Building a Single-cycle RISC V processor in Verilog

Contents

Introduction

Key features

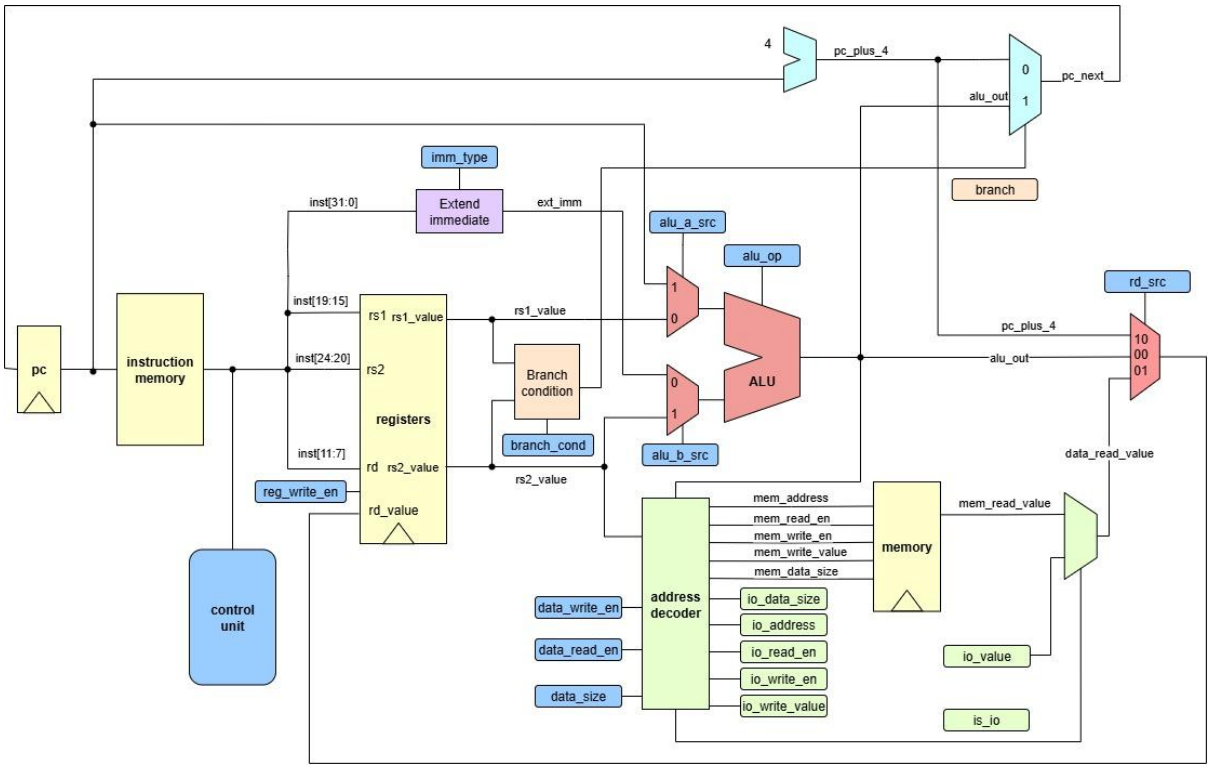
The ALU will process two imputs – a and b. Input a can be either the rs1 value or the program counter. Input b can be either the rs2 vlue or the extended immediate value.

The ALU is used for branch calculations.

There is a separate adder to add 4 to the program counter for the next instruction.

Branches are taken if the branch condition unit calculates a true value. It can compare the rs1 and rs2 values to determine whether to branch. The nature of the comparison is derived by the control unit.

The clocked components are the memory, the register file and the program counter



Not conformance to specification

Misaligned memory loads and stores will not be handled according to the specification – the memory address will be truncated to the lower aligned address. For example, a word access to byte 7 will be truncated down to byte 4.

Control Unit signals

Signal definitions

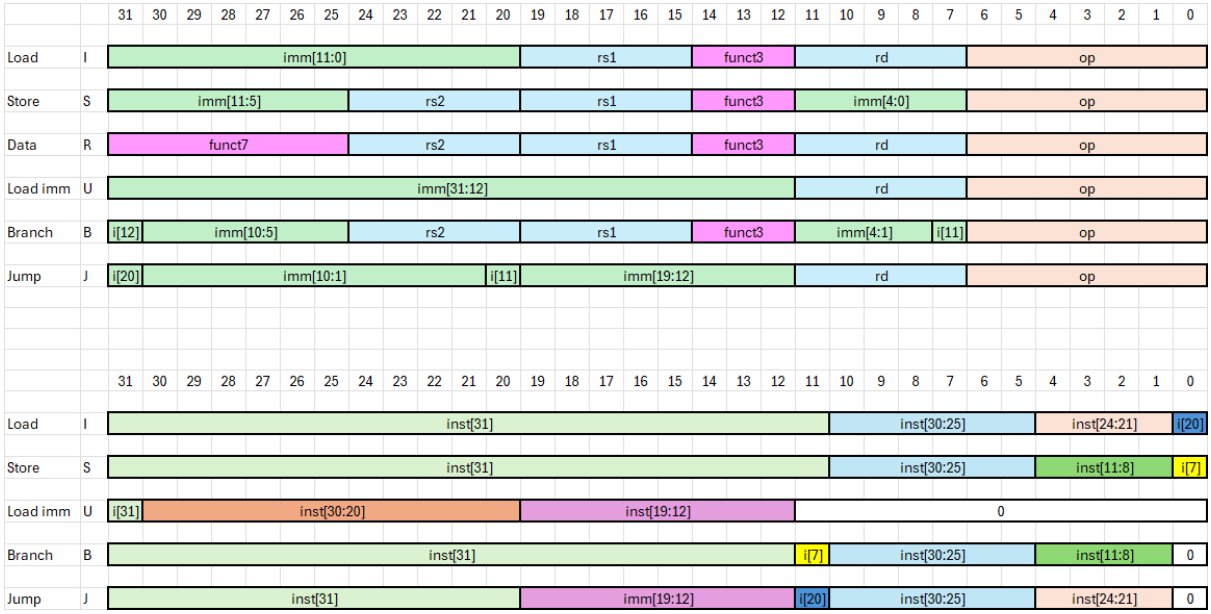
Signal	Definition
imm_type	Type of immediate (B, I, J, S, U, R)
alu_a_src	a input to ALU (pc, rs1)
alu_b_src	b input tp ALU (imm, rs2)
alu_op	Function for ALU to execute
rd_src	Source for rd register (data, pc + 4, alu output)
reg_write_en	Write to rd register
data_read_en	Enable read from data bus
data_write_en	Enable write to data bus
data_size	Data access size of data (byte, half-word, word)
branch_cond	Branch comparison to perform (==, !=, <, >=)

Signals for each opcode

Instruction	opcode	Control Unit signals									
		imm_type	alu_a_src	alu_b_src	rd_src	reg_write_en	data_read_en	data_write_en	data_size	alu_op	branch_cond
arithmetic reg	011_0011	0	0	0	00	1	0	0	000	see table	010
arithmetic imm	001_0011	1	0	1	00	1	0	0	000	see table	010
load	000_0011	1	0	1	01	1	1	0	see table	0000	010
store	010_0011	2	0	1	00	0	0	1	see table	0000	010
branch	110_0011	3	1	1	00	0	0	0	000	0000	see table
jal	110_1111	4	1	1	00	0	0	0	000	0000	011
jalr	110_0111	1	0	1	10	1	0	0	000	0000	011
lui	011_0111	5	0	1	00	1	0	0	000	1001	010
auipc	001_0111	5	1	1	00	1	0	0	000	0000	010

Immediate encoding

instruction	opcode	imm_type	immediate
arithmetic reg	011_0011	0	R
arithmetic imm	001_0011	1	I
load	000_0011	1	I
store	010_0011	2	S
branch	110_0011	3	B
jal	110_1111	4	J
jalr	110_0111	1	I
lui	011_0111	5	U
auipc	001_0111	5	U



Branch condition codes

These specify the comparions for the branch unit to make

For instructions that do not branch this is set to ‘no branch’

For jal and jalr which always branch this is set to ‘always branch’

Fortunately in the ISA there were two values spare to use for these.

Value	Description
000	a == b
001	a != b
010	no branch
011	always branch
100	signed(a) < signed(b)
101	signed(a) >= signed(b)
110	a < b
111	a >= b

Destination register sources

The rd register can store the output of the ALU, the value read from memory or IO, or the location of the next instruction (pc+4)

Value	Description
00	alu_out
01	data_read_value
10	pc + 4
11	

ALU operations

These are the operations the ALU can perform.

The final one will pass the b input to the output (immediate), required for the **lui** instruction.

Apart from the arithmetic instructions, and lui, alu_op will be 0000 (add)

opcode	funct3	funct7[5]	Operation	Function	ALU Op
0?1_0011	000	0	add	alu_out = a + b	0000
0?1_0011	000	1	sub	alu_out = a - b	1000
0?1_0011	001	0	sll	alu_out = a << b[4:0]	0001
0?1_0011	010	0	slt	alu_out = a < signed(b) ? 1 : 0	0010
0?1_0011	011	0	sltu	alu_out = a < unsigned(b) ? 1 : 0	0011
0?1_0011	100	0	xor	alu_out = a ^ b	0100
0?1_0011	101	0	srl	alu_out = a >> b[4:0]	0101
0?1_0011	101	1	sra	alu_out = a >>> b[4:0]	1101
0?1_0011	110	0	or	alu_out = a | b	0110
0?1_0011	111	0	and	alu_out = a & b	0111
				alu_out = b	1001

The arithmetic instruction opcodes are 001_0011 for immediate (addi) and 011_0011for register (add)

For the arithmetic instructions, the input values of a is rs1 and b is either rs2 or the immediate

Value	alu_a	alu_b
001_0011	rs1	immediate
011_0011	rs1	rs2

ALU a and b sources

The a source can be rs1 or pc

The b source can be rs2 or the immediate

Value	Source
0	rs1
1	pc

Value	Source
0	rs2
1	ext_imm

Load and store

Xxxxx

Data access sizes

Memory and IO access can be of various data sizes: byte, half-word or word. Data reads can be extended as signed or unsigned.

Value	Description
000	byte
001	half-word
010	word
011	
100	unsigned byte
101	unsigned half-word
110	
111	

Full design (with Nexys 4 DDR IO)

Verilog file: top.v

```
`timescale 1ns / 1ps

module top(
  input      CLK100MHZ,
  input      [15:0] SW,
  input      [4:0]  BTN,
  output reg [15:0] LED
  );

  wire [31:0] io_address;
  wire [31:0] io_write_value;
  reg  [31:0] io_read_value;
  wire        io_write_en;
  wire        io_read_en;
  wire [2:0]  io_data_size;

  Risc32 risc(
    .clk(CLK100MHZ),
    .io_address(io_address),
    .io_write_value(io_write_value),
    .io_read_value(io_read_value),
    .io_write_en(io_write_en),
    .io_read_en(io_read_en),
    .io_data_size(io_data_size)
    );

   always @(*)
     begin
       case (io_address[1:0])
         2'b01:   io_read_value <= {16'b0, SW};
         2'b10:   io_read_value <= {27'b0, BTN};
         default: io_read_value <= 32'b0;
       endcase
     end

   always @(posedge CLK100MHZ)
      begin
        if (io_write_en && io_address[2])   // bit 2 is set in the write address
          LED <= io_write_value[15:0];
     end
 endmodule
```

Verilog file: processor.v

```
module Risc32(
  input         clk,
  output [31:0] io_address,
  output [31:0] io_write_value,
  input  [31:0] io_read_value,
  output        io_write_en,
  output        io_read_en,
  output [2:0]  io_data_size
  );

  // Control unit signals
  wire [2:0] cu_branch_cond;
  wire       cu_data_read_en;
  wire       cu_data_write_en;
  wire [2:0] cu_data_size;
  wire [1:0] cu_rd_src;
  wire       cu_reg_write_en;
  wire       cu_alu_b_src;
  wire       cu_alu_a_src;
  wire [3:0] cu_alu_op;
  wire [2:0] cu_imm_type;


  // Opcode from datapath to control unit
  wire [6:0] dp_opcode;
  wire [6:0] dp_funct7;
  wire [2:0] dp_funct3;

  // Datapath
  DatapathUnit datapath
  (
    .clk(clk),
    .imm_type(cu_imm_type),
    .branch_cond(cu_branch_cond),
    .data_read_en(cu_data_read_en),
    .data_write_en(cu_data_write_en),
    .alu_b_src(cu_alu_b_src),
    .alu_a_src(cu_alu_a_src),
    .rd_src(cu_rd_src),
    .reg_write_en(cu_reg_write_en),
    .alu_op(cu_alu_op),
    .data_size(cu_data_size),
    .opcode(dp_opcode),
    .funct7(dp_funct7),
    .funct3(dp_funct3),
    .io_address(io_address),
    .io_write_value(io_write_value),
    .io_read_value(io_read_value),
    .io_write_en(io_write_en),
    .io_read_en(io_read_en),
    .io_data_size(io_data_size)
  );

  // control unit
  ControlUnit control
  (
    .opcode(dp_opcode),
    .funct7(dp_funct7),
    .funct3(dp_funct3),
    .imm_type(cu_imm_type),
    .rd_src(cu_rd_src),
    .alu_op(cu_alu_op),
    .branch_cond(cu_branch_cond),
```

```
    .data_read_en(cu_data_read_en),
    .data_write_en(cu_data_write_en),
    .data_size(cu_data_size),
    .alu_b_src(cu_alu_b_src),
    .alu_a_src(cu_alu_a_src),
    .reg_write_en(cu_reg_write_en)
  );
```

Verilog file: control-unit.v

```
module ControlUnit(
  input      [6:0] opcode,
  input      [6:0] funct7,
  input      [2:0] funct3,
  output reg [2:0] imm_type,
  output reg [3:0] alu_op,
  output reg [2:0] branch_cond,
  output reg       data_read_en,
  output reg       data_write_en,
  output reg [2:0] data_size,
  output reg [1:0] rd_src,
  output reg       reg_write_en,
  output reg       alu_b_src,
  output reg       alu_a_src
  );


  always @(*)
  begin
    case(opcode)
    7'b001_0011:   // arithmetic with immediate
                   // only need funct7 when funct 3 is 3'b101 (srli / srai)
                   // could also include for 3'b001 (slli)
      begin
        imm_type       = 3'd1;    // type I
        alu_a_src      = 1'b0;    // rs1
        alu_b_src      = 1'b1;    // imm
        rd_src         = 2'b00;   // data from alu
        reg_write_en   = 1'b1;    // write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = 3'b010;  // no branch
        alu_op         = {funct3 == 3'b101 ? funct7[5] : 0, funct3};   // alu_op has
been encoded to match this
        data_size      = 3'b000;
      end
    7'b011_0011:   // arithmetic with registers
                   // funct7 encoded for all alu_op operations
                   // even if only used for 3'b101 and 3'b000 (add/sub and srl/sra)
      begin
        imm_type       = 3'd0;    // type R
        alu_a_src      = 1'b0;    // rs1
        alu_b_src      = 1'b0;    // rs2
        rd_src         = 2'b00;   // data from alu
        reg_write_en   = 1'b1;    // write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = 3'b010;  // no branch
        alu_op         = {funct7[5], funct3};   // alu_op has been encoded to match this
        data_size      = 3'b000;
      end
    7'b110_0111:   // jalr
      begin
        imm_type       = 3'd1;    // type I
        alu_a_src      = 1'b0;    // rs1
        alu_b_src      = 1'b1;    // ext_imm
        rd_src         = 2'b10;   // pc + 4
        reg_write_en   = 1'b1;    // write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = 3'b011;  // branch always
        alu_op         = 4'b0000; // add
        data_size      = 3'b000;
```

```
      end
    7'b110_1111:   // jal
      begin
        imm_type       = 3'd4;    // type J
        alu_a_src      = 1'b1;    // pc_current
        alu_b_src      = 1'b1;    // ext_imm
        rd_src         = 2'b10;   // pc + 4
        reg_write_en   = 1'b1;    // write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = 3'b011;  // branch always
        alu_op         = 4'b0000; // add
        data_size      = 3'b000;
      end
    7'b010_0011:   // store
      begin
        imm_type      = 3'd2;     // type S
        alu_a_src     = 1'b0;     // rs1
        alu_b_src     = 1'b1;     // ext_imm
        rd_src        = 2'b00;    // data from alu
        reg_write_en  = 1'b0;     // no write to rd
        data_read_en  = 1'b0;     // no read from memory
        data_write_en = 1'b1;     // no write to memory
        branch_cond   = 3'b010;   // no branch
        alu_op        = 4'b0000;  // add
        data_size     = funct3;
      end
    7'b000_0011:   // load
      begin
        imm_type       = 3'd1;     // type I
        alu_a_src      = 1'b0;     // rs1
        alu_b_src      = 1'b1;     // ext_imm
        rd_src         = 2'b01;    // data from memory
        reg_write_en   = 1'b1;     // write to rd
        data_read_en   = 1'b1;     // read from memory
        data_write_en  = 1'b0;     // no write to memory
        branch_cond    = 3'b010;   // no branch
        alu_op         = 4'b0000;  // add
        data_size      = funct3;
      end
    7'b011_0111:   // lui
      begin
        imm_type       = 3'd5;    // type U
        alu_a_src      = 1'b0;    // rs1
        alu_b_src      = 1'b1;    // imm
        rd_src         = 2'b00;   // data from alu
        reg_write_en   = 1'b1;    // write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = 3'b010;  // no branch
        alu_op         = 4'b1001; // pass through alu_b
        data_size      = 3'b000;
      end
    7'b001_0111:   // auipc
      begin
        imm_type       = 3'd5;    // type U
        alu_a_src      = 1'b1;    // pc
        alu_b_src      = 1'b1;    // imm
        rd_src         = 2'b00;   // data from alu
        reg_write_en   = 1'b1;    // write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = 3'b010;  // no branch
        alu_op         = 4'b0000; // add
        data_size      = 3'b000;
      end
```

```
    7'b110_0011:   // branch
      begin
        imm_type       = 3'd3;    // type B
        alu_a_src      = 1'b1;    //  pc_current
        alu_b_src      = 1'b1;    //  ext_imm
        rd_src         = 2'b00;   // data from alu
        reg_write_en   = 1'b0;    // no write to rd
        data_read_en   = 1'b0;    // no read from memory
        data_write_en  = 1'b0;    // no write to memory
        branch_cond    = funct3;  // branch condition is encoded to match funct3
        alu_op         = 4'b0000; // add
        data_size      = 3'b000;
      end
    default:    // ADD
      begin
        imm_type       = 3'd0;    // type R
        alu_a_src      = 1'b0;
        alu_b_src      = 1'b0;
        rd_src         = 2'b00;
        reg_write_en   = 1'b1;
        data_read_en   = 1'b0;
        data_write_en  = 1'b0;
        branch_cond    = 3'b010;   // no branch
        alu_op         = 4'b0000;
        data_size      = 3'b000;
      end
    endcase
```

Verilog file: datapath.v

```
module DatapathUnit(
  input         clk,
  input  [2:0]  imm_type,
  input  [2:0]  branch_cond,
  input         data_read_en,
  input         data_write_en,
  input         reg_write_en,
  input  [2:0]  data_size,
  input  [1:0]  rd_src,
  input         alu_b_src,
  input         alu_a_src,
  input  [3:0]  alu_op,
  output [6:0]  opcode,
  output [6:0]  funct7,
  output [2:0]  funct3,

  output [31:0] io_address,
  output [31:0] io_write_value,
  input  [31:0] io_read_value,
  output        io_write_en,
  output        io_read_en,
  output [2:0]  io_data_size
  );

  reg  [31:0] pc_current;
  wire [31:0] pc_next;
  wire [31:0] pc_plus_4;

  wire        branch_control;

  wire [4:0]  rd;
  wire [31:0] rd_value;
  wire [4:0]  rs1;
  wire [31:0] rs1_value;
  wire [4:0]  rs2;
  wire [31:0] rs2_value;

  wire [31:0] instr;
  reg  [31:0] ext_imm;
  wire [31:0] alu_b_in;
  wire [31:0] alu_a_in;
  wire [31:0] alu_out;

  wire [31:0] data_read_value;

  wire        is_io;
  wire [31:0] mem_address;
  wire [31:0] mem_read_value;
  wire [31:0] mem_write_value;
  wire        mem_read_en;
  wire        mem_write_en;
  wire [2:0]  mem_data_size;

  // Note that io_address is part of the interface
  // Note that io_read_value is part of the interface
  // Note that io_write_value is part of the interface
  // Note that io_read_en is part of the interface
  // Not that  io_write_en are part of the interface

  ////
  //// Program counter
  ////
```

```
  initial begin
    pc_current <= 32'd0;
  end

  // Update to pc_next on rising clock
  // Note - the last bit it set to 0 just in case JALR had set it (the only case where
it could be non-zero)
  always @(posedge clk)
  begin
    pc_current <= {pc_next[31:1], 1'b0};     // last bit set to 0(for JALR)
  end

  assign pc_plus_4 = pc_current + 32'd4;

  ////
  //// Instruction memory
  ////

  InstructionMemory im
  (
    .pc(pc_current),
    .instruction(instr)
  );

  assign opcode = instr[6:0];
  assign funct3 = instr[14:12];
  assign funct7 = instr[31:25];

  assign rs1    = instr[19:15];
  assign rs2    = instr[24:20];
  assign rd     = instr[11:7];

  ////
  //// Registers
  ////

  // Write back the destination register value - either ALU output
  // MEM_READ_MUX

  Mux2_32 mem_read_mux(
    .sel(is_io),
    .out(data_read_value),
    .in0(mem_read_value),
    .in1(io_read_value)
    );

  // RD_VALUE_MUX

  Mux4_32 read_value_mux(
    .sel(rd_src),
    .out(rd_value),
    .in0(alu_out),
    .in1(data_read_value),
    .in2(pc_plus_4),
    .in3(alu_out)           // should never be selected
    );

  // Register allocations

    RegisterUnit reg_file (
    .clk(clk),
    .reg_write_en(reg_write_en),
    .rd(rd),
    .rd_value(rd_value),
    .rs1(rs1),
    .rs1_value(rs1_value),
```

```
    .rs2(rs2),
    .rs2_value(rs2_value)
   );

  ////
  //// ext_imm
  ////

  always @(*)
    case (imm_type)
       3'd1:    ext_imm = { {21{instr[31]}}, instr[30:25], instr[24:21], instr[20] };
// I type (load)
       3'd2:    ext_imm = { {21{instr[31]}}, instr[30:25], instr[11:8],  instr[7]  };
// S type (store)
       3'd3:    ext_imm = { {20{instr[31]}}, instr[7],     instr[30:25], instr[11:8],
1'b0};  // B type - effectively making ext_imm the full offset to branch
       3'd4:    ext_imm = { {12{instr[31]}}, instr[19:12], instr[20],    instr[30:21],
1'b0};  // J type - effectively making ext_imm the full offset to branch
       3'd5:    ext_imm = {     instr[31],   instr[30:20], instr[19:12], 12'b0 };
// U type
       default: ext_imm = { {21{instr[31]}}, instr[30:25], instr[24:21], instr[20] };
// includes R type, does not matter what it is
    endcase


  // ALU_IN_MUX
  // determine input for alu - either the rs2 value or the extended immediate value

   Mux2_32 alu_a_mux (
    .sel(alu_a_src),
    .out(alu_a_in),
    .in0(rs1_value),
    .in1(pc_current)
    );

  Mux2_32 alu_b_mux (
    .sel(alu_b_src),
    .out(alu_b_in),
    .in0(rs2_value),
    .in1(ext_imm)
    );

  // set up the ALU with rs1 and alu_in as inputs - exposes zero flag for branching
  ALU alu_unit (
    .a(alu_a_in),
    .b(alu_b_in),
    .alu_control(alu_op),
    .result(alu_out)
  );

  ////
  //// Branch control
  ////

  // BRANCH_MUX
  // The PC increments by 4
  // If a branch is needed, branch_control is true, and the destination is set a PC +
ext_imm
  // If a jump is needed, the jump destination is calculated
  // Then pc_next set to the correct value - PC + 4, branch destination or jump
destination

  // Branch comparator - do the comparsion based on branch_cond and set branch_control
to 1 if a branch is needed
  BranchComp br_comp (
    .a(rs1_value),
```

```
    .b(rs2_value),
    .branch_cond(branch_cond),
    .branch(branch_control)
    );

  // Then select which is the new pc
  Mux2_32 branch_calc (
    .sel(branch_control),
    .out(pc_next),
    .in0(pc_plus_4),
    .in1(alu_out)
  );

  ////
  //// Address decoder
  ////

  AddressDecoder ad (
    .data_address(alu_out),
    .data_read_en(data_read_en),
    .data_write_en(data_write_en),
    .data_write_value(rs2_value),
    .data_size(data_size),
    .mem_address(mem_address),
    .mem_read_en(mem_read_en),
    .mem_write_en(mem_write_en),
    .mem_write_value(mem_write_value),
    .mem_data_size(mem_data_size),
    .io_address(io_address),
    .io_read_en(io_read_en),
    .io_write_en(io_write_en),
    .io_write_value(io_write_value),
    .io_data_size(io_data_size),
    .is_io(is_io)
   );


  // Data memory

  DataMemory dm
  (
    .clk(clk),
    .mem_access_addr(mem_address),
    .mem_in(mem_write_value),
    .mem_write_en(mem_write_en),
    .mem_read_en(mem_read_en),
    .mem_out(mem_read_value),
    .mem_data_size(mem_data_size)
  );


  // IO
  // io_address, io_read_en and io_write_en set above
  // io_read_value is an input set in the other side of the IO interface
  // so only io_write_value to assign here
  ///assign io_write_value = rs2_value;

endmodule
```

Verilog file: settings.vh

```
`ifndef SETTINGS_H
`define SETTINGS_H

// PROGRAM CHOICE
// There are two base programs, risc_io_prog and test_risc_prog
// risc_io_prog reads the switches and buttons (on a Nexys 4 DDR) and reflects that on
the LEDS
// test_risc_prog runs a basic RISC V program to check the results
// IO_DEMO selects risc_io_prog else test_risc_prog is used
`define IO_DEMO

// MEMORY SETTINGS
// Memory can be either 4 banks each of width one byte, or one array of 32 bit words
// BANKED_MEM selects the 4 banks

//`define BANKED_MEM

// Select the size (in bytes) of data memory and instruction memory
`define data_bytes       128
`define instr_bytes      128

// TESTBENCH SETTINGS

// PROG_BASIC will run the program in instruction memory for 200 steps - works for any
program
// PROG_STEPPED will run each line and check the output (requires test_risc_prog so best
to use with IO_DEMO undefined)
// PROG_INDIV will run specific commands and is not dependent on data memory or
instruction memory being initialised

//`define PROG_BASIC

//`define PROG_STEPPED
//`undef IO_DEMO

`define PROG_INDIV

`endif
```

Verilog file: instruction_memory.v

```
`include "settings.vh"

`define instr_addr_bits       $clog2(`instr_bytes)

module InstructionMemory(
  input  [31:0] pc,
  output [31:0] instruction
  );

  // create the memory
  reg [31:0] memory [`instr_addr_bits - 1:0];

  // memory access will wrap at the limit of the number of words, and is word aligned so
we ignore the lower two bits
  wire [`instr_addr_bits - 1 : 0] rom_addr = pc[`instr_addr_bits + 1 : 2];

  initial
    begin
      `ifdef IO_DEMO
        $readmemb("risc_io_prog.mem", memory);
      `else
        $readmemb("risc_io_prog.mem", memory);
       `endif
    end

  assign instruction = memory[rom_addr];
endmodule
```

Verilog file: data_memory.v

```
`include "settings.vh"

`define data_addr_bits      $clog2(`data_bytes)
`define bank_data_bits      (`data_addr_bits - 2)
`define bank_data_bytes     (`data_bytes >> 2)

module DataMemory(
  input clk,
  // address input, shared by read and write port
  input [31:0]   mem_access_addr,
  input [31:0]   mem_in,
  input          mem_write_en,
  input          mem_read_en,
  input [2:0]    mem_data_size,
  output reg [31:0]  mem_out
  );


`ifdef BANKED_MEM
  // ****************************************
  // ** MEMORY IS 4 BANKS EACH ONE BYTE WIDE **
  // ****************************************

  // four banks, A is msb and D is lsb

  reg [7:0] memA [`bank_data_bytes - 1:0];
  reg [7:0] memB [`bank_data_bytes - 1:0];
  reg [7:0] memC [`bank_data_bytes - 1:0];
  reg [7:0] memD [`bank_data_bytes - 1:0];

  // this needs to split the address range into highest `data_addr_bits-3 and bottom 2
bits
  wire [`data_addr_bits - 3:0] word_addr;   // don't need the bottom two bits - that is
the bank
  wire [1:0] bank_sel;

  // this needs to span the entire address range of [`data_addr_bits-1 : 0]
  // and will also wrap addresses outside of this range into the range
  assign word_addr = mem_access_addr[`data_addr_bits - 1 : 2];
  assign bank_sel  = mem_access_addr[1:0];

  initial
    begin
      `ifdef IO_DEMO
        $readmemb("risc_io_ramA.mem", memA);
        $readmemb("risc_io_ramB.mem", memB);
        $readmemb("risc_io_ramC.mem", memC);
        $readmemb("risc_io_ramD.mem", memD);
      `else
        $readmemb("test_risc_ramA.mem", memA);
        $readmemb("test_risc_ramB.mem", memB);
        $readmemb("test_risc_ramC.mem", memC);
        $readmemb("test_risc_ramD.mem", memD);
      `endif
    end

  always @(posedge clk) begin
    if (mem_write_en)
      begin
        case (mem_data_size)
          3'b000:
            case (bank_sel)
              2'b00:    memD[word_addr] <= mem_in[7:0];
```

```
              2'b01:    memC[word_addr] <= mem_in[7:0];
              2'b10:    memB[word_addr] <= mem_in[7:0];
              2'b11:    memA[word_addr] <= mem_in[7:0];
             endcase
          3'b001:
            case (bank_sel)
            2'b00, 2'b01:   // only allow half-word aligned writes - ignore the lower
bit
              begin
                memD[word_addr] <= mem_in[7:0];
                memC[word_addr] <= mem_in[15:8];
              end
            2'b10, 2'b11:
              begin
                memB[word_addr] <= mem_in[7:0];
                memA[word_addr] <= mem_in[15:8];
              end
            endcase
          default: // really 3'b010
            begin
              memD[word_addr]   <= mem_in[7:0];
              memC[word_addr]   <= mem_in[15:8];
              memB[word_addr]   <= mem_in[23:16];
              memA[word_addr]   <= mem_in[31:24];
            end
        endcase
      end
  end


  always @(*) begin
    if (mem_read_en)
      case (mem_data_size)
        3'b000:          // lb
          case (bank_sel)
            2'b00:       mem_out = { {24{ memD[word_addr][7] }}, memD[word_addr]};
            2'b01:       mem_out = { {24{ memC[word_addr][7] }}, memC[word_addr]};
            2'b10:       mem_out = { {24{ memB[word_addr][7] }}, memB[word_addr]};
            2'b11:       mem_out = { {24{ memA[word_addr][7] }}, memA[word_addr]};
          endcase
        3'b100:           // lbu
          case (bank_sel)
            2'b00:        mem_out = { 24'b0, memD[word_addr]};
            2'b01:        mem_out = { 24'b0, memC[word_addr]};
            2'b10:        mem_out = { 24'b0, memB[word_addr]};
            2'b11:        mem_out = { 24'b0, memA[word_addr]};
          endcase
       3'b001:            // lh
          case (bank_sel)
            2'b00, 2'b01:
              mem_out = { {16{ memC[word_addr][7]}}, memC[word_addr], memD[word_addr]};
            2'b10, 2'b11:
              mem_out = { {16{ memA[word_addr][7]}}, memA[word_addr], memB[word_addr]};
          endcase
        3'b101:
          case (bank_sel) // lhu
            2'b00, 2'b01: mem_out = { 16'b0, memC[word_addr], memD[word_addr]};
            2'b10, 2'b11: mem_out = { 16'b0, memA[word_addr], memB[word_addr]};
          endcase
         default:          // lw      really 3010
          mem_out = {memA[word_addr], memB[word_addr], memC[word_addr],
memD[word_addr]};
      endcase
    else
      mem_out = 32'd0;
  end
```

```
`else
  // ************************************
  // ** MEMORY IS ONE BANK 32 BITS WIDE **
  // ************************************

  reg [31:0] mem [`data_bytes - 1:0];

  // this needs to split the address range into highest `data_addr_bits-3 and bottom 2
bits
  wire [`data_addr_bits - 3:0] word_addr;   // don't need the bottom two bits - that is
the bank
  wire [1:0] byte_sel;

  // this needs to span the entire address range of [`data_addr_bits-1 : 0]
  // and will also wrap addresses outside of this range into the range
  assign word_addr = mem_access_addr[`data_addr_bits - 1 : 2];
  assign byte_sel  = mem_access_addr[1:0];

  initial
    begin
       `ifdef IO_DEMO
          $readmemb("risc_io_mem_word.mem", mem);
       `else
          $readmemb("test_risc_mem_word.mem", mem);
       `endif
    end

  always @(posedge clk) begin
    if (mem_write_en)
      begin
        case (mem_data_size)
          3'b000:
            case (byte_sel)
              2'b00:    mem[word_addr][7:0]   <= mem_in[7:0];
              2'b01:    mem[word_addr][15:8]  <= mem_in[7:0];
              2'b10:    mem[word_addr][23:16] <= mem_in[7:0];
              2'b11:    mem[word_addr][31:24] <= mem_in[7:0];
             endcase
          3'b001:
            case (byte_sel)
            2'b00, 2'b01:   // only allow half-word aligned writes - ignore the lower
bit
              begin
                mem[word_addr][15:0] <= mem_in[15:0];
              end
            2'b10, 2'b11:
              begin
                mem[word_addr][31:16] <= mem_in[15:0];
              end
            endcase
          default: // really 3'b010
            begin
              mem[word_addr] <= mem_in;
            end
        endcase
      end
  end


  always @(*) begin
    if (mem_read_en)
      case (mem_data_size)
        3'b000:          // lb
          case (byte_sel)
            2'b00:
```

```
              mem_out = { {24{ mem[word_addr][7]  }},  mem[word_addr][7:0]};
            2'b01:
              mem_out = { {24{ mem[word_addr][15] }},  mem[word_addr][15:8]};
            2'b10:
              mem_out = { {24{ mem[word_addr][23] }},  mem[word_addr][23:16]};
            2'b11:
              mem_out = { {24{ mem[word_addr][31] }},  mem[word_addr][31:24]};
          endcase
        3'b100:           // lbu
          case (byte_sel)
            2'b00:        mem_out = { 24'b0, mem[word_addr][7:0]};
            2'b01:        mem_out = { 24'b0, mem[word_addr][15:8]};
            2'b10:        mem_out = { 24'b0, mem[word_addr][23:16]};
            2'b11:        mem_out = { 24'b0, mem[word_addr][31:24]};
          endcase
       3'b001:            // lh
          case (byte_sel)
            2'b00, 2'b01: mem_out = { {16{ mem[word_addr][15]}}, mem[word_addr][15:0]};
            2'b10, 2'b11: mem_out = { {16{ mem[word_addr][31]}}, mem[word_addr][31:16]};
          endcase
       3'b101:
          case (byte_sel) // lhu
            2'b00, 2'b01: mem_out = { 16'b0, mem[word_addr][15:0]};
            2'b10, 2'b11: mem_out = { 16'b0, mem[word_addr][31:16]};
          endcase
        default:          // lw      really 3010
                          mem_out = mem[word_addr];
      endcase
    else
      mem_out = 32'd0;
  end

`endif

endmodule
```

Verilog file: address_decoder.v

```
module AddressDecoder(
  input  [31:0] data_address,
  input         data_read_en,
  input         data_write_en,
  input  [31:0] data_write_value,
  input  [2:0]  data_size,

  output [31:0] mem_address,
  output        mem_read_en,
  output        mem_write_en,
  output [31:0] mem_write_value,
  output [2:0]  mem_data_size,

  output [31:0] io_address,
  output        io_read_en,
  output        io_write_en,
  output [31:0] io_write_value,
  output [2:0]  io_data_size,

  output        is_io
  );

  wire        is_mem;

  assign io_address  = data_address;
  assign mem_address = data_address;                  // bit 31 is already 0

  // Is this a memory or IO address?
  assign is_mem = !data_address[31];                  // bit 31 is 0
  assign is_io  =  data_address[31];                  // bit 31 is 1

  // Memory and IO enable read and write flags
  assign mem_read_en =  data_read_en  && is_mem;
  assign mem_write_en = data_write_en && is_mem;
  assign io_read_en =   data_read_en  && is_io;
  assign io_write_en =  data_write_en && is_io;

  assign mem_write_value = data_write_value;
```

Verilog file: br_comp.v

```
module BranchComp(
  input [31:0] a,
  input [31:0] b,
  input [2:0]  branch_cond,
  output reg   branch);

always @(*)
  case (branch_cond)
    3'b000:  branch = (a == b) ? 1 : 0;
    3'b001:  branch = (a != b) ? 1 : 0;
    3'b110:  branch = (a <  b) ? 1 : 0;
    3'b111:  branch = (a >= b) ? 1 : 0;
    3'b100:  branch = ($signed(a) <  $signed(b)) ? 1 : 0;
    3'b101:  branch = ($signed(a) >= $signed(b)) ? 1 : 0;
    3'b010:  branch = 0;    // no branch
    3'b011:  branch = 1;    // always branch
    default: branch = 0;    // no branch
  endcase
endmodule
```

Verilog file: top.v

```
module ALU(
  input  [31:0] a,           // source 1
  input  [31:0] b,           // source 2
  input  [3:0]  alu_control, // function selext

  output reg [31:0] result   //result
  );

  always @(*)
    begin
      case(alu_control)
        4'b0000:  result = a + b;                                  // add
        4'b1000:  result = a - b;                                  // sub
        4'b0001:  result = a << b[4:0];                            // sll
        4'b0010:  result = $signed(a) < $signed(b) ? 32'd1 : 32'd0; // slt
        4'b0011:  result = a < b ? 32'd1 : 32'd0;                  // sltu
        4'b0100:  result = a ^ b;                                  // xor
        4'b0101:  result = a >> b[4:0];                            // srl
        4'b1101:  result = a >>> b[4:0];                           // sra
        4'b0110:  result = a | b;                                  // or
        4'b0111:  result = a & b;                                  // and
        default:  result = a + b;                                  // add
      endcase
    end
endmodule
```

Verilog file: reg_file.v

```
module RegisterUnit(
  input         clk,
  input         reg_write_en,
  input  [4:0]  rd,
  input  [31:0] rd_value,
  input  [4:0]  rs1,
  input  [4:0]  rs2,
  output [31:0] rs1_value,
  output [31:0] rs2_value
  );

  reg    [31:0] reg_array [31:0];

  integer i;
  initial begin
    reg_array[0] = 32'hffff;               // set x0 to be hffff so if anything does
read or write (bug) then we can tell
    for(i = 1; i <= 31; i = i + 1)
      reg_array[i] <= 32'd0;
  end

  always @ (posedge clk ) begin
    if (reg_write_en && (rd != 0)) begin
      reg_array[rd] <= rd_value;
    end
  end

  assign rs1_value = (rs1 == 0) ? 0 : reg_array[rs1];
  assign rs2_value = (rs2 == 0) ? 0 : reg_array[rs2];

endmodule
```

Verilog file: mux.v

```
module Mux2_32(
  input         sel,
  output [31:0] out,
  input  [31:0] in0,    // chosen when sel == 0
  input  [31:0] in1     // chosen when sel == 1
  );

  assign out = sel ? in1 : in0;
endmodule


module Mux4_32(
  input      [1:0]  sel,
  output reg [31:0] out,
  input      [31:0] in0,    // chosen when sel == 0
  input      [31:0] in1,    // chosen when sel == 1
  input      [31:0] in2,    // chosen when sel == 2
  input      [31:0] in3     // chosen when sel == 3
  );

  always @(*)
      begin
        case(sel)
          2'b00:   out = in0;
          2'b01:   out = in1;
          2'b10:   out = in2;
          2'b11:   out = in3;
          default: out = in0;
        endcase
      end
endmodule
```

Testbench

Verilog file: testbench.v

```
`timescale 1ns / 1ps

`include "settings.vh"

`define get_31_24(v)         ((v & 32'hff00_0000) >> 24)
`define get_23_16(v)         ((v & 32'h00ff_0000) >> 16)
`define get_15_8(v)          ((v & 32'h0000_ff00) >> 8)
`define get_7_0(v)            (v & 32'h0000_00ff)

`define make_word(v1, v2, v3,v4)    {v1, v2, v3, v4}
`define make_half(v1, v2)           {v1, v2}

`ifdef BANKED_MEM

`define memA(a)             uut.datapath.dm.memA[a>>2]
`define memB(a)             uut.datapath.dm.memB[a>>2]
`define memC(a)             uut.datapath.dm.memC[a>>2]
`define memD(a)             uut.datapath.dm.memD[a>>2]
`define set_memw(a, v)      `memD(a) <= `get_7_0(v); `memC(a) <= `get_15_8(v); `memB(a)
<= `get_23_16(v); `memA(a) <= `get_31_24(v)
`define show_memw(a)        $display("\tmemw{%1d]:   %8h", a, `make_word(`memA(a),
`memB(a), `memC(a), `memD(a)) )
`define check_memw(a, v, em, sm) if (`make_word(`memA(a), `memB(a), `memC(a), `memD(a))
!= v) $display(em); else $display(sm)

`else

`define memW(a)             uut.datapath.dm.mem[a>>2]
`define set_memw(a, v)     `memW(a) <= v
`define show_memw(a)       $display("\tmemw{%1d]:   %8h", a, `memW(a))
`define check_memw(a, v, em, sm) if (`memW(a) != v) begin $display(em); fails = fails +
1; end else $display(sm)

`endif


`define register(r)         uut.datapath.reg_file.reg_array[r]
`define set_reg(r, v)       uut.datapath.reg_file.reg_array[r] <= v
`define set_pc(a)           uut.datapath.pc_current <= a
`define set_instr(a, v)     uut.datapath.im.memory[a] <= v

`define show_state          $display("PC:  %8h  Instruction: %32b   Opcode: %7b",
uut.datapath.pc_current, uut.datapath.instr, uut.datapath.opcode )

`define tick                clk = ~clk; #5
`define clock_up            clk = 1; #5
`define clock_down          clk = 0; #5
`define run_step            `clock_down; `show_state; `clock_up

`define show_reg(r)         $display("\tx%1d:   %8h", r,
uut.datapath.reg_file.reg_array[r])
`define show_pcnext         $display("\tpc_next: %8h", uut.datapath.pc_next)
`define show_pc             $display("\tpc: %8h", uut.datapath.pc_current)

`define check_pcnext(v, em, sm)  if (uut.datapath.pc_next != v) begin $display(em);
fails = fails + 1; end else $display(sm)
`define check_pc(v, em, sm)      if (uut.datapath.pc_current != v) begin $display(em);
fails = fails + 1; end else $display(sm)
```

```
`define check_reg(r, v, em, sm)  if (uut.datapath.reg_file.reg_array[r] != v) begin
$display(em); fails = fails + 1; end else $display(sm)

`define check_arith_reg(x1, x2, res, cmd, txt)  `set_pc(0); \
                                                `set_instr(0,
32'b0000000_00010_00001_000_00011_0110011 | (cmd << 12)); \
                                                `set_reg(1, x1);    \
                                                `set_reg(2, x2);    \
                                                `run_step;  \
                                                `show_reg(3);   \
                                                if (`register(3) != res) begin
$display("%s %s", txt, "failed"); fails = fails + 1; end else $display("%s %s", txt,
"success")

module test_RISC32;

  // Inputs
  reg clk;

  // Instantiate the Unit Under Test (UUT)
  Risc32 uut (
    .clk(clk)
  );

integer fails = 0;

`ifdef PROG_BASIC
  initial
    begin
      clk <=0;
      #200;  // duration of the simulation
      $finish;
    end

  always
    begin
      #5 clk = ~clk;
    end

`elsif PROG_STEPPED

  initial
    begin
      clk <=0;
    end

  always
    begin
      #10;
      fails = 0;
      $display("RISC-V 32 bit - instruction memory: %4d data memory: %4d", `instr_bytes,
`data_bytes);

      `run_step;
      `show_reg(3);
      `check_reg(3, 32'h0001, "ldw fail", "ldw success");

      `run_step;
      `show_reg(1);
      `check_reg(1, 32'h0002, "ldw fail", "ldw success");

      `run_step;
      `show_reg(2);
      `check_reg(2, 32'h0003, "add fail", "add success");

      `run_step;
```

```
	`show_memw(4);
	`check_memw(4, 32'h0000_0003, "stw fail", "stw success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'hffff_ffff, "sub fail", "sub success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'hffff_fffe, "xori fail", "xori success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'h0000_0004, "sll fail", "sll success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'h0000_0000, "srl fail", "srl success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'h0000_0000, "and fail", "and success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'h0000_0003, "or fail", "or success");

	`run_step;
	`show_reg(2);
	`check_reg(2, 32'h0000_0001, "slt fail", "slt success");

	`run_step;
	`show_reg(3);
	`check_reg(3, 32'h0000_0002, "add fail", "add success");

	`run_step;
	`show_reg(3);
	`check_reg(3, 32'h0000_1000, "lui fail", "lui success");

	`run_step;
	`show_reg(3);
	`check_reg(3, 32'h0000_1034, "auipc fail", "auipc success");

	`run_step;
	`show_pc;
	`check_pc(32'h0000_003c, "beq fail", "beq success");

	`run_step;
	`show_pc;
	`check_pc(32'h0000_0044, "bne fail", "bne success");

	`run_step;
	`show_pc;
	`check_pc(32'h0000_0010, "jalr fail - jump", "jalr success - jump");
	`show_reg(1);
	`check_reg(1, 32'h0000_0048, "jalr fail - store return address", "jalr success -
store return address");

	// can check it like this
	//`clock_down;
	//`show_state;
	//`show_pcnext;
	//`check_pcnext(32'h0000_0010, "jalr fail - jump");
	//`clock_up;
	//`show_reg(1);
	//`check_reg(1, 32'h0000_0048, "jalr fail - store return address");
```

```
      $display("Testbench complete: %d fails", fails);
      #20;
      $finish;
    end

`elsif PROG_INDIV
   initial
     begin
       clk <=0;
     end

   always
     begin
       #10;
       fails = 0;
       $display("RISC-V 32 bit - instruction memory: %4d data memory: %4d",
`instr_bytes,  `data_bytes);

       `check_arith_reg(1, 5, 6, 3'b000, "add");
       `check_arith_reg(32'hffff_ffff, 5, 4, 3'b000, "add");
       `check_arith_reg(3, 4, 7, 3'b110, "or");
       `check_arith_reg(32'h55, 32'h50, 32'h5, 3'b100, "xor");
       `check_arith_reg(32'h55, 32'h55, 32'h0, 3'b100, "xor");
       `check_arith_reg(32'h55, 32'h50, 32'h50, 3'b111, "and");

       // Test lw x1, [0 + x2]
       // Objective - show that a memory word load works
       //                lw rd, rs1(ext_imm)
       //                +++imm++++++_+rs1+_010_++rd+_++op+++
       `set_pc(0);
       `set_instr(0, 32'b000000000000_00010_010_00001_0000011);
       `set_memw(4, 32'h0000_7f7f);
       `set_reg(2, 32'h0004);                            // x2 = 4
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'h0000_7f7f, "lw failure", "lw succcess");

       // Test add x3, x1, x2
       // Objective - show that add of hfff and 1 is h1000
       //                add rd, rs1, rs2
       //                +func7+_+rs2+_+rs1+_fu3_++rd+_+++op++
       `set_pc(0);
       `set_instr(0, 32'b0000000_00010_00001_000_00011_0110011);
       `set_reg(1, 32'h0001);
       `set_reg(2, 32'h0fff);
       `set_reg(3, 32'h2222);
       `run_step;
       `show_reg(3);
       `check_reg(3, 32'h0000_1000, "add failure", "add succcess");

       // Test lui x1, h55555
       // Objective - show that lui loads the upper 20 bits
       //                lui rd, imm
       //                ++++++++++imm+++++++_++rd+_+++op++
       `set_pc(0);
       `set_instr(0, 32'b01010101010101010101_00001_0110111);
       `set_reg(1, 32'h0000_0000);
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'h5555_5000, "lui failure", "lui succcess");

       // Test memory wrap - lw x1, x2(0)   (128) should wrap to 0
       // Objective - show that accessing the word *above* the last word in memory wraps
to 0
       //                lw rd, rs1(ext_imm)
```

```
       //                ++imm++++++_+rs1+_xxx_++rd+_++op+++
       `set_pc(0);
       `set_instr(0, 32'b000000000000_00010_010_00001_0000011);
       `set_memw(0, 32'h7f7f_f7f7);
       `set_memw(124, 32'h8888_8888);
       `set_reg(2, 32'd128);
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'h7f7f_f7f7, "lw memory wrap failure", "lw memory wrap
succcess");

       // Test memory top - lw x1, x2(0)    (124)
       // Objective - show that a read from last word in memory works
       //                lw rd, rs1(ext_imm)
       //                +++imm++++++_+rs1+_010_++rd+_++op+++
       `set_pc(0);
       `set_instr(0, 32'b000000000000_00010_010_00001_0000011);
       `set_memw(0, 32'h7f7f_f7f7);
       `set_memw(124, 32'h8888_8888);
       `set_reg(2, 32'd124);
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'h8888_8888, "lw memory top failure", "lw memory top succcess");

       // Test lh x1, [0 + x2]
       // Objective - show lh does not retrieve the higher two bytes but sets to zeo
       //                lh rd, rs1(ext_imm)
       //                +++imm++++++_+rs1+_001_++rd+_++op+++
       `set_pc(0);
       `set_instr(0, 32'b000000000000_00010_001_00001_0000011);
       `set_memw(0, 32'h7f7f_7f7f);
       `set_reg(2, 32'h0004);
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'h0000_7f7f, "lh failure", "lh succcess");

       // Test lb x1, [0 + x2]
       // Objective - show lb does not sign extend when top bit is 0 (ok, it does - but
sign extends 0)
       //                lb rd, rs1(ext_imm)
       //                +++imm++++++_+rs1+_000_++rd+_++op+++
       `set_pc(0);
       `set_instr(0, 32'b000000000000_00010_000_00001_0000011);
       `set_memw(0, 32'h7f7f_7f7f);
       `set_reg(2, 32'h0004);
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'h0000_007f, "lb (7f) failure", "lb (7f) succcess");

       // Test lb x1, [0 + x2]
       // Objective - show lb does sign extend when top bit is 1
       //                lb rd, rs1(ext_imm)
       //                +++imm++++++_+rs1+_000_++rd+_++op+++
       `set_pc(0);
       `set_instr(0, 32'b000000000000_00010_000_00001_0000011);
       `set_memw(4, 32'h7777_778f);
       `set_reg(2, 32'h0004);
       `run_step;
       `show_reg(1);
       `check_reg(1, 32'hffff_ff8f, "lb (8f) failure", "lb (8f) succcess");

       // Test lhu x1, [0 + x2]
       // Objective - show lhu does not sign extend the half-word on load
       //                lhu rd, rs1(ext_imm)
       //                +++imm++++++_+rs1+_101_++rd+_++op+++
       `set_pc(0);
```

```
        `set_instr(0, 32'b000000000000_00010_101_00001_0000011);
        `set_memw(4, 32'h7777_8f8f);
        `set_reg(2, 32'h0004);
        `run_step;
        `show_reg(1);
        `check_reg(1, 32'h0000_8f8f, "lhu failure", "lhu succcess");

        // Test sh x1, [0 + x2]
        // Objective - show sh only updates lower two bytes in memory
        //                sh rs2, rs1(ext_imm)
        //                +++imm+_+rs2+_+rs1+_001_+imm+_++op+++
        `set_pc(0);
        `set_instr(0, 32'b0000000_00001_00010_001_00000_0100011);
        `set_memw(4, 32'h7777_8f8f);
        `set_reg(2, 32'h0004);
        `set_reg(1, 32'h9999_9999);
        `run_step;
        `show_memw(4);
        `check_memw(4, 32'h7777_9999, "sh failure", "sh succcess");

        $display("Testbench complete: %d fails", fails);
        #20;
        $finish;
      end
`endif

endmodule
```

Another heading

Xxxx

Xxxx

Xxxx

Instruction Set Architecture

instruction	imm	opcode	fun3	fun7	operation
lui rd, imm20	U	0x37			rd = imm20 << 12
auipc rd, imm20	U	0x17			rd = pc + (imm20 <<12)
addi rd, rs1, imm12	I	0x13	000		rd = rs1 + se(imm12)
slli rd, rs1, imm12	I	0x13	001	0x00	rd = rs1 << imm12[4:0]
slti rd, rs1, imm12	I	0x13	010		rd = rs1 < signed se(imm12) ? 1 : 0
sltiu rd, rs1, imm12	I	0x13	011		rd = rs1 < unsign se(imm12) ? 1 : 0
xori rd, rs1, imm12	I	0x13	100		rd = rs1 ^ se(imm12)
srli rd, rs1, imm12	I	0x13	101	0x00	rd = rs1 >> imm12[4:0]
srai rd, rs1, imm12	I	0x13	101	0x20	rd = rs1 >>> imm12[4:0]
ori rd, rs1, imm12	I	0x13	110		rd = rs1 | se(imm12)
andi rd, rs1, imm12	I	0x13	111		rd = rs1 & se(imm12)
add rd, rs1, rs2	R	0x33	000	0x00	rd = rs1 + rs2
sub rd, rs1, rs2	R	0x33	000	0x20	rd = rs1 - rs2
sll rd, rs1, rs2	R	0x33	001		rd = rs1 << rs2[4:0]
slt rd, rs1, rs2	R	0x33	010		rd = rs1 < signed rs2 ? 1 : 0
sltu rd, rs1, rs2	R	0x33	011		rd = rs1 < unsign rs2 ? 1 : 0
xor rd, rs1, rs2	R	0x33	100		rd = rs1 ^ rs2
srl rd, rs1, rs2	R	0x33	101	0x00	rd = rs1 >> rs2[4:0]
sra rd, rs1, rs2	R	0x33	101	0x20	rd = rs1 >>> rs2[4:0]
or rd, rs1, rs2	R	0x33	110		rd = rs1 | rs2
and rd, rs1, rs2	R	0x33	111		rd = rs1 & rs2
lb rd, imm12(rs1)	I	0x03	000		rd = se(mem[rs1 + se(imm12)][7:0])
lh rd, imm12(rs1)	I	0x03	001		rd = se(mem[rs1 + se(imm12)][15:0])
lw rd, imm12(rs1)	I	0x03	010		rd = mem[rs1 + se(imm12)][31:0]
lbu rd, imm12(rs1)	I	0x03	100		rd = ze(mem[rs1 + se(imm12)][7:0])
lhu rd, imm12(rs1)	I	0x03	101		rd = ze(mem[rs1 + se(imm12)][15:0])
sb rs2, imm12(rs1)	S	0x23	000		mem[rs1 + se(imm12)][7:0] = rs2[7:0]
sh rs2, imm12(rs1)	S	0x23	001		mem[rs1 + se(imm12)][15:0] = rs2[15:0]
sw rs2, imm12(rs1)	S	0x23	010		mem[rs1 + se(imm12)][31:0] = rs2
jal rd, targ20	J	0x6f			rd =pc+4; pc += se(targ20 <<1)
jalr rd, imm12(rs1)	I	0x67	000		rd =pc+4; pc = (rs1 + se(imm12)) & ~1
beq rs1, rs2, targ12	B	0x63	000		if (rs1 == rs2) pc += se(targ12<<1)
bne rs1, rs2, targ12	B	0x63	001		if (rs1 != rs2) pc += se(targ12<<1)
blt rs1, rs2, targ12	B	0x63	100		if (rs1 < signed rs2) pc += se(targ12<<1)
bge rs1, rs2, targ12	B	0x63	101		if (rs1 >= signed rs2) pc += se(targ12<<1)
bltu rs1, rs2, targ12	B	0x63	110		if (rs1 < unsign rs2) pc += se(targ12<<1)
bgeu rs1, rs2, targ12	B	0x63	111		if (rs1 >= unsign rs2) pc += se(targ12<<1)

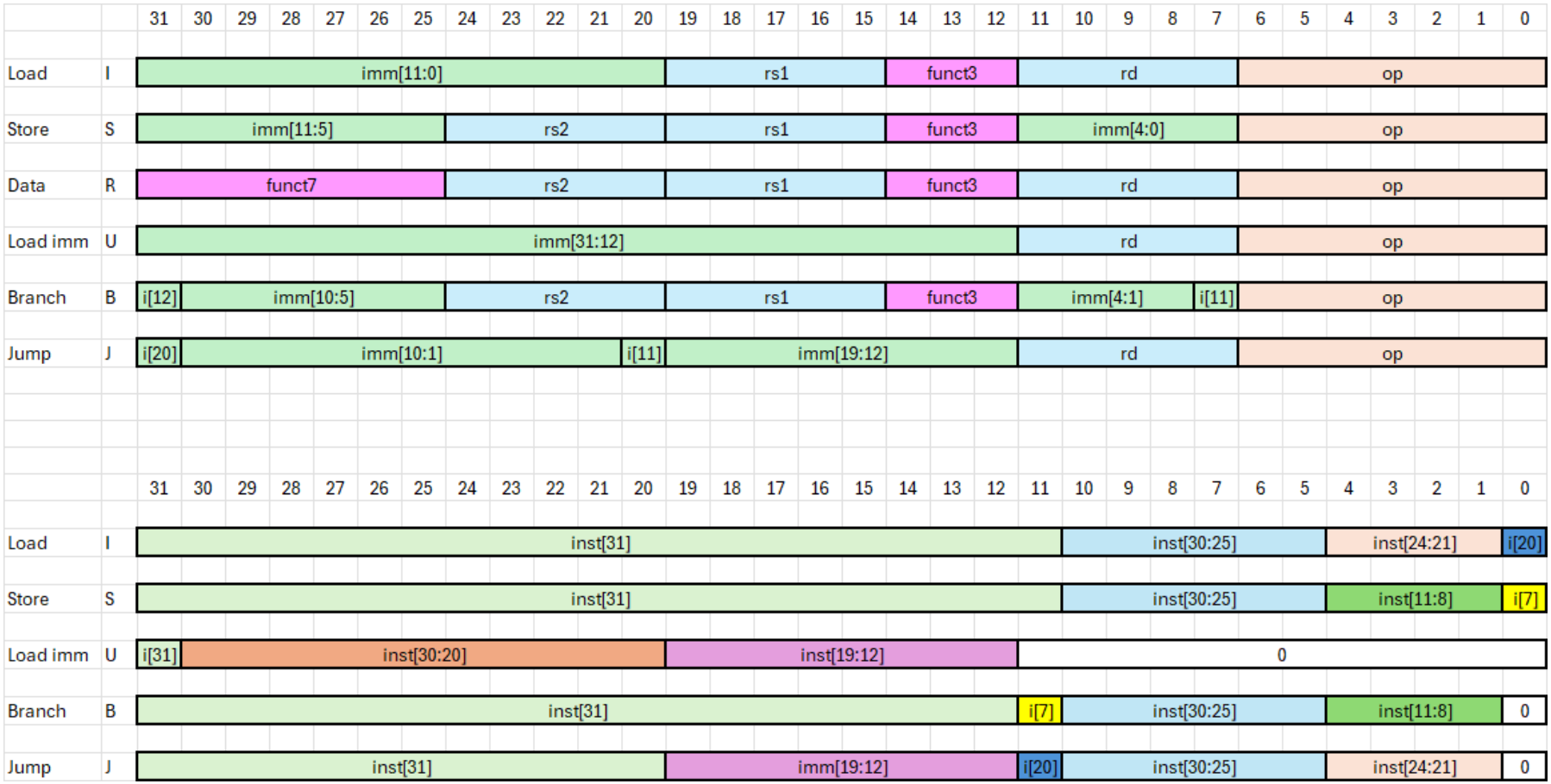
In this implementation all half-word accesses must be half-word aligned, this is enforced by removing the lowest bit of the memory address

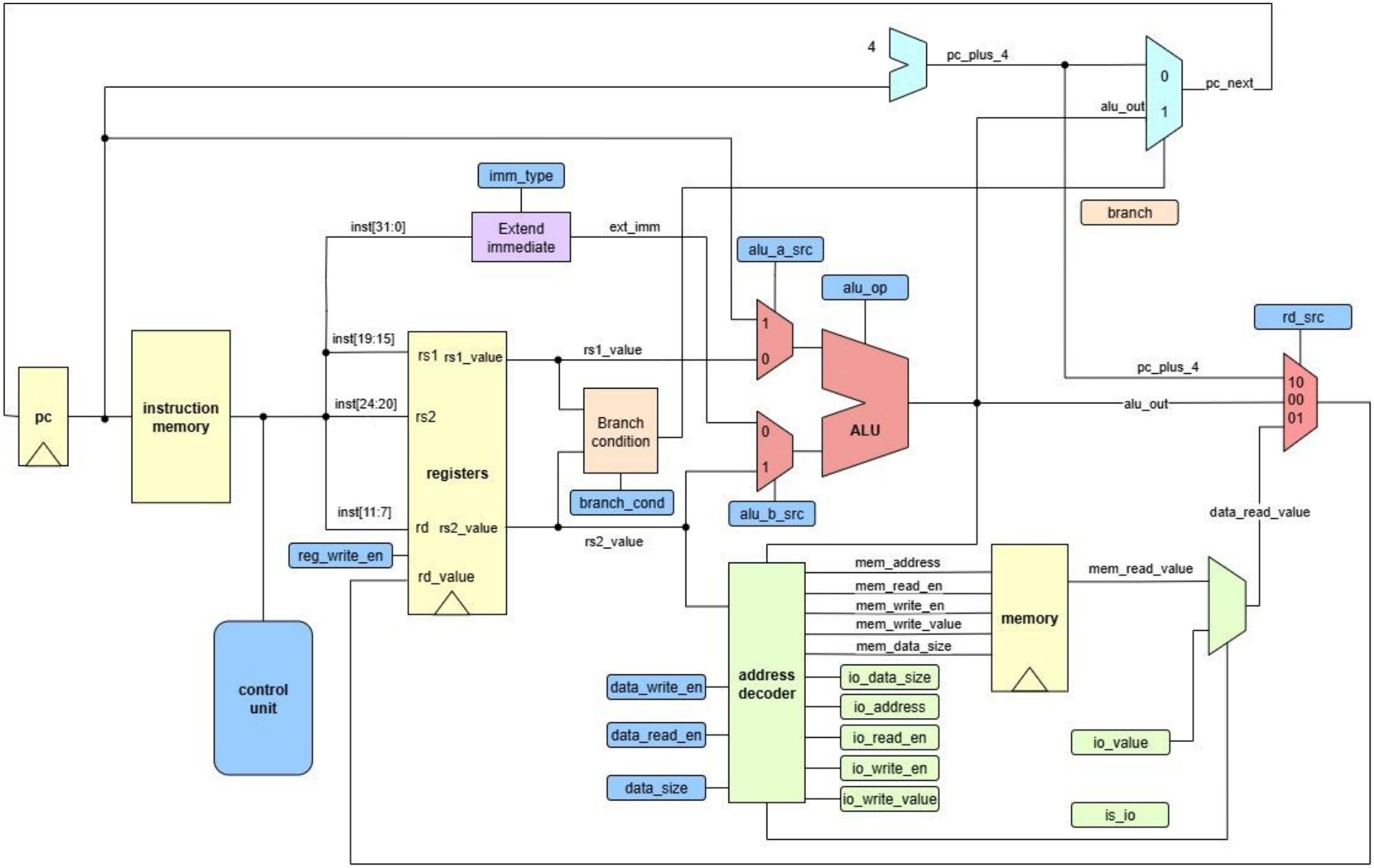
Also all word accesses must be word aligned, this is enforced by removing the lowest two bits of the memory address

instruction	imm	opcode	fun3	fun7	operation
lh rd, imm12(rs1)	I	0x03	001		rd = se((mem[rs1 + se(imm12)) & ~1][15:0])
lhu rd, imm12(rs1)	I	0x03	101		rd = ze((mem[rs1 + se(imm12)) & ~1][15:0])
sh rs2, imm12(rs1)	S	0x23	001		mem[(rs1 + se(imm12)) & ~1][15:0] = rs2[15:0]
lw rd, imm12(rs1)	I	0x03	010		rd = mem[(rs1 + se(imm12)) & ~3][31:0]
sw rs2, imm12(rs1)	S	0x23	010		mem[(rs1 + se(imm12)) & ~3][31:0] = rs2

instruction	imm	opcode	fun3	fun7	encoding
lui rd, imm20	U	0x37			iiii iiii iiii iiii iiii dddd d011 0111
auipc rd, imm20	U	0x17			iiii iiii iiii iiii iiii dddd d001 0111
addi rd, rs1, imm12	I	0x13	000		iiii iiii iiii ssss s000 dddd d001 0011
slli rd, rs1, imm12	I	0x13	001	0x00	0000 000i iiii ssss s001 dddd d001 0011
slti rd, rs1, imm12	I	0x13	010		iiii iiii iiii ssss s010 dddd d001 0011
sltiu rd, rs1, imm12	I	0x13	011		iiii iiii iiii ssss s011 dddd d001 0011
xori rd, rs1, imm12	I	0x13	100		iiii iiii iiii ssss s100 dddd d001 0011
srli rd, rs1, imm12	I	0x13	101	0x00	0000 000i iiii ssss s101 dddd d001 0011
srai rd, rs1, imm12	I	0x13	101	0x20	0100 000i iiii ssss s101 dddd d001 0011
ori rd, rs1, imm12	I	0x13	110		iiii iiii iiii ssss s110 dddd d001 0011
andi rd, rs1, imm12	I	0x13	111		iiii iiii iiii ssss s111 dddd d001 0011
add rd, rs1, rs2	R	0x33	000	0x00	0000 000t tttt ssss s000 dddd d011 0011
sub rd, rs1, rs2	R	0x33	000	0x20	0100 000t tttt ssss s000 dddd d011 0011
sll rd, rs1, rs2	R	0x33	001		0000 000t tttt ssss s001 dddd d011 0011
slt rd, rs1, rs2	R	0x33	010		0000 000t tttt ssss s010 dddd d011 0011
sltu rd, rs1, rs2	R	0x33	011		0000 000t tttt ssss s011 dddd d011 0011
xor rd, rs1, rs2	R	0x33	100		0000 000t tttt ssss s100 dddd d011 0011
srl rd, rs1, rs2	R	0x33	101	0x00	0000 000t tttt ssss s101 dddd d011 0011
sra rd, rs1, rs2	R	0x33	101	0x20	0100 000t tttt ssss s101 dddd d011 0011
or rd, rs1, rs2	R	0x33	110		0000 000t tttt ssss s110 dddd d011 0011
and rd, rs1, rs2	R	0x33	111		0000 000t tttt ssss s111 dddd d011 0011
lb rd, imm12(rs1)	I	0x03	000		iiii iiii iiii ssss s000 dddd d000 0011
lh rd, imm12(rs1)	I	0x03	001		iiii iiii iiii ssss s001 dddd d000 0011
lw rd, imm12(rs1)	I	0x03	010		iiii iiii iiii ssss s010 dddd d000 0011
lbu rd, imm12(rs1)	I	0x03	100		iiii iiii iiii ssss s100 dddd d000 0011
lhu rd, imm12(rs1)	I	0x03	101		iiii iiii iiii ssss s101 dddd d000 0011
sb rs2, imm12(rs1)	S	0x23	000		iiii iiit tttt ssss s000 dddd d010 0011
sh rs2, imm12(rs1)	S	0x23	001		iiii iiit tttt ssss s001 dddd d010 0011
sw rs2, imm12(rs1)	S	0x23	010		iiii iiit tttt ssss s010 dddd d010 0011
jal rd, targ20	J	0x6f			iiii iiii iiii iiii iiii dddd d110 0111
jalr rd, imm12(rs1)	I	0x67	000		iiii iiii iiii ssss s000 dddd d110 0111
beq rs1, rs2, targ12	B	0x63	000		iiii iiit tttt ssss s000 dddd d110 0011
bne rs1, rs2, targ12	B	0x63	001		iiii iiit tttt ssss s001 dddd d110 0011
blt rs1, rs2, targ12	B	0x63	100		iiii iiit tttt ssss s100 dddd d110 0011
bge rs1, rs2, targ12	B	0x63	101		iiii iiit tttt ssss s101 dddd d110 0011
bltu rs1, rs2, targ12	B	0x63	110		iiii iiit tttt ssss s110 dddd d110 0011
bgeu rs1, rs2, targ12	B	0x63	111		iiii iiit tttt ssss s111 dddd d110 0011

instruction	imm	opcode	fun3	fun7	operation	encoding
lui rd, imm20	U	0x37			rd = imm20 << 12	iiii iiii iiii iiii iiii dddd d011 0111
auipc rd, imm20	U	0x17			rd = pc + (imm20 <<12)	iiii iiii iiii iiii iiii dddd d001 0111
addi rd, rs1, imm12	I	0x13	000		rd = rs1 + se(imm12)	iiii iiii iiii ssss s000 dddd d001 0011
slli rd, rs1, imm12	I	0x13	001	0x00	rd = rs1 << imm12[4:0]	0000 000i iiii ssss s001 dddd d001 0011
slti rd, rs1, imm12	I	0x13	010		rd = rs1 < signed se(imm12) ? 1 : 0	iiii iiii iiii ssss s010 dddd d001 0011
sltiu rd, rs1, imm12	I	0x13	011		rd = rs1 < unsign se(imm12) ? 1 : 0	iiii iiii iiii ssss s011 dddd d001 0011
xori rd, rs1, imm12	I	0x13	100		rd = rs1 ^ se(imm12)	iiii iiii iiii ssss s100 dddd d001 0011
srli rd, rs1, imm12	I	0x13	101	0x00	rd = rs1 >> imm12[4:0]	0000 000i iiii ssss s101 dddd d001 0011
srai rd, rs1, imm12	I	0x13	101	0x20	rd = rs1 >>> imm12[4:0]	0100 000i iiii ssss s101 dddd d001 0011
ori rd, rs1, imm12	I	0x13	110		rd = rs1 | se(imm12)	iiii iiii iiii ssss s110 dddd d001 0011
andi rd, rs1, imm12	I	0x13	111		rd = rs1 & se(imm12)	iiii iiii iiii ssss s111 dddd d001 0011
add rd, rs1, rs2	R	0x33	000	0x00	rd = rs1 + rs2	0000 000t tttt ssss s000 dddd d011 0011
sub rd, rs1, rs2	R	0x33	000	0x20	rd = rs1 - rs2	0100 000t tttt ssss s000 dddd d011 0011
sll rd, rs1, rs2	R	0x33	001		rd = rs1 << rs2[4:0]	0000 000t tttt ssss s001 dddd d011 0011
slt rd, rs1, rs2	R	0x33	010		rd = rs1 < signed rs2 ? 1 : 0	0000 000t tttt ssss s010 dddd d011 0011
sltu rd, rs1, rs2	R	0x33	011		rd = rs1 < unsign rs2 ? 1 : 0	0000 000t tttt ssss s011 dddd d011 0011
xor rd, rs1, rs2	R	0x33	100		rd = rs1 ^ rs2	0000 000t tttt ssss s100 dddd d011 0011
srl rd, rs1, rs2	R	0x33	101	0x00	rd = rs1 >> rs2[4:0]	0000 000t tttt ssss s101 dddd d011 0011
sra rd, rs1, rs2	R	0x33	101	0x20	rd = rs1 >>> rs2[4:0]	0100 000t tttt ssss s101 dddd d011 0011
or rd, rs1, rs2	R	0x33	110		rd = rs1 | rs2	0000 000t tttt ssss s110 dddd d011 0011
and rd, rs1, rs2	R	0x33	111		rd = rs1 & rs2	0000 000t tttt ssss s111 dddd d011 0011
lb rd, imm12(rs1)	I	0x03	000		rd = se(mem[rs1 + se(imm12)][7:0])	iiii iiii iiii ssss s000 dddd d000 0011
lh rd, imm12(rs1)	I	0x03	001		rd = se(mem[rs1 + se(imm12)][15:0])	iiii iiii iiii ssss s001 dddd d000 0011
lw rd, imm12(rs1)	I	0x03	010		rd = mem[rs1 + se(imm12)][31:0]	iiii iiii iiii ssss s010 dddd d000 0011
lbu rd, imm12(rs1)	I	0x03	100		rd = ze(mem[rs1 + se(imm12)][7:0])	iiii iiii iiii ssss s100 dddd d000 0011
lhu rd, imm12(rs1)	I	0x03	101		rd = ze(mem[rs1 + se(imm12)][15:0])	iiii iiii iiii ssss s101 dddd d000 0011
sb rs2, imm12(rs1)	S	0x23	000		mem[rs1 + se(imm12)][7:0] = rs2[7:0]	iiii iiit tttt ssss s000 dddd d010 0011
sh rs2, imm12(rs1)	S	0x23	001		mem[rs1 + se(imm12)][15:0] = rs2[15:0]	iiii iiit tttt ssss s001 dddd d010 0011
sw rs2, imm12(rs1)	S	0x23	010		mem[rs1 + se(imm12)][31:0] = rs2	iiii iiit tttt ssss s010 dddd d010 0011
jal rd, targ20	J	0x6f			rd =pc+4; pc += se(targ20 <<1)	iiii iiii iiii iiii iiii dddd d110 0111
jalr rd, imm12(rs1)	I	0x67	000		rd =pc+4; pc = (rs1 + se(imm12)) & ~0x1	iiii iiii iiii ssss s000 dddd d110 0111
beq rs1, rs2, targ12	B	0x63	000		if (rs1 == rs2) pc += se(targ12<<1)	iiii iiit tttt ssss s000 dddd d110 0011
bne rs1, rs2, targ12	B	0x63	001		if (rs1 != rs2) pc += se(targ12<<1)	iiii iiit tttt ssss s001 dddd d110 0011
blt rs1, rs2, targ12	B	0x63	100		if (rs1 < signed rs2) pc += se(targ12<<1)	iiii iiit tttt ssss s100 dddd d110 0011
bge rs1, rs2, targ12	B	0x63	101		if (rs1 >= signed rs2) pc += se(targ12<<1)	iiii iiit tttt ssss s101 dddd d110 0011
bltu rs1, rs2, targ12	B	0x63	110		if (rs1 < unsign rs2) pc += se(targ12<<1)	iiii iiit tttt ssss s110 dddd d110 0011
bgeu rs1, rs2, targ12	B	0x63	111		if (rs1 >= unsign rs2) pc += se(targ12<<1)	iiii iiit tttt ssss s111 dddd d110 0011

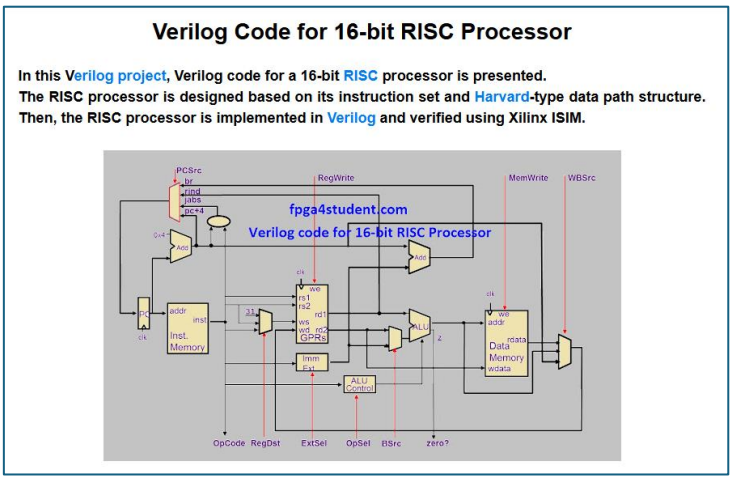

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Load I imm[11:0] rs1 funct3 rd op
Store S imm[11:5] rs2 rs1 funct3 imm[4:0] op
Data R funct7 rs2 rs1 funct3 rd op
Load imm U imm[31:12] rd op
Branch B i[12] imm[10:5] rs2 rs1 funct3 imm[4:1] i[11] op
Jump J i[20] imm[10:1] i[11] imm[19:12] rd op
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Load I inst[31] inst[30:25] inst[24:21] i[20]
Store S inst[31] inst[30:25] inst[11:8] i[7]
Load imm U i[31] inst[30:20] inst[19:12] 0
Branch B inst[31] i[7] inst[30:25] inst[11:8] 0
Jump J inst[31] imm[19:12] i[20] inst[30:25] inst[24:21] 0


4
pc_plus_4
0
1
pc_next
alu_out
imm_type
Extend immediate
inst[31:0]
ext_imm
branch
alu_a_src
alu_op
rd_src
inst[19:15]
rs1
rs1_value
rs1_value
pc
instruction memory
inst[24:20]
rs2
Branch condition
ALU
pc_plus_4
10
00
01
alu_out
registers
branch_cond
alu_b_src
inst[11:7]
rd
rs2_value
rs2_value
data_read_value
reg_write_en
rd_value
mem_address
mem_read_en
mem_write_en
mem_write_value
mem_data_size
memory
mem_read_value
control unit
address decoder
data_write_en
data_read_en
data_size
io_data_size
io_address
io_read_en
io_write_en
io_write_value
io_value
is_io

References

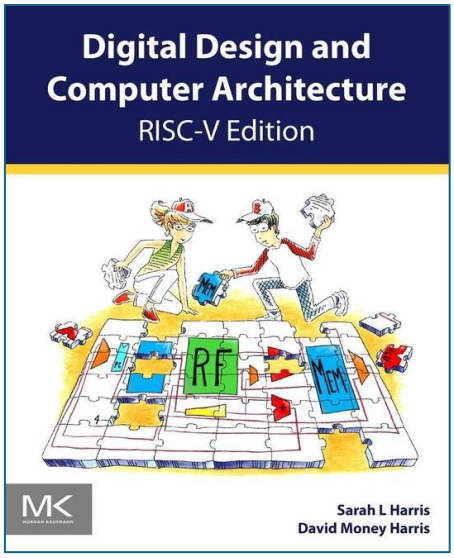
fpga4student RISC processor design

https://www.fpga4student.com/2017/04/verilog-code-for-16-bit-risc-processor.html



Digital Design and Computer Architecture – RISC V Edition

ISBN: 978-0-12-820064-3



ISA documentation

https://www.five-embeddev.com/riscv-user-isa-manual/latest-adoc/