CSC 252: Machine Level Programming: Instruction Set Architectures

Turning C into Object Code
- 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

Assembly Programmer's View

Turning C into Object Code

- 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

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

Compiling Into Assembly

Generated Assembly

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

Obtain with command
gcc -O -S code.c
Produces file code.s
**Operating Systems**

**Assembly Characteristics**

- **Minimal Data Types**
  - "Integer" data of 1, 2, or 4 bytes
  - Addresses (untyped pointers)
  - Data values
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory

- **Primitive Operations**
  - Perform arithmetic function on register or memory data
  - Transfer data between memory and register
    - Load data from memory into register
    - Store register data into memory
  - Transfer control
    - Unconditional jumps to/from procedures
    - Conditional branches

**Object Code**

- **Code for `sum`**
  - 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 `malloc`, `printf`
      - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

- **Machine Instruction Example**
  - C Code
    - Add two signed integers
  - Assembly
    - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - **x**: Register `%eax`
    - **y**: Memory `M[%ebp+8]`
    - **t**: Register `%eax`
    - Return function value in `%eax`
  - **Object Code**
    - 3-byte instruction
    - Stored at address `0x401046`

- **Disassembling Object Code**
  - **Disassembler**
    - `objdump -d p`
    - 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
Disassembled

Object
0x401040: push %ebp
0x401041: mov %esp,%ebp
0x401043: mov 0xc(%ebp),%eax
0x401046: add 0x8(%ebp),%eax
0x401049: mov %ebp,%esp
0x40104b: pop %ebp
0x40104c: ret
0x40104d: lea 0x0(%esi),%esi

Within gdb Debugger
gdb p
– Disassemble procedure
x/13b sum
– Examine the 13 bytes starting at sum

What Can be Disassembled?
% objdump -d WINWORD.EXE
WINWORD.EXE: file format pei-i386
No symbols in "WINWORD.EXE".
Disassembly of section .text:
30001000 <.text>:
30001000: 55 push %ebp
30001001: 8b ec mov %esp, %ebp
30001003: 6a ff push $0xffffffff
30001005: 68 90 10 00 30 push $0x30001090
3000100a: 68 91 10 00 30 push $0x3040c91

– Anything that can be interpreted as executable code
– Disassembler examines bytes and reconstructs assembly source

Moving Data

– 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

– Cannot do memory-memory transfers with single instruction

movl Operand Combinations
Source Destination C Analog

• Various "address modes"
Simple Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
    \texttt{movl \%ecx,\%eax}

- Displacement (D(R)) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    \texttt{movl 8(%ebp),\%edx}

Using Simple Addressing Modes

\textbf{void swap(int *xp, int *yp)}
{\begin{itemize}
  \item \texttt{int t0 = *xp;}
  \item \texttt{int t1 = *yp;}
  \item \texttt{*xp = t1;}
  \item \texttt{*yp = t0;}
\end{itemize}}

\texttt{movl 12(%ebp),\%ecx}
\texttt{movl 8(%ebp),\%edx}
\texttt{movl \%ecx,\%eax}
\texttt{movl \%edx,\%ebx}
\texttt{movl \%eax,\%edx}
\texttt{movl \%ebx,\%ecx}
\texttt{movl -4(%ebp),\%ebx}
\texttt{movl \%ebp,\%esp}
\texttt{popl \%ebp}
\texttt{ret}

Understanding Swap

\begin{itemize}
  \item \texttt{movl 12(%ebp),\%ecx} \# ecx = yp
  \item \texttt{movl 8(%ebp),\%edx} \# edx = xp
  \item \texttt{movl \%ecx,\%eax} \# eax = *xp (t1)
  \item \texttt{movl \%edx,\%ebx} \# ebx = *xp (t0)
  \item \texttt{movl \%eax,\%edx} \# *xp = eax
  \item \texttt{movl \%ebx,\%ecx} \# *yp = ebx
\end{itemize}

Address
\begin{itemize}
  \item 123 \quad 0x124
  \item 456 \quad 0x120
  \item 0x11c
  \item 0x118
  \item 0x114
  \item 0x110
  \item 0x10c
  \item 0x108
  \item 0x104
  \item 0x100
\end{itemize
Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%esax</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

Rtn adr

17

Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%esax</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

Rtn adr

18

Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%esax</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

Rtn adr

19

Understanding Swap

<table>
<thead>
<tr>
<th>Offset</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>yp 12</td>
<td>0x120</td>
</tr>
<tr>
<td>xp 8</td>
<td>0x124</td>
</tr>
<tr>
<td></td>
<td>0x10c</td>
</tr>
<tr>
<td>%ebp</td>
<td>0x104</td>
</tr>
<tr>
<td>%esax</td>
<td>0x100</td>
</tr>
</tbody>
</table>

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

Rtn adr

20

CSC 256/456 - Spring 2007
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
```

Indexed Addressing Modes

- Most General Form
  
  \[ D(Rb,Ri,S) \text{ Mem[Reg[Reg[Rb]+S*Reg[Ri]]+ D]} \]
  
  - \( D \): 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
  
  \( (Rb,Ri) \text{ Mem[Reg[Rb]+Reg[Ri]]} \)
  
  \( D(Rb,Ri) \text{ Mem[Reg[Rb]+Reg[Ri]]+D} \)
  
  \( (Rb,Ri,S) \text{ Mem[Reg[Rb]+S*Reg[Ri]]} \)

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 address without doing 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`
  - `sall` `Src, Dest` `Dest = Dest << Src` Also called `shll`
  - `sarh` `Src, Dest` `Dest = Dest >> Src` Arithmetic
  - `shrh` `Src, Dest` `Dest = Dest >> Src` Logical
  - `xorl` `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 = x + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 + t5;
    return rval;
}
```

```
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 %eax, %ebp
    popl %ebp
    ret
```

Using **leal** for Arithmetic Expressions

Set Up

Body

Finish
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;
}
```

```
# eax = x
movl 8(%ebp), %eax
# edx = y
movl 12(%ebp), %edx
# ecx = x+y (t1)
leal (%edx,%eax), %ecx
# edx = 48*y (t4)
shl $4,%edx
# ecx = z+t1 (t2)
addl 16(%ebp), %ecx
# eax = 4+t4+x (t5)
leal 4(%edx,%eax), %eax
# eax = t5*t2 (rval)
imull %ecx, %eax
```

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;
}
```

```
literal:                  Set Up
    pushl %ebp
    movl %ebp,%esp
    movl 8(%ebp), %eax
    xorl 12(%ebp), %eax
    salr $17,%eax
    andl $8185,%eax
movl %ebp,%esp           Body
ret
```

CISC Properties

- Instruction can reference different operand types
  - Immediate, register, memory
- Arithmetic operations can read/write memory
- Memory reference can involve complex computation
  - Rb + S*Ri + D
- Useful for arithmetic expressions, too
- Instructions can have varying lengths
  - IA32 instructions can range from 1 to 15 bytes
Summary: Abstract Machines

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

Assembly

<table>
<thead>
<tr>
<th>mem</th>
<th>reg</th>
<th>alu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cond. Codes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1) byte 2) 2-byte word 3) 4-byte long word 4) contiguous byte allocation 5) address of initial byte

Whose Assembler?

<table>
<thead>
<tr>
<th>Intel/Microsoft Format</th>
<th>GAS/Gnu Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea eax, [ecx+ecx*2]</td>
<td>leal (%ecx,%ecx,2),%eax</td>
</tr>
<tr>
<td>sub esp, 8</td>
<td>subl $8,%esp</td>
</tr>
<tr>
<td>cmp dword ptr [ebp-8], 0</td>
<td>cmpl $0,-8(%ebp)</td>
</tr>
<tr>
<td>mov eax, dword ptr [eax*4+100h]</td>
<td>movl $0x100(,%eax,4),%eax</td>
</tr>
</tbody>
</table>

- Intel/Microsoft Differences from GAS
  - Operands listed in opposite order
  - Constants not preceded by '$', Denote hex with 'h' at end
  - Operand size indicated by operands rather than operator suffix
  - Addressing format shows effective address computation

Control Flow

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
  - C 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
    \( (a>0 \&\& b>0 \&\& t<0) \ | | (a<0 \&\& b<0 \&\& t>=0) \)

- Not Set by leal instruction
Setting Condition Codes (cont.)

- Explicit Setting by Compare Instruction
  
  \texttt{cmp\,Src2,Src1}
  
  - \texttt{cmp\,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
  
  \texttt{test\,Src2,Src1}
  
  - Sets condition codes based on value of \(Src1 \land Src2\)
    - Useful to have one of the operands be a mask
  - \texttt{test\,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^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</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 \texttt{movzb}l to finish job

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

Body

\[
\texttt{movl\,12(%ebp),%eax \# eax = y}
\texttt{cmp\,%eax,%(ebp) \# Compare x : y}
\texttt{setg %al \# al = x > y}
\texttt{movzb\,%al,%eax \# Zero rest of %eax}
\]

Note inverted ordering!
Jumping

- Jumping 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>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jnz</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jnc</td>
<td>SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>js</td>
<td>(SF OF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jns</td>
<td>(SF OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>CF</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 max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
```

```asm
max:
pushl %ebp
movl %esp,%ebp
movl 8(%ebp),%edx
movl 12(%ebp),%eax
cmpl %eax,%edx
jle L9
movl %edx,%eax
L9:
movl %ebp,%esp
popl %ebp
ret
```

Conditional Branch Example (Cont.)

```c
int goto_max(int x, int y) {
    int rval = y;
    int ok = (x <= y);
    if (ok)
        goto done;
    rval = x;
    done:
    return rval;
}
```

```asm
goto_max:
movl $%ebp,%edx # edx = x
movl 12(%ebp),%eax # eax = y
cmpl %eax,%edx # x : y
jle L9 # if <= goto L9
movl %edx,%eax # eax = x
L9:
    # Done:
```

“Do-While” Loop Example

- C allows “goto” as means of transferring control
- Closer to machine-level programming style
- Generally considered bad coding style

C Code

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

Goto Version

```c
int fact_goto(int x) {
    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

**Goto Version**

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

**Assembly**

```
_fact_goto:
pushl %ebp # Setup
movl %esp,%ebp # Setup
movl $1,%eax # eax = 1
movl %ebp,%edx # edx = x
.L11:
imull %edx,%eax # result *= x
dcl %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

### General “Do-While” Translation

**C Code**

```c
do
    Body (Test);
while (Test);
```

**Goto Version**

```
loop:
    Body
    if (Test)
        goto loop
```

- Body can be any C statement
- Typically compound statement:
  ```c
  { 
      Statement1;
      Statement2;
      ... 
      Statement_n;
  }
  ```
- Test is expression returning integer
  - 0 interpreted as false
  - ≠0 interpreted as true

### “While” Loop Example #1

**C Code**

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

**First Goto Version**

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

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

### Actual “While” Loop Translation

**C Code**

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

**Second Goto Version**

```c
int fact_while_goto2 (int x)
{
    int result = 1;
    if (!(x > 1))
        goto done;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    done:
    return result;
}
```

- Uses same inner loop as do-while version
- Guards loop entry with extra test
### General “While” Translation

**C Code**

```c
while (Test)
    Body
```

**Do-While Version**

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

**Goto Version**

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

### “For” Loop Example

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1) result *= x;
        x = x*x;
    }
    return result;
}

### ipwr Computation

/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1) result *= x;
        x = x*x;
    }
    return result;
}

### “For” Loop Example

```c
int result;
for (result = 1; p != 0; p = p >> 1) {
    if (p & 0x1) {
        result *= x;
        x = x*x;
    }
    return result;
}
```

### General Form

```
for (Init; Test; Update)
    Body
```

<table>
<thead>
<tr>
<th>result</th>
<th>x</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6561</td>
<td>1</td>
</tr>
<tr>
<td>531441</td>
<td>43046721</td>
<td>0</td>
</tr>
</tbody>
</table>
### For” → “While”

**For Version**
```c
for (Init; Test; Update)
    Body
```

**Do-While Version**
```c
Init;
if (!Test)
    goto done;
do {
    Body
    Update;
} while (!Test)
done:
```

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

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

### For” Loop Compilation

**Goto Version**
```c
Init;
if (!Test)
    goto done;
loop:
    Body
    Update;
    if (Test)
        goto loop;
done:
result = 1;
if (p == 0)
goto done;
loop:
    if (p & 0x1)
        result *= x;
    p = p >> 1;
    if (p != 0)
        goto loop;
done:
```

### Switch Statements

- **Implementation Options**
  - Series of conditionals
    - Good if few cases
    - Slow if many
  - Jump Table
    - Lookup branch target
    - Avoids conditionals
    - Possible when cases are small integer constants
  - GCC
    - Picks one based on case structure
    - Bug in example code
    - No default given

```c
typedef enum
    {ADD, MULT, MINUS, DIV, MOD, BAD} op_type;
char unparse_symbol(op_type op)
    {
        switch (op) {
            case ADD:
                return '+';
            case MULT:
                return '*';
            case MINUS:
                return '-';
            case DIV:
                return '/';
            case MOD:
                return '%';
            case BAD:
                return '?';
        }
    }
```

### Jump Table Structure

**Switch Form**
```c
switch(op) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    ...*
    case val_n-1:
        Block n-1
}
```

**Jump Table**

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

**Approx. Translation**
```c
switch(op) {
    case val_0:
        Target = Targ0;
        goto *target;
    case val_1:
        Target = Targ1;
        goto *target;
    ...*
    case val_n-1:
        Target = Targn-1;
        goto *target;
}
```
Switch Statement Example

- Branching

```cpp
typedef enum
{
    ADD, MULT, MINUS, DIV, MOD, BAD
} op_type;

char unparse_symbol(op_type op)
{
    switch (op) {
    case ADD: return 0;
    case MULT: return 1;
    case MINUS: return 2;
    case DIV: return 3;
    case MOD: return 4;
    case BAD: return 5;
    }
}
```

Enumerated Values

- Setup:

```assembly
unparse_symbol:
    pushl %ebp  # Setup
    movl %esp,%ebp  # Setup
    movl (%ebp),%eax  # eax = op
    cmpl $5,%eax  # Compare op : 5
    ja .L49  # If > goto done
    jmp *.L57(,%eax,4)  # goto Table[op]
```

Assembly Setup Explanation

- Symbolic Labels
  - Labels of form .LXX translated into addresses by assembler
- Table Structure
  - Each target requires 4 bytes
  - Base address at .L57
- Jumping
  - Jump target is denoted by label .L49
  - Start of jump table denoted by label .L57
  - Register %eax holds op
  - Must scale by factor of 4 to get offset into table
  - Fetch target from effective Address .L57 + op*4

Jump Table

- Targets & Completion

```assembly
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

Switch Statement Completion

- Puzzle
  - What value returned when op is invalid?
- Answer
  - Register %eax set to op at beginning of procedure
  - This becomes the returned value
- Advantage of Jump Table
  - Can do k-way branch in \( \mathcal{O}(1) \) operations
Object Code

- Setup
  - Label .L49 becomes address 0x804875c
  - Label .L57 becomes address 0x8048bc0

```
08048718 <unparse_symbol>:
  8048718:55  pushl  %ebp
  8048719:89 e5  movl  %esp,%ebp
  804871b:80 45 08  movl  0x8(%ebp),%eax
  804871e:83 f8 05  cmpl  $0x5,%eax
  8048721:77 39  ja  804875c <unparse_symbol+0x44>
  8048723:ff 24 85 c0 8b jmp  *0x8048bc0(,%eax,4)
```

Object Code (cont.)

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
    gdb code-examples
    (gdb) x/6xw 0x8048bc0
    - Examine hexadecimal format "words" (4-bytes each)
    - Use command "help x" to get format documentation
    0x8048bc0 <_fini+32>:
 0x8048730 0x8048737 0x8048740 0x8048747 0x8048750 0x8048757

Extracting Jump Table from Binary

- Jump Table Stored in Read Only Data Segment (.rodata)
  - Various fixed values needed by your code
- Can examine with objdump
  objdump code-examples -s --section=.rodata
  - Show everything in indicated segment.
- Hard to read
  - Jump table entries shown with reversed byte ordering

Contents of section .rodata:
```
08048bc0 30870408 7870408 4870408 0...8...G...
08048bd0 30870408 7870408 46616374 28256429 F...W...Fact (%d)
08048be0 203d2025 6c640a00 43686172 203d2025 = id...Char = %
```

Disassembled Targets

- movl  %esi,%esi does nothing
- Inserted to align instructions for better cache performance

```
8048730:b8 2b 00 00 00 movl  $0x2b,%eax
8048735:eb 25 jmp  804875c <unparse_symbol+0x44>
8048737:b8 2a 00 00 00 movl  $0x2a,%eax
804873c:eb 1e jmp  804875c <unparse_symbol+0x44>
804873e:89 f6 movl  %esi,%esi
8048740:b8 2d 00 00 00 movl  $0x2d,%eax
8048745:eb 15 jmp  804875c <unparse_symbol+0x44>
8048747:b8 2f 00 00 00 movl  $0x2f,%eax
804874c:eb 0e jmp  804875c <unparse_symbol+0x44>
804874e:89 f6 movl  %esi,%esi
8048750:b8 25 00 00 00 movl  $0x25,%eax
8048755:eb 05 jmp  804875c <unparse_symbol+0x44>
8048757:b8 3f 00 00 00 movl  $0x3f,%eax
```

E.g., 30870408 really means 0x8048730
Matching Disassembled Targets

Entry

- 0x08048730: b8 2b 00 00 00 movl
- 0x08048735: eb 2b          jmp
- 0x08048737: b8 2a 00 00 00 movl
- 0x0804873c: eb 1e          jmp
- 0x08048740: b8 2d 00 00 00 movl
- 0x08048745: eb 05          jmp
- 0x08048747: b8 2f 00 00 00 movl
- 0x0804874c: eb 0a          jmp
- 0x08048750: b8 29 00 00 00 movl
- 0x08048755: eb 0b          jmp
- 0x08048757: b8 3f 00 00 00 movl

Disassembled Targets

Summarizing

- C Control
  - if-then-else
  - do-while
  - while
  - switch

- Assembler Control
  - jump
  - Conditional jump

- Compiler
  - Must generate assembly code to implement more complex control

Standard Techniques

- All loops converted to do-while form
- Large switch statements use jump tables

Conditions in CISC

- CISC machines generally have condition code registers

Conditions in RISC

- Use general registers to store condition information
- Special comparison instructions
  - E.g., on Alpha:
    cmple $16,1,$1
    Sets register $1 to 1 when Register $16 <= 1

Disclaimer

- Parts (most in this lecture) of the slides were developed by the course text authors: Dave O'Hallaron and Randy Bryant. The slides are intended for the sole purpose of instruction of computer organization at the University of Rochester. All copyrighted materials belong to their original owner(s).