Lecture 2

Instructions and Instruction Set Architecture

In this lecture, we study *computer instructions* and *instruction set architecture* by using *assembly language programming*. We will study *x86-64 architecture* as an example of ISA since it is the most widely used ISA for personal computers.

2.1 x86-64 Architecture

x86-64 architecture is an extended version of the x86 architecture orginally developed AMD. It supports 64-bit computing and is fully compatible with its predecessors (16-bit and 32-bit architectures). It is a *register-memory architecture*.

2.2 General-purpose Integer Registers

A *register* is a *temporary storage* in the processor used for storing intermediate values while conducting calculation.

x86-64 defines 16 integer registers. Each register is 64 bits wide. The range of integer values that can be stored in each register is shown in Table 2.1 and 2.2.

We refer to each register by its name. Table 2.3 shows the register names.

Table 2.1: Range of unsigned 64-bit integer values

Type	Binary value	Unsigned decimal value
Minimum value	$\underbrace{000\ldots000}_{64\text{ bits}}$	0
Maximum value	$\underbrace{111\ldots111}_{64\text{ bits}}$	$2^{64}-1$

Table 2.2: Range of signed 64-bit integer values

Type	Binary value	Signed decimal value
Minimum value	$1\underbrace{00\ldots000}_{63\text{ bits}}$	-2^{63}
Maximum value	$0\underbrace{11\ldots111}_{63\text{ bits}}$	$+2^{63}-1$

Table 2.3: Register Names for x86-64

`%rax`	`%rbx`	`%rcx`	`%rdx`
`%rbp`	`%rsp`	`%rsi`	`%rdi`
`%r8`	`%r9`	`%r10`	`%r11`
`%r12`	`%r13`	`%r14`	`%r15`

2.3 Assembly Language

An *assembly language* is a way to represent a sequence of machine instructions in a human-friendly form. To execute an assembly program, we need to have an *assembler* to convert the instructions into the *machine code*, or the binary representation of the machine instructions.

Here are how to write assembly instructions:

Assembly Instruction	Meaning
`OP B, A`	`A = A op B`
`OP A`	`A = op A`

where *op* represents an operation, A and B are operands which can be *registers*, *constants*, and *memory*.

For example, `add $1, %rax` means `%rax = %rax + 1`

2.3.1 Operands

As stated above, three types of operands can be used in the assembly instructions:

1. **Register** is referred to by its name.
2. **Constant** is any numerical literal preceded with a `$`.
3. **Memory location** can be referred to in several ways. A way to refer to a memory location is called an *addressing mode*.

Table 2.4 shows some examples of assembly instructions.

2.3.2 Instructions

Table 2.5 shows basic integer instructions in x86-64.

Table 2.4: Example of Operands in Assembly Instructions

Type	Instruction	Meaning
Register	`add %rax, %rcx`	`%rcx = %rcx + %rax`
Decimal constant	`add $4, %rcx`	`%rcx = %rcx + 4`
Hexadecimal constant	`add $0xf, %rcx`	`%rcx = %rcx + 15`
Binary constant	`add $0b1110, %rcx`	`%rcx = %rcx + 12`
Absolute address	`add 1234, %rcx`	`%rcx = %rcx + Memory[1234]`
Indirect address	`add (%rax), %rcx`	`%rcx = %rcx + Memory[%rax]`
Indirect address	`add 4(%rax), %rcx`	`%rcx = %rcx + Memory[%rax+4]`

Table 2.5: Integer Arithmetic and Logical Instructions

Operation	Instruction	Meaning
Copy	`mov S, D`	`D = S`
Add	`add S, D`	`D = D + S`
Subtract	`sub S, D`	`D = D - S`
Negation	`neg D`	`D = -D`
Multiply	`imul S, D`	`D = D * S` *(`D` must be a register.)*
Divide	`idiv S`	`%rax = %rdx:%rax / S; %rdx = %rdx:rax % S` *(`S` must be a register.)*
Convert	`cqto`	Extend `%rax` to `%rdx:%rax`
Increment	`inc D`	`D = D + 1`
Decrement	`dec D`	`D = D - 1`
Bitwise And	`and S, D`	`D = D & S`
Bitwise Or	`or S, D`	`D = D | S`
Bitwise Xor	`xor S, D`	`D = D ^ S`
Bitwise Not	`not S, D`	`D = ~D`
Left shift	`shl S, D`	`D = D << S`
Unsigned right shift	`shr S, D`	`D = D >>> S`
Signed right shift	`sar S, D`	`D = D >> S`
Push	`push S`	Push S on the system stack
Pop	`pop D`	Pop from the system stack into D

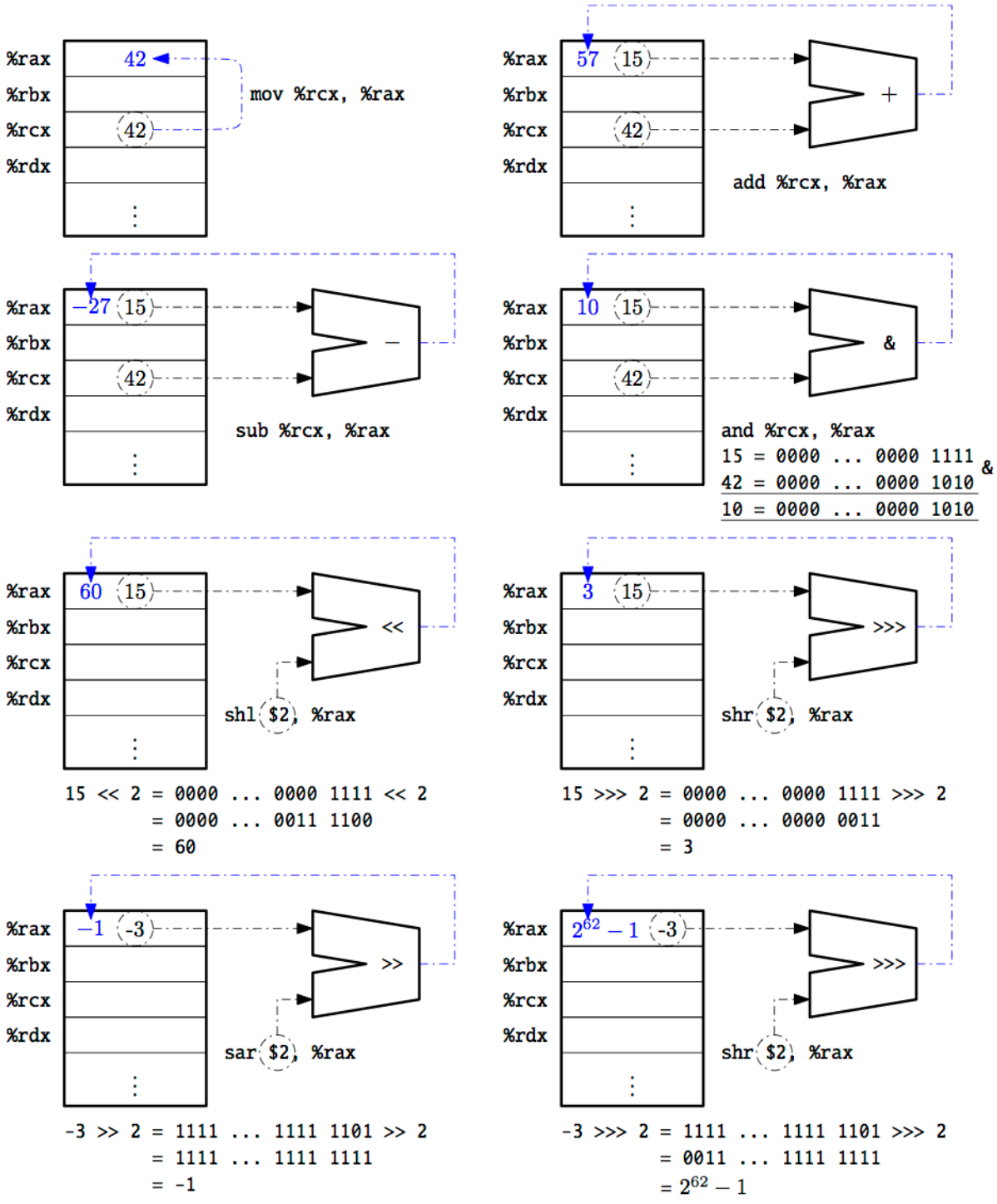

Figure 2.1: Instruction Execution Examples

2.4 Assembly Program

An assembly program mainly composes of a sequence of assembly instructions. However, the processor cannot directly comprehend this type of instructions. We need a program called '*assembler*' to translate assembly instructions into machine code. Each *assembly instruction* can be one-to-one translated into a *machine instruction*.

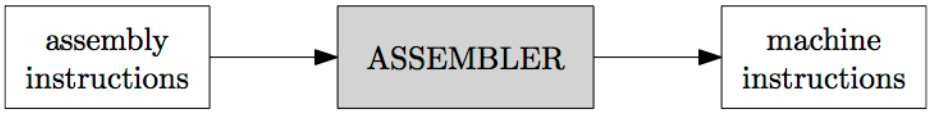

Figure 2.2: Assembler

Each assembly program also includes a number of *assembler directives* to tell the assembler how to translate the program and how to create an executable file that can be run on an operating system. A directive always starts with a dot (`.`).

Table 2.6: Assembler Directives

Directive	Meaning
`.global`	Set a label as a global symbol
`.text`	Start a text section for a sequence of instructions, or read-only data
`.data`	Start a data setion for defining memory locations
`.quad`	Define a 64-bit integer location
`.asciz`	Define a null-terminated character string

An assembly program may include a number of *labels*. Each label is a number ended with a colon (`:`). It represents a memory address of instructions or constants. The assembler also replaces all labels by a memory address after the translation.

Example 2.1 An assembly program that outputs "Hello, world".

```
        .global main
        .text
main:
        mov     $msg, %rdi      # set address of msg as the 1st argument
        call    puts            # call 'puts' function to display a string
        xor     %rax, %rax      # set %rax = 0 (return value)
        ret                     # return (end of program)
msg:    .asciz  "Hello, world"
```

The program is equivalent to the following C program.

```
#include<stdio.h>

int main() {
  puts("Hello, world");
  return 0;
}
```

This is the output that we obtain from the assembler.

```
GAS LISTING hello.s                     page 1


   1                            .global main
   2                            .text
   3                    main:
   4 0000 48C7C700              mov     $msg, %rdi
   4      000000
   5 0007 E8000000              call    puts
   5      00
   6 000c 4831C0                xor     %rax, %rax
   7 000f C3                    ret
   8 0010 48656C6C      msg:    .asciz  "Hello, world"
   8      6F2C2077
   8      6F726C64
   8      00
GAS LISTING hello.s                     page 2


DEFINED SYMBOLS
             hello.s:3      .text:0000000000000000 main
             hello.s:8      .text:0000000000000010 msg

UNDEFINED SYMBOLS
puts
```

Actually, the machine instructions in Example 2.4 still cannot be executed since it calls a C stand library function (`puts`). We need to pass it to a '*linker*' to create an executable program.

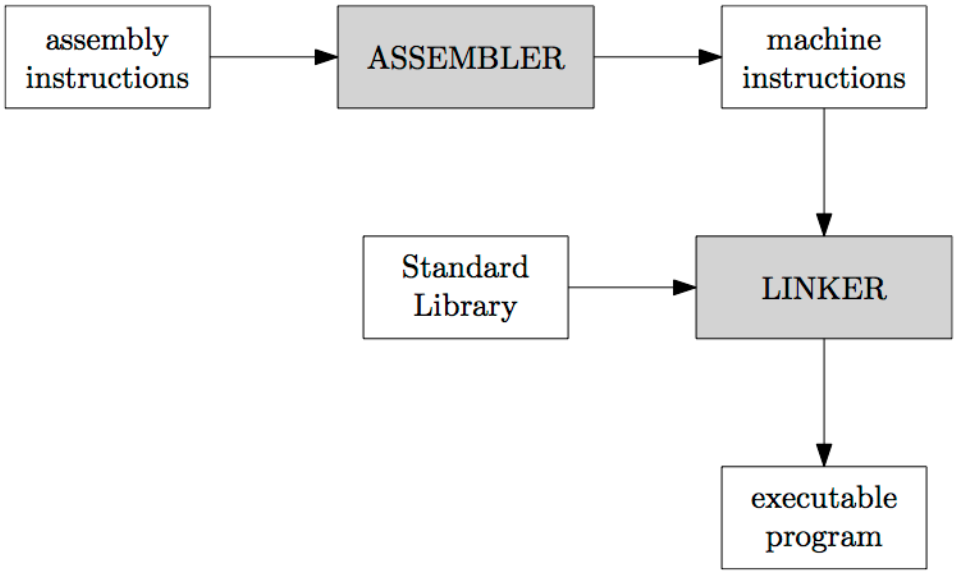

Figure 2.3: Assembler and Linker

2.5 Calling Convention

Since we sometimes use C standard library functions in our assembly programs, we need to follow a standard scheme for calling C functions called '*calling convention*' [Mat+13]. Here are part of them:

- Pass parameters to function via registers in the following order: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`.
- An integer value can be returned from a function via `%rax`.
- The following registers may be changed when we make a function call: `%rax`, `%rcx`, `%rdx`, `%rsp`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10`, `%r11`.
- We need to save the values of the following registers before we make any change: `%rbx`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`.

We study the assembly language in order to understand how machine instructions work. We therefore do not write a stand-alone assembly program, but only implement functions that can be called from C programs in the assembly language.

Example 2.2 Implement a function `sum(a, b, c)` that accepts three integer values and returns the sum of them.

testsum.c

```
#include<stdio.h>

long sum(long a, long b, long c);

int main() {
  long x = sum(1, 2, 3);
  long y = sum(5, 3, 4);

  printf("sum(1, 2, 3) = %ld\n", x);
  printf("sum(5, 3, 4) = %ld\n", y);

  return 0;
}
```

sum.s

```
# long sum(long a, long b, long c);
# according the calling convention:
#       a = %rdi
#       b = %rsi
#       c = %rdx

        .global sum
        .text
sum:
        mov     %rdi, %rax      # %rax = a
        add     %rsi, %rax      # %rax = %rax + b
        add     %rdx, %rax      # %rax = %rax + c
        ret                     # return %rax
```

Exercise 2.1 Implement a function `add1000(a)` that returns `a + 1000`. The function will be used with the following C program.

testadd1000.c

```
#include<stdio.h>

long add1000(long a);

int main() {
  long x;

  print("Enter a number: ");
  scanf("%ld", &x);               // accept an integer from user

  print("Output = %ld\n", add1000(x));
  return 0;
}
```

Exercise 2.2 Implement a function `exercise2(a, b, c)` that returns `c + 2*(a + b) - (b + 2*c)` using with the following C program.

testexercise2.c

```
#include<stdio.h>

long exercise2(long a, long b, long c);

int main() {
  long x, y, z;
  printf("Enter three integers: ");
  scanf("%ld%ld%ld", &x, &y, &z);
  printf("Output = %ld\n", exercise2(x, y, z));
  return 0;
}
```

Example 2.3 Implement a function `ldgt(a)` that returns the last digit of `a`.

testldgt.c

```
#include<stdio.h>

long ldgt(long a);

int main() {
  long x;
  printf("Enter a number: ");
  scanf("%ld", &x);
  printf("Output = %ld\n", ldgt(x));
  return 0;
}
```

ldgt.s

```
        .global ldgt
        .text
ldgt:
        mov     $10, %rcx       # %rcx = 10
        mov     %rdi, %rax      # %rax = a
        cqto                    # extend %rax to %rdx:%rax
        idiv    %rcx            # %rax = %rdx:%rax / %rcx
                                # %rdx = %rdx:%rax % %rcx
        mov     %rdx, %rax      # prepare the return value
        ret
```

Exercise 2.3 Implement a function `exercise3(a, b)` that returns `(a + b) % 2`.

testexercise3.c

```
#include<stdio.h>

long exercise3(long a, long b);

int main() {
  long x, y;
  printf("Enter two numbers: ");
  scanf("%ld%ld", &x, &y);
  printf("Output = %ld\n", exercise3(x, y));
  return 0;
}
```

Exercise 2.4 Implement a function `exercise4(a)` that returns the second last digit of `a`.

testexercise4.c

```
#include<stdio.h>

long exercise4(long a);

int main() {
  long x;
  printf("Enter a number: ");
  scanf("%ld", &x);
  printf("Output = %ld\n", exercise4(x));
  return 0;
}
```

2.6 Control Structures

Each assembly program is just a sequence of instructions. No other structures are provided in the syntax of the language. We can control the instruction execution using two types of instructions, i.e., compare and jump instructions. Table 2.7 shows them.

Table 2.7: Compare and Jump instructions

Instruction	Meaning
`cmp S, D`	Calculate `D - S` and set the *condition flags* without storing the result
`test S, D`	Calculate `D & S` and set the *condition flags* without storing the result; typically used to check the value in a register
`jmp L`	Unconditional jump to `L`
`je L`	Jump to `L` if the previous result $= 0$
`jne L`	Jump to `L` if the previous result $\neq 0$
`jg L`	Jump to `L` if the previous result > 0
`jge L`	Jump to `L` if the previous result ≥ 0
`jl L`	Jump to `L` if the previous result < 0
`jle L`	Jump to `L` if the previous result ≤ 0

2.6.1 Conditional statements

We can combine the compare and jump instructions in order to make the program work in the same manner as the condition statements (`if ... else`).

C Language

```
if (x > 100) {
    //do if true
}
else {
   //do if false
}
```

Assembly Language

```
    cmp $100, %rax
    jg  L1
    # do if false
    jmp LE
L1: # do if true
LE:
```

Figure 2.4: Condition statements in Assembly

Example 2.4 Write a function in assembly that prints `>100` when an integer parameter is greater than 100, otherwise it prints `<=100`.

testg100.c

```
#include<stdio.h>

void g100(long a);

int main() {
  g100(45);
  g100(32090);
  g100(-190);
  g100(100);
  return 0;
}
```

g100.s

```
        .global g100
        .text
g100:
        cmp     $100, %rdi
        jg      LG
        mov     $les, %rdi
        call    puts
        jmp     EXIT
LG:     mov     $gre, %rdi
        call    puts
EXIT:   ret
les:    .asciz  "<=100"
gre:    .asciz  ">100"
```

Example 2.5 Write a function in assembly that checks and displays if an integer parameter is odd or even.

testoddeven.c

```
#include<stdio.h>

void oddeven(long a);

int main() {
  oddeven(10);
  oddeven(11);
  return 0;
}
```

oddeven.s

```
        .global oddeven
        .text
oddeven:
        and     $1, %rdi
        je      LE
        mov     $odd, %rdi
        call    puts
        jmp     EXIT
LE:     mov     $even, %rdi
        call    puts
EXIT:   ret
odd:    .asciz  "It is an odd."
even:   .asciz  "It is an even."
```

All arithmetic instructions also set the condition flags *after conducting the operations.*

2.6.2 Repetition statements

The compare and jump instructions can also be used to form repetition structures.

Example 2.6 Write a function **sumN** that calculates $1+2+\cdots+N$ where N is an argument.

sumN.c

```
long sumN(long N) {
  long i;
  long sum = 0;

  for(i=1; i<=N; i++) {
    sum += i;
  }

  return sum;
}
```

sumN.s

```
        .global sumN
        .text
sumN:
        mov     $1, %rdx        # %rdx = i = 1
        xor     %rax, %rax      # %rax = sum = 0
loop:   cmp     %rdi, %rdx
        jg      done            # go to done if i>N or i-N>0
        add     %rdx, %rax      # sum += i
        add     $1, %rdx        # i++
        jmp     loop            # go to loop
done:
        ret
```

Exercise 2.5 Write a function `maxofthree(a, b, c)` that returns the maximum value among `a`, `b`, and `c`.

Exercise 2.6 Write a function **`grading(x)`** that prints out the grade based on the value of **x**.

$$A = x \geq 90, \qquad B = 60 \leq x < 90, \qquad C = 45 \leq x < 60, \qquad F = x < 45$$

Exercise 2.7 Write a function `fac(n)` that computes the factorial of `n`.

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdots n$$

Exercise 2.8 Write a function `tobin(x)` that prints out a 64-bit binary representation of `x`.

Call `putchar(48)` to print `'0'`, and `putchar(49)` to print `'1'`. Since the function `putchar` will be called inside a loop, be careful that the values of some registers may be changed by the function call.

2.7 Arrays

An *array* is a sequence of values of the same data type. It is stored in the memory as a *consecutive block of memory*. For example, a C statement `long A[5];` defines an array composing of 5 elements. Each element is a 64-bit integer.

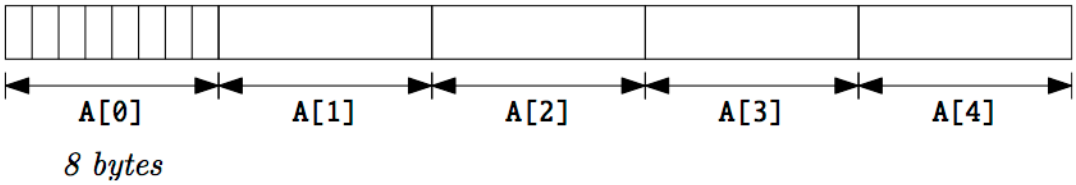

Figure 2.5: Array representation

2.7.1 Accessing array elements

Since each array is a consecutive block of data, we can calculate the *address of an element* in the array from:

1. the starting address of the array,
2. the size in bytes of an element, and,
3. the element index

For example, suppose the starting address of the array `A` is s. The element `A[3]` can be found at $s + 8 \times 3 = s + 24$

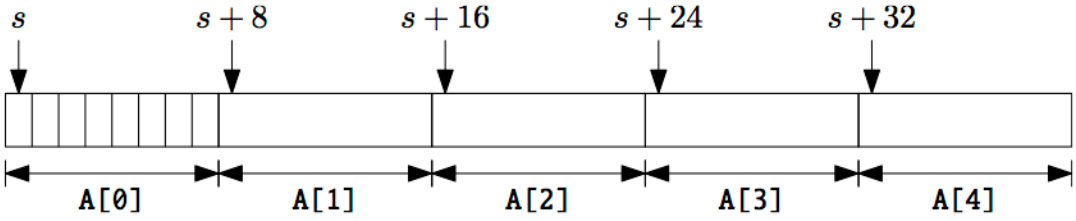

Figure 2.6: Array and element addresses

When we call a function in C and pass an array as an argument, the starting address is passed to the function. We can use the starting address to compute the adresses of elements.

Example 2.7 Write a function `sumArr(A, n)` that computes the sum of the `n` elements in the array `A`.

testsumArr.c

```
#include<stdio.h>

long sumArr(long A[], long n);

int main() {
  long X[] = {1, 4, 5, 8, 3, 7, 6};
  printf("Output = %ld\n", sumArr(X, 7));
  return 0;
}
```

sumArr.s

```
        .global sumArr
        .text
sumArr:
        xor     %rax, %rax
        xor     %rcx, %rcx
loop:   cmp     %rsi, %rcx
        jge     done
        add     (%rdi), %rax
        inc     %rcx
        add     $8, %rdi
        jmp     loop
done:   ret
```

Exercise 2.9 Write a function `maxArr(A, n)` that finds the maximum value of the array `A` of `n` elements.

2.8 Recusion

We can use the `call` instruction to call any function including making *recusive calls*. We need to use **push** and **pop** instructions to store some necessary values in the system stack while conduction the recursive calls.

Example 2.8 Write a function `fac(n)` that computes $n!$ using recursion.

$$n! = n \times (n-1)!, \qquad 0! = 1$$

facR.s

```
        .global fac
        .text
fac:
        cmp     $1, %rdi
        jle     base_case
        push    %rdi
        dec     %rdi
        call    fac
        pop     %rdi
        imul    %rdi, %rax
        ret
base_case:
        mov     $1, %rax
        ret
```

Exercise 2.10 Write a function `pow(a, b)` that computes a^b using recursion where,

$$a^b = a \times a^{b-1}, \qquad a^0 = 1$$

References

[Mat+13] M. Matz et al., eds. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2013.

Appendix A

Compiling Assembly Programs

A.1 Compiling on the ICT Server

Each student is provided an account on our ICT server (`ict.siit.tu.ac.th`), a Linux server. If you do not know your password, please contact me or Dr. Steven Gordon. You can log into the server and use the compiler on the server.

A.1.1 Connecting via SSH

You can connect to the ICT server via the SSH (Secure SHell) service.

Using Linux or Mac OS X

If you are using Mac OS X or Linux, you can open a terminal application and type

```
$ ssh cholwich@ict.siit.tu.ac.th
```

You need to use your own account instead of "`cholwich`".

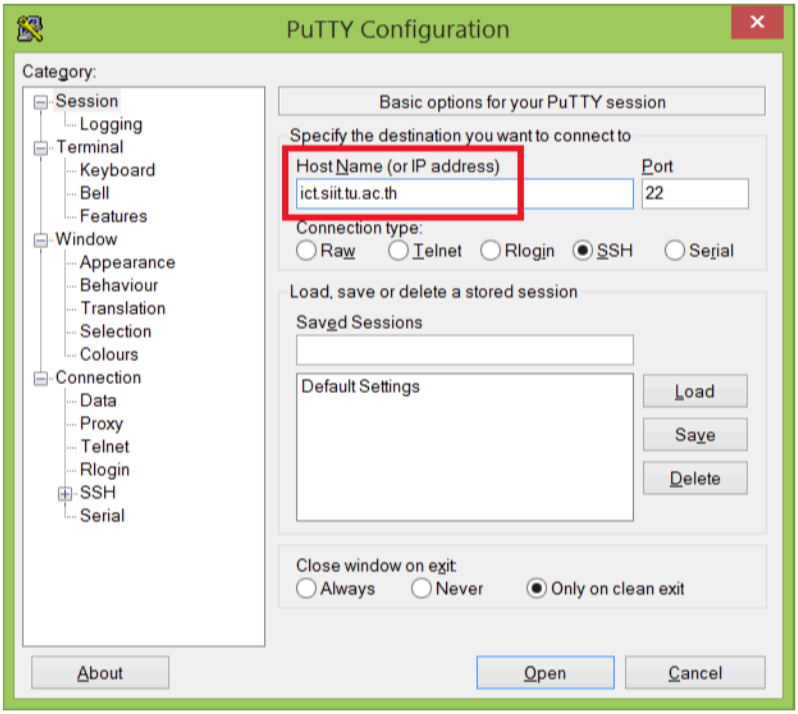

Figure A.1: PuTTY configuration window

Using Microsoft Windows

If you are using Microsoft Windows, you need an SSH client called "PuTTY". It can be downloaded from **`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`**. You can run the downloaded executable (putty.exe) file with installation. When PuTTY starts, it shows the configuration window. Fill in the Host Name with **`ict.siit.tu.ac.th`**, and click "Open" as shown in Figure A.1.

If it is the first time that you are connecting to the ICT server, you will be asked to confirm the server's host key. You can just click "Yes" to close this dialog as shown in Figure A.2.

Then, PuTTY will show the terminal window. Here, you will be asked for your account and password as shown in Figure A.3.

A.1.2 Editing a source file

After logging into the ICT server, your terminal application shows a command prompt waiting for you to input shell commands. Table A.1 shows a list of basic

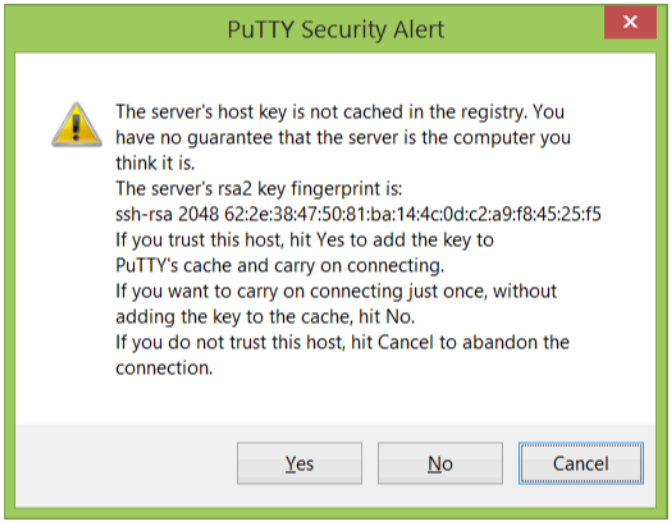

Figure A.2: PuTTY server's host key confirmation

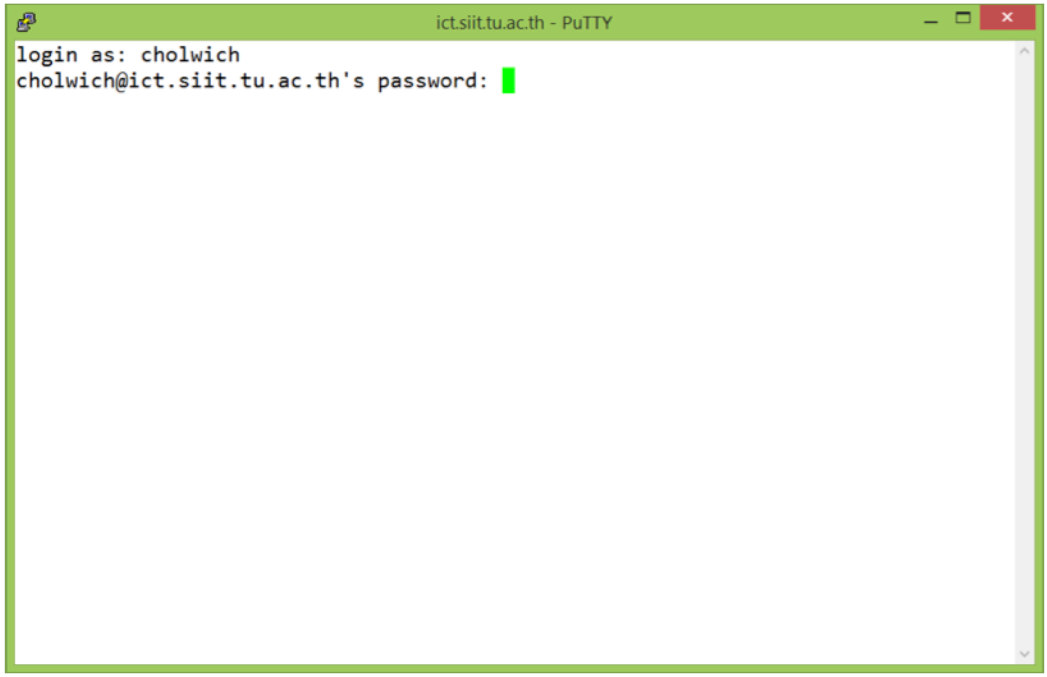

Figure A.3: PuTTY terminal window

commands that you may need to manage your source files. Figure A.4 shows a basic Linux command usage.

If you are not familiar with Linux commands, you are recommended to watch videos provided by Dr. Steven Gordon on YouTube at **http://www.youtube.com/watch?v=a8PsnTgS2OA** and **http://www.youtube.com/watch?v=KVH3dMWefWE**.

Command	Description
pwd	Show the current directory.
ls	Show the list of files in the current directory.
mkdir DIRECTORY	Create an empty directory named **DIRECTORY** under the current directory.
cd DIRECTORY	Change the current directory to **DIRECTORY**. If **DIRECTORY** is **..**, it changes to the parent directory of the current one.
cp SOURCE DEST	Copy file(s) from **SOURCE** to **DEST**.
mv SOURCE DEST	Move file(s) from **SOURCE** to **DEST**.
clear	Clear the screen.
exit	Exit from the server.

Table A.1: Basic Linux Commands

```
cholwich — cholwich@ict: ~ — bash — 80×25
cholwich@ict:~$ pwd
/home/cholwich
cholwich@ict:~$ mkdir asm
cholwich@ict:~$ cd asm
cholwich@ict:~/asm$ ls
cholwich@ict:~/asm$ cd ..
cholwich@ict:~$ exit
logout
Connection to ict.siit.tu.ac.th closed.
tokyo:~ cholwich$
```

Figure A.4: Basic Linux command usage

Nano[1] is a good editor for beginners. You can use it to edit your assembly source file. The editor can be started by using a command **nano** and a file name. It is recommended that an assembly source file is ended with **.s**. For example,

```
cholwich@ict$ nano asm1.s
```

[1] http://www.nano-editor.org

Nano will start editing a new file named **asm.s** as shown in Figure A.5. You can start typing your program. Press **Ctrl-O** to save the file, and press **Ctrl-X** to exit from Nano.

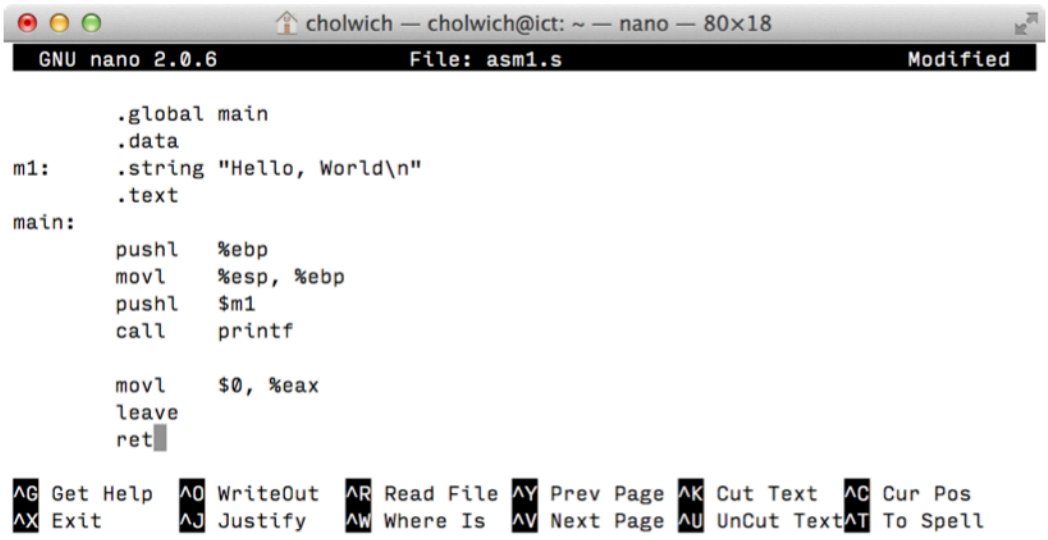

Figure A.5: Nano editor usage

A.1.3 Transferring a source file to ICT server

Instead of editing a source file using Nano, you can create or edit the file on your local machine before transferring it to the ICT server. You need to use **scp** command if you are using Mac OS X and Linux, and **winscp** if you are using Microsoft Windows. Figure A.6 shows how to use **scp** command. The command in the figure copies a file named **asm1.s** to the directory named **asm** under the home directory of user **cholwich** on **ict.siit.tu.ac.th**.

A.1.4 Compiling an assembly file

To compile your assembly file, you need a program called **gcc** on the ICT server. Here is the basic usage:

```
gcc -o OUTPUT SOURCE1 SOURCE2
```

Figure A.6: **scp** usage

where **OUTPUT** is the name of the executable file, **SOURCE1** and **SOURCE2** are the names of the source files (ended with **.s** and **.c**).

To run the compiled program, use **./OUTPUT** where **OUTPUT** is the file output from **gcc**.

A.2 Compiling on an Ubuntu Linux machine

If you are using Ubuntu Linux as a default operating system or on a virtual machine, you can install **gcc** on your machine for compiling assembly programs by installing a package called **build-essential**.

```
$ sudo apt-get update
$ sudo apt-get install build-essential
```