

The examples and discussion in the following slides have been adapted from a variety of sources, including:

Chapter 3 of Computer Systems 3rd Edition by Bryant and O'Hallaron
x86 Assembly/GAS Syntax on WikiBooks
(http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax)

The C code was compiled to assembly with `gcc` version 8.3.1 on CentOS 8.

Unless noted otherwise, the assembly code was generated using the following command line:

```
gcc -S -fno-asynchronous-unwind-tables -mno-red-zone -O0 file.c
```

AT&T assembly syntax is used, rather than Intel syntax, since that is what the `gcc` tools use.

Also:

```
addl rightop, leftop  
      # leftop = leftop - rightop
```

```
subl rightop, leftop  
      # leftop = leftop - rightop
```

```
imull rightop, leftop  
      # leftop = leftop * rightop
```

```
negl op  
      # op = -op
```

```
incl op  
      # op = op + 1
```

```
decl op  
      # op = op - 1
```

Shifting the representation of an integer

```
sall rightop, leftop  
# leftop = leftop << rightop -- C syntax!
```

```
sarl rightop, leftop  
# leftop = leftop >> rightop (preserves sign)
```

```
shll rightop, leftop  
# leftop = leftop << rightop (same as sall)
```

```
shrl rightop, leftop  
# leftop = leftop >> rightop (hi bits set to 0)
```

Shifting an integer operand to the left by k bits is equivalent to multiplying the operand's value by 2^k :

```
sall 1, %eax    # eax = 2*eax
```

```
sall 3, %edx    # edx = 8*edx
```

For example:

```
edx 00000000 00000000 00000000 00000101
```

5

```
edx 00000000 00000000 00000000 00101000
```

40



Since general multiplication is much more expensive (in time) than shifting bits, we should prefer using a shift-left instruction when multiplying by a power of 2.

Shifting an integer operand to the right by k bits might be expected to divide the operand's value by 2^k :

```
shrl 1, %eax           # eax = eax / 2 ?
```

Recall that `shrl` shifts in 0's on the left; so this will indeed perform integer division by 2, provided the value in `eax` is interpreted as an unsigned integer.

For example, if we have an 8-bit unsigned representation of 255_{10} , the instruction above would perform the following transformation:

```
1111 1111    →    0111 1111
```

So it would yield 127_{10} , which is correct for integer division.

But, the following will not yield the correct result for an unsigned integer:

```
sarl 1, %eax           # eax != eax / 2
```

For example, if we consider an 8-bit representation of 200_{10} , the instruction above would produce this transformation:

1100 1000 → 1110 0100

So it would yield 228_{10} , which is incorrect.

The correct result would be 100_{10} which would be represented as 0110 0010.

Note that the correct value would have been found by using `shrl` instead.

Shifting a non-negative (signed) integer operand to the right by k bits will divide the operand's value by 2^k :

```
shrl 1, %eax           # eax = eax / 2
```

```
sarl 1, %eax           # eax = eax / 2
```

If `eax` holds a non-negative signed integer, the left-most bit will 0, and so both of these instructions will yield the same result.

But, if the signed operand is negative, then the high bit will be 1.

Clearly, `shrl` cannot yield the correct quotient in this case. Why?

What about the following instruction, if `eax` holds a negative signed value?

```
sarl 1, %eax           # eax = eax / 2
```

`sarl` replicates the sign bit, so this will yield a negative result...

But, suppose we have an 8-bit representation of -7: 1111 1001

Then applying an arithmetic right shift of 1 position yields: 1111 1100

That represents the value -4... is that correct?

Mathematics says yes by the Division Algorithm:

$$-7 = -4 * 2 + 1$$

Remainders must be ≥ 0 !

C says no:

$$-7 = -3 * 2 + -1$$

$$-7 \% 2 \text{ must equal } -(7 \% 2)$$

There are the usual logical operations, applied bitwise:

```
andl rightop, leftop  
      # leftop = leftop & rightop // C syntax!
```

```
orl rightop, leftop  
     # leftop = leftop | rightop
```

```
xorl rightop, leftop  
     # leftop = leftop ^ rightop
```

```
notl op  
     # op = ~op
```

```
int arith(int x, int y, int z) {
    . . .
}
```

Calling a function causes the creation of a *stack frame* dedicated to that function.

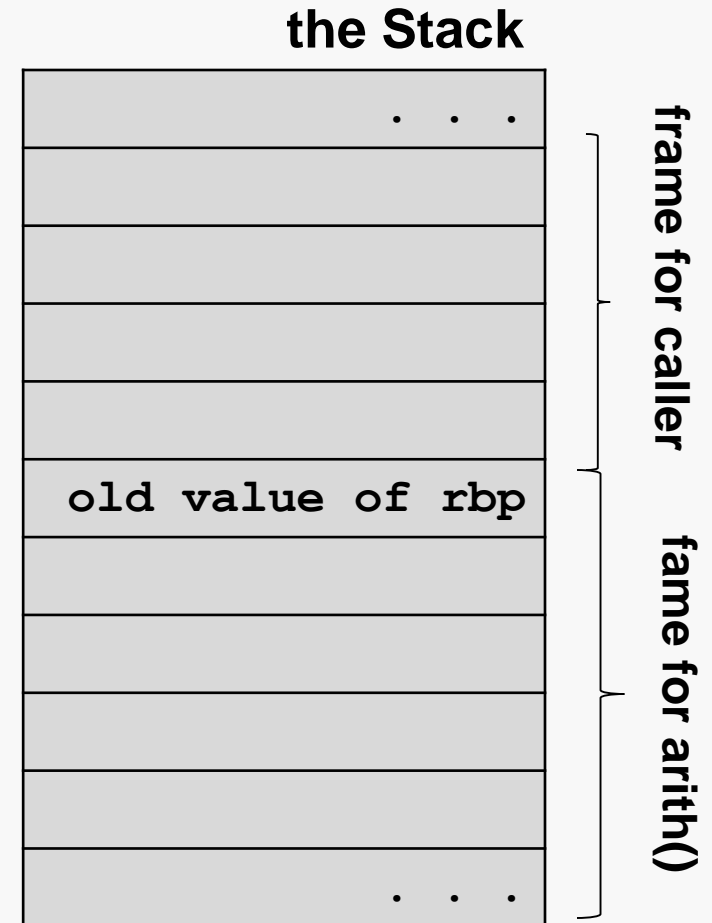
The *frame pointer register*, `rbp`, points to the beginning of the stack frame for the currently-running function.

The *stack pointer register*, `rsp`, points to the last thing that was pushed onto the stack.

(As an optimization, `%rsp` may or may not actually be updated. More on this later).

`rbp`

`rsp`



```

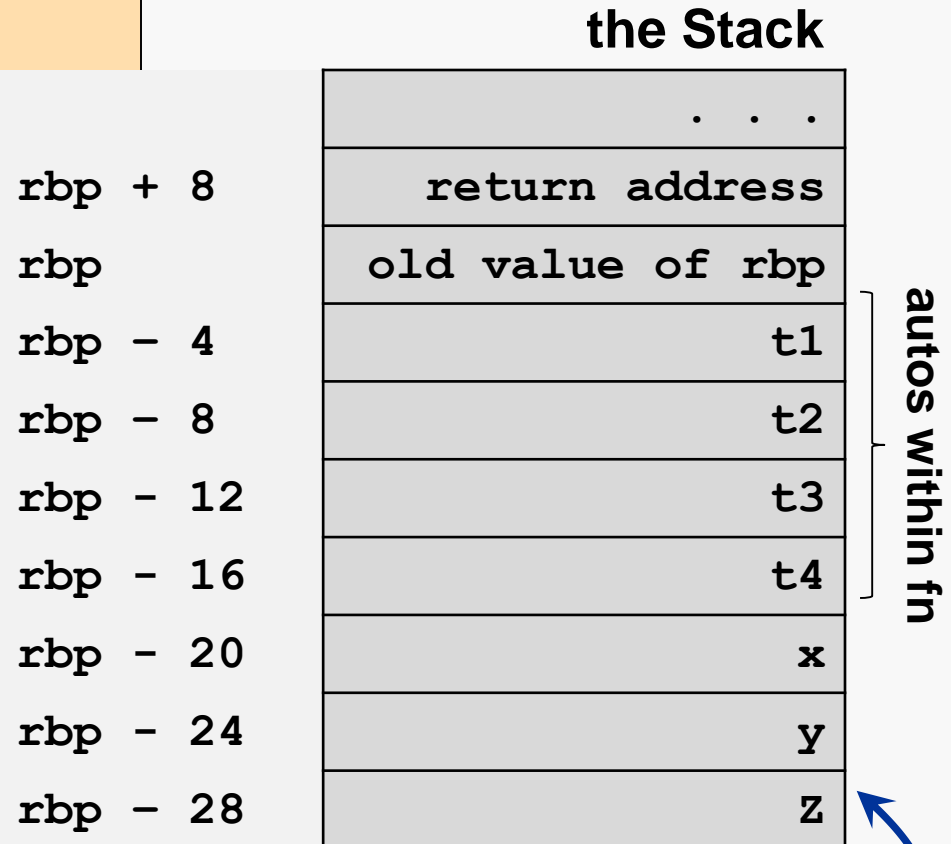
int arith(int x, int y, int z) {
    int t1 = x + y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;
    return t4;
}
    
```

The first 6 function arguments are passed in registers, additional arguments are passed on the stack.

The arguments stored in registers are often moved somewhere else on the stack before any computations.


In this example:

- x is passed in register %edi and is moved to -20 (%rbp).
- y is passed in register %esi and is moved to -24 (%rbp).
- z is passed in register %edx and is moved to -28 (%rbp).



the Stack

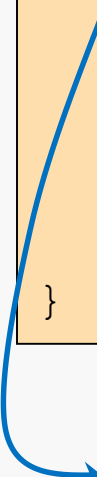
	...	
rbp + 8	return address	8-byte value
rbp	old value of rbp	8-byte value
rbp - 4	t1	4-byte values
rbp - 8	t2	
rbp - 12	t3	
rbp - 16	t4	
rbp - 20	x	
rbp - 24	y	
rbp - 28	z	



```
int arith(int x, int y, int z) {  
  
    int t1 = x + y;  
    int t2 = z*48;  
    int t3 = t1 & 0xFFFF;  
    int t4 = t2 * t3;  
    return t4;  
}
```

Mapping:

	address
x	rbp - 20
y	rbp - 24
t1	rbp - 4



```
movl    -24(%rbp), %eax    # eax = y  
movl    -20(%rbp), %edx    # edx = x  
  
addl    %edx, %eax        # eax = x + y  
  
movl    %eax, -4(%rbp)    # t1 = x + y
```

```
int arith(int x, int y, int z) {  
  
    int t1 = x + y;  
    int t2 = z*48;  
    int t3 = t1 & 0xFFFF;  
    int t4 = t2 * t3;  
    return t4;  
}
```

Mapping:

	address
z	rbp - 28
t2	rbp - 8

```
movl    -28(%rbp), %edx    # edx = z  
movl    %edx, %eax        # eax = z  
addl    %eax, %eax        # eax = z + z = 2z  
addl    %edx, %eax        # eax = 2z + z = 3z  
sall    $4, %eax          # eax = (3z) << 4 = 3z*16 = 48z  
movl    %eax, -8(%rbp)    # t2 = 48z
```

```
int arith(int x, int y, int z) {  
  
    int t1 = x + y;  
    int t2 = z*48;  
    int t3 = t1 & 0xFFFF;  
    int t4 = t2 * t3;  
    return t4;  
}
```

Mapping:

	address
t1	rbp - 4
t3	rbp - 12

```
movl    -4(%rbp), %eax      # eax = t1  
movzwl  $ax, %eax          # eax = t1 & 0xFFFF  
movl    %eax, -12(%rbp)    # t3 = t1 & 0xFFFF
```

You may have noticed the `movzwl` instruction:

```
. . .  
movzwl $ax, %eax          # eax = t1 & 0xFFFF  
. . .
```

This moves a zero extended (z) word (16 bits) stored in `%ax` to `%eax`.

And is equivalent to `t1 & 0xFFFF` since that will zero out the high 16 bits in `%eax` preserving the rest.

We'll see other versions of this instruction later. There are different sizes (`movzb`) and there are signed variants (`movsb`).

In this case, `movzwl` apparently offered a performance (or some other) advantage.


```
int arith(int x, int y, int z) {  
  
    int t1 = x + y;  
    int t2 = z*48;  
    int t3 = t1 & 0xFFFF;  
    int t4 = t2 * t3;  
    return t4;  
}
```

Mapping:

	address
t2	rbp - 8
t3	rbp - 12
t4	rbp - 16

```
movl    -8(%rbp), %eax    # eax = t2  
imull   -12(%rbp), %eax  # eax = t2 * t3  
movl    %eax, -16(%rbp)  # t4 = t2 * t3
```

```

        .file      "arith.c"
        .text
        .globl    arith
        .type     arith, @function
arith:
        pushq    %rbp                # save old frame pointer
        movq     %rsp, %rbp          # move frame pointer to top
        movl     %edi, -20(%rbp)     # move arguments x, y, and z
        movl     %esi, -24(%rbp)
        movl     %edx, -28(%rbp)
        . . .
        movl     -16(%rbp), %eax     # set return value in eax
        popq     %rbp                # rsp = rbp; pop to rbp
        ret                               # return to caller

        .size    arith, .-arith
        .ident   "GCC: (GNU)
        .section .note.G

```

```

int arith(int x, int y, int z) {
    . . .
    int t4 = t2 * t3;
    return t4;
}

```

```
. . .
movl    -24(%rbp), %eax    # eax = y
movl    -20(%rbp), %edx    # edx = x
addl    %edx, %eax        # eax = x + y
movl    %eax, -4(%rbp)    # t1 = x + y

movl    -28(%rbp), %edx    # edx = z
movl    %edx, %eax        # eax = z
addl    %eax, %eax        # eax = z + z = 2z
addl    %edx, %eax        # eax = 2z + z = 3z
sall    $4, %eax          # eax = (3z) << 4 = 3z*16 = 48z
movl    %eax, -8(%rbp)    # t2 = 48z
. . .
```

```
int arith(int x, int y, int z) {

    int t1 = x + y;
    int t2 = z*48;
    . . .
}
```

```
. . .  
movl    -4(%rbp), %eax      # eax = t1  
movzwl  %ax, %eax          # eax = t1 & 0xFFFF  
movl    %eax, -12(%rbp)    # t3 = t1 & 0xFFFF  
  
movl    -8(%rbp), %eax      # eax = t2  
imull   -12(%rbp), %eax    # eax = t2 * t3  
movl    %eax, -16(%rbp)    # t4 = t2 * t3  
. . .
```

```
int arith(int x, int y, int z) {  
  
    . . .  
    int t3 = t1 & 0xFFFF;  
    int t4 = t2 * t3;  
  
    . . .  
}
```