Introduction To Assembler

Dr Carey Pridgeon

# Introduction To Assembler on X86 processors

* Start a C project, in whatever environment you have available called assembler. The only limiting factor is this environment must allow you to enter the compilation commands directly. To complete this worksheet you will need a single C source file, and a text editor. Which one doesn’t matter.

Type in the code C code from the examples below. You will not be typing in the assembly language.

int main() {

return 0;

}

* add a file to your project called main.c. To create it you can either start a new file in your text editor or if you’re on Linux or a Mac, make a new empty file my typing **touch main.c** and opening that. Either method works the same.
* Once you have the c source code listed above in a file, save it, then compile it into assembler by typing the command
* gcc -S main.c
* If all goes well this will finish. The -S tells the compiler you want it to output the compilation results in assembly language, not the native binary used by your computers hardware.
* If you get a compilation error there is a syntax error in your code
* Typing ls will show you have succeeded in generating a file called main.s that contains the assembler version of your c code.
* Read the output by loading main.s into your chosen text editor

.file "main.c"

.text

.globl main

.type main, @function

main:

.LFB0:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

movl $0, %eax

popq %rbp

.cfi\_def\_cfa 7, 8

ret

.cfi\_endproc

.LFE0:

.size main, .-main

.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"

.section .note.GNU-stack,"",@progbits

* This assembler is just setting up the process to run.
* Use this output as a baseline to compare against for further exercises.
* The section that has the new code (beyond fundamental setup and process closing) is:

.cfi\_def\_cfa\_register 6

// new code will be here

movl $0, %eax

* Next we will be typing code in the main function that actually does something, so in the gap above return 0;
* Add this code:

int a = 2;

* Save and compile the code again, then inspect the output as before.

.file "main.c"

.text

.globl main

.type main, @function

main:

.LFB0:

.cfi\_startproc

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

movq %rsp, %rbp

.cfi\_def\_cfa\_register 6

movl $2, -4(%rbp)

movl $0, %eax

popq %rbp

.cfi\_def\_cfa 7, 8

ret

.cfi\_endproc

.LFE0:

.size main, .-main

.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"

.section .note.GNU-stack,"",@progbits

* So now the code

movl $2, -4(%rbp)

* Has appeared.
* This is assembler storing the literal value 2 ($ means literal) in the variable we have decided to call a in one of its 15 counting registers, in the fourth one along specifically and exited.
* So lets retrieve and increment the variable next

a = a+5;

* Now we get, after recompiling main.c:

movl $2, -4(%rbp)

addl $5, -4(%rbp)

* So now we see that the value in -4(%rbp) has had the literal value 5 added to it.
* So, lets add a new variable to the program

int b = 0;

* This now gives us

movl $2, -8(%rbp)

movl $0, -4(%rbp)

addl $5, -8(%rbp)

* So the 8th register in %rbp has been initialised with 0.
* So far we've only worked with literals, lets add a to b.
* add the line

b+=a;

movl $2, -8(%rbp)

movl $0, -4(%rbp)

addl $5, -8(%rbp)

movl -8(%rbp), %eax

addl %eax, -4(%rbp)

* Now the value in -4(%rbp) (our a variable), has been moved into %eax register (an accumulator), from there it has been added to -8(%rbp) (our b variable).

## Subtraction

* Add the code

int c = b - 3;

* Recompile, and our assembly block becomes

movl $2, -12(%rbp)

movl $0, -8(%rbp)

addl $5, -12(%rbp)

movl -12(%rbp), %eax

addl %eax, -8(%rbp)

movl -8(%rbp), %eax

subl $3, %eax

movl %eax, -4(%rbp)

* Of which the new code is

movl -8(%rbp), %eax

subl $3, %eax

movl %eax, -4(%rbp)

* The value in -8(%rbp) is moved into eax
* The literal value 3 is subtracted from it, and the result is stored in the register -4(%rbp) as our new variable c.

## Looping

* In programming we often need to repeat operations. For this we use several forms of loop.
* We will add a simple for loop to our program so we can examine it in assembler.

int i;

for (i=0;i<5;i++) {

c+=2;

}

* Edit your c program adding a new var i to use in the for loop, and a loop that adds 1 to c five times.
* Compile, then view the assembler.
* You will see the places variables are stored has moved. This is because the compiler is deciding where to store things, not us.
* The new block of interest is:

movl $0, -12(%rbp)

jmp .L2

.L3:

addl $2, -16(%rbp)

addl $1, -12(%rbp)

.L2:

cmpl $4, -12(%rbp)

jle .L3

## Looping - Line by Line

movl $0, -12(%rbp)

* This is a for loop, so first line of assembler sets up the loop control var i.

addl $2, -16(%rbp)

jmp .L2

* Loops in assembler work by jumping around the code, using labels to set destination points.
* This loop starts by jumping to label L2.

.L2:

cmpl $4, -4(%rbp)

jle .L3

* At L2 there is a comparison to see whether the loop has ended (is i still less than or equal to 4).

jle .L3

* jle means 'Jump if less than or equal' The jump target is L3, which contains the logic the loop is performing (minimal in this example).

.L3:

addl $2, -8(%rbp)

addl $1, -4(%rbp)

* Here 2 is being added to -8(%rbp) (var c), and one is being added to the iteration varable i -4(%rbp)
* The loop ends when jle returns false (i>4).