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

Название Изучение принципов работы микропроцессорного ядра RISC-V

Дисциплина Архитектура элекронно-вычислительных машин

Студент: _____ Мансуров В. М.
подпись, дата Фамилия, И.О.

Преподаватель: _____ Попов А. Ю.
подпись, дата Фамилия, И. О.

Москва — 2022 г.

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.

1 Основные теоретические сведения

RISC-V является открытым современным набором команд, который может использоваться для построения как микроконтроллеров, так и высокопроизводительных микропроцессоров. 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.

1.1 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.

1.2 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 задаются явно.

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.

2 Общая программа

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 - Листинг 2.1.

Листинг 2.1 – Пример программы

```
1     .section .text (1)
2     .globl _start; (2)
3     len = 8 #Размер массива (3)
4     enroll = 4 #Количество обрабатываемых элементов за одну итерацию
5     elem_sz = 4 #Размер одного элемента массива
6  _start: (4)
7     addi x20, x0, len/enroll (5)
8     la x1, _x (6)
9     loop:
10    lw x2, 0(x1) (7)
11    add x31, x31, x2 (8)
12    lw x2, 4(x1)
13    add x31, x31, x2
14    lw x2, 8(x1)
15    add x31, x31, x2
16    lw x2, 12(x1)
17    add x31, x31, x2
18    addi x1, x1, elem_sz*enroll (9)
19    addi x20, x20, -1 (10)
20    bne x20, x0, loop (11)
21    addi x31, x31, 1
22 forever: j forever (12)
23
24 .section .data (13)
25 _x: .4byte 0x1 (14)
26    .4byte 0x2
27    .4byte 0x3
28    .4byte 0x4
29    .4byte 0x5
30    .4byte 0x6
31    .4byte 0x7
32    .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. Метка.
4. Объявление констант.
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.

Дизассемблерный код представлен на листинге 2.2.

Листинг 2.2 – Дизассемблированный код примера программы

```
1  Disassembly of section .text:
2
3  80000000 <_start>:
4  80000000:       00200a13        addi    x20,x0,2
5  80000004:       00000097        auipc   x1,0x0
6  80000008:       03c08093        addi    x1,x1,60 # 80000040 <_x>
7
8  8000000c <lp>:
9  8000000c:       0000a103        lw      x2,0(x1)
10 80000010:       002f8fb3        add     x31,x31,x2
11 80000014:       0040a183        lw      x3,4(x1)
12 80000018:       003f8fb3        add     x31,x31,x3
13 8000001c:       0080a203        lw      x4,8(x1)
14 80000020:       00c0a283        lw      x5,12(x1)
15 80000024:       004f8fb3        add     x31,x31,x4
16 80000028:       005f8fb3        add     x31,x31,x5
17 8000002c:       01008093        addi    x1,x1,16
18 80000030:       fffa0a13        addi    x20,x20,-1
19 80000034:       fc0a1ce3        bne     x20,x0,8000000c <lp>
20 80000038:       001f8f93        addi    x31,x31,1
21
22 8000003c <lp2>:
23 8000003c:       0000006f        jal     x0,8000003c <lp2>
```

Можно сказать, что данная программа эквивалентна следующему псевдокоду на языке C, представленному на листинге 2.3.

Листинг 2.3 – Псевдокод общей программы

```
#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){}
}
```

3 Результаты исследования программы

Задания выполнялись по варианту 9.

3.1 Задание №1

Условие задания

В процесссе 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 $gmake$. 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 (см. Индивидуальные варианты). Поместить его в каталог src. Текст программы сохранить в файле с расширением $.s$.
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 $gmake$. 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.

Результаты выполнения

Листинг 3.1 – Код программы 9 варианта

```
1     .section .text
2     .globl _start;
3     len = 8
4     enroll = 4
5     elem_sz = 4
6
7  _start:
8     addi x20, x0, len/enroll
9     la x1, _x
10    lp:
11    lw x2, 0(x1)
12    add x31, x31, x2 #!
13    lw x3, 4(x1)
14    add x31, x31, x3
15    lw x4, 8(x1)
16    lw x5, 12(x1)
17    add x31, x31, x4
18    add x31, x31, x5
19    addi x1, x1, elem_sz*enroll
20    addi x20, x20, -1
21    bne x20, x0, lp
22    addi x31, x31, 1
23 lp2: j lp2
24
25    .section .data
26 _x: .4byte 0x1
27    .4byte 0x2
28    .4byte 0x3
29    .4byte 0x4
30    .4byte 0x5
31    .4byte 0x6
32    .4byte 0x7
33    .4byte 0x8
```

Выполнение команды *gmake*, после присвоения SRC = название файла, где содержится код программы варианта 9, в мое случае этот файл был назван *lab*_02_09, это показано на рисунке 3.1

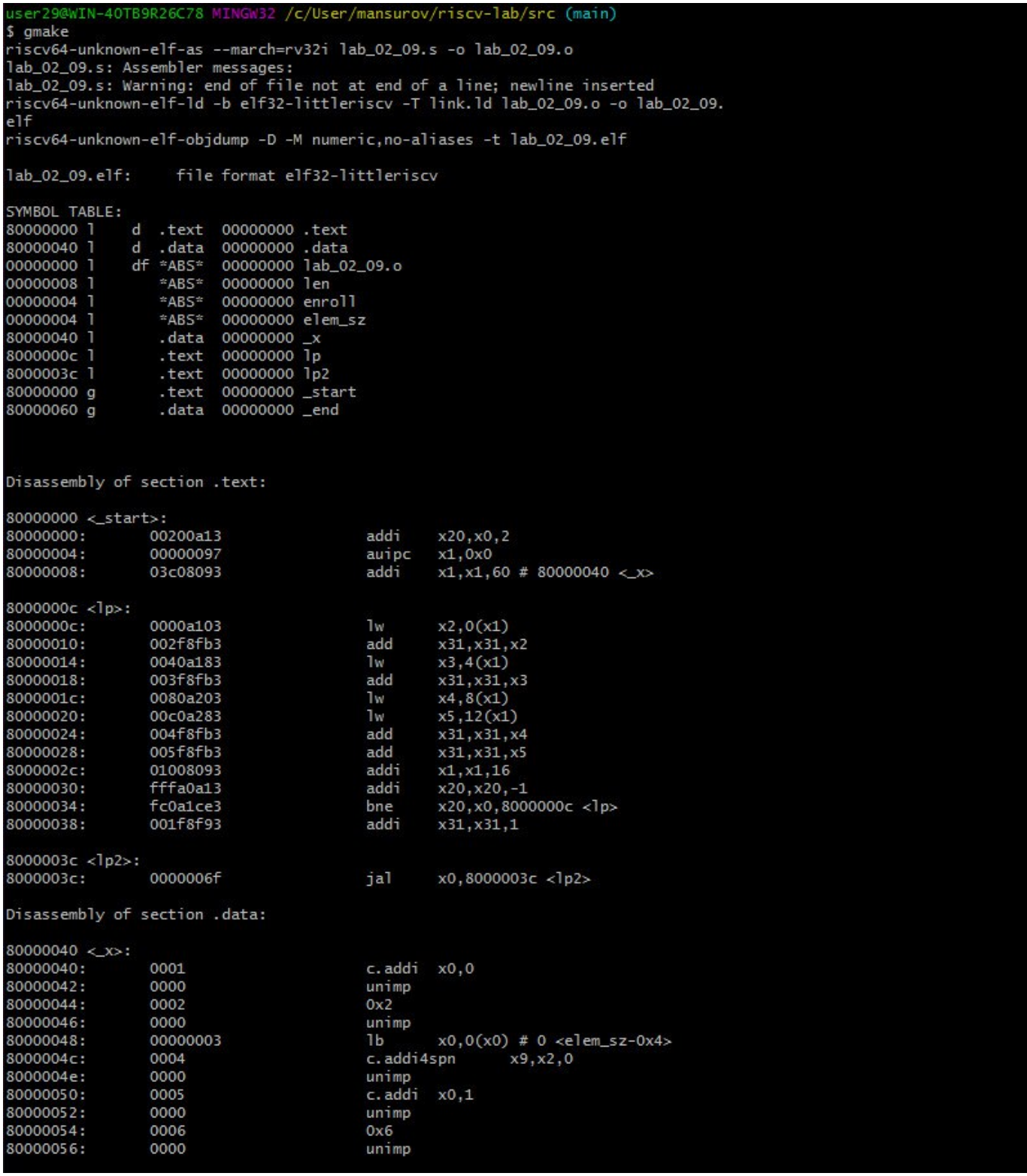

Рисунок 3.1 – Скриншот запуска gmake для программы по варианту 9

Листинг 3.2 – Дизассемблированный код 9 варианта

```
1  Disassembly of section .text:
2
3  80000000 <_start>:
4  80000000:       00200a13        addi    x20,x0,2
5  80000004:       00000097        auipc   x1,0x0
6  80000008:       03c08093        addi    x1,x1,60 # 80000040 <_x>
7
8  8000000c <lp>:
9  8000000c:       0000a103        lw      x2,0(x1)
10 80000010:       0040a183        add     x31,x31,x2
11 80000014:       0080a203        lw      x4,8(x1)
12 80000018:       00c0a283        add     x31,x31,x3
13 8000001c:       002f8fb3        lw      x3,4(x1)
14 80000020:       003f8fb3        add     x31,x31,x4
15 80000024:       004f8fb3        lw      x5,12(x1)
16 80000028:       005f8fb3        add     x31,x31,x5
17 8000002c:       01008093        addi    x1,x1,16
18 80000030:       fffa0a13        addi    x20,x20,-1
19 80000034:       fc0a1ce3        bne     x20,x0,8000000c <lp>
20 80000038:       001f8f93        addi    x31,x31,1
21
22 8000003c <lp2>:
23 8000003c:       0000006f        jal     x0,8000003c <lp2>
```

Можно сказать, что данная программа эквивалентна следующему псевдокоду на языке C, представленному на листинге 3.3.

Листинг 3.3 – Псевдокод программы 9 варианта

```
#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;
        int x3 = x1[1];
        x31 += x3;
        int x4 = x1[2];
        x31 += x4;
        int x5 = x1[3];
        x31 += x5;
        x1 += enroll;
        x20--;
    } while(x20 != 0);
    x31++;
    while(1){}
}
```

После выполнения программы по варианту 9 в $x31$ будет записано число 38 (подсчитано вручную и написанной программы на языке C по псевдокоду).

3.2 Задание №2

Условие задания

В ходе выполнения данного задания необходимо выполнить следующие действия:

1. Открыть проект $riscv-lab/taiga/taiga.qpf$ в среде $Intel\ Quartus$. При запуске $Intel\ Quartus$, если потребуется, выбрать опцию $Run\ the\ Quartus\ Prime\ software$.

2. Выполнить синтез проекта выбрав пункт меню Processing → Start → Start Analysis & Synthesis.

3. Запустить симуляцию в среде Modelsim. Для этого выбрать в меню Quartus пункт Tools → Run Simulation Tool → RTL Simulation.

4. Запустить симуляцию, набрав в командной строке Modelsim команду $run\ 460us$.

5. Изучить список сигналов, приведенных в окне Wave.

6. Получить снимок экрана, содержащий временную диаграмму выполнения стадий выборки и диспетчеризации команды с адресом 8000002c на первой итерации.

Результаты выполнения

После выполнения пунктов 1-4 запускается симуляция в среде Modelsim, как видно на рисунке 3.2.

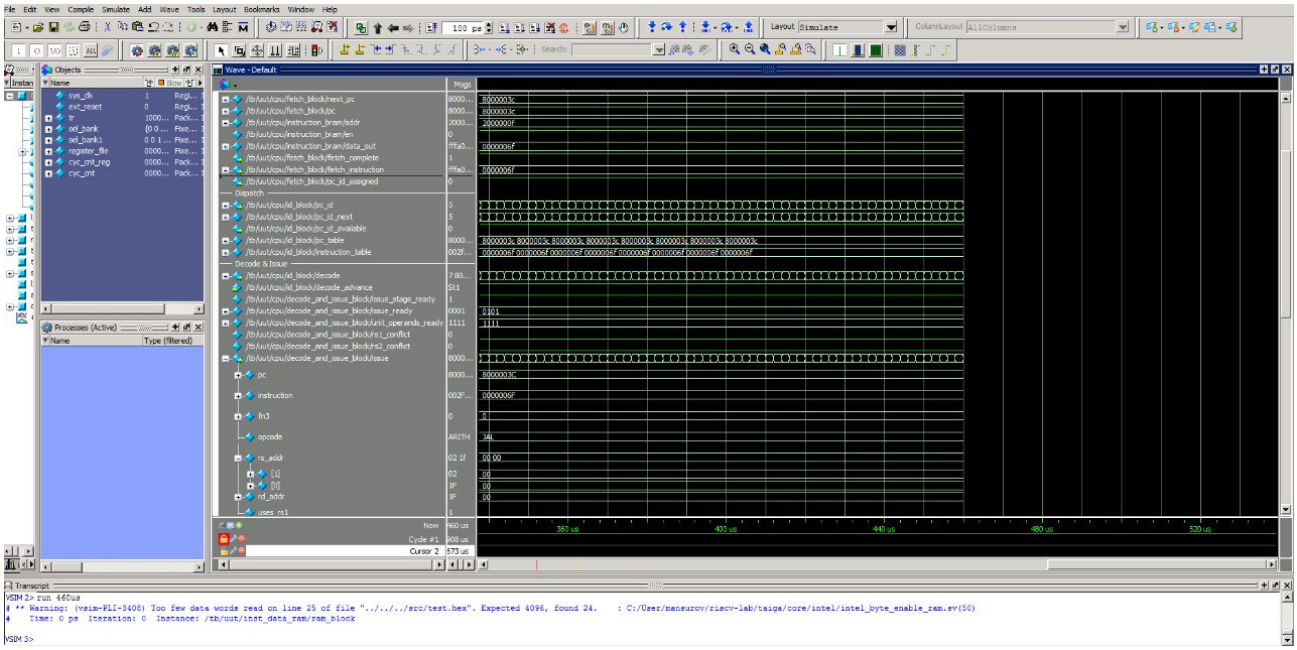

Рисунок 3.2 – Скриншот запуска симуляция в среде Modelsim

На рисунке 3.3 снимок экране симуляции в среде Modelsim на стадии выборки и диспетчеризации команды с адресом 8000002c на первой итерации.

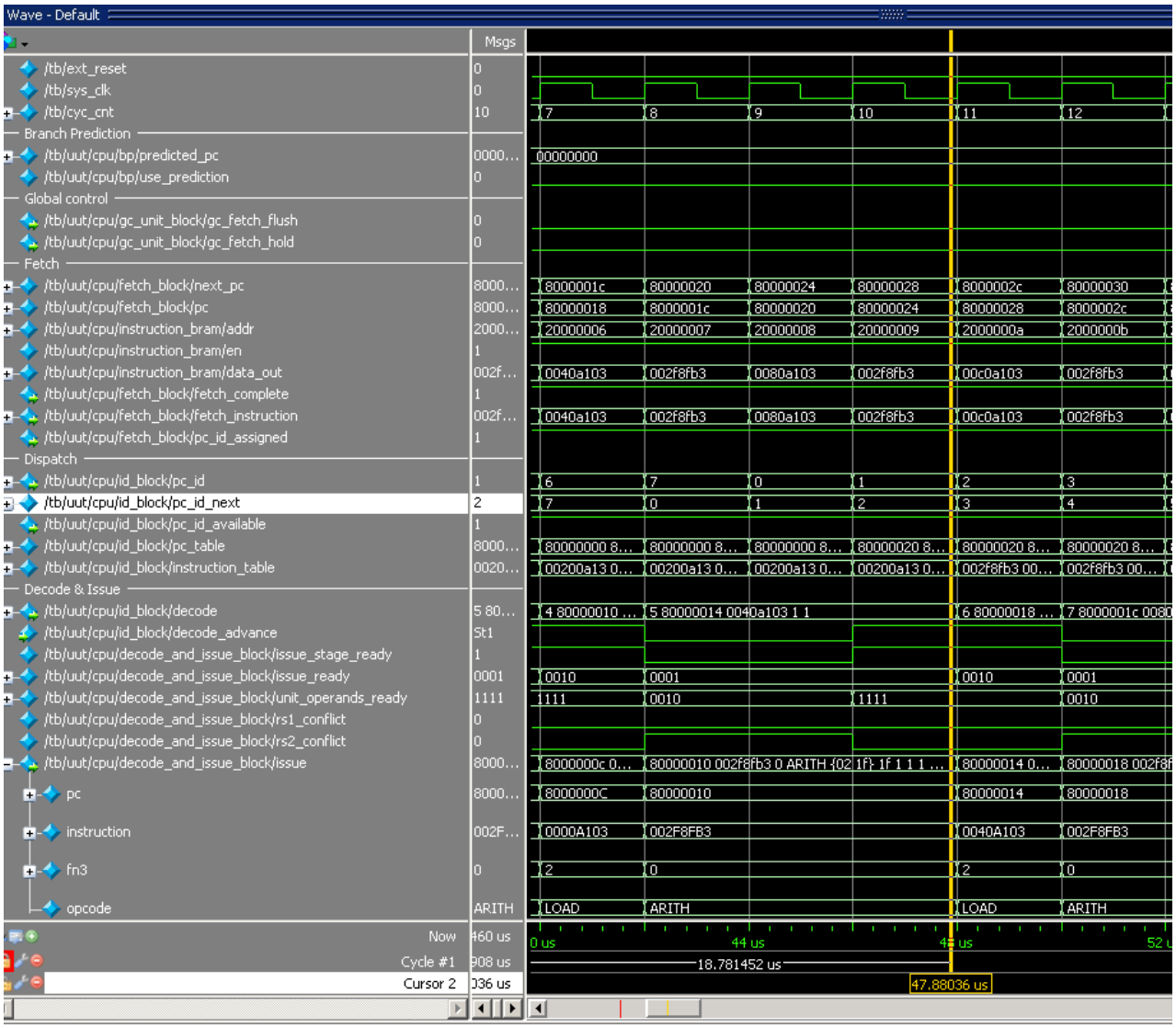

Рисунок 3.3 – Скриншот запуска симуляция в среде Modelsim – команды с адресом 8000002c на первой итерации.

3.3 Задание №3

Условие задания

Получить снимок экрана, содержащий временную диаграмму выполнения стадии декодирования и планирования на выполнение команды с адресом 8000000c на второй итерации.

Результаты выполнения

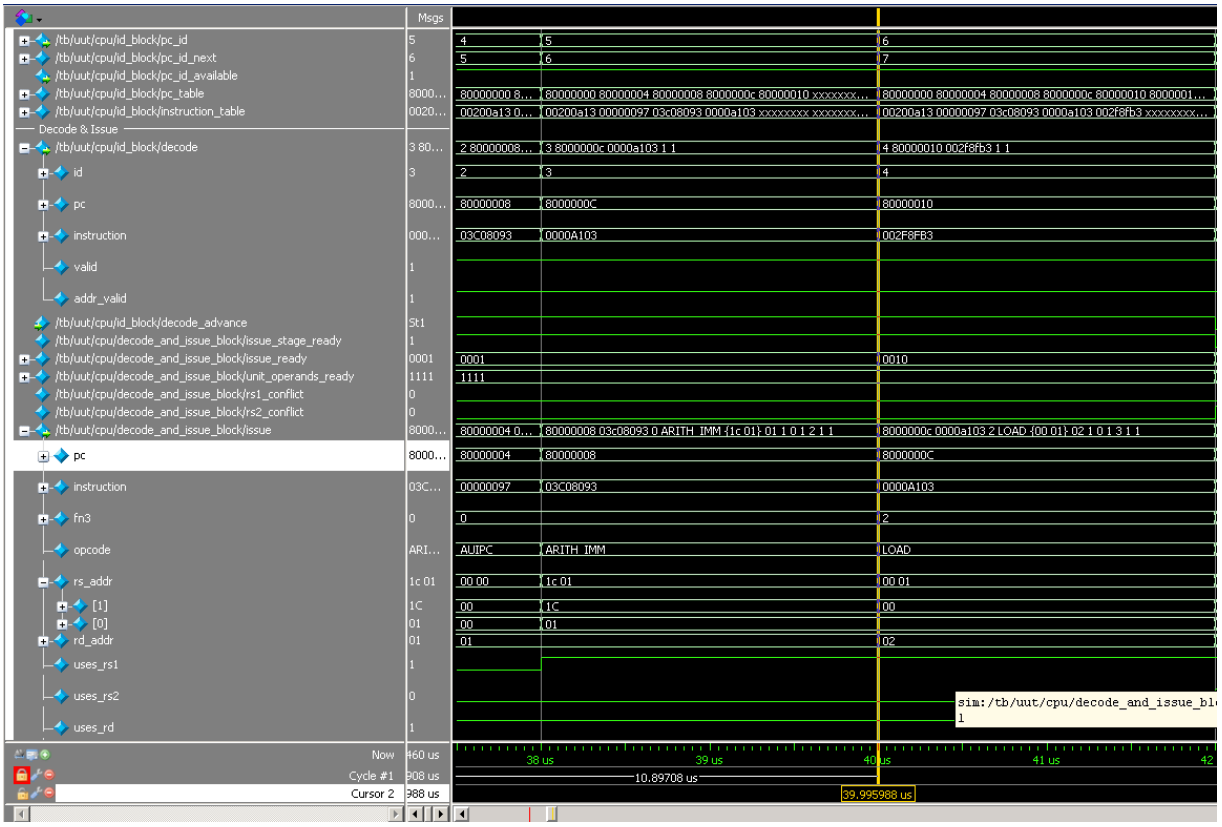

Рисунок 3.4 – Скриншот запуска симуляция в среде Modelsim – команды с адресом 8000000c на второй итерации.

3.4 Задание №4

Условие задания

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

Результаты выполнения

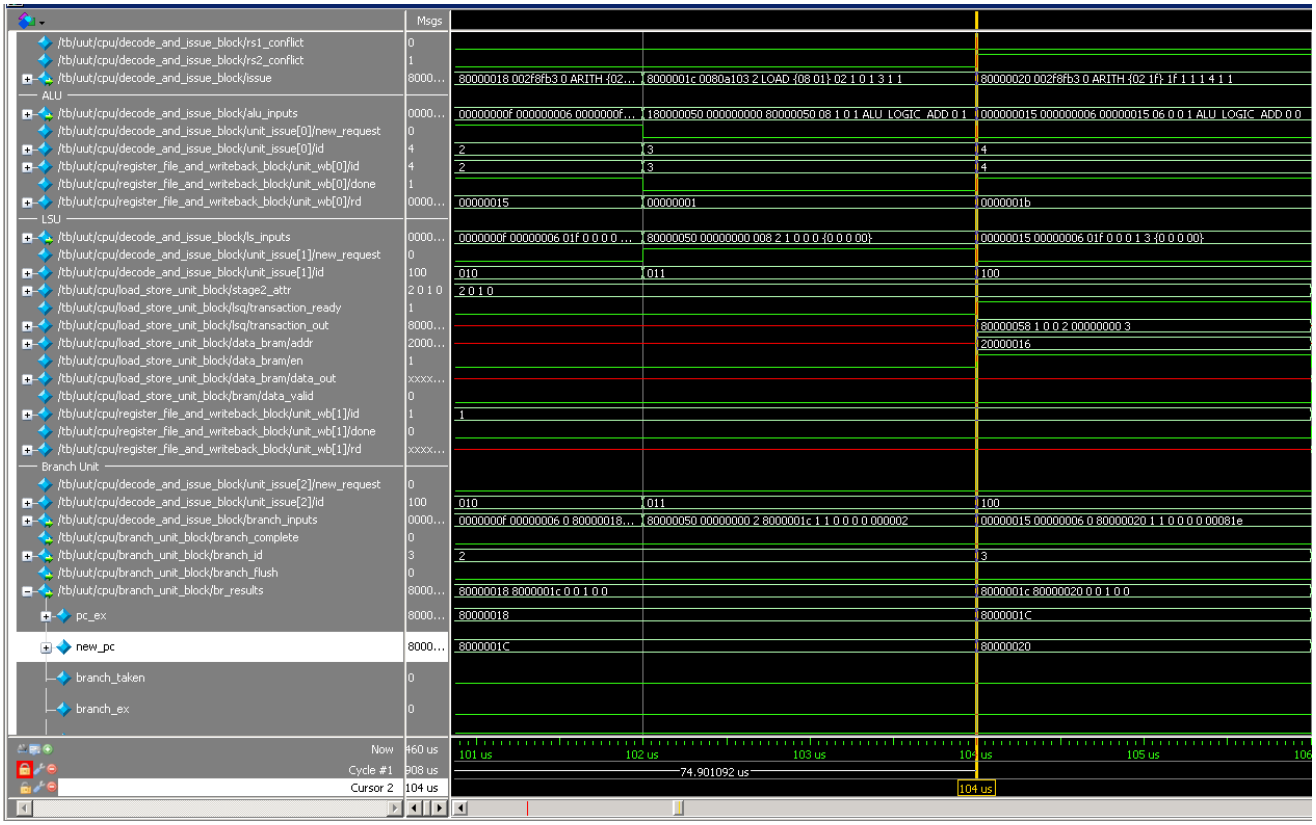

Рисунок 3.5 – Скриншот запуска симуляция в среде Modelsim – команды с адресом 80000020 на первой итерации.

3.5 Задание №5

Условие задания

В процесссе выполнения задания необходимо выполнить следующие действия:

1. Исправить 76-ю строку файла $taiga/examples/zedboard/taiga_wrapper.sv$ так, чтобы там был указан путь к файлу $.hex$, соответствующему программе по индивидуальному варианту. Сохранить файл.

2. Перекомпилировать исправленный файл. Для этого в окне программы Modelsim найти вкладку Library, в этой вкладке найти модуль work → taiga_wrapper. В контекстном меню модуля выбрать пункт Recompile.

3. Ввести в командой строке Modelsim команду $restart$; run $460us$ для перезапуска симуляции.

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.

Результаты выполнения

Трасса работы программы

Трасса работы представлена на рисунке 3.6.

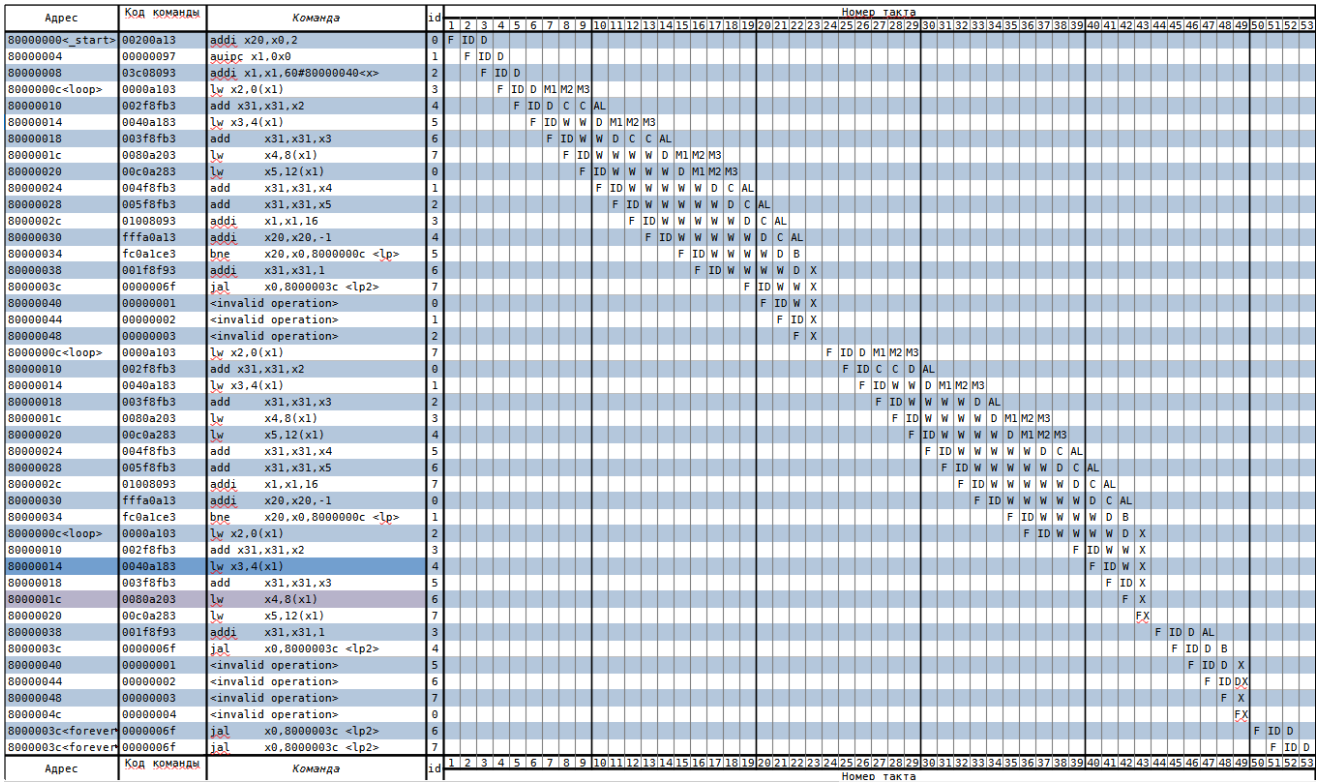

Рисунок 3.6 – Трасса работы программы.

Временные диаграммы

Временные диаграммы сигналов, соответствующих всем стадиям выполнения команды, обозначенной в тексте программы символом #! (add x31, x31, x2) представлены на рисунке 3.7.

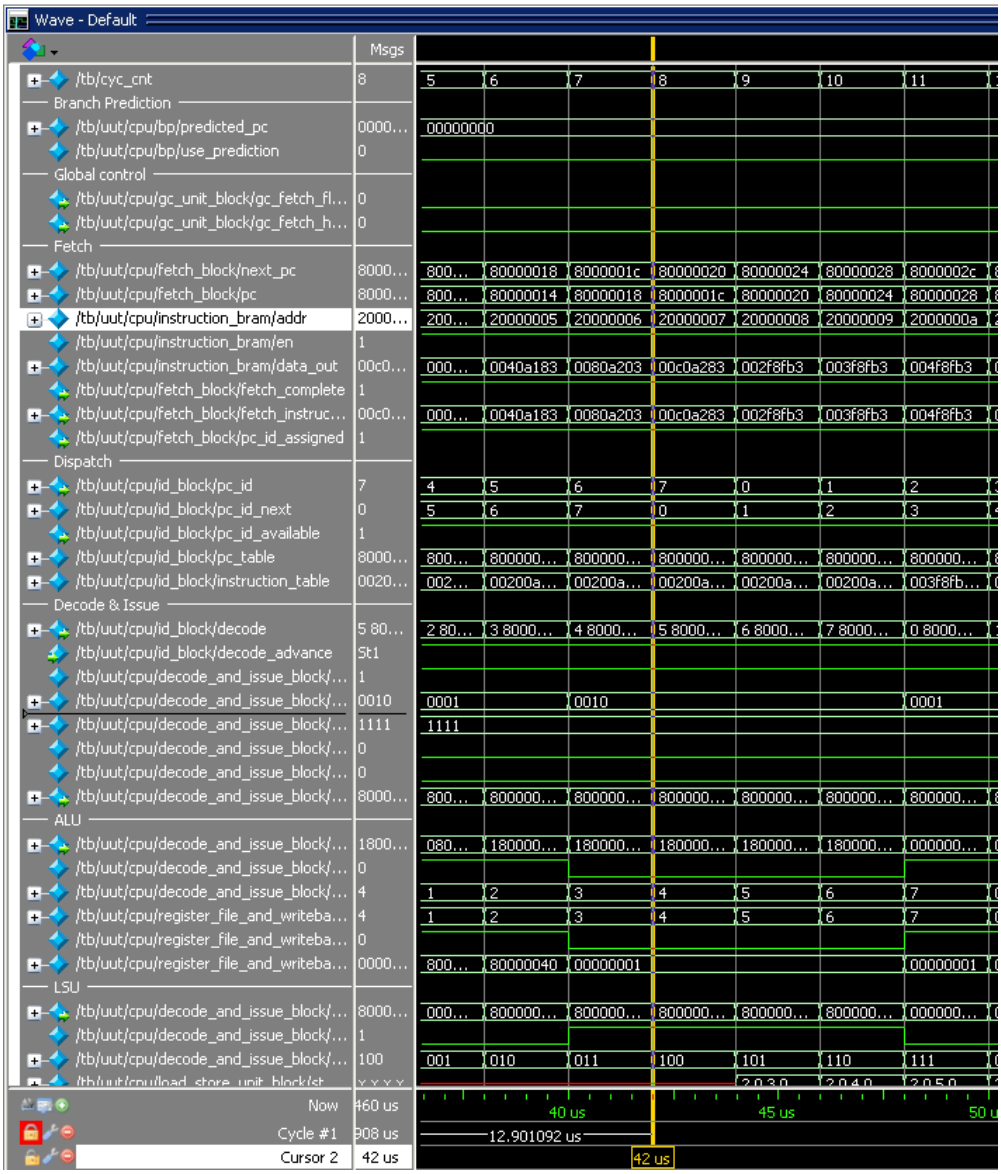

Рисунок 3.7 – Временные диаграммы сигналов.

Вывод и предложение по оптимизации

Как видно на трассе работы программы, представленной на рисунке 3.7, конфликты возникают из-за того, что данные загружаются в память тогда, когда уже готова выполниться операция сложения тех данных, которые загружаются. Из-за этого и возникают конфликты, так как нечего складывать, так как в памяти пока нет ничего.

Можно заметить, что трижды возникает ситуация ошибочной выборки, которая негативно сказывается на производительности, так как приводит к очистки конвейера.

Оптимизировать программы можно тем, что сначала загрузить все данные в память, а потом их складывать. Тем самым у нас не будет конфликтов, не будет ожидания конца загрузки данных в память.

В итоге, можно будет уменьшить программу на 2 такта 4 раза в программе, на 1 такт 3 раза в программе, то есть на $11/53 = 20\%$ программа будет работать быстрее.

Оптимизированная программа

Код программы представлен в листинте 3.4

Листинг 3.4 – Код программы 9 варианта(оптимизированный)

```
1  .section .text
2  .globl _start;
3  len = 8
4  enroll = 4
5  elem_sz = 4
6
7  _start:
8  addi x20, x0, len/enroll
9  la x1, _x
10 lp:
11 lw x2, 0(x1)
12 lw x3, 4(x1)
13 lw x4, 8(x1)
14 lw x5, 12(x1)
15 add x31, x31, x2 #!
16 add x31, x31, x3
17 add x31, x31, x4
18 add x31, x31, x5
19 addi x1, x1, elem_sz*enroll
20 addi x20, x20, -1
21 bne x20, x0, lp
22 addi x31, x31, 1
23 lp2: j lp2
24
25 .section .data
26 _x:     .4byte 0x1
27 .4byte 0x2
28 .4byte 0x3
29 .4byte 0x4
30 .4byte 0x5
31 .4byte 0x6
32 .4byte 0x7
33 .4byte 0x8
```

Дизассемблерный код представлен на листинге 3.5.

Листинг 3.5 – Дизассемблированный код 9 варинта (оптимизированный)

```
Disassembly of section .text:

80000000 <_start>:
80000000:       00200a13        addi    x20,x0,2
80000004:       00000097        auipc   x1,0x0
80000008:       03c08093        addi    x1,x1,60 # 80000040 <_x>

8000000c <lp>:
8000000c:       0000a103        lw      x2,0(x1)
80000010:       0040a183        lw      x3,4(x1)
80000014:       0080a203        lw      x4,8(x1)
80000018:       00c0a283        lw      x5,12(x1)
8000001c:       002f8fb3        add     x31,x31,x2
80000020:       003f8fb3        add     x31,x31,x3
80000024:       004f8fb3        add     x31,x31,x4
80000028:       005f8fb3        add     x31,x31,x5
8000002c:       01008093        addi    x1,x1,16
80000030:       fffa0a13        addi    x20,x20,-1
80000034:       fc0a1ce3        bne     x20,x0,8000000c <lp>
80000038:       001f8f93        addi    x31,x31,1

8000003c <lp2>:
8000003c:       0000006f        jal     x0,8000003c <lp2>
```

Можно сказать, что данная программа эквивалентна следующему псевдокоду на языке C, представленному на листинге 3.6.

Листинг 3.6 – Псевдокод программы 9 варинта (оптимизированный)

```
#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];
        int x3 = x1[1];
        int x4 = x1[2];
        int x5 = x1[3];
        x31 += x2;
        x31 += x3;
        x31 += x4;
        x31 += x5;
        x1 += enroll;
        x20--;
    } while(x20 != 0);
    x31++;
    while(1){}
}
```

Трасса работы оптимизированной программы

Трасса работы представлена на рисунке 3.8.

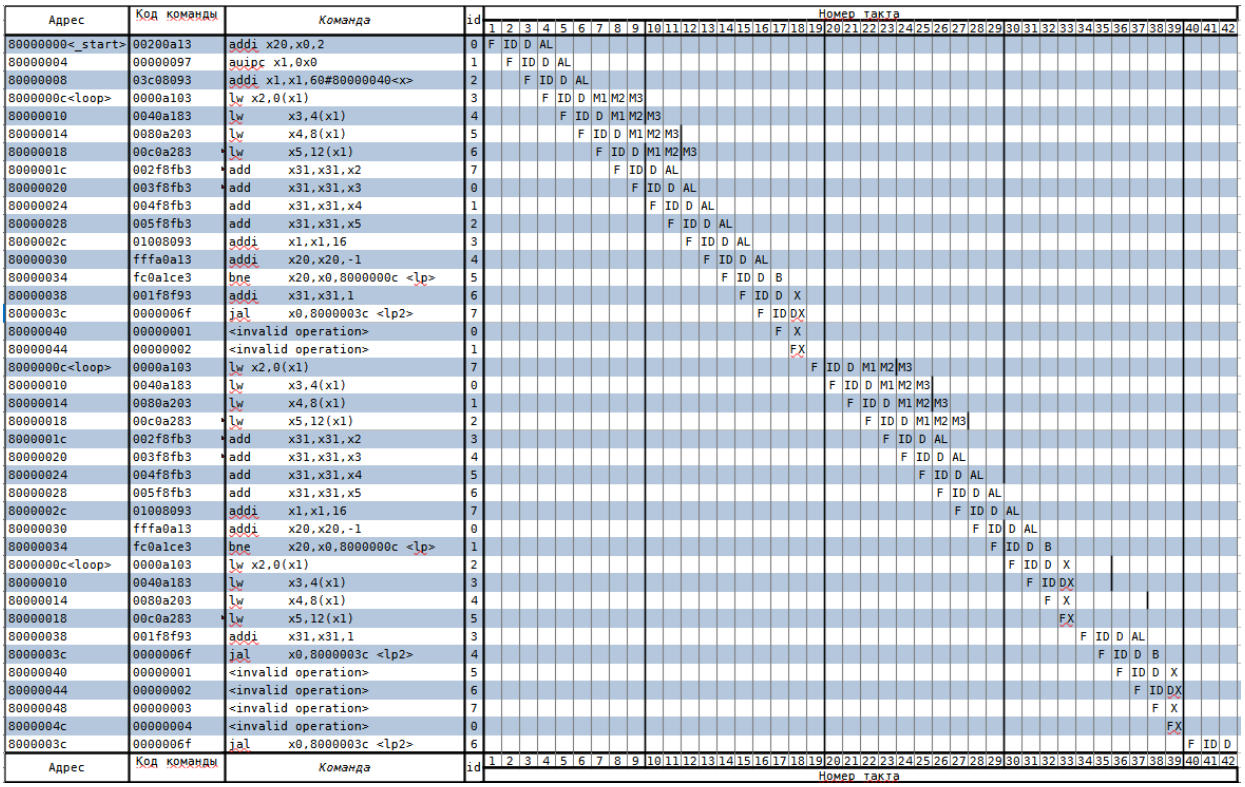

Рисунок 3.8 – Трасса работы оптимизированной программы.

Заключение

В результате выполнения лабораторной работы были изучены принципы функционирования, построения и особенности архитектуры суперскалярных конвейерных микропроцессоров.

Также были рассмотрены принципы проектирования и верификации сложных цифровых устройств с использованием языка описания аппаратуры SystemVerilog и ПЛИС.

На основе изученных материалов был найден способ оптимизации программы.

Поставленная цель достигнута.

Приложение

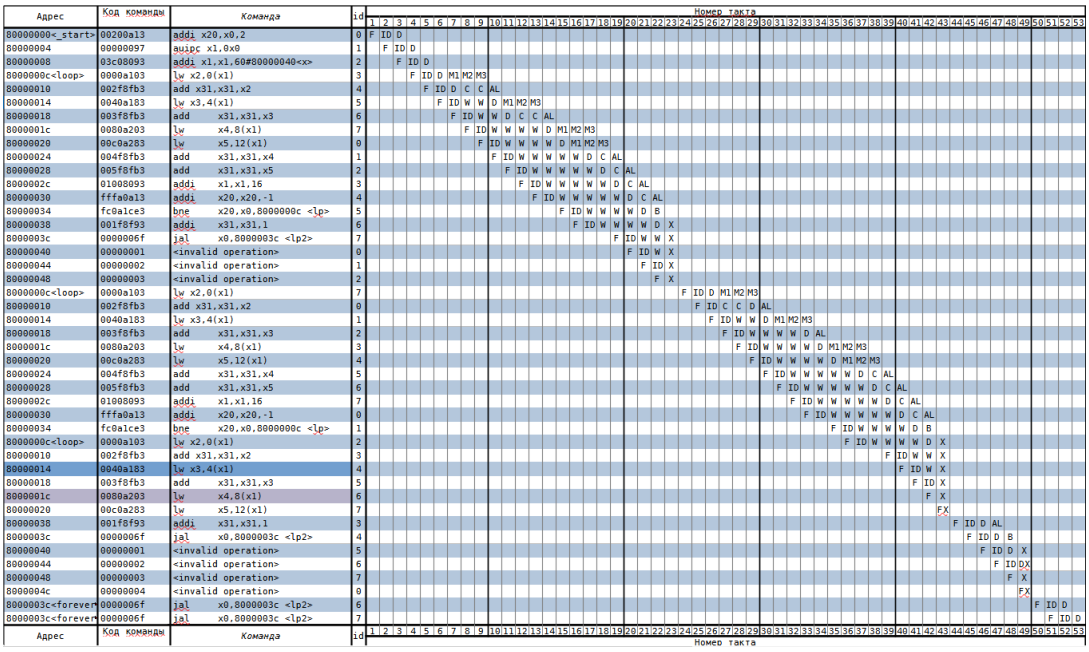

Рисунок 3.9 – Трасса работы программы.

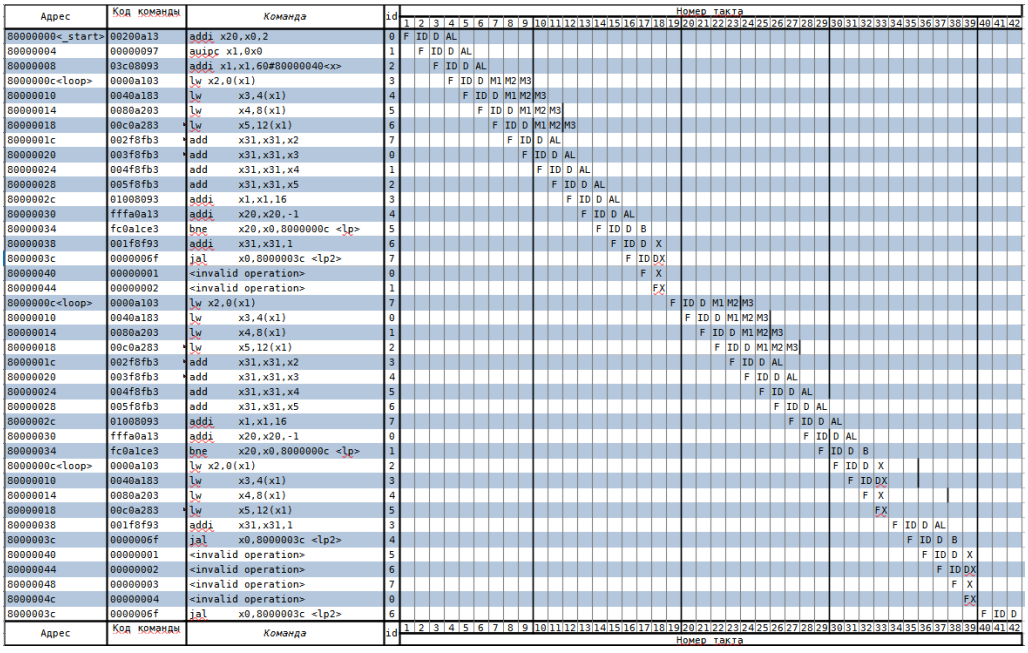

Рисунок 3.10 – Трасса работы оптимизированной программы.