Санкт-Петербургский политехнический университет Петра Великого

Кафедра компьютерных систем и программных технологий

**Отчёт по курсовой работе**

**Дисциплина**: Прикладное программирование

**Тема**: Симулятор RiSC-16

Выполнил студент гр. 23531/2 Н.А. Шинов

Преподаватель Т.Л. Сидорина

Проект защищен с оценкой

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_ 2018 г.

Санкт-Петербург

2018

Оглавление

1. **Техническое задание3**
2. **Алгоритм и структура программы5**
3. **Тесты6**
4. **Использованная литература11**
5. **Приложение11**
   1. Синтаксис комманд RISC-1611
   2. main.c, main.h11
   3. parser.c, parser.h …………………………………………………………………….12
   4. simulation.c, simulation.h14
   5. debug.c, debug.h20
   6. print.c, print.h23
6. Техническое задание

Целью курсового проекта является разработка симулятора RiSC-16 на языке Си. Реализация симулятора предполагает имитацию работы ЭВМ, возможность отладки машины (вывод содержимого регистров, восстановление работы машины).

Структура проекта:

* Входной и выходные файлы:
  + Путь входного файла указывается пользователем в начале работы программы. В данном файле ожидаются инструкции машины RiSC-16.
  + В выходном файле output.txt после выполнения программы находится содержимое регистров, полученное в результате работы.
* Файл с основной частью программы
* Файл, отвечающий за считывание инструкций из файла
* Файл, отвечающий за анализ инструкций
* Файл, отвечающий за функцию отладки
* Файл, отвечающий за печать содержимого регистров в процессе отладки или по окончании работы программы.

Задачи разработанного симулятора:

1. Имитация работы ЭВМ
2. Возможность отладки машины
3. Вывод в консоль информации об ошибках, выявленных во время работы. Печать содержимого регистров в заданный пользователем файл. Завершение работы программы

Программа должна собираться с помощью команды make компилятором gcc (mingw) с опциями –std=c11 –pedantic –Wall –Wextra без предупреждений компилятора.

Пример входного файла:

ADDI R1,R0,50

ADDI R2,r1,100

ADDI r1,r2,100

ADDI r1,r0,-50

ADD r3,r2,r1

NOP

NOP

ADDI R1,R0,12

ADDI R1,R0,25

SW r1,r2,52

LW r4,r2,52

Пример выходного файла output.txt:

registers:

reg[0]: 0

reg[1]: -50

reg[2]: 150

reg[3]: -60

reg[4]: 0

reg[5]: 0

reg[6]: 0

reg[7]: 0

Формат командной строки:

Risc.exe in.txt out.txt [debug]

При неверном формате командной строки в консоль выводится информация о программе и ее аргументах. Также выводится информация о примерном виде входного файла.

Ссылка на репозиторий github: https://github.com/shinov42/risc-16

1. Алгоритм и структура программы

Программа состоит за четырех файлов:

1. main.c (передача аргументов для дальнейшего исполнения)
2. parser.c (считывание информации из файла)
3. simulation.c (анализ команд, проверка правильности введенных данных)
4. debug.c (режим отладки)
5. print.c (вывод результатов)

Работа программы начинается с файла main.c. Здесь читаются аргументы командной строки и передаются для дальнейшей обработки в функцию get\_argc, которая находится в parser.c. Производится анализ аргументов – если их задано неверное количество – на экран выводится предупреждение и краткая справка, содержащая информацию о том, какие аргументы должны подаваться, а также примерный вид входного файла. Производится также дополнительная проверка на наличие ключа [debug]. Если вместо него указано что-либо другое – выводится предупреждение об этом и пример верного использования.

При отсутствии ключа [debug] начинается исполнение функции parse\_file. Производится построчное динамическое считывание команд из входного файла. Проводится проверка на соответствие алфавиту, разрешенному для использования (то есть исключаются различные символы, не воспринимаемые программой, а также любой алфавит кроме латинского). Если встречается символ #, то строка не обрабатывается дальше, а пропускается, так как это обозначение для комментария в машине RISC. После считывания данных строка передаётся длс дальнейшей обработки в функцию string\_procession. Данная строка проходит ряд проверок на соответствие стандарту (в том числе ошибки при вводе инструкций). В процессе проверки, строка разбивается на несколько меньших по размеру. Отдельно выделяется сама инструкция, первый, второй и третий аргумент. Аргументы проходят дополнительный контроль, т.к. количество операндов меняется от нуля до трёх, в связи с чем важно верно их считать и проверить на корректность. В случае нахождения ошибок на данном этапе в консоль выводится сообщение, содержащее номер ошибки, а также её краткое содержание. В случае правильной разбивки программа переходит к этапу выполнения инструкции. Для каждой команды заранее известно необходимое число операндов, в связи с чем проводится ещё одна проверка на соответствие количества (и содержимого) операндов заявленной функции. В случае успешного прохождения данного этапа – производится выполнение команды и дальнейший переход к следующей.

При наличии ключа [debug] происходит аналогичный процесс, однако с некоторыми отличиями. После выполнения команды не осуществляется переход к следующей, а ожидается ввод пользователя. Пользователь может ввести либо d, что означает переход к следующей команде (т.н. пошаговое выполнение), либо ввести e, что означает выход из режима отладки и выполнение программы с точки остановы до конца. После каждого шага в консоль выводится содержимое регистров в текущий момент. При запуске режима отладки показывается сообщение, оповещающее о переходе в данный режим, а также содержащее информацию о том, как пользоваться данным режимом. При вводе кода, отличного от указанных, выводится предупреждение

.

1. Тесты

Программа собирается с помощью команды make компилятором gcc (mingw) с опциями –std=c11 –pedantic –Wall –Wextra без предупреждений компилятора:

Рисунок 3.1. Сборка проекта

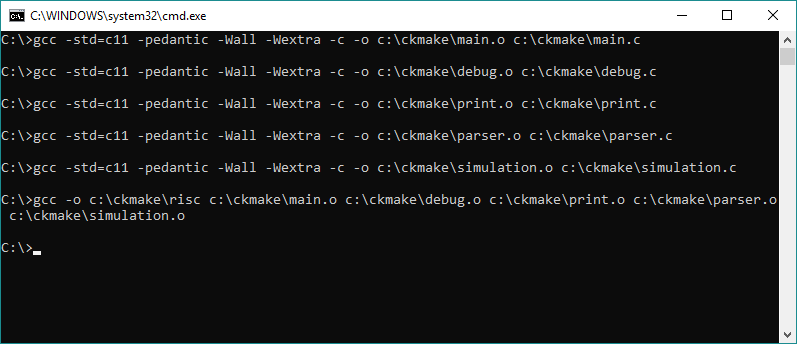
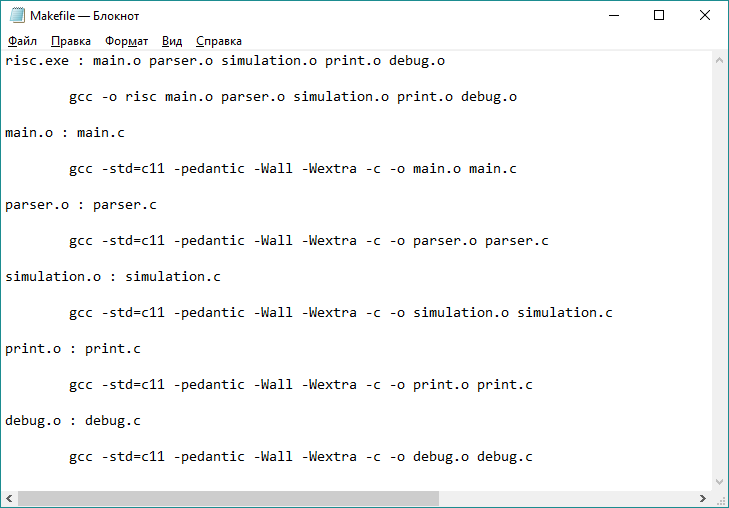
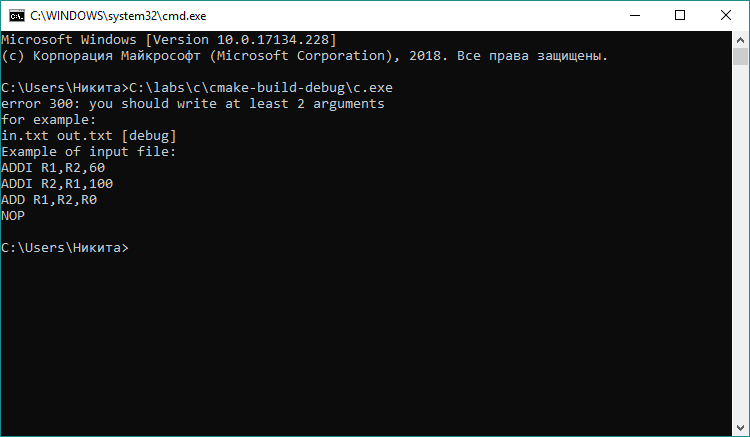


Рисунок 3.2. Makefile



Запуск с некорректными аргументами (не подавались имена файлов):

Рисунок 3.3. Некорректные аргументы



Пример работы программы в режиме отладки:

Рисунок 3.4.1 Входной файл для проверки режима отладки

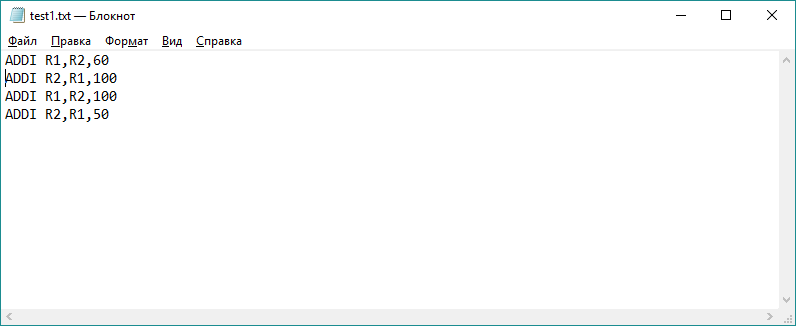


Рисунок 3.4.2 Работа программы в режиме отладки

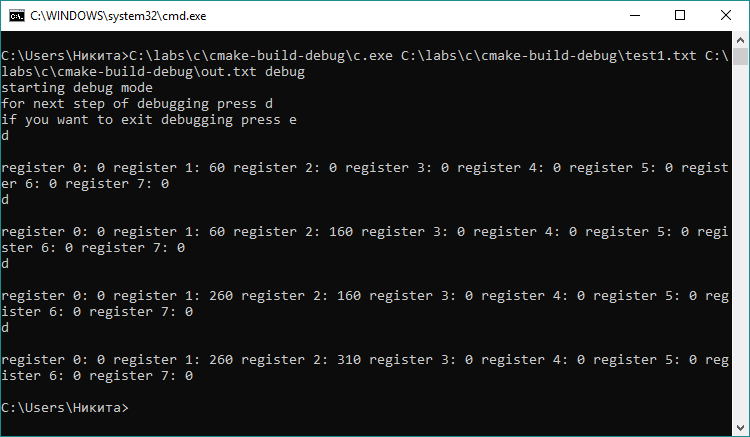
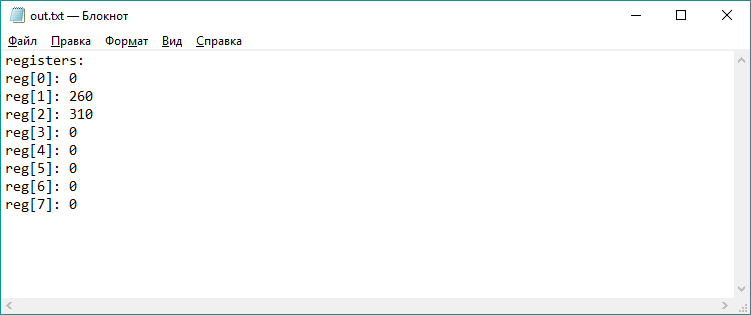


Рисунок 3.5. Пример выходного файла при успешном выполнении



Проведенные тесты:

1. Простые операции.

Входной файл:

ADDI R1,R2,60

ADDI R2,R1,100

ADDI R1,R2,100

ADDI R2,R1,50

Выходной файл:

registers:

reg[0]: 0

reg[1]: 260

reg[2]: 310

reg[3]: 0

reg[4]: 0

reg[5]: 0

reg[6]: 0

reg[7]: 0

1. Простые операции с отрицательными числами и работа с памятью.

Входной файл:

SW r1,r2,52

ADDI R1,R0,50

ADDI R2,r1,100

ADDI r1,r2,100

ADDI r1,r0,-50

ADD r3,r2,r1

#save to mem from r1

SW r1,r2,52

#load from mem to r4

LW r4,r2,52

Выходной файл:

registers:

reg[0]: 0

reg[1]: -50

reg[2]: 150

reg[3]: 100

reg[4]: -50

reg[5]: 0

reg[6]: 0

reg[7]: 0

1. Работа с основными и дополнительными командами.

Входной файл:

ADDI r1,r0,50

ADDI r2,r1,100

ADDI r1,r2,100

ADDI r1,r0,-50

ADD r3,r2,r1

SW r1,r2,52

LW r4,r2,52

BEQ r1,r4,8

ADDI r1,r0,39

ADDI r2,r0,39

ADDI r3,r0,39

ADDI r4,r0,39

ADDI r5,r0,39

ADDI r6,r0,39

ADDI r7,r0,39

ADDI r1,r0,-18

NOP

NOP

NOP

NOP

NOP

NOP

NOP

NOP

MOVI r1,1

MOVI r2,2

MOVI r3,3

MOVI r4,4

MOVI r5,5

MOVI r6,6

MOVI r7,7

Выходной файл:

registers:

reg[0]: 0

reg[1]: 1

reg[2]: 2

reg[3]: 3

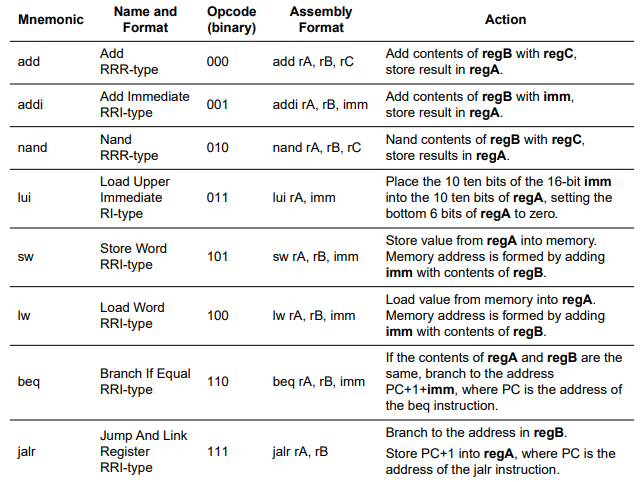
reg[4]: 4

reg[5]: 5

reg[6]: 6

reg[7]: 7

1. Использованная литература
2. B. Jacob «The RiSC-16 Instruction-Set Architecture»
3. Н. Вирт «Алгоритмы и структуры данных»
4. Б. Керниган, Д.Ритчи «Язык программирования Си»
5. Приложение
   1. Синтаксис команд RISC-16



* 1. main.c

#include **"main.h"  
  
int** main(**int** argc, **char** \*argv[])  
{  
 FILE \*in, \*out;  
 **int** reg[8] = {0};  
 **int** mem[1024] = {0};  
  
 *//if we have a wrong ammount of arguments - put an error on screen* **if** (argc < 3 || argc > 4) {  
 printf(**"error 300: you should write at least 2 arguments\nfor example:\nrisc.exe in.txt out.txt [debug]\n"**);  
 printf(**"Example of input file:\n"**);  
 printf(**"ADDI R1,R2,60\nADDI R2,R1,100\nADD R1,R2,R0\nNOP\n"**);  
 exit(300);  
 }  
 in = fopen(argv[1], **"r"**);  
  
 **if** (in == **NULL**) {  
 printf(**"error 200: file %s not found\n"**, argv[1]);  
 exit(200);  
 }  
  
 *//read input file and process it immediately  
 //and close it afterwards  
 //if no debug keyword* **if** (argc == 3) {  
 parse\_file(in, mem, reg, argv[1]);  
 } **else** *//if there is a debug keyword* **if** (argv[3] != **NULL** && strcmp(argv[3], **"debug"**) == 0) {  
 printf(**"starting debug mode\n"**);  
 debug\_file(in, mem, reg, argv[1]);  
 } **else** {  
 *//if not debug* printf(**"unknown command. \nPlease use [debug] for debugging or leave a blank space \n"**);  
 exit(102);  
 }  
  
 *//close input file* fclose(in);  
  
 *//create an output file* out = fopen(argv[2], **"w"**);  
  
 *//print the registers content in it* print\_result(out, reg);  
 fclose(out);  
  
}

main.h

#ifndef **C\_MAIN\_H**#define **C\_MAIN\_H**#include **"parser.h"**#endif *//C\_MAIN\_H*

* 1. parser.c

#include **"parser.h"  
  
void** parse\_file(FILE \*in, **int** \*mem, **int** \*reg, **char** \*filename) {  
  
 **char** \*alphabet = **"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"**;  
 **char** \*symbols = **" 0123456789#,-"**;  
 *//number of line we're at* **int** number = 0;  
  
 *//read file while it's not over* **while** (!feof(in)) {  
  
 *//buf for read symbols* **char** \*buf = calloc(32, **sizeof**(**char**));  
 **if** (buf == **NULL**){  
 printf(**"Error: can't get memory\n"**);  
 exit(1);  
 }  
 *//int to get any symbol from input* **int** read\_symb = 0;  
 **unsigned int** counter = 0;  
  
 **do** {  
 *//get one symbol from file* read\_symb = fgetc(in);  
 **char**\* abc = strchr(alphabet, (**char**) read\_symb);  
 **char**\* num = strchr(symbols, (**char**)read\_symb);  
 *//error if it's not a latin symbol, number or comma or any sign* **if** (read\_symb == 10 || read\_symb == **EOF**) *//10 == '\n' >122 - символы другого алфавита* **break**;  
 **else if** ((abc == **NULL** && num == **NULL**) || read\_symb > 122) {  
 printf(**"Error 102: unknown symbol at line %d\n"**, number);  
 printf(**"%c"**, read\_symb);  
 exit(102);  
 }  
  
 **if** (counter > **sizeof**(buf)) {  
 buf = (**char** \*) realloc(buf, **sizeof**(buf) + **sizeof**(**char**) \* 32);  
 **if** (buf == **NULL**){  
 printf(**"Error: can't get memory\n"**);  
 exit(1);  
 }  
  
 } **else if** (counter >= 100) { *//слишком длинная строка* printf(**"Error 103: command line %d is too long\n"**, number);  
 exit(103);  
 }  
  
 buf[counter++] = (**char**) read\_symb;  
 } **while** (1);  
  
 *//number of string* number++;  
  
 *//if comment* **if** (buf[0] == **'#'**)  
 **continue**;  
 *//if not NULL or comment* **else** *//need to process the string* string\_procession(buf, in, mem, reg, number, filename);  
 free(buf);  
 }  
}

parser.h

#ifndef **C\_PARSER\_H**#define **C\_PARSER\_H**#include **<stdio.h>**#include **"print.h"**#include **"debug.h"**#include **"simulation.h"**#include **<stdlib.h>  
  
int** parse\_file(FILE \*in, **int** mem[], **int** reg[], **char** \*filename);  
#endif *//C\_PARSER\_H*

* 1. simulation.c

#include **"simulation.h"  
  
void** string\_procession(**char** \*string, FILE \*in, **int** mem[], **int** reg[], **int** number, **char** \*filename) {  
 *//флаги чтобы понять что аргумент - номер регистра* **int** ra = 0, rb = 0, rc = 0;  
 *//служебная переменная* **int** null\_comp;  
 *//переменные для хранения чисел их аргументов* **int** arg\_one = 0, arg\_two = 0, arg\_three = 0;  
 *//массивы для проверки допустимых знаков* **char** \*numbers = {**"0123456789"**};  
 **char** \*letterssymb = {**"rR-"**};  
 *//массив для хранения команды* **char** comm[5] = {**'\0'**};  
 *//массив для хранения операндов* **char** \*operands;  
 *//получаем команду из строки  
 //находим пробел* **char** \*temp1 = strchr(string, **' '**);  
 *//ищем запятую (на будущее)* **char** \*commafind = strchr(string, **','**);  
 *//находим количества символов до пробела* size\_t count = strchr(string, **' '**) - string;  
 *//проверяем на отсутствие аргументов* **char**\* countnop = strchr(string, **' '**);  
 *//если нет аргументов - переходим к обработке* **if** (countnop == **NULL**){  
 strcpy(comm, string);  
 **goto** procession;  
 }  
 **else** *//если не смогли найти запятую* **if** (count > **sizeof**(string))  
 count = strchr(string, **'\n'**) - string;  
 *//if command is too long - exit with code 100 - unknown command* **if** (count > 5) {  
 printf(**"Error 100: unknown command at line %d\n"**, number);  
 exit(100);  
 }  
  
 *//если есть аргументы вообще* **if** (temp1 != **NULL**) {  
 *//put a command into 'comm'* strncpy(comm, string, count);  
  
 *//put operands into 'operands'* operands = string + count + 1;  
  
  
 *//operands: r1,r2,imm or r1,r2,r2 or r1,r2 or none  
 //находим где запятая (по идее после первого аргумента)* size\_t comma = strchr(operands, **','**) - operands;  
 *//массивы для операндов* **char** string\_operand\_one[16] = {**'\0'**};  
 **char** string\_operand\_two[16] = {**'\0'**};  
 **char** string\_operand\_three[16] = {**'\0'**};  
 *//берём первый операнд (от начала до запятой)* strncpy(string\_operand\_one, operands, comma);  
 *//проверка на то, нет ли там неразрешенных символов* **unsigned int** k = 0;  
 **for** (k = 0; k < **sizeof**(string\_operand\_one); k++) {  
 **char** \*checksymbols = strchr(letterssymb, string\_operand\_one[k]);  
 **char** \*checknumbers = strchr(numbers, string\_operand\_one[k]);  
 **if** (checksymbols == **NULL** && checknumbers == **NULL** && string\_operand\_one[k] != **'\0'**) {  
 printf(**"Error 300: wrong argument at line %d"**, number);  
 exit(300);  
 }  
 }  
  
 *//проверка на то что после R идёт номер регистра* **char** \*check\_number = strchr(numbers, string\_operand\_one[1]);  
 *//ищем следующую запятую* **char** \*find\_comma = strchr(string\_operand\_one, **','**);  
 *//если не нашли - ошибка* **if** (find\_comma != **NULL**) {  
 printf(**"Error 300: wrong argument at line %d"**, number);  
 exit(300);  
 }  
 *//* **if** (*/\*temp1 == NULL &&\*/* commafind == **NULL**)  
 **goto** procession;  
 *//проверка что первая буква обозначает регистр (числа здесь быть не должно)* **else if** ((string\_operand\_one[0] != **'R'** && string\_operand\_one[0] != **'r'**) || check\_number == **NULL**) {  
 printf(**"Error 301: wrong argument at line %d"**, number);  
 exit(301);  
 } **else** {  
 *//идем брать второй операнд  
 //get second operand string* operands = operands + comma + **sizeof**(**char**);  
 *//сразу ищем и следующую* **char** \*temp = strchr(operands, **','**);  
 **int** check\_comma = strcmp(operands, **","**);  
 **if** (check\_comma == 0) {  
 printf(**"Error 300: wrong ammount of arguments at line %d"**, number);  
 exit(300);  
 }  
 *//если есть второй и третий (по запятой определили)  
 //берём отдельно второй аргумент и проверяем на соответствие списку разрешённых символов* **if** (temp != **NULL** && operands != **NULL**) {  
 comma = strchr(operands, **','**) - operands;  
 strncpy(string\_operand\_two, operands, comma);  
 **for** (k = 0; k < **sizeof**(string\_operand\_two); k++) {  
 **char** \*checksymbols = strchr(letterssymb, string\_operand\_two[k]);  
 **char** \*checknumbers = strchr(numbers, string\_operand\_two[k]);  
 **if** (checksymbols == **NULL** && checknumbers == **NULL** && string\_operand\_two[k] != **'\0'**) {  
 printf(**"Error 300: wrong argument at line %d"**, number);  
 exit(300);  
 }  
 }  
 *//сравниваем, не пустая ли строка* null\_comp = strcmp(string\_operand\_two, **"\0"**);  
 **if** (null\_comp == 0) {  
 printf(**"Error 300: wrong ammount of arguments at line %d"**, number);  
 exit(300);  
 }  
  
 *//проверка на то что после второго операнда лишь запятая* **char** \*temp\_operands = operands + comma;  
 check\_comma = strcmp(temp\_operands, **","**);  
 **if** (check\_comma == 0) {  
 printf(**"Error 300: wrong ammount of arguments at line %d"**, number);  
 exit(300);  
 }  
 *//проверяем строку и если всё нормально - передаём в третий аргумент* operands = operands + comma + **sizeof**(**char**);  
 check\_comma = strcmp(operands, **","**);  
 **if** (check\_comma == 0) {  
 printf(**"Error 300: wrong ammount of arguments at line %d"**, number);  
 exit(300);  
 }  
 strncpy(string\_operand\_three, operands, comma + 1);  
 *//проверяем третий аргумент на соответствие* **for** (k = 0; k < **sizeof**(string\_operand\_three); k++) {  
 **char** \*checksymbols = strchr(letterssymb, string\_operand\_three[k]);  
 **char** \*checknumbers = strchr(numbers, string\_operand\_three[k]);  
 **if** (checksymbols == **NULL** && checknumbers == **NULL** && string\_operand\_three[k] != 0) {  
 printf(**"Error 300: wrong argument at line %d"**, number);  
 exit(300);  
 }  
 }  
  
 *//если осталось ещё что-то - ошибка* operands = operands + **sizeof**(string\_operand\_three);  
 **if** (operands[0] != **'\0'**) {  
 printf(**"Error 300: wrong ammount of arguments at line %d"**, number);  
 exit(300);  
 }  
  
 }  
 *//выполняется если есть только второй аргумент (без третьего)* **else** {  
 strcpy(string\_operand\_two, operands);  
 **for** (k = 0; k < **sizeof**(string\_operand\_two); k++) {  
 **char** \*checksymbols = strchr(letterssymb, string\_operand\_two[k]);  
 **char** \*checknumbers = strchr(numbers, string\_operand\_two[k]);  
 **if** (checksymbols == **NULL** && checknumbers == **NULL** && string\_operand\_two[k] != **'\0'**) {  
 printf(**"Error 300: wrong argument at line %d"**, number);  
 exit(300);  
 }  
 }  
 **if** (string\_operand\_two[0] == **'\0'**) {  
 printf(**"Error 300: wrong ammount of arguments at line %d"**, number);  
 exit(300);  
 }  
 }  
 }  
  
 **int** b;  
 **int** ret = sscanf(string\_operand\_one + **sizeof**(**char**), **"%d"**, &b);  
 **if** (ret != 1)  
 {  
 printf(**"Error: wrong argument \n"**);  
 exit(1);  
 }  
 **else** arg\_one = ret;  
 *//переводим строки в числа и проверяем на номер если регистр* ra = 1;  
 **if** (arg\_one > 7 || arg\_one < 0) {  
 printf(**"error 105: can't reach specified register\n"**);  
 exit(105);  
 }  
 check\_number = strchr(numbers, string\_operand\_two[1]);  
 **if** ((string\_operand\_two[0] == **'r'** || string\_operand\_two[0] == **'R'**) && check\_number != **NULL**) {  
 rb = 1;  
 ret = sscanf(string\_operand\_two + **sizeof**(**char**), **"%d"**, &b);  
 **if** (ret != 1)  
 {  
 printf(**"Error: wrong argument \n"**);  
 exit(1);  
 }  
 **else** arg\_two = ret;  
 **if** (arg\_two > 7 || arg\_two < 0) {  
 printf(**"error 105: can't reach specified register\n"**);  
 exit(105);  
 }  
 } **else** arg\_two = atoi(string\_operand\_two);  
  
 check\_number = strchr(numbers, string\_operand\_three[1]);  
 **if** ((string\_operand\_three[0] == **'r'** || string\_operand\_three[0] == **'R'**) && check\_number != **NULL**) {  
 rc = 1;  
 ret = sscanf(string\_operand\_three + **sizeof**(**char**), **"%d"**, &b);  
 **if** (ret != 1)  
 {  
 printf(**"Error: wrong argument \n"**);  
 exit(1);  
 }  
 **else** arg\_three = ret;  
 **if** (arg\_three > 7 || arg\_three < 0) {  
 printf(**"error 105: can't reach specified register\n"**);  
 exit(105);  
 }  
 } **else** {  
 ret = sscanf(string\_operand\_three + **sizeof**(**char**), **"%d"**, &b);  
 **if** (ret != 1)  
 {  
 printf(**"Error: wrong argument \n"**);  
 exit(1);  
 }  
 **else** arg\_three = ret;  
 arg\_three = atoi(string\_operand\_three);  
 }  
 } **else** strcpy(comm, string);  
  
 *//basic commands* procession:  
 *//reg1 = reg2 + reg3* **if** (strcmp(comm, **"ADD"**) == 0) {  
 *//operands should contain r1,r2,r3* **if** (ra == 1 && rb == 1 && rc == 1)  
 reg[arg\_one] = reg[arg\_two] + reg[arg\_three];  
 **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//reg1 = reg2 + imm* **else if** (strcmp(comm, **"ADDI"**) == 0) {  
 *//operands should contain r1,r2,imm* **if** (ra == 1 && rb == 1 && rc == 0)  
 reg[arg\_one] = reg[arg\_two] + arg\_three;  
 **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//reg1 = nand(reg2, reg3)* **else if** (strcmp(comm, **"NAND"**) == 0) {  
 *//operands should contain r1,r2,r3* **if** (ra == 1 && rb == 1 && rc == 1)  
 reg[arg\_one] = !(reg[arg\_two] & reg[arg\_three]);  
 **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//top 10 bits of imm into 16-bit regA, rest are zeroes* **else if** (strcmp(comm, **"LUI"**) == 0) {  
 *//operands should contain r1,imm* **if** (ra == 1 && rb == 0 && rc == 0)  
 reg[arg\_one] = (**int**) (arg\_two & 01111111111000000);*//первый 0 -двоичное число* **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//store value from r1 to memory formed by imm+reg2* **else if** (strcmp(comm, **"SW"**) == 0) {  
 *//operands should contain r1,r2,imm  
 //save reg into memory* **if** (arg\_three + reg[arg\_two] > 1024) {  
 printf(**"error 400: memory out of bounds\n"**);  
 exit(400);  
 }  
 **if** (ra == 1 && rb == 1 && rc == 0)  
 mem[reg[arg\_two] + arg\_three] = reg[arg\_one];  
 **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//load value from memory to reg1, mem is formed by imm+reg2* **else if** (strcmp(comm, **"LW"**) == 0) {  
 *//operands should contain r1,r2,imm* **if** (arg\_three + reg[arg\_two] > 1024) {  
 printf(**"error 400: memory out of bounds\n"**);  
 exit(400);  
 }  
 **if** (ra == 1 && rb == 1 && rc == 0)  
 reg[arg\_one] = mem[reg[arg\_two] + arg\_three];  
 **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//branch if equal => if r1==r2, branch to PC+1+imm, where PC is address of beq* **else if** (strcmp(comm, **"BEQ"**) == 0) {  
 *//operands should contain r1,r2,imm* **if** (ra == 1 && rb == 1 && rc == 0 && reg[arg\_one] == reg[arg\_two])  
 skip\_strings(arg\_three + 1, in);*//skip imm+1 lines* **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//jump and link to address from regB. Store PC+1 into regA* **else if** (strcmp(comm, **"JALR"**) == 0) {  
 **if** (ra == 1 && rb == 1 && rc == 0) {  
 **if** (reg[arg\_two] > number)  
 skip\_strings(reg[arg\_two] - number, in);*//skip imm+1 lines* **else if** (reg[arg\_two] < number && reg[arg\_two] > 0) {  
 reg[arg\_one] = number + 1;  
 back\_jump(reg[arg\_two], in, filename);  
 }  
 } **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//additional commands* **else if** (strcmp(comm, **"NOP"**) == 0) {  
 **if** (ra == 0 && rb == 0 && rc == 0)  
 reg[0] = 0;  
 **else** {  
 printf(**"Error 301: wrong argument(s)\n"**);  
 exit(301);  
 }  
 }  
  
 *//operands contain nothing* **else if** (strcmp(comm, **"MOVI"**) == 0) {  
 *//operands should contain r1,imm* **if** (ra == 1 && rb == 0 && rc == 0)  
 reg[arg\_one] = arg\_two;  
 **else** {  
 printf(**"error 300: wrong argument(s)\n"**);  
 exit(300);  
 }  
 }  
  
 *//if we didn't recognise command* **else** {  
 printf(**"error 100: unknown command %s"**, comm);  
 exit(100);  
 }  
}  
  
**void** skip\_strings(**int** n, FILE \*in) {  
 **int** count = n;  
  
 *//while file's not over* **while** (!feof(in) && count != 0) {  
 **int** buf = 1;  
 **while** (buf != **'\n'**)  
 buf = fgetc(in);  
 *//counter minus one* count--;  
 }  
}  
  
**void** back\_jump(**int** count, FILE \*in, **char** \*filename) {  
 freopen(filename, **"r"**, in);  
 **int** buf = 1;  
  
 *//while file's not over* **while** (!feof(in) && count != 0) {  
 **while** (buf != **'\n'**)  
 buf = fgetc(in);  
 *//counter minus one* count--;  
 }  
}

simulation.h

#ifndef **C\_SIMULATION\_H**#define **C\_SIMULATION\_H**#include **<stdio.h>**#include **<string.h>**#include **<stdlib.h>  
  
void** string\_procession(**char** \*string, FILE \*in, **int** \*mem, **int** \*reg, **int** number, **char** \*filename);  
  
**void** back\_jump(**int** count, FILE \*in, **char** \*filename);  
  
**void** skip\_strings(**int** n, FILE \*in);  
  
#endif *//C\_SIMULATION\_H*

* 1. debug.c

#include **"debug.h"  
  
void** debug\_file(FILE \*in, **int** \*mem, **int** \*reg, **char** \*filename) {  
 *//number of line we're at* **int** number = 0;  
 *//whole alphabet to check if symbols are correct* **char** \*alphabet = **"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"**;  
 **char** \*symbols = **" 0123456789#,-"**;  
 *//get a symbol from console* **char** code = **'\0'**;  
 restart:  
 printf(**"for next step of debugging press d\n"**);  
 printf(**"if you want to exit debugging press e\n"**);  
 **if** (!feof(in)) {  
 start:  
 scanf(**"%c"**, &code);  
 *//while file's not over* **if** (code == **'d'**) {  
 **if** (!feof(in)) {  
  
 *//buf for read symbols* **char** \*buf = calloc(32, **sizeof**(**char**));  
 *//int to get any symbol from input* **int** read\_symb = 0;  
  
*//number of symbol we read* **unsigned int** counter = 0;  
  
 **do** {  
 *//get one symbol from file* read\_symb = fgetc(in);  
 **char** \*abc = strchr(alphabet, (**char**) read\_symb);  
 **char** \*num = strchr(symbols, (**char**) read\_symb);  
 *//error if it's not a latin symbol, number or comma or any sign* **if** (read\_symb == 10 || read\_symb == **EOF**)  
 **break**;  
 **else if** ((abc == **NULL** && num == **NULL**) || read\_symb > 122) {  
 printf(**"Error 102: unknown symbol at line %d\n"**, number);  
 printf(**"%c"**, read\_symb);  
 exit(102);  
 }  
  
 **if** (counter > **sizeof**(buf)) {  
 buf = (**char** \*) realloc(buf, **sizeof**(buf) + **sizeof**(**char**) \* 32);  
 } **else if** (counter >= 100) {  
 printf(**"Error 103: command line %d is too long\n"**, number);  
 exit(103);  
 }  
  
 buf[counter++] = (**char**) read\_symb;  
 } **while** (1);  
  
 *//number of string* number++;  
  
 *//if comment* **if** (buf[0] == **'#'**);  
 *//if not NULL or comment* **else** *//need to process the string* string\_procession(buf, in, mem, reg, number, filename);  
  
 free(buf);  
 }  
 print\_reg\_debug(reg);  
 } **else if** (code == **'e'**) {  
 **while** (!feof(in)) {  
  
 *//buf for read symbols* **char** \*buf = calloc(32, **sizeof**(**char**));  
 *//int to get any symbol from input* **int** read\_symb = 0;  
 *//number of symbol we read* **unsigned int** counter = 0;  
  
 **do** {  
 *//get one symbol from file* read\_symb = fgetc(in);  
 **char** \*abc = strchr(alphabet, (**char**) read\_symb);  
 **char** \*num = strchr(symbols, (**char**) read\_symb);  
 *//error if it's not a latin symbol, number or comma or any sign* **if** (read\_symb == 10 || read\_symb == **EOF**)  
 **break**;  
 **else if** ((abc == **NULL** && num == **NULL**) || read\_symb > 122) {  
 printf(**"Error 102: unknown symbol at line %d\n"**, number);  
 printf(**"%c"**, read\_symb);  
 exit(102);  
 }  
  
 **if** (counter > **sizeof**(buf)) {  
 buf = (**char** \*) realloc(buf, **sizeof**(buf) + **sizeof**(**char**) \* 32);  
 } **else if** (counter >= 100) {  
 printf(**"Error 103: command line %d is too long\n"**, number);  
 exit(103);  
 }  
  
 buf[counter++] = (**char**) read\_symb;  
 } **while** (1);  
  
 *//number of string* number++;  
  
 *//if comment* **if** (buf[0] == **'#'**)  
 **continue**;  
 *//if not NULL or comment* **else** *//need to process the string* string\_procession(buf, in, mem, reg, number, filename);  
  
 free(buf);  
 }  
 } **else if** (code != **'\n'**) {  
 printf(**"unknown command\n"**);  
 **goto** restart;  
 }  
 **if** (!feof(in))  
 **goto** start;  
  
 }  
}

debug.h

#ifndef **C\_DEBUG\_H**#define **C\_DEBUG\_H**#include **<stdio.h>**#include **<stdlib.h>**#include **<string.h>**#include **"simulation.h"**#include **"print.h"  
  
void** debug\_file(FILE \*in, **int** \*mem, **int** \*reg, **char** \*filename);  
  
#endif *//C\_DEBUG\_H*

* 1. print.c

#include **"print.h"**  
  
**void** print\_result(FILE \*out, **int** \*reg) {  
 fprintf(out, **"registers: \n"**);  
 **for** (**int** i = 0; i < 8; i++)  
 fprintf(out, **"reg[%d]: %d\n"**, i, reg[i]);  
}

print.h

#ifndef **C\_PRINT\_H**#define **C\_PRINT\_H**#include **<stdio.h>  
  
void** print\_result(FILE \*out, **int** \*reg);  
  
#endif *//C\_PRINT\_H*