

CSCI 210: Computer Architecture

Lecture 14: MIPS addressing

Stephen Checkoway

Oberlin College

Nov. 3, 2021

Slides from Cynthia Taylor

Announcements

- Problem Set due Friday
- Lab 3 due Sunday
- Office Hours Friday 13:30 – 14:30

Basic Question of Addressing

- How do we specify which data to operate on (or instruction to jump to)?
- Complication:
 - Instructions are 32 bits.
 - Memory addresses are 32 bits.
 - Data is in 32 bit words.
- Can never full specify address/data in a single instruction

Register Addressing



- Which register the data is in is specified in the instruction
- 32 registers = 5 bits per register address
- Used in add, jr, etc

Immediate Addressing

1. Immediate addressing



- Data is a constant within instruction
- There is no memory address/register, because we are just writing the information in the instruction itself
- 16 bits, can specify numbers up to $2^{16}-1 = 64 \text{ k}$
- Used in addi, ori, etc

32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
- lui rt, constant
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

Which of these will set \$t0 to 0xF0F0F0F0?

A. `lui $t0, 0xF0F0`
`addi $t0, $t0, 0xF0F0`

B. `lui $t0, 0xF0F0`
`ori $t0, $t0, 0xF0F0`

C. `ori $t0, $t0, 0xF0F0`
`lui $t0, 0xF0F0`

D. More than one of these will work

E. None of these will work

Aside: Loading and Storing Bytes

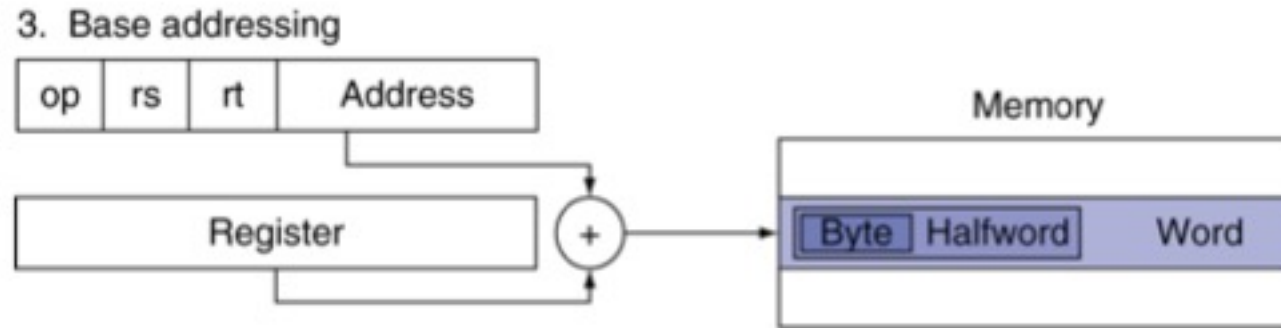
- MIPS provides special instructions to move bytes
 - `lb $t0, 1($s3)` # load byte from memory
 - `sb $t0, 6($s3)` # store byte to memory

opcode	rs	rt	16 bit offset
--------	----	----	---------------

□ What 8 bits get loaded and stored?

- load byte places the byte from memory in the rightmost 8 bits of the destination register
 - Byte is sign extended, other bytes in register erased
- store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
 - Other bytes in word of memory are left intact

Base + Offset Addressing

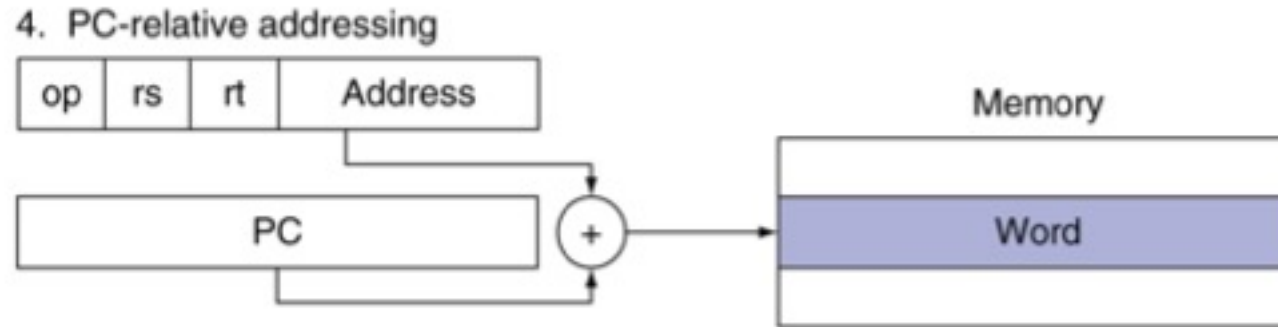


- Problem: 16 bits is not enough to address every word in memory
- Solution: Add the 16-bit offset to the 32-bit address within a register (the base)
- Used in `lw`, `sw`

Branch Instructions' targets are

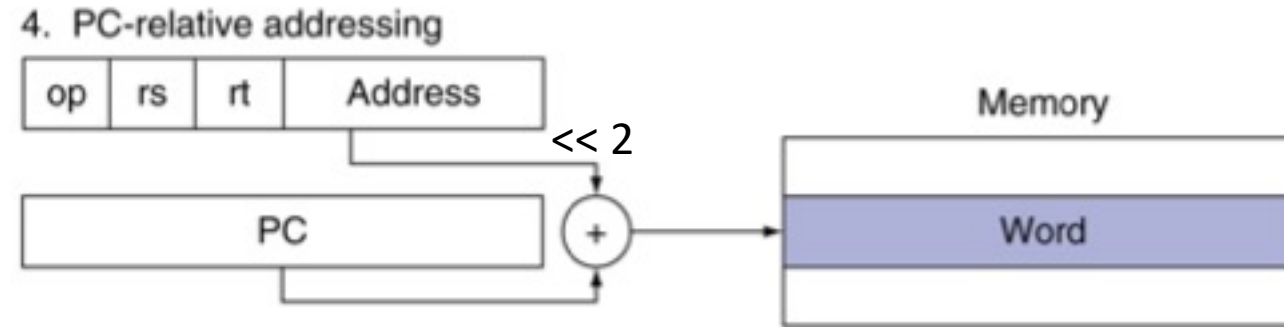
- A. usually within 2^{15} instructions of the branch instruction
- B. always within 2^{15} instructions of the branch instruction
- C. usually more than 2^{15} instructions away from the branch instruction

PC-relative Addressing



- Take 16 bit constant, shift left 2, add to value in PC
- Can access PC +/- 2^{17} bytes
- Used in beq, bne

Why do we shift left by two?



- A. We use the last two bits of the PC instead
- B. We only branch to instructions that are multiples of 4 words away from the current instruction
- C. Instructions are words and addresses specify bytes, so the last two bits of the address will always be 00
- D. None of the above

Which PC value in PC-relative addressing?

0x42000	slt	\$t0, \$t1, \$t2
0x42004	beq	\$t0, \$zero, target
0x42008	addi	\$s0, \$s0, 1
...		
0x?????	target:	ori \$s0, \$s0, 1

If the beq instruction has an immediate field of 0x0572, what is the address of the target ori instruction?

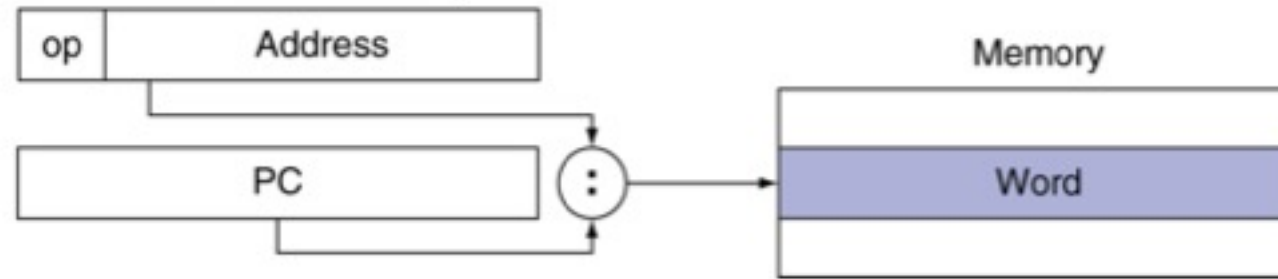
PC is the address of the *following* instruction
target address: $0x42004 + 4 + (0x0572 \ll 2)$

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- `beq $t0, $t1, far_away`
becomes
`bne $t0, $t1, not_equal`
`j far_away`
`not_equal:`

Pseudo-direct Addressing

5. Pseudodirect addressing



- We have 26 bits of address in the instruction
- Shift left by two
- Concatenate first four bits of PC + 4 with address
- Used in j, jal

Consider a jal instruction at address 0xC8001074 whose 26-bit address field has the value 0x00000003. What is the address of the instruction the jal will jump to?

- A. 0x000000003
- B. 0x00000000C
- C. 0xC00000003
- D. 0xC00000007
- E. 0xC0000000C

Assembler directives

- Instructions to the assembler
 - `.data` / `.text` / `.rodata` / `.bss` are used to switch between global (mutable) data, executable code, read-only data, and uninitialized data in the output
 - `.word x` allocates space for 4 bytes with value `x`
 - `.space n` allocates `n` bytes of space
 - `.ascii "string"` writes a 0-terminated string at that location

Arrays!

- How do we declare a 10-word array in our data section?

- Could do

`.data`

`x1: .word 0`

`x2: .word 0`

`x3: .word 0`

`...`

`x10: .word 0`

Declaring an Array

- Instead, just declare a big chunk of memory

```
.data
```

```
arr: .space 40
```

```

.data
arr: .space 40

.text
    li    $t0, 0
    addi  $t1, $t0, 10
    la    $s0, arr
loop:
    beq   $t0, $t1, end
    What goes here?
    addi  $t0, $t0, 1
    j     loop
end:

```

D. More than one of the above

E. None of the above

```

int i;
for (i = 0; i < 10; i++){
    arr[i] = i;
}

```

```
sw    $t0, $t1($s0)
```

A

```
add    $t2, $s0, $t1
sw     $t0, 0($t2)
```

B

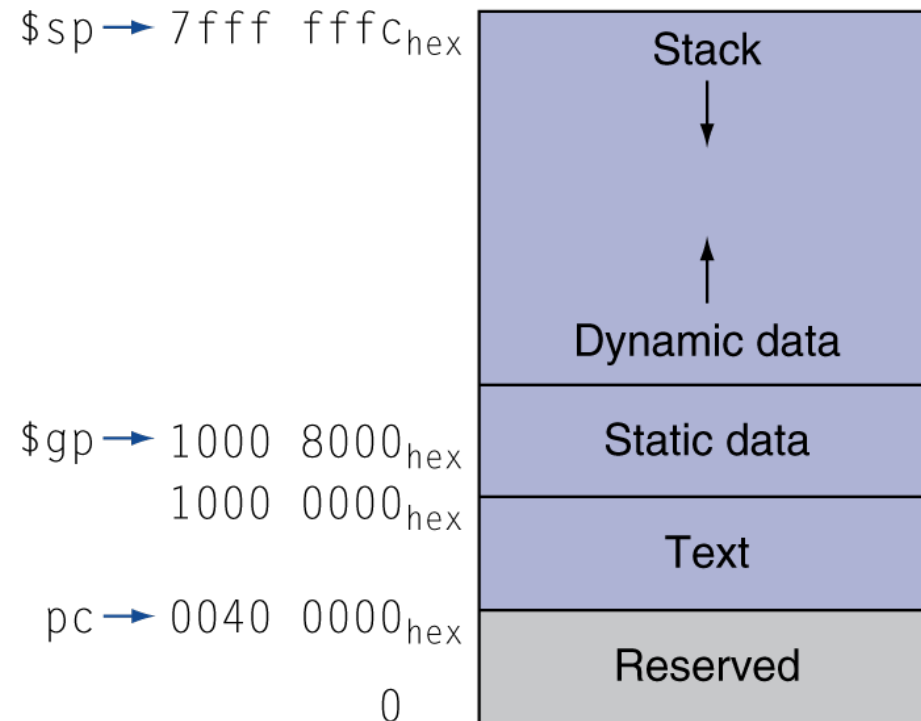
```
sw     $t0, 0($s0)
addi   $s0, $s0, 4
```

C

But what if we don't know how big the array will be before runtime?

sbrk system call

- Allocates memory on the heap and returns its address in \$v0
- Amount of memory is specified in bytes in \$a0



System Calls

- Syscalls (when we need OS intervention)
 - I/O (print/read stdout/file)
 - Exit (terminate)
 - Get system time
 - Random values

System Calls Review

- How to use:
 - Put syscall number into register \$v0
 - Load arguments into argument registers
 - Issue syscall instruction
 - Retrieve return values

- Example (print the integer in \$t0):

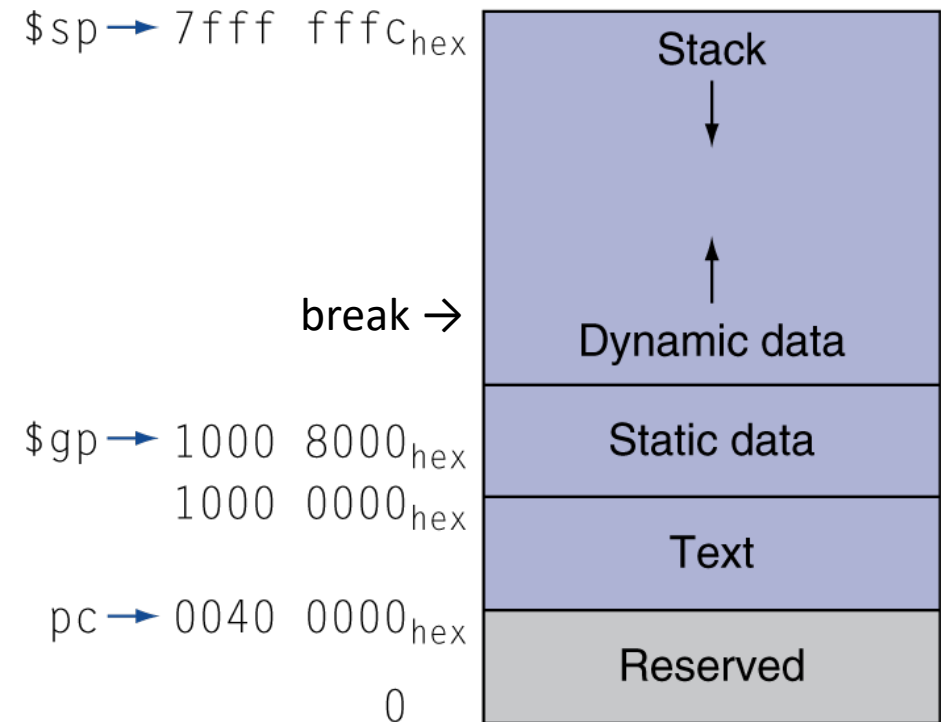
```
li      $v0, 1
move    $a0, $t0
syscall
```

System Call Codes

\$v0 code	Service	Arguments	
1	Print integer	\$a0=integer to print	
2	Print float	\$f12=float to print	
3	Print double	\$f12=double to print	
4	Print string	\$a0=address of string	
5	Read integer		\$v0 = read integer
6	Read float		\$f0 = read float
7	Read double		\$f0 = read double
8	Read string	\$a0 = address of input buffer, \$a1 = max number of characters	
9	Sbrk (allocate heap memory)	\$a0 = number of bytes	\$v0 = address
10	Exit (terminate program)		

What about freeing memory?

- Some operating systems maintain a “program break” which controls the size of the dynamic data
- `sbrk` requests the OS increment/decrement the break
- `malloc()/free()` carve the dynamic data up into chunks which the application can use and maintain lists of free chunks
- Freeing memory adds the chunk to a “free list”
- When more memory is needed, the break is changed



Reading

- Next lecture: Digital logic
- Problem set 4: Due Friday
- Lab 3 due Sunday