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
Readings

- Companion
  - Ch 1, 2.1-2.2.

- Textbook
  - *Historical Perspective. Access to Information and Data Alignment*
    - 2nd Ed: 3.1-3.4, 3.9.3
    - 1st Ed: 3.1-3.4, 3.10

Numbers in Memory
Hexadecimal notation
- “0x” followed by number (e.g., 0x2A3 = 2x16^2 + 10x16^1 + 3x16^0)
- 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
- some examples...

Integers of different sizes
- **byte** is 8 bits, 2 hexits
- **short** is 2 bytes, 16 bits, 4 hexits
- **int** or **word** is 4 bytes, 32 bits, 8 hexits
- **long long** is 8 bytes, 64 bits, 16 hexits

Memory is byte addressed
- every byte of memory has a unique address, number from 0 to N
- 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 its **“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
- we could start with the BIG END of the number (everyone but Intel)

  ![Big Endian Diagram]

- or we could start with the LITTLE END (Intel)

  ![Little Endian Diagram]
Aligned or Unaligned Addresses

- we could allow any number to address a multi-byte integer
  - *disallowed on most architectures
  - *allowed on Intel, but slower
- 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
    - word contains exactly two complete shorts
  - byte address to integer address is division by power to two, which is just shifting bits
    \[ j / 2^k = j \gg k \]  \hspace{1cm} (j shifted k bits to right)

Interlude

A Quick C Primer
A few initial things about C

† source files
  • .c is source file
  • .h is header file

† including headers in source
  • #include <stdio.h>

† pointer types
  • 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[0]) = 1; // treat those four bytes as an INT

† compile and run
  • at UNIX (e.g., Linux, MacOS, or Cygwin) shell prompt
  • gcc -o foo foo.c
  • ./foo
Determining Endianness of a Computer

```c
#include <stdio.h>

t 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]);
} 
```
Questions

Which of the following statement(s) are true

- [A] 6 == 110₂ is aligned for addressing a short
- [B] 6 == 110₂ is aligned for addressing a long
- [C] 20 == 10100₂ is aligned for addressing a long
- [D] 20 == 10100₂ is aligned for addressing a long long (i.e., 8-byte int)

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>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0:</td>
</tr>
<tr>
<td>0x1:</td>
</tr>
<tr>
<td>0x2:</td>
</tr>
<tr>
<td>0x3:</td>
</tr>
<tr>
<td>0x4:</td>
</tr>
<tr>
<td>0x5:</td>
</tr>
<tr>
<td>0x6:</td>
</tr>
<tr>
<td>0x7:</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] 0xffff0000
- [B] 0xff8b0000
- [C] 0x008b0000
- [D] I don't know
In the Lab ...

- 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: any of the other undergrad machines
  - 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 Main Memory Class

- 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

<table>
<thead>
<tr>
<th>CPU</th>
<th>MainMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>isAligned</td>
</tr>
<tr>
<td>execute</td>
<td>length</td>
</tr>
<tr>
<td>read</td>
<td>bytesToInteger</td>
</tr>
<tr>
<td>readInteger</td>
<td>integerToBytes</td>
</tr>
<tr>
<td>write</td>
<td>get</td>
</tr>
<tr>
<td>writeInteger</td>
<td>set</td>
</tr>
</tbody>
</table>
The Code You Will Implement

/**
 * 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 ...
}