CPUID

- CPUID example from Blum, Professional Assembly Language. Programmer to Programmer.
- Very good
- maybe little bit dated
- Available in PB and PDF

CPUID instruction

- Introduced in the early 90s
  - late 486s, Pentiums
- Why?
  - find out what chip type, know feature set
- Controversy?
  - Processor Serial Number option. Disabled.

Chapter 3 part 2

Getting at things on the chip you can’t easily reach from C
CPUID2.S

assembling, linking, running

• our normal way:
  – as –o fname.o cpuid2.s
  – ld –o fname fname.o

• when we’re using C library functions:
  – as –o fflag.o cpuid2.s
  – ld --dynamic-linker /lib/ld-linux.so.2 -lc –o fflag fflag.o

CPUID.S

calling a function

• push args onto the stack in reverse order
• call fflag
• return value will be in %eax
## setting a breakpoint

- simple way: address + offset
  - e.g., `*_start`
  - `*_start+1`
- GDB Bug: misses breakpoint at `*_start`
- Workaround:
  - insert nop

## other useful GDB commands

- `p $regname`
- `info locals`
- `info frame`
- `info registers`

---

## Where do I learn about CPUID?

- or any other operation?
- Intel IA-32, Intel64 manuals are available online for free from Intel
- Volume 2. Instruction set reference
  - details on every instruction, including CPUID

---

## Aside: running assembly under GDB

- Recall that when we compiled C programs to run under GDB, we used `-g` switch
- For assembly programs, we use `-gstabs`
How to do in assembly things you commonly do in C

some things that you know in C

• assignment
• simple arithmetic operations
• functions
• conditionals
• loops

examining the stack with GDB

• info registers
• info frame
• p $esp
• p *(int*)$esp
  – Just what it looks like
  • take what’s in %esp
  • treat it as an int*
  • print what it points to
• p *(int*)($esp+4)
  – if you forget the ( ), you’ll add 4 to *(int*)$esp

• x/Nuf address
  – N number
  – u units
  – f format
• example
  – x/20bx $esp

disassembling in GDB

• disas – disassembles current function
• disas sum – disassemble the sum function
• disas addr – disassemble function at addr
• disas $ad_1$ $ad_2$ – disas between addr $ad_1$, $ad_2$
int x, y;
    movl %ebx, %eax
...
x=y;

---

int x, y;
    movl %ebx, %eax
...
x=y;

*remember: in GAS this is like %eax=%ebx, not vice versa*

---

some things that you know in C

- assignment
- simple arithmetic operations
- functions
- conditionals
- loops

int x, y;
...
x=y;

*I specifies size*

mov{b,w,l,q}
assignment with different widths

• In C, if we have:
  ```c
  int x = 0x01234567;
  char y = 0xFF;
  ...
  x=y;
  ```

• what’s the value of x after the assignment?

What happens?

• What do we expect to find in %eax?
  ```asm
  movl $0x7FFFFFFFFF, %eax
  movb $0x6, %al
  ```

• Tempted to think 0x00000006

more on sizes

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>GAS suffix, e.g., for movz instruction</th>
<th>x86-32 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>unsigned</td>
<td>double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>unsigned long</td>
<td>double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>char *</td>
<td>double word</td>
<td>l</td>
<td>4</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>
<tr>
<td>long double</td>
<td>extended precision</td>
<td>t</td>
<td>10/12</td>
</tr>
</tbody>
</table>

with simple constants

in C
```c
  x=35;
```  

GAS
```asm
  movl $35, %eax
```  
in assembly, $35 is called an immediate value
example. what do we get?

```assembly
1 .section .rodata
2 print_str:
3 .string "x=0x%x\n"
4 .text
5 .globl main
6 .type main, @function
7 main:
8 pushl %ebp
9 movl %esp, %ebp
10 movl $0x01234567, %eax
11 movl $print_str, %ecx
12 pushl %eax
13 pushl %ecx
14 call printf
15 addl $8, %esp
16 movl $0, %eax
17 leave
18 ret
```

prints 0x1234567

---

movb, movzbl, movsbl

- movb – moves a byte to destination
- What if we want to move 1 byte to 32 bit reg?
  - movzbl –
    - expand to 32 bits
    - fill in the LHS with 24 0s
  - movsbl
    - expand to 32 bits
    - fill in LHS with 24 copies of the ms bit of src operand
- Think back to C:
  - movzbl does zero extension
  - movsbl does sign extension

---

What happens?

- What do we expect to find in %eax?
  
  ```assembly
  1 movl $0x7FFFFFFF, %eax
  2 movb $0x6, %al
  ```

- Tempted to think 0x00000006, but we get 2,147,483,398 or 0x7FFFFFF06

---

example. what do we get?

```assembly
1 .section .rodata
2 print_str:
3 .string "x=0x%x\n"
4 .text
5 .globl main
6 .type main, @function
7 main:
8 pushl %ebp
9 movl %esp, %ebp
10 movl $0x01234567, %eax
11 movl $print_str, %ecx
12 pushl %eax
13 pushl %ecx
14 call printf
15 addl $8, %esp
16 movl $0, %eax
17 leave
18 ret
```

prints 0x12345FF
C pointers

- suppose that register %eax contains 1000
  - movl $500, %eax
    - in register %eax replaces 1000 with 500
- movl $500, (%eax)
  - puts the value 500 into memory address 1000

- think of ( ) around registers to be something like the C dereference operator *

assignments: moving data

- we can move things from:
  - “immediate” to anything
  - memory to a register
  - register to memory

```
1 .section .rodata
2 print_str: .string "x=0x%x\n"
3 .text
4 .globl main
5 .type main, @function
6 main:
7   pushl %ebp
8   movl %esp, %ebp
9   movl $0x01234567, %eax
10  movb $0xFF, %dl
11  movzbl %dl, %eax
12  movl $print_str, %ecx
13  pushl %eax
14  pushl %ecx
15  call printf
16  addl $8, %esp
17  movl $0, %eax
18  leave
19  ret
20 .size main, .-main
```

```
1 .section .rodata
2 print_str: .string "x=0x%x\n"
3 .text
4 .globl main
5 .type main, @function
6 main:
7   pushl %ebp
8   movl %esp, %ebp
9   movl $0x01234567, %eax
10  movb $0xFF, %dl
11  movzbl %dl, %eax
12  movl $print_str, %ecx
13  pushl %eax
14  pushl %ecx
15  call printf
16  addl $8, %esp
17  movl $0, %eax
18  leave
19  ret
20 .size main, .-main
```
C pointer arithmetic

- given:
  - int *ip;
  - char *cp = (char*)ip;
- if ip has the address 1000:

<table>
<thead>
<tr>
<th>var</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>1000</td>
</tr>
<tr>
<td>cp</td>
<td>1000</td>
</tr>
<tr>
<td>cp+1</td>
<td>1001</td>
</tr>
<tr>
<td>ip+1</td>
<td>1004</td>
</tr>
<tr>
<td>cp+x</td>
<td>1000+x</td>
</tr>
<tr>
<td>ip+x</td>
<td>1000+x*sizeof(int)</td>
</tr>
</tbody>
</table>

assembly pointer arithmetic

- if %eax contains 1000
  - movl $555, %eax
    - replaces 1000 with the number 555.
  - movl $555, (%eax)
    - puts the number 555 into address 1000
  - movl $555, 4(%eax)
    - puts the number 555 into address 1004
- in this case, 4 is our displacement or offset

moving data. examples.

<table>
<thead>
<tr>
<th>src</th>
<th>dst</th>
<th>assembly example</th>
<th>C equiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm</td>
<td>reg</td>
<td>movl $0x52, %eax</td>
<td>x=0x52;</td>
</tr>
<tr>
<td>imm</td>
<td>mem</td>
<td>movl $0x52, (%eax)</td>
<td>*p=0x52;</td>
</tr>
<tr>
<td>reg</td>
<td>reg</td>
<td>movl %ebx, %eax</td>
<td>a=b;</td>
</tr>
<tr>
<td>reg</td>
<td>mem</td>
<td>movl %eax, (%ebx)</td>
<td>*p=x;</td>
</tr>
<tr>
<td>mem</td>
<td>reg</td>
<td>movl (%eax), %ebx</td>
<td>x=*p;</td>
</tr>
<tr>
<td>mem</td>
<td>mem</td>
<td>can’t be done</td>
<td>*p=*q;</td>
</tr>
</tbody>
</table>

C pointer arithmetic

- given:
  - int *ip;
  - char *cp = (char*)ip;
- if ip has the address 1000, what is:
  - ip+1
  - cp+1
**Understanding Swap**

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

**Disassembly**

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx  # ecx = yp
    movl 8(%ebp),%edx  # edx = xp
    movl (%ecx),%eax  # eax = *yp (t1)
    movl (%edx),%ebx  # ebx = *xp (t0)
    movl %eax,(%edx)  # *xp = eax
    movl %ebx,(%ecx)  # *yp = ebx
    movl -4(%ebp),%ebx
    movl %ebx,(%ecx)  # *ecx = ebx
    movl %esp, %ebp
    popl %ebp
    ret
```
Understanding Swap

movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx
Why? Arrays, etc.

• given int A[5];
• what is &A[2]?

Why? Arrays, etc.

• given int A[5];
• what is &A[2]?
  – the address of A[0] + 2*sizeof(int)
• What is (%eax,%edx,4) if:
  – %eax contains address of A[0]
  – %edx contains 2

Understanding Swap

<table>
<thead>
<tr>
<th>Variable</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>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x124</td>
<td>12</td>
</tr>
<tr>
<td>0x120</td>
<td>8</td>
</tr>
<tr>
<td>0x118</td>
<td>4</td>
</tr>
<tr>
<td>0x114</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td></td>
</tr>
</tbody>
</table>

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

assembly exp resulting address

x(%eax) %eax+x
(%eax, %ebx) %eax+%ebx
x(%eax, %ebx) x+%eax+%ebx
(,%eax,s) (%eax*s)
x(,%eax,s) x+(%eax*s)
(%eax,%ebx,s) %eax+(%ebx*s)
x(%eax,%ebx,s) x+%eax+(%ebx*s)
practice problem

<table>
<thead>
<tr>
<th>operand</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>?</td>
</tr>
<tr>
<td>0x104</td>
<td>?</td>
</tr>
<tr>
<td>$0x108</td>
<td>?</td>
</tr>
<tr>
<td>(%eax)</td>
<td>?</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>?</td>
</tr>
<tr>
<td>9(%eax,%edx)</td>
<td>?</td>
</tr>
<tr>
<td>260(%ecx,%edx)</td>
<td>?</td>
</tr>
<tr>
<td>0xFC(%ecx,%edx)</td>
<td>?</td>
</tr>
<tr>
<td>(%eax,%edx,4)</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>$0x108</td>
<td>0x108</td>
</tr>
<tr>
<td>(%eax)</td>
<td>0xFF</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>0xAB</td>
</tr>
<tr>
<td>9(%eax,%edx)</td>
<td>0x11</td>
</tr>
<tr>
<td>260(%ecx,%edx)</td>
<td>0x13</td>
</tr>
<tr>
<td>0xFC(%ecx,%edx)</td>
<td>0xFF</td>
</tr>
<tr>
<td>(%eax,%edx,4)</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>register</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

practice problem solution

<table>
<thead>
<tr>
<th>operand</th>
<th>value</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
<td>register</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>absolute address</td>
</tr>
<tr>
<td>$0x108</td>
<td>0x108</td>
<td>immediate</td>
</tr>
<tr>
<td>(%eax)</td>
<td>0xFF</td>
<td>addr 0x100</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>0xAB</td>
<td>addr 0x104</td>
</tr>
<tr>
<td>9(%eax,%edx)</td>
<td>0x11</td>
<td>addr 0x10C</td>
</tr>
<tr>
<td>260(%ecx,%edx)</td>
<td>0x13</td>
<td>addr 0x108</td>
</tr>
<tr>
<td>0xFC(%ecx,%edx)</td>
<td>0xFF</td>
<td>addr 0x100</td>
</tr>
<tr>
<td>(%eax,%edx,4)</td>
<td>0x11</td>
<td>addr 0x10C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>register</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

book complete list of 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>E_a</td>
<td>R[E_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>(E_a)</td>
<td>M[R[E_a]]</td>
<td>indirect</td>
</tr>
<tr>
<td>memory</td>
<td>imm(E_a)</td>
<td>M[imm+R[E_a]]</td>
<td>base+displacement</td>
</tr>
<tr>
<td>memory</td>
<td>(E_a, E_i)</td>
<td>M[R[E_a]+R[E_i]]</td>
<td>indexed</td>
</tr>
<tr>
<td>memory</td>
<td>imm(E_a, E_i)</td>
<td>M[imm+R[E_a]+R[E_i]]</td>
<td>indexed</td>
</tr>
<tr>
<td>memory</td>
<td>(E_i,s)</td>
<td>M[R[E_i]*s]</td>
<td>scaled indexed</td>
</tr>
<tr>
<td>memory</td>
<td>imm(E_i,s)</td>
<td>M[imm+R[E_i]*s]</td>
<td>scaled indexed</td>
</tr>
<tr>
<td>memory</td>
<td>(E_a,E_i,s)</td>
<td>M[R[E_a]+R[E_i]*s]</td>
<td>scaled indexed</td>
</tr>
<tr>
<td>memory</td>
<td>imm(E_a,E_i,s)</td>
<td>M[imm+R[E_a]+R[E_i]*s]</td>
<td>scaled indexed</td>
</tr>
</tbody>
</table>

why something like \((E_b,E_i,s)\)?

- to loop through an array:
  - set \(E_b\) is &A[0]
  - \(E_i\) is an index
  - \(s\) is the size of each element

- we’ll do this a little bit later

- \(E_a\) – some register \(a\)
- \(R[E_a]\) – what’s in register \(a\)
- \(M[x]\) – “value at memory address \(x\)”
some leal practice problems

• if %eax contains x, and %ecx contains y, what is in %edx after the following ops?

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal 6(%eax), %edx</td>
<td>6+x</td>
</tr>
<tr>
<td>leal (%eax,%ecx), %edx</td>
<td>x+y</td>
</tr>
<tr>
<td>leal (%eax,%ecx, 4), %edx</td>
<td>x+4y</td>
</tr>
<tr>
<td>leal 7(%eax,%eax,8), %edx</td>
<td>7+x+8x=7+9x</td>
</tr>
<tr>
<td>leal 0xA(,%ecx,4), %edx</td>
<td>0xA+4y=10+4y</td>
</tr>
<tr>
<td>leal 9(%eax,%ecx,2), %edx</td>
<td>9+x+2y</td>
</tr>
</tbody>
</table>

some leal practice problems

• if %eax contains x, and %ecx contains y, what is in %edx after the following ops?

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>leal 6(%eax), %edx</td>
<td>6+x</td>
</tr>
<tr>
<td>leal (%eax,%ecx), %edx</td>
<td>x+y</td>
</tr>
<tr>
<td>leal (%eax,%ecx, 4), %edx</td>
<td>x+4y</td>
</tr>
<tr>
<td>leal 7(%eax,%eax,8), %edx</td>
<td>7+x+8x=7+9x</td>
</tr>
<tr>
<td>leal 0xA(,%ecx,4), %edx</td>
<td>0xA+4y=10+4y</td>
</tr>
<tr>
<td>leal 9(%eax,%ecx,2), %edx</td>
<td>9+x+2y</td>
</tr>
</tbody>
</table>

leal

• load effective address
• more or less assembly equiv of C’s ‘&’ op
• example:

| %eax | 0x1000 |
| %ecx | 0xA   |

• leal (%eax,%ecx,4), %edx
• what’s in %edx?

%eax+%ecx*4 = 0x1000+10*4
= 0x1000+40
= 0x1000+0x28
= 0x1028

leal

• load effective address
• more or less assembly equiv of C’s ‘&’ op
• example:

| %eax | 0x1000 |
| %ecx | 0xA   |

• leal (%eax,%ecx,4), %edx
• what’s in %edx?

%eax+%ecx*4 = 0x1000+10*4
= 0x1000+40
= 0x1000+0x28
= 0x1028
simple arithmetic ops examples

- we’ll save these for another day

some things that you know in C

- assignment
- simple arithmetic operations
- functions
- conditionals
- loops

Some arithmetic ops

<table>
<thead>
<tr>
<th>instruction</th>
<th>effect</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>incl (D)</td>
<td>(D \leftarrow D + 1)</td>
<td>increment</td>
</tr>
<tr>
<td>decl (D)</td>
<td>(D \leftarrow D - 1)</td>
<td>decrement</td>
</tr>
<tr>
<td>negl (D)</td>
<td>(D \leftarrow -D)</td>
<td>negate</td>
</tr>
<tr>
<td>notl (D)</td>
<td>(D \leftarrow D)</td>
<td>complement</td>
</tr>
<tr>
<td>addl (S,D)</td>
<td>(D \leftarrow D + S)</td>
<td>add</td>
</tr>
<tr>
<td>subl (S,D)</td>
<td>(D \leftarrow D - S)</td>
<td>subtract</td>
</tr>
<tr>
<td>imull (S,D)</td>
<td>(D \leftarrow D \cdot S)</td>
<td>multiply</td>
</tr>
<tr>
<td>xorl (S,D)</td>
<td>(D \leftarrow D \oplus S)</td>
<td>exclusive or</td>
</tr>
<tr>
<td>orl (S,D)</td>
<td>(D \leftarrow D \lor S)</td>
<td>or</td>
</tr>
<tr>
<td>andl (S,D)</td>
<td>(D \leftarrow D \land S)</td>
<td>and</td>
</tr>
<tr>
<td>sall (k,D)</td>
<td>(D \leftarrow D \ll k)</td>
<td>left shift</td>
</tr>
<tr>
<td>shll (k,D)</td>
<td>(D \leftarrow D \ll k)</td>
<td>left shift (same as sall)</td>
</tr>
<tr>
<td>sarl (k,D)</td>
<td>(D \leftarrow D \gg k)</td>
<td>shift right arithmetic</td>
</tr>
<tr>
<td>shr1 (k,D)</td>
<td>(D \leftarrow D \gg k)</td>
<td>shift right logical</td>
</tr>
</tbody>
</table>
Where allocated?

```c
#include <stdlib.h>
#define BUF_LEN 1024

int main(void) {
    int x;
    char *buff;
    ...
    if ((buff = (char*)malloc(BUF_LEN))==NULL) {
        fprintf(stderr, "error allocating space. quitting\n");
        return 1;
    }
    ...
    return 0;
}
```

and here?

```java

class Junk {
    int x;
    String s;
}

class WhereAllocated {
    public static void main(String args[]) {
        final int LEN = 1024;
        int x;
        Junk j = new Junk();
        int A[] = new int[LEN];
        ...
        for (int i=0; i<A.length; i++) {
            ...
            /* do something with A[] */
            ...
        }
    }
}
```
function calls

• stack very important for function calls

• high level view of what’s happening:
function calls

int main(int argc, char **argv)
{
...
F(y);
...
}
void F()
{
....
G();
...
}
void H()
{
....
}
void G()
{
....
H();
...
}

function calls

int main(int argc, char **argv)
{
...
F(y);
...
}
void F()
{
....
G();
...
}
void H()
{
....
}
void G()
{
....
H();
...
}
function calls

```c
int main(int argc, char **argv)
{
    ...
    F();
    ...
}

void F()
{
    ...
    G();
    ...
}

void H()
{
    ...
}

void G()
{
    ...
    H();
}
...
```

Stack

* %esp
* %ebp

function calls

```c
int main(int argc, char **argv)
{
    ...
    F();
    ...
}

void F()
{
    ...
    G();
    ...
}

void H()
{
    ...
}

void G()
{
    ...
    H();
}
...
```

Stack

* %esp
* %ebp

where’s the top, bottom of my frame?

- Answer: %esp and %ebp registers
- Used in particular way in every function call
how do we know where to return?

- Remember `%eip`
- Loop forever:
  - fetch
  - decode
  - execute
- how do we get back?

How do we get back?

- In Dr. Bob’s SIM
  - in caller:
    - save return address in `%R8`
  - in callee:
    - copy saved `%R8` into `%R9` (instruction ptr)
- In IA32:
  - save the return address on the stack
  - location: `%ebp+4`
if we add local variables

<table>
<thead>
<tr>
<th>parameter</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>local var 2</td>
<td>-8(%ebp)</td>
</tr>
<tr>
<td>local var 1</td>
<td>-4(%ebp)</td>
</tr>
<tr>
<td>old %ebp</td>
<td>(%ebp)</td>
</tr>
<tr>
<td>return addr</td>
<td>4(%ebp)</td>
</tr>
<tr>
<td>param 1</td>
<td>8(%ebp)</td>
</tr>
<tr>
<td>param 2</td>
<td>12(%ebp)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>param n</td>
<td>4n+4(%ebp)</td>
</tr>
</tbody>
</table>

%ebp, %esp

- inside a function:
  - beginning of each function:
    - pushl %ebp
    - movl %esp, %ebp
  - at the end of each function, restore:
    - movl %ebp, %esp
    - popl %ebp
  - so common that we have the instructions:
    - ENTER
    - LEAVE

calling a function

- in the caller:
  - save caller’s state
  - pass parameters to the function
  - remember return address
  - turn control over to called function
- in the function:
  - allocate space for locals
  - do the work of the function
  - return value
  - turn control back over to caller

closer look at args on stack

<table>
<thead>
<tr>
<th>parameter</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>old %ebp</td>
<td>(%ebp)</td>
</tr>
<tr>
<td>return addr</td>
<td>4(%ebp)</td>
</tr>
<tr>
<td>param 1</td>
<td>8(%ebp)</td>
</tr>
<tr>
<td>param 2</td>
<td>12(%ebp)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>param n</td>
<td>4n+4(%ebp)</td>
</tr>
</tbody>
</table>
another look at swap

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

another example: sum two ints

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

function return value

- %eax used as a return register by convention

a function template

```c
typedef fname, @function
fname:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax, (%edx)
    movl %ebx, (%ecx)
    movl -4(%ebp),%ebx
    movl %ebp, %esp
    popl %ebp
    ret
```
What about my registers? Isn’t the function going to trash them?

Have a roommate?

another example: sum two ints

```assembly
.globl sumTwoMine
.type sumTwoMine, @function
sumTwoMine:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
popl %ebp
ret
```

Where do we find the arguments? Return value?

same thing, but gcc output

```assembly
.globl sumTwo
.type sumTwo, @function
sumTwo:
pushl %ebp
movl %esp, %ebp
movl 12(%ebp), %eax
movl 8(%ebp), %edx
leal (%edx,%eax), %eax
popl %ebp
ret
```
**Function: Reg Saving Convention**

<table>
<thead>
<tr>
<th>Caller Save</th>
<th>Callie Save</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ebx</td>
</tr>
<tr>
<td>%ecx</td>
<td>%esi</td>
</tr>
<tr>
<td>%edx</td>
<td>%edi</td>
</tr>
</tbody>
</table>

- **Caller Save:** if I have something important in one of these, I need to save it before calling a function.
- **Callie Save:** if I need to write to one of these, save it first. (Otherwise, I might be trashing the caller's stuff.)

---

**Have a Roomate?**

![Dirty Dishes](http://commons.wikimedia.org/wiki/File:Dirty_dishes.png)
yet another look at swap

recursive function: factorial

from the PGU book

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

swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)
    movl -4(%ebp),%ebx
    movl %ebx,%esp
    popl %esp
    ret
```

```assembly
.section .text
.globl _start
.globl factorial #this is unneeded unless we want to share
#this function among other programs
_start:
pushl $4 #The factorial takes one argument - the
#number we want a factorial of. So, it
#gets pushed
call factorial #run the factorial function
addl $4, %esp #Scrubs the parameter that was pushed on
#the stack
call factorial #factorial returns the answer in %eax, but
#we want it in %ebx to send it as our exit
#status
movl $1, %eax #call the kernel's exit function
int $0x80
```

save %ebx

save where?

on the stack.

save where?
from the PGU book

```plaintext
.type factorial, function

factorial:
pushl %ebp              #standard function stuff - we have to
    #restore %ebp to its prior state before
    #returning, so we have to push it
movl %esp, %ebp          #This is because we don't want to modify
    #the stack pointer, so we use %ebp.

movl 8(%ebp), %eax      #This moves the first argument to %eax
    #4(%ebp) holds the return address, and
    #8(%ebp) holds the first parameter

cmpl $1, %eax           #If the number is 1, that is our base
    #case, and we simply return (1 is
    #already in %eax as the return value)
je end_factorial

decl %eax                #otherwise, decrease the value
pushl %eax               #push it for our call to factorial

call factorial           #call factorial

movl 8(%ebp), %ebx      #%eax has the return value, so we
    #reload our parameter into %ebx
imull %ebx, %eax         #multiply that by the result of the
    #last call to factorial (in %eax)
    #the answer is stored in %eax, which
    #is good since that's where return
    #values go.

end_factorial:
    movl %ebp, %esp     #standard function return stuff - we
    #have to restore %ebp and %esp to where
    #they were before the function started
    #return from the function (this pops the
    #return value, too)
    popl %ebp
ret
```