



CS107, Lecture 17

Assembly: Arithmetic and Logic, Take II

Reading: B&O 3.5-3.6

Ed Discussion: <https://edstem.org/us/courses/65949/discussion/5594988>

Assembly Exploration

- Let's pull these commands together and see how some C code might be translated to assembly.
- Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let's check it out!

<https://godbolt.org/z/Ecbde99e3>

Code Reference: calculate

```
int calculate(int x, int arr[]) {  
    int sum = x;  
    sum += arr[0];  
    sum <<= x;  
    sum &= 512;  
    return sum;  
}
```

```
calculate:  
    movl %edi, %ecx  
    movl %edi, %eax  
    addl (%rsi), %eax  
    sall %cl, %eax  
    andl $512, %eax  
    ret
```

Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?
- If you specify two operands to **imul**, it multiplies them together and truncates it to fit in the second of the two 64-bit register operands.

`imul S, D` $D \leftarrow D * S$

- If you specify one operand, it multiplies that by **%rax**, and splits the product across **2** registers. It puts the high-order 64 bits in **%rdx** and the low-order 64 bits in **%rax**.

Instruction	Effect	Description
<code>imulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
<code>mulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide

- Terminology: **dividend / divisor = quotient with remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend need to be prepared and stored in **%rdx**, the low-order 64 bits in **%rax**. The divisor is the only listed operand.
- The quotient is stored in **%rax**, and the remainder in **%rdx**.

Division and Remainder

Instruction	Effect	Description
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide
<code>cqto</code>	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word

- Terminology: **dividend** / **divisor** = **quotient with remainder**
- The high-order 64 bits of the dividend need to be prepared and stored in `%rdx`, the low-order 64 bits in `%rax`. The divisor is the only listed operand.
- Most division uses only 64-bit dividends. The **`cqto`** instruction sign-extends the 64-bit value in `%rax` into `%rdx` to fill both registers with the dividend, as the division instruction expects.



Compiler Explorer Demo

<https://godbolt.org/z/4cT75M4nd>

Code Reference: full_divide

```
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}
```

```
full_divide:
    movq %rdi, %rax
    movq %rdx, %rcx
    cqto
    idivq %rsi
    movq %rdx, (%rcx)
    ret
```

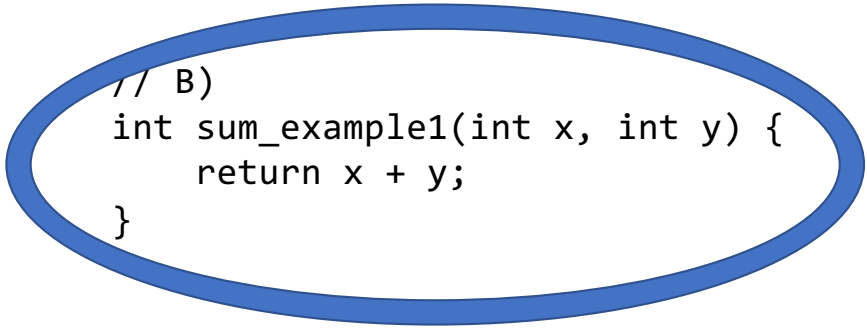

Assembly Exercise 1

```
000000000040116e <sum_example1>:  
40116e: 8d 04 37          lea    (%rdi,%rsi,1),%eax  
401171: c3                retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)  
void sum_example1() {  
    int x;  
    int y;  
    int sum = x + y;  
}
```

```
// C)  
void sum_example1(int x, int y) {  
    int sum = x + y;  
}
```



```
// B)  
int sum_example1(int x, int y) {  
    return x + y;  
}
```

Assembly Exercise 2

```
0000000000401172 <sum_example2>:  
    401172: 8b 47 0c          mov    0xc(%rdi),%eax  
    401175: 03 07            add    (%rdi),%eax  
    401177: 2b 47 18          sub    0x18(%rdi),%eax  
    40117a: c3              retq
```

```
int sum_example2(int arr[]) {  
    int sum = 0;  
    sum += arr[0];  
    sum += arr[3];  
    sum -= arr[6];  
    return sum;  
}
```

What location or value in the assembly above represents the C code's **sum** variable?

%eax

Assembly Exercise 3

```
0000000000401172 <sum_example2>:
    401172: 8b 47 0c          mov    0xc(%rdi),%eax
    401175: 03 07            add    (%rdi),%eax
    401177: 2b 47 18          sub    0x18(%rdi),%eax
    40117a: c3               retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

0x18

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {  
    int sum = ____?____;  
    sum += arr[____?____];  
    return ____?____;  
}
```

```
// x in %edi, arr in %rsi, i in %edx  
add_to:  
    movslq %edx, %rdx  
    movl %edi, %eax  
    addl (%rsi,%rdx,4), %eax  
    ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {  
    int sum = ____?____;  
    sum += arr[____?____];  
    return ____?____;  
}
```

```
// x in %edi, arr in %rsi, i in %edx
```

```
add_to:
```

```
    movslq %edx, %rdx    // sign-extend i into full register  
    movl %edi, %eax      // copy x into %eax  
    addl (%rsi,%rdx,4), %eax // add arr[i] to %eax  
    ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {  
    int sum = x;  
    sum += arr[i];  
    return sum;  
}
```

```
// x in %edi, arr in %rsi, i in %edx
```

```
add_to:
```

```
    movslq %edx, %rdx    // sign-extend i into full register  
    movl %edi, %eax      // copy x into %eax  
    addl (%rsi,%rdx,4), %eax // add arr[i] to %eax  
    ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {  
    int z = nums[___?___] * ___?___;  
    z -= ___?___;  
    z >>= ___?___;  
    return ___?___;  
}
```

```
// nums in %rdi, y in %esi
```

```
elem_arithmetic:
```

```
    movl %esi, %eax
```

```
    imull (%rdi), %eax
```

```
    subl 4(%rdi), %eax
```

```
    sarl $2, %eax
```

```
    addl $2, %eax
```

```
    ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {  
    int z = nums[___?___] * ___?___;  
    z -= ___?___;  
    z >>= ___?___;  
    return ___?___;  
}
```

```
// nums in %rdi, y in %esi
```

```
elem_arithmetic:
```

<pre>movl %esi, %eax</pre>	<pre>// copy y into %eax</pre>
<pre>imull (%rdi), %eax</pre>	<pre>// multiply %eax by nums[0]</pre>
<pre>subl 4(%rdi), %eax</pre>	<pre>// subtract nums[1] from %eax</pre>
<pre>sarl \$2, %eax</pre>	<pre>// shift %eax right by 2</pre>
<pre>addl \$2, %eax</pre>	<pre>// add 2 to %eax</pre>
<pre>ret</pre>	

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {  
    int z = nums[0] * y;  
    z -= nums[1];  
    z >>= 2;  
    return z + 2;  
}
```

```
// nums in %rdi, y in %esi
```

```
elem_arithmetic:
```

movl %esi, %eax	// copy y into %eax
imull (%rdi), %eax	// multiply %eax by nums[0]
subl 4(%rdi), %eax	// subtract nums[1] from %eax
sarl \$2, %eax	// shift %eax right by 2
addl \$2, %eax	// add 2 to %eax
ret	

Our First Assembly

```
int sum_array(int arr[], int nelems) {  
    int sum = 0;  
    for (int i = 0; i < nelems; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

We're 1/2 of the way to understanding assembly!
What looks understandable right now?

0000000000401136 <sum_array>:

401136:	b8 00 00 00 00	mov	\$0x0,%eax
40113b:	ba 00 00 00 00	mov	\$0x0,%edx
401140:	39 f0	cmp	%esi,%eax
401142:	7d 0b	jge	40114f <sum_array+0x19>
401144:	48 63 c8	movslq	%eax,%rcx
401147:	03 14 8f	add	(%rdi,%rcx,4),%edx
40114a:	83 c0 01	add	\$0x1,%eax
40114d:	eb f1	jmp	401140 <sum_array+0xa>
40114f:	89 d0	mov	%edx,%eax
401151:	c3	retq	

