CS 311 Laboratory 1

PART-A

The Godbolt Compiler Explorer is a fantastic tool for assembler programmers. You may read the details about this tool in https://thechipletter.substack.com/p/compiler-explorer In this task, you will explore other ISA's assembly from Compiler Explorer at https://godbolt.org/ .

You will compare the assembly language output for a variety of architectures. For example, How does x86-64 code compare to ARM64 and RISC-V?

You need **to count the number of instructions for the specified compiler and ISAs** for the high level program (For some you first need to write the C program) and input the counts as a table in the format provided below . Labels [those ends with :] and directives [those starts with .] are not instructions, so do not count them as instructions. Instructions are normally highlighted with color and you can apply Filter in the right-side Pan of the web interface to sort out so only instructions show. You should count manually the number of instructions from the output when selecting the specified compilers from https://godbolt.org.

ISA and Compiler	**No. of Instructions**
RISC-V 32-bits gcc 14.2.0	
RISC-V 64-bits gcc (trunk)	
RISC-V rv32gc gcc (trunk)	
RISC-V rv32gc clang(trunk)	
RISC-V rv64gc clang(trunk)	
x86-64 clang 12.0.0	
X86-64 gcc 14.2.0	
ARM64 gcc 14.2.0	
ARM32 gcc 14.2.0	
Armv8-a-clang 19.1.0	
MIPS64 gcc 5.4	

Q1. Write a C program that takes an integer value as an input and prints out the square of this value, together with the original number.

Q2. Write a C program that asks the user for a positive integer and iteratively computes the factorial of this integer.

Q3.

```
int x[10];

void init(int *x) {

   for(int i = 0; i < 10; i++)

       x[i] = i;

}

int sumofarray(int *x) {

   int sum = 0;

   for(int i = 0; i < 10; i++)

       sum = sum + x[i];

return sum;

}

int main(){

   init(x);

   int sum = sumofarray(x);

   return sum;

}
```

Q4.

```
int N=8;
int main() {
   int t1 = 0, t2 = 1, nextTerm;
   for(int i=0;i<N;i++){
       nextTerm = t1 + t2;
       t1 = t2;
       t2 = nextTerm;
   }
   return nextTerm;
}
```

Q5.

```
int source[8] = {0, 1, 2, 3, 4, 5, 6, 7};
int dest[8];
int calculate(int source) { return source * source; }

void loop(int *source, int *dest, int N) {
 for (int k = 0; k < N; k++) {
   int a = 10;
   if (source[k] != 0)
     dest[k] = calculate(source[k]) + 10;
 }
}
```

```
int main() {

 loop(source, dest, 8);

 return 0;

}
```

As a part of this study, you need to summarize the results by answering the following questions

a) For the same high-level program, the instruction sequence for different compilers and different machine architecture (represented by its ISA) - What are your observations ?
b) Even for the same ISA, but for different compilers, can you comment about the generated the instruction sequences

c) Provide your observations about the instruction sequences generated for 32-bit and 64-bit machines of the same ISA and of the same compiler.

PART-B

Write the following programs in the ***ToyRISC*** ISA:

1. Check if a given number is even or odd. If odd, write `1' to register **`x10`**. Else, write `-1' to register **`x10`**.

2. Write a program to sort an array of numbers in the descending order. The sorted array should be placed in the same addresses in memory as the initial array.

3. Write a program to place the first `n' Fibonacci numbers in the memory. The first number is placed at address 2^{16}-1, the second at 2^{16}-2, and so on.

4. Write a program to check if a given number is a palindrome. If yes, place `1' in **`x10`**. If no, place `-1' in **`x10`**.

5. Write a program to check if a given number is prime. If yes, place `1' in **`x10`**. If no, place `-1' in **`x10`**.

Submission Instructions:

PART-A

Submit the answers for the Q1-Q5 along with the observations as a pdf document named “entry-number_assignment1_partA.pdf'.

PART-B

1. **Submit one zipped archive named “entry-number_assignment1.zip".**
2. **The archive must contain files: even-odd.asm, descending.asm, fibonacci.asm, palindrome.asm, and prime.asm. Name your files exactly as mentioned.**
3. **Use the template programs given. You may change the data values for your testing. But do not change the names given to the addresses, for example, { a} and {n} in descending_template.asm. Remember to remove the comment lines.**
4. **Test each individual program using the test_each script. Make sure the test passes. You may change values in the input and expected output files for your testing purposes.**
5. **Test your final zip archive using the test_zip script. Make sure the test passes.**

You must submit both the parts on Moodle on or before the deadline (21st Jan 11.59PM)

Testing Your Programs for PART-B

Download and unzip the supporting files.

To test your program, use the given Java application emulator.jar. The command is

java -jar <path-to-emulator.jar><path-to-assembly-file><starting-address><ending-address> .

This command functionally emulates the program you have written and at the end prints the contents of the register file, as well as the contents of those addresses of the memory specified by starting-address and ending address.

Use this to test and debug your program.

An awkward crash implies your program is syntactically incorrect.

A graceful completion with unexpected contents in the register file and / or the memory implies your program is logically incorrect.

One test case for each program is given. To evaluate, the command is python test_each.py <path-to-assembly-file>.

Make sure the name of your .asm file is as specified above.

To test your zip archive, the command is python test_zip.py <path-to-zip-archive>. Make sure the name of your .zip file is as specified above.

We will use the test_zip script to automatically grade your submissions. So please make sure your submission works before submitting.