CSCI 210: Computer Architecture
Lecture 4: Introduction to MIPS

Stephen Checkoway
Oberlin College
Slides from Cynthia Taylor

Announcements

- Problem Set 1 due Friday 11:59 p.m.

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

CS History: Sophie Wilson

Developed the ARM Instruction Set Architecture

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
  - Most similar to ARM, RISC-V

Three Types of Instruction

- Arithmetic and logical (R)
  - Operates on data entirely in registers
- Immediate (I)
  - One of the operands is encoded directly in the instruction
- Jump (J)
  - Changes the pc to a new location

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
```

```
add a, b, c  # a = b + c
sub a, b, c  # a = b - c
```

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 32 32-bit general purpose registers
  - Numbered 0 to 31
  - 32-bit data called a “word”

- ARM has 37 32-bit general purpose registers
- X86-64 has 16 general purpose registers, around 40 named registers used by the processor
  - Can be used as 8, 16, 32, or 64 bit registers

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, g, h, and j in registers $s0, $s1, $s2, $s3, and $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`
- `mul  dest, src1, src2 # Pseudoinstruction!`
- `div  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`

Assume registers initially have the following values

$a0	$a1	$t0	$t1	$v0
2	100	5	6	7

What values do they have after running this code?

```
move $t0, $a0
add  $t1, $a0, $a0
add  $t1, $t1, $t1
sub  $t0, $t1, $t0
add  $v0, $t0, $a1
```

	$a0	$a1	$t0	$t1	$v0
A	2	100	5	6	7
B	2	100	6	8	106
C	5	-10	-17	22	7
D	5	100	15	20	115
E	None of the above				

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

Load instructions

- **lw** — Loads 4 bytes of memory into a register
  - lw $t0, 8($t4)
- **lh** — Loads 2 bytes of memory into a register
  - lh $t2, 6($t1)
- **lb** — Loads 1 byte of memory into a register
  - lb $t3, 3($t0)

- lw and lb are more common than lh

Store instructions

- **sw** — Stores 4 bytes from a register into memory
  - sw $t0, 8($t4)
- **sh** — Stores 2 bytes from a register into memory
  - sh $t2, 6($t1)
- **sb** — Stores 1 byte from a register into memory
  - sb $t3, 3($t0)

- sw and sb are more common than sh

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	Smaller number to specify which reg/mem location	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:

```
load r3, M(address)
add r1 = r2 + r3
```

can't do

```
add r1 = r2 + M(address)
```

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

- more instructions

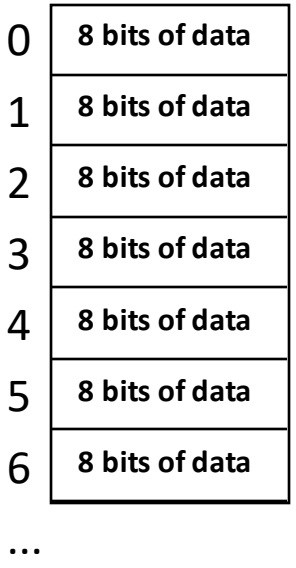
+ fast implementation

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

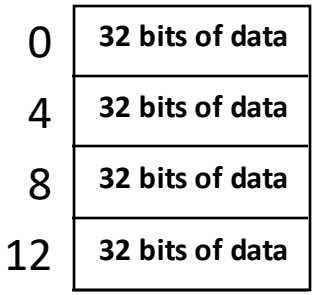
Memory Organization

- 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, ... $2^{32} - 4$

If you have a pointer to address 1000 and you increment it by one to 1001. What does the new pointer point to, relative to the original pointer?

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.

If a 4-byte word is in memory at address 4203084, what is the address of the next word in memory?

A) 4203085

B) 4203088

C) 14203084

D) It depends on the value of the words in memory

E) Since a word is 4 bytes, it’s not possible to have one at address 4203084

Arrays

- Arrays are stored consecutively in memory
- The **base** address points to the first element in the array
- Accessing other elements in the array requires adding an **offset** to the base address
  - The offset to use is the index of the array element * the size of one element

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3
  - A is an array of 4 byte integers

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)
add $s1, $s2, $t0
```

Translate to MIPS

- C code: g = h + A[5];
  - g in $s1, h in $s2, base address of A in $s3.
  - A is an array of 4-byte ints

A.
```
lw  $t0, 5($s3)
add $s1, $s2, $t0
```

B.
```
lw  $t0, 20($s3)
add $s1, $s2, $t0
```

C.
```
lw  $t0, $s5
add $s1, $s2, $t0
```

D.
```
lw  $t0, $s3
add $s1, $s2, $t0
```

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

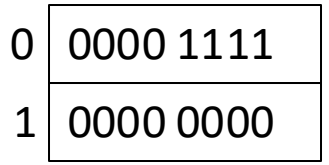
  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32; index 12 requires offset 48

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

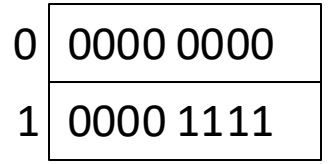
When a 2-byte word is stored in byte-addressed memory (occupying two consecutive bytes), is the most significant byte (MSB) stored in the lower address or the higher address?

A. Low



= 15

B. High



= 15

C. It Depends

Byte ordering

- Big-endian: Most significant byte in lowest address
  - MIPS, Motorola 68000, PowerPC (usually), SPARC (usually), …
- Little-endian: Most significant byte in highest address
  - Intel x86, x86-64, ARM (usually), …
- Bi-endian: Switchable between big and little endian
  - ARM, PowerPC, Alpha, SPARC, …
- Middle-endian/mixed-endian
  - Bytes not stored in either order, at least in some cases

Big-endian means most significant byte/digit/piece comes first, little-endian means least significant byte/digit/piece comes first. Mixed-endian means not in order.

Which row of the table correctly identifies the endianness of date formats?

	US (MM-DD-YYYY)	Most of the world (DD-MM-YYYY)	ISO 8601 (YYYY-MM-DD)
A	Little	Mixed	Big
B	Big	Little	Mixed
C	Mixed	Little	Big
D	Mixed	Big	Little
E	Little	Big	Mixed

Reading

- Next lecture: Assembly
  - 2.3

- Problem Set 1: Due Friday at 10:00 pm