# How to Use

# General Introductions

You can click each radio button to set the value of those registers and click the Set buttons to confirm making changes. And before clicking the Run or Single Step button, you need to click the IPL button first to read initial program. The Run button would run all the instructions stored in the Memory sequentially, and Single Step is used to execute the current instruction.

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

Please note that at the very beginning, before we load anything into memory, the system is all empty. In order to run programs, we need to have a boot program, which can be loaded either by the "Default IPL" button, or by loading a program file that contains the boot program. Under the latter circumstance, you should not click the "Default IPL" button.

# Running Programs

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

If different boot programs are needed, you have to restart the whole simulator to load a different boot program. (Because currently there is no mechanism for clearing boot program that has been loaded before.)

Please note that "Default IPL" and "Load" buttons only change the main memory. As a consequence, the content of registers (such as PC) will not be affected by such operations, which means if you want to jump among multiple programs (meaning not to execute programs sequentially), you have to remember their initial address (which would be provided when loading succeeds) and set 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.

## Program 2.txt:

1) Click "Load Bin" first to load "Boot Program.txt" as a boot program.

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

3) Click "Card Reader" button to read an external text file, "test 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.

## program\_2.txt

This program is almost the same with "Program 2.txt" except that it uses a built-in boot program. To run this program:

1) Click "Default IPL" before doing anything to load built-in boot program.

2) Click "Load Bin" to load "program\_2.txt".

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

4) Click "Run" to execute the program. 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 number in the sentence, the sentence number, and the word.

This version of program 2 is flawed in that the output form is not perfect and it can’t tolerate the space after a comma (under this circumstance, it will count one more to the word number in the sentence).

# 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, MVC pattern, 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

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). But in future we want to make some changes so that there would be Operator Console and Field Engineer Console in the Frame (hopefully one in the West and the other in the East).

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

The source code of program 1 can be found at the loadInitialProgram() method of RomLoader class.

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 use 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

There are some test sample (this piece of code can be put in the loadInitialProgram() method of the RomLoader class)：

this.memory.write(101, 42);

this.memory.write(102, 18468);

this.memory.write(103, 6335);

this.memory.write(104, 26501);

this.memory.write(105, 19170);

this.memory.write(106, 15725);

this.memory.write(107, 11479);

this.memory.write(108, 29359);

this.memory.write(109, 26963);

this.memory.write(110, 24465);

this.memory.write(111, 5706);

this.memory.write(112, 28146);

this.memory.write(113, 23282);

this.memory.write(114, 16828);

this.memory.write(115, 9962);

this.memory.write(116, 492);

this.memory.write(117, 2996);

this.memory.write(118, 11943);

this.memory.write(119, 4828);

this.memory.write(120, 5437);

And if the required one is 140, then the output should be :140,42

With those samples given, we still recommend you input your own test cases in order to validate the program though.

This is the total program1 in the ROM (this piece of code has already been put in the loadInitialProgram() method of the RomLoader class, and is deemed as the initial program):

// Data

this.memory.write(20, 21);

this.memory.write(24, 120);

this.memory.write(26, 200);

this.memory.write(25, 300);

this.memory.write(31, 20);

this.memory.write(29, 65535);

this.memory.write(30, 100);

// in：

// 8 LDX 3, 26 1000010011011010

this.memory.write(8, 34010);

// 9 JSR 3, 0 0011000011000000

this.memory.write(9, 12480);

// LDR 3,0,20

this.memory.write(200, 1812); //0000011100010100

// LDX 1, 24 1000010001011000

this.memory.write(201, 33880);

// IN 2, 0 1100011000000000

this.memory.write(202, 50688);

// OUT 2, 1 1100101000000001

this.memory.write(203, 51713);

// STR 2, 1, 0 0000101001000000

this.memory.write(204, 2624);

// LDA 2, 1, 0 0000111001000000

this.memory.write(205, 3648);

// SIR 2, 1 0001111000000001

this.memory.write(206, 7681);

// STR 2,0,24 0000101000011000

this.memory.write(207, 2584);

// LIX 1, 0 1000110001000000

this.memory.write(208, 35904);

// SOB 3, 3, 1 0011101111000001

this.memory.write(209, 15297);

// AIR 3, 10

this.memory.write(210, 6922); // 0001101100001010

// RFS 0 0011010000000000

this.memory.write(211, 13312);

// M[25]=300

// compare:

// 10 LIX 1, 0 1000110001000000

this.memory.write(10, 35904);

// 11 LIX 3, 0 1000110011000000

this.memory.write(11, 36032);

// 12 LDX 3, 25 1000010011011001

this.memory.write(12, 34009);

// 13 JSR 3, 0 0011000011000000

this.memory.write(13, 12480);

// LDR 3,0,31

this.memory.write(300, 1823); //0000011100011111

// LDX 1, 30

this.memory.write(301, 33886); //1000010001011110

// LDA 2, 1, 1

this.memory.write(302, 3649); //0000111001000001

// STR 2,0,30

this.memory.write(303, 2590); //0000101000011110

// LDX 2, 30

this.memory.write(304, 33950); //1000010010011110

// LDR 0, 2, 0

this.memory.write(305, 1152); //0000010010000000

// LDR 1, 1, 0

this.memory.write(306, 1344); //0000010101000000

// CMT 1, 0

this.memory.write(307, 22784); //0101100100000000

// LDR 0, 1, 0

this.memory.write(308, 1088); //0000010001000000

// LDR 2, 2, 0

this.memory.write(309, 1664); //0000011010000000

// CMB 0, 2

this.memory.write(310, 23680); //0101110010000000

// STR 0,0,27

this.memory.write(311, 2075); //0000100000011011

// LDR 0, 2, 0

this.memory.write(312, 1152); //0000010010000000

// SMR 1,0,27

this.memory.write(313, 5403); //0001010100011011

// STR 1,0,27

this.memory.write(314, 2331); //0000100100011011

// LDR 2,0,29

this.memory.write(315, 1565); //0000011000011101

// CMB 1, 2

this.memory.write(316, 23936); //0101110110000000

// TRR 1, 2

this.memory.write(317, 18816); //0100100110000000

// JCC 4, 3, 21

this.memory.write(318, 11221); //0010101111010101

// STR 0, 0, 28

this.memory.write(319, 2076); //0000100000011100

// STR 1, 0, 29

this.memory.write(320, 2333); //0000100100011101

// LDR 2, 0 ,30

this.memory.write(321, 1566); //0000011000011110

// AIR 2, 1

this.memory.write(322, 6657); //0001101000000001

// STR 2, 0, 30

this.memory.write(323, 2590); //0000101000011110

// LIX 2, 0

this.memory.write(324, 35968); //1000110010000000

// SOB 3, 3, 4

this.memory.write(325, 15300); //0011101111000100

// AIR 3, 14

this.memory.write(326, 6926); // 0001101100001110

// RFS 0

this.memory.write(327, 13312); //0011010000000000

// out：

// 14 LDR 1, 1, 0 0000010101000000

this.memory.write(14, 1344);

// 15 LDR 0, 0, 28 0000010000011100

this.memory.write(15, 1052);

// 16 OUT 1, 1 1100100100000001

this.memory.write(16, 51457);

// 17 OUT 0, 1 1100100000000001

this.memory.write(17, 51201);

# 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  // GETS 0, \_, \_, 2  // STR 0, 1, 0, 0  TRAP \_, \_, \_, 0  0111100000000000  STR 1, 1, 0, 0  0000100101000000  // // output paragraph  // LDR 0, 1, 0, 0  // PUTS 0, \_, \_, 1  // 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  // // output pattern  // LDR 0, 1, 0, 1  // PUTS 0, \_, \_, 1  //  // // M[7] = 0, for I/O control  // LDA 0, 0, 0, 0  // STR 0, 0, 0, 7    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) |