[ВВЕДЕНИЕ 5](#__RefHeading___Toc7326_2062179593)

[1. РАЗРАБОТКА ОБЩЕЙ СТРУКТУРЫ МИКРО-ЭВМ 6](#__RefHeading___Toc7328_2062179593)

[1.1 Функциональный состав микро-ЭВМ 6](#__RefHeading___Toc7330_2062179593)

[1.2 Разработка системы команд 6](#__RefHeading___Toc7332_2062179593)

[1.3 Описание взаимодействия всех блоков микро-ЭВМ при выполнении команд программы 8](#__RefHeading___Toc7334_2062179593)

[2. РАЗРАБОТКА ОСНОВНЫХ УСТРОЙСТВ МИКРО-ЭВМ 9](#__RefHeading___Toc7336_2062179593)

[2.1 Функциональный состав и временные диаграммы работы ПЗУ 9](#__RefHeading___Toc7338_2062179593)

[2.2 Функциональный состав и временные диаграммы работы ОЗУ 10](#__RefHeading___Toc7340_2062179593)

[2.3 Блок стека 11](#__RefHeading___Toc7342_2062179593)

[2.4 Регистры общего назначения 12](#__RefHeading___Toc7344_2062179593)

[2.5 Арифметико-логическое устройство 14](#__RefHeading___Toc7346_2062179593)

[2.5. Устройство управления 15](#__RefHeading___Toc7348_2062179593)

[2.6. Арбитаж шин 19](#__RefHeading___Toc7350_2062179593)

[3. ФУНКЦИОНАЛЬНОЕ МОДЕЛИРОВАНИЕ 22](#__RefHeading___Toc7352_2062179593)

[ЗАКЛЮЧЕНИЕ 29](#__RefHeading___Toc7354_2062179593)

# ВВЕДЕНИЕ

В рамках данного курсового проекта будет разработана микро-ЭВМ. На разработку схемы устройства оказывает влияние тип архитектуры, что является основным фактором при реализации устройства. Различаются Принстонскую и Гарвардскую архитектуру вычислительных машин.

Разрабатываемое устройство соответствует Гарвардскому типу архитектуры. В гарвардской архитектуре принципиально невозможно осуществить операцию записи в память программ, что исключает возможность случайного разрушения управляющей программы в случае ошибки программы при работе с данными или атаки третьих лиц. Кроме того, для работы с памятью программ и с памятью данных организуются отдельные шины обмена данными. Эти особенности определили области применения гарвардской архитектуры. Гарвардская архитектура применяется в микроконтролерах и в сигнальных процессорах, где требуется обеспечить высокую надёжность работы аппаратуры.

В компьютере с использованием гарвардской архитектуры процессор может считывать очередную команду и оперировать памятью данных одновременно и без использования кэш-памяти. Таким образом, компьютер с гарвардской архитектурой при определенной сложности схемы быстрее, чем компьютер с архитектурой фон Неймана, поскольку потоки команд и данных расположены на раздельных физически не связанных между собой аппаратных каналах.

В данном курсовом проекте задача разработки микро-ЭВМ на ПЛИС решалась с помощью программного обеспечения “Quartus Prime” компании “Intel Altera”, языка Verilog и системы тестирования ModelSim. Quartus позволяет писать код и компилировать его, а также проверять синтезируемые схемы и загружать программы на ПЛИС. ModelSim — это гибкое решение для тестирования схем. Тесты в данной программе описываются тем же языком что и схемы.

# РАЗРАБОТКА ОБЩЕЙ СТРУКТУРЫ МИКРО-ЭВМ

Общая структура разрабатываемой микро-ЭВМ, а также связи между ее блоками показаны на структурной схеме. Структурная схема устройства представлена в приложении А. Структурная схема состоит из:

1. ПЗУ;

2. ОЗУ;

2. Устройство управления(УУ);

3. Арифметико-логическое устройство(АЛУ);

4. Стек;

5. Регистры общего назначения(РОН).

6. Арбитр шин.

7. Регистры конвеера.

Все блоки управляются из устройства управления, блок управления связан со всеми переферийными устройствами, такими как: РОН, стек, АЛУ, ОЗУ, ПЗУ. Блок управления пересылает управляющие сигналы, и, в зависимости от исполняемой комманды, организует управление перечисленными выше устройствами.

## Функциональный состав микро-ЭВМ

## Разработка системы команд

В таблице 1.1 представлена реализуемая архитектура системы команд (АСК). Для кодирования одной команды необходимы 2 ячейки памяти по 14 бит.

Таблица 1.1 - Архитектура системы команд

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Первое слово | | | | Второе слово | |
| Первый вариант | | | | | |
| 13-10 | 9-6 | 5-0 | | 13-8 | 7-0 |
| КОП | Регистр 1 | Адрес (ст. ч.) | | Адрес (мл. ч.) | Пусто |
| Другой вариант | | | |
| 5-2 | 1-0 | 13-12 | 11-0 |
| Регистр 2 | Смещение (ст. ч.) | Смещение (мл. ч.) | Адрес |

КОП – код операции;

В команде mov reg, addr (прямая регистровая/адресная) используется оба слова команды, с 13 по 10 биты первого слова содержат код операции, биты с 9 по 6 первого слова содержат адрес регистра, который является источником данных. Биты с 5 по 0 первого слова — старший адрес ячейки памяти, которая является приемником данных или источником в случае прямой адресной команды. Биты с 13 по 8 — младший адрес ячейки памяти. Остальные биты второго слова – безразличные биты.

В команде inc reg используются в качестве адреса регистра биты с 9 по 6 первого слова. Остальные биты первого и второго слова пустые.

В коммандах xor, sra, nand, с прямой регистровой адресацией используется для обозначения адресов такие же биты как и в команде mov с прямой регистровой адресацией. В командах xor, sra, nand, c базовой индексной со смещением используется тот же принцип размещения битов, что и в команде mov с таким же типом адресации.

В команде push/pop reg используются в качестве адреса регистра биты с 9 по 6 первого слова. Остальные биты первого и второго слова пустые.

В команде jmp adr, jnz adr используются биты с 5 по 0 первого слова — старший адрес ячейки памяти и биты с 13 по 8 — младший адрес ячейки памяти. Остальные биты перового и второго слова – безразличные биты.

В команде hlt 4 старшие бита первого слова содержат код операции, остальные биты безразличные.

В таблице 1.2 представлена АСК всех команд, которые используются в данной микро-ЭВМ.

Таблица 1.2 - Система команд микро-ЭВМ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № | Комманда | Слово 1 | | Слово 2 | |
|  |  | 4 бит | 10 бит | 14 бит | |
| 0 | mov reg, addr | 0001 | 4:reg 6:addr-high | 6:addr-low | x..x |
| 1 | mov reg , addr | 0010 | 4:reg 6:addr-high | 6:addr-low | x..x |
| 2 | mov [BI+SI+offset] | 0011 | 4:BI 4:SI 2:Offset | 2:Offset | 12:addr |
| 3 | inc reg | 0100 | 4:reg | x..x | x..x |
| 4 | inc [BI+SI+offset] | 0101 | 4:BI 4:SI 2:Offset | 2:Offset | x..x |
| 5 | xor reg , addr | 0110 | 4:reg 6:addr-high | 6:addr-low | x..x |
| 6 | xor [BI+SI+offset] | 0111 | 4:BI 4:SI 2:Offset | 2:Offset | x..x |
| 7 | nand reg , addr | 1000 | 4:reg 6:addr-high | 6:addr-low | x..x |
| 8 | nand [BI+SI+offset] | 1001 | 4:BI 4:SI 2:Offset | 2:Offset | x..x |
| 9 | sra reg , addr | 1010 | 4:reg 6:addr-high | 6:addr-low | x..x |
| 10 | sra [BI+SI+offset] | 1011 | 4:BI 4:SI 2:Offset | 2:Offset | x..x |
| 11 | push | 1100 | 4:reg | x..x | x..x |
| 12 | pop | 1101 | 4:reg | x..x | x..x |
| 13 | jnz | 1110 | 6:addr-high | 6:addr-low | x..x |
| 14 | jmp | 1111 | 6:addr-high | 6:addr-low | x..x |
| 15 | hlt | 0000 | x..x | x..x | x..x |

## Описание взаимодействия всех блоков микро-ЭВМ при выполнении команд программы

Основными устройствами данного микро-ЭВМ являются: основная память и центральный процессор.

Блок ЦП регулирует и координирует работу блоков регистров общего назначения, стека, арифметико-логического устройства, а также внутренними блоками центрального процессора. Центральный процессор имеет прямой выход к основной памяти.

ОЗУ отвечает за хранение данных.

ПЗУ отвечает за хранение команд устройства.

Регистры общего назначения и стек являются временными хранилищами данных при выполнении микропрограммы устройства. Блок арифметико-логических операций отвечает за выполнение арифметических и логических команд реализуемых в данной микро-ЭВМ.

# РАЗРАБОТКА ОСНОВНЫХ УСТРОЙСТВ МИКРО-ЭВМ

В данном разделе подробно описывается состав и принципы работы основных блоков микро-ЭВМ.

## Функциональный состав и временные диаграммы работы ПЗУ

Блок ПЗУ описан языком Verilog в качестве обозначения ячейки пямяти используется ключевое слово reg, в нашем случае это reg [DATA\_W - 1:0] mem [0:DEPTH-1], где параметр DATA\_W это размерность одной единицы данных, а DEPTH глубина данных, другими словами размерность памяти. Так как по варианту используется аснинхронная память, то организовать работу памяти можно в структурном стилею Схема представлена в приложении Б. Листинг кода представлен ниже:

module ram #(parameter DATA\_W = 14,

ADDR\_W = 12,

DEPTH = ADDR\_W\*\*2

)(

input [DATA\_W - 1:0] data,

input [ADDR\_W - 1:0] address,

input clk,

input wren,

input rden,

output [DATA\_W - 1:0] q

);

reg [DATA\_W - 1:0] mem [0:DEPTH-1];

always @(posedge clk) begin

if(wren == 1'b1)begin

mem[address] <= data;

end

end

assign q = (rden == 1'b1 && wren == 1'b0) ? mem[address] : 14'bzzzzzzzzzzzzzz;

initial begin

$readmemh("ram.mem", mem);

end

endmodule

Блок ПЗУ имеет следующие входы и выходы, временная диаграмма представлена на рисунке 2.1:

* input [DATA\_W - 1:0] datа — входные данные.
* input [ADDR\_W - 1:0] address — адрес ячейки.
* input wren — разрешение на запись.
* input rden — разрешение на чтение.
* output [DATA\_W - 1:0] q — выхоные данные.

Временная диаграмма:

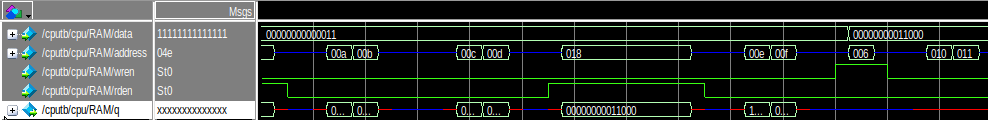


Рисунок 2.1 — Моделирование блока ПЗУ

## Функциональный состав и временные диаграммы работы ОЗУ

Блок ОЗУ описан языком Verilog в качестве обозначения ячейки пямяти используется ключевое слово reg, в нашем случае это reg [DATA\_W - 1:0] mem [0:DEPTH-1], где параметр DATA\_W это размерность одной единицы данных, а DEPTH глубина данных, другими словами размерность памяти. Так как по варианту используется аснинхронная память, то организовать работу памяти можно в структурном стиле. Схема представлена в приложении В. Листинг кода представлен ниже:

module rom #(parameter DATA\_W = 14,

ADDR\_W = 12,

DEPTH = ADDR\_W\*\*2

)(

input [ADDR\_W - 1:0] address,

input clk,

input rden,

output [DATA\_W - 1:0] data

);

reg [DATA\_W - 1:0] mem [DEPTH-1:0];

assign data = (rden) ? mem[address] : 14'bzzzzzzzzzzzzzz;

initial begin

$readmemb("rom.mem", mem);

end

endmodule

Блок OЗУ имеет следующие входы и выходы, временная диаграмма представлена на рисунке 2.2:

input [ADDR\_W - 1:0] address — адрес ячейки.

input rden — разрешение на чтение.

output [DATA\_W - 1:0] data — выхоные данные.

Временная диаграмма:

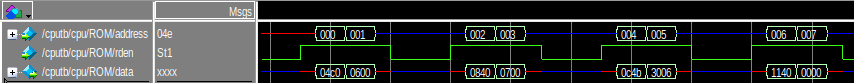


Рисунок 2.2 – Моделирование ОЗУ

## Блок стека

Блок стек имеет 12 регистров и направление вверх, по условию задания. Для работы со стеком предусмотрены 2 сигнала — push и pop, которые на шине управления имеют константное значение `OP\_PUSH[POP]\_R, равные коду комманды. По этим сигналам стек обновляет свой SP (Stack Pointer) и записывает или выдаёт значение соответственно. Листинг кода представлен ниже:

module stack #(parameter DATA\_W = 14, VOLUME = 12)(

input clk,

input [3:0] opcode,

input reset,

input [DATA\_W - 1:0] push,

output reg [DATA\_W - 1:0] pop

);

reg [DATA\_W - 1:0] data[VOLUME -1:0];

reg [VOLUME - 1:0] pointer;

always @(posedge opcode[3]) begin

if (opcode == `OP\_PUSH\_R) begin

pointer <= pointer + 1'd1;

end

if (opcode == `OP\_POP\_R) begin

pointer <= pointer - 1'd1;

end

end

always @(posedge clk) begin

if (reset) begin

pointer <= 0;

end

case(opcode)

`OP\_PUSH\_R:

data[pointer] <= push;

`OP\_POP\_R:

pop <= data[pointer];

endcase

end

endmodule

Входные и выходные пины:

* input clk — тактовый сигнал для синхронизации вывода данных.
* input [3:0] opcode — управляющий сигнал содержащий код команды.
* input reset — сигнал сброса.
* input [DATA\_W - 1:0] push — выходные данные.
* output reg [DATA\_W - 1:0] pop — входные данные.

Схема блока стека представлена в приложении Г.

Временная диаграмма:

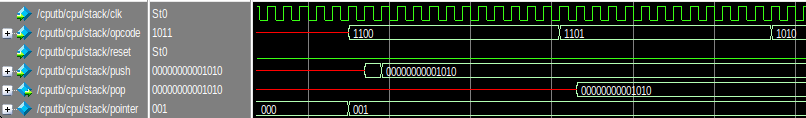


Рисунок 2.3 – Моделирование стека

## Регистры общего назначения

Блок регистров общего назначения (далее - РОН) используется для записи/чтения операндов, которые впоследствии могут использоваться для вычисления операций, либо записываться в другие регистры или память.

Сам РОН представляет из себя 16 регистров размером по 14 бит. Листинг кода представлен ниже:

`define REG\_AX 4'd0

`define REG\_BX 4'd1

`define REG\_CX 4'd2

`define REG\_DX 4'd3

`define REG\_SI 4'd4

`define REG\_DI 4'd5

`define REG\_BP 4'd6

`define REG\_SP 4'd7

// Other Registers

`define REG\_AX1 4'd8

`define REG\_AX2 4'd9

`define REG\_AX3 4'd10

`define REG\_AX4 4'd11

`define REG\_AX5 4'd12

`define REG\_AX6 4'd13

`define REG\_AX7 4'd14

`define REG\_F 4'd15

module GPR #(

parameter DATA\_W = 14, ADDR\_W = 12, REG\_N = 16, REG\_W = 4

) (

input clk,

input [ADDR\_W - 1:0] address\_in,

input [ADDR\_W - 1:0] address\_out,

input [DATA\_W - 1:0] data\_in,

output reg [DATA\_W - 1:0] data\_out,

input GPR\_rd,

input GPR\_wr

);

reg [DATA\_W - 1:0] registers[REG\_N - 1:0];

localparam reg\_offset = ADDR\_W - 1;

always @(\*) begin

if (GPR\_wr) begin

registers[address\_in[ADDR\_W-1:ADDR\_W - REG\_W]] <= data\_in;

end

else if (GPR\_rd) begin

data\_out <= registers[address\_out[ADDR\_W - 1: ADDR\_W - REG\_W]] +

registers[address\_out[ADDR\_W - 1 - REG\_W : ADDR\_W - 2 \* REG\_W]] +

registers[address\_out[ADDR\_W - 1 - 2\*REG\_W : ADDR\_W - 3 \* REG\_W]];

end

end

initial begin

$readmemh("GPR.mem", registers);

end

endmodule

Входные и выходные пины:

* input clk — тактовый сигнал.
* input [ADDR\_W - 1:0] address\_in — адресный сигнал для записи.
* input [ADDR\_W - 1:0] address\_out — адресный сигнал для чтения.
* input [DATA\_W - 1:0] data\_in — запись данных.
* output reg [DATA\_W - 1:0] data\_out — чтение данных.
* input GPR\_rd — сигнал на чтение.
* input GPR\_wr — сигнал на запись.

Принцип работы: так как работа микропроцессора организована как конвеер, для ускорения работы устройства было решено использовать отдельные шины для чтения и записи. Это позволяет не ожидать запись результата во время чтения.

Схема блока регистров общего назначения представлена в приложении Д.

Временная диаграмма:

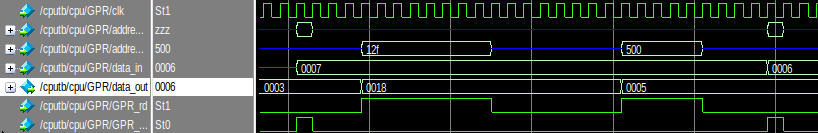


Рисунок 2.5 – Моделирование РОН

## Арифметико-логическое устройство

Блок арифметико-логического устройства должен обязательно уметь выполнять команды:

* inc – увеличение значения на 1 (`),
* xor – логическое «ИЛИ»,
* nand – обратное логическому «И»,
* sra – арифметичесикй сдвиг влево.

Листинг кода:

`include "opcodes.v"

module ALU #(parameter DATA\_W = 14)(

input [DATA\_W - 1:0] data0,

input [DATA\_W - 1:0] data1,

input [3:0] opcode,

output reg [DATA\_W - 1:0] result

);

always @(\*) begin

case (opcode)

`OP\_INC\_SR, `OP\_INC\_BIO:

begin

result <= data0 + 1'b1;

end

`OP\_NAND\_SR, `OP\_NAND\_BIO:

begin

result <= ~(data0 & data1);

end

`OP\_SRA\_BIO, `OP\_SRA\_SR:

begin

result <= data0 >>> data1;

end

`OP\_XOR\_BIO, `OP\_XOR\_SR:

begin

result <= data0 ^ data1;

end

endcase

end

endmodule

Входные и выходные пины:

* input [DATA\_W - 1:0] data0, data1 — входные данные.
* input [3:0] opcode — код комманды.
* output reg [DATA\_W - 1:0] result — результат арифметической или логической операции.

Принцип работы: на вход поступают данные data0 и data1, или только data0, одновременно с этим поступает код операции и если он соответствует операции АЛУ, то выполняются логические или арифметические действия.

Схема блока арифметико-логического устройства представлена в приложении Е.

Временная диаграмма:

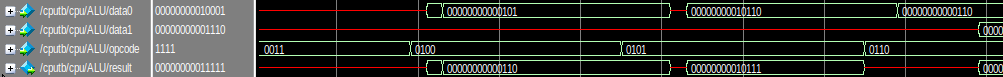


Рисунок 2.6 – Моделирование АЛУ



## Устройство управления

Одной из составных частей устройства управления (далее УУ) является блок выборки и декодирования команд. Блок делится на три стадии выполнения комманды: чтения (READ), расшифровки (DECODE) и записи (WRITE). Так как ЦПУ имеет конвеерную архитектуру, вводятся дополнительные промежуточные регистры хранения данных. Для стадии READ и DECODE вводится промежуточный блок, который хранит 2 считанных строки (reg\_rom). Между стадиями DECODE и WRITE вводится промежуточный блок который хранит данные которые подлежат записи, адрес по которому записать данные и код операции (reg\_ram).

Блок READ выполняет чтение комманд. То есть выполняет оправку разрешающих сигналов для ПЗУ и управляет записью данных в промежуточный блок. Сама реализация блока представляет собой конечный автомат который делится на две стадии: инициализации (INIT) и чтения (READ), стадия инициализации сбрасывает все сигналы, которые могут помешать работе устройства. Стадия чтения выставляет сигнал чтения и ждет разрешения от арбитра шин на выполнение данной стадии. Схема представлена в приложении Ж. Листинг кода представлен в приложении З.

module READ\_ROM #(

parameter DATA\_W = 14,

parameter ADDR\_W = 12

) (

input clk, reset, pause\_READ,

input rom\_rd\_garant,

output reg rom\_rd,

output reg command\_write,

output reg [ADDR\_W - 1:0] addr\_out,

input [ADDR\_W-1 : 0] jmp\_addr

);

reg [DATA\_W - 1:0] command;

reg [ADDR\_W - 1:0] IP;

reg [ADDR\_W - 1:0] IPtoJump;

reg [3:0] latency\_counter;

reg [7:0] state;

localparam INIT = 0, READ =1;

localparam READ\_LAT = 2;

reg wait\_sig;

always @(posedge clk) begin

if (reset) begin

latency\_counter <= 0;

IP <= 0;

rom\_rd <= 1'b0;

command\_write <= 1'b0;

state <= INIT;

wait\_sig <= 0;

end else if(pause\_READ == 1'b0) begin

if(latency\_counter != 0 && wait\_sig == 0) begin

latency\_counter <= latency\_counter - 1'b1;

end

if(jmp\_addr != 12'b111111111111 && command\_write==1) begin

IP <= jmp\_addr;

end

case(state)

INIT: begin

latency\_counter <= READ\_LAT;

state <= READ;

command\_write <= 1'b0;

end

READ: begin

if (latency\_counter == 0) begin

state <= INIT;

addr\_out <= 12'bzzzzzzzzzzzz;

rom\_rd <= 1'b0;

command\_write <= 1'b0;

end else begin

rom\_rd <= 1'b1;

if(rom\_rd\_garant == 1) begin

wait\_sig <= 0;

addr\_out <= IP;

command\_write <= 1'b1;

IP <= IP + 1;

end else begin

wait\_sig <= 1;

end

end

end

endcase

end

end

endmodule

Временная диаграмма данного блока:

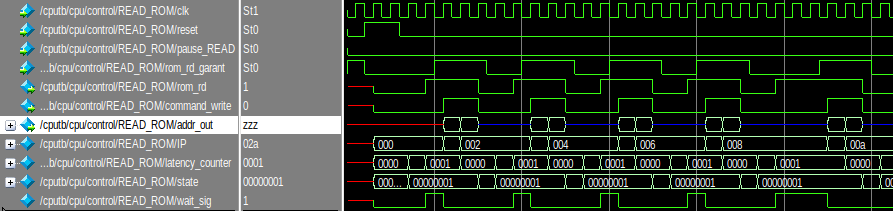


Рисунок 2.7 — Моделирование READ.

Следующий блок DECODE выполняет расшифровку команды и чтения данных, которые необходимо записать. Прочитанная команда вместе с данными записывается в следующий блок ram\_reg, который читает блок WRITE. Данный блок также работает по принципу конечного автомата и делится на множество стадий:

INIT — стадия инициаизации начального состояния автомата.

READ\_COMM — стадия чтения команды из промежуточного блока.

DECODE — расшифровка команды.

WRITE\_COMM — запись команды во внутренние регистры.

EXEC — чтение данных необходимых для далнейшей записи.

WRITE\_DATA — записб в промежуточный блок ram\_reg.

HLT — остановка выполнеия программы, до появления сигнала сброса.

Стадия EXEC делится на множество стадий чтения данных в зависимости от команды. Данная команда выполняется с задержкой в пять тактов. Данная задержка вызвана тем, что для некоторых команд необходимо такое время для выставления всех необходимых данных. Листинг программы представлен в приложении И, а схема в приложении К.

Временная диаграмма данного блока:

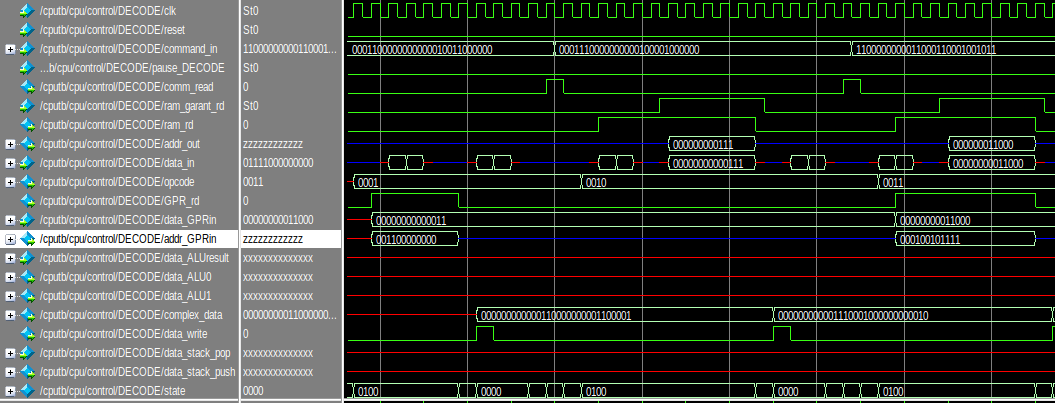


Рисунок 2.8 — Моделирование DECODE.

Следующий блок WRITE выполняет чтение данных из блока ram\_reg. Блок также является конечным автоматом, и имеет 4 стадии:

INIT — стадия инициаизации начального состояния автомата.

READ\_DATA — стадия чтения данных из промежуточного блока.

DECODE — расшифровка данных.

WR\_MEM — запись команды в ПЗУ, стек или РОН.

Листинг кода представлен в приложени Л, а схема в М.

Временная диаграмма:

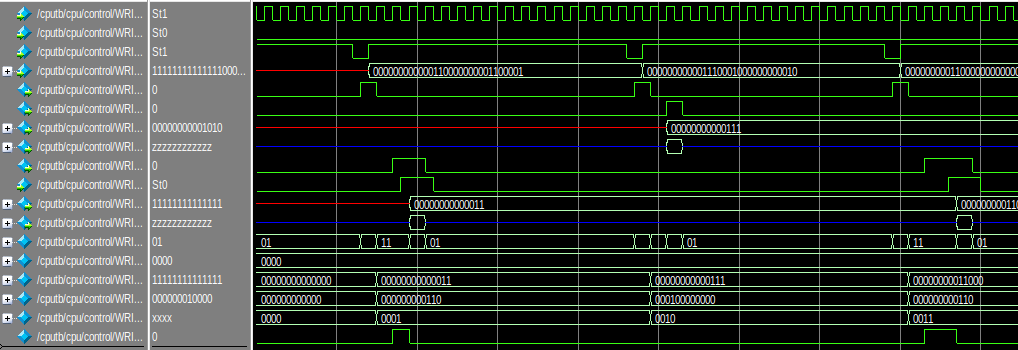


Рисунок 2.8 — Моделирование WRITE.

Промежуточные блоки по своей архитектуре похожи на очередь. Записанные данные помечаются битом 1 в конце, соответсвенно данные, которые прочитаны помечаются битом 0. Каждый блок по принципу работы похож друг на друга. Отличие состоит только в том какие данные хранит тот или иной блок. Листинг кода приведен в приложении Н, а схема в приложении С.

Временная диаграмма reg\_ram и reg\_rom:

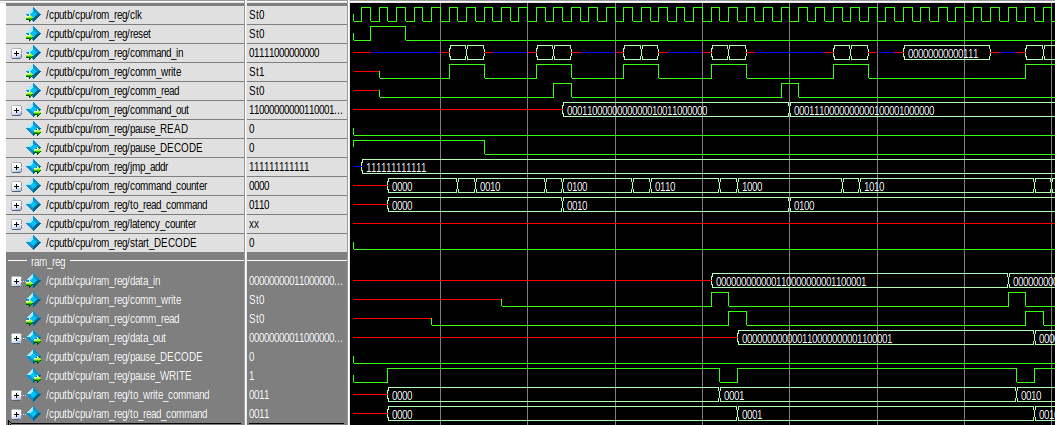


Рисунок 2.9 — Моделирование reg\_ram и read\_rom.

## Арбитаж шин

Арбитаж шин необходим чтобы при одновременном обращении к шине данных или адресу не возникало конфликтов и, следовательно, ошибочных данных. В соответствии с вариантом был выполнен децентрализованный паралельный арбитаж. Данный блок представляет надстройку над ПЗУ и ОЗУ и решает конфликт на шине адреса для трех управляющих сигналов: чтения ОЗУ, чтения ПЗУ и записи в ПЗУ. Для их выставлены приоритеты 1, 2, 3 (наименьший приоритет, команда выполнится последней). Так как для устройства управления выставлены задержки для операций чтения и записи, необходимо применить регистр который будет хранить текущее состояние арбитра и не давать возникнуть конфликтам между сигналами управления, например, сигнал записи данных не должен прерваться сигналом с более высоким приоритетом. Схема представлена в приложении Т. Листинг кода представлен ниже:

module mem\_resolver (

input clk, reset,

input rom\_rd,

input ram\_rd,

input ram\_wr,

output reg rom\_garant,

output reg ram\_garant\_rd,

output reg ram\_garant\_wr

);

reg [1:0] busy\_state;

initial begin

busy\_state <= 0;

rom\_garant <=0;

ram\_garant\_rd <=0;

ram\_garant\_wr <=0;

end

always @(negedge clk) begin

if(reset) begin

busy\_state <= 0;

end else begin

if(busy\_state == 2'b00) begin

casex ({rom\_rd, ram\_rd, ram\_wr})

3'b1xx: begin

busy\_state <= 2'd1;

rom\_garant <= 1;

end

3'bx1x: begin

busy\_state <= 2'd2;

ram\_garant\_rd <= 1;

end

3'bxx1: begin

busy\_state <= 2'd3;

ram\_garant\_wr <= 1;

end

endcase

end

if(busy\_state != 2'b00) begin

casex ({rom\_rd, ram\_rd, ram\_wr, busy\_state})

5'b0xx01: begin

rom\_garant <= 0;

busy\_state <= 2'b00;

end

5'bx0x10: begin

ram\_garant\_rd <= 0;

busy\_state <= 2'b00;

end

5'bxx011: begin

ram\_garant\_wr <= 0;

busy\_state <= 2'b00;

end

endcase

end

end

end

endmodule

Временная диаграмма:

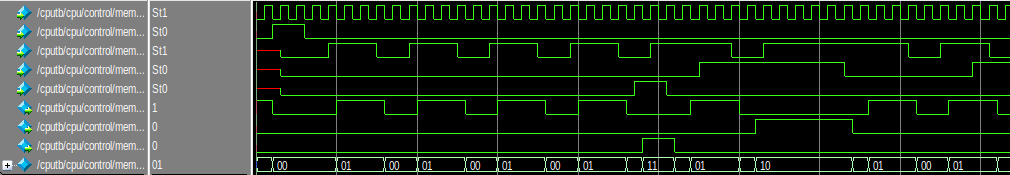


Рисунок 2.10 — Моделирование арбитра шин.

# ФУНКЦИОНАЛЬНОЕ МОДЕЛИРОВАНИЕ

Общая схема разработанной микро-ЭВМ представлена в приложении У. Для общего моделирования системы была разработана следующая микропрограмма, включающая в себя все команды разработанной микро-ЭВМ.

Таблица 3.1 – Микропрограмма для моделирования

|  |  |  |
| --- | --- | --- |
| Номер ячейки ПЗУ | Запись на языке ассемблера | Запись в бинарном виде |
| 0 | Mov reg3, 0x6 | 00010011000000 |
| 1 | 00011000000000 |
| 2 | Mov 0x8, reg1 | 00100001000000 |
| 3 | 00011100000000 |
| 4 | Mov reg1+reg2+4, 0x7 | 00110001001011 |
| 5 | 11000000000111 |
| 6 | inc reg5 | 01000101000000 |
| 7 | 00000000000000 |
| 8 | inc reg3+reg7+2, 0x0 | 01010011011111 |
| 9 | 00000000000000 |
| A | xor reg5, 0xb | 01100101000000 |
| B | 00101100000000 |
| C | xor reg5+reg3+2, 0xe | 01110101001110 |
| D | 00000000001110 |
| E | jnz 0x1e | 11110000000000 |
| F | 01111000000000 |
| 1E | push reg10 | 11001010000000 |
| 1F | 00000000000000 |
| 20 | pop reg13 | 11011101000000 |
| 21 | 00000000000000 |
| 22 | sra reg12, 0xf | 10101100000000 |
| 23 | 00111100000000 |
| 24 | sra reg4+reg2+2, 0xa | 10110100001010 |
| 25 | 00000000001010 |
| 26 | nand reg4, 0x11 | 10000100000000 |
| 27 | 01000100000000 |
| 28 | nand reg5+reg2+2, 0x9 | 10010101001010 |
| 29 | 00000000001001 |
| 2a | Jmp 0x3c | 11100000000000 |
| 2b | 11110000000000 |

Продолжение таблицы 3.1

|  |  |  |
| --- | --- | --- |
| 3c | hlt | 00000000000000 |
| cb | 00000000000000 |

Дамп памяти ПЗУ представлен в листинге кода ниже:

@0 00010011000000

@1 00011000000000

@2 00100001000000

@3 00011100000000

@4 00110001001011

@5 11000000000110

@6 01000101000000

@7 00000000000000

@8 01010011011111

@9 00000000000000

@A 01100101000000

@B 00101100000000

@C 01110101001110

@D 00000000001110

@E 11110000000000

@F 01111000000000

@1E 11001010000000

@1F 00000000000000

@20 11011101000000

@21 00000000000000

@22 10101100000000

@23 00101100000000

@24 10110101001111

@25 00000000001010

@26 10000100000000

@27 01000100000000

@28 10010101001010

@29 00000000001011

@2a 11100000000000

@2b 11110000000000

@3c 00000000000000

@3b 00000000000000

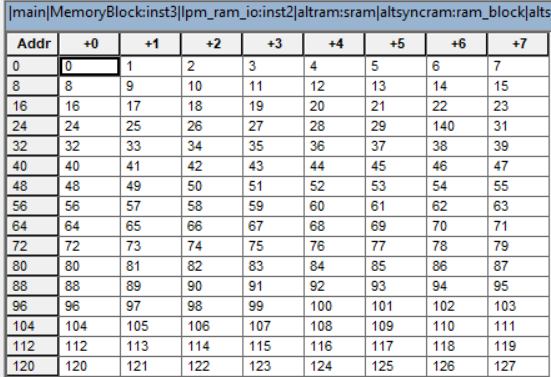
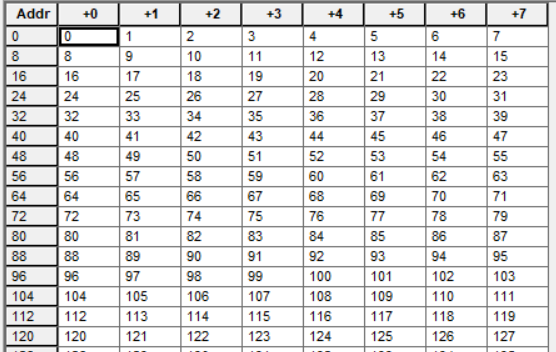


Рисунок 3.1 – Дампы памяти ОЗУ: a – до моделирования; б – после моделирования

На следующих рисунках изображены фрагменты общей временной диаграммы, отражающие выполнение команд микропрограммы.

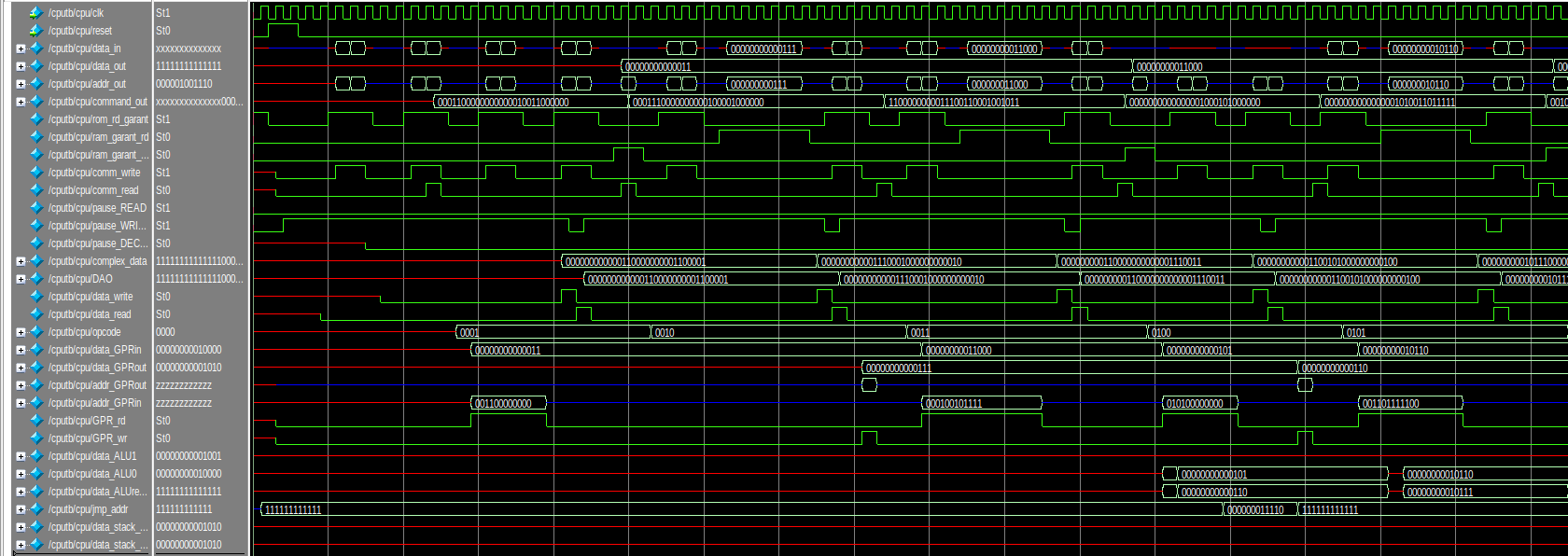
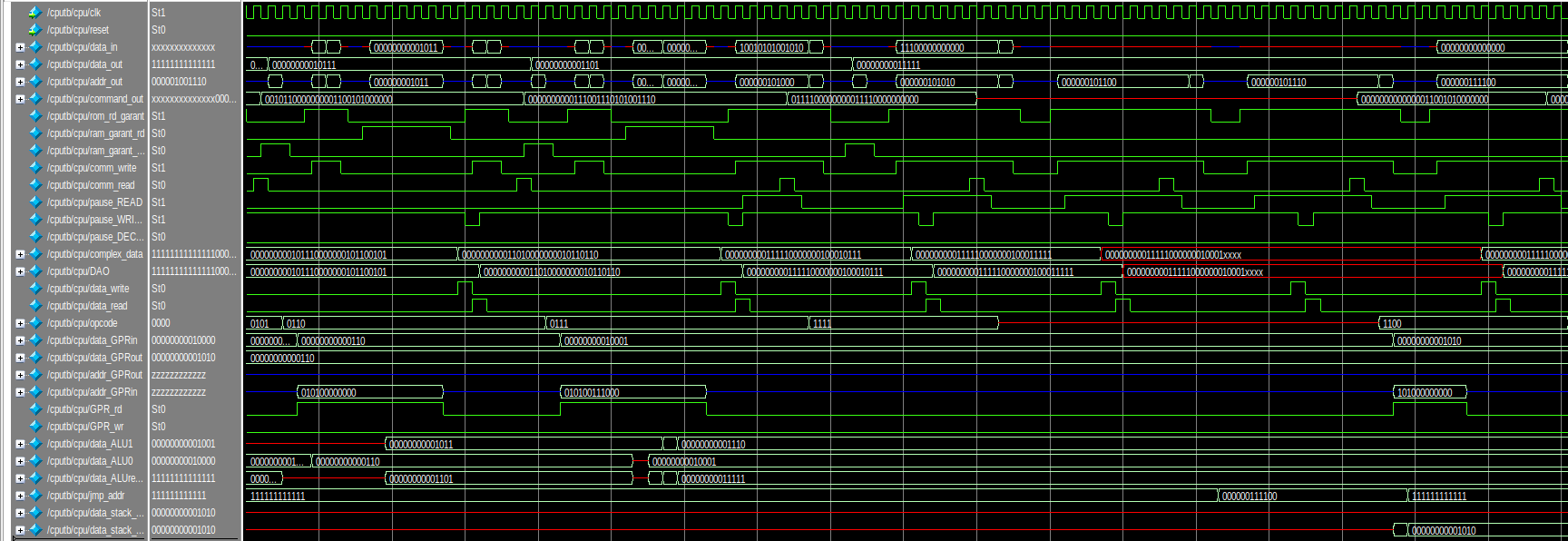


Рисунок 3.3 – Общее моделирование: команды c 1 по 5

Рисунок 3.4 – Общее моделирование: команды с 6 по 9

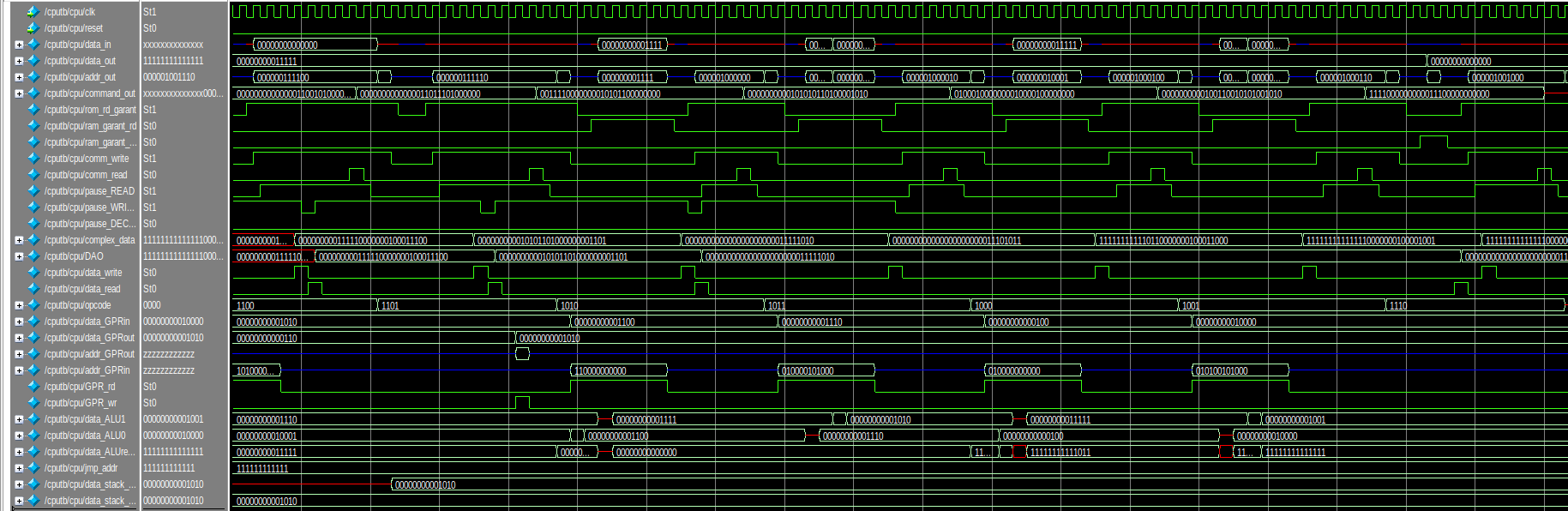


Рисунок 3.5 – Общее моделирование: команды с 9 по 15

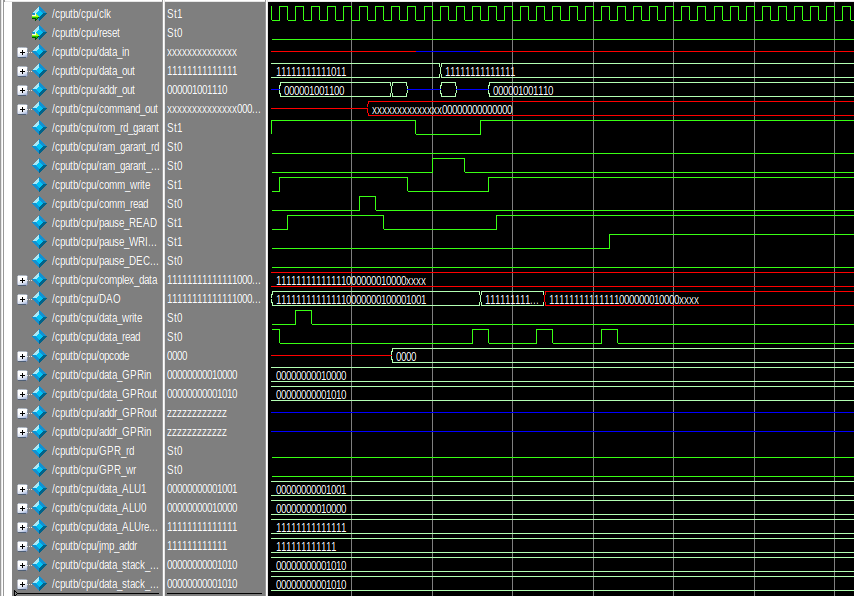


Рисунок 3.6 – Общее моделирование: команда hlt

# ЗАКЛЮЧЕНИЕ

В данном курсовом проекте была разработана микро-ЭВМ с гарвардской архитектурой, которая предполагает наличие общего адресного пространства для команд и данных. Блок памяти содержит модули ПЗУ и ОЗУ размером по 4096 байт. Также разработанное устройство содержит 16 четырнадцатиразрядных регистров общего назначения. Также в устройстве расположены стек, состоящий из 5 регистров, с направлением роста вверх и арифметико-логическое устройство. Разработанное АЛУ позволяет выполнять 4 команды: INC, XOR, NAND, SRA.

Было проведено функциональное моделирование отдельных блоков, а также всей системы с помощью разработанной для этого микропрограммы, содержащей все реализованные команды. Результаты моделирования свидетельствуют о том, что все команды выполняются верно.

Разработанная система может быть расширена путем добавления кэш-памяти, предсказателя переходов и контроллера прямого доступа к памяти (КПДП).