values • Where we place the "operation number" when performing syscall • Where the OS places value read from user input after syscall
$3	$v1	• Where callee places return values
$4	$a0	• Where caller places function parameter args • Where we place the value we want to print when performing syscall
$5	$a1	• Where caller places function parameter args
$6	$a2	• Where caller places function parameter args
$7	$a3	• Where caller places function parameter args
$31	$ra	• Stores return address upon a function call


Example program performing a function call?
→ C program:
int sum3 (int a, int b, int c) {
return a+b+c; }
int main() {
int a = 1;
int b = 5;
int c = 8;
int y = sum3(a,b,c);
printf("%d\n", y);
→ Assembly program (MIPS):
PC Address
.text
main:
0x00003000 addi $a0 $0 1
0x00003004 addi $a1 $0 5
0x00003008 addi $a2 $0 8
Store int a,b, and c in the argument registers a0, a1, a2
0x0000300c jal sum3
jump to Label sum3 AND set $ra to return instruction addr (PC+4): $ra = 0x00003010
0x00003010 addi $a0 $v0 0
We know that the return value y is stored in $v0. Move it to $a0 ($4) so we can print
0x00003014 addi $v0 $0 1
Set $v0 ($2) to 1 to indicate "print an integer from $4"
0x00003018 syscall
0x0000301c addi $v0 $0 10
set $2 to 10 for "exit program".
0x00003020 syscall
NECESSARY b/c otherwise prog will keep executing the stuff below
0x00003024 sum3:
0x00003028 add $v0 $a0 $a1
0x0000302c add $v0 $v0 $a2
add the values in all arg. registers and store ans in $v0
0x00003030 jr $ra
return to main; resume execution at PC = $ra = 0x00003010

Shift Instructions

What is the "shift left logical" instruction?

→ sll $rd $rt shamt

- Shift the contents of $rt to the LEFT by shamt & store result in $rd
- R[rd] = R[rt] << shamt

Why is sll useful?

→ For computing the byte offset when we need to index an array!

→ RECALL: $x << y = x \cdot 2^y$

→ Given an array, if we want to access index # b, we shift b left by 2 and add the result to our base address!

Example?

```
.data
arr: .word 1 5 10 15 20
.text
  addi $a1 $0 3      → we want arr[3]
  la $5 arr          → stores address of arr[0] in $5
  sll $a1 $a1 2      → $a1 = 3<<2 = 3*4 = 12, the byte offset
  add $6 $5 $a1      → stores addr of arr[0] + 12 = addr of arr[3] in $6
  lw $7 0($6)        → $7 = arr[3] = 15
```

What are the 2 right shift instructions?

→ Shift right logical: srl $rd $rt shamt

- Pad left side with 0s

R[rd] = R[rt] >> shamt

→ Shift right arithmetic: sra $rd $rt shamt

- Pad left side with MSB to preserve sign

Storing Variables on the Stack

When is the register $ra not enough to help return from function calls?

→ When we have **nested** function calls.

→ RECALL: $ra is used to store the return address when a function call is made using jal.

→ Ex:

Addr	Code	Notes
	.data	
	hello: .asciiz "Hello "	
	world: .asciiz "world!\n"	
	.text	
	main:	
0	jal print_helloWorld	1) jump to print_helloWorld & set $ra = PC+4 = 4
4	jal print_hello	
8	li $v0 10	
12	syscall	
	print_helloWorld:	
16	jal print_hello	2) jump to print_hello & set $ra = PC+4 = 20
20	jal print_world	4) jump to print_world & set $ra = PC+4 = 24
24	jr $ra	6) jump to address $ra = 24 ????
	print_hello:	
28	la $a0 hello	print "hello"
32	li $v0 4	
36	syscall	
40	jr $ra	3) Finished w/ print_hello. Jump to addr. $ra = 20
	print_world:	
44	la $a0 world	print "world\n"
48	li $v0 4	
52	syscall	
56	jr $ra	5) Finished w/ print_world. Jump to addr. $ra = 24

What has happened in this example?

→ At step 6, we are ready to return to main, as it was the one that called print_helloWorld

- We expected jr $ra to take us to PC = 20

→ BUT, at this point $ra was overwritten, so we have lost the return address to main and are stuck infinitely jumping to address 24 !!!

How can we fix this problem?

→ By saving the ret. addr. somewhere else before we overwrite it. Saving it to another register is not a sustainable solution b/c it can still get overwritten.

→ Instead, we will store the RA on the **stack**

When would we save the RA on the stack?

→ When the callee begins, save the value of $ra to the stack

→ When the callee is ready to return, get the val. from the stack & place it back into $ra, before the jr $ra instruction.

```
print_helloWorld:
    # save $ra=20 on stack
    jal print_hello
    jal print_world
    #restore ra=20 from stack
    jr $ra
```

→ Now, step 6 (from Ex on prev page) will take us back to main by setting PC=20!

– The stack pointer –

RECALL: What is the stack?

→ An area of memory used to store temporary function data, such as local variables.

→ The stack grows DOWN (high → low mem. addresses)

What is the stack pointer?

→ A special register $sp (which is actually $29) that is used to point to the last item that was placed on the stack

- $sp holds the address of the most recently placed val on the stack.

$sp → [memory layout, top to bottom]

Stack ↓ / ↑ Heap (Dynamic Data)
Static Data
Text
Reserved

How do we store a new item on the stack?

→ The stack grows down, so we must decrement the addr. value stored in $sp by 4 to move to a new spot on the stack.

Example?

① Current Stack: $sp = 0x00001034 and points to the last val added, 12.

	Address	Data
$sp →	0x00001034	12
	0x00001030	
	0x0000102C	
	0x00001028	
	0x00001024	

→ To store a new item, the value of $ra:

1) Decrement sp by 4
2) store value in memory

```
addi $sp $sp -4
sw $ra 0($sp)
```

② Updated stack:

	Address	Data
	0x00001034	12
$sp →	0x00001030	0x00003004
	0x0000102C	
	0x00001028	
	0x00001024	

How do we remove an item from the stack?

→ "Remove" aka to fetch back the value we stored

→ When we want to restore a val from the stack, we simply
1. read it into a register (with lw)
2. and then **move the stack pointer back up**

→ We don't have to "clear" the value from memory; by moving sp back up, we've "unallocated" that stack space. B/c next time we want to store on the stack, we will just overwrite this value.

Example?

→ ① Current stack: $sp = 0x00001030

→ ② Read / "remove" / restore value of ra:

```
lw $ra 0($sp)
addi $sp $sp 4
```

	Address	Data
	0x00001034	12
$sp →	0x00001030	0x00003004
	0x0000102C	
	0x00001028	
	0x00001024	

→ ③ Updated stack:

	Address	Data
$sp →	0x00001034	12
	0x00001030	0x00003004
	0x0000102C	
	0x00001028	
	0x00001024	

– Calling Convention with Function-saved values –

RECALL: Why do we have a calling convention?

→ Imagine each function in a program was written by a different person, & they didn't know how the other person wrote their func, or what registers they used.

→ Then, we can't necessarily rely on a val. saved in a reg. to not be overwritten, so we must have rules/conventions for preserving all parts of the program (for ex, param args in $a0-$a3 & return values in $v0-$v1)

What is another example of when we might need to save values to the stack?

→ If a callee overwrites caller-saved values! Ex:

```
main:
  addi $s1 $0 0
  la $s2 array-1
  la $s3 array-2
  lw $a0 0($s2)
  jal fun
fun:
  addi $s1 $a0 1
  sub $s2 $0 $a0
  mul $v0 $s1 $s2
  jr $ra
```

fun overwrites the values in $s1 and $s2!

How can we fix this problem?

→ By having the callee save the values in the caller-saved registers to the stack before overwriting them.

→ Then, before returning to caller, the callee restores the values from the stack!

Example of the fix?

→ The callee "fun":

```
fun:
  addi $sp $sp -8      → allocate space to store 2 reg-values on the stack by decrementing $sp by 8.
  sw $s2 4($sp)        → Store contents of $s2 at addr [sp addr + 4]
  sw $s1 0($sp)        → Store contents of $s1 at addr [sp addr]

  addi $s1 $a0 1   ]
  sub $s2 $0 $a0   ]   → Now, callee can use registers $s1 and $s2 w/o worrying
  mul $v0 $s1 $s2  ]

  lw $s1 0($sp)    ]   → Before returning to the caller, restore the values of
  lw $s2 4($sp)    ]     $s1 and $s2 by reading from the stack
  addi $sp $sp 8       → move sp back up to "deallocate"

  jr $ra
```

What is a caller saved register in the calling convention?

→ Registers that the caller uses, BUT expects that the callee may overwrite them.

- The callee CAN overwrite these registers w/o saving them

→ If caller intends to use them after a func. call, it is the **caller's responsibility to save them.**

What are callee saved registers?

→ Registers that the caller expects the callee to NOT overwrite — caller expects the vals in these registers to be saved.

→ If the callee wants to use these registers, it is the **callee's responsibility** to save their values to the stack and then later restore them.

What are the designated caller- and callee- saved registers in MIPS?

Registers	Name	Use	Type
$2-3	$v0 - $v1	callee stores return values	caller-saved
$4-7	$a0 - $a3	caller stores param. args	caller-saved
$8-15	$t0 - $t7	registers for callee to use	caller-saved
$16-23	$s0 - $s7	registers for caller to use	callee-saved
$24-25	$t8 - $t9	registers for callee to use	caller-saved
$31	$ra	return address pointer	caller-saved

↑ If the caller is ALSO a callee, they are responsible for saving their own RA before calling another func.