CSE 210: Computer Architecture
Lecture 21: Floating Point

Stephen Checkoway

Slides from Cynthia Taylor

Today’s Class

- Finish up performance
- Start floating point

All Together Now

seconds

CPU Execution Time = Instruction Count X CPI X Clock Cycle Time

instructions

cycles/instruction

seconds/cycle

If we run the **same** program on two **different** machines with **different** ISAs, how do the number of instructions, CPI, and clock cycle time compare?

	Number of instructions	CPI	Clock cycle time
A	Same	Same	Same
B	Same	Same	Different
C	Same	Different	Different
D	Different	Different	Different
E	Different	Same	Same

If we run the **same** program on two **different** machines with the **same** ISA, how do the number of instructions, CPI, and clock cycle time compare?

	Number of instructions	CPI	Clock cycle time
A	Same	Same	Same
B	Same	Same	Different
C	Same	Different	Different
D	Different	Different	Different
E	Different	Same	Same

How we can measure CPU performance

- Millions of instructions per second
- Performance on benchmarks—programs designed to measure performance
- Performance on real programs

MIPS (not the name of the architecture)

MIPS = Millions of Instructions Per Second

$$= \frac{\text{Instruction Count}}{\text{Execution Time} * 10^6}$$

$$= \frac{\text{Clock rate}}{\text{CPI} * 10^6}$$

- program-dependent
- deceptive

Speedup

- Often want to compare performance of one hardware or software version against another

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

$$\text{Speedup (A over B)} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

$$\text{Speedup (A over B)} = \frac{ET_B}{ET_A}$$

Amdahl's Law

$$\text{Execution time after improvement} = \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} + \text{Execution Time Unaffected}$$

Amdahl's Law and Parallelism

- Our program is 90% parallelizable (segment of code executable in parallel on multiple cores) and runs in 100 seconds with a single core. What is the execution time if you use 4 cores (assume no overhead for parallelization)?

$$\text{Execution time after improvement} = \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} + \text{Execution Time Unaffected}$$

Selection	Execution Time
A	25 seconds
B	32.5 seconds
C	50 seconds
D	92.5 seconds
E	None of the above

Amdahl’s Law

- So what does Amdalh’s Law ***mean*** at a high level?

Selection	“BEST” message from Amdahl’s Law
A	Parallel programming is critical for improving performance
B	Improving serial code execution is ultimately the most important goal.
C	Performance is strictly tied to the ability to determine which percentage of code is parallelizable.
D	The impact of a performance improvement is limited by the percent of execution time affected by the improvement
E	None of the above

Key Points

- Be careful how you specify “performance”
- Execution time = IC * CPI * CT
- Make the common case fast

CS History: IEEE 754-1985

William Kahan
Photo credit: George M. Bergman,

- Pre-1980, different ISAs used different floating point implementations
- In 1976, John Palmer was managing implementing a floating-point coprocessor at Intel, and wanted a standard floating point
- He went to William Kahan, at UC Berkeley, who worked with Intel to develop a floating point standard
- Kahan, Jerome Coonen and Harold Stone put together a public draft proposal based on Kahan’s work with Intel
- This standard was implemented first by Intel in 1980, and then by other manufacturers
- In 1985 it became the official IEEE standard, and stayed the standard until it was updated in 2008

Floating Point

- Problem: Need a way to store non-integer values

- Including numbers with very large and very small magnitudes

- Want to do this the same way for every computer

How Humans Do This

- Scientific Notation
  - $1.2825 * 10^{2}$
  - $2.004 * 10^{38}$
  - $3.74 * 10^{-27}$
  - $-7.888889 * 10^{40}$

- Normalized Form
  – Always multiply by power of 10
  – Always 1 digit before the decimal point

How Computers Do This

- Floating Point Notation
  - $1.11_2 \times 2^2$
  - $1.0101_2 \times 2^{127}$
  - $1.110001_2 \times 2^{-126}$
  - $-1.0001_2 \times 2^{80}$

- Normalized Form
  – One digit before ~~decimal~~ binary point
  – Multiplied by power of two

101.10001_2

- 101.10001_2
- Integer part is 101_2 =
- Fractional part is 0.10001_2 =
- Total is

We know $101.10001_2 = 5.53125$. What is $1.0110001_2 \times 2^2$

A. 1.37578

B. 5.53125

C. 22.0125

D. None of the above

–17.125 in binary

- Step 1. Convert integer part: 17 =
- Step 2. Convert fractional part: .125 =
- Step 3. Add integer and fractional parts: 17.125 =
- Step 4. Normalize:
- Step 5. Add sign: –17.125 =

-0.75 in Binary is

A. $-1.1_2 \times 2^{-1}$

B. $-1.1_2 \times 2^{-2}$

C. $-1.001011_2 \times 2^{-1}$

D. $-1.001011_2 \times 2^{-2}$

E. None of the above

$1.2825 * 10^2$ in Binary is

A. $1.000000001_2 \times 2^{-7}$

B. $1.000000001_2 \times 2^6$

C. $1.1001000011001_2 \times 2^6$

D. $1.000000001_2 \times 2^7$

E. None of the above

Want to Represent $(-1)^s * 1.x * 2^e$ in 32 bits

- Divide up 32 bits into different sections
- 1 bit for sign s (1 = negative, 0 = nonnegative)
- 8 bits for exponent e
- 23 bits for significand 1.x

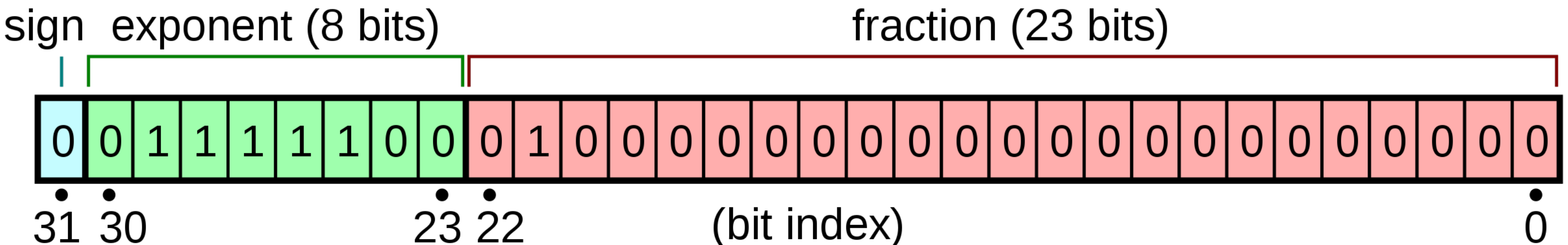
Goal: Get the most out of 32 bits

- The first number before our ~~decimal~~ binary point is always 1
  - $1.0001 * 2^4$
  - $-1.1011 * 2^{-16}$

- We don’t need to represent it in our remaining 23 bits—it is implicit!

$$(-1)^{s} * 1.x * 2^{e}$$

- 1 bit for sign s (1 = negative, 0 = positive)
- 8 bits for exponent e
- 0 bits for implicit leading 1 (called the "hidden bit")
- 23 bits for significand (without hidden bit)/fraction/mantissa x



1.001100101 * 2^7 as a single word

- 1.001100101 * 2^7 as a single word becomes
  - Sign =
  - Exponent =
  - Significand =

If we gave more bits to the exponent, and fewer to the fraction, we could represent

A. Fewer individual numbers

B. More individual numbers

C. Numbers with greater magnitude, but less precision

D. Numbers with smaller magnitude, but greater precision

Want To Make Comparisons Easy

- Can easily tell if number is positive or negative
  - Just check MSB bit
- Exponent is in higher magnitude bits than the fraction
  - Numbers with higher values will look bigger (as integers)
  - 0 00000111 10000000000000000000000 = $1.1 * 2^7$
  - 0 00001000 10000000000000000000000 = $1.1 * 2^8$

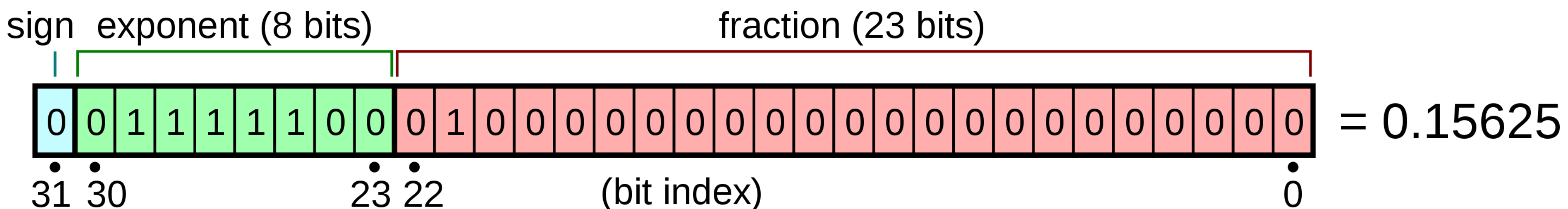
Problem with Two's Compliment

- 0 00000111 10000000000000000000000 = 1.1 * 2^7
- 0 00001000 10000000000000000000000 = 1.1 * 2^8
- 0 11111000 10000000000000000000000 = 1.1 * 2^{-8}

- Solution: Get rid of negative exponents!
  - We can represent 2^8 = 256 numbers: normal exponents -126 to 127 and two special values for zero, infinity, (and NaN and subnormals)
  - Add 127 to value of exponent to encode it, subtract 127 to decode

$$(-1)^s * 1.x * 2^e$$

- 1 bit for sign s (1 = negative, 0 = positive)
- 8 bits for exponent e + 127
- 0 bits for implicit leading 1 (called the "hidden bit")
- 23 bits for significand (without hidden bit)/fraction/~~mantissa~~ x



Encode 1.000000001 * 2^7 in 32-bit Floating Point

A. 0 00000111 00000000100000000000000

B. 0 00000111 10000000010000000000000

C. 0 10000110 00000000100000000000000

D. 0 10000110 10000000010000000000000

E. None of the above

How Can We Represent 0 in Floating Point (as described so far)?

A. 0 00000000 00000000000000000000000

B. 0 01111111 00000000000000000000000

C. 1 00000000 00000000000000000000000

D. More than one of the above

E. We can’t represent 0

Special Cases

Object	Exponent	Fraction
Zero	0	0
Infinity	255	0
NaN	255	Nonzero

Exception Events in Floating Point

- Overflow happens when a positive exponent becomes too large to fit in the exponent field
- Underflow happens when a negative exponent becomes too large (in magnitude) to fit in the exponent field

- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
  - Double precision – takes two MIPS words

s	E (exponent)	F (fraction)
1 bit	11 bits	20 bits

F (fraction continued)
32 bits

Reading

- Next lecture: Floating Point