CSCI 210: Computer Architecture
Lecture 23: MIPS addressing

Stephen Checkoway
Slides from Cynthia Taylor

Today’s Class

- Finishing floating point
- Addressing in MIPS

CS History: The Deep Space Kraken

- Bug in the space simulation game Kerbal Space Program prior to 2012
- The game moved ships through space
- When ships moved at very high velocities, floating point errors would cause parts of the ship to be misaligned
- The game would interpret this as those parts breaking off the ship or colliding with each other
- Fixed it by moving space around the ship, instead of the ship through space

Multiplication in base-10 scientific notation

- Multiply $2.34 * 10^3$ and $4.56 * 10^5$
- Compute sign of the result
- Add together exponents
- Multiply fractions
- Normalize the result and give it the appropriate sign

$$1.000_2 \times 2^{-1} \quad \times \quad -1.110_2 \times 2^{-2}$$

A. $-1.110_2 \times 2^{-1}$

B. $-1.110_2 \times 2^{-2}$

C. $-1.110_2 \times 2^{-3}$

D. $-1.110_2 \times 2^{1}$

Floating point multiplication algorithm

Input: two single-precision, floating point numbers x, and y

Output: x * y

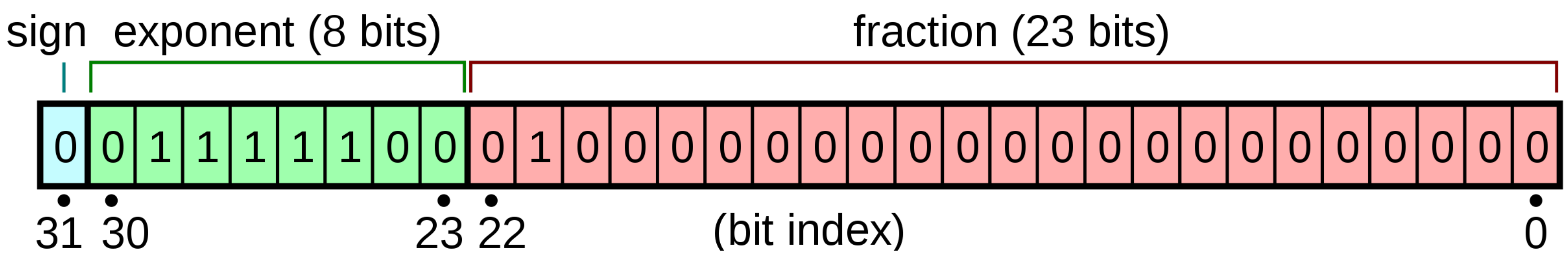
1. If either x or y is 0, return 0
2. Compute the sign of the result
3. Add the exponents, unbiasing first
4. Multiply the significands as 64-bit integers and shift right by 23 bits
5. Normalize the result and give it the appropriate sign

Why shift right?

“Multiply the significands as 64-bit integers and shift right by 23 bits”

Significands are 24 bit values (1 from the hidden bit, 23 from the fraction bits)

The FP number below has a significand of
1.01000000000000000000000



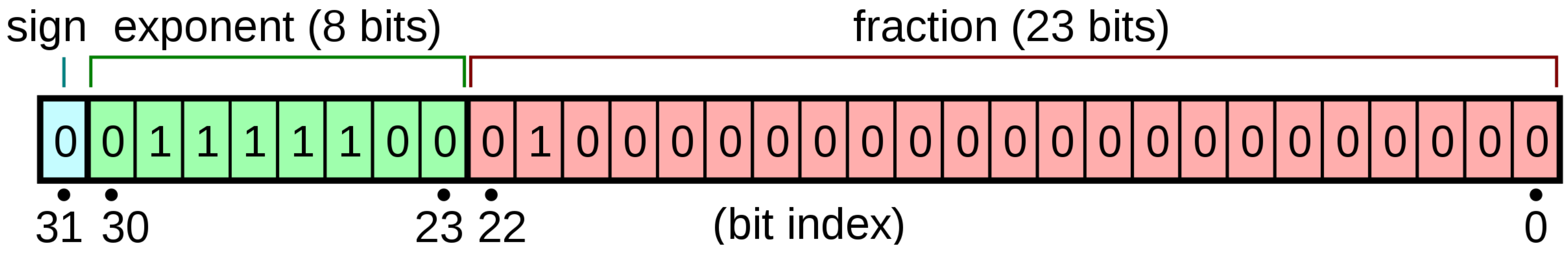
Why shift right?

If we drop the binary point in the significand and treat this as a binary integer we get $101000000000000000000000_2$

This value is the significand times 2^{23}

Multiplying $sig_x * 2^{23}$ by $sig_y * 2^{23} = (sig_x * sig_y) * 2^{46}$

Shifting the result right by 23 divides that by 2^{23} giving $(sig_x * sig_y) * 2^{23}$



FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`
  - Psuedoinstructions are easier to read: l.s, l.d, s.s, s.d

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`

- Double-precision arithmetic (operates on paired registers)
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`

Questions about Floating Point?

- Floating point is a finite approximation of the infinite number space

- This approximation leads to problems

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 fully specify a memory address or word of data in a single instruction

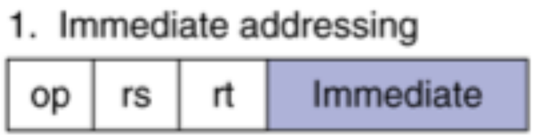
Register Addressing

- Which register the data is in is specified in the instruction

- 32 registers = 5 bits per register address

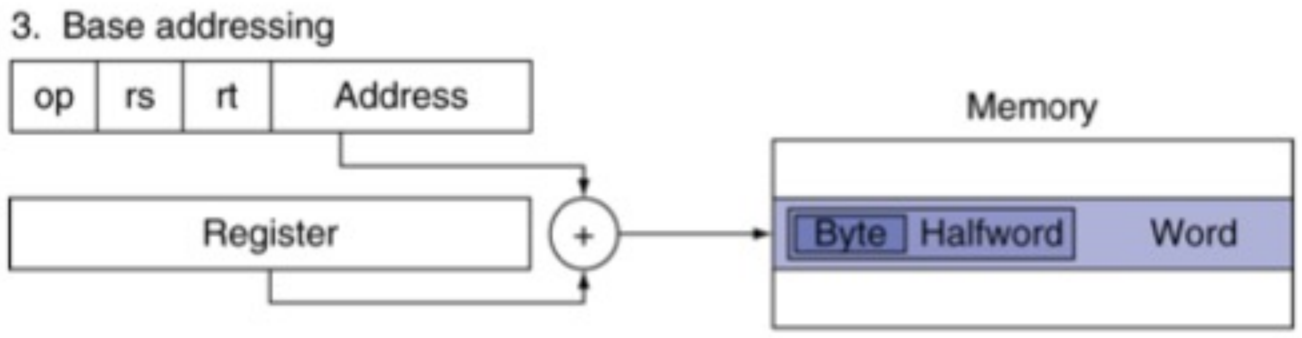
- Used in add, jr, etc

Immediate Addressing



- Data is a constant within instruction
- There is no memory address/register number, because we are just writing the information in the instruction itself
- 16 bits, can specify signed numbers from -2^{15} to $2^{15}-1$ and unsigned numbers up to $2^{16}-1$
- Used in addi, ori, etc

Base + Offset Addressing

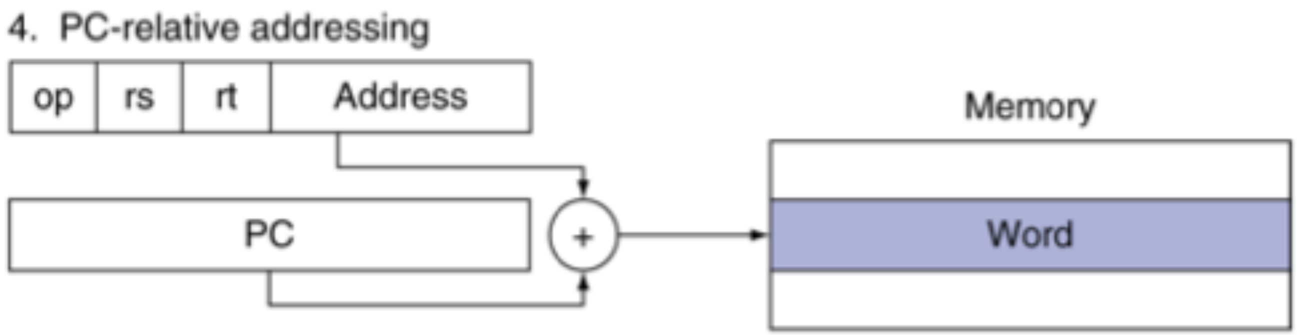


- Problem: 16 bits is not enough to address every word in memory
- Solution: Add a 16-bit offset to the 32-bit address in a register (the base)
- Used in lw, sw (and other loads/stores including floating point)

Branch and Jump: Recall

- Recall the basic instruction cycle
  - IR = Memory[PC]
  - PC = PC + 4
- Both branch and jump instructions change the value of the program counter

PC-relative Addressing



- Problem: Cannot hold a 32-bit memory address in a single 32-bit instruction (that also holds an opcode and two register numbers)

- Solution: Add an offset to the current value of the program counter
  – Used only for branches in MIPS; used for both control flow and data accesses in other architectures

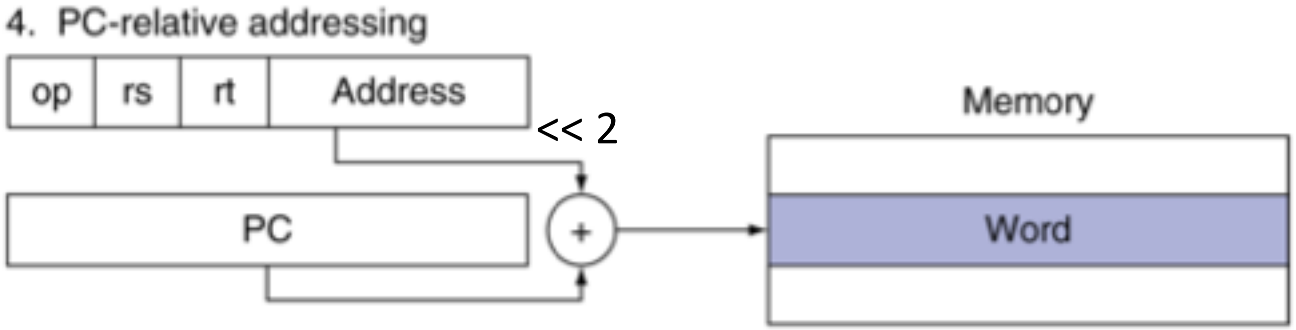
In a program, the target of a branch (if/for) is

A. always within 2^{15} instructions of the branch

B. usually within 2^{15} instructions of the branch

C. usually more than 2^{15} instructions away from the branch

PC-relative Addressing in MIPS

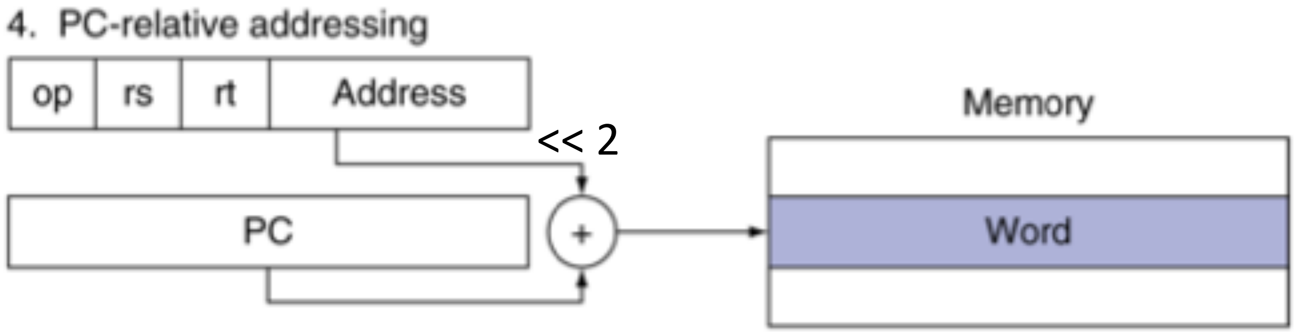


- Take 16 bit constant, shift left 2, add to value in PC

- Can access PC +/- 2^{17} bytes = PC +/- 2^{15} instructions

- 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 so they don’t need to be encoded in the instruction
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 << 2)

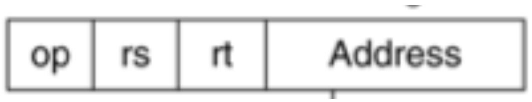
Consider the sequence of instructions:

0x480C bne

0x4810 add

0x4814 sub

0x4818 lw



If the immediate field of the bne instruction is 1, which instruction is the target of the branch?

A. bne
B. add
C. sub
D. lw
E. It's an error because addresses must be multiples of 4

We can create an infinite loop using a beq instruction with rs = rt = $zero and an immediate field of

A. -4
B. -1
C. 0
D. 4
E. Infinite loops are undefined behavior and so aren’t allowed

Branching Far Away

If branch target is too far to encode with 16-bit offset, assembler rewrites the code

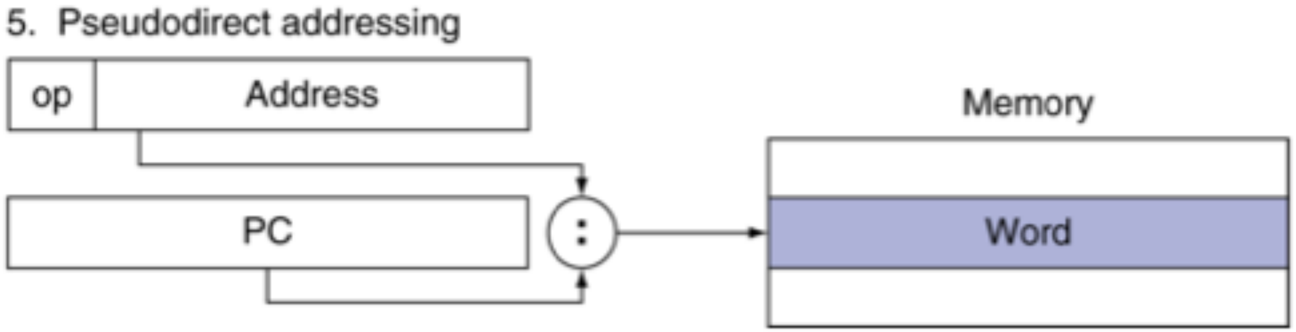
```
    beq   $t0, $t1, far_away
```

becomes

```
    bne   $t0, $t1, not_equal
    j     far_away
not_equal:
```

Questions on PC relative addressing?

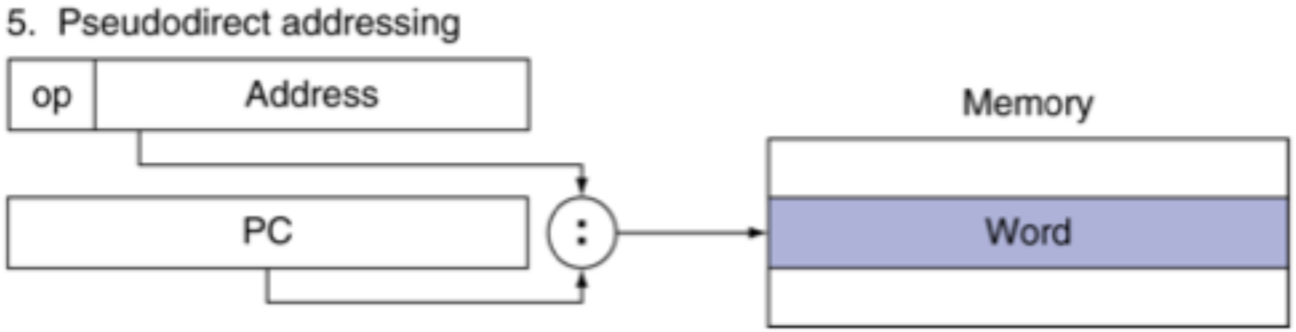
Pseudo-direct Addressing



- Problem: Cannot hold 32 bits of a memory address in the 32-6 bits of an instruction holding an opcode

- Solution: Use the most significant bits of the PC for the missing bits

Pseudo-direct 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 0x0000003. What is the address of the instruction the jal will jump to?

A. 0x00000003
B. 0xC0000003
C. 0xC0000007
D. 0xC000000C
E. 0xC800000C

Psuedo direct addressing

- Shift left by two
- Concatenate first four bits of PC + 4 with address

Questions about addressing?

Reading

- Next lecture: Datapath