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 КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О т ч е т
по лабораторной работе № 2

Название: **Изучение принципов работы микропроцессорного ядра RISCV**

Дисциплина: **Архитектура электронно-вычислительных систем**

Вариант: **19**

Студент гр. ИУ7-55Б _____ О.Н.Талышева
(Подпись, дата) (И.О. Фамилия)

Преподаватель _____ А.Ю. Попов
(Подпись, дата) (И.О. Фамилия)

2024 год

Оглавление

Цель работы 3
1. Основные теоретические сведения: Архитектура набора команд RV32I 4
1.1. Регистровая модель 4
1.2. Модель памяти 4
1.3. Система команд 5
1.4. Пример программы 5
2. Выполнение заданий 9
2.1. Задание №1 9
2.1.1. Подготовительные операции 9
2.1.2. Условие выполнения задания 9
2.1.3. Результат выполнения задания 10
2.2. Задание №2 13
2.2.1. Условие выполнения задания 13
2.2.2. Результат выполнения задания 13
2.3. Задание №3 14
2.3.1. Условие выполнения задания 14
2.3.2. Результат выполнения задания 14
2.4. Задание №4 15
2.4.1. Условие выполнения задания 15
2.4.2. Результат выполнения задания 15
2.5. Задание №5 15
2.5.1. Условие выполнения задания 15
2.5.2. Результат выполнения задания 16
Заключение 23

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: Архитектура набора команд RV32I

RISC-V является открытым современным набором команд, который может использоваться для построения как микроконтроллеров, так и высокопроизводительных микропроцессоров. В связи с такой широкой областью применения в систему команд введена вариативность. 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. В рамках данного набора команд мы не будем рассматривать системные команды, связанные с таймерами, системными регистрами, управлением привилегиями, прерываниями и исключениями.

В настоящем разделе описывается архитектура набора команд, то есть архитектура абстрактной вычислительной машины с точки зрения набора команд без связи с конкретной аппаратной реализацией.

1.1. Регистровая модель

Набор команд RV32I предполагает использование 32 регистров общего назначения x0-x31 размером в 32 бита каждый и регистр pc, хранящего адрес следующей команды. Все регистры общего назначения равноправны, в любой команде могут использоваться любые из регистров. Регистр pc не может использоваться в командах.

Регистр x0 всегда содержит значение 0. Запись в него не производит никакого эффекта.

Существует соглашение, предполагающее использование некоторых регистров для определенных целей (например, для передачи аргументов при вызове функций или для возврата результата), однако, данное соглашение никак не связано с архитектурой и потому не будет приниматься нами во внимание.

1.2. 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.

1.3. 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 (кроме команды auipc) задаются явно. В том случае, если операндами являются регистры, мы будем их называть исходными регистрами (rs, source register), регистр, в который сохраняется результат — целевым регистром (rd, destination register).

В отличие от большинства других архитектур в RISC-V не используется понятие флага, вместо них используются команды условного перехода с использованием сравнения регистров.

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.

1.4. Пример программы

Рассмотрим пример небольшой программы для RV32I, которым мы будем пользоваться далее для исследования процесса выполнения команд.

Данная программа выполняет суммирование значений элементов массива слов и увеличивает это значение на 1.

```
.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 — Объявление констант.

4 — Метка.

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.

Можно сказать, что данная программа эквивалентна следующему псевдокоду 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){}
}
```

Если выполнить компиляцию и дизассемблирование данной программы, то получится следующий результат.

Заметим, что адреса и коды команд приведены в шестнадцатеричной системе счисления.

```
80000000 <_start>:
80000000:   00200a13          addi    x20,x0,2
80000004:   00000097          auipc   x1,0x0 (1)
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>
```

1 — Видно, что команда la x1, _x превращается в 2 команды: auipc и addi. Первая из них используется для формирования значения в старшей части регистра x1. После её выполнения в x1 запишется значение 0x80000004 (адрес команды auipc). После выполнения команды addi, регистр x1 станет равен 0x80000004 + 60

= 0x80000040, то есть адресу нулевого элемента массива _x.

2. Выполнение заданий

2.1. Задание №1

2.1.1. Подготовительные операции

Приступая к выполнению практической части лабораторной работы необходимо получить копию репозитория, содержащего все необходимые файлы.

В результате в каталоге будет создан подкаталог riscv-lab, а в нем, в свою очередь, следующие подкаталоги:

1. taiga. Содержит проект Quartus и все исходные тексты на языке SystemVerilog.
2. src. Содержит исходные тексты тестового примера программы и сборочные файлы.

2.1.2. Условие выполнения 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 (19). Поместить его в каталог src. Текст программы сохранить в файле с расширением .s.

   При выполнении данного пункта не изменяйте файл test.s, но поместите текст программы по индивидуальному варианту в новый файл с другим именем.
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.

2.1.3. Результат выполнения задания

Программа по варианту 19:

```
        .section .text
        .globl _start;
        len = 9 #Размер массива
        enroll = 2 #Количество обрабатываемых элементов за одну итерацию
        elem_sz = 4 #Размер одного элемента массива

_start:
        la x1, _x
        addi x20, x0, (len-1)/enroll
        lw x31, 0(x1)
        addi x1, x1, elem_sz*1
lp:
        lw x2, 0(x1)
        lw x3, 4(x1)
        bltu x2, x31, lt1
        add x31, x0, x2 #!
lt1:    bltu x3, x31, lt2
        add x31, x0, x3
lt2:
        add x1, x1, elem_sz*enroll
        addi x20, x20, -1
        bne x20, x0, lp
lp2: j lp2

        .section .data
_x:     .4byte 0x1
        .4byte 0x2
        .4byte 0x3
        .4byte 0x4
        .4byte 0x5
        .4byte 0x6
        .4byte 0x7
        .4byte 0x8
        .4byte 0x9
```

После анализа данной программы можно сказать, что она эквивалентна следующему коду на C:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define len 9        // Размер массива
#define enroll 2          // Количество обрабатываемых элементов за одну итерацию

int _x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

int main()
{
   int x20 = (len - 1) / enroll;
   int *x1 = _x;
   int x31 = x1[0];  // Инициализация x31 первым элементом массива
   x1 += 1;
   do {
      int x2 = x1[0];
      int x3 = x1[1];
      if (abs(x2) >= abs(x31))
         x31 = x2;
      if (abs(x3) >= abs(x31))
         x31 = x3;
      x1 += enroll;  // Переход к следующей паре элементов
      x20 -= 1;
   } while (x20 != 0);
   printf("Максимальное значение по модулю: %d.\n", x31);
   return 0;
}
```

После выполнения программы по варианту 19 в $x31$ будет записано число 9 (подсчитано вручную и написанной программы на языке С по псевдокоду).

Выполнение команды $make$, после присвоения SRC = название файла, где содержится код программы варианта 19, в мое случае этот файл был назван new. Дизассемблированный код 19 варианта:

```
riscv64-linux-gnu-as --march=rv32i new.s -o new.o
riscv64-linux-gnu-ld -b elf32-littleriscv -T link.ld new.o -o new.elf
riscv64-linux-gnu-objdump -D -M numeric,no-aliases -t new.elf

new.elf:     file format elf32-littleriscv

SYMBOL TABLE:
80000000 l    d  .text  00000000 .text
8000003c l    d  .data  00000000 .data
00000000 l    df *ABS*  00000000 new.o
00000009 l       *ABS*  00000000 len
00000002 l       *ABS*  00000000 enroll
```

```
00000004 l      *ABS*  00000000 elem_sz
8000003c l      .data  00000000 _x
80000014 l      .text  00000000 lp
80000024 l      .text  00000000 lt1
8000002c l      .text  00000000 lt2
80000038 l      .text  00000000 lp2
80000000 g       .text  00000000 _start
80000060 g       .data  00000000 _end


Disassembly of section .text:

80000000 <_start>:
80000000:       00000097            auipc   x1,0x0
80000004:       03c08093            addi    x1,x1,60 # 8000003c <_x>
80000008:       00400a13            addi    x20,x0,4
8000000c:       0000af83            lw      x31,0(x1)
80000010:       00408093            addi    x1,x1,4

80000014 <lp>:
80000014:       0000a103            lw      x2,0(x1)
80000018:       0040a183            lw      x3,4(x1)
8000001c:       01f16463            bltu    x2,x31,80000024 <lt1>
80000020:       00200fb3            add     x31,x0,x2

80000024 <lt1>:
80000024:       01f1e463            bltu    x3,x31,8000002c <lt2>
80000028:       00300fb3            add     x31,x0,x3

8000002c <lt2>:
8000002c:       00808093            addi    x1,x1,8
80000030:       fffa0a13            addi    x20,x20,-1
80000034:       fe0a10e3            bne     x20,x0,80000014 <lp>

80000038 <lp2>:
80000038:       0000006f            jal     x0,80000038 <lp2>

Disassembly of section .data:

8000003c <_x>:
8000003c:       0001                c.addi  x0,0
8000003e:       0000                c.unimp
80000040:       0002                c.slli64        x0
80000042:       0000                c.unimp
```

```
80000044:	00000003	        lb      x0,0(x0) # 0 <enroll-0x2>
80000048:	0004	        .2byte  0x4
8000004a:	0000	        c.unimp
8000004c:	0005	        c.addi  x0,1
8000004e:	0000	        c.unimp
80000050:	0006	        c.slli  x0,0x1
80000052:	0000	        c.unimp
80000054:	00000007	        .4byte  0x7
80000058:	0008	        .2byte  0x8
8000005a:	0000	        c.unimp
8000005c:	0009	        c.addi  x0,2
      ...
riscv64-linux-gnu-objcopy -O binary --reverse-bytes=4 new.elf new.bin
```

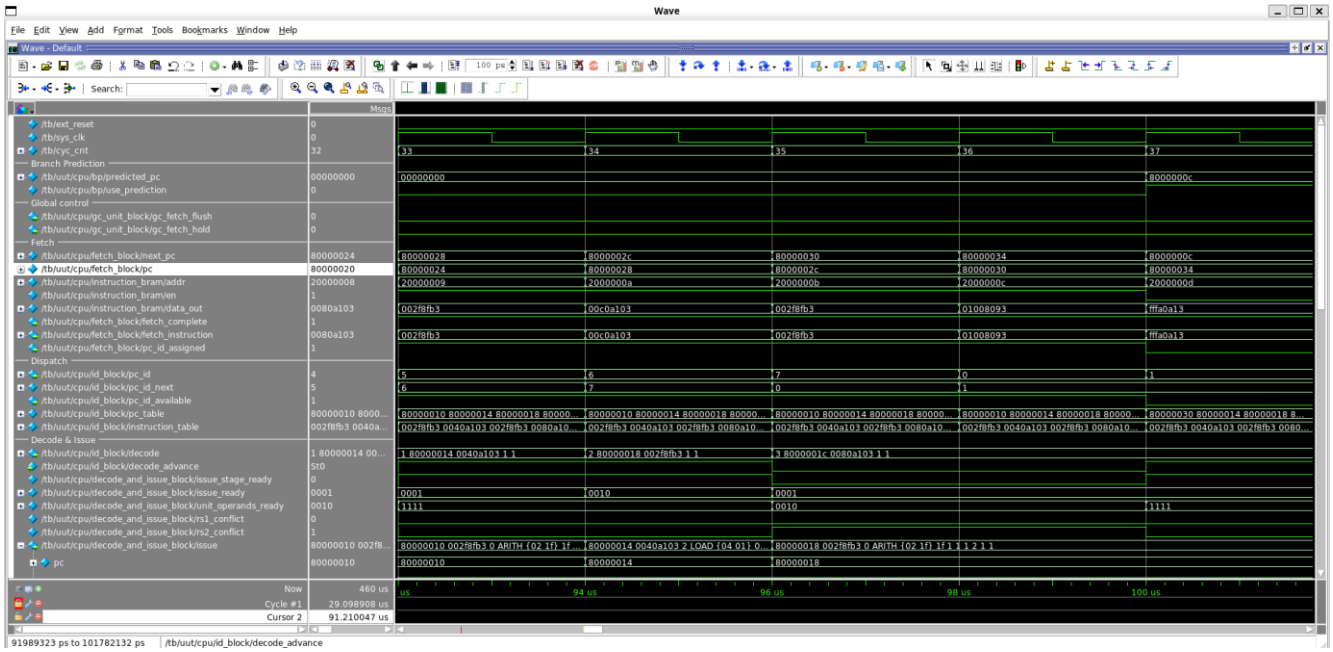
2.2. Задание №2

2.2.1. Условие выполнения задания

1. В ходе выполнения данного задания необходимо выполнить следующие действия:
2. Запустить симуляцию в среде Modelsim. Для этого найти в каталоге taiga файл run.sh и запустить его двойным щелчком мыши.
3. Запустить симуляцию, набрав в командной строке Modelsim команду run 460us.
4. Изучить список сигналов, приведенных в окне Wave.
5. В соответствии с таблицей, получить снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 80000028 на второй итерации.

2.2.2. Результат выполнения задания

Снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 80000028 на второй итерации:



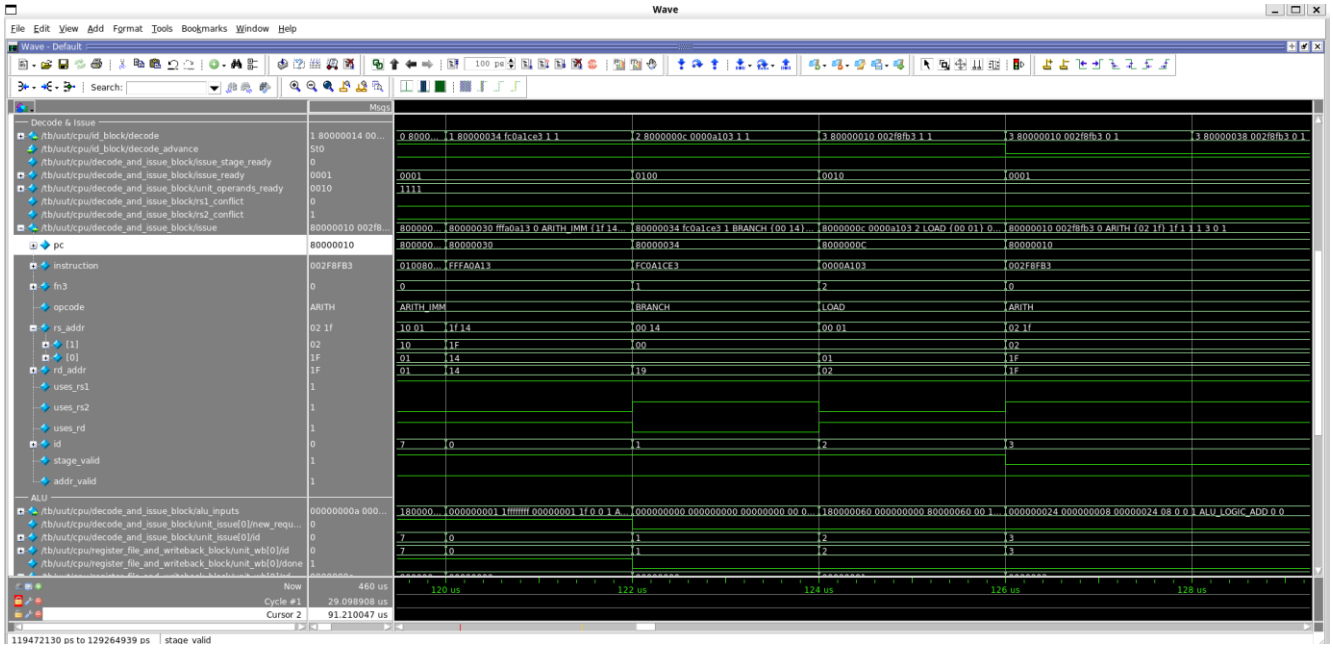
2.3. Задание №3

2.3.1. Условие выполнения задания

Получить снимок экрана, содержащий временную диаграмму выполнения стадии декодирования и планирования на выполнение команды с адресом 80000034 на второй итерации.

2.3.2. Результат выполнения задания

Снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 80000034 на второй итерации:



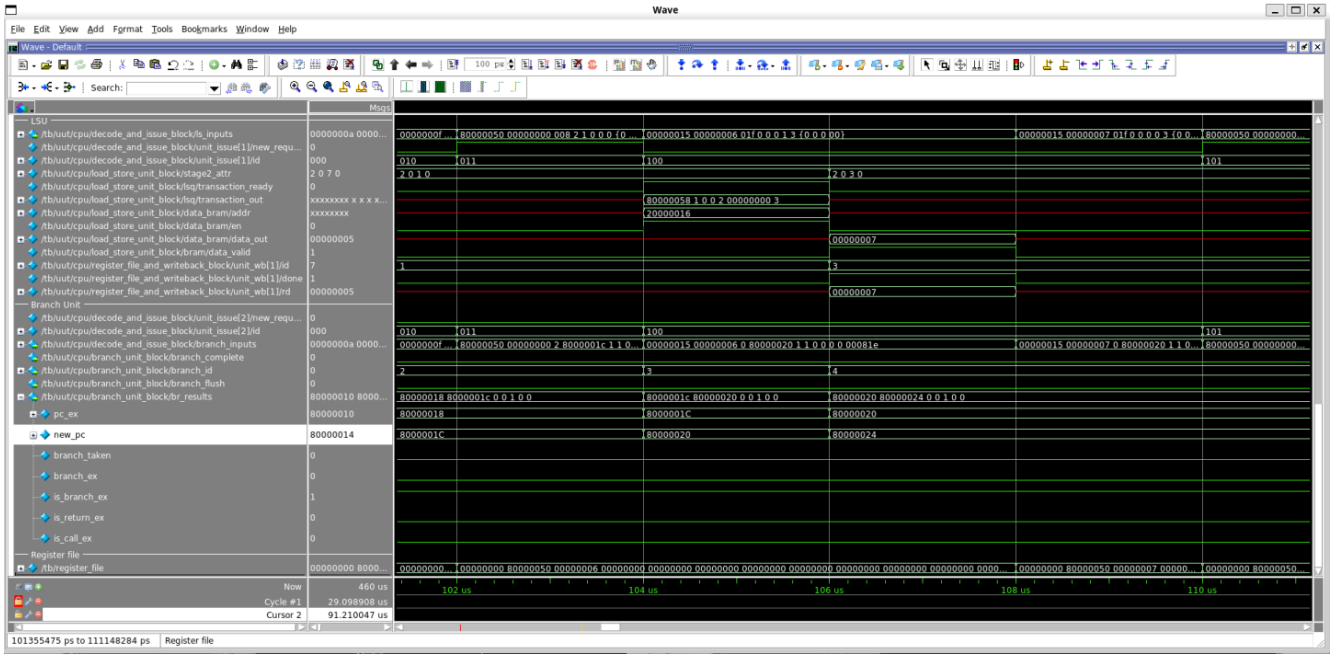
2.4. Задание №4

2.4.1. Условие выполнения задания

Получить снимок экрана, содержащий временную диаграмму выполнения стадии выполнения команды с адресом 80000020 на второй итерации.

2.4.2. Результат выполнения задания

Снимок экрана, содержащий временную диаграмму выполнения стадии выполнения команды с адресом 80000020 на второй итерации:



2.5. Задание №5

2.5.1. Условие выполнения задания

В процессе выполнения этого задания необходимо выполнить следующие действия:

1. Исправить файл taiga/run.sh так, чтобы там был указан путь к файлу new.hex, соответствующему программе по индивидуальному варианту. Сохранить файл.
2. Закрыть Modelsim.
3. Запустить симуляцию 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.

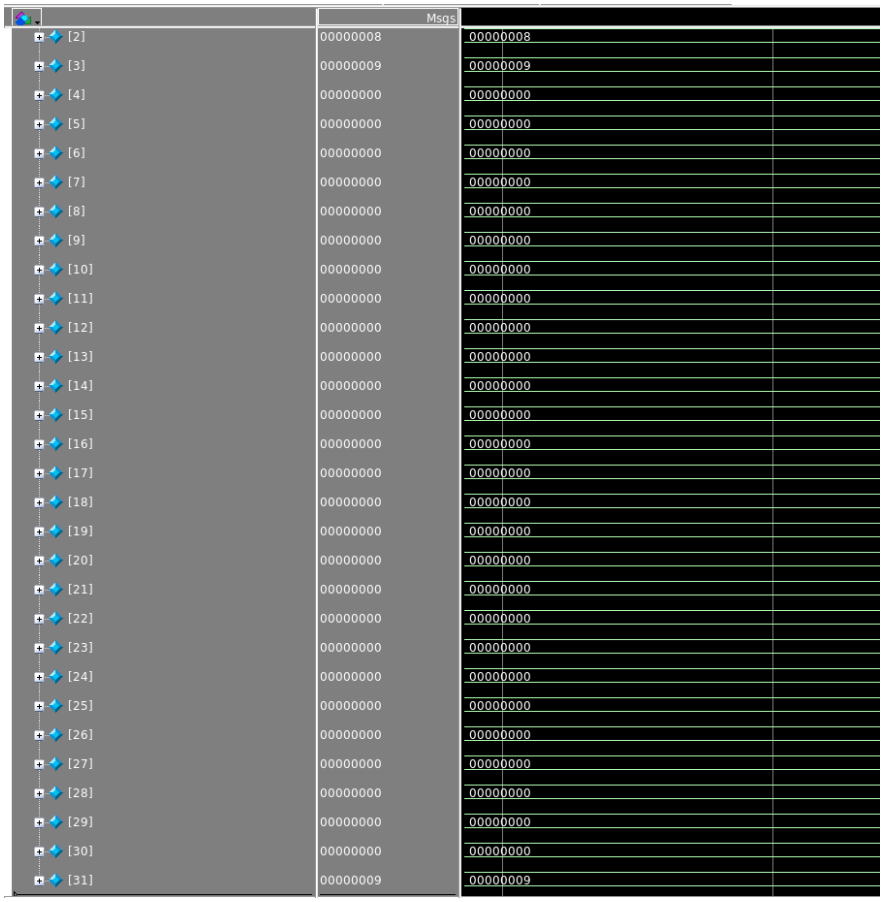
10. Перекомпилировать программу и перезапустить симуляцию.

11. Заполнить трассу выполнения оптимизированной программы.

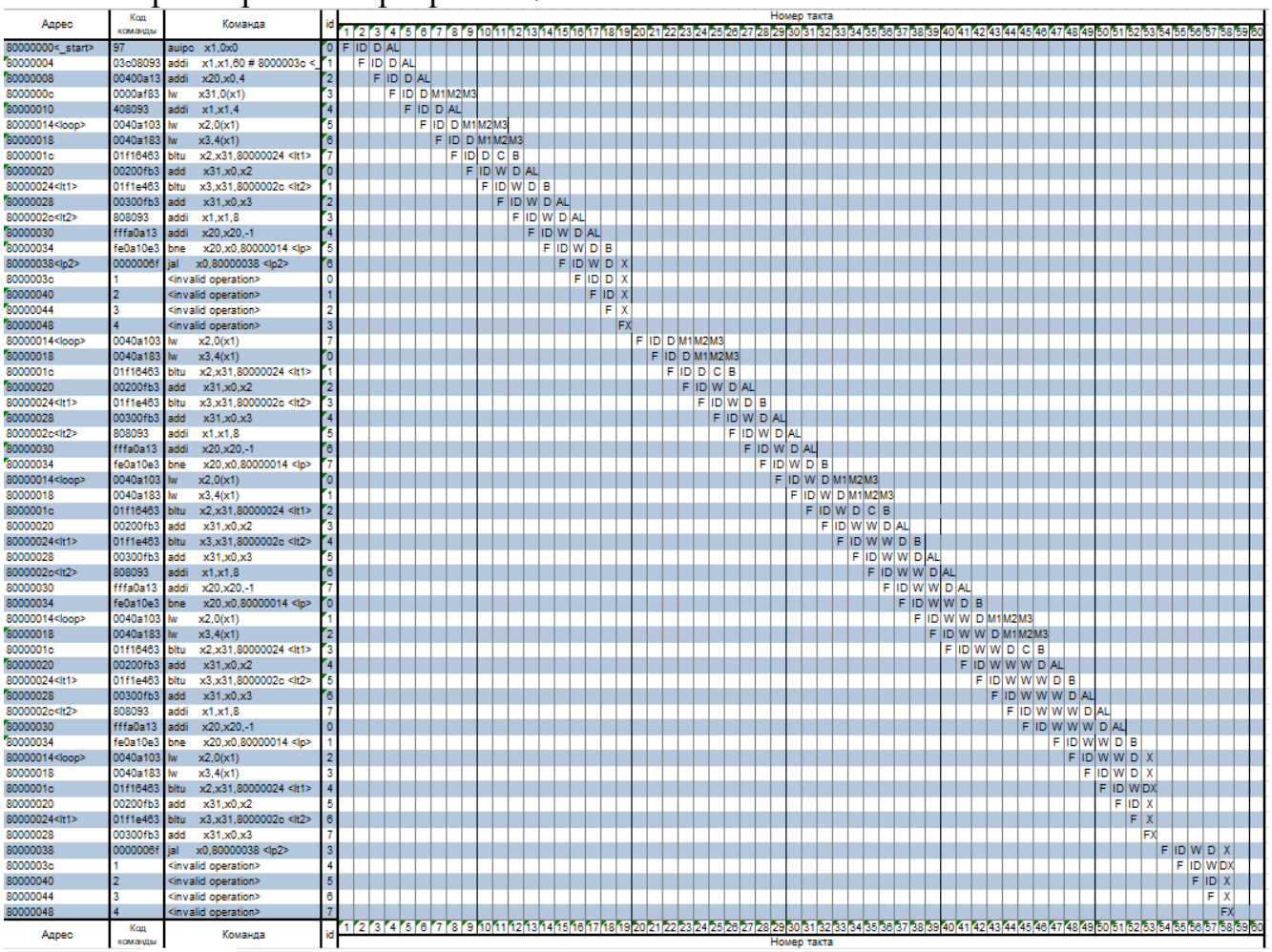
12. Сравнить трассы выполнения неоптимизированной и оптимизированной версии, сделать выводы.

2.5.2. Результат выполнения задания

Результат работы программы (значение регистра x31) действительно равно 9.



Трасса работы программы:

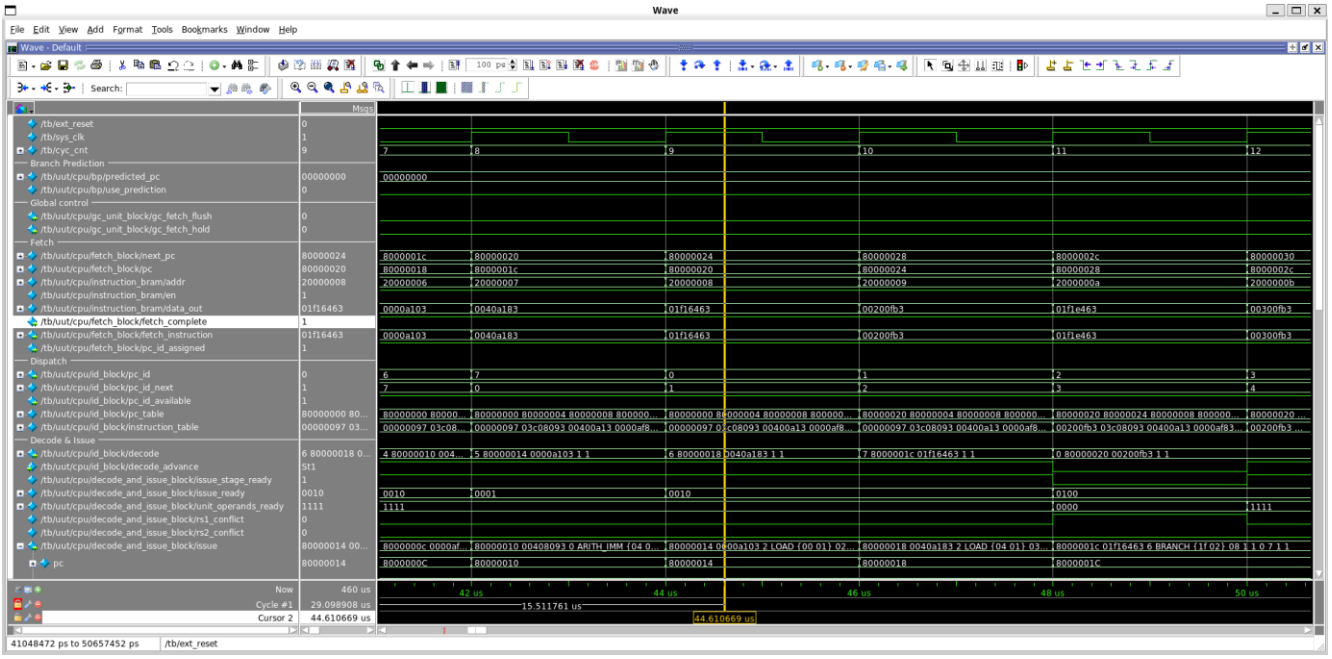


Где:

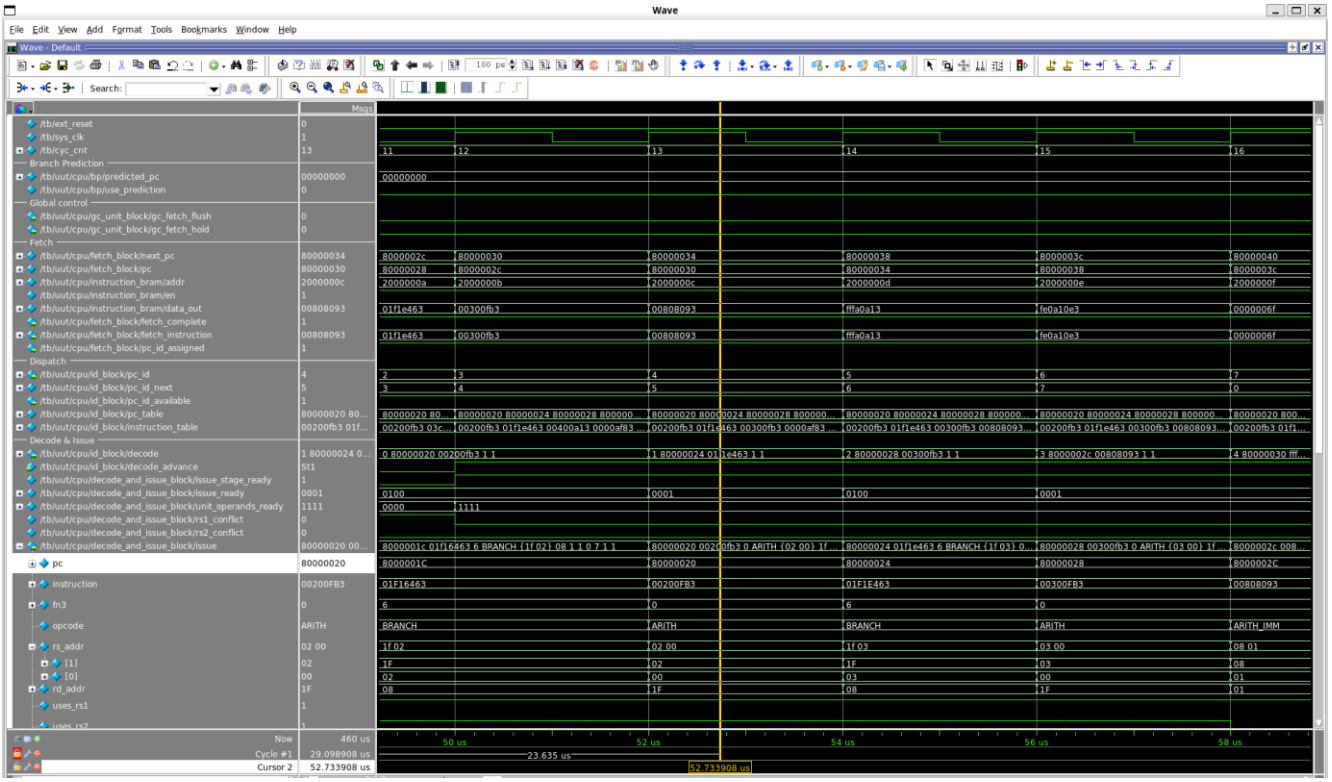
F	Такт, в котором происходит операция выборки;
ID	Такт, в котором происходит операция диспетчеризации;
D	Такт, в котором происходит операция декодирования;
C	Такт, в котором команда не выполняется из-за конфликта;
W	Такт, в котором не происходит декодирования команды из-за загрузки блока декодирования;
X	Такт, в котором происходит сброс команд, находящихся в очереди;
DX	Такт, в котором декодирование команды происходит, но его результаты отбрасываются;
FX	Такт, в котором выборка команды происходит, но его результаты отбрасываются;
M1, M2, M3	Первый, второй и третий такты, в которых происходит выполнение команды доступа к памяти;
AL	Такт, в котором происходит выполнение команды АЛУ;
B	Такт, в котором происходит выполнение команды ветвления.

Далее представлены временные диаграммы сигналов, соответствующих всем стадиям выполнения команды, обозначенной в тексте программы символом #! (add x31, x0, x2) с адресом 80000020.

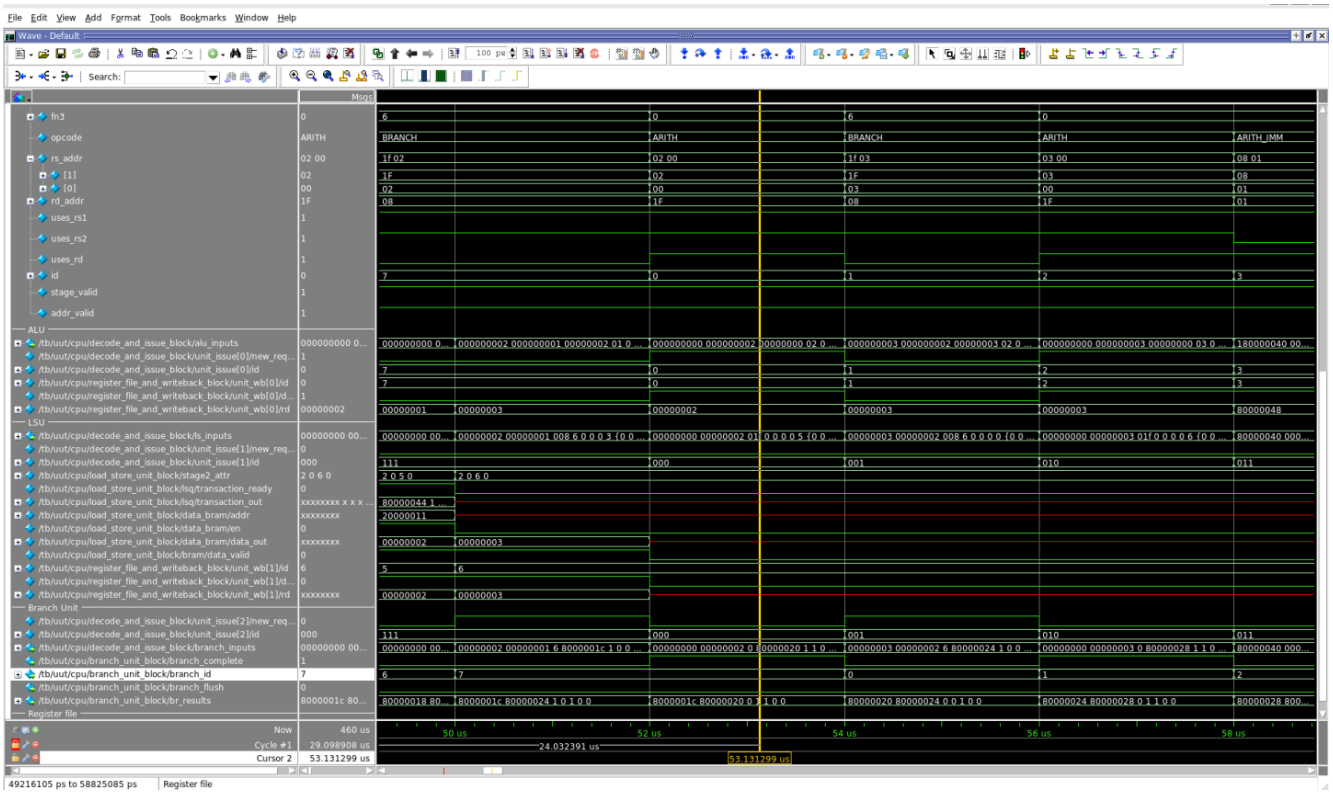
Снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 80000020 на первой итерации:



Снимок экрана, содержащий временную диаграмму выполнения стадии декодирования и планирования команды с адресом 80000020 на первой итерации:



Снимок экрана, содержащий временную диаграмму сигналов ALU с адресом 80000020 на первой итерации:



Вывод об эффективности выполнения программы и о путях оптимизации:

На трассе работы программы, представленной на рисунке, видно, что конфликты возникают из-за попытки выполнить операцию сложения до завершения загрузки необходимых данных в память. Это приводит к задержкам, так как операция сложения требует данные, которые ещё не были загружены.

Для оптимизации программы можно предварительно загрузить все необходимые данные в память, а затем выполнять операции сложения. Такой подход устранит конфликты, поскольку операции будут выполняться после полной загрузки данных, что исключает необходимость ожидания завершения загрузки.

В результате оптимизация программы позволит сократить её выполнение на 3 такта.

Оптимизированная программа:

```
.section .text
    .globl _start;
    len = 9 #Размер массива
    enroll = 2 #Количество обрабатываемых элементов за одну итерацию
    elem_sz = 4 #Размер одного элемента массива
```

```
_start:
        la x1, _x
        addi x20, x0, (len-1)/enroll
        lw x31, 0(x1)
        addi x1, x1, elem_sz*1
lp:
        lw x2, 0(x1)
        lw x3, 4(x1)
        addi x20, x20, -1
        bltu x2, x31, lt1
        add x31, x0, x2 #!
lt1:    bltu x3, x31, lt2
        add x31, x0, x3
lt2:
        add x1, x1, elem_sz*enroll
        bne x20, x0, lp
lp2: j lp2

        .section .data
_x:     .4byte 0x1
        .4byte 0x2
        .4byte 0x3
        .4byte 0x4
        .4byte 0x5
        .4byte 0x6
        .4byte 0x7
        .4byte 0x8
        .4byte 0x9
```

Дизассемблерный код оптимизированной программы:

Disassembly of section .text:

```
80000000 <_start>:
80000000:       00000097            auipc   x1,0x0
80000004:       03c08093            addi    x1,x1,60 # 8000003c <_x>
80000008:       00400a13            addi    x20,x0,4
8000000c:       0000af83            lw      x31,0(x1)
80000010:       00408093            addi    x1,x1,4

80000014 <lp>:
80000014:       0000a103            lw      x2,0(x1)
80000018:       0040a183            lw      x3,4(x1)
8000001c:       fffa0a13            addi    x20,x20,-1
```

```
80000020:       01f16463            bltu    x2,x31,80000028 <lt1>
80000024:       00200fb3            add     x31,x0,x2

80000028 <lt1>:
80000028:       01f1e463            bltu    x3,x31,80000030 <lt2>
8000002c:       00300fb3            add     x31,x0,x3

80000030 <lt2>:
80000030:       00808093             addi    x1,x1,8
80000034:       fe0a10e3            bne     x20,x0,80000014 <lp>

80000038 <lp2>:
80000038:       0000006f            jal     x0,80000038 <lp2>
```

Disassembly of section .data:

```
8000003c <_x>:
8000003c:       0001                c.addi  x0,0
8000003e:       0000                c.unimp
80000040:       0002                c.slli64        x0
80000042:       0000                c.unimp
80000044:       00000003              lb      x0,0(x0) # 0 <enroll-0x2>
80000048:       0004                .2byte  0x4
8000004a:       0000                c.unimp
8000004c:       0005                c.addi  x0,1
8000004e:       0000                c.unimp
80000050:       0006                c.slli  x0,0x1
80000052:       0000                c.unimp
80000054:       00000007              .4byte  0x7
80000058:       0008                .2byte  0x8
8000005a:       0000                c.unimp
8000005c:       0009                c.addi  x0,2
```

После анализа данной программы можно сказать, что она эквивалентна следующему коду на C:

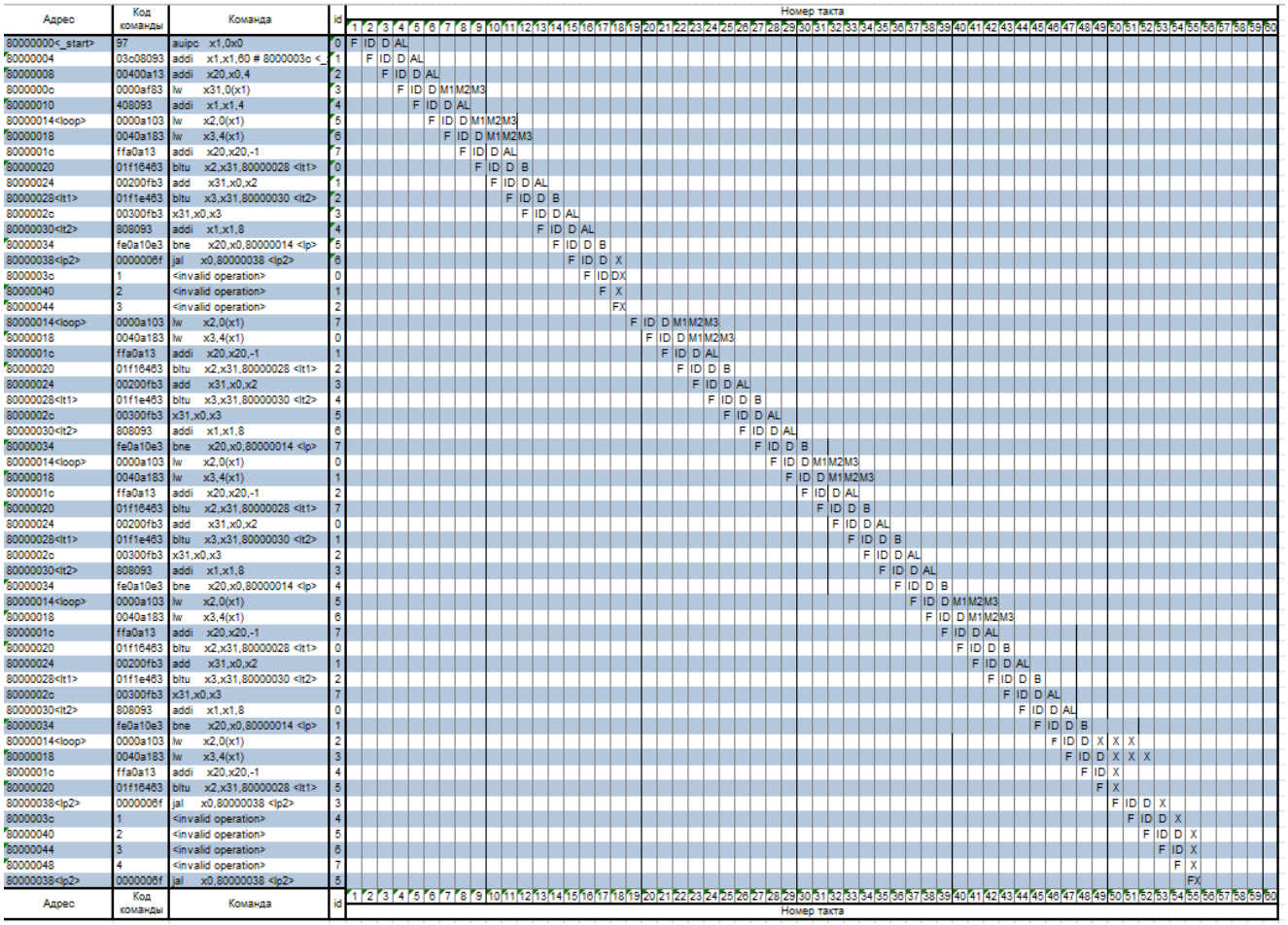
```
#include <stdio.h>
#include <stdlib.h>

#define len 9        // Размер массива
#define enroll 2          // Количество обрабатываемых элементов за одну итерацию

int _x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
int main()
{
    int x20 = (len - 1) / enroll;
    int *x1 = _x;
    int x31 = x1[0];  // Инициализация x31 первым элементом массива
    x1 += 1;
    do {
        int x2 = x1[0];
        int x3 = x1[1];
     x20 -= 1;
        if (abs(x2) >= abs(x31))
            x31 = x2;
        if (abs(x3) >= abs(x31))
            x31 = x3;
        x1 += enroll;  // Переход к следующей паре элементов
    } while (x20 != 0);
    printf("Максимальное значение по модулю: %d.\n", x31);
    return 0;
}
```

Трасса работы оптимизированной программы:



Адрес	Код команды	Команда	id	Номер такта
80000000<_start>	97	auipc x1,0x0	0	F ID D AL
80000004	03c08093	addi x1,x1,60 # 8000003c <_	1	F ID D AL
80000008	00400a13	addi x20,x0,4	2	F ID D AL
8000000c	0000af83	lw x31,0(x1)	3	F ID D M1 M2 M3
80000010	408093	addi x1,x1,4	4	F ID D AL
80000014<loop>	0000a103	lw x2,0(x1)	5	F ID D M1 M2 M3
80000018	0040a183	lw x3,4(x1)	6	F ID D M1 M2 M3
8000001c	ffa0a13	addi x20,x20,-1	7	F ID D AL
80000020	01f16463	bltu x2,x31,80000028 <lt1>	0	F ID D B
80000024	00200fb3	add x31,x0,x2	1	F ID D AL
80000028<lt1>	01f1e463	bltu x3,x31,80000030 <lt2>	2	F ID D B
8000002c	00300fb3	x31,x0,x3	3	F ID D AL
80000030<lt2>	808093	addi x1,x1,8	4	F ID D AL
80000034	fe0a10e3	bne x20,x0,80000014 <lp>	5	F ID D B
80000038<lp2>	0000006f	jal x0,80000038 <lp2>	6	F ID D X
8000003c	1	<invalid operation>	0	F ID DX
80000040	2	<invalid operation>	1	F X
80000044	3	<invalid operation>	2	FX
80000014<loop>	0000a103	lw x2,0(x1)	7	F ID D M1 M2 M3
80000018	0040a183	lw x3,4(x1)	0	F ID D M1 M2 M3
8000001c	ffa0a13	addi x20,x20,-1	1	F ID D AL
80000020	01f16463	bltu x2,x31,80000028 <lt1>	2	F ID D B
80000024	00200fb3	add x31,x0,x2	3	F ID D AL
80000028<lt1>	01f1e463	bltu x3,x31,80000030 <lt2>	4	F ID D B
8000002c	00300fb3	x31,x0,x3	5	F ID D AL
80000030<lt2>	808093	addi x1,x1,8	6	F ID D AL
80000034	fe0a10e3	bne x20,x0,80000014 <lp>	7	F ID D B
80000014<loop>	0000a103	lw x2,0(x1)	0	F ID D M1 M2 M3
80000018	0040a183	lw x3,4(x1)	1	F ID D M1 M2 M3
8000001c	ffa0a13	addi x20,x20,-1	2	F ID D AL
80000020	01f16463	bltu x2,x31,80000028 <lt1>	7	F ID D B
80000024	00200fb3	add x31,x0,x2	0	F ID D AL
80000028<lt1>	01f1e463	bltu x3,x31,80000030 <lt2>	1	F ID D B
8000002c	00300fb3	x31,x0,x3	2	F ID D AL
80000030<lt2>	808093	addi x1,x1,8	3	F ID D AL
80000034	fe0a10e3	bne x20,x0,80000014 <lp>	4	F ID D B
80000014<loop>	0000a103	lw x2,0(x1)	5	F ID D M1 M2 M3
80000018	0040a183	lw x3,4(x1)	6	F ID D M1 M2 M3
8000001c	ffa0a13	addi x20,x20,-1	7	F ID D AL
80000020	01f16463	bltu x2,x31,80000028 <lt1>	0	F ID D B
80000024	00200fb3	add x31,x0,x2	1	F ID D AL
80000028<lt1>	01f1e463	bltu x3,x31,80000030 <lt2>	2	F ID D B
8000002c	00300fb3	x31,x0,x3	7	F ID D AL
80000030<lt2>	808093	addi x1,x1,8	0	F ID D AL
80000034	fe0a10e3	bne x20,x0,80000014 <lp>	1	F ID D B
80000014<loop>	0000a103	lw x2,0(x1)	2	F ID D X X X
80000018	0040a183	lw x3,4(x1)	3	F ID D X X X
8000001c	ffa0a13	addi x20,x20,-1	4	F ID X
80000020	01f16463	bltu x2,x31,80000028 <lt1>	5	F X
80000038<lp2>	0000006f	jal x0,80000038 <lp2>	3	F ID D X
8000003c	1	<invalid operation>	4	F ID D X
80000040	2	<invalid operation>	5	F ID D X
80000044	3	<invalid operation>	6	F ID X
80000048	4	<invalid operation>	7	F X
80000038<lp2>	0000006f	jal x0,80000038 <lp2>	5	FX

Заключение

В ходе лабораторной работы были изучены принципы работы, структуры и особенности архитектуры суперскалярных конвейерных микропроцессоров. Также были рассмотрены основы проектирования и верификации сложных цифровых устройств с использованием языка описания аппаратуры SystemVerilog и ПЛИС. На основе полученных знаний был предложен метод оптимизации программы для устранения задержек, связанных с загрузкой данных. Поставленная цель работы была успешно достигнута.