ECE-251 Computer Architecture
Final Project Report

Di Mei, Zhihao Wang
Professor Mohammed Billoo

Introduction

In this project, we implemented a basic floating-point-number calculator by using the ARM assembler in the Raspberry Pi. After compiling the “calc.s” file, the user is required to input a command line argument to invoke the calculator:

```
pi@raspberrypi: ~ $ ./calc <operation>
```

E.g.

```
pi@raspberrypi: ~ $ ./calc 3+9*(6-5/3)
```

In our program, the mathematical operations include addition, subtraction, multiplication, division, and the parenthesis (“+”, “-”, “*”, “/”, “(”, “)”). For each command line argument, the number of arithmetic operations, which do not include the parenthesis, should be less than or equal to 4; otherwise, an error “Input operations exceeded!” will be printed. Also, if the user inputs any operation that does not belong to the operations shown above, the program will print the error “Invalid expression!”. The result of calculation is always a floating number, and to quit from the program, the user has to input an empty line.

special notice: Because of the linux system’s issue, the user has to input “\(” and “\)” when the user puts the parenthesis in the beginning of the command line argument. An example is shown below.

```
pi@raspberrypi: ~ $ ./calc (9+1)*3
pi@raspberrypi: ~ $ ./calc \(9+1\)*3
```

The first line’s command will return a linux error. To fix this problem, the user has to add “\” in the front of the parenthesis (see the command in the second line) and then the program will return the calculated result.

Program Architecture

Reading Input from Command Line

To read the input from the command line argument, this program makes use of “load/load bytes” and “store/store bytes” instructions. In the beginning of “main:”, the code “ldr r2, [r1, #2]” helps load the second part of the command line (the first part is “./calc”), which is user’s input, into “r2”. Then, the “store:” part enables the program to access every character of user’s input “r2” by using “ldrb” and “strb” through iteration. In the process of iteration, if the current character is not null, the program goes to “expression:” part, and in this part, the program will evaluate the character and perform calculation.

Evaluation of the Character and Calculation

In our program, there exists an order of arithmetic operations. The program will process the operations of “()”, “*”, and “/” first, and then goes to the calculation involving “+” and “-”. Every time an mathematical operation is detected, the program will load the previous calculated result from a stack and process the calculation of it with the next number. Here, the stack instructions “stmbd” and “ldmia” are used. The “expr:” part includes the evaluation of operations “()”, “+”, and “-”; the “term:” part includes the evaluation of operations “*” and “/”. Because the designed calculator processes the calculation of floating numbers, each operand is divided into integer part and fractional part. In the “number:” part, the program firstly evaluates operand’s integer part and then evaluates the fractional part. From the perspective of the program’s whole structure, the part “number:” is a function inside the “term:”, and “term:” is a function inside the “expr:”. The function “expr:” is called inside the “expression:” part.

Instructions of Floating Point Numbers

Since this program takes the input of floating-point operands and outputs the floating point number, it uses different types of floating-point-number instructions including “vmov”, “vldr”, “vadd”, “vsub”, “vmul”, and “vdiv”. For some arithmetic operations, we specify the precision of the floating point numbers by adding “.f64” or “.u32” after the instruction. To enable the assembler to support these operations, the types of CPU and FPU are declared in the beginning of the program.

Challenges & Solutions

ARM Structure

When we tried to apply vector operations such as vmov and vadd.f64, we failed to compile them. The assembler reported that the instruction was not supported in ARM mode. So we researched and found out that we can define CPU and FPU mode in the beginning of the file, so that the assembler can process through vector operations.
The definition we used for our program is :

```
.cpu    cortex-a53
.fpu    neon-fp-armv8
.syntax unified
```

We then are able to use instructions as vldr and vcvt to do floating point arithmetics.

Command Line Input and Argument

We failed to take command line input and argument at first. We tried to find a C style argc and argv to take the inputs. We then find out that r1 and r2 registers are able to store command line inputs. So we successfully stored the inputs from the initial r1 and r2 registers into our buffer.

Operation Counter

We implemented an operation counter. The counter kept reporting bus error. We researched the reason for bus error to occur. It is likely that bus error occurs when there is an issue with memory access. At last, we took a complicated way by loading and storing the counter into its address every time we increment it.