Today

- Hexadecimal
- Little vs Big Endian
- Assembly basics

Assignments

- HW4 out (today)
“Byte” Ordering: Hexadecimal Values

Hexadecimal (Base 16)

- hexadecimal digits: 0, 1, 2, ..., A, B, C, D, E, F
- where A = 10, B = 11, C = 12, ..., F = 15
- one byte (8-bits) can be represented by two hexadecimal digits
- in c/c++/assembly prefixed by 0x, e.g.: 0x1A2B3C01

\[
\begin{array}{c|c|c|c}
\text{Places} & 4096's & 256's & 16's & 1's \\
\hline
4 \text{ bits} & 2 & F & A & 7 \\
\hline
8 \text{ bits} & & & & = 2 \cdot 4096 + 15 \cdot 256 + 10 \cdot 16 + 7 \\
\hline
12 \text{ bits} & & & & = 12,199 \\
\hline
16 \text{ bits} & & & & \\
\end{array}
\]

Exercise: What is the largest decimal value for numbers with ...

- 1 hex digit \( \ldots \quad 16^1 - 1 = 2^4 - 1 = 15 \)
- 2 hex digits \( \ldots \quad 16^2 - 1 = 2^8 - 1 = 255 \)
- \( n \) hex digits \( \ldots \quad 16^n - 1 = 2^{4n} - 1 \)
Exercises: Convert the following bytes to/from (unsigned) decimal

- $0x01$ ... 1
- $0x10$ ... 16
- $0x1C$ ... 28
- $0x2E$ ... 46
- $0xFA$ ... 250

Q: Given a byte in hexadecimal, how do we know if it is a negative (signed) value?
- higher order bit is $\geq 8$

Exercises: Convert the following to hexadecimal

- 15 ... $0x0F$
- 171 ... $0xAB$
- 128 ... $0x80$
- 316 ... $0x13C$

In C:
```
#include <stdio.h>
int main()
{
    unsigned int x = 0xFA;
    printf("%d\n", x); /* prints 250 */
    unsigned int y = 177;
    printf("0x%X\n", y); /* prints 0xB1*/
}
```
**“Byte” Ordering: Big vs Little Endian**

- While bits are stored in contiguous memory cells
- The smallest “chunk” of addressable data is a byte (8 bit addressing)

Given a 4-byte (32-bit) value, each byte is technically addressable ...
- the way the bytes are laid out in memory is called **“endianess”**
- which are just conventions for how to store multi-byte values

**Big Endian**: Given that the value to store is 0x12345678
- In big endian the “big end” (high order) of the number is stored first

<table>
<thead>
<tr>
<th>value</th>
<th>12</th>
<th>34</th>
<th>56</th>
<th>78</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>0x01</td>
<td>0x02</td>
<td>0x03</td>
<td>0x04</td>
</tr>
</tbody>
</table>

**Little Endian**: Given that the value to store is 0x12345678
- In little endian the “little end” (low order) of the number is stored first

<table>
<thead>
<tr>
<th>value</th>
<th>78</th>
<th>56</th>
<th>34</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>0x01</td>
<td>0x02</td>
<td>0x03</td>
<td>0x04</td>
</tr>
</tbody>
</table>

Different manufacturers use different conventions ...
- Most Intel machines use little endian
- IBM / Sun machines use/used big endian
- Most ARM processors are “bi-endian” (can switch conventions) based on OS
- E.g., Android and iOS operate in little-endian mode
Most of the time endianess is invisible to programmers

- handled by compilers / operating systems

Sometimes you have to be aware of endianess though ...

- e.g., when writing network applications (how is data being transferred)
- when accessing (or viewing) integer data as byte sequences ...

To see byte ordering in action:

```c
#include <stdint.h>
#include <stdio.h>

int main()
{
    uint32_t word = 0x0A0B0C0D; /* four-bytes */
    char* ptr = (char*)&word; /* pointer to first byte in word */
    for (int i = 0; i < 4; ++i) /* iterate through low-to-high addresses */
        printf("%02x ", ptr[i]); /* print each byte in the word */
    printf("\n");
}
```

Also when viewing machine code disassembly (via `objdump -d a.out`):

```
58c:   48 8b 05 55 0a 20 00   mov   0x200a55(%rip), %rax
```


Overview of gdb basics

Consider the following simple assembly program (in a file lect11.s):

```assembly
.global _start
.text
_start:
    movl $80155, %eax  # store value in eax register
    movl $10644, %ebx  # store value in ebx register
    # exit program
    movq $60, %rax
    xor %rdi, %rdi
    syscall
```

To compile and run (where “$” denotes command prompt)

```
$ gcc -c -g lect10.s
$ ld lect10.o -o lect10
$ ./lect10
```
Start gdb (GNU Debugger) to “trace” the program ...

$ gdb lect10
(gdb) break _start
(gdb) run
(gdb) layout asm
(gdb) layout reg

These commands:

- start gdb                     ... gdb lect10
- create a breakpoint at _start address   ... break _start
- run the program (to _start     ... run
- open the code viewer          ... layout asm
- open the register viewer     ... layout reg
You can now “step” instruction-by-instruction through your program

- each step updates the registers after the instruction is executed

```
(gdb) step
```

Most commands can be simplified by using their first letter:

- `b` for break
- `r` for run
- `s` for step
Some x86-64 and GNU Assembly Basics

Data types, sizes, and assembly operand sizes

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly-code Suffix</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1 byte (8 bits)</td>
</tr>
<tr>
<td>short</td>
<td>Word</td>
<td>w</td>
<td>2 bytes (16 bits)</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4 bytes (32 bits)</td>
</tr>
<tr>
<td>long</td>
<td>Quad word</td>
<td>q</td>
<td>8 bytes (64 bits)</td>
</tr>
<tr>
<td>char*</td>
<td>Quad word</td>
<td>q</td>
<td>8 bytes (64 bits)</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4 bytes (32 bits)</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>l</td>
<td>8 bytes (64 bits)</td>
</tr>
</tbody>
</table>

Each “general-purpose” register is broken into

![Registers Diagram]

- thus “portions” of (entire) registers are accessible for different data types
There are 16 “general-purpose” registers

• each broken into corresponding parts as in previous RAX example

• each have general conventions for their use in programs (more later)

• however, they are all general purpose (can be used for multiple things)

<table>
<thead>
<tr>
<th>64-bit</th>
<th>32-bit</th>
<th>16-bit</th>
<th>8-bit (h)</th>
<th>8-bit (l)</th>
<th>conventions</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td>accumulator, return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td>base, callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td>loop counter, 4th arg</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td>data (mult/div), 3rd arg</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
<td></td>
<td>source index (strings), 2nd arg</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%sil</td>
<td></td>
<td>dest index (strings), 1st arg</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bpl</td>
<td></td>
<td>stack base pointer, callee saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td>%spl</td>
<td></td>
<td>stack pointer (stack top)</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td></td>
<td>5th arg</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td></td>
<td>6th arg</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td></td>
<td>caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td></td>
<td>caller saved</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
<td>caller saved</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
<td>caller saved</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
<td>caller saved</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></td>
<td>caller saved</td>
</tr>
</tbody>
</table>
Some additional (non general-purpose) registers

instruction pointer: %rip (64 bit)
  • with parts ... %eip (32 bit), %ip (16 bit)

Flag registers: eflag (different bits signify different flags)
  • Carry (CF) ... unsigned carry or borrow out of most significant bit
  • Parity (PF) ... if number of set bits in the result is even (error correction)
  • Zero (ZF) ... if result of previous operation is zero
  • Sign (SF) ... if result of previous operation is negative
  • Overflow (OF) ... signed result won’t fit in number of bits (e.g., 127+127)
  • Adjust (AF) ... if carry or borrow from lowest four bits (lower “nibble”)
  • Interrupt (IF) ... enables process to handle hardware interrupts

... these registers are **not meant** for programs to set directly
The **mov** instruction

The basic syntax:

\[
\text{movs} \ src, \ dst
\]

- **s** is the size of data being moved ... e.g., movb, movw, movl, movq
- **src** is either a literal ($42$), memory location (sum), or a register (%rax)
- **dst** is either a memory location or a register
- for example: "movq $42, %rax" or "movl %eax, sum"

General rules for using **mov** instructions

- both operands should (usually) be the same size
- both operands cannot be memory locations (must use register as in between)
- instruction pointer (e.g., %rip) cannot be a destination operand