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 № 2 по дисциплине "Архитектура ЭВМ"

Тема Изучение принципов работы микропроцессорного ядра RISC-V

Студент Калашков П.А..

Группа ИУ7-56Б

Оценка (баллы)

Преподаватель Ибрагимов С. В.

Москва — 2022 г.

Содержание

Введение

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.

Цель работы: ознакомление с принципами функционирования, построения и особенностями архитектуры суперскалярных конвейерных микропроцессоров, а также знакомство с принципами проектирования и верификации сложных цифровых устройств с использованием языка описания аппаратуры SystemVerilog и ПЛИС. Для достижения данной цели необходимо выполнить следующие задачи:

- ознакомиться с набором команд RV32I;
- ознакомиться с основными принципами работы ядра Taiga: изучить операции, выполняемые на каждой стадии обработки команд;
- на основе полученных знаний проанализировать ход выполнения программы и оптимизировать ее;

В настоящей лабораторной работе используется синтезируемое описание микропроцессорного ядра Taiga, реализующего систему команд RV32I семейства RISC-V. Данное описание выполнено на языке описания аппаратуры SystemVerilog.

Все задания выполняются в соответствии с вариантом №4.

1 Задание № 1

1.1 Текст программы по индивидуальному варианту

Текст программы по индивидуальному варианту приведён на листинге 1.1

Листинг 1.1 – Текст программы по индивидуальному варианту

```
1     .section .text
2     .globl _start;
3     len = 8
4     enroll = 1
5     elem_sz = 4
6
7     _start:
8     la x1, _x                          // into x1 beginning of mas
9     addi x20, x1, elem_sz*(len-1)      // x20 = x1 + (len - 1) * size
10 lp:
11    lw x2, 0(x1)                       // x2 = x1[0]
12    addi x1, x1, elem_sz*enroll #!     // x1 += elem_sz;
13    add x31, x31, x2                   // x31 += x2;
14    bne x1, x20, lp                    // if (x1 != x20) goto lp;
15    addi x31, x31, 1                   // x31 += 1
16    lp2: j lp2
17
18    .section .data
19    _x:     .4byte 0x1
20    .4byte 0x2
21    .4byte 0x3
22    .4byte 0x4
23    .4byte 0x5
24    .4byte 0x6
25    .4byte 0x7
26    .4byte 0x8
```

1.2 Дизассемблерный листинг кода программы

В результате выполнения компиляции был создан файл с расширением .hex, хранящий содержимое памяти команд и данных. В окне терминала отобразился дизассемблерный листинг, который приведён в листинге 1.1.

```
var4.elf:     file format elf32-littleriscv

SYMBOL TABLE:
80000000 l    d  .text  00000000 .text
80000028 l    d  .data  00000000 .data
00000000 l    df *ABS*  00000000 var4.o
00000008 l       *ABS*  00000000 len
00000001 l       *ABS*  00000000 enroll
00000004 l       *ABS*  00000000 elem_sz
80000028 l       .data  00000000 _x
80000010 l       .text  00000000 lp
80000024 l       .text  00000000 lp2
80000000 g       .text  00000000 _start
80000048 g       .data  00000000 _end



Disassembly of section .text:

80000000 <_start>:
80000000:       00000097                auipc   x1,0x0
80000004:       02808093                addi    x1,x1,40 # 80000028 <_x>
80000008:       01c08a13                addi    x20,x1,28
8000000c:       00000fb3                add     x31,x0,x0

80000010 <lp>:
80000010:       0000a103                lw      x2,0(x1)
80000014:       00408093                addi    x1,x1,4
80000018:       002f8fb3                add     x31,x31,x2
8000001c:       ff409ae3                bne     x1,x20,80000010 <lp>
80000020:       001f8f93                addi    x31,x31,1

80000024 <lp2>:
80000024:       0000006f                jal     x0,80000024 <lp2>

Disassembly of section .data:

80000028 <_x>:
80000028:       0001                    c.addi  x0,0
8000002a:       0000                    unimp
8000002c:       0002                    0x2
8000002e:       0000                    unimp
80000030:       00000003                lb      x0,0(x0) # 0 <enroll-0x1>
80000034:       0004                    c.addi4spn      x9,x2,0
80000036:       0000                    unimp
80000038:       0005                    c.addi  x0,1
8000003a:       0000                    unimp
8000003c:       0006                    0x6
8000003e:       0000                    unimp
80000040:       00000007                0x7
80000044:       0008                    c.addi4spn      x10,x2,0
        ...
riscv64-unknown-elf-objcopy -O binary --reverse-bytes=4 var4.elf var4.bin
xxd -g 4 -c 4 -p var4.bin var4.hex
rm var4.bin var4.o var4.elf
```

Рисунок 1.1 – Скриншот дизассемблированной программы по варианту

1.3 Псевдокод, поясняющий работу программы

В рисунке 1.2 и листинге 1.2 приведён псевдокод, поясняющий работу программы.

```
var4.elf:     file format elf32-littleriscv

SYMBOL TABLE:
80000000 l    d  .text  00000000 .text
80000028 l    d  .data  00000000 .data
00000000 l    df *ABS*  00000000 var4.o
00000008 l       *ABS*  00000000 len
00000001 l       *ABS*  00000000 enroll
00000004 l       *ABS*  00000000 elem_sz
80000028 l       .data  00000000 _x
80000010 l       .text  00000000 lp
80000024 l       .text  00000000 lp2
80000000 g       .text  00000000 _start
80000048 g       .data  00000000 _end



Disassembly of section .text:

80000000 <_start>:
80000000:       00000097                auipc   x1,0x0
80000004:       02808093                addi    x1,x1,40 # 80000028 <_x>
80000008:       01c08a13                addi    x20,x1,28
8000000c:       00000fb3                add     x31,x0,x0

80000010 <lp>:
80000010:       0000a103                lw      x2,0(x1)
80000014:       00408093                addi    x1,x1,4
80000018:       002f8fb3                add     x31,x31,x2
8000001c:       ff409ae3                bne     x1,x20,80000010 <lp>
80000020:       001f8f93                addi    x31,x31,1

80000024 <lp2>:
80000024:       0000006f                jal     x0,80000024 <lp2>

Disassembly of section .data:

80000028 <_x>:
80000028:       0001                    c.addi  x0,0
8000002a:       0000                    unimp
8000002c:       0002                    0x2
8000002e:       0000                    unimp
80000030:       00000003                lb      x0,0(x0) # 0 <enroll-0x1>
80000034:       0004                    c.addi4spn      x9,x2,0
80000036:       0000                    unimp
80000038:       0005                    c.addi  x0,1
8000003a:       0000                    unimp
8000003c:       0006                    0x6
8000003e:       0000                    unimp
80000040:       00000007                0x7
80000044:       0008                    c.addi4spn      x10,x2,0
        ...
riscv64-unknown-elf-objcopy -O binary --reverse-bytes=4 var4.elf var4.bin
xxd -g 4 -c 4 -p var4.bin var4.hex
rm var4.bin var4.o var4.elf
```

Рисунок 1.2 – Скриншот псеводкода на языке C, поясняющего работу программы

Листинг 1.2 – Псевдокод

```
1   1. Инициализация массива _x, состоящего из 8 элементов, значениями от
    1 до 8 включительно. Каждое значение по 4 байта.
2   2. Инициализация параметров алгоритма:
3   len = 8
4   enroll = 1
5   elem_sz = 4
6   3. Установка указателя x1 на начало массива:
7   x1 = _x
8   4. Вычисление указателя на последний элемент массива:
9   x20 = x1 + (len - 1) * size
10  5. Цикл: пока x1 не равно x20
11  5.1. x2 = x1[0]
12  5.2. x1 = x1 + elem_sz * enroll
13  5.3. x31 = x31 + x2
14  6. x31 = x31 + 1
```

Очевидно, что в регистре x31 после выполнения программы должно содержаться значение:

$$x31 = \sum_{i=1}^{7} i + 1 = 29.$$

2 Задание №2

2.1 Выполнение

В результате симуляции, был получен снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 80000018 (1 итерация). Снимок экрана приведён на рисунке 2.1.

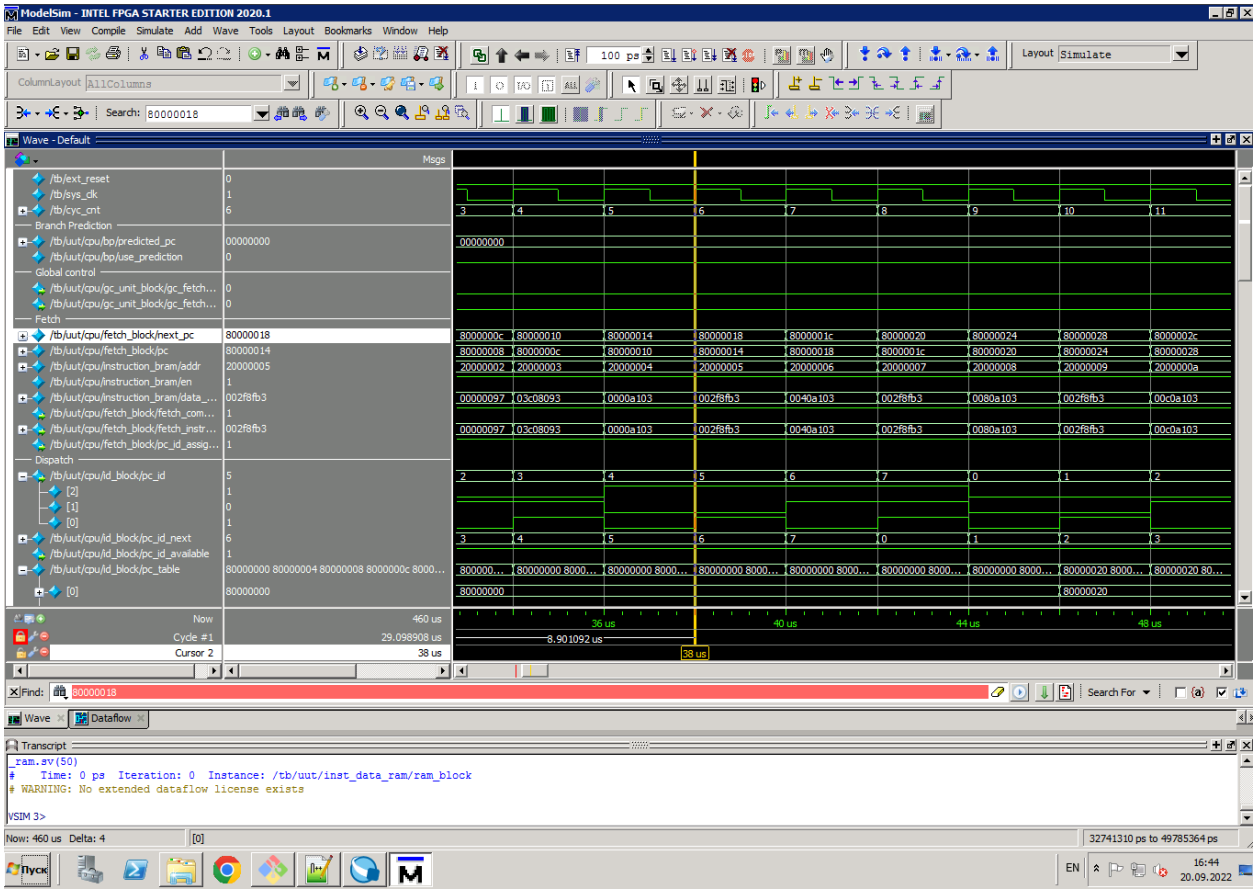

Рисунок 2.1 – Временная диаграмма выполнения стадий выборки и диспетчеризации команды с адресом 80000018 (1 итерация)

Рассмотрим происходящее на рисунке 2.1. Будем называть первым тактом самый левый такт на рисунке.

- На 1 такте `pc` = 80000018. Этот адрес выставляется на ША памяти команд (причём адрес получается с помощью деления на 4, так как память в RISC-V адресуется в байтах, а память команд адресуется блоками по 4 байта). Таким образом, `instruction_bram/addr` = 20000006. Идентификатор команды (`pc_id`) присваивается в момент

начала выборки, то есть одновременно с выставлением адреса команды на ША памяти команд. Кроме того, по фронту `clk`, завершающему такт адрес команды(то есть, значение регистра `pc` на момент выборки) записывается в таблицу `pc_table` по присвоенному идентификатору.

- На 2 такте память команд выдает данные (то есть, код команды), прочитанные по адресу, который был выставлен на ША в предыдущем такте (сигнал `data_out`). Блок выборки выдает эти данные на линию `fetch_instruction`. Устанавливается сигнал `fetch_complete`, подтверждающий выборку и наличие кода команды на линии `fetch_instruction`. В `pc` записывается адрес следующей команды.

- На 3 такте код команды записывается в таблицу `instruction_table` по идентификатору, который был присвоен ранее.

3 Задание №3

3.1 Выполнение

После того, как на выходе блока управления метаинформацией сформирован пакет данных, описывающих очередную инструкцию (то есть, поле decode.valid принимает значение 1) начинается этап декодирования этой инструкции и планирования ее на выполнение. Данный этап выполняется блоком декодирования и планирования на выполнение.

В результате симуляции, был получен снимок экрана, содержащий временную диаграмму выполнения стадии декодирования и планирования на выполнение команды с адресом 80000024 (1 итерация). Снимок экрана приведён на рисунке 3.1.

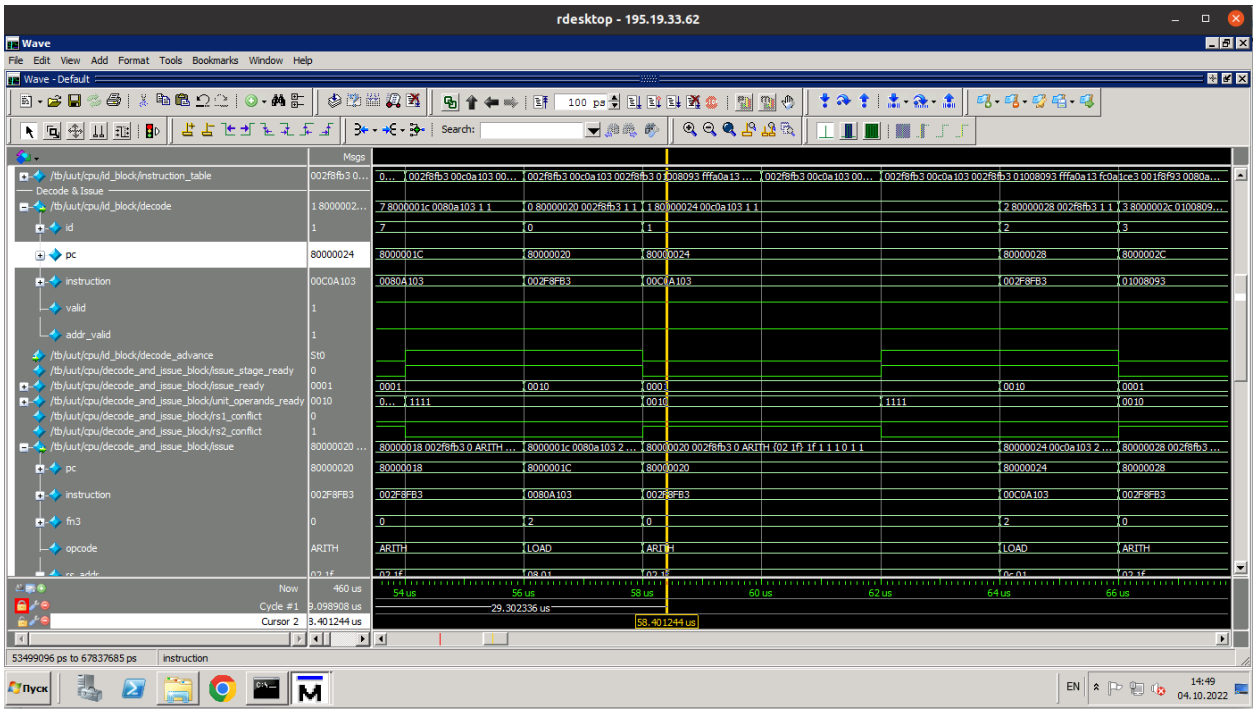

Рисунок 3.1 – Временная диаграмма выполнения стадии декодирования и планирования на выполнение команды с адресом 80000024 (1 итерация)

Рассмотрим происходящее на рисунке 3.1. Будем называть первым тактом самый левый такт на рисунке.

- В такте, предшествующем первому, происходит диспетчеризация команды с id=1. В такте 1 она поступает на вход блока декодирования (это видно по тому признаку, что сигнал decode.valid установлен).

- В такте 1 происходит декодирование команды. Команда не может быть запланирована на выполнение, так как занято устройство LSU (блок управления памятью, выполняющий команду), поэтому проходит 2 такта ожидания. При освобождении блока управления памятью устанавливается сигнал `decode_advance` для передачи блоку управления метаинформацией указания выдать очередную команду для декодирования в следующем такте.

- В начале такта 4 выдаются сигналы для исполнительных блоков. На основании сигнала `issue` формируются сигналы `rs1_conflict`, `rs2_conflict` равные 0, так как конфликта по регистрам нет. Отсутствие конфликта дает возможность точно выполнить данную команду в этом такте, соответственно сигнал `decode_advance` должен быть установлен в 1.

4 Задание №4

4.1 Выполнение

После того, как команда запланирована для выполнения и нет конфликта по регистрам, начинается этап выполнения команды каким-либо исполнительным блоком. Однако, в начале этапа выполнения происходит чтение исходных регистров команды, информация о которых содержится в сигнале issue. Чтение регистрового файла выполняется комбинационно, то есть, данные на выходе регистрового файла выдаются в том же такте, что и сигнал issue.

В результате симуляции, был получен снимок экрана, содержащий временную диаграмму выполнения стадии выполнения команды с адресом 8000000c (1 итерация). Снимок экрана приведён на рисунке 4.1.

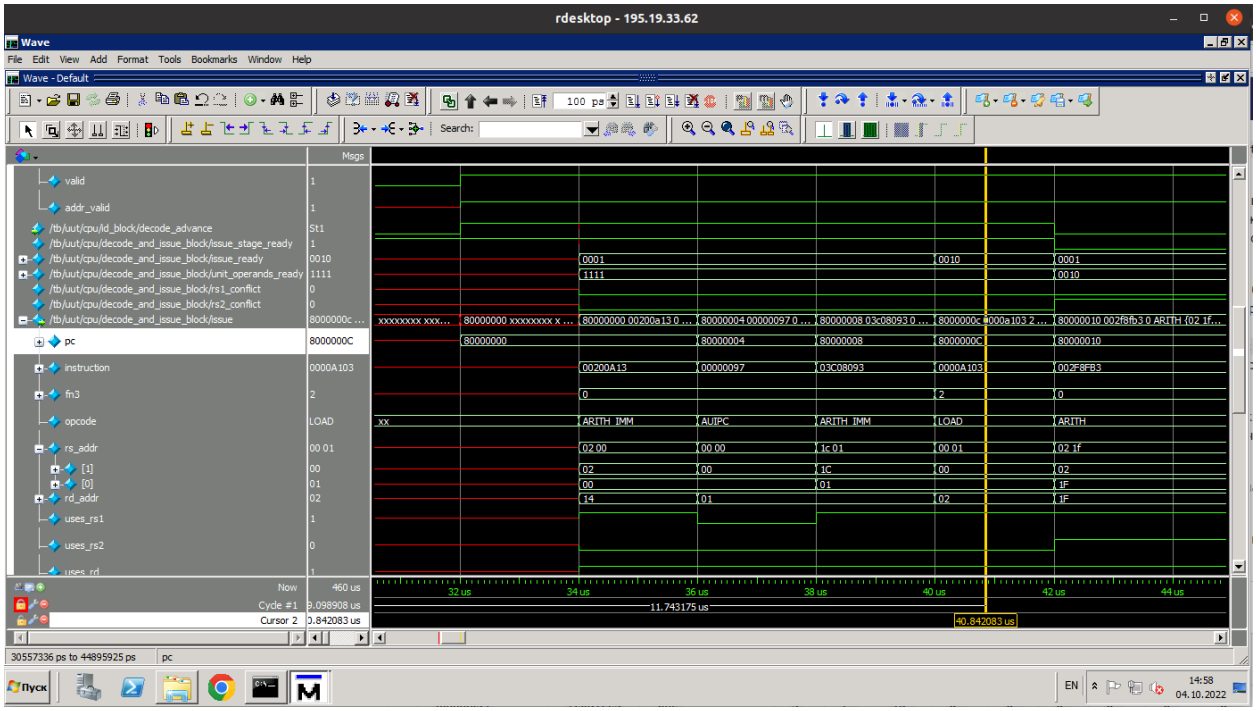

Рисунок 4.1 – Временная диаграмма выполнения стадии декодирования и планирования на выполнение команды с адресом 8000000c (1 итерация)

Рассмотрим происходящее на рисунке 4.1. Видно, что команда будет обработана блоком обращения к памяти.

- В момент планирования на выполнение новой команды (то есть, в момент выставления сигнала `unit_issue[1].new_request`) происходит

формирование на основе сигнала `ls_inputs` атрибутов транзакции доступа к памяти (адрес, тип, размер и пр.) и их запись в очередь транзакций.

- В следующем такте характеристики транзакции становятся доступны на выходе очереди транзакций (сигнал `transaction_out`), что подтверждается сигналом `transaction_-ready`. Выполняется дешифрирование адреса и определение вида памяти к которой происходит доступ. Готовность памяти данных (в нашем случае, готовность памяти имеется всегда, так как время доступа к памяти всегда составляется 1 такт) дает возможность сформировать запрос к памяти, то есть выставить на ША адрес, соответствующий характеристикам транзакции. Адрес формируется комбинационно. Так как блок рассчитан на работу с памятью с неизвестной заранее задержкой, то характеристики запроса к памяти записываются в очередь запросов к памяти.

- В следующем такте память данных выставляет на ШД прочитанные данные (сигнал `data_out_b`) и сигнал готовности данных `data_valid`. Блок фиксирует выполнение команды выставлением сигнала `unit_wb[1].done` (формируется комбинационно по сигналу `data_valid`), `unit_wb[1].id` (берется из выхода очереди запросов к памяти) и `unit_wb[1].rd` (берется с ШД памяти данных). В этом же такте происходит запись в целевой регистр.

Таким образом видно, что выполнение команды доступа к памяти занимает минимум 3 такта.

5 Задание №5

В заданиях 2-4 симуляция проводилась на программе из примера. В задании №5 симуляция проводится на программе из индивидуального варианта, которая описана в 1 задании.

5.1 Сравнение полученного теоретически значения регистра x31 и полученного в результате симуляции

Для проверки сравним значение регистра x31, вычисленного теоретически, с полученным в результате симуляции. Теоретически было получено значение 29. Результат симуляции представлен на рисунке 5.1. Видим, что значения совпадают, так как $1D_{16} = 29_{10}$

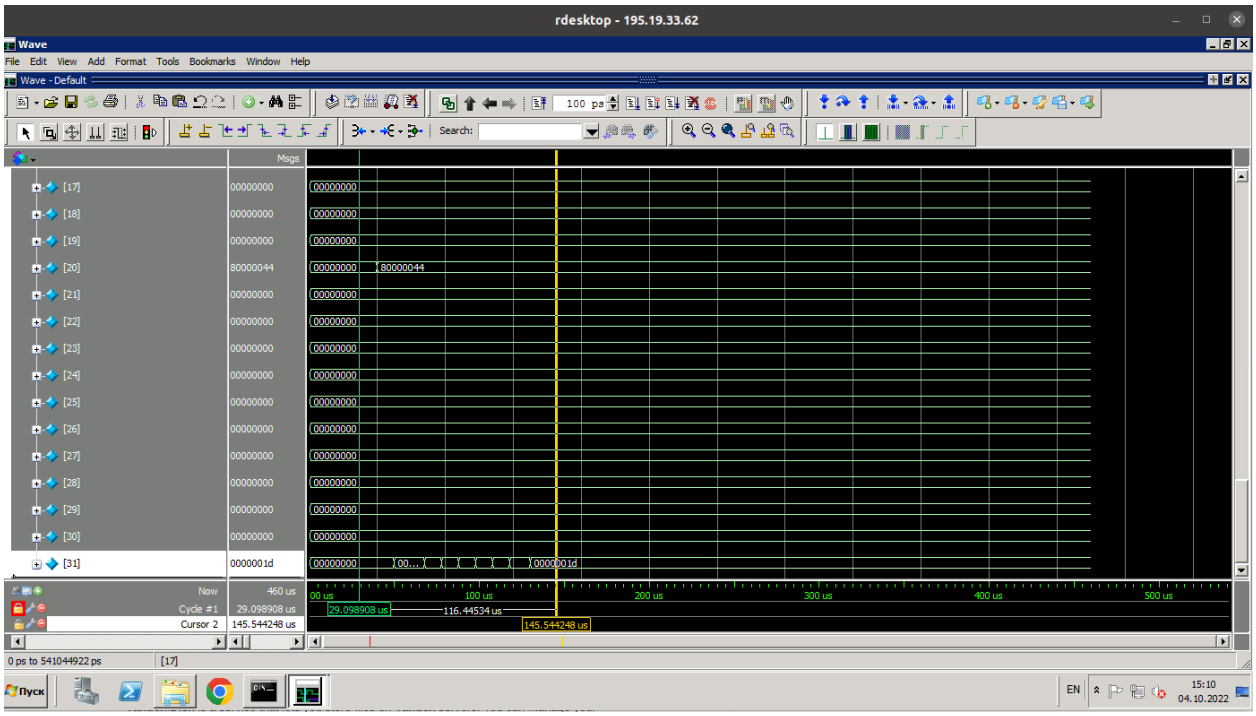

Рисунок 5.1 – Значение регистра x31, полученное в результате симуляции

5.2 Временные диаграммы сигналов, соответствующих всем стадиям выполнения команды 80000010

В тексте программы 1.1 символом `#!` обозначена команда `addi x1, x1, elem_sz * enroll`. Из дизассемблерного листинга кода программы 1.1 очевидно, что эта команда имеет адрес `80000010`.

На рисунках 5.2 и 5.3 приведены временные диаграммы сигналов, соответствующих всем стадиям выполнения команды с адресом 80000010.

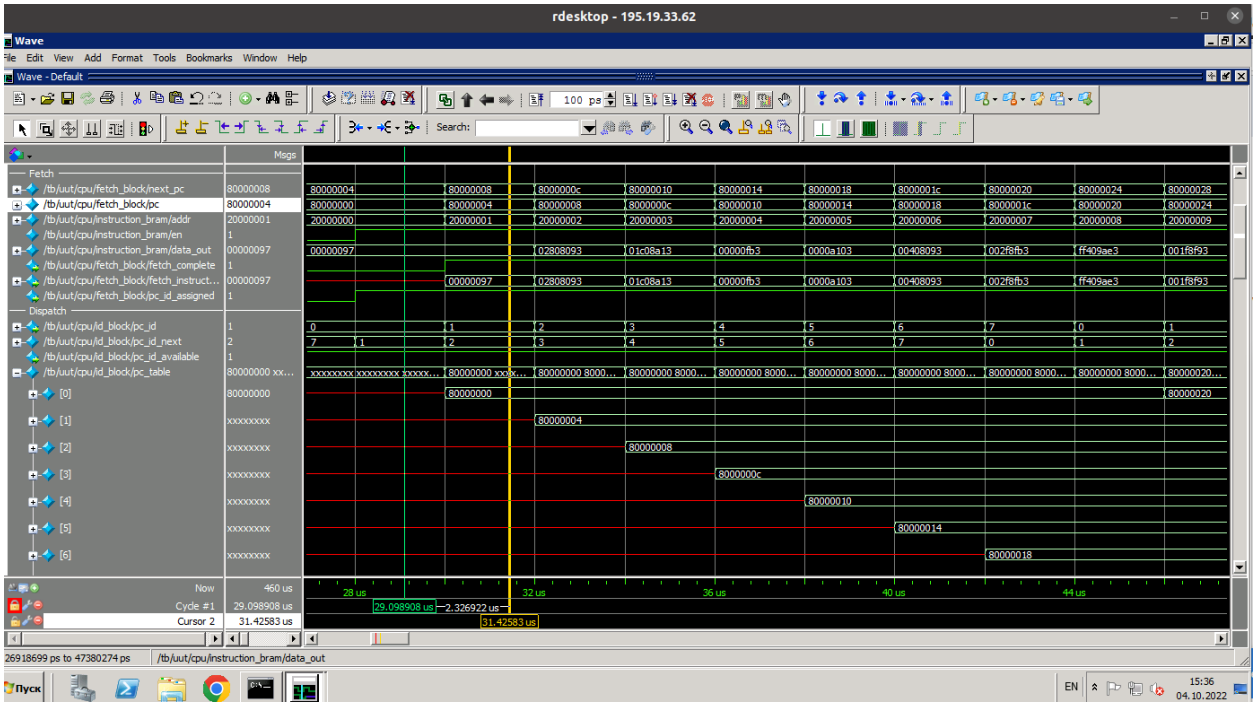

Рисунок 5.2 – Временная диаграмма выполнения стадий выборки и диспетчеризации команды с адресом 80000010

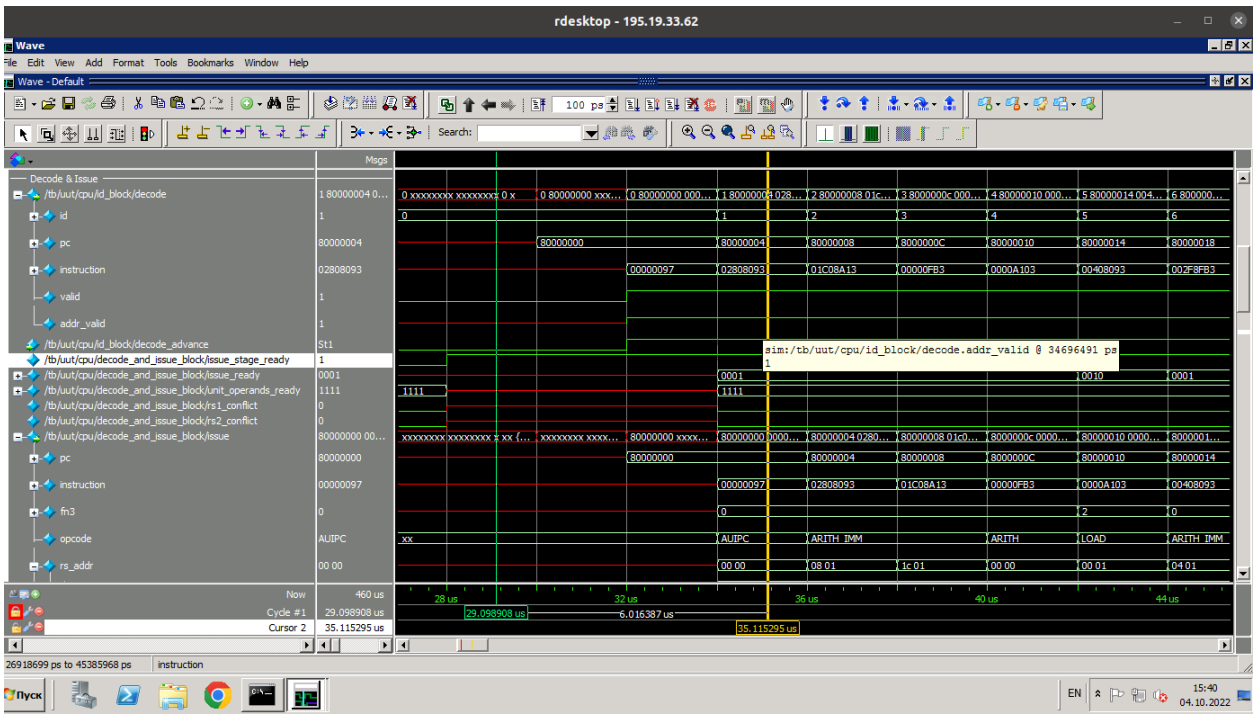

Рисунок 5.3 – Временная диаграмма выполнения стадий декодирования, планирования на выполнение и выполнения команды с адресом 80000010

5.3 Трасса выполнения программы

В результате анализа диаграммы была заполнена трасса выполнения программы. Трасса выполнения показана на рисунке 5.4

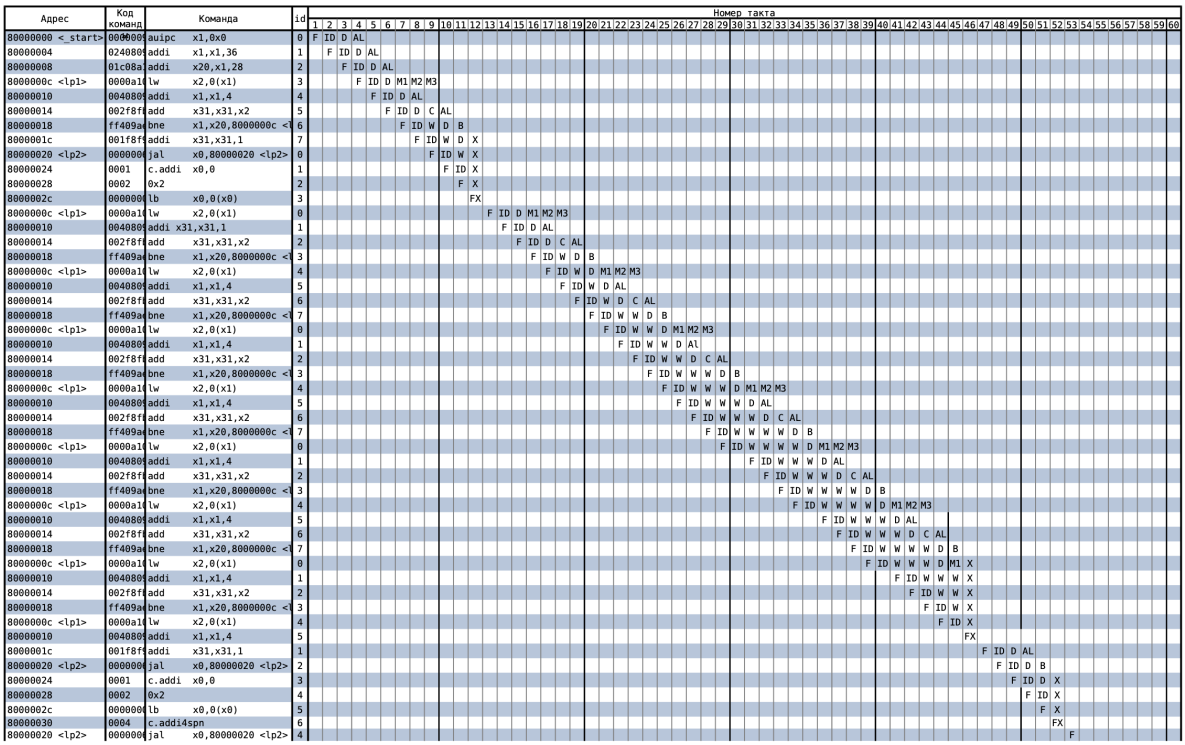

Рисунок 5.4 – Трасса выполнения программы

5.4 Оптимизация программы

Из анализа программы видно, что отсутствует возможность сократить время выполнения путем перестановки команд для ликвидации конфликтов и ожиданий декодера.

Вывод

Выполняемая программа имеет оптимальный порядок команд и не нуждается в оптимизации.

Заключение

В данной лабораторной работе было проведено ознакомление с архитектурой ядра Taiga, а именно с порядком работы вычислительного конвейера: изучены команды RV32I, рассмотрены действия, выполняемые на каждой стадии конвейера, и данные, передаваемые между ними. После ознакомления с теоретической стороной вопроса, был выполнен разбор этапов выполнения программы на симуляции процессора с набором инструкций RV32I. После ее анализа были сделаны выводы, что оптимизация не требуется. В итоге, теоретические знания о порядке исполнения программ на процессорах с RISC архитектурой были закреплены на практике. Таким образом все поставленные задачи решены, основная цель работы достигнута.