

# CSCI 210: Computer Architecture

## Lecture 23: MIPS addressing

Stephen Checkoway

Slides from Cynthia Taylor

# Today's Class

- Addition & Multiplication in 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



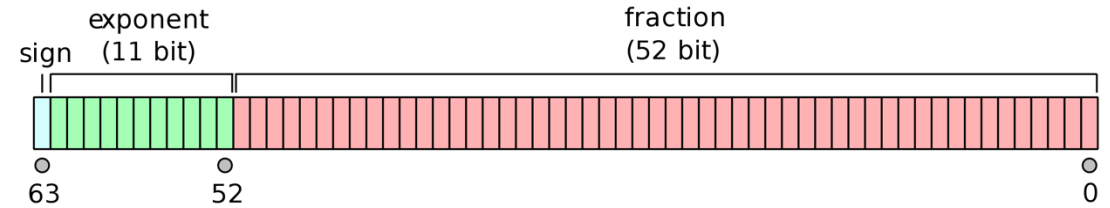
# Floating point addition algorithm

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

Output:  $x + y$

1. If either  $x$  or  $y$  is 0, return the other one
2. Denormalize  $x$  or  $y$  to give them both the larger exponent
3. Add the significands, taking sign into account
4. If the result is 0, return 0
5. Normalize the result

In Javascript, you perform the operation  $9007199254740992 + 1$ . What is the result?



Reminder:  $9007199254740992$  is  $2^{53}$

- A.  $-9007199254740992$
- B.  $9007199254740992$
- C.  $9007199254740993$
- D. This will cause an error
- E. None of the above

# How many times will this loop run in python?

```
a = 1000
while a != 0:
    a -= 0.001
```

- A. 1000 times
- B. 100000 times
- C. 1000000 times
- D. It will run forever
- E. None of the above

# This will run forever

```
a = 1000
while a != 0:
    a -= 0.001
```

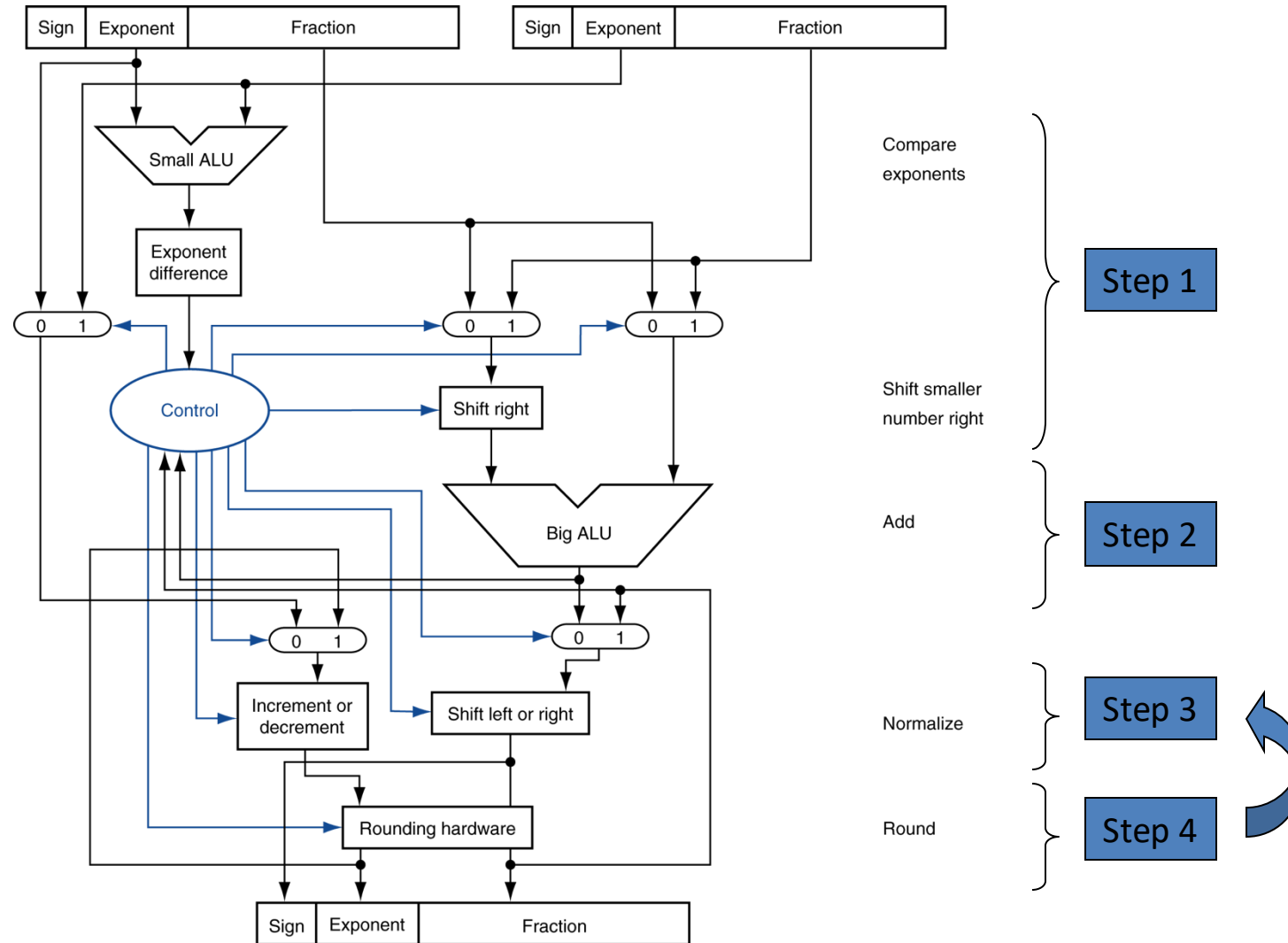
- a is never 0, instead it goes from 1.673494676862619e-08 to -0.00099999832650532314.
- Takeaway: Float equality is hard! Usually want to check within a small range

# FP Adder Hardware

- Much more complex than integer adder
- Doing it in the general purpose ALU/CPU would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles



# FP Adder Hardware



# Questions on Floating Point Addition?

# Multiplication in base-10 scientific notation

- Multiply  $2.34 \times 10^3$  and  $4.56 \times 10^5$
- Check sign
- Add together exponents
- Multiply fractions (with appropriate signs)
- Normalize

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

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

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

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

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

What issues could we run into doing this in binary floating point?

- A. Adding bias in exponent twice
- B. Added exponent could be greater than 127 or less than -126
- C. Multiplied fraction could be longer than 23 bit
- D. More than one of the above

# 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

# 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 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

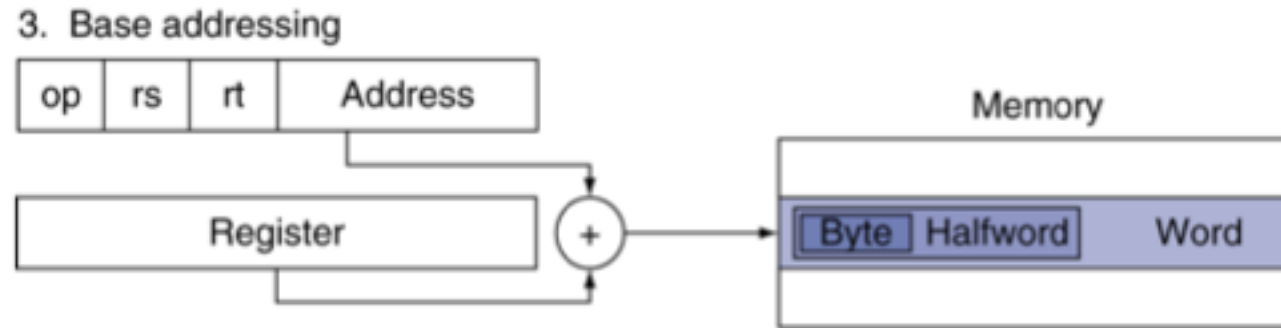
# Immediate Addressing

1. Immediate addressing



- Data is a constant within instruction
- There is no memory address/register number, 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

# Base + Offset Addressing

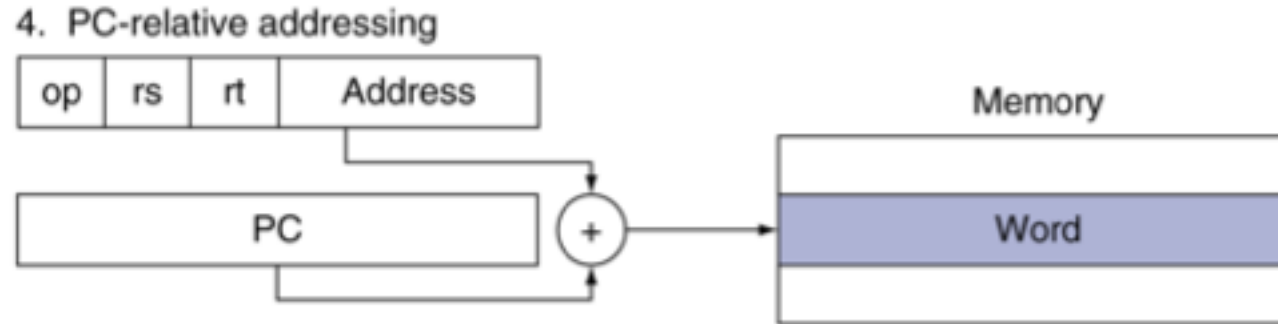


- Problem: 16 bits is not enough to address every word in memory
- Solution: Add the 16-bit offset to the 32-bit address within a register (the base)
- Used in lw, sw

# Branch and Jump: Recall

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

# PC-relative Addressing



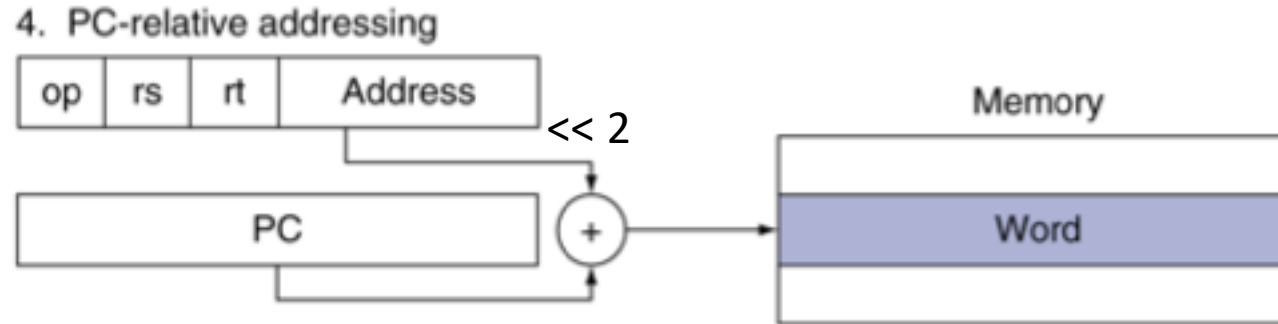
- Problem: Cannot hold a 32-bit memory address in a single 32-bit instruction (that also holds an opcode and two register numbers)
- Solution: Add an offset to the current value of the program counter

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

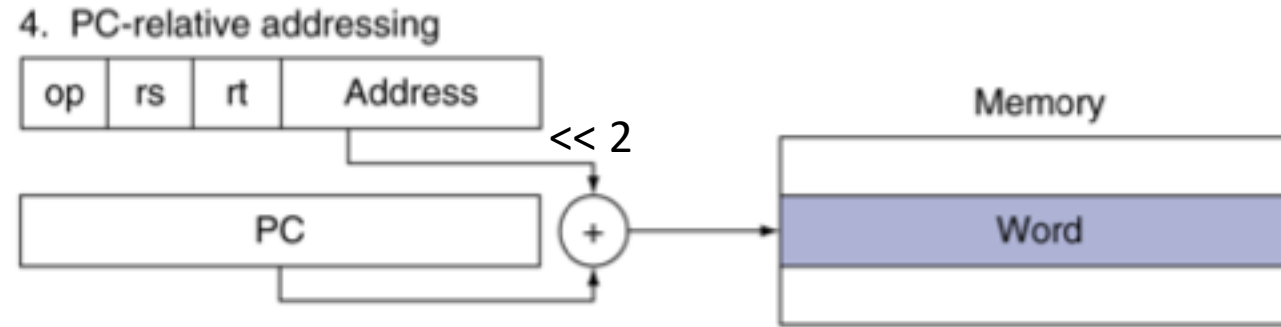


# PC-relative Addressing



- Take 16 bit constant, shift left 2, add to value in PC
- Can access  $PC \pm 2^{17}$  bytes =  $PC \pm 2^{15}$  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
- 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
```

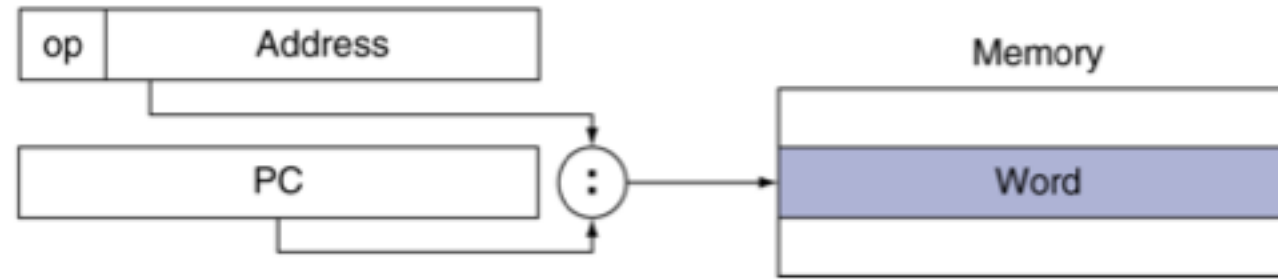
```
    j      far_away
```

```
not_equal:
```

# Questions on PC relative addressing?

# Pseudo-direct Addressing

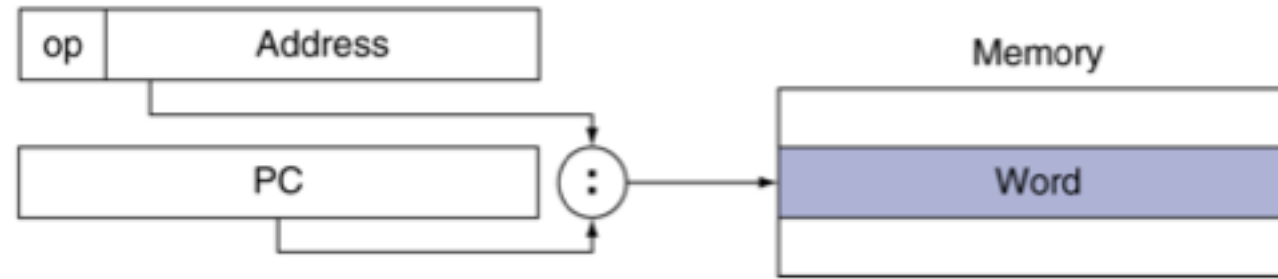
5. Pseudodirect addressing



- 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

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

Pseudo direct addressing

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



Questions about addressing?

# Reading

- Next lecture: Datapath