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

Chapter 3 of Computer Systems 2nd 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 4.5.2 on Ubuntu Linux.

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

```
gcc -S -m32 -00 file.c
```

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

Comparing Operands

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

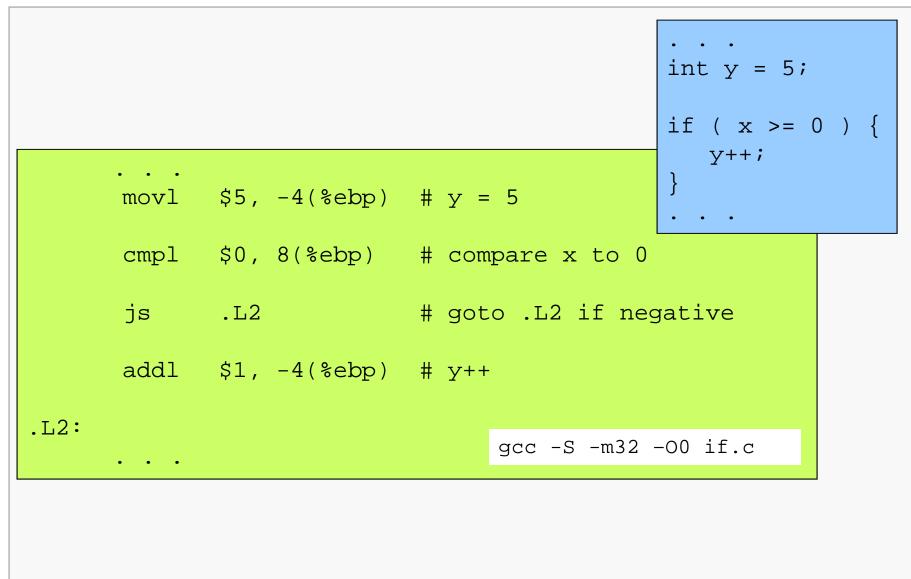
For our purposes, we will most commonly check these codes by using the various jump instructions.

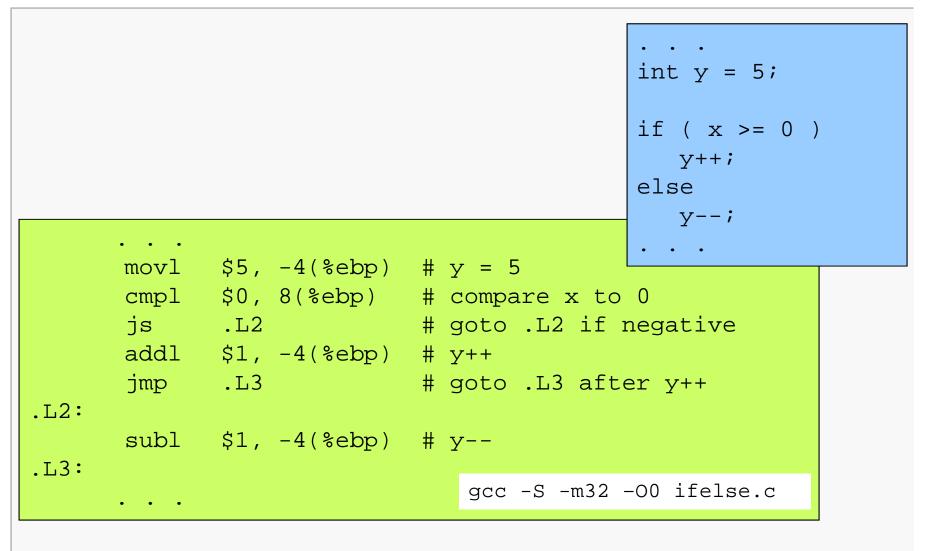
Conditional Jump Instructions

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

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

C to Assembly: if



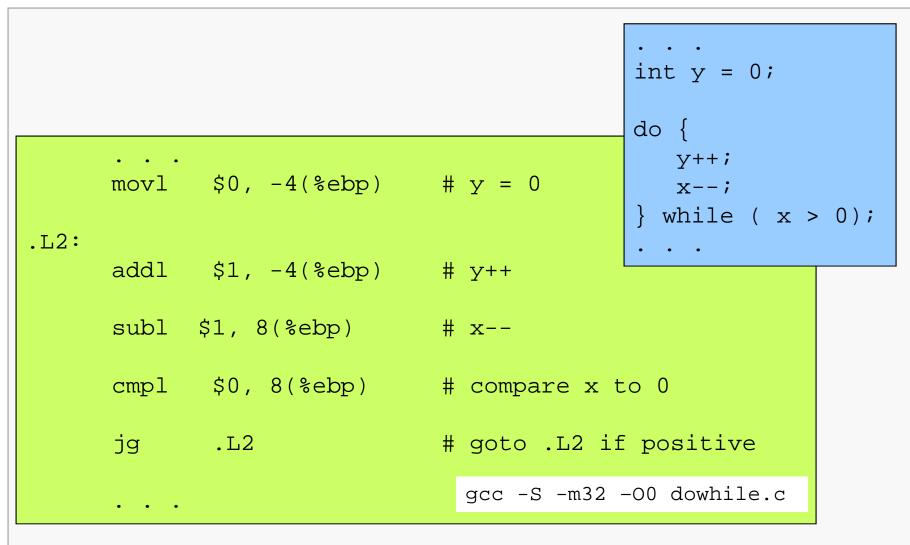


CS@VT

Computer Organization I

©2005-2012 McQuain

C to Assembly: do...while



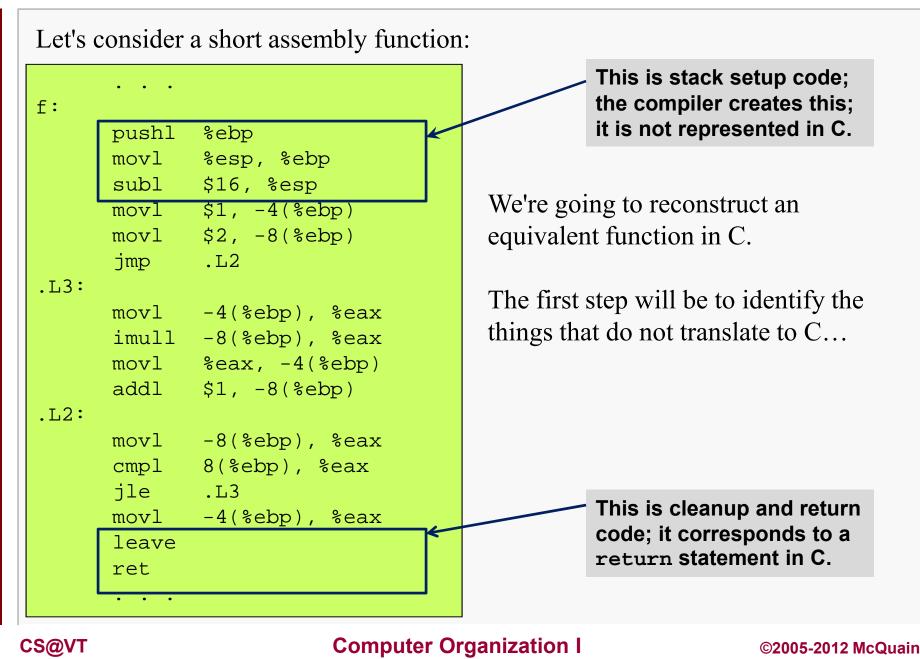
CS@VT

Computer Organization I

©2005-2012 McQuain

C to Assembly: while x86 Assembly 7 Note that the compiler translated the C while int y = 0;loop to a logically-equivalent do-while loop. while (x > 0) { y++; x--; jmp .L2 # goto compare x } # entry testL3: addl \$1, -4(%ebp) # y++ subl \$1, 8(%ebp) # x--.L2: cmpl \$0, 8(%ebp) # compare x to 0 jg .L3 # goto loop entry if positive gcc -S -m32 -O0 while.c • • •





The next step will be to identify variables...

f:		
	 movl movl jmp	\$1, -4(%ebp) \$2, -8(%ebp) .L2
.L3:		
	movl	-4(%ebp), %eax
	imull	-8(%ebp), %eax
	movl	<pre>%eax, -4(%ebp)</pre>
	addl	\$1, -8(%ebp)
.L2:		
	movl	-8(%ebp), %eax
	cmpl	8(%ebp), %eax
	jle	.L3
	movl	-4(%ebp), %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 movl	\$1, -4(%ebp) \$2, -8(%ebp)
	 cmpl	8(%ebp), %eax

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

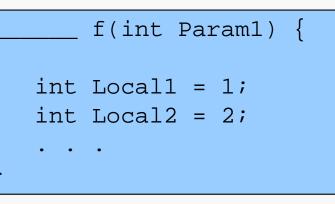
There's another access to a variable on the stack at ebp-8; this must also be a local (auto) variable. Let's call it Local2.

There's an access to a variable on the stack at ebp+8; this must be a parameter passed to the function. Let's call it Param1.



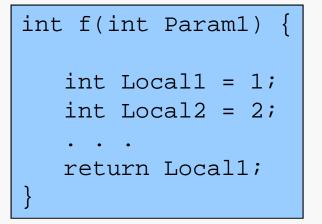
Now we'll assume the variables are all C ints, and considering that the first two accesses are initialization statements, so far we can say the function in

question looks like:



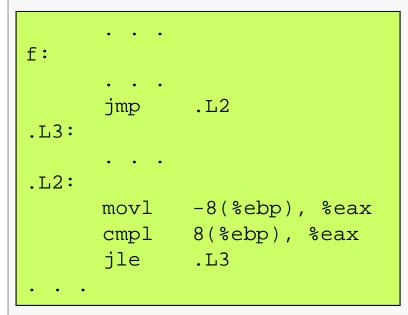
And another clue is the statement that stores the value of the variable we're calling Locall into the register eax right before the function returns.

That indicates what's returned and the return type:





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



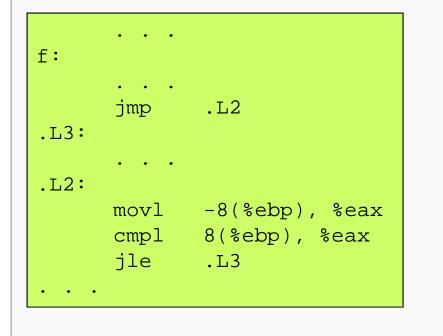
The first jump is unconditional... that looks like a C goto.

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).

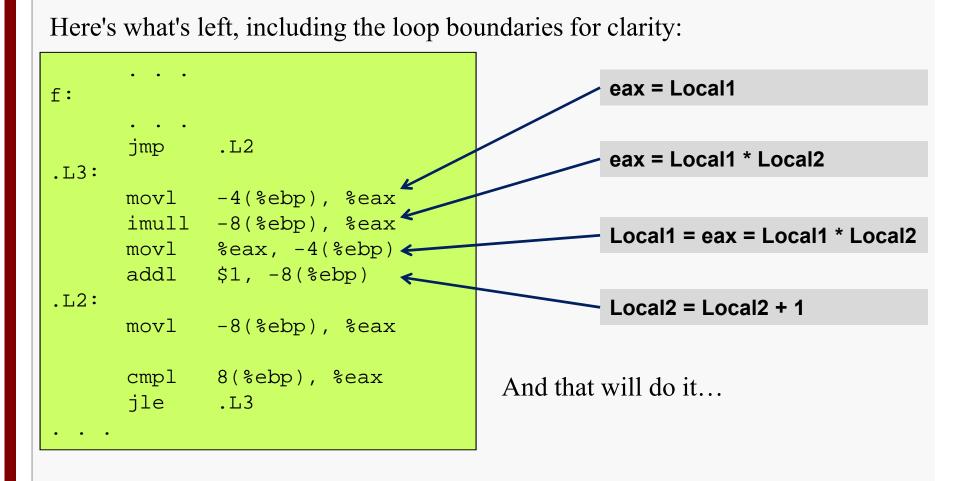
Moreover, the conditional jump statement that follows the comparison causes a jump back to the label at the top of the loop, if Local2 <= Param1.

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



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





CS@VT

Computer Organization I

©2005-2012 McQuain

Here's our function:

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

So, what is it computing... really?