CS3320 – Mini Assignment 1

Name: Taha Adeel Mohammed
Roll Number: CS20BTECH11052

Q1) Write a short note on various options in common compilers: GCC and LLVM.

GCC

The GNU Compiler Collection (GCC) is a compiler supporting various programming languages such as C, C++, Objective C, Fortran, GO, etc. A few common options provided by GCC are shown below.

- **Overall Options**
  - --help : Prints the command line options understood by gcc along with their descriptions onto the terminal.
  - --version : Displays the version of gcc being used.

Compilation consists of four stages: preprocessing, compilation proper, assembly and linking. The following options control the stage upto which the source files are compiled.

  - -c : The source files are compiled, but not linked, hence producing an object file with the extension .o
  - -S : Stop after the compilation proper stage, i.e. do not assemble and link the files. Produces an assembler file with the extension .s
  - -E : Stop after the preprocessing stage, without compiling, assembling, or linking the code. Produces preprocessed source code, which is sent to the standard output by default.

  - -o <file> : Places the output(executable file/object file/assembler file/etc) into the file specified by <file>.
- There are many other options supported by gcc, including options to set the standard for the language(-std=standard, etc), warning options(-Werror, -Wall, etc), debugging options(-g, etc) and optimization options(-O1, -O2, etc).

LLVM

LLVM is a set of compilers that supports multiple programming languages and instruction set architectures, using Intermediate Representation as its core concept. Below we detail the options supported by clang, a front end for C/C++/Objective that uses LLVM as its backend.

- **Stage Selection Options**
  - -E : Run the preprocessor stage only.
  - -fsyntax-only : Runs the preprocessing, parsing, and type checking stages only.
  - -S : Produces an assembly file after running the above stages as well as LLVM generation and optimization stages and target specific code generation.
  - -c : Produces an object file after running all the stages besides linking.
- Similar to gcc, we have many other options supported, a few of which are: options to set the standard for the language(-std=standard, etc), target selection options(-target, etc), debugging options(-g, etc) and optimization options(-O1, -O2, etc).

Q2) Write a note on the various front-ends that these compilers support.

<u>GCC</u>

The latest release version of GCC supports front ends for the following languages, and the command used to compile the language.

- C : gcc
- C++ : g++
- Objective C : gcc
- Fortran : gfortran
- Ada : GNAT
- Go : gccgo
- D : gdc

There are also various third party front ends for different languages that haven't been integrated into the main distribution of GCC. A few of these are for the languages Pascal(gpc- GNU Pascal Compiler), Mercury, Modula-2, Modula-3, VHDL(GHDL), etc.

<u>LLVM</u>

LLVM is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and out and IR from the source code for that language. Languages with compilers that use LLVM(i.e LLVM is the backend) include:

- C, C++, Objective C : clang
- Ada
- D
- Fortran
- Rust
- D, Delphi, Haskell, Julia, Swift, etc

Q3) Use these compilers to generate code for various architectures using its various backends and report your findings.

We use the simple Hello world program shown below to generate code for various backends. The assembly code generated by these different compilers and backends is shown below(the code is generated using godbolt.org)

```c
#include <stdio.h>

int main(){
    printf("Hello World!");

    return 0;
}
```

- **ARM gcc 11.2(linux)**

```
.LC0:
        .ascii  "Hello World!\000"
main:
        push    {r7, lr}
        add     r7, sp, #0
        movw    r0, #:lower16:.LC0
        movt    r0, #:upper16:.LC0
        bl      printf
        movs    r3, #0
        mov     r0, r3
        pop     {r7, pc}
```

- **X86-64 gcc 11.2**

```
.LC0:
        .string "Hello World!"
main:
        push    rbp
        mov     rbp, rsp
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        pop     rbp
        ret
```

- **X86-64 clang 14.0.0**

```
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     dword ptr [rbp - 4], 0
        movabs  rdi, offset .L.str
        mov     al, 0
        call    printf
        xor     eax, eax
        add     rsp, 16
        pop     rbp
        ret
.L.str:
        .asciz  "Hello World!"
```

- **Mips64 gcc 11.2.0**

```
.LC0:
        .ascii  "Hello World!\000"
main:
        daddiu  $sp,$sp,-32
        sd      $31,24($sp)
        sd      $fp,16($sp)
        sd      $28,8($sp)
        move    $fp,$sp
        lui     $28,%hi(%neg(%gp_rel(main)))
        daddu   $28,$28,$25
        daddiu  $28,$28,%lo(%neg(%gp_rel(main)))
        ld      $2,%got_page(.LC0)($28)
        daddiu  $4,$2,%got_ofst(.LC0)
        ld      $2,%call16(printf)($28)
        mtlo    $2
        mflo    $25
        jalr    $25
        nop

        move    $2,$0
        move    $sp,$fp
        ld      $31,24($sp)
        ld      $fp,16($sp)
        ld      $28,8($sp)
        daddiu  $sp,$sp,32
        jr      $31
        nop
```

As we can see, different backends generate different assembly language codes. However, the general sequence of instructions is around the same for all these assembly codes.

Q4) Compilers come with various optimization levels: Focusing on options O0, O1, O2, O3 as well as -Os, -Oz. Run various codes using these predetermined passes and report your findings.

We use the below simple code that adds two numbers, and then increments the sum by a fixed amount through a for loop. We analyze the assembly code generated through different levels of optimization (Using ARM64 gcc 11.2.0) and analyze it.

```c
#include <stdio.h>

int sum_function(int a, int b){
    int sum = a + b;
    return sum;
}

int main(){
    int a = 4;
    int b = 3;

    int sum = sum_function(a, b);

    int n=5;
    for(int i=0; i<n; i++){
        ++sum;
    }

    printf("Final result: %d\n", sum);

    return 0;
}
```

- **-O0**

There is no optimization taking place and the assembly code is a direct conversion of the c code.

```
sum_function(int, int):
        sub     sp, sp, #32
        str     w0, [sp, 12]
        str     w1, [sp, 8]
        ldr     w1, [sp, 12]
        ldr     w0, [sp, 8]
        add     w0, w1, w0
        str     w0, [sp, 28]
        ldr     w0, [sp, 28]
        add     sp, sp, 32
        ret
.LC0:
        .string "Final result: %d\n"
main:
        stp     x29, x30, [sp, -48]!
        mov     x29, sp
        mov     w0, 4
        str     w0, [sp, 36]
        mov     w0, 3
        str     w0, [sp, 32]
        ldr     w1, [sp, 32]
        ldr     w0, [sp, 36]
        bl      sum_function(int, int)
        str     w0, [sp, 44]
        mov     w0, 5
        str     w0, [sp, 28]
        str     wzr, [sp, 40]
        b       .L4
```

```
.L5:
        ldr     w0, [sp, 44]
        add     w0, w0, 1
        str     w0, [sp, 44]
        ldr     w0, [sp, 40]
        add     w0, w0, 1
        str     w0, [sp, 40]
.L4:
        ldr     w1, [sp, 40]
        ldr     w0, [sp, 28]
        cmp     w1, w0
        blt     .L5
        ldr     w1, [sp, 44]
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 48
        ret
```

- **-O1**
  - We see that the assembly code is highly optimized now.
  - The sum function now directly returns the sum of the two parameters, rather than first copying them to different variable, then adding them, and then returning a copy of the sum.
  - In the main function, instead of the loop, the sum is directly incremented by n.

```
sum_function(int, int):
        add     w0, w0, w1
        ret
.LC0:
        .string "Final result: %d\n"
main:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        mov     w1, 12
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
```

- **-O2, -O3**

  The order of instructions is slightly changed for better optimization.

```
sum_function(int, int):
        add     w0, w0, w1
        ret
.LC0:
        .string "Final result: %d\n"
main:
        stp     x29, x30, [sp, -16]!
        mov     w1, 12
        adrp    x0, .LC0
        mov     x29, sp
        add     x0, x0, :lo12:.LC0
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        ret
```

- **-Os, -Oz**

  These flags optimize the code for size. In this case we get the same output as the -O2 and -O3 flags.