Monday, October 30, 2017

Lecture 11: Assembly Part I

Reading: Chapter 3.1-3.4

Lecturers: Julie Zelenski and Chris Gregg

```
# looper.c

void count_from_offset(int offset)
{
    for (int i=0; i < 10; i++) {
        printf("Count: %d\n",i+offset);
    }
}

(gdb)
--looper.c--
| 3 |
| 4 | void count_from_offset(int offset) |
| 5 | { |
| 6 | for (int i=0; i < 10; i++) { |
| 7 | printf("Count: %d\n",i+offset);
| 8 | } |
| 9 | } |
| 10 |

0x400531 <count_from_offset+4>  sub    $0x20,%rsp
0x400535 <count_from_offset+8>  mov    %edi,-0x14(%rbp)
0x400538 <count_from_offset+11> movl   $0x0,-0x4(%rbp)
0x40053f <count_from_offset+18> jmp  0x40055e <count_fro|
0x400541 <count_from_offset+20> mov   -0x14(%rbp),%eax
0x400544 <count_from_offset+23> mov   -0x4(%rbp),%edx
0x400547 <count_from_offset+26> add   %edx,%eax
0x400549 <count_from_offset+28> mov   %eax,%esi

child process 8824 In: count_from_off* Line: 6    PC: 0x400538
```

Today's Topics

• Logistics
  • Midterm Friday, during class, **Dinkelspiel Auditorium**
  • Will use computer program (BlueBook) that requires a laptop
  • You will be able to test your laptop beforehand, and we will have power strips for those that need them.
• Reading: Chapter 3.1-3.4
• Programs from class: /afs/ir/class/cs107/samples/lect11
• Introduction to x86 Assembly Language
  • Overview of assembly code and the weirdness of x86 (primarily historical)
    • First example: HelloWorld, gcc -S, gdbtui
    • Second Example: Looper
• Registers
• Data formats
• Addressing Modes
• The `mov` instruction
• Access to variables of various types
What is Assembly Code?

- Computers execute "machine code," which is a sequence of bytes that encode low-level operations for manipulating data, managing memory, read and write from storage, and communicate with networks.

- The "assembly code" for a computer is a textual representation of the machine code giving the individual instructions to the underlying machine.
What is Assembly Code?

• **gcc** generates assembly code from C code
• Assembly is raw — there is no type checking, and the instructions are simple. It is unique to the type of processor (e.g., the assembly for your computer cannot run on your phone)
• Humans can write assembly (and, in fact, in the early days of computing they had to write assembly), but it is more productive to be able to read and understand what the compiler produces.
• **gcc** is almost always going to produce better optimized code than a human could, and understanding what the compiler produces is important.
• The Intel-based computers we use has its roots in Intel's 16-bit, 1978 processor with the name 8086.
• Intel has taken a strict backwards-compatibility approach to new processors, and their 32- and 64-bit processors have built upon the original 8086 Assembly code.
• These days, when we learn x86 assembly code, we have to keep this history in mind. Naming of "registers," for example, has historical roots, so bear with it.
Before we look at some assembly code, let's talk about some things that have been hidden from us when writing C code.

Machine code is based on the "instruction set architecture" (ISA), which defines the behavior and layout of the system. Behavior is defined as if instructions are run one after the other, and memory appears as a very large byte array.
New things that have been hidden:

- The *program counter* (PC), called "%rip" indicates the address of the next instruction. We cannot modify this directly.
- The "register file" contains 16 named locations that store 64-bit values. Registers are the fastest memory on your computer.
  - Registers can hold addresses, or integer data. Some registers are used to keep track of your program's state, and others hold temporary data.
  - Registers are used for arithmetic, local variables, and return values for functions.
- The condition code registers hold status information about the most recently executed arithmetic or logical instruction. These are used to control program flow — e.g., if the result of an addition is negative, exit a loop.
- There are vector registers, which hold integer or floating point values.
• Unlike C, there is no model of different data types, and memory is simply a large, byte-addressable array.

• There is no distinction between signed and unsigned integers, between different types of pointers, or even between pointers and integers.

• A single machine instruction performs only a very elementary operation. For example:
  • there is an instruction to add two numbers in registers. That's all the instruction does.
  • there is an instruction that transfers data between a register and memory.
  • there is an instruction that conditionally branches to a new instruction address.

• Often, one C statement generates multiple assembly code instructions.
• Let's look at some assembly code!

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

int main()
{
    int i = 1;
    printf("Hello, World %d!\n", i);
    return 0;
}
```

```assembly
.LC0:
.string "Hello, World %d!\n"
main:
    subq $8, %rsp
    movl $1, %esi
    movl $.LC0, %edi
    movl $0, %eax
    call printf
    movl $0, %eax
    addq $8, %rsp
    ret
```

$ gcc -S -g -Og -std=gnu99 -Wall $warnflags hello.c
$ vim hello.s

Lots of extra stuff taken away
x86 Assembly: a first look

• Let's look at some assembly code!

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

int main()
{
    int i = 1;
    printf("Hello, World %d!\n", i);
    return 0;
}
```

$ make
gcc -g -Og -std=gnu99 -Wall $warnflags
hello.c -o hello
$ gdbtui hello
(gdb) layout split
(or type ctrl-x, 2 to get split view)

(gdb) disas main
Dump of assembler code for function main:
  0x000000000040055d <+0>:   sub $0x8,%rsp
  0x0000000000400561 <+4>:   mov $0x1,%edx
  0x0000000000400566 <+9>:   mov $0x400614,%esi
  0x000000000040056b <+14>:  mov $0x1,%edi
  0x0000000000400570 <+19>:  mov $0x0,%eax
  0x0000000000400575 <+24>:  callq 0x400460 <__printf_chk@plt>
  0x000000000040057a <+29>:  mov $0x0,%eax
  0x000000000040057f <+34>:  add $0x8,%rsp
  0x0000000000400583 <+38>:  retq
#include<stdlib.h>
#include<stdio.h>

int main()
{
    int i = 1;
    printf("Hello, World %d!\n", i);
    return 0;
}

- One C statement can lead to multiple assembly instructions
- "mov" is a pretty common instruction
- It also has different forms
- Setting up function calls takes some work.
- Something is going on with the stack
- Return values go into a register (%rax, as it turns out)
Data Formats

- Because of its 16-bit origins, Intel uses "word" to mean 16-bits (two bytes)
- 32-bit words are referred to as "double words" ("l" suffix)
- 64-bit quantities are referred to as "quad words" ("q" suffix)
- This table shows the x86 primitive data types of C (Figure 3.1 in the textbook)

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

- Notice:
  - Pointers are 8-byte quad words
  - The suffixes will become important very soon
x86 CPUs have 16 *general purpose registers*, which store 64-bit values.

- Registers store integer data, and pointers
- The names begin with "r", but the naming is historical:
  - The original 16-bit registers were %ax, %bx, %cx, %dx, %si, %di, %bp, and %sp.
  - Each had a purpose, and were named as such.
  - When 32-bit x86 arrived, the register names expanded to 32-bits each, and changed to %eax, %ebx, etc.
  - When x86-64 arrived, the registers were again renamed to %rax, %rbx, etc., and expanded to 64-bits. Additionally, eight more registers were added, %r8 - %r15.

- The following page shows the registers, which have nested naming behavior.
- The least flexible register is %rsp, the "stack pointer", but the others are relatively flexible, and have multiple uses.
## The Integer Registers (part I)

<table>
<thead>
<tr>
<th>Register</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
<td></td>
<td></td>
<td>return value</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td></td>
<td></td>
<td>callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
<td></td>
<td></td>
<td>4th argument</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
<td></td>
<td>3rd argument</td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
<td></td>
<td></td>
<td>2nd argument</td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
<td></td>
<td></td>
<td>1st argument</td>
</tr>
</tbody>
</table>

- %rax: %eax, %ax, %al
- %rbx: %ebx, %bx, %bl
- %rcx: %ecx, %cx, %cl
- %rdx: %edx, %dx, %dl
- %rsi: %esi, %si, %sil
- %rdi: %edi, %di, %sil

- %rax: 63
- %rbx: 31
- %rcx: 15
- %rdx: 7
- %rsi
- %rdi
## The Integer Registers (part II)

<table>
<thead>
<tr>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bpl</td>
<td>callee saved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td>%spl</td>
<td>stack pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td>5th argument</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td>6th argument</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td>caller saved</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td>caller saved</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Integer Registers (part III)

<table>
<thead>
<tr>
<th>Register</th>
<th>First 32 Bits</th>
<th>Last 32 Bits</th>
<th>Storage Options</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w %r12b</td>
<td>callee saved</td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w %r13b</td>
<td>callee saved</td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w %r14b</td>
<td>callee saved</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w %r15b</td>
<td>callee saved</td>
<td></td>
</tr>
</tbody>
</table>

The last column designates the standard programming conventions — we will get to that later, but it denotes how registers manage the stack, passing function arguments, returning from function calls, and storing local and temporary data.
The Integer Registers are nested!

<table>
<thead>
<tr>
<th>$%rax$</th>
<th>$%eax$</th>
<th>$%ax$</th>
<th>$%al$</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>31</td>
<td>15</td>
<td>7</td>
<td>15</td>
</tr>
</tbody>
</table>

```
$ gdb hello
(gdb) b main
(gdb) run
(gdb) p/x $rax=0x445566778899aabb
$1 = 0x445566778899aabb
(gdb) p/x $eax
$2 = 0x8899aabb
(gdb) p/x $ax
$3 = 0xaabb
(gdb) p/x $al
$4 = 0xbb
(gdb) p/x $ah
$5 = 0xaa
```
## Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm$</td>
<td>$Imm$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>$R[r_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm$</td>
<td>$M[Imm]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_a)$</td>
<td>$M[R[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b)$</td>
<td>$M[Imm + R[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i)$</td>
<td>$M[R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i)$</td>
<td>$M[Imm + R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(*,r_i,s)$</td>
<td>$M[R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(*,r_i,s)$</td>
<td>$M[Imm + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i,s)$</td>
<td>$M[R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i,s)$</td>
<td>$M[Imm + R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Figure 3.3** *Operand forms.* Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor $s$ must be either 1, 2, 4, or 8.
Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm$</td>
<td>$Imm$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>$R[r_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm$</td>
<td>$M[Imm]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_a$)</td>
<td>$M[R[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b)$</td>
<td>$M[Imm + R[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b, r_i$)</td>
<td>$M[R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b, r_i)$</td>
<td>$M[Imm + R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>(s, $r_i$)</td>
<td>$M[R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(s, r_i)$</td>
<td>$M[Imm + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b, r_i, s$)</td>
<td>$M[R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b, r_i, s)$</td>
<td>$M[Imm + R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Immediate**: for constant values. Examples: $1$, $0x1A$, $-42$
### Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm$</td>
<td>$Imm$</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>$R[r_a]$</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm$</td>
<td>$M[Imm]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_a)$</td>
<td>$M[R[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b)$</td>
<td>$M[Imm + R[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i)$</td>
<td>$M[R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i)$</td>
<td>$M[Imm + R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(,r_i,s)$</td>
<td>$M[R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(,r_i,s)$</td>
<td>$M[Imm + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i,s)$</td>
<td>$M[R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i,s)$</td>
<td>$M[Imm + R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Register:** for constant values. Represents the value of the register. Examples: `%rax`, `%edx`, `%r8d`
Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>($Imm)</td>
<td>(Imm)</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>(r_a)</td>
<td>(R[r_a])</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>(Imm)</td>
<td>(M[Imm])</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>((r_a))</td>
<td>(M[R[r_a]])</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>(Imm(r_b))</td>
<td>(M[Imm + R[r_b]])</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>((r_b,r_i))</td>
<td>(M[R[r_b] + R[r_i]])</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>((.r_i,s))</td>
<td>(M[R[r_i] \cdot s])</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>(Imm(.r_i,s))</td>
<td>(M[Imm + R[r_i] \cdot s])</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>((r_b,.r_i,s))</td>
<td>(M[R[r_b] + R[r_i] \cdot s])</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>(Imm(r_b,.r_i,s))</td>
<td>(M[Imm + R[r_b] + R[r_i] \cdot s])</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Memory**: for accessing some memory location according to a *computed* address, often called the *effective address*. As seen above, there are many different *addressing modes* to allow different forms of memory references.
Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>$Imm$</td>
<td>$M[Imm]$</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_a)$</td>
<td>$M[R[r_a]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b)$</td>
<td>$M[Imm + R[r_b]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i)$</td>
<td>$M[R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i)$</td>
<td>$M[Imm + R[r_b] + R[r_i]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_i,s)$</td>
<td>$M[R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(,r_i,s)$</td>
<td>$M[Imm + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r_b,r_i,s)$</td>
<td>$M[R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b,r_i,s)$</td>
<td>$M[Imm + R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Most general form:** $Imm(r_b,r_i,s)$

This has four parts: an immediate offset, $Imm$, a base register, $r_b$, an index register, $r_i$, and a scale factor, $s$, which must be 1, 2, 4, or 8. The effective address is computed as: $Imm + R[r_b] + R[r_i] \cdot s$

Often, we see this when referencing elements in arrays.
Practice with Operand Forms

Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%rdx</td>
<td>0x3</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the table to the right showing the values for the indicated operands.

**Reminder:**

**Most general form:** $Imm(r_b, r_i, s) = Imm + R[r_b] + R[r_i] \times s$

Also: $260d = 0x104$
### Practice with Operand Forms

Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%rdx</td>
<td>0x3</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the table to the right showing the values for the indicated operands.

**Reminder:**

**Most general form:** \( \text{Imm}(r_b, r_i, s) \)

\[ \text{Imm} + R[r_b] + R[r_i] \times s \]

**Also:** \( 260 \text{d} = 0x104 \)
• Copying data from location to another is one of the most common instructions in assembly code.
• The x86 processors have a "Complex Instruction Set Architecture" (CISC), as opposed to some other processors—like the ARM that is most likely in your phone—called "Reduced Instruction Set Architecture" (RISC). The many ways to copy data is a hallmark of a CISC processor.
• We will discuss many different types of data movement instructions, starting with the mov instruction. The simple data movement instructions are as follows:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>S, D</td>
<td>Move</td>
</tr>
<tr>
<td>movb</td>
<td></td>
<td>Move byte</td>
</tr>
<tr>
<td>movw</td>
<td></td>
<td>Move word</td>
</tr>
<tr>
<td>movl</td>
<td></td>
<td>Move double word</td>
</tr>
<tr>
<td>movq</td>
<td></td>
<td>Move quad word</td>
</tr>
<tr>
<td>movabsq</td>
<td>I, R</td>
<td>Move absolute quad word</td>
</tr>
</tbody>
</table>
The `mov` instruction has a source and a destination, but only one can potentially be a memory location (you need two instructions to do that: first copy to a register from memory, then copy to memory from the register).

For most cases, the `mov` instruction only updates the specific register bytes or memory locations indicated by the destination operand.

The exception is for the `movl` instruction: if it has a register as a destination, it will also set the high order 4 bytes of the register to 0.

Examples:

1. `movl $0x4050, %eax`  Immediate–Register, 4 bytes
2. `movw %bp, %sp`  Register–Register, 2 bytes
3. `movb (%rdi, %rcx), %al`  Memory–Register, 1 byte
4. `movb $-17, (%rsp)`  Immediate–Memory, 1 byte
5. `movq %rax, -12(%rbp)`  Register–Memory, 8 bytes
The `movabsq` instruction is used when a 64-bit immediate (constant) value is needed in a register. The regular `movq` instruction can only take a 32-bit immediate value (because of the way the instruction is represented in memory).

The `movabsq` instruction can have a 64-bit immediate as a source, and only a register as a destination.

Example:

```
movabsq $0x0011223344556677, %rax
```
There are two `mov` instructions that can be used to copy a smaller source to a larger destination: `movz` and `movs`.

- `movz` fills the remaining bytes with zeros
- `movs` fills the remaining bytes by sign-extended the most significant bit in the source.

- The source must be from memory or a register, and the destination is a register.

- There are six ways to move a 1- or 2-byte size to a 2- or 4-byte size, for each case:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MOVZ</code></td>
<td>( S, R ) ( R \leftarrow \text{ZeroExtend}(S) )</td>
<td>Move with zero extension</td>
</tr>
<tr>
<td><code>movzbw</code></td>
<td></td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td><code>movzbl</code></td>
<td></td>
<td>Move zero-extended byte to double word</td>
</tr>
<tr>
<td><code>movzwl</code></td>
<td></td>
<td>Move zero-extended word to double word</td>
</tr>
<tr>
<td><code>movzwbq</code></td>
<td></td>
<td>Move zero-extended byte to quad word</td>
</tr>
<tr>
<td><code>movzwq</code></td>
<td></td>
<td>Move zero-extended word to quad word</td>
</tr>
</tbody>
</table>

- There isn't a 4-byte source to 8-byte destination, as it is already covered by the `movl` instruction with a register destination, which always populates the upper 4 bytes with 0s.
3 minute break
movz and movs

- movs fills the remaining bytes by sign-extending the most significant bit in the source.
- There is also a cltq instruction, which is a more compact encoding of movslq %eax,%rax

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVZ S, R</td>
<td>( R \leftarrow \text{SignExtend}(S) )</td>
<td>Move with sign extension</td>
</tr>
<tr>
<td>movsbw</td>
<td>Move sign-extended byte to word</td>
<td></td>
</tr>
<tr>
<td>movsbl</td>
<td>Move sign-extended byte to double word</td>
<td></td>
</tr>
<tr>
<td>movswl</td>
<td>Move sign-extended word to double word</td>
<td></td>
</tr>
<tr>
<td>movsbq</td>
<td>Move sign-extended byte to quad word</td>
<td></td>
</tr>
<tr>
<td>movswq</td>
<td>Move sign-extended word to quad word</td>
<td></td>
</tr>
<tr>
<td>movslq</td>
<td>Move sign-extended double word to quad word</td>
<td></td>
</tr>
<tr>
<td>cltq</td>
<td>( %rax \leftarrow \text{SignExtend}(%eax) )</td>
<td>Sign-extend %eax to %rax</td>
</tr>
</tbody>
</table>
Practice with \texttt{mov}

- For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (e.g., \texttt{mov} can be \texttt{movb}, \texttt{movw}, \texttt{movl}, \texttt{movq})

\begin{verbatim}
\texttt{mov___ \%eax, (%rsp)}
\texttt{mov___ \%(rax), \%dx}
\texttt{mov___ $0xFF, \%bl}
\texttt{mov___ (@rsp,@rdx,4), \%dl}
\texttt{mov___ (@rdx), \%rax}
\texttt{mov___ \%dx, (@rax)}
\end{verbatim}
Practice with \textbf{mov}

- For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (e.g., \texttt{mov} can be \texttt{movb, movw, movl, movq})

\begin{verbatim}
movl %eax, (%rsp)
movw %(%rax), %dx
movb $0xFF, %bl
movb (%%rsp, %rdx,4), %dl
movq (%rdx), %rax
movw %dx, (%rax)
\end{verbatim}
Practice with \texttt{mov}

- Each of the following lines of code generate an error message if we use the assembler. Explain what is wrong with each line:

\begin{verbatim}
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax),4(%rsp)
movb %al, %sl
movq %rax,$0x123
movl %eax,%dx
movb %si, 8(%rbp)
\end{verbatim}
• Each of the following lines of code generate an error message if we use the assembler. Explain what is wrong with each line:

```asm
movb $0xF, (%ebx)  Cannot use ebx as address register.
movl %rax, (%rsp)  Mismatch between instruction suffix and register ID
movw (%rax),4(%rsp)  Cannot have both source and destination be memory registers
movb %al, %sl  No register named %sl
movq %rax,$0x123  Cannot have immediate destination
movl %eax,%dx  Destination operand incorrect size
movb %si, 8(%rbp)  Mismatch between instruction suffix and register ID (%si is a word)
```
More C to Assembly

#include<stdio.h>
#include<stdlib.h>

long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}

int main()
{
    long x = 1000;
    long y = 42;

    printf("Before exchange: x:%lu, y:%lu\n",x,y);
    y = exchange(&x, y);
    printf("After exchange: x:%lu, y:%lu\n",x,y);
    return 0;
}

Compile line:
gcc -g -Og -std=gnu99 -Wall $warnflags exchange.c -o exchange

Compiler Explorer:
https://gcc.godbolt.org

Use compiler:
x86-64 gcc 4.8.5

Flags:
-g -Og -std=gnu99
```c
#include<stdio.h>
#include<stdlib.h>

void count_from_offset(int offset)
{
    for (int i=0; i < 10; i++) {
        printf("Count: %d\n", i+offset);
    }
}

int main()
{
    count_from_offset(5);
    return 0;
}
```

Slightly different compile line:
gcc -g -Og -std=gnu99 -Wall $warnflags exchange.c -o exchange
The **lea** instruction

- The **lea** instruction is related to the **mov** instruction. It has the form of an instruction that reads from memory to a register, but it *does not reference memory at all*.
- It's first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination.
- You can think of it as the "&" operator in C — it retrieves the address of a memory location:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaq S,D</td>
<td>D ← &amp;S</td>
<td>Load effective address</td>
</tr>
</tbody>
</table>

Examples: if `%rax` holds value `x` and `%rcx` holds value `y`:

- `leaq 6(%rax), %rdx` : `%rdx` now holds `x + 6`
- `leaq (%rax,%rcx), %rdx` : `%rdx` now holds `x + y`
- `leaq (%rax,%rcx,4), %rdx` : `%rdx` now holds `x + 4*y`
- `leaq 7(%rax,%rax,8), %rdx` : `%rdx` now holds `7 + 9x`
- `leaq 0xA(,%rcx,4), %rdx` : `%rdx` now holds `10 + 4y`
- `leaq 9(%rax,%rcx,2), %rdx` : `%rdx` now holds `9 + x + 2y`
Pushing and Popping from the Stack

• As we have seen from stack-based memory allocation in C, the stack is an important part of our program, and assembly language has two built-in operations to use the stack.
• Just like the stack ADT, they have a first-in, first-out discipline.
• By convention, we draw stacks upside down, and the stack "grows" downward.
Pushing and Popping from the Stack

- The push and pop operations write and read from the stack, and they also modify the stack pointer, `%rsp`:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pushq</code></td>
<td>S</td>
<td>R[%rsp] ← R[%rsp]-8; M[R[%rsp]] ← S</td>
</tr>
<tr>
<td><code>popq</code></td>
<td>D</td>
<td>D ← M[R[%rsp]]; R[%rsp] ← R[%rsp]+8</td>
</tr>
</tbody>
</table>

Stack "bottom"

Stack "top"

Increasing address

0x108
Pushing and Popping from the Stack

• Example:

<table>
<thead>
<tr>
<th>Initially</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
</tr>
<tr>
<td>%rdx</td>
</tr>
<tr>
<td>%rsp</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing address

Stack "top"
Pushing and Popping from the Stack

Example:

Initially

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

pushq %rax

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x100</td>
</tr>
</tbody>
</table>

Increasing address.

Stack "bottom"

Stack "top"
Pushing and Popping from the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq %rax</th>
<th>popq %rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
<td>%rax</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
<td>%rdx</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
<td>%rsp</td>
</tr>
</tbody>
</table>

Initially:
- %rax: 0x123
- %rdx: 0
- %rsp: 0x108

After `pushq %rax`:
- %rax: 0x123
- %rdx: 0
- %rsp: 0x100

After `popq %rdx`:
- %rax: 0x123
- %rdx: 0x123
- %rsp: 0x108

Increasing address:
- Stack "bottom"
- Stack "top"
Pushing and Popping from the Stack

• As you can tell, pushing a quad word onto the stack involves first decrementing the stack pointer by 8, and then writing the value at the new top-of-stack address.

• Therefore, the behavior of the instruction `pushq %rbp` is equivalent to the pair of instructions:
  
  \[
  \begin{align*}
  \text{subq } & \, 8, \, %rsp \\
  \text{movq } & \, %rbp, (\%rsp)
  \end{align*}
  \]  
  (\text{subq} \text{ is a subtraction, and this decrements the stack pointer})
  
  (\text{Store } \%rbp \text{ on the stack})

• The behavior of the instruction `popq %rax` is equivalent to the pair of instructions:

  \[
  \begin{align*}
  \text{movq } & \, (\%rsp), \, %rax \\
  \text{addq } & \, 8, \, %rsp
  \end{align*}
  \]  
  (Read \%rax from the stack)
  
  (Increment the stack pointer)
References and Advanced Reading

• References:
  • Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  • CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  • gdbtui: https://beej.us/guide/bgdb/
  • More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  • Compiler explorer: https://gcc.godbolt.org

• Advanced Reading:
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64
The aside on page 184 of the textbook is interesting: you should understand how data movement changes the destination register:

**Aside** Understanding how data movement changes a destination register

As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register. This distinction is illustrated by the following code sequence:

```bash
1. movabsq $0x001223344556677, %rax  \( %rax = 0011223344556677 \)
2. movb  $-1, %al               \( %rax = 0011223344455667 \)
3. movw  $-1, %ax               \( %rax = 0011223344455667 \)
4. movl  $-1, %eax              \( %rax = 00000000FFFFFEFF \)
5. movq  $-1, %rax              \( %rax = FFFFFFFFEFFFFFEFF \)
```

In the following discussion, we use hexadecimal notation. In the example, the instruction on line 1 initializes register %rax to the pattern 00112233444556677. The remaining instructions have immediate value \(-1\) as their source values. Recall that the hexadecimal representation of \(-1\) is of the form FF· · · F, where the number of F’s is twice the number of bytes in the representation. The `movb` instruction (line 2) therefore sets the low-order byte of %rax to FF, while the `movw` instruction (line 3) sets the low-order 2 bytes to FFFF, with the remaining bytes unchanged. The `movl` instruction (line 4) sets the low-order 4 bytes to FFFFFFFF, but it also sets the high-order 4 bytes to 00000000. Finally, the `movq` instruction (line 5) sets the complete register to FFFFFFFFEFFFFFEFF.