

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)

Using Assembly Language in Linux by Phillip ??

(<http://asm.sourceforge.net/articles/linasm.html>)

The C code was compiled to assembly with `gcc` version 4.8.3 on CentOS 7.

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

```
gcc -S -m64 -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.

The compare instruction facilitates the comparison of operands:

```
    cmpl rightop, leftop
```

The instruction performs a subtraction of its operands, discarding the result.

The instruction sets flags in the *machine status word* register (EFLAGS) that record the results of the comparison:

CF	<i>carry flag</i> ;	indicates overflow for unsigned operations
OF	<i>overflow flag</i> ;	indicates operation caused 2's complement overflow
SF	<i>sign flag</i> ;	indicates operation resulted in a negative value
ZF	<i>zero flag</i> ;	indicates operation resulted in zero

For our purposes, we will most commonly check these codes by using the various jump instructions.

The conditional jump instructions check the relevant EFLAGS flags and jump to the instruction that corresponds to the label if the flag is set:

```
                                # make jump if last result was:
je  label                       # zero
jne label                       # nonzero
js  label                       # negative
jns label                       # nonnegative
jg  label                       # positive (signed >)
jge label                       # nonnegative (signed >=)
jl  label                       # negative (signed <)
jle label                       # nonpositive (signed <=)
ja  label                       # above (unsigned >)
jae label                       # above or equal (unsigned >=)
jb  label                       # below (unsigned <)
jbe label                       # below or equal (unsigned <=).
```

```
gcc -S -m64 -O0 if.c
```

```
. . .  
int y = 5;  
  
if ( x >= 0 ) {  
    y++;  
}  
  
. . .
```

```
. . .  
  
movl    $5, -8(%rbp)  
  
cmpl    $0, -4(%rbp)  
  
js      .L1  
  
addl    $1, -8(%rbp)  
.L1:    . . .
```

```
    . . .  
    movl    $5, -8(%rbp)    # y = 5  
  
    cmpl    $0, -4(%rbp)    # compare x to 0  
  
    js     .L1              # goto .L1 if negative  
  
    addl    $1, -8(%rbp)    # y++  
  
.L1:  
    . . .
```



```
    . . .  
    int y = 5;  
  
    if ( x < 0 ) goto L1;  
  
    y++;  
  
L1:  
    . . .
```

```
. . .  
int y = 5;  
  
if ( x >= 0 )  
    y++;  
else  
    y--;  
. . .
```

```
. . .  
movl    $5, -8(%rbp)  
cmpl    $0, -4(%rbp)  
js      .L4  
addl    $1, -8(%rbp)  
jmp     .L3  
.L4:  
    subl    $1, -8(%rbp)  
.L3:  
. . .
```

```
gcc -S -m64 -O0 ifelse.c
```

```
    . . .
    movl    $5, -8(%rbp)      # y = 5
    cmpl    $0, -4(%rbp)     # compare x to 0
    js     .L4               # goto .L2 if negative
    addl    $1, -8(%rbp)     # y++
    jmp     .L3              # goto .L3 after y++
.L4:
    subl    $1, -8(%rbp)     # y--
.L3:
    . . .
```

```
    . . .
    int y = 5;

    if ( x < 0 ) goto L4;
    y++;
    goto L3;
L4:  y--;
L3:
    . . .
```

```
gcc -S -m64 -O0 ifelse.c
```

```
    . . .
    movl    $0, -8(%rbp)      # y = 0
.L2:
    addl    $1, -8(%rbp)      # y++
    subl    $1, -4(%rbp)      # x--
    cmpl    $0, -4(%rbp)      # compare x to 0
    jg     .L2                # goto .L2 if positive
    . . .
```

```
. . .
int y = 0;

do {
    y++;
    x--;
} while ( x > 0 );
. . .
```

```
gcc -S -m64 -O0 dowhile.c
```



```
    . . .  
    movl    $0, -8(%rbp)      # y = 0  
  
.L2:  
    addl    $1, -8(%rbp)      # y++  
    subl    $1, -4(%rbp)      # x--  
    cmpl    $0, -4(%rbp)      # compare x to 0  
    jg     .L2                 # goto .L2 if positive  
    . . .
```

```
    . . .  
    int y = 0;  
L2:  
    y++;  
    x--;  
    if ( x > 0) goto L2;  
    . . .
```

```
gcc -S -m64 -O0 dowhile.c
```

```
    . . .
    movl  $0, -8(%rbp)    # y = 0
    jmp   .L2            # goto compare x
                          # entry test
.L3:
    addl  $1, -8(%rbp)    # y++
    subl  $1, -4(%rbp)    # x--
.L2:
    cmpl  $0, -4(%rbp)    # compare x to 0
    jg    .L3            # goto loop entry if positive
    . . .
```

```
. . .
int y = 0;

while ( x > 0 ) {
    y++;
    x--;
}
. . .
```

```
gcc -S -m64 -O0 while.c
```

```

    . . .
    movl    $0, -8(%rbp)    # y = 0
    jmp     .L2             # goto compare x to 0
                                # entry test

.L3:
    addl    $1, -8(%rbp)    # y++
    subl    $1, -4(%rbp)    # x--

.L2:
    cmpl    $0, -4(%rbp)    # compare x to 0
    jg      .L3             # goto loop entry if positive

    . . .

```

Note that the compiler translated the C while loop to a logically-equivalent do-while loop.

```
gcc -S -m64 -O0 while.c
```

```

    . . .
    int y = 0;
    goto L2;

L3:
    y++;
    x--;

L2:  if (x > 0) goto L3;

    . . .

```

Let's consider a short assembly function:

```
f:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    20, %rsp
    movl    %edi, -20(%rbp)
    movl    $1, -4(%rbp)
    movl    $2, -8(%rbp)
    jmp     .L2

.L3:
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -4(%rbp)
    addl    $1, -8(%rbp)

.L2:
    movl    -8(%rbp), %eax
    cmpl    -20(%rbp), %eax
    jle     .L3
    movl    -4(%rbp), %eax
    leave
    ret
    . . .
```

This is stack setup code; the compiler creates this; it is not represented in C.

We're going to reconstruct an equivalent function in C.

The first step will be to identify the things that do not translate to C...

This is cleanup and return code; it corresponds to a return statement in C.

The next step will be to identify variables...

```
f:      . . .
      . . .
      movl    %edi, -20(%rbp)
      movl    $1, -4(%rbp)
      movl    $2, -8(%rbp)
      jmp     .L2
.L3:
      movl    -4(%rbp), %eax
      imull   -8(%rbp), %eax
      movl    %eax, -4(%rbp)
      addl   $1, -8(%rbp)
.L2:
      movl    -8(%rbp), %eax
      cmpl   -20(%rbp), %eax
      jle    .L3
      movl    -4(%rbp), %eax
      . . .
```

We're going to reconstruct an equivalent function in C.

The next step will be to identify variables...

Variables will be indicated by memory accesses.

Filtering out repeat accesses yields these assembly statements:

```
f:      . . .
        movl    $1, -4(%rbp)
        movl    $2, -8(%rbp)
        . . .
        cmpl   -20(%rbp), %eax
        . . .
```

There's an access to a variable on the stack at `rbp - 4`; this must be a local (auto) variable. Let's call it `Local1`.

There's another access to a variable on the stack at `rbp - 8`; this must also be a local (auto) variable. Let's call it `Local2`.

A parameter is passed in `%edi` and stored in `rbp - 20`; let's call it `Param1`.

Now we'll assume the variables are all C `ints`, and considering that the first two accesses are initialization statements, so far we can say the function in question looks like:

```
f(int Param1) {  
    int Local1 = 1;  
    int Local2 = 2;  
    . . .  
}
```

And another clue is the statement that stores the value of the variable we're calling `Local1` into the register `eax` (or `rax`) right before the function returns.

That indicates what's returned and the return type:

```
int f(int Param1)  
{  
  
    int Local1 = 1;  
    int Local2 = 2;  
    . . .  
    return Local1;  
}
```

Now, there are two jump statements, a comparison statement, and two labels, all of which indicate the presence of a loop...

```
f:      . . .  
      jmp      .L2  
.L3:   . . .  
.L2:   . . .  
      movl    -8(%rbp), %eax  
      cmpl   -20(%rbp), %eax  
      jle    .L3  
. . .
```

The first jump is unconditional... that looks like a C `goto`.

So, this skips the loop body the first time through...

The comparison is using the parameter we're calling `Param1` (first argument) and we see that the register `eax` is holding the value of the variable we're calling `Local2` (second argument).

Moreover, the conditional jump statement that follows the comparison causes a jump back to the label at the top of the loop, if `Local2 <= Param1`.

What we've just discovered is that there is a while loop:

```
f:      . . .  
      . . .  
      jmp     .L2  
.L3:      . . .  
.L2:      . . .  
      movl   -8(%rbp), %eax  
      cmpl   -20(%rbp), %eax  
      jle    .L3  
. . .
```

```
int f(int Param1) {  
    int Local1 = 1;  
    int Local2 = 2;  
    . . .  
    while (Local2 <= Param1)  
    {  
        . . .  
    }  
    . . .  
    return Local1;  
}
```

The final step is to construct the body of the loop, and make sure we haven't missed anything else...

Here's what's left, including the loop boundaries for clarity:

```
f:      . . .
      . . .
      jmp     .L2
.L3:
      movl   -4(%rbp), %eax
      imull  -8(%rbp), %eax
      movl   %eax, -4(%rbp)
      addl   $1, -8(%rbp)
.L2:
      movl   -8(%rbp), %eax

      cmpl  -20(%rbp), %eax
      jle   .L3
      . . .
```

eax = Local1

eax = Local1 * Local2

Local1 = eax = Local1 * Local2

Local2 = Local2 + 1

And that will do it...

Here's our function:

```
int f(int Param1) {  
  
    int Local1 = 1;  
    int Local2 = 2;  
  
    while (Local2 <= Param1) {  
        Local1 = Local1 * Local2;  
        Local2++;  
    }  
  
    return Local1;  
}
```

So, what is it computing... really?

Let's consider the same function, just lightly optimized using `-O1`:

```
f:
    cmpl    $1, %edi
    jle     .L4
    movl    $2, %edx
    movl    $1, %eax
.L3:
    imull   %edx, %eax
    addl    $1, %edx
    cmpl    %edx, %edi
    jge     .L3
    rep ret
.L4:
    movl    $1, %eax
    ret     . . .
```

The is stack setup code has been omitted. There are only a few locals, and one parameter, so we don't need the stack.

Registers are used instead of the stack.

`%edi` holds Param1.

`%eax` is used as Local1.

`%edx` is used as Local2.

The stack clean up code is also mostly gone. Only the `ret` instruction remains. More on this later.

Reproducing the earlier slide, we have the exact same pieces in fewer steps:

```
f:
    cmpl    $1, %edi
    jle     .L4
    movl    $2, %edx
    movl    $1, %eax

.L3:
    imull   %edx, %eax
    addl    $1, %edx
    cmpl    %edx, %edi
    jge     .L3
    rep ret

.L4:
    movl    $1, %eax
    ret     . . .
```

edx = Local2

eax = Local1

Local1 = eax = Local1 * Local2

Local2 = Local2 + 1

And that will do it...