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»

Отчёт по лабораторной работе №2 по курсу «Архитектура ЭВМ»

Тема Изучение принципов работы микропроцессорного ядра RISC-V

Студент Ляпина Н.В.

Группа ИУ7-52Б

Преподаватель Дубровин Е.Н.

Москва — 2022 г.

Оглавление

Введение

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.

Для достижения поставленной цели необходимо решить 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.

1 Аналитическая часть

1.1 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.

1.2 Микроархитектура

Теперь перейдем от рассмотрения абстрактной архитектуры системы команд к рассмотрению микроархитектуры ядра Taiga.

Будем рассматривать систему, состоящую из вычислительного ядра Taiga и локальной памяти, реализованной с помощью блочной памяти ПЛИС. Данная память является статической, синхронной и двухпортовой. Один и тот же блок памяти используется для реализации как памяти команд (ПК), так и памяти данных (ПД). Таким образом команды и данные находятся в едином адресном пространстве. Дешифратор адресов настроен таким образом, что блок памяти ПЛИС отображается в адресное пространство RISC-V с адреса 0x80000000, как мы это видели из рассмотрения примера выше.

Благодаря двухпортовой организации имеется возможность чтения и записи одновременно и команд и данных. Кроме того, блочная память

ПЛИС имеет фиксированную задержку доступа в 1 такт. Таким образом, в нашей системе не будут возникать задержки доступа к памяти, в связи с чем отпадает необходимость в кеш-памяти.

Taiga является конвейерным микропроцессором с элементами суперскалярности. При конвейерной организации микропроцессора различные команды одновременно проходят различные стадии своей обработки. Конвейер Taiga насчитывает 4 стадии. В скобках приведены сокращенные обозначения стадий.

1) выборка(F). Стадия, на которой команда извлекается из ПК. Выполняется в блоке выборки;
2) диспетчеризация (ID). Стадия, на которой происходит запись команды в очередь команд для декодирования. Выполняется в блоке управления метаданными;
3) декодирование и планирование на выполнение (D). Стадия на которой происходит определение типа и полей команды и определение вычислительного блока, способного ее исполнить. Выполняется в блоке декодирования и планирования на выполнение;
4) выполнение (AL, M1..M3, в зависимости от исполнительного блока). Стадия, на которой команда передается в блок выполнения.

"Ширина"конвейера Taiga (то есть, количество команд, которые одновременно могут находиться на одной и той же стадии конвейера) равна 1 для всех стадий, кроме стадии выполнения. В лучшем случае, каждая стадия конвейера (кроме выполнения) выполняется за один такт.

В состав рассматриваемой конфигурации Taiga входит 3 блока выполнения команд: Арифметико-логическое устройство (АЛУ), блок доступа к памяти (LSU) и блок ветвлений. АЛУ и блок ветвлений выполняют команды за 1 такт, LSU – минимум за 3. Таким образом, возможна ситуация когда команда обращения к памяти выполняется одновременно с арифметической командой.

На рисунке 1.1 показана упрощенная и укрупненная структурная схема ядра Taiga.

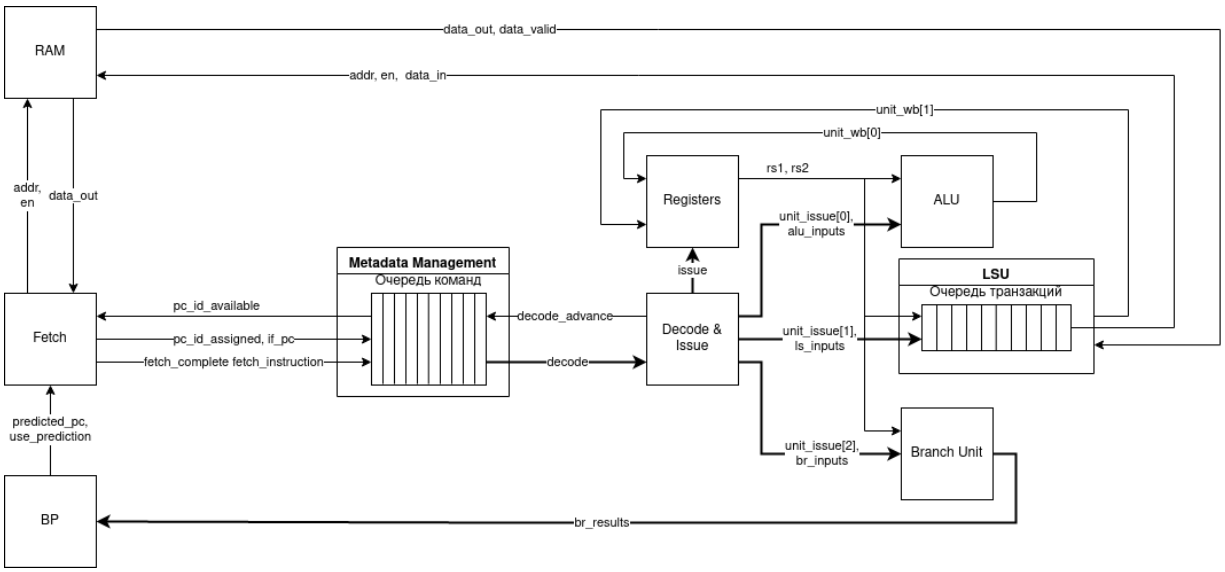

Рисунок 1.1 – Схема ядра Taiga

2 Ход выполнения работы

Все задания лабораторной работы выполнялись по варианту 10.

2.1 Задание 1

В листинге 2.1 представлен код программы по индивидуальному варианту. На рисунке 2.1 представлен дизассемблированный код программы. А в листинге 2.2 представлен псевдокод программы по индивидуальному варианту на языке C.

Листинг 2.1 – Код программы по индивидуальному варианту

```
.section .text
        .globl _start;
        len = 8 #Размер массива
        enroll = 4 #Количество обрабатываемых элементов за одну итерацию
    elem_sz = 4 #Размер одного элемента массива

_start:
        addi x20, x0, len/enroll
        la x1, _x
        add x31, x0, x0
lp:
        lw x2, 0(x1)
        lw x3, 4(x1) #!
        add x31, x31, x2
        add x31, x31, x3
        lw x4, 8(x1)
        lw x5, 12(x1)
        add x31, x31, x4
        add x31, x31, x5
        addi x1, x1, elem_sz*enroll
        addi x20, x20, -1
        bne x20, x0, 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
```

```
Disassembly of section .text:

80000000 <_start>:
80000000:       00200a13                addi    x20,x0,2
80000004:       00000097                auipc   x1,0x0
80000008:       04008093                addi    x1,x1,64 # 80000044 <_x>
8000000c:       00000fb3                add     x31,x0,x0

80000010 <lp>:
80000010:       0000a103                lw      x2,0(x1)
80000014:       0040a183                lw      x3,4(x1)
80000018:       002f8fb3                add     x31,x31,x2
8000001c:       003f8fb3                add     x31,x31,x3
80000020:       0080a203                lw      x4,8(x1)
80000024:       00c0a283                lw      x5,12(x1)
80000028:       004f8fb3                add     x31,x31,x4
8000002c:       005f8fb3                add     x31,x31,x5
80000030:       01008093                addi    x1,x1,16
80000034:       fffa0a13                addi    x20,x20,-1
80000038:       fc0a1ce3                bne     x20,x0,80000010 <lp>
8000003c:       001f8f93                addi    x31,x31,1

80000040 <lp2>:
80000040:       0000006f                jal     x0,80000040 <lp2>

Disassembly of section .data:

80000044 <_x>:
80000044:       0001                    c.addi  x0,0
80000046:       0000                    unimp
80000048:       0002                    0x2
8000004a:       0000                    unimp
8000004c:       00000003                lb      x0,0(x0) # 0 <elem_sz-0x4>
80000050:       0004                    c.addi4spn      x9,x2,0
80000052:       0000                    unimp
80000054:       0005                    c.addi  x0,1
80000056:       0000                    unimp
80000058:       0006                    0x6
8000005a:       0000                    unimp
8000005c:       00000007                0x7
80000060:       0008                    c.addi4spn      x10,x2,0
        ...
riscv64-unknown-elf-objcopy -O binary --reverse-bytes=4 task_1.elf task_1.bin
xxd -g 4 -c 4 -p task_1.bin task_1.hex
rm task_1.bin task_1.elf task_1.o
```

Рисунок 2.1 – Дизассемблированный код программы

Листинг 2.2 – Псевдокод на языке Си

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

```
11      int x31 += 0;
12
13      do
14      {
15          int x2 = x1[0];
16          int x3 = x1[4];
17          x31 += x2;
18          x31 += x3;
19          int x4 = x1[8];
20          int x5 = x1[12];
21          x31 += x4;
22          x31 += x5;
23          x1 += elem_sz*enroll;
24          int x20 -= 1;
25      }
26      while (x20 != 0);
27      x31 += 1;
28
29      while (1) {};
30  }
```

После выполнения программы в $x31$ будет содержаться сумма всех элементов массива + 1. Или $\sum_0^7 x[i] + 1 = 37$.

2.2 Задание 2

В результате симуляции был получен снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 8000000c (1-я итерация). Результат представлен на рисунке 2.2.

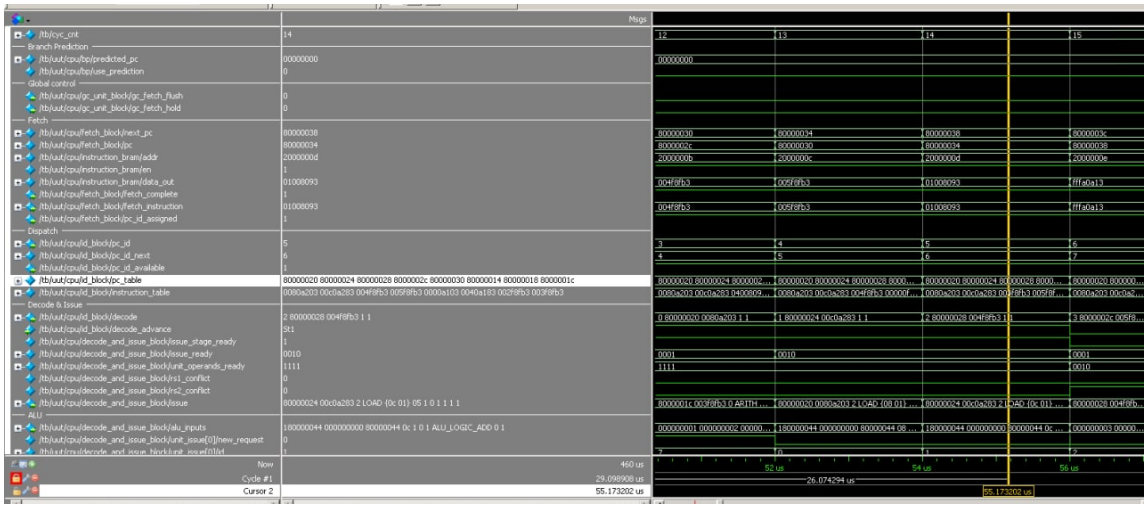

Рисунок 2.2 – Временная диаграмма выполнения стадий выборки и диспетчеризации команды с адресом 80000030 (1-я итерация)

В такте 14 fetch_complete = 1, следовательно, в предыдущем такте произошла выборка команды 80000030.

2.3 Задание 3

В результате симуляции был получен снимок экрана, содержащий временную диаграмму выполнения стадий декодирования и планирования команды с адресом 80000010 (2-я итерация). Результат представлен на рисунке 2.3.

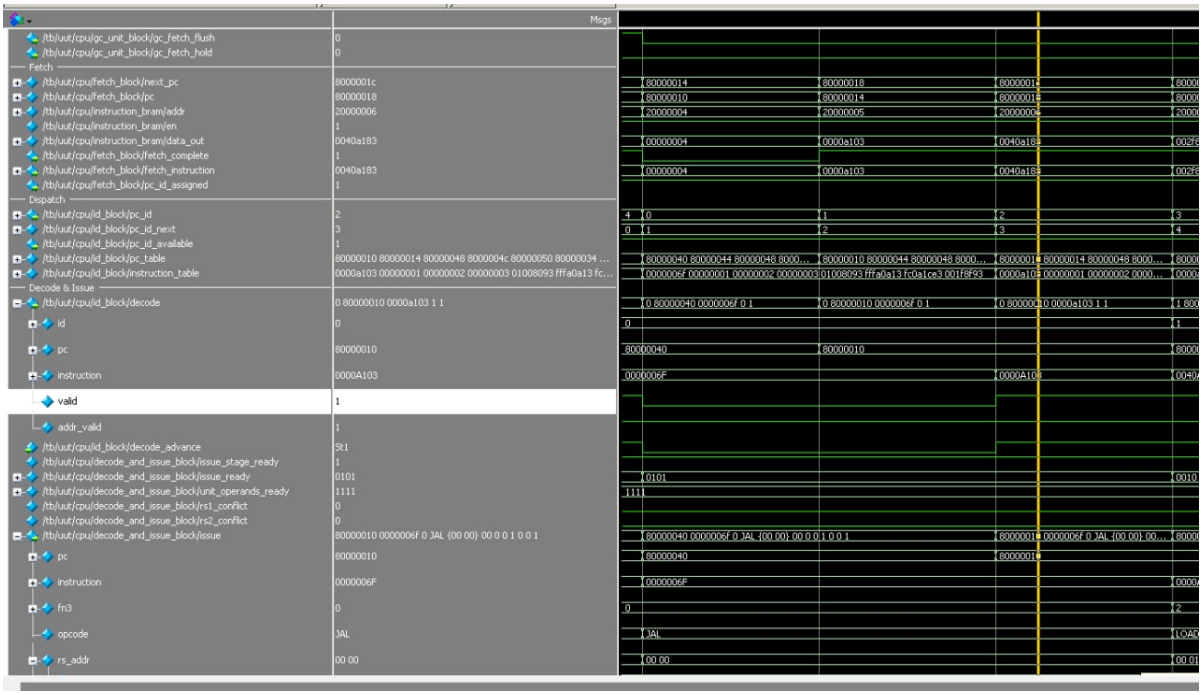

Рисунок 2.3 – Временная диаграмма выполнения стадий декодирования и планирования команды с адресом 80000010

На 24 такте декодируется команда 80000010 с id=0. На 25 такте планируется ее выполнение.

2.4 Задание 4

В результате симуляции был получен снимок экрана, содержащий временную диаграмму выполнения стадии выполнения команды с адресом 80000024. Результат представлен на рисунке 2.4.

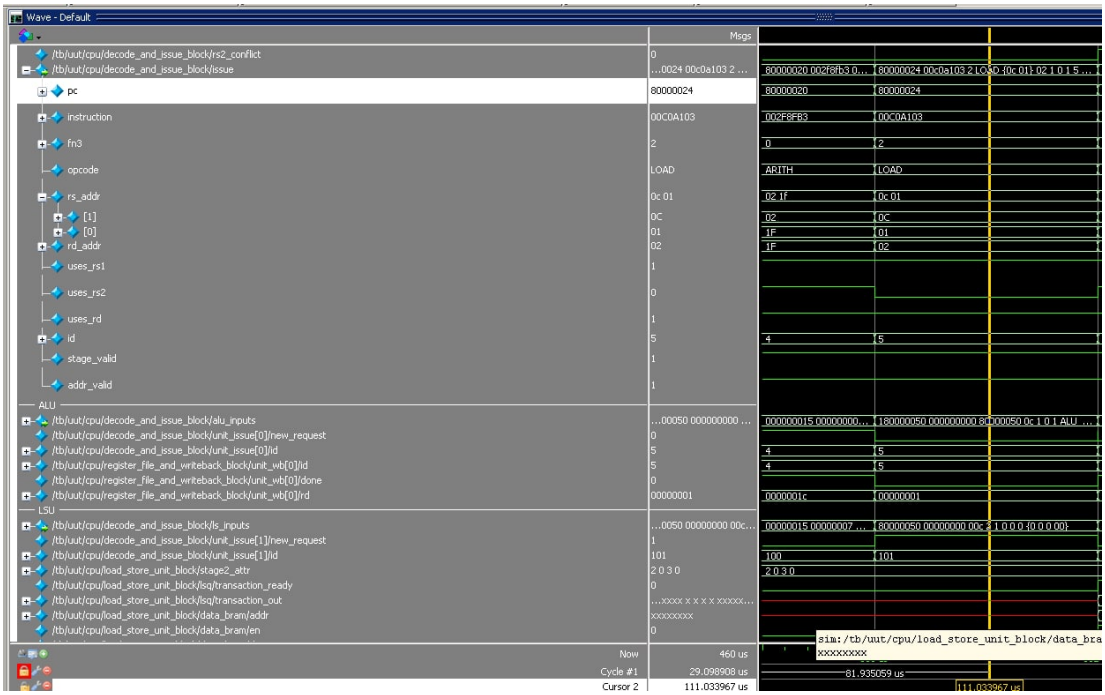

Рисунок 2.4 – Временная диаграмма выполнения стадии выполнения команды с адресом 80000024

Так как команда 80000024 – команда доступа к памяти, то ее выполнение занимает 3 такта. Она начнет выполняться в 14 такте и закончит в 16 такте.

2.5 Задание 5

В данном задании симуляция происходит на программе из задания 1.

Для проверки сравним теоретические значение с значением, полученным с помощью симуляции. На рисунке 2.5 представлен результат из симуляции. Он равен $0x25_{16} = 37_{10}$. Теоретическое значение сошлось с действительным.

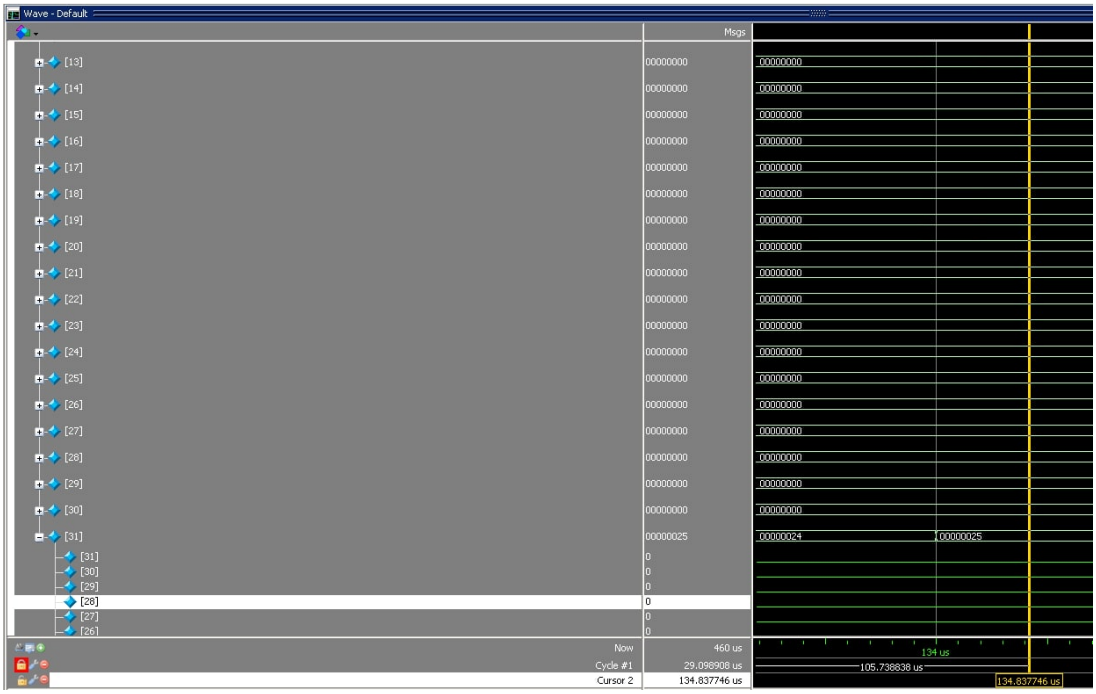

Рисунок 2.5 – Результат работы программы в симуляции

В тексте программы 2.1 символом #! обозначена команда `lw x3, 4(x1)`. Из дизассемблированного кода, приведенного на рисунке 2.1, можно увидеть, что эта команда имеет адрес 80000014. На рисунках 2.6 – 2.7 можно увидеть временные диаграммы сигналов, соответствующих всем стадиям выполнения этой команды.

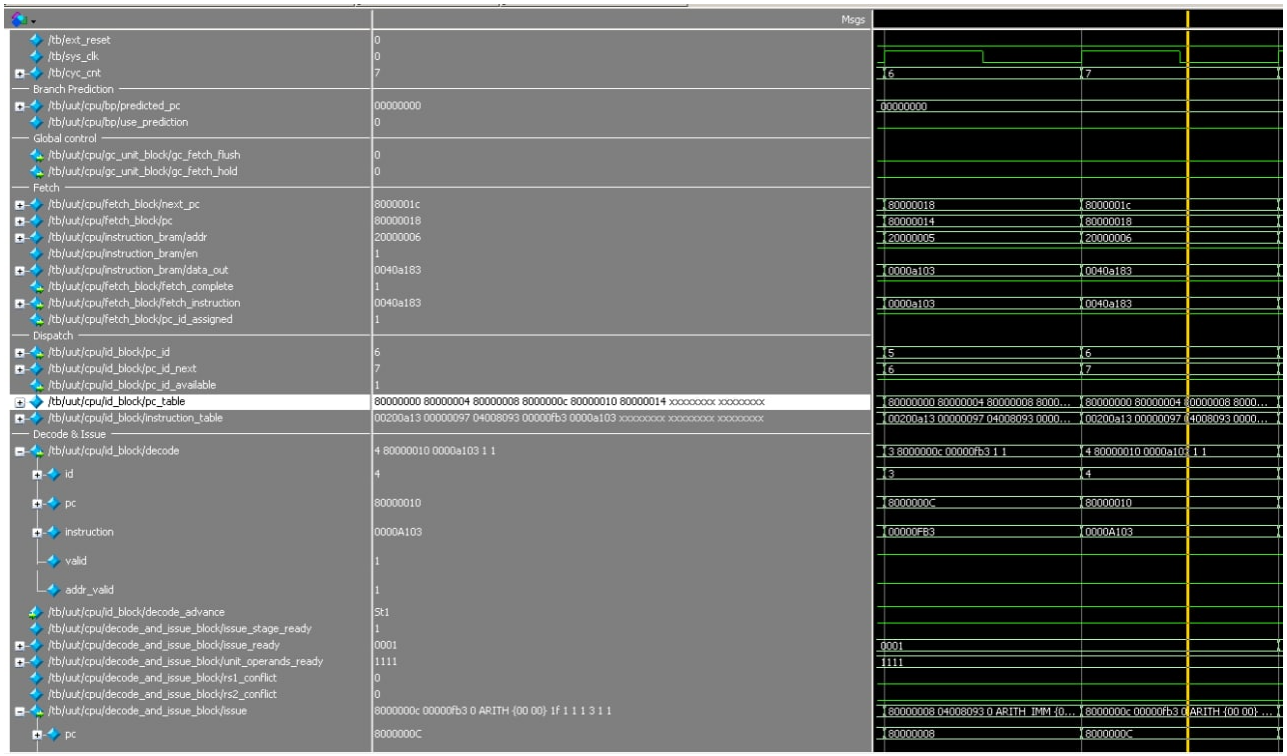

Рисунок 2.6 – Временные диаграммы выполнения стадий выборки и диспетчеризации

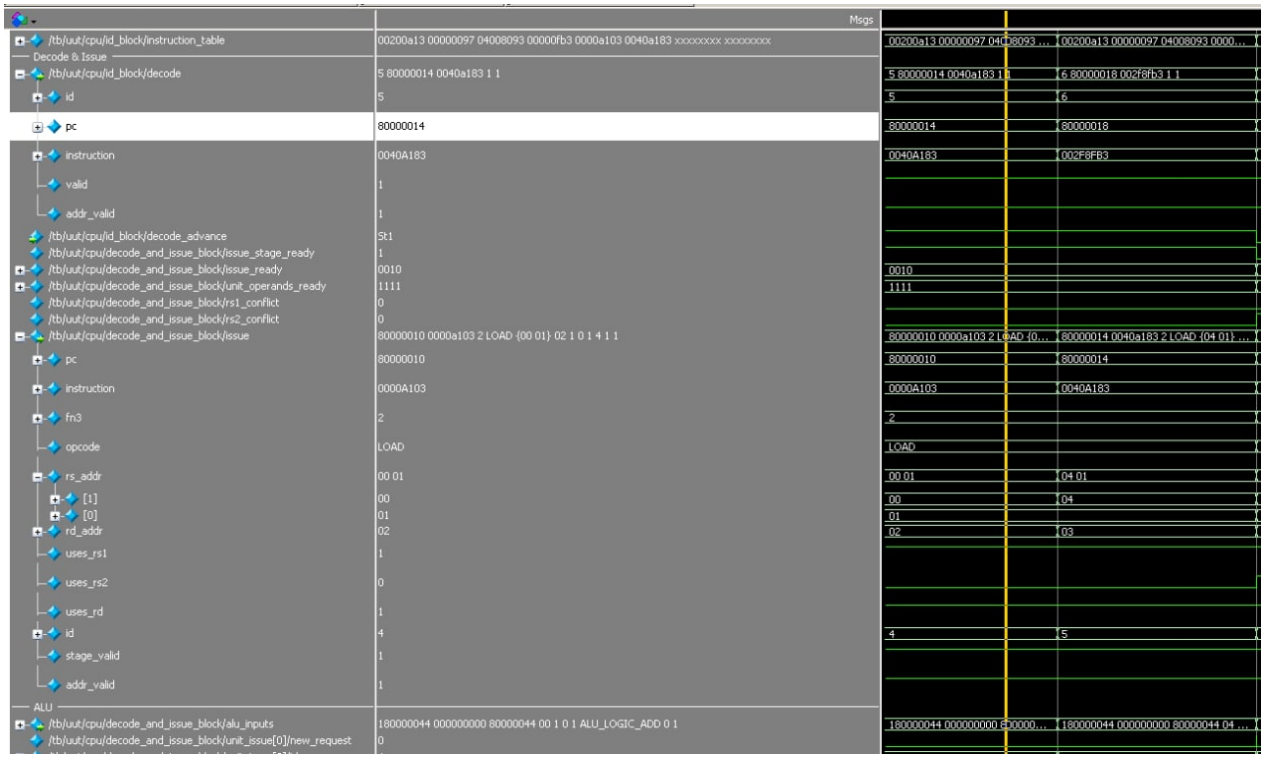

Рисунок 2.7 – Временные диаграммы выполнения стадий декодирования и планирования

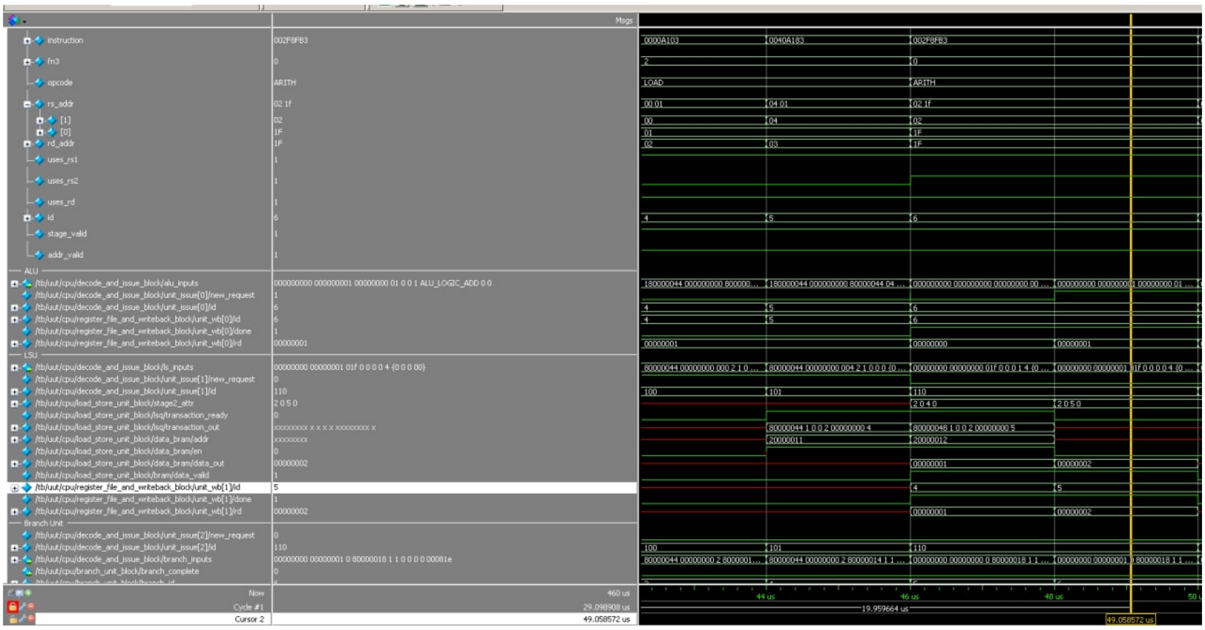

Рисунок 2.8 – Временная диаграмма выполнения стадии выполения

В 8 такте происходит декодирование команды. В 9 такте – ее планирование. С 9 такта по 11 программа выполнялась. Наглядно порядок выполнения стадий конвейера процесса можно увидеть на трассе выполнения программы, представленной на 2.9.

Адрес	Код команды	Команда	id	Номер такта																																																
				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
80000000<_start>	00200a13	addi x20,x0,2	0	F	ID	D	AL																																												
80000004	00000097	auipc x1,0x0	1		F	ID	D	AL																																											
80000008	4008093	addi x1,x1,64#80000040<x>	2			F	ID	D	AL																																										
8000000c	00000fb3	add x31,x0,x0	3				F	ID	D	AL																																									
80000010<lp>	0000a103	lw x2,0(x1)	4					F	ID	D	M1	M2	M3																																						
80000014	0040a103	lw x3,4(x1)	5						F	ID	D	M1	M2	M3																																					
80000018	002f8fb3	add x31,x31,x2	6							F	ID	D	C	AL																																					
8000001c	003f8fb3	add x31,x31,x3	7								F	ID	W	D	AL																																				
80000020	0080a203	lw x4,8(x1)	0									F	ID	W	D	M1	M2	M3																																	
80000024	00c0a283	lw x5,12(x1)	1										F	ID	W	D	M1	M2	M3																																
80000028	004f8fb3	add x31,x31,x4	2											F	ID	W	D	C	AL																																
8000002c	005f8fb3	add x31,x31,x5	3												F	ID	W	W	D	AL																															
80000030	1008093	addi x1,x1,16	4													F	ID	W	W	D	AL																														
80000034	fffa0a13	addi x20,x20,-1	5														F	ID	W	W	D	AL																													
80000038	fc0a1ce3	bne x20,x0,80000010<lp>	6															F	ID	W	W	D	C	B																											
8000003c	001f8f93	addi x31,x31,1	7																F	ID	W	W	D	X																											
80000040<lp2>	0000006f	jal x0,80000040<forever>	0																	F	ID	W	W	DX																											
80000044	00000002	<invalid operation>	1																		F	ID	W	X																											
80000048	00000003	<invalid operation>	2																			F	ID	X																											
8000004c	00000004	<invalid operation>	3																				F	X																											
80000050	00000005	<invalid operation>	4																					FX																											
80000010<loop>	0000a103	lw x2,0(x1)	0																						F	ID	D	M1	M2	M3																					
80000014	0040a103	lw x3,4(x1)	1																							F	ID	D	M1	M2	M3																				
80000018	002f8fb3	add x31,x31,x2	2																								F	ID	D	C	AL																				
8000001c	003f8fb3	add x31,x31,x3	3																									F	ID	W	D	AL																			
80000020	0080a203	lw x4,8(x1)	4																										F	ID	W	D	M1	M2	M3																
80000024	00c0a283	lw x5,12(x1)	5																											F	ID	W	D	M1	M2	M3															
80000028	004f8fb3	add x31,x31,x4	6																												F	ID	W	D	C	AL															
8000002c	005f8fb3	add x31,x31,x5	7																													F	ID	W	W	D	AL														
80000030	01008093	addi x1,x1,16	0																														F	ID	W	W	D	AL													
80000034	fffa0a13	addi x20,x20,-1	1																															F	ID	W	W	D	AL												
80000038	fc0a1ce3	bne x20,x0,80000010<loop>	2																																F	ID	W	W	D	C	B										
80000010<loop>	0000a103	lw x2,0(x1)	3																																	F	ID	W	W	D	X										
80000014	0040a103	lw x3,4(x1)	4																																		F	ID	W	W	DX										
80000018	0040a103	add x31,x31,x2	0																																			F	ID	W	X										
8000001c	002f8fb3	add x31,x31,x3	1																																				F	ID	X										
80000020	003f8fb3	lw x4,8(x1)	2																																					F	X										
80000024	0080a203	lw x5,12(x1)	3																																						FX										
8000003c	00c0a283	addi x31,x31,1	4																																							F	ID	D	AL						
80000040<lp2>	001f8f93	jal x0,80000040<forever>	5																																								F	ID	D	B					
80000044	0000006f	<invalid operation>	3																																									F	ID	D	X				
80000048	00000001	<invalid operation>	4																																										F	ID	X				
8000004c	00000002	<invalid operation>	5																																											F	X				
80000050	00000003	<invalid operation>	6																																												FX				
80000040<lp2>	00000004	jal x0,80000040<forever>	7																																													F	ID	D	B	
80000040<lp2>	0000006f	jal x0,8000004c<forever>	0																																														F	ID	D	B

Рисунок 2.9 – Трасса выполнения программы

Из трассы видно, что конфликты происходят из-за того, что пока данные загружаются в память, уже готова к выполнению операция сложения с этими данными. Оптимизировать программу можно следующим путем: пока в память загружаются данные, производить вычислительные операции не связанные с этими данными. В результате можно будет уменьшить программу на 4 такта или на $4/49 = 8\%$.

Код оптимизированной программы представлен на листинге 2.3, дизассемблированный код – на рисунке 2.10, псевдокод – в листинге 2.4.

Листинг 2.3 – Код оптимизированной программы по индивидуальному варианту

```
        .section .text
        .globl _start;
        len = 8 #Размер массива
        enroll = 4 #Количество обрабатываемых элементов за одну итерацию
    elem_sz = 4 #Размер одного элемента массива

_start:
        addi x20, x0, len/enroll
        la x1, _x
    add x31, x0, x0
lp:
        lw x2, 0(x1)
        lw x3, 4(x1) #!
        lw x4, 8(x1)
        lw x5, 12(x1)
        add x31, x31, x2
        add x31, x31, x3
        add x31, x31, x4
        add x31, x31, x5
        addi x1, x1, elem_sz*enroll
        addi x20, x20, -1
        bne x20, x0, 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
```

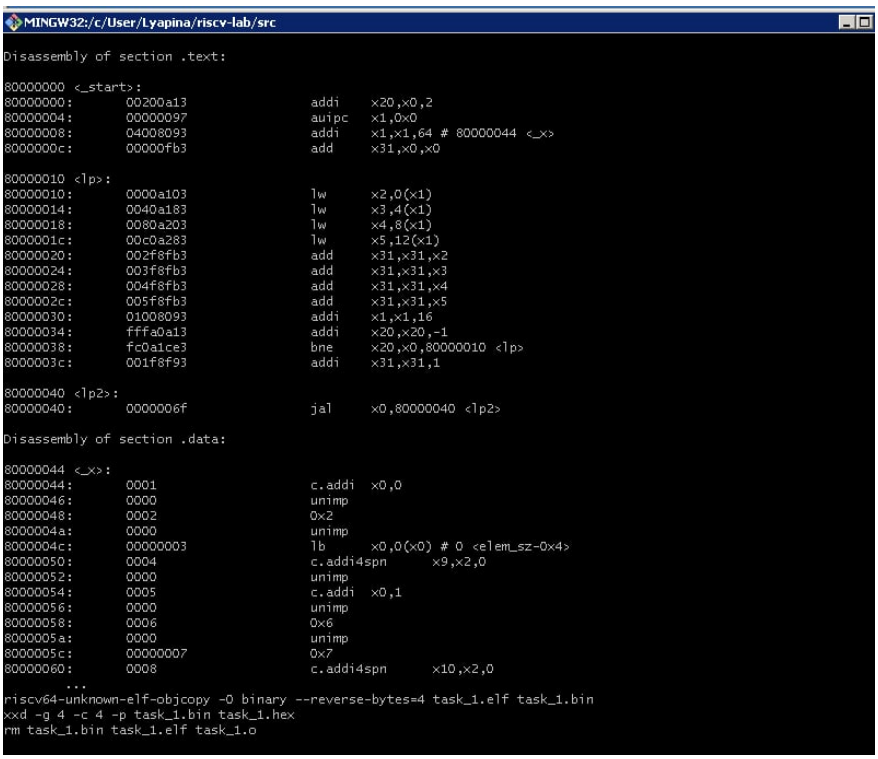

Рисунок 2.10 – Дизассемблированный код оптимизированной программы

Листинг 2.4 – Псевдокод оптимизировнной программы на языке Си

```
#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;
    int x31 += 0;

    do
    {
        int x2 = x1[0];
        int x3 = x1[4];
        int x4 = x1[8];
        int x5 = x1[12];
        x31 += x2;
        x31 += x3;
        x31 += x4;
        x31 += x5;
        x1 += elem_sz*enroll;
        int x20 -= 1;
    }
```

```
    while (x20 != 0);
    x31 += 1;

    while (1) {};
}
```

Теперь трасса выполнения программы, представленная на рисунке 2.11, выглядит следующим образом.

Адрес	Код команды	Команда	id	Номер такта																																											
				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
80000000<_start>	00200a13	addi x20,x0,2	0	F	ID	D	AL																																										
80000004	00000097	auipc x1,0x0	1		F	ID	D	AL																																									
80000008	4008093	addi x1,x1,64#80000040<x>	2			F	ID	D	AL																																						
8000000c	00000fb3	add x31,x0,x0	3				F	ID	D	AL																																					
80000010<lp>	0000a103	lw x2,0(x1)	4					F	ID	D	M1	M2	M3																																		
80000014	0040a103	lw x3,4(x1)	5						F	ID	D	M1	M2	M3																																	
80000018	0080a103	lw x4,8(x1)	6							F	ID	D	M1	M2	M3																																
8000001c	00c0a103	lw x5,12(x1)	7								F	ID	D	M1	M2	M3																															
80000020	002f8fb3	add x31,x31,x2	0									F	ID	D	AL																																
80000024	003f8fb3	add x31,x31,x3	1										F	ID	D	AL																															
80000028	004f8fb3	add x31,x31,x4	2											F	ID	D	AL																														
8000002c	005f8fb3	add x31,x31,x5	3												F	ID	D	AL																													
80000030	1008093	addi x1,x1,16	4													F	ID	D	AL																												
80000034	fffa0a13	addi x20,x20,-1	5														F	ID	D	AL																											
80000038	fc0a1ce3	bne x20,x0,80000010<lp>	6															F	ID	D	B																										
8000003c	001f8f93	addi x31,x31,1	7																F	ID	D	AL																									
80000040<lp2>	0000006f	jal x0,80000040<lp2>	0																	F	ID	DX																									
80000044	00000002	<invalid operation>	1																		F	ID	X																								
80000048	00000003	<invalid operation>	2																			FX																									
80000010<lp>	0000a103	lw x2,0(x1)	0																				F	ID	D	M1	M2	M3																			
80000014	0040a103	lw x3,4(x1)	1																					F	ID	D	M1	M2	M3																		
80000018	0080a103	lw x4,8(x1)	2																						F	ID	D	M1	M2	M3																	
8000001c	00c0a103	lw x5,12(x1)	3																							F	ID	D	M1	M2	M3																
80000020	002f8fb3	add x31,x31,x2	4																								F	ID	D	AL																	
80000024	003f8fb3	add x31,x31,x3	5																									F	ID	D	AL																
80000028	004f8fb3	add x31,x31,x4	6																										F	ID	D	AL															
8000002c	005f8fb3	add x31,x31,x5	7																											F	ID	D	AL														
80000030	1008093	addi x1,x1,16	0																												F	ID	D	AL													
80000034	fffa0a13	addi x20,x20,-1	1																													F	ID	D	AL												
80000038	fc0a1ce3	bne x20,x0,80000010<lp>	2																														F	ID	D	B											
80000010<lp>	0000a103	lw x2,0(x1)	0																															F	ID	D	X										
80000014	0040a103	lw x3,4(x1)	1																																F	ID	DX										
80000018	0080a103	lw x4,8(x1)	2																																	F	ID	X									
8000001c	00c0a103	lw x5,12(x1)	3																																		FX										
8000003c	00c0a283	addi x31,x31,1	4																																			F	ID	D	AL						
80000040<lp2>	0000006f	jal x0,80000040<lp2>	5																																				F	ID	D	B					
80000044	00000002	<invalid operation>	6																																					F	ID	DX					
80000048	00000003	<invalid operation>	7																																						F	ID	X				
8000004c	00000002	<invalid operation>	0																																							F	X				
80000050	00000003	<invalid operation>	1																																								FX				
80000040<lp2>	00000004	jal x0,80000040<lp2>	2																																									F	ID	D	AL	
80000040<lp2>	0000006f	jal x0,8000004c<lp2>	3																																										F	ID	D	AL

Рисунок 2.11 – Трасса выполнения оптимизированной программы

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 требуется оптимизация. Программу можно было оптимизировать на 20%.

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.