Outline of Introduction

- Administrivia
- What is computer architecture?
- What do computers do?
- Representing high level things in binary
  - Data objects: integers, decimals, characters, etc.
  - Memory locations
  - (We’ll get to instruction representations a bit later in course)

Number Systems for Computers

- Today’s computers are digital systems
  - Each signal in digital system is either off or on
- Need to represent numbers using only off and on
  - Two symbols
  - off and on can represent the digits 0 and 1
  - BIT is Binary Digit
  - A bit can have a value of 0 or 1
- Binary representation
  - Weighted positional notation using base 2
    \[11_2 = 1*2^3 + 1*2^1 + 1*2^0 = 1011_2\]
    \[11_2 = 8 + 2 + 1\]
  - What is largest 4-bit number? (Hint: it’s my office number)

Binary, Octal and Hexadecimal numbers

- Computers can input/output decimal numbers, but internally they use binary representation
- Binary is good for computers, hard for us to read
  - We often like to use numbers that are easily computed from binary
- Binary numbers use only two digits: {0,1}
  - Example: 120010 = 00000100101100002
- Octal numbers (base 8) use 8 digits: {0 - 7}
  - Example: 120010 = ?8
  - Octal numbers often begin with 0 (e.g., 0317)
- Hexadecimal (base 16) uses 16 digits: {0-9,A-F}
  - Example: 120010 = ?16
  - Hex numbers should begin with 0x (e.g., 0x7ad24)
  - Hex notation doesn’t distinguish between upper/lower case

Issues for Binary Representation of Numbers

- There are many ways to represent numbers in binary
  - What are the issues that we must address?
- Issue #1: Complexity of arithmetic operations
- Issue #2: Negative numbers
- Issue #3: Maximum representable number
  - Choose representation that makes these issues easy for machine, even if it’s not easy for humans
    - Why? Machine has to do all the work!
Sign and Magnitude Representation for Integers

- **Unsigned Integers:**
  - $100101_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
  - 4 bits $\Rightarrow$ max number is 15 ($2^4 - 1$)
  - Can represent signed integers with sign and magnitude

- **Add a sign bit**
  - Example: $010110_2 = 22_{10}$; $110110_2 = -22_{10}$

- **Advantages:**
  - Simple extension of unsigned numbers
  - Same number of positive and negative numbers
  - Easy for humans (although we recall that this isn't so important)

- **Disadvantages:**
  - Two representations for 0: 0 = 000000 and -0 = 100000
  - Algorithm (circuit) for addition depends on the arguments’ signs

1's Complement Representation for Integers

- **Key:** use largest positive binary numbers to represent negative numbers
  - $(x) = 2^n - 1 - x$

- This is 1's complement + 1
  - $(x) = 2^n - 1 - x + 1$

- So, just invert bits and add 1
  - $010110_2 = 22_{10}$; $101010_2 = -22_{10}$

- 6-bit examples:
  - $010110_2 = 22_{10}$; $101010_2 = -22_{10}$
  - $1_{10} = 000001_2$; $-1_{10} = 111111_2$
  - $0_{10} = 000000_2$; $-0_{10} = 111111_2$; good!

- **Advantages:**
  - Only one representation for 0: 0 = 000000
  - Addition algorithm is much easier than with sign and magnitude
  - Independent of sign bits

- **Disadvantage:**
  - One more negative number than positive
  - Example: 6-bit 2’s complement number $100000_2 = -32_{10}$; but $32_{10}$ could not be represented

2's Complement Representation for Integers

- Still use large positives to represent negatives
  - $(x) = 2^n - x$

- This is 1’s complement + 1
  - $(x) = 2^n - 1 - x + 1$

- So, just invert bits and add 1
  - 6-bit examples:
    - $010110_2 = 22_{10}$; $101010_2 = -22_{10}$
    - $1_{10} = 000001_2$; $-1_{10} = 111111_2$
    - $0_{10} = 000000_2$; $-0_{10} = 111111_2$; good!

- **Advantages:**
  - Only one representation for 0: 0 = 000000
  - Addition algorithm is much easier than with sign and magnitude
  - Independent of sign bits

- **Disadvantage:**
  - One more negative number than positive
  - Example: 6-bit 2’s complement number $100000_2 = -32_{10}$; but $32_{10}$ could not be represented

All modern computers use 2's complement for integers
2’s Complement Negation and Addition

• To negate a number (positive or negative):
  • Step 1. complement the digits
  • Step 2. add 1

Example

\[
\begin{align*}
14_{10} &= 001110_2 \\
-14_{10} &= 110001_2 \\
+1 &= 110010_2
\end{align*}
\]

• To add signed numbers, use regular addition, but disregard carry out (Why? Hint: what does carry out equal?)

Example: \(18_{10} - 14_{10} = 18_{10} + (-14_{10}) = 4_{10}\)

2’s Complement (cont.)

• Example: \(A = 0x0C, B = 0x3B\)

• Compute: \(A + B\) and \(A - B\) in 8-bit 2’s complement arithmetic

• What’s the answer (in hex)?

2’s Complement Precision Extension

• Most computers today support 32-bit (int) or 64-bit integers
  • Specify 64-bit using gcc C compiler with long long
• To extend precision, use sign bit extension
  • Integer precision is number of bits used to represent a number

Examples

\[
\begin{align*}
14_{10} &= 001110, \text{ in 6-bit representation.} \\
14_{10} &= 000000001110, \text{ in 12-bit representation} \\
-14_{10} &= 110010, \text{ in 6-bit representation} \\
-14_{10} &= 111111110010, \text{ in 12-bit representation.}
\end{align*}
\]

What About Non-integer Numbers?

• There are infinitely many real numbers between two integers
  • Many important numbers are real
    • Speed of light \(\approx 3 \times 10^8\)
    • \(\pi = 3.1415...\)
  • Fixed number of bits limits range of integers
    • Can’t represent some important numbers
  • Humans use Scientific Notation
    • \(1.3 \times 10^4\)
  • We’ll learn how computers represent floating point numbers later in the semester
What about strings?

- Many important things stored as strings...
  - E.g., your name
- How should we store strings?

ASCII Character Representation

<table>
<thead>
<tr>
<th>Oct. Char</th>
<th>ASCII Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>nul</td>
</tr>
<tr>
<td>001</td>
<td>soh</td>
</tr>
<tr>
<td>002</td>
<td>stx</td>
</tr>
<tr>
<td>003</td>
<td>etx</td>
</tr>
<tr>
<td>004</td>
<td>eot</td>
</tr>
<tr>
<td>005</td>
<td>enq</td>
</tr>
<tr>
<td>006</td>
<td>ack</td>
</tr>
<tr>
<td>007</td>
<td>bel</td>
</tr>
<tr>
<td>010</td>
<td>bs</td>
</tr>
<tr>
<td>011</td>
<td>ht</td>
</tr>
<tr>
<td>012</td>
<td>nl</td>
</tr>
<tr>
<td>013</td>
<td>vt</td>
</tr>
<tr>
<td>014</td>
<td>np</td>
</tr>
<tr>
<td>015</td>
<td>cr</td>
</tr>
<tr>
<td>016</td>
<td>so</td>
</tr>
<tr>
<td>017</td>
<td>si</td>
</tr>
<tr>
<td>018</td>
<td>dle</td>
</tr>
<tr>
<td>019</td>
<td>dc1</td>
</tr>
<tr>
<td>020</td>
<td>dc2</td>
</tr>
<tr>
<td>021</td>
<td>dc3</td>
</tr>
<tr>
<td>022</td>
<td>dc4</td>
</tr>
<tr>
<td>023</td>
<td>nak</td>
</tr>
<tr>
<td>024</td>
<td>syn</td>
</tr>
<tr>
<td>025</td>
<td>etb</td>
</tr>
<tr>
<td>026</td>
<td>can</td>
</tr>
<tr>
<td>027</td>
<td>em</td>
</tr>
<tr>
<td>028</td>
<td>sub</td>
</tr>
<tr>
<td>029</td>
<td>esc</td>
</tr>
<tr>
<td>030</td>
<td>fs</td>
</tr>
<tr>
<td>031</td>
<td>gs</td>
</tr>
<tr>
<td>032</td>
<td>rs</td>
</tr>
<tr>
<td>033</td>
<td>us</td>
</tr>
<tr>
<td>034</td>
<td>sp</td>
</tr>
<tr>
<td>035</td>
<td>!</td>
</tr>
<tr>
<td>036</td>
<td>&quot;</td>
</tr>
<tr>
<td>037</td>
<td>#</td>
</tr>
<tr>
<td>038</td>
<td>$</td>
</tr>
<tr>
<td>039</td>
<td>%</td>
</tr>
<tr>
<td>040</td>
<td>&amp;</td>
</tr>
<tr>
<td>041</td>
<td>'</td>
</tr>
<tr>
<td>042</td>
<td>(</td>
</tr>
<tr>
<td>043</td>
<td>)</td>
</tr>
<tr>
<td>044</td>
<td>*</td>
</tr>
<tr>
<td>045</td>
<td>+</td>
</tr>
<tr>
<td>046</td>
<td>,</td>
</tr>
<tr>
<td>047</td>
<td>.</td>
</tr>
<tr>
<td>048</td>
<td>/</td>
</tr>
<tr>
<td>049</td>
<td>0</td>
</tr>
<tr>
<td>050</td>
<td>1</td>
</tr>
<tr>
<td>051</td>
<td>2</td>
</tr>
<tr>
<td>052</td>
<td>3</td>
</tr>
<tr>
<td>053</td>
<td>4</td>
</tr>
<tr>
<td>054</td>
<td>5</td>
</tr>
<tr>
<td>055</td>
<td>6</td>
</tr>
<tr>
<td>056</td>
<td>7</td>
</tr>
<tr>
<td>057</td>
<td>8</td>
</tr>
<tr>
<td>058</td>
<td>9</td>
</tr>
<tr>
<td>059</td>
<td>:</td>
</tr>
<tr>
<td>060</td>
<td>;</td>
</tr>
<tr>
<td>061</td>
<td>&lt;</td>
</tr>
<tr>
<td>062</td>
<td>=</td>
</tr>
<tr>
<td>063</td>
<td>&gt;</td>
</tr>
<tr>
<td>064</td>
<td>?</td>
</tr>
<tr>
<td>065</td>
<td>@</td>
</tr>
<tr>
<td>066</td>
<td>A</td>
</tr>
<tr>
<td>067</td>
<td>B</td>
</tr>
<tr>
<td>068</td>
<td>C</td>
</tr>
<tr>
<td>069</td>
<td>D</td>
</tr>
<tr>
<td>070</td>
<td>E</td>
</tr>
<tr>
<td>071</td>
<td>F</td>
</tr>
<tr>
<td>072</td>
<td>G</td>
</tr>
<tr>
<td>073</td>
<td>H</td>
</tr>
<tr>
<td>074</td>
<td>I</td>
</tr>
<tr>
<td>075</td>
<td>J</td>
</tr>
<tr>
<td>076</td>
<td>K</td>
</tr>
<tr>
<td>077</td>
<td>L</td>
</tr>
<tr>
<td>078</td>
<td>M</td>
</tr>
<tr>
<td>079</td>
<td>N</td>
</tr>
<tr>
<td>080</td>
<td>O</td>
</tr>
<tr>
<td>081</td>
<td>P</td>
</tr>
<tr>
<td>082</td>
<td>Q</td>
</tr>
<tr>
<td>083</td>
<td>R</td>
</tr>
<tr>
<td>084</td>
<td>S</td>
</tr>
<tr>
<td>085</td>
<td>T</td>
</tr>
<tr>
<td>086</td>
<td>U</td>
</tr>
<tr>
<td>087</td>
<td>V</td>
</tr>
<tr>
<td>088</td>
<td>W</td>
</tr>
<tr>
<td>089</td>
<td>X</td>
</tr>
<tr>
<td>090</td>
<td>Y</td>
</tr>
<tr>
<td>091</td>
<td>Z</td>
</tr>
<tr>
<td>092</td>
<td>[</td>
</tr>
<tr>
<td>093</td>
<td>]</td>
</tr>
<tr>
<td>094</td>
<td>^</td>
</tr>
<tr>
<td>095</td>
<td>_</td>
</tr>
<tr>
<td>096</td>
<td>`</td>
</tr>
<tr>
<td>097</td>
<td>a</td>
</tr>
<tr>
<td>098</td>
<td>b</td>
</tr>
<tr>
<td>099</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>126</td>
<td>~</td>
</tr>
</tbody>
</table>

- Each character represented by 7-bit ASCII code.
- Packed into 8-bits

Basic Data Types

- **Bit**: 0, 1
- **Bit String**: sequence of bits of a particular length
  - 8 bits is a byte
  - 16 bits is a half-word
  - 32 bits is a word
  - 64 bits is a double-word
- **Character**: ASCII 7 bit code
- **Decimal**: (binary coded decimal or BCD)
  - digits 0-9 encoded as 0000-1001 2 decimal digits packed per byte
- **Integers**: 2’s complement (32-bit representation)
- **Floating Point**: Single Precision (32-bit representation)
  - Double Precision (64-bit representation)
  - Extended Precision (128-bit representation)

Summary of Data Representations

- Computers operate on binary numbers (0s and 1s)
- Conversion to/from binary, octal, hex
- Signed binary numbers
  - 2’s complement
  - arithmetic, negation
- Real number representation → later in course
- Characters and strings
Computer Memory

- What is computer memory?
- What does it "look like" to the program?
- How do we find things in computer memory?

A Program’s View of Memory

- What is memory? a bunch of bits
- Looks like a large linear array
- Find things by indexing into array
  - Index is unsigned integer
- Most computers support byte (8-bit) addressing
  - Each byte has a unique address (location).
  - Byte of data at address 0x100 and 0x101
  - Word of data at address 0x100 and 0x104
- 32-bit v.s. 64-bit addresses
  - We will assume 32-bit for rest of course, unless otherwise stated
  - How many bytes can we address with 32 bits? With 64 bits?

Memory Partitions

- Text for instructions
  - add res, src1, src2
  - mem[res] = mem[src1] + mem[src2]
- Data
  - Static (constants, globals)
  - Dynamic (heap, new allocated)
  - Grows upward
- Stack
  - Local variables
  - Grows down from top of memory
  - Variables are names for memory locations
    - int x; // x is a location in memory

A Simple Program’s Memory Layout

```
... int result; // global variable
main()
{
    int x; // allocated on stack
    ... result = x + result;
    ...
}
```
Pointers

- A pointer is a memory location that contains the address of another memory location
- "address of" operator & in C/C++
  - Don’t confuse with bitwise AND operator (which is && operator)

Given

```c
int x; int* p;
p = &x;  // p points to x (i.e., p is the address of x)
Then
*p = 2; and x = 2; produce the same result
```

On 32-bit machine, p is 32-bits

| x | 0x26cf0 |
| p | 0x26d00 |

Arrays

- In C++: allocate using array form of `new`
  ```c
  int* a = new int[100];
  double* b = new double[300];
  ```
- `new[]` returns a pointer to a block of memory
  - How big? Where?
- Size of chunk can be set at runtime
- `delete[]` a; // storage returned
- In C:
  ```c
  int* ptr = malloc(nbytes);
  free(ptr);
  ```

Address Calculation

- x is a pointer, what is x+33?
  - A pointer, but where?
    - What does calculation depend on?
  - Result of adding an int to a pointer depends on size of object pointed to
  - Result of subtracting two pointers is an int
  - \( (d + 3) - d = \) 32

<table>
<thead>
<tr>
<th>int * a = new int[100]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>double * d = new double[200];</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20</td>
</tr>
</tbody>
</table>

| *d[33] is the same as *(a+33) |
|*(a+33) is the same as *(a+33) |

More Pointer Arithmetic

- address one past the end of an array is ok for pointer comparison only

- what’s at *(begin+44) ?

- what does `begin++` mean?

- how are pointers compared using < and using == ?

- what is value of `end - begin` ?
More Pointers & Arrays

```c
int* a = new int[100];
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
```

Array Example

```c
#include <iostream.h>
main()
{
    int* a = new int[100];
    int* p = a;
    int k;
    for (k = 0; k < 100; k++)
    {
        *p = k;
        p++;
    }
    cout << "entry 3 = " << a[3] << endl;
}
```

Strings as Arrays

- A string is an array of characters with '\0' at the end
- Each element is one byte, ASCII code
- '\0' is null (ASCII code 0)

Bit Manipulations

**Problem**
- 32-bit word contains many values
  - e.g., input device, sensors, etc.
  - Current x,y position of mouse and which button (left, mid, right)
- Assume x, y position is 0-255
- How many bits for position?
- How many for button?

**Goal**
- Extract position and button from 32-bit word
- Need operations on individual bits of binary number
Bitwise AND / OR

• & operator performs bitwise AND
• | operator performs bitwise OR
• Per bit (remember your truth tables from ECE 151)
  0 & 0 = 0  0 | 0 = 0
  0 & 1 = 0  0 | 1 = 1
  1 & 0 = 0  1 | 0 = 1
  1 & 1 = 1  1 | 1 = 1
• For multiple bits, apply operation to individual bits in same position

Mouse Example

• 32-bit word with x, y, and button fields
  • Bits 0-7 contain x position
  • Bits 8-15 contain y position
  • Bits 16-17 contain button (0 = left, 1 = middle, 2 = right)
• To extract value, we need to clear all other bits
• How do we use bitwise operations to do this?

Mouse Solution

• AND with a bit mask
  • Specific values that clear some bits, but pass others through
  • To extract x position, use mask 0x000ff
    xpos = 0x1a34c & 0x000ff

  button  y  x
  0x1a34c = 01 1010 0011 0100 1100
  0x000ff = 00 0000 0000 1111 1111
  0x0004c = 00 0000 0000 0100 1100

More of the Mouse Solution

• To extract y position use mask 0x00ff00
  ypos = 0x1a34c & 0x00ff00

  button  y  x
  0x1a34c = 01 1010 0011 0100 1100
  0x000ff = 00 0000 0000 1111 1111
  0x0a300 = 00 1010 0011 0000 0000

• Similarly, button is extracted with mask 0x30000
  button = 0x1a34c & 0x30000

  button  y  x
  0x1a34c = 01 1010 0011 0100 1100
  0x000ff = 00 1111 1111 0000 0000
  0x0a300 = 00 1010 0011 0000 0000
The SHIFT operator

- >> is shift right, << is shift left, operands are int and number of positions to shift
- \((1 << 3)\) is \(\ldots000001 \to \ldots0001000\) (it's \(2^3\))
- \(0xff00\) is \(0xff << 8\), and \(0xff\) is \(0xff00 >> 8\)
- So, true ypos value is
  
  \[
  \text{ypos} = (0x1a34c \& 0x0ff00) >> 8
  \]
  
  \[
  \text{button} = (0x1a34c \& 0x30000) >> 16
  \]

Summary: Representing High Level in Computer

- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- Bitwise operations
- We'll visit these topics again throughout semester

Outline of Introduction

- Administrivia
- What is computer architecture?
- What do computers do?
- Representing high level things in binary

What You Will Learn In This Course

- The basic operation of a computer
  - Primitive operations (instructions)
  - Computer arithmetic
  - Instruction sequencing and processing
  - Memory
  - Input/output
  - Doing all of the above, just faster!
- Understand the relationship between abstractions
  - Interface design
  - High-level program to control signals (SW \(\rightarrow\) HW)
Course Outline

- Introduction to Computer Architecture
- Instruction Sets & Assembly Programming (next!)
- Central Processing Unit (CPU)
- Memory Hierarchy
- I/O Devices and Networks
- Pipelined and Parallel Processors
- Computer Performance Analysis (time permitting)