ASSEMBLY CODE STUDY

ADDRESSING MODES

1. The mov instruction — Move

- The mov instruction copies the data item referred to by its second operand into the location referred to by its first operand.
- The mov instruction that is used for moving data from one storage space to another. The mov instruction takes two operands.

Syntax

mov <reg>, <reg>

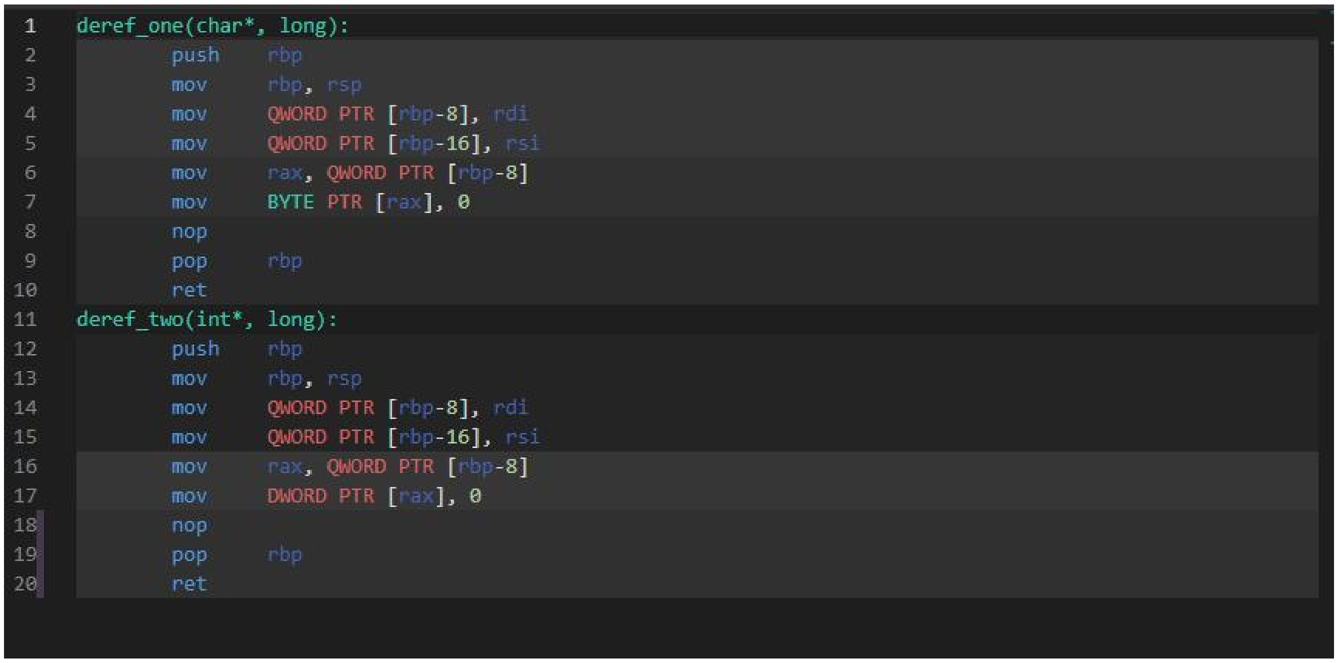
mov <reg>, <mem>

mov <mem>, <reg>

mov <reg>, <const>

mov <mem>, <const>

2. There is one difference between the assembly instruction sequences of the two functions. What is the difference? Why it is different?

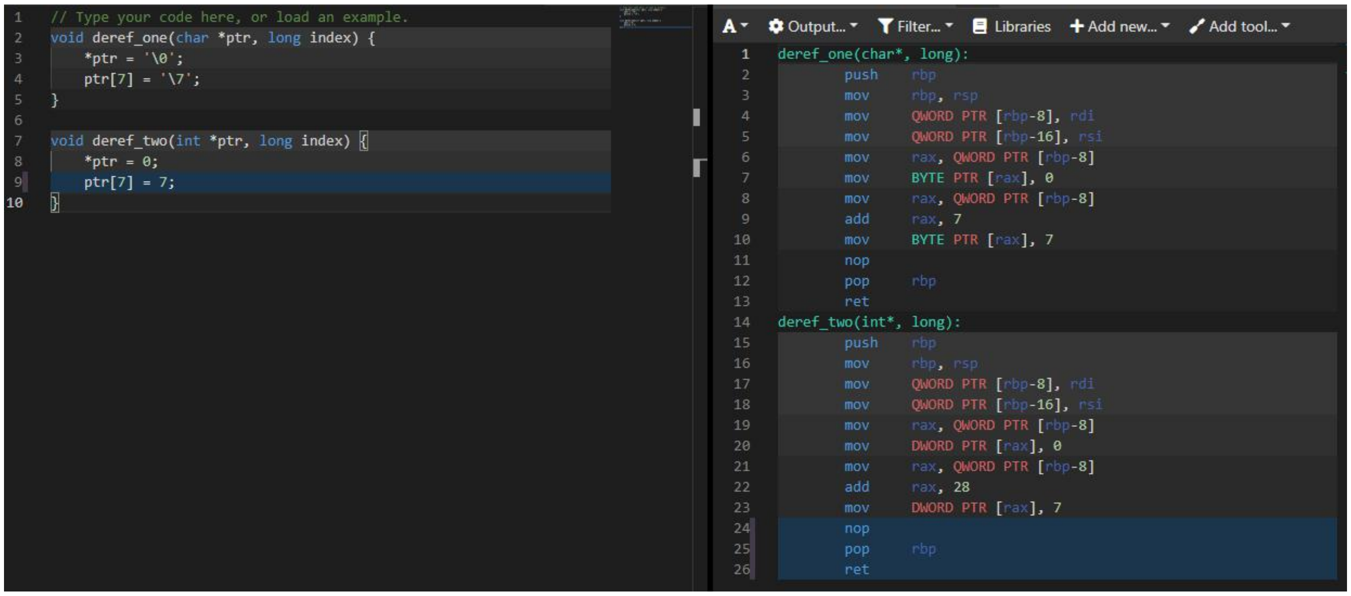


```
deref_one(char*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        mov     BYTE PTR [rax], 0
        nop
        pop     rbp
        ret
deref_two(int*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax], 0
        nop
        pop     rbp
        ret
```

There is a difference between lines 7 and 17. Because both the operands in mov operation should be of same size.

deref_one(char*, long):	deref_two(int*, long):
mov BYTE PTR [rax], 0	mov DWORD PTR [rax], 0
The char data type stores single-byte. BYTE has 1 byte addressed.	The int data type stores 4 bytes. DWORD has 4 bytes address.

3. Edit both functions to instead assign ptr[7] to their respective values. How does this change the addressing mode using for the destination operand of the mov instruction? Do both deref functions change in the same way?

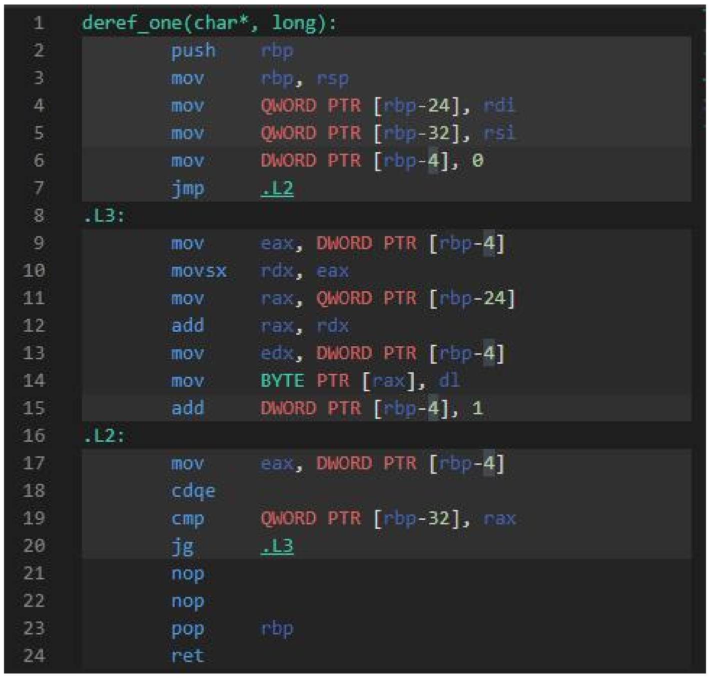

```c
// Type your code here, or load an example.
void deref_one(char *ptr, long index) {
    *ptr = '\0';
    ptr[7] = '\7';
}

void deref_two(int *ptr, long index) {
    *ptr = 0;
    ptr[7] = 7;
}
```

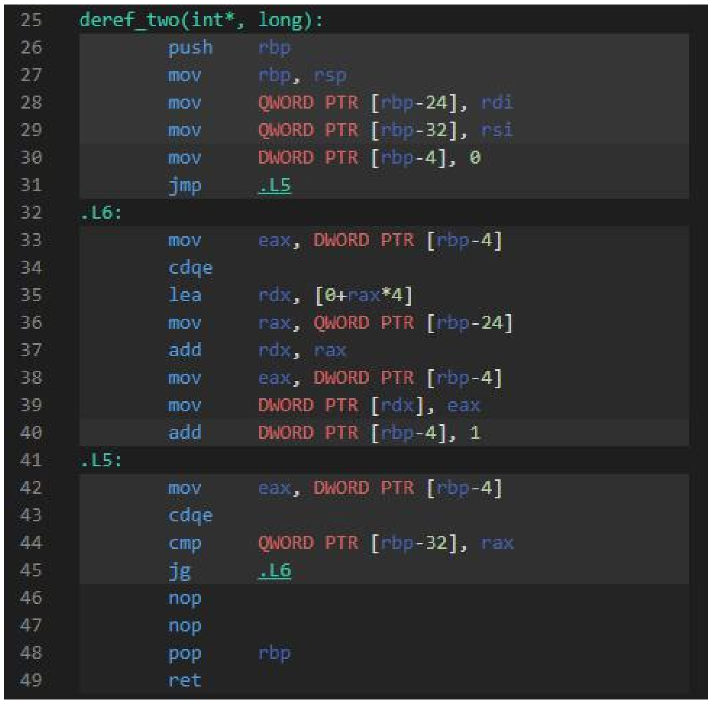
```
1   deref_one(char*, long):
2           push    rbp
3           mov     rbp, rsp
4           mov     QWORD PTR [rbp-8], rdi
5           mov     QWORD PTR [rbp-16], rsi
6           mov     rax, QWORD PTR [rbp-8]
7           mov     BYTE PTR [rax], 0
8           mov     rax, QWORD PTR [rbp-8]
9           add     rax, 7
10          mov     BYTE PTR [rax], 7
11          nop
12          pop     rbp
13          ret
14  deref_two(int*, long):
15          push    rbp
16          mov     rbp, rsp
17          mov     QWORD PTR [rbp-8], rdi
18          mov     QWORD PTR [rbp-16], rsi
19          mov     rax, QWORD PTR [rbp-8]
20          mov     DWORD PTR [rax], 0
21          mov     rax, QWORD PTR [rbp-8]
22          add     rax, 28
23          mov     DWORD PTR [rax], 7
24          nop
25          pop     rbp
26          ret
```

The addressing mode changes from indirect to indirect with offset to take account of subsequent indexing. The scale factor is expressed in bytes, depending on the type of pointer used. deref_one has a char* so it's 1 x 7, while deref_two has an int * so it's 4 x 7. Both change identically otherwise only the scale factor is different.

4. Edit both functions to now assign ptr[index] to their respective values. How does this change the addressing mode using for the destination operand of the mov instruction? Do both deref functions change in the same way?



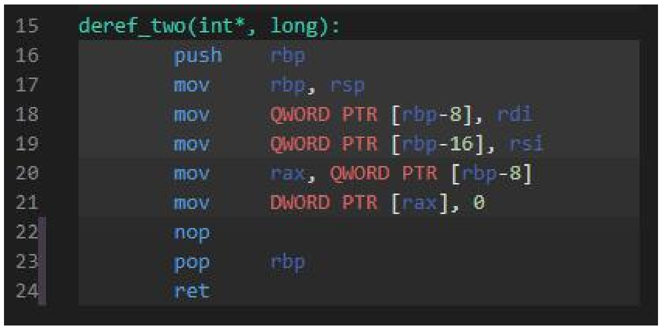
```
deref_one(char*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     QWORD PTR [rbp-32], rsi
        mov     DWORD PTR [rbp-4], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-4]
        movsx   rdx, eax
        mov     rax, QWORD PTR [rbp-24]
        add     rax, rdx
        mov     edx, DWORD PTR [rbp-4]
        mov     BYTE PTR [rax], dl
        add     DWORD PTR [rbp-4], 1
.L2:
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        cmp     QWORD PTR [rbp-32], rax
        jg      .L3
        nop
        nop
        pop     rbp
        ret
```



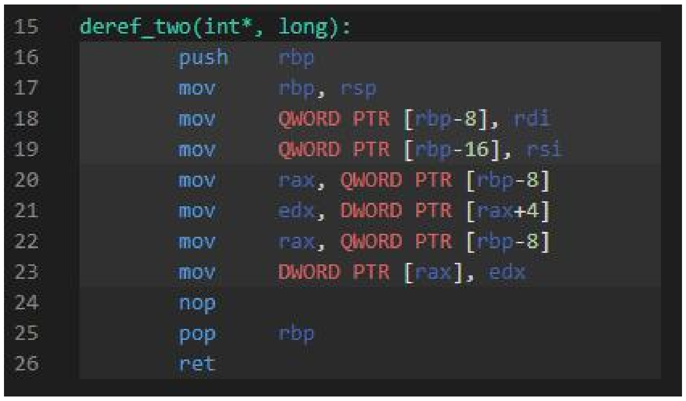
```
deref_two(int*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-24], rdi
        mov     QWORD PTR [rbp-32], rsi
        mov     DWORD PTR [rbp-4], 0
        jmp     .L5
.L6:
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        lea     rdx, [0+rax*4]
        mov     rax, QWORD PTR [rbp-24]
        add     rdx, rax
        mov     eax, DWORD PTR [rbp-4]
        mov     DWORD PTR [rdx], eax
        add     DWORD PTR [rbp-4], 1
.L5:
        mov     eax, DWORD PTR [rbp-4]
        cdqe
        cmp     QWORD PTR [rbp-32], rax
        jg      .L6
        nop
        nop
        pop     rbp
        ret
```

The addressing mode changes to indirect with offset for the first and indirect with offset and graduated index for the second. This is because in both cases we use the index, which is the second parameter, to resize. deref_one is scaled by a multiplier of 1, so it can omit it. deref_two should also be scaled by a multiplier of 4 times the size of an integer. They both change identically otherwise only the scaling is different.

5. Change the assignment statement to ptr[0] = ptr[1]. For both functions, the assembly sequence is one instruction longer. Previously, only one mov instruction was needed. This assignment statement requires two movs. Why?



```
deref_two(int*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax], 0
        nop
        pop     rbp
        ret
```



```
deref_two(int*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rax+4]
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax], edx
        nop
        pop     rbp
        ret
```

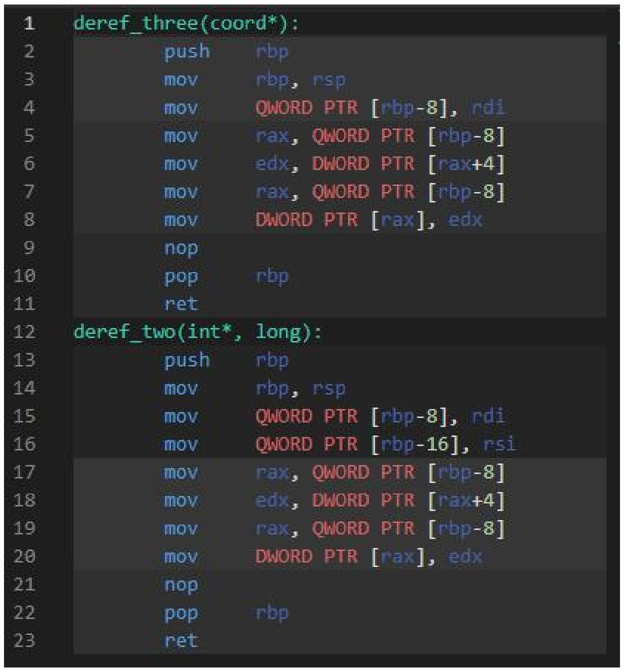
The addressing mode changes to indirect with offset for the first and indirect with offset and scaled index for the second. This is because in both cases we use the index, which is the second parameter, to resize. The second version should also be scaled by a multiplier of 4 times the size of an integer.

6. The mov instructions

- Move instructions are used to copy data between registers and between RAM and registers.

mov eax, [ebx]	Move the 4 bytes in memory at the address contained in EBX into EAX
mov [var], ebx	Move the contents of EBX into the 4 bytes at memory address var.
mov eax, [esi-4]	Move 4 bytes at memory address ESI + (-4) into EAX
mov [esi+eax], cl	Move the contents of CL into the byte at address ESI+EAX
mov edx, [esi+4*ebx]	Move the 4 bytes of data at address ESI+4*EBX into EDX

7. The assembly for deref_three is identical to deref_two, but the C source for the two functions seem to have nothing to do with one another! How is possible that both can generate the same assembly instructions?

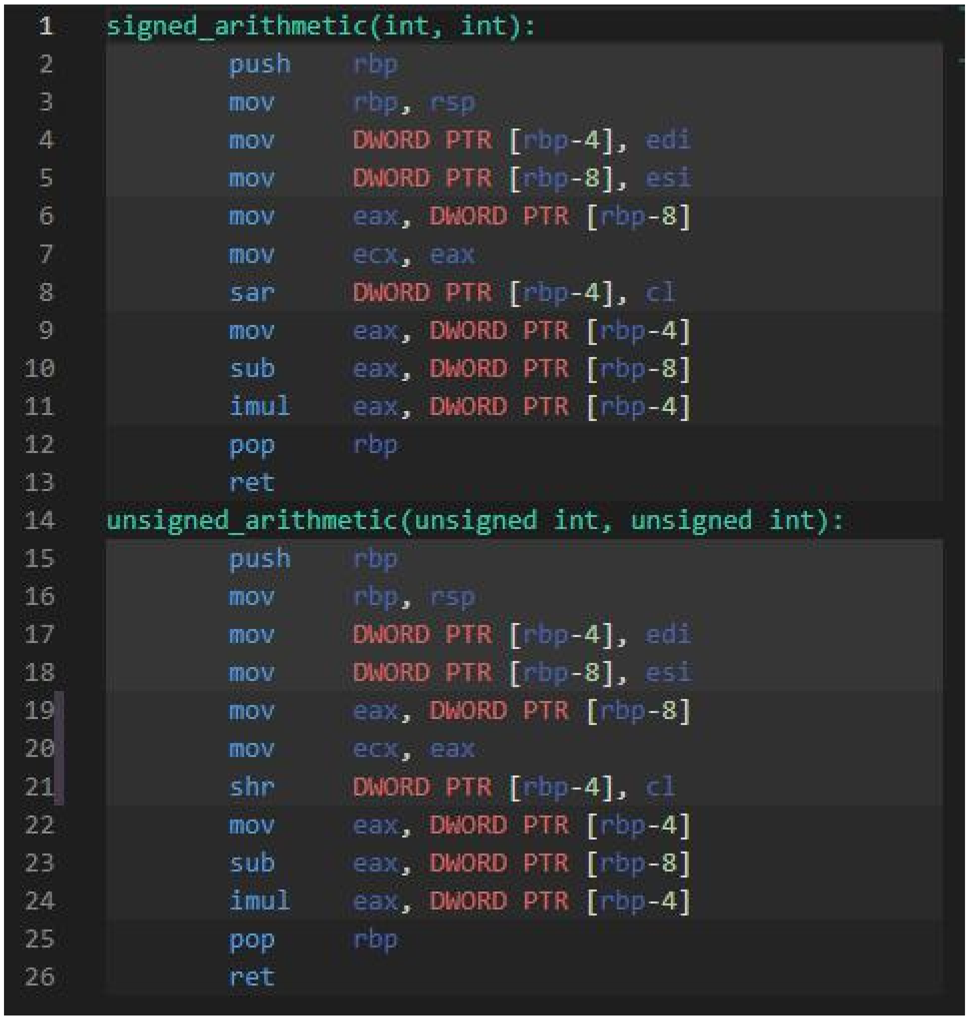


```
deref_three(coord*):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rax+4]
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax], edx
        nop
        pop     rbp
        ret
deref_two(int*, long):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     QWORD PTR [rbp-16], rsi
        mov     rax, QWORD PTR [rbp-8]
        mov     edx, DWORD PTR [rax+4]
        mov     rax, QWORD PTR [rbp-8]
        mov     DWORD PTR [rax], edx
        nop
        pop     rbp
        ret
```

Both perform a similar transfer operation. Both locate 4 bytes addresses from a base, copy 4 bytes from there, and use them to overwrite 4 bytes at the base.

Signed/Unsigned Arithmetic

1. Edit both functions to add the following line at the start: a >>= b;. This performs a right shift on one of its arguments. When doing a right-shift, does gcc emit an arithmetic (sar) or logical (shr) shift? Does it matter whether the argument is signed or unsigned?



```
signed_arithmetic(int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     eax, DWORD PTR [rbp-8]
        mov     ecx, eax
        sar     DWORD PTR [rbp-4], cl
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, DWORD PTR [rbp-8]
        imul    eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
unsigned_arithmetic(unsigned int, unsigned int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     eax, DWORD PTR [rbp-8]
        mov     ecx, eax
        shr     DWORD PTR [rbp-4], cl
        mov     eax, DWORD PTR [rbp-4]
        sub     eax, DWORD PTR [rbp-8]
        imul    eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

The arithmetic shift comes up with the signed types, and the logical shift is used for the unsigned types.

Load Effective Address

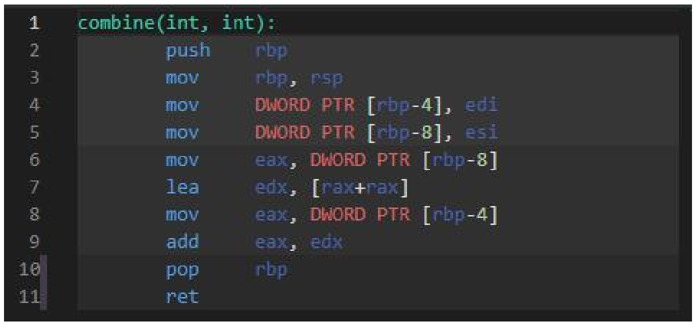
1. The lea instruction — Load effective address

- The lea instruction places the address specified by its second operand into the register specified by its first operand. The contents of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

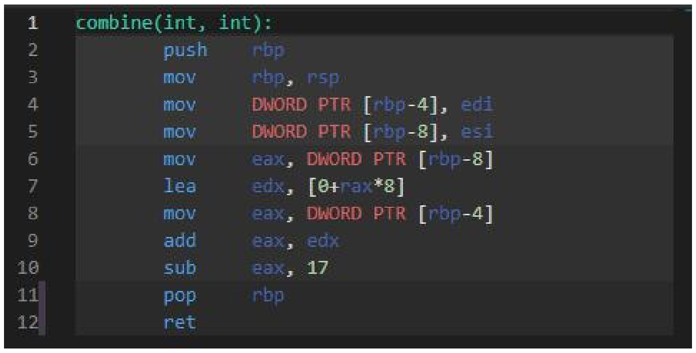
Syntax

lea <reg32>, <mem>

2. Edit the combine function to return x + 2 * y, and then return x + 8 * y - 17, and observe how a single lea instruction can also compute these more complex expressions, such as with multiplication!

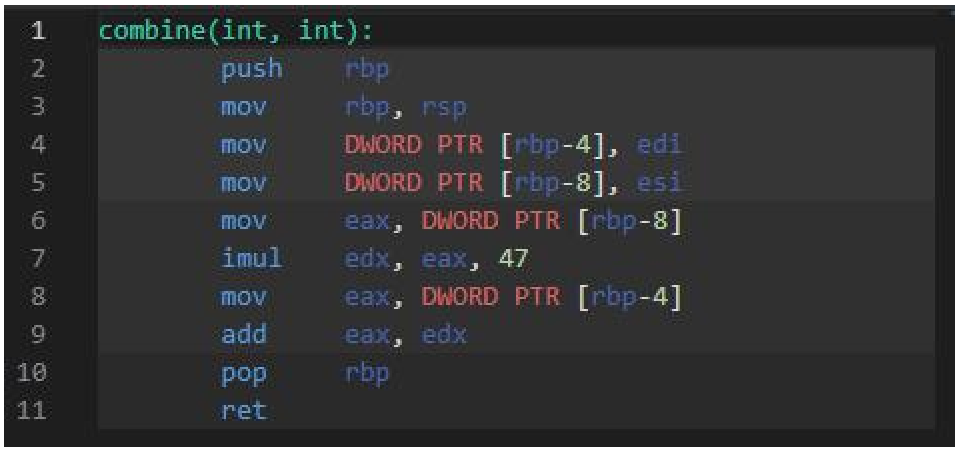


```
combine(int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     eax, DWORD PTR [rbp-8]
        lea     edx, [rax+rax]
        mov     eax, DWORD PTR [rbp-4]
        add     eax, edx
        pop     rbp
        ret
```



```
combine(int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     eax, DWORD PTR [rbp-8]
        lea     edx, [0+rax*8]
        mov     eax, DWORD PTR [rbp-4]
        add     eax, edx
        sub     eax, 17
        pop     rbp
        ret
```

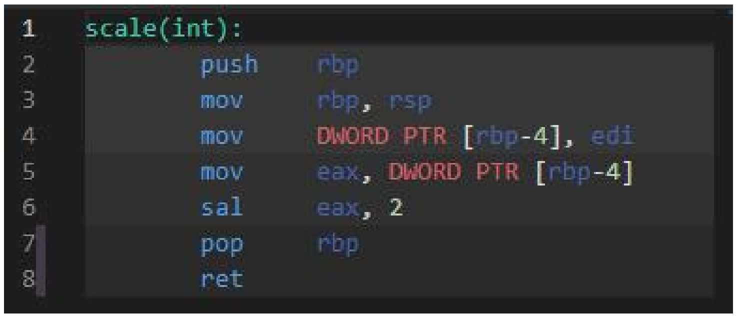
3. Edit the function to now be return x + 47 * y and the result will no longer fit the pattern for an lea because of the multiplication factor. What sequence of assembly instructions is generated instead?



```
combine(int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     eax, DWORD PTR [rbp-8]
        imul    edx, eax, 47
        mov     eax, DWORD PTR [rbp-4]
        add     eax, edx
        pop     rbp
        ret
```

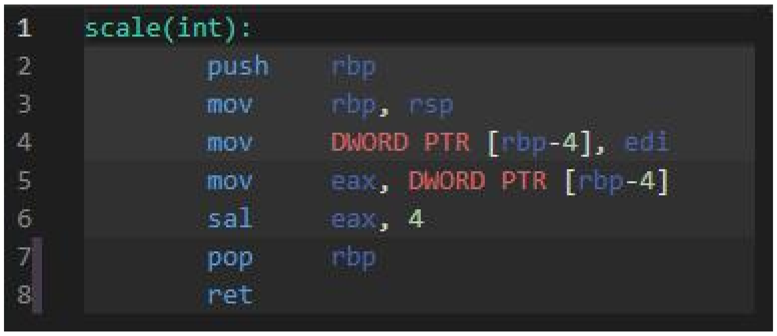
The imul and add instructions are used because 47 is not suitable as a valid scale multiplier for lea.

4. The scale function multiplies its argument by 4. Look at its generated assembly-- there is no multiply instruction. What has the compiler used instead? Edit the scale function to return x * 16. What assembly instruction is used now?



```
scale(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        sal     eax, 2
        pop     rbp
        ret
```

It uses sal. That's because all we're interested in is resizing x by 4. We can do a multiplication by doing a left shift by 2 (x << 2).



```
scale(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        sal     eax, 4
        pop     rbp
        ret
```

It uses sal. We can do a multiplication by doing a left shift by 4 (x << 4).

5. It is perhaps unsurprising that the compiler treats multiplication by powers of 2 as a special case given an underlying representation in binary, but it's got a few more tricks up its sleeve than just that! Edit the scale function to instead multiply its argument by a small constant that is not a power of 2, e.g. x * 3, x * 7, x * 12, x * 17, ... For each, look at the assembly

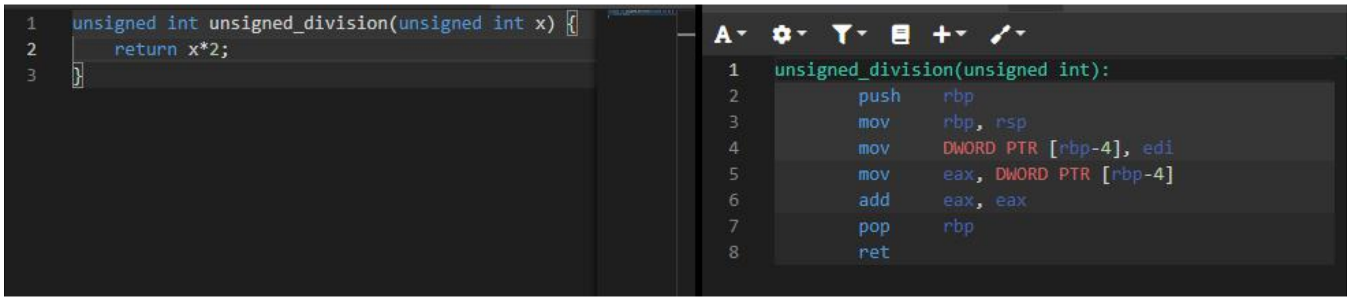
and see what instructions are used. GCC sure goes out of its way to avoid multiply!

For 3, we leverage the reality that x + x * 2 == 3 * x inside a name to lea.

For 17, we leverage the left shift and addition of itself to the left shifted end result through 4.

For 25, we use the reality that temp = x + x * 4; temp + temp * 4 == x * 25.

6. Experiment to find a small integer constant C such that return x * C is expressed as an actual imul instruction



```
unsigned int unsigned_division(unsigned int x) {
    return x*2;
}
```

```
unsigned_division(unsigned int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        add     eax, eax
        pop     rbp
        ret
```

C = 2;

Division

1. In the generated assembly for unsigned_division, there is no div instruction in sight, which is the instruction to perform division. How then did it implement unsigned division by 2? What is the interpretation of a one operand shr instruction?

It uses shr (right shift), and omits the first operand because it is 1. The shr instruction is used to shift the bits of the operand destination to the right, by the number of bits specified in the count operand.

2. Change the function to divide by a different power of 2 instead (e.g. 4 or 8 or 64). What changes in the generated assembly?

The shift amount operand is added since we are shifting by more than 1.

3. What is the bit pattern for 3 and then 3 >> 1? What is the bit pattern for -3 and then -3 >> 1? (Assume arithmetic right shift) Do you see why the right shift (discard lsb) will round a positive value toward zero and rounds a negative value away from zero? This difference in rounding is the crux of why right-shift alone is insufficient if the result is negative.

3 is 0b11, 3 >> 1 is 0b1, which is 1.

-3 is 0b111...1101, -3 >> 1 is 0b111...110, which is -2

4. Compare the assembly for unsigned_division to signed_division. The signed version has a pair of instructions (shr, add) inserted and uses sar in place of shr. First, consider that last substitution. If the number being shifted is positive, there is no difference, but arithmetic right shift versus logical on a negative number has a profound impact. What is different? Why?

We need to do an arithmetic shift because the sign bit needs to be preserved.

5. Now let's dig into the shr and add instructions that were inserted in the signed version. Trace through their operation when the dividend is positive and confirm these instructions made no change in the quotient. But these instructions do have an effect when the dividend is negative. They adjust the negative dividend by a "fixup" amount before the divide (shift). This pushes the dividend to the next larger multiple of the divisor so as to get the proper rounding for a negative quotient.

If the dividend is positive, these instructions did not change the quotient. However, these instructions take effect in the event of a negative dividend. You correct the negative dividend by a “fixed” amount before the division (change). This pushes the dividend to the next higher multiple of the divisor to get a proper rounding for a negative quotient.

6. The necessary fixup amount when dividing by 2 is 1, the fixup for dividing by 4 is 3, for 8 it is 7 and so on. This calculation should be reminiscent of the roundup function we studied way back in lab1! Change the signed_division function to divide by 4 instead. The fixup amount is now 3. The assembly instructions handle the fixup slightly differently than before. It computes the dividend plus fixup and uses a "conditional mov" instruction to select between using the fixed or unmodified dividend based on whether the dividend is negative. This is a new type of mov instruction

in addition to the ones we've already seen. Lecture 12 discusses more about this kind of mov instruction to understand how it is used here.

cmovns S, R is like mov S, R, but only copies S to R when the sign flag is 0. Therefore, in this case, the mov will only happen if %edi is non-negative because of test %edi %edi.