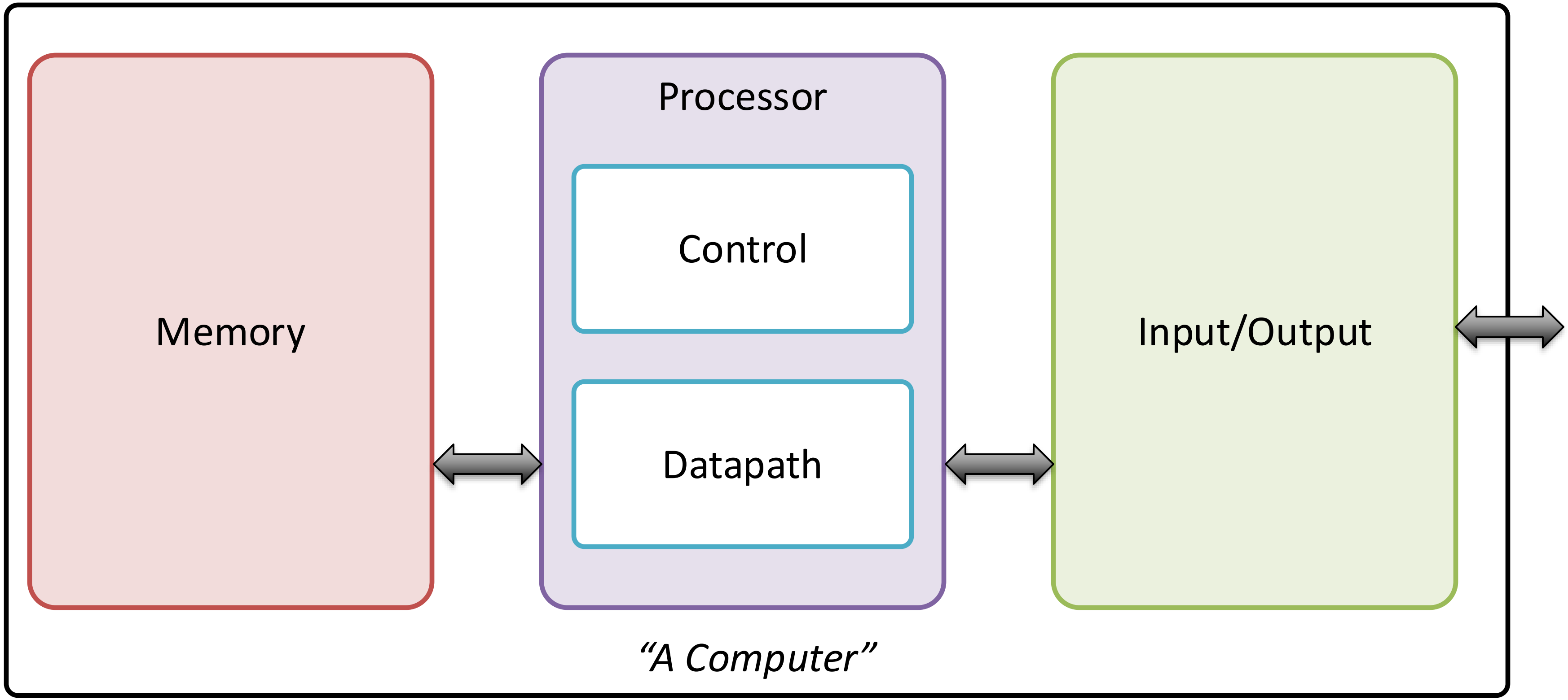
CSE 141: Introduction to Computer Architecture

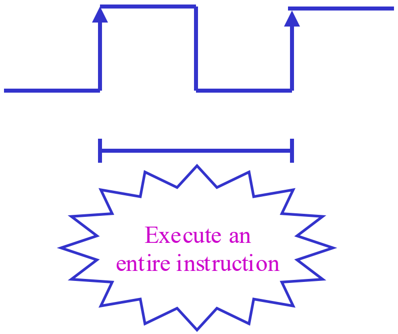
The Single Cycle Machine

Zooming out for a moment…
The major building blocks of a computer



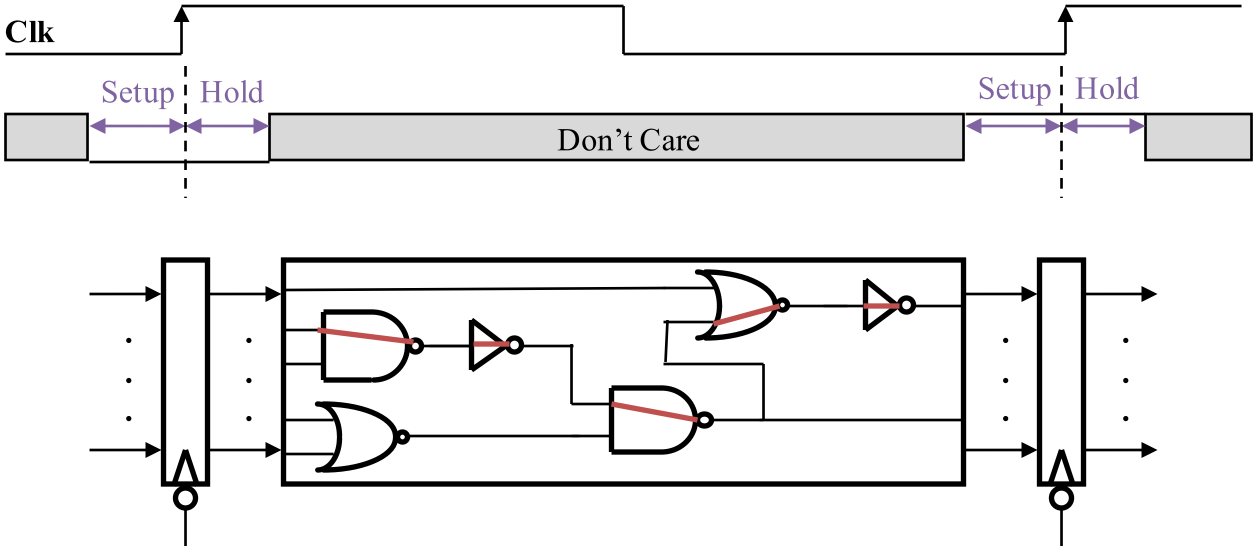
The Big Picture: The Performance Perspective

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Starting today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time



- ET = Insts * CPI * Cycle Time

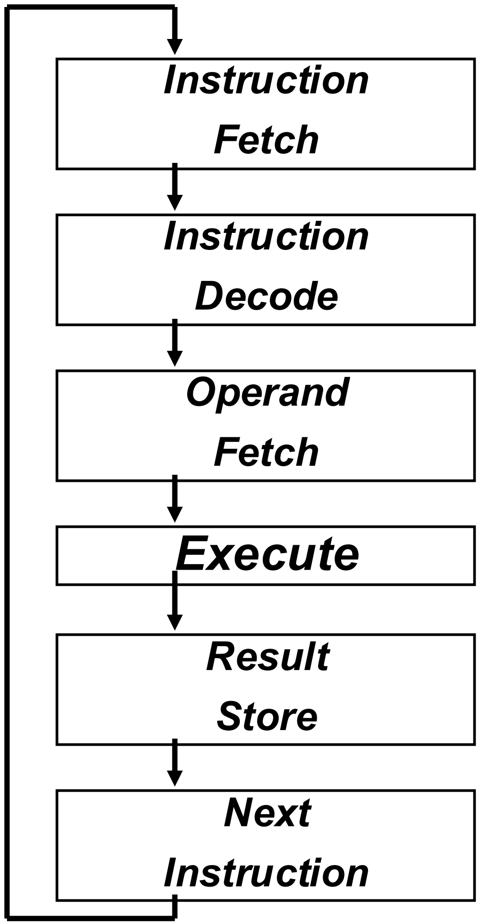
Review: Synchronous and Asynchronous logic



- All storage elements are clocked by the same clock edge

The Processor: Datapath & Control

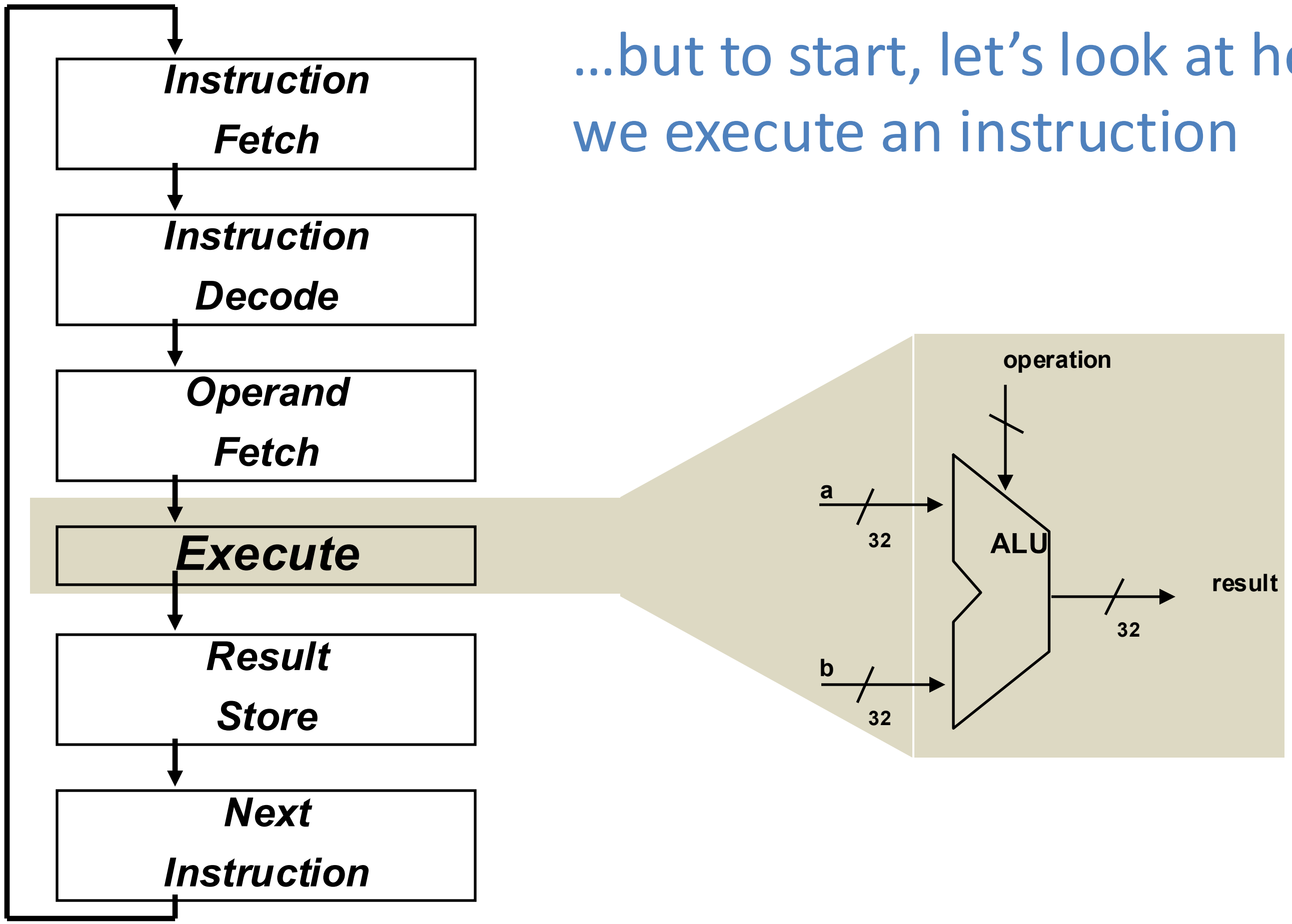
- We're ready to look at a simplified MIPS with only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do



Recall…

Computing is much more than just executing instructions!

…but to start, let’s look at how we execute an instruction



Recall: 2's complement

- Need a ***number system*** that provides
  - obvious representation of 0,1,2...
  - uses an adder for both unsigned and signed addition
  - single value of 0
  - equal coverage of positive and negative numbers
  - easy detection of sign
  - easy negation

binary	**unsigned**	**signed**
0001		
	14	
		-8

[Review on your own]
Questions About Numbers

- How do you represent
  - negative numbers?
  - fractions?
  - really large numbers?
  - really small numbers?
- How do you
  - do arithmetic?
  - identify errors (e.g. overflow)?

[Review on your own]
Two's Complement Representation

- 2's complement representation of negative numbers
  - Take the bitwise inverse and add 1
- Biggest 4-bit Binary Number: 7
- Smallest 4-bit Binary Number: -8

Decimal	**Two's Complement Binary**
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Point out negatives,
sign bit,

how to convert a 2's
complement number

[Review on your own]
Some Things We Want To Know About Our Number System

- How does negation work?
- Sign extension?
  - +3 => 0011, 00000011, 0000000000000011
  - -3 => 1101, 11111101, 1111111111111101

[Review on your own]
Introduction to Binary Numbers

- Consider a 4-bit binary number

Decimal	Binary	Decimal	Binary
0	0000	4	0100
1	0001	5	0101
2	0010	6	0110
3	0011	7	0111

Walk through the add

- Examples of binary arithmetic:
  - 3 + 2 = 5   3 + 3 = 6

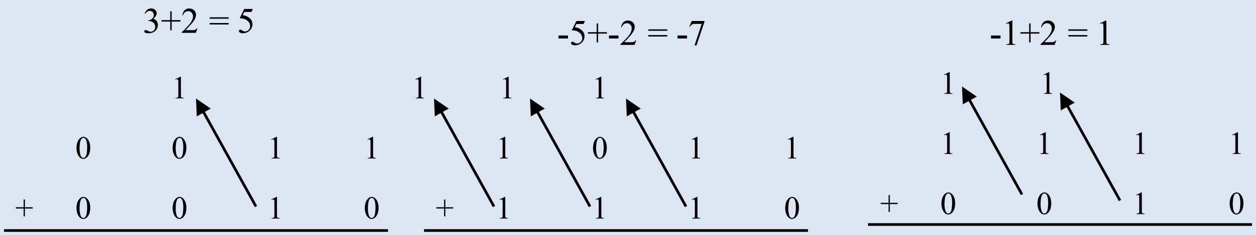
3 + 2 = 5:

		1		
	0	0	1	1
+	0	0	1	0

3 + 3 = 6:

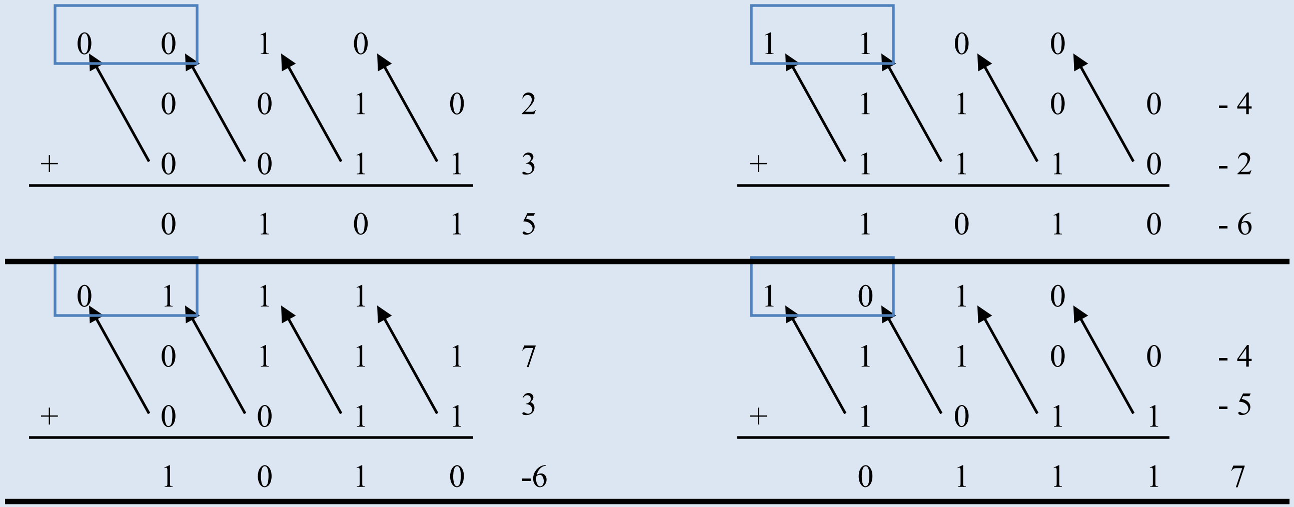
		1	1	
	0	0	1	1
+	0	0	1	1

[Review on your own]
Can we use that same procedure for adding 2’s complement negative #s as unsigned #s?



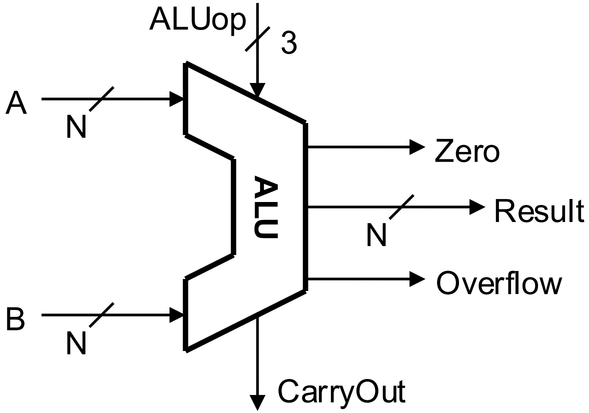
Selection	“Best” Statement
A	Yes – the same procedure applies
B	Yes – the same “procedure” applies but it changes overflow detection
C	No – we need a new procedure
D	No – we need a new procedure and new hardware to implement it
E	None of the above

[Review on your own]
Overflow Detection



So how do we detect overflow for signed arithmetic?

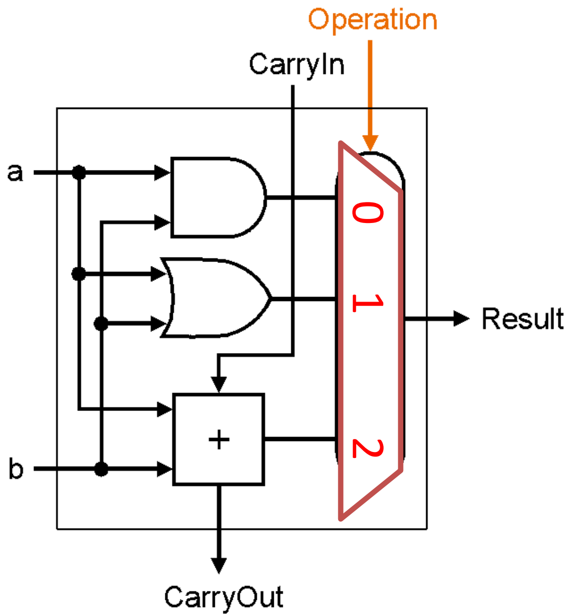
“Arithmetic Logic Units” are the computing part of computers — how do they work?



ALU Control Lines (ALUop)	Function
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

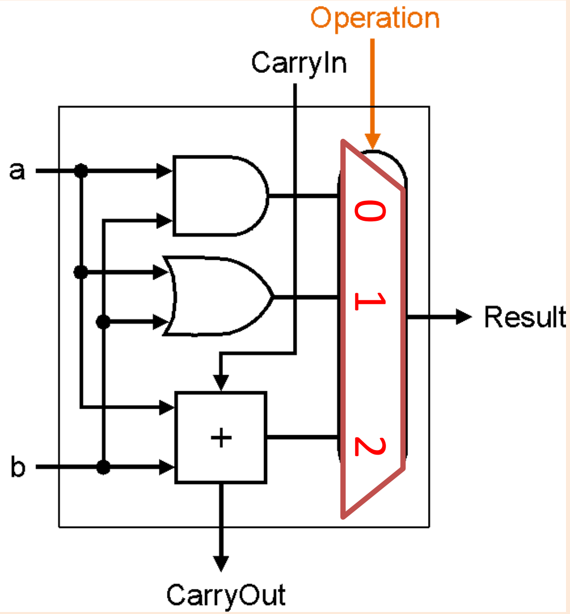
Start small: A one-bit ALU

- This 1-bit ALU will perform AND, OR, and ADD



Poll Q

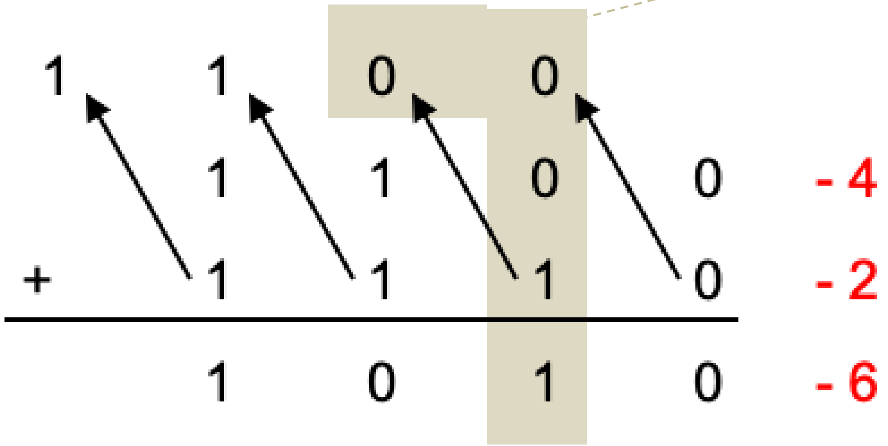
- During each cycle where `Operation==1`, this ALU will **compute**…

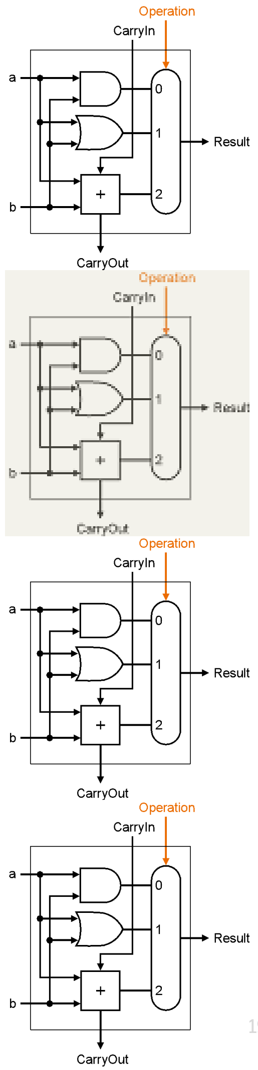


A	a & b
B	a | b
C	a + b
D	All of the above
E	None of the above

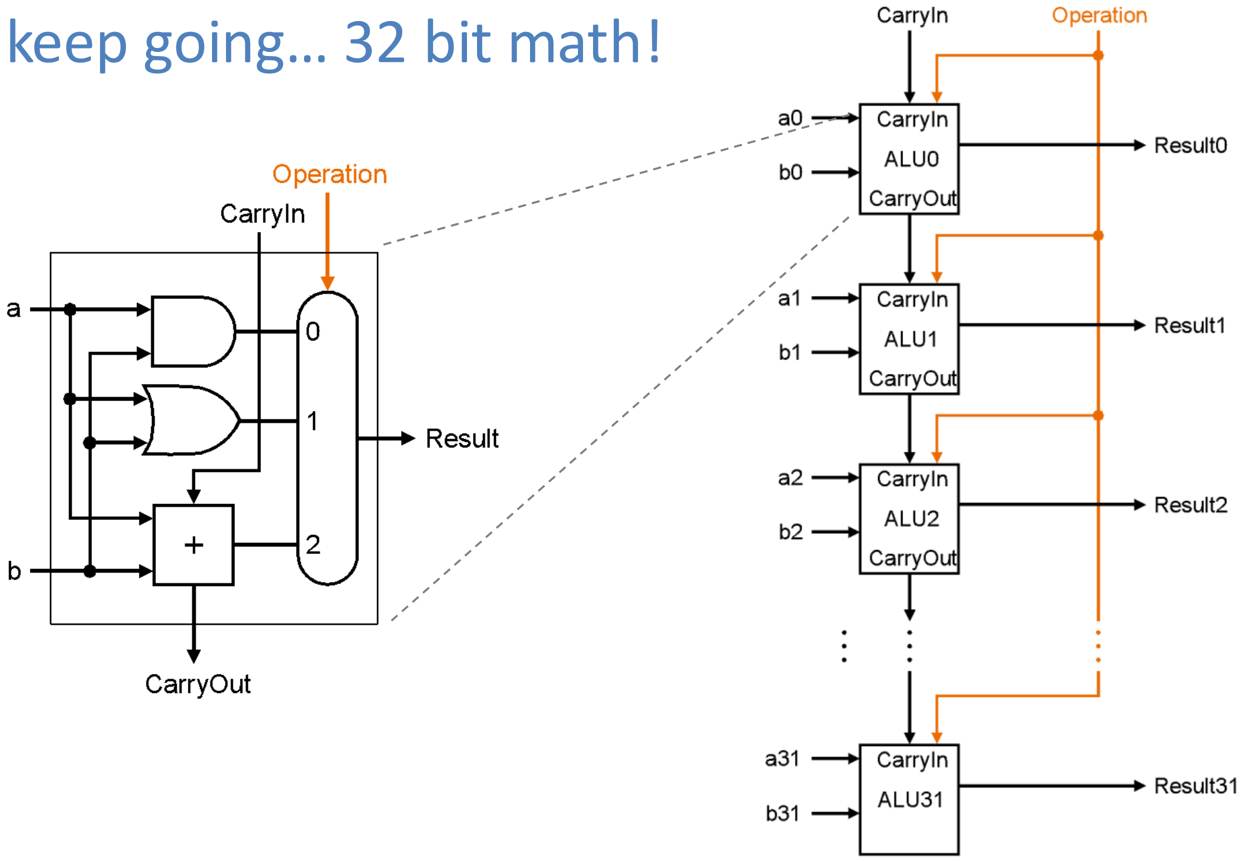
Recall: Binary addition works just like “normal” (base 10), but you end up “carrying” more often

- A 4-bit ALU can be made from four 1-bit ALUs





And if you keep going… 32 bit math!

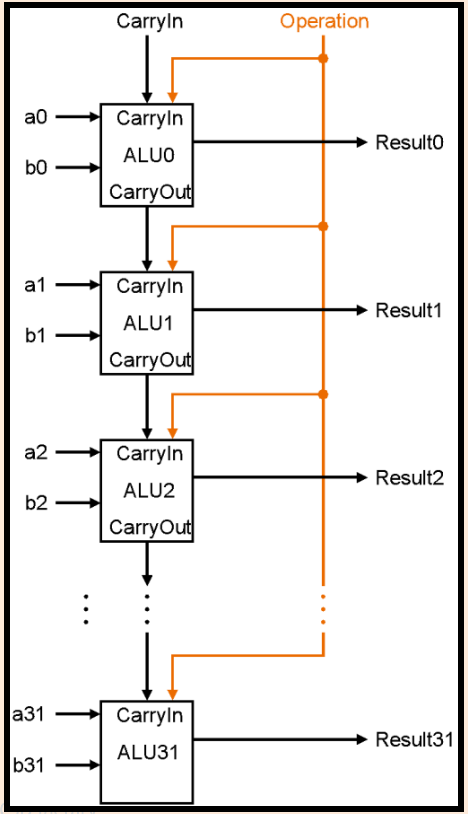


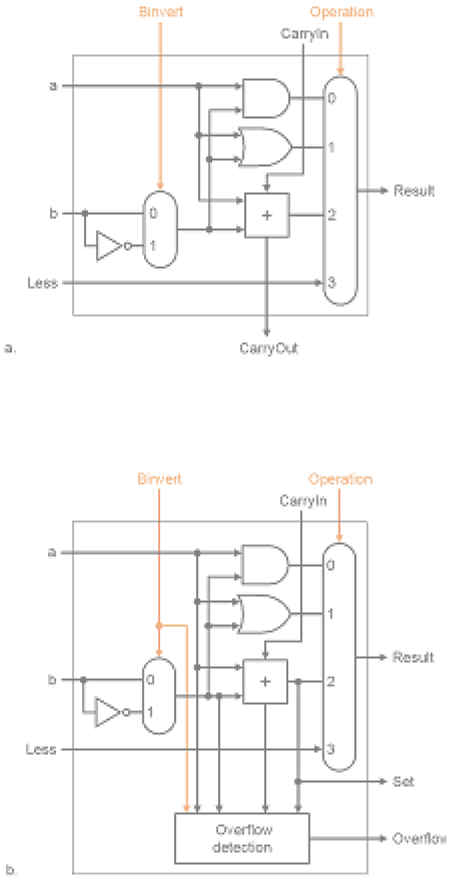
Hint: A-B is the same as A + (-B)

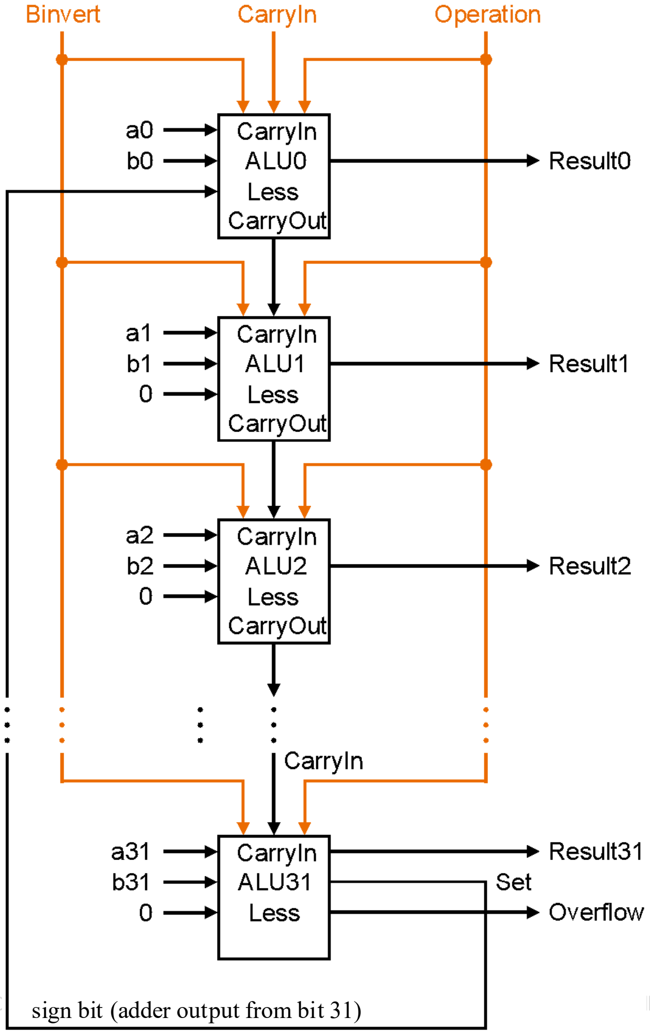
Poll Q: We'd like to implement a means of doing A-B (subtract) but with only minor changes to our hardware. How?

1. Provide an option to use bitwise NOT A
2. Provide an option to use bitwise NOT B
3. Provide an option to use bitwise A XOR B
4. Provide an option to use 0 instead of the first Carry_{In}
5. Provide an option to use 1 instead of the first Carry_{In}

Selection	Choices
A	1 alone
B	Both 1 and 2
C	Both 3 and 4
D	Both 2 and 5
E	None of the above





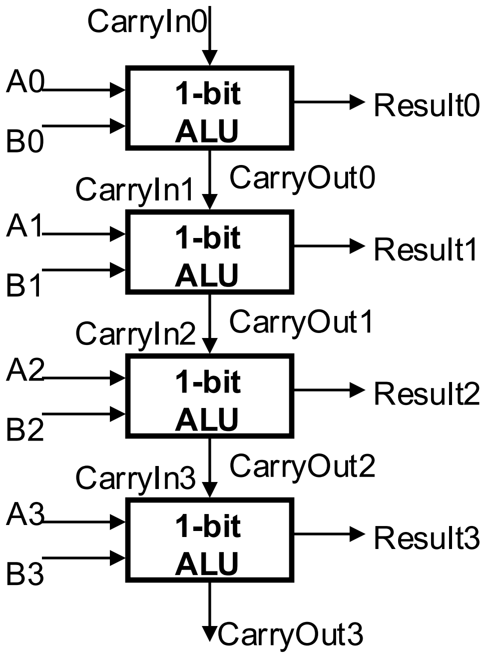


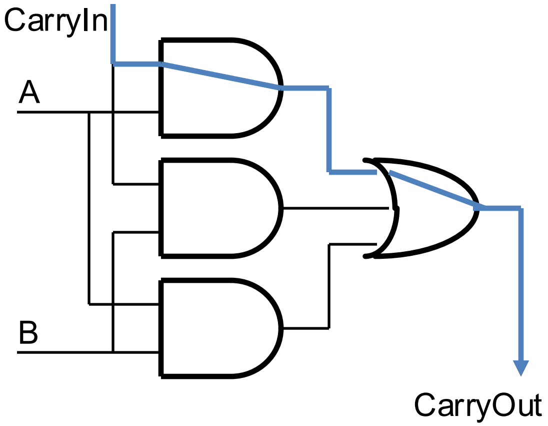
The full ALU

	B_{invert}	$Carry_{In}$	Operation
and			
or			
add			
sub			
beq			
slt			

The Disadvantage of Ripple Carry

- The adder we just built is called a "Ripple Carry Adder"
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an N-bit RC adder: 2N-gate delay





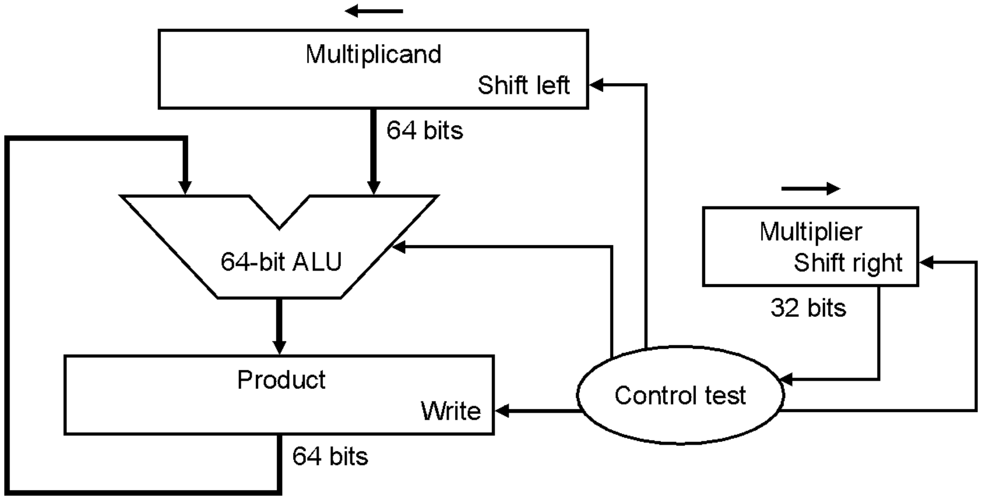
The point: ripple carry adders are slow. Faster addition schemes are possible that *accelerate* the movement of the carry from one end to the other. Optimizing this is *digital logic* (CSE 140).

Why doesn't our (simplified) single-cycle machine support multiplication or division?

We're ready to look at a simplified MIPS with only:
- memory-reference instructions: `lw, sw`
- arithmetic-logical instructions: `add, sub, and, or, slt`
- control flow instructions: `beq`

- How does a computer multiply?
  - How do you multiply?

  123
  <u>x 321</u>



The point: Multiplication (and division) is a lot of work to try to do in a single cycle

Poll Q: Which of these real-world processors supports single-cycle multiply?

A) "Biggest, best"
- 4.1 GHz Intel [core i7]
- ~$500

10th Generation Intel Core Processor based on Ice Lake				
iform	regsize	mask	Throughput	Latency
IMUL_GPRv_GPRv	16	no	1.0	3.0
IMUL_GPRv_GPRv	32	no	1.0	3.0
IMUL_GPRv_GPRv	64	no	1.0	3.0
IMUL_GPRv_MEMv	16	no	1.0	8.0
IMUL_GPRv_MEMv	32	no	1.0	8.0
IMUL_GPRv_MEMv	64	no	1.0	8.0
IMUL_GPRv_MEMv_IMMb	16	no	1.0	9.0
IMUL_GPRv_MEMv_IMMb	32	no	1.0	8.0
IMUL_GPRv_MEMv_IMMb	64	no	1.0	8.0
IMUL_GPRv_MEMv_IMMz	16	no	1.0	9.0
IMUL_GPRv_MEMv_IMMz	32	no	1.0	8.0
IMUL_GPRv_MEMv_IMMz	64	no	1.0	8.0

B) "Smallest, cheapest"
- 48 MHz ARM [Cortex M0]
- ~$0.50

The Cortex-M0 processor is built on a highly area and power optimized 32-bit processor core, with a 3-stage pipeline von Neumann architecture. The processor delivers exceptional energy efficiency through a small but powerful instruction set and extensively optimized design, providing high-end processing hardware including a single-cycle multiplier.

Modern Concerns about Execute

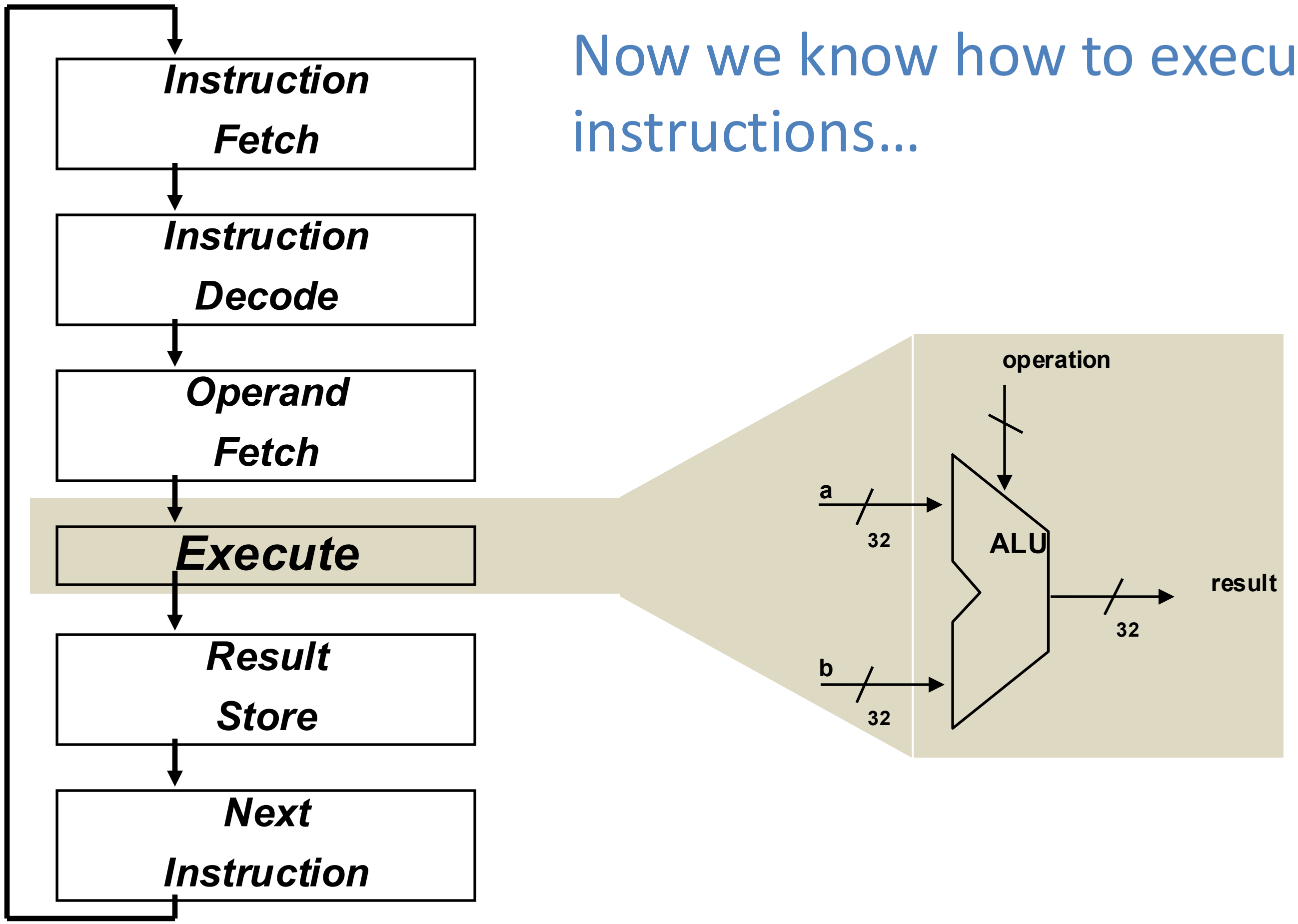
(aka, why is no one angry that an i7 can't do single-cycle multiply?)

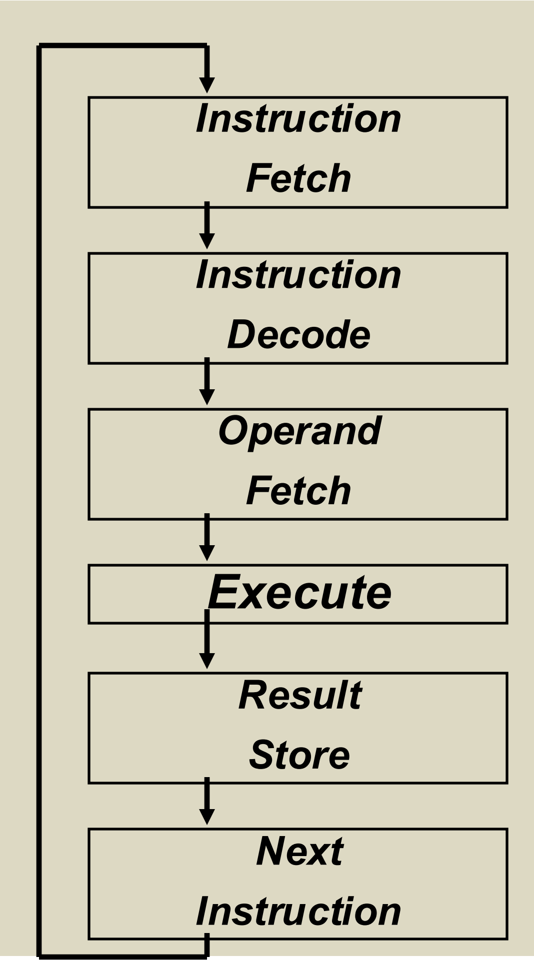
- Hardware designers have done an excellent job optimizing multiply/FP hardware, but additions are still faster, than, say multiply. Divides are even slower and have other problems.

- More complex topics in later lectures will show how multiply/FP/divide may not be on the "critical path" and hence may not hurt performance as much as expected.

- More recent years have taught us that even "slow" multiply is not nearly as important as cache/memory issues we'll discuss in later lessons.

Now we know how to execute instructions…

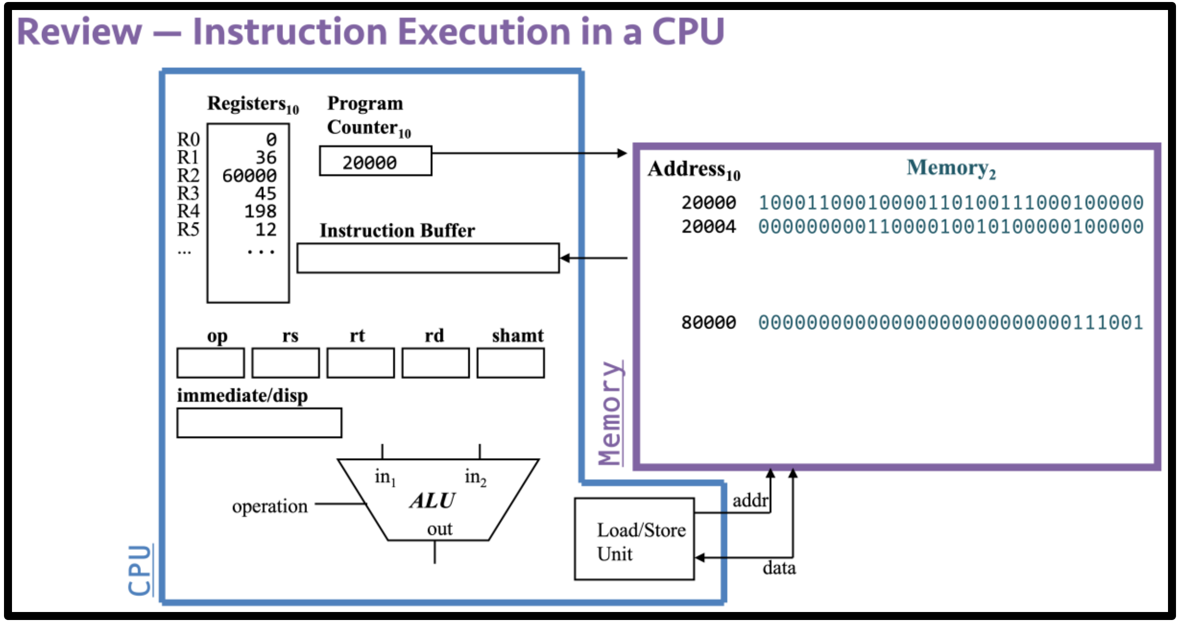




…so let's look at the rest of the machine!

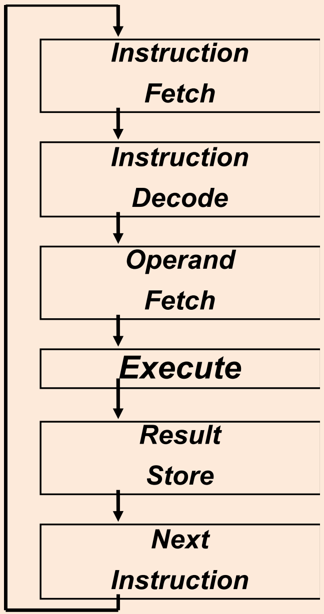
Our previous view of a computer had no organization

- From Part I…



Think about how a MIPS machine executes instructions… Which correctly describes the *order* things must happen in?

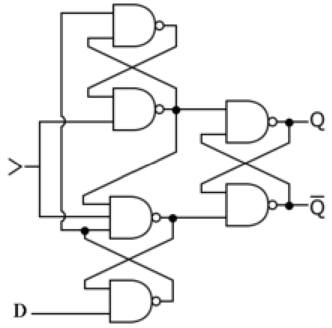
A. The ALU *always* performs an operation before accessing data memory
B. The ALU *sometimes* performs an operation before accessing data memory
C. Data memory is *always* accessed before performing an ALU operation
D. Data memory is *sometimes* accessed before performing an ALU operation
E. None of the above.

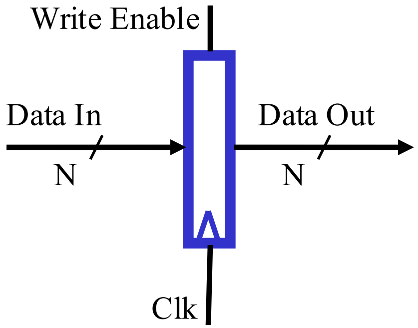


So what does this tell us about what the machine might look like?

Storage Element: Register

- Review: D Flip Flop
- New: Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - Write Enable:
    - 0: Data Out will not change
    - 1: Data Out will become Data In (on the clock edge)

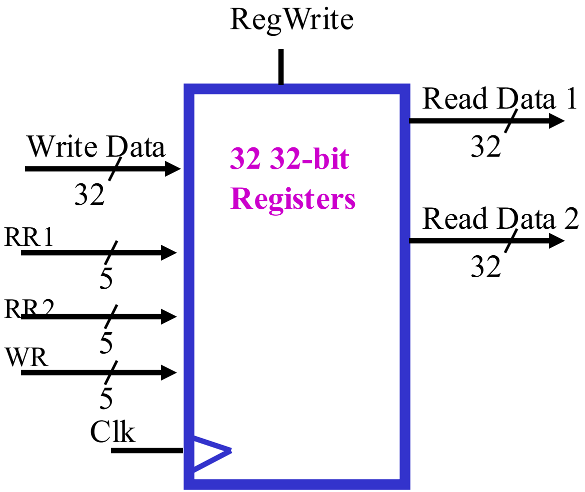




A *register file* is a structure that holds many registers. What kinds of signals will we need for our MIPS register file?

	Number of bits for register output	Number of bits for register selection	Control Inputs?	Control Outputs?
A	5	32	clk	read/write
B	5	5	clk, read/write	clk
C	32	5	clk, read/write	(none)
D	32	32	clk, read/write	clk, read/write
E	32	5	read/write	(none)

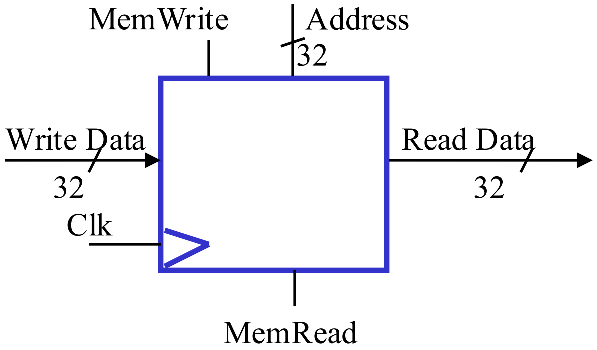
Let's try to make a *Register File*



Which of these describes our memory interface (for now)?

A	One 32-bit output	One 5-bit input	One 32-bit input	Clk input	Two 1-bit control inputs
B	One 32-bit output	Two 5-bit inputs		Clk input	Two 1-bit control inputs
C	One 32-bit output		Two 32-bit inputs	Clk input	Two 1-bit control inputs
D	One 32-bit output		One 32-bit input	Clk input	Two 1-bit control inputs
E	*None of these are correct*				

Let's describe the signals to interface to *Memory*

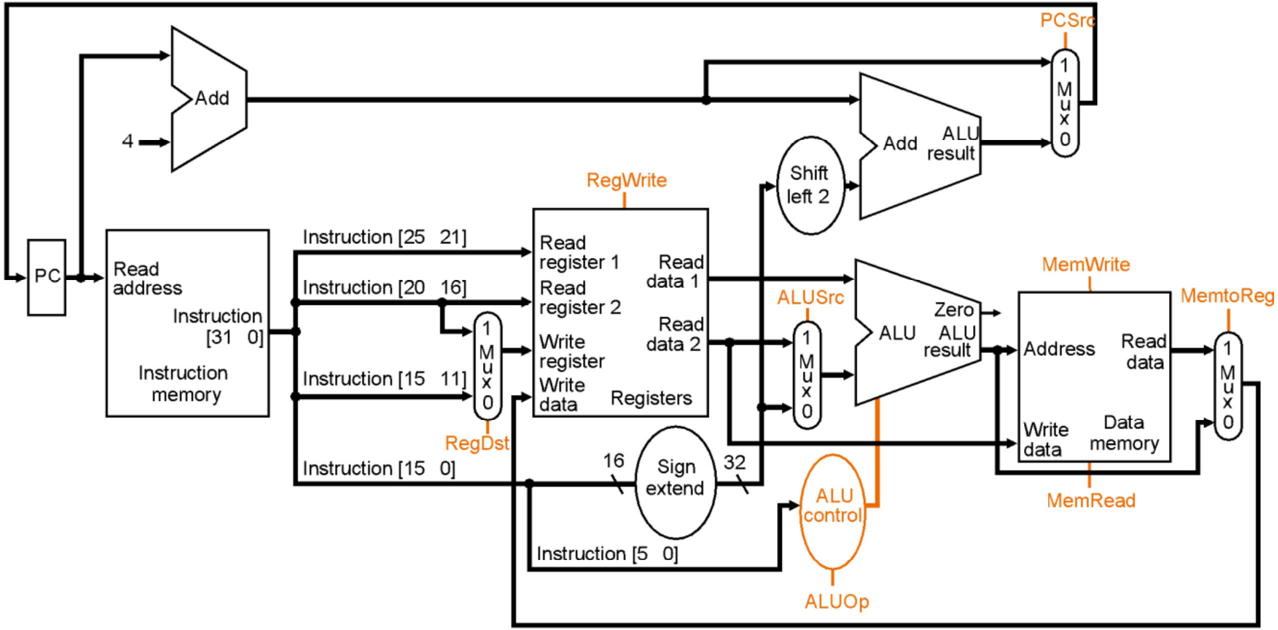


Can we layout a high-level design to do everything?

We're ready to look at a simplified MIPS with only:
- memory-reference instructions: `lw, sw`
- arithmetic-logical instructions: `add, sub, and, or, slt`
- control flow instructions: `beq`

Putting it All Together: A Single Cycle Datapath

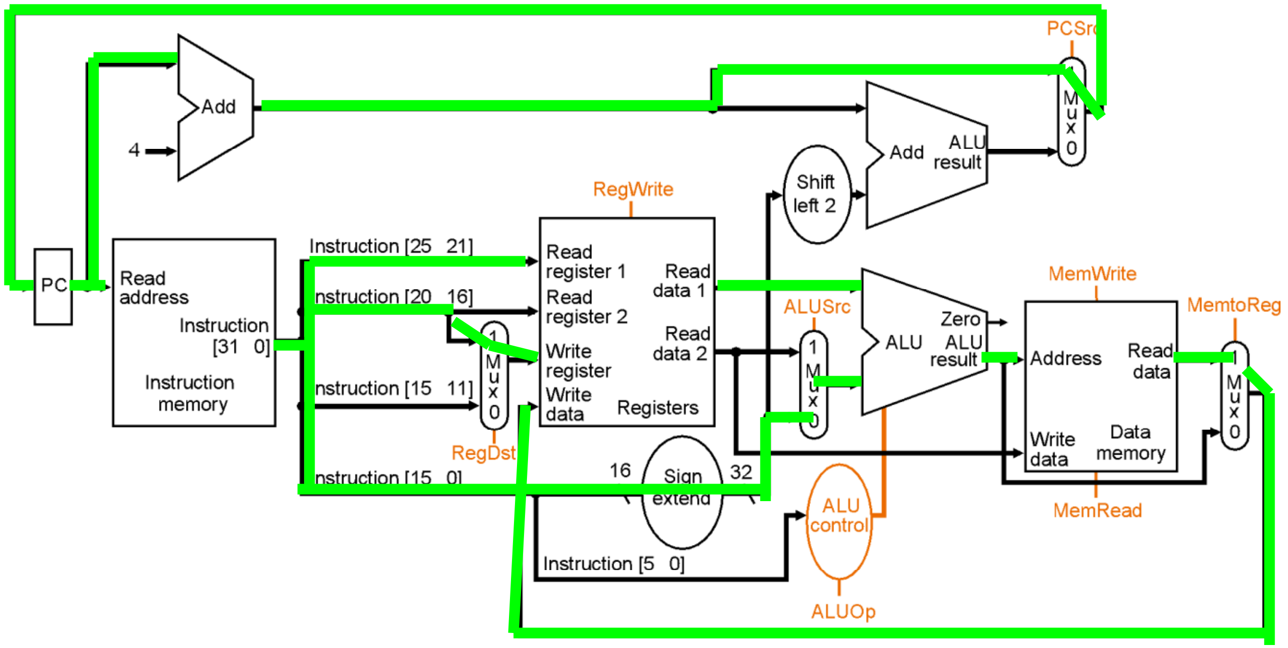
- We have everything except control signals (later)



Active Single-Cycle Datapath

Ignoring control – which instruction does this active datapath represent

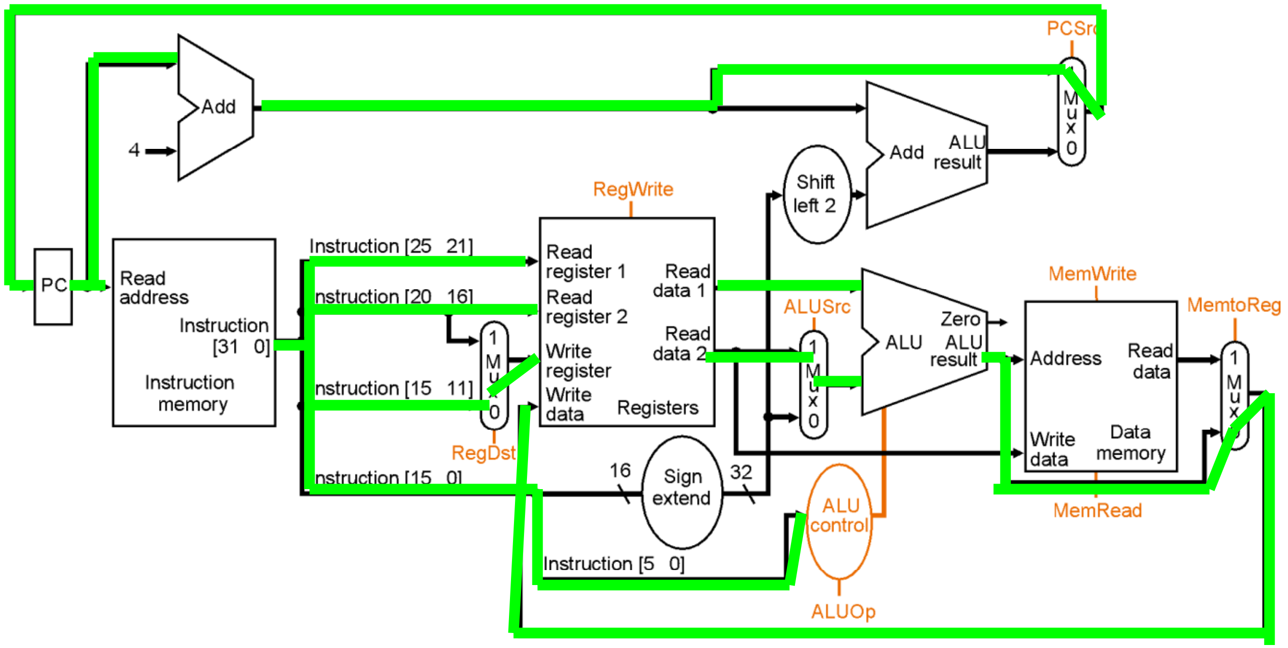
A. R-type
B. lw
C. sw
D. Beq
E. None of the above



Active Single-Cycle Datapath

Ignoring control – which instruction does this active datapath represent

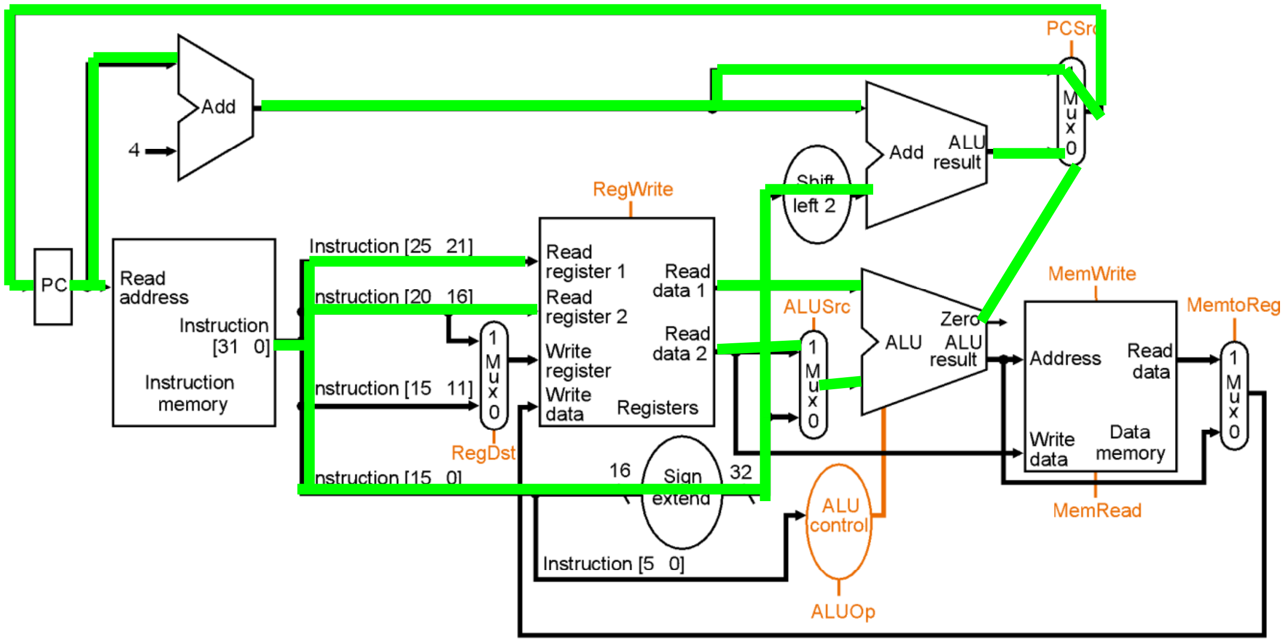
A. R-type
B. lw
C. sw
D. Beq
E. None of the above



Active Single-Cycle Datapath

Ignoring control – which instruction does this active datapath represent

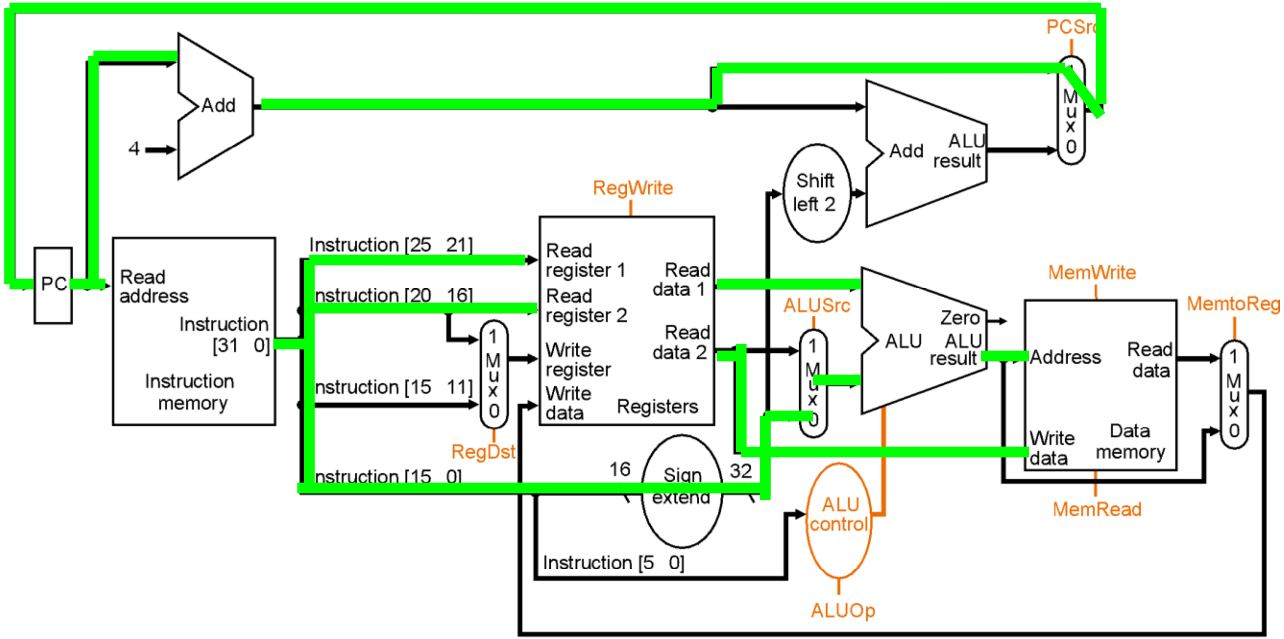
A. R-type
B. lw
C. sw
D. Beq
E. None of the above



Active Single-Cycle Datapath

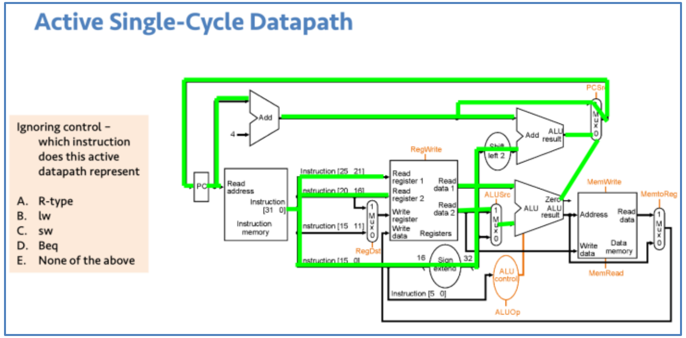
Ignoring control – which instruction does this active datapath represent

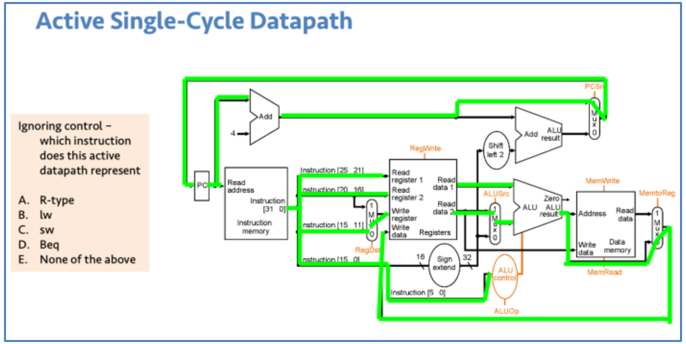
A. R-type
B. lw
C. sw
D. Beq
E. None of the above

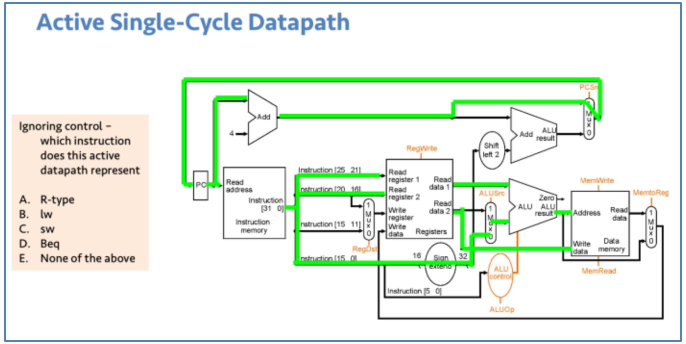


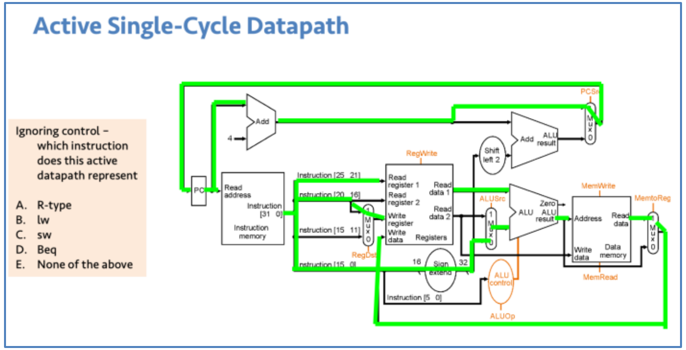
“The Control Path”

aka, what controls which wires are green?

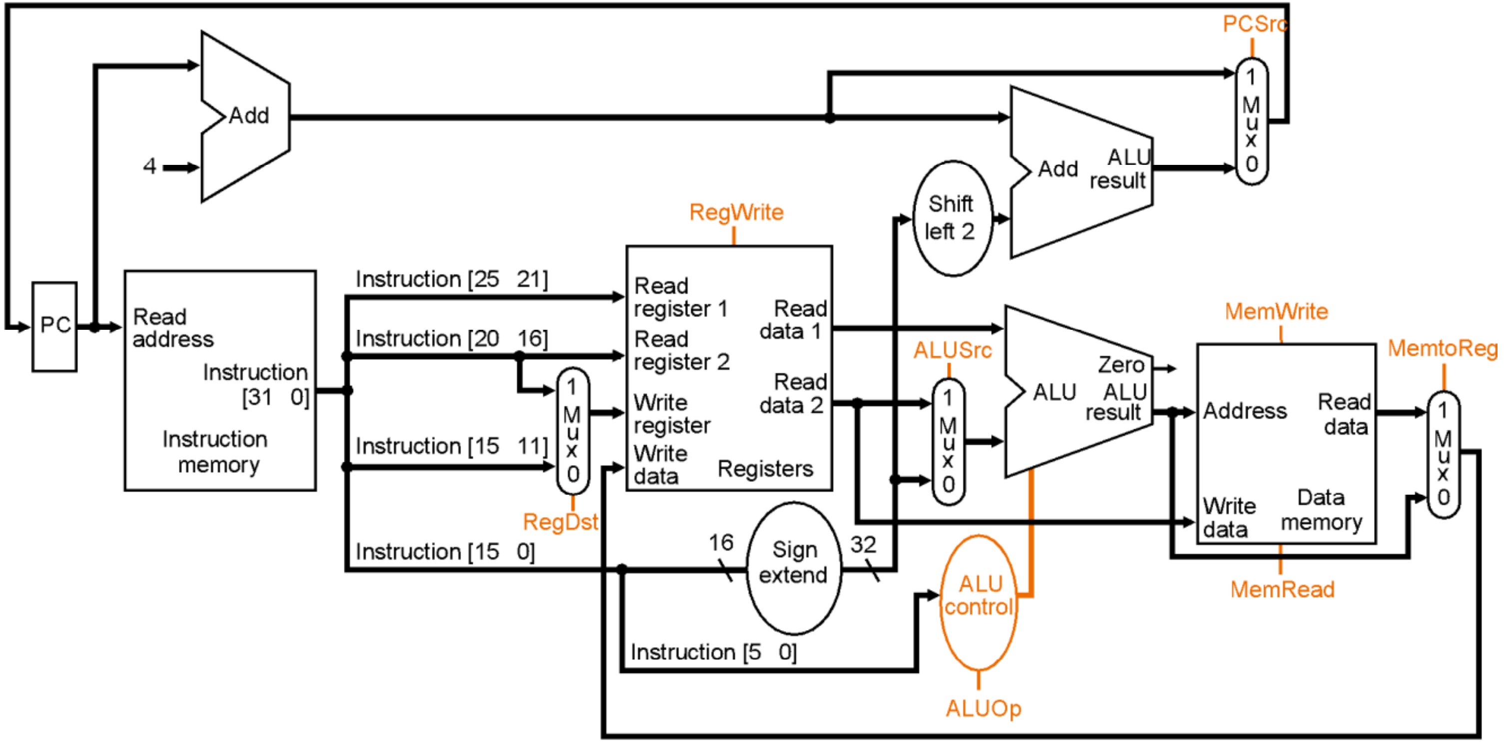








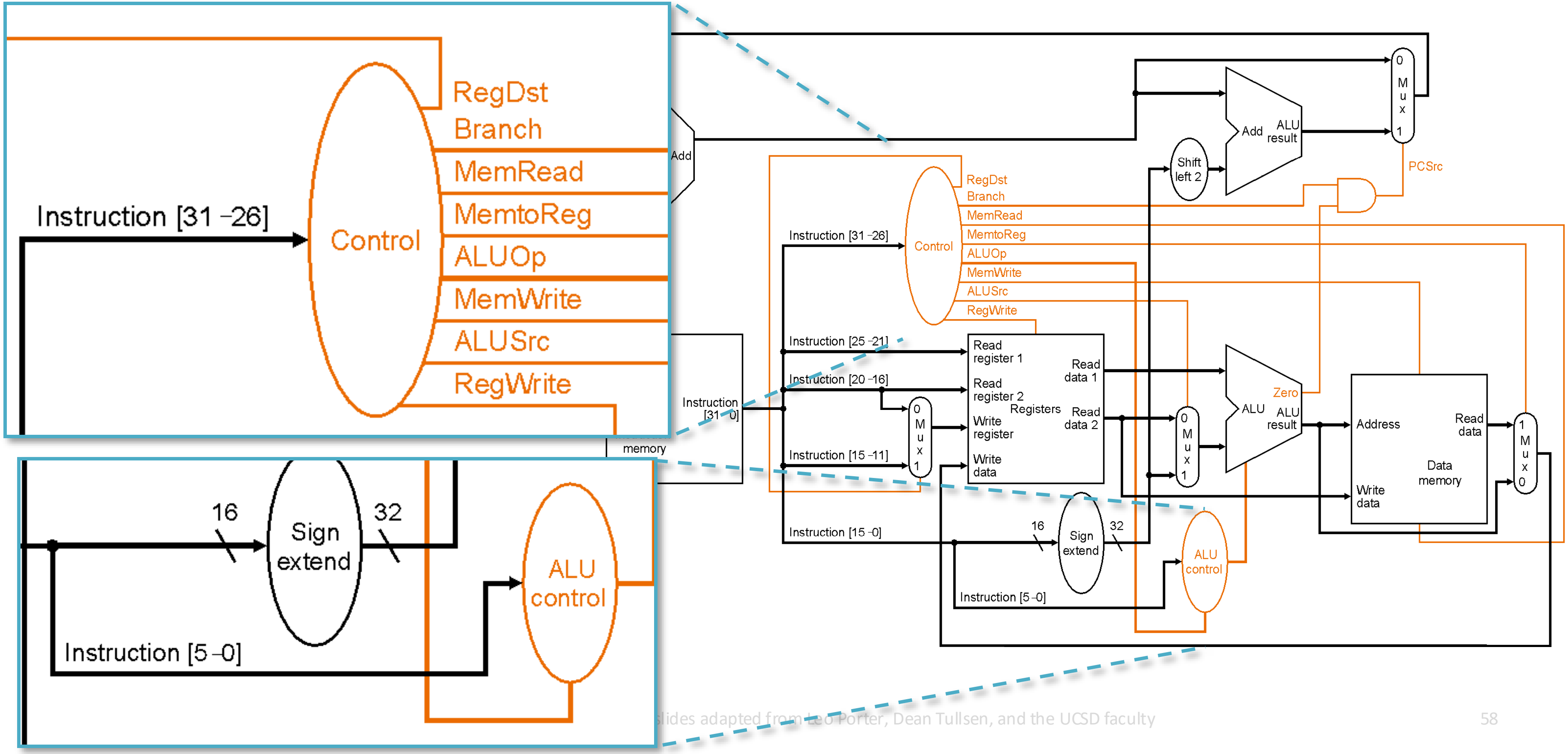
Control signals are all the parts in red

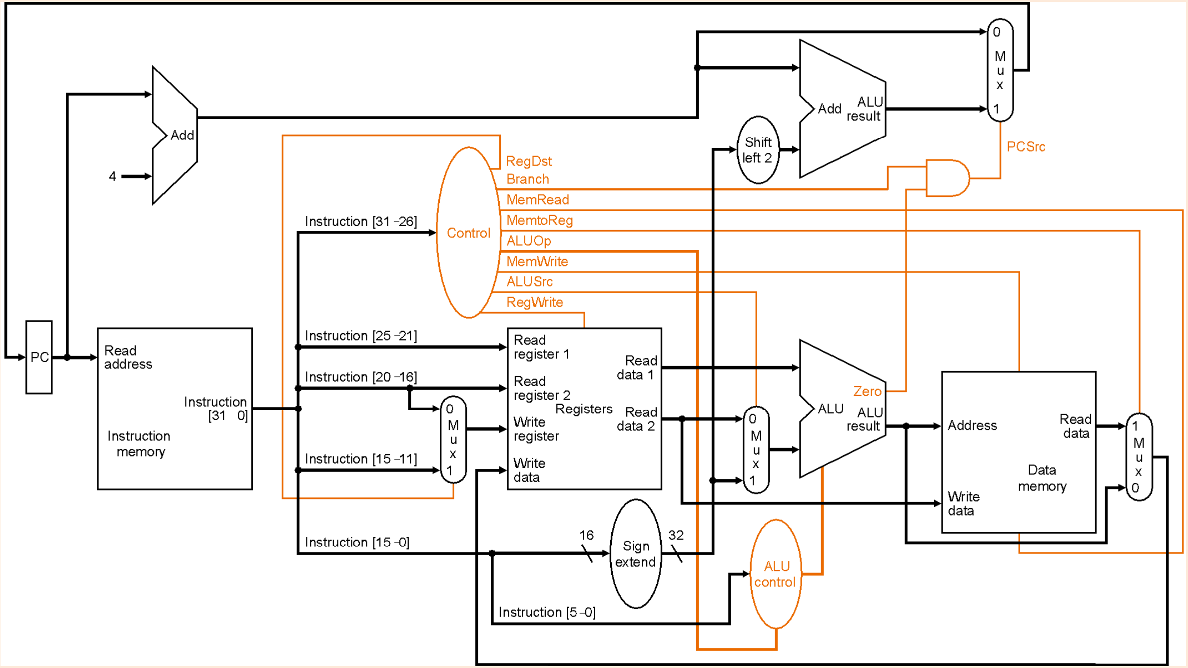


Where might we get control signals?

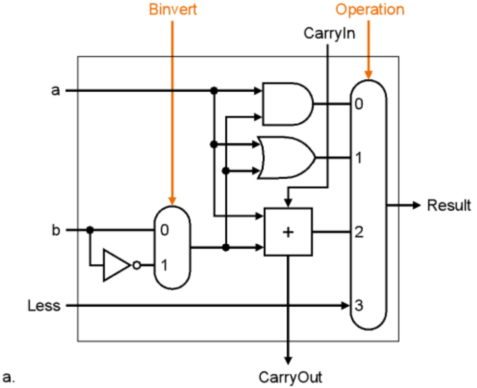
- Ideas?

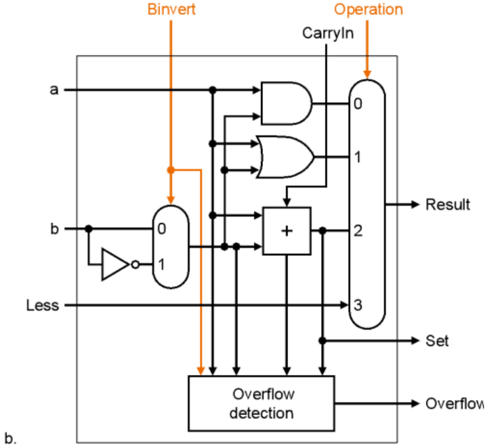
Where do we get control signals?

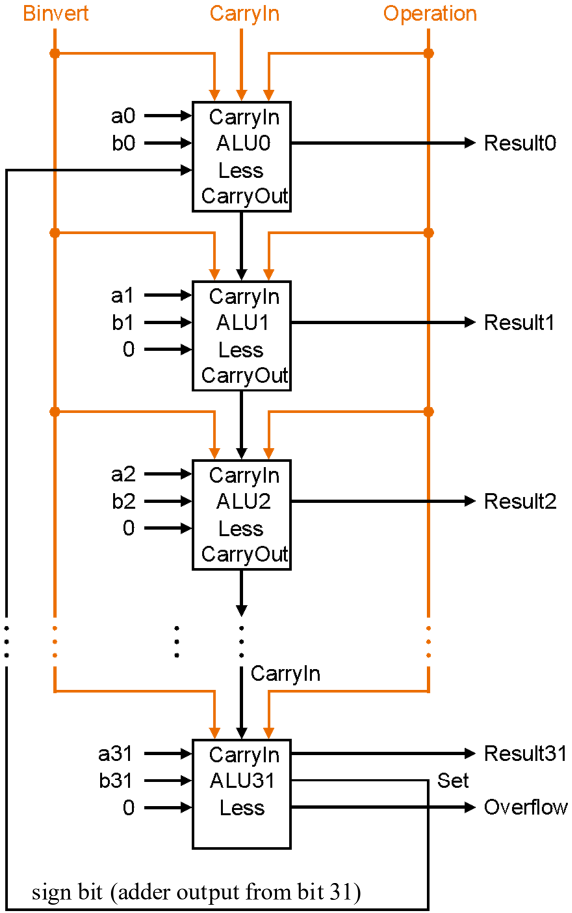




	Select the true statement for MIPS
A	Registers can be read in parallel with control signal generation
B	Instruction Read can be done in parallel with control signal generation
C	Registers can be written in parallel with control signal generation
D	The main ALU can execute in parallel with control signal generation
E	None of the above







Recall: The full ALU

How many bits to control?

	B_{invert}	$Carry_{In}$	Oper-ation
and	0	x	0
or	0	x	1
add	0	0	2
sub	1	1	2
beq	1	1	2
slt	1	1	3

ALU control bits

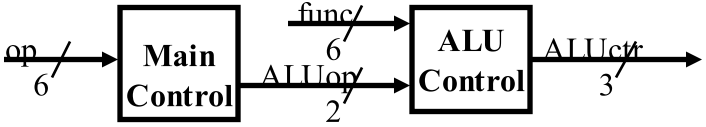
Note – book presents a 6-function ALU and a fourth ALU control input bit that never gets used (in simplified MIPS machine).

Don't let that confuse you.

- Recall: 5-function ALU

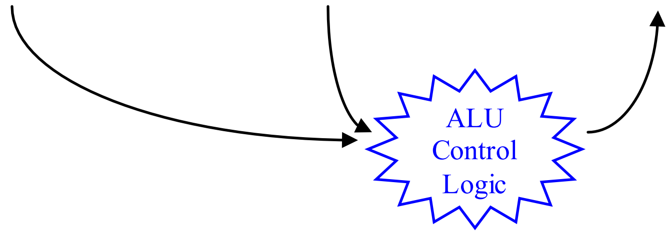
ALU control input	Function	Operations
000	And	and
001	Or	or
010	Add	add, lw, sw
110	Subtract	sub, beq
111	Slt	slt

- based on opcode (bits 31-26) and function code (bits 5-0) from instruction
- ALU doesn't need to know all opcodes!
  - Can summarize opcode with ALUOp (2 bits): 00 - lw,sw 01 - beq 10 - R-format



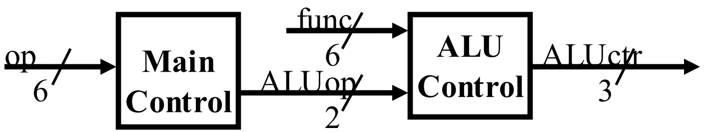
Generating ALU control

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
lw	00	load word	xxxxxx	add	010
sw	00	store word	xxxxxx	add	010
beq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	slt	101010	slt	111



Generating individual ALU signals

ALUop	Function	ALUCtr signals
00	xxxx	010
01	xxxx	110
10	0000	010
10	0010	110
10	0100	000
10	0101	001
10	1010	111

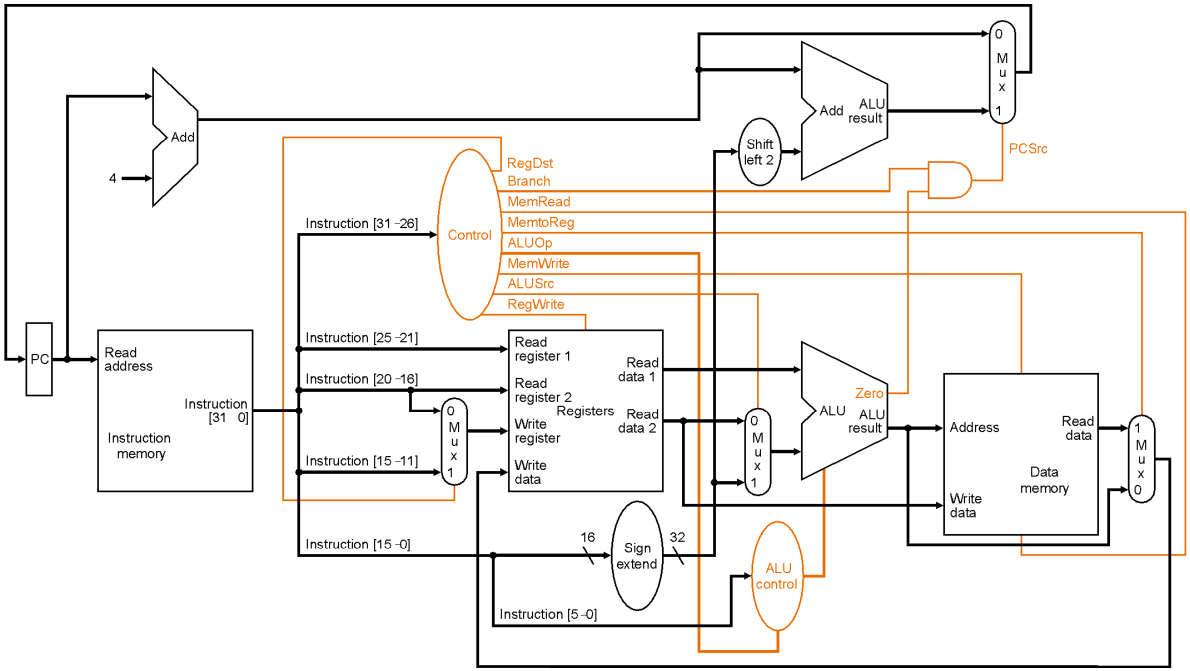


ALUctr2 = (!ALUop1 & ALUop0) | (ALUop1 & Func1)

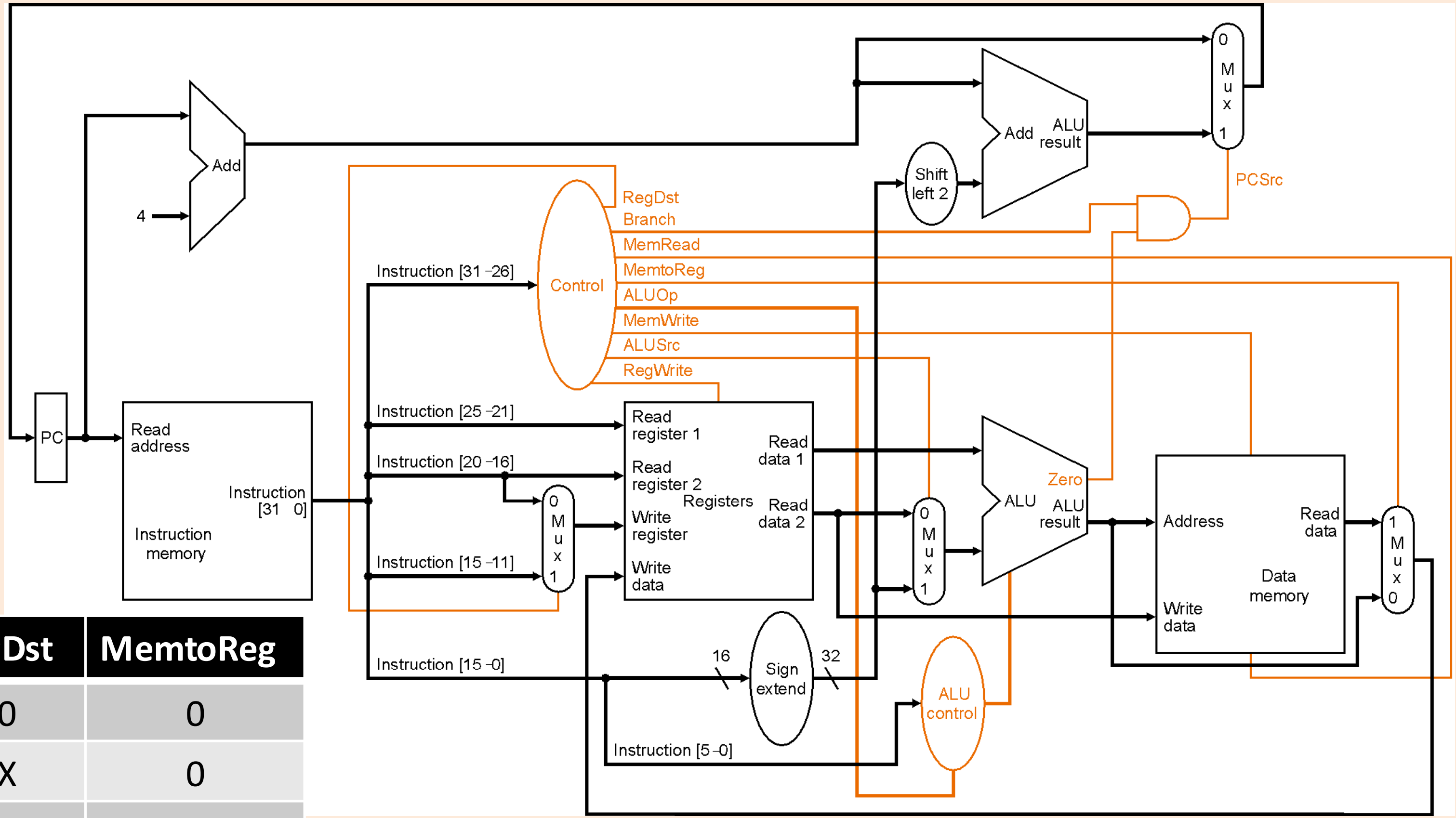
ALUctr1 = !ALUop1 | (ALUop1 & !Func2)

ALUctr0 = ALUop1 & (Func0 | Func3)

R-Format Instructions (e.g., add $d, $s0, $s1)



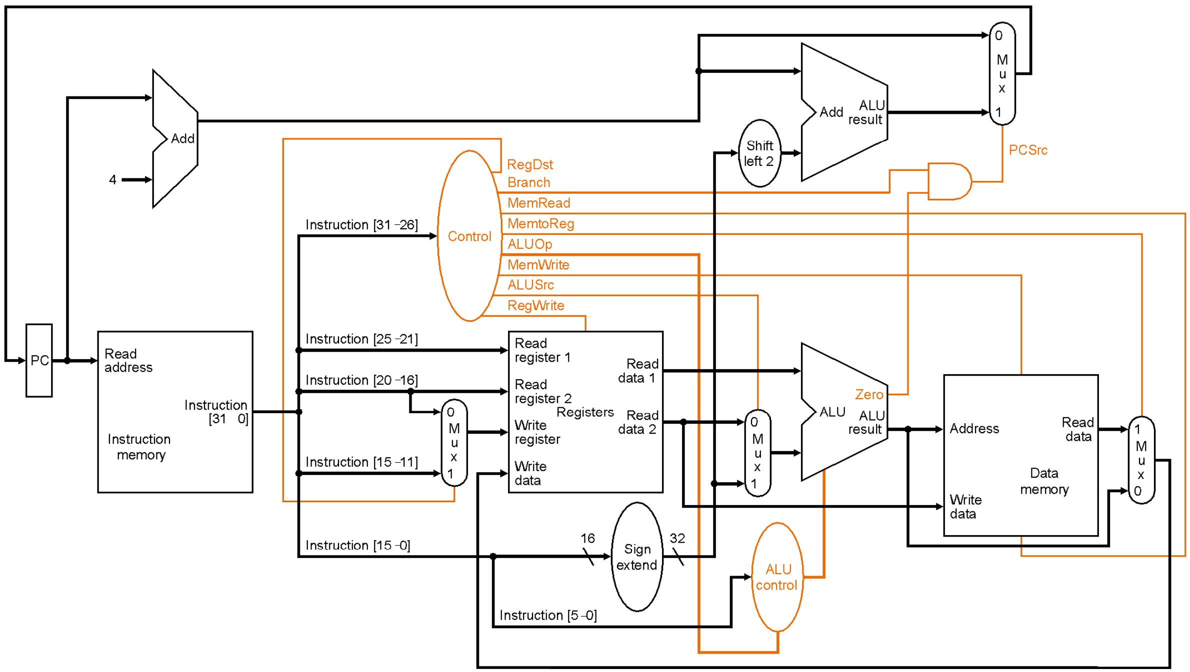
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format								1	0
lw								0	0
sw								0	0
beq								0	1



lw instruction control signals?

	ALUSrc	RegDst	MemtoReg
A	0	0	0
B	1	X	0
C	1	0	1
D	1	1	1
E	None of the above		

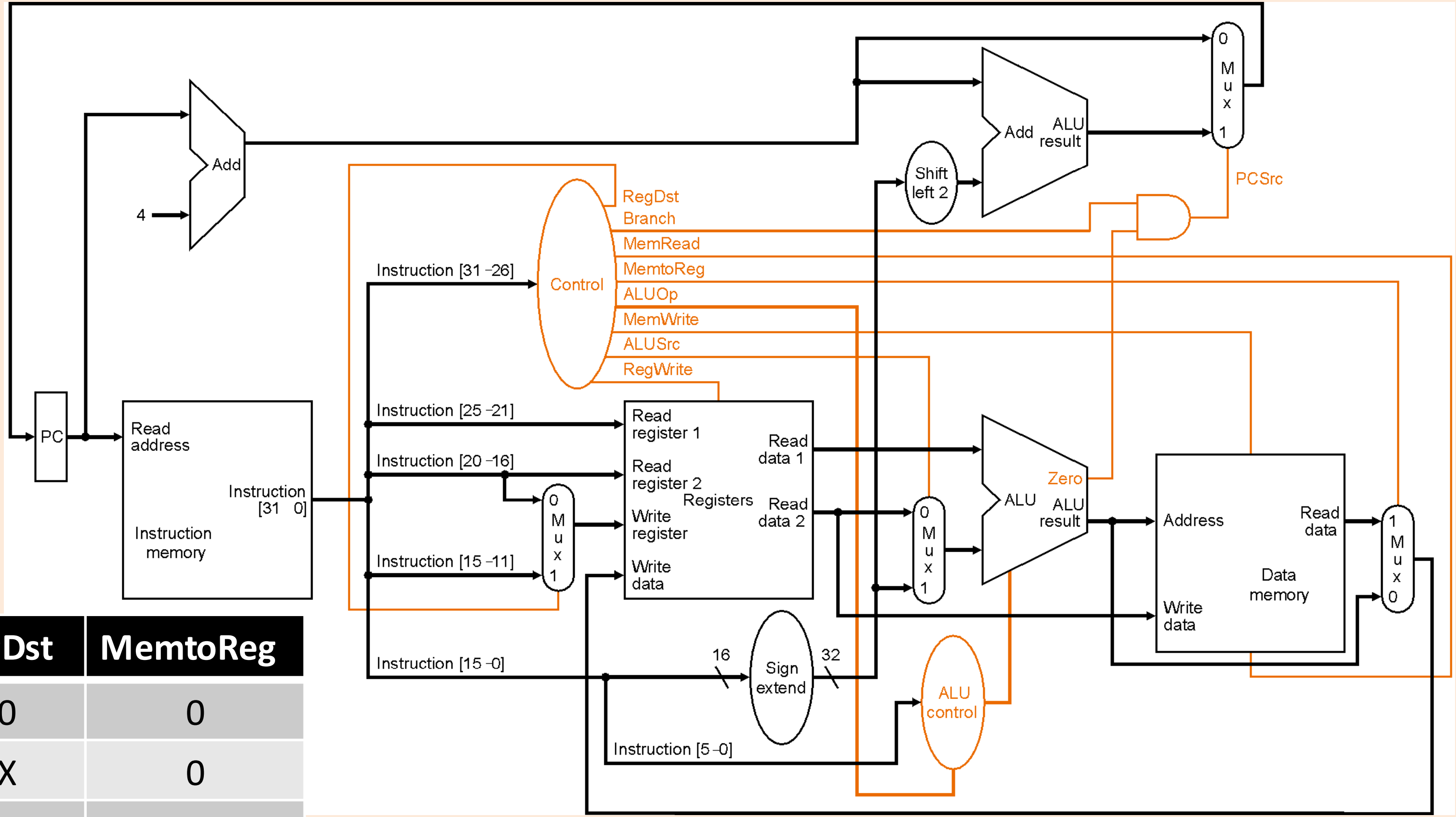
lw Control



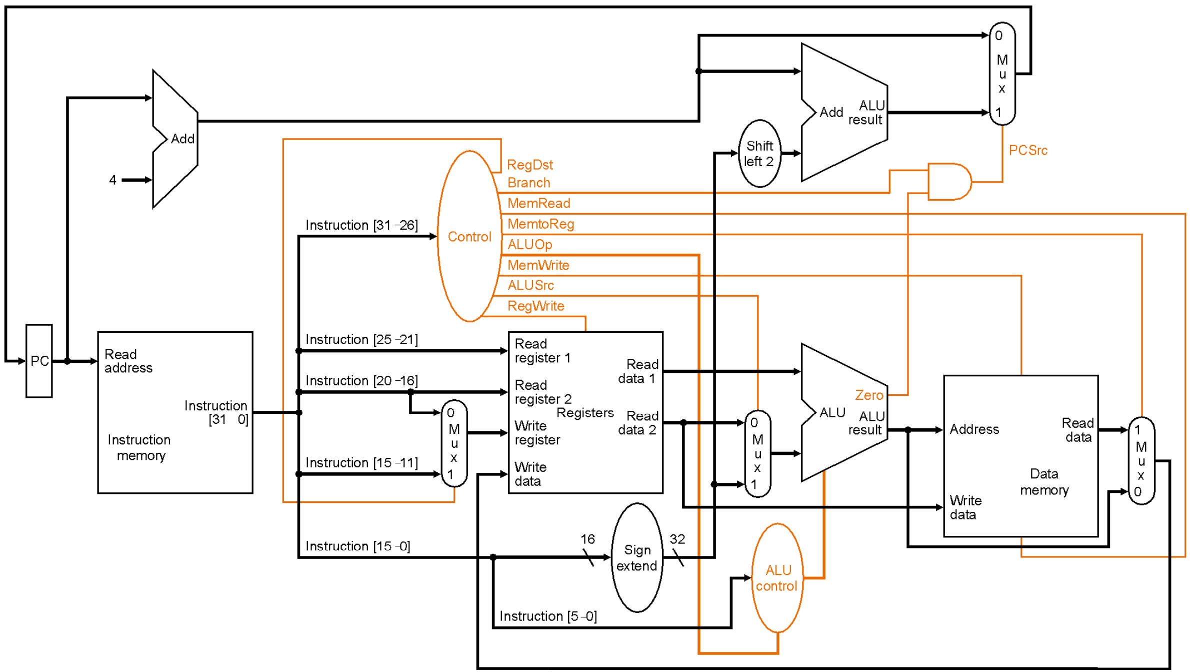
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw								0	0
sw								0	0
beq								0	1

sw instruction control signals?

	ALUSrc	RegDst	MemtoReg
A	0	0	0
B	1	X	0
C	0	0	X
D	1	X	1
E	None of the above		

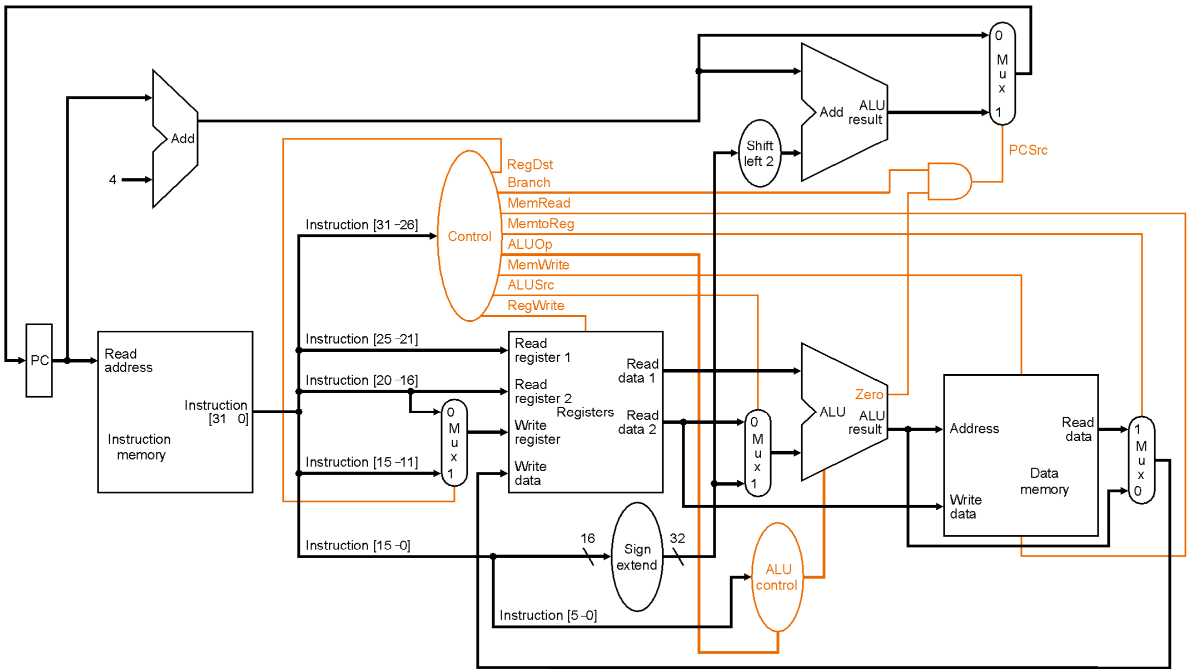


sw Control



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw								0	0
beq								0	1

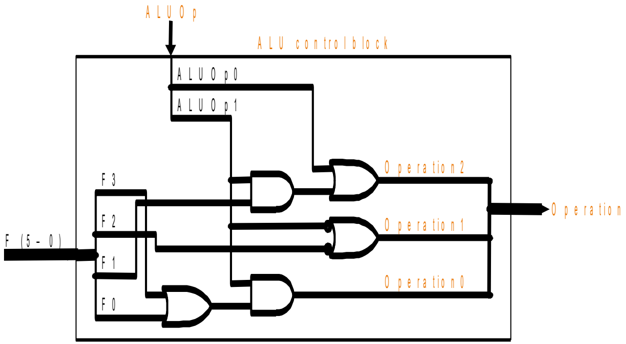
beq Control

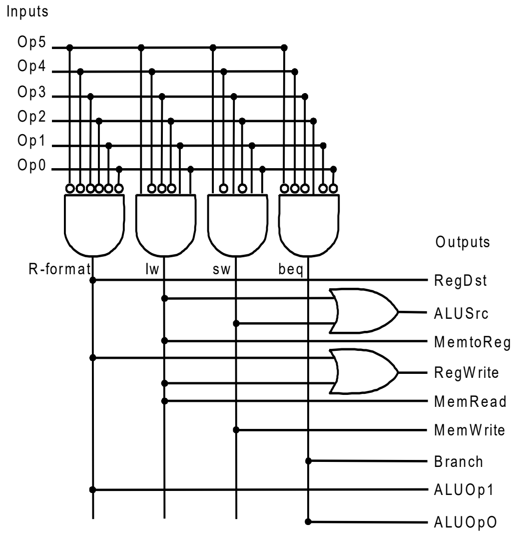


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`								0	1

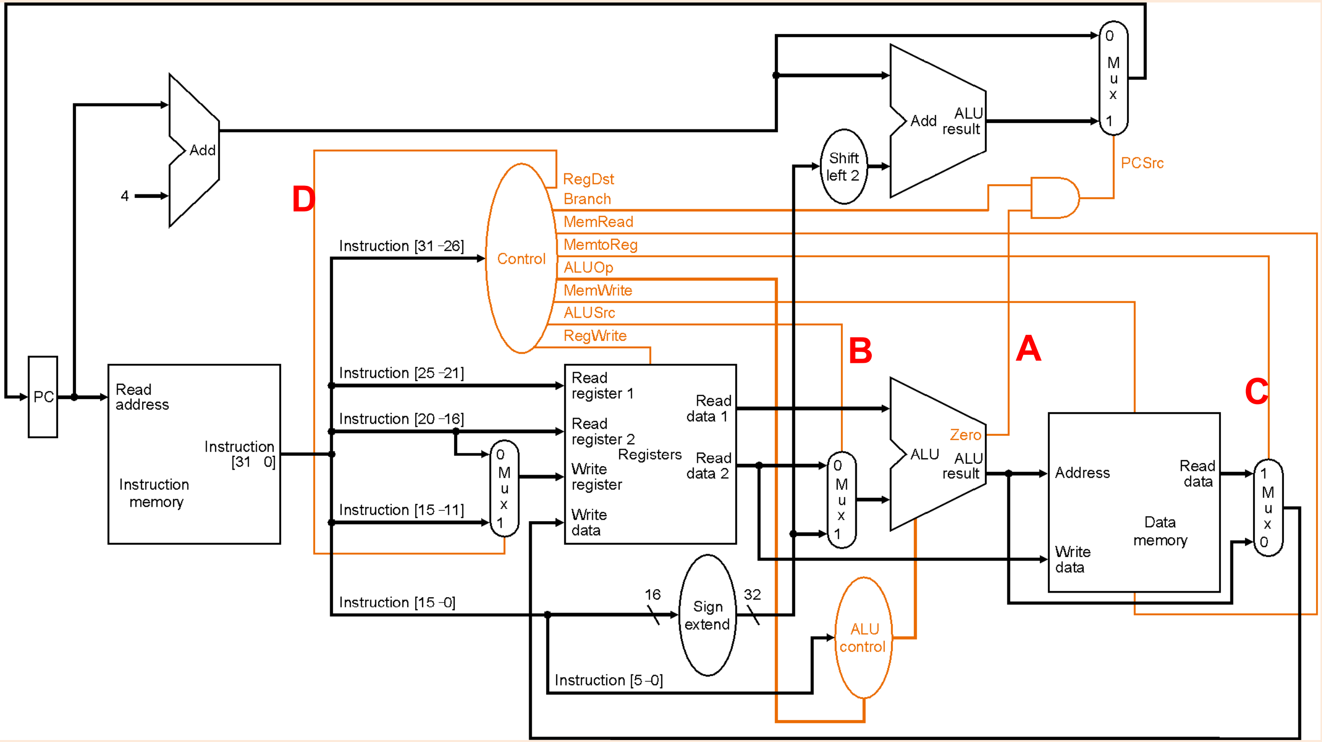
Control Truth Table

		R-format	lw	sw	beq
Opcode		000000	100011	101011	000100
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



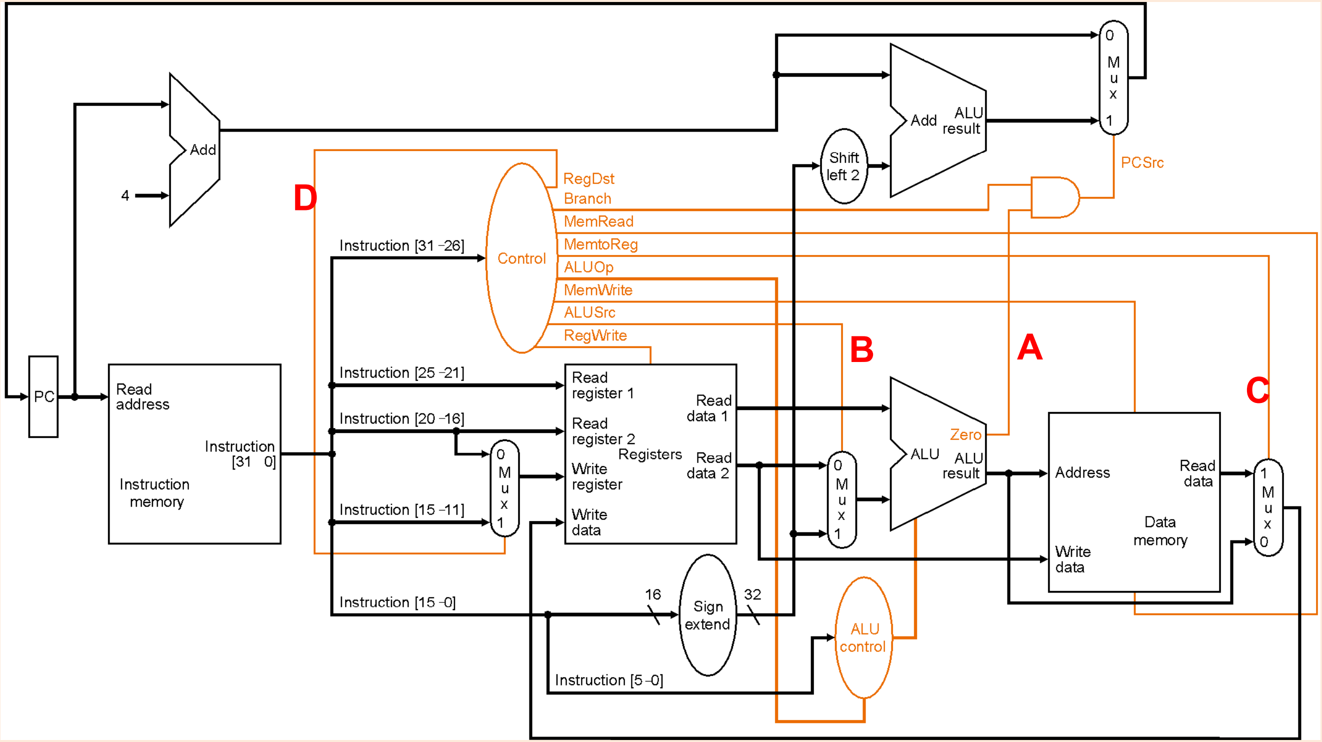


Which wire – if always ZERO – would break add?



A	ALU ‘iszero’ out
B	ALUsrc
C	MemtoReg
D	RegDst
E	None of these

Which wire – if always ONE – would break lw?



A	ALU 'iszero' out
B	ALUsrc
C	MemtoReg
D	RegDst
E	None of these

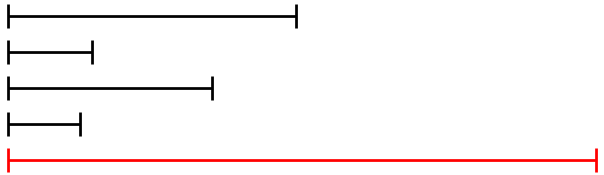
Single-Cycle CPU Summary

- Easy, particularly the control
- Which instruction takes the longest?
  - By how much? Why is that a problem?
- ET = IC * CPI * CT
- What else can we do?
- When does a **multi-cycle implementation** make sense?
  - e.g., 70% of instructions take 75 ns, 30% take 200 ns?
  - suppose 20% overhead for extra latches
- Real machines have ***much more*** variable instruction latencies than this.

Let’s think about this multicycle processor…

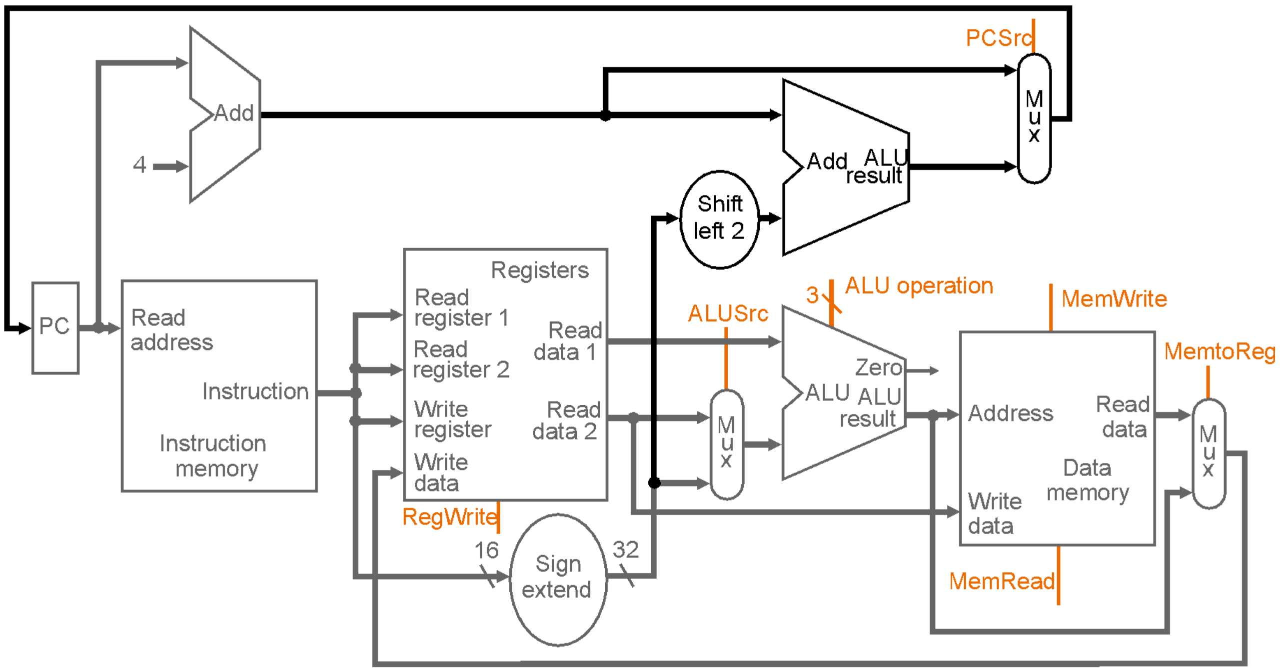
- (a very brief introduction…)

Why a Multiple Clock Cycle CPU?



- the problem => single-cycle cpu has a cycle time long enough to complete the longest instruction in the machine
- the solution => break up instruction execution into smaller tasks, each task taking one cycle
  - different instructions require different numbers of tasks (of cycles)
- other advantages => reuse of functional units (e.g., alu, memory)

High-level View



So Then,

- How many cycles does it take to execute
  - Add
  - BNE
  - LW
  - SW

- What about control logic?

- ET = IC * CPI * CT

Summary of instruction execution steps

"step" == "task" == "__________"

Step	R-type	Memory	Branch
Instruction Fetch	IR = Mem[PC] PC = PC + 4		
Instruction Decode/ register fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUout = PC + (sign-extend(IR[15-0]) << 2)		
Execution, address computation, branch completion	ALUout = A op B	ALUout = A + sign-extend(IR[15-0])	if (A==B) then PC=ALUout
Memory access or R-type completion	Reg[IR[15-11]] = ALUout	memory-data = Mem[ALUout] *or* Mem[ALUout]= B	
Write-back		Reg[IR[20-16]] = memory-data	

What is the fastest, slowest class of instruction in this MC machine?

Multicycle Questions

Recall	
R-type	4 cycles
Mem	5 cycles
Branch	3 cycles

- How many cycles will it take to execute this code?

```
  lw  $t2, 0($t3)
  lw  $t3, 4($t3)
  beq $t2, $t3, Label  #assume not taken
  add $t5, $t2, $t3
  sw  $t5, 8($t3)
Label:...
```

	How many cycles to execute these 5 instructions?
A	5
B	25
C	22
D	21
E	None of the above

Multi-Cycle CPU Summary

- Break up large instructions into smaller steps
  - Each step should be ~the same execution time (Why?)
- Saving work between steps is cheap, but not free
- More complex control, harder to reason about performance
  - But *worth it* — nearly all real-world machines were multi-cycle

Multicycle Implications

```
        lw  $t2, 0($t3)
        lw  $t3, 4($t3)
        #assume not taken
        beq $t2, $t3, Label
        add $t5, $t2, $t3
        sw  $t5, 8($t3)
Label:      ...
```

- What is the CPI of this program?
- What about a program that is 20% loads, 10% stores, 50% R-type, and 20% branches?

Single-Cycle, Multicycle CPU Summary

- Single-cycle CPU
  - CPI = 1, CT = **LONG**, simple design, simple control
  - No one has built a single-cycle machine in many decades
- Multi-cycle CPU
  - CPI > 1, CT = fairly short, complex control
  - Common up until maybe early 1990s, and dominant for many decades before that.