Lab 4: x86 Crash

CYB633 Spring 2021

Due: Wednesday, March 10 at 11:59:59pm

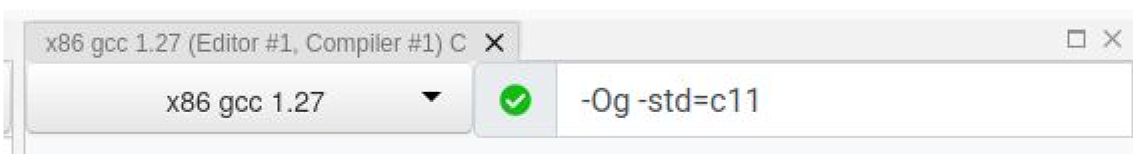
Learning Outcomes

There are specific objectives to this assignment:

- Reinforce your understanding of x86 assembly memory addressing mode and control flow.
- Understand how C programs are translated into assembly and how assembly supports high level language features such as decisions and function calls.
- Reverse-engineer simple assembly instructions to C.

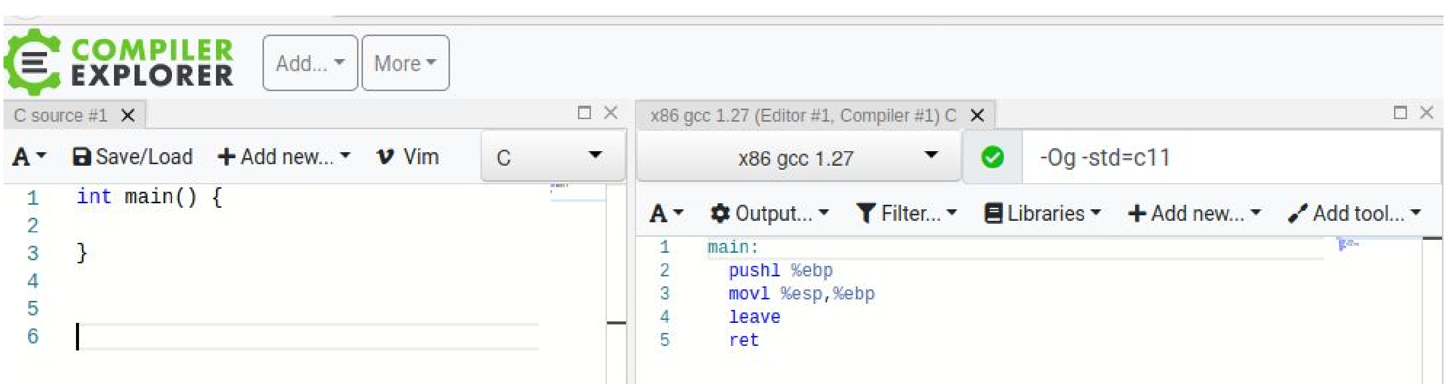
Task 1. Warm-up: Where are my local variables? (25%)

In this lab, we will be using the [Godbolt compiler explorer](), which allows the visitor to compile using a slew of compilers and compare their output to inspect assembly code of C programs. Please choose x86 gcc 1.27 and -0g -std=c11 shown as following:



One reason we switched to x86 for online instruction mode is because x86 is widely supported by various tool sets and thus saves us overhead of setting up labs remotely. As long as you have a web browser, you will be able to use [Godbolt compiler explorer](). :-)

This task is to help you get familiar with [Godbolt compiler explorer]() and x86 assembly syntax. To start with, simply type an empty main function into the C source editor:



Godbolt compiler explorer generates x86 Assembly code in real-time from the C code you supplied and you may notice that although there is nothing inside the main function, there are still 4 instructions injected into assembly text. Why? These are the instructions of the calling convention generated by your compiler at the beginning and ending of each function. You do not need to understand them for this lab.

Now add code to main(). Because we are using an old gcc compiler to build x86 compiler code which generates simple x86 32bit assembly (why are we using such an old compiler? Because it does not over-optimize assembly code, and it uses more decimals than hex numbers which hopefully makes your life easier. :-/), please make sure all your variables are declared as the first thing inside the function which is required by the older version of compilers.

C source #1 | Save/Load | Add new... | Vim | C

```c
int main() {
    int a, b, c,d;
    a=1;
    b=2;
    c=3;
    d=4;
}
```

Pause after each line-- each line of newly added C code, if error free, will update the generated assembly code right away, which enables you to observe how each line is converted to assembly code dynamically. Now observe the assembly code generated on the right and answer the following questions in the lab2_questions.txt using your assembly skills:

1. What instruction is "`int a, b, c, d;`" converted to?
2. Does "`int a, b, c, d;`" shrink or grow the stack? How many bytes are allocated on stack for a, b, c, and d? Explain them using instruction from question 1.
3. What instruction is "`b=2;`" converted to? How is the address of variable b formed in the instruction?
4. Based on the instructions converted from four assignments, can we claim the declaration order determines the allocation order? For instance, a is declared before b, does it mean a is to be allocated before b on stack? Verify your theory by changing the declaration to "`int a, c, b, d;`"

[What to submit]

Task1_questions.txt should be submitted to Gradescope.

Task 2. Arithmetic Expressions (25%)

Copy code from lab4/task2.c to Godbolt compiler explorer. Please observe generated x86 assembly code and answer the following questions in task2_questions using your assembly skills:

1. What instructions is “`c=a+b;`” converted to?

Now your instructor has modified the expression `c=a+b;` to assign some other expression to c. Based on the assembly output as follows (which is also in lab4/task2.s), please infer what expression has been assigned to c (c=?). Write down your answer in main.c and verify it using compiler explorer.

x86 gcc 1.27 (Editor #1, Compiler #1) C ✕

x86 gcc 1.27 ▾ | -Og -std=c11

A ▾ Output... ▾ Filter... ▾ Libraries ▾ + Add new... ▾

```
main:
  pushl %ebp
  movl %esp,%ebp
  subl $12,%esp
  pushl %ebx
  pushl %esi
  movl $13,-4(%ebp)
  movl $5,-8(%ebp)
  movl -4(%ebp),%eax
  cltd
  idivl -8(%ebp)
  movl %edx,%esi
  movl %eax,%ebx
  movl -8(%ebp),%ebx
  subl %esi,%ebx
  movl -4(%ebp),%eax
  imull %ebx,%eax
  movl %eax,-12(%ebp)
  leal -20(%ebp),%esp
  popl %esi
  popl %ebx
  leave
  ret
```

Please note that cltd converts the signed long in EAX to a signed double long in EDX:EAX by extending the most-significant bit (sign bit) of EAX into all bits of EDX.

[What to submit] task2_questions and task2.c.

Task 3. Control-Flow(25%)

Copy code from lab4/task3.c to [Godbolt compiler explorer](). Please observe generated x86 assembly code and answer the following questions in task3_questions using your assembly skills:

1. What instructions evaluate “`a>6`”?
2. What happens if we delete the “`jmp`” instruction?
3. Can you rewrite the condition with “`jg`” in place of “`jle`”? Why or why not?

Now your instructor has modified the code. Based on the assembly output as follows (which is also in lab4/task3.s), please reverse-engineer the assembly code to C code and write down your answer in task3.c and verify it using compiler explorer.

x86 gcc 1.27 (Editor #1, Compiler #1) C

x86 gcc 1.27 -Og -std=c

A Output... Filter... Libraries +

```
main:
  pushl %ebp
  movl %esp,%ebp
  subl $12,%esp
  movl $13,-4(%ebp)
  movl $5,-8(%ebp)
  movl $0,-12(%ebp)
  cmpl $6,-4(%ebp)
  jle .L2
  movl $15,-8(%ebp)
.L2:
.L3:
  cmpl $14,-12(%ebp)
  jg .L4
  movl -12(%ebp),%eax
  addl %eax,-4(%ebp)
  incl -12(%ebp)
  jmp .L3
.L4:
  leave
  ret
```

[What to submit] task3_questions and task3.c in updated by you should be checked in and pushed to github repo.

Task 4. Hard coded passwords are bad, right?(25%)

Copy code from lab4/task4.c to Godbolt compiler explorer. Please observe generated x86 assembly code and answer the following questions in task4_questions using your assembly skills:

1. Let us find the function name "`strcmp`" in the assembly text. What instruction calls `strcmp`?
2. Where is the password "`secret`" stored?

Now your instructor has modified the code. Based on the assembly output in lab4/task4.s, please reverse-engineer the assembly code to C code and write down your answer in task4.c and verify it using compiler explorer.

[What to submit] task4_questions.txt and task4.c updated by you should be checked in and pushed to github repo.