Contents

[How to Use 1](#_Toc468495658)

[1. Recommended Procedure 1](#_Toc468495659)

[2. General Introductions 1](#_Toc468495660)

[3. Running Programs 2](#_Toc468495661)

[3.1 Program 1.txt 3](#_Toc468495662)

[3.2 Program 2.txt 3](#_Toc468495663)

[3.3 Program 3.txt 4](#_Toc468495664)

[System Design 5](#_Toc468495665)

[UI Design 7](#_Toc468495666)

[Cache Design and Implementation 9](#_Toc468495667)

[Program 1 10](#_Toc468495668)

[Explanation 10](#_Toc468495669)

[Source Code 12](#_Toc468495670)

[Program 2 18](#_Toc468495671)

[The C++ program: 18](#_Toc468495672)

[The instructions: 19](#_Toc468495673)

[Program 3 31](#_Toc468495674)

# How to Use

## 1. Recommended Procedure

The simplest way to run all the 3 programs without understanding of the system is:

1) Open G3Computer.jar.

2) Click “Load Bin” to load “Boot Program.txt” found in “programs” directory.

3) Click “Load Bin” to load “Program 1.txt” found in “programs” directory.

4) Click “Load Bin” to load “Program 2.txt” found in “programs” directory.

5) Click “Load Bin” to load “Program 3.txt” found in “programs” directory.

6) Click “Card Reader” to load “Paragraph Sample.txt” found in “programs” directory.

7) Click “Run” to run the boot program. A “Halt” will be shown after it is complete.

8) Click “Run” to run the program 1. You will be asked to input 21 integers and the program will give you the answer. Once complete, a “Halt” will be shown.

9) Click “Run” to run the program 2. After the paragraph sample is printed, you’ll be asked to input a word and the program will give you the answer. Once complete, a “Halt” will be shown.

10) Click “Run” to run the program 3. This is just a simple test program for Float and Vector. Refer to the file itself for more details. Once complete, a “Halt” will be shown.

## 2. General Introductions

You can click each radio button to set the value of those registers and click the Set buttons to confirm making changes. The Run button would run all the instructions stored in the memory sequentially (until a halt instruction), and Single Step is used to execute the current instruction. Upon start, nothing is in the memory.

New instructions, or programs, can be loaded into memory in one of the three format: hexadecimal, octal and binary.

This computer should always have one and only one boot program, which can be loaded either by the "Default IPL" button, or by loading a program file that contains the boot program.

How cache behaves as the system runs is recorded in "trace.txt".

## 3. Running Programs

All the files are located in the ***"programs"*** directory.

If different boot programs are needed, please click "Clear All" button in the Field Engineer console to clear all things in the simulator, including the boot program. Then load a different one.

When there are multiple programs in the simulator, you can randomly execute any of them by remembering their initial address (which would be provided upon loaded) and setting the value of PC manually.

Once encountered an error (unexpected instruction or accessing wrong memory address), our computer would reboot automatically with boot program automatically executed and PC pointing to the initial address of the first user program.

### 3.1 Program 1.txt

1) Click "Load Bin" first to load "Boot Program.txt" as a boot program (If it is already set, ignore this step and do step 2, 4 instead).

2) Click "Load Bin" again to load "Program 2.txt".

3) Click "Run" to run boot program first. The computer will halt after the boot program is done.

4) Click "Run" again to execute the program. It reads 20 numbers (integers) from the keyboard, prints the numbers to the console printer, requests a number from the user, and searches the 20 numbers read in for the number closest to the number entered by the user. Finally, it prints the number entered by the user and the number closest to that number.

### 3.2 Program 2.txt

1) Click "Load Bin" first to load "Boot Program.txt" as a boot program (If it is already set, ignore this step and do step 2, 3, 5 instead).

2) Click "Load Bin" again to load "Program 2.txt".

3) Click "Card Reader" button to read an external text file, "Paragraph Sample.txt", into the card reader.

4) Click "Run" to run boot program first. The computer will halt after the boot program is done.

5) Click "Run" again to execute Program 2. It reads the file into memory, prints out the contents of the text file, and then asks the user for a word. Finally, it searches the paragraph to see if it contains the word. If so, it prints out the word, the sentence number, and the word number in the sentence.

### 3.3 Program 3.txt

1) Click "Load Bin" first to load "Boot Program.txt" as a boot program (If it is already set, ignore this step and do step 2, 4 instead).

2) Click "Load Bin" again to load "Program 3.txt".

3) Click "Run" to run boot program first. The computer will halt after the boot program is done.

4) Click "Run" again to execute the program. This program demonstrates floating point add/subtract, vector add/subtract, and floating point conversion by observing the relevant registers and output. More Details can be found in the program file.

# System Design

The figure below depicted our system hierarchy.



Figure System Hierarchy

For Memory System, it has a main memory implemented as an array of 2048 integers and a simple cache. Its size of is designed to be expandable. Also, a Rom Loader and a Card Reader are integrated into the memory system as outer devices.

Our system is based on certain Instruction Set Architecture (ISA). It is not explicitly “owned” by CPU or part of it, though. A bunch of decoding schemes and instruction definition conventions would be included in this element.

The Control Unit would be in charge of lots of tasks as it is in practice. Inside the CU class Data Handling Operations and Control Flow Operations are implemented. What’s more important, it’s the one that executes the Instruction Cycle, which is would allow us to realize pipelines later on.

What Processor Registers include INITIALLY has already been put on the figure. Their bit length is not necessarily the same as required in the project description. We packaged them together into the CPU Java class.

ALU would implement some arithmetic and logic operations.

The object design derives from the above figure. In addition, we made use of a few software engineering techniques for the implementation. Examples are Java interface, multi-thread programming, etc. Lots of classes are in the source code and the code is arranged as well as possible in order for further development.

# UI Design

Here we only briefly introduce the design of the operator console. The outmost would be a JFrame of BorderLayout, which is described below:



Figure Frame

For the moment we only made use of the center and south part to settle the Register Panel and Control Panel (to form the Operator Console).

Now we would mainly focus on the Register Panel since Control Panel would just consist of a few buttons.



Figure Register Panel & SubPanels

There should be some things to be clarified for the components of Left Panel or Right Panel, Register GUI. First, the points of black and white, implemented with radio buttons, represent the value of that register. Second, the “Value Display” is a text field showing the decimal content of the register. Third, there would always be a Set button to enable making changes to the value of certain registers at any time. Fourth, the Name and Index are optional because:

1. For GPR and IX, all would have indices but only the first has name.

2. For others, all would have name but none has index.

# Cache Design and Implementation

Our cache is implemented as a fully associative, unified cache. It has 16 cache lines, with each line having a tag indicating the address and space for storing 8 data. What’s behind the implementation is actually a queue. To demonstrate it is working, we printed relevant trace information both to system output console and to file (named trace.txt). In addition, we used write-through strategy. And when a write miss occurred, write no-allocate was employed.

The implementation lies in Memory class. New classes such as CacheLine and Cache are defined there.

# Program 1

## Explanation

For the program1, I think it should be divided into three parts: In, Compare, and Out. In the first part, it should be a loop for receiving the input numbers. So I choose the instruction SOB to loop. And the increase or decrees of the subscripts is use an index register to achieve. (By the way, LIX is a customized instruction invented by us, used for loading an immediate value to X.)

// LDR 3,0,20

// LDX 1,24

// IN 2,0

// STR 2,1,0

// LDA 2,1,0

// SIR 2,1

// STR 2,0,24

// LIX 1,0

// SOB 3,3,1

It is a sample loop, the index 1 is a like a subscripts.

And then is the part for comparing, it also uses the SOB and the index register to loop, and use two new instructions CMB, CMT to compare and replace.

This is one loop:

// LDA 2,1,1

// STR 2,0,30

// LDX 2,30

// LDR 0,2,0

// LDR 1,1,0

// CMT 1,0

// LDR 0,1,0

// LDR 2,2,0

// CMB 0,2

// STR 0,0,27

// LDR 0,2,0

// SMR 1,0,27

// STR 1,0,27

// LDR 2,0,29

// CMB 1,2

// TRR 1,2

// JCC 4,3,21

// STR 0,0,2

// STR 1,0,29

Memory[29] is the mark to compare if it less than last one’s result. If did, then replace, and record the number to Memory[28].

And the final part is quite simple for those program, so I just use the LDR and OUT to make it.

//LDR 1,1,0

//LDR 0,0,28

// OUT 1,1

// OUT 0,1

## Source Code

|  |
| --- |
| LIX 1,0  1000110001000000  LIX 2,0  1000110010000000  LIX 3,0  1000110011000000  LDR 0,0,5  0000010000000101  LDR 1,0,5  0000010100000101  LDR 2,0,5  0000011000000101  LDR 3,0,5  0000011100000101  // load index 3 with the m[9]=512  LDX 3,9  1000010011001001  LDA 0, 0, 1  0000110000000001  SRC I SET 0, 1, 15  0110010001101111  SIR 0,0,1  0001110000000001  STR 0,0,8  0000100000001000  LDA 0, 0, 1  0000110000000001  SRC I SET 0, 1, 15  0110010001101111  AMR 0,0,8  0001000000001000  //set 65535 to var.29 And var.31 as a max value to compare  STR 0,3,29  0000100011011101  STR 0,3,31  0000100011011111  // set the array address(712) for input number to index register 1  LDA 3,3,0  0000111111000000  AIR 3,200  0001101111001000  STR 3,0,8  0000101100001000  LDR 3,0,5  0000011100000101  //set counter(var.21)=21  AIR 3,21  0001101100010101  STR 3,3,21  0000101111010101  //loop for in and out 21 numbers  LDX 1,8  1000010001001000  IN 2,0  1100011000000000  OUT 2,1  1100101000000001  STR 2,1,0  0000101001000000  LDA 2,1,0  0000111001000000  SIR 2,1  0001111000000001  STR 2,0,8  0000101000001000  LIX 1,0  1000110001000000  LDX 1,8  1000010001001000  JSR 0,10  0011000000001010  SIR 3,9  0001111100001001  STR 3,0,8  0000101100001000  LDR 3,3,21  0000011111010101  SIR 3,1  0001111100000001  STR 3,3,21  0000101111010101  AIR 3,1  0001101100000001  \*SOB 3,0,8 I SET  0011101100101000  LIX 1,0  1000110001000000  LDR 3,0,5  0000011100000101  //set counter(var.20)=20  AIR 3,20  0001101100010100  STR 3,3,20  0000101111010100  LDR 1,0,5  0000010100000101  //loop for compare 20 numbers with the required one  \*\*\*\*\*LDA 1,3,0  0000110111000000  AIR 1,180  0001100110110100  STR 1,0,8  0000100100001000  LDX 1,8  1000010001001000  LDA 2,1,1  0000111001000001  STR 2,3,30  0000101011011110  STR 2,0,8  0000101000001000  LDX 2,8  1000010010001000  LDR 0,2,0  0000010010000000  LDR 1,1,0  0000010101000000  CMT 1,0  0101100100000000  LDR 0,1,0  0000010001000000  LDR 2,2,0  0000011010000000  CMB 0,2  0101110010000000  STR 0,3,27  0000100011011011  SMR 1,3,27  0001010111011011  STR 1,3,27  0000100111011011  LDR 2,3,29  0000011011011101  CMB 1,2  0101110110000000  TRR 1,2  0100100110000000  JSR 0,10  0011000000001010  AIR 3,6  0001101100000110  STR 3,0,8  0000101100001000  \*JCC 3,0,8 I SET  0010101100101000  LDR 0,2,0  0000010010000000  STR 0,3,28  0000100011011100  STR 1,3,29  0000100111011101  LDR 2,3,30  0000011011011110  AIR 2,1  0001101000000001  STR 2,3,30  0000101011011110  LIX 2,0  1000110010000000  JSR 0,10  0011000000001010  \*SIR 3,26  0001111100011010  STR 3,0,8  0000101100001000  LDR 3,3,20  0000011111010100  SIR 3,1  0001111100000001  STR 3,3,20  0000101111010100  AIR 3,1  0001101100000001  \*SOB 3,0,8 I set  0011101100101000  //output the required number and closet one(var.28)  LDR 1,1,0  0000010101000000  LDR 0,3,28  0000010011011100  OUT 1, 1  1100100100000001  OUT 0, 1  1100100000000001  LDR 0,3,31  0000010011011111  STR 0,3,29  0000100011011101  LIX 3,0  1000110011000000  LIX 1,0  1000110001000000  LIX 2,0  1000110010000000  LDA 0, 0, 31  0000110000011111  SRC I SET 0, 1, 6  0110010001100110  STR 0, 0, 8  0000100000001000 |

# Program 2

## The C++ program:

|  |
| --- |
| #include <iostream>  using namespace std;  int main() {  char para[1000] = "we can get learn from the movie... that the father are badly sick, so. he just urges. to teach hushpuppy. how to be strong enough to survive by herself.";  char pattern[100] = "can";  cout << para << endl;  cout << pattern << endl;  int word, sentence;  word = 0;  sentence = 1;  int start = -1;  for (int i = 0; para[i] != '\0'; ++i) {  // got a word  if (!(para[i] >= 'a' && para[i] <= 'z' || para[i] == '\'')) {  if (start != -1) {  // compare  int j;  for (j = 0; pattern[j] != '\0' && start + j < i && pattern[j] == para[start + j]; ++j);  if (pattern[j] == '\0' && start + j == i) {  cout << sentence << endl;  cout << word << endl;  cout << pattern << endl;  }  // got a sentence  if (para[i] == '.') {  word = 0;  ++sentence;  }  start = -1;  }  }  else if (start == -1) {  start = i;  ++word;  }  }  return 0;  } |

## The instructions:

|  |
| --- |
| /\*  \*\* Instruction Format:  \*\* "HEAD R, IX, I, ADDR"  \*\* Example: STR 0, 1, 0, 0  \*\* Part of the content of certain instructions might be omitted with a symbol of '\_'.  \*\* Example: AIR, 0, \_, \_, 1  \*/  // start of variables  // M[8] = 1984  // RFS 0  // M[10] = 13312  // initialize X[1] for variables  LIX \_, 1, \_, 0  1000110001000000  LDX \_, 1, 0, 8  1000010001001000  /\*  \*\* Variable Definition:  \*\* STR 0, 1, 0, 0 para  \*\* STR 0, 1, 0, 1 pattern  \*\* STR 0, 1, 0, 2 word  \*\* STR 0, 1, 0, 3 sentence  \*\* STR 0, 1, 0, 4 start  \*\* STR 0, 1, 0, 5 i  \*\* STR 0, 1, 0, 6 j  \*\* (Disregarded) STR 0, 1, 0, 7 constant 0  \*\* STR 0, 1, 0, 8 constant -1  \*\* STR 0, 1, 0, 9 constant 97 (a)  \*\* STR 0, 1, 0, 10 constant 122 (z)  \*\* STR 0, 1, 0, 11 constant 39 (')  \*\* STR 0, 1, 0, 12 constant 46 (.)  \*\* STR 3, 1, 0, 13 loop 1 start  \*\* STR 3, 1, 0, 14 loop 1 end  \*\* STR 3, 1, 0, 15 loop 2 start  \*\* STR 3, 1, 0, 16 loop 2 end  \*\* STR 3, 1, 0, 17 branch 1  \*\* STR 3, 1, 0, 18 branch 2  \*\* STR 3, 1, 0, 19 branch 3  \*\* STR 3, 1, 0, 20 branch 4  \*\* STR 3, 1, 0, 21 branch 5  \*\* STR 0, 1, 0, 22 addr(para[start + j])  \*\* STR 0, 1, 0, 23 addr(para[i])  \*\* STR 0, 1, 0, 24 addr(pattern[j])  \*\* STR 0, 1, 0, 25 constant 108  \*\* STR 0, 1, 0, 26 constant 86  \*\* STR 0, 1, 0, 27 constant 85  \*/  // store the initial address of paragraph into M[M[8]], and output  TRAP \_, \_, \_, 0  0111100000000000  STR 1, 1, 0, 0  0000100101000000  // store the initial address of word into M[M[8] + 1]  // M[7] = 1, for I/O control  LDA 0, 0, 0, 1  0000110000000001  STR 0, 0, 0, 7  0000100000000111  IN 0, \_, \_, 0  1100010000000000  STR 0, 1, 0, 1  0000100001000001    LDR 1, 1, 0, 1  0000010101000001  TRAP \_, \_, \_, 1  0111100000000001  // STR 0, 1, 0, 2 word, word = 0  LDA 0, 0, 0, 0  0000110000000000  STR 0, 1, 0, 2  0000100001000010  // STR 0, 1, 0, 3 sentence, sentence = 1  LDA 0, 0, 0, 1  0000110000000001  STR 0, 1, 0, 3  0000100001000011  // STR 0, 1, 0, 8 constant -1  LDA 0, 0, 0, 0  0000110000000000  SIR 0, \_, \_, 1  0001110000000001  STR 0, 1, 0, 8  0000100001001000  // STR 0, 1, 0, 4 start, start = -1  LDR 0, 1, 0, 8  0000010001001000  STR 0, 1, 0, 4  0000100001000100  // STR 0, 1, 0, 5 i, i = 0  LDA 0, 0, 0, 0  0000110000000000  STR 0, 1, 0, 5  0000100001000101  // STR 0, 1, 0, 9 constant 97 (a)  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 7  0110010001100111  SIR 0, \_, \_, 31  0001110000011111  STR 0, 1, 0, 9  0000100001001001  // STR 0, 1, 0, 10 constant 122 (z)  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 7  0110010001100111  SIR 0, \_, \_, 6  0001110000000110  STR 0, 1, 0, 10  0000100001001010  // STR 0, 1, 0, 11 constant 39 (')  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 5  0110010001100101  AIR 0, \_, \_, 7  0001100000000111  STR 0, 1, 0, 11  0000100001001011  // STR 0, 1, 0, 12 constant 46 (.)  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 5  0110010001100101  AIR 0, \_, \_, 14  0001100000001110  STR 0, 1, 0, 12  0000100001001100  // STR 0, 1, 0, 25 constant 108  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 7  0110010001100111  SIR 0, \_, \_, 20  0001110000010100  STR 0, 1, 0, 25  0000100001011001  // STR 0, 1, 0, 26 constant 86  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 6  0110010001100110  AIR 0, \_, \_, 22  0001100000010110  STR 0, 1, 0, 26  0000100001011010  // STR 0, 1, 0, 27 constant 85  SIR 0, \_, \_, 1  0001110000000001  STR 0, 1, 0, 27  0000100001011011  // -----------------------------------------------------------------------------  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 4 (to get the loop 1 start address for jumping back)  AIR 3, \_, \_, 4  0001101100000100  // STR 3, 1, 0, 13 loop 1 start  STR 3, 1, 0, 13  0000101101001101  // On the basis of previous addition, Add 108 (to get the loop 1 end address for jumping out)  AMR 3, 1, 0, 25  0001001101011001  // STR 3, 1, 0, 14 loop 1 end  STR 3, 1, 0, 14  0000101101001110  // loop 1 start, R[1] = para[i]  LDR 0, 1, 0, 0  0000010001000000  AMR 0, 1, 0, 5  0001000001000101  STR 0, 1, 0, 23  0000100001010111  LDR 1, 1, 1, 23  0000010101110111  // if R[1] == 0 jump out to loop 1 end  JZ 1, 1, 1, 14  0010000101101110  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 86 (to get the branch 1 address)  AMR 3, 1, 0, 26  0001001101011010  // STR 3, 1, 0, 17 branch 1  STR 3, 1, 0, 17  0000101101010001  // construct the condition for if 1 to R[2]  LDR 0, 1, 0, 9  0000010001001001  GE 1, 0, \_, 2  0111000100000010  LDR 0, 1, 0, 10  0000010001001010  LE 1, 0, \_, 3  0110110100000011  AND 2, 3, \_, \_  0100111011000000  LDR 0, 1, 0, 11  0000010001001011  ET 1, 0, \_, 3  0111010100000011  ORR 2, 3, \_, \_  0101001011000000  NOT 2, \_, \_, \_  0101011000000000  // if R[2] == 0 jump to branch 1  JZ 2, 1, 1, 17  0010001001110001  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 85 (to get the branch 2 address)  AMR 3, 1, 0, 27  0001001101011011  // STR 3, 1, 0, 18 branch 2  STR 3, 1, 0, 18  0000101101010010  // construct the condition for if 2  LDR 1, 1, 0, 4  0000010101000100  LDR 2, 1, 0, 8  0000011001001000  TRR 1, 2, \_, \_  0100100110000000  // if start == -1 jump to branch 2  JCC 3, 1, 1, 18  0010101101110010    // STR 1, 1, 0, 6 j = 0  LDA 1, 0, 0, 0  0000110100000000  STR 1, 1, 0, 6  0000100101000110  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 4 (to get the loop 2 start address for jumping back)  AIR 3, \_, \_, 4  0001101100000100  // STR 3, 1, 0, 15 loop 2 start  STR 3, 1, 0, 15  0000101101001111  // On the basis of previous addition, add 25 (to get the loop 2 end address for jumping out)  AIR 3, \_, \_, 25  0001101100011001  // STR 3, 1, 0, 16 loop 2 end  STR 3, 1, 0, 16  0000101101010000  // loop 2 start, R[1] = pattern[j]  LDR 1, 1, 0, 1  0000010101000001  AMR 1, 1, 0, 6  0001000101000110  STR 1, 1, 0, 24  0000100101011000  LDR 1, 1, 1, 24  0000010101111000  // construct condition for loop 2 into R[3]  LDA 0, 0, 0, 0  0000110000000000  ET 1, 0, \_, 3  0111010100000011  NOT 3, \_, \_, \_  0101011100000000  LDR 0, 1, 0, 4  0000010001000100  AMR 0, 1, 0, 6  0001000001000110  LDR 2, 1, 0, 5  0000011001000101  SIR 2, \_, \_, 1  0001111000000001  LE 0, 2, \_, 2  0110110010000010  AND 3, 2, \_, \_  0100111110000000  LDR 0, 1, 0, 0  0000010001000000  AMR 0, 1, 0, 4  0001000001000100  AMR 0, 1, 0, 6  0001000001000110  STR 0, 1, 0, 22  0000100001010110  LDR 0, 1, 1, 22  0000010001110110  ET 1, 0, \_, 2  0111010100000010  AND 3, 2, \_, \_  0100111110000000  // if R[3] == 0 jump out to loop 2 end  JZ 3, 1, 1, 16  0010001101110000  // loop 2 increment  LDR 0, 1, 0, 6  0000010001000110  AIR 0, \_, \_, 1  0001100000000001  STR 0, 1, 0, 6  0000100001000110    // jump back to loop 2 start  JMA \_, 1, 1, 15  0010110001101111  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 16 (to get the branch 3 address)  AIR 3, \_, \_, 16  0001101100010000  // STR 3, 1, 0, 19 branch 3  STR 3, 1, 0, 19  0000101101010011  // construct the condition for if 3  LDA 0, 0, 0, 0  0000110000000000  ET 1, 0, \_, 3  0111010100000011  LDR 1, 1, 0, 4  0000010101000100  AMR 1, 1, 0, 6  0001000101000110  LDR 2, 1, 0, 5  0000011001000101  ET 1, 2, \_, 2  0111010110000010  AND 3, 2, \_, \_  0100111110000000  // if R[3] == 0 jump to branch 3  JZ 3, 1, 1, 19  0010001101110011  // output pattern, sentence, word  LDR 1, 1, 0, 1  0000010101000001  TRAP \_, \_, \_, 1  0111100000000001  LDR 2, 1, 0, 3  0000011001000011  LDR 3, 1, 0, 2  0000011101000010  OUT 2, \_, \_, 1  1100101000000001  OUT 3, \_, \_, 1  1100101100000001  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 14 (to get the branch 4 address)  AIR 3, \_, \_, 14  0001101100001110  // STR 3, 1, 0, 20 branch 4  STR 3, 1, 0, 20  0000101101010100  // construct the condition for if 4  LDR 0, 1, 0, 0  0000010001000000  AMR 0, 1, 0, 5  0001000001000101  STR 0, 1, 0, 23  0000100001010111  LDR 0, 1, 1, 23  0000010001110111  LDR 1, 1, 0, 12  0000010101001100  ET 0, 1, \_, 2  0111010001000010  // if R[2] == 0 jump to branch 4  JZ 2, 1, 1, 20  0010001001110100    // reset word and increment sentence  LDA 0, 0, 0, 0  0000110000000000  STR 0, 1, 0, 2  0000100001000010  LDR 0, 1, 0, 3  0000010001000011  AIR 0, \_, \_, 1  0001100000000001  STR 0, 1, 0, 3  0000100001000011  // reset start  LDR 0, 1, 0, 8  0000010001001000  STR 0, 1, 0, 4  0000100001000100  // jump to brach 2  JMA \_, 1, 1, 18  0010110001110010  // JSR to 10 and RFS in order to obtain PC + 1 into R[3]  JSR \_, 0, 0, 10  0011000000001010  // Add 11 (to get the branch 5 address)  AIR 3, \_, \_, 11  0001101100001011  // STR 3, 1, 0, 21 branch 5  STR 3, 1, 0, 21  0000101101010101  // construct the condition for if 5  LDR 0, 1, 0, 4  0000010001000100  LDR 1, 1, 0, 8  0000010101001000  ET 0, 1, \_, 2  0111010001000010  // if R[2] == 0 jump to branch 5  JZ 2, 1, 1, 21  0010001001110101  // set start and increment word  LDR 0, 1, 0, 5  0000010001000101  STR 0, 1, 0, 4  0000100001000100  LDR 0, 1, 0, 2  0000010001000010  AIR 0, \_, \_, 1  0001100000000001  STR 0, 1, 0, 2  0000100001000010  // loop 1 increment  LDR 0, 1, 0, 5  0000010001000101  AIR 0, \_, \_, 1  0001100000000001  STR 0, 1, 0, 5  0000100001000101  // jump back to loop 1 start  JMA \_, 1, 1, 13  0010110001101101  // END OF PROGRAM  // (nothing here) |

# Program 3

This program validates the functionality of FADD/FSUB, VADD/VSUB and CNVRT. It’s a very small program. Please follow the comment of the program and single step to evaluate.

|  |
| --- |
| /\*\*  \* Instruction Format:  \* "HEAD R, IX, I, ADDR"  \* Example: STR 0, 1, 0, 0  \* Part of the content of certain instructions might be omitted with a symbol of '\_'.  \* Example: AIR, 0, \_, \_, 1  \*/  // start of variables  // M[8] = 1984  // initialize X[1] for variables  LIX \_, 1, \_, 0  1000110001000000  LDX \_, 1, 0, 8  1000010001001000  /\*\*  \* Variable Definition:  \* (STR 0, 1, 0, 0, STR 0, 1, 0, 1) Float1 1.3  \* STR 0, 1, 0, 2 Float2 2.6 (0 1111111 00011010)  \* STR 0, 1, 0, 4 constant 127 (complement of -1 in 7-bit 2's)  \* (STR 0, 1, 0, 5, STR 0, 1, 0, 6) Vector1  \* (STR 0, 1, 0, 7, STR 0, 1, 0, 8) Vector2  \* (STR 0, 1, 0, 9, STR 0, 1, 0, 10) Float 2  \* STR 0, 1, 0, 11 addr(Vector1)  \* STR 0, 1, 0, 12 addr(Vector2)  \*/  // construct constant -1  LDA 0, 0, 0, 1  0000110000000001  SRC 0, 1, 1, 7  0110010001100111  SIR 0, \_, \_, 1  0001110000000001  STR 0, 1, 0, 4  0000100001000100  // construct Float1 1.3  LDR 0, 1, 0, 4  0000010001000100  STR 0, 1, 0, 0  0000100001000000  LDA 0, 0, 0, 13  0000110000001101  STR 0, 1, 0, 1  0000100001000001  // construct Float2 2.6  LDR 0, 1, 0, 4  0000010001000100  SRC 0, 1, 1, 8  0110010001101000  AIR 0, \_, \_, 26  0001100000011010  STR 0, 1, 0, 2  0000100001000010  // test FADD/FSUB  LDFR 0, 1, 0, 0  1010000001000000  FADD 0, 1, 0, 2  0110110001000010  FSUB 0, 1, 0, 2  0111000001000010  // -----------------------------------------------------------------------------  // construct Vector1 (1, 2)  LDA 0, 0, 0, 1  0000110000000001  STR 0, 1, 0, 5  0000100001000101  LDA 0, 0, 0, 2  0000110000000010  STR 0, 1, 0, 6  0000100001000110  // construct Vector1 (3, 4)  LDA 0, 0, 0, 3  0000110000000011  STR 0, 1, 0, 7  0000100001000111  LDA 0, 0, 0, 4  0000110000000100  STR 0, 1, 0, 8  0000100001001000  // construct Float 2  LDA 0, 0, 0, 0  0000110000000000  STR 0, 1, 0, 9  0000100001001001  LDA 0, 0, 0, 2  0000110000000010  STR 0, 1, 0, 10  0000100001001010  // set addr(Vector1), addr(Vector2)  LDA 0, 1, 0, 5  0000110001000101  STR 0, 1, 0, 11  0000100001001011  LDA 0, 1, 0, 7  0000110001000111  STR 0, 1, 0, 12  0000100001001100  // test VADD/VSUB  LDFR 1, 1, 0, 9  1010000101001001  VADD 1, 1, 0, 11  0111010101001011  LDR 0, 1, 0, 5  0000010001000101  OUT 0, \_, \_, 1  1100100000000001  LDR 0, 1, 0, 6  0000010001000110  OUT 0, \_, \_, 1  1100100000000001  VSUB 1, 1, 0, 11  0111100101001011  LDR 0, 1, 0, 5  0000010001000101  OUT 0, \_, \_, 1  1100100000000001  LDR 0, 1, 0, 6  0000010001000110  OUT 0, \_, \_, 1  1100100000000001  // -----------------------------------------------------------------------------  // test CNVRT  // convert 2.6 to 2  LDA 1, 0, 0, 0  0000110100000000  CNVRT 1, 1, 0, 2  0111110101000010  // convert 2 to 2.0  LDA 0, 0, 0, 1  0000110000000001  CNVRT 0, 1, 0, 10  0111110001001010  // END OF PROGRAM  // (nothing here) |