Министерство цифрового развития, связи

и массовых коммуникаций Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики» (СибГУТИ)

Кафедра вычислительных систем

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе

по дисциплине «**Архитектура ЭВМ**»

|  |  |  |
| --- | --- | --- |
| Выполнил:  студент гр. ИС-142  «\_\_» июня 2023 г. | \_\_\_\_\_\_\_\_\_\_\_\_ | /Григорьев Ю.В./ |
|  |  |  |
| Проверил:  преподаватель  «\_\_» июня 2023 г. | **\_\_\_\_\_\_\_\_\_\_\_\_** | /Майданов Ю.С./ |

Оценка «\_\_\_\_\_\_\_\_\_\_\_\_\_»

Новосибирск 2023

**ОГЛАВЛЕНИЕ**

[ПОСТАНОВКА ЗАДАЧИ 3](#_Toc35593781)

[БЛОК-СХЕМЫ АЛГОРИТМОВ 6](#_Toc35593782)

[ПРОГРАММНАЯ РЕАЛИЗАЦИЯ 9](#_Toc35593783)

[РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ 53](#_Toc35593783)

[ЗАКЛЮЧЕНИЕ 55](#_Toc35593783)

[ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА 55](#_Toc35593783)

# **ПОСТАНОВКА ЗАДАЧИ**

В рамках курсовой работы необходимо доработать модель *Simple Computer* так, чтобы она обрабатывала команды, записанные в оперативной памяти. Система команд представлена в таблице 1. Из пользовательских функций необходимо реализовать только одну согласно варианту задания (номеру вашей учетной записи). Для разработки программ требуется создать трансляторы с языков *Simple Assembler* и *Simple Basic*.

**Обработка команд центральным процессором**

Для выполнения программ моделью *Simple Computer* необходимо реализовать две функции:

**int *ALU*** (*int command, int operand*) – реализует алгоритм работы арифметико-логического устройства. Если при выполнении функции возникла ошибка, которая не позволяет дальше выполнять программу, то функция возвращает -1, иначе 0;

**int *CU*** (void) – обеспечивает работу устройства управления. Обработку команд осуществляет устройство управления. Функция *CU* вызывается либо обработчиком сигнала от системного таймера, если не установлен флаг «игнорирование тактовых импульсов», либо при нажатии на клавишу *t*.

Алгоритм работы функции следующий:

1. из оперативной памяти считывается ячейка, адрес которой хранится в регистре *instructionCounter*;  
2. полученное значение декодируется как команда;  
3. если декодирование невозможно, то устанавливаются флаги «указана неверная команда» и «игнорирование тактовых импульсов» (системный таймер можно отключить) и работа функции прекращается.  
4. Если получена арифметическая или логическая операция, то вызывается функция *ALU*, иначе команда выполняется самим устройством управления.  
5. Определяется, какая команда должна быть выполнена следующей и адрес её ячейки памяти заносится в регистр *instructionCounter*.  
6. Работа функции завершается.

**Транслятор с языка Simple Assembler**

Разработка программ для *Simple Computer* может осуществляться с использованием низкоуровневого языка *Simple Assembler*. Для того чтобы программа могла быть обработана *Simple Computer*необходимо реализовать транслятор, переводящий текст *Simple Assembler* в бинарный формат, которым может быть считан консолью управления.

Пример программы на **Simple Assembler:**

00 READ 09 ; (Ввод А)  
01 READ 10 ; (Ввод В)  
02 LOAD 09 ; (Загрузка А в аккумулятор)  
03 SUB 10 ; (Отнять В)  
04 JNEG 07 ; (Переход на 07, если отрицательное)  
05 WRITE 09 ; (Вывод А)  
06 HALT 00 ; (Останов)  
07 WRITE 10 ; (Вывод В)  
08 HALT 00 ; (Останов)  
09 = +0000 ; (Переменная А)  
10 = +9999 ; (Переменная В)

Программа транслируется по строкам, задающим значение одной ячейки памяти. Каждая строка состоит как минимум из трех полей: адрес ячейки памяти, команда (символьное обозначение), операнд. Четвертым полем может быть указан комментарий, который обязательно должен начинаться с символа точка с запятой. Название команд представлено в таблице 1. Дополнительно используется команда =, которая явно задает значение ячейки памяти в формате вывода его на экран консоли (+XXXX).

Команда запуска транслятора должна иметь вид: *sat* файл.*sa* файл.*o*, где файл.*sa*– имя файла, в котором содержится программа на *Simple Assembler*, файл.*o* – результат трансляции.

**Транслятор с языка Simple Basic**

Для упрощения программирования пользователю модели *Simple Computer* должен быть предоставлен транслятор с высокоуровневого языка *Simple Basic*. Файл, содержащий программу на *Simple Basic*, преобразуется в файл с кодом *Simple Assembler*. Затем *Simple Assembler*-файл транслируется в бинарный формат. В языке *Simple Basic* используются следующие операторы: *rem, input, output, goto, if, let, end*.

Пример программы на **Simple Basic:**

10 REM Это комментарий  
20 INPUT A  
30 INPUT B  
40 LET C = A – B  
50 IF C < 0 GOTO 20  
60 PRINT C  
70 END

Каждая строка программы состоит из номера строки, оператора *Simple Basic* и параметров. Номера строк должны следовать в возрастающем порядке. Все команды за исключением команды конца программы могут встречаться в программе многократно. *Simple Basic* должен оперировать с целыми выражениями, включающими операции +, -, \*, и /. Приоритет операций аналогичен C. Для того чтобы изменить порядок вычисления, можно использовать скобки.

Транслятор должен распознавания только букв верхнего регистра, то есть все символы в программе на *Simple Basic* должны быть набраны в верхнем регистре (символ нижнего регистра приведет к ошибке). Имя переменной может состоять только из одной буквы. *Simple Basic* оперирует только с целыми значениями переменных, в нем отсутствует объявление переменных, а упоминание переменной автоматически вызывает еѐ объявление и присваивает ей нулевое значение. Синтаксис языка не позволяет выполнять операций со строками.

**Оформление отчета по курсовой работе**

Отчет о курсовой работе представляется в виде пояснительной записки (ПЗ), к которой прилагается разработанное программное обеспечение. В пояснительную записку должны входить:

• титульный лист;  
• полный текст задания к курсовой работе;  
• реферат (объем ПЗ, количество таблиц, рисунков, схем, программ, приложений, краткая характеристика и результаты работы);  
• содержание:  
• постановка задачи исследования;  
• блок-схемы используемых алгоритмов;  
• программная реализация;  
• результаты проведенного исследования;  
• выводы;  
• список использованной литературы;  
• подпись, дата.

Пояснительная записка должна быть оформлена на листах формата А4, имеющих поля. Все листы следует сброшюровать и пронумеровать.

**БЛОК-СХЕМЫ АЛГОРИТМОВ**

1. CU()

Считывание ячейки памяти, номер которой обозначен в instructionCounter

Декодирование полученного значения в команду и операнд

Нет

Команда и операнд корректны?

Да

Выполнение команды управления / вызов ALU() для выполнения арифметической операции

Обновление интерфейса mySimpleComputer

Вывод сообщения об ошибке, установка флагов IGNORE, WRONG\_COMMAND / WRONG\_OPERAND

**2. Транслятор с Simple Assembler**

Получение имён входного и выходного файлов

Входной файл существует?

**Да**

**Нет**

Считывание строки из входного файла

Строка корректна?

**Нет**

**Да**

Кодирование команды, запись в память

Это последняя строка входного файла?

**­­**

**Нет**

**Да**

Вывод сообщения об ошибке

Запись полученного в выходной файл

**3.** Транслятор с Simple Basic

Входной файл существует?

Получение имён входного и выходного файлов

Нет

Да

Считывание строки из входного файла

Строка корректна? (проверка номера строки, команды, операнда)

Нет

Да

Обработка команды: проверка используемых переменных на корректность, получение адресов переменных, выполнение операции (в случае LET – транслирование выражения в RPN, выполнение RPN с помощью стек-машины)

Запись оттранслированного кода команды на Simple Assembler в выходной файл

Это последняя строка входного файла?

Нет

Вывод сообщения об ошибке

Да

# **ПРОГРАММНАЯ РЕАЛИЗАЦИЯ**

alu.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #include "alu.h"  int  ALU (int command, int operand)  {  int tmp, accum, a, b;  sc\_memoryGet (operand, &tmp);  sc\_accumGet (&accum);  switch (command)  {  case 0x30: // summation  sc\_accumSet (accum + tmp);  break;  case 0x31: // substraction  sc\_accumSet (accum - tmp);  break;  case 0x32: // division  if (tmp == 0)  {  sc\_regSet (FLAG\_DIV\_BY\_ZERO, 1);  return -1;  }  sc\_regSet (FLAG\_DIV\_BY\_ZERO, 0);  sc\_accumSet (accum / tmp);  break;  case 0x33: // multiplication  sc\_accumSet (accum \* tmp);  break;  case 0x52: // conjunction  a = accum;  b = tmp;  for (int i = 0; i < 8; i++)  {  if ((a & (1 << i)) & (b & (1 << i)))  {  sc\_accumSet (accum | (1 << i));  }  }  break;  }  return 0;  } |

alu.h

|  |  |
| --- | --- |
| 1  2  3  4  5 | #pragma once  #include "msc.h"  int ALU (int command, int operand); |

bc.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152 | #include "bc.h"  int  bc\_printA (char \*str)  {  ssize\_t len = strlen (str) \* sizeof (char);  write (STDOUT\_FILENO, "\E(0", 3);  if (write (STDOUT\_FILENO, str, len) != len)  {  return -1;  }  write (STDOUT\_FILENO, "\E(B", 3);  return 0;  }  int  bc\_box (int x1, int y1, int x2, int y2)  {  for (int i = 0; i < x2; i++)  {  for (int j = 0; j < y2; j++)  {  mt\_gotoXY (x1 + i, y1 + j);  if (i == 0 && j == 0)  {  bc\_printA ("l");  }  else if (i == 0 && j == y2 - 1)  {  bc\_printA ("k\n");  }  else if (i == x2 - 1 && j == 0)  {  bc\_printA ("m");  }  else if (i == x2 - 1 && j == y2 - 1)  {  bc\_printA ("j\n");  }  else if ((i == 0 || i == x2 - 1)  && (j > 0 && j < y2)) // horizontal line  {  bc\_printA ("q");  }  else if ((i > 0 && i < x2)  && (j == 0 || j == y2 - 1)) // vertical line  {  bc\_printA ("x");  }  else  {  write (STDOUT\_FILENO, " ", sizeof (char));  }  }  }  return 0;  }  int  bc\_printbigchar (int \*big, int x, int y, enum colors fgcolor,  enum colors bgcolor)  {  mt\_setbgcolor (bgcolor);  mt\_setfgcolor (fgcolor);  for (int i = 0; i < 8; i++)  {  for (int j = 0; j < 8; j++)  {  mt\_gotoXY (x + i, y + j);  short value;  bc\_getbigcharpos (big, i, j, &value);  if (!value)  {  write (STDOUT\_FILENO, " ", sizeof (char));  }  else  {  bc\_printA ("a");  }  }  write (STDOUT\_FILENO, "\n", sizeof (char));  }  mt\_setbgcolor (GREY);  mt\_setfgcolor (LIGHT\_GREY);  return 0;  }  int  bc\_setbigcharpos (int \*big, int x, int y, short int value)  {  if (x < 0 || x > 7 || y < 0 || y > 7 || value > 1 || value < 0)  {  return 1;  }  int part = x / 4;  x %= 4;  if (value)  {  big[part] |= (1 << (8 \* x + y));  }  else  {  big[part] &= ~(1 << (8 \* x + y));  }  return 0;  }  int  bc\_getbigcharpos (int \*big, int x, int y, short int \*value)  {  if (x < 0 || x > 7 || y < 0 || y > 7)  {  return -1;  }  short int part = x / 4;  x %= 4;  if (big[part] & (1 << (8 \* x + y)))  {  \*value = 1;  }  else  {  \*value = 0;  }  return 0;  }  int  bc\_bigcharwrite (int fd, int \*big, int count)  {  for (int i = 0; i < count \* 2; i++)  {  if (write (fd, &big[i], sizeof (int)) == -1)  {  return 1;  }  }  return 0;  }  int  bc\_bigcharread (int fd, int \*big, int need\_count, int \*count)  {  for (\*count = 0; (\*count < need\_count \* 2); \*count += 1)  {  if (read (fd, &big[\*count], sizeof (int)) == -1)  {  return 1;  }  }  return 0;  } |

bc.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #pragma once  #include "term.h"  int bc\_printA (char \*str);  int bc\_box (int x1, int y1, int x2, int y2);  int bc\_printbigchar (int \*big, int x, int y, enum colors fgcolor,  enum colors bgcolor);  int bc\_setbigcharpos (int \*big, int x, int y, short int value);  int bc\_getbigcharpos (int \*big, int x, int y, short int \*value);  int bc\_bigcharwrite (int fd, int \*big, int count);  int bc\_bigcharread (int fd, int \*big, int need\_count, int \*count); |

cu.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189 | #include "cu.h"  #include "alu.h"  #include "term.h"  #include "tui.h"  int  READ (int operand) // read new content for memory unit from console  {  g\_clearfields ();  mt\_gotoXY (INPUTFIELD\_X, 1);  mt\_printtext (" Input (hex):> ");  rk\_mytermsave ();  rk\_mytermregime (0, 0, 4, 1, 1);  char buff[5];  mt\_readtext (buff, sizeof (buff));  int value;  sscanf (buff, "%x", &value);  sc\_regSet (FLAG\_WRONG\_COMMAND, 0);  sc\_regSet (FLAG\_WRONG\_OPERAND, 0);  int cmd, oper;  int err = sc\_commandDecode (value, &cmd, &oper);  mt\_gotoXY (RESULTFIELD\_X, 1);  mt\_printtext (" ");  if (err == 0)  {  sc\_memorySet (operand, value);  mt\_setbgcolor (GREEN);  mt\_printtext (" SUCCESS ");  }  else  {  sc\_memorySet (operand, value);  mt\_setbgcolor (RED);  mt\_printtext (" FAIL : WRONG ");  if (err == ERR\_WRONG\_COMMAND)  {  sc\_regSet (FLAG\_WRONG\_COMMAND, 1);  mt\_printtext ("COMMAND");  }  else  {  sc\_regSet (FLAG\_WRONG\_OPERAND, 1);  mt\_printtext ("OPERAND");  }  mt\_printtext (" : WROTE ANYWAY ");  }  mt\_setbgcolor (GREY);  rk\_mytermrestore ();  g\_drawmemorybox ();  return 0;  }  int  WRITE (int operand) // write memory unit contents to console  {  g\_clearfields ();  char buff[6];  g\_getunit (operand, &buff);  char tmp[16];  sprintf (tmp, " Output:> %s", buff);  mt\_gotoXY (RESULTFIELD\_X, 1);  mt\_printtext (tmp);  return 0;  }  int  LOAD (int operand) // put value from accumulator to operand# memory cell  {  int value = 0;  sc\_memoryGet (operand, &value);  sc\_accumSet (value);  return 0;  }  int  STORE (int operand) // put operand from accumulator to memory  {  int accum;  sc\_accumGet (&accum);  sc\_memorySet (operand, accum);  return 0;  }  int  JUMP (int operand) // jump to instruction  {  sc\_countSet (operand);  CU ();  return 0;  }  int  JNEG (int operand) // jump to instruction if accumulator is negative  {  int accum;  sc\_accumGet (&accum);  if (accum < 0)  {  sc\_countSet (operand);  CU ();  }  return 0;  }  int  JZ (int operand) // jump if accumulator equals to zero  {  int accum;  sc\_accumGet (&accum);  if (accum == 0)  {  sc\_countSet (operand);  CU ();  }  return 0;  }  int  HALT () // set ignore flag to 1  {  sc\_regSet (FLAG\_IGNORE, 1);  alarm (0);  sc\_countSet (0);  return 0;  }  int  JNS (int operand)  {  int accum;  sc\_accumGet (&accum);  if (accum > 0)  {  sc\_countSet (operand);  }  return 0;  }  int  CU ()  {  int value = 0, count;  sc\_countGet (&count);  sc\_memoryGet (count, &value);  int command, operand;  if (sc\_commandDecode (value, &command, &operand) < 0)  {  sc\_regSet (FLAG\_IGNORE, 1);  return 0;  }  if (((command >= 0x30) && (command <= 0x33)) || (command == 0x52))  {  ALU (command, operand);  }  else  {  switch (command)  {  case 0x10:  READ (operand);  break;  case 0x11:  WRITE (operand);  break;  case 0x20:  LOAD (operand);  break;  case 0x21:  STORE (operand);  break;  case 0x40:  JUMP (operand);  break;  case 0x41:  JNEG (operand);  break;  case 0x42:  JZ (operand);  break;  case 0x43:  HALT ();  break;  case 0x55:  JNS (operand);  break;  }  }  return 0;  } |

cu.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #pragma once  #include "alu.h"  #include "msc.h"  #include "term.h"  #include "tui.h"  #include <string.h>  int READ (int operand);  int WRITE (int operand);  int LOAD (int operand);  int STORE (int operand);  int JUMP (int operand);  int JNEG (int operand);  int JZ (int operand);  int HALT (void);  int JNS (int operand);  int CU (void); |

main.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | #include "bc.h"  #include "msc.h"  #include "readkey.h"  #include "sig.h"  #include "term.h"  #include "tui.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int  main ()  {  sc\_memoryInit ();  sc\_regInit ();  sc\_regSet (FLAG\_IGNORE, 1);  sc\_accumSet (0);  sc\_countSet (0);  sig\_set ();  rk\_mytermsave ();  rk\_mytermregime (0, 0, 1, 0, 1);  mt\_clrscr ();  g\_interface ();  rk\_mytermrestore ();  return 0;  } |

msc.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187 | #include "msc.h"  int sc\_memory[MEMSIZE];  int sc\_register;  int sc\_accum;  int sc\_count;  int  sc\_memoryInit () // initializes the array of 100 elements  {  memset (sc\_memory, 0, MEMSIZE \* sizeof (sc\_memory[0]));  return 0;  }  int  sc\_memorySet (int address,  int value) // sets the value of [address] memory unit  {  if (address < 0 || address >= MEMSIZE)  {  BIT\_SET (sc\_register, FLAG\_WRONG\_ADDRESS);  return ERR\_WRONG\_ADDRESS;  }  sc\_memory[address] = value;  return 0;  }  int  sc\_memoryGet (int address, // gets the value of [address] memory unit and  int \*value) // returns it into value var  {  if (address < 0 || address >= MEMSIZE)  {  BIT\_SET (sc\_register, FLAG\_WRONG\_ADDRESS);  return ERR\_WRONG\_ADDRESS;  }  \*value = sc\_memory[address];  return 0;  }  int  sc\_memorySave (  char \*filename) // saves memory into a binary file (write/fwrite)  {  FILE \*f = fopen (filename, "wb");  if (!f)  {  return 1;  }  fwrite (sc\_memory, sizeof (int), sizeof (sc\_memory) / sizeof (int), f);  fclose (f);  return 0;  }  int  sc\_memoryLoad (char \*filename) // loads RAM from a file (read/fread)  {  FILE \*f = fopen (filename, "rb");  if (!f)  {  return 1;  }  fread (sc\_memory, sizeof (int), sizeof (sc\_memory) / sizeof (int), f);  fclose (f);  return 0;  }  int  sc\_regInit (void) // inits the register of flags with 0  {  sc\_register = 0;  return 0;  }  int  sc\_regSet (int reg, // sets the flag register value, #define-s are used for  int value) // register numbers, if wrong register number - error  {  if (reg < 1 || reg > 64)  {  return ERR\_WRONG\_FLAG;  }  if (!value)  {  BIT\_DEL (sc\_register, reg);  return 0;  }  if (value != 1)  {  BIT\_SET (sc\_register, FLAG\_OVERFLOW);  return ERR\_WRONG\_VALUE;  }  BIT\_SET (sc\_register, reg);  return 0;  }  int  sc\_regGet (int reg,  int \*value) // gets the flag value, if wrong register - error  {  if (reg < 1 || reg > 64)  {  return ERR\_WRONG\_FLAG;  }  \*value = BIT\_GET (sc\_register, reg);  return 0;  }  int  sc\_commandEncode (int command, // encodes command with a specific number and  int operand, // operand, puts the result in value, if wrong  int \*value) // command or operand - error, value not changes  {  if ((command > 0x0 && command < 0x10) || (command > 0x11 && command < 0x20)  || (command > 0x21 && command < 0x30)  || (command > 0x33 && command < 0x40)  || (command > 0x44 && command < 0x51) || command > 0x79)  {  sc\_regSet (FLAG\_WRONG\_COMMAND, 1);  return ERR\_WRONG\_COMMAND;  }  if (operand < 0 || operand > 127)  {  sc\_regSet (FLAG\_WRONG\_OPERAND, 1);  return ERR\_WRONG\_OPERAND;  }  \*value = \*value | (command << 7);  \*value = \*value | operand;  return 0;  }  int  sc\_commandDecode (  int value,  int \*command, // decodes value as a sc command, if decoding is impossible -  int \*operand) // sets error command and returns an error  {  if ((value >> 14) != 0)  {  sc\_regSet (FLAG\_WRONG\_COMMAND, 1);  return ERR\_WRONG\_COMMAND;  }  \*command = (value >> 7);  if ((\*command > 0x0 && \*command < 0x10)  || (\*command > 0x11 && \*command < 0x20)  || (\*command > 0x21 && \*command < 0x30)  || (\*command > 0x33 && \*command < 0x40)  || (\*command > 0x44 && \*command < 0x51) || \*command > 0x79)  {  sc\_regSet (FLAG\_WRONG\_COMMAND, 1);  return ERR\_WRONG\_COMMAND;  }  \*operand = value & 0b1111111;  return 0;  }  int  sc\_accumSet (int value)  {  sc\_accum = value;  return 0;  }  int  sc\_accumGet (int \*value)  {  \*value = sc\_accum;  return 0;  }  int  sc\_countSet (int value)  {  if (value < 0 || value > 99)  {  value = 0;  }  sc\_count = value;  return 0;  }  int  sc\_countGet (int \*value)  {  \*value = sc\_count;  return 0;  } |

msc.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | #pragma once  #include <inttypes.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MEMSIZE 100  // flags  #define FLAG\_IGNORE 32  #define FLAG\_WRONG\_COMMAND 16  #define FLAG\_WRONG\_OPERAND 8  #define FLAG\_WRONG\_ADDRESS 4  #define FLAG\_DIV\_BY\_ZERO 2  #define FLAG\_OVERFLOW 1  // errors  #define ERR\_WRONG\_ADDRESS -1  #define ERR\_WRONG\_FLAG -2  #define ERR\_WRONG\_VALUE -3  #define ERR\_WRONG\_COMMAND -4  #define ERR\_WRONG\_OPERAND -5  // bit operations  #define BIT\_SET(X, Y) X = X | (1 << (Y - 1))  #define BIT\_DEL(X, Y) X = X & (~(1 << (Y - 1)))  #define BIT\_GET(X, Y) X >> (Y - 1) & 0x1  int sc\_memoryInit (); // initializes the array of 100 elements  int sc\_memorySet (int address,  int value); // sets the value of [address] memory unit  int sc\_memoryGet (int address, // gets the value of [address] memory unit and  int \*value); // returns it into value var  int sc\_memorySave (  char \*filename); // saves memory into a binary file (write/fwrite)  int sc\_memoryLoad (char \*filename); // loads RAM from a file (read/fread)  int sc\_regInit (void); // inits the register of flags with 0  int sc\_regSet (  int register, // sets the flag register value, #define-s are used for  int value); // register numbers, if wrong register number - error  int sc\_regGet (int register,  int \*value); // gets the flag value, if wrong register - error  int  sc\_commandEncode (int command, // encodes command with a specific number and  int operand, // operand, puts the result in value, if wrong  int \*value); // command or operand - error, value not changes  int sc\_commandDecode (  int value, // decodes value as a sc command, if decoding is impossible  int \*command, // - sets error command and returns an error  int \*operand);  int sc\_accumGet (int \*value);  int sc\_accumSet (int value);  int sc\_countSet (int value);  int sc\_countGet (int \*value); |

readkey.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148 | #include "readkey.h"  struct termios current, backup;  int  rk\_readkey (enum keys \*key)  {  char buff[16];  int numRead;  if (tcgetattr (STDOUT\_FILENO, &backup) != 0)  {  return -1;  }  if (rk\_mytermregime (0, 0, 1, 0, 1) != 0)  {  return -1;  }  numRead = read (STDOUT\_FILENO, buff, 15);  buff[numRead] = '\0';  if (strcmp (buff, "l") == 0)  {  \*key = KEY\_L;  }  else if (strcmp (buff, "s") == 0)  {  \*key = KEY\_S;  }  else if (strcmp (buff, "r") == 0)  {  \*key = KEY\_R;  }  else if (strcmp (buff, "q") == 0)  {  \*key = KEY\_Q;  }  else if (strcmp (buff, "t") == 0)  {  \*key = KEY\_T;  }  else if (strcmp (buff, "i") == 0)  {  \*key = KEY\_I;  }  else if ((strcmp (buff, "\n") == 0) || (strcmp (buff, "\r") == 0))  {  \*key = KEY\_ENTER;  }  else if ((strcmp (buff, "\E[15~") == 0) || (strcmp (buff, "\E[[E") == 0))  {  \*key = KEY\_F5;  }  else if (strcmp (buff, "\E[17~") == 0)  {  \*key = KEY\_F6;  }  else if (strcmp (buff, "\E[A") == 0)  {  \*key = KEY\_UP;  }  else if (strcmp (buff, "\E[B") == 0)  {  \*key = KEY\_DOWN;  }  else if (strcmp (buff, "\E[C") == 0)  {  \*key = KEY\_RIGHT;  }  else if (strcmp (buff, "\E[D") == 0)  {  \*key = KEY\_LEFT;  }  if (tcsetattr (STDOUT\_FILENO, TCSANOW, &backup) != 0)  {  return -1;  }  return 0;  }  int  rk\_mytermsave ()  {  tcsetattr (STDOUT\_FILENO, TCSANOW, &current);  memcpy (&backup, &current, sizeof (backup));  return 0;  }  int  rk\_mytermrestore ()  {  return tcsetattr (STDOUT\_FILENO, TCSANOW, &backup);  }  int  rk\_mytermregime (int regime, int vtime, int vmin, int echo, int sigint)  {  if (tcgetattr (STDOUT\_FILENO, &current) != 0)  {  return -1;  }  // канонический режим = 1  if (regime)  {  current.c\_lflag |= ICANON;  }  else if (!regime)  {  current.c\_lflag &= ~ICANON;  }  else  {  return -1;  }  // неканонический режим работы  if (!regime)  {  // количество символов в очереди, чтобы read завершился  current.c\_cc[VTIME] = vtime;  // сколько времени ждать появления символа  current.c\_cc[VMIN] = vmin;  }  // символы будут отражаться по мере набора  if (echo == 1)  {  current.c\_lflag |= ECHO;  }  else if (!echo)  {  current.c\_lflag &= ~ECHO;  }  else  {  return -1;  }  // обработка клавиш прерывания  if (sigint)  {  current.c\_lflag |= ISIG;  }  else if (!sigint)  {  current.c\_lflag &= ~ISIG;  }  else  {  return -1;  }  return tcsetattr (STDOUT\_FILENO, TCSANOW, &current);  } |

readkey.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #pragma once  #include "term.h"  #include <string.h>  #include <termios.h>  #include <unistd.h>  enum keys  {  KEY\_S,  KEY\_L,  KEY\_R,  KEY\_T,  KEY\_I,  KEY\_F5,  KEY\_F6,  KEY\_UP,  KEY\_DOWN,  KEY\_RIGHT,  KEY\_LEFT,  KEY\_ENTER,  KEY\_Q,  KEY\_DEFAULT,  };  int rk\_readkey (enum keys \*key);  int rk\_mytermsave (void);  int rk\_mytermrestore (void);  int rk\_mytermregime (int regime, int vtime, int vmin, int echo, int sigint); |

rpn.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128 | #include "rpn.h"  void  stack\_push (char data, node \*\*top)  {  node \*tmp = (node \*)malloc (sizeof (node));  tmp->data = data;  tmp->next = \*top;  \*top = tmp;  }  char  stack\_pop (node \*\*top)  {  node \*tmp;  char d;  if (\*top == NULL)  {  return -1;  }  else  {  tmp = \*top;  \*top = tmp->next;  d = tmp->data;  free (tmp);  return d;  }  }  char  stack\_top (node \*top)  {  if (top != NULL)  {  return top->data;  }  return NULL;  }  int  checkPriority (char sign)  {  switch (sign)  {  case '\*':  case '/':  return 4;  case '+':  case '-':  return 2;  case '(':  case ')':  return 1;  }  return 0;  }  char \*  translateToRPN (char \*inf, char \*rpn)  {  node \*root = NULL;  int i = 0, j = 0;  while (inf[i] != '\0' && inf[i] != '\n')  {  char x = inf[i];  if ((x >= 'a' && x <= 'z') || (x >= 'A' && x <= 'Z'))  {  rpn[j] = x;  j++;  }  else if (x == '(')  {  stack\_push (x, &root);  }  else if (x == ')')  {  while (stack\_top (root) != '(')  {  char c = stack\_pop (&root);  if (c != 0)  {  rpn[j] = c;  j++;  }  }  stack\_pop (&root);  }  else if (x == '+' || x == '-' || x == '\*' || x == '/')  {  while (root != NULL  && checkPriority (root->data) >= checkPriority (x))  {  char c = stack\_pop (&root);  if (c != 0)  {  rpn[j] = c;  j++;  }  }  stack\_push (x, &root);  }  else if (x != ' ')  {  fprintf (stderr, "Wrong expression\n");  exit (EXIT\_FAILURE);  }  i++;  }  while (root != NULL)  {  char c = stack\_pop (&root);  if (c != 0)  {  rpn[j] = c;  j++;  }  }  for (int k = 0; k < j; k++)  {  if (rpn[k] == '(' || rpn[k] == ')')  {  fprintf (stderr, "Check your expression!\n");  exit (EXIT\_FAILURE);  }  }  rpn[j] = '\0';  return rpn; |

rpn.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | // RPN (Reverse Polish Notation) - logical operations form, where operands are  // located before operations stack machine - algorithm that performs  // calculations using RPN  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  typedef struct node  {  char data;  struct node \*next;  } node;  // stack machine functions  void stack\_push (char data, node \*\*top);  void stack\_print (node \*top);  char stack\_pop (node \*\*top);  char stack\_top (node \*top);  // RPN functions  int check\_priority (char sign);  char \*translateToRPN (char \*inf, char rpn[]); |

sat.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156 | #include "msc.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  FILE \*input = NULL;  void  load\_file (const char \*filename)  {  if ((input = fopen (filename, "r")) == NULL)  {  fprintf (stderr, "Can't open file: no such file.\n");  exit (EXIT\_FAILURE);  }  return;  }  void  translation (const char \*filename)  {  for (int i = 0; !feof (input); i++)  {  char line[255];  if (!fgets (line, 255, input))  {  if (feof (input))  {  break;  }  else  {  fprintf (stderr, "Line %d: can't read line from file.\n", i++);  break;  }  }  char \*addr;  char \*command;  char \*oper;  int num\_of\_cmd = 0, operand = 0, address = 0;  char \*ptr = strtok (line, " ");  addr = ptr;  ptr = strtok (NULL, " ");  command = ptr;  ptr = strtok (NULL, " +");  oper = ptr;  address = atoi (addr);  ptr = strtok (NULL, " ");  if (ptr != NULL && ptr[0] != ';')  {  fprintf (stderr, "Line %d: unexpected symbols.\n", ++i);  break;  }  if (!atoi (addr) && strcmp (addr, "00") != 0)  {  fprintf (stderr, "Line %d: expected address of memory cell.\n", ++i);  break;  }  operand = atoi (oper);  char buffer[255];  sprintf (buffer, "%02x", operand);  sscanf (buffer, "%02x", &operand);  if (!strcmp (command, "READ"))  {  num\_of\_cmd = 0x10;  }  else if (!strcmp (command, "WRITE"))  {  num\_of\_cmd = 0x11;  }  else if (!strcmp (command, "LOAD"))  {  num\_of\_cmd = 0x20;  }  else if (!strcmp (command, "STORE"))  {  num\_of\_cmd = 0x21;  }  else if (!strcmp (command, "ADD"))  {  num\_of\_cmd = 0x30;  }  else if (!strcmp (command, "SUB"))  {  num\_of\_cmd = 0x31;  }  else if (!strcmp (command, "DIVIDE"))  {  num\_of\_cmd = 0x32;  }  else if (!strcmp (command, "MUL"))  {  num\_of\_cmd = 0x33;  }  else if (!strcmp (command, "JUMP"))  {  num\_of\_cmd = 0x40;  }  else if (!strcmp (command, "JNEG"))  {  num\_of\_cmd = 0x41;  }  else if (!strcmp (command, "JZ"))  {  num\_of\_cmd = 0x42;  }  else if (!strcmp (command, "HALT"))  {  num\_of\_cmd = 0x43;  }  else if (!strcmp (command, "AND"))  {  num\_of\_cmd = 0x52;  }  else if (!strcmp (command, "JNS"))  {  num\_of\_cmd = 0x55;  }  else if (command[0] == '=')  {  sscanf (oper, "%x", &operand);  sc\_memorySet (address, operand);  continue;  }  else  {  fprintf (stderr, "Line %d: wrong command.\n", ++i);  break;  }  int value = 0;  if (sc\_commandEncode (num\_of\_cmd, operand, &value) < 0)  {  fprintf (stderr,  "Line %d: error by encode. Check command or operand.\n",  ++i);  break;  }  sc\_memorySet (address, value);  }  sc\_memorySave (strdup (filename));  }  int  main (int argc, const char \*\*argv)  {  if (argc < 3)  {  fprintf (stderr, "Usage: %s <input\_file.sa> <output\_object\_file.o>\n",  argv[0]);  exit (EXIT\_FAILURE);  }  sc\_memoryInit ();  load\_file (argv[1]);  translation (argv[2]);  return 0;  } |

sbt.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425  426  427  428  429  430  431  432  433  434  435  436  437  438  439  440  441  442  443  444  445  446  447  448  449  450  451  452  453  454  455  456  457  458  459  460  461  462  463  464  465  466  467  468  469  470  471  472  473  474  475  476  477  478  479  480  481  482  483  484  485  486  487  488  489  490  491  492  493  494  495  496  497  498  499  500  501  502  503  504  505  506  507  508  509  510  511  512  513  514  515  516  517  518  519  520  521  522  523  524  525  526  527  528  529  530  531  532  533  534  535  536  537  538  539  540  541  542  543  544  545  546  547  548  549  550  551  552  553  554  555  556  557  558  559  560  561  562  563  564  565  566  567  568  569  570  571  572  573  574  575  576  577  578  579  580  581  582  583  584  585  586  587  588  589  590  591  592  593  594  595  596  597  598  599  600  601  602  603  604  605  606  607  608  609  610  611  612  613  614  615  616  617  618  619  620  621  622  623  624  625  626  627  628  629  630  631  632  633  634 | #include "rpn.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void INPUT (char \*arguments);  void PRINT (char \*arguments);  void IF (char \*arguments);  void GOTO (int address, int operand);  void function (char \*command, char \*arguments);  void parseRPN (char \*rpn, char var);  void LET (char \*arguments);  void END ();  #define ERR\_EXPECTED\_ADDRESS\_OF\_MEMORY\_CELL -1  #define ERR\_WRONG\_COMMAND -2  #define ERR\_WRONG\_OPERAND -3  #define ERR\_BY\_ENCODE -4  struct variable  {  char name;  int address;  int value;  };  struct variable variables[52];  int variableCount = 0;  char lastConstantName = 'a';  struct command  {  int num;  char \*command;  int address;  };  FILE \*input = NULL;  FILE \*output = NULL;  const char commandID[][7]  = { "REM", "INPUT", "PRINT", "GOTO", "IF", "LET", "END" };  int basicCommandCouner = 0, assemblerCommandCounter = 0, gotoCounter = -1;  struct command \*program;  struct command \*gotoRecords;  int  getVariableAddress (char name)  {  for (int i = 0; i < 52; i++)  {  if (variables[i].name == name)  {  return variables[i].address;  }  if (variables[i].name == NULL)  {  variables[i].name = name;  variables[i].address = 99 - i;  variableCount++;  return variables[i].address;  }  }  return 0;  }  char  intToConstant (int value)  {  for (int i = 0; i < 52; i++)  {  if (variables[i].name == NULL)  {  variables[i].name = lastConstantName;  lastConstantName++;  variables[i].address = 99 - i;  variables[i].value = value;  fprintf (output, "%.2i = %04x\n", assemblerCommandCounter,  abs (variables[i].value));  variableCount++;  assemblerCommandCounter++;  return variables[i].name;  }  if (variables[i].value != NULL)  {  if (variables[i].value == value)  {  return variables[i].name;  }  }  }  }  char  preCalcProcessing (char \*expr)  {  char \*ptr = strtok (expr, " =");  char val;  sscanf (ptr, "%s", &val);  if (val < 'A' || val > 'Z')  {  fprintf (stderr, "Incorrect variable\n");  exit (EXIT\_FAILURE);  }  ptr = strtok (NULL, " ");  char \*equal = ptr;  if (strcmp (equal, "=") != 0)  {  fprintf (stderr, "Wrong expression\n");  exit (EXIT\_FAILURE);  }  ptr = strtok (NULL, "");  char \*exp = ptr;  int i = 0, j = 0, pos = 0, operat = 0, flg = 1, m = 0;  char \*assign = (char \*)malloc (sizeof (char) \* 255);  for (int k = 0; k < (int)strlen (exp); k++)  {  if (exp[k] == '-' && flg)  {  assign[m] = '0';  m++;  }  flg = 0;  if (exp[k] == '+' || exp[k] == '-' || exp[k] == '/' || exp[k] == '\*')  {  operat++;  }  if (exp[k] == '+' || exp[k] == '-' || exp[k] == '/' || exp[k] == '\*'  || exp[k] == '(')  {  flg = 1;  }  assign[m] = exp[k];  m++;  }  if (operat == 0) // 0+ перед ним, если перед минусом нет аргумента, то пишем  // 0 перед миунсом  {  sprintf (exp, "0 + %s", assign);  }  else  {  sprintf (exp, "%s", assign);  }  while (exp[i] != '\n' && exp[i] != '\0')  {  if (exp[i] >= '0' && exp[i] <= '9')  {  char num[256];  j = 0;  num[j] = exp[i];  j++;  pos = i;  exp[i] = ' ';  i++;  while (exp[i] >= '0' && exp[i] <= '9')  {  num[j] = exp[i];  j++;  exp[i] = ' ';  i++;  }  num[j] = '\0';  exp[pos] = intToConstant (atoi (num));  }  i++;  }  sprintf (expr, "%s", exp);  return val;  }  void  loadFile (const char \*input\_file, const char \*output\_file)  {  if ((input = fopen (input\_file, "r")) == NULL)  {  fprintf (stderr, "Input file '%s' does not exist\n", input\_file);  exit (EXIT\_FAILURE);  }  output = fopen (output\_file, "w");  }  void  basic\_translate ()  {  int instructionCounter = 1;  while (1)  {  char temp[255];  fgets (temp, 254, input);  if (feof (input))  {  break;  }  instructionCounter++;  }  basicCommandCouner = instructionCounter;  rewind (input);  program = (struct command \*)malloc (sizeof (struct command)  \* instructionCounter);  gotoRecords = (struct command \*)malloc (sizeof (struct command)  \* instructionCounter);  for (int i = 0; i < instructionCounter; i++)  {  program[i].command = (char \*)malloc (sizeof (char) \* 255);  if (!fgets (program[i].command, 254, input))  {  if (feof (input))  {  return;  }  else  {  fprintf (stderr, "Line %d: can't read line from file\n", i++);  return;  }  }  }  for (int i = 0; i < instructionCounter; i++)  {  char \*lin;  char \*thisCommand = (char \*)malloc (sizeof (char) \* 255);  sprintf (thisCommand, "%s", program[i].command);  char \*ptr = strtok (thisCommand, " ");  lin = ptr;  int line = atoi (lin);  if ((strcmp (lin, "0") == 0) || (strcmp (lin, "00") == 0))  {  fprintf (stderr, "Line %d: expected line number\n", i++);  break;  }  if (i - 1 >= 0)  {  if (line <= program[i - 1].num)  {  fprintf (stderr, "Line %d: wrong line number\n", i++);  break;  }  }  program[i].num = line;  char \*command;  ptr = strtok (NULL, " ");  command = ptr;  char \*arguments;  ptr = strtok (NULL, "");  arguments = ptr;  program[i].address = assemblerCommandCounter;  if (strcmp (command, "GOTO") != 0)  {  function (command, arguments);  }  else  {  gotoCounter++;  gotoRecords[gotoCounter].num = program[i].num;  gotoRecords[gotoCounter].command = program[i].command;  gotoRecords[gotoCounter].address = program[i].address;  assemblerCommandCounter++;  }  }  for (int i = 0; i <= gotoCounter; i++)  {  int address = gotoRecords[i].address;  char \*ptr = strtok (gotoRecords[i].command, " ");  ptr = strtok (NULL, " ");  ptr = strtok (NULL, "");  int operand = atoi (ptr);  GOTO (address, operand);  }  }  void  INPUT (char \*arguments)  {  if ((strlen (arguments) != 2) || (arguments[0] < 'A')  || (arguments[0] > 'Z'))  {  fprintf (stderr, "Wrong variable name\n");  exit (EXIT\_FAILURE);  }  fprintf (output, "%.2i READ %d\n", assemblerCommandCounter,  getVariableAddress (arguments[0]));  assemblerCommandCounter++;  }  void  PRINT (char \*arguments)  {  if ((strlen (arguments) != 2) || (arguments[0] < 'A')  || (arguments[0] > 'Z'))  {  fprintf (stderr, "Wrong variable name.\n");  exit (EXIT\_FAILURE);  }  fprintf (output, "%.2i WRITE %d\n", assemblerCommandCounter,  getVariableAddress (arguments[0]));  assemblerCommandCounter++;  }  void  parseRPN (char \*rpn, char var)  {  int i = 0, depth = 0, operand1, operand2;  char memoryCounter = '1';  while (rpn[i] != '\0' && rpn[i] != '\n')  {  char x = rpn[i];  if ((x >= 'a' && x <= 'z') || (x >= 'A' && x <= 'Z'))  {  fprintf (output, "%.2i LOAD %d\n", assemblerCommandCounter,  getVariableAddress (x));  assemblerCommandCounter++;  fprintf (output, "%.2i STORE %d\n", assemblerCommandCounter,  getVariableAddress (memoryCounter));  memoryCounter++;  assemblerCommandCounter++;  depth++;  }  if (x == '+' || x == '-' || x == '\*' || x == '/')  {  if (depth < 2)  {  fprintf (stderr, "Incorrect LET statement\n");  exit (EXIT\_FAILURE);  }  else  {  operand1 = getVariableAddress (memoryCounter - 2);  operand2 = getVariableAddress (memoryCounter - 1);  fprintf (output, "%.2i LOAD %d\n", assemblerCommandCounter,  operand1); // right operand goes to accum  assemblerCommandCounter++;  switch (x)  {  case '+': // summation  fprintf (output, "%.2i ADD %d\n", assemblerCommandCounter,  operand2);  assemblerCommandCounter++;  break;  case '-': // substraction  fprintf (output, "%.2i SUB %d\n", assemblerCommandCounter,  operand2);  assemblerCommandCounter++;  break;  case '/': // division  fprintf (output, "%.2i DIVIDE %d\n", assemblerCommandCounter,  operand2);  assemblerCommandCounter++;  break;  case '\*': // multiplication  fprintf (output, "%.2i MUL %d\n", assemblerCommandCounter,  operand2);  assemblerCommandCounter++;  break;  }  fprintf (output, "%.2i STORE %d\n", assemblerCommandCounter,  getVariableAddress (memoryCounter - 2));  assemblerCommandCounter++;  depth--;  memoryCounter = memoryCounter - 1;  }  }  i++;  }  if (depth != 1)  {  fprintf (stderr, "Incorrect LET statement\n");  exit (EXIT\_FAILURE);  }  fprintf (output, "%.2i STORE %d\n", assemblerCommandCounter,  getVariableAddress (var));  assemblerCommandCounter++;  }  void  GOTO (int address, int operand)  {  for (int i = 0; i < basicCommandCouner; i++)  {  if (program[i].num == operand)  {  fprintf (output, "%.2i JUMP %d\n", address, program[i].address);  return;  }  }  fprintf (stderr, "Reference to an inspecifed memory location\n");  exit (EXIT\_FAILURE);  }  void  IF (char \*arguments)  {  int mySign = -1, before = 0, after = 0;  for (int i = 0; i < (int)strlen (arguments); i++)  {  if ((arguments[i] == '>') || (arguments[i] == '<')  || (arguments[i] == '='))  {  mySign = i;  if (!(arguments[i - 1] == ' '))  {  before = 1;  }  if (!(arguments[i + 1] == ' '))  {  after = 1;  }  break;  }  }  char \*expression = (char \*)malloc (sizeof (char) \* 255);  if (!(before) && !(after))  {  expression = strtok (arguments, "");  }  else  {  int j = 0;  for (int i = 0; i < (int)strlen (arguments); i++)  {  if (i == mySign)  {  if (before)  {  expression[j] = ' ';  j++;  }  expression[j] = arguments[i];  j++;  if (after)  {  expression[j] = ' ';  j++;  }  }  else  {  expression[j] = arguments[i];  j++;  }  }  expression[j] = '\0';  }  char \*ptr = strtok (expression, " ");  char \*operand1 = ptr;  char operand1Name;  if (strlen (operand1) > 1)  {  if (atoi (operand1))  {  operand1Name = intToConstant (atoi (operand1));  }  }  else  {  if ((operand1[0] >= '0') && (operand1[0] <= '9'))  {  operand1Name = intToConstant (atoi (operand1));  }  else if ((operand1[0] >= 'A') && (operand1[0] <= 'Z'))  {  operand1Name = operand1[0];  }  else  {  fprintf (stderr, "Wrong statement\n");  exit (EXIT\_FAILURE);  }  }  ptr = strtok (NULL, " ");  char \*logicalSign = ptr;  ptr = strtok (NULL, " ");  char \*operand2 = ptr;  char operand2Name;  if (strlen (operand2) > 1)  {  if (atoi (operand2))  {  operand2Name = intToConstant (atoi (operand2));  }  }  else  {  if ((operand2[0] >= '0') && (operand2[0] <= '9'))  {  operand2Name = intToConstant (atoi (operand2));  }  else if ((operand2[0] >= 'A') && (operand2[0] <= 'Z'))  {  operand2Name = operand2[0];  }  else  {  fprintf (stderr, "Wrong statement\n");  exit (EXIT\_FAILURE);  }  }  int falsePosition = -1;  if (logicalSign[0] == '<')  {  fprintf (output, "%.2i LOAD %d\n", assemblerCommandCounter,  getVariableAddress (operand1Name));  assemblerCommandCounter++;  fprintf (output, "%.2i SUB %d\n", assemblerCommandCounter,  getVariableAddress (operand2Name));  assemblerCommandCounter++;  fprintf (output, "%.2i JNEG %d\n", assemblerCommandCounter,  assemblerCommandCounter + 2);  assemblerCommandCounter++;  falsePosition = assemblerCommandCounter;  assemblerCommandCounter++;  }  else if (logicalSign[0] == '>')  {  fprintf (output, "%.2i LOAD %d\n", assemblerCommandCounter,  getVariableAddress (operand2Name));  assemblerCommandCounter++;  fprintf (output, "%.2i SUB %d\n", assemblerCommandCounter,  getVariableAddress (operand1Name));  assemblerCommandCounter++;  fprintf (output, "%.2i JNEG %d\n", assemblerCommandCounter,  assemblerCommandCounter + 2);  assemblerCommandCounter++;  falsePosition = assemblerCommandCounter;  assemblerCommandCounter++;  }  else if (logicalSign[0] == '=')  {  fprintf (output, "%.2i LOAD %d\n", assemblerCommandCounter,  getVariableAddress (operand1Name));  assemblerCommandCounter++;  fprintf (output, "%.2i SUB %d\n", assemblerCommandCounter,  getVariableAddress (operand2Name));  assemblerCommandCounter++;  fprintf (output, "%.2i JZ %d\n", assemblerCommandCounter,  assemblerCommandCounter + 2);  assemblerCommandCounter++;  falsePosition = assemblerCommandCounter;  assemblerCommandCounter++;  }  ptr = strtok (NULL, " ");  char \*command = ptr;  ptr = strtok (NULL, "");  char \*commandArguments = ptr;  if (strcmp (command, "IF") == 0)  {  fprintf (stderr, "Multiple IFs\n");  exit (EXIT\_FAILURE);  }  else if (strcmp (command, "GOTO") != 0)  {  function (command, commandArguments);  }  else  {  gotoCounter++;  gotoRecords  = realloc (gotoRecords, sizeof (struct command) \* gotoCounter + 1);  struct command gotoCommand;  gotoCommand.address = assemblerCommandCounter;  char \*buff = (char \*)malloc (sizeof (char) \* 255);  sprintf (buff, "%d GOTO %s", falsePosition, commandArguments);  gotoCommand.command = buff;  gotoRecords[gotoCounter] = gotoCommand;  assemblerCommandCounter++;  }  fprintf (output, "%.2i JUMP %d\n", falsePosition, assemblerCommandCounter);  }  void  LET (char \*arguments)  {  char fin[255];  char var = preCalcProcessing (arguments);  translateToRPN (arguments, fin);  parseRPN (fin, var);  }  void  END ()  {  fprintf (output, "%.2i HALT 00\n", assemblerCommandCounter);  assemblerCommandCounter++;  }  void  function (char \*command, char \*arguments)  {  if (strcmp (command, "REM") == 0)  {  }  else if (strcmp (command, "INPUT") == 0)  {  INPUT (arguments);  }  else if (strcmp (command, "PRINT") == 0)  {  PRINT (arguments);  }  else if (strcmp (command, "IF") == 0)  {  IF (arguments);  }  else if (strcmp (command, "LET") == 0)  {  LET (arguments);  }  else if (strcmp (command, "END") == 0)  {  END ();  }  else  {  fprintf (stderr, "%s is not a command.\n", command);  }  }  int  main (int argc, const char \*\*argv)  {  if (argc < 3)  {  fprintf (stderr,  "Usage: %s <input\_basic\_file.sb> <output\_assembler\_file.sa>\n",  argv[0]);  exit (EXIT\_FAILURE);  }  loadFile (argv[1], argv[2]);  basic\_translate ();  fclose (input);  fclose (output);  return 0;  } |

sig.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include "sig.h"  #include "cu.h"  #include "tui.h"  int  sig\_handle\_reset ()  {  sc\_memoryInit ();  sc\_regInit ();  sc\_regSet (FLAG\_IGNORE, 1);  alarm (0);  return 0;  }  int  sig\_handle\_alarm ()  {  CU ();  int value;  sc\_countGet (&value);  sc\_countSet (value + 1);  sc\_countGet (&value);  g\_drawboxes ();  int flag;  int x = value / 10;  int y = value % 10;  g\_highlightmemory (x, y);  g\_drawbcbox ();  sc\_regGet (FLAG\_IGNORE, &flag);  alarm (!flag);  return 0;  }  int  sig\_set ()  {  signal (SIGUSR1, sig\_handle\_reset);  signal (SIGALRM, sig\_handle\_alarm);  return 0;  } |

sig.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #pragma once  #include "msc.h"  #include <signal.h>  #include <unistd.h>  int sig\_handle\_reset (void);  int sig\_handle\_alarm (void);  int sig\_set (void); |

term.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79 | #include "term.h"  int  mt\_clrscr (void)  {  if ((unsigned long)write (STDOUT\_FILENO, CLEAR, strlen (CLEAR))  < sizeof (char) \* strlen (CLEAR))  {  return -1;  }  return 0;  }  int  mt\_gotoXY (int x, int y)  {  char go[30];  sprintf (go, "\E[%d;%dH", x, y);  if ((unsigned long)write (STDOUT\_FILENO, go, strlen (go))  < sizeof (char) \* strlen (go))  {  return -1;  }  return 0;  }  int  mt\_getscreensize (int \*rows, int \*cols)  {  struct winsize ws;  if (ioctl (1, TIOCGWINSZ, &ws))  {  return -1;  }  \*rows = ws.ws\_row;  \*cols = ws.ws\_col;  return 0;  }  int  mt\_setfgcolor (enum colors color)  {  char foreground[30];  sprintf (foreground, "\E[38;5;%dm", color);  if ((unsigned long)write (STDOUT\_FILENO, foreground, strlen (foreground))  < sizeof (char) \* strlen (foreground))  {  return -1;  }  return 0;  }  int  mt\_setbgcolor (enum colors color)  {  char background[30];  sprintf (background, "\E[48;5;%dm", color);  if ((unsigned long)write (STDOUT\_FILENO, background, strlen (background))  < sizeof (char) \* strlen (background))  {  return -1;  }  return 0;  }  int  mt\_readtext (char \*text, int size)  {  int numRead = read (STDOUT\_FILENO, text, size);  text[numRead] = '\0';  return 0;  }  int  mt\_printtext (char \*text)  {  write (STDOUT\_FILENO, text, strlen (text));  return 0;  } |

term.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #pragma once  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/ioctl.h>  #include <unistd.h>  #define CLEAR "\E[H\E[2J"  enum colors  {  LIGHT\_GREY = 247,  GREY = 237,  BLACK = 16,  RED = 124,  GREEN = 41,  BLUE = 20,  YELLOW = 184,  };  int mt\_clrscr (void); // clears screen and moves cursor to upper left corner  int mt\_gotoXY (  int x,  int y); // moves cursor to the entered coordinates (x, y) = (row, col)  int mt\_getscreensize (  int \*rows, int \*cols); // gets terminal screen size (num of rows and cols)  int mt\_setfgcolor (enum colors color); // sets a background color for all rows  // and cols, entire terminal  int mt\_setbgcolor (  enum colors color); // sets a background color only for upcoming characters  int mt\_readtext (char \*text, int size);  int mt\_printtext (char \*text); |

tui.c

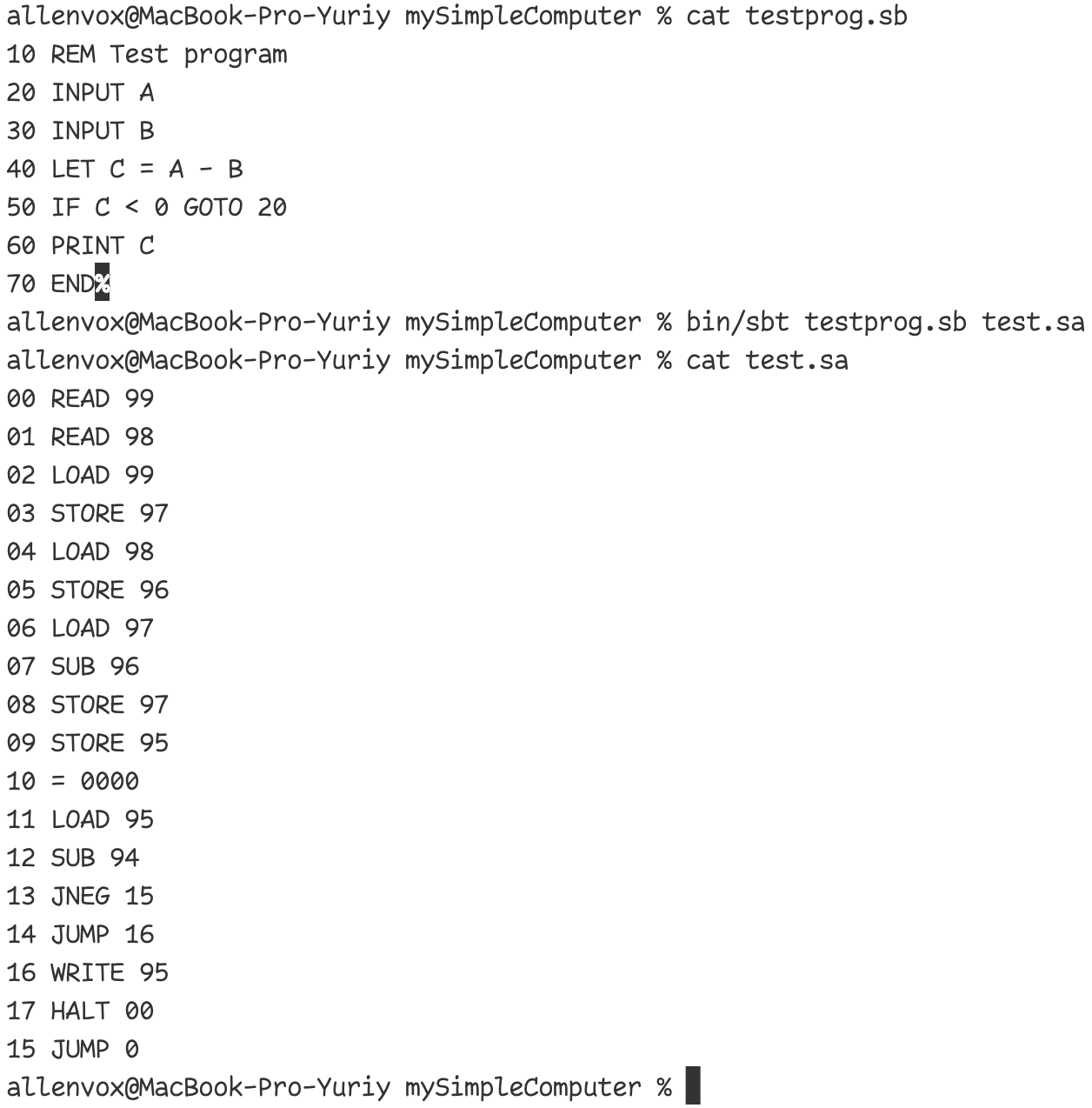
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425  426  427  428 | #include "tui.h"  #include "cu.h"  int  g\_writeflags (char \*\*val)  {  int flag = 0;  sc\_regGet (1, &flag);  char F = flag == 1 ? 'F' : ' ';  sc\_regGet (2, &flag);  char D = flag == 1 ? 'D' : ' ';  sc\_regGet (4, &flag);  char A = flag == 1 ? 'A' : ' ';  sc\_regGet (8, &flag);  char O = flag == 1 ? 'O' : ' ';  sc\_regGet (16, &flag);  char C = flag == 1 ? 'C' : ' ';  sc\_regGet (32, &flag);  char I = flag == 1 ? 'I' : ' ';  char buff[24] = { 0 };  sprintf (buff, "%c %c %c %c %c %c\0", F, D, A, O, C, I);  \*val = buff;  return 0;  }  int  g\_drawborders (void)  {  bc\_box (1, 1, 12, 63);  mt\_gotoXY (0, 28);  write (STDOUT\_FILENO, " Memory ", 8 \* sizeof (char));  bc\_box (1, 64, 3, 39);  mt\_gotoXY (1, 77);  write (STDOUT\_FILENO, " accumulator ", 13 \* sizeof (char));  bc\_box (4, 64, 3, 39);  mt\_gotoXY (4, 73);  write (STDOUT\_FILENO, " instructionCounter ", 20 \* sizeof (char));  bc\_box (7, 64, 3, 39);  mt\_gotoXY (7, 78);  write (STDOUT\_FILENO, " Operation ", 11 \* sizeof (char));  bc\_box (10, 64, 3, 39);  mt\_gotoXY (10, 79);  write (STDOUT\_FILENO, " Flags ", 7 \* sizeof (char));  bc\_box (13, 1, 12, 63);  bc\_box (13, 64, 12, 39);  mt\_gotoXY (13, 67);  write (STDOUT\_FILENO, " Keys ", 7 \* sizeof (char));  char \*str[7] = { "l - load",  "s - save",  "r - run",  "t - step",  "i - reset",  "F5 - accumulator",  "F6 - instructionCounter" };  for (int i = 0; i < 7; i++)  {  mt\_gotoXY (15 + i, 66);  write (STDOUT\_FILENO, str[i], strlen (str[i]));  }  mt\_gotoXY (33, 0);  return 0;  }  int  g\_drawaccumbox (void)  {  mt\_gotoXY (2, 80);  char buff[5];  int val;  sc\_accumGet (&val);  sprintf (buff, "%d", val);  write (STDOUT\_FILENO, buff, 5 \* sizeof (char));  mt\_gotoXY (33, 0);  return 0;  }  int  g\_drawflagbox (void)  {  mt\_gotoXY (11, 78);  char \*val;  g\_writeflags (&val);  write (STDOUT\_FILENO, val, 24 \* sizeof (char));  mt\_gotoXY (33, 0);  return 0;  }  int  g\_drawbcbox ()  {  int big[18][2] = {  { 0x4242423C, 0x3C424242 }, // 0  { 0x48506040, 0x40404040 }, // 1  { 0x2042423C, 0x7E020418 }, // 2  { 0x3840423C, 0x3C424040 }, // 3  { 0x7E424478, 0x40404040 }, // 4  { 0x3E02027E, 0x3C424040 }, // 5  { 0x3E02423C, 0x3C424242 }, // 6  { 0x2040407E, 0x10101010 }, // 7  { 0x3C42423C, 0x3C424242 }, // 8  { 0x7C42423C, 0x3C424040 }, // 9  { 0x66663C18, 0x66667E7E }, // A  { 0x3E66663E, 0x3E666666 }, // B  { 0x0202423C, 0x3C420202 }, // C  { 0x4444443E, 0x3E444444 }, // D  { 0x3E02027E, 0x7E020202 }, // E  { 0x1E02027E, 0x02020202 }, // F  { 0x7E181800, 0x0018187E }, // +  { 0x7E000000, 0x7E } // -  };  int address, val;  sc\_countGet (&address);  char buff[6];  g\_getunit (address, &buff);  int \*digit = malloc (2 \* sizeof (int));  for (int i = 0; i < 5; i++)  {  int k = 0;  switch (buff[i])  {  case '0':  k = 0;  break;  case '1':  k = 1;  break;  case '2':  k = 2;  break;  case '3':  k = 3;  break;  case '4':  k = 4;  break;  case '5':  k = 5;  break;  case '6':  k = 6;  break;  case '7':  k = 7;  break;  case '8':  k = 8;  break;  case '9':  k = 9;  break;  case 'a':  k = 10;  break;  case 'b':  k = 11;  break;  case 'c':  k = 12;  break;  case 'd':  k = 13;  break;  case 'e':  k = 14;  break;  case 'f':  k = 15;  break;  case '+':  k = 16;  break;  default:  k = 17;  break;  }  digit[0] = big[k][0];  digit[1] = big[k][1];  bc\_printbigchar (digit, BC\_X, BC\_START + i \* BC\_STEP, GREEN, GREY);  }  mt\_gotoXY (33, 0);  return 0;  }  int  g\_clearfields ()  {  for (int j = 25; j < 28; j++)  {  mt\_gotoXY (j, 1);  for (int i = 0; i < 102; i++)  {  mt\_printtext (" ");  }  }  return 0;  }  int  g\_loadmemory (void)  {  g\_clearfields ();  mt\_gotoXY (INPUTFIELD\_X, 1);  mt\_printtext (" Load from file:> ");  rk\_mytermregime (0, 0, 1, 1, 1);  char \*buff = calloc (0, sizeof (char) \* 100);  scanf ("%s", buff);  mt\_gotoXY (RESULTFIELD\_X, 1);  mt\_printtext (" ");  if (sc\_memoryLoad (buff) == 0)  {  mt\_setbgcolor (GREEN);  mt\_printtext (" SUCCESS ");  }  else  {  mt\_setbgcolor (RED);  mt\_printtext (" FAIL ");  }  mt\_setbgcolor (GREY);  g\_drawboxes ();  return 0;  }  int  g\_savememory (void)  {  g\_clearfields ();  mt\_gotoXY (INPUTFIELD\_X, 1);  mt\_printtext (" Save to file:> ");  rk\_mytermregime (0, 0, 1, 1, 1);  char \*buff = calloc (0, sizeof (char) \* 100);  scanf ("%s", buff);  sc\_memorySave (buff);  mt\_gotoXY (RESULTFIELD\_X, 1);  mt\_printtext (" ");  mt\_setbgcolor (GREEN);  mt\_printtext (" SUCCESS ");  mt\_setbgcolor (GREY);  g\_drawboxes ();  return 0;  }  int  g\_getunit (int address, char \*buff)  {  int val, command, operand;  sc\_memoryGet (address, &val);  if (val < 0)  {  sprintf (buff, "-%04x", abs (val));  return 0;  }  char temp[6];  int err = sc\_commandDecode (val, &command, &operand);  sprintf (temp, " %04x", val);  if (err == 0)  {  temp[0] = '+';  }  else  {  temp[0] = '-';  }  strcpy (buff, temp);  return 0;  }  int  g\_highlightmemory (int x, int y)  {  g\_drawmemorybox ();  int address = x \* 10 + y;  char buff[6];  g\_getunit (address, &buff);  mt\_gotoXY (X\_START + x, Y\_START + y \* Y\_STEP);  mt\_setbgcolor (BLUE);  mt\_printtext (buff);  mt\_setbgcolor (GREY);  return 0;  }  int  g\_drawoperationbox (void)  {  mt\_gotoXY (8, 79);  int address, val, cmd = 0, oper = 0;  sc\_countGet (&address);  sc\_memoryGet (address, &val);  char buff[11];  int err = sc\_commandDecode (val, &cmd, &oper);  sprintf (buff, " %02x:%d", cmd, oper);  if (err == 0)  {  buff[0] = '+';  }  else  {  buff[0] = '-';  }  mt\_printtext (buff);  mt\_gotoXY (33, 0);  return 0;  }  int  g\_drawcounterbox (void)  {  mt\_gotoXY (5, 80);  char buff[5];  int val;  sc\_countGet (&val);  sprintf (buff, "%04d", val);  write (STDOUT\_FILENO, buff, 5 \* sizeof (char));  mt\_gotoXY (33, 0);  return 0;  }  int  g\_drawmemorybox (void)  {  int k = 0;  for (int i = X\_START; i < 12; i++)  {  for (int j = Y\_START; j < 63; j += Y\_STEP)  {  mt\_gotoXY (i, j);  char buff[6];  g\_getunit (k++, &buff);  write (STDERR\_FILENO, buff, 6 \* sizeof (char));  }  }  return 0;  }  int  g\_drawboxes ()  {  g\_drawmemorybox ();  g\_drawaccumbox ();  g\_drawcounterbox ();  g\_drawoperationbox ();  g\_drawflagbox ();  g\_drawbcbox ();  return 0;  }  int  g\_interface ()  {  mt\_setbgcolor (GREY);  g\_drawborders ();  g\_clearfields ();  int exit = 0, address = 0, flag = 0;  while (!exit)  {  sc\_countGet (&address);  g\_drawboxes ();  int x = address / 10, y = address % 10;  g\_highlightmemory (x, y);  enum keys key = KEY\_DEFAULT;  rk\_readkey (&key);  switch (key)  {  case KEY\_Q: // exit  exit++;  break;  case KEY\_UP:  address -= 10;  if (address < 0)  {  address += 100;  }  sc\_countSet (address);  break;  case KEY\_DOWN:  address += 10;  if (address > 99)  {  address -= 100;  }  sc\_countSet (address);  break;  case KEY\_LEFT:  address -= 1;  if (address % 10 == 9 || address == -1)  {  address += 10;  }  sc\_countSet (address);  break;  case KEY\_RIGHT:  address += 1;  if (address % 10 == 0)  {  address -= 10;  }  sc\_countSet (address);  break;  case KEY\_L: // load memory  g\_loadmemory ();  break;  case KEY\_S: // save memory  g\_savememory ();  break;  case KEY\_I: // reset  raise (SIGUSR1);  x = 0, y = 0;  g\_drawboxes ();  break;  case KEY\_ENTER: // setting values  READ (address);  break;  case KEY\_R: // ignore flag  sc\_regGet (FLAG\_IGNORE, &flag);  sc\_regSet (FLAG\_IGNORE, !flag);  alarm (flag);  break;  case KEY\_T: // step key - calling control unit  CU ();  break;  case KEY\_F5:  break;  case KEY\_F6:  break;  }  }  return 0;  } |

tui.h

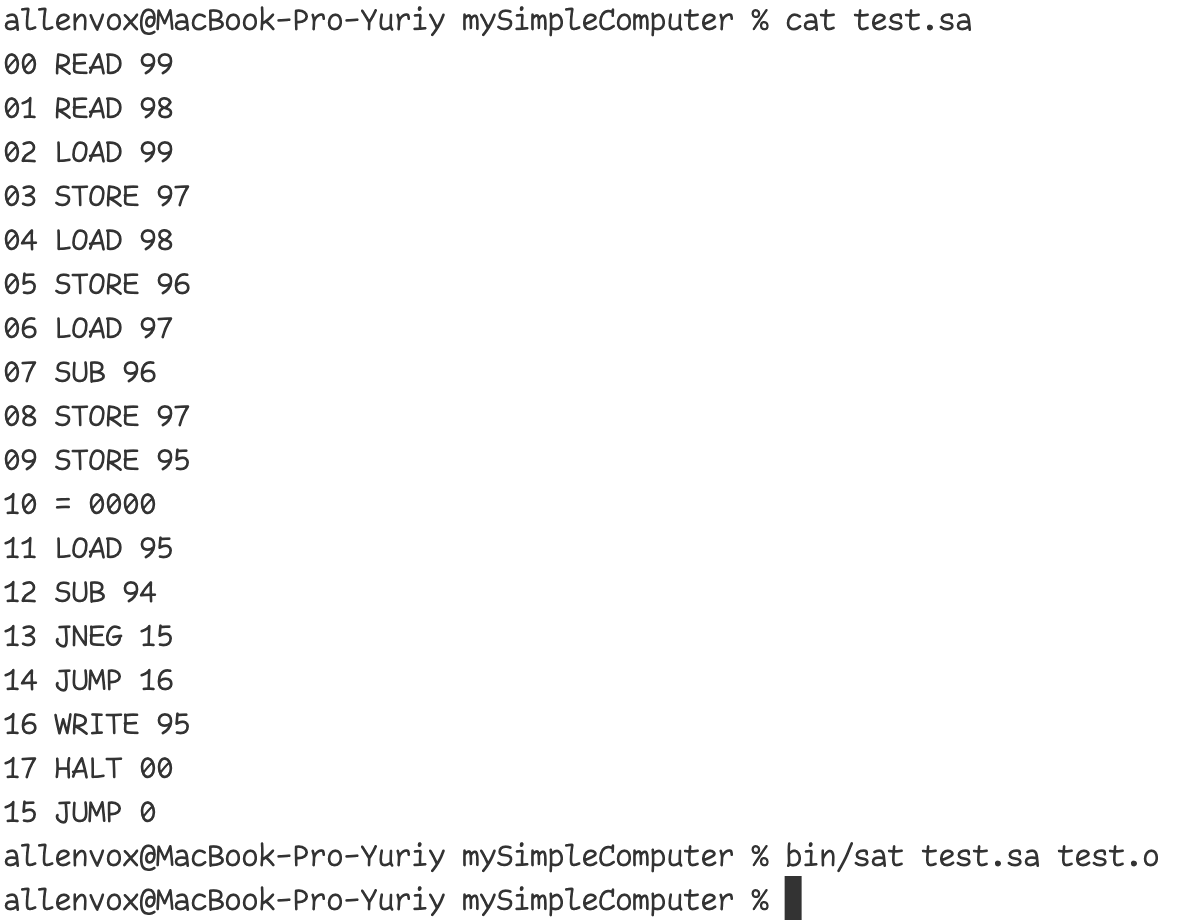
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #pragma once  #include "bc.h"  #include "msc.h"  #include "readkey.h"  #include "sig.h"  #include "term.h"  #include <math.h>  #include <stdlib.h>  #include <string.h>  #include <sys/param.h>  #include <unistd.h>  #define BC\_X 15  #define BC\_START 8  #define BC\_STEP 8  #define X\_START 2  #define Y\_START 3  #define Y\_STEP 6  #define INPUTFIELD\_X 25  #define RESULTFIELD\_X 26  int g\_writeflags (char \*\*val);  int g\_drawborders (void);  int g\_drawaccumbox (void);  int g\_drawflagbox (void);  int g\_drawbcbox (void);  int g\_clearfields (void);  int g\_loadmemory (void);  int g\_savememory (void);  int g\_getunit (int address, char \*buff);  int g\_highlightmemory (int x, int y);  int g\_drawoperationbox (void);  int g\_drawcounterbox (void);  int g\_drawmemorybox (void);  int g\_drawboxes (void);  int g\_interface (void); |

# **РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ**

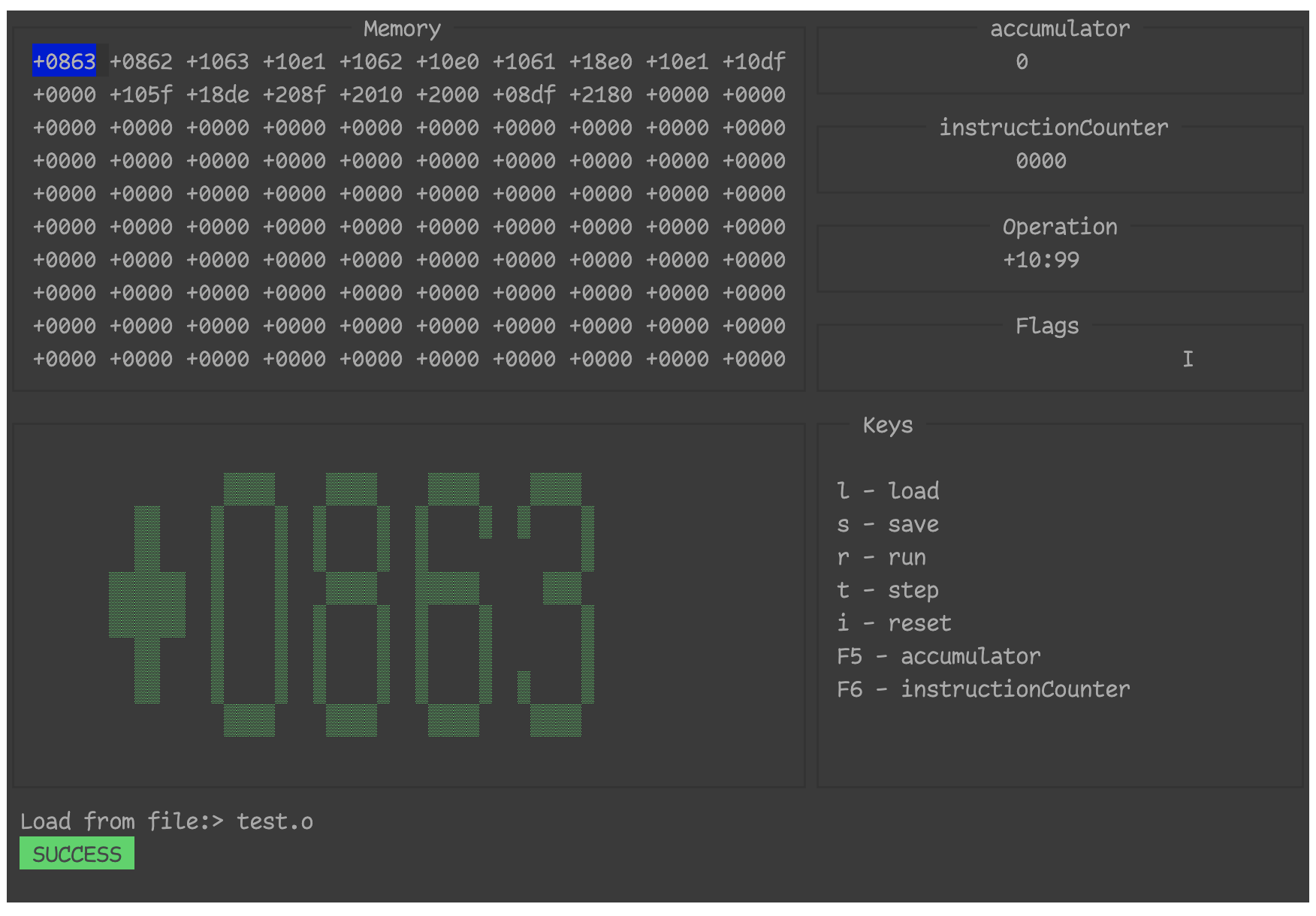
Трансляция тестовой программы на Simple Basic в программу на Simple Assembler:



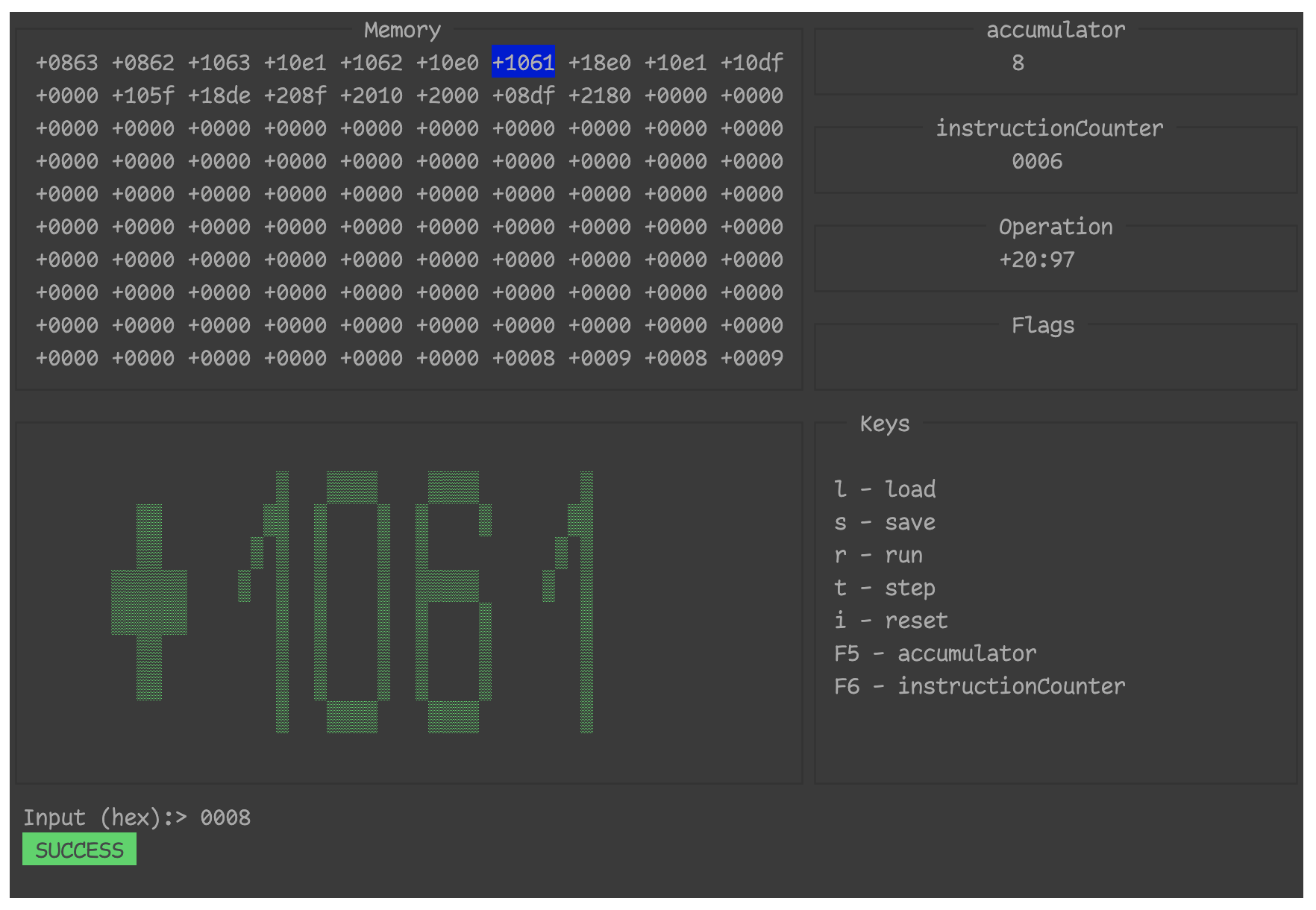
Трансляция программы на Simple Assembler в объектный файл:



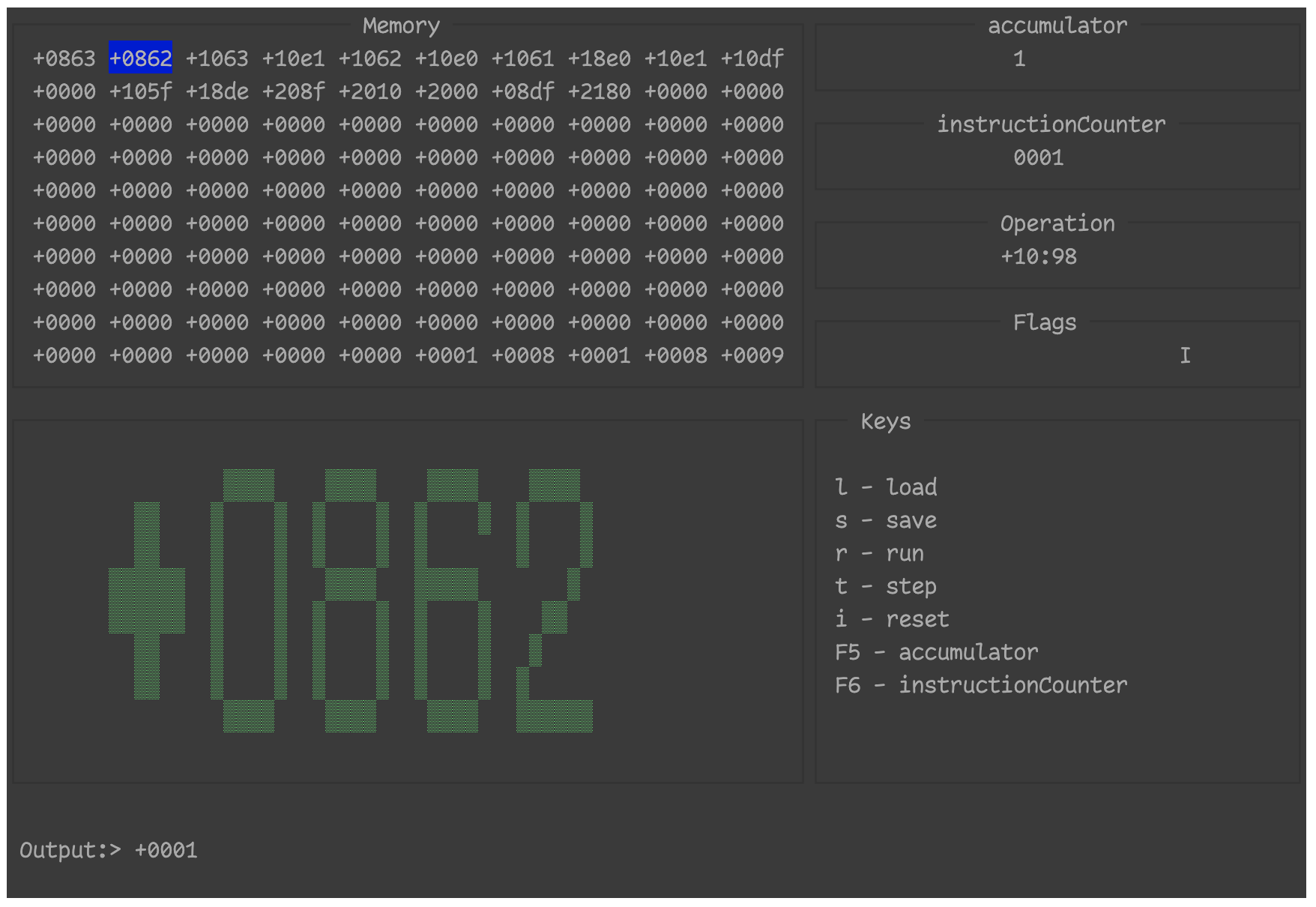
Загрузка и запуск тестовой программы в mySimpleComputer: вычисление разности двух чисел, вывод, если она больше нуля, иначе повторить ввод.



Ввод двух чисел: 9 и 8.



Вывод результата: 1.



# **ЗАКЛЮЧЕНИЕ**

В рамках выполнения курсовой работы мной была доработана модель *Simple Computer* так, чтобы она обрабатывала команды, записанные в оперативной памяти с помощью функций *ALU()* и *CU()*. Также были разработаны транслятор с языка программирования *Simple Assembler*, переводящий код с него в бинарный формат, и транслятор с высокоуровнего языка программирования *Simple Basic*, переводящий его в код на Simple Assembler.

**ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА**

1. Организация ЭВМ и систем. Практикум // С.Н. Мамойленко, Новосибирск: ГОУ ВПО «СибГУТИ», 2005 г.

2. Архитектура компьютера. 4-е изд. // Э. Танненбаум. – СПб.: Издательство «Питер», 2003.

3. Организация ЭВМ. 5-е изд. / К. Хамахер, З. Вранешич, С. Заки. – СПб.: Издательство «Питер», 2003.