Today

• Assembly basics (cont)

Assignments

• HW4 due Thurs
• Quiz on Thurs
REVIEW: Overview of gdb basics

Consider the following simple assembly program (in a file example1.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 example1.s
$ ld example1.o -o example1
$ ./example1
```
Start gdb (GNU Debugger) to “trace” the program ...

```bash
$ gdb example1
(gdb) break _start
(gdb) run
(gdb) layout asm
(gdb) layout reg
```

These commands:

- start gdb
- create a breakpoint at _start address
- run the program (to _start
- open the code viewer
- open the register viewer

... gdb lect10
... break _start
... run
... layout asm
... 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

![Register 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>%dil</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

Basic syntax:

```
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
Q: What happens to rest of register when moving a “small” value into it?

```
movw $512, %ax # 0000 0010 0000 0000 (512)
movb $42, %al # 0000 0010 0010 1010 (554)
```

- the high order bits don’t change!

The `movz{s_1}^{s_2}` instruction helps with this ...

- zero extended `mov` instruction
- similar to coercion (e.g., converting int value to long value)

```
movw $512, %ax # 0000 0010 0000 0000 (512)
movb $42, %al # 0000 0010 0010 1010 (554)
movzbw %al, %ax # 0000 0000 0010 1010 (42)
```
The **add** instruction

Basic syntax:

```
add s src, dst                          ... dst = src + dst
```

- `s` is the size of data being moved
- `src` is either a literal, memory loc, or register
- `dst` is either a memory loc or register

* example:

```
movw $42, %ax  # ax = 42
addw $15, %ax  # ax = 15 + ax
```

The **sub** instruction

Basic syntax:

```
sub s src, dst                          ... dst = dst - src
```

- `s` is the size of data being moved
- `src` is either a literal, memory loc, or register
- `dst` is either a memory loc or register

* example:

```
movl $42, %eax  # eax = 42
subl $15, %eax  # eax = eax - 15
```
The \texttt{neg} instruction: Two's complement negation

Basic syntax:

\begin{verbatim}
  negs dst ... dst = 0 - dst
\end{verbatim}

– \texttt{s} is the size of data being negated
– \texttt{dst} is either a memory loc or register

• example:

\begin{verbatim}
  movl $42, %eax # eax = 42
  negl %eax # eax = -42
\end{verbatim}

Note that this is a single-operand instruction

• in x86, the size of each instruction can differ
• depends in part on literal sizes as well as number of operands
• thus \texttt{movb $42 \%al} will be smaller than \texttt{movw $42 \%ax}
The inc / dec instructions

Basic syntax:

\[
\begin{align*}
\text{inc} & \quad s \quad \text{dst} \quad \Rightarrow \quad \text{dst} \rightarrow \text{dst} + s \\
\text{dec} & \quad s \quad \text{dst} \quad \Rightarrow \quad \text{dst} \rightarrow \text{dst} - s
\end{align*}
\]

- \( s \) is the size of data being negated
- \( \text{dst} \) is either a memory loc or register

• example:

\[
\begin{align*}
\text{movq} & \quad $42, %rax \quad \# \text{rax} = 42 \\
\text{incq} & \quad %rax \quad \# \text{rax} = 43 \\
\text{decq} & \quad %rax \quad \# \text{rax} = 42
\end{align*}
\]