

# Unit 1: Number Systems

CSE 220: System Fundamental I

Stony Brook University

Joydeep Mitra

# RISC vs CISC Machines

- Simple instructions, fewer in number
  - Fixed length instructions
  - Complexity in compiler
  - Only load/store instructions access memory
  - Three operands per instruction
  - Single-cycle instructions
  - Highly pipelined
  - Many complex instructions
  - Variable-length instructions
  - Complexity in hardware
  - Many instructions can access memory
  - One or two register operands per instruction
  - Multiple-cycle instructions
  - Less pipelined
- CISC machines are commercially more successful for historical reasons (e.g., Intel's x86).
  - Despite its advantages, RISC machines have taken time to develop due to lack of software support.
  - **Recommended reading: The Case for the Reduced Instruction Set Computer, Patterson & Ditzel.**

# Typical RISC Instructions

- We will focus on the **MIPS** (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) architecture – a RISC type
  - **Load** data from memory to CPU register
  - Copy (**Store**) data from CPU register to memory
  - **Add, Subtract, Multiply, Divide**, etc. data in CPU registers
  - **AND, OR, XOR, NOT**, etc. data in CPU registers
  - **Shift** and **Rotate** data in CPU registers
  - **Jump** codes to execute specific instructions
  - **Call** a subroutine (function) and **return** to caller

# Digital Abstraction

- Modern digital systems are made of complicated hardware.
  - We need a way to abstract away the hardware details and reason about data.
- What is the key idea behind representing data in digital systems?
  - Electronic machines are made of transistors.
  - Transistors act as switches with two distinct states (0 and 1).
  - The switches control the flow of electrons in a machine.
  - We can work with 0s and 1s to understand and develop systems that work with discrete values.
- The fundamental unit of data is the *binary digit (bit)*.
  - introduced by Claude Shannon in his seminal work “*A Mathematical Theory of Communication*”.
- With  $N$  bits we can have  $2^N$  combinations; each representing a unique value.

# Decimal Numbers

- To understand bits, let's take inspiration from decimal numbers (base 10).
- There are 10 decimal digits (0-9).
- Each column in a number has a weight ( $10^i$ ), where  $i$  is the  $i^{\text{th}}$  column starting at 0 (right to left)
  - $437 = 4 \cdot (10^2) + 3 \cdot (10^1) + 7 \cdot (10^0) = 400 + 30 + 7$
- In general, with  $N$  digits we can represent  $10^N$  decimal numbers.

# Binary Numbers

- Binary digits can be 0 or 1.
- Each column in a binary number has a weight ( $2^i$ ), where  $i$  is the  $i^{\text{th}}$  column starting at 0 (right to left). E.g.,
  - $1011 = 1*(2^3) + 0*(2^2) + 1*(2^1) + 1*(2^0) = 11$
- In general,  $N$  bits can be used to represent  $2^N$  binary numbers.

# Converting Decimal to Binary

- Continuously divide the number by 2 till the quotient is 0
- After every division record the remainder
- The binary number is the remainder in every step from last to first

Convert  $123_{10}$  to binary

$$123 / 2 = 61 \text{ (rem. 1)}$$

$$61 / 2 = 30 \text{ (rem. 1)}$$

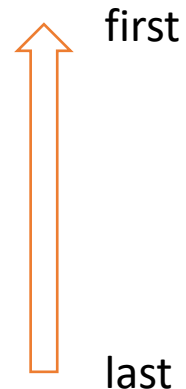
$$30 / 2 = 15 \text{ (rem. 0)}$$

$$15 / 2 = 7 \text{ (rem. 1)}$$

$$7 / 2 = 3 \text{ (rem. 1)}$$

$$3 / 2 = 1 \text{ (rem. 1)}$$

$$1 / 2 = 0 \text{ (rem. 1)}$$



Answer:  $1111011_2$

# Hexadecimal Numbers

- It maybe inconvenient to work with binary numbers.
- An alternate representation in hexadecimal (4 bits at a time).
  - Each digit can be one of 16 ( $2^4$ ) numbers – 0,1,2,...,9,A,B,C,D,E,F
  - 10110011 = B3
- What is the equivalent decimal form?
  - Each column in a hexadecimal number has a weight ( $16^i$ ), where i is the  $i^{\text{th}}$  column starting at 0 (right to left)
    - 10110011 = B3 =  $11 \cdot (16^1) + 3 \cdot (16^0) = 179$



# Converting Decimal to Hexadecimal

- Continuously divide the number by 16 till the quotient is 0
- After every division record the remainder
- The hexadecimal number is the remainder in every step from last to first

Convert  $123_{10}$  to hex

$$\begin{array}{l} 123 / 16 = 7 \text{ (rem. } 11 \text{ )} \\ 7 / 16 = 0 \text{ (rem. } 7 \text{ )} \end{array}$$



first  
last

Answer:  $7B_{16}$

# Octal Numbers

- Alternatively, one could use the octal representation (base 8)!
- In octal, each digit is one of 8 ( $2^3$ ) numbers : 0,1,2,...,7
  - E.g., 10110011 = 263
- What is the equivalent decimal form?
  - Each column in an octal number has a weight ( $8^i$ ), where i is the  $i^{\text{th}}$  column starting at 0 (right to left)
    - $263_8 = 2*(8^2) + 6*(8^1) + 3*(8^0) = 179_{10}$

# Converting Decimal to Octal

- Continuously divide the number by 8 till the quotient is 0
- After every division record the remainder
- The octal number is the remainder in every step from last to first

Convert  $179_{10}$  to octal

$$179 / 8 = 22 \text{ (rem. 3)}$$

$$22 / 8 = 2 \text{ (rem. 6)}$$

$$2 / 8 = 0 \text{ (rem. 2)}$$

 first  
last

Answer:  $263_8$

# Data Formats

- A group of *eight bits* is a *byte*.
  - One of  $2^8 = 256$  possibilities.
- Microprocessors handle data in *words*.
  - Word size depends on the microprocessor's architecture.
  - Modern microprocessors use 64-bit *words*.
  - Older ones used 32-bit *words*.
- Bits in the 1<sup>st</sup> column are called *least significant bit (lsb)*.
- Bits in the other end are called *most significant bit (msb)*.
- *1024 bits* is a *Kb*; a million bits is a *Mb* and a billion *Gb*.
  - Similar units for bytes – *KB*, *MB*, and *GB*.

1011001  
↑                    ↑  
*msb*                    *lsb*


# Binary Addition

- Binary addition is like decimal addition
  - Add digits in each column
  - If result is greater than digit *carry over* to adjacent column
  - Axioms of binary addition
    - $1+1 = 10$ ;  $1+0 = 0+1 = 1$ ;  $0+0 = 0$

$$\begin{array}{r} \phantom{+} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{1} \phantom{0} \phantom{0}_2 \\ + 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} 0 \phantom{0}_2 \\ \hline 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0} 1 \phantom{0} 0 \phantom{0}_2 \end{array}$$

# Overflow

- *Overflow* is an erroneous condition where the result is bigger than the no. of digits available.
- When overflow occurs, the leftmost carry is simply discarded.
- Not being mindful of such size constraints leads to incorrect results.

$$\begin{array}{r} \phantom{+} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{0} \overset{1}{1} \overset{1}{1} 1_2 \\ + 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1_2 \\ \hline \textcolor{red}{1} \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0_2 \end{array}$$


**Overflow! (assuming only 8 bits are available)**

- What is the least decimal number for which an overflow will occur when adding two positive 4-bit binary numbers?
  - *Ans: 16*

# Signed Binary Numbers

- How can we represent negative numbers in binary?
- Two schemes are used commonly
  - Sign/magnitude
  - Two's complement

# Sign/Magnitude Numbers

- The *msb* indicates the sign; remaining  $N-1$  bits represent magnitude
  - 0 indicates +ve number and 1 indicates -ve number
  - $5_{10} \Rightarrow 0101_2$
  - $-5_{10} \Rightarrow 1101_2$
- An  $N$ -bit sign/magnitude represents numbers  $[-2^{N-1}+1, 2^{N-1}-1]$
- The Sign/magnitude representation has issues:
  - Two representations of 0 (+0 and -0); weird!
  - Binary addition does not work!
    - $-5_{10} + 5_{10} = 1101_2 + 0101_2 = 10010_2 \Rightarrow$  **garbage value!**



# Two's Complement Representation

- Introduced to address issues of sign/magnitude
- Same as unsigned binary numbers, except *msb* has weight  $-2^{N-1}$ 
  - $0101 \Rightarrow 0*(-2^3) + 1*2^2 + 0*2^1 + 1*2^0 = 5$
  - $1011 \Rightarrow 1*(-2^3) + 0*2^2 + 1*2^1 + 1*2^0 = -5$
- Common two's complement N-bit representations
  - $0_{10} \Rightarrow 00...000_2$  (N zeroes)
  - $-1_{10} \Rightarrow 11...111_2$  (N ones)
  - Highest +ve number  $\Rightarrow 01...111_2 \Rightarrow 2^{N-1}-1$
  - Lowest -ve number  $\Rightarrow 10...000_2 \Rightarrow -2^{N-1}$

# Taking Two's Complement

- A technique to find two's complement representation of a binary number
  - Flip each bit from 1 to 0 and 0 to 1 (Also called ***One's complement representation***)
  - Add 1 to the *lsb*

00000011      One's complement      →      11111100       $\xrightarrow{+1}$       11111101

- Useful for representing negative numbers in binary
  - What is the 8-bit two's complement representation of  $-99$ ?
    - $99_{10} = 01100011_2$
    - Flip all bits to left of the rightmost 1 (same as taking 1's complement and adding 1)
    - Answer: 10011101
  - What is the decimal equivalent of the two's complement number 11001010?
    - 1 in *msb* indicates -ve number
    - Flip all bits to left of the rightmost 1
    - $00110110_2 = 54_{10}$
    - Answer:  $-54$

# Magic of Two's Complement

- Consider two's complement of all 3-bit numbers

-4 -> 100 -> 100

-3 -> 011 -> 101

-2 -> 010 -> 110

-1 -> 001 -> 111

-0 -> 000 -> 000

0 -> 000 -> 000

} Only 1 representation of 0!

1 -> 001 -> 001

2 -> 010 -> 010

3 -> 011 -> 011

- Two's complement addition works for both +ve and -ve numbers!
  - $-2_{10} + 1_{10} \Rightarrow 1110_2 + 0001_2 \Rightarrow 1111_2 \Rightarrow -1_{10}$
  - $-7_{10} + 7_{10} \Rightarrow 1001_2 + 0111_2 \Rightarrow 10000_2 \Rightarrow 0_{10}$  (1 in 5<sup>th</sup> bit discarded)

# Overflow In Two's Complement Addition

- Unlike unsigned numbers, *carry in msb does not indicate overflow.*
- Overflow will occur if add result is more than  $2^{N-1}-1$  or less than  $-2^{N-1}$ .
  - Adding numbers with ***same sign bit and result with opposite sign bit will cause overflow.***
- Adding numbers with opposite sign will *never cause overflow.*
- Add two positive 4-bit two's complement numbers
  - $4_{10} + 5_{10} \Rightarrow 0100_2 + 0101_2 \Rightarrow 1001_2 \Rightarrow -7_{10}$  (Overflow!)

# Previous Class

- Number Representations
  - Binary (base 2), Hexadecimal (base 16), Octal (base 8)
  - E.g., represent 49 in 8-bit binary, hexadecimal, and octal.
- Signed numbers are represented in
  - Sign/magnitude form
    - 1 in MSB indicates -ve; 0 in MSB indicates +ve
  - Two's complement form
    - Number is still -ve if MSB is set to 1.
    - If number is -ve then flip all bit (1's complement) and add 1.
  - E.g., represent -49 in Two's complement.
  - What about 49 in Two's complement?

# Extending Signed Bits

- Extending unsigned numbers is easy! Append 0s
  - $3_{10}$  in 3 bits  $\Rightarrow 011_2$
  - $3_{10}$  in 8 bits  $\Rightarrow 00000011_2$
- For two's complement numbers, *signed bit must be copied to all most significant bits*
  - $3_{10}$  in 3 bits  $\Rightarrow 011_2$
  - $3_{10}$  in 8 bits  $\Rightarrow 00000011_2$  (after extending)
  - $-3_{10}$  in 8 bits  $\Rightarrow 101_2$
  - $-3_{10}$  in 8 bits  $\Rightarrow 11111101_2$  (after extending)

# Comparing Number Systems

- Range of N-bit numbers

System	Range
Unsigned	$[0, 2^N - 1]$
Sign/Magnitude	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Two's Complement	$[-2^{N-1}, 2^{N-1} - 1]$

- 4-bit binary encoding

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111    Unsigned

1000 1001 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110 0111    Two's Complement

1111 1110 1101 1100 1011 1010 1001    0000  
1000    0001 0010 0011 0100 0101 0110 0111    Sign/Magnitude

# What About Fractions?

- Two schemes to represent fractions:
  - **Fixed-point system** – *fixed* no. of bits represent integer part and remaining denote fractional part.
  - **Floating-point system** – akin to scientific notation with a *mantissa* and *exponent*.
    - *E.g.*, 4100 in scientific notation is  $4.1 \times 10^3$
    - The decimal point floats to the position right after the most significant bit.



# Fixed Point Representation

- Rule: generate the bits in *left-to-right order*, starting from the radix point:
  - Multiply the decimal value by 2. If the product is greater than or equal to 1, the next bit is 1. Otherwise, the next bit is 0.
  - Drop the integer part to get a value less than 1.
  - Continue until 0 is reached (a terminating expansion) or a pattern of digits repeats (a non-terminating expansion)

# Fixed Point Representation

- Convert  $0.4_{10}$  to binary
  - $0.4 \times 2 = 0.8$      $0.8 < 1$ , so write a **0**     $\cong 0.\textcolor{red}{0}$
  - $0.8 \times 2 = 1.6$      $1.6 \geq 1$ , so write a **1**     $\cong 0.\textcolor{red}{0}\textcolor{blue}{1}$ 
    - Drop the integer part
  - $0.6 \times 2 = 1.2$      $1.2 \geq 1$ , so write a **1**     $\cong 0.\textcolor{red}{0}\textcolor{blue}{1}\textcolor{green}{1}$ 
    - Drop the integer part
  - $0.2 \times 2 = 0.4$      $0.4 < 1$ , so write a **0**     $\cong 0.\textcolor{red}{0}\textcolor{blue}{1}\textcolor{green}{1}\textcolor{violet}{0}$
  - *Since we arrived at a decimal fraction we have already seen, the pattern will repeat*
- Final answer:  $0.\overline{0110}_2$

# Fixed-Point Representation

- Convert  $13.85_{10}$  to binary
- $13_{10} = 1101_2$  (integer part)
- Fractional part:
  - $0.85 \times 2 = 1.7$        $1.7 \geq 1$ , so write a 1       $\cong 0.1$
  - $0.7 \times 2 = 1.4$        $1.4 \geq 1$ , so write a 1       $\cong 0.11$
  - $0.4 \times 2 = 0.8$        $0.8 < 1$ , so write a 0       $\cong 0.110$
  - $0.8 \times 2 = 1.6$        $1.6 \geq 1$ , so write a 1       $\cong 0.1101$
  - $0.6 \times 2 = 1.2$        $1.2 \geq 1$ , so write a 1       $\cong 0.11011$
  - $0.2 \times 2 = 0.4$        $0.4 < 1$ , so write a 0       $\cong 0.110110$
  - We will get  $\cong 0.1101100110 \dots = 0.11\overline{0110}$
- Final answer:  $1101.11\overline{0110}_2$

# Fixed-Point To Decimal

- Every column *after* the “.” has weight  $i$ , where  $i$  is the column number from *left to right* starting at -1
- Every column *before* the “.” has weight  $i$ , where  $i$  is the column number from *right to left* starting at 0
- Convert  $0110.1100_2$  to decimal
  - $0110.1100_2$   
 $\Rightarrow 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 + 1*2^{-1} + 1*2^{-2} + 0*2^{-3} + 0*2^{-4}$   
 $\Rightarrow 6.75$  (Final Answer)

# Floating-Point Representation

- Fixed-point numbers have a constant number of integer and fractional bits
  - Hinders expressing very small and large numbers
  - What is the solution? Reserve some bits to represent an exponent
- A format that is inspired by scientific notation
  - $M * 10^N$ , M is a real number (or mantissa) and N is the exponent
    - $300 \Rightarrow 3 * 10^2$
    - $4321.768 \Rightarrow 4.321768 * 10^3$
    - $0.2 \Rightarrow 2 * 10^{-1}$

# IEEE 754 Floating-Point Standard

- IEEE 754 is essentially a form of **scientific notation** but written in binary:  
 $\pm 2^{\text{exponent}} \times 1.\text{fraction}$
- This format can be encoded using three fields: a **sign bit** ( $s$ ), an **exponent** ( $e$ ) and a **fraction** ( $f$ ), also called the **mantissa**
- IEEE 754 **single-precision format** requires 32 bits
- IEEE 754 **double-precision format** requires 64 bits

# IEEE 754 Floating-Point Standard

- 32-bit version:

1 bit	8 bits	23 bits
sign	exponent	fraction (mantissa)

- 64-bit version:

1 bit	11 bits	52 bits
sign	exponent	fraction (mantissa)

- Sign bit: 0 (positive) or 1 (negative)
- Fraction: contains the digits to the right of the binary point
  - **Normalized**: the digit to the left of the point is always 1 and is not represented.

# IEEE 754 Floating-Point Standard

- What will be the IEEE 754 form of  $228_{10}$ ?
  - $228_{10} \Rightarrow 11100100_2 \Rightarrow 1.1100100_2 * 2^7$

1 bit (sign)	8 bits (exponent)	23 bits (mantissa)
0	00000111	11 0010 0000 0000 0000 0000

*Notice how the leftmost 1 is not part of mantissa*



# Biased Exponent

- The exponent may be +ve or -ve; we haven't accounted for that!
- Soln: use a ***biased*** exponent
  - Original exponent + constant bias
  - For 8-bit exponent we use 127 as the bias
- So now, what will be the IEEE 754 form of  $228_{10}$ ?
  - $228_{10} \Rightarrow 11100100_2 \Rightarrow 1.1100100_2 * 2^7$
  - Biased exponent =  $127 + 7 = 134 = 10000110$

1 bit (sign)	8 bits (exponent)	23 bits (mantissa)
0	10000110	11 0010 0000 0000 0000 0000

# Decimal to IEEE 754 Example

- Encode  $13.4_{10}$  in 32-bit IEEE 754 floating-point format
- Positive number  $\rightarrow s = 0$
- From earlier:  $0.4_{10} = 0.\overline{0110}_2$
- $1101.011001100110011001100110...$
- Normalize: move binary point 3 places to left
  - $e = 3$       Add 127:  $3 + 127 = 130_{10} = 10000010$
- $101011001100110011001100110...$  (dropped leading 1)
- Take 23 bits: 1010 1100 1100 1100 1100 110

<i>sign</i>	<i>exponent</i>	<i>mantissa</i>
0	10000010	10101100110011001100110

# IEEE 754 to Decimal Example

- What decimal value has the following IEEE 754 encoding?

10111110011000000000000000000000

1 01111100 110000000000000000000000

- $sign = 1$
- $exponent = (64 + 32 + 16 + 8 + 4) - 127 = -3$
- $mantissa = 0.5 + 0.25 = 0.75$
- Answer:  $-1.75 \times 2^{-3} = -0.21875_{10}$

# IEEE 754 Special Values

- The smallest 000...0 and largest 111...1 exponents are reserved for the encoding of special values:
  - Zero (two encodings):
    - $s = 0 \text{ or } 1, e = 000 \dots 0, f = 000 \dots 0$
  - Infinity:
    - $+\infty: s = 0, e = 111 \dots 1, f = 000 \dots 0$
    - $-\infty: s = 1, e = 111 \dots 1, f = 000 \dots 0$
  - NaN (not a number):
    - $s = 0 \text{ or } 1, e = 111 \dots 1, f = \text{non-zero}$
    - Can result from division by zero,  $\sqrt{-1}$ ,  $\log(-5)$ , etc.

# IEEE 754 Format Summary

Property	Single-Precision	Double-Precision
Bits in Sign	1	1
Bits in Exponent	8	11
Bits in Fraction	23	52
Total Bits	32	64
Exponent Encoding	excess-127	excess-1023
Exponent Range	$-127$ to $128$	$-1023$ to $1024$
Decimal Range	$\cong 10^{-38}$ to $10^{38}$	$\cong 10^{-308}$ to $10^{308}$

# Limitations of Floating-Point Numbers

- There are  $\sim 2^{32}$  different values that can be represented in single-precision floating point.
- This is the same as the number of values that can be represented using 32-bit integer encodings.
- Many values do not have accurate floating-point representations:
  - Examples:  $(33554431.0)_{10}$  and  $0.33554431_{10}$
- Try assigning these to a **float** variable in Java and then printing them out.
- *Caution: results of floating-point calculations are usually not exact.*
- Never use floating point when exact results are essential or in equality checks, such as in conditional statements