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**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.03 Прикладная информатика**

ОТЧЕТ

по лабораторной работе № 1

вариант № 3

Название Изучение принципов работы микропроцессорного ядра RISC-V

Дисциплина Организация ЭВМ и систем

Студент гр. ИУ6-74Б _____ М.А. Гейне
(Подпись, дата) (И.О.Фамилия)

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

Москва, 2022

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.

Задание

В ходе выполнения лабораторной работы необходимо:

1. Провести сборку программы по индивидуальному варианту, изучить её исходный код;
2. Изучить процесс выборки и диспетчеризации команды;
3. Изучить процесс декодирования команды;
4. Изучить процесс выполнения команды;
5. Провести анализ выполнения программы по индивидуальному варианту. Провести трассировку программы. Выявить пути для оптимизации, оптимизировать и привести трассировку оптимизированной программы.

1 Сборка и изучение программы

В соответствии с индивидуальным вариантом была выбрана программа, исходный код которой приведён ниже.

```
        .section .text
        .globl _start;
        len = 8 #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-1) #Адрес последнего элемента
lp:
        lw x2, 0(x1)
        add x31, x31, x2 #!
        addi x1, x1, elem_sz*enroll
        bne x1, x20, lp
        addi x31, x31, 1
lp2: j lp2

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

По результатам сборки был получен следующий дизассемблерный листинг.

```
SYMBOL TABLE:
80000000 l    d  .text        00000000 .text
80000024 l    d  .data        00000000 .data
00000000 l    df *ABS*        00000000 task.o
00000008 l       *ABS*        00000000 len
00000001 l       *ABS*        00000000 enroll
00000004 l       *ABS*        00000000 elem_sz
80000024 l       .data        00000000 _x
8000000c l       .text        00000000 lp
80000020 l       .text        00000000 lp2
80000000 g       .text        00000000 _start
```

```
80000044 g       .data         00000000 _end


Disassembly of section .text:

80000000 <_start>:
80000000:        00000097                  auipc        x1,0x0
80000004:        02408093                  addi         x1,x1,36
 ↪  # 80000024 <_x>
80000008:        01c08a13                  addi         x20,x1,28

8000000c <lp>:
8000000c:        0000a103                  lw         x2,0(x1)
80000010:        002f8fb3                  add         x31,x31,x2
80000014:        00408093                  addi         x1,x1,4
80000018:        ff409ae3
 ↪          bne         x1,x20,8000000c <lp>
8000001c:        001f8f93                  addi         x31,x31,1

80000020 <lp2>:
80000020:        0000006f
 ↪          jal         x0,80000020 <lp2>

Disassembly of section .data:

80000024 <_x>:
80000024:        0001                        c.addi         x0,0
80000026:        0000                        unimp
80000028:        0002                        0x2
8000002a:        0000                        unimp
8000002c:        00000003                  lb         x0,0(x0) #
 ↪  0 <enroll-0x1>
80000030:        0004
 ↪          c.addi4spn        x9,x2,0
80000032:        0000                        unimp
80000034:        0005                        c.addi         x0,1
80000036:        0000                        unimp
80000038:        0006                        0x6
8000003a:        0000                        unimp
8000003c:        00000007                  0x7
80000040:        0008
 ↪          c.addi4spn        x10,x2,0
```

Изучив исходный код программы был составлен псевдокод, приведённый ниже.

```
DATA:
_x := [1, 2, 3, 4, 5, 6, 7, 8]
```

```
begin
    x1 := указатель на _x[0]
    x20 := указатель на _x[7]

    do
        x2 := значение по x1
        x31 := x31 + x2
        x1++
    while (x1!=x20)

    x31++
end
```

Изучив алгоритм работы программы было установлено, что в конце её выполнения **x31 = 29**.

2 Изучение процессов выборки и диспетчеризации

В процессе выборки команды происходит выборка из памяти команд по адресу, который записан в PC. Выбранные данные помещаются на линию *fetch_instruction*. Выборка команды занимает один такт, однако может быть приостановлена, если далее по конвейеру данные некому обработать.

В рамках процесса диспетчеризации выбранные данные на предыдущем шаге помещаются в очередь команд, получая при этом id<8. В случае, если очередь полностью заполнена, выдаётся сигнал, приостанавливающий выборку.

По индивидуальному варианту была сохранена временная диаграмма выборки и диспетчеризации команды с адресом **80000014, 1-я итерация**, приведённая на рисунке 1.

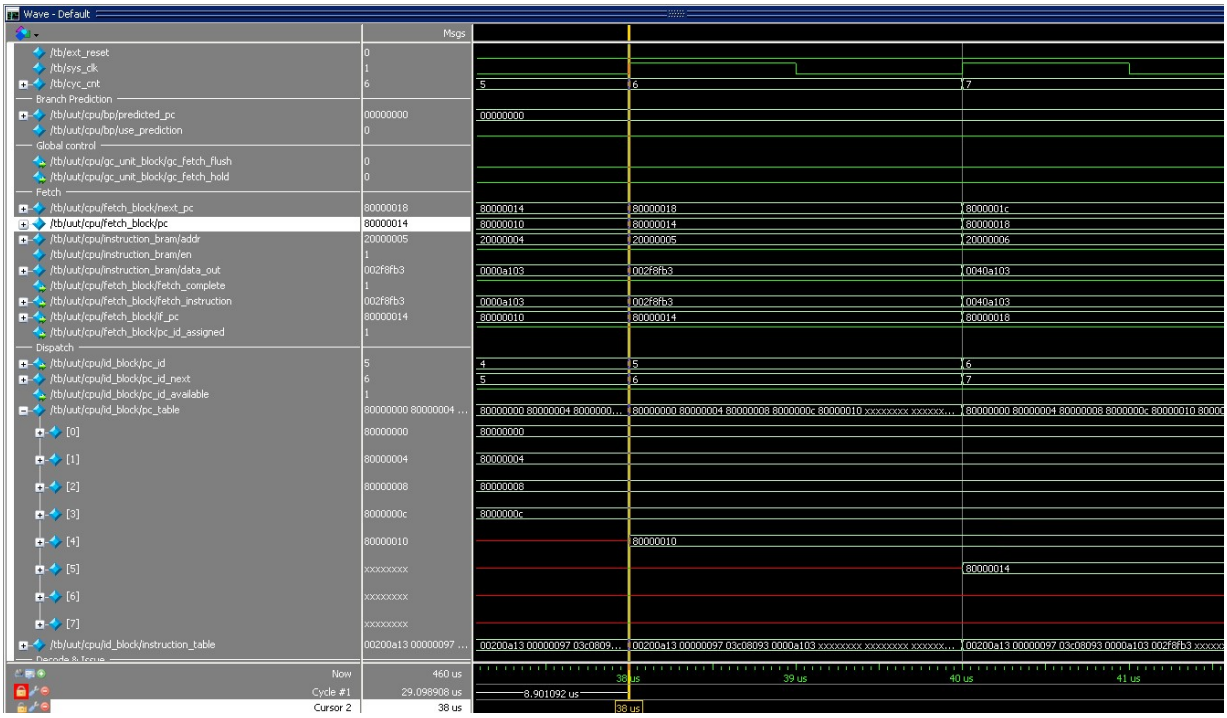

Рисунок 1 – Временная диаграмма выборки и диспетчеризации

3 Изучение процесса декодирования команды

Из очереди команд данные поступают на декодер, где происходит декодирование команды. В рамках этого процесса определяются код операции, исполнительный блок, исходные регистры и регистры назначения, специфические для команды характеристики. Декодирование производится комбинационно и занимает один такт. Следующим тактом производится планирование на выполнение. Однако в некоторых случаях можно наблюдать конфликты при выполнении операции, например при параллельной попытке доступа к одному регистру разных исполнительных блоков. В таком случае команда не выдаётся на исполнение, новая команда не выбирается из очереди.

По индивидуальному варианту была сохранена временная диаграмма декодирования команды с адресом **80000020, 1-я ите-**

рация, приведённая на рисунке 2.

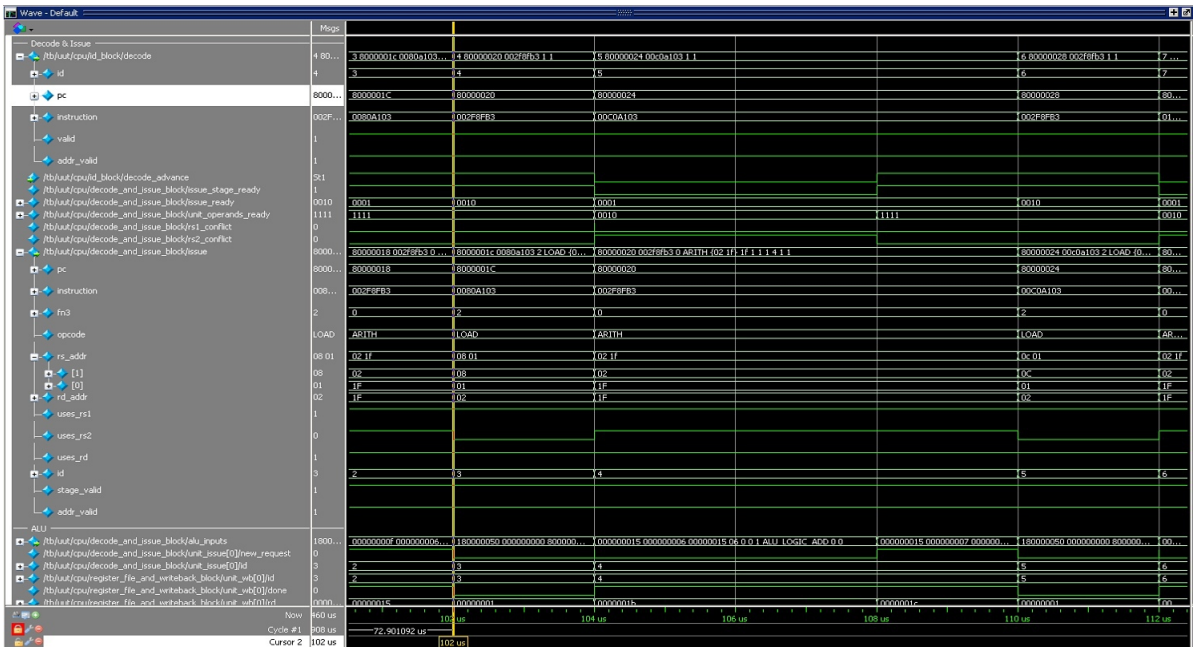

Рисунок 2 – Временная диаграмма декодирования

На диаграмме можно наблюдать, что после декодирования и выдачи на исполнение был обнаружен конфликт (выставлен сигнал *rs2_conflict*). По этой причине сигнал *decode_advance* снят, новая команда не выбирается из очереди, а уже поступившая на декодер - не подаётся на исполнение. После снятия сигнала о конфликте процесс возобновляется.

4 Изучение процесса исполнения команды

После декодирования команда отдаётся на исполнение. Исполнительный блок определяется сигналом *new_request[i]*, где i=0,1,2 - номер блока. Блоку АЛУ соответствует i=0, блоку обращения к памяти - i=1, блоку ветвления - i=2.

На АЛУ выполняются арифметические операции. Блок является комбинационным, результат предоставляет в течение 1 так-

та. Блок ветвления предоставляет результат за 1 такт. В случае ошибки предсказания адреса в следующем после поступления на исполнение такте будет выдан сигнал сброса, который приведёт к очистке уже выбранной очереди команд. Блок обращения к памяти выполняет запрос за 3 такта. Однако если бы было обращение к внешней памяти, то на его работу потребовалось бы значительно больше тактов.

По индивидуальному варианту была сохранена временная диаграмма выполнения команды с адресом **80000008**, приведённая на рисунке 3.

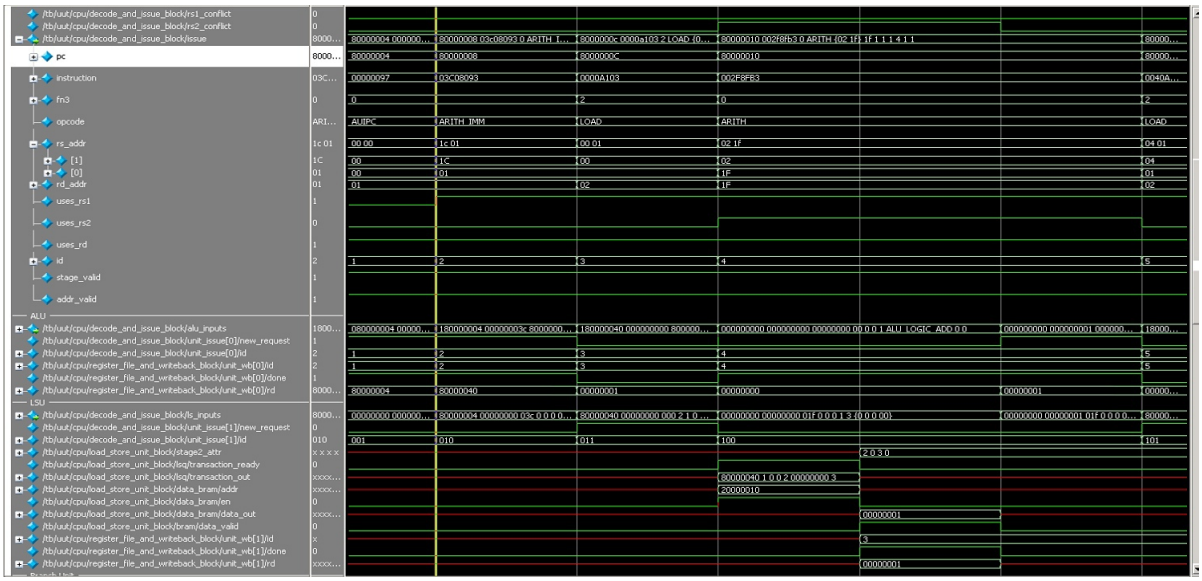

Рисунок 3 – Временная диаграмма выполнения

5 Трассировка программы

На основе полученных знаний была проведена трассировка индивидуальной программы. В ходе выполнения было подтверждено, что **x31 = 29**. Результат трассировки представлен на рисунке 4.

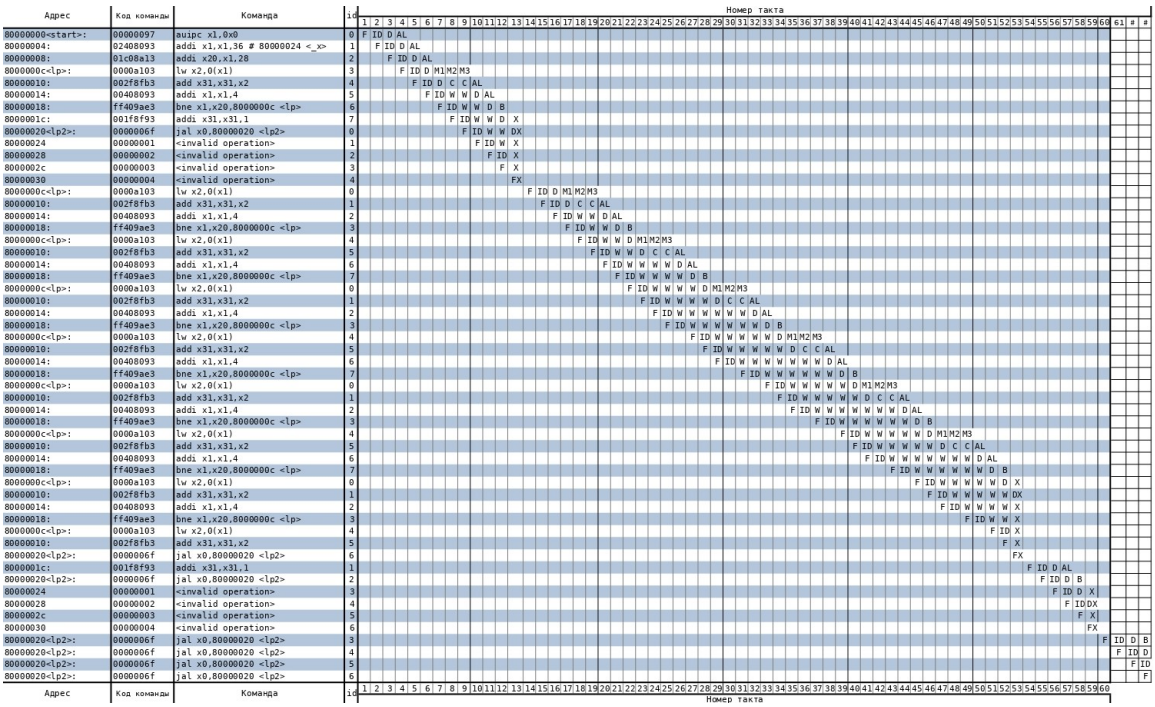

Рисунок 4 – Трассировка программы

Была получена временная диаграмма команды **add x31, x31, x2** с адресом **80000010**, приведённая на рисунке 5.

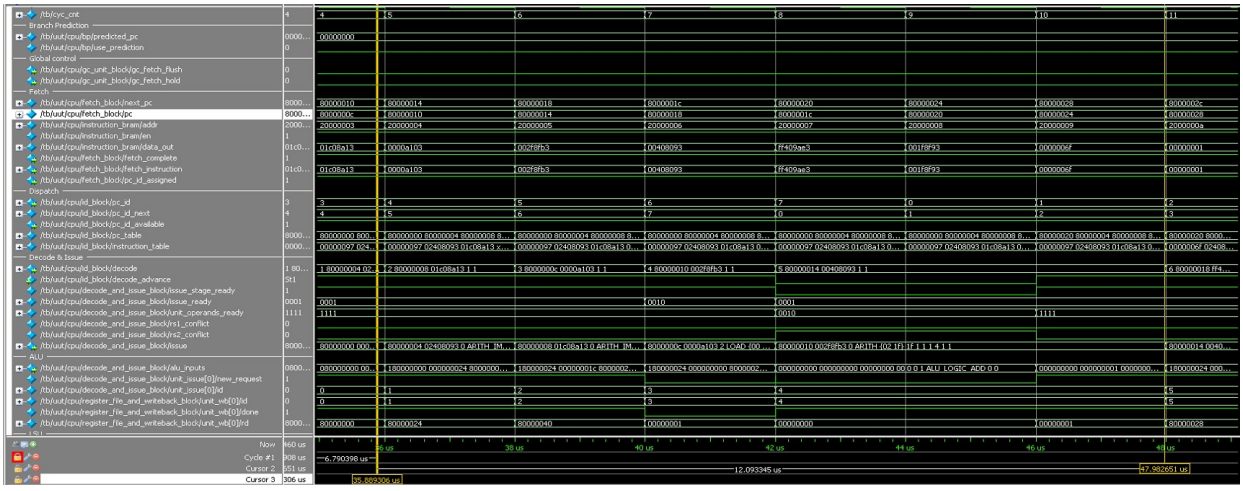

Рисунок 5 – Временная диаграмма обработки команды

В таблице видно, что после выборки данных из памяти в следующей команде случается конфликт. Это происходит из-за того, что в этой команде используется регистр, в который записываются данные из памяти. Чтобы избежать конфликта было решено производить выборку нового аргумента перед началом

очередной итерации цикла и после обработки предыдущего аргумента. Таким образом, к моменту выполнения команды с конфликтом пройдет достаточное количество тактов для выполнения пересылки из памяти. Однако на первой итерации будет также наблюдаться конфликт.

В соответствии с приведёнными выше предложениями по оптимизации была составлена оптимизированная программа, текст которой приведён ниже.

```
        .section .text
        .globl _start;
        len = 8 #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-1) #Адрес последнего элемента

                lw x2, 0(x1)
lp:
        addi x1, x1, elem_sz*enroll
        add x31, x31, x2 #!

                lw x2, 0(x1)
        bne x1, x20, lp
        addi x31, x31, 1
lp2: j lp2

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

По результатам сборки был получен следующий дизассемблерный листинг.

```
SYMBOL TABLE:
80000000 l    d  .text        00000000 .text
80000024 l    d  .data        00000000 .data
00000000 l    df *ABS*        00000000 task.o
00000008 l       *ABS*        00000000 len
00000001 l       *ABS*        00000000 enroll
00000004 l       *ABS*        00000000 elem_sz
80000024 l       .data        00000000 _x
8000000c l       .text        00000000 lp
80000020 l       .text        00000000 lp2
80000000 g       .text        00000000 _start
80000044 g       .data        00000000 _end



Disassembly of section .text:

80000000 <_start>:
80000000:        00000097                    auipc        x1,0x0
80000004:        02408093                    addi        x1,x1,36
 ↪  # 80000024 <_x>
80000008:        01c08a13                    addi        x20,x1,28

8000000c <lp>:
8000000c:        0000a103                    lw        x2,0(x1)
80000010:        002f8fb3                    add        x31,x31,x2
80000014:        00408093                    addi        x1,x1,4
80000018:        ff409ae3
 ↪          bne        x1,x20,8000000c <lp>
8000001c:        001f8f93                    addi        x31,x31,1

80000020 <lp2>:
80000020:        0000006f
 ↪          jal        x0,80000020 <lp2>

Disassembly of section .data:

80000024 <_x>:
80000024:        0001                          c.addi        x0,0
80000026:        0000                          unimp
80000028:        0002                          0x2
8000002a:        0000                          unimp
8000002c:        00000003                    lb        x0,0(x0) #
 ↪  0 <enroll-0x1>
80000030:        0004
 ↪          c.addi4spn        x9,x2,0
80000032:        0000                          unimp
80000034:        0005                          c.addi        x0,1
80000036:        0000                          unimp
80000038:        0006                          0x6
8000003a:        0000                          unimp
```

```
8000003c:          00000007                    0x7
80000040:          0008
 ↪              c.addi4spn          x10,x2,0
```

Была проведена повторная трассировка программы, результат которой отображен на рисунке 6

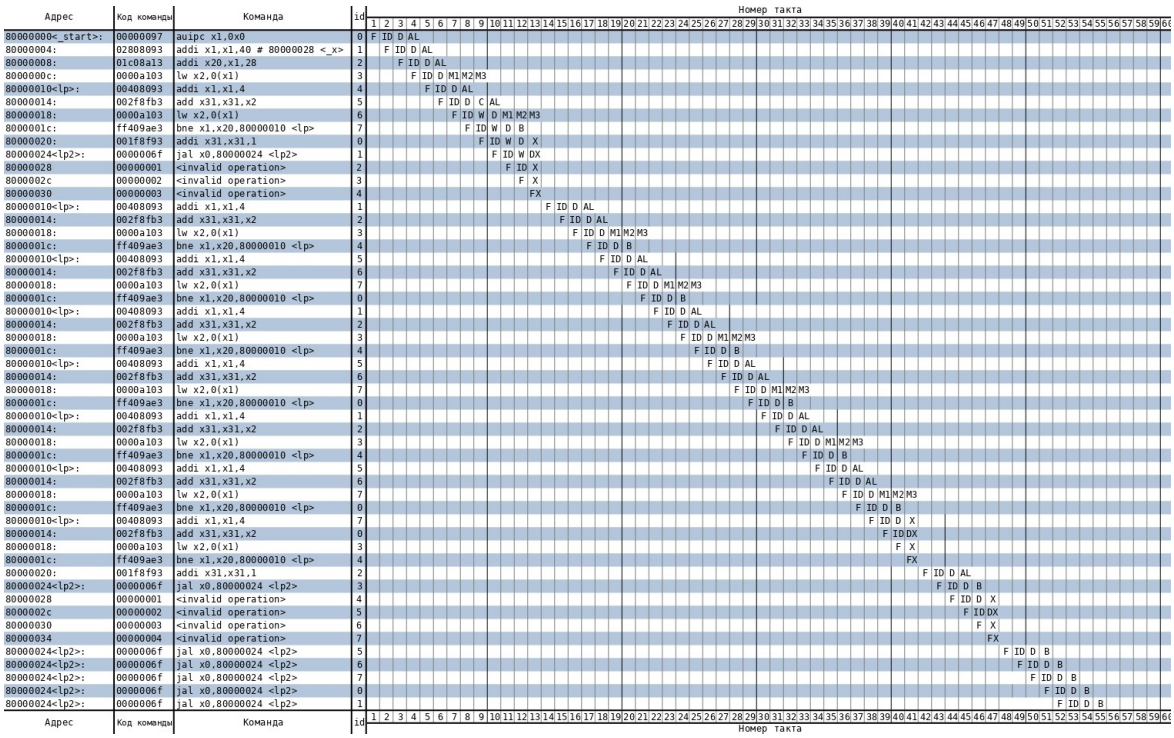

Рисунок 6 – Трассировка оптимизированной программы

Была получена временная диаграмма команды **add x31, x31, x2** с адресом **80000014**, приведённая на рисунке 7.

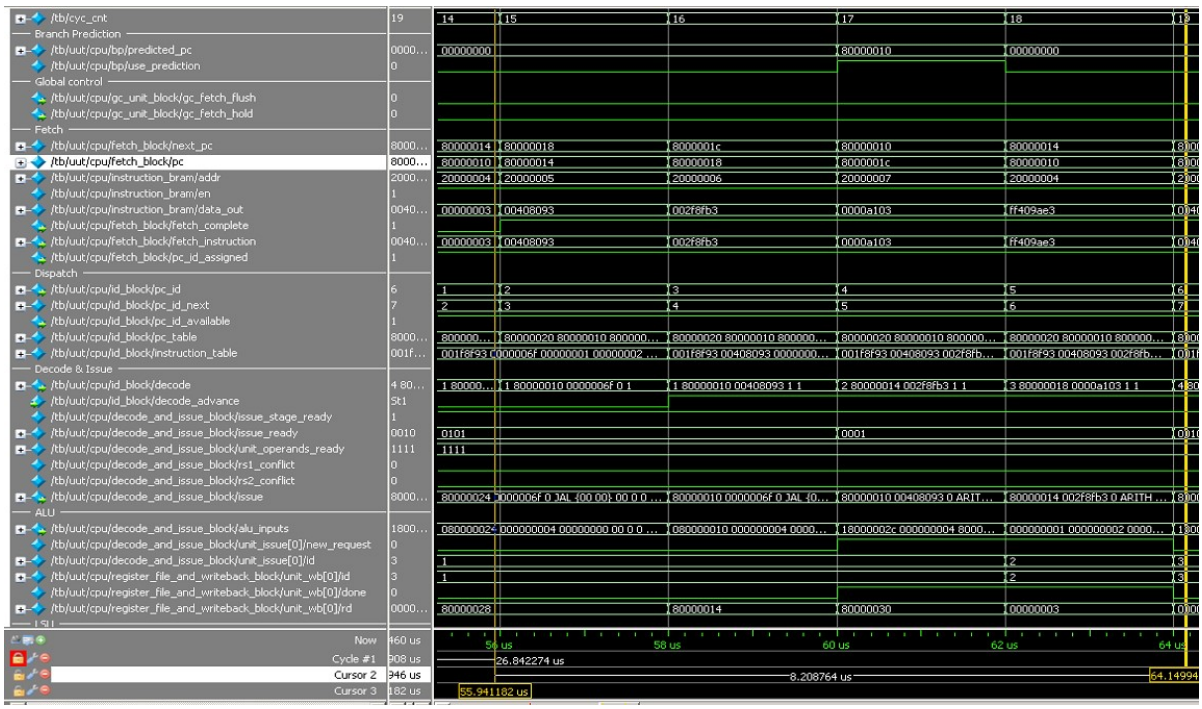

Рисунок 7 – Временная диаграмма обработки команды

Из таблицы видно, что конфликты больше не возникают и команды не находятся в ожидании лишние такты. В результате оптимизации удалось сократить время выполнения основной части программы с 60 тактов до 48, т.е. на 12 тактов.

Выводы

В ходе выполнения лабораторной работы были изучены принципы работы микропроцессорного ядра RISC-V.

Изучены процессы выборки, диспетчеризации, декодирования и выполнения команд.

Проведена трассировка программы, предоставленной по индивидуальному заданию. В ходе трассировки выявлены конфликты в выполнении команд.

Предложены варианты оптимизации программы. В соответствии с ними программа была изменена.

Проведена трассировка оптимизированной программы. Оптимизация позволила избежать конфликтов и увеличить скорость исполнения программы на 12 тактов.