DISCUSSION – WEEK 1

TA : Mathanky
Email : mathanky04@ucla.edu
Office Hours : Thursday, 11.30PM - 1.30PM,
Boelter Hall 3256F
REVIEW OF WEEK 1

- Bits and Bytes
- Integers
Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires
Encoding Byte Values

- **Byte = 8 bits**
  - Binary 00000000₂ to 11111111₂
  - Decimal: 0₁₀ to 255₁₀
  - Hexadecimal 00₁₆ to FF₁₆
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xfa1d37b

One byte is represented by 2 hexadecimal characters
General Boolean Algebra

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>x • y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

XOR?
Perform the following BitWise operations:

1. \(7 \& 8\)
2. \(12 \mid 11\)
3. \(5 \^ 7\)
4. \(\sim 15\)
Shift Operations

**Left Shift:** \( x \ll y \)
- Shift bit-vector \( x \) left \( y \) positions
  - Throw away extra bits on left
  - Fill with 0’s on right

**Right Shift:** \( x \gg y \)
- Shift bit-vector \( x \) right \( y \) positions
  - Throw away extra bits on right
  - Logical shift
  - Fill with 0’s on left
  - Arithmetic shift
  - Replicate most significant bit on left

**Undefined Behavior**
- Shift amount &lt; 0 or &ge; word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>
Integers can be **signed** or **unsigned**.

**Unsigned**

\[ B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i \]

**Two’s Complement**

\[ B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i \]

- short int \( x = 15213; \)
- short int \( y = -15213; \)

**C short 2 bytes long**

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( 15213 )</td>
<td>( 3B \ 6D \ 00111011 \ 01101101 )</td>
</tr>
<tr>
<td>( y )</td>
<td>( -15213 )</td>
<td>( C4 \ 93 \ 11000100 \ 10010011 )</td>
</tr>
</tbody>
</table>

**Sign Bit**

- For 2’s complement, most significant bit indicates sign
  - 0 for nonnegative
  - 1 for negative
Numeric Ranges

**Unsigned Values**
- **$UMin$** = 0
  - 000...0
- **$UMax$** = $2^w - 1$
  - 111...1

**Two’s Complement Values**
- **$TMin$** = $-2^{w-1}$
  - 100...0
- **$TMax$** = $2^{w-1} - 1$
  - 011...1

**Other Values**
- Minus 1
  - 111...1

Values for $W = 16$

<table>
<thead>
<tr>
<th></th>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMax</td>
<td>65535</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>Tmax</td>
<td>32767</td>
<td>7F FF</td>
<td>01111111 11111111</td>
</tr>
<tr>
<td>Tmin</td>
<td>-32768</td>
<td>80 00</td>
<td>10000000 00000000</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>FF FF</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00 00</td>
<td>00000000 00000000</td>
</tr>
</tbody>
</table>
## Unsigned & Signed Numeric Values

<table>
<thead>
<tr>
<th>$X$</th>
<th>$\text{B2U}(X)$</th>
<th>$\text{B2T}(X)$</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>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

- **Equivalence**
  - Same encodings for nonnegative values

- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

- **Can Invert Mappings**
  - $U2B(x) = \text{B2U}^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = \text{B2T}^{-1}(x)$
    - Bit pattern for two’s comp integer
Mapping Between Signed & Unsigned

Two’s Complement

\[ x \rightarrow T2B \rightarrow T2U \rightarrow B2U \rightarrow ux \]

Maintain Same Bit Pattern

Unsigned

\[ ux \rightarrow U2B \rightarrow U2T \rightarrow B2T \rightarrow x \]

Maintain Same Bit Pattern

Mappings between unsigned and two’s complement numbers:

Keep bit representations and reinterpret
Conversion Visualized

- 2’s Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive

2’s Complement Range

Unsigned Range

UMax
UMax - 1
TMax + 1
TMax

TMax
TMin
0
-1
-2
0

UMax
UUnsigned Range
### Casting Surprises

#### Expression Evaluation
- If there is a mix of unsigned and signed in single expression, _signed values implicitly cast to unsigned_
- Including comparison operations `<`, `>`, `==`, `<=`, `>=`
- Examples for $W = 32$: $TMIN = -2,147,483,648$, $TMAX = 2,147,483,647$

<table>
<thead>
<tr>
<th>Constant$_1$</th>
<th>Constant$_2$</th>
<th>Relation</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0U</td>
<td>==</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>&lt;</td>
<td>signed</td>
</tr>
<tr>
<td>-1</td>
<td>0U</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>-2147483647-1</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>2147483647U</td>
<td>-2147483647-1</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>-1</td>
<td>-2</td>
<td>&gt;</td>
<td>signed</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>-2</td>
<td>&gt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>2147483648U</td>
<td>&lt;</td>
<td>unsigned</td>
</tr>
<tr>
<td>2147483647</td>
<td>(int) 2147483648U</td>
<td>&gt;</td>
<td>signed</td>
</tr>
</tbody>
</table>
Sign Extension

Task:
- Given \( w \)-bit signed integer \( x \)
- Convert it to \( w+k \)-bit integer with same value

Rule:
- Make \( k \) copies of sign bit:
- \( X' = x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0 \)
Unsigned Addition

Operands: $w$ bits

\[
\begin{array}{c}
\hline
u \\
+ v \\
\hline
\end{array}
\]

True Sum: $w+1$ bits

\[
\begin{array}{c}
\hline
u + v \\
\hline
\end{array}
\]

Discard Carry: $w$ bits

\[
\begin{array}{c}
\hline
\text{UAdd}_w(u, v) \\
\hline
\end{array}
\]

- **Standard Addition Function**
  - Ignores carry output

- **Implements Modular Arithmetic**

  \[
  s = \text{UAdd}_w(u, v) = (u + v) \mod 2^w
  \]
ADDISON OF BINARY NUMBERS

- How do we add 10 + 5 using binary representation?
- What about 10 - 5?

LEFT SHIFT: MULTIPLY BY POWER OF 2

RIGHT SHIFT: DIVIDE BY POWER OF 2

- How do we multiply an integer x with 5?
- How do we divide an integer x by 8?
Problems to think about:

1. Check if a given number is odd or even, by using only bitwise operators.
2. Check if an integer $x$ is divisible by 4.
3. Check if an integer $x$ is a power of 2.
Write a function that, given a number \( n \), returns another number where the \( k \)th bit from the right is set to 0.

Examples:

\[
\text{killKthBit}(37, 3) = 33 \quad \text{because} \quad 37_{10} = 100101_2 \rightarrow 100001_2 = 33_{10} \\
\text{killKthBit}(37, 4) = 37 \quad \text{because the 4th bit from the right is already 0.}
\]
Machine Level Programming - Basics

Assembly/Machine Code View

CPU

Registers

Condition Codes

Addresses

Data

Instructions

Memory

Code

Data

Stack

Programmer-Visible State

- PC: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)

- Register file
  - Heavily used program data

- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures
Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
  - Data values
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes

- (SIMD vector data types of 8, 16, 32 or 64 bytes)

- Code: Byte sequences encoding series of instructions

- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory
<table>
<thead>
<tr>
<th>x86-64 Integer Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rbx</td>
</tr>
<tr>
<td>%rcx</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsi</td>
</tr>
<tr>
<td>%rdi</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
<tr>
<td>%rbp</td>
</tr>
<tr>
<td>%r8</td>
</tr>
<tr>
<td>%r9</td>
</tr>
<tr>
<td>%r10</td>
</tr>
<tr>
<td>%r11</td>
</tr>
<tr>
<td>%r12</td>
</tr>
<tr>
<td>%r13</td>
</tr>
<tr>
<td>%r14</td>
</tr>
<tr>
<td>%r15</td>
</tr>
</tbody>
</table>

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)
# movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imm</strong></td>
<td><strong>Reg</strong></td>
<td>movq $0x4,%rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq $-147,(%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td><strong>Reg</strong></td>
<td><strong>Reg</strong></td>
<td>movq %rax,%rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq %rax,(%rdx)</td>
<td>*p = temp;</td>
</tr>
<tr>
<td><strong>Mem</strong></td>
<td><strong>Reg</strong></td>
<td>movq (%rax),%rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>
Simple Memory Addressing Modes

- **Normal (R)** \( \text{Mem}[\text{Reg}[R]] \)
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

  \[
  \text{movq} \ (\%rcx),\%rax
  \]

- **Displacement D(R)** \( \text{Mem}[\text{Reg}[R]+D] \)
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \[
  \text{movq} \ 8(\%rbp),\%rdx
  \]
Complete Memory Addressing Modes

■ Most General Form

\[ D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri] + D] \]

- **D:** Constant “displacement” 1, 2, or 4 bytes
- **Rb:** Base register: Any of 16 integer registers
- **Ri:** Index register: Any, except for %rsp
- **S:** Scale: 1, 2, 4, or 8 (*why these numbers?*)

■ Special Cases

\[ (Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \]
\[ D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \]
\[ (Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri]] \]
Example of Simple Addressing Modes

```c
void swap  
   (long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

swap:
```
movq    (%rdi), %rax
movq    (%rsi), %rdx
movq    %rdx, (%rdi)
movq    %rax, (%rsi)
ret
```
Address Computation Instruction

- **leaq** *Src, Dst*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

**Example**

```c
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax  # t = x+2*x
salq $2, %rax            # return t<<2
```
mov vs lea

\[
\begin{align*}
\text{movl} &\quad (%rdx), \%rax \\
\text{leal} &\quad (%rdx), \%rax
\end{align*}
\]

movl takes the \textbf{contents} of what’s stored in register %rdx and moves it to %rax.

leal computes the load effective \textbf{address} and stores it in %rax. leal analogous to returning a pointer, whereas movl is analogous to returning a dereferenced pointer.