Under the Hood: Data Representation
Memory

CPS 104
Lecture 2

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
**Levels of Abstraction**

Temporary variable declaration:

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Instruction set example:

```
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

Transistors turning on and off:

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

---

**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**
  - 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_{10} = 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 1011_{2}
11_{10} = 8 + 2 + 1
```

What is largest number, given 4 bits?
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_{10} = 0000010010110000_2
• Octal numbers use 8 digits: {0 - 7}
  ➢ Example: 1200_{10} = 04260_8
• Hexadecimal numbers use 16 digits: {0-9, A-F}
  ➢ Example: 1200_{10} = 04B0_{16} = 0x04B0
  ➢ does not distinguish between upper and lower case

Issues for Binary Representation

• Complexity of arithmetic operations
• Negative numbers
• Maximum representable number
• Choose representation that’s easy for machine not easy for humans
Binary Integers

- **Unsigned Integers:**
  - $i = 100101_2$; $i = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
  - 4 bits => max number is 15

- **Sign 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.
  - **Disadvantages:**
    - Two representations for 0: 0=000000; -0=100000.
    - Algorithm (circuit) for addition depends on the arguments' signs.

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**
  - 6-bit examples:
    - $010110 = 22_{10}$; $101001 = -22_{10}$
    - $0_{10} = 000000$; $0 = 111111$
    - $1_{10} = 000001$; $-1_{10} = 111110$
    - $1000_2 = -7$
    - $1001_2 = -6$
    - $1010_2 = -5$
    - $1011_2 = -4$
    - $1100_2 = -3$
    - $1101_2 = -2$
    - $1110_2 = -1$
    - $1111_2 = 0$
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:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0</td>
</tr>
<tr>
<td>000001</td>
<td>1</td>
</tr>
<tr>
<td>00010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
</tr>
</tbody>
</table>

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
2’s Complement Negation and Addition

• To negate a number do:
  ➢ Step 1. complement the digits
  ➢ Step 2. add 1

Example

\[ 14_{10} = 001110_2 \]
\[ -14_{10} = 110001_2 \]
\[ +1 \]
\[ 110010_2 \]

• 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: \[ A = 0x0ABC; \quad B = 0x0FEB. \]

• Compute: \[ A + B \] and \[ A - B \] in 16-bit 2’s complement arithmetic.

• Give answer in HEX
Answer

- A + B = 0x1AA7
- A – B = 0xFAD1

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

\[
14_{10} = 001110_2 \text{ in 6-bit representation.}
\]

\[
14_{10} = 000000001110_2 \text{ in 12-bit representation}
\]

-14_{10} = 110010_2 \text{ in 6-bit representation}
-14_{10} = 111111110010_2 \text{ in 12-bit representation.}
What About Non-integer Numbers?

- There are infinitely many real numbers between two integers
- Many important numbers are real
  - speed of light \( \approx 3 \times 10^8 \)
  - \( 3.145 \ldots \)
- Fixed number of bits limits range of integers
  - Can’t represent some important numbers
- Humans use Scientific Notation
  - \( 1.3 \times 10^4 \)

Floating Point Representation

Numbers are represented by:

\[ X = (-1)^s \times 1.M \times 2^{E-127} \]

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

Single precision floating point number uses 32-bits for representation:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>8-bit</td>
<td>23-bit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exp</td>
<td>Mantissa</td>
<td></td>
</tr>
</tbody>
</table>
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: \( X = \text{-0.75}_{10} \) in single precision (\(-\text{(1/2 + 1/4)}\))

\[ -0.75_{10} = -0.11_2 = (-1) \times 1.1_2 \times 2^{-1} = (-1) \times 1.1_2 \times 2^{126-127} \]

\[ S = 1; \quad \text{Exp} = 126_{10} = 0111 1110_2; \]

\[ M = 100 0000 0000 0000 0000 0000_2 \]

\[ \begin{array}{cccccc}
31 & 30 & 23 & 22 & 0 \\
\hline
1 & 0111 & 1110 & 100 0000 0000 0000 0000 0000 \\
\end{array} \]

Example:

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

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

\[
\begin{array}{cccccccc}
1 & 1000 & 0010 & 101 & 1000 & 0000 & 0000 & 0000 \\
\hline
s & E & M
\end{array}
\]

\[
X = (-1)^s \times 2^{E-127} \times 1.M
\]

- \( E = 128 + 2 - 127 = 3 \)
- \( M = 1011 \)
- \( -1.1011 \times 2^3 = -1101.1 = -13.5 \)

Floating Point Representation

- Double Precision Floating point:
  - 64-bit representation: 1-bit sign, 11-bit (biased) exponent; 52-bit mantissa (with hidden 1).

\[
X = (-1)^s \times 2^{E-1023} \times 1. M
\]

<p>| Double precision floating point number |</p>
<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bit</td>
<td>20-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit</td>
</tr>
</tbody>
</table>
What about strings?

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

ASCII Character Representation

<table>
<thead>
<tr>
<th>Oct. Chr.</th>
<th>nul</th>
<th>soh</th>
<th>stx</th>
<th>etx</th>
<th>eot</th>
<th>enq</th>
<th>ack</th>
<th>bel</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td>015</td>
<td>016</td>
<td>017</td>
<td>018</td>
</tr>
<tr>
<td>010</td>
<td>bs</td>
<td>011</td>
<td>012</td>
<td>013</td>
<td>014</td>
<td>015</td>
<td>016</td>
<td>017</td>
</tr>
<tr>
<td>020</td>
<td>dle</td>
<td>021</td>
<td>022</td>
<td>023</td>
<td>024</td>
<td>025</td>
<td>026</td>
<td>027</td>
</tr>
<tr>
<td>030</td>
<td>can</td>
<td>031</td>
<td>032</td>
<td>033</td>
<td>034</td>
<td>035</td>
<td>036</td>
<td>037</td>
</tr>
<tr>
<td>040</td>
<td>sp</td>
<td>041</td>
<td>042</td>
<td>043</td>
<td>044</td>
<td>045</td>
<td>046</td>
<td>047</td>
</tr>
<tr>
<td>050</td>
<td>(</td>
<td>051</td>
<td>052</td>
<td>053</td>
<td>054</td>
<td>055</td>
<td>056</td>
<td>057</td>
</tr>
<tr>
<td>060</td>
<td>0</td>
<td>061</td>
<td>062</td>
<td>063</td>
<td>064</td>
<td>065</td>
<td>066</td>
<td>067</td>
</tr>
<tr>
<td>070</td>
<td>8</td>
<td>071</td>
<td>072</td>
<td>073</td>
<td>074</td>
<td>075</td>
<td>076</td>
<td>077</td>
</tr>
<tr>
<td>100</td>
<td>H</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>110</td>
<td>I</td>
<td>111</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
</tr>
<tr>
<td>120</td>
<td>P</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
</tr>
<tr>
<td>130</td>
<td>X</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
</tr>
<tr>
<td>140</td>
<td>a</td>
<td>141</td>
<td>142</td>
<td>143</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
</tr>
<tr>
<td>150</td>
<td>b</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
</tr>
<tr>
<td>160</td>
<td>c</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
</tr>
<tr>
<td>170</td>
<td>d</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
<td>176</td>
<td>177</td>
</tr>
</tbody>
</table>

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

**Bit:** 0, 1

**Bit String:** sequence of bits of a particular length
- 4 bits is a nibble
- 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: (BCD code)
  - digits 0-9 encoded as 0000 thru 1001
two decimal digits packed per 8 bit 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).

- How many +/- #'s?
- Where is decimal pt?
- How are +/- exponents represented?

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
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 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
**Buzz Word Definition: Endianess**

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

**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 ?
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;`

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
  int x; int *p;
  p = &x;

Then
  *p = 2; and x = 2; produce the same result

On 32-bit machine, p is 32-bits

Vector Class vs. Arrays

• Vector Class
  ➢ insulates programmers
  ➢ array bounds checking
  ➢ automagically 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?
### Arrays

- In C++ allocate using array form of `new`
  ```cpp
  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
  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
  ```plaintext
  (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?

```c
char * a = new char[44];
char * begin = a;
char * end = a + 44;

while (begin < end)
{
    *begin = 'z';
    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
```
Next time

Finish Memory
• Pointers
• Arrays
• Strings

Bitwise operations

• Instruction Set Architecture

Reading
• Start Chapter 3