



# Introduction to Instruction Set Architecture

Ergun Simsek

Lecture 4

1

## Representing Instructions

### Last class

- Representing Operands (Part 2)
  - Integers
  - Negative numbers
    - Signed Magnitude Representation
    - Two's Complement Representation (How to negate)
  - Non-integer numbers
    - IEEE 754

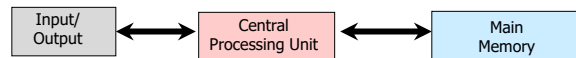
### Today

- von Neumann model of a computer
- Introduction to ISA (Instruction Set Architecture)

2

## A General-Purpose Computer

- Most of today's computers based on the model proposed by John von Neumann in the late 1940s
- Major components of the von Neumann Model are:



Central Processing Unit (CPU): Fetches, interprets, and executes a specified set of operations called **Instructions**.

Memory: storage of  $N$  words of  $W$  bits each, where  $W$  is a fixed architectural parameter, and  $N$  can be expanded to meet needs.

I/O: Devices for communicating with the outside world.

3

CMSC 411 – Lecture 5

3

## Open Questions in our Simple Model

We will answer the following questions today and in the following two lectures

- WHAT are INSTRUCTIONS?
- WHERE are INSTRUCTIONS stored?
- HOW are instructions represented?
- WHERE are VARIABLES stored?
- How are labels associated with particular instructions?
- How do you access more complicated variable types:
  - Arrays?
  - Structures?
  - Objects?
- Where does a program start executing?
- How does it stop?

CMSC 411 – Lecture 5

4

4

## Instructions and Programs

What are **instructions**?

- Words of a computer's language

**Instruction Set**

- The full vocabulary

**Stored Program Concept**

- Instructions and data are stored in a common (memory) memory as numbers

**Sequential semantics**

- All instructions execute sequentially (or at least appear sequential to the programmer)

Note: MIPS is a family of reduced instruction set computer (RISC) instruction set architectures (ISA)

There is also CISC: Complex Instruction Set Computers

CMSC 411 – Lecture 5

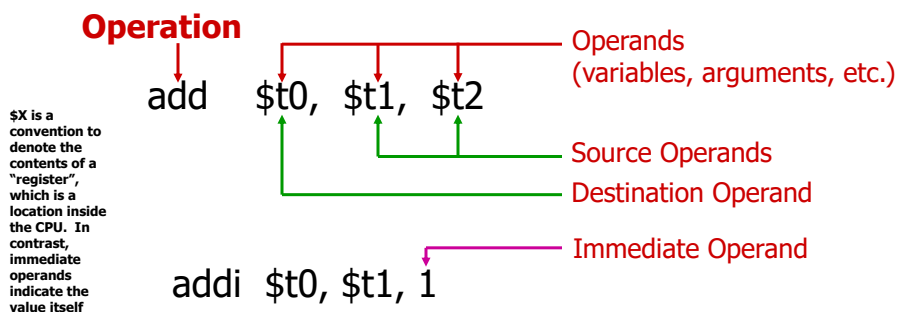
5

5

## Anatomy of an Instruction

An instruction is a primitive operation

- Instructions specify an operation and its operands (the necessary variables to perform the operation)
- Types of operands: immediate, source, and destination



CMSC 411 – Lecture 5

6

6

## Meaning of an Instruction

Operations are abbreviated into opcodes (1-4 letters)

Instructions are specified with a very regular syntax

- First an opcode followed by arguments
- Usually (but not always) the destination is next, then source
- Why this order? Arbitrary...

CMSC 411 – Lecture 5


7

7

## Being the Machine!

Instruction sequence

- Instructions are executed sequentially from a list ...
  - ... unless some special instructions alter this flow
- Instructions execute one after another
  - therefore, results of all previous instructions have been computed

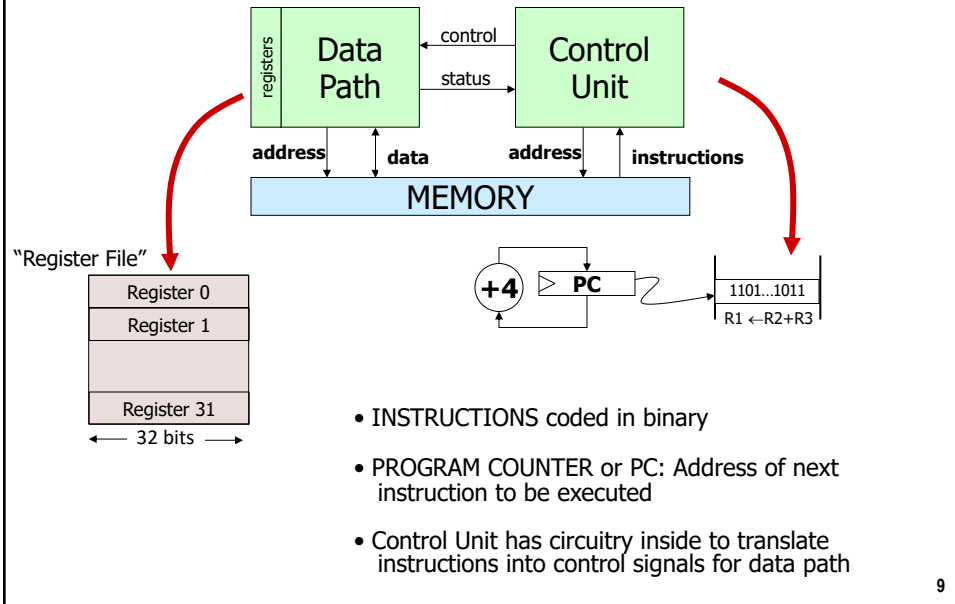
Instructions		Variables	
	add \$t0, \$t1, \$t1	\$t0:	<del>0</del> <del>12</del> <del>24</del> 48
	add \$t0, \$t0, \$t0	\$t1:	<del>0</del> 42
	add \$t0, \$t0, \$t0	\$t2:	8
	sub \$t1, \$t0, \$t1	\$t3:	10

CMSC 411 – Lecture 5

8

8

## Anatomy of a von Neumann Computer



9

## The big picture: RISC

- Memory is distinct from data path
- Registers are in data path
- Program is stored in memory
- Control unit fetches instructions from memory
- Control unit tells data path what to do
- Data can be moved from memory to registers, or from registers to memory
- All data processing (e.g., arithmetic) takes place within the data path

$a = b + c$

`add $10, $11, $9`

00000001011010010101000000100000

CMSC 411 – Lecture 5

11

11

## MIPS Register Usage Conventions

Some MIPS registers assigned to specific uses

- by convention, so programmers can combine code pieces
  - will cover the convention later
- \$0 is hard-wired to the value 0

Name	Register number	Usage
\$zero	0	the constant value 0
\$at	1	assembler temporary (for assembler use)
\$v0-\$v1	2-3	values returned by procedures/functions
\$a0-\$a3	4-7	arguments provided to procedures/functions
\$t0-\$t7	8-15	temporaries (for scratch work)
\$s0-\$s7	16-23	saved registers (saved across procedure calls)
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer (tracks start of process's space)
\$sp	29	stack pointer (tracks top of stack)
\$fp	30	frame pointer (tracks start of procedure's space)
\$ra	31	return address (where to return from procedure)

CMSC 411 – Lecture 5

12

12

# Bit, Byte, and Word (in MIPS-lish)

Bit: 0 or 1

Byte: 8-bits

Word: 32 bits = 4 bytes

## Little Endian vs Big Endian?

Every BYTE has a unique address = MIPS is a byte-addressable machine

Every instruction is one word

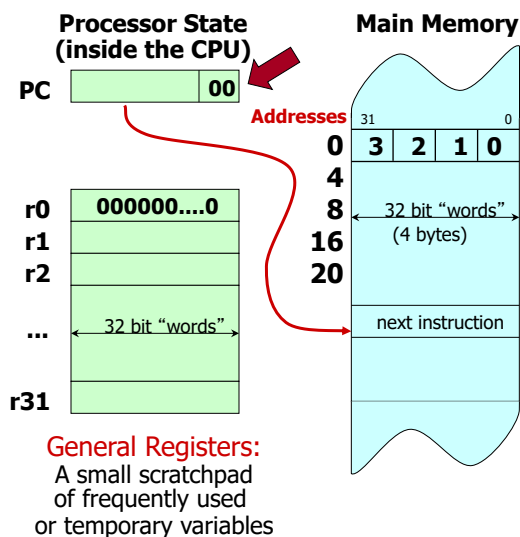
CMSC 411 – Lecture 5

13

13

# MIPS Programming Model

a representative simple RISC machine



We'll use a clean and sufficient subset of the MIPS-32 core Instruction set.

### Fetch/Execute loop:

- fetch Mem[PC]
- $PC = PC + 4^{\dagger}$
- execute fetched instruction (may change PC!)
- repeat!

†MIPS uses byte memory addresses. However, each instruction is 32-bits wide, and \*must\* be aligned on a multiple of 4 (word) address. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

14

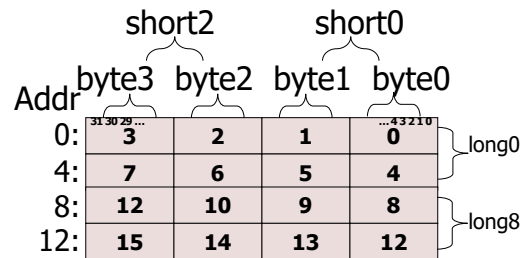
14

## Some MIPS Memory Nits

Memory locations are 32 bits wide

BUT, they are addressable in different-sized chunks

- 8-bit chunks (bytes)
- 16-bit chunks (shorts)
- 32-bit chunks (words)
- 64-bit chunks (longs/double)



We also frequently need access to individual bits!  
(Instructions help w/ this)

CMSC 411 – Lecture 5

15

15

## Programming a Processor

C code:

$f = (g+h) - (i+j);$

Compiler

Assembly:

add t0,g,h  
add t1, i,j  
sub f, t0,t1

Memory	
0	add R1, R4, R5
4	add R2, R6, R7
8	sub R3, R1, R2
12	
16	
20	
24	
28	
32	

OS Loader

MIPS Assembly

add R1, R4, R5  
add R2, R6, R7  
sub R3, R1, R2

CMSC 411 – Lecture 5

16

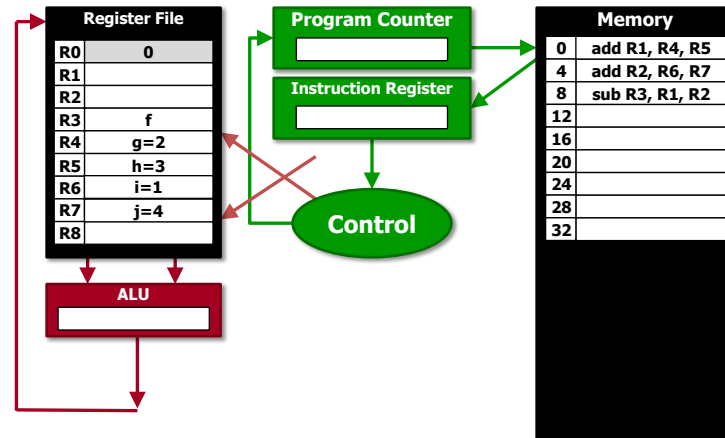
16



### Execution of $f = (g+h) - (i+j)$ in MIPS

Assume the following registers

R3=f  
R4=g  
R5=h  
R6=i  
R7=j



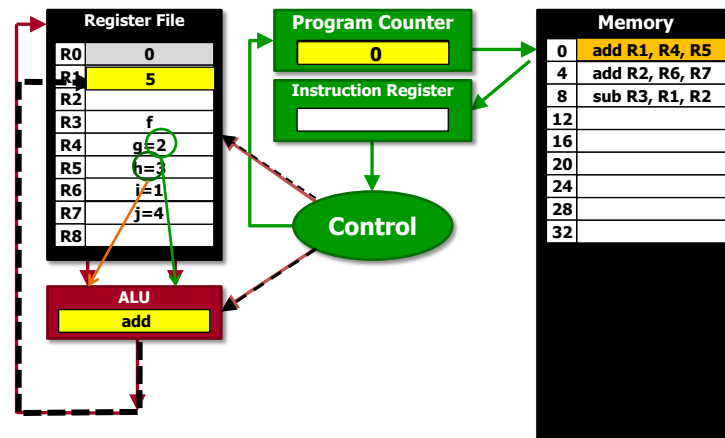
CMSC 411 – Lecture 5

17

### Execution of $f = (g+h) - (i+j)$ in MIPS

Assume the following registers

R3=f  
R4=g  
R5=h  
R6=i  
R7=j



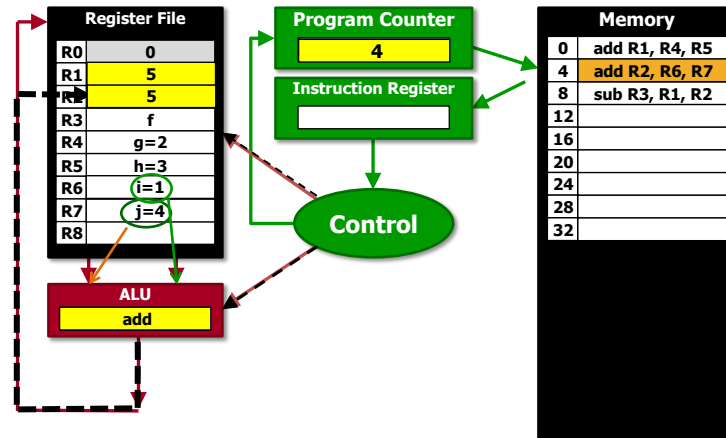
CMSC 411 – Lecture 5

18

Execution of  $f = (g+h) - (i+j)$  in MIPS

Assume the following registers

R3=f  
R4=g  
R5=h  
R6=i  
R7=j



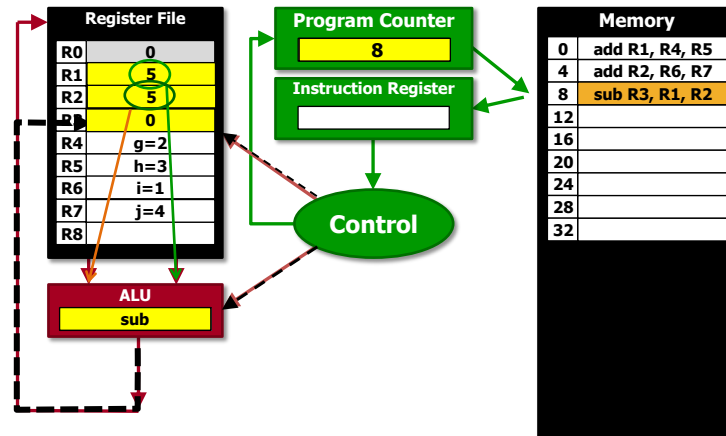
CMSC 411 – Lecture 5

19

Execution of  $f = (g+h) - (i+j)$  in MIPS

Assume the following registers

R3=f  
R4=g  
R5=h  
R6=i  
R7=j



CMSC 411 – Lecture 5

20

## MIPS Register Nits

There are 32 named registers [\$0, \$1, .... \$31]

The operands of **all** ALU instructions are registers

- This means to operate on a variables in memory you must:
  - Load the value/values from memory into a register
  - Perform the instruction
  - Store the result back into memory
- Going to and from memory can be expensive
  - (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!

Special purpose and conventions

- 2 registers have specific "side-effects"
  - (ex: \$0 always contains the value '0' ... more later)
- 4 registers dedicated to specific tasks by convention
- 26 available for general use, but constrained by convention

CMSC 411 – Lecture 5

21

21

## MIPS Instruction Formats

All MIPS instructions fit into a single 32-bit word

Every instruction includes various "fields":

- a 6-bit operation or "OPCODE"
  - specifies which operation to execute (fewer than 64)
- up to three 5-bit OPERAND fields
  - each specifies a register (one of 32) as source/destination
- embedded constants
  - also called "literal" or "immediate"
  - 16-bits, 5-bits or 26-bits long (you will see in a minute, why)
  - sometimes treated as signed values, sometimes unsigned

There are three basic instruction formats:

- **R-type**, 3 register operands (2 sources, destination)
- **I-type**, 2 register operands, 16-bit constant
- **J-type**, no register operands, 26-bit constant



CMSC 411 – Lecture 5

22

22