Machine-Level Programming II: Control Flow

Today
- Condition codes
- Control flow structures

Next time
- Procedures
Processor state (ia32, partial)

- Information about currently executing program
  - Temporary data
  - Location of runtime stack
  - Location of current code control point
  - Status of recent tests

General purpose registers:

- %eax
- %ecx
- %edx
- %ebx
- %esi
- %edi
- %esp
- %ebp
- %eip

Condition codes:

- CF
- ZF
- SF
- OF
Condition codes (Implicit setting)

- **Single bit registers**
  
<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>

- **Implicitly set by arithmetic operations**
  
  ```c
  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 (positive or negative)
    
      \[(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)\]
  
- **Logical operations leave** **CF and OF set to 0**
- **Not set by** **leal instruction (why?)**
Condition codes (Explicit setting: compare)

- Explicit setting by compare instruction
  
  ```
  cmp{b,w,l} 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 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
  ```
Condition codes (Explicit setting: test)

- Explicit setting by test instruction
  
  \[ \text{test}\{b,w,l\} \ Src2, Src1 \]
  
  - Sets condition codes based on value of \( Src1 \) \& \( Src2 \)
    
    • Useful to have one of the operands be a mask
  
  - \text{testl}\ b, a \ like\ a \& b \ without\ setting\ destination
  
  - \text{ZF} \ set \ when\ a \& b == 0
  
  - \text{SF} \ set \ when\ a \& b < 0
## Reading condition codes

- **SetX Instructions**
  - Set single byte based on combinations of condition codes
  - Note the suffixes do not indicate operand sizes, but conditions

<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

- **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;
}
```

Body

```asm
movl 12(%ebp),%eax  # eax = y
cmpl %eax,8(%ebp)   # Compare x : y
setg %al            # al = x > y
movzbl %al,%eax     # Zero rest of %eax
```

Note inverted ordering!
Breaking with sequential execution

- “Normal” execution executes instructions in listed order
- To move to a different location – jump
  - Jump to different part of code depending on condition codes
  - Destination of a jump – label
  - Label addresses are determined when generating the object code

```plaintext
movl  8(%ebp),%edx
movl 12(%ebp),%eax
cmpl  %eax,%edx
jle  .L9
movl  %edx,%eax
.L9:
leave
ret
```
Jumping

- **jX Instructions**
  - `jmp` can be direct (to a label) or indirect (e.g. `*(%eax)`, to M[R[%eax]]); everything else only direct

<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>
int max(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}

max:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%edx
    movl 12(%ebp),%eax
    cmpl %eax,%edx
    jle .L9
    movl %edx,%eax

.L9:
    leave
    ret
C allows “goto” as means of transferring control

- Closer to machine-level programming style

- Generally considered bad coding style

Skipped when $x \leq y$!
### General “if-else” translation

#### C Code

```c
If (test-expr)
  then-statement
else
  else-statement
```

#### Goto Version

```c
t = test-expr;
if (!t)
  goto false;
then-statement
  goto done;
false:
else-statement;
```

- **test-expr** is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- Only one of the two branches is executed
- There are other models/goto versions
Conditional expression

**C Code**

```c
val = test? then_expr : else_expr;
```

- Test is an expression returning an int
  - 0 is false; !0 is true

```c
int absdiff(int x, int y)
{
    return x<y? y-x:x-y;
}
```

**Goto Version**

```c
nt = !test;
if (nt) goto else;
val = then_expr;
goto done;
else:
    val = else_expr;
done:
```

```assembly
movl 8(%ebp), %eax
movl 12(%ebp), %edx
cmpl %edx, %eax
jge .L2
subl %eax, %edx
movl %edx, %eax
jmp .L3
.L2: subl %edx, %eax
.L3: ...
```
Loops

- C provides different looping constructs
  - do-while, while, for
- No corresponding instruction in machine code
- Most compilers
  - Transform general loops into do-while
  - Then compile them into machine code
“Do-While” loop example

- Compute factorial of n \((n!\))

### 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

```assembly
fact_goto:
    pushl %ebp     # Setup
    movl %esp,%ebp # Setup
    movl $1,%eax  # eax = 1
    movl 8(%ebp),%edx # edx = x

.L2:
    imull %edx,%eax # result *= x
    decl %edx # x--
    cmpl $1,%edx # Compare x : 1
    jg .L2 # if > goto loop

    leave       # Finish
    ret         # Finish
```

- **Registers**
  - %edx  x
  - %eax  result
General “Do-While” translation

**C Code**

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

**Goto Version**

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

- **Body** can be any C statement
  - Typically compound statement:
    ```c
    {
      Statement_1;
      Statement_2;
      ...
      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;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test
General “While” translation

C Code

while (Test)
Body

Do-While Version

if (!Test)
goto done;
do
Body
while (Test);
done:

Goto Version

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

Body

Init
i = 2;

Test
i <= n;

Update
i++;

Body
{
  result *= i;
}

int fact_for(int n)
{
    int i;
    for (i = 2; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
“For” → “While”

For Version

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

While Version

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

Goto Version

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

Do-While Version

```
Init;
    if (! Test)
        goto done;
do {
    Body
    Update;
} while (Test)
done:
```
"For" loop compilation

**Goto Version**

Init;
if (!Test)
  goto done;

loop:
  Body
  Update ;
  if (Test)
    goto loop;

done:

**Init**

i = 2;

**Test**

i <= n;

**Update**

i++;

**Body**

{ result *= i; 
}

{ int result = 1;
i = 2;
if (! (i <= n)) 
  goto done;

loop:
  result *= i;
i++;
  if (i <= n)
    goto loop;

done:
  return result; 
}
"For" loop compilation result

```c
{ int result = 1;
i = 2;
if (!(i <= n))
    goto done;
loop:
    result *= i;
i++;
if (i <= n)
    goto loop;
done:
    return result;
}
```

Assembly

```
fact_for:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %ecx
    Get n
    movl $2, %edx
    i <- 2
    movl $1, %eax
    result <- 1
    cmp $1, %ecx
    Comp n:1
    jle .L3
    Comp n:i
    .L6:
    imull %ecx, %eax
    addl $1, %edx
    result *= i
    i++
    .L3:
    cmpl %edx, %ecx
    jge .L6
    If >=, goto loop
    .L3:
    popl %ebp
    done
    ret
```
Conditional moves

- Conditional move instructions
  - Instruction supports if (test) dest ← src
  - Supported starting with PentiumPro, since 1995
  - GCC tends not to use it for backward compatibility; it’s starting to change (use switch -march=i686 -i.e. PentiumPro)

- Why?
  - Branches disrupt the instruction flows of pipelines
  - Conditional moves avoid control transfers

C Code

```c
val = test
? then_expr
: else_expr;
```

Goto Version

```c
tval = then_expr;
result = else_expr;
t = test;
if (t) result = tval;
return result;
```
When using conditional moves maybe a bad idea
- Expensive computation; one of the sides will be discarded
- Computation has side effects (must be side-effect free!)

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

```assembly
x in %ecx  y in %edx
movl %edx, %eax
tval=y-x
subl %ecx, %eax
movl %ecx, %ebx
result=x-y
subl %edx, %ebx
Comp x:y
cmpl %edx, %ecx
if >, result=tval
```

...
A multi-way branching capability based on the value of an `int`

Useful when many possible outcomes

Switch cases
  - Multiple case labels
    • Here 5 & 6;
  - Fall through cases:
    • Here 2
  - Missing cases:
    • Here 4

Easier to read C code and more efficient implementation with jump tables

```c
long switch_eg (long x, long y, long z) {
    long w = 1;
    switch(x) {
        case 1:
            w = y * x;
            break;
        case 2:
            w = y/z;
            /* fall through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```
Jump table structure

- An array where entry \( i \) is the address of a code segment to run when switch index equals \( i \)

Switch form

```c
switch(x) {
  case val_0:  
    Block 0
  case val_1:  
    Block 1
  • • •
  case val_{n-1}:  
    Block \( n-1 \)
}
```

Jump table

```
jtab:  
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ0</td>
</tr>
<tr>
<td>Targ1</td>
</tr>
<tr>
<td>Targ2</td>
</tr>
<tr>
<td>•</td>
</tr>
<tr>
<td>•</td>
</tr>
<tr>
<td>•</td>
</tr>
<tr>
<td>Targ_{n-1}</td>
</tr>
</tbody>
</table>
```

Jump targets

- Targ0: Code Block 0
- Targ1: Code Block 1
- Targ2: Code Block 2
- Targ_{n-1}: Code Block \( n-1 \)

Approx. translation

```c
target = JTab[op];
goto *target;
```
Switch statement example

Setup:

```
long switch_eg (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        ...
    }
    return w;
}
```

```
switch_eg:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl $1, %ebx
    movl 8(%ebp),%edx
    movl 16(%ebp), %ecx
    cmpl $6, %edx
    ja .L8
    jmp * .L9(%edx,4)
```
Switch statement example

```c
long switch_eg (long x, long y, long z) {
    long w = 1;
    switch (x) {
        ...
    }
    return w;
}
```

**Setup:**

```assembly
switch_eg:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    movl $1, %ebx
    movl 8(%ebp), %edx
    movl 16(%ebp), %ecx
    cmpl $6, %edx
    ja .L8
    jmp * .L9(, %edx, 4)
```

**Jump table**

```assembly
.sect .rodata
.align 4
.L9:
    .long .L8 # x = 0
    .long .L3 # x = 1
    .long .L4 # x = 2
    .long .L5 # x = 3
    .long .L8 # x = 4
    .long .L7 # x = 5
    .long .L7 # x = 6
```

**Indirect jump**
Assembly setup explanation

- **Table structure**
  - Each target requires 4 bytes
  - Base address at .L9

- **Jumping**
  jmp .L8
  - Jump target is denoted by label .L9
  jmp *.*L9(%edx,4)
  - Start of jump table denoted by label .L9
  - Register %edx holds op
  - Must scale by factor of 4 to get offset into table (labels have 32-bits, 4 bytes on IA32)
  - Fetch target from effective Address .L9 + op*4

**Jump table**

```assembly
.section .rodata
.align 4
.L9:
.long   .L8 # x = 0
.long   .L3 # x = 1
.long   .L4 # x = 2
.long   .L5 # x = 3
.long   .L8 # x = 4
.long   .L7 # x = 5
.long   .L7 # x = 6
```
### Jump table

```assembly
.section .rodata
.align 4
.L9:
.long .L8 # x = 0
.long .L3 # x = 1
.long .L4 # x = 2
.long .L5 # x = 3
.long .L8 # x = 4
.long .L7 # x = 5
.long .L7 # x = 6
```

```c
switch(x) {  
  case 1:    /* .L3 */  
    w = y * x;  
    break;
  case 2:    /* .L4 */  
    w = y/z;  
    /* fall through */
  case 3:    /* .L5 */  
    w += z;  
    break;
  case 5:  
  case 6:    /* .L7 */  
    w -= z;  
    break;
  default:  /* .L8 */  
    w = 2;
}
```
switch(x) {
    case 1:    /* .L3 */
        w = y * x;
        break;
    case 2:    /* .L4 */
        w = y/z;
        /* fall through */
    case 3:    /* .L5 */
        w += z;
        break;
    case 5:
    case 6:    /* .L7 */
        w -= z;
        break;
    default:    /* .L8 */
        w = 2;
}

.L3:
    movl    12(%ebp), %ebx
    imull   %edx, %ebx
    jmp     .L2

.L4:
    movl    12(%ebp), %eax
    cltd    
    idivl   %ecx
    movl    %eax, %ebx

.L5:
    addl    %ecx, %ebx
    jmp     .L2

.L7:
    subl    %ecx, %ebx
    jmp     .L2

.L8:
    movl    $2, %ebx
Object code

- Setup
  - Label .L8 becomes address 0x8048367
  - Label .L9 becomes address 0x8048470

```assembly
08048334 <switch_eg>:
8048334:       55                      push   %ebp
8048335:       89 e5                  mov    %esp,%ebp
8048337:       53                      push   %ebx
8048338:       8b 55 08               mov    0x8(%ebp),%edx
804833b:       8b 4d 10               mov    0x10(%ebp),%ecx
804833e:       bb 01 00 00 00         mov    $0x1,%ebx
8048343:       83 fa 06               cmp    $0x6,%edx
8048346:       77 1f                  ja     8048367 <switch_eg+0x33>
8048348:       ff 24 95 70 84 04 08   jmp    *0x8048470(%edx,4)
```
Object code

- **Jump table**
  - Doesn’t show up in disassembled code
  - Can inspect using GDB

```
gdb code-examples
(gdb) x/7yw 0x8048470
```

- **Examine 7 hexadecimal format “words” (4-bytes each)**
- **Use command “help x” to get format documentation**

```
0x8048470:
  0x08048367
  0x0804834f
  0x08048357
  0x0804835f
  0x08048363
  0x08048363
```
### Disassembled targets

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>804834f</td>
<td>8b 5d</td>
<td>mov $0xc(%ebp),%ebx</td>
<td>8048352</td>
<td>0f af</td>
<td>imul %edx,%ebx</td>
</tr>
<tr>
<td></td>
<td>0c</td>
<td></td>
<td>8048355</td>
<td>eb 15</td>
<td>jmp 0x080483c &lt;switch_eg+0x38&gt;</td>
</tr>
<tr>
<td>8048357</td>
<td>8b 45</td>
<td>mov $0xc(%ebp),%eax</td>
<td>804835a</td>
<td>99</td>
<td>cltd</td>
</tr>
<tr>
<td></td>
<td>0c</td>
<td></td>
<td>804835b</td>
<td>f7 f9</td>
<td>idiv %ecx</td>
</tr>
<tr>
<td>804835d</td>
<td>89 c3</td>
<td>mov %eax,%ebx</td>
<td>804835f</td>
<td>01 cb</td>
<td>add %ecx,%ebx</td>
</tr>
<tr>
<td>8048361</td>
<td>eb 09</td>
<td>jmp 0x080483c &lt;switch_eg+0x38&gt;</td>
<td>8048363</td>
<td>29 cb</td>
<td>sub %ecx,%ebx</td>
</tr>
<tr>
<td>8048365</td>
<td>eb 05</td>
<td>jmp 0x080483c &lt;switch_eg+0x38&gt;</td>
<td>8048367</td>
<td>bb 02 00 00 00</td>
<td>mov $0x2,%ebx</td>
</tr>
<tr>
<td>804836c</td>
<td>89 d8</td>
<td>mov %ebx,%eax</td>
<td>804836e</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>804836f</td>
<td>c9</td>
<td>leave</td>
<td>8048370</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
Sparse switch example

- Not practical to use jump table
  - Would require 1000 entries
- Obvious translation into if-then-else would have max. of 9 tests

```c
/* Return x/111 if x is multiple && <= 999.
   -1 otherwise */
int div111(int x)
{
    switch(x) {
    case  0: return 0;
    case 111: return 1;
    case 222: return 2;
    case 333: return 3;
    case 444: return 4;
    case 555: return 5;
    case 666: return 6;
    case 777: return 7;
    case 888: return 8;
    case 999: return 9;
    default: return -1;
    }
}
```
Sparse switch code

- Compares x to possible case values
- Jumps different places depending on outcomes

```assembly
movl 8(%ebp),%eax  # get x
cmp $444,%eax     # x=444
  je L8
ej L16
cmpl $111,%eax   # x=111
  je L5
ej L17
testl %eax,%eax  # x=0
  je L4
jmp L14

L5:
mov $1,%eax
  jmp L19
L6:
mov $2,%eax
  jmp L19
L7:
mov $3,%eax
  jmp L19
L8:
mov $4,%eax
  jmp L19

...
Sparse switch code structure

- Organizes cases as binary tree
- Logarithmic performance
Summarizing

- **C Control**
  - if-then-else, do-while, while, switch
  - Conditional transfer of data

- **Assembler control**
  - Jump & conditional jump

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

- **Standard techniques**
  - All loops → do-while form
  - Large switch statements use jump tables