Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 1

по курсу «Архитектура ЭВМ»

на тему: «Изучение принципов работы микропроцессорного ядра RISC-V.»

Вариант № 14

Студент ИУ7-53Б _____ Лысцев Н. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Ибрагимов С. В.
(Подпись, дата) (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

Цель работы

Основной целью работы является ознакомление с принципами функционирования, построения и особенностями архитектуры суперскалярных конвейерных микропроцессоров. Дополнительной целью работы является знакомство с принципами проектирования и верификации сложных цифровых устройств с использованием языка описания аппаратуры SystemVerilog и ПЛИС.

1 Основные теоретические сведения

RISC-V является открытым современным набором команд, который может использоваться для построения как микроконтроллеров, так и высокопроизводительных микропроцессоров. Таким образом, термин RISC-V фактически является названием для семейства различных систем команд, которые строятся вокруг базового набора команд, путем внесения в него различных расширений.

В данной работе исследуется набор команд RV32I, который включает в себя основные команды 32-битной целочисленной арифметики кроме умножения и деления.

1.1 Модель памяти

Архитектура RV32I предполагает плоское линейное 32-х битное адресное пространство. Минимальной адресуемой единицей информации является 1 байт. Используется порядок байтов от младшего к старшему (Little Endian), то есть, младший байт 32-х битного слова находится по младшему адресу (по смещению 0). Отсутствует разделение на адресные пространства команд, данных и ввода-вывода. Распределение областей памяти между различными устройствами (ОЗУ, ПЗУ, устройства ввода-вывода) определяется реализацией.

1.2 Система команд

Большая часть команд RV32I является трехадресными, выполняющими операции над двумя заданными явно операндами, и сохраняющими результат в регистре. Операндами могут являться регистры или константы, явно заданные в коде команды. Операнды всех команд задаются явно.

Архитектура RV32I, как и большая часть RISC-архитектур, предполагает разделение команд на команды доступа к памяти (чтение данных из памяти в регистр или запись данных из регистра в память) и команды обработки данных в регистрах.

2 Практическая часть

2.1 Задание №1

Листинг 2.1 – Программа для примера

```
    .section .text (1)
    .globl _start; (2)
    len = 8 #Размер массива (3)
    enroll = 4 #Количество обрабатываемых элементов за одну
       итерацию
    elem_sz = 4 #Размер одного элемента массива
_start: (4)
    addi x20, x0, len/enroll (5)
    la x1, _x (6)
loop:
    lw x2, 0(x1) (7)
    add x31, x31, x2 (8)
    lw x2, 4(x1)
    add x31, x31, x2
    lw x2, 8(x1)
    add x31, x31, x2
    lw x2, 12(x1)
    add x31, x31, x2
    addi x1, x1, elem_sz*enroll (9)
    addi x20, x20, -1 (10)
    bne x20, x0, loop (11)
    addi x31, x31, 1
forever: j forever (12)

    .section .data (13)
_x: .4byte 0x1 (14)
    .4byte 0x2
    .4byte 0x3
    .4byte 0x4
    .4byte 0x5
    .4byte 0x6
    .4byte 0x7
    .4byte 0x8
```

1) Объявление секции *.text*, содержащей исполняемый код.

2) Объявление символа *_start*, имеющего глобальную видимость. Символ *_start* это специальный символ, обозначающий точку входа в программу.

3) Метка.

4) Объявление констант.

5) Арифметические выражения над константами могут использоваться в командах на месте непосредственного операнда.

6) Загрузка в $x1$ адреса символа *_x* (то есть, начала массива).

7) Загрузка в $x2$ числа по адресу, содержащемуся в $x1$ по смещению 0.

8) Добавление к $x31$ (который хранит результат) значения $x2$.

9) Смещение указателя $x1$.

10) Уменьшение счетчика цикла.

11) Условный переход на метку *loop*.

12) Бесконечный цикл.

13) Объявление секции данных.

14) Начало описания массива.

Программа, представленная на листинге 2.1 выполняет суммирование значений элементов массива слов и увеличивает это значение на 1. Примером данной небольшой программы для RV32I мы будем пользоваться далее для исследования процесса выполнения команд.

Дизассемблированный код представлен на листинге 2.2.

Листинг 2.2 – Дизассемблированный код программы для примера

```
80000000 <_start>:
80000000:   00200a13            addi    x20,x0,2
80000004:   00000097            auipc   x1,0x0
80000008:   03c08093            addi    x1,x1,60 # 80000040 <_x>

8000000c <loop>:
8000000c:   0000a103            lw  x2,0(x1)
80000010:   002f8fb3            add x31,x31,x2
80000014:   0040a103            lw  x2,4(x1)
80000018:   002f8fb3            add x31,x31,x2
8000001c:   0080a103            lw  x2,8(x1)
80000020:   002f8fb3            add x31,x31,x2
80000024:   00c0a103            lw  x2,12(x1)
80000028:   002f8fb3            add x31,x31,x2
8000002c:   01008093            addi    x1,x1,16
80000030:   fffa0a13            addi    x20,x20,-1
80000034:   fc0a1ce3            bne x20,x0,8000000c <loop>
80000038:   001f8f93            addi    x31,x31,1

8000003c <forever>:
8000003c:   0000006f            jal x0,8000003c <forever>
```

Можно сказать, что данная программа эквивалентна следующему псевдокоду на языке C, представленному на листинге 2.3.

Листинг 2.3 – Псевдокод программы для примера на языке C

```
#define len 8
#define enroll 4
#define elem_sz 4
int _x[]={1,2,3,4,5,6,7,8};
void _start() {
    int x20 = len/enroll;
    int *x1 = _x;

    do {
      int x2 = x1[0];
      x31 += x2;
      x2 = x1[1];
      x31 += x2;
      x2 = x1[2];
      x31 += x2;
      x2 = x1[3];
      x31 += x2;
      x1 += enroll;
      x20--;
    } while(x20 != 0);
    x31++;
    while(1){}
}
```

Условие задания

В процессе выполнения задания необходимо выполнить следующие действия:

1) Ознакомиться с теоретической частью, внимательно изучить примеры.

2) Перейти в подкаталог src командой cd riscv-lab/src.

3) Выполнить сборку, запустив команду make. Убедиться, что был создан файл test.hex, содержащий шестнадцатеричное представление программы, а в окне терминала отобразился дизассемблерный листинг. Сравнить дизассемблерный листинг с тем, который приведен в примере.

4) Создать новый файл, содержащий текст программы по индивидуальному варианту. Поместить его в каталог src. Текст программы сохранить в файле с расширением .s.

5) Изучить текст программы по индивидуальному варианту. Поместить в отчете псевдокод, соответствующий данной программе.

6) Анализируя исходный текст программы, ответьте на вопрос: какое значение должно содержаться в регистре x31 в конце выполнения программы?

7) Изменить в Makefile строку SRC= так, чтобы ее содержимое соответствовало имени файла с текстом программы без расширения .s.

8) Выполнить компиляцию командой make. В процессе будет создан файл с расширением .hex, хранящий содержимое памяти команд и данных, а в окне терминала отобразится дизассемблерный листинг, который необходимо поместить в отчет вместе с исходным текстом.

Результаты выполнения

При переходе в подкаталог $riscv-lab/src$ и запуске сборки командой $make$, был получен файл $test.hex$ и в окне терминала отобразился следующий листинг:

```
^^/b/s/a/l/lab_01 >>> cd riscv-lab/src/
^^/b/s/a/l/l/r/src >>> make
riscv64-linux-gnu-as --march=rv32i test.s -o test.o
riscv64-linux-gnu-ld -b elf32-littleriscv -T link.ld test.o -o test.elf
riscv64-linux-gnu-objdump -D -M numeric,no-aliases -t test.elf

test.elf:     формат файла elf32-littleriscv

SYMBOL TABLE:
80000000 l    d  .text  00000000 .text
80000040 l    d  .data  00000000 .data
00000000 l    df *ABS*  00000000 test.o
00000008 l       *ABS*  00000000 len
00000004 l       *ABS*  00000000 enroll
00000004 l       *ABS*  00000000 elem_sz
80000040 l       .data  00000000 _x
8000000c l       .text  00000000 loop
8000003c l       .text  00000000 forever
80000000 g       .text  00000000 _start
80000060 g       .data  00000000 _end



Дизассемблирование раздела .text:

80000000 <_start>:
80000000:       00200a13                addi    x20,x0,2
80000004:       00000097                auipc   x1,0x0
80000008:       03c08093                addi    x1,x1,60 # 80000040 <_x>

8000000c <loop>:
8000000c:       0000a103                lw      x2,0(x1)
80000010:       002f8fb3                add     x31,x31,x2
80000014:       0040a103                lw      x2,4(x1)
80000018:       002f8fb3                add     x31,x31,x2
8000001c:       0080a103                lw      x2,8(x1)
80000020:       002f8fb3                add     x31,x31,x2
80000024:       00c0a103                lw      x2,12(x1)
80000028:       002f8fb3                add     x31,x31,x2
8000002c:       01008093                addi    x1,x1,16
80000030:       fffa0a13                addi    x20,x20,-1
80000034:       fc0a1ce3                bne     x20,x0,8000000c <loop>
80000038:       001f8f93                addi    x31,x31,1

8000003c <forever>:
8000003c:       0000006f                jal     x0,8000003c <forever>

Дизассемблирование раздела .data:

80000040 <_x>:
80000040:       0001                    c.addi  x0,0
80000042:       0000                    c.unimp
80000044:       0002                    c.slli64        x0
80000046:       0000                    c.unimp
80000048:       00000003                lb      x0,0(x0) # 0 <elem_sz-0x4>
8000004c:       0004                    .2byte  0x4
8000004e:       0000                    c.unimp
80000050:       0005                    c.addi  x0,1
80000052:       0000                    c.unimp
80000054:       0006                    c.slli  x0,0x1
80000056:       0000                    c.unimp
80000058:       00000007                .4byte  0x7
8000005c:       0008                    .2byte  0x8
        ...
riscv64-linux-gnu-objcopy -O binary --reverse-bytes=4 test.elf test.bin
xxd -g 4 -c 4 -p test.bin test.hex
rm test.bin test.o test.elf
^^/b/s/a/l/l/r/src >>> ls
link.ld  Makefile  my14var.s  test.hex  test.s
```

Рисунок 2.1 – Выполнение команды $make$ к программе для примера

Полученный дизасемблерный листинг в точности повторяет тот, что был приведен в примере (листинг 2.2)

Был создан файл, содержащий текст программы по индивидуальному варианту, с названием $my14var.s$ и помещен в каталог src.

Код программы для 14-го варианта представлен на листинге 2.4.

Листинг 2.4 – Код программы для 14-го варианта

```
    .section .text
    .globl _start;
    len = 9 #Размер массива
    enroll = 1 #Количество обрабатываемых элементов за одну
       итерацию
    elem_sz = 4 #Размер одного элемента массива

_start:
    la x1, _x
    addi x20, x1, elem_sz*len #Адрес элемента, следующего за
       последним
    lw x31, 0(x1)
    add x1, x1, elem_sz*1
lp:
    lw x2, 0(x1)
    bltu x2, x31, lt
    add x31, x0, x2 #!
lt:
    add x1, x1, elem_sz*enroll
    bne x1, x20, lp
lp2: j lp2

    .section .data
_x: .4byte 0x1
    .4byte 0x2
    .4byte 0x3
    .4byte 0x4
    .4byte 0x8
    .4byte 0x6
    .4byte 0x7
    .4byte 0x5
    .4byte 0x4
```

Псевдокод на языке C, соответствующий данной программе представлен на листинге 2.5.

Листинг 2.5 – Псевдокод программы для 14-го варианта на языке C

```
#include <stdio.h>

#define len 9
#define enroll 1
#define elem_sz 4

int _x[] = {1, 2, 3, 4, 8, 6, 7, 5, 4};

int main(void) {
  int *x1 = _x;
  d x1, x1, elem_sz *enroll int *x20 = x1 + len + 1;
  int x31 = x1[0];

  x1 += 1;

  do {
    int x2 = x1[0];
    if (x2 >= x31) {
      x31 = x2;
    }
    x1 += enroll;
  } while (x1 != x20);

  printf("x31 = %d\n", x31);

  while (1) {
  }

  return 0;
}
```

В ходе анализа исходного текста программы, соответствующего 14-му варианту, было установлено, что ее задачей является поиск максимального элемента в массиве. Следовательно, в регистре $x31$ в конце работы программы будет лежать значение максимального элемента массива (в данном случае – 8).

После присвоения в Makefile строки SRC имени файла программы без расширения, соответствующее 14-му варианту, была выполнена команда *make* (рисунок 2.2):

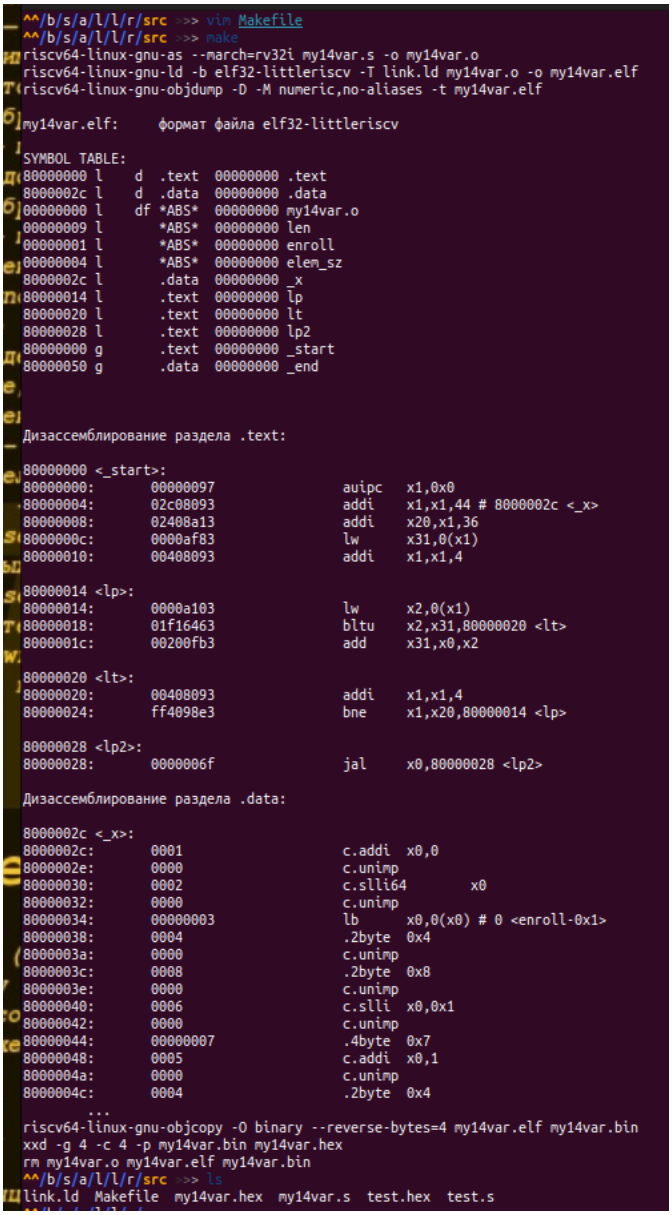

Рисунок 2.2 – Выполнение команды *make* к программе 14-го варианта

Дизассемблированный код программы 14-го варианта представлен на листинге 2.6

Листинг 2.6 – Дизассемблированный код программы для 14-го варианта

```
Дизассемблирование раздела .text:

80000000 <_start>:
80000000:   00000097            auipc   x1,0x0
80000004:   02c08093            addi    x1,x1,44 # 8000002c <_x>
80000008:   02408a13            addi    x20,x1,36
8000000c:   0000af83            lw  x31,0(x1)
80000010:   00408093            addi    x1,x1,4

80000014 <lp>:
80000014:   0000a103            lw  x2,0(x1)
80000018:   01f16463            bltu    x2,x31,80000020 <lt>
8000001c:   00200fb3            add x31,x0,x2

80000020 <lt>:
80000020:   00408093            addi    x1,x1,4
80000024:   ff4098e3            bne x1,x20,80000014 <lp>

80000028 <lp2>:
80000028:   0000006f            jal x0,80000028 <lp2>
```

2.2 Задание №2

Условие задания

В ходе выполнения данного задания необходимо выполнить следующие действия:

1) Запустить симуляцию в среде Modelsim. Для этого найти в каталоге taiga файл run.sh (если лабораторная работа выполняется в среде ОС Linux) или run.bat (для ОС Windows) и запустить его двойным щелчком мыши.

2) Запустить симуляцию, набрав в командной строке Modelsim команду run 460us.

3) Изучить список сигналов, приведенных в окне Wave.

4) В соответствии с таблицей, приведенной ниже, получить снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с указанным адресом. Для команд, входящих в тело цикла, приведен номер итерации.

   Мой вариант: команда с адресом 80000014, 2-я итерация.

Результаты выполнения

После выполнения пунктов 1-2 запускается симуляция в среде Modelsim, как видно на рисунке 2.3.

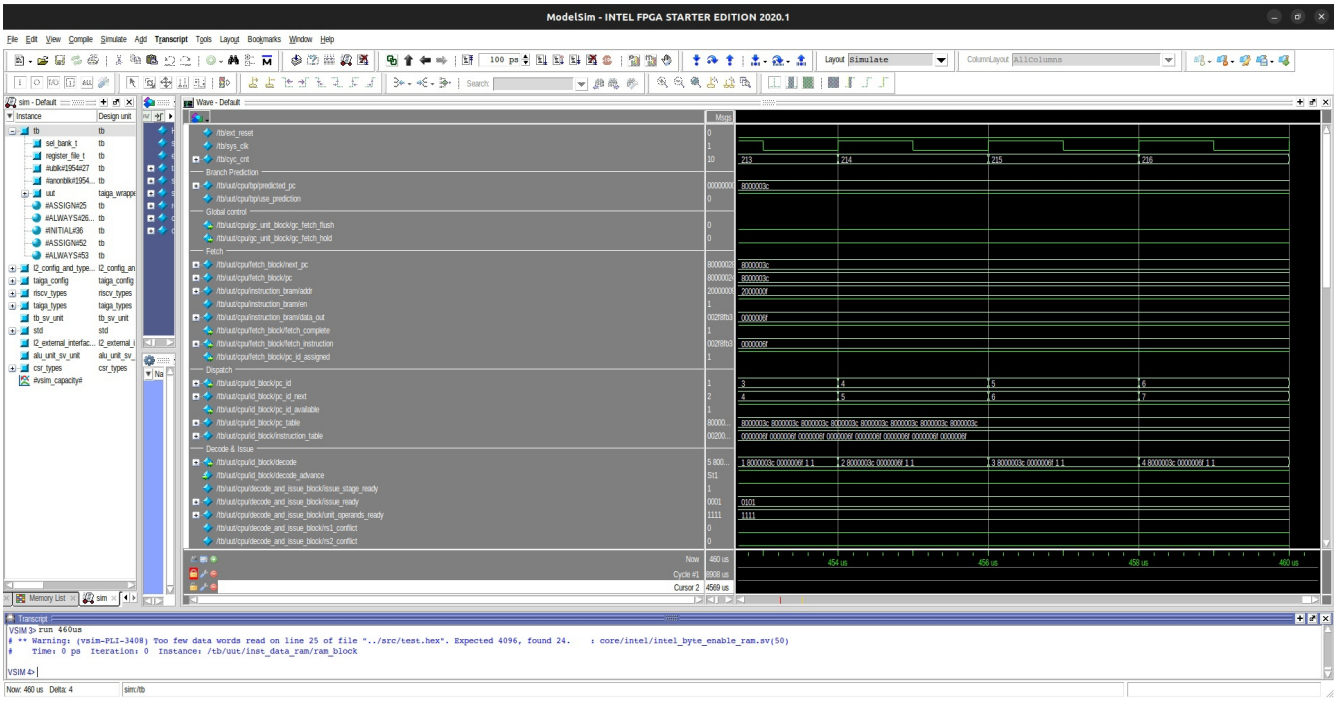

Рисунок 2.3 – Скриншот запуска симуляции в среде Modelsim

На рисунке 2.4 показан снимок экрана симуляции в среде Modelsim на стадии выборки (29-й такт) и диспетчеризации (30-й такт) команды с адресом 80000014 на 2-й итерации.

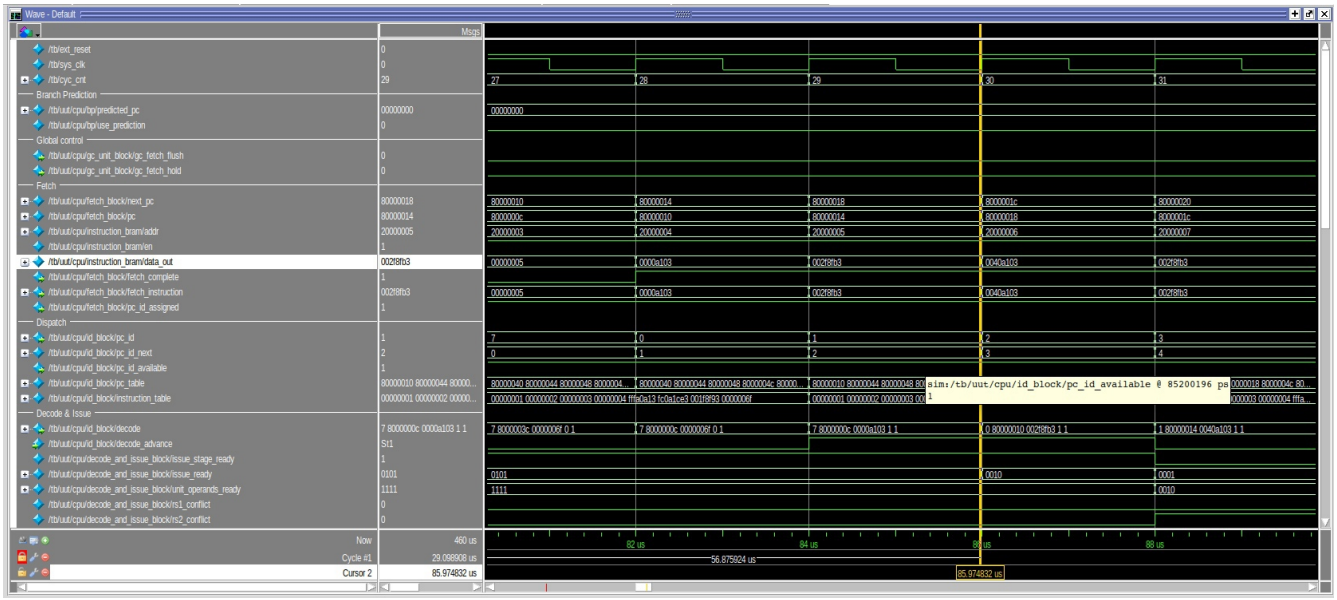

Рисунок 2.4 – Стадии выборки и диспетчеризаци команды с адресом 80000014 на 2-й итерации.

2.3 Задание №3

Условие задания

Получить снимок экрана, содержащий временную диаграмму выполнения стадии декодирования и планирования на выполнение команды с указанным адресом. Для команд, входящих в тело цикла, приведен номер итерации.

Мой вариант: команда с адресом 80000020, 2-я итерация.

Результаты выполнения

На рисунке 2.5 показан снимок экрана симуляции в среде Modelsim на стадии декодирования (38-й такт) команды с адресом 80000020 на 2-й итерации.

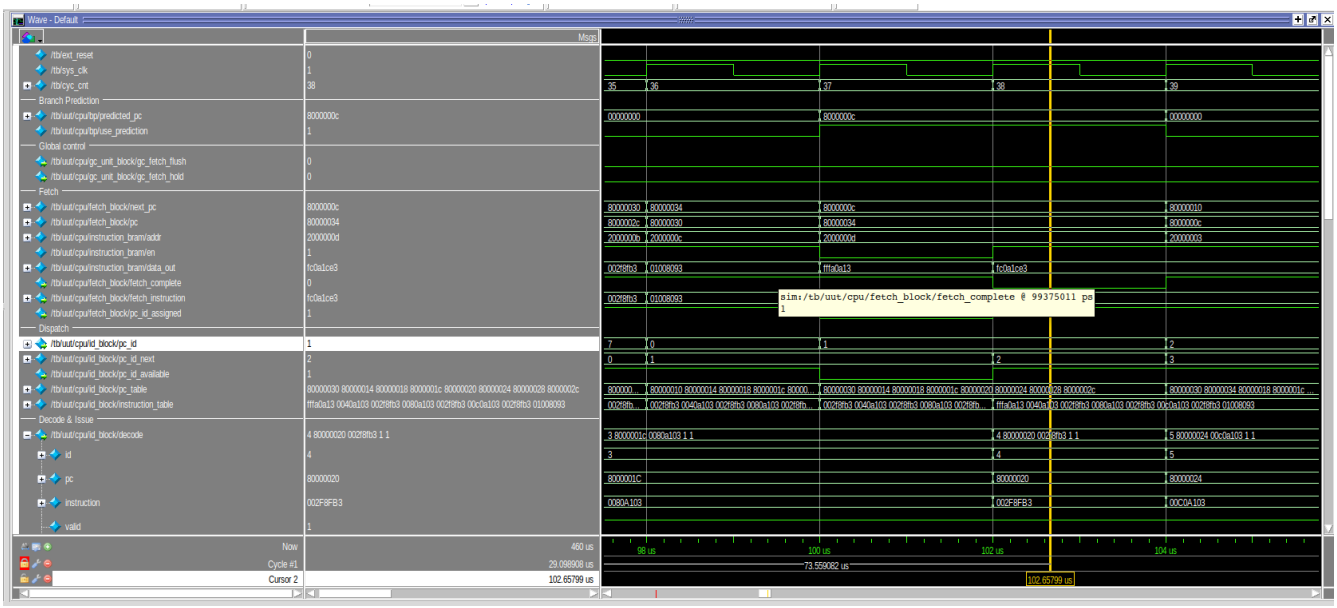

Рисунок 2.5 – Стадии декодирования и планирования на выполнение команды с адресом 80000020 на 2-й итерации.

2.4 Задание №4

Условие задания

Получить снимок экрана, содержащий временную диаграмму выполнения стадии выполнения команды с указанным адресом. Для команд, входящих в тело цикла, приведен номер итерации.

Мой вариант: команда с адресом 8000000c, 2-я итерация.

Результаты выполнения

На рисунке 2.6 показан снимок экрана симуляции в среде Modelsim на стадии выполнения (30-й, 31-й и 32-й такты) команды с адресом 8000000c на 2-й итерации.

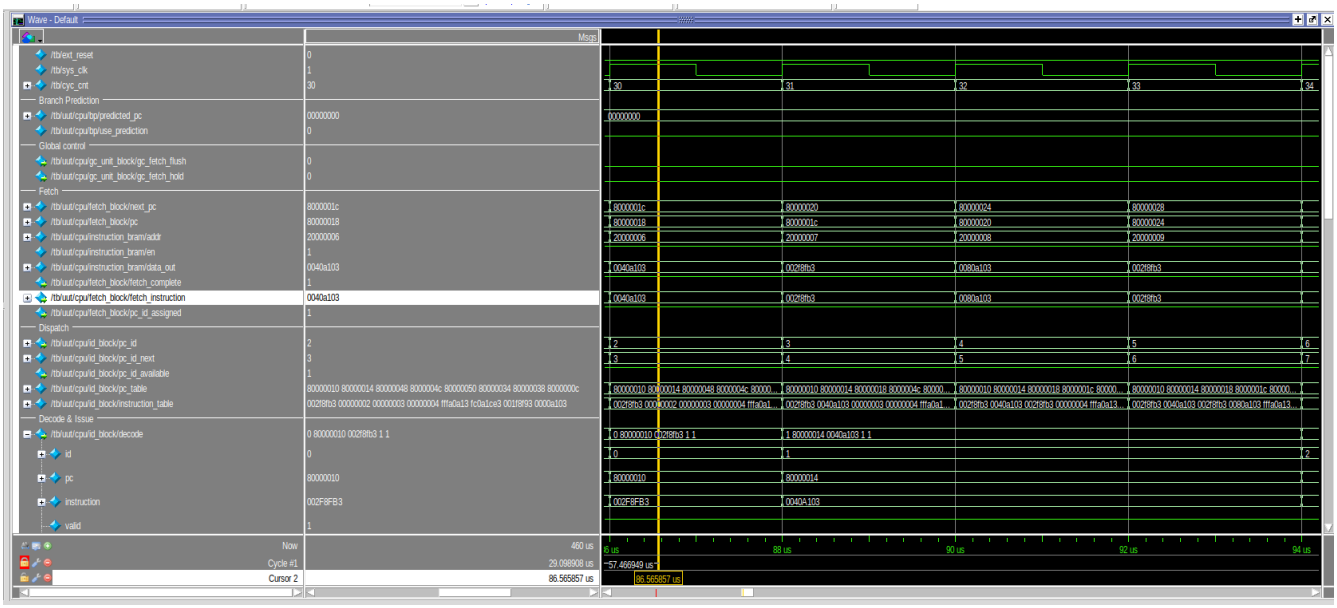

Рисунок 2.6 – Стадии декодирования и планирования на выполнение команды с адресом 8000000c на 2-й итерации.

2.5 Задание №5

Условие задания

В процесссе выполнения этого задания необходимо выполнить следующие действия:

1) Исправить файл taiga/run.sh или taiga/run.bat так, чтобы там был указан путь к файлу .hex, соответствующему программе по индивидуальному варианту. Сохранить файл.

2) Закрыть Modelsim.

3) Запустить симуляция заново.

4) Получить временную диаграмму сигналов выполнения программы индивидуального варианта.

5) Сравнить значение регистра $x31$ (сигнал $/tb/register\_file[31]$) на момент окончания выполнения программы с тем, который был получен в Задании №1.

6) Получить снимок экрана, содержащий временные диаграммы сигналов, соответствующих всем стадиям выполнения команды, обозначенной в тексте программы символом #!.

7) Анализируя диаграмму заполнить трассу выполнения программы. Рекомендуется использовать для этого файл pipeline.ods, содержащий трассу тестового примера.

8) Сделать вывод об эффективности выполнения программы и о путях оптимизации.

9) Провести оптимизацию программы путем перестановки команд для устранения конфликтов.

10) Перекомпилировать программу и перезапустить симуляцию.

11) Заполнить трассу выполнения оптимизированной программы.

12) Сравнить трассы выполнения неоптимизированной и оптимизированной версии, сделать выводы.

Результаты выполнения

Узнаем значение, хранящееся в регистре $x31$ по окончании выполнения программы. На рисунке 2.7 видно, что вычисленное в первом задании значение совпадает с хранящимся там.

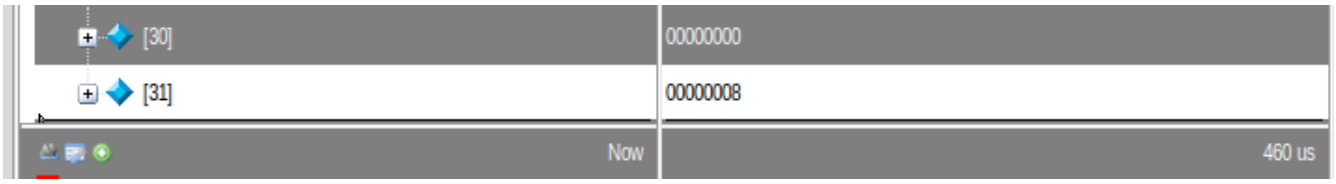

Рисунок 2.7 – Значение регистра $x31$ после выполнения программы

Символом #! помечена команда $add\ \ x31, x0, x2$ с адресом $8000001c$. На рисунке 2.8 представлены стадии выборки (8-й такт) и диспетчеризации (9-й такт), а на рисунке 2.9 показаны стадии декодирования и планирования на выполнение (12-й такт) и выполнения (13-й такт).

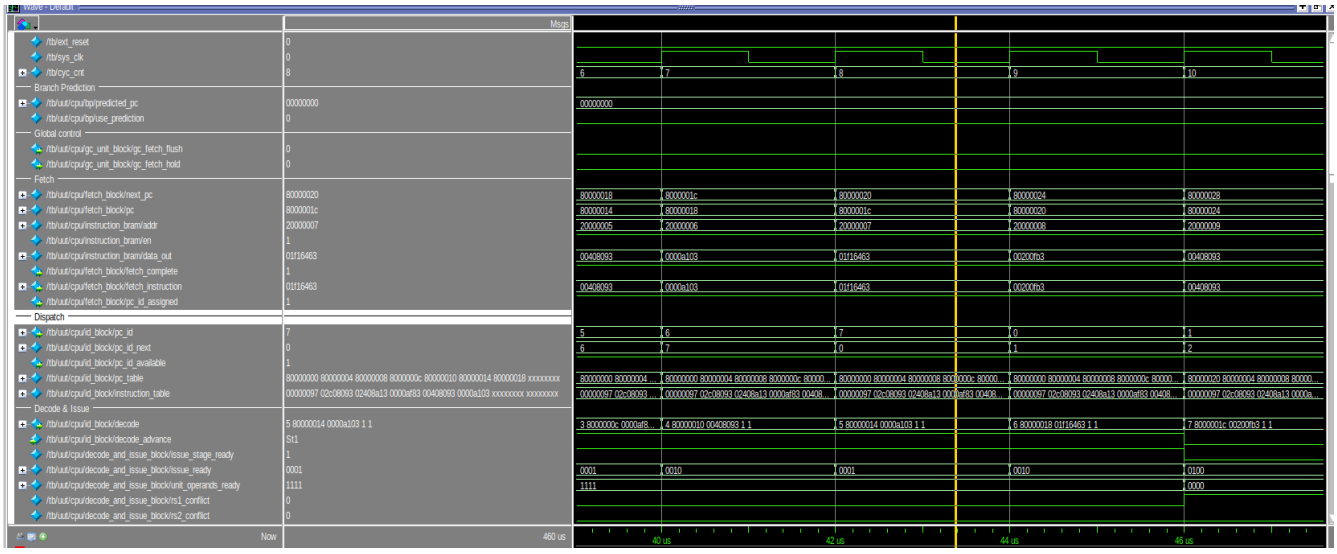

Рисунок 2.8 – Стадии выборки и диспетчеризации команды с адресом $8000001c$

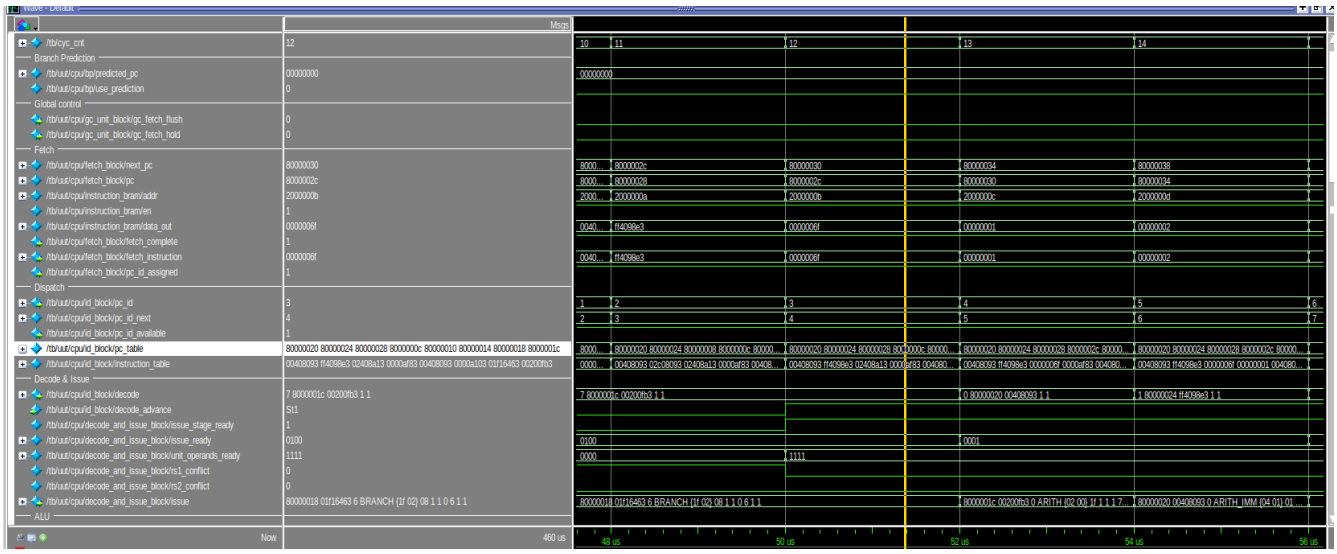

Рисунок 2.9 – Стадии декодирования и выполнения команды с адресом 8000001c

Трасса выполнения программы представлена на рисунке 2.10.

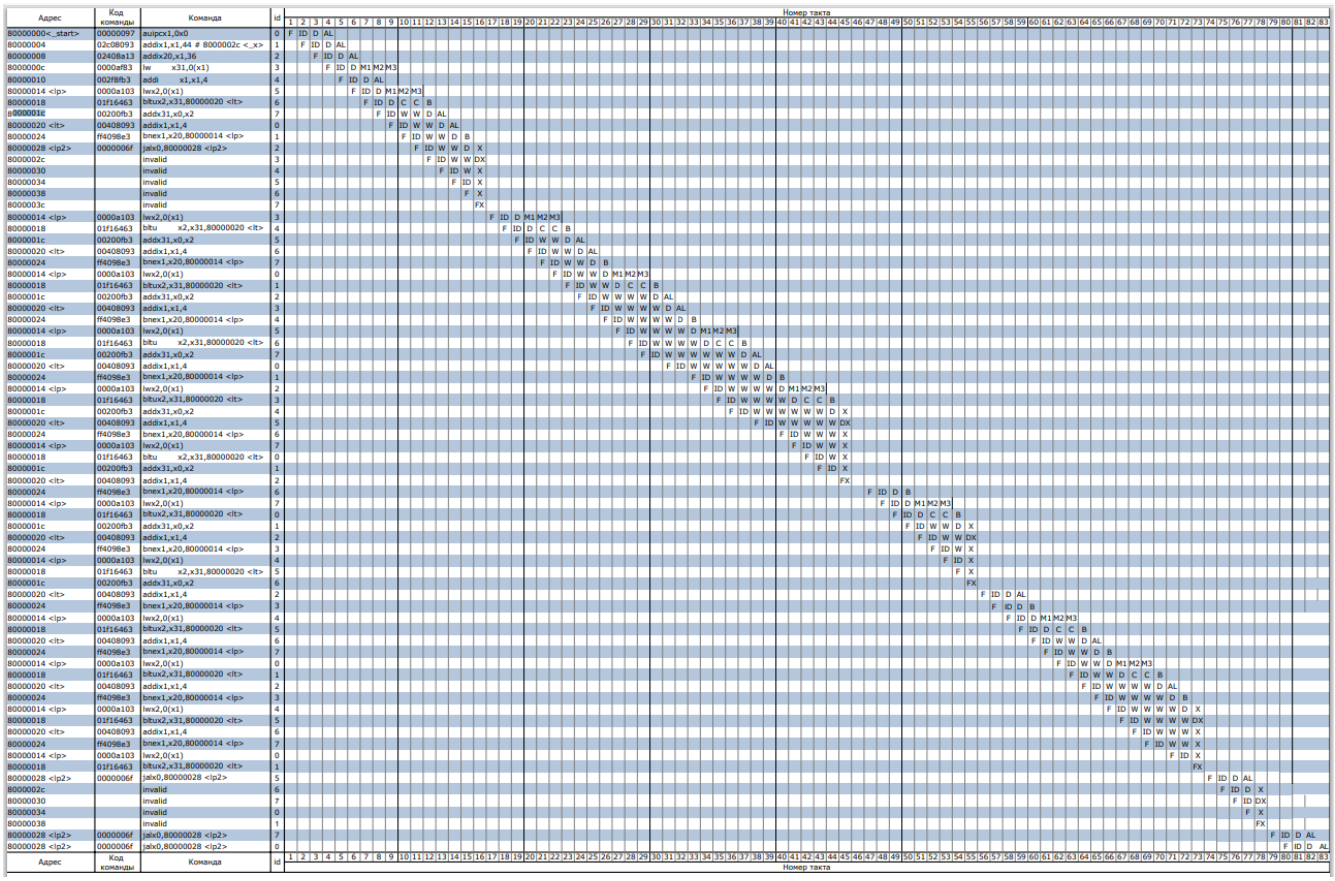

Рисунок 2.10 – Трасса выполнения программы 14-го варианта

Из трассы видно возникновение конфликтов, которые замедляют работу программы. Для оптимизации можно перенести команду $addi\ \ x1, x1, 4$ в место между конфликтующими командами.

Ниже приведены ассемблерный и дизассемблерный коды оптимизированной программы.

Листинг 2.7 – Исходный код оптимизированной программы для 14-го варианта

```
 .section .text
        .globl _start;
        len = 9 #Размер массива
        enroll = 1 #Количество обрабатываемых элементов за одну
           итерацию
        elem_sz = 4 #Размер одного элемента массива

_start:
        la x1, _x
        addi x20, x1, elem_sz*len #Адрес элемента, следующего за
           последним
        lw x31, 0(x1)
        add x1, x1, elem_sz*1
lp:
        lw x2, 0(x1)
        add x1, x1, elem_sz*enroll
        bltu x2, x31, lt
        add x31, x0, x2 #!
lt:
        bne x1, x20, lp
lp2: j lp2

        .section .data
_x:     .4byte 0x1
        .4byte 0x2
        .4byte 0x3
        .4byte 0x4
        .4byte 0x8
        .4byte 0x6
        .4byte 0x7
        .4byte 0x5
        .4byte 0x4
```

Листинг 2.8 – Дизассемблированный код оптимизированной программы для 14-го варианта

```
Дизассемблирование раздела .text:

80000000 <_start>:
80000000:   00000097            auipc   x1,0x0
80000004:   02c08093            addi    x1,x1,44 # 8000002c <_x>
80000008:   02408a13            addi    x20,x1,36
8000000c:   0000af83            lw  x31,0(x1)
80000010:   00408093            addi    x1,x1,4

80000014 <lp>:
80000014:   0000a103            lw  x2,0(x1)
80000018:   00408093            addi    x1,x1,4
8000001c:   01f16463            bltu    x2,x31,80000024 <lt>
80000020:   00200fb3            add x31,x0,x2

80000024 <lt>:
80000024:   ff4098e3            bne x1,x20,80000014 <lp>

80000028 <lp2>:
80000028:   0000006f            jal x0,80000028 <lp2>
```

Трасса выполнения оптимизированной программы представлена на рисунке 2.11.

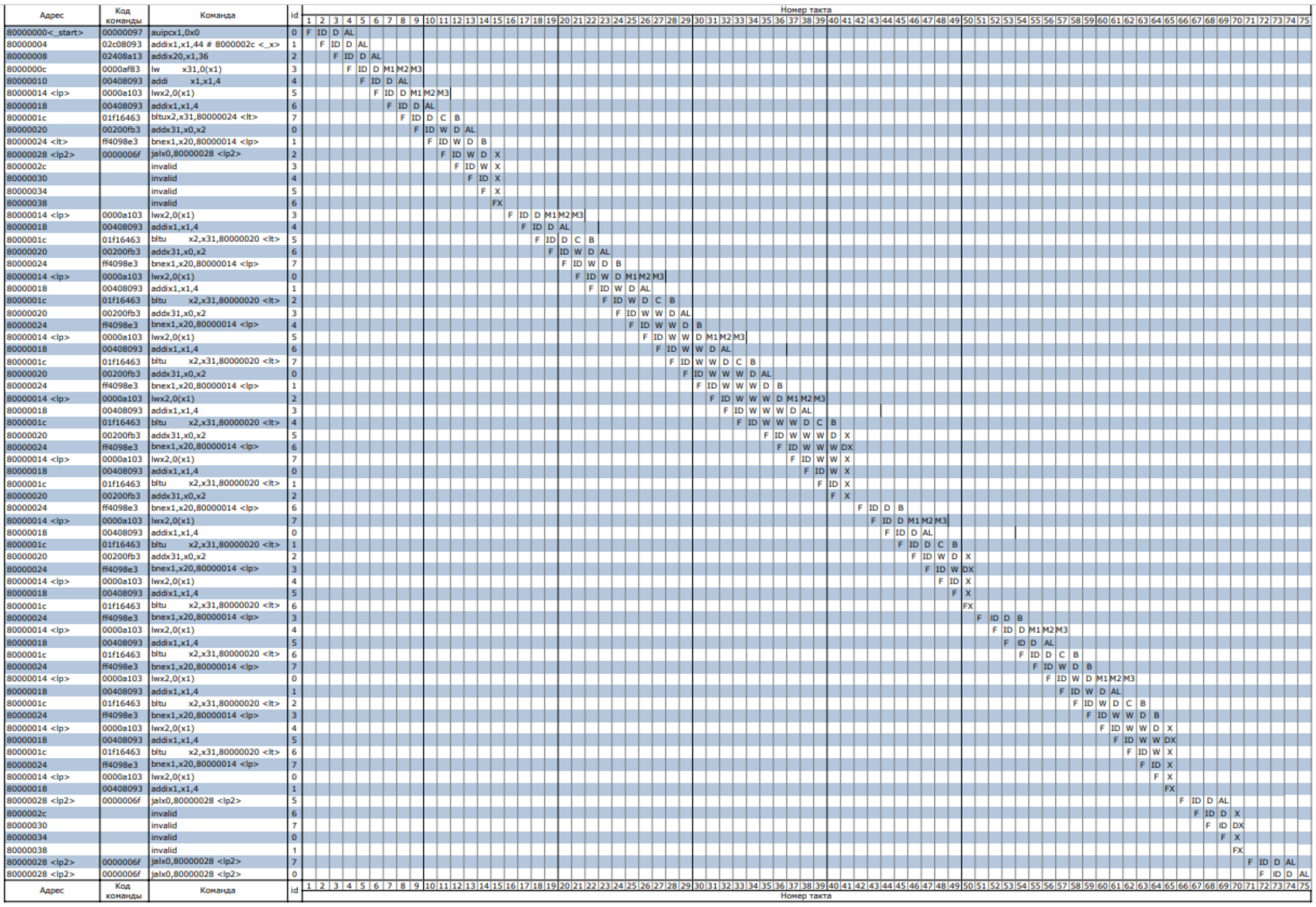

Рисунок 2.11 – Трасса выполнения оптимизированной программы 14-го варианта

Проанализировав обе трассы, можно увидеть, что после оптимизаций программа стала работать на 8 тактов быстрее.

Заключение

В результате выполнения лабораторной работы были изучены принципы функционирования, построения и особенности архитектуры суперскалярных конвейерных микропроцессоров, были рассмотрены принципы проектирования и верификации сложных цифровых устройств с использованием языка описания аппаратуры SystemVerilog и ПЛИС.

На основе изученных материалов был найден способ оптимизации программы.

Поставленная цель достигнута.