**Многопоточное процессорное ядро**

На работу над многопоточным вариантом вдохновили процессоры семейства Xcore компании XMOS, специализирующейся на процессорах/микроконтроллерах для встраиваемых систем. Решения XMOS традиционно включают в себя элементы, которые традиционно требовали бы использования компонентов другого класса [1]. Там, где традиционно можно использовать микроконтроллер для управления конструкцией, DSP для обработки сигналов и, возможно, CPLD для подключения к сложному цифровому интерфейсу, процессоры XMOS могут выполнять эти три задачи в одном устройстве, используя единый программный процесс на основе программного обеспечения. Их характерные черты – аппаратная многопоточность, возможность масштабирования количества ядер/процессоров в системе, гибкие программно-конфигурируемые порты ввода-вывода. В ранних версиях своих процессоров XMOS применяли свои RISC ядра, были серии с комбинацией RISC-ядер и ARM-ядра. В 2023м году компания анонсировала вариант процессора с ядрами RISC-V (рис.1).

Вторым побуждающим мотивом экспериментов с является желание в решениях софт-ядер для FPGA уйти от необходимости введения в состав софт-процессора контроллера прерываний (кто любит обрабатывать прерывания, да еще и вложенные? Да – мало кто).

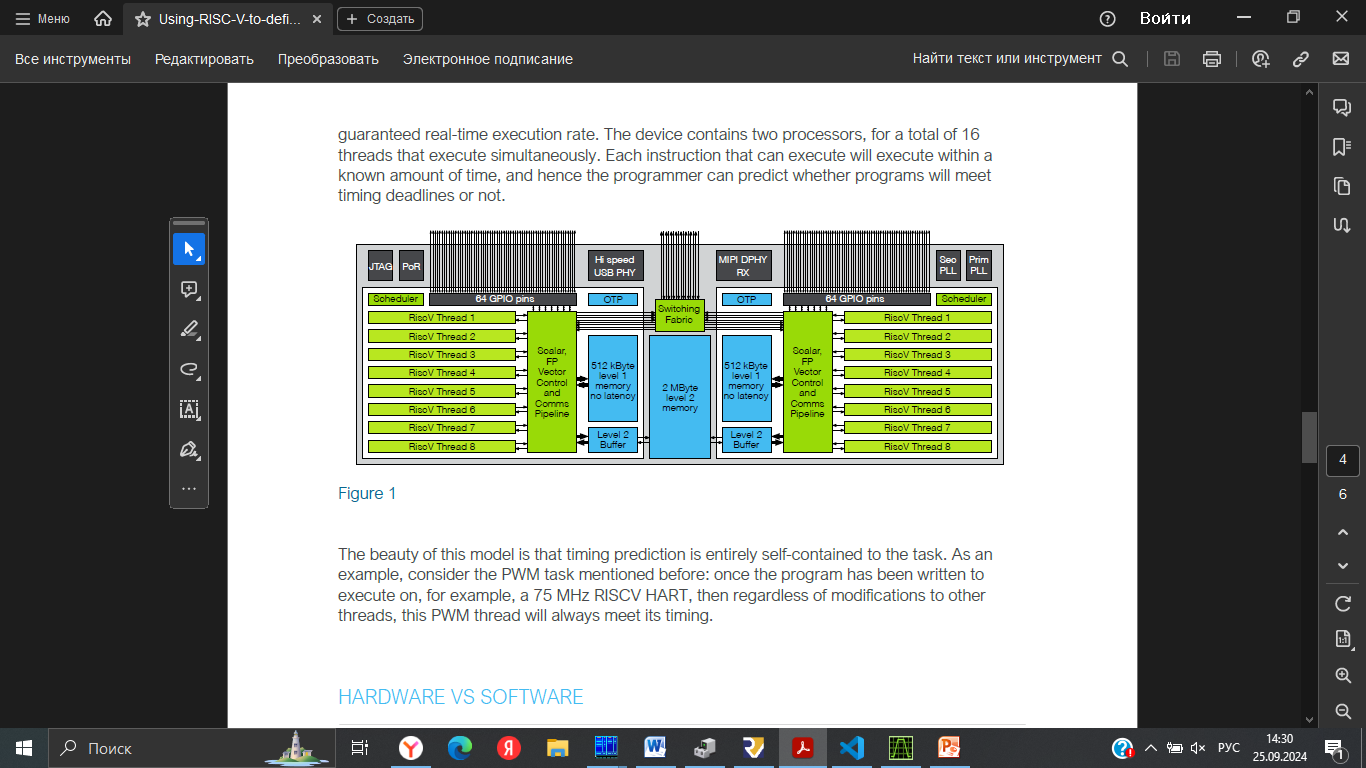


Рис. 1 Двухядерный процессор XMOS с восьмипоточными RISC-V ядрами [2].

Попробуем несколько модифицировать многотактный процессор, добавив ему поддержку многопоточности в виде «теневых» копий основных архитектурных регистров. В терминологии RISC-V аппаратно-поддерживаемый поток называется хартом (hart) [2-4]. Общая структура ядра пока останется почти прежней – рис.2.

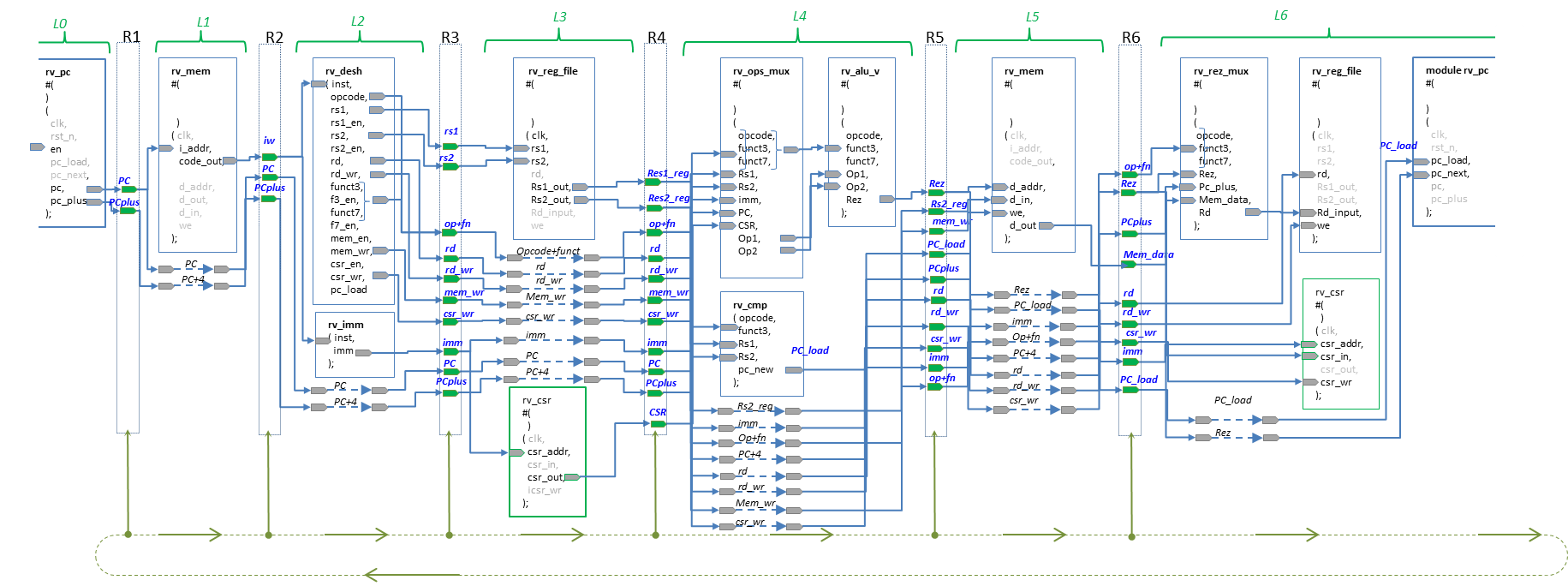


Рис. 1 Базовая структура ядра.

Микроархитектурные блоки:

**rv\_pc** – программный счетчик;

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

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

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

**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.

Ключевая идея – для каждого из хартов хранение состояния потока(нити) в массиве теневых регистров. При этом каждая из копий регистров хранит архитектурное состояние нити. Общими ресурсами остаются логические блоки, память. Также имеет смысл сохранить общим для всех хартов блок регистров специальных функций (как правило, это системные таймеры, счетчики производительности и тп).

Блок CSR-регистров в таком варианте должен будет иметь два раздельных адресных входа – на чтение его данных и на запись данных в него (из-за того, что он оставлен общим для всех хартов и операции чтения/записи осуществляются на разных этапах):

module rv\_csr

#(

  parameter DATA\_WIDTH=32,

  parameter ADDR\_WIDTH=12,

  parameter CSR\_size = 32

  )

( input clk,

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

  input [(DATA\_WIDTH-1):0] csr\_in,

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

  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\_in)

    32'h0 : begin

      if (csr\_wr&en) begin

        csr\_reg[0] <= csr\_in;

      end

    end

    default: begin

      csr\_reg[1] <= 32'h555;

    end

  endcase

  case (csr\_addr\_out)

    32'h0 : begin

      csr\_out <= csr\_reg[0];

    end

    default: begin

      csr\_out<=32'hAAA;

    end

  endcase

end

endmodule

Аналогично файл-регистр также должен иметь возможность записывать и считывать данные регистров, соотнесенных различным хартам – фактически определяем массив или, если угодно, стек файл-регистров. (для отладочных целей и бОльшей наглядности в коде далее использован вариант файл-регистра с дополнительными выходами для отслеживания состояния отдельных рнегистров).

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 [2:0] hart\_in, // hard to read out data from

  input [2:0] hart\_out, // hard to write data to

  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][0:7];

  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][hart\_in] <= Rd\_input;

  end

always @ (\*)

  begin

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

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

    x1\_out <= ram[1][hart\_out];

    x2\_out <= ram[2][hart\_out];

    x3\_out <= ram[3][hart\_out];

    x4\_out <= ram[4][hart\_out];

    x5\_out <= ram[5][hart\_out];

  end

// \*/

endmodule

Программный счетчик также будет массивом регистров.

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,

  input [2:0] hart\_in,

  input [2:0] hart\_out

);

reg [WIDTH-1:0] pc\_reg[0:7];

// read initial PCs data values

initial

$readmemh("pc.txt",pc\_reg);

always@(posedge clk or negedge rst\_n)

  begin

    if (en) begin

      if(pc\_load==1'b1)

        pc\_reg[hart\_in] <= pc\_next;

      else

        pc\_reg[hart\_in] <= pc\_reg[hart\_in] + 3'h4;

      end

    end

  //end///\*

assign pc = pc\_reg[hart\_out];

always @ \*

  begin

    pc\_plus <= pc\_reg[hart\_out] + 3'h4;

  end

// \*/

endmodule

В файле «pc.txt» - начальные значения регистров.

Как и ожидалось, «ленивый» вариант преобразования многотактного процессора в многопоточный простой заменой файл-регистра и счетчика не работоспособен, требуются некоторый механизм управления.

Для управления выборкой данных хартов введём группу регистров – *hart-reg*, объединенных в линейку. Выходные сигналы – номер харта и бит активности харта. При сбросе регистры инициируются нулями и биты активности хартов сброшены.

На линейку в процессе функционирования должна подаваться последовательность хартов с кодами активности. Сама последовательность может, например, хранится в отдельной небольшой памяти – назовем ее hart\_table. Hart\_table адресуется циклическим счетчиком.

Для текущего примера создадим простенький «системный таймер» разрядностью 64-бита (на самом деле можно любой приемлемой разрядности, в данном случае 64 бита выбрано по аналогии с традиционными таймерами-счетчиками в реализациях RISC-V архитектуры). Младшие разряды счетчика будут адресовать память таблицы хартов. Сам счетчик потом можно будет использовать в качестве счетчика тактов работы с момента старта системы, например.

Биты активности харта также должны будут использованы для управления записью в такие модули процессорного ядра, как программный счетчик, файл-регистр, запись в память (если харт не активен – запись не будет произведена). Также инверсный сигнал активности харта подается на вход «сброса конвейера» АЛУ (да, для данной реализации будет взята версия АЛУ конвейеризированного процессорного ядра).

Уровни «разделяются» промежуточными регистрами (обычные регистры хранения), которые фиксируют данные (регистры R0 – R6). Ряд сигналов просто «пробрасываются» по этапам (напрямую между регистрами) в том случае, если они не задействованы в них, но требуются на последующих – всё как и было в простом многотактном варианте.

Общая структура многопоточного процессора представлена на рис. 2.

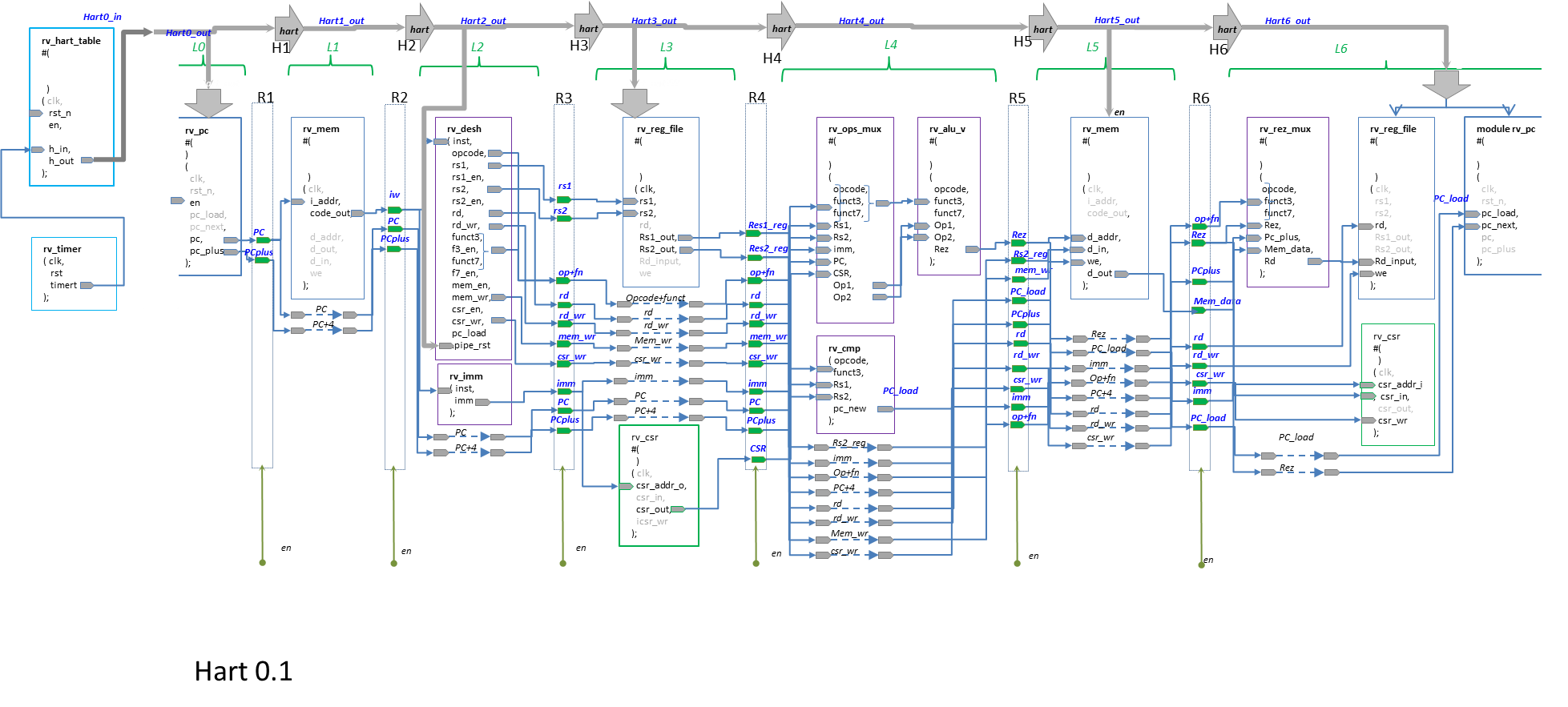


Рис. 3 Структура многопоточного процессора.

Описание памяти таблицы хартов – в текущем варианте – память на 8 записей, выходная разрядность 4 бита, старший бит – бит активности харта, младшие три – номер харта:

module rv\_hart\_table

#(

  parameter DATA\_WIDTH=4,

  parameter ADDR\_WIDTH=3

  )

( input clk,

  input [(ADDR\_WIDTH-1):0] h\_addr,

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

  input we,

  input en

);

// ROM array

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

// read ROM content from file

initial

$readmemh("hart\_table.txt",rom);

always @ (\*)

  begin

    h\_out <= rom[h\_addr];

  end

endmodule

Системный таймер:

module rv\_timer

#(

  parameter DATA\_WIDTH=64

  )

( input clk,

  input rst,

  output [(DATA\_WIDTH-1):0] timer

);

reg [(DATA\_WIDTH-1):0] cnt;

always@(posedge clk or negedge rst)

  begin

    if(!rst)

      cnt <= 64'h0;

    else

      cnt <= cnt + 1'b1;

  end

assign timer = cnt;

endmodule

Соединение таблицы хартов со счетчиком – рис.3.

|  |  |
| --- | --- |
|  | wire [63:0] tmr\_out;  rv\_timer tmr(    .clk(clk),    .rst(rst),    .timer(tmr\_out)  );  //==========================================================  rv\_hart\_table hart\_table(    .clk(clk),    .h\_addr(tmr\_out[2:0]),    .h\_out(hart0\_out)  );  //==========================================================  wire [3:0] hart0\_out; |

Рис. 4 Соединение таблицы хартов со счетчиком.

Соединение линейки регистров для хранения хартов:

wire [3:0] hart1\_out;

rv\_hart\_reg H1(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart0\_out),

  .h\_out(hart1\_out)

);

wire [3:0] hart2\_out;

rv\_hart\_reg H2(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart1\_out),

  .h\_out(hart2\_out)

);

wire [3:0] hart3\_out;

rv\_hart\_reg H3(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart2\_out),

  .h\_out(hart3\_out)

);

wire [3:0] hart4\_out;

rv\_hart\_reg H4(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart3\_out),

  .h\_out(hart4\_out)

);

wire [3:0] hart5\_out;

rv\_hart\_reg H5(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart4\_out),

  .h\_out(hart5\_out)

);

wire [3:0] hart6\_out;

rv\_hart\_reg H6(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart5\_out),

  .h\_out(hart6\_out)

);

Общий код процессора:

`include "rv\_pc.v"

`include "rv\_mem.v"

`include "rv\_desh\_pipe.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\_hart\_reg.v"

`include "rv\_timer.v"

`include "rv\_hart\_table.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 [63:0] tmr\_out;

rv\_timer tmr(

  .clk(clk),

  .rst(rst),

  .timer(tmr\_out)

);

//===============================================================

rv\_hart\_table hart\_table(

  .clk(clk),

  .h\_addr(tmr\_out[2:0]),

  .h\_out(hart0\_out)

);

// ==============================================================

wire [3:0] hart0\_out;

wire [3:0] hart1\_out;

rv\_hart\_reg H1(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart0\_out),

  .h\_out(hart1\_out)

);

wire [3:0] hart2\_out;

rv\_hart\_reg H2(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart1\_out),

  .h\_out(hart2\_out)

);

wire [3:0] hart3\_out;

rv\_hart\_reg H3(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart2\_out),

  .h\_out(hart3\_out)

);

wire [3:0] hart4\_out;

rv\_hart\_reg H4(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart3\_out),

  .h\_out(hart4\_out)

);

wire [3:0] hart5\_out;

rv\_hart\_reg H5(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart4\_out),

  .h\_out(hart5\_out)

);

wire [3:0] hart6\_out;

rv\_hart\_reg H6(

  .clk(clk),

  .rst(rst),

  .en(1'b1),

  .h\_in(hart5\_out),

  .h\_out(hart6\_out)

);

// ==============================================================

// 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(hart6\_out[3]),

  .pc\_load(r6l6\_PC\_load),

  .pc\_next(r6l6\_Rez),

  .pc(l0r1\_PC),

  .pc\_plus(l0r1\_PCplus),

  .hart\_in(hart6\_out[2:0]),

  .hart\_out(hart0\_out[2:0])

);

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(1'b1),

  .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(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(hart5\_out[3])

);

// 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(1'b1),

  .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(),

  .pipe\_rst(~hart2\_out[3])

);

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(1'b1),

  .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(hart6\_out[3]),

  .hart\_in(hart6\_out[2:0]),

  .hart\_out(hart3\_out[2:0])

);

rv\_csr csr(

  .clk(~clk),

  .csr\_addr\_out(r3l3\_imm[11:0]),

  .csr\_addr\_in(r6l6\_imm[11:0]),

  .csr\_in(r6l6\_Rez),

  .csr\_out(l3r4\_CSR),

  .csr\_wr(r6l6\_csr\_wr),

  .en(hart6\_out[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(1'b1),

  .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(1'b1),

  .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(1'b1),

  .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)

);

endmodule

Традиционно небольшая, тестовая программа:

lui x1,0x1

srli x1,x1,12

l1:

addi x1, x1, 1

jal x2,l1

lui x1,0x99

lui x1,0x2

srli x1,x1,12

l2:

addi x1, x1, 2

jal x2,l2

lui x1,0x99

lui x1,0x3

srli x1,x1,12

l3:

addi x1, x1, 3

jal x2,l3

lui x1,0x99

lui x1,0x4

srli x1,x1,12

l4:

addi x1, x1, 4

jal x2,l4

lui x1,0x99

lui x1,0x5

srli x1,x1,12

l5:

addi x1, x1, 5

jal x2,l5

lui x1,0x99

lui x1,0x6

srli x1,x1,12

l6:

addi x1, x1, 6

jal x2,l6

lui x1,0x99

lui x1,0x7

srli x1,x1,12

l7:

addi x1, x1, 7

jal x2,l7

lui x1,0x99

lui x1,0x8

srli x1,x1,12

l8:

addi x1, x1, 8

jal x2,l8

lui x1,0x99

На рис.5 – она же в симуляторе RARS.

В программе заданы несколько коротких бесконечных цикла, работающих с регистрами х1, х2. Регистр х1 инициируется определенным значением, регистр х2 будет использован, как регистр связи. При простом линейном исполнении программа зациклится на метке l1.

Дамп программы:

Address Code Basic Line Source

0x00400000 0x000010b7 lui x1,1 1 lui x1,0x1

0x00400004 0x00c0d093 srli x1,x1,12 2 srli x1,x1,12

0x00400008 0x00108093 addi x1,x1,1 4 addi x1, x1, 1

0x0040000c 0xffdff16f jal x2,0xfffffffc 5 jal x2,l1

0x00400010 0x000990b7 lui x1,0x00000099 6 lui x1,0x99

0x00400014 0x000020b7 lui x1,2 8 lui x1,0x2

0x00400018 0x00c0d093 srli x1,x1,12 9 srli x1,x1,12

0x0040001c 0x00208093 addi x1,x1,2 11 addi x1, x1, 2

0x00400020 0xffdff16f jal x2,0xfffffffc 12 jal x2,l2

0x00400024 0x000990b7 lui x1,0x00000099 13 lui x1,0x99

0x00400028 0x000030b7 lui x1,3 15 lui x1,0x3

0x0040002c 0x00c0d093 srli x1,x1,12 16 srli x1,x1,12

0x00400030 0x00308093 addi x1,x1,3 18 addi x1, x1, 3

0x00400034 0xffdff16f jal x2,0xfffffffc 19 jal x2,l3

0x00400038 0x000990b7 lui x1,0x00000099 20 lui x1,0x99

0x0040003c 0x000040b7 lui x1,4 22 lui x1,0x4

0x00400040 0x00c0d093 srli x1,x1,12 23 srli x1,x1,12

0x00400044 0x00408093 addi x1,x1,4 25 addi x1, x1, 4

0x00400048 0xffdff16f jal x2,0xfffffffc 26 jal x2,l4

0x0040004c 0x000990b7 lui x1,0x00000099 27 lui x1,0x99

0x00400050 0x000050b7 lui x1,5 29 lui x1,0x5

0x00400054 0x00c0d093 srli x1,x1,12 30 srli x1,x1,12

0x00400058 0x00508093 addi x1,x1,5 32 addi x1, x1, 5

0x0040005c 0xffdff16f jal x2,0xfffffffc 33 jal x2,l5

0x00400060 0x000990b7 lui x1,0x00000099 34 lui x1,0x99

0x00400064 0x000060b7 lui x1,6 36 lui x1,0x6

0x00400068 0x00c0d093 srli x1,x1,12 37 srli x1,x1,12

0x0040006c 0x00608093 addi x1,x1,6 39 addi x1, x1, 6

0x00400070 0xffdff16f jal x2,0xfffffffc 40 jal x2,l6

0x00400074 0x000990b7 lui x1,0x00000099 41 lui x1,0x99

0x00400078 0x000070b7 lui x1,7 43 lui x1,0x7

0x0040007c 0x00c0d093 srli x1,x1,12 44 srli x1,x1,12

0x00400080 0x00708093 addi x1,x1,7 46 addi x1, x1, 7

0x00400084 0xffdff16f jal x2,0xfffffffc 47 jal x2,l7

0x00400088 0x000990b7 lui x1,0x00000099 48 lui x1,0x99

0x0040008c 0x000080b7 lui x1,8 50 lui x1,0x8

0x00400090 0x00c0d093 srli x1,x1,12 51 srli x1,x1,12

0x00400094 0x00808093 addi x1,x1,8 53 addi x1, x1, 8

0x00400098 0xffdff16f jal x2,0xfffffffc 54 jal x2,l8

0x0040009c 0x000990b7 lui x1,0x00000099 55 lui x1,0x99

Для исполнения на многопоточном процессоре массив программных счетчиков инициируется значениями адресов начала каждого из циклов – в данном примере – адреса:

**0x0, 0x14, 0x28, 0x3С, 0x50, 0x64, 0x78, 0x8C.**

Таблица хартов – перечисление номеров хартов с 0 по 7 в 4-битном представлении. Например, если активен должен быть харт с номером 0, то в таблице он будет представлен цифрой 8 (4`b1000), если с номером 3, то 0xB (4`b1011) и так далее.

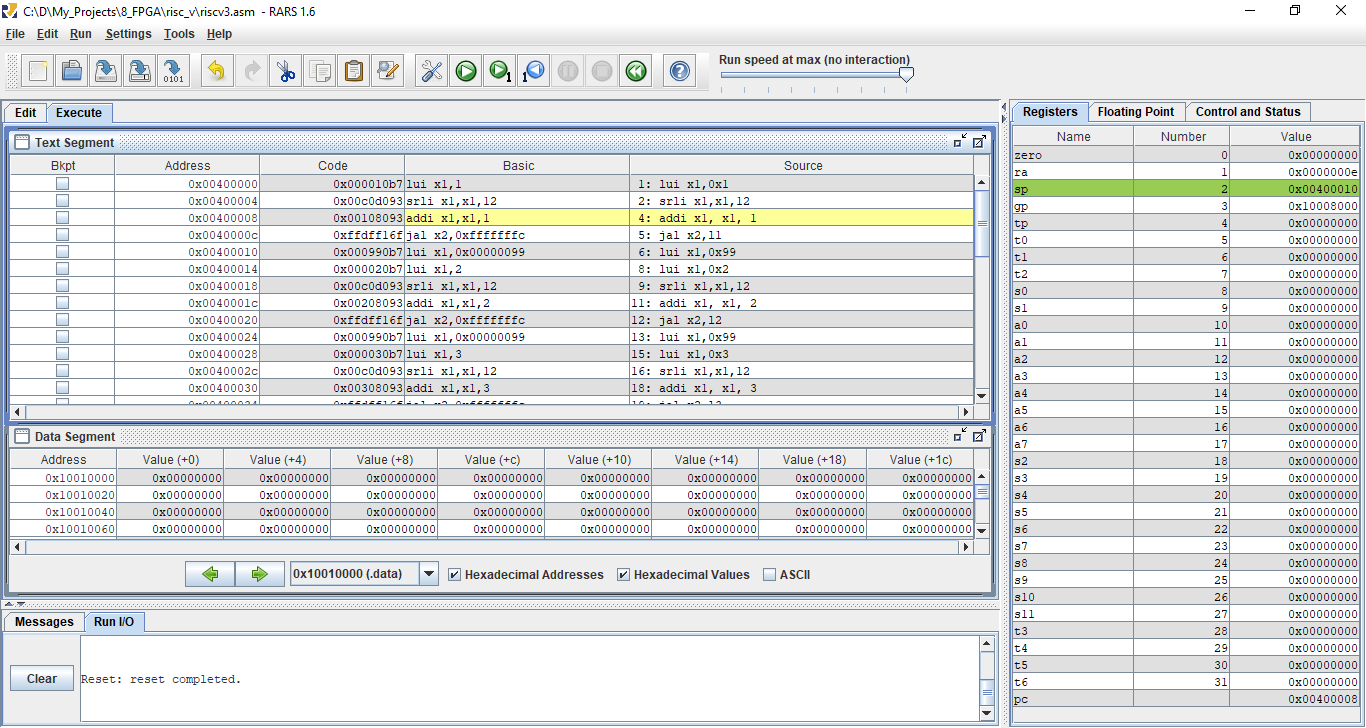


Рис. 6 Тестовая программа в симуляторе RARS.

Тест 1:

|  |  |
| --- | --- |
| **PC.txt** | **Hart\_table.txt** |
| @00  0  14  28  3c  50  64  78  8c | @00  8  1  2  3  4  5  6  7 |

Рис. 6-7 – временные диаграммы исполнения кода.

На ней представлены значения:

- сигналы сброса и тактовый;

- номера хартов (с битом активности);

- значение программного счетчика и выбранное из памяти слово инструкции;

- для АЛУ - опкод команды, операнды и результат;

- для файл-регистра текущие выходы и вход с разрешением записи;

- тестовые выходы первых 4 регистров файл-регистра с указанием, к какому харту они относятся.

Для регистра Х1 0-го активного харта видна сначала его загрузка значением 0х1000, потом сдвиг его до 0х1.

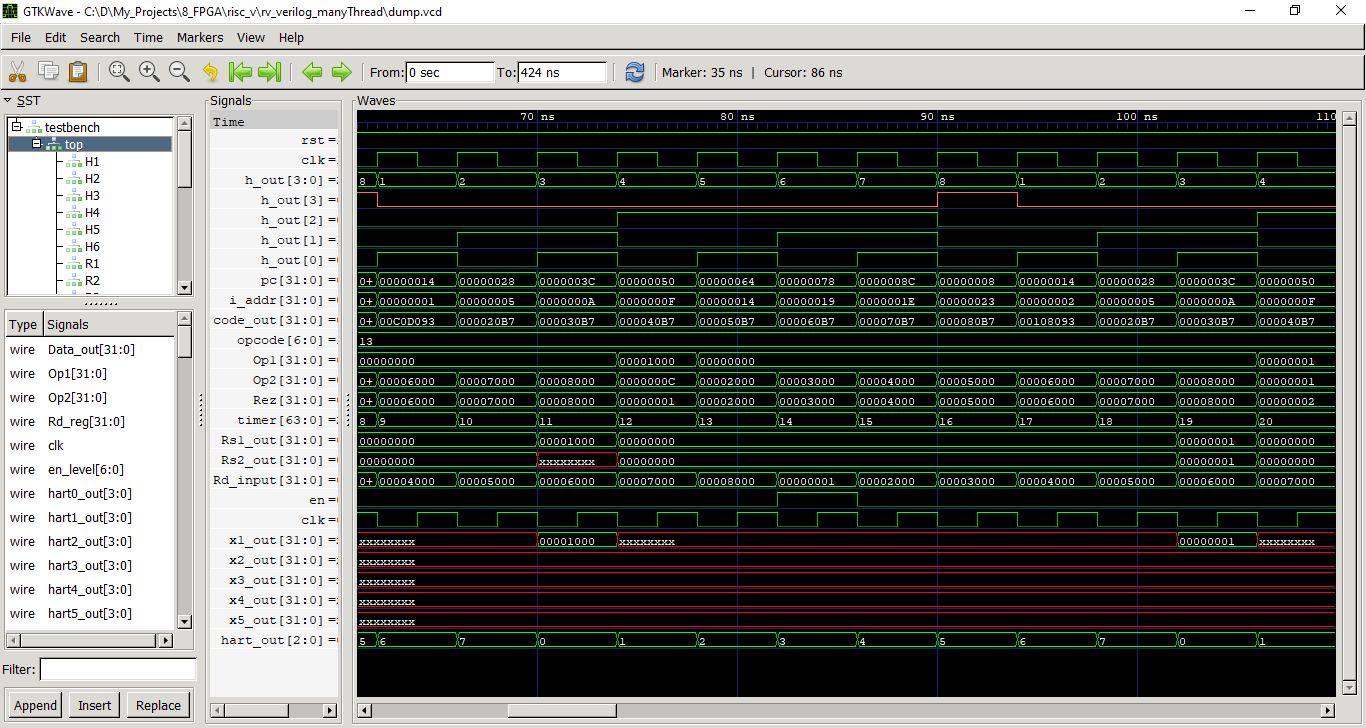


Рис. 7 Временные диаграммы исполнения программы в многопоточном ядре – активен харт0.

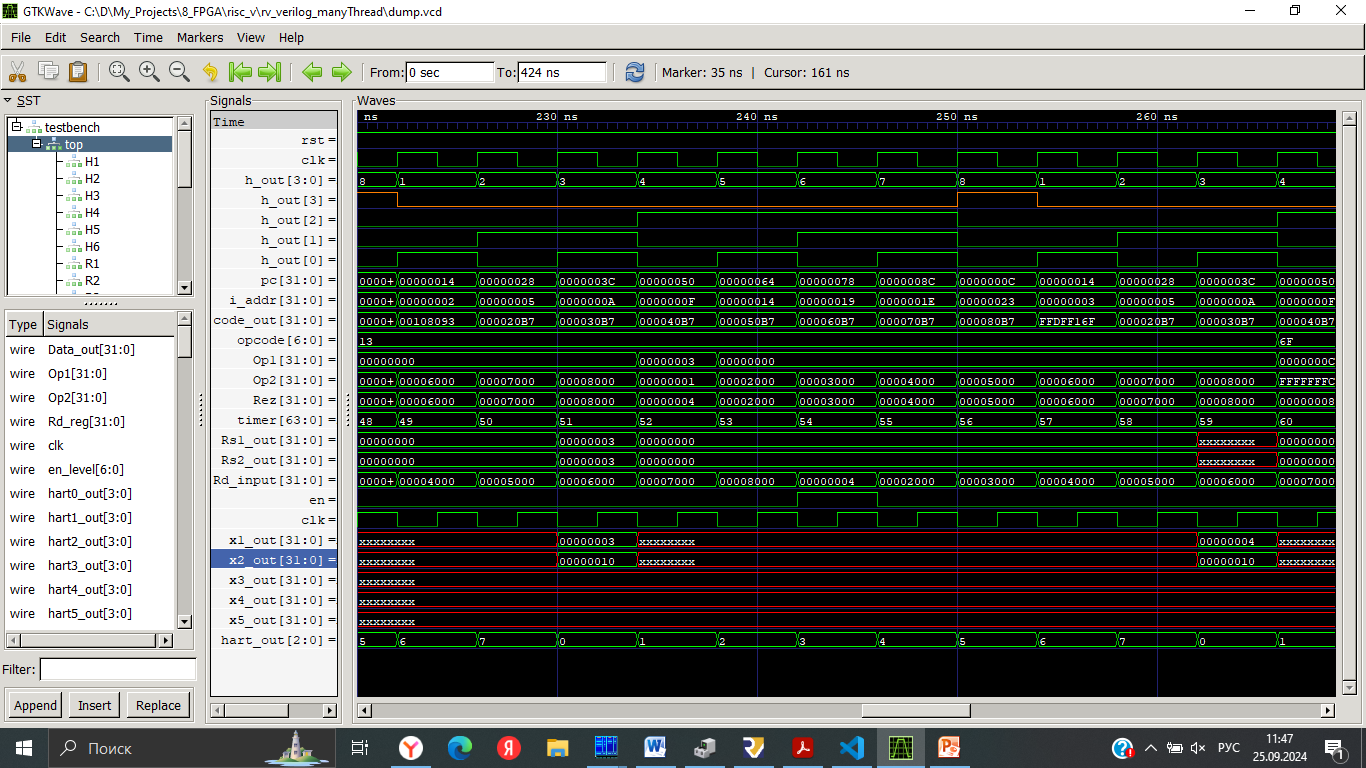


Рис. 8 Временные диаграммы исполнения программы в многопоточном ядре – активен харт0.

Тест 3 – распределение адресов тоже самое, активируем 2й харт (номер 1):

|  |  |
| --- | --- |
| **PC.txt** | **Hart\_table.txt** |
| @00  0  14  28  3c  50  64  78  8c | @00  8  9  2  3  4  5  6  7 |

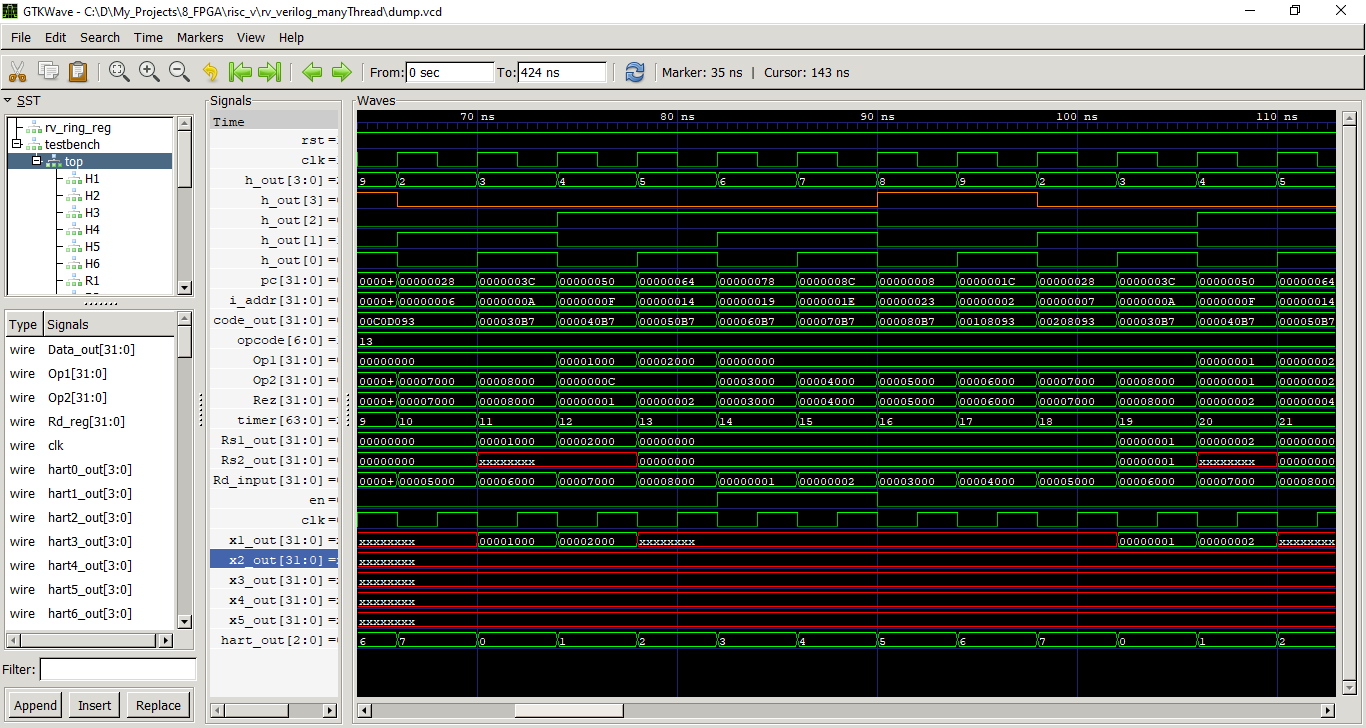


Рис. 9 Временные диаграммы исполнения программы в многопоточном ядре – активены харт0 и харт1.

Итоговые микроархитектурные блоки многопоточного варианта процессора:

**rv\_pc** – программный счетчик (массив из нескольких регистров);

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

**rv\_desh** – дешифратор команд (слова-инструкции) с входом «сброса конвейера»;

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

**rv\_reg\_file** – файл-регистр (массив из нескольких файл-регистров);

**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;

**rv\_r\_reg** – регистры хранения результатов промежуточных этапов;

**rv\_hart\_reg** – регистры хранения номеров и состояния активности хартов;

**rv\_timer** – системный таймер;

**rv\_hart\_table** – память хранения списка и состояния хартов.

**Литература**

1. Мультиядерные микроконтроллеры семейства XCore от XMOS // https://cyberleninka.ru/article/n/multiyadernye-mikrokontrollery-semeystva-xcore-ot-xmos
2. Using RISC-V to define SoCs in software \_ XMOS // <https://www.xmos.com/using-risc-v-to-define-socs-in-software/>
3. The RISC-V Instruction Set Manual. Volume I: User-Level ISA Document Version 2.2 // Editors: Andrew Waterman1, Krste Asanov
4. <https://electronics.stackexchange.com/questions/580645/what-is-a-hardware-thread-in-risc-v>
5. <https://www.computer.org/csdl/proceedings-article/trustcom/2022/942500b587/1LFM9tik1IQ>