General Information

- Instructor: Alvin R. Lebeck (alvy)
  Office: D304 LSRC
  email: alvy@cs.duke.edu
  Office Hours: Tues 2:00 - 3:00, Fri 10:00 - 11:00, or by appointment
- Course Web Page
  http://www.cs.duke.edu/courses/spring03/cps104
  Lecture slides available on web page
- Course News Group
  duke.cs.cps104
  You are required to monitor web page and newsgroup
  - Home work will appear on web page
  - If necessary, additional information about homework on newsgroup
  - You can post questions about homework to newsgroup

Administrivia

No Recitation this semester
- Fold material into class (examples, etc.)
- Kevin more office hours

Homework
- Homework #1 Due Jan 21
- Two parts
  - written due in class,
  - program submit by midnight

Reading
- Ch. 1, skim Ch. 2
- Ch 4.1-4.3, 4.8 pages 275-280
- Start Ch. 3

Overview

- First step in mapping high-level to machine
  - Data representations

Outline

- Review
  - Binary Numbers
  - Integer numbers
  - Floating-point numbers
  - Characters
  - Storage sizes: Bit, Byte, Word, Double-word
  - Memory
  - Arrays
  - Pointers

Review

Goal
- Understand basic operation of a computer

Why?
- Software performance is affected/determined by HW capabilities
- Future Computer Architects (Processor or System)

Review: The Big Picture

- The Five Classic Components of a Computer
High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Control Signal Specification

Levels of Abstraction

<table>
<thead>
<tr>
<th>High Level Language Program</th>
<th>temp = v[k];</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>v[k] = v[k+1];</td>
</tr>
<tr>
<td>lw $15, 0($2)</td>
<td>v[k+1] = temp;</td>
</tr>
<tr>
<td>lw $16, 4($2)</td>
<td></td>
</tr>
<tr>
<td>sw $16, 0($2)</td>
<td></td>
</tr>
<tr>
<td>sw $15, 4($2)</td>
<td></td>
</tr>
</tbody>
</table>

Number Systems for Computers

- Today’s computers are built from transistors
- Transistor is either off or on
- Need to represent numbers using only off and on
  - 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
      \[1110 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 1011_2\]
      \[1110 = 8 + 4 + 2 + 1\]
  - What is largest number, given 4 bits?

Binary Integers

- Unsigned Integers:
  - \[i = 100101; i = 1010112 = 2210\]
  - 4 bits \(\Rightarrow\) max number is 15
- Sign Magnitude
  - Add a sign bit
    - Example: \(0101102 = 2210\); \(1101102 = -2210\)
  - Advantages:
    - Simple extension of unsigned numbers.
    - Same number of positive and negative numbers.
  - Disadvantages:
    - Two representations for 0: 0=000000; -0=100000.
    - Algorithm (circuit) for addition depends on the arguments’ signs.

Binary, Octal and Hexadecimal numbers

- Computers can input and output decimal numbers but they convert them to internal binary representation.
- Binary is good for computers, hard for us to read
  - Use numbers easily computed from binary
- Binary numbers use only two different digits: \(0, 1\)
  - Example: \(1200_2 = 0000010010110000\)
- Octal numbers use 8 digits: \(0 - 7\)
  - Example: \(1200_8 = 04260\)
- Hexadecimal numbers use 16 digits: \(0-9, A-F\)
  - Example: \(1200_{16} = 04B8_{16}\)
  - does not distinguish between upper and lower case

1's Complement Representation for Integers

- Key is to use largest positive binary numbers to represent negative numbers
  - \(i = 2^n - 1 - x\)
- Simply invert each bit \(0->1, 1->0\)
- Two zeros
  - \(0111 = 7\)
- 8-bit examples:
  - \(0000 = 0\)
  - \(0001 = 1\)
  - \(0010 = 2\)
  - \(0011 = 3\)
  - \(0100 = 4\)
  - \(0101 = 5\)
  - \(0110 = 6\)
  - \(0111 = 7\)
  - \(1000 = 8\)
  - \(1001 = 9\)
  - \(1010 = 10\)
  - \(1011 = 11\)
  - \(1100 = 12\)
  - \(1101 = 13\)
  - \(1110 = 14\)
  - \(1111 = 15\)

- Example: \(010100 = 16\)
- \(00000000 = 0\)
- \(11111111 = -1\)
- \(01010101 = 253\)
- \(10101010 = -253\)
- \(00000001 = 1\)
- \(11111110 = -1\)
2's Complement Representation for Integers

- Still use large positives to represent negatives
- $i = 2^n - x$
- This is 1's complement + 1
- $i = 2^n - 1 - x + 1$
- So, invert bits and add 1

6-bit examples:

$010110_2 = 22_{10} \; \; 101010_2 = -22_{10}$

$010 = 000000_2; 110 = 000001_2; \; -110 = 111111_2$

2's Complement

- Advantages:
  - Only one representation for 0: $0 = 000000$
  - Addition algorithm 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

To negate a number do:

- Step 1. complement the digits
- Step 2. add 1

Example

$14_{10} = 001110_2$
$-14_{10} = 110001_2$

$+1 \quad \quad 110010_2$

To add signed numbers use regular addition but disregard carry out:

Example

$18_{10} - 14_{10} = 18_{10} + (-14_{10}) = 4_{10}$

$010010_2$
$+110010_2$
$000100_2$

2's Complement (cont.)

- Example: $A = 0x0ABC; \; B = 0xFEB$

- Compute: $A + B$ and $A - B$ in 16-bit 2's complement arithmetic.

- Give answer in HEX

2's Complement Precision Extension

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

  Example

  $14_{10} = 001110_2$ in 6-bit representation.
  $14_{10} = 000000001110_2$ in 12-bit representation

  $-14_{10} = 110010_2$ in 6-bit representation
  $-14_{10} = 111111110010_2$ in 12-bit representation.

Answer

- $A + B = 0x1AA7$
- $A - B = 0xFAD1$
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers.
- Many important numbers are real:
  - speed of light ≈ 3x10^8
  - 3.145…
- Fixed number of bits limits range of integers:
  - Can't represent some important numbers.
- Humans use Scientific Notation:
  - 1.3x10^4

Floating Point Representation

Numbers are represented by:

\[ X = (-1)^s \times 1.M \times 2^{E} \]

- **S**: 1-bit field; Sign bit
- **E**: 8-bit field; Exponent; Biased integer, 0 ≤ E ≤ 255.
- **M**: 23-bit field; Mantissa; Normalized fraction with hidden 1 (don't actually store it)

Single precision floating point number uses 32-bits for representation

Example:

\[ X = -0.75_{10} \text{ in single precision } \]

\[ \begin{align*}
S &= 1 ; \text{Exp} = 126_{10} = 0111 1110_{2} ; \\
M &= 100 0000 0000 0000 0000 0000_{2}
\end{align*} \]

Answer

What floating-point number is \( 0xC1580000 \)?

\[ \begin{align*}
X &= \frac{1100 0001 0101 1000 0000 0000 0000 0000}{101 1000 0000 0000 0000 0000_{2}} \\
&= 2^{128} + 2^{-127} = 3 \quad M = 1011 \\
&= 1.1011 x 2^3 = -1101.1 = -13.5
\end{align*} \]

Floating Point Representation

- The mantissa represents a fraction using binary notation:
  \[ M = .s_1 s_2 s_3 \ldots = 1.0 + s_1 \times 2^{-1} + s_2 \times 2^{-2} + s_3 \times 2^{-3} + \ldots \]
- Example: \( X = -0.75_{10} \) in single precision \((-11/2 + 1/4))\):
  \[ \begin{align*}
S &= 1 ; \text{Exp} = 126_{10} = 0111 1110_{2} ; \\
M &= 100 0000 0000 0000 0000 0000_{2}
\end{align*} \]

Floating Point Representation

- Double Precision Floating point:
  - 64-bit representation: 1-bit sign, 11-bit (biased) exponent; 52-bit mantissa (with hidden 1).
  \[ X = (-1)^s \times 2^{E-1023} \times 1.M \]

Double precision floating point number

\[ \begin{align*}
S &= 1111 1110_{2} ; \\
E &= 100 0000 0000 0000 0000 0000_{2} \\
M &= 100 0000 0000 0000 0000 0000_{2}
\end{align*} \]
What about strings?

- Many important things stored as strings...
- Your name
- How should we store strings?

ASCII Character Representation

Oct. Chr.

<table>
<thead>
<tr>
<th>Character</th>
<th>Octal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>nul</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>soh</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>stx</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>etx</td>
<td>3</td>
</tr>
<tr>
<td>020</td>
<td>eot</td>
<td>4</td>
</tr>
<tr>
<td>021</td>
<td>enq</td>
<td>5</td>
</tr>
<tr>
<td>030</td>
<td>ack</td>
<td>6</td>
</tr>
<tr>
<td>031</td>
<td>bel</td>
<td>7</td>
</tr>
<tr>
<td>040</td>
<td>bs</td>
<td>8</td>
</tr>
<tr>
<td>041</td>
<td>ht</td>
<td>9</td>
</tr>
<tr>
<td>050</td>
<td>nl</td>
<td>10</td>
</tr>
<tr>
<td>051</td>
<td>vt</td>
<td>11</td>
</tr>
<tr>
<td>060</td>
<td>np</td>
<td>12</td>
</tr>
<tr>
<td>061</td>
<td>cr</td>
<td>13</td>
</tr>
<tr>
<td>070</td>
<td>so</td>
<td>14</td>
</tr>
<tr>
<td>071</td>
<td>si</td>
<td>15</td>
</tr>
<tr>
<td>080</td>
<td>dle</td>
<td>16</td>
</tr>
<tr>
<td>081</td>
<td>dc1</td>
<td>17</td>
</tr>
<tr>
<td>090</td>
<td>dc2</td>
<td>18</td>
</tr>
<tr>
<td>091</td>
<td>dc3</td>
<td>19</td>
</tr>
<tr>
<td>100</td>
<td>dc4</td>
<td>20</td>
</tr>
<tr>
<td>101</td>
<td>nak</td>
<td>21</td>
</tr>
<tr>
<td>110</td>
<td>syn</td>
<td>22</td>
</tr>
<tr>
<td>111</td>
<td>etb</td>
<td>23</td>
</tr>
<tr>
<td>120</td>
<td>can</td>
<td>24</td>
</tr>
<tr>
<td>121</td>
<td>em</td>
<td>25</td>
</tr>
<tr>
<td>122</td>
<td>sub</td>
<td>26</td>
</tr>
<tr>
<td>123</td>
<td>esc</td>
<td>27</td>
</tr>
<tr>
<td>124</td>
<td>fs</td>
<td>28</td>
</tr>
<tr>
<td>125</td>
<td>gs</td>
<td>29</td>
</tr>
<tr>
<td>126</td>
<td>rs</td>
<td>30</td>
</tr>
<tr>
<td>127</td>
<td>us</td>
<td>31</td>
</tr>
</tbody>
</table>

- Each character is represented by a 7-bit ASCII code.
- It is packed into 8-bits

Summary of Data Representations

- Computers operate on binary numbers (0s and 1s)
- Conversion to/from binary, oct, hex
- Signed binary numbers
  - 2's complement
  - arithmetic, negation
- Floating point representation
  - hidden 1
  - biased exponent
  - single precision, double precision

Basic Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>0, 1</td>
</tr>
<tr>
<td>Bit String</td>
<td>sequence of bits of a particular length</td>
</tr>
<tr>
<td>8 bits</td>
<td>a byte</td>
</tr>
<tr>
<td>16 bits</td>
<td>a half-word</td>
</tr>
<tr>
<td>32 bits</td>
<td>a word</td>
</tr>
<tr>
<td>64 bits</td>
<td>a double-word</td>
</tr>
<tr>
<td>Character</td>
<td>7 bit code</td>
</tr>
<tr>
<td>DecimaL</td>
<td>BCD code</td>
</tr>
<tr>
<td>8 bits</td>
<td>two decimal digits packed per 8 bit byte</td>
</tr>
<tr>
<td>Integer</td>
<td>2's Complement (32-bit representation).</td>
</tr>
<tr>
<td>Floating</td>
<td>Single Precision (32-bit representation).</td>
</tr>
<tr>
<td>Point</td>
<td>Double Precision (64-bit representation).</td>
</tr>
<tr>
<td></td>
<td>Extended Precision (128-bit representation).</td>
</tr>
</tbody>
</table>

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
- unsigned integer
- Most computers support byte (8-bit) addressing
  - Each byte has a unique address (location).
  - Byte of data at address 0x100
  - Word of data at address 0x100 and 0x101
- 32-bit v.s. 64-bit addresses
  - we will assume 32-bit for rest of course, unless otherwise stated
Buzz Word Definition: Endianess

Byte Order
- **Big Endian**: byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian**: byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC Alpha

![Diagram of byte order and endianess](image)

---

Buzz Word Definition: Alignment

- **Alignment**: require that objects fall on address that is multiple of their size.
- 32-bit integer
  - Aligned if address % 4 = 0
- 64-bit integer?
  - Aligned if ?

![Diagram of alignment](image)

---

A Simple Program’s Memory Layout

```c
... int result;
main()
{
   int x;
   ... result = x + result;
   ...
}
mem[0x208] = mem[0x400] + mem[0x208]
```

---

Pointers

- A pointer is a memory location that contains the address of another memory location
- “address of” operator &
  - don’t confuse with bitwise AND operator (later today)

Given
```c
int x; int *p;
p = &x;
```
Then
```c
*p = 2; and x = 2; produce the same result
```

On 32-bit machine, p is 32-bits

```c
x 0x26cf0
p 0x26d00
```

---

Memory Partitions

- **Text for instructions**
  - add res, src1, src2
  - mem[ res ] = mem[ src1 ] + mem[ src2 ]
- **Data**
  - static (constants, globals)
  - dynamic (heap, new allocated)
  - grows up
- **Stack**
  - local variables
  - grows down
- Variables are names for memory locations
  - int x;

![Diagram of memory partitions](image)

---

Vector Class vs. Arrays

- **Vector Class**
  - insulates programmers
  - array bounds checking
  - automatically growing/shrinking when more items are added/deleted
- **How are Vectors implemented?**
  - real understanding comes when more levels of abstraction are understood
- **Programming close to HW**
  - e.g., operating system, device drivers, etc.
- **Arrays can be more efficient**
  - but be leery of claims that C-style arrays required for efficiency
- **Can talk about memory easier in terms of arrays**
  - pointer to a vector?

![Diagram of vector class vs. arrays](image)
Arrays

- In C++ allocate using array form of `new`
  ```
  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:
  ```
  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 =________
  ```

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

```
int * a = new int[100];
```