Meet & Greet!

Come hang out with your TAs and Fellow Students (& eat free insomnia cookies)

When: TODAY!! 5-6 pm
Where: 3rd Floor Atrium, CIT
Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
The early computers of the x86 family had 16-bit words, starting with the 386, they supported 32-bit words.
\[2^{64}\]

- \[2^{32}\] used to be considered a large number
  - one couldn’t afford \[2^{32}\] bytes of memory, so no problem with that as an upper bound
- Intel (and others) saw need for machines with 64-bit addresses
  - devised IA64 architecture with HP
    » became known as Itanium
    » very different from x86
- AMD also saw such a need
  - developed 64-bit extension to x86, called x86-64
- Itanium flopped
- x86-64 dominated
- Intel, reluctantly, adopted x86-64

\[2^{32} = 4\] gigabytes.
\[2^{64} = 16\] exbibytes

All SunLab computers are x86-64.
Data Types on IA32 and x86-64

• “Integer” data of 1, 2, or 4 bytes (plus 8 bytes on x86-64)
  – data values
    » whether signed or unsigned depends on interpretation
  – addresses (untyped pointers)

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

• No aggregate types such as arrays or structures
  – just contiguously allocated bytes in memory
Most instructions come in three (on IA32) or four (on x86-64) forms, one for each possible operand size.

- Rather than \texttt{mov} ...
  - \texttt{movb}
  - \texttt{movs}
  - \texttt{movl}
  - \texttt{movq} (x86-64 only)
### General-Purpose Registers (IA32)

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
<td></td>
</tr>
</tbody>
</table>

#### Origin
- mostly obsolete

#### Use
- accumulate
- counter
- data
- base
- source
- index
- destination
- index
- stack
- pointer
- base
- pointer

#### 16-bit virtual registers
(backwards compatibility)
### Moving Data: IA32

- **Moving data**
  
  ```
  movl source, dest
  ```

- **Operand types**
  
  - **Immediate**: constant integer data
    ```
    example: $0x400, $-533
    ```
  
  - **Register**: one of 8 integer registers
    ```
    example: %eax, %edx
    ```
  
  - **Memory**: 4 consecutive bytes of memory at address given by register(s)
    ```
    simplest example: (%eax)
    ```
    ```
    various other “address modes”
    ```

Note that though `esp` and `ebp` have special uses, they may also be used in both source and destination operands.

Note that some assemblers (in particular, those of Intel and Microsoft) place the operands in the opposite order. Thus the example of the slide would be “addl %eax,8(%ebp)”. The order we use is that used by gcc, known as the “AT&T syntax” because it was used in the original Unix assemblers, written at Bell Labs, then part of AT&T.
## movl Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src, Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg</td>
<td>movl $0x4,%eax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movl $-147,(%eax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl (%eax),%edx</td>
<td>temp = *p;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movl %eax,%edx</td>
<td>temp2 = temp1;</td>
</tr>
</tbody>
</table>

*Cannot (normally) do memory-memory transfer with a single instruction*
**Simple Memory Addressing Modes**

- **Normal (R) Mem[Reg[R]]**
  - register R specifies memory address
  
  ```
  movl (%ecx),%eax
  ```

- **Displacement D(R) Mem[Reg[R]+D]**
  - register R specifies start of memory region
  - constant displacement D specifies offset

  ```
  movl 8(%ebp),%edx
  ```

Supplied by CMU.

If one thinks of there being an array of registers, then “Reg[R]” selects register “R” from this array.
We discuss the “set up” and “finish” in a subsequent lecture. They have to do with facilitating the calling of functions.
void swap(int *xp, int *yp) {
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Register | Value
---|---
%edx | xp
%ecx | yp
%ebx | t0
%eax | t1

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

Supplied by CMU.
Understanding Swap

%eax
%edx
%ecx
%ebx
%esi
%edi
%esp
%ebp 0x104

movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

Address
123 0x124
456 0x120
0x11c 0x118
0x114 0x110
0x120 0x10c
0x124 0x108
Rtn adr 0x104
0x104 0x100

Supplied by CMU.
Understanding Swap

%eax
%edx 0x124
%ecx
%ebx
%esi
%edi
%esp
%ebp 0x104

movl $%ebp, %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

Supplied by CMU.
Understanding Swap

| %eax | | Address |
|-------|-----------|
|       |           | 0x124   |
| %edx  | 0x124     |
| %ecx  | 0x120     |
| %ebx  |           |
| %esi  |           |
| %edi  |           |
| %esp  |           |
| %ebp  | 0x104     |

<table>
<thead>
<tr>
<th>Offset</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>0x120</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>%ebp</td>
<td>0</td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td>-4</td>
<td>0x100</td>
</tr>
</tbody>
</table>

movl $(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx # ecx = yp
movl (%edx), %ebx  # ebx = *xp (t0)
movl (%ecx), %eax  # eax = *yp (t1)
movl %eax, (%edx)  # *xp = t1
movl %ebx, (%ecx)  # *yp = t0

Supplied by CMU.
Supplied by CMU.
### Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
</tr>
<tr>
<td>%edi</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>YP</th>
<th>12</th>
<th>0x120</th>
<th>0x110</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp</td>
<td>8</td>
<td>0x12c</td>
<td>0x10c</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Rtn adr</td>
<td>0x108</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

```assembly
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx    # ebx = *xp (t0)
movl (%ecx), %eax    # eax = *yp (t1)
movl %eax, (%edx)    # *xp = t1
movl %ebx, (%ecx)    # *yp = t0
```
Supplied by CMU.
# Understanding Swap

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>456</td>
<td>0x124</td>
</tr>
<tr>
<td>%edx</td>
<td>0x124</td>
<td>0x110</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x120</td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebx</td>
<td>123</td>
<td>0x108</td>
</tr>
<tr>
<td>%esi</td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>%edi</td>
<td></td>
<td>0x100</td>
</tr>
<tr>
<td>%esp</td>
<td></td>
<td>0x104</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
<td>0x100</td>
</tr>
</tbody>
</table>

```asm
movl 8(%ebp), %edx  # edx = xp
movl 12(%ebp), %ecx  # ecx = yp
movl (%edx), %ebx   # ebx = *xp (t0)
movl (%ecx), %eax   # eax = *yp (t1)
movl %eax, (%edx)   # *xp = t1
movl %ebx, (%ecx)   # *yp = t0
```

Supplied by CMU.
Quiz 1

```
movl -4(%ebp), %eax
movl (%eax), %eax
movl (%eax), %eax
movl %eax, -8(%ebp)
```

Which C statements best describe the assembler code?

```
// a          // b          // c          // d
int x;       int *x;      int **x;      int ***x;
int y;       int y;       int y;       int y;
y = x;       y = *x;      y = **x;      y = ***x;
```
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”
– Rb: base register: any of 8 integer registers
– Ri: index register: any, except for %esp
  » unlikely you’d use %ebp either
– S: scale: 1, 2, 4, or 8

• Special cases

\begin{align*}
(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]] \\
D & \quad \text{Mem}[D]
\end{align*}

Supplied by CMU.
# Address-Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0x8(%edx)$</td>
<td>$0xf000 + 0x8$</td>
<td>$0xf008$</td>
</tr>
<tr>
<td>$(%edx,%ecx)$</td>
<td>$0xf000 + 0x1000$</td>
<td>$0xf100$</td>
</tr>
<tr>
<td>$(%edx,%ecx,4)$</td>
<td>$0xf000 + 4*0x1000$</td>
<td>$0xf400$</td>
</tr>
<tr>
<td>$0x80(,%edx,2)$</td>
<td>$2*0xf000 + 0x80$</td>
<td>$0x1e080$</td>
</tr>
</tbody>
</table>

Supplied by CMU.
Supplied by CMU.

Note that a function returns a value by putting it in %eax.
## Quiz 2

What value ends up in %ecx?

```assembly
movl $1000, %eax
movl $1, %ebx
movl 2(%eax, %ebx, 4), %ecx
```

| 1009: | 0x09 |
| 1008: | 0x08 |
| 1007: | 0x07 |
| 1006: | 0x06 |
| 1005: | 0x05 |
| 1004: | 0x04 |
| 1003: | 0x03 |
| 1002: | 0x02 |
| 1001: | 0x01 |
| %eax → 1000: | 0x00 |

a) 0x02030405  
b) 0x05040302  
c) 0x06070809  
d) 0x09080706

**Hint:**
Supplied by CMU.

Note that %ebp/%rbp may be used as a base register as on IA32, but they don’t have to be used that way. This will become clearer when we explore how the runtime stack is accessed. The convention on Linux is for the first 6 arguments of a function to be in registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9. The return value of a function is put in %rax.

Note also that each register, in addition to having a 32-bit version, also has an 8-bit (one-byte) version. For the numbered registers, it’s, for example, %r10b. For the other registers it’s the same as for IA32.
On x86-64, for instructions with 32-bit (long) operands that produce 32-bit results going into a register, the register must be a 32-bit register; the higher-order 32 bits are filled with zeroes.
Note that using single-byte versions of registers has different behavior than using with 4-byte versions of registers. Putting data into the latter using mov causes the upper bytes to be zeroed. But with the byte versions, putting data into them does not affect the upper bytes.
Supplied by CMU.

Note that for the IA32 architecture, arguments are passed on the stack.
Note that no more than six arguments can be passed in registers. If there are more than six arguments (which is unusual), then remaining arguments are passed on the stack, and referenced via %rsp.
64-bit code for long int swap

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

- **64-bit data**
  - data held in registers %rax and %rdx
  - `movq` operation
    » “q” stands for quad-word
Supplied by CMU.

Note that normally one does not ask gcc to produce assembler code, but instead it compiles C code directly into machine code (producing an object file). Note also that the gcc command actually invokes a script; the compiler (also known as gcc) compiles code into either assembler code or machine code; if necessary, the assembler (as) assembles assembler code into object code. The linker (ld) links together multiple object files (containing object code) into an executable program.
Example

```c
int sum(int a, int b) {
    return (a+b);
}
```
Object Code

Code for `sum`:
0x401040 `<sum>`:
  0x55
  0x89
  0xe5
  0x8b
  0x45
  0x0c
  0x03
  0x45
  0x08
  0x5d
  0xc3

- Total of 11 bytes
- Each instruction: 1, 2, or 3 bytes
- Starts at address 0x401040

- **Assembler**
  - translates `.s` into `.o`
  - binary encoding of each instruction
  - nearly-complete image of executable code
  - missing linkages between code in different files

- **Linker**
  - resolves references between files
  - combines with static run-time libraries
    » e.g., `code for printf`
  - some libraries are *dynamically linked*
    » linking occurs when program begins execution

Supplied by CMU.
Disassembling Object Code

Disassembled

```
080483c4 <sum>:
080483c4: 55 push %ebp
080483c5: 89 e5 mov %esp,%ebp
080483c7: 8b 45 0c mov 0xc(%ebp),%eax
080483ca: 03 45 08 add 0x0(%ebp),%eax
080483cd: 5d pop %ebp
080483ce: c3 ret
```

- **Disassembler**
  - `objdump -d <file>`
  - useful tool for examining object code
  - analyzes bit pattern of series of instructions
  - produces approximate rendition of assembly code
  - can be run on either executable or object (.o) file

Supplied by CMU.
**Alternate Disassembly**

<table>
<thead>
<tr>
<th>Object</th>
<th>Disassembled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401040: 0x55 0x89 0xe5 0x8b 0x45 0x0c 0x03 0x45 0x08 0x5d 0xc3</td>
<td>Dump of assembler code for function sum: 0x080483c4 &lt;sum+0&gt;: push %ebp 0x080483c5 &lt;sum+1&gt;: mov %esp,%ebp 0x080483c7 &lt;sum+3&gt;: mov 0xc(%ebp),%eax 0x080483ca &lt;sum+6&gt;: add 0x8(%ebp),%eax 0x080483cd &lt;sum+9&gt;: pop %ebp 0x080483ce &lt;sum+10&gt;: ret</td>
</tr>
</tbody>
</table>

- Within gdb debugger
  - `gdb <file>`
  - `disassemble sum`
    - `disassemble procedure`
    - `x/11xb sum`
  - `examine the 11 bytes starting at sum`
How Many Instructions are There?

- We cover ~30
- Implemented by Intel:
  - 80 in original 8086 architecture
  - 7 added with 80186
  - 17 added with 80286
  - 33 added with 386
  - 6 added with 486
  - 6 added with Pentium
  - 1 added with Pentium MMX
  - 4 added with Pentium Pro
  - 8 added with SSE
  - 8 added with SSE2
  - 2 added with SSE3
  - 14 added with x86-64
  - 10 added with VT-x
  - 2 added with SSE4a
- Total: 198
- Doesn’t count:
  - floating-point instructions
    - ~90
  - SIMD instructions
    - lots
  - AMD-added instructions
  - undocumented instructions

The source for this is http://en.wikipedia.org/wiki/X86_instruction Listings, viewed on 6/20/2017, which comes with the caveat that it may be out of date.
**Some Arithmetic Operations**

- **Two-operand instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>(Dest = Dest + Src)</td>
</tr>
<tr>
<td>subl</td>
<td>(Dest = Dest - Src)</td>
</tr>
<tr>
<td>imull</td>
<td>(Dest = Dest \times Src)</td>
</tr>
<tr>
<td>sall</td>
<td>(Dest = Dest \ll Src)</td>
</tr>
<tr>
<td>sarl</td>
<td>(Dest = Dest \gg Src)</td>
</tr>
<tr>
<td>shr1</td>
<td>(Dest = Dest \gg Src)</td>
</tr>
<tr>
<td>xorl</td>
<td>(Dest = Dest \oplus Src)</td>
</tr>
<tr>
<td>andl</td>
<td>(Dest = Dest &amp; Src)</td>
</tr>
<tr>
<td>orl</td>
<td>(Dest = Dest \midim Or \ Src)</td>
</tr>
</tbody>
</table>

  - watch out for argument order!
  - no distinction between signed and unsigned int (why?)

Note that for shift instructions, the Src operand (which is the size of the shift) must either be a immediate operand or be a designator for a one-byte register (e.g., \%cl – see the slide on general-purpose registers for IA32).
Some Arithmetic Operations

- **One-operand Instructions**
  
  incl \( Dest \) = \( Dest + 1 \)
  
  decl \( Dest \) = \( Dest - 1 \)
  
  negl \( Dest \) = \( -Dest \)
  
  notl \( Dest \) = \( \sim Dest \)

- **See book for more instructions**

Supplied by CMU.
Arithmetic Expression Example

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```
arith:
    leal (%rdi,%rsi), %eax
    addl %edx, %eax
    leal (%rsi,%rsi,2), %edx
    sall $4, %edx
    leal 4(%rdi,%rdx), %ecx
    imull %ecx, %eax
    ret
```

Supplied by CMU, but converted to x86-64.
Understanding arith

```c
int arith(int x, int y, int z)
{
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

```assembly
leal (%rdi,%rsi), %eax
addl %edx, %eax
leal (%rsi,%rsi,2), %edx
sall $4, %edx
leal 4(%rdi,%rdx), %ecx
imull %ecx, %eax
ret
```

Supplied by CMU, but converted to x86-64.
By convention, the first three arguments to a procedure are placed in registers rdi, rsi, and rdx, respectively. Note that, also by convention, procedures put their return values in register eax/rax.
Supplied by CMU, but converted to x86-64.

```
int arith(int x, int y, int z) {
    int t1 = x+y;
    int t2 = z+t1;
    int t3 = x+4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

- Instructions in different order from C code
- Some expressions might require multiple instructions
- Some instructions might cover multiple expressions

```
leal (%rdi,%rsi), %eax  # eax = x+y   (t1)
addl %edx, %eax        # eax = t1+z   (t2)
leal (%rsi,%rsi,2), %edx  # edx = 3*y   (t4)
sall $4, %edx           # edx = t4*16   (t4)
leal 4(%rdi,%rdx), %ecx  # ecx = x+4+t4 (t5)
imull %ecx, %eax  # eax *= t5  (rval)
ret
```
Another Example

```c
int logical(int x, int y)
{
    int t1 = x ^ y;
    int t2 = t1 >> 17;
    int mask = (1<<13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

\[ 2^{13} = 8192, 2^{13} - 7 = 8185 \]

- xorl %esi, %edi  # edi = x ^ y  (t1)
- sarl $17, %edi   # edi = t1 >> 17  (t2)
- movl %edi, %eax  # eax = edi
- andl $8185, %eax # eax = t2 & mask (rval)

Supplied by CMU, but converted to x86-64.
Quiz 3

- What is the final value in %ecx?

    xorl %ecx, %ecx
    incl %ecx
    sall %cl, %ecx  # %cl is the low byte of %ecx
    addl %ecx, %ecx

da) 2  
b) 4  
c) 8  
d) indeterminate