Machine Programming with Assemble

COMP201 Lab Session
Spring 2021

KOÇ UNIVERSITY

GDB Recap

- Gdb is a debugger for C (and C++).
- It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.
- It uses a command line interface.

Debugging using Assembly Language

- Sometimes, debugging is easier when seeing what is happening to the memory registers.
- To go deeper, one must look at Assembly Language code.
- The command in GDB command line: ‘disassemble’ outputs the assembly translation of the function currently being executed, or the translation of a target function if one is supplied.
- For example, *(gdb) disassemble interestCal* gives the equivalent assembly code for the function in exercise 2.

Assembly Language

- Low-level programming language
- Designed for a specific type of processor
- It may be produced by compiling source code from a high-level programming language (such as C/C++)
- It can also be written from scratch.
- Assembly code can be converted to machine code using an assembler.

Assembly Language

- Assembly languages differ between processor architectures
- They often include similar instructions and operators
- Below are some examples of instructions supported by x86 processors:
  - MOV - move data from one location to another
  - ADD - add two values
  - SUB - subtract a value from another value
  - PUSH - push data onto a stack (will be covered in this week’s lectures
  - POP - pop data from a stack (will be covered in this week’s lectures)
  - JMP - jump to another location
  - INT - interrupt a process

Registers

- Registers are data storage locations directly on the CPU
- Usually, the size, or width, of a CPU’s registers define its architecture
- In a 64-bit CPU, the registers will be 64 bits wide
- The same is true of 32-bit CPUs (32-bit registers), 16-bit CPUs, and so on.
- Registers are very fast to access and are often the operands for arithmetic and logic operations.
- rbp and rsp are special purpose registers
    - rbp is the base pointer, which points to the base of the current stack frame
    - rsp is the stack pointer, which points to the top of the current stack frame
    - rbp always has a higher value than %rsp because the stack starts at a high memory address and grows downwards.

Understanding Assembly

Consider the following Assembly code:

```
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -4(%rbp)
movl    -4(%rbp), %eax
imull   -4(%rbp), %eax
popq    %rbp
ret
```

Understanding Assembly

Normally these are the first 2 instructions of all Assembly codes:

```
pushq   %rbp
movq    %rsp, %rbp
```

- The first two instructions are called the function prologue or preamble.
- First we push the old base pointer onto the stack to save it for later.
- Then we copy the value of the stack pointer to the base pointer.
- After this, %rbp points to the base of main‘s stack frame.

Understanding Assembly

```
movl    %edi, -4(%rbp)
```

- The first integer argument is passed in the edi register.
- So this line copies the argument to a local (offset -4 bytes from the frame pointer value stored in rbp).

```
movl    -4(%rbp), %eax
```

- This copies the value in the local to the eax register.

Understanding Assembly

```
imull   -4(%rbp), %eax
```

- Multiply the contents of eax register with eax register

```
popq    %rbp
```

- pop original register out of stack

```
ret
```

- return

Let’s Revisit

```
square:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        imull   -4(%rbp), %eax
        popq    %rbp
        ret
```

Yes, it is just simple squaring function:

```
int square(int num) {

    return num * num;

}
```

Try to understand this!

```
weirdProduct:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -20(%rbp)
        movl    -20(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -8(%rbp)
        cmpl    $2, -20(%rbp)
        jle     .L2
        movl    -20(%rbp), %eax
        subl    $1, %eax
        movl    %eax, -4(%rbp)
.L2:
        movl    -8(%rbp), %eax
        imull   -4(%rbp), %eax
        movl    %eax, -12(%rbp)
        movl    -12(%rbp), %eax
        popq    %rbp
        ret
```

Did you get it right?

```
int weirdProduct(int num1) {
    int x;
    int y;
    x = num1 + 1;
    if (num1 > 2){
        y = num1 - 1;
    }
    int z = x*y;
    return z;
}
```

References

[1] “Assembly Language,” *Assembly Language Definition*. [Online]. Available: https://techterms.com/definition/assembly_language . [Accessed: 21-Nov-2020].

[2] "Understanding C by learning assembly - Blog - Recurse Center", Recurse Center, 2020. [Online]. Available: https://www.recurse.com/blog/7-understanding-c-by-learning-assembly#:~:text=%25rbp%20is%20the%20base%20pointer,memory%20address%20and%20grows%20downwards . [Accessed: 21- Nov- 2020].