**Разработка простейшего декодера команд RISC-V**

RISC-V своеобразная архитектура, ведущая свои корни изначально из микроконтроллерного сегмента и только потом расширенная на работу с операционными системами общего назначения, работу в условиях многозадачности, необходимости разделения и защиты областей памяти.

Это отражено в спецификации архитектуры в виде ряда понятий:

* Hart (Аппаратный поток) — архитектура RISC-V поддерживает многопоточность, поэтому может быть несколько аппаратных потоков исполнения кода. Под потоком (hart) подразумевается аппаратный поток – выполнение которого (или точнее «хранение архитектурного состояния») обеспечивается дублирующими регистрами (теневыми регистрами). Естественно полагать, что микропроцессор должен иметь как минимум один поток (hart) с ID равным 0.

Практически любой микропроцессор функционирует в окружении некоторого количества периферийных устройств, практически все из них могут функционировать независимо от него, выполнять те или иные функции, генерировать или принимать сигналы, как от самого процессорного ядра, так и извне. Взаимодействие с периферийными устройствами может прерывать выполнение основного кода программы. Также само выполнение кода программы может вызывать ситуации, требующие отдельной реакции. Механизм, описывающий такие ситуации в терминологии RISC-V называется «Trap»:

* Trap(Ловушка) — ловушка это совокупное объедение смысла таких слов, как прерывание и исключение. Ловушки бывают нескольких типов:
  + Ловушка исключения (exception) — это понятие означает синхронное событие, которое прерывает исполнения кода (наверное, самое банальное – попытка деления на ноль). Исключение может прерываться другим исключением, или NMI.
  + Ловушка прерывания (interrupt) — внешнее асинхронное событие, которое может привести к тому, что поток неожиданно может передать управление. Прерывание может прерываться другим прерыванием, NMI, или исключением.
  + Ловушка немаскируемого прерывания(NMI) — немаскируемое прерывание. NMI не может прерываться другим NMI, но может перейти из обработчика NMI в режим обработки исключения, если в момент обработки NMI произойдет исключение.
* Machine (машинный) — В ядре все машинное — регистры, таймер, режим. Поэтому всё, что связано со словом machine (машинный) должно поддерживается на уровне ядра. К примеру большинство микроконтроллеров имеет один уровень привилегий– машинный.

**Уровни привилегий**

Поддержка многопоточности или операционных систем практически всегда порождает понятия уровня привилегий текущего исполняемого кода. В RISC-V архитектуре существует 3 уровня привилегий:

В архитектуре RISC-V существует три уровня привилегий:

* Машинный.
* Пользовательский.
* Уровень супервизора.

Уровни привилегий используются для обеспечения защиты между различными компонентами программного обеспечения (например, пользовательским приложением и ядром операционной системы). Любые попытки выполнения операций, не разрешенных текущим режимом привилегий, вызовут исключение.

Существует две спецификации набора инструкций [1-3]:

* Непривилегированный набор инструкций;
* Привилегированный набор инструкций.

**Непривилегированный набор инструкций [1]**

Спецификация на этот набор описывает инструкции и функциональность которые обычно используются во всех режимах привилегий, т.е. общие для всех архитектур набор инструкций и функций.

**Привилегированный набор инструкций[3]**

Основное её назначение — это разделение уровня приложений и уровня ядра, а также поддержка операционных систем вплоть до нескольких разных операционных систем типа Linux, работающих через виртуальную машину.

Спецификация на привилегированный набор описывает возможную архитектуру привилегированных режимов, в том числе специальные инструкции и дополнительную функциональность для каждого из них [4-6].

Следует уточнить, что эта спецификация носит рекомендованный характер, и она описывает только одно из возможных решений. Основное её преимущество, в том, что привилегированная архитектура никак не задевает основную непривилегированную функциональность и является её расширением.

**Разработка простейшего декодера команд RISC-V**

Чтобы не усложнять задачу, на данном этапе ограничимся непривелигированным базовым набором команд.

Рассмотрим базовую систему команд (I). Несмотря на начальную парадигму RISC, от неё можно заметить только то, что в базовом наборе, да и почти во всех популярных расширениях, команды фиксированной длины – 32-битное слово.

При этом мы имеем регистровые операции, операции с числами (непосредственные значения), работа с памятью, переходы.

При построении декодера команд приятно видеть, что есть жёстко фиксированной поле опкодов, определяющее, по факту, тип команды (да даже и текущий набор команд). Логика расположения дальнейших полей возможно и обоснована [7], но....откровенно говоря, «не блещет красотой»(особенно, что касается непосредственного операнда – imm) - рис.1.

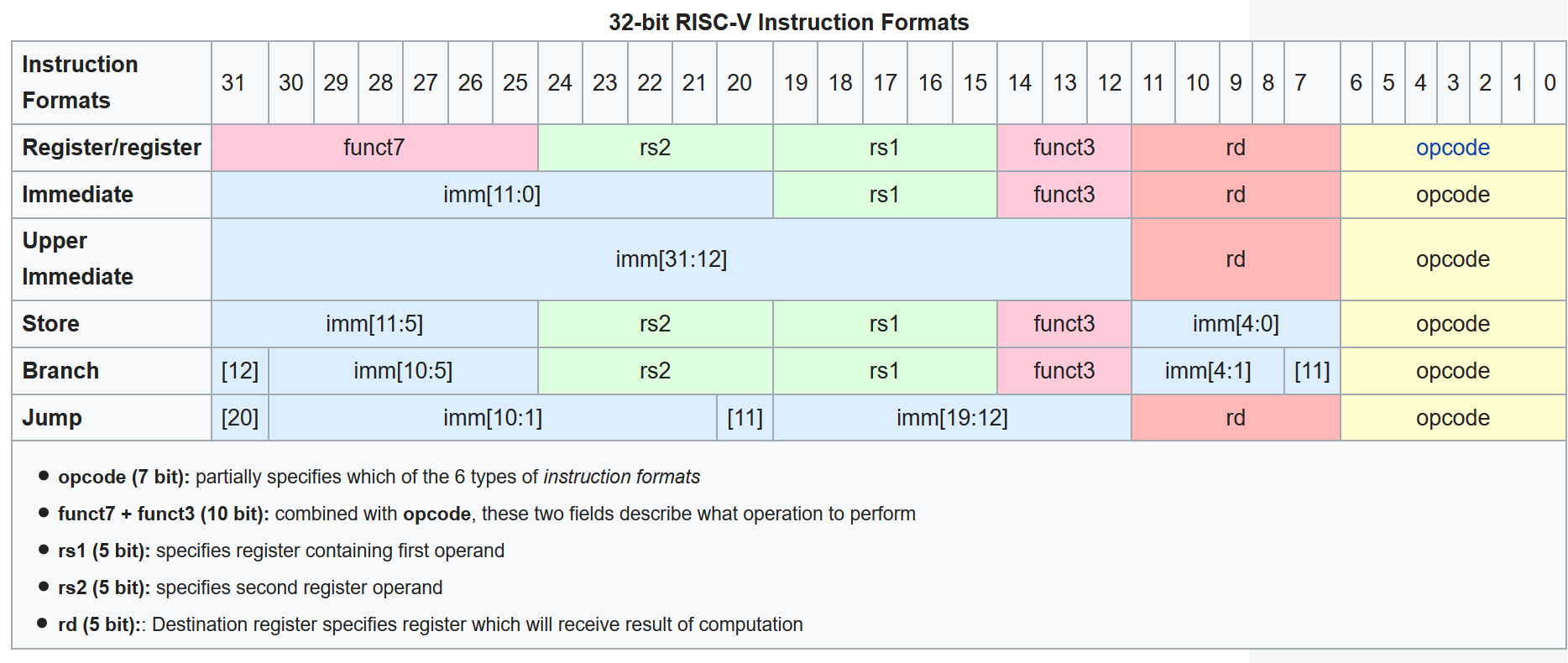


Рис.1 Формат 32-битных инструкций RISC-V.

**Постановка задачи**

Задачи декодера – выбрать опкоды, их расширения (функциональные поля), сформировать числовую константу, сформировать управляющие сигналы разрешения записи с регистровый файл, память, блок специальных регистров, программный счётчик.

Константу проще всего (как кажется автору) проще всего сразу формировать 32-битную, ибо далее везде в базовом наборе команд они рассматриваются как 32-битные знаковые величины.

Отдельная «головная боль» – переходы, тут попадаем в двоякую ситуацию – с одной стороны понятно стремление разработчиков архитектуры к позиционно-независимому коду (все переходы относительно программного счётчика), с другой – а адресам этих переходы придётся считать, учитывая и значения регистров, и программного счетчика, и смещение заданное числовым значением....

**Пробуем провести декомпозицию**

Исходя из описания системы команд (или ISA) можно выделить следующие ключевые блоки процессорного ядра архитектуры RISC-V:

* регистровый файл (31 активный регистр, два выходных порта, один порт на запись);
* программный счётчик (PC);
* память программ;
* память данных (ОЗУ);
* блок регистров специального назначения (CSR);
* арифметико-логическое устройство;
* блок дешифратора команд.

Объединять ли память программ и память данных в единую память по большей части будет зависеть от целевого приложения и можно отнести на уровень микроархитектурных решений. Зачастую, даже для единого блока оперативной памяти стараются выделить независимые каналы для выборки команд, и для чтения/записи данных.

Реализация АЛУ, как дешифратора команд также будет зависеть от микроархитектуры, и отчасти, от тех множеств команд, которые планируется поддерживать. Обычно в минимальных ядра поддерживаются наборы I, E, C., в более «серьёзных» ядрах, ещё и М.

По общей концепции и тенденциях в проектировании расширений архитектуры RISC-V и поддержки наборов команд наподобие плавающей точки, векторных, тензорных операций – для них добавляются свои выделенные регистры, при этом сами команды остаются 32-разрядными.

Структура и техническая реализация дешифратора команд также будет зависеть от поддерживаемых наборов команд, но, по возможности и в силу пониманию, будем придерживаться модульного подхода с тем, чтобы дальнейшая модификация была возможна. Поскольку прямых упоминаний, как должны выбираться из памяти команды, какой память должна быть разрядности, что происходит в случае одновременного присутствия в исполняемом коде обычных инструкций и сокращенных, а также для упрощения дальнейшего рассмотрения, ограничимся разбором только команд в 32-битном формате, равно как и примем размер ячеек памяти равным 32 битам.

В базовой ISA существует четыре основных формата команд (R/I/S/U), как показано на рисунке 2. Все они имеют фиксированную длину в 32 бита и должна быть выровнена по четырехбайтовой границе в памяти.

Для принятой ветви или безусловного перехода генерируется исключение со смещенным адресом инструкции, если целевой адрес не выровнен по четырем байтам. Для условной ветви, которая не принята, не генерируется исключение со смещенным адресом выборки команд.

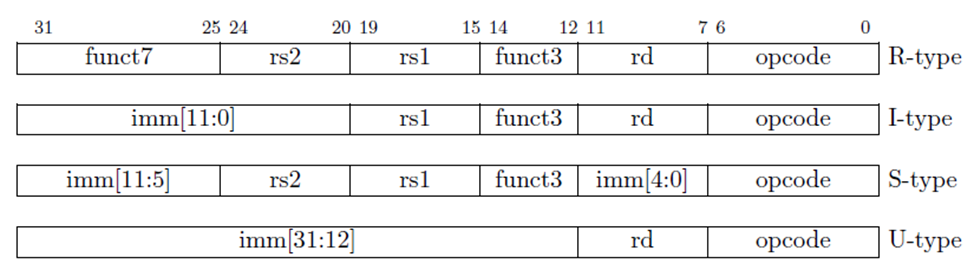
****

Рис.2 Типы команд базового набора инструкций RISC-V.

При разборе опкодов команд по типам и по расширениям очень помогает битовая карта опкодов - рис.3.

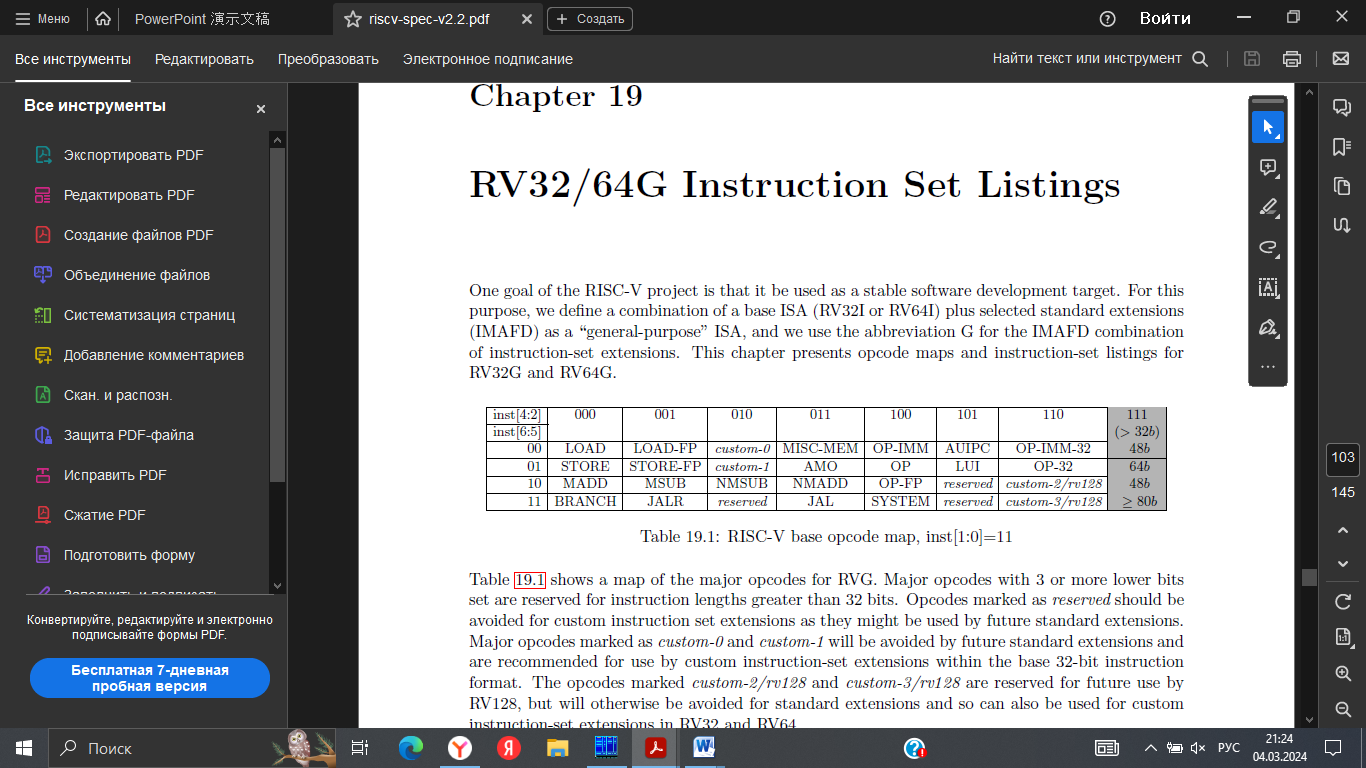


Рис. 3 «Созвездие» опкодов инструкций**.**

Каждое непосредственное подполе помечается позицией бита (imm[x]) в создаваемом непосредственном значении, а не позицией бита в непосредственном поле инструкции, как это обычно делается.

RISC-V ISA (рис.3) сохраняет исходный (rs1 и rs2) и целевой (rd) регистры в одном и том же положении во всех форматах для упрощения декодирования (почти единственный видимый плюс кодирования команд, да простят автора истовые поклонники RV). За исключением 5-разрядных немедленных значений, используемых в инструкциях CSR, непосредственные значения всегда расширяются по знаку и, как правило, упаковываются в крайние левые доступные биты в инструкции и были выделены для уменьшения аппаратной сложности. В частности, знаковый бит для всех непосредственных действий всегда находится в бите 31 инструкции для ускорения схемы расширения знака.

Существуют еще два варианта форматов команд (B/J). Единственное различие между форматами S и B заключается в том, что 12-разрядное поле immediate используется для кодирования наборов ответвлений o, кратных 2 в формате B. Вместо того, чтобы сдвигать все биты в закодированном инструкцией непосредственном левом на единицу в аппаратном обеспечении, как это обычно делается, средние биты (imm[10:1]) и знаковый бит остаются в фиксированных положениях, в то время как младший бит в формате S (inst[7]) кодирует бит старшего разряда в инструкциях формата B.

Аналогично, единственное различие между форматами U и J заключается в том, что 20-битный immediate сдвигается влево на 12 бит для формирования U immediate и на 1 бит для формирования J immediate. Расположение командных битов в форматах U и J immediate выбрано таким образом, чтобы максимально перекрывать другие форматы и друг друга.

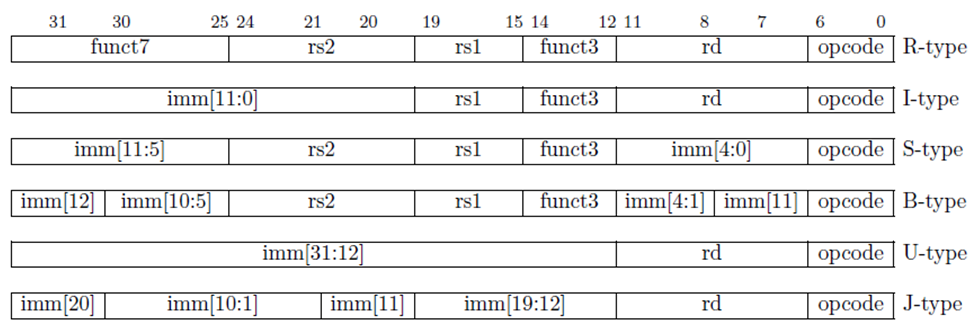
****

Рис. 4 Базовые форматы команд RISC-V, показывающие варианты размещения immediate с слове инструкции.

Ну и самое «весёлое» - собирать immediate по слову инструкции. Вишенкой на торте великой и свободной архитектуры RV, победно шествующей по миру – является именно кодировка в командах констант и непосредственных значений. Она вызывает прям бурю эмоций и восторга своей красотой и элегантностью технических решений. Небольшая шпаргалка на эту тему есть в спецификации – рис.5.

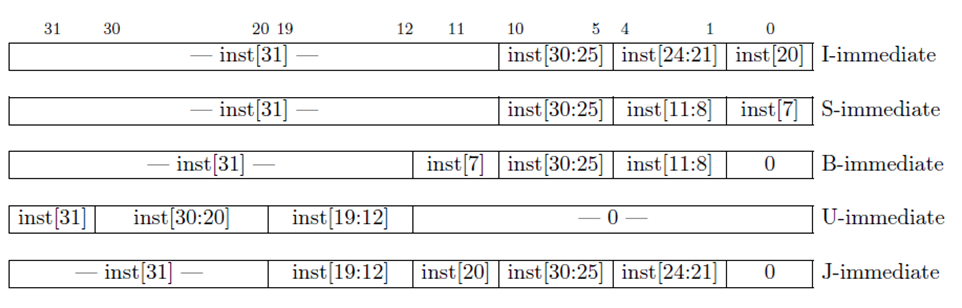
****

Рис. 5 Непосредственные значения, создаваемые инструкциями RISC-V. Поля помечены битами команд, используемыми для построения их значения. Расширение знака всегда использует бит inst[31].

Расширение знака всегда использует inst[31]. Расширение знака является одной из наиболее важных операций с immediates (особенно в RV64I), и в RISC-V бит знака для всех immediate всегда хранится в бите 31 инструкции, чтобы позволить расширению знака выполняться параллельно с декодированием команды.

Один из удобных инструментариев полезных при освоении ассемблера RISC-V, а также для целей тестирования разрабатываемых узлов процессора - симулятор RV32 - RARS- <https://github.com/TheThirdOne/rars/releases> – чтоб спокойно генерировать бинарные опкоды команд, а не писать их вручную с риском ошибиться в паре-тройке бит.

## **Декомпозиция…**

## Попробуем разбить декодер на несколько блоков для упрощения описания каждого из них, и для возможно более простой дальнейшей модификации. По факту - надо из кода инструкции выделить сам опкод, коды доп функций команд, адреса регистров и непосредственные значения декодируем отдельно, валидные значения декодированных полей отдельными сигналами.

## Таким образом, декодер составится из блоков:

## - формирователь непосредственных значений;

## - декодер адресов регистров-операндов и функциональных полей.

## **Формирователь непосредственных значений**

## Формирователь непосредственных значений – принимает на вход 32 бита инструкции, на выходе формирует 32-битное непосредственное значение и сигнал «валидности» значения – 1, если из инструкции декодирован immediate, и 0, если его в команде нет – например, для команд R-типа.

## Код:

module rv\_imm

( input [31:0] inst,

  output reg [31:0] imm,

output reg imm\_en

);

wire [6:0] opcode;

wire [19:0] sign;

//wire zero;

assign opcode[6:0] = inst[6:0];

assign sign = {

    inst[31],inst[31],inst[31],inst[31],inst[31],//inst[31],

    inst[31],inst[31],inst[31],inst[31],inst[31],//inst[31],

    inst[31],inst[31],inst[31],inst[31],inst[31],//inst[31],

    inst[31],inst[31],inst[31],inst[31],inst[31]//,inst[31]

            };

//assign zero = 1'b0;

always @ \*

begin

  case (opcode)

    //I-type

    7'b00000\_11 : begin

        imm <= {sign,inst[31:20]}; $display ("LOAD");

imm\_en <= 1'b1;

        end

    7'b00011\_11 : begin

        imm <= {sign,inst[31:20]}; $display ("Misc-MEM");

imm\_en <= 1'b1;

        end

    7'b00100\_11 : begin

        imm <= {sign,inst[31:20]}; $display ("OP-Imm");

imm\_en <= 1'b1;

        end

    7'b11100\_11 : begin

        imm <= {sign,inst[31:20]}; $display ("SYSTEM");

imm\_en <= 1'b1;

        end

    7'b11001\_11 : begin

        imm <= {sign,inst[31:20]}; $display ("JALR");

imm\_en <= 1'b1;

        end

    //J-type

    7'b11011\_11 : begin

        imm <= {sign[19:8],inst[19:12],inst[20],inst[30:25],inst[24:21],1'b0}; //J-type

        $display("JAL");

imm\_en <= 1'b1;

        end

    //S-type

    7'b01000\_11 : begin

        imm <= {sign,inst[31:25],inst[11:7]};

        $display("Store");

imm\_en <= 1'b1;

        end

    //U-type

    7'b01101\_11 : begin

        imm <= {inst[31:12],12'b0};

        $display("LUI");

imm\_en <= 1'b1;

        end

    7'b00101\_11 : begin

        imm <= {inst[31],inst[30:20],inst[19:12],12'b0};

        $display("AUIPC");

imm\_en <= 1'b1;

        end

    //B-type

    7'b11000\_11 : begin

        imm <= {sign,inst[7],inst[30:25],inst[11:8],1'b0};

        $display("BRANCH");

imm\_en <= 1'b1;

        end

    default: begin

        imm <= 32'h00; $display ("default");

imm\_en <= 1'b0;

        end

  endcase

end

endmodule

## Тестбэнч:

`include "rv\_imm.v"

`timescale 10ns/1ns

/\*

  input [31:0] inst,

  output reg [31:0] imm,

output reg imm\_en

\*/

module testbench; // input and output test signals

reg  [31:0] inst;

wire [31:0] imm;

wire [31:0] imm\_en;

// creating the instance of the module we want to test

//  bcd\_to\_sseg - module name

//  dut  - instance name ('dut' means 'device under test')

rv\_imm dut( inst, imm, imm\_en );

// do at the beginning of the simulation

initial

  begin

    inst = 32'h50b7;    // set test signals value

    #10;            // pause

    inst = 32'h0137;    // set test signals value

    #10;            // pause

    inst = 32'h508193;    // set test signals value

    #10;            // pause

    inst = 32'h502083;    // set test signals value

    #10;            // pause

    inst = 32'hfe000ce3;    // set test signals value

    #10;            // pause

    inst = 32'h77237;    // set test signals value

    #10;            // pause

    inst = 32'hc0d093;    // set test signals value

    #10;            // pause

    inst = 32'hff9ff2ef;    // set test signals value

    #10;            // pause

  end    // do at the beginning of the simulation

//  print signal values on every change

initial

  $monitor("inst=%h imm=%h", inst, imm, imm\_en);

// do at the beginning of the simulation

initial

  $dumpvars;  //iverilog dump init

endmodule

## Для проверки в симуляторе введем небольшую программу, а если точнее, то небольшой список инструкций разных типов для тестирования реакции модуля формирования непосредственных значений.

## 

## Рис. 6 Генерируем последовательность команд средствами симулятора RARS.

## 

Рис. 7 Временные диаграммы работы схемы формирования непосредственного операнда.

Декодер адресов регистров-операндов и функциональных полей – принимает на вход 32-бита инструкции, выделяет из него адреса регистров-операндов, сопровождаемые сигналами подтверждения (валидности), генерируемые в том случае, если они присутствуют в коде инструкции, также добавим несколько сигналов для управления такими функциональными блоками, как память(в частности память данных или ОЗУ), регистры специального назначения – разрешения работу блоку, сигнал разрешения записи. Поскольку в нашей декомпозиции программный счетчик выделен в отдельный функциональный блок выделим ему также отдельный сигнал который переводит его из режима инкремента (+4 по спецификации) в режим загрузки нового значения.

## Код:

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,

  output reg pc\_load

);

always @ \*

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

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'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

## Тестбэнч:

`include "rv\_desh.v"

`timescale 10ns/1ns

/\*

  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\_en,

  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,

  output reg pc\_load

\*/

module testbench; // input and output test signals

reg  [31:0] inst;

  wire [6:0] opcode;

  wire [4:0] rs1;

  wire rs1\_en;

  wire [4:0] rs2;

  wire rs2\_en;

  wire [4:0] rd;

  wire rd\_en;

  wire [2:0] funct3;

  wire f3\_en;

  wire [6:0] funct7;

  wire f7\_en;

  wire mem\_en;

  wire mem\_wr;

  wire csr\_en;

  wire csr\_wr;

  wire pc\_load;

// creating the instance of the module we want to test

//  dut  - instance name ('dut' means 'device under test')

rv\_desh dut( inst, opcode, rs1, rs1\_en, rs2, rs2\_en, rd, rd\_en, funct3, f3\_en, funct7, f7\_en,

  mem\_en, mem\_wr, csr\_en, csr\_wr, pc\_load );

// do at the beginning of the simulation

initial

  begin

    inst = 32'h50b7;    // set test signals value

    #10;            // pause

    inst = 32'h0137;    // set test signals value

    #10;            // pause

    inst = 32'h508193;    // set test signals value

    #10;            // pause

    inst = 32'h502083;    // set test signals value

    #10;            // pause

    inst = 32'hfe000ce3;    // set test signals value

    #10;            // pause

    inst = 32'h77237;    // set test signals value

    #10;            // pause

    inst = 32'hc0d093;    // set test signals value

    #10;            // pause

    inst = 32'hff9ff2ef;    // set test signals value

    #10;            // pause

  end    // do at the beginning of the simulation

//  print signal values on every change

initial

  $monitor("inst=%h op=%b %h %h %h %h %h %h %h %h %h %h", inst, opcode, rs1\_en, rs2\_en, rd\_en, f3\_en, f7\_en, mem\_en, mem\_wr, csr\_en, csr\_wr, pc\_load);

// do at the beginning of the simulation

initial

  $dumpvars;  //iverilog dump init

endmodule

## Консольный вывод:

[Running] testbench\_desh.v

VCD info: dumpfile dump.vcd opened for output.

LUI

inst=000050b7 op=0110111 0 0 1 0 0 0 0 0 0 0

LUI

inst=00000137 op=0110111 0 0 1 0 0 0 0 0 0 0

OP-IMM

inst=00508193 op=0010011 1 1 1 1 1 0 0 0 0 0

LOAD

inst=00502083 op=0000011 1 1 1 1 0 1 0 0 0 0

BRANCH

inst=fe000ce3 op=1100011 1 1 0 1 0 0 0 0 0 1

LUI

inst=00077237 op=0110111 0 0 1 0 0 0 0 0 0 0

OP-IMM

inst=00c0d093 op=0010011 1 1 1 1 1 0 0 0 0 0

JAL

inst=ff9ff2ef op=1101111 0 0 1 0 0 0 0 0 0 1

[Done] exit with code=0 in 1.201 seconds

## 

## Рис. 8 Временная диаграмма работы дешифратора команд.

Литература:

1. <https://github.com/riscv/riscv-isa-manual>
2. <https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-spec.pdf>
3. <https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-privileged.pdf>
4. <https://riscv.org/technical/specifications/privileged-isa/>
5. <https://uim.fei.stuba.sk/wp-content/uploads/2018/02/riscv-privileged-2022.pdf>
6. <https://www.linux.org.ru/news/hardware/11636749>
7. The RISC-V Instruction Set Manual. Volume I: User-Level ISA Document Version 2.2 // Editors: Andrew Waterman1, Krste Asanov