The Big Picture

- Build machine model of execution
  - for Java and C programs
  - by examining language features
  - and deciding how they are implemented by the machine

- What is required
  - design an ISA into which programs can be compiled
  - implement the ISA in the hardware simulator

- Our approach
  - examine code snippets that exemplify each language feature in turn
  - look at Java and C, pausing to dig deeper when C is different from Java
  - design and implement ISA as needed

- The simulator is an important tool
  - machine execution is hard to visualize without it
  - this visualization is really our WHOLE POINT here
write a C program to determine Endianness
- prints “Little Endian” or “Big Endian”
- get comfortable with Unix command line and tools (important)

compile and run this program on two architectures
- IA32: lin01.ugrad.cs.ubc.ca
- Sparc: galiano.ugrad.cs.ubc.ca
- you can tell what type of arch you are on
  - % uname -a

SimpleMachine simulator
- load code into Eclipse and get it to build
- write and test MainMemory.java
- additional material available on the web page at lab time
The SM213 simulator has two main classes

- CPU implements the fetch-execute cycle
- MainMemory implements memory

The first step in building our processor

- implement 6 main internal methods of MainMemory
The Code You Will Implement

```java
/**
 * Determine whether an address is aligned to specified length.
 * @param address memory address
 * @param length byte length
 * @return true iff address is aligned to length
 */
protected boolean isAccessAligned (int address, int length) {
    return false;
}
```
/**
 * Convert an sequence of four bytes into a Big Endian integer.
 * @param byteAtAddrPlus0 value of byte with lowest memory address
 * @param byteAtAddrPlus1 value of byte at base address plus 1
 * @param byteAtAddrPlus2 value of byte at base address plus 2
 * @param byteAtAddrPlus3 value of byte at base address plus 3
 * @return Big Endian integer formed by these four bytes
 */

public int bytesToInteger (UnsignedByte byteAtAddrPlus0,
                          UnsignedByte byteAtAddrPlus1,
                          UnsignedByte byteAtAddrPlus2,
                          UnsignedByte byteAtAddrPlus3) {
    return 0;
}

/**
 * Convert a Big Endian integer into an array of 4 bytes
 * @param i an Big Endian integer
 * @return an array of UnsignedByte
 */

public UnsignedByte[] integerToBytes (int i) {
    return null;
}
**
* Fetch a sequence of bytes from memory.
* @param address address of the first byte to fetch
* @param length number of bytes to fetch
* @return an array of UnsignedByte
*/

protected UnsignedByte[] get (int address, int length) throws ... {
    UnsignedByte[] ub = new UnsignedByte [length];
    ub[0] = new UnsignedByte (0); // with appropriate value
    // repeat to ub[length-1] ...
    return ub;
}

/**
* Store a sequence of bytes into memory.
* @param address address of the first memory byte
* @param value an array of UnsignedByte values
* @throws InvalidAddressException if any address is invalid
*/
protected void set (int address, UnsignedByte[] value) throws ... {
    byte b[] = new byte [value.length];
    for (int i=0; i<value.length; i++)
        b[i] = (byte) value[i].value();
    // write b into memory ...
}
Companion

- previous module: 1, 2.1
- new: 2.2 (focus on 2.2.2 for this week)

Textbook

- 2nd ed: 3.1-3.2.1, 3.3, "New to C" sidebar of 3.4, 3.9.3
  - (skip 3.2.2 and 3.2.3)
- 1st ed: 3.1-3.2.1, 3.3, "New to C" sidebar of 3.4, 3.10
Numbers in Memory
Hexadecimal notation

- number starts with “0x”, each digit is base 16 not base 10
- e.g.: \(0x2a3 = 2 \times 16^2 + 10 \times 16^1 + 3 \times 16^0\)
- a convenient way to describe numbers when binary format is important
- each hex digit (hexit) is stored by 4 bits:
  \((0|1)x8 + (0|1)x4 + (0|1)x2 + (0|1)x1\)

Examples

- 0x10 in binary? in decimal?
- 0x2e in binary? in decimal?
- 1101 1000 1001 0110 in hex? in decimal?
- 102 in binary? in hex?
Memory and Integers

- Memory is byte addressed
  - every byte of memory has a unique address, numbered from 0 to N
  - N is huge: billions is common these days (2-16 GB)

- Integers can be declared at different sizes
  - `byte` is 1 byte, 8 bits, 2 hexits
  - `short` is 2 bytes, 16 bits, 4 hexits
  - `int` or `word` or `long` is 4 bytes, 32 bits, 8 hexits
  - `long long` is 8 bytes, 64 bits, 16 hexits

- Integers in memory
  - reading or writing an integer requires specifying a range of byte addresses
Making Integers from Bytes

Our first architectural decisions
• assembling memory bytes into integer registers

Consider 4-byte memory word and 32-bit register
• it has memory addresses i, i+1, i+2, and i+3
• we’ll just say it's “at address i and is 4 bytes long”
• e.g., the word at address 4 is in bytes 4, 5, 6 and 7.

Big or Little Endian (end means where start from, not finish)
• we could start with the BIG END of the number (most everyone but Intel)
  \[
  \begin{align*}
  i & : 2^{31} \text{ to } 2^{24} \\
  i+1 & : 2^{23} \text{ to } 2^{16} \\
  i+2 & : 2^{15} \text{ to } 2^{8} \\
  i+3 & : 2^{7} \text{ to } 2^{0}
  \end{align*}
  \]
  
• or we could start with the LITTLE END (Intel x86, some others)
  \[
  \begin{align*}
  i+3 & : 2^{31} \text{ to } 2^{24} \\
  i+2 & : 2^{23} \text{ to } 2^{16} \\
  i+1 & : 2^{15} \text{ to } 2^{8} \\
  i & : 2^{7} \text{ to } 2^{0}
  \end{align*}
  \]
Aligned or Unaligned Addresses

- we could allow any number to address a multi-byte integer
- or we could require that addresses be aligned to integer-size boundary

address modulo chunk-size is always zero

• Power-of-Two Aligned Addresses Simplify Hardware
  - smaller things always fit complete inside of bigger things
  - byte address from integer address: divide by power to two, which is just shifting bits

\[ j / 2^k = j >> k \]  
(j shifted k bits to right)
Computing Alignment

› boolean align(number, size)
  • does a number fit nicely for a particular size (in bytes)?
  ‣ divide number n by size s (in bytes), aligned if no remainder
    • easy if number is decimal
    • otherwise convert from hex or binary to decimal
  ‣ check if n mod s = 0
    • mod notation usually '%'. same as division, of course...
  ‣ check if certain number of final bits are all 0
    • pattern?
      - last 1 digit for 2-byte short
      - last 2 digits for 4-byte world
      - last 3 digits for 8-byte longlong
    • last k digits, where $2^k = s$ (size in bytes)
    • easy if number is hex: convert to binary and check

<table>
<thead>
<tr>
<th>B</th>
<th>H</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>1010</td>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>b</td>
<td>11</td>
</tr>
<tr>
<td>1100</td>
<td>c</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>d</td>
<td>13</td>
</tr>
<tr>
<td>1110</td>
<td>e</td>
<td>14</td>
</tr>
<tr>
<td>1111</td>
<td>f</td>
<td>15</td>
</tr>
</tbody>
</table>
Question

Which of the following statement(s) are true

• [A] 6 == 110_2 is aligned for addressing a short
• [B] 6 == 110_2 is aligned for addressing a long
• [C] 20 == 10100_2 is aligned for addressing a long
• [D] 20 == 10100_2 is aligned for addressing a long long (i.e., 8-byte int)
Interlude
A Quick C Primer
<table>
<thead>
<tr>
<th>Java Syntax...</th>
<th>vs. C Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>source files</strong></td>
<td><strong>source files</strong></td>
</tr>
<tr>
<td>.java is source file</td>
<td>.c is source file</td>
</tr>
<tr>
<td>.h is header file</td>
<td>.h is header file</td>
</tr>
<tr>
<td><strong>including packages in source</strong></td>
<td><strong>including headers in source</strong></td>
</tr>
<tr>
<td>import java.io.*</td>
<td>#include &lt;stdio.h&gt;</td>
</tr>
<tr>
<td><strong>printing</strong></td>
<td><strong>printing</strong></td>
</tr>
<tr>
<td>System.out.println(&quot;blah blah&quot;);</td>
<td>printf(&quot;blah blah\n&quot;);</td>
</tr>
<tr>
<td><strong>compile and run</strong></td>
<td><strong>compile and run</strong></td>
</tr>
<tr>
<td>javac foo.java</td>
<td>gcc -o foo foo.c</td>
</tr>
<tr>
<td>java foo</td>
<td>./foo</td>
</tr>
</tbody>
</table>

- do this at a Unix shell prompt (Linux, Mac Terminal, Sparc, Cygwin on Windows)
Java Hello World...

```java
import java.io.*;
public class HelloWorld {
    public static void main (String[] args) {
        System.out.println("Hello world");
    }
}
```

C Hello World...

```c
#include <stdio.h>
main() {
    printf("Hello world\n");
}
```
Java and C: Similarities

- declaration, assignment
  - int a = 4;

- control flow (often)
  - if (a == 4) ... else ...
  - for (int i = 0; i < 10; i++) {...}
  - while (i < 10) {...}

- casting
  
  int a;
  long b;
  a = (int) b;
New in C: Pointers

- pointers: addresses in memory
  - locations are first-class citizens in C
  - can go back and forth between location and value!

- pointer declaration: `<type>*`
  - `int* b;`  // b is a POINTER to an INT

- getting address of object: `&`
  - `int a;`  // a is an INT
  - `int* b = &a;`  // b is a pointer to a

- de-referencing pointer: `*`
  - `a = 10;`  // assign the value 10 to a
  - `*b = 10;`  // assign the value 10 to a

- type casting is not typesafe
  - `char a[4];`  // a 4 byte array
  - `*((int*) a) = 1;`  // treat those four bytes as an INT

- Example addresses:
  - `0x00000000`
  - `0x00000001`
  - `0x00000002`
  - `0x00000003`
  - `0x00000004`
  - `0x00000005`
  - `0x00000006`
  - `0x3e47ad40`
  - `0x3e47ad41`
  - `0x3e47ad42`
  - `0xffffffff`
Back to Numbers ...
#include <stdio.h>

int main () {
    char a[4];

    *((int*)a) = 1;

    printf("a[0]=%d a[1]=%d a[2]=%d a[3]=%d\n",a[0],a[1],a[2],a[3]);
}
Which of the following statements are true

- [A] memory stores Big Endian integers
- [B] memory stores bytes interpreted by the CPU as Big Endian integers
- [C] Neither
- [D] I don’t know
Which of these are true

• [A] The Java constants 16 and 0x10 are exactly the same integer
• [B] 16 and 0x10 are different integers
• [C] Neither
• [D] I don't know
What is the Big-Endian integer value at address 4 below?

- [A] 0x1c04b673
- [B] 0xc1406b37
- [C] 0x73b6041c
- [D] 0x376b40c1
- [E] none of these
- [F] I don’t know

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0xfe</td>
</tr>
<tr>
<td>0x1</td>
<td>0x32</td>
</tr>
<tr>
<td>0x2</td>
<td>0x87</td>
</tr>
<tr>
<td>0x3</td>
<td>0x9a</td>
</tr>
<tr>
<td>0x4</td>
<td>0x73</td>
</tr>
<tr>
<td>0x5</td>
<td>0xb6</td>
</tr>
<tr>
<td>0x6</td>
<td>0x04</td>
</tr>
<tr>
<td>0x7</td>
<td>0x1c</td>
</tr>
</tbody>
</table>
What is the value of i after this Java statement executes?

```java
int i = (byte)(0x8b) << 16;
```

- [A] 0x8b
- [B] 0x0000008b
- [C] 0x008b0000
- [D] 0xff8b0000
- [E] None of these
- [F] I don’t know
What is the value of i after this Java statement executes?

```java
i = 0xff8b0000 & 0x00ff0000;
```

- [A] 0xffffffff
- [B] 0xffffffff
- [C] 0x008b0000
- [D] I don’t know