**Построение однотактного простейшего ядра RISC-V**

**Введение**

Приступая к реализации собственно процессорного узла для определенного набора команд, мы ступаем в область создания т.н. микроархитектуры процессора – его уникальной для каждой реализации внутренней структуры. По факту уже, разбирая вариант построения декодера команд в предыдущей части, мы уже начали работу над микроархитектурой.

Вернемся к основным архитектурным блокам процессора:

* регистровый файл/файл-регистр (31 активный регистр, плюс «регистр ноля», два выходных порта, один порт на запись);
* программный счётчик (PC);
* блок регистров специального назначения (CSR);
* арифметико-логическое устройство (АЛУ);
* блок дешифратора команд.

Дополнительными блоками не входящими в архитектурные блоки процессорного ядра, но необходимые для работы:

* память программ;
* память данных (оперативная память).

## Согласно теории - однотактная микроархитектура подразумевает выполнение команды процессора за один такт. Все операции выполняются за один такт, эта микроархитектура не требует никакого неархитектурного состояния. Все исполнительные узлы процессорного ядра (за исключением, конечно же его регистров, определенных или других элементов с памятью состояния) архитектурой представлены комбинационно-логическими схемами. Минимальная длительность такта при этом определяется временем выполнения самой медленной команды (например, временем распространения сигнала переноса в сумматоре).

## **Файл-регистр**

## Файл-регистр данных архитектур устроен довольно просто – особенно для базовых конфигураций архитектур.

## Тридцать два – 32-разрядных регистра, два порта на чтение данных (для упрощения делаем их асинхронными), один синхронный порт на запись данных. Регистр файла с адресом 0 – всегда при чтении выдает ноль, на запись не работает.

## Файл-регистр принимает адреса регистров-операндов (rs1, rs2), асинхронно выдает содержимое соответствующих регистров на выходные порты. Параллельно при наличии разрешающего сигнала записывает в регистр с адресом rd значение, поданное на вход.

## При такой реализации выдача значений регистров производится всегда, безотносительно к операции, соответственно, дальнейшая «судьба» данных регистров, поданных на выходы зависит от последующих узлов и требует аккуратного обращения.

**Код 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

);

// 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 (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

## **Тест-бенч:**

`include "rv\_reg\_file.v"

`timescale 10ns/1ns

/\*

 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

\*/

module testbench; // input and output test signals

reg clk;

reg [4:0] rs1;

reg [4:0] rs2;

reg [4:0] rd;

wire [31:0] Rs1\_out;

wire [31:0] Rs2\_out;

reg [31:0] Rd\_input;

reg we;

rv\_reg\_file dut ( clk, rs1, rs2, rd, Rs1\_out, Rs2\_out, Rd\_input, we);

integer i;

initial

begin

   clk = 1'b0; rs1 = 5'h00; rs2 = 5'h01; rd = 5'h02; we=1'b0;

   #10;

   i = 0;

   rs1 <= #i 5'h00; rs2 <= #i 5'h01; rd <= #i 5'h02; Rd\_input <= #i 32'h01; we <= #i 1'b0; i = i + 20;

   rs1 <= #i 5'h01; rs2 <= #i 5'h01; rd <= #i 5'h01; Rd\_input <= #i 32'h77; we <= #i 1'b0; i = i + 20; //we = 1'b1;

   rs1 <= #i 5'h02; rs2 <= #i 5'h01; rd <= #i 5'h02; Rd\_input <= #i 32'h01; we <= #i 1'b0; i = i + 20; //we = 1'b1;

   rs1 <= #i 5'h03; rs2 <= #i 5'h02; rd <= #i 5'h03; Rd\_input <= #i 32'h01; we <= #i 1'b0; i = i + 20;

   rs1 <= #i 5'h04; rs2 <= #i 5'h04; rd <= #i 5'h04; Rd\_input <= #i 32'h01; we <= #i 1'b1; i = i + 20;

   rs1 <= #i 5'h05; rs2 <= #i 5'h04; rd <= #i 5'h05; Rd\_input <= #i 32'h01; we <= #i 1'b0; i = i + 20; //we = 1'b0;

   rs1 <= #i 5'h00; rs2 <= #i 5'h01; rd <= #i 5'h06; Rd\_input <= #i 32'h01; we <= #i 1'b0; i = i + 20;

   rs1 <= #i 5'h00; rs2 <= #i 5'h01; rd <= #i 5'h07; Rd\_input <= #i 32'h01; we <= #i 1'b0; i = i + 20;

   for(i=0; i < 80; i = i+1)

      #10 clk = ~clk;

end

initial

  $monitor("Rs1=%h Rs2=%h ",

            Rs2\_out, Rs2\_out);

initial

  $dumpvars;  //iverilog dump init

endmodule

## 

Рис.1 Временные диаграммы тестирования файл-регистра.

**АЛУ**

Арифметико-логическое устройство является одним из центральных узлов любого процессора. Структура АЛУ и его функциональность определяется ISA, поэтому вернемся к еще раз к базовым операциям RISC-V.

**Операции с регистрами и константами** (immediate/непосредственными значениями) – тип-I - оперируют с регистрами и непосредственными операндами – арифметико-логические операции, сравнения с константами, загрузкой данных в регистры

|  |  |  |  |
| --- | --- | --- | --- |
|  | **funct3** | **opcode** | **I-type** |
| Rd = Rs1+imm  (NOP encoded as ADDI x0, x0, 0.) | 000 | 0010011 | ADDI |
| Rd= 1 if Rs< imm else 0 | 010 | 0010011 | SLTI |
| Rd= 1 if Rs< uimm else 0 | 011 | 0010011 | SLTIU |
| Rd = rs1 XOR imm | 100 | 0010011 | XORI |
| Rd = rs1 OR imm | 110 | 0010011 | ORI |
| Rd = rs1 AND imm | 111 | 0010011 | ANDI |

**Загрузка в регистры непосредственных значения (immediate)**

|  |  |  |
| --- | --- | --- |
|  | **opcode** | **U-type** |
| Rd = imm[31:12][012] | 0110111 | LUI |
| Rd=PC+ imm[31:12] [012] | 0010111 | AUIPC |

**Группа регистровых операций (R-типа)**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **funct7** | **funct3** | **opcode** | **R-type** |
| Rd = Rs1 + Rs2 | 0000000 | 000 | 0110011 | ADD |
| Rd = Rs1 - Rs2 | **0100000** | **000** | **0110011** | **SUB** |
| Rd = Rs1 << Rs2 | 0000000 | 001 | 0110011 | SLL |
| Rd= 1 if Rs1 < Rs2 else 0 | 0000000 | 010 | 0110011 | SLT |
| Rd= 1 if Rs1 < Rs2 else 0(unsign)  SLTU rd, x0, rs2 sets rd to 1  if rs2 is not equal to zero | 0000000 | 011 | 0110011 | SLTU |
| Rd = Rs1 XOR Rs2 | 0000000 | 100 | 0110011 | XOR |
| Rd = Rs1 >> Rs2 | 0000000 | 101 | 0110011 | SRL |
| Rd = Rs1 >> Rs2 (sign ext) | **0100000** | **101** | **0110011** | **SRA** |
| Rd = Rs1 OR Rs2 | 0000000 | 110 | 0110011 | OR |
| Rd = Rs1 AND Rs2 | 0000000 | 111 | 0110011 | AND |
| Rd = Rs1 << imm | 0000000 | 001 | 0010011 | SLLI |
| Rd = Rs1 >> imm (0 ext) | 0000000 | 101 | 0010011 | SRLI |
| Rd = Rs1 >> imm (sign ext) | **0100000** | **101** | **0010011** | **SRAI** |

**Переходы**. Все инструкции условных переходов используют формат команд B-типа (переход в пределах 4КБ).

|  |  |  |  |
| --- | --- | --- | --- |
|  | **funct3** | **opcode** | **B-type** |
| If (Rs1 == Rs2)  PC=PC+imm | 000 | 1100011 | BEQ |
| If (Rs1 != Rs2)  PC=PC+imm | 001 | 1100011 | BNE |
| If (Rs1 <Rs2)  PC=PC+imm | 100 | 1100011 | BLT |
| If (Rs1 >Rs2)  PC=PC+imm | 101 | 1100011 | BGE |
| If (Rs1 <Rs2) –usign  PC=PC+imm | 110 | 1100011 | BLTU |
| If (Rs1 >Rs2) –usign  PC=PC+imm | 111 | 1100011 | BGEU |

**Безусловный переход** относительно значения регистра с записью в регистр связи

|  |  |  |  |
| --- | --- | --- | --- |
|  | **funct3** | **opcode** | **I-type** |
| **Rd = PC+4 PC= Rs + imm** | 000 | 1100111 | JALR |

**Команда простого безусловного перехода** с сохранением адреса возврата в регистр

(переход без сохранения адреса возврата осуществляется, если в качестве регистра указан x0)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Imm[20|10:1|11|19:12]** | **rd** | **opcode** | **J-type** |
| **Rd = PC + 4**  **PC = PC + imm** | imm[20|10:1|11|19:12] | rd | 1101111 | JAL |

**Операции с памятью**

Инструкции записи в память – S-тип.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **imm[11:5]** | **rs2** | **rs1** | **funct3** | **imm[4:0]** | **opcode** | **S-type** |
| [Rs1+imm]=Rs2(byte) | imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| [Rs1+imm]=Rs2(half) | imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| [Rs1+imm]=Rs2(word) | imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

Чтение из памяти (I-тип, как бы неожиданно это не выглядело)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **funct3** | **opcode** | **I-type** |
| Rd = @(Rs + imm) 8bit | 000 | 0000011 | LB |
| Rd = @(Rs + imm) 16bit | 001 | 0000011 | LH |
| Rd = @(Rs + imm) 32bit | 010 | 0000011 | LW |
| Rd = @(Rs + imm) 8bit zero ext | 100 | 0000011 | LBU |
| Rd = @(Rs + imm) 16bit zero ext | 101 | 0000011 | LHU |

**Работа с регистрами специального назначения**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **funct3** | **Опкод** | **I-type** |
| Rd= CSR[imm]  CSR = Rs1 | 001 | 1110011 | CSRRW |
| Rd= CSR[imm]  CSR[imm] = CSR[imm] or Rs1 | 010 | 1110011 | CSRRS |
| Rd= CSR[imm]  CSR[imm] = CSR[imm] and (~Rs1) | 011 | 1110011 | CSRRC |
| Rd= CSR[imm]  CSR[imm] = zimm | 101 | 1110011 | CSRRWI |
| Rd= CSR[imm]  CSR[imm] = CSR[imm] or zimm | 110 | 1110011 | CSRRSI |
| Rd= CSR[imm]  CSR[imm] = CSR[imm] and (~zimm) | 111 | 1110011 | CSRRCI |

Анализируем далее систему команд :

- основные операции осуществляются между двумя операндами;

- есть операции, требующие аппаратного параллелизма – сравнение операндов и вычисление нового адреса перехода, сохранение инкрементированного программного счетчика и вычисление нового адреса (JAL, JAR).

Программный счётчик в нашей микро архитектуре уже вынесен в отдельный блок с функциями счетчика с опцией параллельной загрузки – при линейном выполнении программного кода PC автоматически инкрементируется (+4 к текущему значению), в случае переходов – инициируется новым значением.

Поскольку в командах перехода со связыванием (*jal, jarl*) в регистр связи записывается значение инкрементированное значение программного счётчика (PC+4), в его модуле организуем параллельно к выходу значения PC и вывод PC+4. Такое решение, возможно, не слишком каноническое, но оно позволяет уже иметь значение PC+4 «под рукой» .

module rv\_pc

#(

parameter WIDTH=32

)

(

  input clk,

  input rst\_n,

  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(pc\_load==1'b1)

        pc\_reg <= pc\_next;

      else

        pc\_reg <= pc\_reg + 3'h4;

    end

///\*

assign pc = pc\_reg;

always @ \*

  begin

  pc\_plus <= pc\_reg + 3'h4;

  end

// \*/

endmodule

Для поддержки операций условных переходов выделим отдельный блок сравнения принимающий на вход значения регистров, код операции сравнения (funct3) и выдающий сигнал, разрешающий или не разрешающий переход. Модуль представляет собой относительно небольшой модуль на case-ах.

В этот же модуль вынесем формирование сигнала разрешения параллельной загрузки в регистр программного счётчика нового значения – при командах переходов и возвратов из обработчиков прерываний (mret, например, или также можно добавить в этот список команды ecall, ebreak, хотя для ядер микроконтроллерного класса их поддержка не сильно актуальна).

module rv\_cmp

( input [6:0] opcode,

  input [2:0] funct3,

  input [31:0] Rs1,

  input [31:0] Rs2,

  output reg pc\_new

);

always @ \*

begin

  case (opcode)

    7'b11011\_11 : begin

        $display("JAL");

        pc\_new <= 1'b1;

    end

    7'b11001\_11 : begin

        $display("JALR");

        pc\_new <= 1'b1;

    end

    7'b11100\_11 : begin

        $display("MRET/ecall/ect...");

        pc\_new <= 1'b1;

    end

    7'b11000\_11 : begin // Relative (rs1) jump and link in register

        $display("BRANCH");

        case (funct3)

            3'h0: begin

                pc\_new <= Rs1 == Rs2;

            end

            3'h1: begin

                pc\_new <= Rs1 != Rs2;

            end

            3'h4: begin

                pc\_new <= Rs1 < Rs2;

            end

            3'h5: begin

                pc\_new <= Rs1 >= Rs2;

            end

            3'h6: begin

                pc\_new <= {1'b0,Rs1} < {1'b0,Rs2};

            end

            3'h7: begin

                pc\_new <= {1'b0,Rs1} >= {1'b0,Rs2};

            end

            default: begin

                pc\_new <= 1'b0;

            end

        endcase

        end

    default: begin

        $display ("default");

        pc\_new <= 1'b0;

        end

  endcase

end

endmodule

Чтобы не усложнять структуру и поведенческое описание АЛУ (не делать его очень многовходовым – как минимум Rs1, Rs2, PC, imm, вынесем функционал выбора операндов на откуп специальному мультиплексору. АЛУ таким образом, имеет два входных порта-операнда, выход результата и вход управления выбором функции (зависит от полей opcode, funct3, funct7), что соответствует тому факту, что все операции в базовом наборе RV32I выполняются исключительно над двумя операндами – трех и более операндных действий не требуется.

Код ALU:

module rv\_alu\_v

#(

parameter WIDTH=32

)

(

  input [6:0] opcode,

  input [2:0] funct3,

  input [6:0] funct7,

  input [31:0] Op1,

  input [31:0] Op2,

  output reg [31:0] Rez

);

always @ \*

begin

  case (opcode)

    //I-type

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

        $display("LOAD");

            Rez <= Op1 + Op2;

        end

    7'b00011\_11 : begin // fence

        $display("fence");

            Rez <= Op1;

        end

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

        $display("OP-IMM");

            `include "alu\_op\_base.v"

        end

    7'b01100\_11 : begin // reg with reg operations

        $display("OP-REG");

            `include "alu\_op\_base.v"

        end

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

        $display("RET\_SYST or CSR");

            `include "alu\_op\_CSR.v"

        end

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

        $display("JALR");

        Rez <= Op1 + Op2;

        end

    //J-type

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

        $display("JAL");

        Rez <= Op1 + Op2;

        end

    //S-type

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

        $display("Store");

        Rez <= Op1 + Op2;

        end

    //U-type

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

        $display("LUI");

        Rez <= Op1;

        end

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

        $display("AUIPC");

        Rez <= Op1 + Op2;

        end

    //B-type

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

        $display("BRANCH");

        Rez <= Op1 + Op2;

        end

    default: begin

        $display ("default");

        Rez <= Op1;

        end

  endcase

end

endmodule

Как видно, арифметико-логических операций в базовом варианте не много. Адресные операции (и для подготовки адресов переходов и для адресов обращения к памяти) сводятся к сложению.

Описания арифметических операций с регистрами (включая и операции с регистрами специального назначения) вынесены в отдельные файлы - "alu\_op\_base.v" и "alu\_op\_CSR.v". Это несколько сократило само текстовое описание АЛУ и проще их модифицировать при необходимости.

"alu\_op\_base.v" – набор арифметико-логических операций базового набора команд.

        case (funct3)

            3'h0: begin

                if (funct7 == 7'h20) begin Rez <= Op1 + Op2; end

                else begin Rez <= Op1 - Op2; end

                end

            3'h1: begin Rez <= Op1 << Op2; end

            3'h2: begin Rez <= (Op1 < Op2) ? 32'h1 : 32'h0; end

            3'h3: begin Rez <= ({1'b0,Op1} < {1'b0,Op2}) ? 32'h1 : 32'h0; end

            3'h4: begin Rez <= Op1 << Op2; end

            3'h5: begin

                if (funct7 == 7'h20) begin Rez <= Op1 >>> Op2; end

                else begin Rez <= Op1 >> Op2; end

                end

            3'h6: begin Rez <= Op1 | Op2; end

            3'h7: begin Rez <= Op1 & Op2; end

            default: begin Rez <= Op2; end

        endcase

***Как упоминалось ранее, расширить набор математических операций в рамках набора I можно за счет поля funct7. В представленной версии кода достаточно, например, добавить выборы типа “case (funct7)” в вариантах выбора по полю funct3.***

Набор операций практически типовой. Также можно видеть возможность расширения набора команд в рамках регистровых за счет дополнительного кодирования операций в поле *‘funct7’*.

Аналогичная ситуация наблюдается и в операциях с регистрами специального назначения.

        case (funct3)

            3'h0: begin Rez <= Op1; end

            3'h1: begin Rez <= Op1; end

            3'h2: begin Rez <= Op1 | Op2; end

            3'h3: begin Rez <= Op1 & (~Op2); end

            3'h4: begin Rez <= Op1; end

            3'h5: begin Rez <= Op1; end

            3'h6: begin Rez <= Op1 | Op2; end

            3'h7: begin Rez <= Op1 & (~Op2); end

            default: begin Rez <= Op1; end

        endcase

Ещё один модуль который будет необходим – мультиплексор данных для записи в файл-регистр по адресу rd – в него будут записаны данные или с выхода АЛУ, или данные из оперативной памяти по вычисленному адресу.

module rv\_rez\_mux

#(

parameter WIDTH=32

)

(

  input [6:0] opcode,

  input [2:0] funct3,

  input [6:0] funct7,

  input [WIDTH-1:0] Rez,

  input [WIDTH-1:0] Pc\_plus,

  input [WIDTH-1:0] Mem\_data,

  output reg [WIDTH-1:0] Rd

);

always @ \*

begin

  case (opcode)

    //I-type

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

        $display("LOAD");

            Rd <= Rez;

        end

    7'b00011\_11 : begin // fence

        $display("fence");

            Rd <= Rez;

        end

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

        $display("OP-IMM");

            Rd <= Rez;

        end

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

        $display("OP-REG");

            Rd <= Rez;

        end

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

        $display("RET\_SYST or CSR");

            Rd <= Rez;

        end

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

        $display("JALR");

        case (funct3)

            3'h0: begin Rd <= Pc\_plus; end

            default: begin Rd <= Pc\_plus; end

        endcase

        end

    //J-type

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

        $display("JAL");

            Rd <= Pc\_plus;

        end

    //S-type

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

        $display("Store");

            Rd <= Rez;

        end

    //U-type

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

        $display("LUI");

            Rd <= Rez;

        end

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

        $display("AUIPC");

        case (funct3)

            3'h0: begin Rd <= Rez; end

            default: begin Rd <= Rez; end

        endcase

        end

    //B-type

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

        $display("BRANCH");

        case (funct3)

            3'h0: begin Rd <= Rez; end

            3'h1: begin Rd <= Rez; end

            3'h4: begin Rd <= Rez; end

            3'h5: begin Rd <= Rez; end

            3'h6: begin Rd <= Rez; end

            3'h7: begin Rd <= Rez; end

            default: begin Rd <= Rez; end

        endcase

        end

    default: begin

        $display ("default");

            Rd <= Rez;

        end

  endcase

end

endmodule

И финальный блок формально не входящий в архитектуру процессорного ядра – оперативная память и память программ. С учетом специфики реализации блоков памяти в большинстве семейств FPGA память для процессорного ядра логичнее реализовать в виде (или на основе) двухпортовой памяти – один порт будет использован для чтения данных из памяти – функция памяти программ, второй порт – для чтения/записи данных – функция оперативной памяти.

Вариант кода описания модуля:

module rv\_mem

#(

  parameter DATA\_WIDTH=32,

  parameter ADDR\_WIDTH=10

  )

( input clk,

  input [(ADDR\_WIDTH-1):0] i\_addr,

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

  input [(ADDR\_WIDTH-1):0] d\_addr,

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

  input [(DATA\_WIDTH-1):0] d\_in,

  input we

);

// ROM array

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

// read ROM content from file

initial

$readmemh("rom.txt",rom);

always @ (posedge clk)

  begin

    if (we) rom[d\_addr] <= d\_in;

  end

always @ (\*)

  begin

    code\_out <= rom[i\_addr];

  end

endmodule

В некоторых случаях удобнее будет реализовать две отдельные памяти – память для кода, память для данных – как некие сильно упрощенные варианты кэша инструкций и кэша данных.

Общая структура процессора (вместе с блоками памятей – и программ и данных) представлена на рис.2. Блоки выстроены в условно-линейном порядке прохождения сигнала, сквозные сигналы также проброшены вдоль блоков. Это

Один из портов блока памяти **rv\_mem** адресуется программным счетчиком (выход РС), соответствующее выходное слово инструкции - *iw* поступает параллельно на входы модулей **rv\_desh** и **rv\_imm**. Данные модули «разбирают» поступившую инструкцию выделяя из нее адреса регистров-операндов файл-регистра, формируют непосредственное числовое значение (immediate), сопровождая декодированные элементами сигналами стробов.

Значения выбранных регистров регистрового файла, числовое значение (immediate), программный счетчик, значения выбранных регистров специального назначения (опционально) поступают на вход мультиплексора операндов **rv\_ops\_mux**, параллельно выхода файл-регистра подаются на вход компаратора (вернее его называть блоков разрешения перехода) совместно с комбинацией опкода с модификаторами (*opcode*, *funct3*, *funct7*).

Парные выходы мультиплексора также вместе с кодами операции поступают на вход АЛУ (модуль **rv\_alu\_v**). АЛУ применяется и для выполнения операций над операндами, и для расчета адресов – или адресов переходов, или адресов для обращения к памяти данных (ОЗУ). В зависимости от операции результат АЛУ, инкрементированный программный счетчик (PC+4) или данные из оперативной памяти записывается в регистр с номером *rd* файл-регистра.

Блок регистров специального назначения добавлен, как опциональный (выделен зелёным цветом).

Выходной сигнал блока сравнения/управления переносом и новое вычисленное значение подаются на соответствующие входы программного счетчика **rv\_pc**.

Возможные недостатки предлагаемого решения:

- операции с памятью становятся критичными и скорее всего их придётся делать за два такта;

- исполнение программного счётчика далеко от «классических» вариантов;

- наличие относительно большого числа «сквозных» сигналов;

- наличие двух достаточно «широких» мультиплексоров – в перспективе (если ее, конечно представлять) масштабирования разрядности реализация процессорного ядра может быть «весомой».

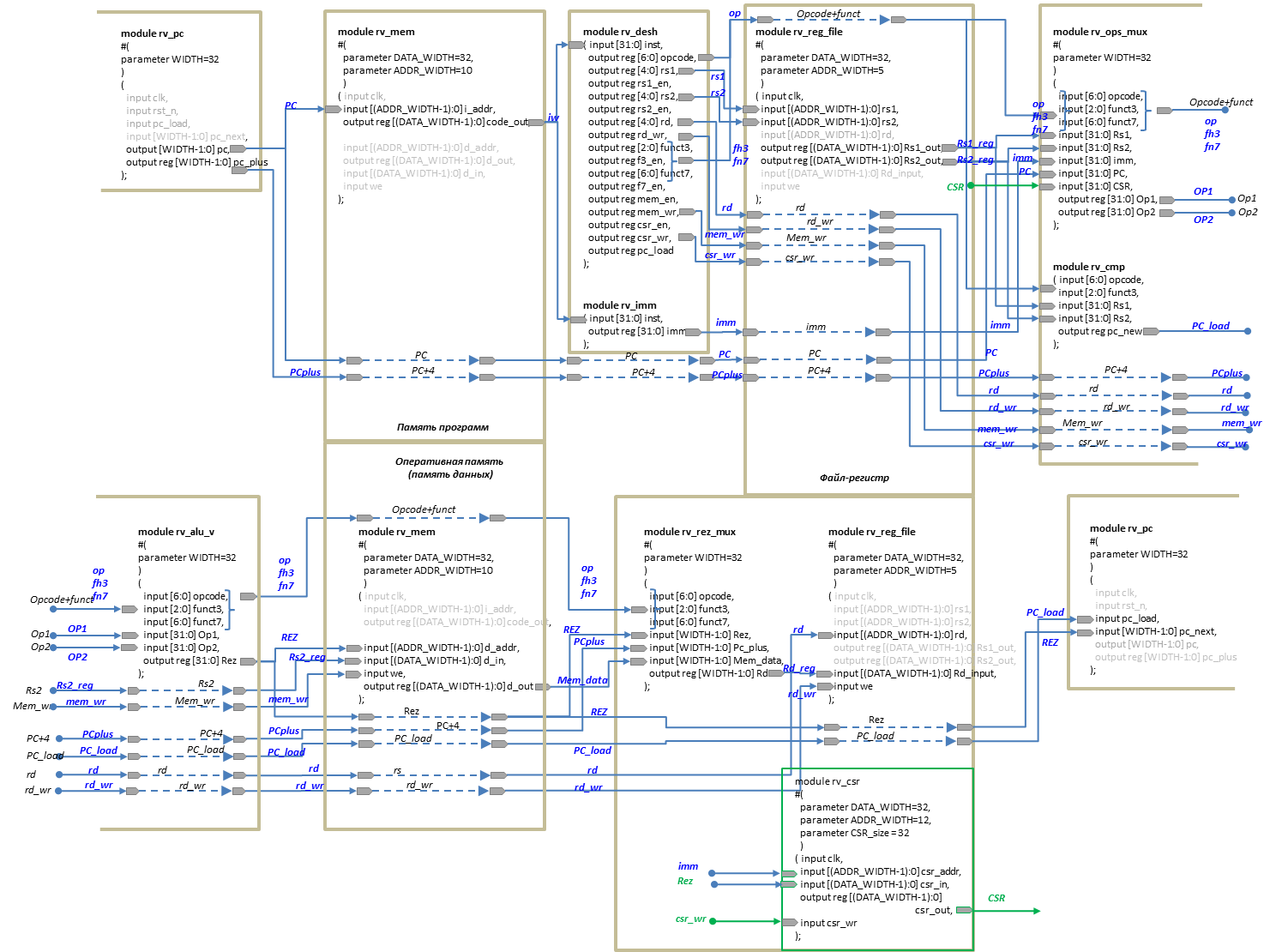


Рис. 2 Структурная схема процессора.

Код-заглушка для блока CSR-регистров:

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

);

// 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) 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

И, собственно, «код объединяющий всё» (нотации и сигналы сохранены по возможности близко к рис.2):

`include "rv\_pc.v"

`include "rv\_mem.v"

`include "rv\_desh.v"

`include "rv\_imm.v"

`include "rv\_reg\_file.v"

`include "rv\_ops\_mux.v"

`include "rv\_cmp.v"

`include "rv\_alu\_v.v"

`include "rv\_rez\_mux.v"

`include "rv\_csr.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

);

// wires

wire [(DATA\_WIDTH-1):0] PC;

wire [(DATA\_WIDTH-1):0] PCplus;

wire [(DATA\_WIDTH-1):0] iw;

wire [(DATA\_WIDTH-1):0] imm;

wire [(DATA\_WIDTH-1):0] Rs1\_reg;

wire [(DATA\_WIDTH-1):0] Rs2\_reg;

wire [(DATA\_WIDTH-1):0] Rd\_reg;

wire [(DATA\_WIDTH-1):0] Op1;

wire [(DATA\_WIDTH-1):0] Op2;

wire [(DATA\_WIDTH-1):0] Rez;

wire [(DATA\_WIDTH-1):0] CSR;

wire [(DATA\_WIDTH-1):0] Mem\_data;

wire [6:0] op;

wire [2:0] fn3;

wire [6:0] fn7;

wire [4:0] rs1;

wire [4:0] rs2;

wire [4:0] rd;

wire rd\_wr;

wire mem\_wr;

wire csr\_wr;

wire PC\_load;

//assign Data\_In = CSR;

assign Data\_out = CSR;

// CPU modules

rv\_pc pc(

  .clk(clk),

  .rst\_n(rst),

  .pc\_load(PC\_load),

  .pc\_next(REZ),

  .pc(PC),

  .pc\_plus(PCplus)

);

rv\_mem mem(

  .clk(clk),

  .i\_addr(PC),

  .code\_out(iw),

  .d\_addr(REZ),

  .d\_out(Mem\_data),

  .d\_in(Rs2\_reg),

  .we(mem\_wr)

);

rv\_desh desh(

  .inst(iw),

  .opcode(op),

  .rs1(rs1),

  .rs1\_en(),

  .rs2(rs2),

  .rs2\_en(),

  .rd(rd),

  .rd\_wr(rd\_wr),

  .funct3(fn3),

  .f3\_en(),

  .funct7(fn7),

  .f7\_en(),

  .mem\_en(),

  .mem\_wr(mem\_wr),

  .csr\_en(),

  .csr\_wr(crsr\_wr),

  .pc\_load()

);

rv\_imm immed(

  .inst(iw),

  .imm(imm)

);

rv\_reg\_file reg\_file(

  .clk(clk),

  .rs1(rs1),

  .rs2(rs2),

  .rd(rd),

  .Rs1\_out(Rs1\_reg),

  .Rs2\_out(Rs2\_reg),

  .Rd\_input(Rd\_reg),

  .we(rd\_wr)

);

rv\_cmp cmp(

  .opcode(op),

  .funct3(fn3),

  .Rs1(Rs1\_reg),

  .Rs2(Rs2\_reg),

  .pc\_new(PC\_load)

);

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\_alu\_v alu\_v(

  .opcode(op),

  .funct3(fn3),

  .funct7(fn7),

  .Op1(Op1),

  .Op2(Op2),

  .Rez(Rez)

);

rv\_rez\_mux rez\_mux(

  .opcode(op),

  .funct3(fn3),

  .funct7(fn7),

  .Rez(REZ),

  .Pc\_plus(PCplus),

  .Mem\_data(Mem\_data),

  .Rd(Rd\_reg)

);

rv\_csr csr(

  .csr\_addr(imm[11:0]),

  .csr\_in(REZ),

  .csr\_out(CSR),

  .csr\_wr(csr\_wr)

);

endmodule