# CSCE 312 Lab manual

## Lab 5 - Assembly language programming

Instructor: Dr. Eun Jung Kim

Fall 2022
Department of Computer Science & Engineering
Texas A&M University

# Chapter 7: Assembly language programming, compilation and use of simulators

In this chapter you will learn assembly language programming. You will use **Y86-64 assembly language**, which is a subset and a slightly modified version of x86-64 assembly language. This will make you familiar with x86-64 assembly language which is suitable for the Intel processor chips found in the IBM compatible machines. You will also learn how to relate the "C" code which you generally use, with the Y86-64/x86-64 assembly counterpart which the "C" compiler generates during compilation. **You will be asked to write, verify, and illustrate operations of the assembly codes which are equivalent to "C" code fragments**. In addition, you will develop a basic understanding of the complex compilation and linking process. An understanding of the compilation process is related to processor design (that is computer architecture), performance and behavior of the software code and the hardware. All these encompass an essential understanding of the interrelated domains - computer organization, operating system, and compilers. We will utilize "**gcc**" as the model compiler to understand all these.

An example: A "C" code and its assembly equivalent are illustrated below (taken from Bryant & Hallaron's book "Computer Systems: A programmer's perspective"). Note that this assembly code fragment is not ready to be executed even when it is transformed to its binary/ machine code equivalent. Additional supporting code before and after this following assembly code fragment is necessary to make it a standalone executable. The complete standalone assembly code is available in Bryant & Hallaron's book Fig. 4.6 and in the example code file "asum.ys" which came along with the Y86-64 toolkit.

| "C" code to compute sum of an integer array | Y86-64 Assembly equivalent |
|---|---|
| int Sum(int *Start, int Count) {<br>  int sum = 0;<br>  while (Count) {<br>  sum+= *Start;<br>  Start++;<br>  Count--;<br>  }<br>  return sum;<br>} | Sum:<br>  pushq %rbp<br>  rrmovq  %rsp, %rbp<br>  mrmovq 8(%rbp),%rcx<br>  mrmovq 12(%rbp), %rdx<br>  xorq %rax,%rax<br>  andq %rdx, %rdx<br>  je End<br><br>Loop:<br>  mrmovq (%rcx), %rsi<br>  addq %rsi, %rax<br>  irmovq $4,%rbx<br>  addq %rbx,%rcx<br>  irmovq $-1,%rbx<br>  addq %rbx,%rdx<br>  jne Loop<br><br>End:<br>  rrmovq %rbp,%rsp<br>  popq %rbp<br>  ret |

Compiler and assembler: **Compiler** is the tool that transforms the high level source code to executable machine code. Whereas "**assembler**" is the tool that transform your assembly language code to machine code. Compilers first transform high level source codes to equivalent assembly codes, then assembler component of the compiler "assembles" the assembly code to corresponding machine formats.

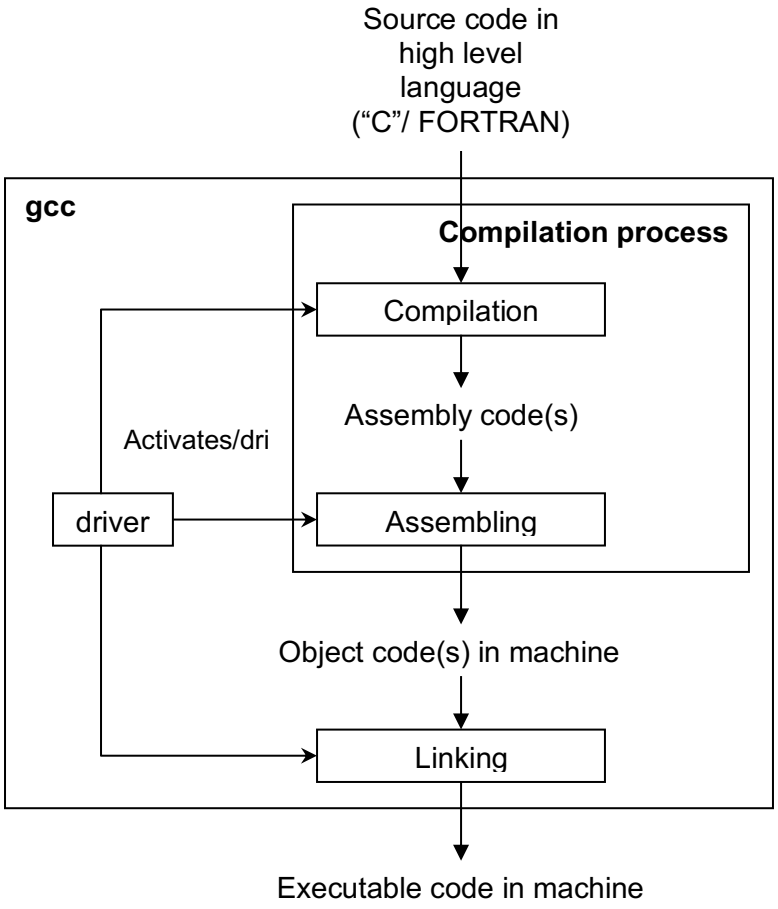


**Fig. 1: Steps in compilation process**

Two phase compilation, linker, object code, executable and loader: Different parts of the source codes might be written in separate source code files, but they are finally compiled together to generate a single binary executable file. Today high-level source code compilation is a two-step process. In the first step all the separate source code files are "compiled" to generate separate "object code" files in machine language format, then in the second step all these object codes are "linked" together to generate a single executable. When you use "gcc" these two phases – **"compilation" and "linking" are carried out one after other automatically as a result of a single command** –

```
gcc source_file.c -o executable_file
```

unless you specifically ask the "gcc" to do only the first phase by using the "–c" flag (asks gcc to stop after the compilation process, before linking) –

```
gcc -c source_file1.c -o object_file1.o
```

This first compilation phase generates the object code files with an extension ".o", in the second linking phase you use the object file (or files if there multiple source and object files) to generate the final single executable. Then you should type –

```
gcc object_file1.o object_file2.o -o executable_file
```

Actually, "gcc" is two tools "compiler" and "linker" combined into one (Fig. 1). Though the object files are in machine language format, but they cannot be executed individually, because they are not complete codes that can run and produce meaningful results by themselves. **The linking process** is a gluing process which adds additional glue code to the multiple object codes to hold them together to form the single standalone executable code file. On the other hand, loader is the part of the OS which loads the executable in the memory and prepares it for execution. The loading process is also fairly complicated, perhaps you will learn it later in an OS course. Sometimes by "loader" some people actually mean the linker (especially the linker that does its job at run time during "dynamic linking", but now you would know the difference. You can see the relative role of the loader in Fig. 7.15 in Bryant's book, pg. 567.

Two phase compilation and linking to enable modularization: Developers wanted to have modularity in their code during application development. Given your past experience in designing digital hardware you should be able to appreciate the need for modularization. Modularization allows "concurrent engineering". In this form of engineering practice senior members of a large engineering team first agree with each other about the "requirements specifications" of the entire application. Then they divide the entire application into several hardware and/or software "functional/logical modules". They specify the requirement specifications of each module so that when integrated these modules should properly work with each other to manifest the intended behavior. Once the modules have been specified then they assign the module design tasks and responsibilities to other members ("task assignment"). This enables the team members to design, develop and test various modules concurrently and progressively integrate them to finally get the application. This engineering discipline makes development of large applications manageable. This software engineering rationale was behind the need to have multiple modular code segments which are structured as functions, procedures, classes etc. instead of a single monolithic source code.

However, this scheme required that somebody integrates the various components to get the application working. This job is done by the linker. It links all the components together to generate a single functional application. This linking can be done at compile time and/or run time. For now, you will only learn about "static linking". To allow greater flexibility in the applications, modern "run time" systems also allow run time dynamic linking when the program is loaded and executed.

Linker performs two primary tasks – "symbol resolutions" and "relocation". When parameters are passed to functions or when two code segments (functions/ procedures or class methods) share a global or external variable (variable with "extern" modifier in "C") common memory locations are

shared between different code segments, even though they may be referred by different variable names ("symbols") in different code segments. In the following example –

```
//File 1, named "hello.c"
#include <stdio.h>
     void print_hello(char *a_string);
     int main(int argc, char *argv[]) {
          print_hello("Hello world\n");
          return 0;
     }


//File 2, named "print.c"
     void print_hello(char *passed_string){
          printf(passed_string);
     };
```

The parameter "a_string" and "passed_string" actually points to a single memory location, though it is referred using two different names ("symbols") in the main() and the print_hello() functions/ code segments stored in two different files. It is linker's job to resolve them to a common memory location when the final executable is made. The linker also arranges/joins or "relocates" these two separate code segments to form a single executable.

Without this two-phase scheme - translation followed by automated linking or gluing, programmers would have to explicitly and very meticulously compartmentalize and exclusively reserve memory for each section of the codes to avoid inadvertent memory corruption by another code segment. That would have been quite labor intensive, painful and error prone for a large application with multiple functions, classes, and components.

How to use gcc's inbuilt assembler: Actually, the compilation process is again a combination of two sub-processes – "compilation proper" and "assembling" (Fig. 1). "Compilation proper" phase does two things – "translation" and optimization" (not shown), which you will learn later in advanced computer architecture and compiler courses. The "driver" component of the compiler activates these processes in sequence, automatically inside gcc on behalf of the user.

The inbuilt assembler inside gcc, which called "gas", is available to users to compile assembly codes (if you provide assembly code files with extension ".s" instead of ".c"). Therefore, gcc can generate assembly codes from a given "C" source code file when you mention the "-S" flag, which asks it to stop after the first "compilation proper" phase**. To generate assembly equivalent of a "C" code you should type** -

```
gcc -S source_file1.c -o source_file1.s
```

The output is the assembly code file, which you can open to learn how a "C" code is transformed to equivalent assembly code. You will note that in addition to the core logic code which you wrote, additional code has been added. These additional codes either came from the standard ANSI "C" and/or the system call library for the given OS (Linux in your case). You should now remember the activities of Lab 1 (Chap 2).

Processor simulators and their various roles: In real-life "production" environment, compiled codes are executed on physical processors. However, loading and testing a code on a physical processor is cumbersome and time consuming, so often a "virtual" processor is used. This virtual processor is a program which can run on your desktop. It can read the machine language code specifically written for a "target" processor and generate the memory and registers state as the real physical processor would have this same program been executed on it. Sometime such a program is also called "processor emulator" or "processor simulator" or "instruction simulator". Emulators or simulators for the entire system are also available. These simulate or emulate the entire behavior of the system, processor chips, motherboard and all the IO systems. These are often used by system developers and designers to evaluate OS, kernel codes and embedded system applications. Simulators are especially valuable to troubleshoot codes, this is because it is possible to see and tracks values inside the register and memory locations inside a simulator while a code is stepped through one instruction at a time. Whereas in real physical processor, the entire code will get executed so fast that you will not get any opportunity to know what had happened in between the instructions.

One can have both software and hardware based (FPGA implementations) simulators for a hardware component like processor. Software based simulators which simulates hardware systems are slower than the actual hardware, whereas hardware-based simulators are faster than the software simulators. The benefit of software simulators is that they can be readily modified, and invasively instrumented to note any arbitrary internal state at will. With the present technology, hardware-based simulators do not readily provide such flexibility and richness. Simulators may only simulate a part of the system functionality or limited behavioral aspects, so it is important to know what behavior the simulator is actually simulating, and how to interpret the results.

Specificity of gcc: A platform is a combination of the hardware (processor and the peripherals), OS and the compiler. gcc is available for only few specific OS like Linux, BSD and Mac OS. Gcc can only compile, assemble or link codes for specific set of "target" processors. If no specific target is mentioned, then by default gcc assumes the target to be the host processor on which it is currently running. Note that gcc's assembler can only understand assembly languages for the target processors. This means vanilla gcc's assembler would not be able to understand Y86-64 assembly language, unless somebody modifies gcc to do that.

AT&T and Intel assembly syntax: This pertains to source-destination ordering. In Intel syntax the first operand is the destination, and the second operand is the source whereas in AT&T it is opposite. Intel syntax assembly will say "Op-code dst src" but AT&T syntax will say it as "Op-code src dst". This is good to know. So, you would know what the Y86-64 toolset and the textbook follow. gcc's assembler uses AT&T syntax.

How to embed assembly instruction inside "C" code: The good news is assembly and "C" programming can be mixed with each other provided the compiler supports it. Gcc for Linux supports x86-64 statement embedding in "C". The basic form of the embedded statement is either of the following two alternatives –

```
asm("assembly code"); //Plain old embedding

__asm__("assembly code"); //Just in case asm keyword is used for something else
```

Examples -

```
asm("movq %rax %rcx"); /* Moves the contents of rax to rcx */

__asm__(("movq %rax %rcx") /*The same as above */
```

To include multiple consecutive assembly statements, use "\n\t" or ";" after each assembly instruction, like this –

```
__asm__ ("movq %rbx, %rax\n\t"
                "movq $10, %rsi\n\t"
                "movb %ah, (%rcx)");
```

**1. Learning duration:** 1 week (See e-campus for exact due)

**2. Required Tools:** Y86-64 assembler, Y86-64 ISA simulator and gcc.

**3. Objective:** *To learn* -

Primary topics

1. Assembly language programming.
2. Assembly language equivalents of "C" code constructs.
3. Basic understanding of the compilation process.
4. How "C" function calls are handled by the compiler-linker and the operating system.
5. How to embed assembly code in "C" code to gain low level access to the processor's features.

Secondary topics

6. Role of a simulator and its limitations, how to use an ISA or processor simulator.
7. Basic understanding of the structure of the assembly code generated by "gcc" compiler for an Intel processor.
8. Basic understanding of the linking process (interaction of the compiler-linker with the operating system).

**4. Instructions:**

1. **This is an individual lab assignment** to be submitted separately. You may discuss the problem with other members of the class, but you should present your own solutions for the given problems.
2. Use the Y86-64 distribution that you can download from: http://csapp.cs.cmu.edu/3e/students.html (Look under Chapter-4). You can either download the source code and compile it on your machine, or you may use the precompiled binaries that are already present.
3. Some example Y86-64 assembly codes can be found in the "/y86-code" folder in the same Y86-64 toolkit installation. You may transfer these and the yas, yis binaries to a common working folder before running the simulations.

4. To generate the Y86-64 object code for the Y86-64 assembly code (for example the file "asum.ys") type the following –

```
./yas asum.ys
```

This will generate an object file with name "asum.yo".

5. To execute this object file generated in the previous step, on the "Y86-64 instruction simulator (yis)" type the following –

```
./yis asum.yo
```

You will get to see how the values in the registers and memory locations change as result of this. A trace of the above commands for the program named "asum.ys", which can be found in the example Y86-64 assembly code folder in the tool installation, is illustrated below.

```
Stopped in 34 steps at PC = 0x13.  Status 'HLT',
CC Z=1 S=0 O=0
Changes to registers:
%rax:   0x0000000000000000      0x0000abcdabcdabcd
%rsp:   0x0000000000000000      0x0000000000000200
%rdi:   0x0000000000000000      0x0000000000000038
%r8:    0x0000000000000000      0x0000000000000008
%r9:    0x0000000000000000      0x0000000000000001
%r10:   0x0000000000000000      0x0000a000a000a000

Changes to memory:
0x01f0: 0x0000000000000000      0x0000000000000055
0x01f8: 0x0000000000000000      0x0000000000000013
```

**5. Useful resources:**

1. Read Bryant's book section 4.1, 3.1 to 3.7, 3.15, 4.3.2 to 4.3.5, 7.1 to 7.9, 7.14

2. Complete reading all the necessary reading materials, tips and additional materials given in the Chap 5 of this lab manual.

## 6. Exercises to do

**6.1 Problem 1:** The following "C" program (code fragment) is given –

```
int i,j;
…..
if (i > j) {
     i= i+5;
}
else {
     i=0;
     j++;
}
```

*Activities to do-*

1. Provide the Y86-64 assembly language code for the "C" program.
2. Verify this assembly code using the Y86-64 tool set, provide a screenshot of the output (the change in value of each register).
   **Tip:** See the example Y86-64 code files. Read Bryant's book section 3.1 to 3.6.

**6.2 Problem 2:** For the following "C" program (code fragment) –

```
int j,k;
…..
for (int i=0; i <5; i++) {
     j = i*2;
     k = j+1;
}
```

*Activities to do-*

1. Provide the Y86-64 assembly language code for the "C" program.
3. Verify this assembly code using the Y86-64 tool set, provide a screenshot of the output (the change in value of each register).

**6.3 Problem 3:** For the following two "C" programs –

```
//Program 1, file name "lab5_prob3_1.c"
     #include <stdio.h>
     int main(int argc, char *argv[]){
          printf("Hello, world\n");
          return 0;
     }
```

```
//Program 2, file name "lab5_prob3_2.c"
     #include <stdio.h>
     int main(int argc, char *argv[]) {
          int i = 1;
          i++;
          printf("The value of  i is %d\n", i);
          return 0;
     }
```

*Activities to do-*

1. Use gcc to generate the equivalent assembly codes for these two "C" programs. Provide the printouts (or copy paste in the report) of these two assembly codes.

2. **Generate the executable codes** from the generated assembly codes.

3. **Compare and analyze the structure of the two assembly codes** generated by gcc, identify the various assembly code segments and their respective roles on the printouts. Specifically state what those assembly code statements are achieving. Do not state the obvious facts like this way - "…this statement is moving the value from memory location "z" to register %rax….". You should rather mention what purpose these statements are serving.

**Tips:** Glance through 7.1 to 7.9 of Bryant's book and this link - http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

**6.4 Problem 4:** Following single "C" file are given –

```
//File 1, named "lab5_prob4_main.c"
     #include <stdio.h>
     void print_hello();
     int main(int argc, char *argv[])
     {
          print_hello();
          return 0;
     }


     void print_hello(){
          printf("Hello, world\n");
     };
```

*Activities to do-*

1. Generate assembly codes for this file. Compare it with the assembly code generated from the file "**lab5_prob3_1.c**" of problem 3, analyze and explain how the function call "print_hello()" was materialized inside the computer (compiler, OS and the hardware) for the above example. Use figures and text descriptions if necessary.

**Tips:** Read sec 3.7, 7.14 of Bryant's book.

**6.5 Problem 5:** Following two "C" files are given –

```
//File 1, named "lab5_prob5_main.c"
     void print_hello();
     int main(int argc, char *argv[]) {
          print_hello();
          return 0;
     }


//File 2, named "lab5_prob5_print.c"
     #include <stdio.h>
     void print_hello() {
          printf("Hello, world\n");
     };
```

*Activities to do-*

1. Generate assembly codes for these two files. Then using these assembly files generate the object code files, then link the generated object files to make a single executable file. Use gcc toolset. Provide printout (or copy paste) of the assembly files.

2. Compare the assembly codes generated in this problem against the assembly code generated in problem 4. Is there any difference how the function call "print_hello()" was materialized for this example? Explain the differences if any, in the assembly codes generated in this problem with that of problem 4.

**6.6 Problem 6:** The following "C" code is given –

```
//File named "lab5_prob6.c"

     #include <stdio.h>
     int very_fast_function(int i){
          if ( (i*64 +1) > 1024) return i++;
          else return 0;
     }

     int main(int argc, char *argv[]) {
          int i;
          i=40;
          printf("The function value of  i is %d\n",
          very_fast_function(i) );
          return 0;
     }
```

*Activities to do-*

1. Rewrite the above "C" code such that the "very_fast_function()" is implemented in Y86-64 or x86-64 assembly language embedded in the "C" code.

*Note: You won't be able to run embedded Y86-64 assembly code in GCC, since GCC does not understand Y86-64.*

**Tips:** Read Chap 3, sec 3.15 in Bryant's book. You can also read the following links -

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

http://tldp.org/HOWTO/html_single/Assembly-HOWTO/

http://www.cs.virginia.edu/~clc5q/gcc-inline-asm.pdf

http://www.codeproject.com/Articles/15971/Using-Inline-Assembly-in-C-C