CSCI 210: Computer Architecture
Lecture 23: RISC-V addressing

Stephen Checkoway

Slides from Cynthia Taylor

Today’s Class

- Finishing floating point
- Addressing in RISC-V

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 significands
- Normalize the result and give it the appropriate sign

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

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

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

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

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

Floating point multiplication algorithm

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

Output: x * y

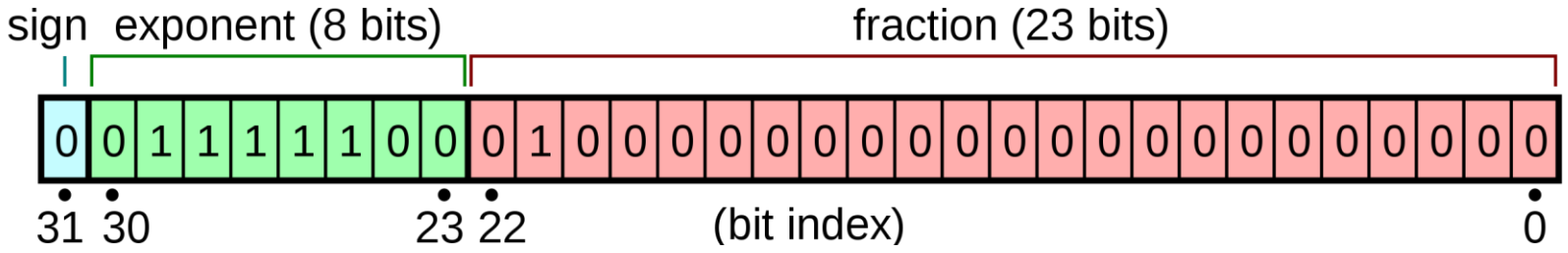
1. If either x or y is 0, return 0
2. Compute the sign of the result
3. Add the exponents, unbiasing first
4. Multiply the significands as 64-bit integers and shift right by 23 bits
5. Normalize the result and give it the appropriate sign

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



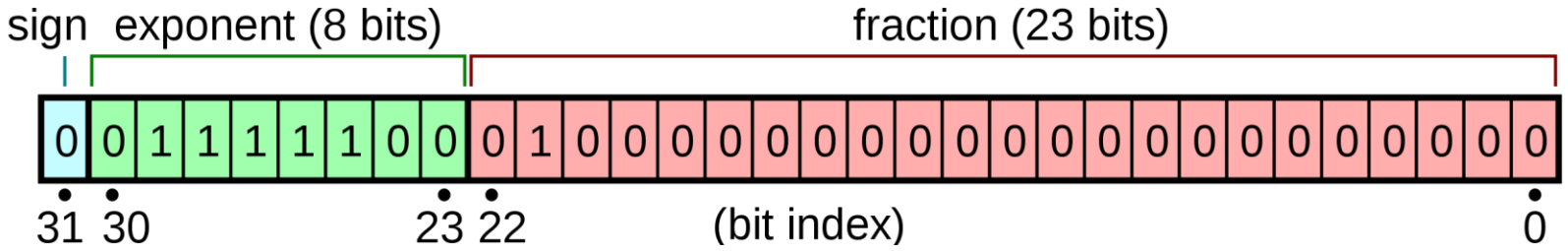
Why shift right?

If we drop the binary point in the significand and treat this as a binary integer we get $10100000000000000000000_2$

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 $(sig_x * sig_y) * 2^{23}$



Floating point instructions in RISC-V

- Defined in ISA extensions F (for single-precision) and D (for double-precision)
- Thirty-two 64-bit floating point registers (for D, 32-bit for F) f0–f31
- Load and store instruction examples
  - `flw f10, 0(s3)    # Floating point load word`
  - `fsw f8, 4(s3)     # Floating point store word`
  - `fld f0, 16(s3)    # Floating point load double`
  - `fsd f2, 0(s3)     # Floating point store double`

Other floating point instructions

- Addition: `fadd.s`, `fadd.d`
- Subtraction: `fsub.s`, `fsub.d`
- Multiplication: `fmul.s`, `fmul.d`
- Division: `fdiv.s`, `fdiv.d`
- Comparison: `feq.s`, `feq.d`, `flt.s`, `flt.d`, `fle.s`, `fle.d`
- Plus some others like min/max, and fused multiply-add

Questions about Floating Point?

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

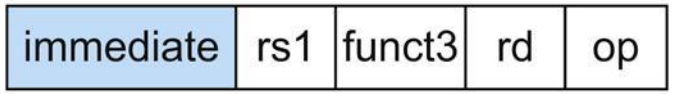
- This approximation leads to problems

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 64 bits
  - Data is in 32 bit words or 64-bit doublewords
- Can never fully specify a memory address or (double)word of data in a single instruction

Immediate Addressing

1. Immediate addressing



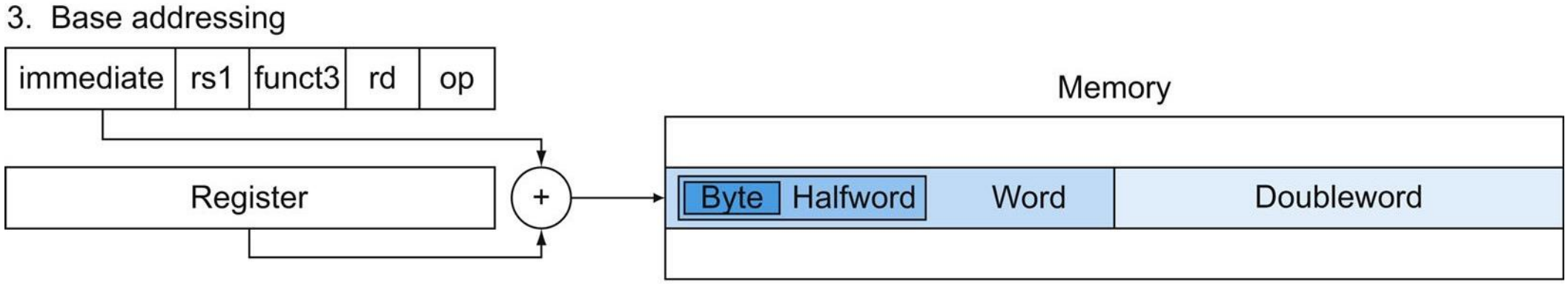
- Data is a constant within instruction
- There is no memory address/register number, because we are just writing the information in the instruction itself
- 12 bits, can specify signed numbers from -2^{11} to $2^{11} - 1$
- Used in addi, ori, etc.

Register Addressing



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

Base + Offset Addressing



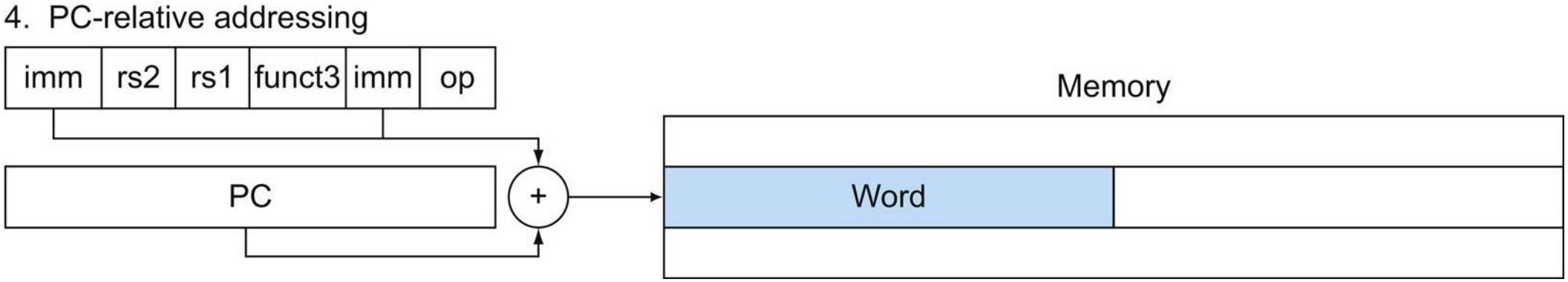
- Problem: 12 bits is not enough to address every word in memory
- Solution: Add a 12-bit offset to the 64-bit address in a register (the base)
- Used in ld, sd (and other loads/stores including floating point)

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 64-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 for control flow (jumps/branches) and global-data accesses in most architectures

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

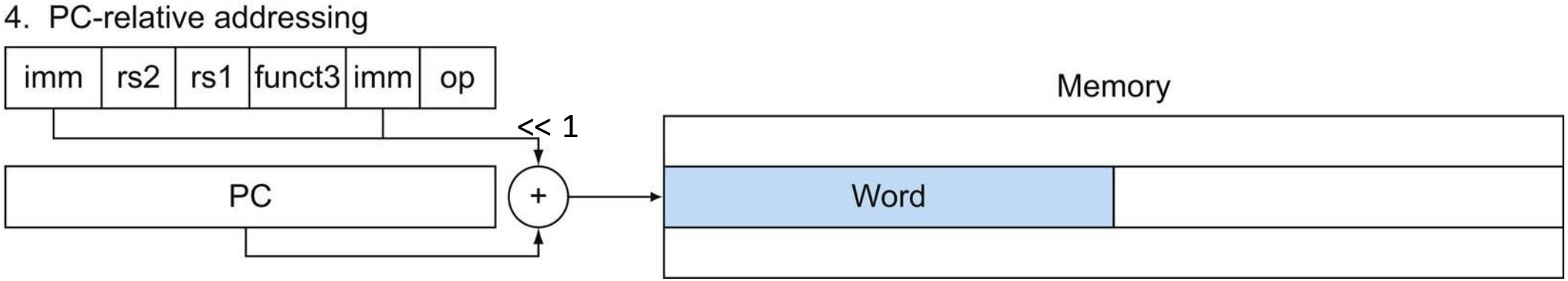
A. always within 1024 instructions of the branch

B. usually within 1024 instructions of the branch

C. usually more than 1024 instructions away from the branch

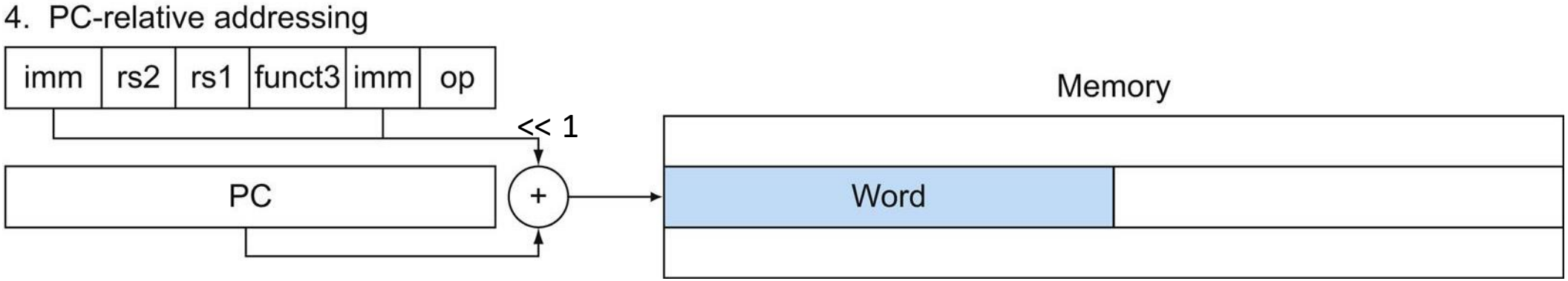
[Equivalently, do loop bodies (or if-else branches) usually/always have fewer than or more than 1024 instructions]

PC-relative Addressing in RISC-V



- Branch instructions: beq, bne, blt, bge, bltu, bgeu
  - Take 12 bit immediate field, shift left by 1, add to pc
  - Can access pc ± 2^{12} bytes = pc ± 2^{10} instructions
- Jump and link: jal
  - 20-bit immediate field, shift left by 1, add to PC
  - Can access pc ± 2^{20} bytes = pc ± 2^{18} instructions

Why do we shift left by one?



A. We use the last bit of the PC instead
B. We only branch to instructions that are multiples of 2 words away from the current instruction
C. Instructions are words and addresses specify bytes, so the last bit of the address will always be 0 so it doesn't need to be encoded in the instruction
D. None of the above

Why << 1 and not << 2?

- Instruction addresses are multiples of 4 so the least significant 2 bits are always zero
- We could branch farther away if we shifted by 2 instead of 1
  - (MIPS's branch instructions shift left by 2 for precisely this reason)
- RISC-V supports **compressed** instructions which are 2-bytes in length so branches target even addresses

Branching Far Away

If branch target is too far to encode with an effective 13-bit offset, the assembler rewrites the code

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

becomes

```
    bne t0, t1, not_equal
    jal zero, far_away
not_equal:
```

If father away than jal can reach (21-bit effective offset), auipc/jalr can target pc ± 2^{31} bytes

Questions about addressing?

Reading

- Next lecture: Datapath