

Computer Architecture & Microprocessor System

RISC-V SINGLE CYCLE IMPLEMENTATION

Dennis A. N. Gookyi





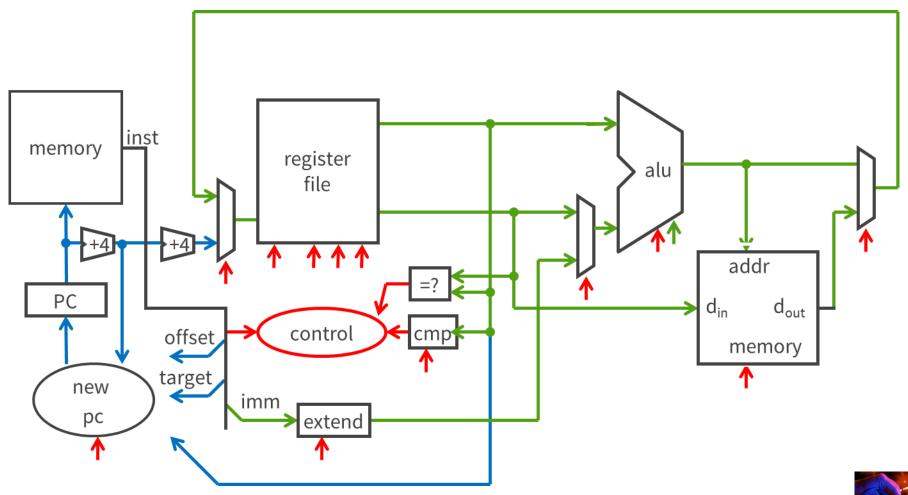
RISC-V Single Cycle Implementation





BIG PICTURE: BUILDING A PROCESSOR

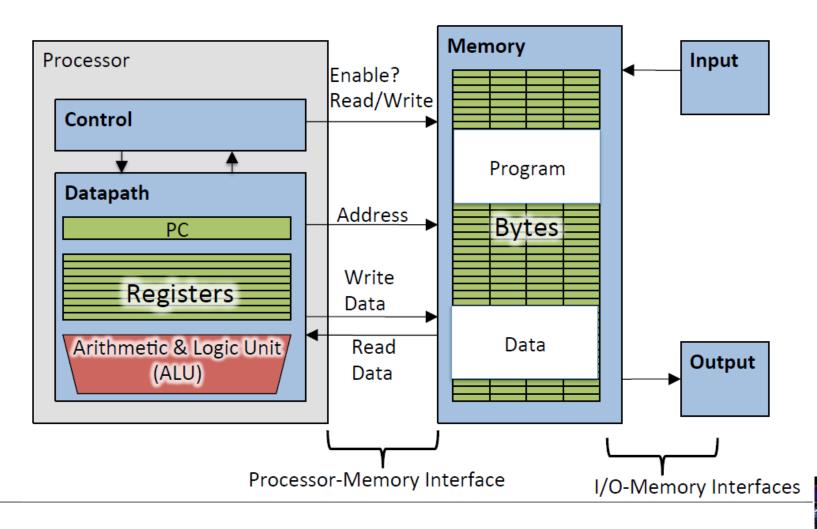
Single cycle processor





THE CPU

Components of a computer







THE CPU

Processor (CPU)

 The active part of the computer that does all the work (data manipulation and decision-making)

Datapath

 Portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)

Control

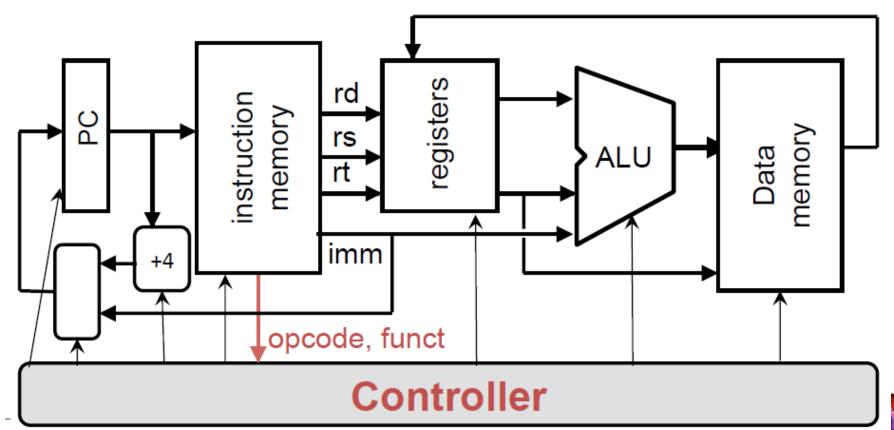
 Portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)





THE DATAPATH AND CONTROL

- Datapath designed to support data transfers required by instructions
- Controller causes correct transfers to happen





LOGIC DESIGN BASICS

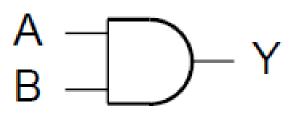
- Information encoded in binary
 - □ Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses
- Combinational circuit
 - Operate on data
 - Output is a function of input
- State (sequential) circuit
 - Store information



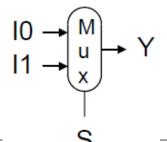


COMBINATIONAL CIRCUITS

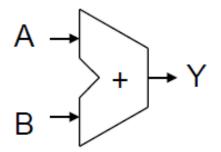
- AND gate
 - □ Y = A & B



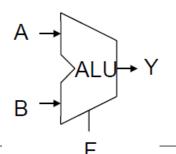
- Multiplexer
 - □ Y = S \$ |1 : |0



- AND gate
 - □ Y = A & B



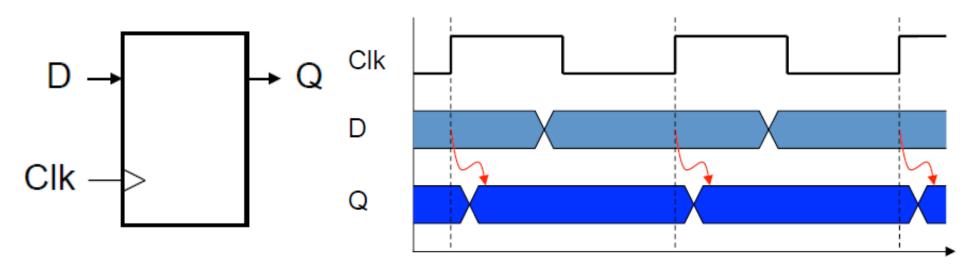
- Multiplexer
 - □ Y = S \$ |1 : |0





SEQUENTIAL CIRCUITS

- Register: stores data in a circuit
 - Uses a clock signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1

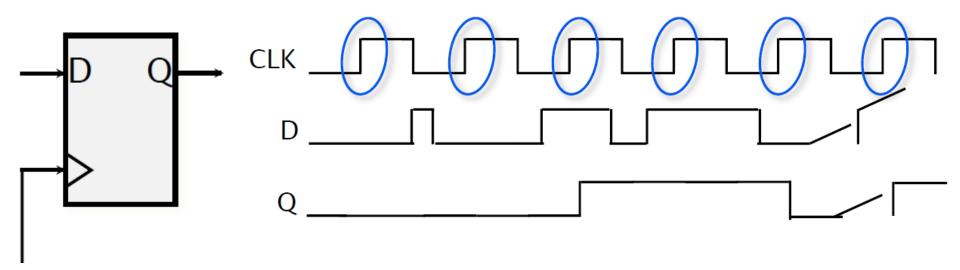






EDGE-TRIGGERED D FLIP FLOPS

- Value of D is sampled on positive clock edge
- Q outputs sampled value for rest of cycle

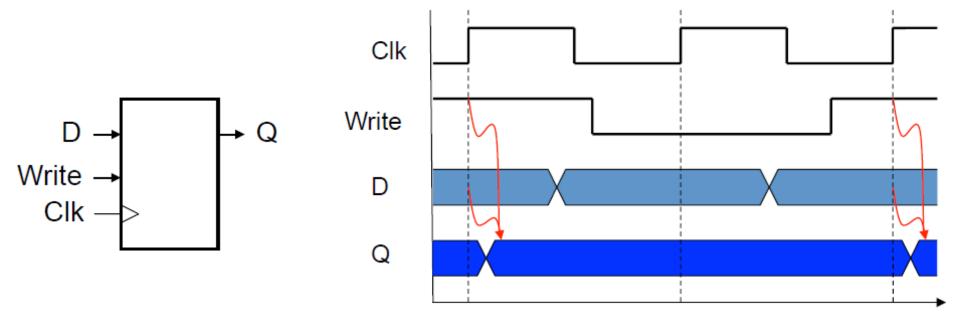






EDGE-TRIGGERED D FLIP FLOPS

- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later

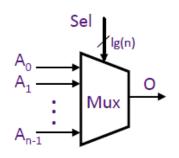


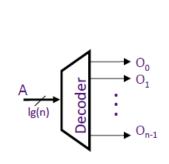


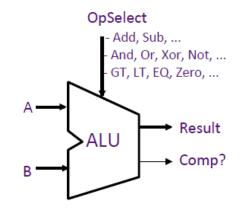


HARDWARE ELEMENTS OF CPU

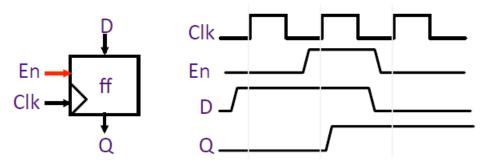
- Combinational circuits
 - Mux, Decoder, ALU, etc







- Synchronous state elements
 - Flipflop, Register, Register file, SRAM, DRAM



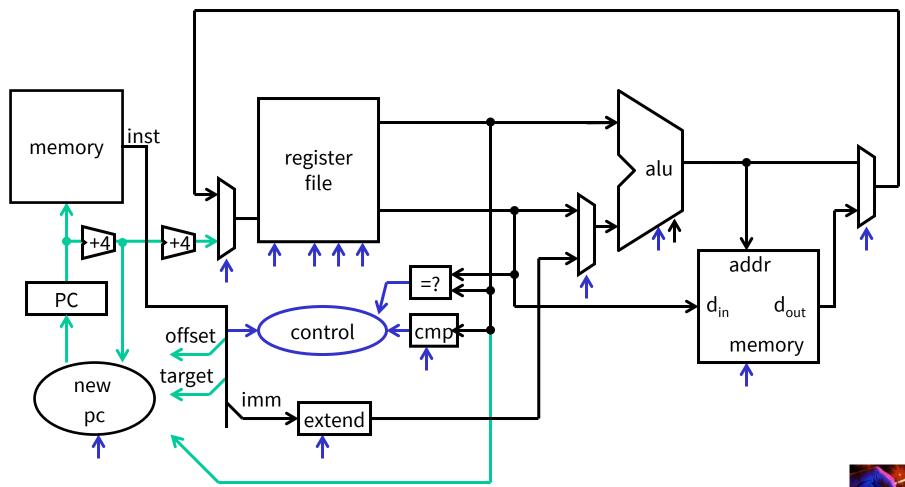
Edge-triggered: Data is sampled at the rising edge





BUILDING A PROCESSOR

* A single cycle processor





GOALS FOR THIS LECTURE

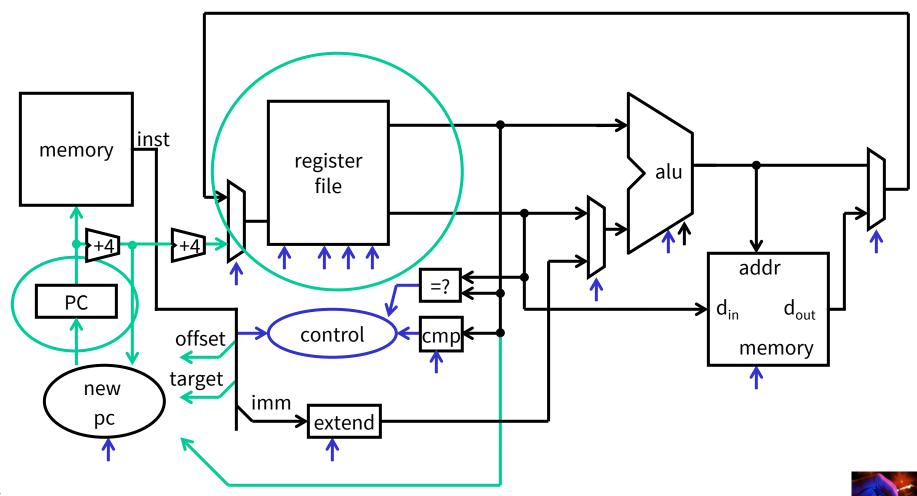
- Understanding the basics of a processor
 - We now have the technology to build a CPU
- Putting it all together:
 - Arithmetic Logic Unit (ALU)
 - Register File
 - Memory
 - SRAM: cache
 - DRAM: main memory
 - RISC-V Instructions & how they are executed





RISC V REGISTER FILE

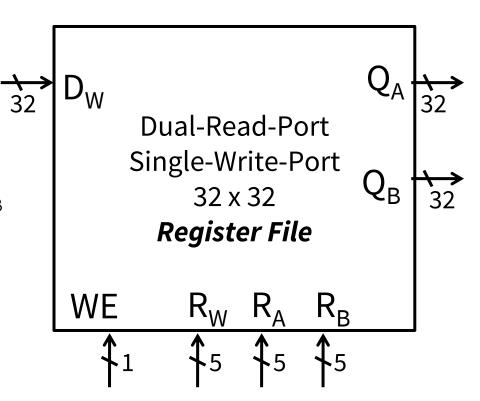
* A single cycle processor





RISC V REGISTER FILE

- RISC-V register file
 - 32 registers, 32-bits each
 - x0 wired to zero
 - Write port indexed via RW
 - on falling edge when WE=1
 - \square Read ports indexed via R_A , R_B

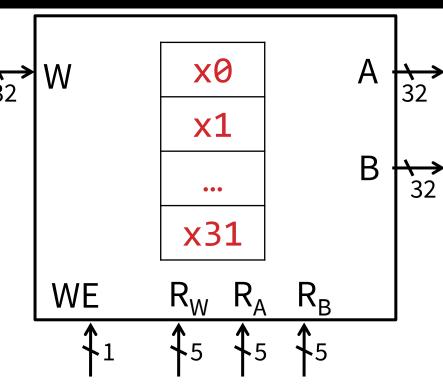






RISC V REGISTER FILE

- RISC-V register file
 - 32 registers, 32-bits each
 - x0 wired to zero
 - Write port indexed via RW
 - on falling edge when WE=1
 - \square Read ports indexed via R_A , R_B
- RISC-V register file
 - Numbered from 0 to 31
 - Can be referred by number: x0, x1, x2, ... x31
 - Convention, each register also has a name:
 - $x10 x17 \rightarrow a0 a7$, $x28 x31 \rightarrow t3 t6$



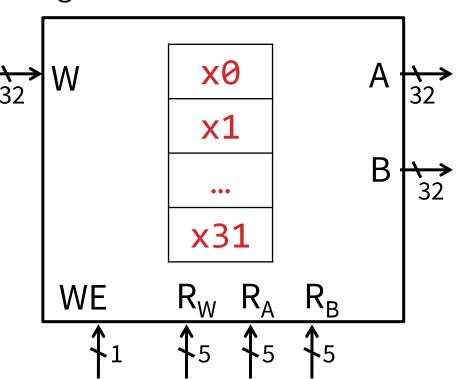




QUESTION?

If we wanted to support RV64 registers, what would change?

- ♦ (A) W, A, B → 64
- \Leftrightarrow (B) RW, RA, RB 5 \rightarrow 6
- \clubsuit (C) W 32 \rightarrow 64, RW 5 \rightarrow 6
- ♦ (D) A & B only



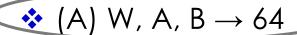




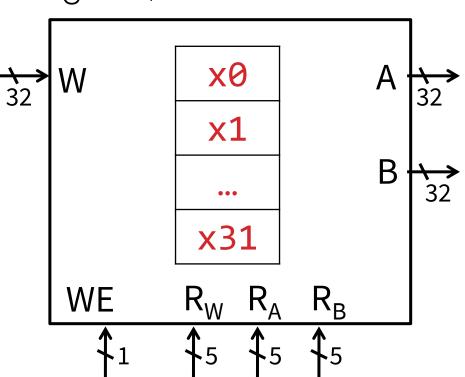
QUESTION?

If we wanted to support RV64 registers, what would

change?



- \Leftrightarrow (B) RW, RA, RB 5 \rightarrow 6
- **♦** (C) W 32 → 64, RW 5 → 6
- ♦ (D) A & B only

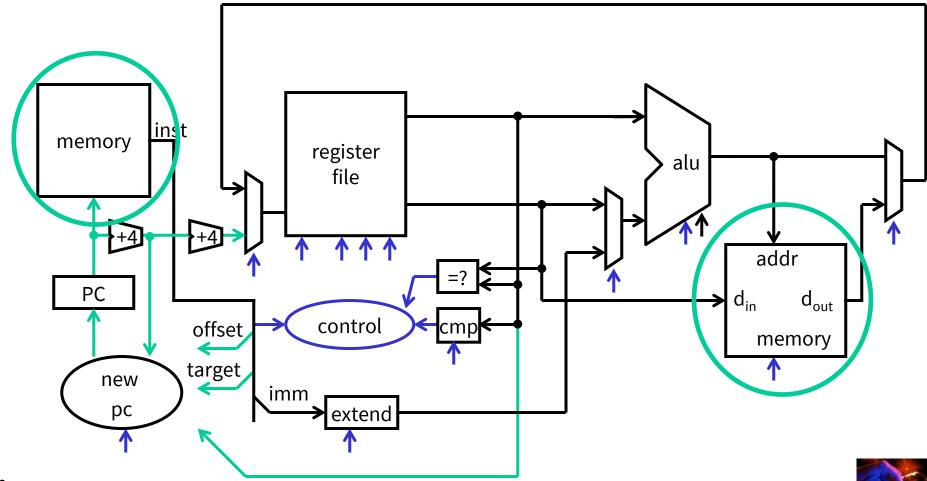






RISC V MEMORY

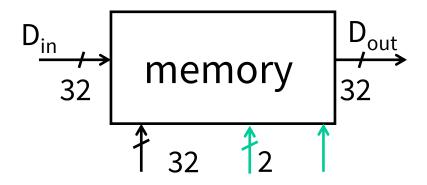
* A single cycle processor





RISC V MEMORY

- 32-bit address
- 32-bit data (but byte addressed)
- Enable + 2-bit memory control (mc)
 - 00: read word (4 byte aligned)
 - 01: write byte
 - 10: write halfword (2 byte aligned)
 - 11: write word (4 byte aligned)



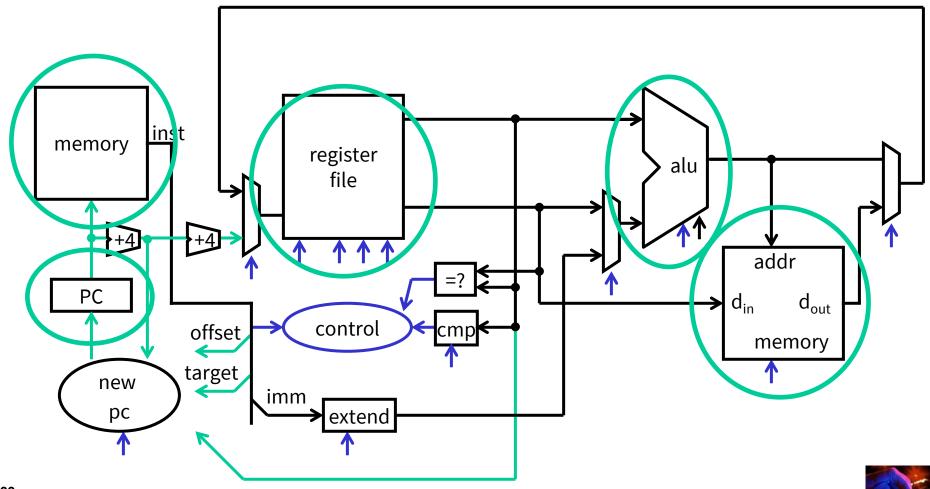
1 byte	address
	0x000fffff
	• • •
	0x0000000b
0x05	0x0000000a
	0x00000009
	0x00000008
	0x00000007
	0x00000006
	0x00000005
	0x00000004
	0x00000003
	0x00000002
	0x00000001
	0x00000000





OVERALL ARCHITECTURE: BASIC PROCESSOR

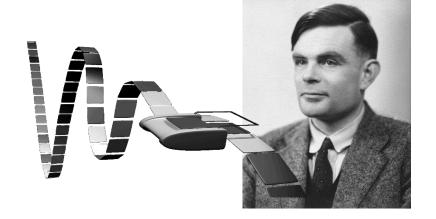
* A single cycle processor



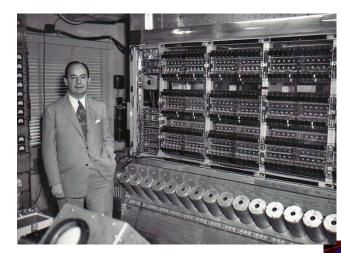


OVERALL ARCHITECTURE: BASIC PROCESSOR

- To make a computer
 - Need a program
 - Stored program computer



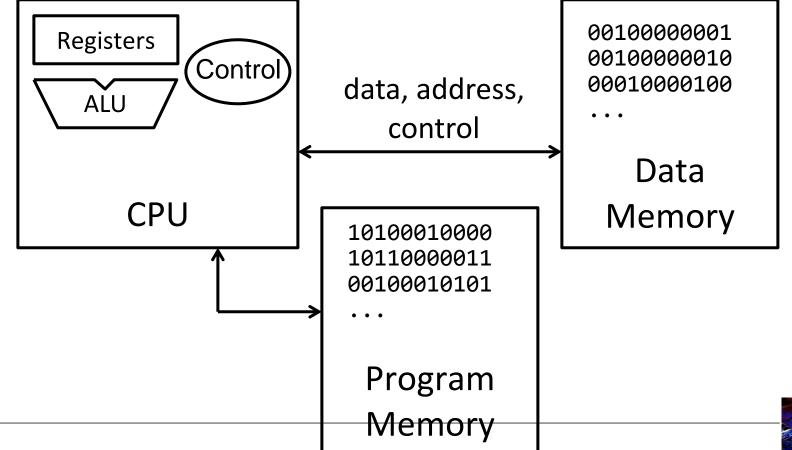
- Architectures
 - Von Neumann architecture
 - Harvard (modified) architecture





OVERALL ARCHITECTURE: BASIC PROCESSOR

- * A RISC-V CPU with a (modified) Harvard architecture
 - Modified: instructions & data in common address space, separate instr/data caches can be accessed in parallel







NEXT GOAL

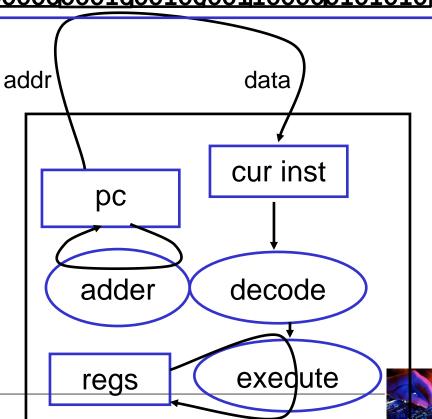
How to program and execute instructions on a RISC-V processor?





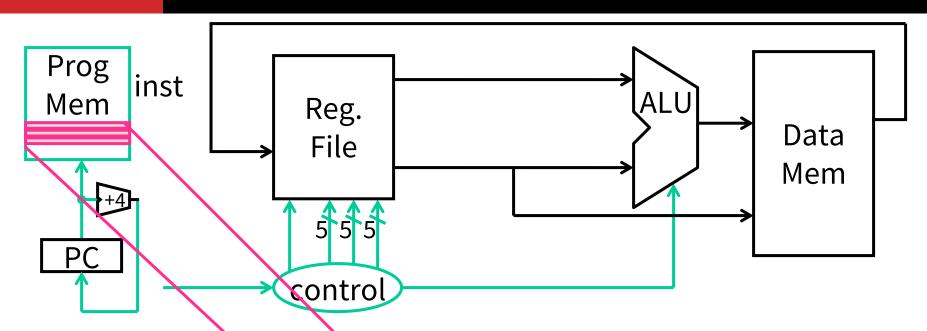
INSTRUCTION USAGE

- Instructions are stored in memory, encoded in binary 10 x2 x0 op=addi
- A basic processor
 - fetches
 - decodes
 - executes
 - one instruction at a time





INSTRUCTION PROCESSING



- A basic processor
 - fetches
 - decodes
 - executes
 - one instruction at a time

Instructions:

stored in memory, encoded in binary

00100000000010000000000001010

0000000001000100001100000101010





LEVELS OF INTERPRETATION: INSTRUCTIONS

Levels

main: addi x2, x0, 10

addi x1, x0, 0

loop: slt x3, x1, x2

•••

10 x2 x0 op=addi

High Level Language

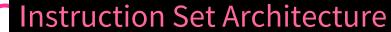
- C, Java, Python, ADA, ...
- Loops, control flow, variables

Assembly Language

- No symbols (except labels)
- One operation per statement
- "human readable machine language"

Machine Language

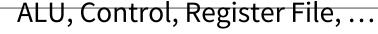
- Binary-encoded assembly
- Labels become addresses
- The language of the CPU



·

Machine Implementati

(Microarchitecture)





INSTRUCTION SET ARCHITECTURE (ISA)

- Different CPU architectures specify different instructions
- Two classes of ISAs
 - Reduced Instruction Set Computers (RISC)
 - IBM Power PC, Sun Sparc, MIPS, Alpha
 - Complex Instruction Set Computers (CISC)
 - Intel x86, PDP-11, VAX
- Another ISA classification: Load/Store Architecture
 - Data must be in registers to be operated on
 - For example: array[x] = array[y] + array[z]
 - 1 add?
 OR
 2 loads, an add, and a store?
 - □ Keeps HW simple → many RISC ISAs are load/store





QUESTION?

- What does it mean for an architecture to be called a load/store architecture?
 - (a) Load and Store instructions are supported by the ISA
 - (b) Load and Store instructions can also perform arithmetic instructions on data in memory
 - (c) Data must first be loaded into a register before it can be operated on
 - (d) Every load must have an accompanying store at some later point in the program





ANSWER

- What does it mean for an architecture to be called a load/store architecture?
 - (a) Load and Store instructions are supported by the ISA
 - (b) Load and Store instructions can also perform arithmetic instructions on data in memory
 - (c) Data must first be loaded into a register before it can be operated on
 - (d) Every load must have an accompanying store at some later point in the program





NEXT GOAL

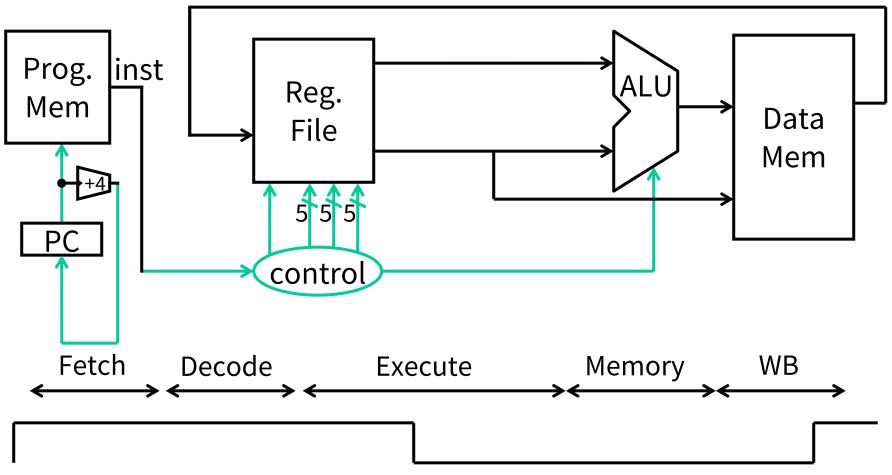
- How are instructions executed?
- What is the general datapath to execute an instruction?





FIVE STAGES OF RISC V DATAPATH

Stages of RISC V datapath



33 A single cycle processor – this diagram is not 100% spatial





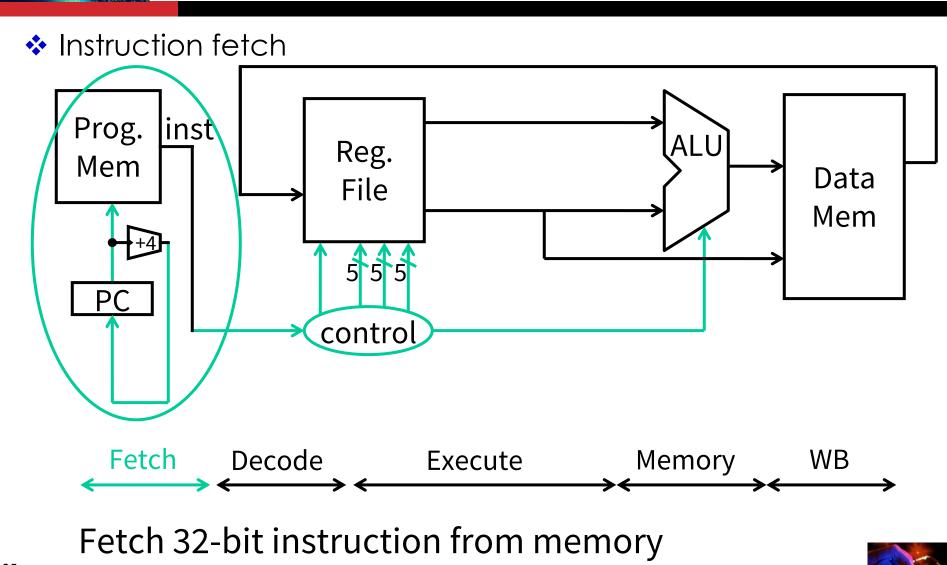
FIVE STAGES OF RISC V DATAPATH

- Basic CPU execution loop
 - Instruction Fetch
 - Instruction Decode
 - □ Execution (ALU)
 - ☐ Memory Access
 - Register Writeback





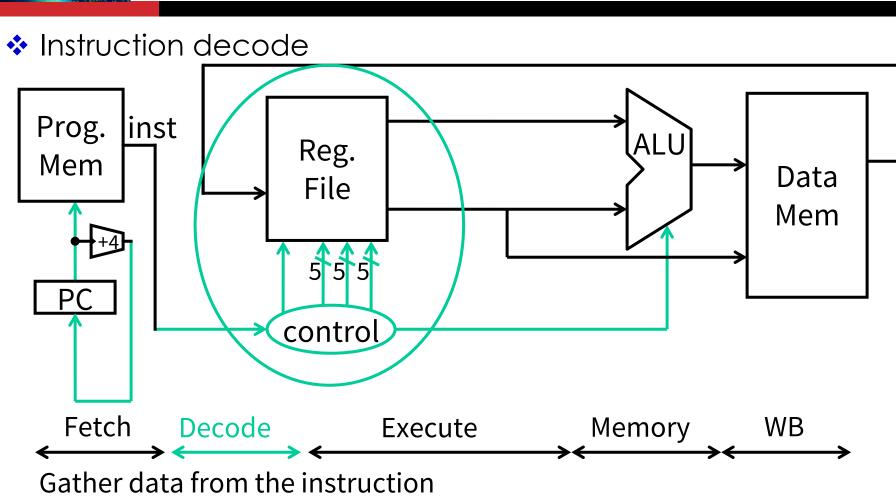
STAGE 1: INSTRUCTION FETCH



Increment PC = PC + 4



STAGE 2: INSTRUCTION DECODE



Read opcode; determine instruction type, field lengths

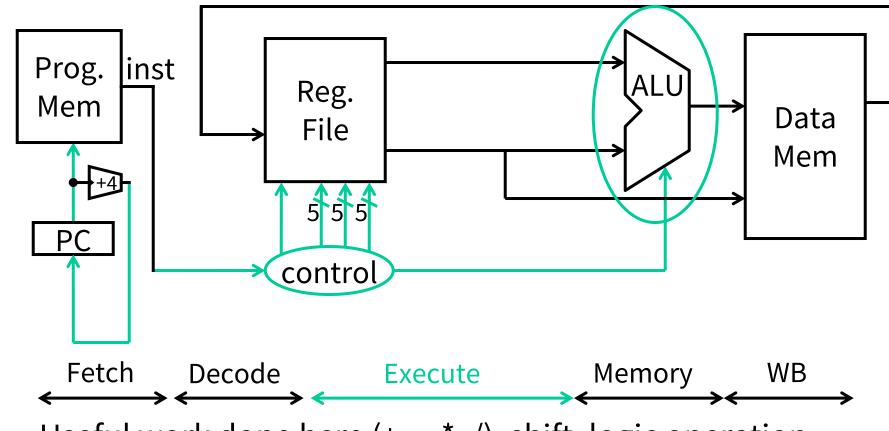
Read in data from register file

(0, 1, or 2 reads for jump, addi, or add, respectively)



STAGE 3: EXECUTION (ALU)





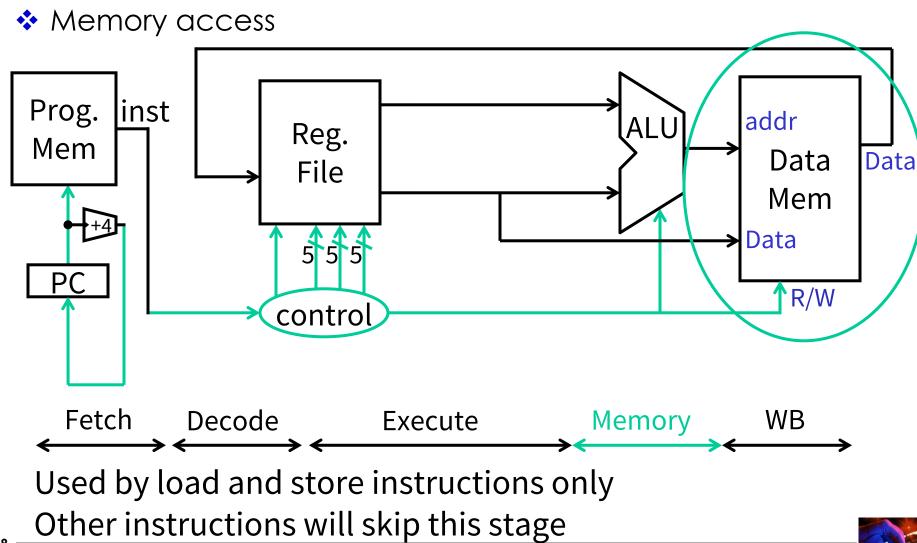
Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)

Load/Store? lw x2, x3, 32 → Compute address



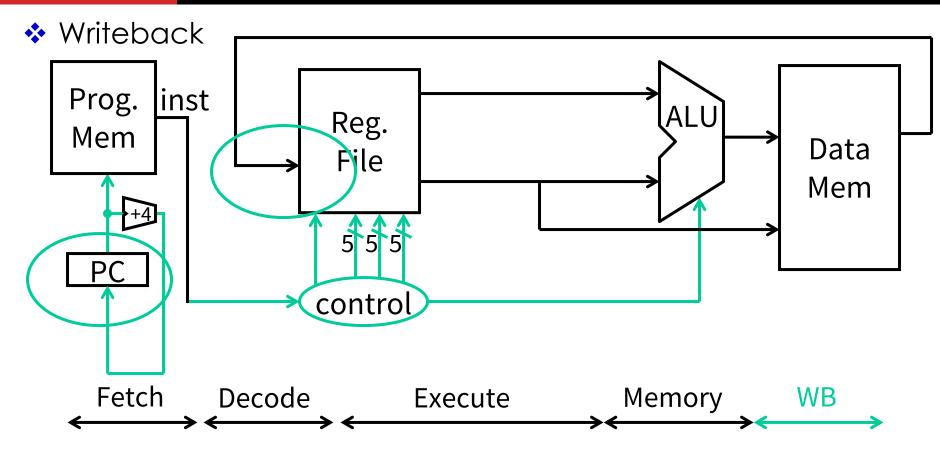


STAGE 4: MEMORY ACCESS





STAGE 5: WRITEBACK



Write to register file

- For arithmetic ops, logic, shift, etc, load. What about stores?
 Update PC
- For branches, jumps



QUESTION?

- Which of the following statements is true?
 - (A) All instructions require an access to Program Memory
 - □ (B) All instructions require an access to Data Memory
 - (C) All instructions write to the register file
 - □ (D) Some RISC-V instructions are shorter than 32 bits
 - □ (E) A & C





ANSWER

- Which of the following statements is true?
 - □ (A) All instructions require an access to Program Memory
 - □ (B) All instructions require an access to Data Memory
 - (C) All instructions write to the register file
 - □ (D) Some RISC-V instructions are shorter than 32 bits
 - □ (E) A & C





NEXT GOAL

Specific datapaths RISC-V Instructions





RISC V DESIGN PRINCIPLES

- Simplicity favors regularity
 - 32-bit instructions
- Smaller is faster
 - Small register file
- Make the common case fast
 - Include support for constants
- Good design demands good compromises
 - Support for different types of interpretations/classes





INSTRUCTION TYPES

- Arithmetic
 - add, subtract, shift left, shift right, multiply, divide
- Memory
 - load value from memory to a register
 - store value to memory from a register
- Control flow
 - conditional jumps (branches)
 - jump and link (subroutine call)
- Many other instructions are possible
 - vector add/sub/mul/div, string operations
 - manipulate coprocessor
 - □ I/O





RISC V INSTRUCTION TYPES

- Arithmetic/Logical
 - R-type: result and two source registers, shift amount
 - □ I-type: result and source register, shift amount in 6-bit immediate with sign/zero extension
 - □ U-type: result register, 12-bit immediate with sign/zero extension
- Memory Access
 - I-type for loads and S-type for stores
 - load/store between registers and memory
 - word, half-word and byte operations
- Control flow
 - □ UJ-type: jump-and-link
 - I-type: jump-and-link register
 - SB-type: conditional branches: pc-relative addresses





RISC V INSTRUCTION FORMAT

All RISC-V instructions are 32 bits long, have 4 formats							
	31 25	24 20	19 15	14 12	11 7	6 0	
□ R-type	funct7	rs2	rs1	Funct3	Rd	ор	
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
	31	20	19 15	14 12	11 7	6 0	
□ I-type	im	m	Rs1	Funct3	rd	ор	
□ 1-туре	12 bits		5 bits	3 bits	5 bits	7 bits	
	31 25	24 20	19 15	14 12	11 7	6 0	
□ S-type	imm	rs2	rs1	funct3	imm	Ор	
• (SB-type)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
□ U-type	31			12	11 7	6 0	
• (UJ-type	imm				rd	ор	
5	20 bits					7 bits	



R-TYPE (1): ARITHMETIC AND LOGIC

ADD and SUB have the same funct3, 000, but funct7 differs between the two, 0000000 vs 0100000, respectively

000000001100100010001000110011

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	Funct3	Rd	ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

ор	funct3	mnemonic	description
0110011	000	ADD rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
0110011	000	SUB rd, rs1, rs2	R[rd] = R[rs1] - R[rs2]
0110011	110	OR rd, rs1, rs2	R[rd] = R[rs1] R[rs2]
0110011	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$



R-TYPE (1): ARITHMETIC AND LOGIC

Arithmetic and Logic 000000001100100010001000110011

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	Funct3	Rd	ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

ор	funct3	mnemonic	description
0110011	000	ADD rd, rs1, rs2	R[rd] = R[rs1] + R[rs2]
0110011	000	SUB rd, rs1, rs2	R[rd] = R[rs1] - R[rs2]
0110011	110	OR rd, rs1, rs2	R[rd] = R[rs1] R[rs2]
0110011	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$

rd, rs1, rs2

Example: $x4 = x8 \oplus x6 \# XOR x4, x8, x6$





ARITHMETIC AND LOGIC

Arithmetic and Logic XOR x4, x8, x6 Prog. Reg. Mem File x8 ⊕ x6 PC contro XOR x4 x8 x6 Fetch Decode Memory WB Execute skip

Example: $x4 = x8 \oplus x6 \# XOR x4, x8, x6$

rd, rs1, rs2



R-TYPE (2): SHIFT INSTRUCTIONS

SRL and SRA have the same funct3, 101, but funct7 differs between the two, 0000000 vs 0100000, respectively

0000000011000100001010000110011

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	Funct3	Rd	ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

ор	funct3	mnemonic	description
0110011	001	SLL rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
0110011	101	SRL rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2] (zero ext.)
0110011	101	SRA rd, rs1, rs2	R[rd] = R[rt] >>> R[rs2] (sign ext.)





R-TYPE (2): SHIFT INSTRUCTIONS

Shift instructions

0000000011000100001010000110011

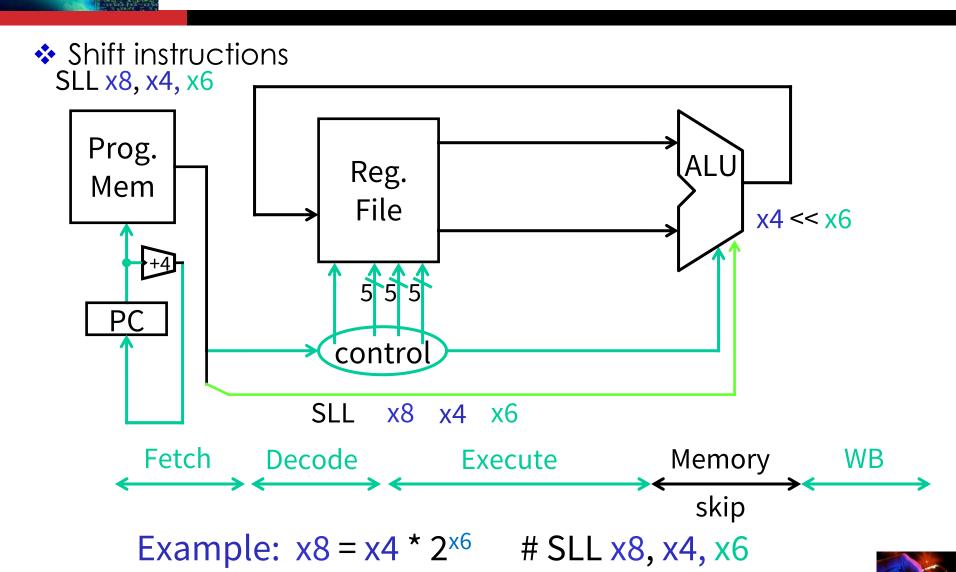
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	Funct3	Rd	ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

	ор	funct3	mnemonic	description
17	0110011	001	SLL rd, rs1, rs2	R[rd] = R[rs1] << R[rs2]
	0110011	101	SRL rd, rs1, rs2	R[rd] = R[rs1] >>> R[rs2] (zero ext.)
	0110011	101	SRA rd, rs1, rs2	R[rd] = R[rt] >>> R[rs2] (sign ext.)

Example: $x8 = x4 * 2^{x6}$ # SLL x8, x4, x6



S - E



$$x8 = x4 << x6$$



I-TYPE (1): ARITHMETIC WITH IMMEDIATES

Arithmetic with immediates

0000000010100101000001010010011

31 26	19 15	14 12	11 7	6 0
imm	rs1	funct3	rd	ор
12 bits	5 bits	3 bits	5 bits	7 bits

ор	funct3	mnemonic	description
0010011	000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + sign_extend(imm)$
0010011	111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
0010011	110	ORI rd, rs1, imm	$R[rd] = R[rs1] sign_extend(imm)$





I-TYPE (1): ARITHMETIC WITH IMMEDIATES

Arithmetic with immediates

0000000010100101000001010010011

31 20	1 9 1 5	14 12	11 7	6 0
imm	rs1	funct3	rd	ор
12 bits	5 bits	3 bits	5 bits	7 bits

ор	funct3	mnemonic	description
0010011	000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + sign_extend(imm)$
0010011	111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
0010011	110	ORI rd, rs1, imm	$R[rd] = R[rs1] sign_extend(imm)$

Example: x5 = x5 + 5

ADDI x5, x5, 5

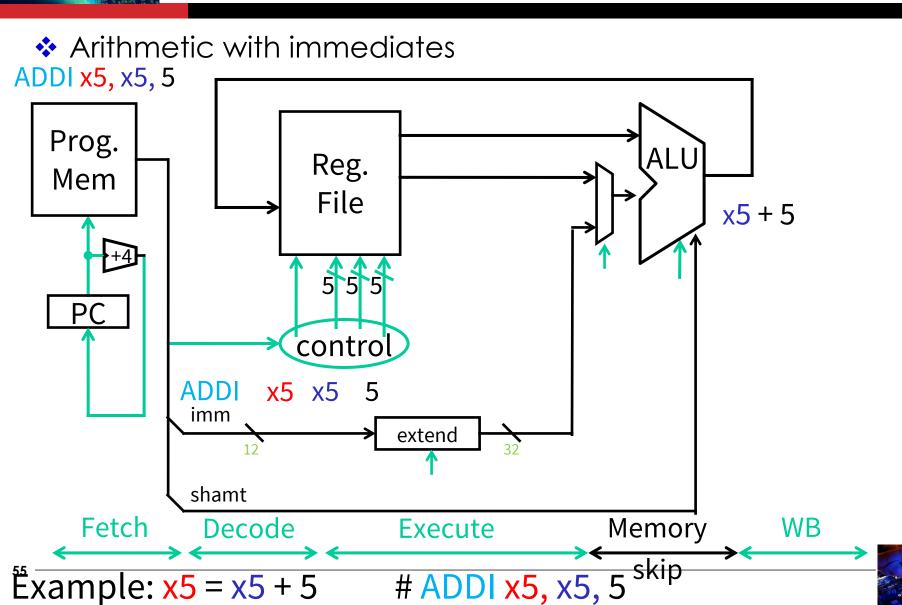
$$x5 += 5$$







ARITHMETIC WITH IMMEDIATES





QUESTION?

- To compile the code y = z + 1, assuming y is stored in X1 and z is stored in X2, you can use the ADDI instruction
- What is the largest number for which we can continue to use ADDI?
 - □ (A) 12
 - \square (B) $2^{12-1} 1 = 2,047$
 - \Box (C) $2^{12-1} = 4.095$
 - \square (D) $2^{16-1} = 65,535$
 - \Box (E) $2^{32-1} = \sim 4.3$ billion





QUESTION?

- To compile the code y = z + 1, assuming y is stored in X1 and z is stored in X2, you can use the ADDI instruction
- What is the largest number for which we can continue to use ADDI?
 - □ (A) 12
 - \Box (B) $2^{12-1} 1 = 2.047$
 - \Box (C) $2^{12-1} = 4.095$
 - \square (D) $2^{16-1} = 65,535$
 - \Box (E) $2^{32-1} = \sim 4.3$ billion





U-TYPE (1): LOAD UPPER IMMEDIATE

Load Upper Immediate

0000000000000000001010010110111

31			12	11 7	6	0
	imr	n		rd	ор	
	20 bi	ts		5 bits	7 bits	

ор	mnemonic	description
0110111	LUI rd, imm	R[rd] = sign_ext(imm) << 12





U-TYPE (1): LOAD UPPER IMMEDIATE

Load Upper Immediate

31			12	11 7	6 6
	imr	n		rd	ор
	20 bi	ts		5 bits	7 bits

ор	mnemonic	description
0110111	LUI rd, imm	R[rd] = sign_ext(imm) << 12

Example: x5 = 0x5000 # LUI x5, 5

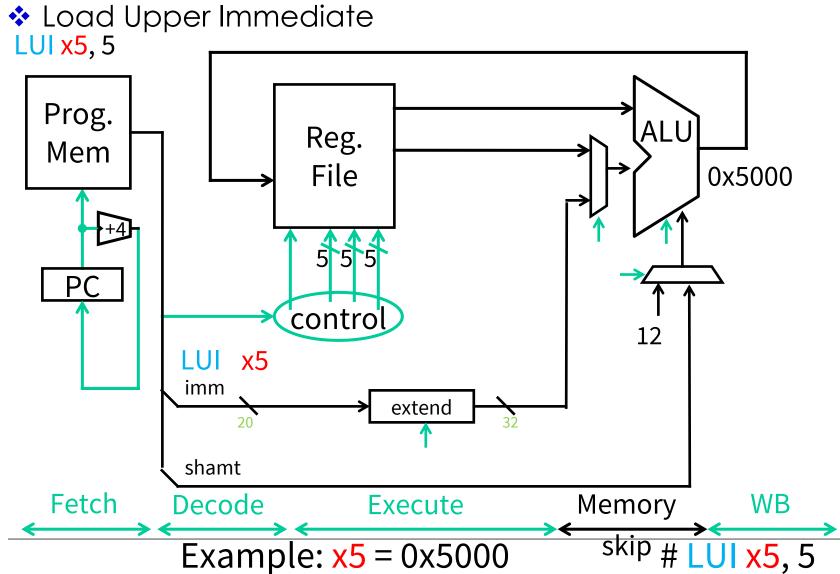
Example: LUI x5, 0xbeef1

ADDI x5, x5 0x234





U-TYPE (1): LOAD UPPER IMMEDIATE





RISC V INSTRUCTION TYPES

- Arithmetic/Logical
 - R-type: result and two source registers, shift amount
 - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
 - □ U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
 - I-type for loads and S-type for stores
 - load/store between registers and memory
 - word, half-word and byte operations
- Control flow
 - □ UJ-type: jump-and-link
 - □ I-type: jump-and-link register
 - SB-type: conditional branches: pc-relative addresses





I-TYPE (2): LOAD INSTRUCTIONS

Load instructions

99999999919999191919999911

<u> </u>	\overline{CCCCT}	-0001	\overline{O}	OOTO	OOOT.
31	20	19 15	14 12	11 7	6
im	m	rs1	funct3	rd	ор
12 b	its	5 bits	3 bits	5 bits	7 bits

base + offset addressing

ор	funct3	mnemonic	Description
0000011	000	LB rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	001	LH rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	010	LW rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	011	LD rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	100	LBU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	101	LHU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	110	LWU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]

signed offsets





I-TYPE (2): LOAD INSTRUCTIONS

Load instructions 0000000010000101010000010000011

31	20	19 15	14 12	11 7	6 0
im	m	rs1	funct3	rd	ор
12 b	its	5 bits	3 bits	5 bits	7 bits

base + offset addressing

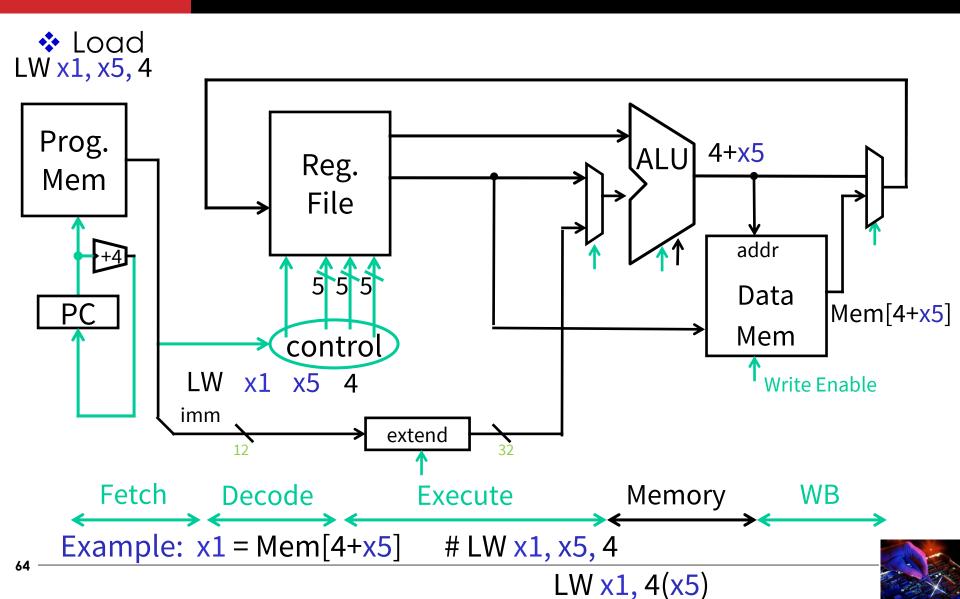
ор	funct3	mnemonic	Description
0000011	000	LB rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	001	LH rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	010	LW rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	011	LD rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	100	LBU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	101	LHU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]
0000011	110	LWU rd, rs1, imm	R[rd] = Mem[imm+R[rs1]]

Example: x1 = Mem[4+x5] # LW x1, x5, 4

offsets



MEMORY OPERATIONS: LOAD





S-TYPE (1): STORE INSTRUCTIONS

Store instructions

000010000000001010100000000010011

31 25	24 20	1 9 15	14 12	11 7	6 0
imm	rs2	rs1	funct3	imm	Ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

base + offset addressing

ор	funct3	mnemonic	description	
0100011	000	SB rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]	
0100011	001	SH rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]	
0100011	010	SW rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]	

signed offsets





S-TYPE (1): STORE INSTRUCTIONS

Store instructions 00001000000000101010000000010011

31 25	24 20	1 9 15	14 12	11 7	6 0
imm	rs2	rs1	funct3	imm	Ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

base + offset addressing

ор	funct3	mnemonic	description	
0100011	000	SB rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]	
0100011	001	SH rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd	
0100011	010	SW rs2, rs1, imm	Mem[sign_ext(imm)+R[rs1]] = R[rd]

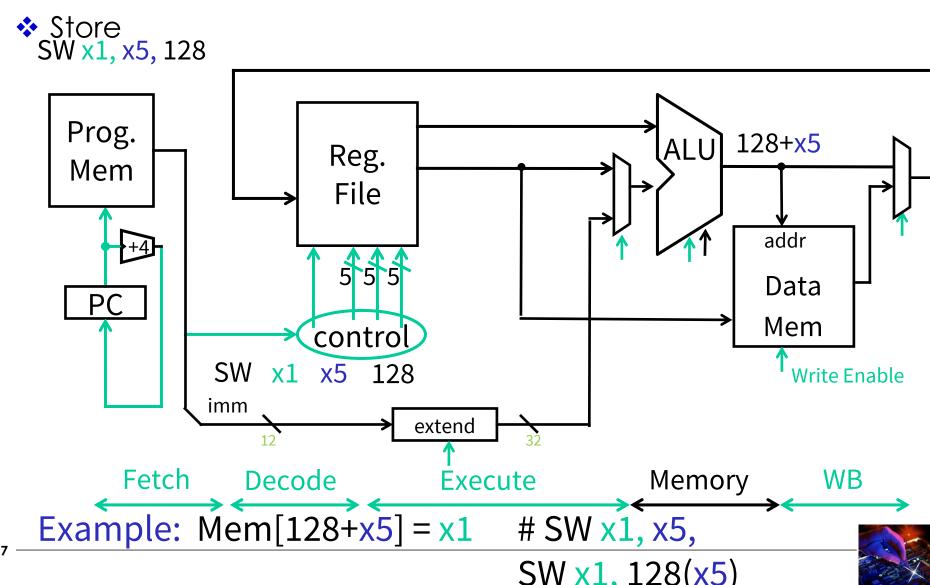
signed offsets

Example: Mem[128+x5] = x1 # SW x1, x5, 128





MEMORY OPERATIONS: STORE





MEMORY OPERATIONS: STORE

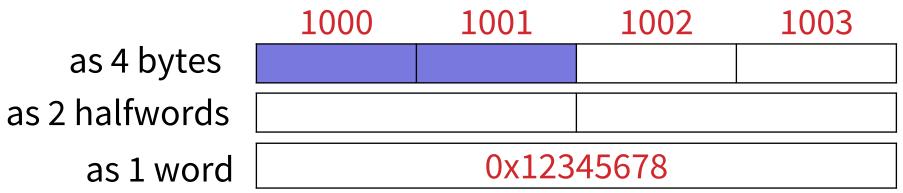
- # x5 contains 5 (0x00000005)
- ❖ SB x5, x0, 0
- ❖ SB x5, x0, 2
- ❖ SW x5, x0, 8
- Two ways to store a word in memory
- Endianness
 - Ordering of bytes within a memory word

,		
0x000fffff		
•••		
0x0000000b		
0x0000000a		
0x00000009		
0x00000008		
0x00000007		
0x00000006		
0x00000005		
0x00000004		
0x00000003		
0x00000002		
0x00000001		
0x00000000		



LITTLE ENDIAN

- Endianness: Ordering of bytes within a memory word
 - □ Little Endian = least significant part first (RISC-V, x86)



Question: What values go in the byte-sized boxes with addresses 1000 and 1001?

a) 0x8, 0x7

d) 0x12, 0x34

b) 0x78, 0x56

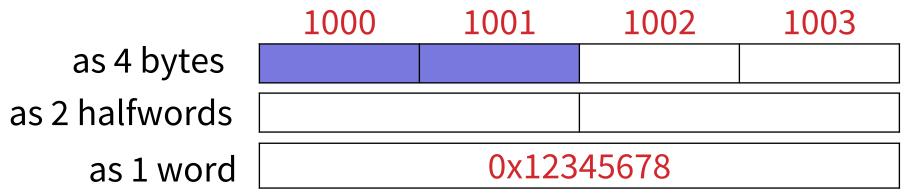
e) 0x1, 0x2

c) 0x87, 0x65



LITTLE ENDIAN

- Endianness: Ordering of bytes within a memory word
 - □ Little Endian = least significant part first (RISC-V, x86)



Question: What values go in the byte-sized boxes with addresses 1000 and 1001?

a) 0x8, 0x7

d) 0x12, 0x34

b) 0x78, 0x56

e) 0x1, 0x2

c) 0x87, 0x65



BIG ENDIAN

- Endianness: Ordering of bytes within a memory word
 - □ Big Endian = most significant part first (MIPS, networks)

	1000	1001	1002	1003	
as 4 bytes					
as 2 halfwords					
as 1 word	0x12345678				

Question: What values go in the half-word sized box with address 1000?

a) 0x1

d) 0x4321

b) 0x12

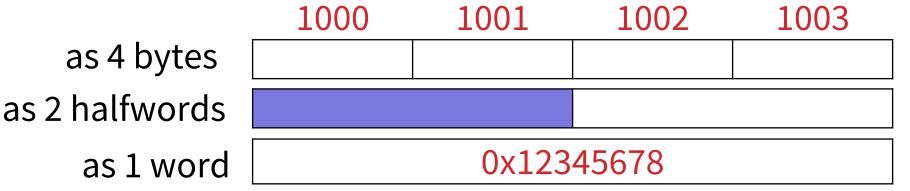
e) 0x5678

c) 0x1234

00101010101

BIG ENDIAN

- Endianness: Ordering of bytes within a memory word
 - Big Endian = most significant part first (MIPS, networks)



Question: What values go in the half-word sized box with address 1000?

a) 0x1

d) 0x4321

b) 0x12

e) 0x5678

Ox1234



LITTLE ENDIAN

- Little Endian = least significant part first (RISC-V, x86)
- Example:
 - □ r5 contains 5 (0x00000005)
 - □ SW r5, 8(r0)

- a) 0x00000008
- b) 0x00000009
- c) 0x0000000a
- d) 0x000000b
- e) I don't know

0x000fffff
• • •
0x0000000b
0x0000000a
0x00000009
0x00000008
0x00000007
0x00000006
0x00000005
0x00000004
0x00000003
0x00000002
0x00000001
 0x00000000





LITTLE ENDIAN

- Little Endian = least significant part first (RISC-V, x86)
- Example:
 - □ r5 contains 5 (0x00000005)
 - □ SW r5, 8(r0)

- a) 0x00000008
- b) 0x0000009
- c) 0x0000000a
- d) 0x000000b
- e) I don't know

0x000fffff
•••
0x0000000b
0x00000000a
0x00000009
0x00000008
0x00000007
0x00000006
0x00000005
0x00000004
0x00000003
0x00000002
0x00000001
0x00000000





BIG ENDIAN

- Big Endian = most significant part first (some MIPS, networks)
- Example:
 - r5 contains 5 (0x00000005)
 - □ SW r5, 8(r0)

- a) 0x00000008
- b) 0x00000009
- c) 0x0000000a
- d) 0x000000b
- e) I don't know

0x000fffff
•••
0x0000000b
0x00000000a
0x00000009
0x00000008
0x00000007
0x00000006
0x00000005
0x00000004
0x00000003
0x00000002
0x00000001
0×00000000



BIG ENDIAN

- Big Endian = most significant part first (some MIPS, networks)
- Example:
 - r5 contains 5 (0x00000005)
 - □ SW r5, 8(r0)

- a) 0x00000008
- b) 0x00000009
- c) 0x0000000a
- d) 0x0000000b
- e) I don't know

0x000fffff
•••
0x0000000b
0x00000000a
0x00000009
0x00000008
0x00000007
0x00000006
0x00000005
0x00000004
0x00000003
0x00000002
0x00000001
0×00000000



BIG ENDIAN MEMORY LAYOUT

0x00000000

The state of the s				0x000fffff
Layout		x0		
,		• • •	0x05	0x0000000b
	0x0000005	x 5	0x00	0x0000000a
	0x0000005	x6	0x00	0x00000009
	0x0000000	x7	0x00	0x00000008
	0x00000005			0x00000007
■ SB x5, x0, 2				0x00000006
■ LB x6, x0, 2				0x00000005
SW x5, x0, 8				0x00000004
■ LB x7, x0, 8				0x00000003
■ LB x7, x0, 0 ■ LB x8, x0, 1			0x05	0x00000002
	T			0x00000001



RISC V INSTRUCTION TYPES

- Arithmetic/Logical
 - R-type: result and two source registers, shift amount
 - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
 - □ U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
 - I-type for loads and S-type for stores
 - load/store between registers and memory
 - word, half-word and byte operations
- Control flow
 - □ UJ-type: jump-and-link
 - I-type: jump-and-link register
 - SB-type: conditional branches: pc-relative addresses





UJ-TYPE (2): JUMP AND LINK

Jump and Link

000000000000000001000001011101111

31			12	11 7	6	0
	imr	n		rd	ор	
	20 bi	ts		5 bits	7 bits	

ор	Mnemonic	Description
1101111	JAL rd, imm	R[rd] = PC+4; PC=PC + sext(imm)

Why?

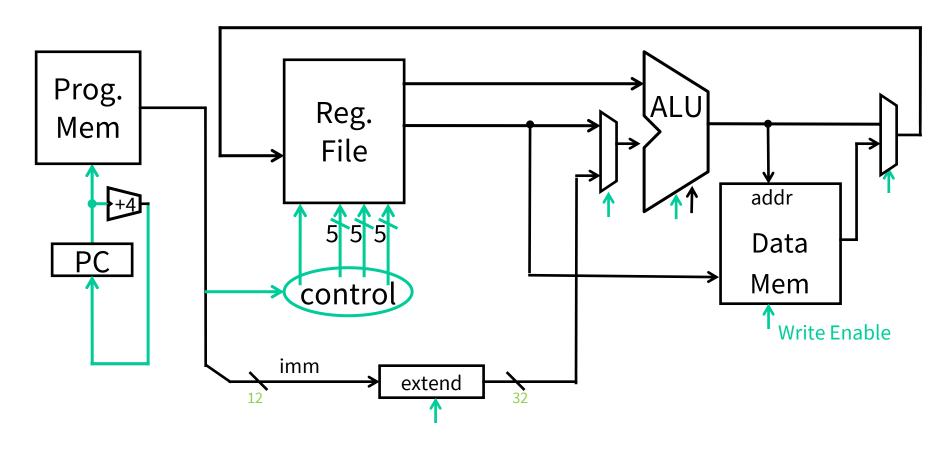
Function/procedure calls





JUMP AND LINK

Jump and Link



Example:

$$x5 = PC + 4$$

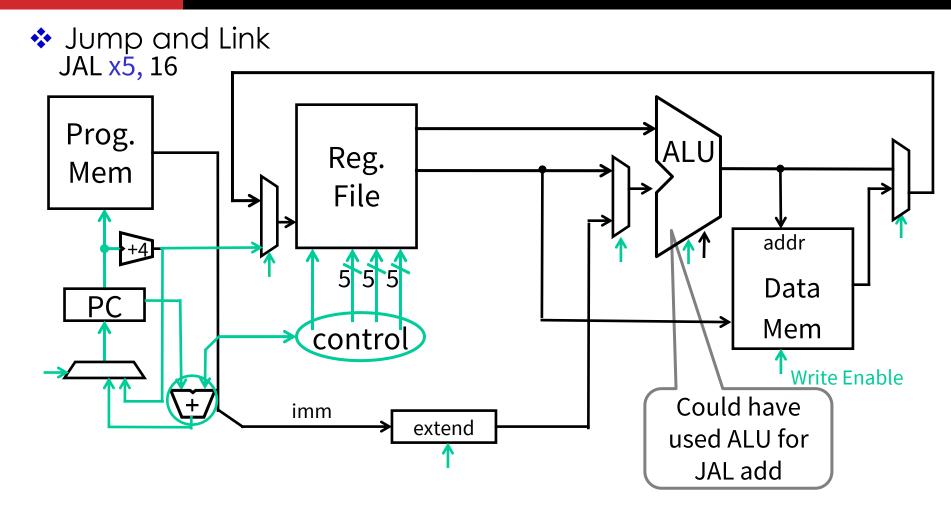
JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)





JUMP AND LINK



Example:

$$x5 = PC + 4$$

JAL x5, 16

PC = PC + 16 (i.e. 16 == 8<<1)





I-TYPE (3): JUMP AND LINK REGISTER

Jump and Link Register

0000001000000100000001011100111

31	20	19 15	14 12	11 7	6 0
im	m	Rs1	funct3	rd	ор
12 b	its	5 bits	3 bits	5 bits	7 bits

ор	funct3	Mnemonic	Description
1100111	000	, ,	R[rd] = PC+4; PC=(R[rs1]+sign_ex(imm))&0xfffffffe

Example: x5 = PC+4

PC = x4 + 16

JALR x5, x4, 16

Why?

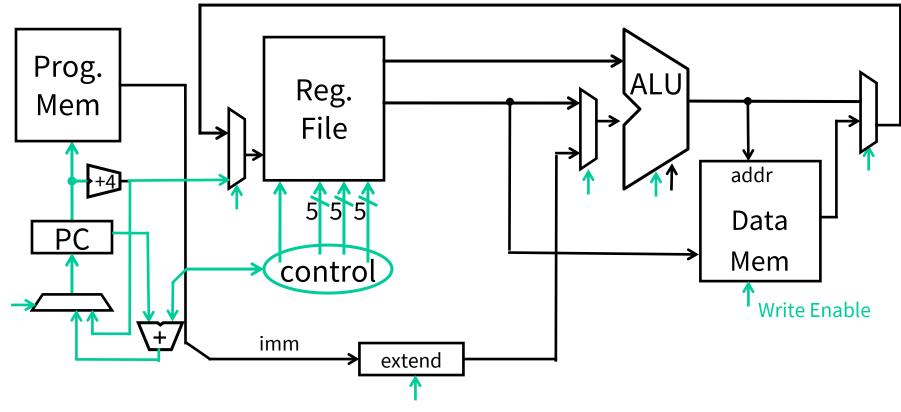
Function/procedure calls





JUMP AND LINK REGISTER

Jump and Link Register



Example:

$$x5 = PC + 4$$

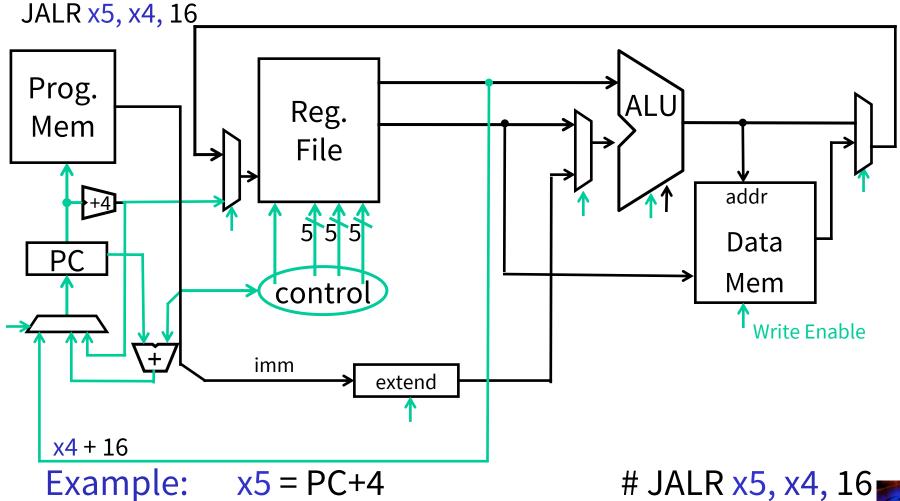
$$PC = x4 + 16$$

JALR x5, x4, 16



JUMP AND LINK REGISTER

Jump and Link Register



PC = x4 + 16





MOVING BEYOND JUMPS

- Can use Jump and link (JAL) or Jump and Link Register (JALR) instruction to jump to 0xabcd1234
- What about a jump based on a condition?
 - □ # assume 0 <= x3 <= 1
 </p>
 - if (x3 == 0) jump to 0xdecafe00 else jump to 0xabcd1234





SB-TYPE (2): BRANCHES

Branches

000001000000001010000000000010011

31	25	24 20	1 9 15	14 12	11 7	6 0
	imm	rs2	rs1	funct3	imm	Ор
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

signed

ор		mnemonic	description	
11000	011	BEQ rs1, rs2, imm	PC=(R[rs1] == R[rs2] ? PC + sext(imm) :	: PC+4)
11000	011	BNE rs1, rs2, imm	PC=(R[rs1] != R[rs2] ? PC + sext(imm) :	PC+4)

Example: BEQ x5, x1, 128

$$PC = PC + 128$$
 (i.

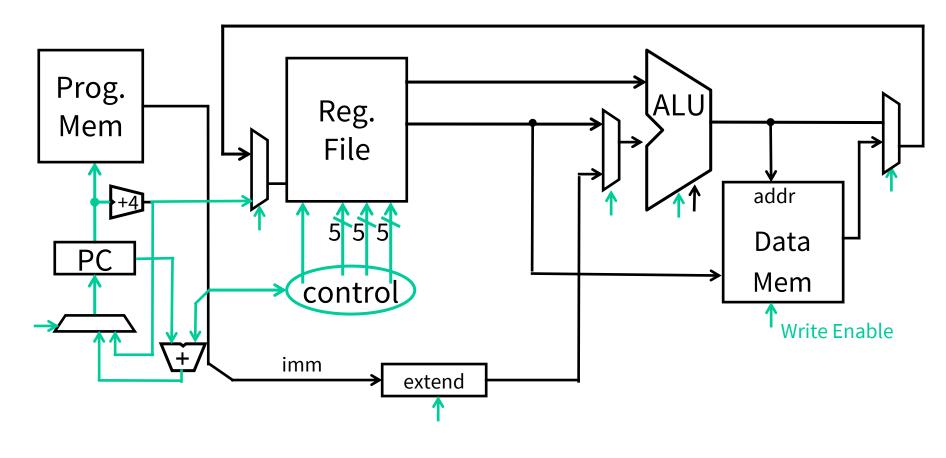
$$PC = PC + 128$$
 (i.e. $128 == 64 <<1$)





CONTROL FLOW: BRANCHES

Branches

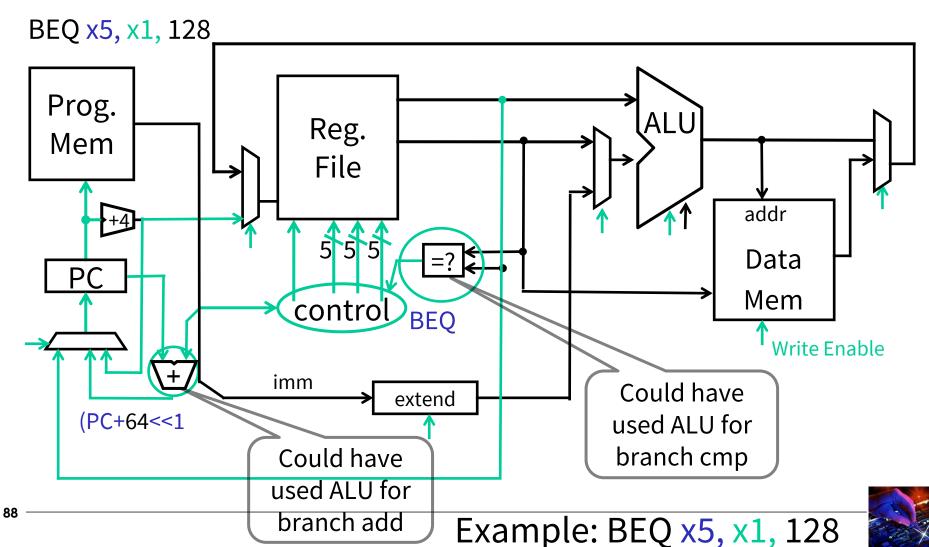






CONTROL FLOW: BRANCHES

Branches





SB-TYPE (3): CONDITIONAL JUMPS

Conditional jumps

000000000000001010001000000010011

31 25	24 20	19 15	14 12	11 7	6 0
imm	rs2	rs1	funct3	imm	Ор
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

ор	funct3	mnemonic	description
1100011	100	BLT rs1, rs2, imm	PC=(R[rs1] <s +="" :="" ?="" pc="" pc+4)<="" r[rs2]="" sext(imm)="" td=""></s>
1100011	101	BGE rs1, rs2, imm	PC=(R[rs1] >=s R[rs2] ? PC + sext(imm) : PC+4)
1100011	110	BLTU rs1, rs2 imm	PC=(R[rs1] <u +="" :="" ?="" pc="" pc+4)<="" r[rs2]="" sext(imm)="" td=""></u>
1100011	111	BGEU rs1, rs2, imm	PC=(R[rs1] >=u R[rs2] ? PC + sext(imm) : PC+4)

Example: BGE x5, x0, 32

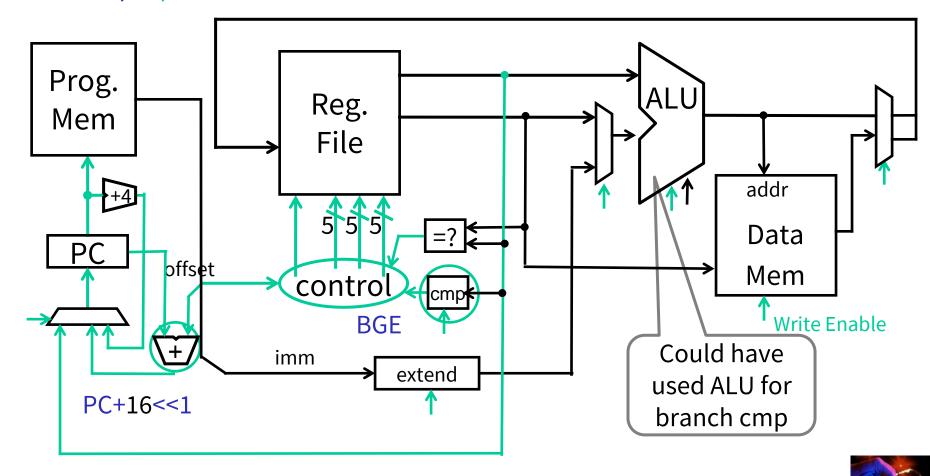
$$if(R[x5] \ge s R[x0])$$

$$PC = PC + 32$$
 (i.e. $32 == 16 <<1$)



CONTROL FLOW: MORE BRANCHES

Branches BGE x5, x0, 32





Example: BGE x5, x0, 32



SUMMARY

- We have all that it takes to build a processor
 - Arithmetic Logic Unit (ALU)
 - Register File
 - Memory
- RISC-V processor and ISA is an example of a Reduced Instruction Set Computers (RISC)
 - Simplicity is key, thus enabling us to build it
- We now know the data path for the MIPS ISA:
 - register, memory and control instructions

