

COEN 175

Lecture 18: Code Generation for Unary Operators

Negation

- Arithmetic negation uses the `neg` instruction.

- Generate code for $-x + y$.

```
movl    x, %eax
negl    %eax
addl    y, %eax
```

- The logical negation of x , $!x$, is equivalent to $x == 0$.

- Generate code for $!x$.

```
movl    x, %eax
cmpl    $0, %eax
sete    %al
movzbl  %al, %eax
```

Type Cast

- Converting to a smaller sized type is not necessary, as we simply use the appropriately sized register.
 - For example, we will simply use AL if we need to read the contents of EAX (or RAX) as a byte.
- Converting to a larger sized type requires sign extension, which is done by the `movs` instruction with the appropriate suffix.
- Generate code for `(int) c + 123`.

```
movsbl c, %eax    # sign extend byte to long
addl    $123, %eax
```

Address

- The address operator has two cases.
- If its operand is a dereference expression then the operator does nothing as `&*p` is equivalent to `p`.
 - All that is necessary is to pass along its operand's location.
- If its operand is an identifier then we use the `leal` instruction to “load the effective address”.
 - Note that addresses are always 32-bits.
- Generate code for `&x`.

```
leal    x, %eax
```

Strings

- To allocate space for a zero-terminated ASCII string, we use the `.asciz` assembler directive.

```
.L0:    .asciz "hello world"
```

- Labels beginning with a period are local to the file.
- When the string is used, the address of its first character must be computed using `lea` as before.
- Generate code for `"hello world" + 4`.

```
leal    .L0, %eax  
addl    $4, %eax
```

Dereference: Load

- A dereference expression could be used as either an lvalue or an rvalue.
- If used as an rvalue, we read the memory whose address is specified by the operand.
- Generate code for `*p + x`.

```
movl    p, %eax
movl    (%eax), %eax
addl    x, %eax
```

- Every use of a dereference as an **rvalue** should have an indirect memory reference as a **source** operand.

Dereference: Store

- If a dereference is used as an lvalue then no operation is done by the dereference operator itself.
- Rather, the parent operation controls what is done:
 - As previously seen, the address operator simply ignores the dereference and passes along its operand.
 - An assignment statement will do a store.
- Every **assignment** via a dereference should result in an indirect memory reference as a **target** operand.
 - Note that in `**p = x`, the inner dereference is an rvalue and the outer dereference is an lvalue.

Assignment: Variable

- The left-hand side of an assignment statement is either a variable or a dereference.
- If it is a variable, then do the following:
 1. Generate code for both operands.
 2. If the right operand is a memory reference, then load it into a register.
 3. Move the right operand into the memory location given by the left operand using the `mov` instruction.
- In my implementation, a type cast has been inserted to truncate or extend the right operand.

Assignment: Dereference

- If the left-hand side is a dereference, then:
 1. Generate code for the right operand and the child of the left operand.
 2. If the right operand is a memory reference, then load it into a register.
 3. Load the **child** of the dereference expression into a register if necessary.
 4. Move the right operand into the indirect memory location given by the left operand. This step is where the dereference operation is actually performed.
- Again, a type cast has already been inserted to truncate or extend the right operand.

Example 1: Assignment

- Generate code for $x = y + z$.

```
movl    y, %eax
addl    z, %eax
movl    %eax, x
```

- Generate code for $*p = y + z$.

```
movl    y, %eax
addl    z, %eax
movl    p, %ecx
movl    %eax, (%ecx)
```


- What if all the operands are not the same size?

Example 2: Assignment

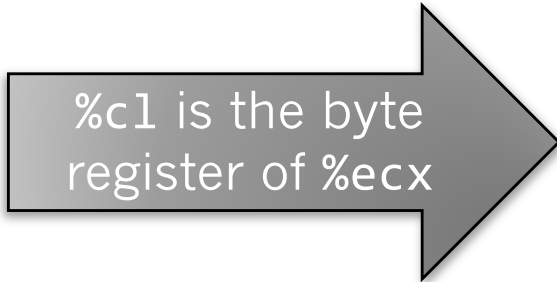
- Generate code for the following code snippet:

```
char *p, c;  
int *q, x;  
  
*p = *(q + c) + x;
```

```
movsbl  c, %eax  
imull   $4, %eax  
movl    q, %ecx  
addl    %eax, %ecx  
movl    (%ecx), %ecx  
addl    x, %ecx  
movl    p, %eax  
movb    %cl, (%eax)
```



Advance by
objects not bytes



%cl is the byte
register of %ecx

- Note that my implementation does the following:
 - Coercions are represented as explicit type casts;
 - Adjustments in pointer arithmetic are made explicit.

Logical Operators

- The logical operators in C are **short-circuiting**.
 - If the left operand of a logical-OR is true, the right operand is not evaluated.
 - If the left operand of a logical-AND is false, the right operand is not evaluated.
- Like the comparison operators, these operators yield a 1 or 0 “truth value” as a result.
- However, we do not always evaluate both operands.
- Therefore, they are more like conditional statements.

Short-Circuiting Code

- The expression $E_1 \ || \ E_2$ is equivalent to:

```
if (E1 != 0)
    result = 1;
else if (E2 != 0)
    result = 1;
else
    result = 0;
```

- The expression $E_1 \ \&\& \ E_2$ is equivalent to:

```
if (E1 == 0)
    result = 0;
else if (E2 == 0)
    result = 0;
else
    result = 1;
```

Example: Logical Or

- Generate code for `*p || y + z`.

```
        movl    p, %eax
        movl    (%eax), %eax
        cmpl    $0, %eax
        jne     .L1
        movl    y, %eax
        addl    z, %eax
        cmpl    $0, %eax
        jne     .L1
        movl    $0, %eax
        jmp     .L2
.L1:
        movl    $1, %eax
.L2:
```

While Statements

- The statement `while (expr) stmt` is translated as:

loop:

```
<code for expr>  
cmp    $0, expr  
je      exit  
<code for stmt>  
jmp     loop
```

1 conditional and 1
unconditional jump

Place each label
on its own line

exit:

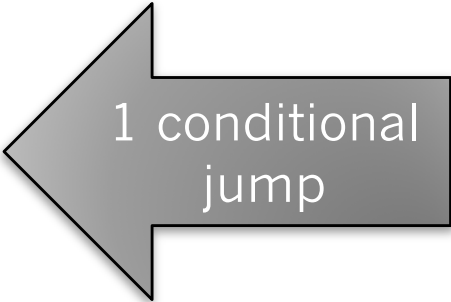
- It's important that each label is placed on its own line in case loops or conditionals are nested.

```
label1: label2:    # won't assemble
```

If Statements

- The statement `if (expr) stmt` is translated as:

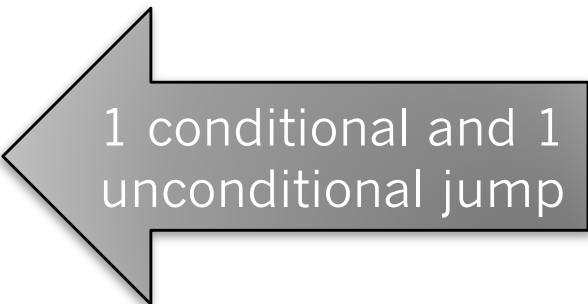
```
    <code for expr>
    cmp    $0, expr
    je     skip
    <code for stmt>
skip:
```



1 conditional jump

- And, `if (expr) stmt1 else stmt2` is translated as:

```
    <code for expr>
    cmp    $0, expr
    je     skip
    <code for stmt1>
    jmp     exit
skip:
    <code for stmt2>
exit:
```



1 conditional and 1 unconditional jump