CSE 210: Computer Architecture
Lecture 4: Introduction to MIPS

Stephen Checkoway

Oberlin College

Oct. 11, 2021

Slides from Cynthia Taylor

Announcements

- Problem Set 1 due Friday at 11:59 pm
- Office hours Tuesday 13:30 – 14:30

Why you should learn (a little) assembly

- Learn what your computer is fundamentally capable of

- By learning about how high level mechanisms are created in assembly, we learn what is fast, what is slow . . .

- Might use it for reverse engineering, embedded systems, compilers

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (owned by John L. Hennessy, who wrote your book.)
- Used in Embedded Systems
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs

Three Types of Instruction

- Arithmetic (R)
- Immediate (I)
- Jump (J)

Arithmetic and Logical Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a = b + c
sub a, b, c  # a = b - c
and a, b, c  # a = b & c (bit-wise AND)
```

- All arithmetic and logical operations have this form

Convert to MIPS: f = (g + h) - (i + j);

A.

```
add    f, g, h
sub    f, i, j
```

B.

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

C.

```
sub    f, (add g,h), (add i,j)
```

D. More than one of these is correct

Register Operands

- Arithmetic instructions use register operands

- MIPS has a 32 × 32-bit register file
  - Numbered 0 to 31
  - 32-bit data called a “word”

Aside: MIPS Register Convention

Name	Register Number	Usage
$zero	0	constant 0 (hardware)
$at	1	reserved for assembler
$v0 - $v1	2-3	returned values
$a0 - $a3	4-7	arguments
$t0 - $t7	8-15	temporaries
$s0 - $s7	16-23	saved values
$t8 - $t9	24-25	temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return addr (hardware)

Register Operand Example

- C code:

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

  – f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add  $t0, $s1, $s2
add  $t1, $s3, $s4
sub  $s0, $t0, $t1
```

Some R-type instructions

- `add  dest, src1, src2`
- `sub  dest, src1, src2`
- `div  dest, src1, src2 # Pseudoinstruction!`
- `mul  dest, src1, src2`
- `move dest, src        # add dest, $zero, src`
- `and  dest, src1, src2`
- `or   dest, src1, src2`
- `nor  dest, src1, src2`
- `xor  dest, src1, src2`

Questions about Arithmetic Operations?

Memory Instructions

- `lw   $t0, 0($t1)`
  - `$t0 = Mem[$t1+0]`
  - Loads 4 bytes from `$t1, $t1+1, $t1+2, and $t1+3`
- `sw   $t0, 4($t1)`
  - `Mem[$t1+4] = $t0`
  - Stores 4 bytes at `$t1+4,   $t1+5,   $t1+6, and $t1+7`
- These instructions are the cornerstones of our being able to go to and from memory

Accessing the Operands

There are typically two locations for operands – registers (internal storage e.g., $t0 or $a0) and memory. In each column we have which—reg or mem—is better. Which row is correct?

	Faster access	**Fewer bits to specify address**	**More locations**
A	Mem	Mem	Reg
B	Mem	Reg	Mem
C	Reg	Mem	Reg
D	Reg	Reg	Mem
E	None of the above		

Load-store architectures

can do:

add r1=r2+r3

and

load r3, M(address)

$\Rightarrow$ forces heavy dependence on registers, which is exactly what you want in today's CPUs

can't do

add r1 = r2 + M(address)

- more instructions

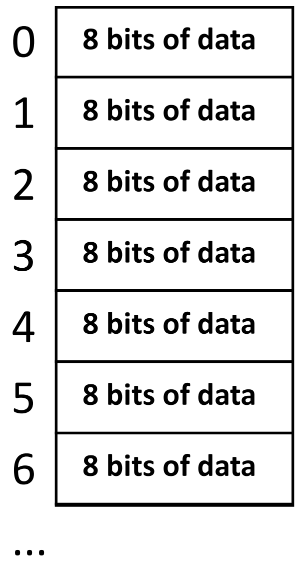
+ fast implementation

Memory

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address of a word must be a multiple of 4

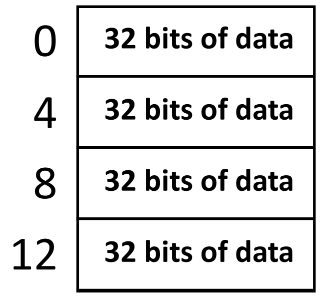
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- “Byte Addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32} - 1$

- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32} - 4$

Processor X is 8 bit **byte-addressable**. If you have a pointer at address 0000 0000 0000 1000 and you increment it by one (0000 0000 0000 1001). What does the new pointer (0000 0000 0000 1001) point to, relative to the original pointer (0000 0000 0000 1000)?

A) The next word in memory
B) The next byte in memory
C) Either the next word or byte – depends on if you use that address for a load byte or load word
D) Pointers are a high level construct – they don't make sense pointing to raw memory addresses.
E) None of the above.

Processor Y is 32 bit **word-addressable**. If you have a pointer at address 00 0000 0000 1000 and you increment it by one (00 0000 0000 1001). What does the new pointer (00 0000 0000 1001) point to, relative to the original pointer (00 0000 0000 1000)?

A) The next word in memory
B) The next byte in memory
C) Either the next word or byte – depends on if you use that address for a load byte or load word
D) Pointers are a high level construct – they don't make sense pointing to raw memory addresses.
E) None of the above.

Reading

- Next lecture: Assembly
  - 2.3

- Problem Set 1: Due Friday at 11:59pm via Gradescope