x86 Programming 740  
Sept. 12, 2012

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
  - 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</td>
<td>1) loops</td>
</tr>
<tr>
<td></td>
<td>2) int, float</td>
<td>2) conditionals</td>
</tr>
<tr>
<td></td>
<td>3) double</td>
<td>3) goto</td>
</tr>
<tr>
<td></td>
<td>4) struct, array</td>
<td>4) Proc. call</td>
</tr>
<tr>
<td></td>
<td>5) pointer</td>
<td>5) Proc. return</td>
</tr>
</tbody>
</table>

| ASM           | 1) byte | 3) branch/jump |
|               | 2) word | 4) jump & link |
|               | 3) doubleword |  |
|               | 4) contiguous word allocation |  |
|               | 5) address of initial byte |  |

Assembly Programmer’s View

- PC (Program Counter)
  - 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
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack used to support procedures

- Stack
Translation Process

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

\begin{itemize}
  \item Compiler (\texttt{gcc -S})
  \item Assembler (\texttt{gcc or as})
  \item Linker (\texttt{gcc or ld})
\end{itemize}

\begin{itemize}
  \item C program (\textit{p1.c p2.c})
  \item Asm program (\textit{p1.s p2.s})
  \item Object program (\textit{p1.o p2.o})
  \item Static libraries (.a)
  \item Executable program (p)
\end{itemize}

Compiling Into Assembly

\textbf{C Code}

\begin{verbatim}
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
\end{verbatim}

\textbf{Generated IA32 Assembly}

\begin{verbatim}
    _sum:
pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
\end{verbatim}

Object Code

\begin{itemize}
  \item Code for \textit{sum}
    \begin{verbatim}
    0x401040 <sum>:  
    0x55
    0x89  
    0xe5  
    0x45  
    0xc5  
    0x0c  
    0x03  
    0x45  
    0xe8  
    0x89  
    0xec  
    0x5d  
    0x5d  
    0xe3
    \end{verbatim}
  \item Assembler
    \begin{itemize}
      \item Translates .c into .o
      \item Binary encoding of each instruction
      \item Nearly-complete image of executable code
      \item Missing linkages between code in different files
      \item Starts at address 0x401040
    \end{itemize}
  \item Linker
    \begin{itemize}
      \item Resolves references between files
      \item Combines with static run-time libraries
      \item Some libraries are \textit{dynamically linked}
      \item Linking occurs when program begins execution
    \end{itemize}
\end{itemize}

Machine Instruction Example

\textbf{C Code}

\begin{verbatim}
int t = x+y;
\end{verbatim}

\textbf{Assembly}

- \texttt{addl 8(%ebp),%eax}

\textbf{Object Code}

\begin{verbatim}
0x401046:  03 45 08
\end{verbatim}

Similar to expression:

\begin{itemize}
  \item \texttt{x += y}
  \item \texttt{int eax; int *ebp; eax += ebp[2]}
\end{itemize}
Disassembling Object Code

Disassembled

00401040 <sum>:
  0: 55  push %ebp
  1: 89 e5  mov %esp,%ebp
  3: 8b 45 0c  mov 0xc(%ebp),%eax
  6: 03 45 08  add 0x8(%ebp),%eax
  9: 89 ec  mov %ebp,%esp
  b: 5d  pop %ebp
  c: c3  ret
  d: 8d 76 00  lea 0x0(%esi),%esi

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

Within gdb Debugger

gdb p disassemble sum
- Disassemble procedure
x/13b sum
- Examine the 13 bytes starting at sum

Moving Data: IA32

Moving Data

movl Source, Dest:
- 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
  - But %esp and %ebp reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various "address modes"

Operand Combinations

movl

C Analog

\[
\begin{align*}
\text{movl} & \quad \text{Reg} \quad \text{Reg} \quad \text{Mem} \quad \text{Reg} \quad \text{Reg} \\
\text{temp} &= 0x4; \quad \text{temp}2 = \text{temp}; \\
\text{temp} &= \text{temp}1; \quad \text{temp} &= \text{temp}; \\
\end{align*}
\]

Cannot do memory-memory transfer with a single instruction
Simple Addressing Modes

Normal \( (R) \) \( \text{Mem}[\text{Reg}[R]] \)

- Register \( R \) specifies memory address
  - \text{movl} (%ecx),%eax

Displacement \( D(R) \) \( \text{Mem}[\text{Reg}[R]+D] \)

- Register \( R \) specifies start of memory region
- Constant displacement \( D \) specifies offset
  - \text{movl} 8(%ebp),%edx

Using Simple Addressing Modes

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

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

Understanding Swap

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>yp</td>
<td>12</td>
</tr>
<tr>
<td>%edx</td>
<td>xp</td>
<td>8</td>
</tr>
<tr>
<td>%eax</td>
<td>t1</td>
<td>4</td>
</tr>
<tr>
<td>%ebx</td>
<td>t0</td>
<td>0</td>
</tr>
</tbody>
</table>

Indexed Addressing Modes

Most General Form

\( D(Rb,Ri,S) \) \( \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+D] \)

- \( D \): 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

- \( (Rb,Ri) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]] \)
- \( D(Rb,Ri) \) \( \text{Mem}[\text{Reg}[Rb]+\text{Reg}[Ri]+D] \)
- \( (Rb,Ri,S) \) \( \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]] \)
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  | Computation
---      | ---
Two Operand Instructions
addl Src, Dest | Dest = Dest + Src
subl Src, Dest | Dest = Dest - Src
imull Src, Dest | Dest = Dest * Src
sal1 Src, Dest | Dest = Dest << Src
sal1 Src, Dest | Dest = Dest >> Src
shr1 Src, Dest | Dest = Dest >> Src
xorl Src, Dest | Dest = Dest ^ Src
andl Src, Dest | Dest = Dest & Src
orl Src, Dest  | Dest = Dest | Src

One Operand Instructions

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 t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;
    return rval;
}
```

Understanding `arith`

```assembly
arith:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Another Example

```c
int logical(int x, int y)
{
    int t1 = x^y;
    int t2 = t1 >> 17;
    int t4 = 8192 - 7;
    int rval = t2 & mask;
    return rval;
}
```

Data Representations: IA32 + x86-64

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

» Or any other pointer

Sizes of C Objects (in Bytes)

- `unsigned` 4 4 4
- `int` 4 4 4
- `long int` 4 4 8
- `char` 1 1 1
- `short` 2 2 2
- `float` 4 4 4
- `double` 8 8 8
- `long double` 8 10/12 16
- `char*` 4 4 8

2^3 = 8192, 2^3 − 7 = 8185
### x86-64 General Purpose Registers

- %rax
- %rdx
- %rcx
- %rbx
- %rsi
- %rdi
- %rsp
- %rbp

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

### Swap in 32-bit Mode (Review)

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

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

Set Up

Body

Finish

### Swap in 64-bit Mode

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

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers %eax and %edx
  - movl operation

### Swap Long Ints in 64-bit Mode

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

swap_l:
- movq (%rdi), %rdx
- movq (%rsi), %rax
- movq %rax, (%rdi)
- movq %rdx, (%rsi)
- movl -4(%ebp),%ebx
- movl %ebp,%esp
- popl %ebp
- ret

- 64-bit data
  - Data held in registers %rax and %rdx
  - movq operation
  - "q" stands for quad-word
Condition Codes

Single Bit Registers
- CF: Carry Flag
- SF: Sign Flag
- ZF: Zero Flag
- OF: Overflow Flag

Implicitly Set By Arithmetic Operations
- addl Src,Dest: \( t = a + b \) (a = Src, b = Dest)
  - CF set if carry out from most significant bit
  - 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
- cmpq Src2,Src1
  - CMPL b,a like computing \( a-b \) without setting destination
  - 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
    \( (a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0) \)

Setting Condition Codes (cont.)

Explicit Setting by Test Instruction
- testl Src2,Src1
- testq Src2,Src1
  - Sets condition codes based on value of \( Src1 \land Src2 \)
    - Usefull to have one of the operands be a mask
  - CMPL 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>setns</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

```c
int gt (int x, int y)
{
    return x > y;
}
```

```c
long lgt (long x, long y)
{
    return x > y;
}
```

**Reading condition codes: x86-64**

**SetX Instructions**
- Set single byte based on combinations of condition codes
  - Does not alter remaining 7 bytes

```c
int gt (long x, long y)
{
    return x > y;
}
```

```c
long lgt (long x, long y)
{
    return x > y;
}
```

**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>ZF</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**

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

```c
absdiff:                Set Up
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %edx
    movl 12(%ebp), %eax
    cmpl %eax, %edx
    jle .L7
    subl %eax, %edx
    movl %edx, %eax
    .L8:
    leave
    ret
    .L7:
    subl %edx, %eax
    jmp .L8
```

**Body1**

**Body2**
Conditionals: x86-64

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

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

```asm
absdiff: # x in %edi, y in %esi
    movl %edi, %eax  # v   = x
    movl %esi, %edx  # ve  = y
    subl %esi, %eax  # v  -= y
    subl %edi, %edx  # ve -= x
    cmpl %esi, %edi  # x:y
    cmovle %edx, %eax  # v=ve if <= ret
```

General Form with Conditional Move

C Code

<table>
<thead>
<tr>
<th>C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val = Test ? Then-Expr : Else-Expr;</code></td>
</tr>
</tbody>
</table>

Conditional Move Version

<table>
<thead>
<tr>
<th>Conditional Move Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val = Then-Expr;</code></td>
</tr>
<tr>
<td><code>vale = Else-Expr;</code></td>
</tr>
<tr>
<td><code>val = vale if !Test;</code></td>
</tr>
</tbody>
</table>

Limitations of Conditional Move

<table>
<thead>
<tr>
<th>Limitations of Conditional Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don't use when:</td>
</tr>
<tr>
<td>• Then-Expr or Else-Expr has side effect</td>
</tr>
<tr>
<td>• Then-Expr or Else-Expr requires significant computation</td>
</tr>
</tbody>
</table>

```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

<table>
<thead>
<tr>
<th>C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int fact_do(int x) {</code></td>
</tr>
</tbody>
</table>
|     int result = 1;
|     do { |
|         result *= x;
|         x = x-1;
|     } while (x > 1);
|     return result; |
| `}` |

Goto Version

<table>
<thead>
<tr>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int fact_goto(int x) {</code></td>
</tr>
</tbody>
</table>
|     int result = 1;
|     loop: |
|         result *= x;
|         x = x-1;
|         if (x > 1) |
|             goto loop; |
|     return result; |
| `}` |

• Use backward branch to continue looping
• Only take branch when "while" condition holds
### “Do-While” Loop Compilation

<table>
<thead>
<tr>
<th>Goto Version</th>
<th>Assembly</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fact_goto(int x) { int result = 1; loop: result *= x; x = x-1; if (x &gt; 1) goto loop; return result; }</td>
<td>fact_goto: pushl %ebp # Setup movl %esp,%ebp # Setup movl $1,%eax # eax = 1 movl 8(%ebp),%edx # edx = x</td>
<td>%edx x %eax result</td>
</tr>
</tbody>
</table>

### “While” Loop Translation

<table>
<thead>
<tr>
<th>C Code</th>
<th>Goto Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (Test) Body</td>
<td>if (!Test) goto done; Body while (Test); done:</td>
</tr>
</tbody>
</table>

### “For” → “While” → “Do-While”

<table>
<thead>
<tr>
<th>For Version</th>
<th>While Version</th>
<th>Do-While Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (Init; Test; Update) Body</td>
<td>Init; while (Test) { Body Update ; }</td>
<td>Init; if (!Test) goto done; do { Body Update ; } while (Test) done:</td>
</tr>
</tbody>
</table>

### 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(x) {
  case val_0:
    Block 0
    *  
  case val_1:
    Block 1
    *  
  case val_n-1:
    Block n-1
}
```

**Jump Table**

- Targ0
- Targ1
- Targ2
- Targ[n-1]

**Jump Targets**

- Code Block 0
- Code Block 1
- Code Block 2
- Code Block n-1

**Approx. Translation**

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

**Switch Statement Example**

```
long switch_eg
(long x, long y, long z)
{
  long w = 1;
  switch(x) {
    case val_0: ...
    case val_1: ...
    case val_n-1: ...
  }
  return w;
}
```

**Setup:**

```
pushl %ebp    # Setup
movl %esp, %ebp # Setup
pushl %ebx    # Setup
movl $1, %ebx # w = 1
movl 8(%ebp), %edx # edx = x
movl 16(%ebp), %ecx # ecx = z
cmpl $6, %edx # x:6
ja .L61 # if > goto default
jmp *.L62(%edx,4) # goto JTab[x]
```

**IA32 Stack**

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

**Procedure Calls**

- x86 (IA32):
  - stack discipline

- x86-64:
  - argument passing in registers
IA32 Stack Pushing

Pushing
- `pushl` *Src*
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`

Stack Grows Down

Stack "Top"  Stack Pointer `%esp`

IA32 Stack Popping

Popping
- `popl` *Dest*
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*

Stack Grows Down

Stack "Bottom"  Stack Pointer `%esp`

Procedure Control Flow

- Use stack to support procedure call and return

Procedure call:
  - `call label`
  - Push return address on stack; Jump to *label*

Return address value
  - Address of instruction beyond `call`
  - Example from disassembly:
    - `0x804854e: e8 3d 06 00 00` call `0x8048b90 <main>`
    - `0x8048553: 50` pushl `%eax`
      - Return address = `0x8048553`

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

Procedure Call Example

```
0x804854e: e8 3d 06 00 00 call 0x8048b90 <main>
0x8048553: 50 pushl %eax
call 0x8048b90
```

- `%esp` 0x108  0x110
- `%esp` 0x104  0x8048553
- `%eip` 0x804854e  0x8048b90

%eip is program counter
Procedure Return Example

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

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

Stack Pointer (%esp)

Revisiting swap

int zip1 = 15213;
int zip2 = 91125;

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

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;
}
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`
- `%zip2`
- `%zip1`
- Rtn adr
- `%esp`

**Body**
- `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`

### swap Setup #2

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

**Resulting Stack**
- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- Rtn adr
- `%esp`

**Body**
- `movl %esp,%ebp`
- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
- `ret`

### swap Setup #3

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

**Resulting Stack**
- `%ebp`
- `%esp`
- `%zip2`
- `%zip1`
- Rtn adr
- `%esp`

**Body**
- `movl %esp,%ebp`
- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
- `ret`
**Effect of swap Setup**

**Entering Stack**

<table>
<thead>
<tr>
<th>Offset (relative to %ebp)</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>%esp</td>
<td>%ebp</td>
</tr>
<tr>
<td>yp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn adr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resulting Stack**

<table>
<thead>
<tr>
<th>Offset (relative to %ebp)</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td>%esp</td>
<td>%ebp</td>
</tr>
<tr>
<td>yp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn adr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp

Body
```

**Observation**

- Saved & restored register %ebx

**swap Finish #1**

**swap's Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
<td></td>
</tr>
</tbody>
</table>

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

**swap Finish #2**

**swap's Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
<td></td>
</tr>
</tbody>
</table>

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

**swap Finish #3**

**swap's Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>%ebp</th>
<th>%esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>yp</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>xp</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>Old %ebx</td>
<td></td>
</tr>
</tbody>
</table>

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
Register Saving Conventions

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

Can Register be Used for Temporary Storage?

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

Observation
- Saved & restored register %ebx
- Didn’t do so for %eax, %ecx, or %edx

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

Caller-Save Temporaries
%eax
%edx
%ecx
%esp
%ebp
%edi
%esi
%ebp
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**

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

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

---

**x86-64 Register Conventions**

<table>
<thead>
<tr>
<th>%rax</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r9</th>
<th>Argument #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r10</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Used for linking</td>
</tr>
<tr>
<td>%r12</td>
<td>C: Callee Saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee Saved</td>
</tr>
<tr>
<td>%r15</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
### 32-bit code for swap (review)

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

### 64-bit code for swap

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

**Body**
- `pushl %ebp`
- `movl %esp,%ebp`
- `pushl %ebx`
- `movl 8(%ebp), %edx`
- `movl 12(%ebp), %ecx`
- `movl (%edx), %ebx`
- `movl (%ecx), %eax`
- `movl %eax, (%edx)`
- `popl %ebx`
- `popl %ebp`
- `ret`

**Set Up**
- `movl (%edi), %edx`
- `movl (%rsi), %eax`
- `movl %eax, (%rsi)`

**Finish**
- `ret`

**Operands passed in registers (why useful?)**
- First (xp) in %edi, second (yp) in %rsi
- 64-bit pointers

No stack operations required

32-bit data
- Data held in registers %eax and %edx
- `movl` operation

### 64-bit code for long int swap

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

**Body**
- `movq (%edi), %rdx`
- `movq (%rsi), %rax`

**Set Up**
- `movq (%rdi), %rdx`
- `movq (%rsi), %rax`

**Finish**
- `ret`

**64-bit data**
- Data held in registers %rax and %rdx
- `movq` operation
  - "q" stands for quad-word

### 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 l 8 double
  - Extended t 10/12/16 long double
Array Accessing Example

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

**IA32 Memory Reference Code**

```c
int get_digit(zip_dig z, int dig)
{
    return z[dig];
}
```

```assembly
# %edx = z  
# %eax = dig  
movl (%edx,%eax,4),%eax  # z[dig]
```

**Structures**

**Memory Layout**

```
\[
\begin{array}{cccc}
  & a & a & p \\
0 & & & \\
4 & & & \\
16 & & & \\
\end{array}
\]
```

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

**IA32 Assembly**

```
# %eax = val  
# %edx = r  
movl %eax,(%edx)  # Mem[r] = val
```

Generating Pointer to Struct. Member

**C Code**

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

```assembly
# %edx = r  
movl (%edx),%ecx  # r->i
leal 0(,%ecx,4),%eax  # 4*(r->i)
leal 4(%edx,%eax),%eax  # r+4*(r->i)
movl %eax,16(%edx)  # Update r->p
```

**Structure Referencing (Cont.)**

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

```assembly
int *
find_a(0(,%ecx,4),%eax)  # 4*(r->idx)
leal 4(%edx,%eax),%eax  # r+4*(r->idx)
movl %eax,16(%edx)  # Update r->p
```

```assembly
# %edx = r  
movl (%edx),%ecx  # r->i
leal 0(,%ecx,4),%eax  # 4*(r->i)
leal 4(%edx,%eax),%eax  # r+4*(r->i)
movl %eax,16(%edx)  # Update r->p
```

Generating Pointer to Array Element

**C Code**

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

```assembly
# %edx = idx  
# %edx = r  
leal 0(,%ecx,4),%eax  # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```