Building a Single-cycle RISC V processor in Verilog

Contents

Introduction

Key features

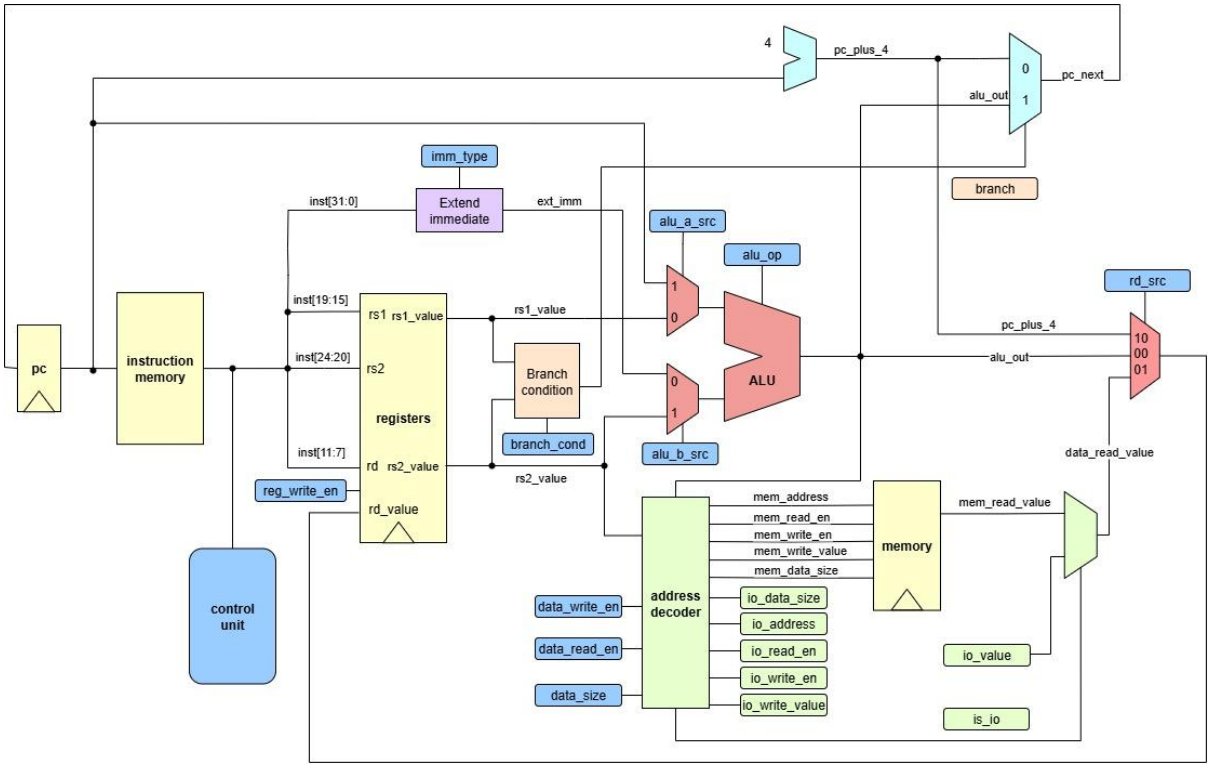
The ALU will process two imputs – a and b. Input a can be either the rs1 value or the program counter. Input b can be either the rs2 vlue or the extended immediate value.

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



Control Unit signals

Signal definitions

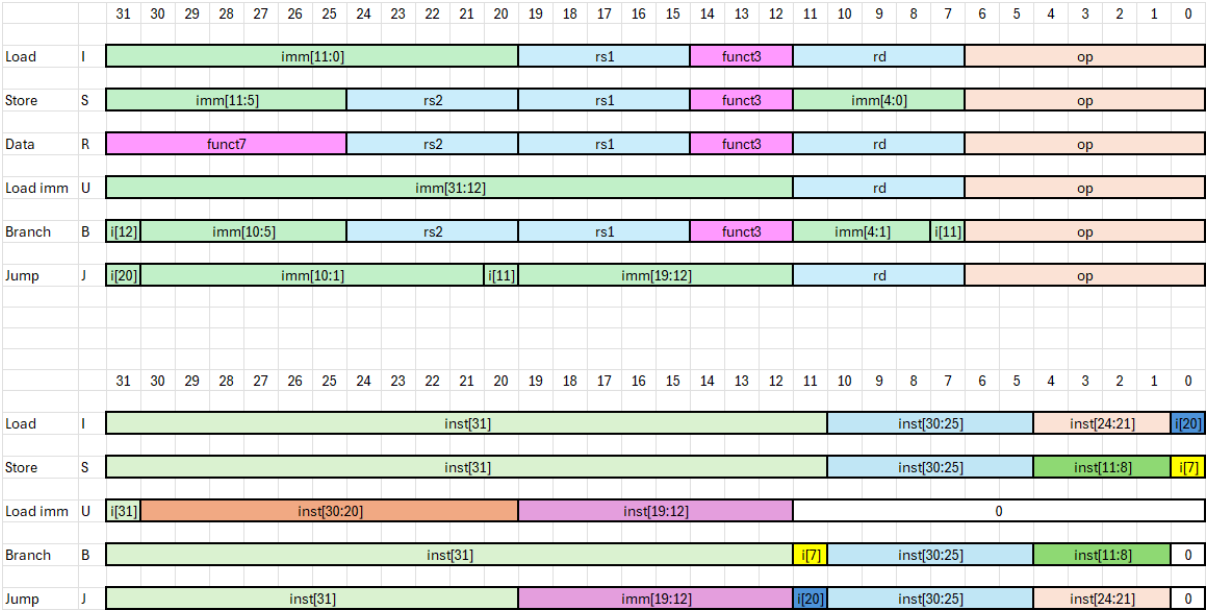
Signal	Definition
imm_type	Type of immediate (B, I, J, S, U, R)
alu_a_src	a input to ALU (pc, rs1)
alu_b_src	b input tp ALU (imm, rs2)
alu_op	Function for ALU to execute
rd_src	Source for rd register (data, pc + 4, alu output)
reg_write_en	Write to rd register
data_read_en	Enable read from data bus
data_write_en	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

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	

Instruction memory

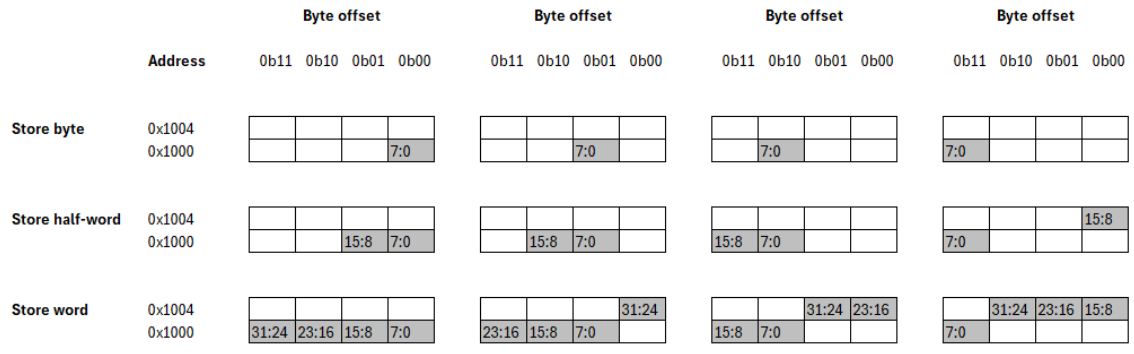
RISC V has byte-addressable instruction memory but only allows aligned access – word or half-word. As this implementation only allows 32 bit instructions, the memory is an array of 32 bit words.

Data memory

RISC V has byte-addressable memory and allows unaligned data access. This adds complexity to the implementation of data memory because it must be capable to spanning word boundaries.

Memory has been defined as four banks of 8 bits. A 32 bit word is created from the four bytes, starting at the base address.

This is more complex than enforcing word or half-word aligned memory, which could have been created as an array of 32 bit words.



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 has been encoded to match this
        alu_op         = {funct3 == 3'b101 ? funct7[5] : 0, funct3};
        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 has been encoded to match this
        alu_op         = {funct7[5], funct3};
        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
    // last bit set to 0(for JALR)
    pc_current <= {pc_next[31:1], 1'b0};
  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)
     // I type (load)
     3'd1:  ext_imm = { {21{instr[31]}}, instr[30:25], instr[24:21], instr[20] };
     // S type (store)
     3'd2:  ext_imm = { {21{instr[31]}}, instr[30:25], instr[11:8],  instr[7]  };
     // B type - effectively making ext_imm the full offset to branch
     3'd3:  ext_imm = { {20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
     // J 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};
     // U type
     3'd5:  ext_imm = {     instr[31],   instr[30:20], instr[19:12], 12'b0 };
     // includes R type, does not matter what it is
     default:
            ext_imm = { {21{instr[31]}}, instr[30:25], instr[24:21], instr[20] };
  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 to 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;
  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
            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
  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
              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
    // set x0 to be hffff so if anything does read or write (bug) then we can tell
    reg_array[0] = 32'hffff;
    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) \
                            begin $display(em); fails = fails + 1; end \
                            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
```

Assembler and disassembler

Arguments for lui, auipc, branch, jal, jalr

Argument and immediate size

Many of the RISC V instructions include a non-register value – referred to a *value*

```
lui x1, 0x12
addi x1, x2, 23
beq x1, x2, 4
```

To calculate the range of that can be stored in x bits, use 1 << x

For example, 12 bits can store 1<<12 = 4096 values.

For unsigned, the minimum value is 0 and the maximum value is (1<<12) -1 = 4095.

If they are signed, then the minimum value is –(1 <<11) = -2048 and the maxiumum is (1<<11) -1 = 2047

Number of bit	Signed / Unsigned	Min	Max	Min	Max
12	Signed	-2048	2047	-0x0800	0x07ff
13	Signed	-4096	4095	-0x1000	0x0fff
13 (final bit zero)	Signed	-4096	4094	-0x1000	0x0ffe
20	Unsigned	0	1048575	0	0xf_ffff
21	Signed	-1048576	1048575	-0x10_0000	0xf_ffff
21 (final bit zero)	Signed	-1048576	1048574	-0x10_0000	0xf_fffe

lui

Format	Description	Immediate
`lui rd, val20`	`rd = val20 << 12`	`imm20 = val20`

The value for lui is a 20 bit unsigned number.

This is stored in the instruction immediate as a 20 bit number.

This is shifted left 20 bits and stored in the register.

```
lui x1, 0x12345
```
will store 0x1234_5000 in x1

Valid instructions are shown in the table below

```
lui x1, 0x00000
lui x1, 0xfffff

lui x1, 0
lui x1, 1048575
```

auipc

Format	Description	Immediate
`auipc rd,  val20`	`rd = pc + (val20 << 12)`	`imm20 = val20`

The value for auipc is a 20 bit unsigned number.

This is stored in the instruction immediate as a 20 bit number.

This is shifted left 20 bits, added to the pc and stored in the register.

```
auipc x1, 0x12345
```
will store pc + 0x1234_5000 in x1

Valid instructions are shown in the table below

```
auipc x1, 0x00000
auipc x1, 0xfffff

auipc x1, 0
auipc x1, 1048575
```

branches

All branches are specified by the byte offset from the *current* instruction.

```
beq x1, x2, 4
```
will just continue with the next instruction (pc+4)

```
beq x1, x2, 0
```
will loop to the current instruction

Format	Description	Immediate
`beq   rs1, rs2, val13`	`if (rs1 == rs2) pc += se(val13)`	`imm12 = val13 >> 1`

The value for a branch is a 13 bit signed number.

All instruction addresses are 16-bit aligned, so the lowest address bit *must be zero*.

Therefore the offset can be stored in the instruction immediate as a 12 bit number.

The range of the argument is from -4096 to +4094, so 8192.

The range of the instruction immediate is -2048 to +2047, so 4096.

Valid instructions are shown in the table below

```
beq x1, x2, -0x1000
beq x1, x2,  0
beq x1, x2,  0x0ffe

beq x1, x2, -4096
beq x1, x2,  0
beq x1, x2,  4094
```

jal

Format	Description	Immediate
`jal rd, val21`	`rd =pc+4; pc += se(val21)`	`imm20 = val21 >> 1`

The value for jal is a 21 bit signed offset of the instruction to jump to relative to the pc.

All instruction addresses are 16-bit aligned, so the lowest address bit *must be zero*.

Therefore the offset can be stored in the instruction immediate as a 20 bit number.

```
jal x1, 1234
```

Valid instructions are shown in the table below

```
jal x1, -0x10_0000
jal x1,  0x0f_fffe
jal x1, -1048576
jal x1,  0
jal x1,  1048574
```

jalr

Format	Description	Immediate
`jalr rd, val12(rs1)`	`rd =pc+4; pc = (rs1 + se(val12)) & ~0x1`	`imm12 = val12`

The value for jalr is a 12 bit byte signed offset of the instruction to jump to relative to the the source register.

The is stored in the instruction immediate as a 12 bit number.

```
 jalr x1, 123(x2)
```

Valid instructions are shown in the table below

```
jalr x1, -0x0800(x2)
jalr x1,  0x07fe(x2)
jalr x1, -2048(x2)
jalr x1,  0(x2)
jalr x1,  2046(x2)
```

Technically the two instructions below are valid and the lowest bit will be set to zero after the source register is added.

```
jalr x1, 0x07ff(x2)
jalr x1, 2047(x2)
```

Python file: ass.py

```
# My RISC simple assembler
# Creates machine code from My RISC assembler

# Writes output to the screen and two files
# Screen and .lrs file have line numbers
# The .rsc file doesn't have line numbers

# Assemble a My RISC machine code file

# Format is:
# {line_number} {label} {code} {comment}
# Where any item may be present or missing
# Comments start //
# Anything after a left brace will be removed  - {

# Example:

# label_here:
#      ld rd, rs1(2)
#      jmp label       // comment at end of line
#      // standalone comment
# end: jmp end

# NOTE - any jump or banch offsets / immediate values are the actual
#        number of bytes, rather than the number of instructions


import re


# Check for leading negative sign - no need to check for plus sign
# as that is removed as whitespace

def is_uint(s):
    return s.isnumeric()

def is_int(s):
    return s.isnumeric() or (s[0] == "-" and s[1:].isnumeric())


def int_to_twos_complement(val, bits):
    if val < 0:
        return val + (1 << bits)
    else:
        return val


def get_bits(value, start, end):
    mask = (1 << (end - start + 1)) -1
    return (value >> start) & mask

def tokenise(txt) :
    # remove () and [] and + that might surround / precede an integer
    # make the left bracket and + into a space - for split()
    # and remove the right brackets
    # so:
    # ld x0, x2(0) => ld x0 x2 0
    # ld x0, x2+10 => ld x0 x2 10

    txt = txt.replace("[", " ")
    txt = txt.replace("]","")
    txt = txt.replace("(", " ")
    txt = txt.replace(")","")
```

```
    txt = txt.replace("+", " ")
    txt = txt.lower()

    # remove anything in braces {} - only allowed once in any line
    l_brace = txt.find("{")
    r_brace = txt.find("}")
    if r_brace > -1 and l_brace > -1:
        txt = txt[ : l_brace] + txt [r_brace + 1: ]

    # process comments - anything from // to end of the line
    comment = ""
    comment_location = txt.find("//")
    if comment_location != -1:
        comment = txt[comment_location : ]
        txt = txt[ : comment_location].strip()

    sp = re.split("[,\s]+", txt)

    label = None
    cmd   = None
    regA  = None
    regB  = None
    regC  = None
    value = None
    jmp_label = None

    for ind, c in enumerate(sp):
        if len(c) > 0:             # don't process an empty cell
            if ind == 0 and is_int(c):
                # ignore line numbers
                pass
            elif c[-1] == ":":
                label = c[:-1]
            elif cmd == None:
                cmd = c
            elif regA == None and c[0] == "x" and is_uint(c[1:]):
                regA = int(c[1])
            elif regB == None and c[0] == "x" and is_uint(c[1:]):
                regB = int(c[1])
            elif regC == None and c[0] == "x" and is_uint(c[1:]):
                regC = int(c[1])
            elif value == None  and is_int(c):
                value = int(c)
            else:
                jmp_label = c

    return label, cmd, regA, regB, regC, value, jmp_label, comment

# Instruction formats

# lw     rd,  rs1(imm)
# sw     rs2, rs1(imm)
# add    rd,  rs1, rs2
# beq    rs1, rs2, imm
# jal    rd,  imm
# jalr   rd,  rs1(imm)
# lui    rd,  imm
# auipc  rd,  imm


# Machine code

# add    -func7- --rs2 --rs1 fu3 --rd- 01100 11
# addi   -func7- --rs2 --rs1 fu3 --rd- 00100 11
# lw     -----imm----- --rs1 dw- --rd- 00000 11
# sw     --imm-- --rs2 --rs1 dw- -imm- 01000 11
```

```
# beq    --imm-- --rs2 --rs1 fu3 -imm- 11000 11
# jalr   -----imm----- --rs1 000 --rd- 11001 11
# jal      ----------imm-------  --rd- 11011 11
# lui      ----------imm-------  --rd- 01101 11
# auipc    ----------imm-------  --rd- 00101 11

# As written

# cmd regA, regB, regC, imm
# add r1,   r2,   r3
# ld  r1,   r2          (-12)

# add    -func7- --rgC --rgB fu3 --rgA 01100 11
# addi   -func7- --rgC --rgB fu3 --rgA 00100 11
# lw     -----imm----- --rgB dw- --rgA 00000 11
# sw     --imm-- --rgA --rgB dw- -imm- 01000 11
# beq    --imm-- --rgB --rgA fu3 -imm- 11000 11
# jalr   -----imm----- --rgB 000 --rdA 11001 11
# jal      ----------imm-------  --rgA 11011 11
# lui      ----------imm-------  --rgA 01101 11
# auipc    ----------imm-------  --rgA 00101 11

def assemble(code):
    result = []
    label_to_line = {}
    line_to_label = {}

    old_arith_cmds = {"add": 2, "sub": 3, "lsl": 5,
                  "lsr": 6, "and": 7, "or" : 8,
                  "slt": 9}

    arith_r_cmds = {"add": 0, "sub": 8, "sll": 1,
                    "slt": 2, "sltu": 3, "xor" : 4,
                    "srl": 5, "sra": 13, "or": 6 , "and": 7}

    arith_i_cmds = {"addi": 0,  "slti": 2, "sltiu": 3,
                    "xori": 4, "ori": 6 , "andi": 7}

    arith_i_shift_cmds = {"slli": 1, "srli": 5, "srai": 13}

    load_cmds =    {"lb": 0, "lh": 1, "lw": 2, "lbu": 4, "lhu": 5}
    store_cmds =   {"sb": 0, "sh": 1, "sw": 2}
    branch_cmds =  {"beq": 0, "bne": 1, "blt": 4, "bge": 5, "bltu":6, "bgeu": 7}

    # Pass 1 - for labels
    line_number = 0
    for line in code:
        (label, cmd, regA, regB, regC, value, jmp_label, comment) = tokenise(line)
        if label:
            label_to_line[label] = line_number
            line_to_label[line_number] = label
        if cmd:
            line_number += 4

    # Pass 2 - assembly
    line_number = 0
    for line in code:
        (label, cmd, regA, regB, regC, value, jmp_label, comment) = tokenise(line)
        code = ""

        # Replace a jump label with the value
        if jmp_label:
            value = label_to_line[jmp_label] - line_number

        if cmd:
            if   cmd in load_cmds:
```

```
            imm = int_to_twos_complement(value, 12)
            imm_11_0 = get_bits(imm, 0,  11)
            dw = load_cmds[cmd]
            code = (f"   {imm_11_0:012b}_{regB:05b}_{dw:03b}_" +
                    f"{regA:05b}_00000_11")
        elif cmd in store_cmds:
            imm = int_to_twos_complement(value, 12)
            imm_11_5 = get_bits(imm, 5,  11)
            imm_4_0  = get_bits(imm, 0,  4)
            dw = store_cmds[cmd]
            code = (f"  {imm_11_5:07b}_{regA:05b}_{regB:05b}_" +
                    f"{dw:03b}_{imm_4_0:05b}_01000_11")
        elif cmd in arith_r_cmds:
            val = arith_r_cmds[cmd]
            func7 = (val >> 3) & 1
            func3 = val & 7
            code = (f"  0{func7:01b}00000_{regC:05b}_{regB:05b}_" +
                    f"{func3:03b}_{regA:05b}_01100_11")
        elif cmd in arith_i_cmds:
            imm = int_to_twos_complement(value, 12)
            imm_11_0 = get_bits(imm, 0,  11)
            val = arith_i_cmds[cmd]
            func7 = (val >> 3) & 1
            func3 = val & 7
            code = (f"   {imm_11_0:012b}_{regB:05b}_{func3:03b}_" +
                    f"{regA:05b}_00100_11")
        elif cmd in arith_i_shift_cmds:
            imm = int_to_twos_complement(value, 5)
            imm_0_4 = get_bits(imm, 0,  4)
            val = arith_i_shift_cmds[cmd]
            func7 = (val >> 3) & 1
            func3 = val & 7
            code = (f"  0{func7:01b}00000_{imm_0_4:05b}_{regB:05b}_" +
                    f"{func3:03b}_{regA:05b}_00100_11")
        elif cmd in branch_cmds:
            #         imm[12]    imm[10:5]      imm[4:1]   imm[11]
            # get 13 bit twos complement as we drop the final bit
            imm = int_to_twos_complement(value, 13)
            imm_12   = get_bits(imm, 12, 12)
            imm_10_5 = get_bits(imm, 5,  10)
            imm_4_1  = get_bits(imm, 1,  4)
            imm_11   = get_bits(imm, 11, 11)
            func3 = branch_cmds[cmd]
            code = (f"{imm_12:01b}_{imm_10_5:06b}_{regB:05b}_" +
                    f"{regA:05b}_{func3:03b}_{imm_4_1:04b}_" +
                    f"{imm_11:01b}_11000_11")
        elif cmd == "jal":
            #         imm[20]    imm[10:1]      imm[11]    imm[19:12]
            # get 21 bit twos complement as we drop the final bit
            imm = int_to_twos_complement(value, 21)
            imm_20    = get_bits(imm, 20, 20)
            imm_10_1  = get_bits(imm, 1,  10)
            imm_11    = get_bits(imm, 11, 11)
            imm_19_12 = get_bits(imm, 12, 19)
            code = (f"  {imm_20:01b}_{imm_10_1:010b}_{imm_11:01b}_" +
                    f"{imm_19_12:08b}_{regA:05b}_11011_11")
        elif cmd == "jalr":
            imm = int_to_twos_complement(value, 12)
            imm_11_0 = get_bits(imm, 0,  11)
            code = (f"   {imm_11_0:012b}_{regB:05b}_000_{regA:05b}_"+
                    f"11001_11")
        elif cmd == "auipc":
            imm = int_to_twos_complement(value, 20)
            imm_19_0 = get_bits(imm, 0,  19)
            code = f"     {imm_19_0:020b}_{regA:05b}_00101_11"
        elif cmd == "lui":
```

```
                imm = int_to_twos_complement(value, 20)
                imm_19_0 = get_bits(value, 0,  19)
                code = f"     {imm_19_0:020b}_{regA:05b}_01101_11"
            else:
                code = "ERROR"

        result.append((line_number, label, code, comment))
        if code:
            line_number += 4

    return result
#################################################################


import sys, os
filename = ""

if len(sys.argv) == 2:
    filename = sys.argv[1]
    splitname = re.split("\.", filename)
    outname1 = splitname[0] + ".mc"
    outname2 = splitname[0] + ".lmc"
else:
    filename = "risc_test.rscin"
    outname1 = None
    outname2 = None

# read input file  into code_clean
f = open(filename, mode='r')
code = f.readlines()
f.close()
code_clean =[line.strip() for line in code]

# assemble
fmc = assemble(code_clean)

# Print out the result
for l in code_clean:
    print(l)

print()

# helper to reduce number of checks on files in the main loop
def printfile(txt, f):
    if f != None:
        print(txt, file = f)

# Print machine code to two files, one with line numbers, one without
if outname1:
    f1 = open(outname1, mode='w')
    f2 = open(outname2, mode='w')
else:
    f1 = None
    f2 = None

for line_no, label, code, comment in fmc:
    if label:
        s = f"// [{label:s}:{line_no:d}]"
        print(s)
        printfile(s, f1)
        printfile(s, f2)

    if comment and not code:
        s = f"       {comment:s}"
        print(s)
        printfile(s, f1)
```

```
        printfile(s, f2)

    if code:
        s1 = f"        {code:22s} {comment:s}"
        s2 = f"{line_no:<4d}    {code:22s} {comment:s}"
        print(s2)
        printfile(s1, f1)
        printfile(s2, f2)

if outname1:
    f1.close()
    f2.close()
```

Python file: dis.py

```
# My RISC simple disassembler
# Creates assembler from My RISC machine code

# Disassemble a My RISC machine code file

# Writes output to the screen and two files
# Screen and .lmc file have line numbers
# The .mc file doesn't have line numbers

# Format is:

# {label_in_comment}
# {line_number} {binary} {comment}

# {comment} starts with // and run to the end of the line
# {label_in_comment} is formatted: // [label:line_number]
#                    - should be the only thing on that line
# {label} is any set of characters (except a :)
#                    - but A-Z a-z 0-9 and underscore are preferred
# {line_number} is a simple positive integer
# {binary} is a 16 bit binary number with underscores permitted

# Example:

# // comment
# // [start:0]
# // [start:0]
#
#           000000000000_00010_000_00011_00000_11
#           000000000100_00010_000_00001_00000_11  // with comment
# 12      000000_0_00001_00011_000_00010_00010_11
# 12       0000000_00010_00001_000_00000_00001_11  // with comment

# Instruction formats

# lw     rd,  imm(rs1)
# sw     rs2, imm(rs1)
# add    rd,  rs1, rs2
# beq    rs1, rs2, imm
# jal    rd,  imm
# jalr   rd,  imm(rs1)
# lui    rd,  imm
# auipc  rd,  imm


# Machine code

# add    -func7- --rs2 --rs1 fu3 --rd- 01100 11
# addi   -func7- --rs2 --rs1 fu3 --rd- 00100 11
# lw     -----imm----- --rs1 dw- --rd- 00000 11
# sw     --imm-- --rs2 --rs1 dw- -imm- 01000 11
# beq    --imm-- --rs2 --rs1 fu3 -imm- 11000 11
# jalr   -----imm----- --rs1 000 --rd- 11001 11
# jal      ----------imm-------  --rd- 11011 11
# lui      ----------imm-------  --rd- 01101 11
# auipc    ----------imm-------  --rd- 00101 11

# As written

# cmd regA, regB, regC, imm
# add r1,   r2,   r3
# ld  r1,   r2          (-12)
```

```
# add    -func7- --rgC --rgB fu3 --rgA 01100 11
# addi   -func7- --rgC --rgB fu3 --rgA 00100 11
# lw     -----imm----- --rgB dw- --rgA 00000 11
# sw     --imm-- --rgA --rgB dw- -imm- 01000 11
# beq    --imm-- --rgB --rgA fu3 -imm- 11000 11
# jalr   -----imm----- --rgB 000 --rdA 11001 11
# jal      ----------imm-------  --rgA 11011 11
# lui      ----------imm-------  --rgA 01101 11
# auipc    ----------imm-------  --rgA 00101 11

def is_int(s):
    return s.isnumeric() or (s[0] == "-" and s[1:].isnumeric())

def twos_complement_to_int(val, bits):
    limit = 1 << bits
    max_positive_int = (1 << (bits - 1)) - 1

    if val > max_positive_int:
        return val - limit
    else:
        return val

def get_bits(value, start, end):
    mask = (1 << (end - start + 1)) -1
    return (value >> start) & mask

def sign_extend(value, bits):
    if value == 1:
        return (1 << bits) - 1
    else:
        return 0

def disassemble(code):
    full_assembly = []
    opcodes_old = ["ld ", "st ", "add", "sub", "inv", "lsl", "lsr",
                   "and", "or ", "slt", "",    "beq", "bne", "jmp",
                   "lui", "lli"]

    line_number = 0
    label_names = {}

    # Pass 1 - add all labels to the label table
    for line in code:
        find_label = line.find("// [")
        find_colon = line.find(":")
        find_end   = line.find("]")
        if find_label > -1 and find_colon > -1 and find_end > -1:
            label = line[find_label + 4: find_colon]
            value = line[find_colon + 1: find_end]
            if is_int(value):
                label_names[int(value)] = label


    # Pass 2 - disassemble
    for line in code:
        comment = ""
        assembly = ""

        # remove anything between braces {}
        brace_start = line.find("{")
        brace_end  = line.find("}")
        if brace_start > -1 and brace_end > -1:
            line = line[:brace_start] + line[brace_end + 1:]

        # split on comment //
        line = line.strip()
```

```
        comment_location = line.find("//")
        if comment_location > -1:
            comment = line[comment_location :]
            line = line[:comment_location]

        # check for a label in a comment (at start of comment) // [xxx:yy]
        if comment_location == 0:
            label_start = comment.find("[")
            label_end = comment.find(":")
            if label_start > -1 and label_end > -1:
                output = comment[label_start + 1 : label_end + 1]
            else:
                output = comment
            full_assembly.append((None, output))

        # otherwise process the line for disassembly
        elif line != "":
            value = int(line, 2)

            opcode = get_bits(value, 0, 6)
            rd     = get_bits(value, 7, 11)
            func3  = get_bits(value, 12, 14)
            rs1    = get_bits(value, 15, 19)
            rs2    = get_bits(value, 20, 24)
            func7  = get_bits(value, 25, 31)

            #opcode = opcode >> 2 # drop bottom two bits as not needed

            sign      = get_bits(value, 31, 31)   # 1 bit
            val_30_20 = get_bits(value, 20, 30)   # 11 bits
            val_30_25 = get_bits(value, 25, 30)   # 6 bits
            val_24_21 = get_bits(value, 21, 24)   # 5 bits
            val_20    = get_bits(value, 20, 20)   # 1 bit
            val_19_12 = get_bits(value, 12, 19)   # 8 bits
            val_11_8  = get_bits(value, 8, 11)    # 4 bits
            val_7     = get_bits(value, 7, 7)     # 1 bit

            imm_I = ( (sign_extend(sign, 21) << 11) +  val_30_20 )
            imm_S = ( (sign_extend(sign, 21) << 11) + (val_30_25 << 5)  +
                      (val_11_8 << 1)  + val_7 )
            imm_B = ( (sign_extend(sign, 20) << 12) + (val_7 << 11)     +
                      (val_30_25 << 5) + (val_11_8 << 1) )
            imm_U = ( (sign << 31) + (val_30_20 << 20) +
                      (val_19_12 << 12) )
            imm_J = ( (sign_extend(sign, 12) << 20) + (val_19_12 << 12) +
                      (val_20 << 11) + (val_30_25 << 5) + (val_24_21 << 1) )

            signed_imm_I  = twos_complement_to_int(imm_I, 32)
            signed_imm_S  = twos_complement_to_int(imm_S, 32)
            signed_imm_B  = twos_complement_to_int(imm_B, 32)
            signed_imm_U  = twos_complement_to_int(imm_U, 32)
            signed_imm_J  = twos_complement_to_int(imm_J, 32)

            # and process all the optional registers and value


            if   opcode == 0b01100_11:
                # arithmetic r type
                arith_r_cmds = ["add", "sll", "slt", "sltu", "xor",
                                "srl", "or",  "and", "sub",  "",
                                "",    "",    "",    "sra"]
                ind = (func7 >> 2) + func3
                assembly = arith_r_cmds[ind] + "\t"
                assembly += f"x{rd:d}, x{rs1:d}, x{rs2:d}"
            elif opcode == 0b00100_11:
                # arithmetic i type
```

```
                arith_i_cmds = ["addi", "slli", "slti", "sltiu", "xori",
                                "srli", "ori",  "andi", "",      "",
                                "",     "",     "",     "srai"]
                # check for the three shift instructions with 5 bit immediate
                if (func3 == 1 or func3 == 5):
                    ind = (func7 >> 2) + func3
                    assembly = arith_i_cmds[ind] + "\t"
                    assembly += f"x{rd:d}, x{rs1:d}, {imm_I & 31:d}"
                else:
                    assembly = arith_i_cmds[func3] + "\t"
                    assembly += f"x{rd:d}, x{rs1:d}, {signed_imm_I:d}"
            elif opcode == 0b00000_11:
                # load
                load_cmds = ["lb", "lh", "lw", "", "lbu", "lhu"]
                assembly = load_cmds[func3] + "\t"
                assembly += f"x{rd:d}, {signed_imm_I:d}(x{rs1:d})"
            elif opcode == 0b01000_11:
                # store
                store_cmds = ["sb", "sh", "sw"]
                assembly = store_cmds[func3] + "\t"
                assembly += f"x{rs2:d}, {signed_imm_S:d}(x{rs1:d})"
            elif opcode == 0b11000_11:
                # branch
                branch_cmds = ["beq", "bne", "", "", "blt", "bge", "bltu", "bgeu"]
                assembly = branch_cmds[func3] + "\t"
                assembly += f"x{rs1:d}, x{rs2:d}, {signed_imm_B:d}"
            elif opcode == 0b11001_11:
                # jalr
                assembly = "jalr" + "\t"
                assembly += f"x{rd:d}, {signed_imm_I:d}(x{rs1:d})"
            elif opcode == 0b11011_11:
                # jal
                assembly = "jal" + "\t"
                assembly += f"x{rd:d}, {signed_imm_J:d}"

            elif opcode == 0b01101_11:
                # lui
                assembly = "lui" + "\t"
                shift_imm = signed_imm_U >> 12
                assembly += f"x{rd:d}, {shift_imm:d}"
            elif opcode == 0b00101_11:
                # auipc
                assembly = "auipc" + "\t"
                shift_imm = signed_imm_U >> 12
                assembly += f"x{rd:d}, {shift_imm:d}"


            # create the output with the assembly plus a comment
            # (which could be empty)
            output = f"{assembly:25s}" + comment

            full_assembly.append((line_number, output))
            line_number += 4

    return full_assembly

#############################################################

import sys, os

if len(sys.argv) > 1:
    filename = sys.argv[1]
    splitname = filename.split(".")
    outname1 = splitname[0] +".rsc"
    outname2 = splitname[0] +".lrs"
else:
```

```
    #filename = "test1.mc"
    filename = "risc_test.mc"
    outname1 = None

f = open(filename, mode='r')
code = f.readlines()
f.close()
code_clean =[line.strip() for line in code]


ass = disassemble(code_clean)

# helper to reduce number of checks on files in the main loop
def printfile(txt, f):
    if f != None:
        print(txt, file = f)

# Print out the result

if outname1:
    f1 = open(outname1, mode='w')
    f2 = open(outname2, mode='w')
else:
    f1 = None
    f2 = None

for line_no, line in ass:
    if line_no != None:
        s1 = f"          {line:s}"
        s2 = f"{line_no:<4d}      {line:s}"
        print(s2)
        printfile(s1, f1)
        printfile(s2, f2)
    else:
        print(line)
        printfile(line, f1)
        printfile(line, f2)

if outname1:
    f1.close()
    f2.close()
```

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)

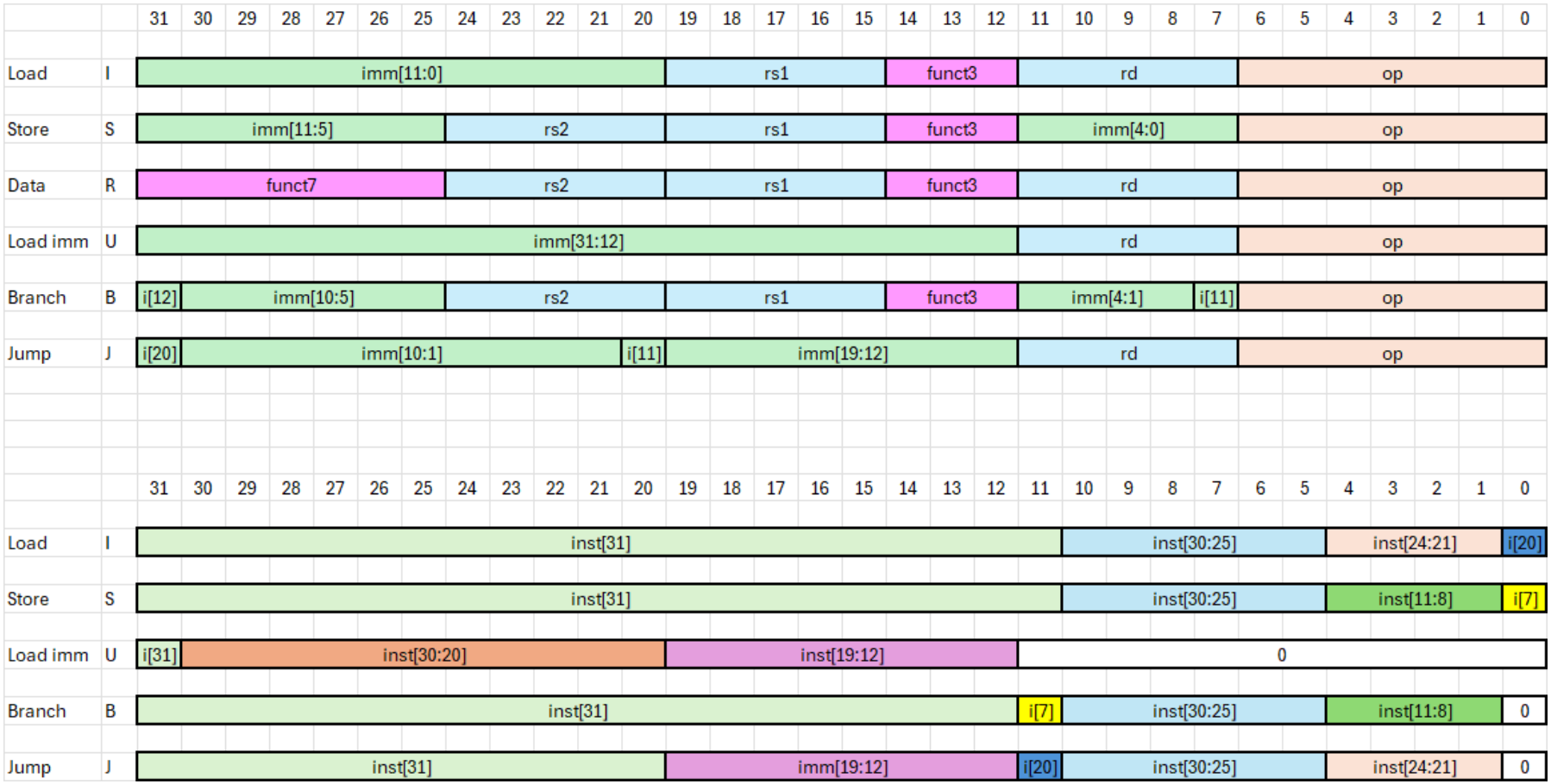
In this implementation all half-word accesses must be half-word aligned, this is enforced by removing the lowest bit of the memory address

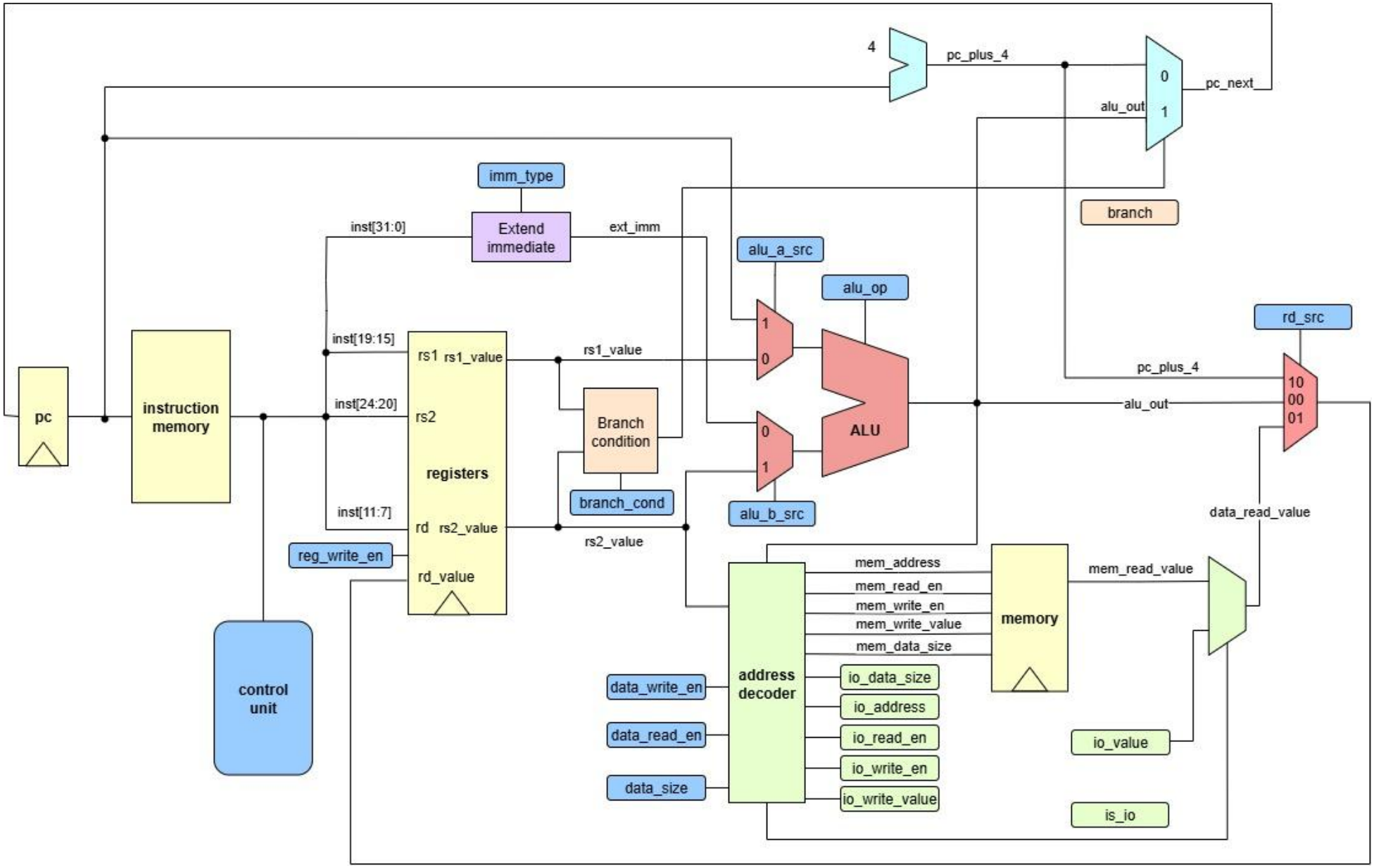
Also all word accesses must be word aligned, this is enforced by removing the lowest two bits of the memory address

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

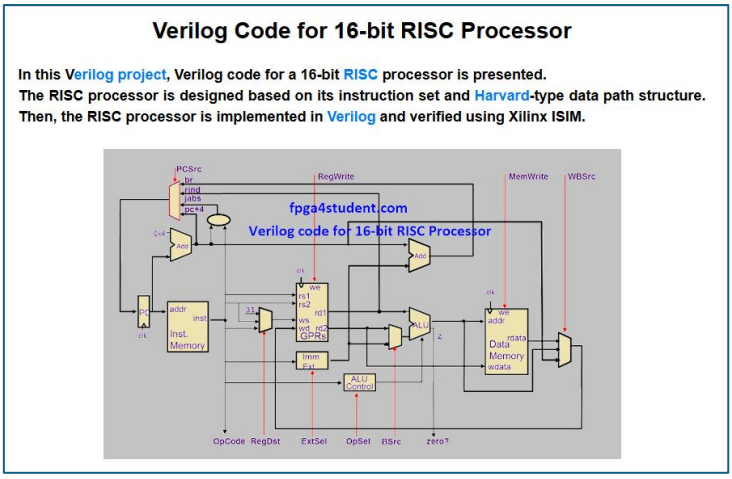

4
pc_plus_4
0
1
pc_next
alu_out
imm_type
Extend immediate
inst[31:0]
ext_imm
branch
alu_a_src
alu_op
rd_src
inst[19:15]
rs1
rs1_value
rs1_value
pc
instruction memory
inst[24:20]
rs2
Branch condition
ALU
pc_plus_4
10
00
01
alu_out
registers
branch_cond
alu_b_src
inst[11:7]
rd
rs2_value
rs2_value
data_read_value
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

		Byte offset				Byte offset				Byte offset				Byte offset			
	Address	0b11	0b10	0b01	0b00	0b11	0b10	0b01	0b00	0b11	0b10	0b01	0b00	0b11	0b10	0b01	0b00
Store byte	0x1004																
	0x1000				7:0			7:0			7:0			7:0			
Store half-word	0x1004																15:8
	0x1000			15:8	7:0		15:8	7:0		15:8	7:0			7:0			
Store word	0x1004								31:24			31:24	23:16		31:24	23:16	15:8
	0x1000	31:24	23:16	15:8	7:0	23:16	15:8	7:0		15:8	7:0			7:0			

References

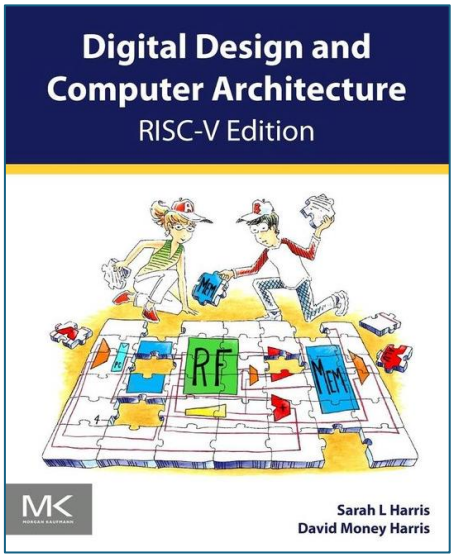
fpga4student RISC processor design

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/