```
/*****************************************************************************************/
/* RVCore_Base, RISC-V RV32I Baseline Version v2024-10-22a          ArchLab, Science Tokyo */
/*****************************************************************************************/
`default_nettype none

`define VERIFY                  // define this to generate verify.txt
`define MEM_TXT   "sample1.txt"  // file name of memory initialization file
`define MEM_SIZE  4096           // 4KB instructio memory and 4KB data memory
`define START_PC  0              // initial PC value
`define UART_CNT  50             // UART wait count, 50MHz / 500 = 1Mbaud
`define D_I_TYPE  0
`define D_R_TYPE  1
`define D_S_TYPE  2
`define D_B_TYPE  3
`define D_J_TYPE  4
`define D_U_TYPE  5
`define D_JA__IS  6
`define D_LD__IS  7
`define D_MUL_IS  8
`define D_DIV_IS  9

/*****************************************************************************************/
module m_rvcore ( ///// RVCore Simple Version
    input  wire        w_clk, w_rst,
    input  wire [31:0] I_IN, D_IN,
    output wire [31:0] I_ADDR, D_ADDR,
    output wire [31:0] D_OUT,
    output wire  [3:0] D_WE
);
    reg r_rst = 1;
    always @(posedge w_clk) r_rst <= w_rst;

    reg [31:0] r_pc = `START_PC;
    always @(posedge w_clk) r_pc <= (r_rst) ? `START_PC : (w_b_rslt) ? w_tkn_pc : r_pc+4;
    assign I_ADDR = r_pc;

    wire [31:0] w_ir = I_IN;
    wire [31:0] w_rrs1, w_rrs2_t, w_imm_t, w_rslt;
    wire [ 4:0] w_rs1, w_rs2, w_rd;
    wire        w_op_im;
    wire [9:0]  w_itype;
    wire [10:0] w_alu_c;
    wire [6:0]  w_bru_c;
    wire        w_b_rslt;
    wire        w_ja   = w_itype[6];
    wire [4:0]  w_op   = w_ir[6:2];
    wire [31:0] w_imm  = (w_ja) ? r_pc+4 : (w_op==5'b00101) ? r_pc+w_imm_t : w_imm_t;
    wire [31:0] w_rrs2 = (w_op_im) ? w_imm : w_rrs2_t;

    m_decode decode1 (w_ir, w_rd, w_rs1, w_rs2, w_op_im, w_itype, w_alu_c, w_bru_c, w_imm_t);

    m_regfile regfile1 (w_clk, w_rs1, w_rs2, w_rrs1, w_rrs2_t, w_ma_rd, w_ma_rslt);

    m_alu alu1 (w_rrs1, w_rrs2, w_alu_c, w_rslt);

    /***** for branch and jump instructions *****/
    wire [31:0] w_tkn_pc = (w_ir[6:2]==5'b11001) ? w_rrs1+w_imm_t : r_pc+w_imm_t;
    m_bru bru0(w_rrs1, w_rrs2, w_bru_c, w_b_rslt);

    /***** for load and store instructions *****/
    wire [2:0]  w_fct3 = w_ir[14:12];
    wire [31:0] w_mem_adr = w_rrs1 + w_imm;
    wire        w_we0 = (w_itype[`D_S_TYPE] & w_fct3[1:0]==0);
    wire        w_we1 = (w_itype[`D_S_TYPE] & w_fct3[0]);
    wire        w_we2 = (w_itype[`D_S_TYPE] & w_fct3[1]);
    wire [3:0]  w_we_sb = (w_we0) ? 4'b0001 << D_ADDR[1:0]       : 0;
    wire [3:0]  w_we_sh = (w_we1) ? 4'b0011 << {D_ADDR[1], 1'b0} : 0;
    wire [3:0]  w_we_sw = (w_we2) ? 4'b1111                      : 0;

    assign D_WE   = (w_we_sh ^ w_we_sw ^ w_we_sb);
    assign D_ADDR = w_mem_adr;
    assign D_OUT  = (w_fct3[0]) ? {2{w_rrs2[15:0]}} :
                    (w_fct3[1]) ? w_rrs2 : {4{w_rrs2[7:0]}};

    /***** for load instructions *****/
    wire [15:0] w_ma_lh_t = (w_mem_adr[1]) ? D_IN[31:16]     : D_IN[15:0];     // align
    wire [7:0]  w_ma_lb_t = (w_mem_adr[0]) ? w_ma_lh_t[15:8] : w_ma_lh_t[7:0]; // align
    wire [31:0] w_ma_lb  = (w_fct3==3'b000) ? {{24{w_ma_lb_t[7]}},  w_ma_lb_t[7:0]}  : 0;
    wire [31:0] w_ma_lbu = (w_fct3==3'b100) ? {24'd0,               w_ma_lb_t[7:0]}  : 0;
    wire [31:0] w_ma_lh  = (w_fct3==3'b001) ? {{16{w_ma_lh_t[15]}}, w_ma_lh_t[15:0]} : 0;
    wire [31:0] w_ma_lhu = (w_fct3==3'b101) ? {16'd0,               w_ma_lh_t[15:0]} : 0;
    wire [31:0] w_ma_lw  = (w_fct3==3'b010) ? D_IN                                   : 0;
    wire [31:0] w_ma_ld_rslt = w_ma_lb ^ w_ma_lbu ^ w_ma_lh ^ w_ma_lhu ^ w_ma_lw;

    wire [31:0] w_ma_rslt_t = (w_itype[`D_LD__IS]) ? w_ma_ld_rslt : w_rslt;

    wire [4:0]  w_ma_rd   = w_rd ;
    wire [31:0] w_ma_rslt = (w_rd!=0) ? w_ma_rslt_t : 0;

    /*************************************************************************************/
`ifdef VERIFY
    initial $write("\n== VERIFY is defined and generate verify.txt\n");
    integer fd;
    initial fd = $fopen("verify.txt", "w");
    reg [31:0] r_tc = 1;
    always @(posedge w_clk) if(!r_rst) r_tc <= r_tc + 1;
    integer i, j;
    always @(posedge w_clk) if(!r_rst) begin
        $fwrite(fd, "%08d %08x %08x\n", r_tc, r_pc, w_ir);
        for (i=0; i<4; i=i+1) for (j=0; j<8; j=j+1) begin
            $fwrite(fd, "%08x", ((i*8+j == {27'd0, w_rd}) && (i*8+j != 0)) ?
                    w_ma_rslt : regfile1.mem[i * 8 + j]);
            $fwrite(fd, "%s", (j != 7 ? " " : "\n"));
        end
    end
`endif
endmodule

/*****************************************************************************************/
module m_regfile ( ///// register file
    input  wire        w_clk,
    input  wire [ 4:0] rs1, rs2,
    output wire [31:0] rdata1, rdata2,
    input  wire [ 4:0] rd,
    input  wire [31:0] wdata
);
    reg [31:0] mem [0:31];
    integer i; initial begin for(i=0; i<32; i=i+1) mem[i]=0; end
    assign rdata1 = mem[rs1];
    assign rdata2 = mem[rs2];
    always @(posedge w_clk) mem[rd] <= wdata;
endmodule

/*****************************************************************************************/
module m_bru ( ///// Branch Resolution Unit
    input  wire [31:0] in1, in2,
    input  wire [ 6:0] s,
    output wire        rslt
);
    wire signed [31:0] sin1 = in1;
    wire signed [31:0] sin2 = in2;
    wire ex0 = s[0] & ( in1 ==  in2);  // BEQ
    wire ex1 = s[1] & ( in1 !=  in2);  // BNE
    wire ex2 = s[2] & (sin1 <  sin2);  // BLT
    wire ex3 = s[3] & (sin1 >= sin2);  // BGE
    wire ex4 = s[4] & ( in1 <   in2);  // BLTU
    wire ex5 = s[5] & ( in1 >=  in2);  // BGEU
    wire ex6 = s[6];                   // JAL, JALR  -> always taken
    assign rslt = ex0 ^ ex1 ^ ex2 ^ ex3 ^ ex4 ^ ex5 ^ ex6;
endmodule

/*****************************************************************************************/
module m_alu ( ///// Arithmetic and Logic Unit
    input  wire [31:0] in1, in2,
    input  wire [10:0] s,
    output wire [31:0] rslt
);
    wire signed [31:0] sin1 = in1;
    wire signed [31:0] sin2 = in2;
    wire        ex0  = (s[0])  ?  in1 <  in2      : 0;
    wire        ex1  = (s[1])  ? sin1 < sin2      : 0;
    wire [31:0] ex2  = (s[2])  ?  in1 +  in2      : 0;
    wire [31:0] ex3  = (s[3])  ?  in1 -  in2      : 0;
    wire [31:0] ex4  = (s[4])  ?  in1 ^  in2      : 0;
    wire [31:0] ex5  = (s[5])  ?  in1 |  in2      : 0;
    wire [31:0] ex6  = (s[6])  ?  in1 &  in2      : 0;
    wire [31:0] ex7  = (s[7])  ?  in1 << in2[4:0] : 0;
    wire [31:0] ex8  = (s[8])  ?  in1 >> in2[4:0] : 0;
    wire [31:0] ex9  = (s[9])  ? sin1 >>>in2[4:0] : 0;
    wire [31:0] ex10 = (s[10]) ?          in2     : 0;
    assign rslt = {31'd0, ex0} ^ {31'd0, ex1} ^
                  ex2 ^ ex3 ^ ex4 ^ ex5 ^ ex6 ^ ex7 ^ ex8 ^ ex9 ^ ex10;
endmodule

/*****************************************************************************************/
module m_get_imm(ir, i, s, b, u, j, imm);
  input wire [31:0] ir;
  input wire i, s, b, u, j;
  output wire [31:0] imm;
  assign imm= (i) ? {{20{ir[31]}},ir[31:20]} :
              (s) ? {{20{ir[31]}},ir[31:25],ir[11:7]} :
              (b) ? {{20{ir[31]}},ir[7],ir[30:25],ir[11:8],1'b0} :
              (u) ? {ir[31:12],12'b0} :
              (j) ? {{12{ir[31]}},ir[19:12],ir[20],ir[30:21],1'b0} : 0;
endmodule
/*****************************************************************************************/
module m_decode (
    input  wire [31:0] ir,
    output wire [ 4:0] rd, rs1, rs2,
    output wire        op_imm,
    output wire [ 9:0] itype,
    output wire [10:0] alu_c,
    output wire [ 6:0] bru_c,
    output wire [31:0] w_imm
```

```
);
    wire [4:0] opcode5 = ir[6:2]; // use 5-bit, cause lower 2-bit are always 2'b11
    wire j = (opcode5==5'b11011);                       // J-type
    wire b = (opcode5==5'b11000);                       // B-type
    wire s = (opcode5==5'b01000);                       // S-type
    wire r = (opcode5==5'b01100);                       // R-type
    wire u = (opcode5==5'b01101 || opcode5==5'b00101);  // U-type
    wire i = ~(j | b | s | r | u);                      // I-type

    wire ja = (opcode5==5'b11001 || opcode5==5'b11011); // JAL, JALR insn
    wire ld = (opcode5==5'b00000);                      // LB, LH, LW, LBU, LHU insn
    wire mul= r & ir[25] & !ir[14];                     // MUL, MULH, MULHSU, MULHU insn
    wire div= r & ir[25] &  ir[14];                     // DIV, DIVU, REM, REMU insn
    wire reg_we = (ir[11:7]!=0) && (!s && !b);          // rd!=0 & (not store & not branch)
    assign op_imm = (opcode5==5'b00100 || u | ja);      // use immediate for rrs2, Note
    assign rd     = (reg_we)    ? ir[11:7]  : 5'd0;
    assign rs1    = (!u && !j)  ? ir[19:15] : 5'd0;
    assign rs2    = (b | s | r) ? ir[24:20] : 5'd0;
    assign itype = {div, mul, ld, ja, u, j, b, s, r, i};

    wire [2:0] fct3   = ir[14:12]; // funct3
    wire [6:0] fct7   = ir[31:25]; // funct7
    wire [3:0] alu_op = (opcode5==5'b01100) ? {fct7[5], fct3} :
                        (opcode5==5'b00100) ? {(fct3==3'h5 & fct7[5]), fct3} : 4'hf;
    /********** one-hot encoding of ALU control *********************/
    assign alu_c[0] = (alu_op==4'h3);      // `ALU_CTRL_SLTU
    assign alu_c[1] = (alu_op==4'h2);      // `ALU_CTRL_SLT
    assign alu_c[2] = (alu_op==4'h0 || u); // `ALU_CTRL_ADD
    assign alu_c[3] = (alu_op==4'h8);      // `ALU_CTRL_SUB
    assign alu_c[4] = (alu_op==4'h4);      // `ALU_CTRL_XOR
    assign alu_c[5] = (alu_op==4'h6);      // `ALU_CTRL_OR
    assign alu_c[6] = (alu_op==4'h7);      // `ALU_CTRL_AND
    assign alu_c[7] = (alu_op==4'h1);      // `ALU_CTRL_SLL
    assign alu_c[8] = (alu_op==4'h5);      // `ALU_CTRL_SRL
    assign alu_c[9] = (alu_op==4'hd);      // `ALU_CTRL_SRA
    assign alu_c[10]= ja;                  // `JAL, JALR
    /********** one-hot encoding of branch control ******************/
    assign bru_c[0] = (b & fct3==3'b000);  // BEQ
    assign bru_c[1] = (b & fct3==3'b001);  // BNE
    assign bru_c[2] = (b & fct3==3'b100);  // BLT
    assign bru_c[3] = (b & fct3==3'b101);  // BGE
    assign bru_c[4] = (b & fct3==3'b110);  // BLTU
    assign bru_c[5] = (b & fct3==3'b111);  // BGEU
    assign bru_c[6] = ja;                  // JAL, JALR  -> always taken

    m_get_imm m_get_imm1 (ir, i, s, b, u, j, w_imm);
endmodule

/*****************************************************************************************/
module m_imem (
    input  wire        w_clk,
    input  wire        w_we,
    input  wire [9:0]  w_adr,
    input  wire [31:0] w_din,
    output wire [31:0] w_dout
);
    reg [31:0] mem [0:1023];
`ifndef SYNTHESIS
initial begin
`include `MEM_TXT
end
`endif

    always @(posedge w_clk) if (w_we) mem[w_adr] <= w_din;
    assign w_dout = mem[w_adr];
endmodule

/*****************************************************************************************/
module m_dmem (
    input  wire        w_clk,
    input  wire [3:0]  w_we,
    input  wire [9:0]  w_adr,
    input  wire [31:0] w_din,
    output wire [31:0] w_dout
);
    reg [31:0] mem [0:1023];
`ifndef SYNTHESIS
initial begin
`include `MEM_TXT
end
`endif

    always @(posedge w_clk) begin
        if (w_we[0]) mem[w_adr][ 7: 0] <= w_din[ 7: 0];
        if (w_we[1]) mem[w_adr][15: 8] <= w_din[15: 8];
        if (w_we[2]) mem[w_adr][23:16] <= w_din[23:16];
        if (w_we[3]) mem[w_adr][31:24] <= w_din[31:24];
    end
    assign w_dout = mem[w_adr];
endmodule

`ifndef SYNTHESIS
```

```
/*****************************************************************************************/
module m_top();
    reg r_clk = 0;
    initial forever #50 r_clk = ~r_clk;

    wire w_uart_rx;
    wire w_uart_tx;
    m_main main1 (r_clk, w_uart_rx, w_uart_tx);
endmodule

module clk_wiz_0(
    output wire w_clock_out,
    output wire w_locked,
    input  wire w_clock_in
);
    assign w_locked = 1;
    assign w_clock_out = w_clock_in;
endmodule
/*****************************************************************************************/
`define DONE_INIT 1
`else
`define DONE_INIT 0
`endif

/*****************************************************************************************/
module m_main (
    input  wire w_clk,      // 100MHz clock signal
    input  wire w_uart_rx,  // UART rx, data line from PC to FPGA
    output wire w_uart_tx   // UART tx, data line from FPGA to PC
);
    wire w_clk50m, locked;
    clk_wiz_0 clk_wiz1 (w_clk50m, locked, w_clk);

    reg r_rst = 1;
    always @(posedge w_clk50m) r_rst <= ~locked;

    reg core_rst = 1;
    always @(posedge w_clk50m) core_rst <= (r_rst | r_done==0);

    wire w_en;
    wire [7:0] w_char;
    m_uart_rx uart_rx1 (w_clk50m, w_uart_rx, w_char, w_en);

    reg r_done = `DONE_INIT;
    reg r_we = 0;
    reg [31:0] r_wcnt = 0, r_adr = 0, r_data = 0;
    always @(posedge w_clk50m) if (r_done==0) begin
        if(w_en) begin
            r_adr  <= r_wcnt;
            r_data <= {w_char, r_data[31:8]};
            r_we   <= (r_wcnt[1:0]==3);
            r_wcnt <= r_wcnt + 1;
        end else begin
            r_we <= 0;
            if (r_wcnt>=`MEM_SIZE) r_done <= 1;
        end
    end

    wire [3:0]  D_WE;
    wire [31:0] I_IN, I_ADDR, D_IN, D_OUT, D_ADDR;
    m_rvcore rvcore1 (w_clk50m, core_rst, I_IN, D_IN, I_ADDR, D_ADDR, D_OUT, D_WE);

    m_imem imem1 (w_clk50m, r_we, r_done ? I_ADDR[11:2] : r_adr[11:2], r_data, I_IN);
    m_dmem dmem1 (w_clk50m, {4{r_we}} | D_WE, r_done ? D_ADDR[11:2] : r_adr[11:2],
                  r_done ? D_OUT : r_data, D_IN);

    wire [1:0] tohost_cmd  = ((D_ADDR==32'h40008000) && D_WE==4'hf) ? D_OUT[17:16] : 0;
    wire [7:0] tohost_char = D_OUT[7:0];

    reg [7:0] ubuf [0:4095];
    reg [11:0] r_head=0, r_tail=0;
    always @(posedge w_clk50m) if (core_rst==0 && tohost_cmd==1) begin
        ubuf[r_tail] <= tohost_char;
        r_tail <= r_tail + 1;
    end

    reg [31:0] r_cnt = 1;
    always @(posedge w_clk50m) r_cnt <= (r_cnt>=(`UART_CNT*12)) ? 1 : r_cnt + 1;

    wire w_uart_we = ((core_rst==0) && (r_cnt==1) && (r_head!=r_tail));
    wire w_uart_tx_t;
    always @(posedge w_clk50m) if(core_rst==0 && w_uart_we) r_head <= r_head + 1;
    m_uart_tx uart_tx1 (w_clk50m, w_uart_we, ubuf[r_head], w_uart_tx);

    always@(posedge w_clk50m) begin ///// for simulation
        if (tohost_cmd==1) $write("%c", tohost_char);
        if (tohost_cmd==2) begin
            $write("\nsimulation finished.\n");
            $finish();
        end
    end
endmodule
```

```
369
370 /*************************************************************************************/
371 module m_uart_tx (
372     input  wire       w_clk,     // 50MHz clock signal
373     input  wire       w_we,      // write enable
374     input  wire [7:0] w_din,     // data in
375     output wire       w_uart_tx  // UART tx, data line from FPGA to PC
376 );
377     reg [8:0] r_buf = 9'b1_1111_1111;
378     reg [7:0] r_cnt = 1;
379     always @(posedge w_clk) begin
380         r_cnt <= (w_we) ? 1 : (r_cnt==`UART_CNT) ? 1 : r_cnt + 1;
381         r_buf <= (w_we) ? {w_din, 1'b0} : (r_cnt==`UART_CNT) ? {1'b1, r_buf[8:1]} : r_buf;
382     end
383     assign w_uart_tx = r_buf[0];
384 endmodule
385
386 /*************************************************************************************/
387 module m_uart_rx (
388     input  wire       w_clk,   // 100MHz clock signal
389     input  wire       w_rxd,   // UART rx, data line from PC to FPGA
390     output wire [7:0] w_char,  // 8-bit data received
391     output reg        r_en = 0 // data enable
392 );
393     reg [2:0] r_detect_cnt = 0; /* to detect the start bit */
394     always @(posedge w_clk) r_detect_cnt <= (w_rxd) ? 0 : r_detect_cnt + 1;
395     wire w_detected = (r_detect_cnt>2);
396
397     reg       r_busy = 0; // r_busy is set while receiving 9-bits data
398     reg [3:0] r_bit  = 0; // the number of received bits
399     reg [7:0] r_cnt  = 0; // wait count for 1Mbaud
400     always@(posedge w_clk) r_cnt <= (r_busy==0) ? 1 : (r_cnt==`UART_CNT) ? 1 : r_cnt + 1;
401
402     reg [8:0] r_data = 0;
403     always@(posedge w_clk) begin
404         if (r_busy==0) begin
405             {r_data, r_bit, r_en} <= 0;
406             if (w_detected) r_busy <= 1;
407         end
408         else if (r_cnt>= `UART_CNT) begin
409             r_bit <= r_bit + 1;
410             r_data <= {w_rxd, r_data[8:1]};
411             if (r_bit==8) begin r_en <= 1; r_busy <= 0; end
412         end
413     end
414     assign w_char = r_data[7:0];
415 endmodule
416 /*************************************************************************************/
```