Numbers and Memory

**Masking**
- bitmask: pattern of bits you construct with/for logical operations
  - mask with 0 to throw bits away
  - mask with 1 to let bit values pass through
- Examples:
  - mask with 8:0 to turn bits off
  - mask with 8:0 (in binary) to let bit values pass through
  - example: 0x00ff & 0x3a2b = 0x002b

**Bit Shifting**
- Two's Complement: Reminder
  - unsigned: all possible values interpreted as positive numbers
    
    | B | H | Signed (Decimal) | Unsigned |
    |---|---|------------------|----------|
    | 1111 | 1111 | 0xff | 255 |
    | 1110 | 1110 | 0xfe | 254 |
    | 1101 | 1101 | 0xfd | 253 |
    | 1100 | 1100 | 0xfc | 252 |
    | 1100 | 1000 | 0xfb | 250 |
    | 1100 | 1001 | 0xfa | 249 |
    | 1100 | 1010 | 0xf9 | 247 |
    | 1100 | 1011 | 0xf8 | 246 |
    | 1100 | 1100 | 0xf7 | 244 |
    | 1100 | 1101 | 0xf6 | 243 |
    | 1100 | 1110 | 0xf5 | 241 |
    | 1100 | 1111 | 0xf4 | 240 |

- signed: two's complement
  - the first half of the numbers are positive, the second half are negative
  - start at 0, go to top positive value, "wrap around" to most negative value, end up at -1

<table>
<thead>
<tr>
<th>B</th>
<th>H</th>
<th>Signed (Decimal)</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>0xff</td>
<td>255</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>0xfe</td>
<td>254</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>0xfd</td>
<td>253</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfc</td>
<td>252</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfb</td>
<td>250</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfa</td>
<td>249</td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
<td>0xf9</td>
<td>247</td>
</tr>
<tr>
<td>1100</td>
<td>1110</td>
<td>0xf8</td>
<td>246</td>
</tr>
<tr>
<td>1100</td>
<td>1111</td>
<td>0xf7</td>
<td>244</td>
</tr>
<tr>
<td>1101</td>
<td>0000</td>
<td>0xf6</td>
<td>243</td>
</tr>
<tr>
<td>1101</td>
<td>0001</td>
<td>0xf5</td>
<td>241</td>
</tr>
<tr>
<td>1101</td>
<td>0010</td>
<td>0xf4</td>
<td>240</td>
</tr>
<tr>
<td>1101</td>
<td>0011</td>
<td>0xf3</td>
<td>239</td>
</tr>
<tr>
<td>1101</td>
<td>0100</td>
<td>0xf2</td>
<td>238</td>
</tr>
<tr>
<td>1101</td>
<td>0101</td>
<td>0xf1</td>
<td>236</td>
</tr>
<tr>
<td>1101</td>
<td>0110</td>
<td>0xf0</td>
<td>235</td>
</tr>
<tr>
<td>1101</td>
<td>0111</td>
<td>0xff</td>
<td>255</td>
</tr>
</tbody>
</table>

- Two’s Complement: Byte
  - unsigned: all possible values interpreted as positive numbers

<table>
<thead>
<tr>
<th>B</th>
<th>H</th>
<th>Signed (Decimal)</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>0xff</td>
<td>255</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>0xfe</td>
<td>254</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>0xfd</td>
<td>253</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfc</td>
<td>252</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfb</td>
<td>250</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfa</td>
<td>249</td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
<td>0xf9</td>
<td>247</td>
</tr>
<tr>
<td>1100</td>
<td>1110</td>
<td>0xf8</td>
<td>246</td>
</tr>
<tr>
<td>1100</td>
<td>1111</td>
<td>0xf7</td>
<td>244</td>
</tr>
<tr>
<td>1101</td>
<td>0000</td>
<td>0xf6</td>
<td>243</td>
</tr>
<tr>
<td>1101</td>
<td>0001</td>
<td>0xf5</td>
<td>241</td>
</tr>
<tr>
<td>1101</td>
<td>0010</td>
<td>0xf4</td>
<td>240</td>
</tr>
<tr>
<td>1101</td>
<td>0011</td>
<td>0xf3</td>
<td>239</td>
</tr>
<tr>
<td>1101</td>
<td>0100</td>
<td>0xf2</td>
<td>238</td>
</tr>
<tr>
<td>1101</td>
<td>0101</td>
<td>0xf1</td>
<td>236</td>
</tr>
<tr>
<td>1101</td>
<td>0110</td>
<td>0xf0</td>
<td>235</td>
</tr>
<tr>
<td>1101</td>
<td>0111</td>
<td>0xff</td>
<td>255</td>
</tr>
</tbody>
</table>

- signed: two’s complement
  - the first half of the numbers are positive, the second half are negative
  - start at 0, go to top positive value, "wrap around" to most negative value, end up at -1

<table>
<thead>
<tr>
<th>B</th>
<th>H</th>
<th>Signed (Decimal)</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1111</td>
<td>0xff</td>
<td>255</td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>0xfe</td>
<td>254</td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>0xfd</td>
<td>253</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfc</td>
<td>252</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfb</td>
<td>250</td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>0xfa</td>
<td>249</td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
<td>0xf9</td>
<td>247</td>
</tr>
<tr>
<td>1100</td>
<td>1110</td>
<td>0xf8</td>
<td>246</td>
</tr>
<tr>
<td>1100</td>
<td>1111</td>
<td>0xf7</td>
<td>244</td>
</tr>
<tr>
<td>1101</td>
<td>0000</td>
<td>0xf6</td>
<td>243</td>
</tr>
<tr>
<td>1101</td>
<td>0001</td>
<td>0xf5</td>
<td>241</td>
</tr>
<tr>
<td>1101</td>
<td>0010</td>
<td>0xf4</td>
<td>240</td>
</tr>
<tr>
<td>1101</td>
<td>0011</td>
<td>0xf3</td>
<td>239</td>
</tr>
<tr>
<td>1101</td>
<td>0100</td>
<td>0xf2</td>
<td>238</td>
</tr>
<tr>
<td>1101</td>
<td>0101</td>
<td>0xf1</td>
<td>236</td>
</tr>
<tr>
<td>1101</td>
<td>0110</td>
<td>0xf0</td>
<td>235</td>
</tr>
<tr>
<td>1101</td>
<td>0111</td>
<td>0xff</td>
<td>255</td>
</tr>
</tbody>
</table>

**Reading**
- Companion
  - previous module: 1, 2.1
  - new: 2.2 (focus on 2.2.2 for this week)
- Textbook
  - A Historical Perspective, Machine-Level Code, Data Formats, Data Alignment.
    - 2ed: 3.1-3.2.1, 3.3, 3.9.3
    - 1ed: 3.1-3.2.1, 3.3, 3.10

**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 a 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 ... address is aligned to length
 */
protected boolean isAccessAligned (int address, int length) {
  return false;
}
```

**Lab/Assignment 1**
- SimpleMachine simulator
  - load code into Eclipse and get it to build/run
  - write and test MainMemory.j ava
  - set:
    - isAccessAligned
    - bytesToInteger
    - IntegerToBytes
```java
protected UnsignedByte get (int address, int length) throws {
  // return an array of UnsignedByte
  return null;
}
```

**Languages and Tools**
- SM213 Assembly
  - you will trace, write, read
  - use SM213 simulator to trace and execute
- Java
  - you will read, write
  - use Eclipse IDE to edit, compile, debug, run
  - SM213 simulator written in Java; you will implement specific pieces
- C
  - you will read, write
  - go to compile, gdb to debug, command line to run

**CPU fetch execute**
MainMemory
  - isAligned
  - bytesToInteger
  - integerToBytes
  - get
  - set
  - read
  - readInteger
  - write
  - writeInteger

**Numbers and Bits**
- Hexadecimal, Hex, and Decimal Refresher
  - Bit shifting: multiply/divide by powers of 2
    - left shift by k bits, "<< k": multiply by 2^k
      - old bits on left end drop off, new bits on right end set to 0
      - examples:
        - 0000 0100 = 0 x 2^4 + 1 x 2^3 + 0 x 2^2 + 0 x 2^1 + 0 x 2^0
        - 0011 0100 = 2 x 2^4 + 0 x 2^3 + 1 x 2^2 + 0 x 2^1 + 0 x 2^0
    - right shift by k bits, ">> k": divide by 2^k
      - old bits on right end drop off, new bits on left end set to 0
      - examples:
        - 0000 0100 = 0 x 2^4 + 1 x 2^3 + 0 x 2^2 + 0 x 2^1 + 0 x 2^0
        - 0000 0010 = 0 x 2^3 + 1 x 2^2 + 0 x 2^1 + 0 x 2^0
    - why do this? two good reasons:
      - much faster than multiply, much, much faster than division
      - good way to move bits around to where you need them

<insert diagram of memory layout and addressing>
Two’s Complement and Sign Extension
‣ normally, pad with 0s when extending to larger size
  • 0x8b byte (139) becomes 0x00000000 int (0)
‣ reminder: why do all this?
  • add/subtract works without checking if number positive or negative

Bit Shifting in Java
‣ signed/arithmetic right shift by k bits, ">> k": divide by 2^k
  • old bits on right end drop off, new bits on left end set to (sign) bit
    • examples
      • 1010 >> 1 = 1110
      • 1110 >> 2 = 0000
  • unsigned/logical right shift by k bits, ">>> k":
    • old bits on right end drop off, new bits on left set to 0
    • but... be careful - requires int/long and automatically promotes up
      • so bytes automatically promoted, but with sign extension

Making Integers from Bytes
‣ Our first architectural decisions
  • assembing memory bytes into integer registers
  • computing alignment

In the Lab ... Revisited
‣ SimpleMachine simulator
  • load code into Eclipse and get it to build/run
  • write and test MainMemory.java
    • get/set should check for out of bounds access but not alignment
    • isAccessAligned checks for alignment

Questions
‣ Which of the following statements are true
  • [A] memory stores Big Endian integers
  • [B] memory stores bytes intepreted by the CPU as Big Endian integers
  • [C] Neither
  • [D] I don't know

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 word
    • last 3 digits for 8-byte longword
    • last k digitst, where 2^k = n (base in bytes)
  • easy if number is hex: convert to binary and check

What is the Big-Endian integer value at address 4 below?
  • [A] 0x1c84b673
  • [B] 0x14b037
  • [C] 0x73b8041c
  • [D] 0x376b4801
  • [E] none of these
  • [F] I don’t know

What is the value of i after this Java statement executes?
  • int i = (0x00000000) << 16;
  • [A] 0
  • [B] 0x00000000
  • [C] 0x00000000
  • [D] I don’t know
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