COM 410 SUMMARY

MORITZ M. KONARSKI

Contents

Date: March 29, 2020.

1. Introduction

Summary for the course Computer Architecture and Organization (COM 410) at the American University of Central Asia (https://www.auca.kg) in Bishkek, Kyrgyzstan for the spring semester of 2020. COM 410 is focused on x86_64 assembly language coded on Linux-based machines. This documents is exclusively for my own note taking and summarizing and not for distribution or the like. None of the information herein is my own nor do I claim to be the author of any of it.

1.1. Assembly Language.

Assembly language (**asm**) is a low-level programming language that has a strong correspondence between the instructions in the code and the machine code it is compiled to. Because asm is closely tied to specific architectures and their instructions, each architecture has its own assembly language.

The **assembler** is a program that converts asm into executable machine code. The process is called *assembly*, as in the assembler *assembles* the source code. Assembling turns text-based asm into numerical machine code that can be executed by a CPU. Some assemblers also perform optimizations of jump sizes or instruction ordering for optimal pipelining. Generally, one asm instruction corresponds to one machine instruction (notwithstanding comments, directives, macros, symbolic labels).

1.2. Language Syntax.

In asm, each low-level machine instruction (opcode, specifies which operation is supposed to be performed) is represented by a **mnemonic** (abbreviations of what an instruction does, e.g. `MOV` for "move"). Mnemonics are also used to represent **registers** and **flags**.

Most opcodes require multiple **operands** to become complete instructions, e.g. two registers or variables to move data from one to the other. Assemblers generally permit labels for registers, variables, constants and memory locations to make programs shorter and easier to read and develop.

One architecture can have multiple assemblers with multiple different syntax. For example in x86 assembly using the **Intel syntax** one writes `add eax, [ebx]` while the same code in **AT&T syntax** is `addl (%ebx), %eax`. This course uses the AT&T style of assembly. Both AT&T and Intel syntax generally compile into the same machine code.

1.3. Compilation.

Programs are compiled using `gcc`. Compiling an assembly code file to an executable is done as follows:

```
gcc file.s -o file.out
```

If one starts with a C file, a preprocessed file can be obtained as follows:

```
gcc file.c -o file.i  -E
```

Assembly code can be obtained as follows:

```
gcc file.c -o file.s  -S
```

An object file can be obtained as follows:

```
gcc file.c -o file.o  -c
```

2. Commands

2.1. **call.** Pushes the return address to the stack and branches to the specified procedure. The memory address that is branched to belongs to the first instruction of the called procedure. Leaving a procedure requires `ret`

```
call <function name>
call print_message
```

2.1.2. *plt.* `plt` stands for **procedure linkage table** – basically enables code to call functions regardless of where it sits in virtual memory. Thus the code does not have to store its own version of `printf`, code sharing is enabled.

```
call <function>@plt
call printf@plt
```

2.1.3. *call printf@plt.* Call the c-equivalent function printf.

```
lea <format string>(%rip), %rdi
lea <variable for format print>(%rip), %rsi
# then rdx, r8, r9, xmm0-xmm7, ymm0 - ymm7, zmm0 - zmm7
call printf@plt
```

2.1.4. *call puts@plt.* Call the c-equivalent function puts.

```
lea <string>(%rip), %rdi
call printf@plt
```

2.1.5. *call scanf@plt.* Call the c-equivalent function scanf.

```
lea <format string>(%rip), %rdi
lea <variable for format read>(%rip), %rsi
# then rdx, r8, r9, xmm0-xmm7, ymm0 - ymm7, zmm0 - zmm7
call scanf@plt
```

2.2. **lea.** Stands for **load effective address** basically loads the memory address of the variable into the register.

```
lea <variable>(%rip), %<register>
```

One can also access local variables by specifying how far away a variable is from the current stack pointer. For example 8 bytes away or 0 bytes away.

```
lea 0x8(%rsp), %<register>
lea (%rsp), %<register>
```

2.3. **syscall.** System calls directly call kernel functions without relying on other pieces of code. Using the example of the `write` syscall, `rax` holds the syscall ID, `rdi` specifies where to output, `rsi` holds the memory address of the string, and `rdx` holds the length of the message (size of the char array).

```
mov $0x1, %rax
mov $0x1, %rdi
lea message(%rip), %rsi
mov $message_size, %rdx
syscall
```

2.4. **xor.** Computes a logical exclusive `OR` between two registers or a register and an immediate value. Can be used to set a register to 0.

```
xor %<reg a>, %<reg b>
xor %<reg a>, %<reg a>
xor $<immediate>, %<register>
```

2.5. **ret.** Returns from a procedure by popping the return address off of the stack and branching to it.

```
procedure:
    # code
    ret
```

2.6. **mov.** Move the data at the source to the destination. One can move an immediate value into a register or data from a register to another register, the data from a variable to a register, or local variables to a register by specifying offset from the stack pointer.

```
mov $<immediate>, %<register>
mov %<register>, %<register>
mov $<variable>, %<register>
mov 0x8(%rsp), %<register>
mov (%rsp), %<register>
```

2.7. **dec.** Decrements the register by 1.

```
dec %<register>
```

2.8. **inc.** Increments the register by 1.

```
inc %<register>
```

2.9. **add.** Perform addition of the two operands, storing the result in the second operand. Two registers can be added together or an immediate value can be added to a register.

```
add %<register>, %<register>
add $<immediate>, %<register>
```

2.10. **sub.** Perform subtraction of the two operands, storing the result in the second operand. Two registers can be subtracted or an immediate value can be subtracted from a register.

```
sub %<register>, %<register>
sub $<immediate>, %<register>
```

2.11. **push.** Pushes the data onto the top of the stack.

```
push %<register>
push $<immediate>
```

2.12. **pop.** Pops the first element off of the stack and puts it into the specified destination.

```
pop %<register>
```

2.13. **cltq.** Converts a double word to a quad word, basically an `int` to `long`. This is done as `%eax` to `%rax` with sign extension.

```
cltq
```

2.14. **cmp.** Compares two sources by subtracting the first from the second (reg a - reg b) and setting the condition codes accordingly.

```
cmp %<reg a>, %<reg b>
cmp %<register>, %<immediate>
```

2.15. **test.** Sets the condition codes for the logical `AND` of both sources.

```
test %<reg a>, %<reg b>
cmp %<register>, %<immediate>
```

2.16. **jmp.** Jumps to a label in code without recording any return information.

```
jmp <label>
```

- JG: jmp if greater
- JL: jmp if less
- JGE: jmp if greater or equal (signed)
- JAE: jmp if above or equal (unsigned)
- JLE: jmp if less or equal
- JZ: jmp if zero
- JE: jmp if equal
- JNZ: jmp if not zero
- JNE: jmp if not equal

2.17. **cqo.** Convert quad word to octo word sign extends `rax` into `rdx:rax`.

```
cqo
```

2.18. **mul.** Multiplies `rax` with the specified register. Result is stored in `rdx:rax`. For signed multiply use `imul`. This command can also multiply immediate values with registers.

```
mul %<register>
imul %<register>
```

2.19. **div.** Divides `rdx:rax` by the specified register. Quotient is stored in `rax`, remainder in `rdx`. For signed division use `idiv`

```
div %<register>
idiv %<register>
```

3. Common Constructs

3.1. **Aligning Stack and Frame Pointers.** Push the frame pointer onto the stack. Then move the value of the stack pointer into the frame pointer. This means that the frame pointer now points to the current stack frame. At the end of the program these two steps need to be undone.

```
push %rbp
mov  %rsp, %rbp
# code
mov  %rbp, %rsp
pop  %rbp        # or leave
```

3.2. Local Variables.

This reserves space for local variables that could be needed for `scanf` or for direct use in a program. Sizes should be multiples of the desired variables size, so `0x8` for 64 bits.

```
sub $0x8, %rsp
# code
add $0x8, %rsp
```

3.3. Variable Declaration.

All variable declaration starts with the name and a colon. Variables are declared in the data section `.section .data`.

3.3.1. *Strings.*

The string is surrounded by double quotes.

```
string: .string "Hello, World!"
```

3.3.2. *Ints.*

For integers the initial value can be specified, if one wants 64 bits, two values need to be specified for each long.

```
a: .int 0
b: .int 0, 0
```

3.3.3. *Arrays.*

Specify the data type of the elements and then list them.

```
array: .quad 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Accessing an array is done like this:

```
lea array(%rip), %<reg base>
mov <offset>(%<reg base>, %<reg index>, <element size>), %<reg
    destination>
```

3.4. General Program Structure.

A program generally looks like this:

```
.section .data
# variables go here
.section .text
# functions and labels go here
.global main
main:
    # main program code goes here
    xor %eax, %eax
    ret                 # returns 0 when execution is successful
```

3.5. If Else Statements.

3.6. Switch Statements.

3.7. Modulus.

3.8. Loops.

3.9. Recursion.

4. Lab 1

4.1. **Task 1.**

What is there to do in this task? What's the crux of the issue?