Санкт-Петербургский политехнический университет Петра Великого
Кафедра компьютерных систем и программных технологий

Отчёт по курсовой работе

Дисциплина: Прикладное программирование

Тема: Симулятор RiSC-16

Выполнил студентгр. 23531/2 _____ А.А.Гаврилов

Преподаватель _____

Проект защищен с оценкой

“___”_____ 2018 г.

Санкт-Петербург

2018

Оглавление

1. 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 выходномфайле output.txt после выполнения программы находится содержимое регистров, и памяти, полученные в результате работы.
- Файл с основной частью программы
- Файл, отвечающий за выполнение программы
- Файл, отвечающий за считывание инструкций из файла и их анализ
- Файл, отвечающий за запись инструкций в машинном коде в память ЭВМ
- Файл, отвечающий за выполнение инструкций
- Файл, описывающий структуру инструкции
- Файл, отвечающий за функцию отладки
- Файл, отвечающий за обработку ошибок

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 input1.txt:

```
ADDI 1,0,50
ADDI 2,1,60
ADDI 1,2,60
ADDI 1,0,-50
ADD 3,2,1
ADDI 1,0,12
ADDI 1,0,25
SW 1,3,-40
LW 4,3,-40
JALR 0,0
```

Пример выходного файла output1.txt:

```
Registers:
register[0]=  0000000000000000
register[1]=  0000000000011001
register[2]=  0000000001101110
register[3]=  0000000000111100
register[4]=  0000000000011001
register[5]=  0000000000000000
register[6]=  0000000000000000
register[7]=  0000000000000000

Memory:
memory[0]=  0000000000000000
memory[1]=  0010010000110010
memory[2]=  0010100010111100
memory[3]=  0010010100111100
memory[4]=  0010010001001110
memory[5]=  0000110100000001
memory[6]=  0010010000001100
memory[7]=  0010010000011001
memory[8]=  1000010111011000
memory[9]=  1011000111011000
memory[10]=  1110000000000000
memory[11]=  0000000000000000
```

Формат командойстроки:

```
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/Kento0k/RISC

2. Алгоритм и структура программы

Программа состоит из следующих файлов:

1. main.c (передача аргументов для дальнейшего исполнения)
2. execute.c (выполнение программы)
3. file_parser.c (считывание и обработка файла)
4. write_to_memory.c (запись программы в память ЭВМ)
5. run.c (выполнение отдельных инструкций)
6. debug.c (режим отладки)
7. instruction.c (структура инструкции)

Работа программы начинается с файла main.c. Здесь читаются аргументы командной строки и передаются для дальнейшей обработки в функцию exec_program, которая находится в execute.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 exec_program. На первом этапе данная функция инициализирует массив команд. Происходит это следующим образом. В функцию parse_file (file_parcer.c) подаются элементы структуры instruction: name (название команды),*args (массив аргументов команды). Затем данная функция считывает строку из входного файла, проверяет ее на корректность, игнорирует комментарии и пустые строки, и, в случае корректности строки команды, инициализирует поданные аргументы своими значениями. Проверки включают в себя: проверка на соответствие допустимому алфавиту, проверка количества аргументов. Затем инструкции присваевается адрес в памяти ЭВМ и она подается на вход функции memory_write (write_to_memory.c), которая осуществляет запись иструкции в машинном коде в соответствующую ячейку памяти ЭВМ. Далее функция exec_program подает команду, регистры, память и адрес команды в функцию run_instruction (run.c). В ней производится проверка формата аргументов и выполнение инструкции. В конце выполнения программы содержимое регистров и памяти выводится в файл. Содержимое памяти выводится частично, до адреса самого дальнего обращения к памяти включительно, так как содержимое ячеек памяти, ни разу не использовавшихся в данной программе, не представляет интереса в данной ситуации.

Отладка производится с помощью функции debug_program. Она почти полностью повторяет функцию exec_program, за исключением того, что выполняет программу по одной инструкции, выводя содержимое регистров и памяти на экран, или всю сразу, в зависимости от введенной команды.

Регистры и память ЭВМ в данной реализации являются двумерными массивами. Каждая строка этих массивов состоит из 16 элементов, принимающих значение 0 или 1. Адресация в памяти и обращение к регистрам осуществляется по номеру строки двумерного массива (первому индексу), а считывание битов из ячейки памяти или регистра- по номеру элемента в строке (номеру столбца). Запись в нулевой регистр и нулевой адрес памяти не осуществляется, в соответствии с архитектурой RISC-16. Работа машины выглядит следующим образом: с помощью счетчика команд осуществляется проход по памяти, пока не будут достигнуты либо конец адресного пространства памяти, либо точка останова. Точкой останова является инструкция JALR 0,0, в соответствии с архитектурой RISC-16.

3. Тесты

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

Рисунок 3.1. Сборка проекта

```
C:\Users\asass\CLionProjects\RISC>make
gcc -std=c11 -pedantic -Wall -Wextra -c -o main.o main.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o errors.o errors.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o file_parser.o file_parser.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o instruction.o instruction.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o run.o run.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o execute.o execute.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o debug.o debug.c
gcc -std=c11 -pedantic -Wall -Wextra -c -o write_to_memory.o write_to_memory.c
gcc -o RISC main.o errors.o file_parser.o instruction.o run.o execute.o debug.o write_to_memory.o
```

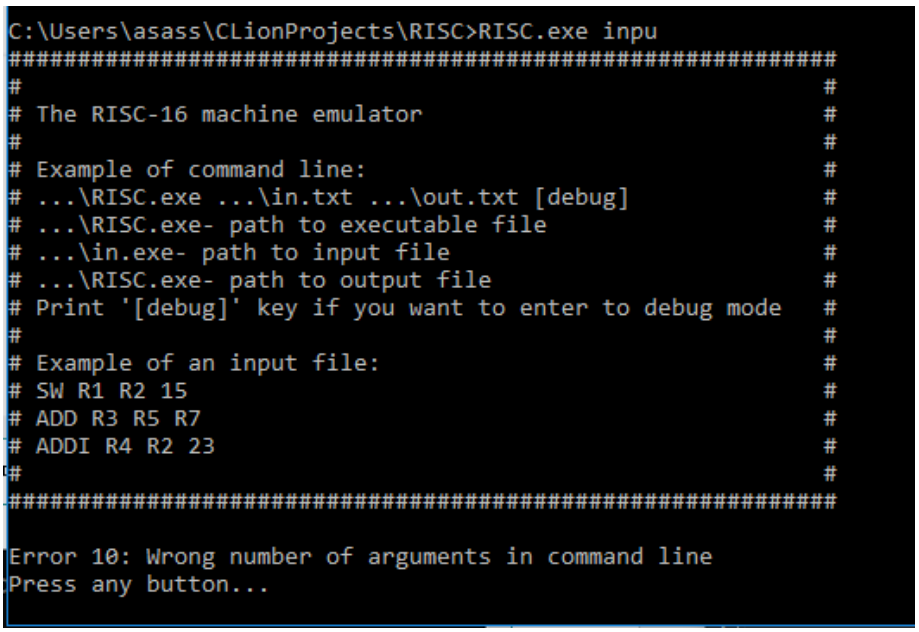
Рисунок 3.2. Makefile

```
.PHONY: all clean

all: RISC.exe

RISC.exe : main.o errors.o file_parser.o instruction.o run.o execute.o debug.o write_to_memory.o
	gcc -o RISC main.o errors.o file_parser.o instruction.o run.o execute.o debug.o write_to_memory.o

main.o : main.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o main.o main.c

errors.o : errors.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o errors.o errors.c

file_parser.o : file_parser.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o file_parser.o file_parser.c

instruction.o : instruction.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o instruction.o instruction.c

run.o : run.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o run.o run.c

debug.o : debug.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o debug.o debug.c

execute.o : execute.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o execute.o execute.c

write_to_memory.o : write_to_memory.c
	gcc -std=c11 -pedantic -Wall -Wextra -c -o write_to_memory.o write_to_memory.c
clean :
	del *.o
```

Команда make clean удаляет все .o файлы из директории проекта.

Запуск с некорректными аргументами:

Рисунок 3.3. Некорректные аргументы

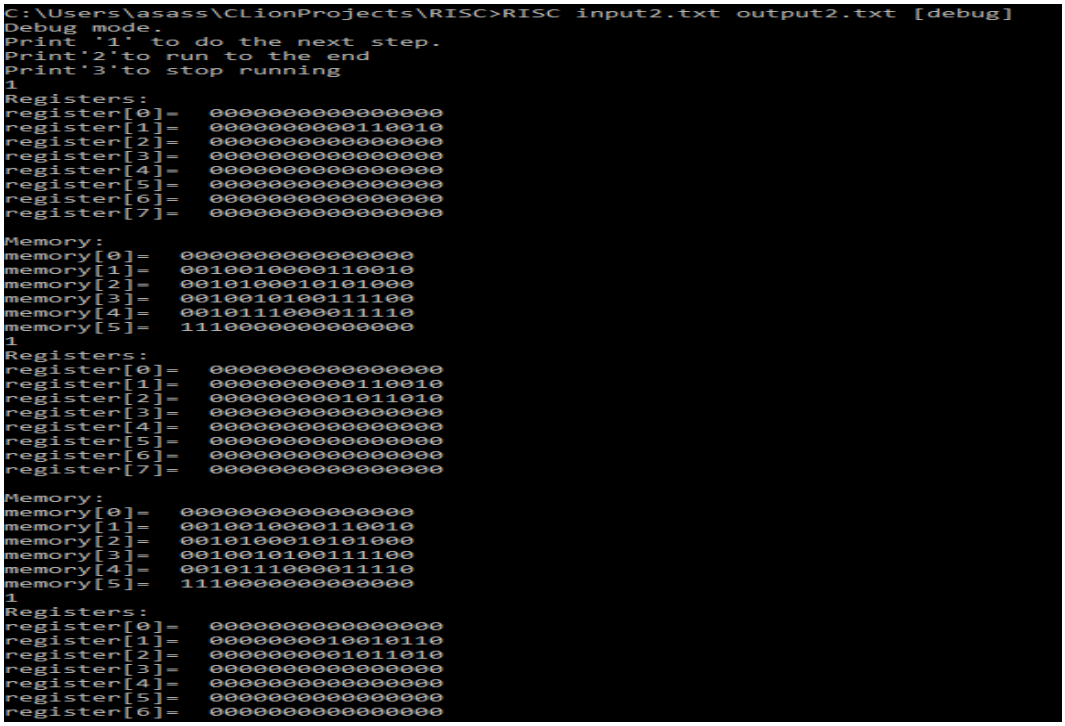


Пример работы программы в режиме отладки.

Входной файл input2.txt для проверки режима отладки :

```
ADDI 1,0,50
ADDI 2,1,40
ADDI 1,2,60
ADDI 3,4,30
JALR 0,0
```

Рисунок 3.4. Работа программы в режиме отладки



Выходной файл output2.txt при успешном выполнении:

```
Registers:
register[0]=  0000000000000000
register[1]=  0000000010010110
register[2]=  0000000001011010
register[3]=  0000000000011110
register[4]=  0000000000000000
register[5]=  0000000000000000
register[6]=  0000000000000000
register[7]=  0000000000000000

Memory:
memory[0]=  0000000000000000
memory[1]=  0010010000110010
memory[2]=  0010100010101000
memory[3]=  0010010100111100
memory[4]=  0010111000011110
memory[5]=  1110000000000000
```

Проведенные тесты.

Входной файл input.txt:

```
#Добавление чисел
ADDI 1,2,15
ADDI 1,1,19
ADD 2,1,1
ADDI 3,2,-18
ADDI 4,3,-53
#Операция NAND
NAND 5,3,4
#Операция LUI
LUI 3,102
#Добавление чисел
ADDI 6,7,-64
ADDI 7,6,63
ADDI 7,7,2
#Запись числа(инструкции) в память
SW 3,7,16
#Загрузка числа из памяти
LW 1,7,16
ADDI 7,7,19
#Загрузка точки останова по адресу 20
LUI 1,896
SW 1,7,0
#Прыжок к точке останова
JALR 0,7
```

Выходной файл output.txt:

```
Registers:
register[0]=  0000000000000000
register[1]=  1110000000000000
register[2]=  0000000001000100
register[3]=  0001100110000000
register[4]=  1111111111111101
register[5]=  1111111111001111
register[6]=  1111111111000000
register[7]=  0000000000010100

Memory:
memory[0]=  0000000000000000
memory[1]=  0010010100001111
memory[2]=  0010010010010011
memory[3]=  0000100010000001
memory[4]=  0010110101101110
memory[5]=  0011000111001011
memory[6]=  0101010110000100
memory[7]=  0110110001100110
memory[8]=  0011101111000000
memory[9]=  0011111100111111
memory[10]=  0011111110000010
memory[11]=  1000111110010000
memory[12]=  1010011110010000
memory[13]=  0011111110010011
memory[14]=  0110011110000000
memory[15]=  1000011110000000
memory[16]=  1110001110000000
memory[17]=  0001100110000000
memory[18]=  0000000000000000
memory[19]=  0000000000000000
memory[20]=  1110000000000000
```

4. Использованная литература

1. B. Jacob «The RiSC-16 Instruction-Set Architecture»
2. Н.Вирт «Алгоритмы и структуры данных»
3. Б. Керниган, Д.Ритчи «Язык программирования Си»

5. Приложение

5.1. Синтаксис команд RISC-16

FORMATS:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RRR-type:	3 bits: opcode			3 bits: reg A			3 bits: reg B			4 bits: 0				3 bits: reg C		
RRI-type:	3 bits: opcode			3 bits: reg A			3 bits: reg B			7 bits: signed immediate (-64 to 63)						
RI-type:	3 bits: opcode			3 bits: reg A			10 bits: immediate (0 to 0x3FF)									

INSTRUCTIONS:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD:	3 bits: 000			3 bits: reg A			3 bits: reg B			4 bits: 0				3 bits: reg C		
ADDI:	3 bits: 001			3 bits: reg A			3 bits: reg B			7 bits: signed immediate (-64 to 63)						
NAND:	3 bits: 010			3 bits: reg A			3 bits: reg B			4 bits: 0				3 bits: reg C		
LUI:	3 bits: 011			3 bits: reg A			10 bits: immediate (0 to 0x3FF)									
SW:	3 bits: 100			3 bits: reg A			3 bits: reg B			7 bits: signed immediate (-64 to 63)						
LW:	3 bits: 101			3 bits: reg A			3 bits: reg B			7 bits: signed immediate (-64 to 63)						
BEQ:	3 bits: 110			3 bits: reg A			3 bits: reg B			7 bits: signed immediate (-64 to 63)						
JALR:	3 bits: 111			3 bits: reg A			3 bits: reg B			7 bits: 0						
Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mnemonic	Name and Format	Opcode (binary)	Assembly Format	Action
add	Add RRR-type	000	add rA, rB, rC	Add contents of **regB** with **regC**, store result in **regA**.
addi	Add Immediate RRI-type	001	addi rA, rB, imm	Add contents of **regB** with **imm**, store result in **regA**.
nand	Nand RRR-type	010	nand rA, rB, rC	Nand contents of **regB** with **regC**, store results in **regA**.
lui	Load Upper Immediate RI-type	011	lui rA, imm	Place the 10 ten bits of the 16-bit **imm** into the 10 ten bits of **regA**, setting the bottom 6 bits of **regA** to zero.
sw	Store Word RRI-type	101	sw rA, rB, imm	Store value from **regA** into memory. Memory address is formed by adding **imm** with contents of **regB**.
lw	Load Word RRI-type	100	lw rA, rB, imm	Load value from memory into **regA**. Memory address is formed by adding **imm** with contents of **regB**.
beq	Branch If Equal RRI-type	110	beq rA, rB, imm	If the contents of **regA** and **regB** are the same, branch to the address PC+1+**imm**, where PC is the address of the beq instruction.
jalr	Jump And Link Register RRI-type	111	jalr rA, rB	Branch to the address in **regB**. Store PC+1 into **regA**, where PC is the address of the jalr instruction.