Today

- Finish-up control flow: loops and switch statements
- If interested: talk on floating point precision 1pm in CSE 305
- HW 1 due tomorrow 11pm
- You have the knowledge to start Lab 2 (due in 1 week).
  - Today, Thursday section, and Friday should give you the rest.
  - Do your part to thwart Dr. Evil!

Typical control flow code in assembly

```
cmpl %eax, %ebx
je label
...
label: ...
...```

Condition Codes (Explicit Setting: Compare)

- Single-bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
  - ZF: Zero Flag
  - OF: Overflow Flag (for signed)
- Explicit Setting by Compare Instruction
  - `cmp`/`cmpq` Src2,Src1
  - `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 (as signed)
    - OF set if two’s complement (signed) overflow
      (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

Condition Codes (Explicit Setting: Test)

- Single-bit registers
  - CF: Carry Flag (for unsigned)
  - SF: Sign Flag (for signed)
  - ZF: Zero Flag
  - OF: Overflow Flag (for signed)
- Explicit Setting by Test instruction
  - `testl/testq` Src2,Src1
  - `testl` b,a like computing a & b without setting destination
    - ZF set if a&b == 0
    - SF set if a&b < 0
    - `testl` %eax, %eax
      - Sets SF and ZF, check if eax is +,0,-
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) &amp;ZF</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 (Cont.)

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

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
go to Exit;
}
```

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

Conditional Branch Example (Cont.)

```c
int goto_ad(int x, int y)
{
    int result;
    if (x <= y) goto Else;
    result = x-y;
    Exit:
    return result;
Else:
    result = y-x;
    goto Exit;
}
```

```
absdiff:
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
```

- PC relative branches are relocatable
- Absolute branches are not
Compiling Loops

C/Java code:
```java
while ( sum != 0 ) {
    <loop body>
}
```

Machine code:
```asm
loopTop:  cmpl $0, %eax
          je  loopDone
          <loop body code>
          jmp  loopTop
loopDone:
```

- How to compile other loops should be straightforward
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop

“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;
}
```

Goto Version
```c
int fact_goto(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

“Do-While” Loop Compilation

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

Assembly
```asm
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:

<table>
<thead>
<tr>
<th>%edx</th>
<th>x</th>
<th>result</th>
</tr>
</thead>
</table>

Translation?

“Do-While” Loop Compilation

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

Assembly
```asm
fact_goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx

.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:

<table>
<thead>
<tr>
<th>%edx</th>
<th>x</th>
<th>result</th>
</tr>
</thead>
</table>

Translation?
General “Do-While” Translation

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

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

- **Body:**
  ```c
  {  
    Statement;
    Statement;
    ...
    Statement;
  }
  ```

- **Test** returns integer
  - 0 interpreted as false
  - ≠ 0 interpreted as true

“While” Loop Translation

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

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

- **Used by GCC for both IA32 & x86-64**
- **First iteration jumps over body computation within loop straight to test**

“While” Loop Example

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

“For” Loop Example: Square-and-Multiply

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

- **Algorithm**
  - **Exploit bit representation:** \( p = p_{n-1} + 2p_{n-1} + \ldots \) \( 2^{n-1}p_{n-1} \)
  - **Gives:** \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot ((z_{n-1})^2)^2 \) \( n-1 \) times
  - \( z_0 = 1 \) when \( p_i = 0 \)
  - \( z_i = x \) when \( p_i = 1 \)
  - **Complexity \( O(\log p) = O(\text{sizeof}(p)) \)**

Example

\( 3^{10} = 3^4 \cdot 3^4 \cdot 3^2 = (3^3)^3 \)
**ipwr Computation**

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

<table>
<thead>
<tr>
<th>before iteration</th>
<th>result</th>
<th>x=3</th>
<th>p=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10=1010₂</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5=101₂</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>81</td>
<td>2=10₂</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6561</td>
<td>1=1₂</td>
</tr>
<tr>
<td>5</td>
<td>59049</td>
<td>43046721</td>
<td>0₂</td>
</tr>
</tbody>
</table>
Switch Statement Example

- Multiple case labels
  - Here: 5, 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4
- Lots to manage, we need a *jump table*

Jump Table Structure

C code:
```c
switch(x) {
  case 1: <some code>
  break;
  case 2: <some code>
  break;
  case 3: <some code>
  break;
  case 5:
  case 6: <some code>
  break;
  default: <some code>
}
```

We can use the jump table when `x <= 6`:
```c
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```

Jump Table Structure (IA32)

Declaring data, not instructions:
```assembly
.switch(x) {
  case 1: // .L56
    w = y*z;
    break;
  case 2: // .L57
    w = y/z;
    /* Fall Through */
  case 3: // .L58
    w += z;
  case 5:
  case 6: // .L60
    w = 2;
  default: // .L61
    w = 2;
}
```

“long” as in movl: 4 bytes would be .quad in x86-64
Switch Statement Example (IA32)

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

Assembly Setup Explanation (IA32)

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at .L62

- **Jumping: different address modes for target**
  - Direct: jmp .L61
    - Jump target is denoted by label .L61
  - Indirect: jmp *.L62(%edx,4)
    - Start of jump table: .L62
    - Must scale by factor of 4 (labels are 32-bits = 4 bytes on IA32)
    - Fetch target from effective address .L62 + edx*4

```
switch(x) {
    . . .
    case 2:   /* Fall Through */
        w = y/z;
        break;
    case 3:   // .L58
        w += z;
        break;
    default: // .L61
        w = 2;
    }
    return w;
```

Code Blocks (Partial)

```
.L61: // Default case
    movl $2, %ebx  # w = 2
    jmp .L63
.L57: // Case 2:
    movl 12(%ebp), %eax  # y
    cltd  # Div prep
    idivl %ecx  # y/z
    movl %eax, %ebx  # w = y/z
    jmp .L63  # Fall through - no jmp
.L58: // Case 3:
    addl %ecx, %ebx  # w += z
    jmp .L63
...
.L63
    movl %ebx, %eax  # return w
    popl %ebx
    leave
    ret
```
### Code Blocks (Rest)

```assembly
switch(x) {
    case 1:      // .L56
        w = y*z;
        break;
    case 5:
    case 6:      // .L60
        w -= z;
        break;
    . . .
}
return w;
```

### Code Blocks (Partial, return inlined)

```assembly
.switch(x) {
    .L60: // Cases 5&6:
        subl %ecx, %ebx # w -= z
        jmp .L63
    .L56: // Case 1:
        movl 12(%ebp), %ebx # w = y
        imull %ecx, %ebx # w*= z
        jmp .L63
    . . .
    .L63:
        movl %ebx, %eax # return w
        popl %ebx
        leave
        ret

    .L61: // Default case
    .L61: // Default case
        movl $2, %ebx # w = 2
        movl %ebx, %eax # Return w
        popl %ebx
        leave
        ret

    .L57: // Case 2:
        movl 12(%ebp), %eax # y
        cltd # Div prep
        idivl %ecx # y/z
        movl %eax, %ebx # w = y/z
        jmp .L58 # Fall through - no jmp
    .L58: // Case 3:
        addl %ecx, %ebx # w+= z
        movl %ebx, %eax # Return w
        popl %ebx
        leave
        ret
```

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

### Code Blocks (Rest, return inlined)

```assembly
.switch(x) {
    case 1:      // .L56
        w = y*z;
        break;
    . . .
    case 5:
    case 6:      // .L60
        w -= z;
        break;
    . . .
}
```

### IA32 Object Code

- **Setup**
  - Label `.L61` becomes address `0x08048630`
  - Label `.L62` becomes address `0x080488dc`

#### Assembly Code

```assembly
.switch_eg:
    . . .
    ja .L61 # if > goto default
    jmp *.L62(%edx,4) # goto JTab[x]
```

#### Disassembled Object Code

```
08048610 <switch_eg>:
    . . .
08048622:  77 0c     ja 8048630
08048624:  ff 24 95 dc 88 04 08 jmp *0x80488dc(%edx,4)
```
IA32 Object Code (cont.)

- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
    - `gdb asm-cntl` (gdb) x/7xw 0x080488dc
      - Examine 7 hexadecimal format "words" (4-bytes each)
      - Use command "help x" to get format documentation

```
0x080488dc:
0x08048630
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649
```

Disassembled Targets

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048630:</td>
<td>bb 02 00 00 00</td>
</tr>
<tr>
<td>8048635:</td>
<td>89 d8</td>
</tr>
<tr>
<td>8048637:</td>
<td>5b</td>
</tr>
<tr>
<td>8048638:</td>
<td>c9</td>
</tr>
<tr>
<td>8048639:</td>
<td>c3</td>
</tr>
<tr>
<td>804863a:</td>
<td>8b 45 0c</td>
</tr>
<tr>
<td>804863d:</td>
<td>99</td>
</tr>
<tr>
<td>8048642:</td>
<td>01 cb</td>
</tr>
<tr>
<td>8048644:</td>
<td>89 d8</td>
</tr>
<tr>
<td>8048645:</td>
<td>5b</td>
</tr>
<tr>
<td>8048647:</td>
<td>c9</td>
</tr>
<tr>
<td>8048648:</td>
<td>c3</td>
</tr>
<tr>
<td>804864b:</td>
<td>29 cb</td>
</tr>
<tr>
<td>804864d:</td>
<td>89 d8</td>
</tr>
<tr>
<td>8048650:</td>
<td>8b 5d 0c</td>
</tr>
<tr>
<td>8048653:</td>
<td>0f af d9</td>
</tr>
<tr>
<td>8048656:</td>
<td>89 d8</td>
</tr>
<tr>
<td>8048658:</td>
<td>5b</td>
</tr>
<tr>
<td>8048659:</td>
<td>c9</td>
</tr>
<tr>
<td>804865a:</td>
<td>c3</td>
</tr>
</tbody>
</table>
```

Matching Disassembled Targets

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08048630</td>
<td>bb 02 00 00 00</td>
</tr>
<tr>
<td>0x08048650</td>
<td>89 d8</td>
</tr>
<tr>
<td>0x0804863a</td>
<td>c9</td>
</tr>
<tr>
<td>0x08048642</td>
<td>8b 45 0c</td>
</tr>
<tr>
<td>0x08048649</td>
<td>89 d8</td>
</tr>
<tr>
<td>0x0804864a</td>
<td>c9</td>
</tr>
<tr>
<td>0x0804865a</td>
<td>c3</td>
</tr>
</tbody>
</table>
```

Question

- Would you implement this with a jump table?

```
switch(a) {
    case 0:     <some code>
    break;
    case 10:    <some code>
    break;
    case 52000: <some code>
    break;
    default:    <some code>
    break;
}
```

- Probably not:
  - Don't want a jump table with 52001 entries for only 4 cases (too big)
  - about 200KB = 200,000 bytes
  - text of this switch statement = about 200 bytes
Quick Review

- **x86-64 vs. IA32**
  - Integer registers: 16 x 64-bit vs. 8 x 32-bit
  - `movq`, `addq`, ... vs. `movl`, `addl`, ...
    - `movq` -> “move quad word” or 4*16-bits
  - x86-64: better support for passing function arguments in registers

- **Complete memory addressing mode**
  - `(eax)`, 17`(eax)`, 2`(ebx`, `ecx`, 8), ...

- **Immediate (constant), Register, and Memory Operands**
  - `subl %eax, %ecx`  # ecx = ecx + eax
  - `sall $4,%edx`  # edx = edx << 4
  - `addl 16(%ebp),%ecx`  # ecx = ecx + Mem[16+ebp]
  - `imull %ecx,%eax`  # eax = eax * ecx

---

Quick Review

- **Control**
  - 1-bit condition code registers
  - Set as side effect by arithmetic instructions or by `cmp`, `test`
    - Used:
      - Read out by setx instructions (`setg`, `setle`, ...)
      - Or by conditional jumps (`jle .L4`, `je .L10`, ...)
      - Or by conditional moves (`cmovle %edx, %eax`)

- **Arithmetic operations also set condition codes**
  - `subl`, `addl`, `imull`, `shrl`, etc.

- **Load Effective Address does NOT set condition codes**
  - `leal 4(%edx,%eax),%eax`  # eax = 4 + edx + eax

---

Quick Review

- **Do-While loop**
  ```c
  while (Test) {
    Body;
  }
  ```

- **While-Do loop**
  ```c
  do
    Body;
  while (Test);
  ```

© 2022 University of Washington