COA
Assignment 1

ASRAR UL HAQ

2020BITE092

SHAHZAREEN

2020BITE039

JAMEE BASHIR

2020BITE091

Cross Compilation:

When a compiler running on one architecture creates a binary executable which is capable of running on another architecture. Example: Creating an ARM binary executable on an X86 system.

Updating Packages:

sudo apt update
sudo apt upgrade

Installing Dependencies:

sudo apt install gcc gcc-aarch64-linux-gnu
sudo apt install g++ g++-aarch64-linux-gnu

hello.c:

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

Generating x86_64 Assembly File:

```
gcc hello.c -S -o hello_x86_asm
cat hello_x86_asm
```

```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```

Generating arm64 Assembly File:

aarch64-linux-gnu-gcc hello.c -S -o hello_arm64_asm
cat hello_arm64_asm

```
        .arch armv8-a
        .file   "hello.c"
        .text
        .section    .rodata
        .align  3
.LC0:
        .string "Hello World!"
        .text
        .align  2
        .global main
        .type   main, %function
main:
.LFB0:
        .cfi_startproc
        stp     x29, x30, [sp, -16]!
        .cfi_def_cfa_offset 16
        .cfi_offset 29, -16
        .cfi_offset 30, -8
        mov     x29, sp
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        bl      puts
        mov     w0, 0
        ldp     x29, x30, [sp], 16
        .cfi_restore 30
        .cfi_restore 29
        .cfi_def_cfa_offset 0
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
        .section    .note.GNU-stack,"",@progbits
```

Generating x86_64 Binary Executable:

```
gcc hello.c -o hello_x86

file hello_x86
hello_x86: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=052027a0a045abf419a7c865f9f0224ee798365a,
for GNU/Linux 3.2.0, not stripped

./hello_x86
Hello World!
```

Generating arm64 Binary Executable:

```
aarch64-linux-gnu-gcc hello.c -o hello_arm64

file hello_arm64
hello_arm64: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
BuildID[sha1]=6be5428204f81852d7e2d626cb83e646a3e343ad,
for GNU/Linux 3.7.0, not stripped

./hello_arm64
/lib/ld-linux-aarch64.so.1: No such file or directory
```

sudo apt install qemu-user

Running arm64 Binary using QEMU:

qemu-aarch64 -L /usr/aarch64-linux-gnu hello_arm64

Hello World!

Static Linking (x86_64):

gcc hello.c -static -o hello_x86_stat
./hello_x86_stat

Hello World!

Static Linking (arm64):

aarch64-linux-gnu-gcc hello.c -static -o hello_arm64_stat
qemu-aarch64 hello_arm64_stat

Hello World!

COA
Assignment 2

Design the circuits for the addition, multiplication and division using logisim or using any HDL.

Installing logisim:

Step 1: `sudo apt-get update`

Step 2: `sudo apt install default-jdk`

Step 3: `java –version`

Output :

openjdk version "11.0.7" 2020-04-14
OpenJDK Runtime Environment (build 11.0.7+10-post-Ubuntu-3ubuntu1)
OpenJDK 64-Bit Server VM (build 11.0.7+10-post-Ubuntu-3ubuntu1, mixed mode, sharing)

Step 4: `sudo apt-get install logisim`

Step 5: `Search for logisim in Apps and open it.`

Creating 32BIT ALU:

Make The circuit for Addition, Subtraction, division.
Your file will be saved as file_name.circ

COA
Assignment 3

Design an assembler for RV32I programs.

Installation

Step 1:

The assembler works on `Python3`. Plese create a `python3` virtualenvironment in your system. For linux systems with `virtualenv` installed, this is as simple as running

```
virtualenv -p python3 rvi
```

This creates a new virtual environment named `rvi`. For Windows, Anaconda, or other environments, plese refer to your environment specifc instructions.

Step 2:

After activating the environment created in step 1, install the requirements specified in `requirements.txt`. In `src/assembler`, run

pip install -r requirements.txt

Step 3:

Create a .py file that has RISC-V assembler for subset of instructions.

```
from lib.parser import parse_input
import argparse



def get_arguments():
    descr = '''
    RVI v''' + str(VERSION) + '''
    - A simple RV32I assembler developed for testing
    RV32I targeted hardware designs.
    '''
```

```
    ap = argparse.ArgumentParser(description=descr)
    ap.add_argument("INFILE", help="Input file containing assembly code.")
    ap.add_argument('-o', "--outfile",
                    help="Output file name.", default = 'a.b')
    ap.add_argument('-e', "--echo", help="Echo converted code to console",
                    action="store_true")
    ap.add_argument('-nc', "--no-color", help="Turn off color output.",
                    action="store_true")
    ap.add_argument('-n32', "--no-32", help="Turn of 32 bit core warnings.",
                    action="store_true")
    ap.add_argument('-x', "--hex", action="store_true",
                    help="Output generated code in hexadecimal format" +
                    " instead of binary.")
    ap.add_argument('-t', '--tokenize', action="store_true",
                    help="Echo tokenized instructions to console" +
                    " for debugging.")
    ap.add_argument("-es", "--echo-symbols", action="store_true",
                    help="Echo the symbols table.")
    args = ap.parse_args()
    return args


def main():
    args = get_arguments()
    infile = args.INFILE
    return parse_input(infile, **vars(args))


if __name__ == '__main__':
    main()
```

Step 4:

Create file for Constants and variable declaring various machine instructions.

Step 5:

Create file for Converting the tokenized assembly instruction to corresponding machine code

Step 6:

Create file for Parser for a simple assembler for subset of RV32I