**Многотактное процессорное ядро**

Попробуем немного улучшить предыдущий вариант процессорного ядра, проведя повторную ревизию модулей ядра, проанализировав систему команд и возможные преобразования сигналов и данных, которые необходимо будет сделать.

Подход к построению многотактного процессорного ядра выбран не совсем классический, введем изначально некоторое количество т.н. «неархитектурных регистров» которые будут хранить промежуточные состояния.

И так, еще раз наши микроархитектурные блоки однотактного варианта:

**rv\_pc** – программный счетчик;

**rv\_mem** – блок памяти (программная и оперативная;

**rv\_desh** – дешифратор команд (слова-инструкции);

**rv\_imm** – формирователь непосредственного значения из слова-интсрукции;

**rv\_reg\_file** – файл-регистр

**rv\_ops\_mux** – коммутатор операндов для АЛУ;

**rv\_cmp** – формирователь сигнала разрешения перехода;

**rv\_alu\_v** – АЛУ;

**rv\_rez\_mux** – коммутатор результатов;

**rv\_csr** – блок регистров специального назначения.

Пробуем рассмотреть возможные сигналы и действия на каждом из этапов. Промежуточные результаты каждого из этапов фиксируются в неархитектурных регистрах. Да, таким образом количество задействованных ресурсов возрастает, но в качестве упражнения такой подход может иметь место.

Предлагается разбиение функциональности на несколько уровней:

L0 – выборка текущего значения программного счетчика (точнее пары значений – PC и РС+4).

L1 – выборка слова-инструкции из памяти.

L2 – дешифрация инструкции – выделение адресов операндов, типа инструкции, формирование управляющих сигналов.

L3 – выборка операндов из файл-регистров (в рассматриваемой микроархитектуре – это файл-регистр и блок регистров специальных функций).

L4 – работа АЛУ.

L5 – чтение/запись данных из оперативной памяти.

L6 – запись данных в файл-регистры и нового значения в программный счётчик.

Вся структура управляется кольцевым счетчиком поочередно разрешающим работу модулей каждого из уровней. Для этого модули процессорного ядра должны быть дополнены сигналами разрешения работы (особенно модули с элементами памяти). Такой модификации подвергнутся – программный счетчик, добавим разрешение на запись в файл-регистр (запись результата в порт Rd/регистра по адресу rd), разрешение на запись в блок регистров специальных функций.

«Новый» программный счетчик:

module rv\_pc

#(

parameter WIDTH=32

)

(

  input clk,

  input rst\_n,

  input en,

  input pc\_load,

  input [WIDTH-1:0] pc\_next,

  output [WIDTH-1:0] pc,

  output reg [WIDTH-1:0] pc\_plus

);

reg [WIDTH-1:0] pc\_reg;

always@(posedge clk or negedge rst\_n)

  begin

    if(!rst\_n)

      pc\_reg   <= 32'h00;

    else

    if (en) begin

      if(pc\_load==1'b1)

        pc\_reg <= pc\_next;

      else

        pc\_reg <= pc\_reg + 3'h4;

      end

    end

  //end///\*

assign pc = pc\_reg;

always @ \*

  begin

  pc\_plus <= pc\_reg + 3'h4;

  end

// \*/

endmodule

Модификация файл-регистра:

module rv\_reg\_file

#(

  parameter DATA\_WIDTH=32,

  parameter ADDR\_WIDTH=5

  )

( input clk,

  input [(ADDR\_WIDTH-1):0] rs1,

  input [(ADDR\_WIDTH-1):0] rs2,

  input [(ADDR\_WIDTH-1):0] rd,

  output reg [(DATA\_WIDTH-1):0] Rs1\_out,

  output reg [(DATA\_WIDTH-1):0] Rs2\_out,

  input [(DATA\_WIDTH-1):0] Rd\_input,

  input we,

  input en

);

  reg [DATA\_WIDTH-1:0] ram[0:2\*\*ADDR\_WIDTH-1];

  wire rd\_nonzero;

  wire rs1\_nonzero;

  wire rs2\_nonzero;

  assign rd\_nonzero = |rd;

  assign rs1\_nonzero = |rs1;

  assign rs2\_nonzero = |rs2;

always @ (posedge clk)

  begin

    if (en & we & rd\_nonzero) ram[rd] <= Rd\_input;

  end

always @ (\*)

  begin

    Rs1\_out <= rs1\_nonzero ? ram[rs1] : 32'h0;

    Rs2\_out <= rs2\_nonzero ? ram[rs2] : 32'h0;

  end

endmodule

И блок регистров специального назначения (пока это просто код-заглушка):

module rv\_csr

#(

  parameter DATA\_WIDTH=32,

  parameter ADDR\_WIDTH=12,

  parameter CSR\_size = 32

  )

( input clk,

  input [(ADDR\_WIDTH-1):0] csr\_addr,

  input [(DATA\_WIDTH-1):0] csr\_in,

  output reg [(DATA\_WIDTH-1):0] csr\_out,

  input csr\_wr,

  input en

);

// csr register file

reg [ADDR\_WIDTH-1:0] csr\_reg[0:CSR\_size-1];

always @ (posedge clk)

begin

  case (csr\_addr)

    32'h0 : begin

      if (csr\_wr&en) begin

        csr\_reg[0] <= csr\_in;

      end

      else begin

        csr\_out <= csr\_reg[0];

      end

    end

    default: begin

      csr\_out<=32'h0;

    end

  endcase

end

endmodule

Уровни «разделяются» промежуточными регистрами (обычные регистры хранения), которые фиксируют данные (регистры R0 - R7). Ряд сигналов просто «пробрасываются» по этапам (напрямую между регистрами) в том случае, если они не задействованы в них, но требуются на последующих.

**L0 – выборка текущего значения программного счетчика**

Текущее значение программного счетчика и его инкремента (PC+4) запоминаются в регистре R1. Выходными линиями, соответствующими PC, адресуется память (память программ). Также пара PC, PC+4 с выхода R1 будет подана на вход регистра R2. Программному счетчику добавлен вход разрешения работы (en).

**L1 – выборка слова-инструкции из памяти**

В регистре R2 фиксируется слово инструкции из памяти и текущие значения пары PC, PC+4. Выход регистра подается на декодер инструкции и на схемы формирования непосредственного значения (immediate). Значения счетчиков передаются на вход следующего регистра (R3).

|  |  |
| --- | --- |
|  | wire [31:0]l0r1\_PC;  wire [31:0]l0r1\_PCplus;  wire [31:0]r1l1\_PC;  wire [31:0]r1l1\_PCplus;  rv\_pc pc( // programm counter    .clk(clk),    .rst\_n(rst),    .en(en\_level[6]),    .pc\_load(r6l6\_PC\_load),    .pc\_next(r6l6\_Rez),    .pc(l0r1\_PC),    .pc\_plus(l0r1\_PCplus)  );  wire [255:0] r1\_in;  assign r1\_in = {l0r1\_PC, l0r1\_PCplus, 192'h0};  wire [255:0] r1\_out;  wire [191:0] r1\_null;  assign {r1l1\_PC, r1l1\_PCplus, r1\_null} = r1\_out;  rv\_r\_reg R1(    .clk(clk),    .rst\_n(rst),    .en(en\_level[0]),    .r\_in(r1\_in),    .r\_out(r1\_out)  ); |

Рис. 1 Выборка текущего значения программного счетчика.

Адрес инструкции на память подается со сдвигом на два бита – таким образом учитывается, что реализованная в примере микроархитектура использует 32-битную память.

|  |  |
| --- | --- |
|  | wire [31:0] l1r2\_iw;  wire [31:0] l1r2\_PC;  wire [31:0] l1r2\_PCplus;  rv\_mem mem( // system memory    .clk(clk),    //.i\_addr(32'h0), //(r1l1\_PC),    .i\_addr(r1l1\_PC>>2),    .code\_out(l1r2\_iw),    .d\_addr(r5l5\_Rez>>2),    .d\_out(l5r6\_Mem\_data),    .d\_in(r5l5\_Rs2\_reg),    .we(r5l5\_mem\_wr),    .en(en\_level[5])  );  // L1\_R2  wire [31:0]r2l2\_iw;  wire [31:0]r2l2\_PC;  wire [31:0]r2l2\_PCplus;  wire [255:0] r2\_in;  wire [255:0] r2\_out;  assign r2\_in = {l1r2\_iw, r1l1\_PC, r1l1\_PCplus, 160'h0};  wire [159:0] r2\_null;  assign {r2l2\_iw, r2l2\_PC, r2l2\_PCplus, r2\_null} = r2\_out;  rv\_r\_reg R2(    .clk(clk),    .rst\_n(rst),    .en(en\_level[1]),    .r\_in(r2\_in),    .r\_out(r2\_out)  ); |

Рис. 2 Выборка слова-инструкции из памяти.

**L2 – дешифрация инструкции**

Из слова-инструкции идет выделение адресов регистров-операндов, типа (опкодов и функциональных полей) инструкции, формирование управляющих сигналов на запись в файл-регистры (регистров общего назначения и специальных регистров).

|  |  |
| --- | --- |
|  | wire [4:0] l2r3\_rs1;  wire [4:0] l2r3\_rs2;  wire [6:0] l2r3\_op;  wire [2:0] l2r3\_fn3;  wire [6:0] l2r3\_fn7;  wire [4:0] l2r3\_rd;  wire l2r3\_rd\_wr;  wire l2r3\_mem\_wr;  wire l2r3\_csr\_wr;  wire [31:0] l2r3\_imm;  wire [31:0] l2r3\_PC;  wire [31:0] l2r3\_PCplus;  rv\_desh desh( // instruction decoder    .inst(r2l2\_iw),    .opcode(l2r3\_op),    .rs1(l2r3\_rs1),    .rs1\_en(),    .rs2(l2r3\_rs2),    .rs2\_en(),    .rd(l2r3\_rd),    .rd\_wr(l2r3\_rd\_wr),    .funct3(l2r3\_fn3),    .f3\_en(),    .funct7(l2r3\_fn7),    .f7\_en(),    .mem\_en(),    .mem\_wr(l2r3\_mem\_wr),    .csr\_en(),    .csr\_wr(l2r3\_csr\_wr),    .pc\_load()  );  rv\_imm immed(  // immediate decoder    .inst(r2l2\_iw),    .imm(l2r3\_imm)  );  wire [4:0] r3l3\_rs1;  wire [4:0] r3l3\_rs2;  wire [6:0] r3l3\_op;  wire [2:0] r3l3\_fn3;  wire [6:0] r3l3\_fn7;  wire [4:0] r3l3\_rd;  wire r3l3\_rd\_wr;  wire r3l3\_mem\_wr;  wire r3l3\_csr\_wr;  wire [31:0] r3l3\_imm;  wire [31:0] r3l3\_PC;  wire [31:0] r3l3\_PCplus;  wire [255:0] r3\_in;  wire [255:0] r3\_out;  assign r3\_in = {l2r3\_rs1, l2r3\_rs2, l2r3\_op, l2r3\_fn3, l2r3\_fn7, l2r3\_rd, l2r3\_rd\_wr,      l2r3\_mem\_wr, l2r3\_csr\_wr, l2r3\_imm,  r2l2\_PC, r2l2\_PCplus, 125'h0};  wire [124:0] r3\_null;  assign {r3l3\_rs1, r3l3\_rs2, r3l3\_op, r3l3\_fn3, r3l3\_fn7, r3l3\_rd, r3l3\_rd\_wr,      r3l3\_mem\_wr, r3l3\_csr\_wr, r3l3\_imm,  r3l3\_PC, r3l3\_PCplus, r3\_null} = r3\_out;  rv\_r\_reg R3(    .clk(clk),    .rst\_n(rst),    .en(en\_level[2]),    .r\_in(r3\_in),    .r\_out(r3\_out)  ); |

Рис. 3 Дешифровка инструкции.

**L3 – выборка операндов из файл-регистров**

Адреса регистров-операндов подаются на адресные входы файл-регистров (основного и регистров специальных функций). Оставшиеся адрес регистра-приёмника, числовая константа, программный счетчик, сигналы разрешения записью поступают на входы регистра R4.

Выходы Rs1, Rs2 и выбранного CSR также записываются в регистр R4.

|  |  |
| --- | --- |
|  | wire [31:0] l3r4\_Rs1\_reg;  wire [31:0] l3r4\_Rs2\_reg;  wire [31:0] l3r4\_CSR;  rv\_reg\_file reg\_file(   // register's file    .clk(clk),    .rs1(r3l3\_rs1),    .rs2(r3l3\_rs2),    .rd(r6l6\_rd),    .Rs1\_out(l3r4\_Rs1\_reg),    .Rs2\_out(l3r4\_Rs2\_reg),    .Rd\_input(Rd\_reg),    .we(r6l6\_rd\_wr),    .en(en\_level[6])  );  rv\_csr csr(    .csr\_addr(r3l3\_imm[11:0]),    .csr\_in(r6l6\_Rez),    .csr\_out(l3r4\_CSR),    .csr\_wr(r6l6\_csr\_wr),    .en(en\_level[3])  );  assign Data\_out = l3r4\_CSR; //CSR;  wire [31:0] r4l4\_Rs1\_reg;  wire [31:0] r4l4\_Rs2\_reg;  wire [6:0] r4l4\_op;  wire [2:0] r4l4\_fn3;  wire [6:0] r4l4\_fn7;  wire [4:0] r4l4\_rd;  wire r4l4\_rd\_wr;  wire r4l4\_mem\_wr;  wire r4l4\_csr\_wr;  wire [31:0] r4l4\_imm;  wire [31:0] r4l4\_PC;  wire [31:0] r4l4\_PCplus;  wire [31:0] r4l4\_CSR;  wire [255:0] r4\_in;  wire [255:0] r4\_out;  assign r4\_in = {l3r4\_Rs1\_reg, l3r4\_Rs2\_reg, r3l3\_op, r3l3\_fn3, r3l3\_fn7, r3l3\_rd, r3l3\_rd\_wr,      r3l3\_mem\_wr, r3l3\_csr\_wr, r3l3\_imm,  r3l3\_PC, r3l3\_PCplus, l3r4\_CSR, 39'h0};  wire [38:0] r4\_null;  assign {r4l4\_Rs1\_reg, r4l4\_Rs2\_reg, r4l4\_op, r4l4\_fn3, r4l4\_fn7, r4l4\_rd, r4l4\_rd\_wr,      r4l4\_mem\_wr, r4l4\_csr\_wr, r4l4\_imm,  r4l4\_PC, r4l4\_PCplus, r4l4\_CSR, r4\_null} = r4\_out;  rv\_r\_reg R4(    .clk(clk),    .rst\_n(rst),    .en(en\_level[3]),    .r\_in(r4\_in),    .r\_out(r4\_out)  ); |

Рис. 4 Выборка операндов из файл-регистров.

**L4 – этап работы арифметико-логического устройства( АЛУ)**

В рассматриваемой архитектуре АЛУ применяется для действий над операндами из файл-регистров, для вычисления адресов переходов, действий над считанными значениями регистров специального назначения. Из R4 на мультиплексор поступают возможные операнды, выбор двух операндов производится в зависимости от управляющих кодов (opcode, funct3, funct7).

Параллельно Rs1, Rs2 и опкоды подаются на блок разрешения переноса (по факту – компаратор с входами разрешения).

Все остальные сигналы плюс копия Rs2 транзитом подаются на вход регистра R5.

**L5 – чтение/запись данных из оперативной памяти**

На этом шаге производятся операции с памятью (памятью данных). Из регистра R5 выбираются результаты АЛУ, значение регистра Rs2, сигнал управления записью. Остальные сигналы транзитом подаются на вход регистра R6.

**L6 – запись данных в файл-регистры**.

Потенциальные результаты для записи в Rd подаются на вход мультиплексора, определяющего какое из значений будет записано в регистр Rd. Данные для записи также подаются на входы блока регистра специального назначения (CSR). Транзитом следует только новое значение программного счетчика и сигнал разрешения его модификации.

|  |  |
| --- | --- |
|  | wire [31:0] Op1;  wire [31:0] Op2;  rv\_ops\_mux ops\_mux(    .opcode(r4l4\_op),    .funct3(r4l4\_fn3),    .funct7(r4l4\_fn7),    .Rs1(r4l4\_Rs1\_reg),    .Rs2(r4l4\_Rs2\_reg),    .imm(r4l4\_imm),    .PC(r4l4\_PC),    .CSR(r4l4\_CSR),    .Op1(Op1),    .Op2(Op2)  );  rv\_cmp cmp(    .opcode(r4l4\_op),    .funct3(r4l4\_fn3),    .Rs1(r4l4\_Rs1\_reg),    .Rs2(r4l4\_Rs2\_reg),    .pc\_new(l4r5\_PC\_load)  );  rv\_alu\_v alu\_v(    .opcode(r4l4\_op),    .funct3(r4l4\_fn3),    .funct7(r4l4\_fn7),    .Op1(Op1),    .Op2(Op2),    .Rez(l4r5\_Rez)  );  wire [31:0] l4r5\_Rez;  wire l4r5\_PC\_load;  wire [31:0] r5l5\_Rez;  wire [31:0] r5l5\_Rs2\_reg;  wire [6:0] r5l5\_op;  wire [2:0] r5l5\_fn3;  wire [6:0] r5l5\_fn7;  wire [4:0] r5l5\_rd;  wire r5l5\_rd\_wr;  wire r5l5\_mem\_wr;  wire r5l5\_csr\_wr;  wire [31:0] r5l5\_imm;  wire [31:0] r5l5\_PC;  wire [31:0] r5l5\_PCplus;  wire r5l5\_PC\_load;  //L5  wire [255:0] r5\_in;  wire [255:0] r5\_out;  assign r5\_in = {l4r5\_Rez, r4l4\_Rs2\_reg, r4l4\_op, r4l4\_fn3, r4l4\_fn7, r4l4\_rd, r4l4\_rd\_wr,      r4l4\_mem\_wr, r4l4\_csr\_wr, r4l4\_imm,  r4l4\_PC, r4l4\_PCplus, l4r5\_PC\_load, 70'h0};  wire [69:0] r5\_null;  assign {r5l5\_Rez, r5l5\_Rs2\_reg, r5l5\_op, r5l5\_fn3, r5l5\_fn7, r5l5\_rd, r5l5\_rd\_wr,      r5l5\_mem\_wr, r5l5\_csr\_wr, r5l5\_imm,  r5l5\_PC, r5l5\_PCplus, r5l5\_PC\_load, r5\_null} = r5\_out;  rv\_r\_reg R5(    .clk(clk),    .rst\_n(rst),    .en(en\_level[4]),    .r\_in(r5\_in),    .r\_out(r5\_out)  ); |

Рис. 5 Подача операндов на АЛУ.

|  |  |
| --- | --- |
|  | wire [31:0] l5r6\_Mem\_data;  wire [31:0] r6l6\_Mem\_data;  wire [31:0] r6l6\_Rez;  wire [31:0] r6l6\_Rs2\_reg;  wire [6:0] r6l6\_op;  wire [2:0] r6l6\_fn3;  wire [6:0] r6l6\_fn7;  wire [4:0] r6l6\_rd;  wire r6l6\_rd\_wr;  wire r6l6\_mem\_wr;  wire r6l6\_csr\_wr;  wire [31:0] r6l6\_imm;  wire [31:0] r6l6\_PC;  wire [31:0] r6l6\_PCplus;  wire r6l6\_PC\_load;  wire [255:0] r6\_in;  wire [255:0] r6\_out;  assign r6\_in = {l5r6\_Mem\_data, r5l5\_Rez,  r5l5\_op, r5l5\_fn3, r5l5\_fn7, r5l5\_rd, r5l5\_rd\_wr,      r5l5\_mem\_wr, r5l5\_csr\_wr, r5l5\_imm,  r5l5\_PC, r5l5\_PCplus, r5l5\_PC\_load, 70'h0};  wire [69:0] r6\_null;  assign {r6l6\_Mem\_data, r6l6\_Rez, r6l6\_op, r6l6\_fn3, r6l6\_fn7, r6l6\_rd, r6l6\_rd\_wr,      r6l6\_mem\_wr, r6l6\_csr\_wr, r6l6\_imm,  r6l6\_PC, r6l6\_PCplus, r6l6\_PC\_load, r6\_null} = r6\_out;  rv\_r\_reg R6(    .clk(clk),    .rst\_n(rst),    .en(en\_level[5]),    .r\_in(r6\_in),    .r\_out(r6\_out)  );  wire [31:0] Rd\_reg;  rv\_rez\_mux rez\_mux(    .opcode(r6l6\_op),    .funct3(r6l6\_fn3),    .funct7(r6l6\_fn7),    .Rez(r6l6\_Rez),    .Pc\_plus(r6l6\_PCplus),    .Mem\_data(r6l6\_Mem\_data),    .Imm(r6l6\_imm),    .Rd(Rd\_reg)  ); |

Рис. 6 Чтение/запись данных из оперативной памяти.

|  |  |
| --- | --- |
|  | wire [31:0] Rd\_reg;  rv\_rez\_mux rez\_mux(    .opcode(r6l6\_op),    .funct3(r6l6\_fn3),    .funct7(r6l6\_fn7),    .Rez(r6l6\_Rez),    .Pc\_plus(r6l6\_PCplus),    .Mem\_data(r6l6\_Mem\_data),    .Imm(r6l6\_imm),    .Rd(Rd\_reg)  ); |

Рис.7 Запись в файл-регистры.

Общая структура процессора представлена на рис.8.

Регистры R1-R6 являются достаточно «широкими» - их разрядность с запасом взята 256 бит (да, немного расточительно с точки зрения ресурсов кристалла/FPGA, но достаточно наглядно). Кольцевой счетчик, выдающий сигналы разрешения и «широкие» регистры хранения промежуточных результатов.

|  |  |
| --- | --- |
| module rv\_ring\_reg  #(  parameter WIDTH=7  )  (    input clk,    input rst\_n,    output  [WIDTH-1:0] L\_en  );  reg [WIDTH-1:0] ring;  always@(posedge clk or negedge rst\_n)    begin      if(!rst\_n)        ring   <= 8'b00000001;      else        ring <= {ring[WIDTH-2:0],ring[WIDTH-1]};      end  assign L\_en = ring;  endmodule | module rv\_r\_reg  #(    parameter WIDTH=256    )  (    input clk,    input rst\_n,    input en,    input  [WIDTH - 1:0] r\_in,    output reg [WIDTH - 1:0] r\_out  );  always@(posedge clk or negedge rst\_n)    begin      if(!rst\_n)        r\_out <= 0;      else        if(en) begin r\_out <= r\_in; end      end  endmodule |

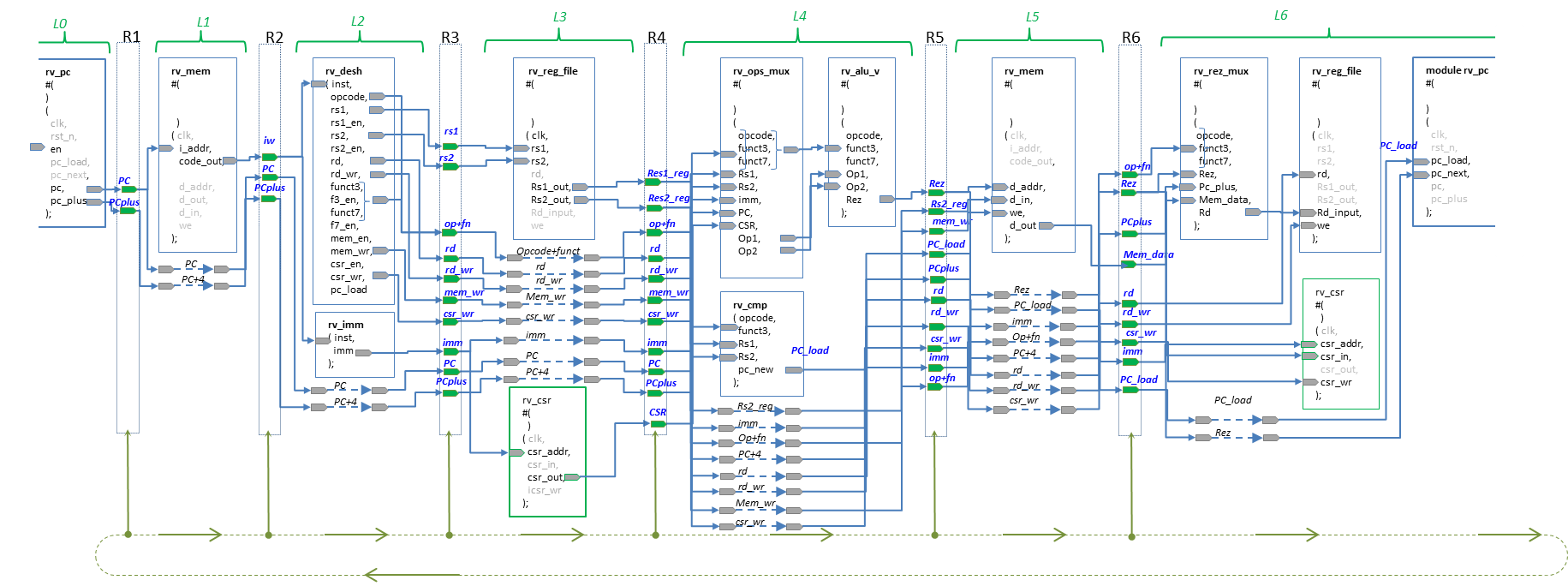


Рис. 8 Структура многотактного процессора.

Общий код процессора:

`include "rv\_pc.v"

`include "rv\_mem.v"

`include "rv\_desh.v"

`include "rv\_imm.v"

`include "rv\_reg\_file\_tst.v"

`include "rv\_ops\_mux.v"

`include "rv\_cmp.v"

`include "rv\_alu\_v.v"

`include "rv\_rez\_mux.v"

`include "rv\_csr.v"

`include "rv\_r\_reg.v"

`include "rv\_ring\_reg.v"

module rv\_cpu\_top

#(

  parameter DATA\_WIDTH=32,

  parameter ADDR\_WIDTH=32

  )

( input clk,

  input rst,

//  input [(ADDR\_WIDTH-1):0] Data\_In,

  output[(DATA\_WIDTH-1):0] Data\_out

);

// CPU modules

wire [6:0] en\_level;

rv\_ring\_reg ring\_reg(

  .clk(clk),

  .rst\_n(rst),

  .L\_en(en\_level)

);

// L0\_R1

wire [31:0]l0r1\_PC;

wire [31:0]l0r1\_PCplus;

wire [31:0]r1l1\_PC;

wire [31:0]r1l1\_PCplus;

rv\_pc pc( // programm counter

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[6]),

//  .pc\_load(r7l7\_PC\_load),

//  .pc\_next(r7l7\_Rez),

  .pc\_load(r6l6\_PC\_load),

  .pc\_next(r6l6\_Rez),

  .pc(l0r1\_PC),

  .pc\_plus(l0r1\_PCplus)

);

wire [255:0] r1\_in;

assign r1\_in = {l0r1\_PC, l0r1\_PCplus, 192'h0};

wire [255:0] r1\_out;

wire [191:0] r1\_null;

assign {r1l1\_PC, r1l1\_PCplus, r1\_null} = r1\_out;

rv\_r\_reg R1(

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[0]),

  .r\_in(r1\_in),

  .r\_out(r1\_out)

);

// R1\_L1

wire [31:0] l1r2\_iw;

wire [31:0] l1r2\_PC;

wire [31:0] l1r2\_PCplus;

rv\_mem mem( // system memory

  .clk(clk),

  //.i\_addr(32'h0), //(r1l1\_PC),

  .i\_addr(r1l1\_PC>>2),

  //.i\_addr(l0r1\_PC>>2),

  .code\_out(l1r2\_iw),

  .d\_addr(r5l5\_Rez>>2),

  .d\_out(l5r6\_Mem\_data),

  .d\_in(r5l5\_Rs2\_reg),

  .we(r5l5\_mem\_wr),

  .en(en\_level[5])

);

// L1\_R2

wire [31:0]r2l2\_iw;

wire [31:0]r2l2\_PC;

wire [31:0]r2l2\_PCplus;

wire [255:0] r2\_in;

wire [255:0] r2\_out;

assign r2\_in = {l1r2\_iw, r1l1\_PC, r1l1\_PCplus, 160'h0};

//assign r2\_in = {l1r2\_iw, l0r1\_PC, l0r1\_PCplus, 160'h0};

wire [159:0] r2\_null;

assign {r2l2\_iw, r2l2\_PC, r2l2\_PCplus, r2\_null} = r2\_out;

rv\_r\_reg R2(

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[1]),

  .r\_in(r2\_in),

  .r\_out(r2\_out)

);

// L2\_R3

wire [4:0] l2r3\_rs1;

wire [4:0] l2r3\_rs2;

wire [6:0] l2r3\_op;

wire [2:0] l2r3\_fn3;

wire [6:0] l2r3\_fn7;

wire [4:0] l2r3\_rd;

wire l2r3\_rd\_wr;

wire l2r3\_mem\_wr;

wire l2r3\_csr\_wr;

wire [31:0] l2r3\_imm;

wire [31:0] l2r3\_PC;

wire [31:0] l2r3\_PCplus;

rv\_desh desh( // instruction decoder

  .inst(r2l2\_iw),

  .opcode(l2r3\_op),

  .rs1(l2r3\_rs1),

  .rs1\_en(),

  .rs2(l2r3\_rs2),

  .rs2\_en(),

  .rd(l2r3\_rd),

  .rd\_wr(l2r3\_rd\_wr),

  .funct3(l2r3\_fn3),

  .f3\_en(),

  .funct7(l2r3\_fn7),

  .f7\_en(),

  .mem\_en(),

  .mem\_wr(l2r3\_mem\_wr),

  .csr\_en(),

  .csr\_wr(l2r3\_csr\_wr),

  .pc\_load()

);

rv\_imm immed(  // immediate decoder

  .inst(r2l2\_iw),

  .imm(l2r3\_imm)

);

wire [4:0] r3l3\_rs1;

wire [4:0] r3l3\_rs2;

wire [6:0] r3l3\_op;

wire [2:0] r3l3\_fn3;

wire [6:0] r3l3\_fn7;

wire [4:0] r3l3\_rd;

wire r3l3\_rd\_wr;

wire r3l3\_mem\_wr;

wire r3l3\_csr\_wr;

wire [31:0] r3l3\_imm;

wire [31:0] r3l3\_PC;

wire [31:0] r3l3\_PCplus;

wire [255:0] r3\_in;

wire [255:0] r3\_out;

assign r3\_in = {l2r3\_rs1, l2r3\_rs2, l2r3\_op, l2r3\_fn3, l2r3\_fn7, l2r3\_rd, l2r3\_rd\_wr,

    l2r3\_mem\_wr, l2r3\_csr\_wr, l2r3\_imm,  r2l2\_PC, r2l2\_PCplus, 125'h0};

wire [124:0] r3\_null;

assign {r3l3\_rs1, r3l3\_rs2, r3l3\_op, r3l3\_fn3, r3l3\_fn7, r3l3\_rd, r3l3\_rd\_wr,

    r3l3\_mem\_wr, r3l3\_csr\_wr, r3l3\_imm,  r3l3\_PC, r3l3\_PCplus, r3\_null} = r3\_out;

rv\_r\_reg R3(

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[2]),

  .r\_in(r3\_in),

  .r\_out(r3\_out)

);

//L3\_R4

wire [31:0] l3r4\_Rs1\_reg;

wire [31:0] l3r4\_Rs2\_reg;

wire [31:0] l3r4\_CSR;

rv\_reg\_file reg\_file(   // register's file

  .clk(clk),

  .rs1(r3l3\_rs1),

  .rs2(r3l3\_rs2),

  .rd(r6l6\_rd),

  .Rs1\_out(l3r4\_Rs1\_reg),

  .Rs2\_out(l3r4\_Rs2\_reg),

  .Rd\_input(Rd\_reg),

  .we(r6l6\_rd\_wr),

  .en(en\_level[6])

);

rv\_csr csr(

  .csr\_addr(r3l3\_imm[11:0]),

  .csr\_in(r6l6\_Rez),

  .csr\_out(l3r4\_CSR),

  .csr\_wr(r6l6\_csr\_wr),

  .en(en\_level[3])

);

assign Data\_out = l3r4\_CSR; //CSR;

wire [31:0] r4l4\_Rs1\_reg;

wire [31:0] r4l4\_Rs2\_reg;

wire [6:0] r4l4\_op;

wire [2:0] r4l4\_fn3;

wire [6:0] r4l4\_fn7;

wire [4:0] r4l4\_rd;

wire r4l4\_rd\_wr;

wire r4l4\_mem\_wr;

wire r4l4\_csr\_wr;

wire [31:0] r4l4\_imm;

wire [31:0] r4l4\_PC;

wire [31:0] r4l4\_PCplus;

wire [31:0] r4l4\_CSR;

wire [255:0] r4\_in;

wire [255:0] r4\_out;

assign r4\_in = {l3r4\_Rs1\_reg, l3r4\_Rs2\_reg, r3l3\_op, r3l3\_fn3, r3l3\_fn7, r3l3\_rd, r3l3\_rd\_wr,

    r3l3\_mem\_wr, r3l3\_csr\_wr, r3l3\_imm,  r3l3\_PC, r3l3\_PCplus, l3r4\_CSR, 39'h0};

wire [38:0] r4\_null;

assign {r4l4\_Rs1\_reg, r4l4\_Rs2\_reg, r4l4\_op, r4l4\_fn3, r4l4\_fn7, r4l4\_rd, r4l4\_rd\_wr,

    r4l4\_mem\_wr, r4l4\_csr\_wr, r4l4\_imm,  r4l4\_PC, r4l4\_PCplus, r4l4\_CSR, r4\_null} = r4\_out;

rv\_r\_reg R4(

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[3]),

  .r\_in(r4\_in),

  .r\_out(r4\_out)

);

// L4

wire [31:0] Op1;

wire [31:0] Op2;

rv\_ops\_mux ops\_mux(

  .opcode(r4l4\_op),

  .funct3(r4l4\_fn3),

  .funct7(r4l4\_fn7),

  .Rs1(r4l4\_Rs1\_reg),

  .Rs2(r4l4\_Rs2\_reg),

  .imm(r4l4\_imm),

  .PC(r4l4\_PC),

  .CSR(r4l4\_CSR),

  .Op1(Op1),

  .Op2(Op2)

);

rv\_cmp cmp(

  .opcode(r4l4\_op),

  .funct3(r4l4\_fn3),

  .Rs1(r4l4\_Rs1\_reg),

  .Rs2(r4l4\_Rs2\_reg),

  .pc\_new(l4r5\_PC\_load)

);

rv\_alu\_v alu\_v(

  .opcode(r4l4\_op),

  .funct3(r4l4\_fn3),

  .funct7(r4l4\_fn7),

  .Op1(Op1),

  .Op2(Op2),

  .Rez(l4r5\_Rez)

);

wire [31:0] l4r5\_Rez;

wire l4r5\_PC\_load;

wire [31:0] r5l5\_Rez;

wire [31:0] r5l5\_Rs2\_reg;

wire [6:0] r5l5\_op;

wire [2:0] r5l5\_fn3;

wire [6:0] r5l5\_fn7;

wire [4:0] r5l5\_rd;

wire r5l5\_rd\_wr;

wire r5l5\_mem\_wr;

wire r5l5\_csr\_wr;

wire [31:0] r5l5\_imm;

wire [31:0] r5l5\_PC;

wire [31:0] r5l5\_PCplus;

wire r5l5\_PC\_load;

//L5

wire [255:0] r5\_in;

wire [255:0] r5\_out;

assign r5\_in = {l4r5\_Rez, r4l4\_Rs2\_reg, r4l4\_op, r4l4\_fn3, r4l4\_fn7, r4l4\_rd, r4l4\_rd\_wr,

    r4l4\_mem\_wr, r4l4\_csr\_wr, r4l4\_imm,  r4l4\_PC, r4l4\_PCplus, l4r5\_PC\_load, 70'h0};

wire [69:0] r5\_null;

assign {r5l5\_Rez, r5l5\_Rs2\_reg, r5l5\_op, r5l5\_fn3, r5l5\_fn7, r5l5\_rd, r5l5\_rd\_wr,

    r5l5\_mem\_wr, r5l5\_csr\_wr, r5l5\_imm,  r5l5\_PC, r5l5\_PCplus, r5l5\_PC\_load, r5\_null} = r5\_out;

rv\_r\_reg R5(

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[4]),

  .r\_in(r5\_in),

  .r\_out(r5\_out)

);

// L6

wire [31:0] l5r6\_Mem\_data;

wire [31:0] r6l6\_Mem\_data;

wire [31:0] r6l6\_Rez;

wire [31:0] r6l6\_Rs2\_reg;

wire [6:0] r6l6\_op;

wire [2:0] r6l6\_fn3;

wire [6:0] r6l6\_fn7;

wire [4:0] r6l6\_rd;

wire r6l6\_rd\_wr;

wire r6l6\_mem\_wr;

wire r6l6\_csr\_wr;

wire [31:0] r6l6\_imm;

wire [31:0] r6l6\_PC;

wire [31:0] r6l6\_PCplus;

wire r6l6\_PC\_load;

wire [255:0] r6\_in;

wire [255:0] r6\_out;

assign r6\_in = {l5r6\_Mem\_data, r5l5\_Rez,  r5l5\_op, r5l5\_fn3, r5l5\_fn7, r5l5\_rd, r5l5\_rd\_wr,

    r5l5\_mem\_wr, r5l5\_csr\_wr, r5l5\_imm,  r5l5\_PC, r5l5\_PCplus, r5l5\_PC\_load, 70'h0};

wire [69:0] r6\_null;

assign {r6l6\_Mem\_data, r6l6\_Rez, r6l6\_op, r6l6\_fn3, r6l6\_fn7, r6l6\_rd, r6l6\_rd\_wr,

    r6l6\_mem\_wr, r6l6\_csr\_wr, r6l6\_imm,  r6l6\_PC, r6l6\_PCplus, r6l6\_PC\_load, r6\_null} = r6\_out;

rv\_r\_reg R6(

  .clk(clk),

  .rst\_n(rst),

  .en(en\_level[5]),

  .r\_in(r6\_in),

  .r\_out(r6\_out)

);

// L7

wire [31:0] Rd\_reg;

rv\_rez\_mux rez\_mux(

  .opcode(r6l6\_op),

  .funct3(r6l6\_fn3),

  .funct7(r6l6\_fn7),

  .Rez(r6l6\_Rez),

  .Pc\_plus(r6l6\_PCplus),

  .Mem\_data(r6l6\_Mem\_data),

  .Imm(r6l6\_imm),

  .Rd(Rd\_reg)

);

endmodule

Небольшая, не делающая ничего «умного» тестовая программа:

lui x3,0x5

srli x3,x3,12

lui x1,0x1

lui x4,0x7

l1:

add x5, x1, x4

addi x3, x3,5

jal x2,l1

lui x3,0x99

lui x4,0x77

beq x0,x0,l1

srli x1,x1,12

lui x2,0x0

lui x4,0x7

add x5, x1, x4

На рис.9 – она же в симуляторе RARS.

Рис. 10 – временные диаграммы исполнения кода. На ней представлены значения программного счетчика, адресных входов памяти команд, вход Rd файл-регистра, входы и выходы АЛУ, расшифровку кодов инструкции – поля opcode и funct3, funct7, значение immediate (да, они не всегда имеют значение, но при обработке будут игнорированы), сигналы разрешения работы каждого из этапов.

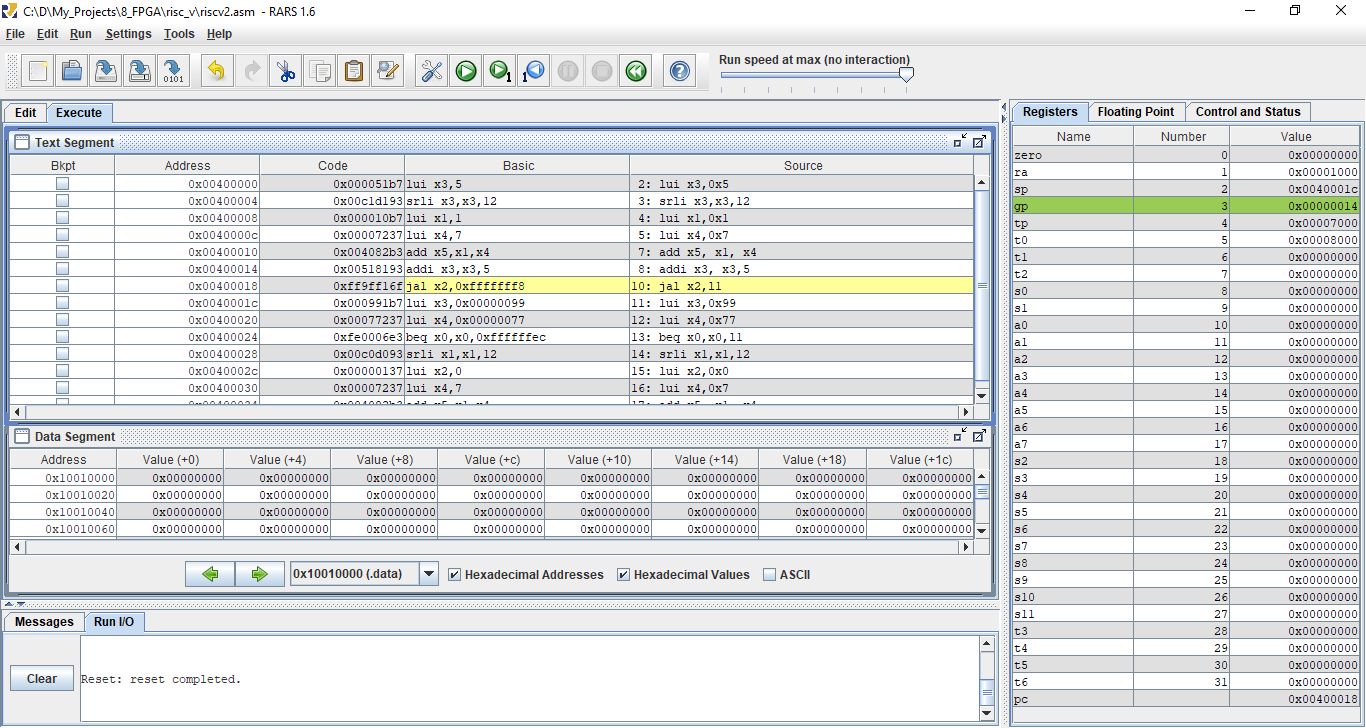


Рис. 10 Тестовая программа в симуляторе RARS.

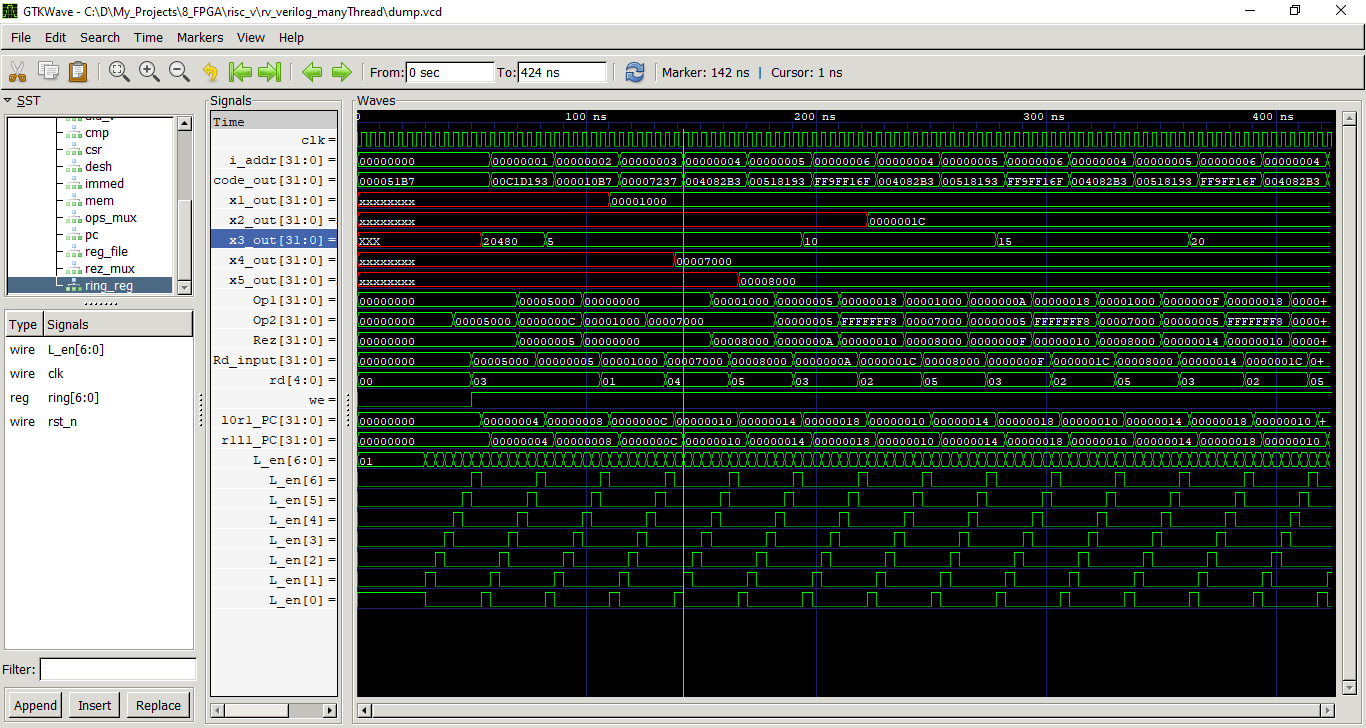


Рис. 11 Временные диаграммы исполнения программы в многотактном ядре.

На диаграммах видно выполнение математических операций, операций перехода – код доходит до адреса 1C и в соответствии с программной переходит на нулевой адрес.

Очевидные недостатки представленной микроархитектуры:

- большое количество «сквозных» многоразрядных сигналов, проходящих через все этапы и требующие для промежуточного хранения соответствующего количества бит в промежуточных регистрах (Rx).

- регистры Rx в представленном коде фиксированной разрядности, без учета реально требуемого количества бит на каждом из этапов.

PS

«Отладочный вариант» файл-регистра с дополнительными выходами, позволяющими на временных диаграммах наблюдать значения подключенных к ним выходов регистров.

module rv\_reg\_file

#(

  parameter DATA\_WIDTH=32,

  parameter ADDR\_WIDTH=5

  )

( input clk,

  input [(ADDR\_WIDTH-1):0] rs1,

  input [(ADDR\_WIDTH-1):0] rs2,

  input [(ADDR\_WIDTH-1):0] rd,

  output reg [(DATA\_WIDTH-1):0] Rs1\_out,

  output reg [(DATA\_WIDTH-1):0] Rs2\_out,

  input [(DATA\_WIDTH-1):0] Rd\_input,

  input we,

  input en,

  output reg [(DATA\_WIDTH-1):0] x1\_out,

  output reg [(DATA\_WIDTH-1):0] x2\_out,

  output reg [(DATA\_WIDTH-1):0] x3\_out,

  output reg [(DATA\_WIDTH-1):0] x4\_out,

  output reg [(DATA\_WIDTH-1):0] x5\_out

);

// RAM array

  reg [DATA\_WIDTH-1:0] ram[0:2\*\*ADDR\_WIDTH-1];

  wire rd\_nonzero;

  wire rs1\_nonzero;

  wire rs2\_nonzero;

  assign rd\_nonzero = |rd;

  assign rs1\_nonzero = |rs1;

  assign rs2\_nonzero = |rs2;

// read RAM content from file

//initial

//$readmemh("ram.txt",ram);

always @ (posedge clk)

  begin

    if (en & we & rd\_nonzero) ram[rd] <= Rd\_input;

  end

always @ (\*)

  begin

    Rs1\_out <= rs1\_nonzero ? ram[rs1] : 32'h0;

    Rs2\_out <= rs2\_nonzero ? ram[rs2] : 32'h0;

    x1\_out <= ram[1];

    x2\_out <= ram[2];

    x3\_out <= ram[3];

    x4\_out <= ram[4];

    x5\_out <= ram[5];

  end

// \*/

endmodule