# Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address 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>

\[
D(Rb,Ri,S) \quad \text{Mem}[\text{Reg}[Rb]+S*\text{Reg}[Ri]+ D]
\]
• 3.32
• 3.33
“Do-While” Loop Example

• Use backward branch to continue looping
• Only take branch when “while” condition holds
```c
int
fact_goto(int x)
{
    int result = 1;

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

```
Registers:
%edx   x
%eax   result

Assembly

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

.L11:
imull %edx,%eax       # result *= x
decl %edx              # x--
cmpl $1,%edx           # Compare x : 1
jg  .L11               # if > goto loop

movl %ebp,%esp        # Finish
popl %ebp              # Finish
ret                    # Finish
```
General “Do-While” Translation

C Code

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

Goto Version

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

- **Body:**
  ```c
  { 
      Statement_1;
      Statement_2;
      ...
      Statement_n;
  }
  ```

- **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
While version

while \((\text{Test})\)
    Body

do-while version

if (!\(\text{Test}\))
    goto \textit{done};
done:

while (\textit{Test});

Goto Version

if (!\(\text{Test}\))
    goto \textit{done};
loop:
    Body
    if (\textit{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

while (Test)
  Body

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