Machine Programming with Assembly

COMP201 Lab Session
Fall 2022

KOÇ UNIVERSITY

GDB Recap

- Gdb is a debugger for C (and C++), which allows:
    - Run the program up to a certain point,
    - Pause execution and see the current state,
    - Continue execution step by step

- Higher level debugging
    - Simpler to interpret,
    - but not always useful

- **What if we want to dive deeper?**

Debugging using Assembly Language

- Debugging can be easier if we can see what actually happens under the hood:
  - the individual CPU operations,
  - registers,
  - or the memory.

- To go deeper, one must look at the Assembly 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.

  - `disassemble`
  - `disassemble [Function]`

Assembly

- A 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
- Often similar instructions and operators
- Below are some examples of instructions supported by x86 processors:

  - `mov` - copy data from one location to another
  - `add` - add two values
  - `sub` - subtract a value from another value
  - `push` - push data onto a stack
  - `pop` - pop data from a stack (will be covered later)
  - `jmp` - jump to another execution point
  - `int` - interrupt a process
  - `cmp` - compares two operands

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
}
```

Example 1:

What is the equivalent C code?

```
    cmpq %rbx, %rax
    ja L1
    jmp next
 L1:
    cmpq %rcx, %rbx
    ja L2
    jmp next
 L2:
    movq $1, %rdx
next
```

Example 2:

What is the equivalent C code?

```
    cmpl $0x0A, %eax
    jg end
beginning:
    addl $1, %eax
    cmpl $0x0A, %eax
    jle beginning
end:
```

Example 3:

What is the equivalent C code?

```
    movl $0, %ecx
for:

    cmpl $100, %ecx

    je endfor

    movl $0, %eax

    movl (%edx, %ecx, 4), %eax

    addl $1, %ecx

    jmp for

endfor:
```

Example 4:

Seems familiar?

```
    movl $0, %eax

    movl $1, %ebx

L1:

    movl %eax, %ecx

    addl %ebx, %ecx

    movl %ebx, %eax

    movl %ecx, %ebx

    jmp L1
```

Example 5:

What is the equivalent C code?

```
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)      # -20(rbp) = num1
    movl    -20(%rbp), %eax      # eax = -20(rbp)        # eax = num1
    addl    $1, %eax             # eax += 1
    movl    %eax, -8(%rbp)       # -8(rbp) = eax ------- #  x = num1 + 1
    cmpl    $2, -20(%rbp)        #
    jle     .L2                  # if num1 <= 2, then jump to L2
    movl    -20(%rbp), %eax      # eax = num1
    subl    $1, %eax             # eax -= 1
    movl    %eax, -4(%rbp)       # -4(rbp) = eax         # y = num1 - 1
 .L2:                            #
   movl    -8(%rbp), %eax        # eax = -8(rbp)         # y = x
   imull   -4(%rbp), %eax        # eax *= -4(rbp)        # y *= y_old
   movl    %eax, -12(%rbp)       # -12(rbp) = eax        # = y*y_old = y*x
   movl    -12(%rbp), %eax       # eax = -12(rbp)        # z = ans
   popq    %rbp                  # function end
   ret
```

Example 6:

What is the equivalent assembly code?

```
if (((x < y) && (z > t)) || (a != b)){
    // Some stmt
}
```

Example 7:

Write an assembly code to find the **max** of array of 100 elements