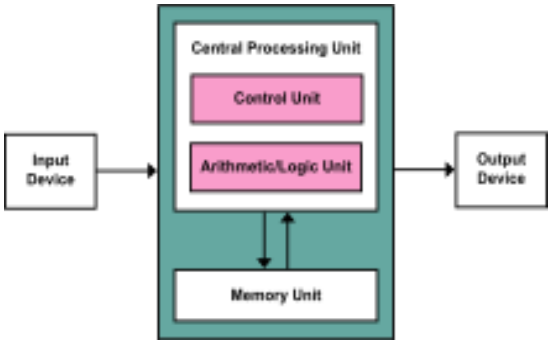
COMPUTER
ARCHITECTURE

CS 10 Computer Architecture and Organization
Instruction Set Architecture

Foothill College
Computer Science Department

Representing Instructions

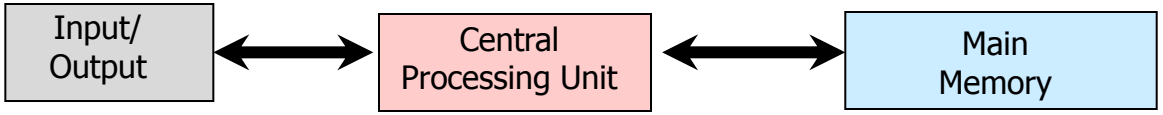
- This Week's topics
  - von Neumann model of a computer



  - Instruction set architecture
  - MIPS instruction formats
  - Some MIPS instructions

A General-Purpose Computer

- The von Neumann Model
  - Many architectural models for a general-purpose computer have been explored
  - Most of today's computers based on the model proposed by John von Neumann in the late 1940s
  - Its major components 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.

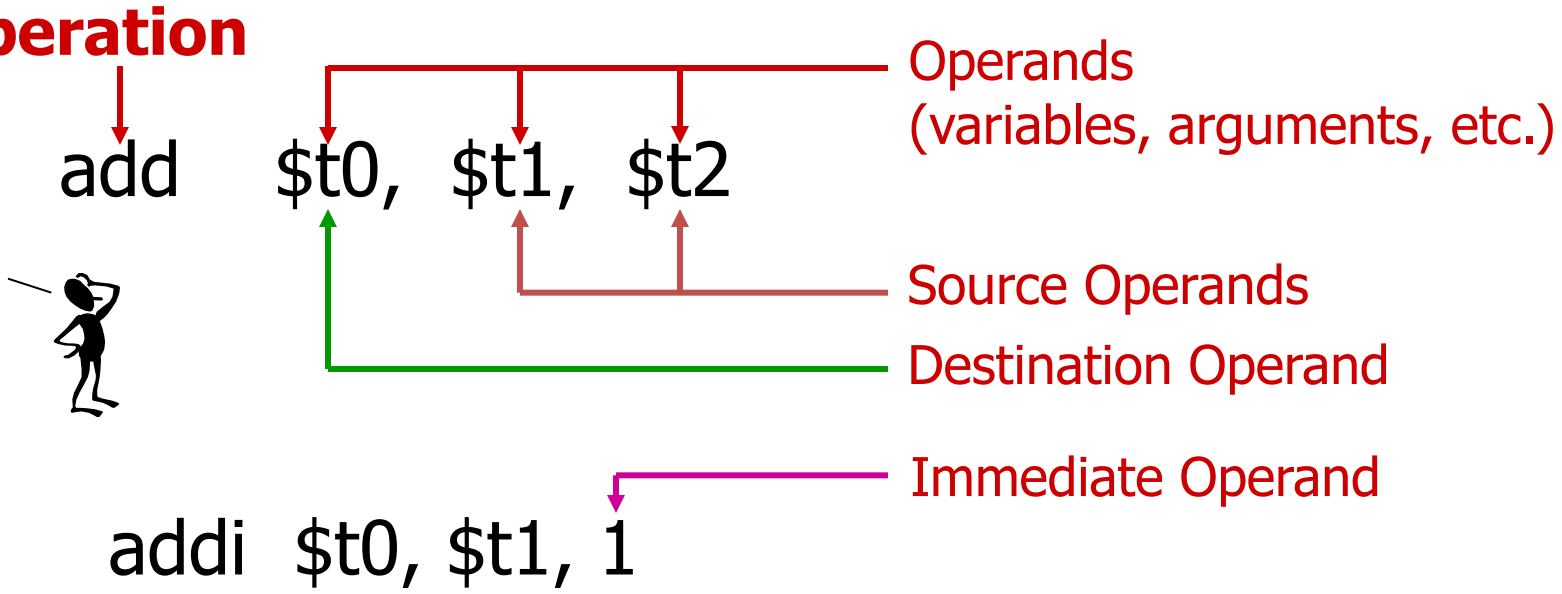
I/O: Devices for communicating with the outside world.

Instructions and Programs

- What are *instructions?*
  - the words of a computer's language
- Instruction Set
  - the full vocabulary
- Stored Program Concept
  - The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer
    - Distinct from "application-specific" hardware, which is "hardwired" to perform "fixed-function" processing on inputs
    - Distinct from punched tape computers (e.g., looms) where instructions were not stored, but streamed in one at a time

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



$X is a convention to denote the contents of a "register", which is a location inside the CPU. In contrast, immediate operands indicate the value itself

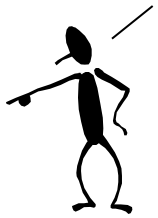
Meaning of an Instruction

- 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…
    - … but analogous to high-level language like Java or C

```
add  $t0, $t1, $t2
```

implies

```
int t0, t1, t2
t0 = t1 + t2
```



The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

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

```
add $t0, $t1, $t1
add $t0, $t0, $t0
add $t0, $t0, $t0
sub $t1, $t0, $t1
```

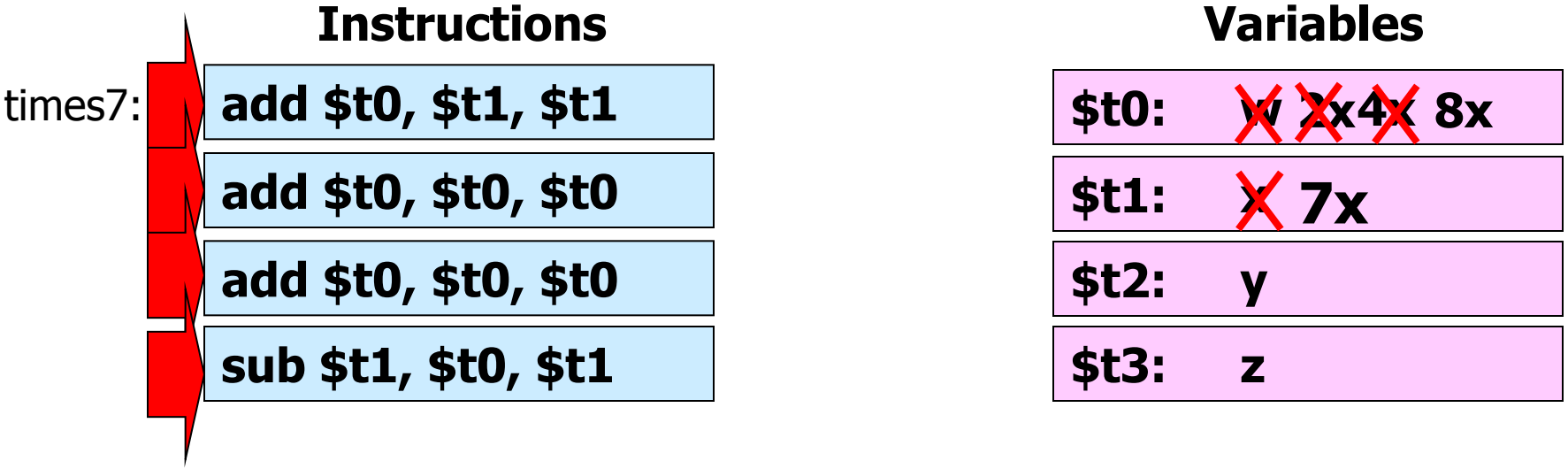
What is this program doing?

Variables

$t0:	~~0~~ ~~32~~ ~~24~~ 48
$t1:	~~8~~ 42
$t2:	8
$t3:	10

What did this machine do?

- Let's repeat the simulation, this time using unknowns
  - What is this machine doing?
- Knowing what the program does allows us to write down its specification, and give it a meaningful name

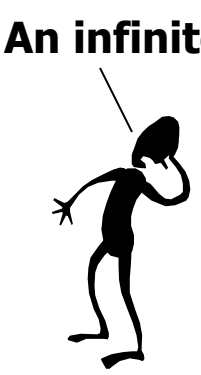


Looping the Flow

- Need something to change the instruction flow
  - "go back" to the beginning
  - a *jump* instruction with opcode 'j'
    - the operand refers to a label of some other instruction
    - for now, this is a text label you assign to an instruction
    - in reality, the text label becomes a numerical address

Instructions

```
times7: add $t0, $t1, $t1
        add $t0, $t0, $t0
        add $t0, $t0, $t0
        sub $t1, $t0, $t1
        j   times7
```



Variables

$t0:	w 8x 56x 392x
$t1:	x 7x49x 343x
$t2:	y
$t3:	z

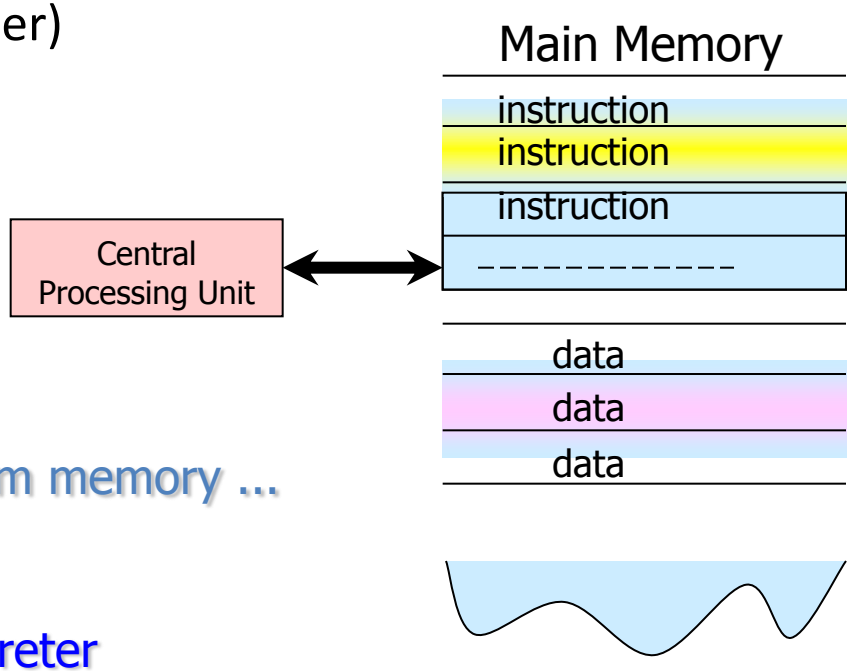
Open Questions in our Simple Model

- We will answer the following questions next
  - 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?

The Stored-Program Computer

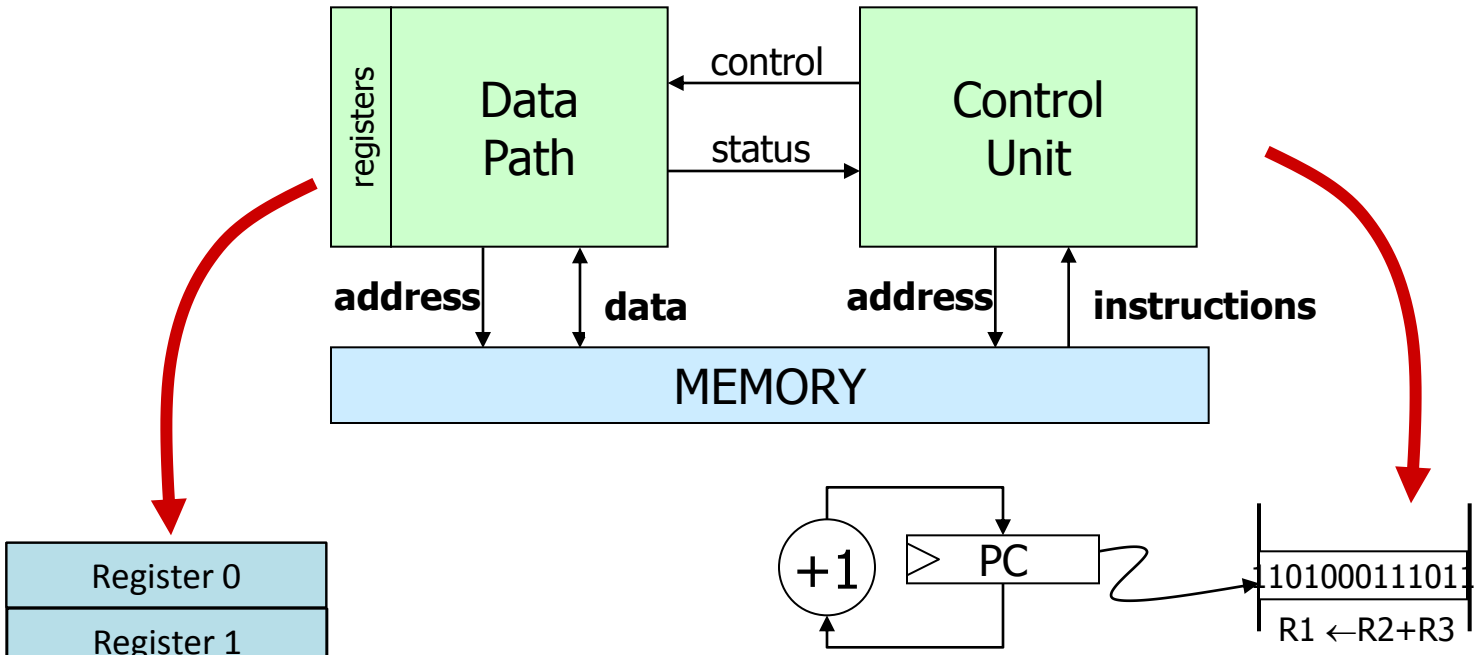
- The von Neumann model:
  - Instructions and Data stored in a common memory ("main memory")
  - Sequential semantics: All instructions execute sequentially (or at least appear sequential to the programmer)

Key idea: Memory holds not only data, but *coded instructions* that make up a *program*.



- CPU fetches and executes instructions from memory ...
  - The CPU is a H/W interpreter
  - Program IS simply data for this interpreter
  - Main memory: Single expandable resource pool
    - constrains both data and program size
    - don't need to make separate decisions of how large of a program or data memory to buy

Anatomy of a von Neumann Computer



- INSTRUCTIONS coded in binary
- PROGRAM COUNTER or PC: Address of next instruction to be executed
- Control Unit has circuitry inside to translate instructions into control signals for data path

Instruction Set Architecture

- Definition:
  - The part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O
  - An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor

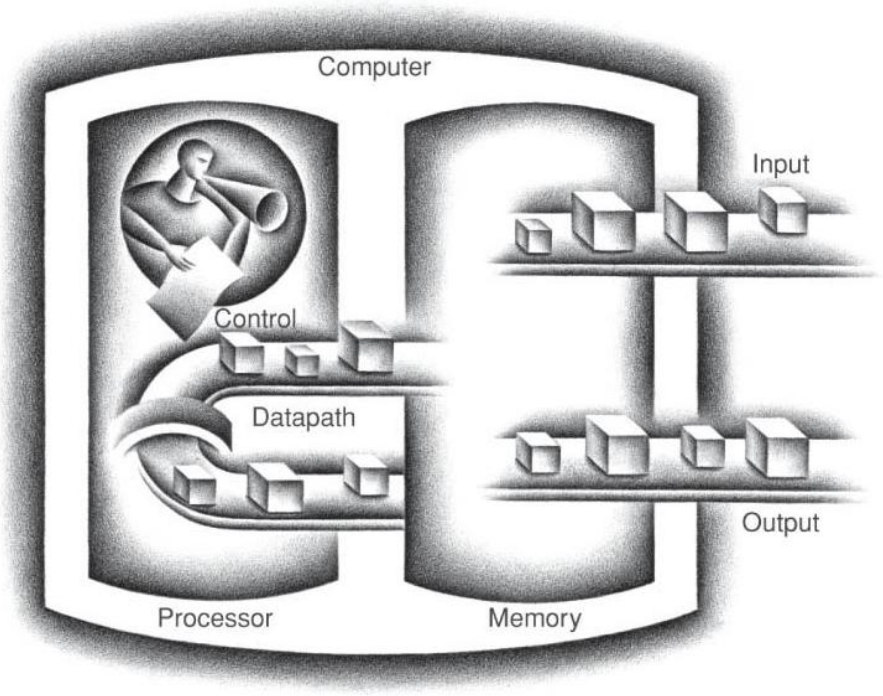
Instruction Set Architecture (ISA)

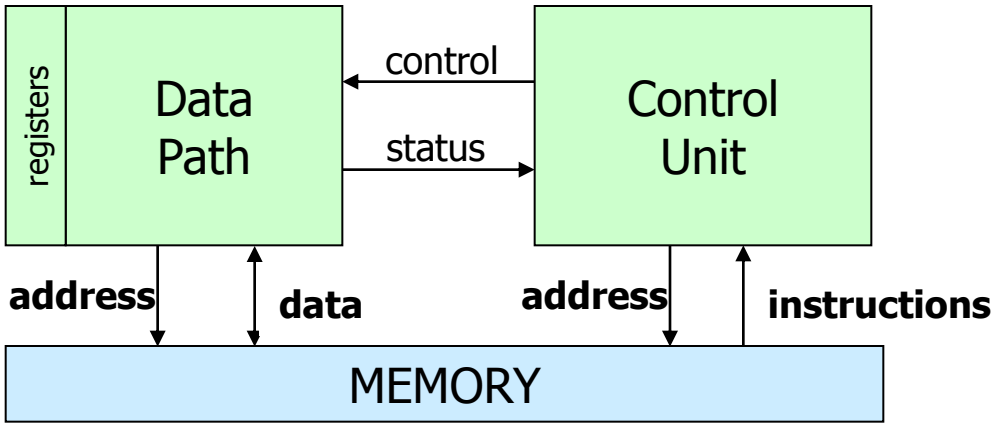
- Encoding of instructions raises interesting choices...
  - Tradeoffs: performance, compactness, programmability
  - Complexity
    - How many different instructions? What level operations?
      - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc.
    - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed
  - Uniformity
    - Should different instructions be same size?
    - Take the same amount of time to execute?
    - Trend favors uniformity ➔ simplicity, speed, cost/power
- Mix of Engineering & Art...
  - Trial (by simulation) is our best technique for making choices!

Our representative example: the MIPS architecture!

The Big Picture

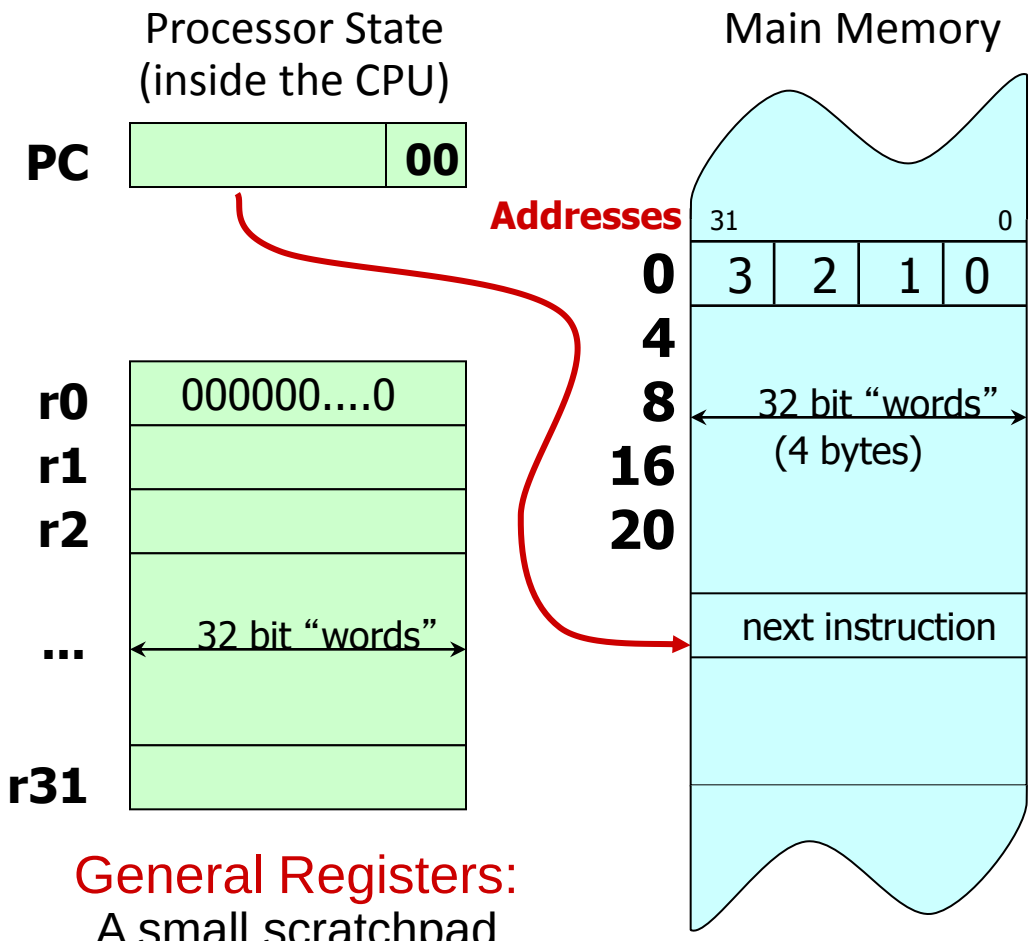
- A few things to note:
  - Memory is distinct from data path
  - Registers are in data path
  - Program is stored in memory
  - Control unit fetches instructions from memory
  - Control unit tells data path what to do
  - Data can be moved from memory to registers, or from registers to memory
  - All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





MIPS Programming Model

a representative simple RISC machine



General Registers:
A small scratchpad
of frequently used
or temporary variables

In CS 10 we'll use a clean and sufficient subset of the MIPS-32 core Instruction set.

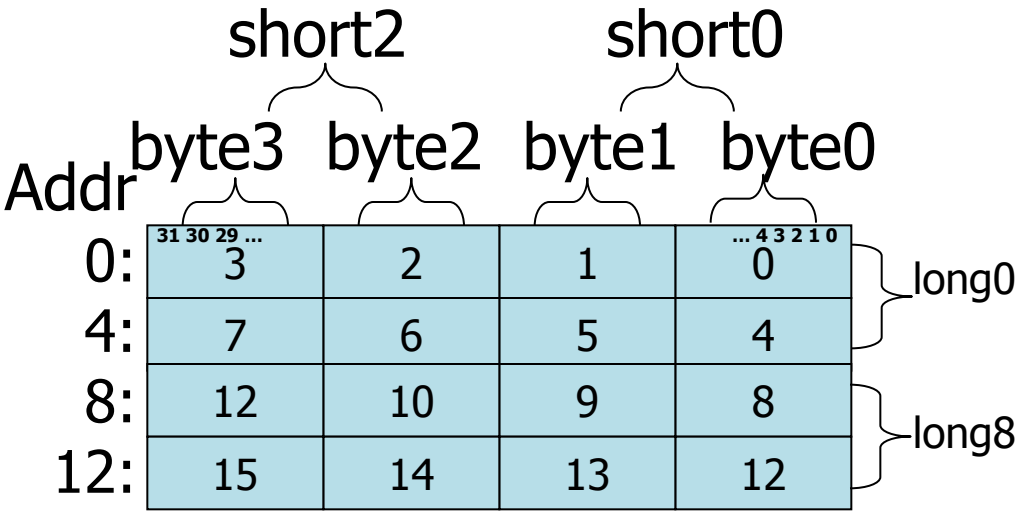
Fetch/Execute loop:

- fetch Mem[PC]
- PC = PC + 4†
- 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.*

MIPs Memory

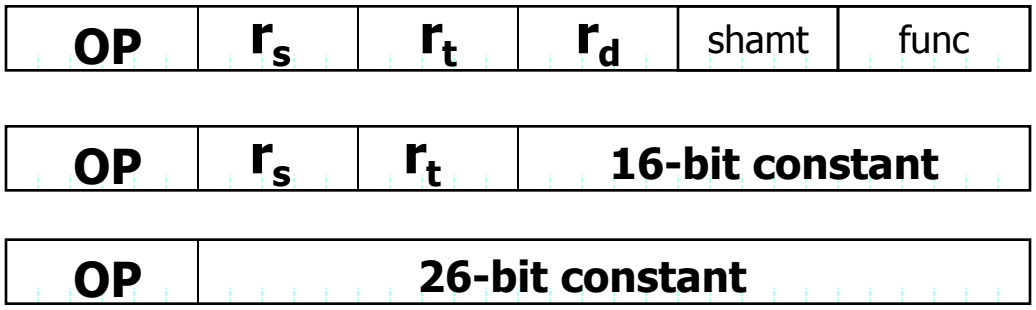
- Memory locations are 32 bits wide
  - BUT, they are addressable in different-sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/double)
- We also frequently need access to individual bits! (Instructions help w/ this)
- Every BYTE has a unique address (MIPS is a byte-addressable machine)
- Every instruction is one word



MIPS Instruction Formats

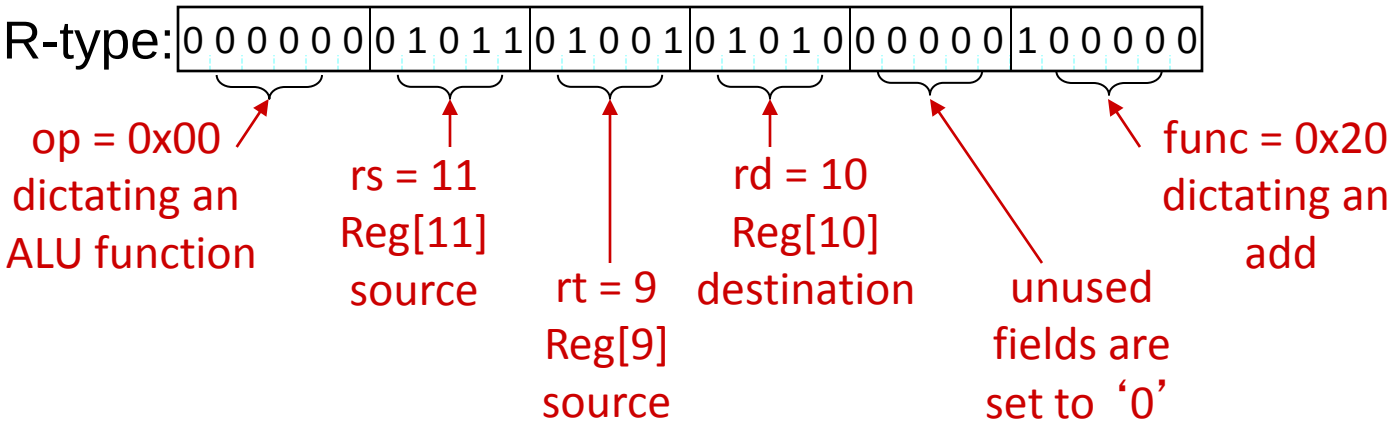
- All MIPS instructions fit into a single 32-bit word
- Every instruction includes various "fields":
  - a 6-bit operation or "OPCODE"
    - specifies which operation to execute (fewer than 64)
  - up to three 5-bit OPERAND fields
    - each specifies a register (one of 32) as source/destination
  - embedded constants
    - also called "literals" or "immediates"
    - 16-bits, 5-bits or 26-bits long
    - 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



MIPS ALU Operations

Sample coded operation: ADD instruction



References to register contents are prefixed by a "$" to distinguish them from constants or memory addresses

What *we* prefer to write: `add $10, $11, $9` ("assembly language")

The convention with MIPS assembly language is to specify the destination operand first, followed by source operands.

`add rd, rs, rt`:

Reg[rd] = Reg[rs] + Reg[rt]

"Add the contents of rs to the contents of rt; store the result in rd"

Similar instructions for other ALU operations:

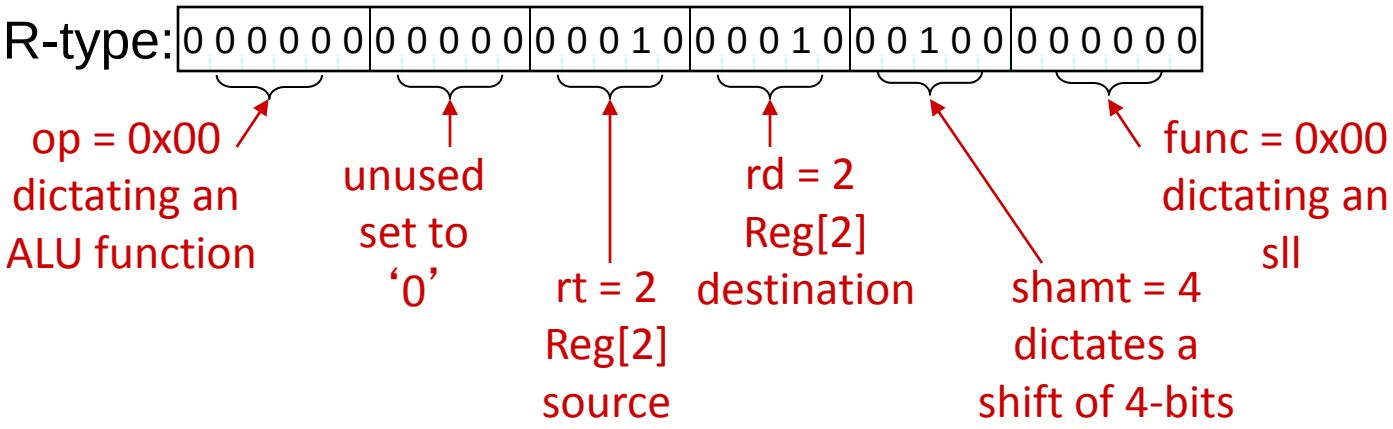
- arithmetic: add, sub, addu, subu
- compare: slt, sltu
- logical: and, or, xor, nor
- shift: sll, srl, sra, sllv, srav, srlv

Shift operations

- Shifting is a common operation
  - applied to groups of bits
  - used for alignment
  - used for "short cut" arithmetic operations
    - X << 1 is often the same as 2*X
    - X >> 1 can be the same as X/2
- For example:
  - $X = 20_{10} = 00010100_2$
  - Left Shift:
    - $(X << 1) = 00101000_2 = 40_{10}$
  - Right Shift:
    - $(X >> 1) = 00001010_2 = 10_{10}$
  - Signed or "Arithmetic" Right Shift:
    - $(-X >>> 1) = (11101100_2 >>> 1) = 11110110_2 = -10_{10}$

MIPS Shift Operations

Sample coded operation: SHIFT LOGICAL LEFT instruction



Assembly: `sll $2, $2, 4`

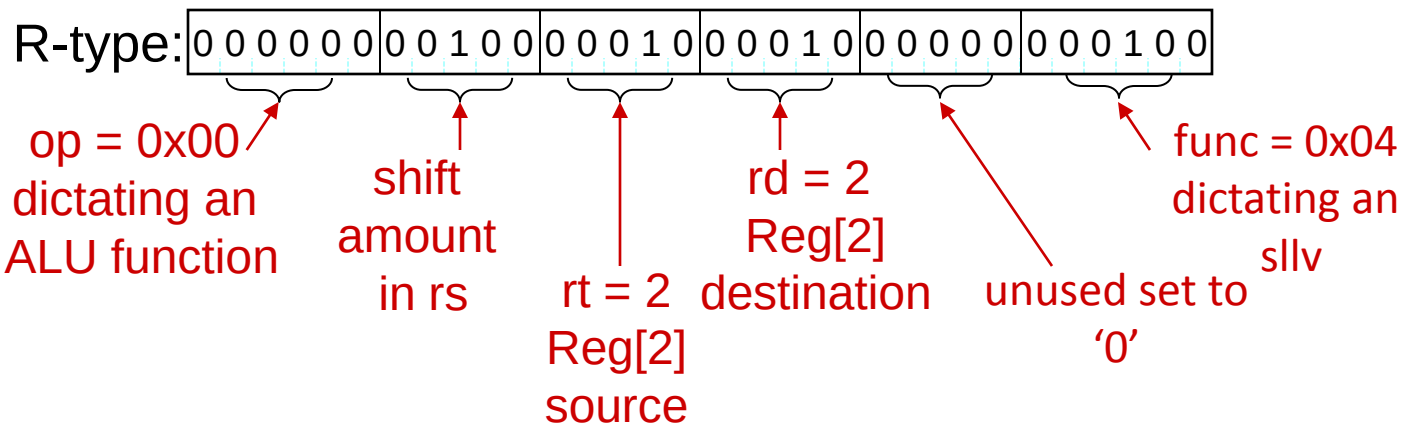
`sll rd, rt, shamt`:

Reg[rd] = Reg[rt] << shamt

"Shift the contents of *rt* to the left by *shamt*; store the result in *rd*"

MIPS Shift Operations

Sample coded operation: SLLV (SLL Variable)



This is peculiar syntax for MIPS, in this ALU instruction the rt operand precedes the rs operand. Usually, it's the other way around

Different flavor:
Shift amount is not in instruction, but in a register

Assembly: `sllv $2, $2, $8`

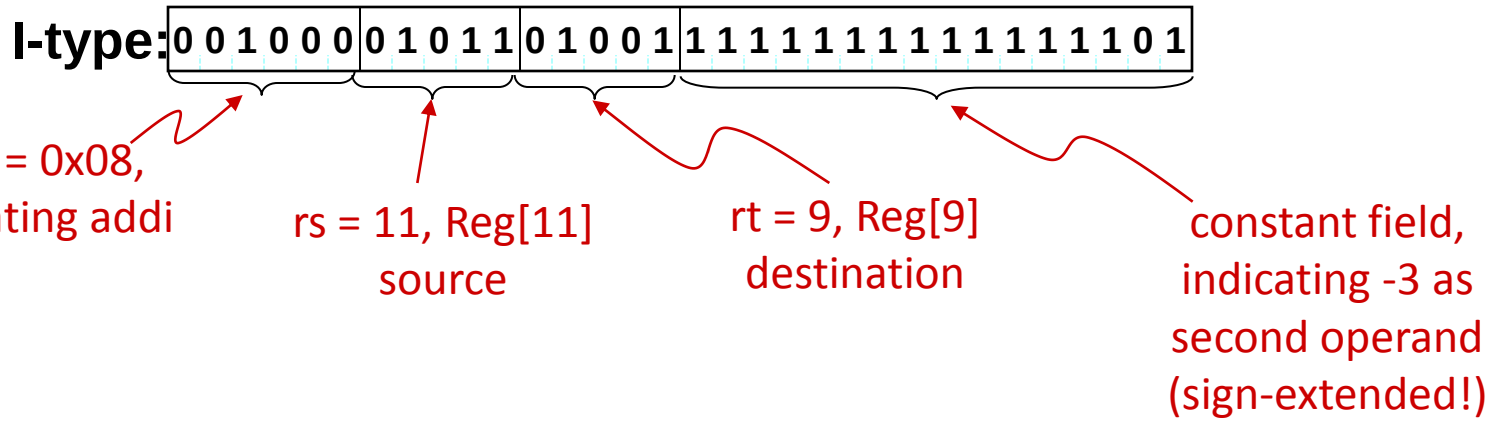
`sllv rd, rt, rs`:

Reg[rd] = Reg[rt] << Reg[rs]

"Shift the contents of *rt* left by the contents of *rs*; store the result in *rd*"

MIPS ALU Operations with Immediate

addi instruction: adds register contents, signed-constant:



Symbolic version: `addi $9, $11, -3`

`addi rt, rs, imm`:

Reg[rt] = Reg[rs] + sxt(imm)

"Add the contents of rs to const; store result in rt"

Similar instructions for other ALU operations:

arithmetic: addi, addiu
compare: slti, sltiu
logical: andi, ori, xori, lui

Immediate values are sign-extended for arithmetic and compare operations, but not for logical operations.

Why Built-in Constants? (Immediate)

- Where are constants/immediates useful?
  - SMALL constants used frequently (50% of operands)
    - In a C compiler (gcc) 52% of ALU operations use a constant
    - In a circuit simulator (spice) 69% involve constants
    - e.g., B = B + 1; C = W & 0x00ff; A = B + 0;

- Examples:

```
addi  $29, $29,  4
slti   $8, $18, 10
andi  $29, $29,  6
ori   $29, $29,  4
```

First MIPS Program (fragment)

- Suppose you want to compute the expression:

  f = (g + h) – (i + j)

  - where variables f, g, h, i, and j are assigned to registers $16, $17, $18, $19, and $20 respectively
  - what is the MIPS assembly code?

```
add $8,$17,$18          # (g + h)
add $9,$19,$20          # (i + j)
sub $16,$8,$9           # f = (g + h) – (i + j)
```

  - Questions to answer:
    - How did these variables come to reside in registers?
    - Answer: We need more instructions which allow data to be explicitly ***loaded*** from memory to registers, and ***stored*** from registers to memory

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)

Continue next module…

- More MIPS instructions
  - accessing memory
  - branches and jumps
  - larger constants
  - multiply, divide
  - etc.