Under the Hood: Data Representation

Computer Science 104
Lecture 2

Homework
- Homework #1 Due Jan 24
- Two parts
  - written due in class,
  - program submit by midnight (blackboard drop box)

Reading
- Ch. 1
- Ch. 3.1-3.2 (pgs 160-168, 3.6 pgs 189-197)
- Start Ch. 2

Levels of Abstraction

High Level Language Program
- temp = v[k];
- v[k] = v[k+1];
- v[k+1] = temp;

Compiler
- lw $15, 0($2)
- lw $16, 4($2)
- sw $16, 0($2)
- sw $15, 4($2)

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

Transistors turning on and off

Changing Base (Radix)

- Given 4 positions, what is the largest number you can represent?

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
  - \(11_{10} = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1011_2\)
  - \(11_{10} = 8 + 2 + 1\)
- 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
### Binary and Octal

- Easy to convert Binary numbers To/From Octal.
- Group the binary digits in groups of three bits and convert each group to an Octal digit.

#### Example:

<table>
<thead>
<tr>
<th>Bin</th>
<th>Oct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

11 000 010 011 001 110 100 111 101 010 1012 = 3 0 2 3 1 6 4 7 5 2 58

### Binary and Hex

- To convert to and from hex: group binary digits in groups of four and convert according to table.
- 24 = 16

<table>
<thead>
<tr>
<th>Hex</th>
<th>Bin</th>
<th>Hex</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

#### Example:

1100 0110 0111 0100 1111 1101 01012 = C 2 6 7 4 F D 516

### 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

#### Signed 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.

### 1’s Complement Representation for Integers

- Key is to use largest positive binary numbers to represent negative numbers
- If $i = 2^n - 1 - x$
- Simply invert each bit (0->1, 1->0)
- Two zeros
- 6-bit examples:
  - 010110 = 2210 ; 1010102 = -2210
  - 0, $i = 000000$; 0 = 111111
  - $i_{10} = 000011$; -$i_{10} = 111111$

### 2’s Complement Representation for Integers

- Still use large positives to represent negatives
- If $i = 2^n - x$
- This is 1’s complement + 1
- If $i = 2^n - 1 - x + 1$
- So, invert bits and add 1
- 6-bit examples:
  - 010110 = 2210 ; 1010102 = -2210
  - 0, $i = 000000$; 0 = 111111
  - $i_{10} = 000011$; -$i_{10} = 111111$
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.
    - 100000₂ = -32₁₀; but 32₁₀ could not be represented

2's Complement Negation and Addition

- To negate a number do:
  1. Step 1: Complement the digits
  2. Step 2: Add 1

  **Example**
  - 1₄₁₀ = 001110₂
  - -1₄₁₀ = 110010₂
  - +1 = 110010₂

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

  **Example**
  - 18₁₀ - 1₄₁₀ = 18₁₀ + (-1₄₁₀) = 4₁₀
  - 010010₂
  - 110010₂
  - 000100₂

2's Complement (cont.)

- **Example:** A = 0x0ABC; 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**
  - 1₄₁₀ = 001110₂ in 6-bit representation.
  - 1₄₁₀ = 000000001110₂ in 12-bit representation
  - -1₄₁₀ = 110010₂ in 6-bit representation
  - -1₄₁₀ = 111111110010₂ 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 \( \approx 3 \times 10^8 \)
  - \( \approx 3.45 \)
- 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>E-bit</th>
<th>23-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>s exp</td>
<td>Mantissa</td>
</tr>
</tbody>
</table>
```

Floating Point Representation

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

```
| 31 30 29 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| s | E | M |
| 0111 | 1110 | 100 0000 0000 0000 0000 0000 |
```

\[ X = (-1)^{111} \times 1.1110 \times 2^{128-127} \]

- \(S = 1\)
- \(E = 128 + 2 - 127 = 3\)
- \(M = 10111\)

Double precision floating point:

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

Double precision floating point number

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-bit</td>
<td>52-bit</td>
<td></td>
</tr>
</tbody>
</table>

What about strings?

- Many important things stored as strings...
  - Your name
- What is a string?
- How should we represent strings?
### ASCII Character Representation

<table>
<thead>
<tr>
<th>Oct. Chr.</th>
<th>ASCII Char.</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 nul</td>
<td>0</td>
</tr>
<tr>
<td>001 soh</td>
<td>1</td>
</tr>
<tr>
<td>010 sta</td>
<td>2</td>
</tr>
<tr>
<td>011 eol</td>
<td>3</td>
</tr>
<tr>
<td>012 stx</td>
<td>4</td>
</tr>
<tr>
<td>013 exa</td>
<td>5</td>
</tr>
<tr>
<td>014 bks</td>
<td>6</td>
</tr>
<tr>
<td>015 tab</td>
<td>7</td>
</tr>
<tr>
<td>016 bar</td>
<td>8</td>
</tr>
<tr>
<td>017 form</td>
<td>9</td>
</tr>
<tr>
<td>020 asp</td>
<td>a</td>
</tr>
<tr>
<td>021 aux</td>
<td>b</td>
</tr>
<tr>
<td>022 asv</td>
<td>c</td>
</tr>
<tr>
<td>023 asd</td>
<td>d</td>
</tr>
<tr>
<td>024 asc</td>
<td>e</td>
</tr>
<tr>
<td>025 asc</td>
<td>f</td>
</tr>
<tr>
<td>026 asc</td>
<td>g</td>
</tr>
<tr>
<td>027 asc</td>
<td>h</td>
</tr>
<tr>
<td>028 asc</td>
<td>i</td>
</tr>
<tr>
<td>029 asc</td>
<td>j</td>
</tr>
<tr>
<td>030 asc</td>
<td>k</td>
</tr>
<tr>
<td>031 asc</td>
<td>l</td>
</tr>
<tr>
<td>032 asc</td>
<td>m</td>
</tr>
<tr>
<td>033 asc</td>
<td>n</td>
</tr>
<tr>
<td>034 asc</td>
<td>o</td>
</tr>
<tr>
<td>035 asc</td>
<td>p</td>
</tr>
<tr>
<td>036 asc</td>
<td>q</td>
</tr>
<tr>
<td>037 asc</td>
<td>r</td>
</tr>
<tr>
<td>038 asc</td>
<td>s</td>
</tr>
<tr>
<td>039 asc</td>
<td>t</td>
</tr>
<tr>
<td>040 asc</td>
<td>u</td>
</tr>
<tr>
<td>041 asc</td>
<td>v</td>
</tr>
<tr>
<td>042 asc</td>
<td>w</td>
</tr>
<tr>
<td>043 asc</td>
<td>x</td>
</tr>
<tr>
<td>044 asc</td>
<td>y</td>
</tr>
<tr>
<td>045 asc</td>
<td>z</td>
</tr>
<tr>
<td>046 asc</td>
<td>{</td>
</tr>
<tr>
<td>047 asc</td>
<td></td>
</tr>
<tr>
<td>048 asc</td>
<td>}</td>
</tr>
<tr>
<td>049 asc</td>
<td>~</td>
</tr>
<tr>
<td>050 asc</td>
<td>del</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**: 7 bit code (packed in a byte)
- **Decimal (BCD code)**
  - digits 0-9 encoded as 0000 thru 1001
two decimal digits packed per 8 bit byte
- **Integer**:
  - 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, 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**
  - **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

- **Little endian byte 0**
  - big endian byte 0

- **msb**
  - 3 2 1 0
  - Isb

- **lsb**
  - 0 1 2 3

- **2’s complement**
  - 2^-1 → 2^-1-1
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 $\ldots$

<table>
<thead>
<tr>
<th>Byte #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Not Aligned</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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
  ```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

<table>
<thead>
<tr>
<th>p 0x26d00</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0x26cf0</td>
</tr>
<tr>
<td>result 0x208</td>
</tr>
</tbody>
</table>

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?

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
  malloc(nbytes); /* 100*sizeof(int) */
  free(ptr);
  ```
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 = _______

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?

int * a = new int[100];

a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)

double * d = new double[200];

d[33] is the same as d[33]
if d is 0x00b0, then d+1 is 0x00b8, d+2 is 0x00c0
(decimal 176, 184, 192)

int * a = new int[100];

a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] id 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

char * a = new char[44];
char * begin = a;
char * end = a + 44;
while (begin < end)
{
    *(begin++) = 'z';
}

Finish Memory

Pointers
Arrays
Strings
Bitwise operations

Instruction Set Architecture
Reading
Start Chapter 3