**Конвейеризированное процессорное ядро**

Несложно заметить, что предложенную микроархитектуру многотактного процессора (рис.1) можно относительно просто превратить в конвейеризированную. Для этого достаточно (вот сейчас будет страшно – а точно ли?) разрешить работу всех промежуточных регистров Rx, а также блоков программного счетчика, файл-регистра, регистров специальных функций, и памяти на запись.

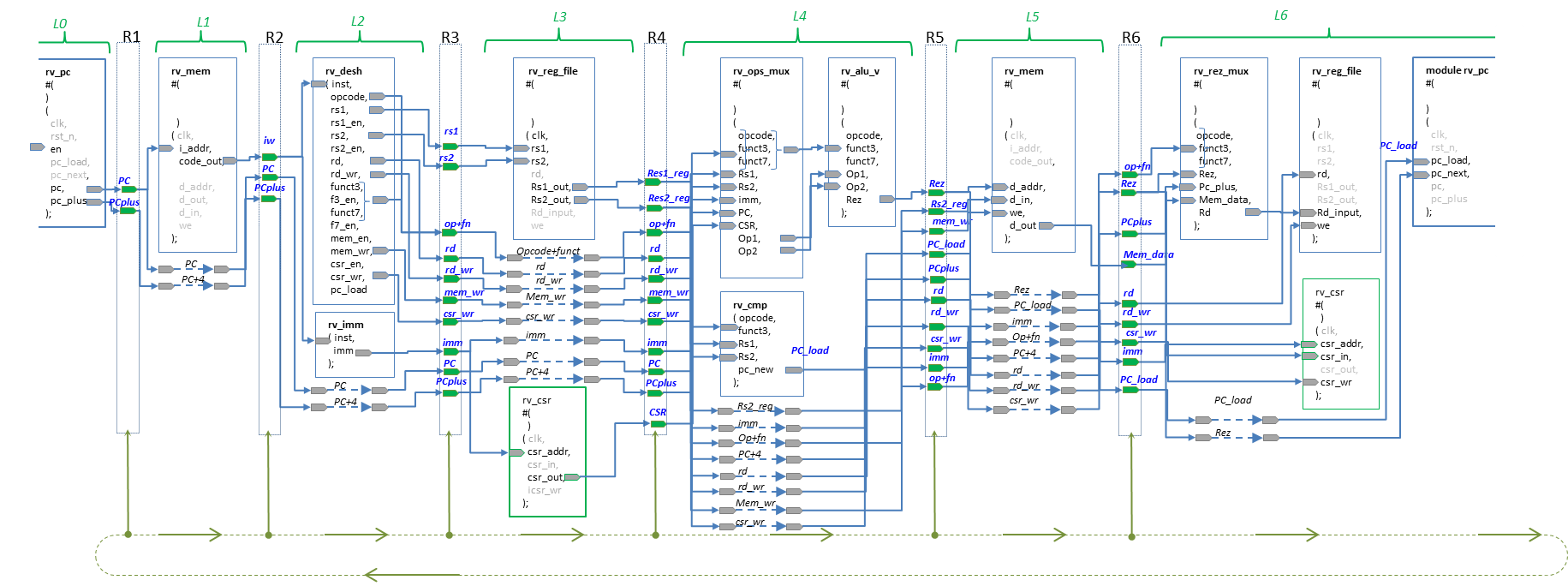


Рис. 1 Структурная схема многотактного процессора.

Первым вариантом модификации будет исключение из микроархитектуры кольцевого сдвигового регистра, подача входы разрешения работы модулей с памятью логической единицы («разрешено»).

При таком варианте в случае линейного кода на каждом такте значение программного счетчика будет инкрементироваться и выбираться новые инструкции программы – рис.2.

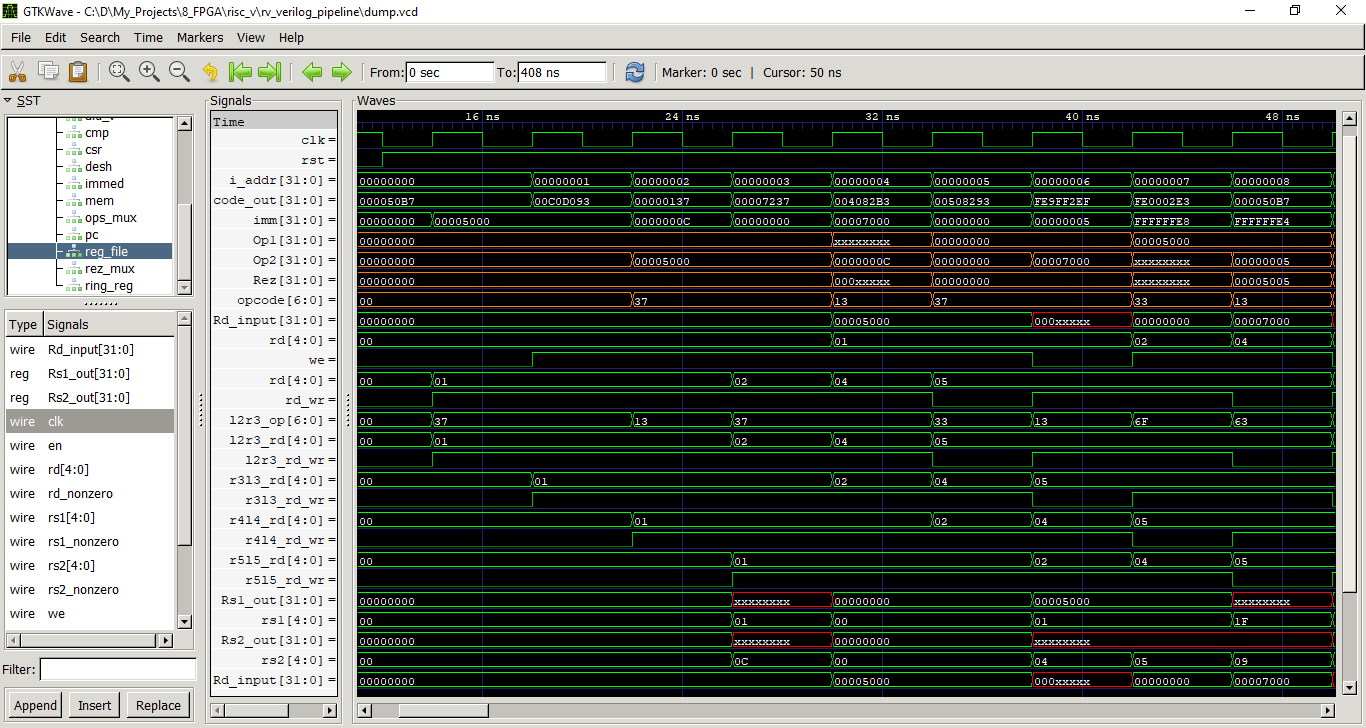


Рис. 2 Временные диаграммы конвейеризованного варианта

Смотря на рис.2 можно с одной стороны порадоваться – выборка инструкций действительно конвейеризовалась, но вот выполнение последовательности команд наталкивается на неопределенность при выборке операндов из файл-регистра.

Последующий разбор показывает, что в текущем варианте разбиения конвейера по этапам может нарушаться ( и для тестового программного кода нарушается) временнАя последовательность выполнения команд – выборка операнда последующей команды начинается раньше, чем производится запись результата предыдущей.

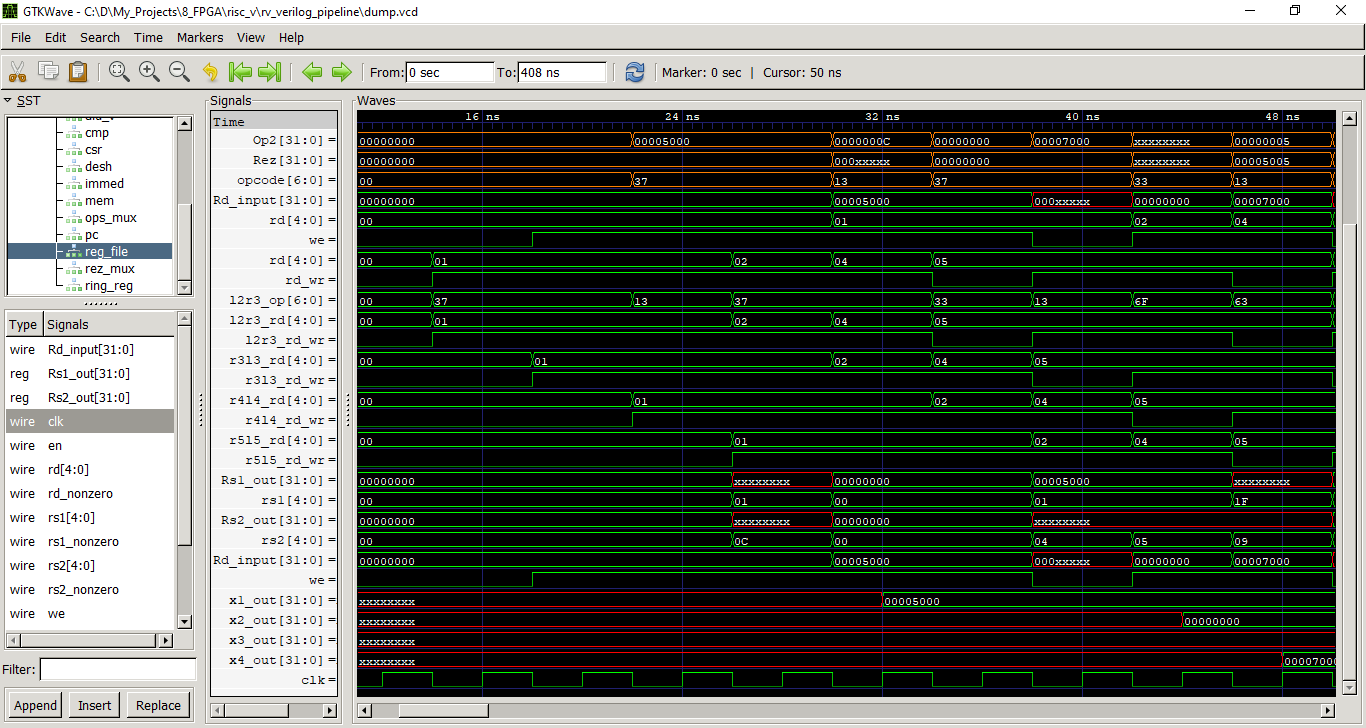


Рис.3 «Разбор» причин нарушения корректности работы.

Следовательно, необходимо немного пересмотреть вариант разбиения конвейера по этапам для корректной работы.

Микроархитектурные блоки процессора оставляем пока как и были:

**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.

Новый вариант разбиения по этапам:

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 – вычисления - выборка операндов из файл-регистров, работа АЛУ, чтение/запись данных из оперативной памяти, запись данных в файл-регистры..

L4 – запись нового значения в программный счётчик.

Пробуем рассмотреть возможные сигналы и действия на каждом из этапов. Промежуточные результаты каждого из этапов по прежнему фиксируются в неархитектурных «буферных» регистрах, ожидаемо их стало меньше (R0 – R4), и разрядность их также можно уменьшить. Ряд сигналов, аналогично предыдущей версии, «пробрасываются» по этапам (напрямую между регистрами) в том случае, если они не задействованы в них. Работа блоков с элементами памяти и промежуточных регистров разрешена всё время.

Структурная схема конвейеризированного варианта процессора представлена на рис.4.

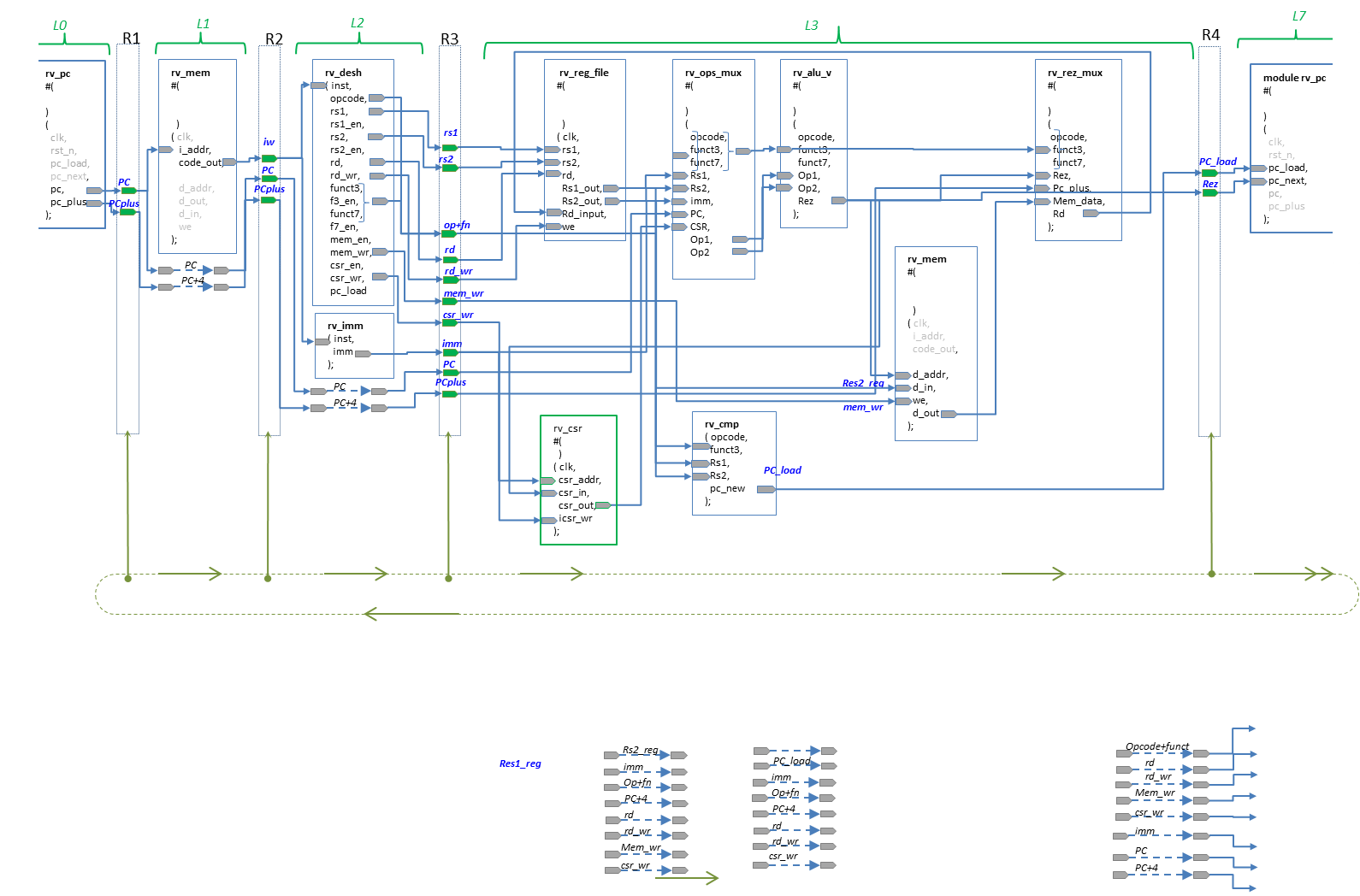


Рис. 4 Структурная схема конвейеризированного варианта процессора.

Этапы L0-L2 остаются пока без изменений.

**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.

**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),    .pc\_load(r6l7\_PC\_load),    .pc\_next(l6r7\_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;  assign r1\_out = {r1l1\_PC, r1l1\_PCplus, 192'h0};  rv\_r\_reg R1(    .clk(clk),    .rst\_n(rst),    .en(en\_level[0]),    .r\_in(r1\_in),    .r\_out(r1\_out)  ); |

Рис. 5 Выборка текущего значения программного счетчика.

|  |  |
| --- | --- |
|  | 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),    .code\_out(l1r2\_iw),    .d\_addr(r5l5\_Rez),    .d\_out(l5r6\_Mem\_data),    .d\_in(r5l5\_Rs2\_reg),    .we(r5l5\_mem\_wr)  );  // 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\_out = {r2l2\_iw, r2l2\_PC, r2l2\_PCplus, 160'h0};  rv\_r\_reg R2(    .clk(clk),    .rst\_n(rst),    .en(en\_level[1]),    .r\_in(r2\_in),    .r\_out(r2\_out)  ); |

Рис. 6 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};  assign r3\_out = {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, 125'h0};  rv\_r\_reg R3(    .clk(clk),    .rst\_n(rst),    .en(en\_level[2]),    .r\_in(r3\_in),    .r\_out(r3\_out)  ); |

Рис. 7 Дешифровка инструкции.

**L3 – вычисления**

Адреса регистров-операндов подаются на адресные входы файл-регистров (основного и регистров специальных функций).

Сигналы опкодов и функций-расширений подаются на мультиплексоры операндов и результатов. Выходы файл-регистра Rs1, Rs2 и опкоды подаются на блок разрешения переноса (по факту – компаратор с входами разрешения).

Также выходы файл-регистра Rs1, Rs2 совместно с константой и значением регистра CSR и программного счетчика подается на выход мультиплексора операндов, выходы его – на входы операндов АЛУ. Выходные сигналы АЛУ поступают на вход блока CSR, на вход мультиплексора операндов для записи в файл-регистр (по адресу rd), адресуют память (ОЗУ).

В регистр R4 записывается новое значение программного счетчика и сигнал разрешения записи в него.

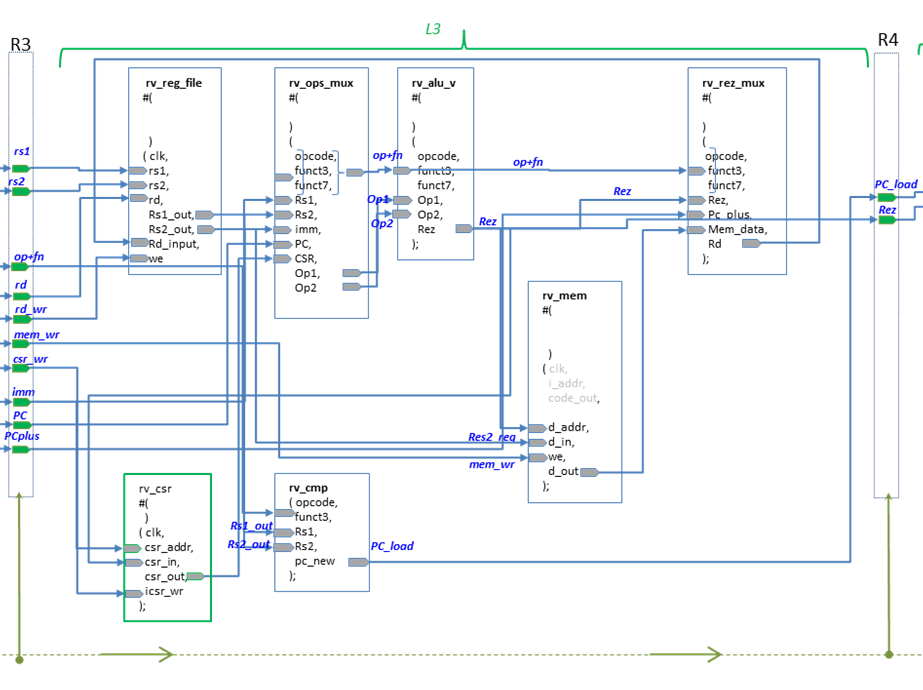


Рис. 8 Структура логики этапа L3 - вычисления.

Описание логики этапа «вычисления»:

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)

);

wire [31:0] Rs1\_reg;

wire [31:0] Rs2\_reg;

wire [31:0] CSR\_mux;

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

  .clk(~clk),

  .rs1(r3l3\_rs1),

  .rs2(r3l3\_rs2),

  .rd(r3l3\_rd),

  .Rs1\_out(Rs1\_reg),

  .Rs2\_out(Rs2\_reg),

  .Rd\_input(Rd\_reg),

  .we(r3l3\_rd\_wr),

  .en(1'b1)

);

rv\_csr csr(

  .clk(~clk),

  .csr\_addr(r3l3\_imm[11:0]),

  .csr\_in(Rez),

  .csr\_out(CSR\_mux),

  .csr\_wr(r3l3\_csr\_wr),

  .en(1'b1)

);

wire [31:0] Op1;

wire [31:0] Op2;

rv\_ops\_mux ops\_mux(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .imm(r3l3\_imm),

  .PC(r3l3\_PC),

  .CSR(CSR\_mux),

  .Op1(Op1),

  .Op2(Op2)

);

rv\_cmp cmp(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .pc\_new(l3r4\_PC\_load)

);

wire [31:0] Rez;

wire [31:0] Rd\_reg;

wire [31:0] Mem\_data;

rv\_alu\_v alu\_v(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Op1(Op1),

  .Op2(Op2),

  .Rez(Rez)

);

rv\_rez\_mux rez\_mux(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Rez(Rez),

  .Pc\_plus(r3l3\_PCplus),

  .Mem\_data(Mem\_data),

  .Imm(r3l3\_imm),

  .Rd(Rd\_reg)

);

wire r4l4\_PC\_load;

wire [31:0] r4l4\_Rez;

wire [255:0] r4\_in;

wire [255:0] r4\_out;

assign r4\_in = {Rez, l3r4\_PC\_load, 223'h0};

wire [222:0] r4\_null;

assign {r4l4\_Rez, r4l4\_PC\_load, r4\_null} = r4\_out;

**L4 – запись нового значения в программный счётчик**.

Возможно, данный этап можно было совместить с предыдущим, но пока для наглядности оставим его. Вычисленный результат нового адреса перехода и сигнал записи в программный счетчик из регистра R4 подаются на соответствующие входы программного счётчика.

|  |  |
| --- | --- |
|  | wire r4l4\_PC\_load;  wire [31:0] r4l4\_Rez;  wire [255:0] r4\_in;  wire [255:0] r4\_out;  assign r7\_in = {Rez, l3r4\_PC\_load, 223'h0};  wire [222:0] r4\_null;  assign {r4l4\_Rez, r4l4\_PC\_load, r4\_null} = r4\_out;  rv\_r\_reg R4(    .clk(clk),    .rst\_n(rst),    //.en(en\_level[6]),    .en(1'b1),    .r\_in(r4\_in),    .r\_out(r4\_out)  ); |

Рис. 9 Запись нового значения в программный счётчик.

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]),

  .en(1'b1),

  .pc\_load(r4l4\_PC\_load),

  .pc\_next(r4l4\_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]),

  .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(32'h0), //(r1l1\_PC),

  .i\_addr(r1l1\_PC>>2),

  .code\_out(l1r2\_iw),

  .d\_addr(Rez>>2),

  .d\_out(Mem\_data),

  .d\_in(Rs2\_reg),

  .we(r3l3\_mem\_wr),

  .en(1'b1)

);

// 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]),

  .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()

);

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]),

  .en(1'b1),

  .r\_in(r3\_in),

  .r\_out(r3\_out)

);

//L3\_R4

wire [31:0] Rs1\_reg;

wire [31:0] Rs2\_reg;

wire [31:0] CSR\_mux;

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

  .clk(clk),

  .rs1(r3l3\_rs1),

  .rs2(r3l3\_rs2),

  .rd(r3l3\_rd),

  .Rs1\_out(Rs1\_reg),

  .Rs2\_out(Rs2\_reg),

  .Rd\_input(Rd\_reg),

  .we(r3l3\_rd\_wr),

  .en(1'b1)

);

rv\_csr csr(

  .clk(clk),

  .csr\_addr(r3l3\_imm[11:0]),

  .csr\_in(Rez),

  .csr\_out(CSR\_mux),

  .csr\_wr(r3l3\_csr\_wr),

  .en(1'b1)

);

wire [31:0] Op1;

wire [31:0] Op2;

rv\_ops\_mux ops\_mux(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .imm(r3l3\_imm),

  .PC(r3l3\_PC),

  .CSR(CSR\_mux),

  .Op1(Op1),

  .Op2(Op2)

);

rv\_cmp cmp(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .pc\_new(l3r4\_PC\_load)

);

wire [31:0] Rez;

wire [31:0] Rd\_reg;

wire [31:0] Mem\_data;

rv\_alu\_v alu\_v(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Op1(Op1),

  .Op2(Op2),

  .Rez(Rez)

);

rv\_rez\_mux rez\_mux(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Rez(Rez),

  .Pc\_plus(r3l3\_PCplus),

  .Mem\_data(Mem\_data),

  .Imm(r3l3\_imm),

  .Rd(Rd\_reg)

);

wire r4l4\_PC\_load;

wire [31:0] r4l4\_Rez;

wire [255:0] r4\_in;

wire [255:0] r4\_out;

assign r4\_in = {Rez, l3r4\_PC\_load, 223'h0};

wire [222:0] r4\_null;

assign {r4l4\_Rez, r4l4\_PC\_load, r4\_null} = r4\_out;

rv\_r\_reg R4(

  .clk(clk),

  .rst\_n(rst),

  .en(1'b1),

  .r\_in(r4\_in),

  .r\_out(r4\_out)

);

endmodule

При выполнении всё той же тестовой программы, где фиксировалась неточность (ошибочность функционирования) видно, что ситуация с регистрами выправилась. И до определенного момента выполнение кода адекватно, а именно – до инструкций переходов/ветвления.

Перенос операции записи нового адреса перехода в программный счетчик лишь немного решает проблему – да, переход совершается, но происходит выполнение команд, уже попавших в конвейер (рис. 12-13), что, конечно же, недопустимо.

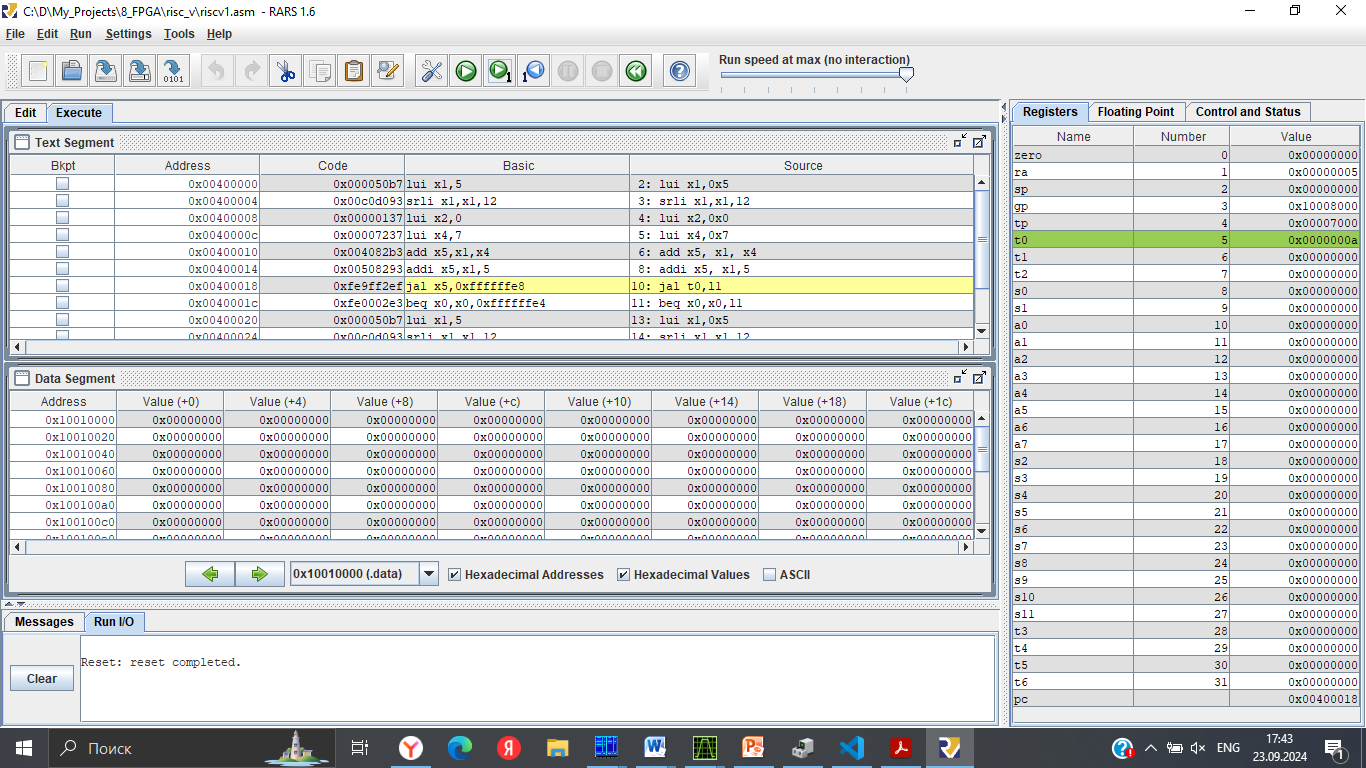


Рис. 10 Тестовая программа в симуляторе RARS.

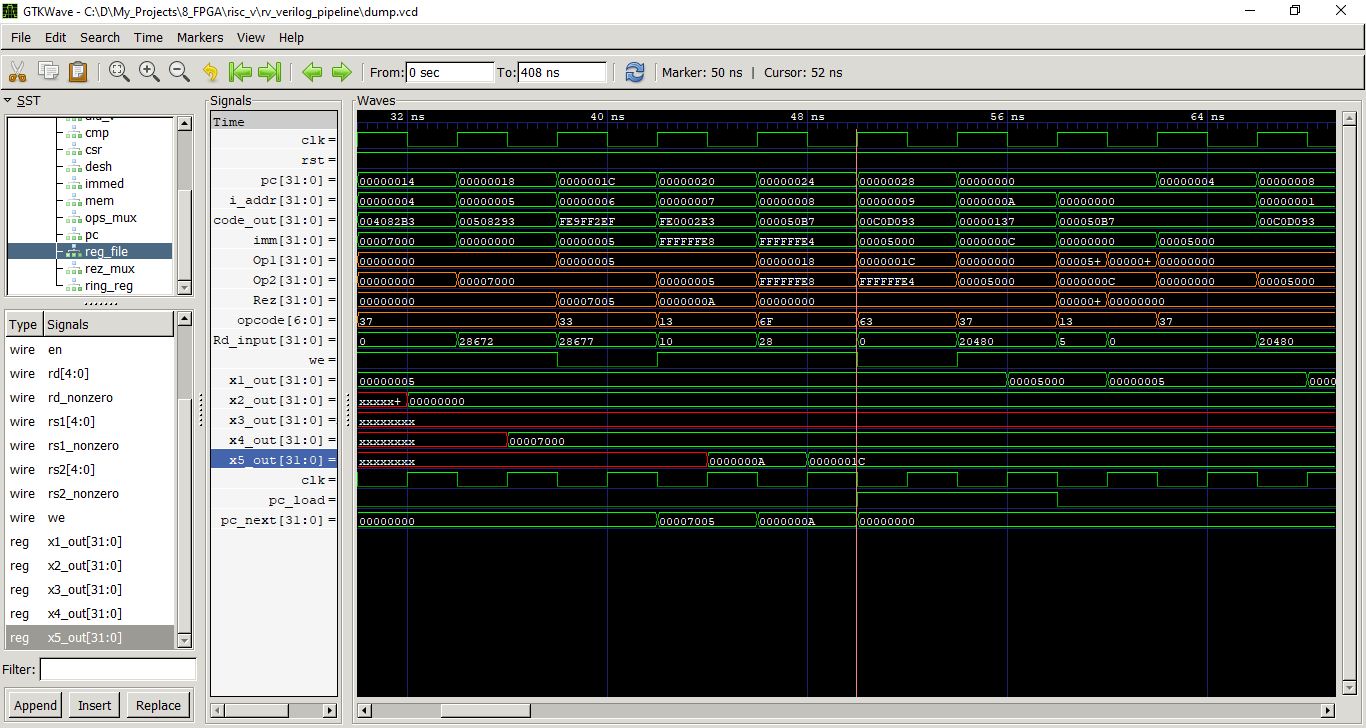


Рис. 11 Временные диаграммы.

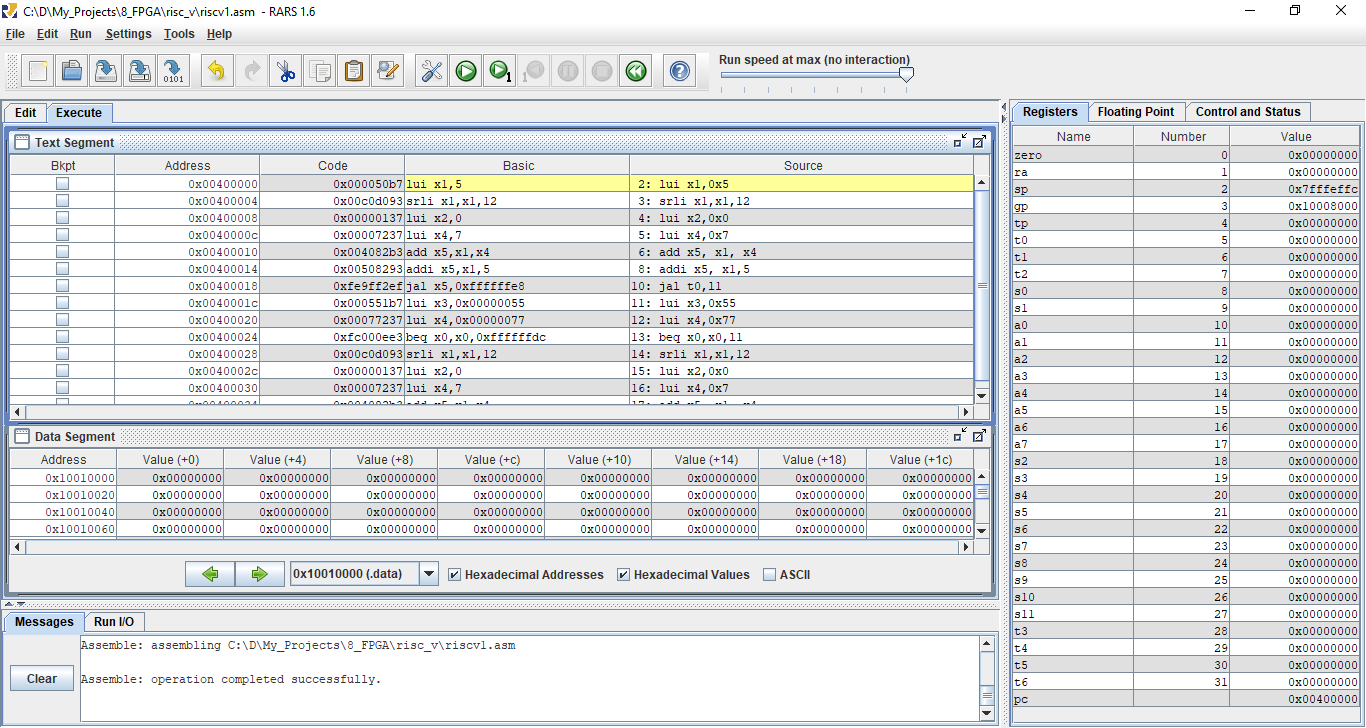


Рис. 12 Тестовая программа в симуляторе RARS

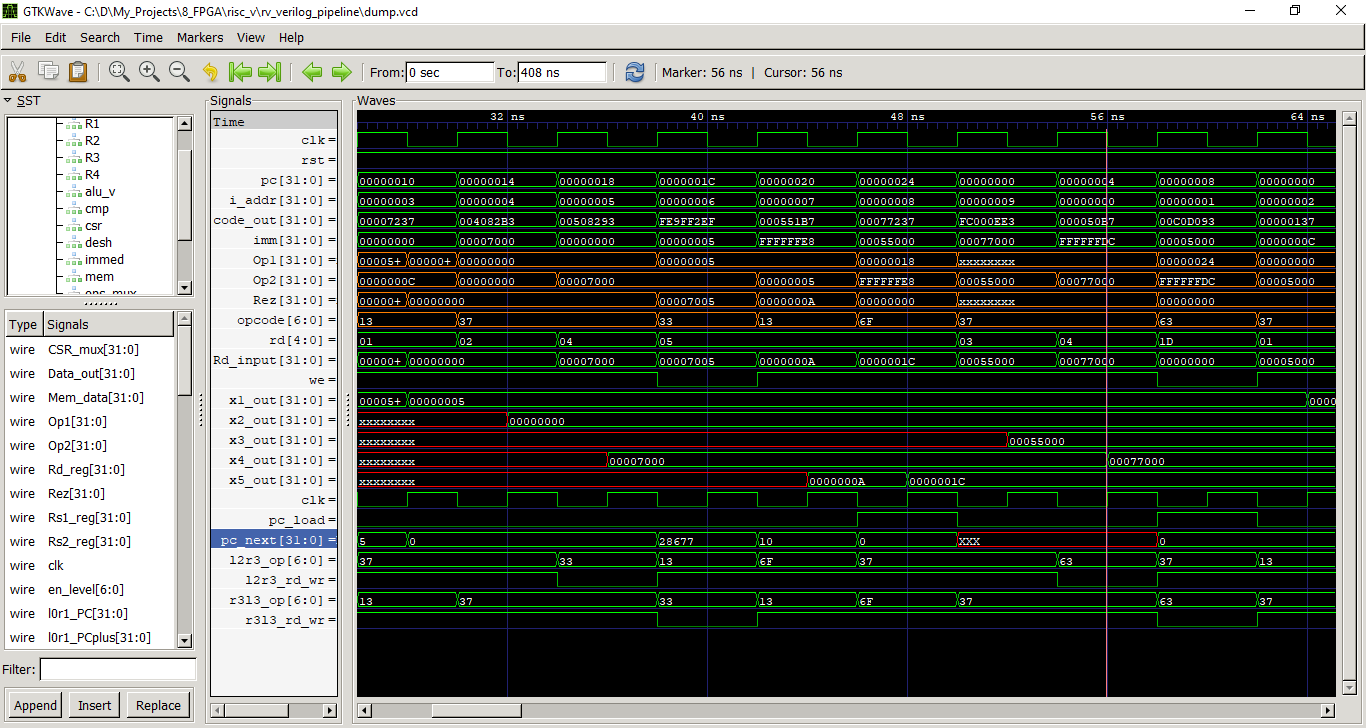


Рис. 13 Временные диаграммы тестирования работы варианта конвейера.

Из всего этого следует, что необходимо как-то разрешать данную ситуацию. Варианты могут быть следующие.

1) Немного модифицируем логику выборки команд так, чтобы при обнаружении команд перехода (любого) запрещается дальнейшая загрузка команд в конвейер (это можно сделать даже на первом этапе. Но это приведет к тому, что данные команды в любом случае будут выполняться несколько тактов, вне зависимости от того, выполняется условие перехода или нет.

2) Можно добавить опцию сброса конвейера при обнаружении перехода – например все этапы заменяем на выполнение операции «*nop*» - ее роль в архитектуре RISC-V исполняет команда «*addi x0,x0,0*» (как один из вариантов).

Попытаемся пойти вторым путём, так как первый действительно порождает много циклов простоя.

Немного модифицируется дешифратор команд – вводим дополнительный входной сигнал «сброса конвейера», по которому дешифратор при любом входном слове выдает комбинацию сигналов, соответствующих «nop» (в наборе команд RV32I nop соответствует, например, команда сложения add x0,x0,x0):

module rv\_desh

( input [31:0] inst,

  output reg [6:0] opcode,

  output reg [4:0] rs1,

  output reg rs1\_en,

  output reg [4:0] rs2,

  output reg rs2\_en,

  output reg [4:0] rd,

  output reg rd\_wr,

  output reg [2:0] funct3,

  output reg f3\_en,

  output reg [6:0] funct7,

  output reg f7\_en,

  output reg mem\_en,

  output reg mem\_wr,

  output reg csr\_en,

  output reg csr\_wr,

  input pipe\_rst,

  output reg pc\_load

);

always @ \*

begin

    if (pipe\_rst) begin

        opcode[6:0] <= 7'b0010011;

        rs1 <= 5'h0;

        rs2 <= 5'h0;

        rd <= 5'h0;

        funct3 <= 3'b0;

        funct7 <= 3'b0;

        end

    else begin

        opcode[6:0] <= inst[6:0];

        rs1 <= inst[19:15];

        rs2 <= inst[24:20];

        rd <= inst[11:7];

        funct3 <= inst[14:12];

        funct7 <= inst[31:25];

    end

end

always @ \*

begin

  case (opcode)

    //I-type

    7'b00000\_11 : begin // load data from mem

        $display("LOAD");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b1;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b1;

        f7\_en <= 1'b0;

        mem\_en <= 1'b1;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    7'b00011\_11 : begin // fence

        $display("fence");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b1;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b1;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    7'b00100\_11 : begin // reg with immediate operations

        $display("OP-IMM");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b1;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b1;

        f7\_en <= 1'b1;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    7'b00100\_11 : begin // reg with reg operations

        $display("OP-reg");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b1;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b1;

        f7\_en <= 1'b1;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    7'b11100\_11 : begin // mret - system return

        $display("RET\_SYST");

        rs1\_en <= 1'b0;

        rs2\_en <= 1'b0;

        rd\_wr <= 1'b0;

        f3\_en <= 1'b1;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        if (funct3 == 3'b000) begin

            csr\_en <= 1'b0;

            csr\_wr <= 1'b0;

        end

        else begin  // operations with CSR

            csr\_en <= 1'b1;

            csr\_wr <= 1'b1;

        end

        pc\_load <= 1'b0;

        end

    7'b11001\_11 : begin // Relative (rs1) jump and link in register

        $display("JALR");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b0;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b0;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b1;

        end

    //J-type

    7'b11011\_11 : begin // pc relative jump and link in register

        $display("JAL");

        rs1\_en <= 1'b0;

        rs2\_en <= 1'b0;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b0;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b1;

        end

    //S-type

    7'b01000\_11 : begin // store register value in memory

        $display("Store");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b1;

        rd\_wr <= 1'b0;

        f3\_en <= 1'b1;

        f7\_en <= 1'b0;

        mem\_en <= 1'b1;

        mem\_wr <= 1'b1;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    //U-type

    7'b01101\_11 : begin // load upper immediate

        $display("LUI");

        rs1\_en <= 1'b0;

        rs2\_en <= 1'b0;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b0;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    7'b00101\_11 : begin // add upper immediate to PC

        $display("AUIPC");

        rs1\_en <= 1'b0;

        rs2\_en <= 1'b0;

        rd\_wr <= 1'b1;

        f3\_en <= 1'b0;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

    //B-type

    7'b11000\_11 : begin // conditional PC relative branch - PC+imm

        $display("BRANCH");

        rs1\_en <= 1'b1;

        rs2\_en <= 1'b1;

        rd\_wr <= 1'b0;

        f3\_en <= 1'b1;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b1; // must be 'AND' with compatator output signal

        end

    default: begin

        $display ("default");

        rs1\_en <= 1'b0;

        rs2\_en <= 1'b0;

        rd\_wr <= 1'b0;

        f3\_en <= 1'b0;

        f7\_en <= 1'b0;

        mem\_en <= 1'b0;

        mem\_wr <= 1'b0;

        csr\_en <= 1'b0;

        csr\_wr <= 1'b0;

        pc\_load <= 1'b0;

        end

  endcase

end

endmodule

Вводим биты сброса конвейера в сам конвейер.

Этап L0:

wire [255:0] r1\_in;

assign r1\_in = {l3r4\_PC\_load, l0r1\_PC, l0r1\_PCplus, 191'h0};

wire [255:0] r1\_out;

wire [190:0] r1\_null;

assign {r1l1\_pipe\_rst, 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)

);

Этап L1:

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 = {r1l1\_pipe\_rst, l1r2\_iw, r1l1\_PC, r1l1\_PCplus, 159'h0};

wire [158:0] r2\_null;

assign {r2l2\_pipe\_rst, 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)

);

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;

В случае опознавания перехода необходимо, чтобы команды, уже попавшие в конвейер, игнорировались. Суммирующий сигнал *pipe\_rst* для дешифратора команд объединятся по «or» из несколки сигналов, чтобы охватить «заранее» эффекты от перехода.

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),

  .pipe\_rst(r2l2\_pipe\_rst | l3r4\_PC\_load | r1l1\_pipe\_rst),

  .pc\_load()

);

Финальная структура процессора представлена на рис.14

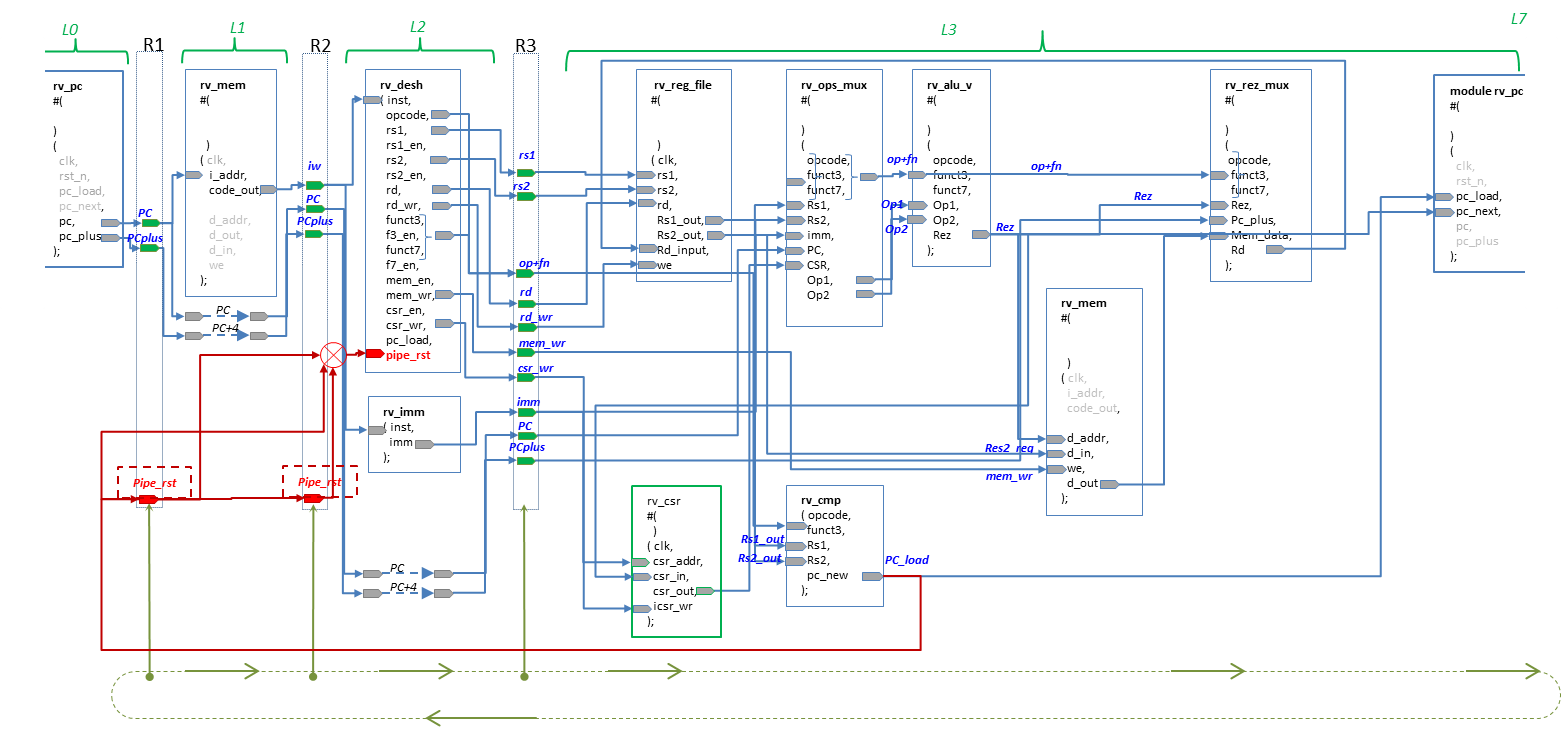


Рис. 14 Итоговый вариант конвейеризированного процессора.

Временные диаграммы приведены на рис.15 – видно, что блокирование выполнения операций предыдущих этапов конвейера, равно как и переход – работают штатно.

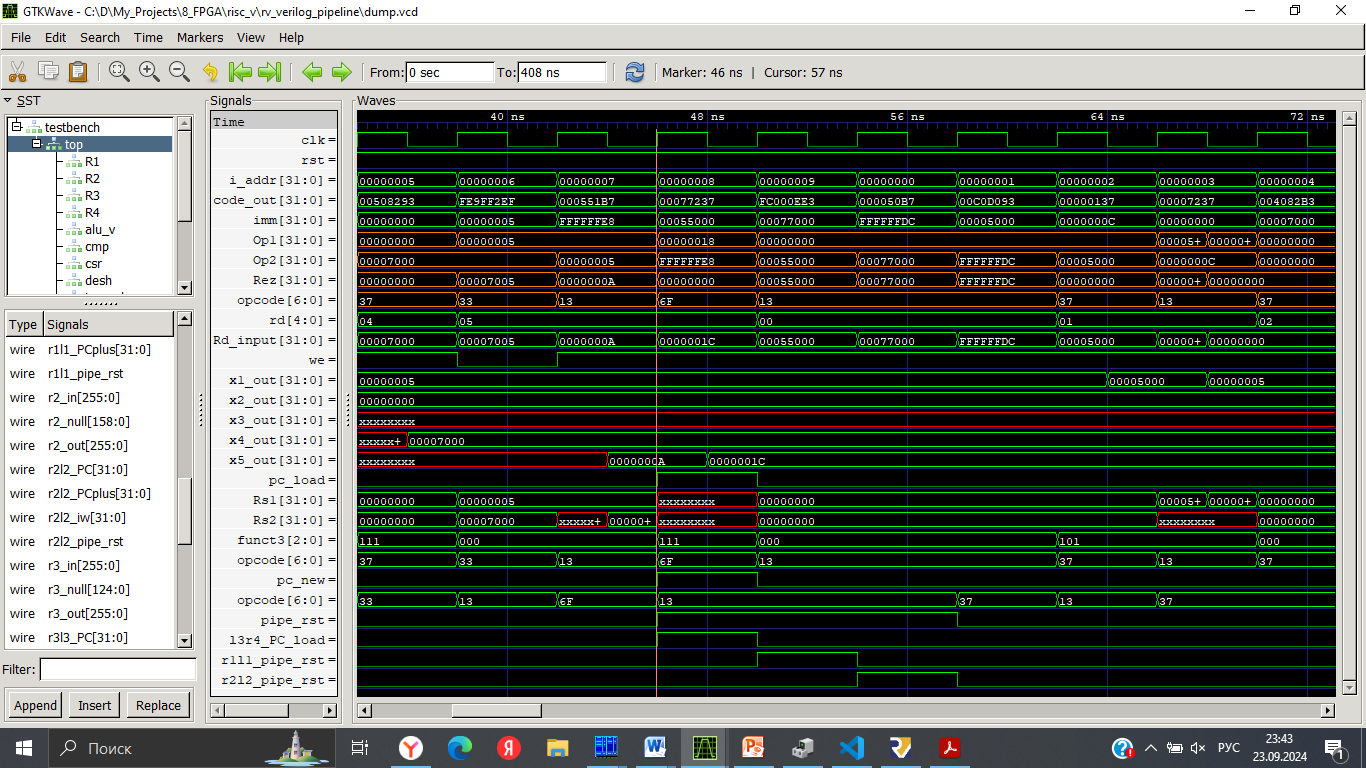


Рис. 15 Временные диаграммы тестирования работы конвейера.

Финальный код процессорного ядра:

`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(1'b1),

  .pc\_load(l3r4\_PC\_load),

  .pc\_next(Rez),

  .pc(l0r1\_PC),

  .pc\_plus(l0r1\_PCplus)

);

wire [255:0] r1\_in;

assign r1\_in = {l3r4\_PC\_load, l0r1\_PC, l0r1\_PCplus, 191'h0};

wire [255:0] r1\_out;

wire [190:0] r1\_null;

assign {r1l1\_pipe\_rst, 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(32'h0), //(r1l1\_PC),

  .i\_addr(r1l1\_PC>>2),

  .code\_out(l1r2\_iw),

  .d\_addr(Rez>>2),

  .d\_out(Mem\_data),

  .d\_in(Rs2\_reg),

  .we(r3l3\_mem\_wr),

  .en(1'b1)

);

// 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;

//assign r2\_in = {l3r4\_PC\_load, l1r2\_iw, r1l1\_PC, r1l1\_PCplus, 159'h0};

assign r2\_in = {r1l1\_pipe\_rst, l1r2\_iw, r1l1\_PC, r1l1\_PCplus, 159'h0};

wire [158:0] r2\_null;

assign {r2l2\_pipe\_rst, 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),

  .pipe\_rst(r2l2\_pipe\_rst | l3r4\_PC\_load | r1l1\_pipe\_rst),

  .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(1'b1),

  .r\_in(r3\_in),

  .r\_out(r3\_out)

);

wire [31:0] Rs1\_reg;

wire [31:0] Rs2\_reg;

wire [31:0] CSR\_mux;

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

  .clk(~clk),

  .rs1(r3l3\_rs1),

  .rs2(r3l3\_rs2),

  .rd(r3l3\_rd),

  .Rs1\_out(Rs1\_reg),

  .Rs2\_out(Rs2\_reg),

  .Rd\_input(Rd\_reg),

  .we(r3l3\_rd\_wr),

  .en(1'b1)

);

rv\_csr csr(

  .clk(~clk),

  .csr\_addr(r3l3\_imm[11:0]),

  .csr\_in(Rez),

  .csr\_out(CSR\_mux),

  .csr\_wr(r3l3\_csr\_wr),

  .en(1'b1)

);

wire [31:0] Op1;

wire [31:0] Op2;

rv\_ops\_mux ops\_mux(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .imm(r3l3\_imm),

  .PC(r3l3\_PC),

  .CSR(CSR\_mux),

  .Op1(Op1),

  .Op2(Op2)

);

rv\_cmp cmp(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .pc\_new(l3r4\_PC\_load)

);

wire [31:0] Rez;

wire [31:0] Rd\_reg;

wire [31:0] Mem\_data;

rv\_alu\_v alu\_v(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Op1(Op1),

  .Op2(Op2),

  .Rez(Rez)

);

rv\_rez\_mux rez\_mux(

  .opcode(r3l3\_op),

  .funct3(r3l3\_fn3),

  .funct7(r3l3\_fn7),

  .Rez(Rez),

  .Pc\_plus(r3l3\_PCplus),

  .Mem\_data(Mem\_data),

  .Imm(r3l3\_imm),

  .Rd(Rd\_reg)

);

wire r4l4\_PC\_load;

wire [31:0] r4l4\_Rez;

wire [255:0] r4\_in;

wire [255:0] r4\_out;

assign r4\_in = {Rez, l3r4\_PC\_load, 223'h0};

wire [222:0] r4\_null;

assign {r4l4\_Rez, r4l4\_PC\_load, r4\_null} = r4\_out;

rv\_r\_reg R4(

  .clk(clk),

  .rst\_n(rst),

  .en(1'b1),

  .r\_in(r4\_in),

  .r\_out(r4\_out)

);

endmodule