A number in context

- Look closely …
  - a 1 or a 0 in the right place
  - Is also a sign

- Break a sequence of bits here or there
  - And it gives you powers
  - To size up who-ville or
    the universe’s atoms

- The only clarity that matters? Context
- The interpretation that follows? Unambiguous

Frequently asked questions from the previous class survey

- Signed and unsigned numbers
  - How does the system know what’s a positive or negative number?
- Multiplication of binary numbers
- Word sizes and what they can represent
- Two’s complement!
Topics covered in this lecture

- Signed numbers
  - Two's complement
  - One’s complement
- Floating point numbers
- Hexadecimal numbers

Signed Numbers

Words are but the signs of ideas.
— Samuel Johnson (1709-1784)
Signed Numbers

- An n-bit binary system can code $2^n$ different things
- If we have to represent signed (+ and −) numbers in binary code, a solution is to split the available space into two subsets
  - One for representing nonnegative (+) numbers, and
  - The other for representing negative (−) numbers

Desirable properties of signed number representations

- Ideally, the coding scheme should be chosen such that the introduction of signed numbers
  - Complicates the hardware implementation of arithmetic operations as little as possible
- This challenge has led to the development of several coding schemes for representing signed numbers in binary code
The solution used today in almost all computers?

- Two's complement method, also known as radix complement

Example:
- Consider a binary system that uses a word size of $n$ bits
- The two's complement binary code that represents negative $x$ is taken to be the code that represents $2^n - x$
- Representation of $-x \rightarrow 2^n - x$

Two’s complement representation of 4-bit numbers

- Recall, in a $n$-bit number system
- $-x \rightarrow 2^n - x$
- For example, in a 4-bit binary system:
  - $-7$ is represented using the binary code associated with $2^4 - 7 = 9$
    - Which happens to be 1001

<table>
<thead>
<tr>
<th>4-bit binary</th>
<th>Base-10</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>$-8$</td>
<td>$2^4 - 8 = 16 - 8 = 8$</td>
</tr>
<tr>
<td>1001</td>
<td>$-7$</td>
<td>$2^4 - 7 = 16 - 7 = 9$</td>
</tr>
<tr>
<td>1010</td>
<td>$-6$</td>
<td>$2^4 - 6 = 16 - 6 = 10$</td>
</tr>
<tr>
<td>1011</td>
<td>$-5$</td>
<td>$2^4 - 5 = 16 - 5 = 11$</td>
</tr>
<tr>
<td>1100</td>
<td>$-4$</td>
<td>$2^4 - 4 = 16 - 4 = 12$</td>
</tr>
<tr>
<td>1101</td>
<td>$-3$</td>
<td>$2^4 - 3 = 16 - 3 = 13$</td>
</tr>
<tr>
<td>1110</td>
<td>$-2$</td>
<td>$2^4 - 2 = 16 - 2 = 14$</td>
</tr>
<tr>
<td>1111</td>
<td>$-1$</td>
<td>$2^4 - 1 = 16 - 1 = 15$</td>
</tr>
</tbody>
</table>
Let’s take a closer look at +7 and −7

- +7 = 0111
- −7 = 1001
- +7 − −7 = 0000 (ignoring the overflow bit)

An n-bit binary system with two’s complement representation has attractive properties

- The system codes $2^n$ signed numbers, ranging from $-2^{n-1}$ to $(2^{n-1} - 1)$
- The code of any nonnegative number begins with a 0
- The code of any negative number begins with a 1
- To obtain the binary code of $-x$ from the binary code of $x$?
  - Flip all the bits of $x$ and add 1 to the result
Two’s complement: binary code of $-x$ from the binary code of $x$?

- Flip all the bits of $x$ and add 1 to the result

<table>
<thead>
<tr>
<th>4-bit binary</th>
<th>Base-10</th>
<th>Flip the bits</th>
<th>Add 1</th>
<th>Base-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>1</td>
<td>1110</td>
<td>1111</td>
<td>- 1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>1101</td>
<td>1110</td>
<td>- 2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>1100</td>
<td>1101</td>
<td>- 3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>1011</td>
<td>1100</td>
<td>- 4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>1110</td>
<td>1011</td>
<td>- 5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>1101</td>
<td>1010</td>
<td>- 6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>1100</td>
<td>1001</td>
<td>- 7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>0111</td>
<td>1000</td>
<td>- 8</td>
</tr>
</tbody>
</table>

Another attractive feature of two’s complement

- Subtraction is handled as a special case of addition
- To illustrate, consider $5 - 7$ in our 4-bit binary number system
  - This $5 + (-7)$
  - $0101 + 1001$
  - = 1110
  - Which is $-2$
Implications of the two’s complement method? [1/2]

- The two's complement method enables adding and subtracting signed numbers
  - Using nothing more than the hardware required for adding positive numbers!
- Every arithmetic operation, from multiplication to division to square root, can be implemented reductively using binary addition

Implications of the two’s complement method? [2/2]

- A huge range of computer capabilities rides on top of binary addition
- The two's complement method obviates the need for special hardware for adding and subtracting signed numbers
- The two's complement method is one of the most remarkable and unsung heroes of applied computer science
Another useful concept to know ... one’s complement

- A not so successful attempt to represent signed numbers
- Get to negative numbers by taking positive numbers and flipping all the bits (i.e., 1 becomes 0 and 0 becomes 1)
### One’s complement

- Flipping each bit of 0111 (+7) yields 1000 (–7)
- Has **two** different representations for zero

<table>
<thead>
<tr>
<th>Sign</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>+7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>+6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>+5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>+4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>+3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>+2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>+0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>–0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>–1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>–2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>–3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>–4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>–5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>–6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>–7</td>
</tr>
</tbody>
</table>

---

### One’s complement: Addition is a little more complicated

- End-around carry

0 0 1 0 +2

1 1 1 0 -1

0 0 0 0 0
Floating Point Representation

Counting is the religion of this generation, its hope and salvation.
—Gertrude Stein (1874–1946)

General-purpose computers are built to solve general-purpose problems

- Which involve a wide range of numbers
- You can get an idea of this range by skimming a physics textbook
  - There are tiny numbers such as Planck’s constant \((6.63 \times 10^{-34} \text{ joule-seconds})\) and
  - Huge numbers such as Avogadro’s constant \((6.02 \times 10^{23} \text{ molecules/mole})\)
  - This is a range of \(10^{57}\), which comes out to about \(2^{191}\)
  - That’s almost 200 bits!
- Bits just aren’t cheap enough to use a few hundred of them to represent every number, so we need a different approach
Look to what we have done in the past

- **Scientific notation** represents a large range of numbers by (how else?) creating a *new context* for interpretation
  - It uses a number with a *single digit* to the left of the decimal point, called the *mantissa*, multiplied by 10 raised to some power, called the *exponent*

\[
\begin{align*}
6.63 \times 10^{-34} \\
\text{Mantissa} & \quad \text{Exponent}
\end{align*}
\]

Floating points

- Computers use the same system, except that the mantissa and exponent are binary numbers and 2 is used instead of 10
- This is called the *floating point representation*
More on the scientific notation: Let’s come up with a representation

\[ \pm \text{number}. \text{number} \times 10^{\pm \text{number}} \]

- A value like 9,876,543,210 would be approximated with \( 9.88 \times 10^9 \) (or 9.88e+9 in programming language notation)

Scientific notation complicates arithmetic somewhat

- When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same.

- For example, when adding 1.23e1 and 4.56e0, you could convert 4.56e0 to 0.456e1 and then add them.
  - The result (1.23e1 + 0.456e1 = 1.686e1), does not fit into the three significant digits of our current format (in the previous slide).
  - So, we must either round or truncate the result to three significant digits.
    - Rounding generally produces the more accurate result, so let’s round to obtain 1.69e1.
The implications of such adjustments?

- The lack of **precision** (the number of digits or bits maintained in a computation) affects the **accuracy** (the correctness of the computation).

Floating point representation: **Binary version** of the scientific notation to represent a wide range of numbers

- The naming convention is confusing because the binary (or decimal) point is always in the same place:
  - Between the ones and halves (tenths in decimal)
  - The “float” is just another way of saying “scientific notation,” which allows us to write $1.2 \times 10^{-3}$ instead of 0.0012

- By **separating** the significant digits from the exponents:
  - The floating-point system allows us to represent very small or very large numbers without having to store all those zeros
IEEE 754 Floating standard [1/2]

- When Intel planned to introduce a floating-point unit (FPU) for its original 8086 microprocessor?
  - Intel knew their electrical engineers and solid-state physicists didn’t have the numerical analysis background to design a good floating-point representation
  - Went out and hired the best numerical analyst they could find to design a floating-point format for its 8087 FPU
- That person then hired two other experts in the field, and the three of them (Kahan, Coonen, and Stone) designed the KCS Floating-Point Standard
  - They did such a good job that the IEEE organization used this format as the basis for the IEEE Std 754 floating-point format

IEEE 754 Floating standard [2/2]

- The floating-point system is the standard way to represent real numbers in computing
- There are two signs:
  - One for the mantissa and a hidden one that is part of the exponent
- There are also a lot of tricks to make sure that things like rounding work as well as possible and to minimize the number of wasted bit combinations
The mantissa and the exponent

- The mantissa is a base value that usually falls within a **limited range** (for example, between 0 and 1).
- The exponent is a **multiplier** that, when applied to the mantissa, produces values outside this range.
- The big advantage of the mantissa/exponent configuration:
  - Floating-point format can represent values across a **wide range**.

IEEE Floating point number formats

- **Single Precision**:
  - 8-bits for the exponent
  - Exponent: 31-23
  - Mantissa: 22-0

- **Double Precision**:
  - 11-bits for the exponent
  - Exponent: 63-52
  - Mantissa (high): 51-32
  - Mantissa (low): 31-0
A couple of the tricks that are used in the standard

- **Normalization**, which adjusts the mantissa so that there are no leading (that is, on the left) zeros
- A second trick, from Digital Equipment Corporation (DEC), doubles the accuracy
  - By throwing away the leftmost bit of the mantissa since we know that it will always be 1, which makes room for one more bit
- Exponent values of all 0s and all 1s would have special meaning

Two types of floating point numbers: Single and double-precision

- **Single-precision** numbers use 32 bits and can represent numbers approximately in the range $\pm 10^{\pm 38}$ with about 7 digits of accuracy
  - Although there is an infinite number of values between 1 and 2, we can represent only 8 million ($2^{23}$) of them because we use a 23-bit mantissa
    - Therefore, have only 23 bits of precision
- **Double-precision** numbers use 64 bits and can represent a wider range of numbers, approximately $\pm 10^{\pm 308}$, with about 15 digits of accuracy
  - Number of atoms in the know universe is between $10^{78}$~$10^{82}$
IEEE 754 also uses some special bit patterns

- To represent things like division by zero, which evaluates to positive or negative infinity
- It also specifies a special value called NaN, which stands for “not a number”
  - If you find yourself in the NaNny state, it probably means that you did some illegal arithmetic operation

HEXADECIMAL
Hexadecimal

- Hexadecimal is **base 16**!
- Given what we’ve already seen so far, you probably know what that means
- Hexadecimal, or just **hex** for short, is a place-value system where
  - Each place represents a power of 16
  - and each place can be one of 16 symbols

Hexadecimal number representation

- As in all place-value systems, the rightmost place will still be the ones place
- The next place to the left will be the sixteens place, then the 256s \((16 \times 16)\) place, then the 4,096s \((16 \times 16 \times 16)\) place, and so on
- Simple enough!
But what about the other requirement that each place can be one of 16 symbols? [1/2]

- We usually have ten symbols to use to represent numbers, 0 through 9
- We need to add six more symbols to represent the other values

But what about the other requirement that each place can be one of 16 symbols? [2/2]

- We could pick some random symbols like & @ #, but these symbols have no obvious order
- Instead, the standard is to use A, B, C, D, E, and F
  - Either uppercase or lowercase is fine!
- In this scheme, A represents ten, B represents eleven, and so on, up to F, which represents fifteen
Consider the number 0x1A5 in hexadecimal

- What’s the value of this number in decimal?
- The rightmost place is worth 5
- The next place has a weight of 16, and there’s an A there, which is 10 in decimal, so the middle place is worth 16 × 10 = 160
- The leftmost place has a weight of 256, and there’s a 1 in that place, so that place is worth 256
- The total value then is 256 + 160 + 5 = 421 in decimal
Another view of the number $0x1A5$ in hexadecimal

\[
\begin{array}{ccc}
1 & A & 5 \\
256s place & Sixteens place & Ones place \\
16^2 & 16^1 & 16^0 \\
16 \times 16 = 256 & 16 & 1 \\
\end{array}
\]

\[
1 \times 256 + 10 \times 16 + 1 \times 5 \\
= 256 + 160 + 5 \\
= 421 \text{ in decimal}
\]

Conversion from decimal to hex

- The hex number is constructed from **right to left**
- Step 1: Divide the decimal number by 16 and note down the remainder
- Step 2: Divide the obtained quotient by 16, and note remainder again
- Step 3: Repeat the above steps until you get 0 as the quotient
  - Stopping criteria
Example: Decimal number 421

- \(421 \div 16 = 26\) (Remainder 5)
- \(26 \div 16 = 1\) (Remainder 10 or A)
- \(1 \div 16 = 0\) (Remainder 1)

Hex representation: \(0x1A5\)

\[= 1 \times 256 + 10 \times 16 + 1 \times 5\]
\[= 256 + 160 + 5\]
\[= 421 \text{ in decimal}\]

Top to Bottom is Right to Left \{LSB to MSB\}

Some conversions across number systems

<table>
<thead>
<tr>
<th></th>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>1111 0000 0000 1111</td>
<td>1000 1000 1000 0001</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>F00F</td>
<td>8881</td>
</tr>
<tr>
<td>Decimal</td>
<td>61,455</td>
<td>34,945</td>
</tr>
</tbody>
</table>
Boolean Algebra

- Boolean algebra manipulates two-state binary values that are typically labeled true/false, 1/0, yes/no, on/off, and so forth
- We will use 1 and 0.
Boolean Functions

- A **boolean function** is a function that operates on binary inputs and returns binary outputs.

- Since computer hardware is based on representing and manipulating binary values ...
  - Boolean functions play a central role in the specification, analysis, and optimization of hardware architectures.

Three commonly used Boolean functions, also known as Boolean operators

- **And**, **Or** and **NOT**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>And y</th>
<th>x</th>
<th>y</th>
<th>Or y</th>
<th>x</th>
<th>Not x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
What makes And, Or, and Not more interesting, or privileged, than any other subset of Boolean operators?

- The short answer is that indeed there is nothing special about And, Or, and Not.
- A deeper answer is that various subsets of logical operators can be used for expressing any Boolean function, and \{And, Or, Not\} is one such subset.

If you find this claim impressive, consider this: any one of these three basic operators can be expressed using yet another operator—**Nand**

- The name of the Nand operator is shorthand for Not-And, coming from the observation that Nand \((x, y)\) is equivalent to Not (And \((x, y)\))

- Now, that’s impressive!
- It follows that any boolean function can be realized using Nand gates only.
The contents of this slide-set are based on the following references


