x86 Programming
15-740/18-740
Sept. 9, 2009

Topics
• Basics
• Accessing and Moving Data
• Arithmetic operations
• Control Flow
• Procedures
• Data Structures

x86 Processors
Ubiquitous in the desktop, laptop & server markets

Instruction set has evolved over the past ~30 years
• 8086 (1978) was a 16-bit processor
• 386 (1985) extended to 32-bits with a flat address space
  - Capable of running UNIX/Linux
  - 32-bit Linux/gcc uses no instructions introduced in later models
• 64-bit extensions (2003–2004):
  - AMD’s x86-64 and Intel’s “Intel 64” are nearly identical
  - (not to be confused with Intel’s IA-64 in the Itanium machines)

Constraints on the original x86 instruction set:
• Limited memory and silicon space
• Features to facilitate assembly-language programming
  - More recent (RISC) ISAs focus on compiler-generated code

Abstract Machines

<table>
<thead>
<tr>
<th>Machine Model</th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1) char, 2) int, 3) double, 4) struct, array, 5) pointer</td>
<td>1) loops, 2) conditionals, 3) goto, 4) Proc. call, 5) Proc. return</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASM

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1) byte, 2) word, 3) doubleword, 4) contiguous word allocation, 5) address of initial byte</td>
<td>1) branch/jump, 2) word, 3) jump &amp; link, 4) Proc. call, 5) Proc. return</td>
</tr>
</tbody>
</table>

Assembly Programmer’s View

CPU
- Registers
- Condition Codes
- Program Counter (PC)
  - Address of next instruction
  - Called “EIP” (IA32) or “RIP” (x86-64)
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching

Memory
- Object Code
- Program Data
- OS Data
- Stack
- Memory
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

Addresses
- Data
- Instructions

Programmer-Visible State
• PC
  - Address of next instruction
• Register File
  - Heavily used program data
• Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
Translation Process

- Code in files: `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (-O)
  - Put resulting binary in file `p`

Compiling Into Assembly

**C Code**

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

**Generated IA32 Assembly**

```
sum:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    addl 8(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```

Obtain with command `gcc -O -S code.c`
Produces file `code.s`

Object Code

<table>
<thead>
<tr>
<th>Code for <code>sum</code>&lt;br&gt; 0x401040 &lt;<code>sum&gt;</code>:</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x55 0x89 0x8b 0x45 0x0c 0x03 0x45 0x08 0x89 0xec 0x5d 0xca</td>
<td>Translates <code>.s</code> into <code>.o</code></td>
</tr>
<tr>
<td>Total of 13 bytes</td>
<td>Binary encoding of each instruction</td>
</tr>
<tr>
<td>Each instruction 1, 2, or 3 bytes</td>
<td>Nearly-complete image of executable code</td>
</tr>
<tr>
<td>Starts at address 0x401040</td>
<td>Missing linkages between code in different files</td>
</tr>
</tbody>
</table>

**Linker**

- Resolves references between files
- Combines with static run-time libraries
  - E.g., code for `malloc`, `printf`
- Some libraries are dynamically linked
  - Linking occurs when program begins execution

Machine Instruction Example

**C Code**

```
int t = x+y;
```

**Assembly**

```
addl 8(%ebp), %eax
```

Similar to expression:

```
x += y
```

Or

```
int eax;
int *ebp;
eax += ebp[2]
```

**Object Code**

- 3-byte instruction
- Stored at address 0x401046
Disassembling Object Code

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00401040 &lt;_sum&gt;:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x00401041 &lt;_sum+1&gt;:</td>
<td>89 e5</td>
<td>mov %esp, %ebp</td>
</tr>
<tr>
<td>0x00401043 &lt;_sum+3&gt;:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp), %eax</td>
</tr>
<tr>
<td>0x00401046 &lt;_sum+6&gt;:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp), %eax</td>
</tr>
<tr>
<td>0x00401049 &lt;_sum+9&gt;:</td>
<td>89 ec</td>
<td>mov %ebp, %esp</td>
</tr>
<tr>
<td>0x0040104b &lt;_sum+11&gt;:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x0040104c &lt;_sum+12&gt;:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x0040104d &lt;_sum+13&gt;:</td>
<td>8d 76 00</td>
<td>lea 0x0(%esi), %esi</td>
</tr>
</tbody>
</table>

Disassembler

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o file

Alternate Disassembly

Disassembled

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00401040 &lt;_sum&gt;:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>0x00401041 &lt;_sum+1&gt;:</td>
<td>89 e5</td>
<td>mov %esp, %ebp</td>
</tr>
<tr>
<td>0x00401043 &lt;_sum+3&gt;:</td>
<td>8b 45 0c</td>
<td>mov 0xc(%ebp), %eax</td>
</tr>
<tr>
<td>0x00401046 &lt;_sum+6&gt;:</td>
<td>03 45 08</td>
<td>add 0x8(%ebp), %eax</td>
</tr>
<tr>
<td>0x00401049 &lt;_sum+9&gt;:</td>
<td>89 ec</td>
<td>mov %ebp, %esp</td>
</tr>
<tr>
<td>0x0040104b &lt;_sum+11&gt;:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>0x0040104c &lt;_sum+12&gt;:</td>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>0x0040104d &lt;_sum+13&gt;:</td>
<td>8d 76 00</td>
<td>lea 0x0(%esi), %esi</td>
</tr>
</tbody>
</table>

Within gdb Debugger

gdb p
disable sum

- Disassemble procedure
- `x/13b sum`
- Examine the 13 bytes starting at sum

Moving Data: IA32

Moving Data

- Move 4-byte (“long”) word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with $'
  - E.g., $0x400, $-533
  - Encoded with 1, 2, or 4 bytes
- Register: One of 8 integer registers
  - %eax, %edx, %ecx, %ebx, %esi, %edi, %esp, %ebp
- Memory: 4 consecutive bytes of memory
  - Various “address modes”

Operand Combinations

- Movl Source, Dest:
  - %eax, %edx, %ecx, %ebx, %esi, %edi, %esp, %ebp

movl $0x4,%eax

movl $-147,(%eax)

movl %eax,%edx

Cannot do memory-memory transfer with a single instruction

movl %eax, %edx
temp = *p

movl $0x4,%eax
temp = 0x4;

movl $-147,(%eax)
*p = -147;

movl %eax,%edx
temp2 = temp1;

movl %eax,%edx
*p = temp;

movl (%eax),%edx
*temp = *p;
### Simple 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`

### Using Simple Addressing Modes

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

#### Instruction Set
- **Set Up**
  - `pushl %ebp`
  - `movl %esp,%ebp`
  - `pushl %ebx`
- **Body**
  - `movl 12(%ebp),%ecx`
  - `movl 8(%ebp),%edx`
  - `movl (%ecx),%eax`
  - `movl (%edx),%ebx`
  - `movl %eax,(%edx)`
  - `movl %ebx,(%ecx)`
- **Finish**
  - `movl -4(%ebp),%ebx`
  - `movl %ebp,%esp`
  - `popl %ebp`
  - `ret`

### Indexed Addressing Modes

#### Most General Form
- `D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]`
  - **D**: Constant "displacement" 1, 2, or 4 bytes
  - **Rb**: Base register: Any of 8 integer registers
  - **Ri**: Index register: Any, except for %esp
  - **S**: Scale: 1, 2, 4, or 8

#### Special Cases
- `(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]`
- `(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]`
- `(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]`
- `(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]`

### Understanding Swap

#### Stack Diagram
- `void swap(int *xp, int *yp)`
- `movl 12(%ebp),%ecx`  # ecx = yp
- `movl 8(%ebp),%edx`  # edx = xp
- `movl (%ecx),%eax`  # eax = *xp (t1)
- `movl (%edx),%ebx`  # ebx = *xp (t0)
- `movl %eax,(%edx)`  # *xp = eax
- `movl %ebx,(%ecx)`  # *yp = ebx
### Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>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 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%edx,%ecx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%edx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>

### Address Computation Instruction

**leal**  
*Src, Dest*  
- *Src* is address mode expression  
- Set *Dest* to address denoted by expression

**Uses**
- Computing addresses without a memory reference
  - E.g., translation of p = &x[i];
- Computing arithmetic expressions of the form *x + k*y*
  - *k* = 1, 2, 4, or 8.

### Some Arithmetic Operations

#### Format

**Two Operand Instructions**

- **addl**  
  *Src, Dest*  
  *Dest* = *Dest* + *Src*
- **subl**  
  *Src, Dest*  
  *Dest* = *Dest* - *Src*
- **imull**  
  *Src, Dest*  
  *Dest* = *Dest* * *Src*
- **sall**  
  *Src, Dest*  
  *Dest* = *Dest* <<= *Src*  
  Also called **shll**
- **sarl**  
  *Src, Dest*  
  *Dest* = *Dest* >>= *Src*  
  Arithmetic
- **shrl**  
  *Src, Dest*  
  *Dest* = *Dest* >>= *Src*  
  Logical
- **xorl**  
  *Src, Dest*  
  *Dest* = *Dest* ^ *Src*
- **andl**  
  *Src, Dest*  
  *Dest* = *Dest* & *Src*
- **orl**  
  *Src, Dest*  
  *Dest* = *Dest* | *Src*

#### Computation

- **incl**  
  *Dest*  
  *Dest* = *Dest* + 1
- **decl**  
  *Dest*  
  *Dest* = *Dest* - 1
- **negl**  
  *Dest*  
  *Dest* = - *Dest*
- **notl**  
  *Dest*  
  *Dest* = ~ *Dest*
Using `leal` for Arithmetic Expressions

```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;
}
```

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;
}
```

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;
}
```

Data Representations: IA32 + x86-64

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>- unsigned</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>- int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>- long int</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>- char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>- short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>- float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>- double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>- long double</td>
<td>8</td>
<td>10/12</td>
<td>16</td>
</tr>
<tr>
<td>- char</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

> Or any other pointer
x86-64 General Purpose Registers

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose

Swap in 32-bit Mode (Review)

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

Set Up
pushl %ebp
movl %esp, %ebp
pushl %ebx
movl 12(%ebp), %ecx
movl 8(%ebp), %edx
movl %ecx, %eax
movl (%edx), %eax
movl %eax, (%edx)
movl %ebx, (%ecx)
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret

Body
void swap_l
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Finish
void swap_l
{
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret

Swap in 64-bit Mode

void swap_l
{
    long int t0 = *xp;
    long int t1 = *yp;
    *xp = t1;
    *yp = t0;
}

Swap Long Ints in 64-bit Mode

- Data held in registers %rax and %rdx
- movl operation
  » "q" stands for quad-word
**Condition Codes**

**Single Bit Registers**
- **CF** Carry Flag
- **ZF** Zero Flag
- **OF** Overflow Flag

**Implicitly Set By Arithmetic Operations**
- **addl Src, Dest**
  - Analog: \( t = a + b \)
  - CF set if carry out from most significant bit
  - Used to detect unsigned overflow
  - ZF set if \( t = 0 \)
  - SF set if \( t < 0 \)
  - OF set if two's complement overflow
    \((a>0 \land b>0 \land t<0) \lor (a<0 \land b<0 \land t>0)\)

*Not set by lea, inc, or dec instructions*

---

**Setting Condition Codes (cont.)**

**Explicit Setting by Compare Instruction**
- **cmpl Src2, Src1**
  - CF set if carry out from most significant bit
  - Used for unsigned comparisons
  - ZF set if \( a = b \)
  - SF set if \( (a-b) < 0 \)
  - OF set if two's complement overflow
    \((a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0)\)

**Explicit Setting by Test Instruction**
- **testl Src2, Src1**
  - Sets condition codes based on value of \( Src1 \& Src2 \)
  - Useful to have one of the operands be a mask
  - **testl b,a** like computing \( a\&b \) without setting destination
  - ZF set when \( a\&b = 0 \)
  - SF set when \( a\&b < 0 \)

**Reading Condition Codes**

**SetX Instructions**
- Set single byte based on combinations of condition codes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setsn</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~SF&amp;OF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~SF&amp;OF</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>SF&amp;OF</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>SF&amp;OF&amp;ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF&amp;ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Reading Condition Codes (Cont.)

SetX Instructions
- Set single byte based on combinations of condition codes
- One of 8 addressable byte registers
  - Embedded within first 4 integer registers
  - Does not alter remaining 3 bytes
  - Typically use movzbl to finish job

\[
\begin{align*}
\text{int } & \text{gt} (\text{int } x, \text{int } y) \{
\text{return } x > y; \}
\end{align*}
\]

\[
\begin{align*}
\text{movl } 12(%ebp), %eax & \quad \# \text{eax} = y \\
\text{cmpl } %eax, 8(%ebp) & \quad \# \text{Compare } x : y \\
\text{setg } %al & \quad \# \text{al} = x > y \\
\text{movzbl } %al, %eax & \quad \# \text{Zero rest of } %eax
\end{align*}
\]

Jumping

jX Instructions
- Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>¬ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>¬SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>(SF  OF) &amp; ¬ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>(SF  OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF  OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF  OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>CF &amp; ¬ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Conditional Branch Example

\[
\text{int } \text{absdiff}(\text{int } x, \text{int } y) \{
\text{if } (x > y) \{
\text{result} = x - y;
\} \text{ else } \{
\text{result} = y - x;
\}
\text{return } \text{result};
\}
\]

\[
\begin{align*}
\text{absdiff}: & \quad \text{pushl } %ebp \\
& \quad \text{movl } %esp, %ebp \\
& \quad \text{movl } 8(%ebp), %edx \\
& \quad \text{movl } 12(%ebp), %eax \\
& \quad \text{cmpl } %eax, %edx \\
& \quad \text{jle } .L7 \\
& \quad \text{subl } %eax, %edx \\
& \quad \text{movl } %edx, %eax \quad \text{.L8:} \\
& \quad \text{leave} \\
& \quad \text{ret} \quad \text{.L7:} \\
& \quad \text{subl } %edx, %eax \\
& \quad \text{jmp } .L8
\end{align*}
\]
**Conditionals: x86-64**

```c
int absdiff(  
    int x, int y)  
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

- Conditional move instruction
  - `cmovc src, dest`
  - Move value from src to dest if condition `c` holds
  - More efficient than conditional branching
    - Simple & predictable control flow

**Absdiff**: # x in %edi, y in %esi

```c
movl %edi, %eax  # v = x
movl %esi, %edx # ve
```  

```c
subl %esi, %eax  # v -= y
subl %edi, %edx # ve -= x
```  

```c
cmpl %esi, %edi  # x:y
cmovle %edx, %eax  # v=ve if <=
ret
```  

```c
int absdiff(  
    int x, int y)  
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

**General Form with Conditional Move**

**C Code**

```c
val = Test ? Then-Expr : Else-Expr;
```  

- Both values get computed
- Overwrite then-value with else-value if condition doesn't hold

**Conditional Move Version**

```c
val = Then-Expr;
vale = Else-Expr;
val = vale if !Test;
```  

**Limitations of Conditional Move**

Don't use when:
- Then-Expr or Else-Expr has side effect
- Then-Expr or Else-Expr requires significant computation

```c
int xgty = 0, xltey = 0;
int absdiff_se(
    int x, Int y)
{
    int result;
    if (x > y) {  
        xgty++; result = x-y;
    } else {  
        xltey++; result = y-x;
    }
    return result;
}
```

**“Do-While” Loop Example**

**C Code**

```c
int fact_do(int x)
{
    int result = 1;
    do {  
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}
```

- Use backward branch to continue looping
- Only take branch when “while” condition holds

**Goto Version**

```c
int fact_goto(int x)
{
    int result = 1;
    loop:
        result *= x;
        x = x-1;
    if (x > 1)  
        goto loop;
    return result;
}
```
“Do-While” Loop Compilation

Goto Version

```c
int fact_goto(int x) {
    int result = 1;
    loop:
        result *= x;
        x--; // x = x-1;
        if (x > 1) goto loop;
    return result;
}
```

Assembly

```assembly
fact_goto:
    pushl %ebp # Setup
    movl %esp,%ebp # Setup
    movl $1,%eax # eax = 1
    movl 8(%ebp),%edx # edx = x
L11:
    imull %edx,%eax # result *= x
    decl %edx # x--
    cmpl $1,%edx # Compare x : 1
    jg L11 # if > goto loop
    movl %ebp,%esp # Finish
    popl %ebp # Finish
    ret # Finish
```

Registers

- %edx : x
- %eax : result

“While” Loop Translation

C Code

```c
while (Test) {
    Body
}
done:
```

Do-While Version

```c
if (!Test) goto done;
Body
while (Test);
done:
```

Goto Version

```c
if (!Test) goto done;
loop:
    Body
    while (Test);
done:
```

“For” → “While” → “Do-While”

For Version

```c
for (Init; Test; Update) {
    Body
}
```

While Version

```c
Init;
while (Test) {
    Body
    Update;
}
```

Do-While Version

```c
Init;
if (!Test) goto done;
Body
Update;
while (Test);
done:
```

Goto Version

```c
Init;
if (!Test) goto done;
loop:
    Body
    Update;
    if (Test) goto loop;
done:
```

Switch Statements

Implementation Options

- Series of conditionals
  - Organize in tree structure
  - Logarithmic performance
- Jump Table
  - Lookup branch target
  - Constant time
  - Possible when cases are small integer constants
- GCC
  - Picks one based on case structure
### Jump Table Structure

#### Switch Form

```java
switch(x) {
    case val_0:
        Block 0
        ...
    case val_n-1:
        Block n-1
}
```

#### Jump Table

<table>
<thead>
<tr>
<th>Target</th>
<th>Code Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ0</td>
<td>0</td>
</tr>
<tr>
<td>Targ1</td>
<td>1</td>
</tr>
<tr>
<td>Targ2</td>
<td>2</td>
</tr>
<tr>
<td>Targn-1</td>
<td>n-1</td>
</tr>
</tbody>
</table>

#### Jump Targets

```
target = JTab[x];
goto *target;
```

#### Approx. Translation

```
switch(x) {
    case val_0:
        Block 0
        ...
    case val_n-1:
        Block n-1

    default:
        Default Block
}
```

### Procedure Calls

#### x86 (IA32):
- stack discipline

#### x86-64:
- argument passing in registers

### IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
- Address of top element

```
    Stack Pointer
     %esp

        Stack Grows Down
```

### Switch Statement Example

```
long switch_eg (long x, long y, long z) {
    long w = 1;
    switch(x) {
        case val_0:
            Block 0
            ...
        case val_n:
            Block n
            ...
    }
    return w;
}
```

#### Setup:

```
pushl %ebp
movl %esp, %ebp
pushl %ebx
    # w = 1
pushl %edx
    movl %edx, %eax
    movl 8(%ebp), %eax
    movl 16(%ebp), %eax
    cmpl $6, %eax
    je .L61
    jmp *.L62(,%eax,4) # goto JTab[x]
```

---

Page 12
**IA32 Stack Pushing**

**Pushing**
- pushl \textit{Src}
- Fetch operand at \textit{Src}
- Decrement %esp by 4
- Write operand at address given by %esp

**IA32 Stack Popping**

**Popping**
- popl \textit{Dest}
- Read operand at address given by %esp
- Increment %esp by 4
- Write to \textit{Dest}

**Procedure Control Flow**

- Use stack to support procedure call and return

**Procedure call:**
- call \textit{label} Push return address on stack; Jump to \textit{label}

**Return address value**
- Address of instruction beyond call
- Example from disassembly
  
  804854e: e8 3d 06 00 00 call 8048b90 <main>
  8048553: 50 pushl %eax
  
- Return address = 0x8048553

**Procedure return:**
- ret Pop address from stack; Jump to address

**Procedure Call Example**

```
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 50 pushl %eax
```

```
0x108 123
0x10c 0x8048553
0x110 0x804854e
```

%esp is program counter
## Procedure Return Example

8048591: c3, `ret

```
%esp 0x104
%eip 0x8048553
```

%eip is program counter

## Stack-Based Languages

Languages that support recursion
- e.g., C, Pascal

**Stack Allocated in Frames**
- state for procedure invocation
  - return point, arguments, locals

**Code Example**

```
yoo(…)
{
  who(…)
  {
    amI(…)
    {
      yoo
      who
      amI
      amI
    }
  }
}
```

## IA32/Linux Stack Frame

**Current Stack Frame (“Top” to Bottom)**
- Parameters for function about to call
  - "Argument build"
- Local variables
  - If can’t keep in registers
- Saved register context
- Old frame pointer

**Caller Stack Frame**
- Return address
  - Pushed by call instruction
- Arguments for this call

![IA32/Linux Stack Frame Diagram](image)

## Revisiting swap

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

**Calling swap from call_swap**

```
call_swap:
   pushl $zip2  # Global Var
   pushl $zip1  # Global Var
   call swap
```

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

![Resulting Stack Diagram](image)
Revisiting swap

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

### swap Setup #1

**Entering Stack**
- `%ebp`
- `%esp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**
- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`

### swap Setup #2

**Entering Stack**
- `%ebp`
- `%esp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**
- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`

### swap Setup #3

**Entering Stack**
- `%ebp`
- `%esp`
- `&zip2`
- `&zip1`
- `Rtn adr`

**Resulting Stack**
- `%ebp`
- `%esp`
- `yp`
- `xp`
- `Rtn adr`
- `Old %ebp`
- `Old %ebx`
Register Saving Conventions

When procedure yoo calls who:
- yoo is the caller, who is the callee

Can Register be Used for Temporary Storage?

- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using

Contents of register %edx overwritten by who

Register Saving Conventions

IA32/Linux Register Usage

Integer Registers
- Two have special uses
  %ebp, %esp
- Three managed as callee-save
  %ebx, %esi, %edi
  - Old values saved on stack prior to using
- Three managed as caller-save
  %eax, %edx, %ecx
  - Do what you please, but expect any callee to do so, as well
- Register %eax also stores returned value
Recursive Factorial

```c
int rfact(int x)
{
    int rval;
    if (x <= 1) return 1;
    rval = rfact(x-1);
    return rval * x;
}
```

Registers
- `%eax` used without first saving
- `%ebx` used, but save at beginning & restore at end

x86-64 General Purpose Registers

```
%rax  %eax  %edi  %esi  %edi
%rbx  %ebx  %edi  %esi  %edi
%rcx  %ecx  %edi  %esi  %edi
%rdx  %edx  %edi  %esi  %edi
%rsi  %rsi  %edi  %esi  %edi
%rdi  %rdi  %edi  %esi  %edi
%rsp  %esp  %esp  %esp  %esp
%rbp  %ebp  %ebp  %ebp  %ebp
%rbx  %rbx  %rbx  %rbx  %rbx
%rdx  %rdx  %rdx  %rdx  %rdx
%rsi  %rsi  %rsi  %rsi  %rsi
%rdi  %rdi  %rdi  %rdi  %rdi
%r8   %r8   %r8   %r8   %r8
%r9   %r9   %r9   %r9   %r9
%r10  %r10  %r10  %r10  %r10
%r11  %r11  %r11  %r11  %r11
%r12  %r12  %r12  %r12  %r12
%r13  %r13  %r13  %r13  %r13
%r14  %r14  %r14  %r14  %r14
%r15  %r15  %r15  %r15  %r15
```

- Twice the number of registers
- Accessible as 8, 16, 32, or 64 bits

x86-64 Register Conventions

```
<table>
<thead>
<tr>
<th>Register</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rax</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%rbx</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%rcx</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rdx</td>
<td>Used for linking</td>
</tr>
<tr>
<td>%rsi</td>
<td>C: Callee Saved</td>
</tr>
<tr>
<td>%rdi</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rsp</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee Saved</td>
</tr>
</tbody>
</table>
```

x86-64 Registers

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- All References to Stack Frame via Stack Pointer
  - Eliminates need to update `%ebp`

Other Registers
- 6+1 callee saved
- 2 or 3 have special uses
Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used
  - Intel GAS Bytes C
  - byte b 1 [unsigned] char
  - word w 2 [unsigned] short
  - double word l 4 [unsigned] int
  - quad word q 8 [unsigned] long int (x86-64)

Floating Point
- Stored & operated on in floating point registers
  - Intel GAS Bytes C
  - Single s 4 float
  - Double d 8 double
  - Extended e 10/12/16 long double

Array Accessing Example

Computation
- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at d+%eax + %edx
- Use memory reference (%edx, %eax, 4)

IA32 Memory Reference Code
- int get_digit (zip_digit z, int dig)
  { return z[dig]; }

Generating Pointer to Struct. Member

IA32 Assembly
- struct rec {
  int i;
  int a[3];
  int *p;
}
- void set_i (struct rec *r, int val)
  { r->i = val; }

Structures

Memory Layout
- i
- a
- p

Generating Pointer to Array Element
- Offset of each structure member determined at compile time

IA32 Assembly
- int * find_a (struct rec *r, int idx)
  { return &r->a[idx]; }

- struct rec {
  int i;
  int a[3];
  int *p;
};

- void set_i (struct rec *r, int val)
  { r->i = val; }

- int * find_a (struct rec *r, int idx)
  { return &r->a[idx]; }
C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

The code snippet demonstrates how to set the pointer `p` of a `struct rec` based on the value of `i` and `a`. The diagram illustrates the structure and the pointers involved in this process.