Homework 0: 2.84

• Given: k-bit exponent, n-bit fraction
• Find:
  – Exponent E,
  – Significand M,
  – Fraction f,
  – Value V,
  – Bit representation
Homework 0: 2.84

- Given: k-bit exponent, n-bit fraction
- 7.0: 0111 = 1.11 × 2²
  - Exponent E = 2
  - Significand M = 1.11
  - Fraction f = 0.11
  - Value V = 7
  - Bit representation
    - E = 2 = e - Bias = e - (2^{k-1} - 1)
    - e = 2 + 2^{k-1} - 1 = 1 + 2^{k-1} = 0...01 + 10...0 = 10...01
    - f = 110....
Homework 0: 2.84

• Given: k-bit exponent, n-bit fraction

• **Largest odd int**: n+1 1’s (to represent exactly)
  – Exponent E = n
  – Significand M = 1.11….1
  – Fraction f = 0.1111....
  – Value V = n+1 1’s = 2^{n+1}-1

• Bit representation
  • E = n = e - Bias = e - (2^{k-1} – 1 )
  • e = n + 2^{k-1} - 1
  • f = 1...1
Condition Codes (Explicit Setting: Compare)

• Explicit Setting by Compare Instruction
  \texttt{cmp}l/\texttt{cmp}q \ Src2,Src1
  \texttt{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 \( (a-b) < 0 \) (as signed)
  – **OF set** if two’s complement (signed) overflow
    \( (a>0 \ && \ b<0 \ && \ (a-b)<0) \ || \ (a<0 \ && \ b>0 \ && \ (a-b)>0) \)
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>ZF</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>
Let’s Work Through an Example

And when you gaze long into an abyss the abyss also gazes into you. ~ Friedrich Nietzsche
Reading Condition Codes (Cont.)

• **SetX Instructions:**
  Set single byte based on combination of condition codes

• **One of 8 addressable byte registers**
  – Does not alter remaining 3 bytes
  – Typically use `movzb1` to finish job

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

---

**Body**

```assembly
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzb1 %al,%eax
```
Reading Condition Codes (Cont.)

• **SetX Instructions:**
  Set single byte based on combination of condition codes

• **One of 8 addressable byte registers**
  – Does not alter remaining 3 bytes
  – Typically use `movzbl` to finish job

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

**Body**

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

Note inverted ordering!
```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x - y;
    } else {
        result = y - x;
    }
    return result;
}
```

```
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
```

Body1

Setup

Body2

Finish
### 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;
}
```

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

```assembly
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
```
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;
}
```

```asm
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx  
    jle   .L7
    subl   %eax, %edx
    jmp   .L8

.L7:  
    subl   %edx, %eax
    jmp   .L8

.L8:  
    leave
    ret
```

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
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
cmp %eax, %edx
jle .L7
subl %eax, %edx
movl %edx, %eax
.L8:
leave
ret
.L7:
subl %edx, %eax
jmp .L8
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
General Conditional Expression Translation

C Code

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

Goto Version

```
nt = !Test;
if (nt) goto Else;
val = Then-Expr;
Done:
    ...
Else:
    val = Else-Expr;
goto Done;
```

— *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
— Create separate code regions for then & else expressions
— Execute appropriate one
General Form with Conditional Move

C Code

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

Conditional Move Version

```c
val1 = Then-Expr;
val2 = Else-Expr;
val1 = val2 if !Test;
```

- Both values get computed
- Overwrite then-value with else-value if condition doesn’t hold
- **Don’t use when:**
  - Then or else expression have side effects
  - Then and else expression are to expensive
“Do-While” Loop Example

- 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
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx
    # initialization
.L11:
    imull %edx,%eax
    decl %edx
    cmpl $1,%edx
    jg .L11
    movl %ebp,%esp
    popl %ebp
    ret
```

Registers:

- %edx  \( x \)
- %eax  \( \text{result} \)
Goto Version

```c
int fact_goto(int x)
{
    int result = 1;

loop:
    result *= x;
    x = x - 1;
    if (x > 1)
        goto loop;

    return result;
}
```

Assembly

```
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>edx</td>
<td>x</td>
</tr>
<tr>
<td>eax</td>
<td>result</td>
</tr>
</tbody>
</table>
General “Do-While” Translation

C Code

do
  Body
while (Test);

Goto Version

loop:
  Body
  if (Test)
    goto loop

• **Body:**
  
  
  \[
  \begin{align*}
  \text{Statement}_1; \\
  \text{Statement}_2; \\
  \ldots \\
  \text{Statement}_n;
  \end{align*}
  \]

• **Test** returns integer
  
  = 0 interpreted as false
  ≠0 interpreted as true
“While” Loop Example

C Code

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

Goto Version #1

```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
Alternative “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 #2

```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

While version

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:
New Style “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_goto3(int x)
{
    int result = 1;
    goto middle;
    loop:
        result *= x;
        x = x-1;
    middle:
        if (x > 1)
            goto loop;
    return result;
}
```

• Recent technique for GCC
  –Both IA32 & x86-64

• First iteration jumps over body computation within loop
Jump-to-Middle While Translation

C Code

\[
\text{while (Test)} \\
\quad Body
\]

- Avoids duplicating test code
- Unconditional `goto` incurs no performance penalty
- for loops compiled in similar fashion

Goto Version

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

Goto (Previous) Version

```
if (!Test)
    goto done;
loop:
        Body
        if (Test)
            goto loop;
done:
```
Jump-to-Middle Example

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

```
# x in %edx, result in %eax
jmp .L34  # goto Middle
.L35:    # Loop:
imull %edx, %eax # result *= x
decl %edx  # x--
.L34:     # Middle:
cmpl $1, %edx # x:1
jg .L35     # if >, goto Loop
```
Implementing Loops

• IA32
  – All loops translated into form based on “do-while”

• x86-64
  – Also make use of “jump to middle”

• Why the difference
  – IA32 compiler developed for machine where all operations costly
  – x86-64 compiler developed for machine where unconditional branches incur (almost) no overhead