CSC 3210 Computer Organization and programming

LAB 4

Introduction to NASM, along with add and call

INSTRUCTIONS FOR LAB 4

Tasks to be done in this lab:

- Examine how assembly language code from the gcc compiler differs from code written for NASM
- Learn about the "mov", "add", and "call" commands
- Learn about how variables are stored in a data section
- Learn about how code is stored in a "text" section
- Compare the outputs from several versions of an assembly language program
- Learn about the return value

ASSIGNMENT FOR LAB 4

- This lab uses the SNOWBALL server. You will need to log in and create a log like you did in previous labs.
- **Remember to turn in a version of the log that has the control characters removed. Also, there are prompts/questions to answer in bold.**
- For your information, NASM is the "Netwide ASseMbler" for x86 CPUs. It is commonly used to program x86-based computers in assembly. This lab uses the nasm program.
- After this lab, we will use the Venus simulator. You will notice that the code in future labs is a bit different from this and the previous labs, and you will likely find it to be simpler.
- After this lab, we will use the Venus assembler for this course unless specifically stated otherwise.

INTRODUCTION TO NASM

- First, we will use the pound-sign ("#") for comments. NASM uses the semi-colon (";") for comments. Assembly language code contains directives, which inform the assembler of what to do, but are not actually assembly language commands since they do not have a machine-language equivalent. Here are some points to observe about the directives.

Venus	gcc output	NASM	Comment
.data	.section .rodata	section .data	The data section
.asciiz, .string	.string "hello world."	db "Hello world", 0	Defining a string
.text	.text	section .text	The code section
.globl	.globl	global	Associated label will be global
	.globl main	global main	Where the code actually starts
	pushq %rbp	push rbp	Put the stack frame reference on the stack. We are going to change rbp.
	movl $.LC0, %edi	mov rdi,fmt	di holds a pointer to the start of the format string
la a1, msg	movq %rsi, -16(%rbp)	mov rsi,msg	a1 holds a pointer to the start of string that "msg" defines si holds a pointer to the start of the string to print
li a0, 4 ecall	call puts	call printf	Call the function that prints the text
li a0, 0	movl $0, %eax	mov rax,0	Put the value 0 into the A (or a0) register
	leave	pop rbp	Restore the rbp value from the start.
.byte		db	8 bit value
.word		dw	32 bit value

INTRODUCTION TO NASM

- There is a 0 (also called a null-character) after the "Hello world" string, which is a convention for strings. We know the start of the string, specified with a label.
- In non-object oriented languages, how do you know the length or end of a string? One way is to encode the end of the string with a special character, and here it is the NUL character (0).
- Given the start of a string, any function can then iterate over the string's characters until it comes to a 0, and it then know that the string's end has been reached.
- Programs often use "main:" as a label, defining where the code starts. While this is mandatory with some assemblers, Venus does not appear to need it.
- Both "puts" and "printf“ are functions to send output to stdout. It's interesting to note that the original lab1.c program specified printf, but that the compiler generated code to call puts instead. Changes like this happen when a compiler optimizes our code for us; the result may in fact be more efficient.
- However, as the programmer you are responsible for your code. If there is some obscure bug on your particular system in puts but not in printf, looking at the higher-level language (HLL) code you might conclude "it cannot be that because my code uses printf". A bug that hides at the HLL level does not hide at the assembly language level.

STEPS TO USE NASM

- Compile the gcc version is done with “**gcc -c lab1.s**”
- Then link it using command "**gcc lab1.o -o lab1**".
- Compile the NASM version with “**nasm -f elf64 hello_64.asm**”
- Then link it with "**gcc hello_64.o -o hello_64**".
- Using command "**-f elf64**" specify the file format for the output.
- On SNOWBALL, omitting the "**-f elf64**" will generate some errors.
- There is an optional "**-l hello_64.lst**" that creates a "listing" file, with both the assembly language instructions and the machine language results.

PART-1

- Write a program for NASM
- Here is an example program from the Kip Irvine book (see the link for more details). It has been adapted to work with NASM.
- You can download this here. Once you have a copy on SNOWBALL, use the "nasm" command to assemble it. Then use the "gcc" command to link it. Then run it.

QUESTIONS

- **Describe what this program does from the "main:" label to the end.**
- **What do you observe when you run it?**
- **Does the program work?**

PART-2

- Now let's look at an expanded version. You can download this here.
- Once you have a copy on SNOWBALL, use the "nasm" command to assemble it. Then use the "gcc" command to link it. Then run it.
- **What do you observe? Does the program work? What does this program do differently from the first one? (Describe what the assembly language commands do.)**
- Compile this again, only this time use "nasm -f elf64 -l AddTwoSum_64_pt2.lst AddTwoSum_64_pt2.asm".
- Use "**cat**" to show the AddTwoSum_64_pt2.lst file.
- **What do you observe in the file?**
- Run the command "**xxd AddTwoSum_64_pt2**" (which creates a hexadecimal dump of the file's contents).
- **What do you observe there, and how does it relate to the .lst file? (Hint: look for the values B8 in the AddTwoSum_64_pt2.lst and b8 in the xxd output.)**

PART-3

- Now we'll see a slightly different version, called AddTwoSum_64_pt3.asm.
- Download it, put it on SNOWBALL, and use the "nasm" command to assemble it then gcc to link it. Then run it.
- **Do you observe any differences between this and AddTwoSum_64_pt2.asm? Use the "diff" command to show the differences between them, then explain what they are.**
- Now run AddTwoSum_64_pt2, and then enter the command **echo $?**
- Next, run AddTwoSum_64_pt3, and then enter the command **echo $?**
- **What do you observe about the output from these two commands? Look up what a "return value" value is under Unix/Linux, describe what it is, and say how it relates to this lab.**
- Be sure to document where you got your answer, and use double-quotes for anything you do not say yourself.

IMPORTANT NOTE:

When turning this in (as with all other labs), you should submit the cleaned-up .txt file, as well as the lab report (i.e. a .pdf file). We will grade the lab based on the report, and look at the .txt file if the report is not clear or if there is a problem.