

Оглавление

Введение	3
1 Задание № 1	4
1.1 Текст программы по индивидуальному варианту	4
1.2 Дизассемблерный листинг кода программы	4
1.3 Псевдокод, поясняющий работу программы	6
2 Задание №2	7
2.1 Выполнение	7
3 Задание №3	9
3.1 Выполнение	9
4 Задание №4	11
4.1 Выполнение	11
5 Задание №5	13
5.1 Сравнение полученного теоретически значения регистра x31 и полученного в результате симуляции	13
5.2 Временные диаграммы сигналов, соответствующих всем стадиям выполнения команды 80000010	13
5.3 Трасса выполнения программы	15
5.4 Оптимизация программы	15
Вывод	15
Заключение	16

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:
20         .4 byte 0x1
21         .4 byte 0x2
22         .4 byte 0x3
23         .4 byte 0x4
24         .4 byte 0x5
25         .4 byte 0x6
26         .4 byte 0x7
27         .4 byte 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 1.2.

Листинг 1.2: Дизассемблерный листинг кода программы

```

1 SYMBOL TABLE:
2 80000000 I d .text 00000000 .text
3 80000024 I d .data 00000000 .data
4 00000000 I df *ABS* 00000000 main.o
5 00000008 I *ABS* 00000000 len
6 00000001 I *ABS* 00000000 enroll
7 00000004 I *ABS* 00000000 elem_sz
8 80000024 I .data 00000000 _x
9 8000000c I .text 00000000 lp
10 80000020 I .text 00000000 lp2
11 80000000 g .text 00000000 _start
12 80000044 g .data 00000000 _end

13
14 Disassembly of section .text:
15
16 80000000 <_start>:
17 80000000: 00000097 auipc x1,0x0
18 80000004: 02408093 addi x1,x1,36 # 80000024 <_x>
19 80000008: 01c08a13 addi x20,x1,28

20
21 8000000c <lp>:
22 8000000c: 0000a103 lw x2,0(x1)
23 80000010: 00408093 addi x1,x1,4
24 80000014: 002f8fb3 add x31,x31,x2
25 80000018: ff409ae3 bne x1,x20,8000000c <lp>
26 8000001c: 001f8f93 addi x31,x31,1

27
28 80000020 <lp2>:
29 80000020: 0000006f jal x0,80000020 <lp2>

30
31 Disassembly of section .data:
32
33 80000024 <_x>:
34 80000024: 0001 c.addi x0,0
35 80000026: 0000 unimp
36 80000028: 0002 0x2
37 8000002a: 0000 unimp
38 8000002c: 00000003 lb x0,0(x0) # 0 <enroll-0x1>
39 80000030: 0004 c.addi4spn x9,x2,0
40 80000032: 0000 unimp
41 80000034: 0005 c.addi x0,1
42 80000036: 0000 unimp
43 80000038: 0006 0x6
44 8000003a: 0000 unimp
45 8000003c: 00000007 0x7
46 80000040: 0008 c.addi4spn x10,x2,0

```

1.3 Псевдокод, поясняющий работу программы

В листинге 1.3 приведён псевдокод, поясняющий работу программы.

Листинг 1.3: Псевдокод

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

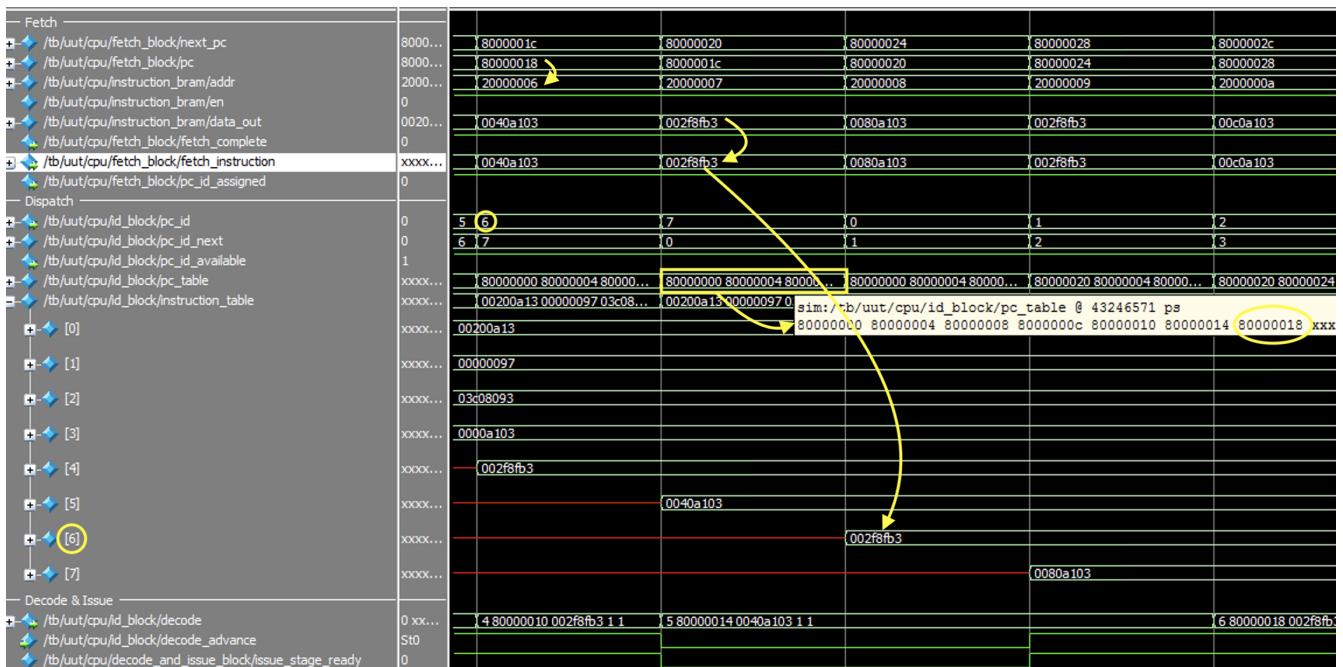


Рис. 2.1: Временная диаграмма выполнения стадий выборки и диспетчеризации команды с адресом 80000018 (1 итерация)

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 адреса 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.

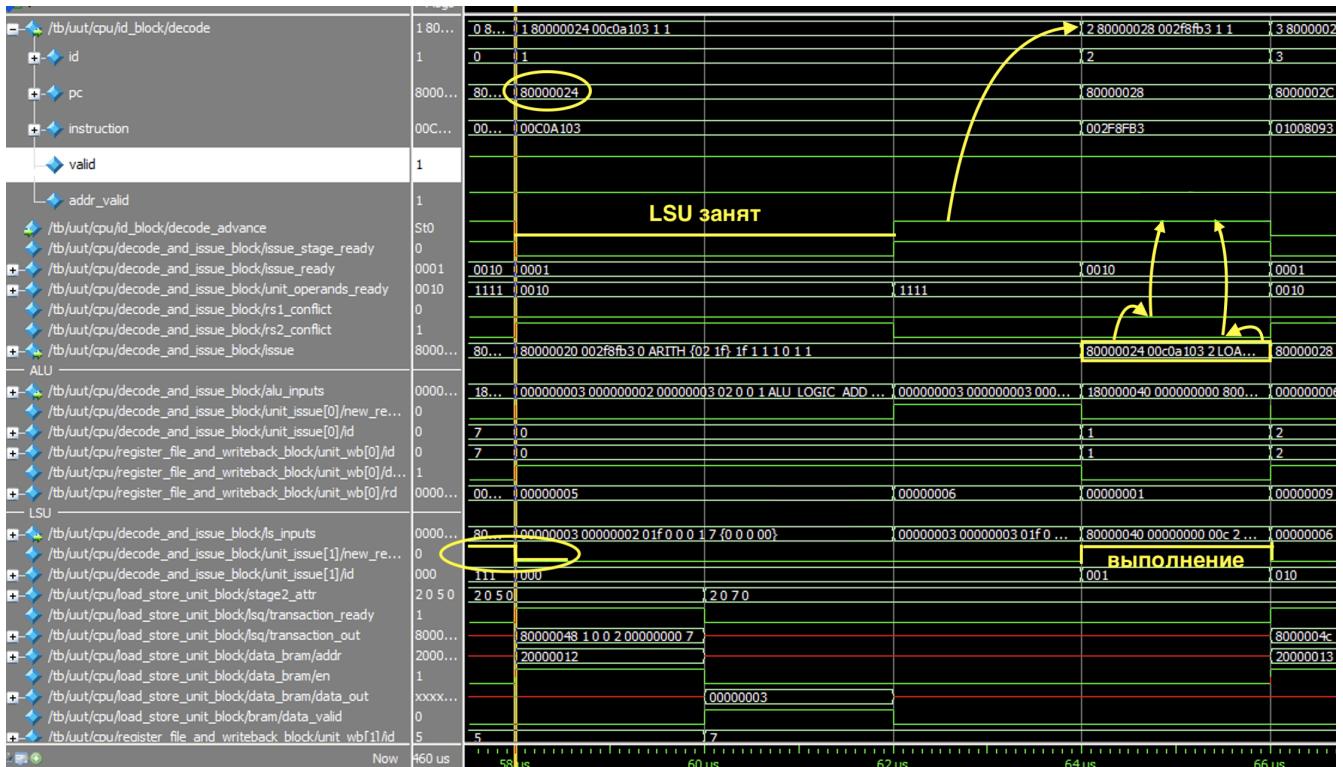


Рис. 3.1: Временная диаграмма выполнения стадии декодирования и планирования на выполнение команды с адресом 80000024 (1 итерация)

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.

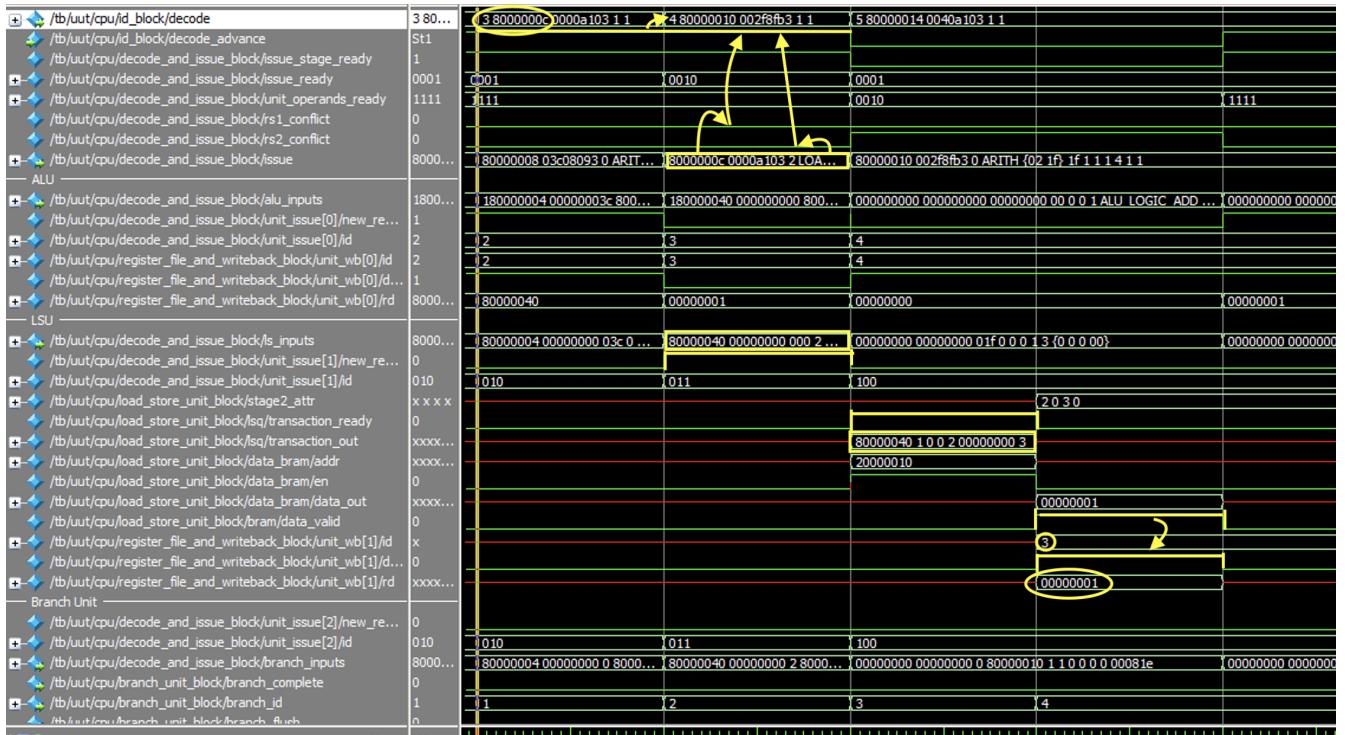


Рис. 4.1: Временная диаграмма выполнения стадии декодирования и планирования на выполнение команды с адресом 8000000c (1 итерация)

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 выставле-

ния 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 проходит 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 $1D_{16} = 29_{10}$



Рис. 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 1.2 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.

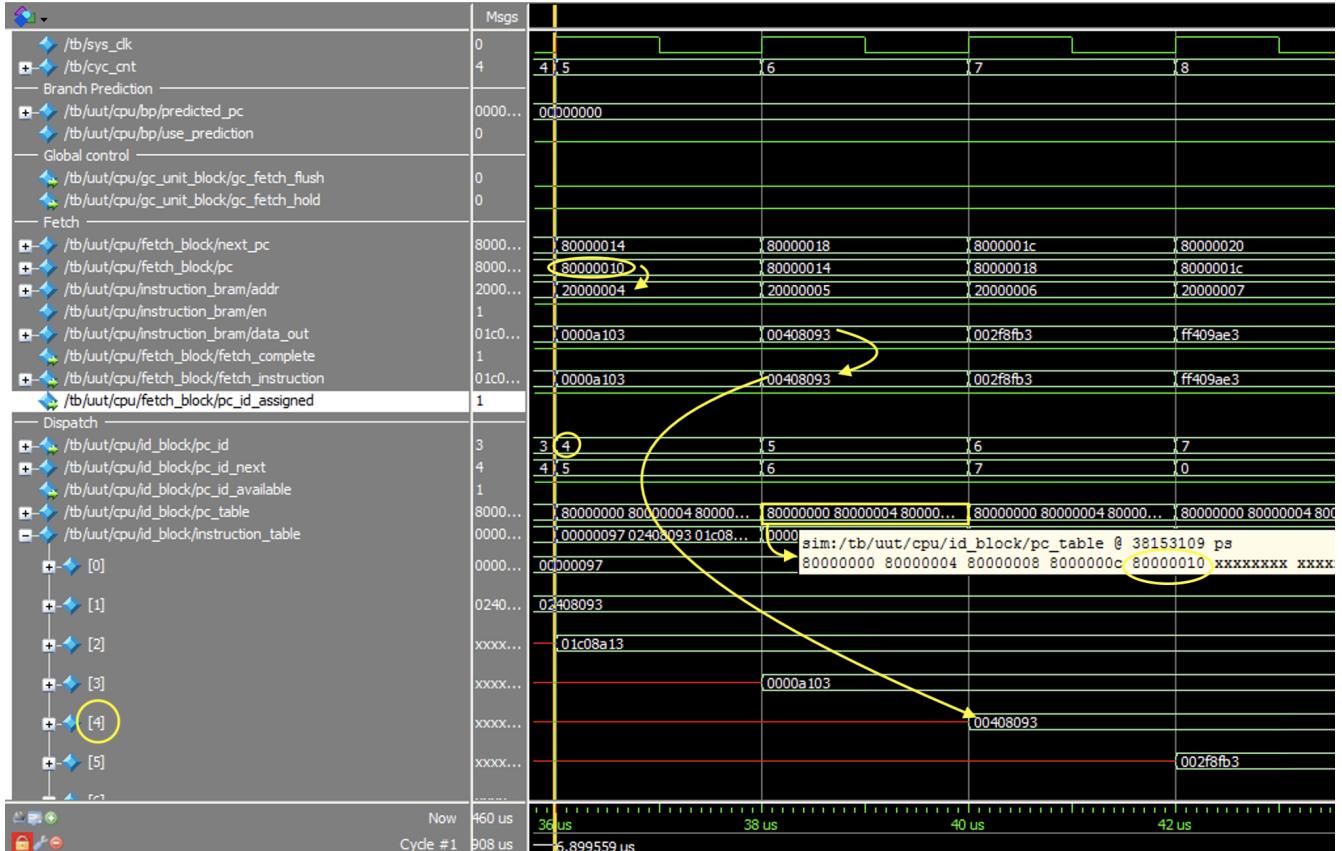


Рис. 5.2: Временная диаграмма выполнения стадий выборки и диспетчеризации команды с адресом 80000010

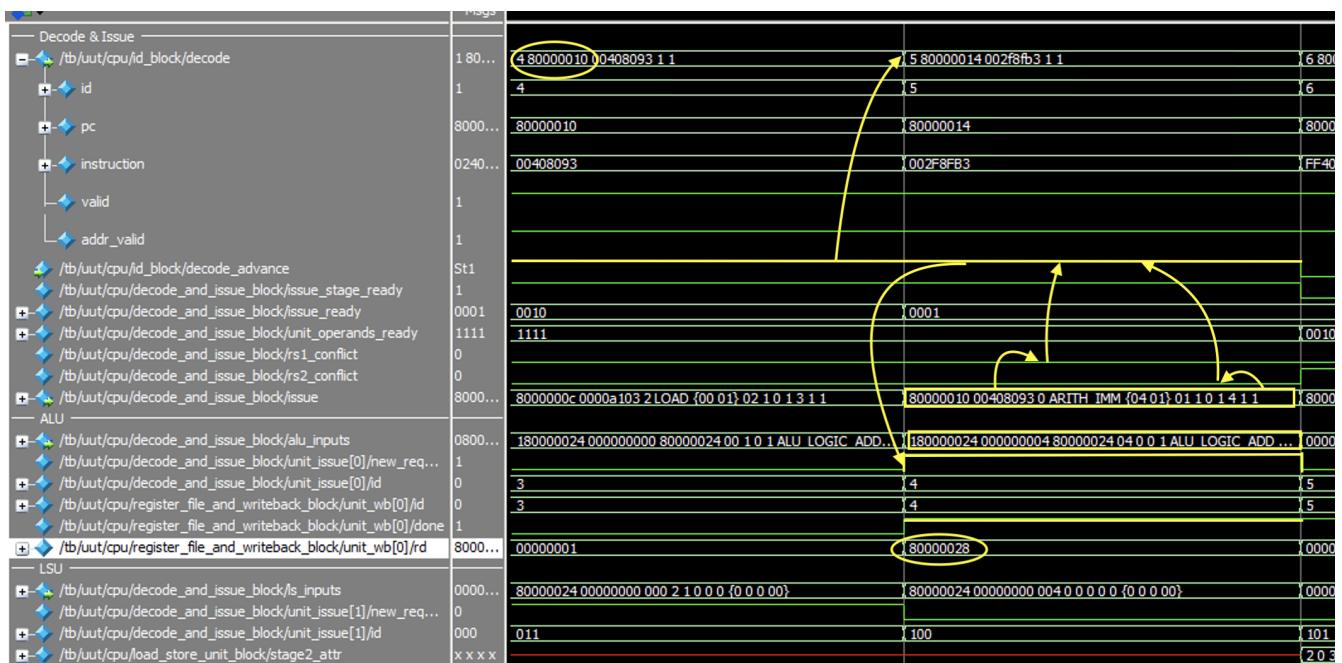


Рис. 5.3: Временная диаграмма выполнения стадий декодирования, планирования на выполнение и выполнения команды с адресом 80000010

5.3 Трасса выполнения программы

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

Рис. 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.