HW#02: Memory model, RISC-V Instruction, Decoder

SATYAM (2023-81784)

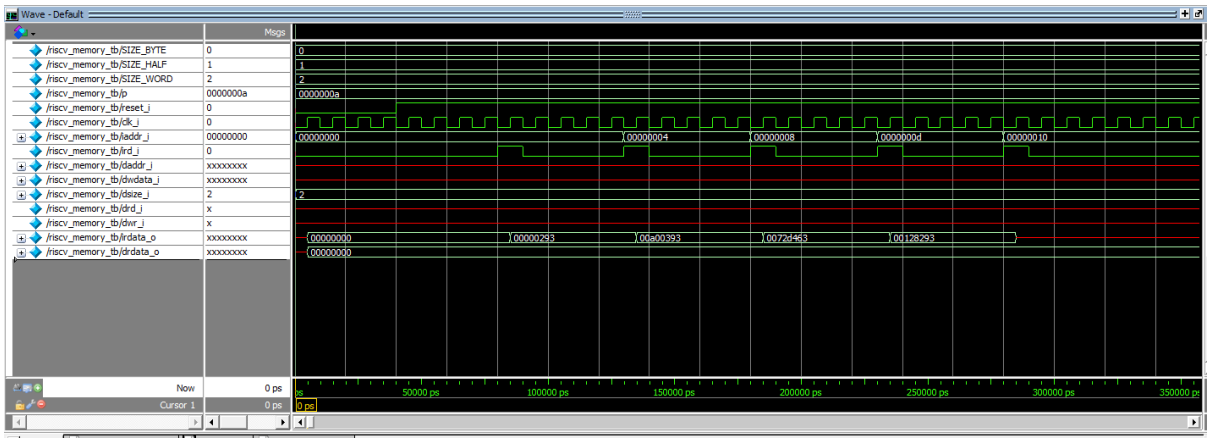
Problem 1 (15p): Memory model

(a) Memory model (15p)

Files:

1. **Lab03/ex1_riscv_memory/a/riscv_memory.v**
2. **Lab03/ex1_riscv_memory/a/riscv_memory_tb.v**

Waveform -

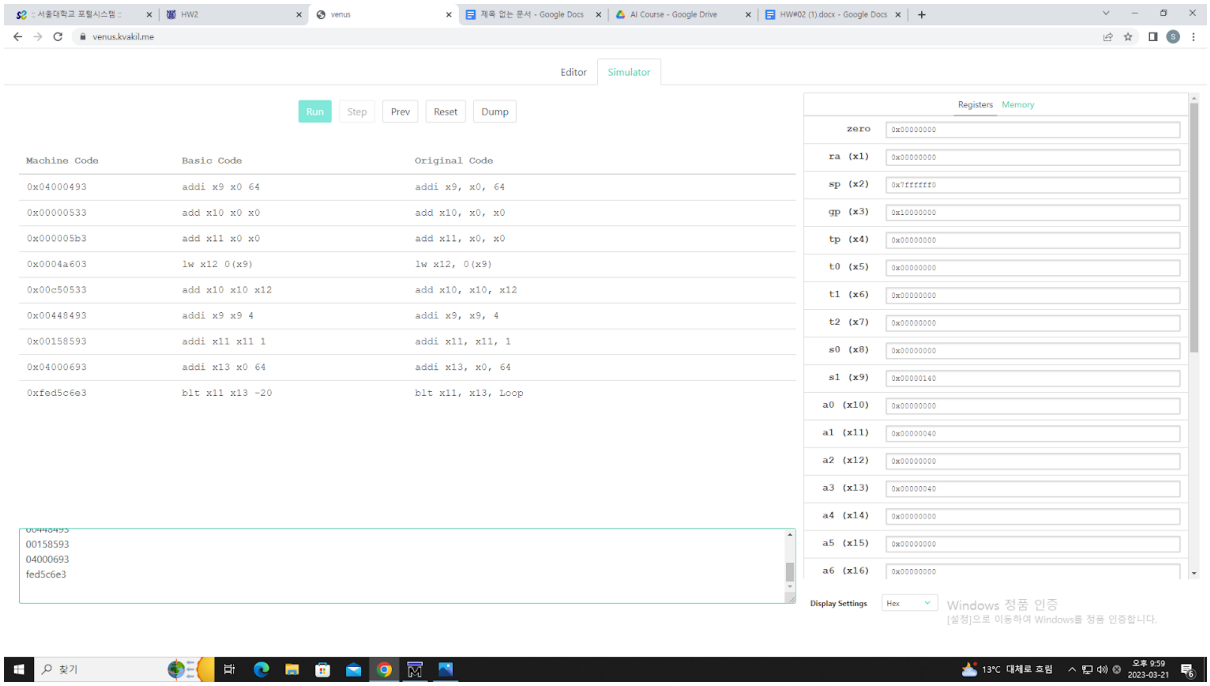


(b) Program memory file (5p)

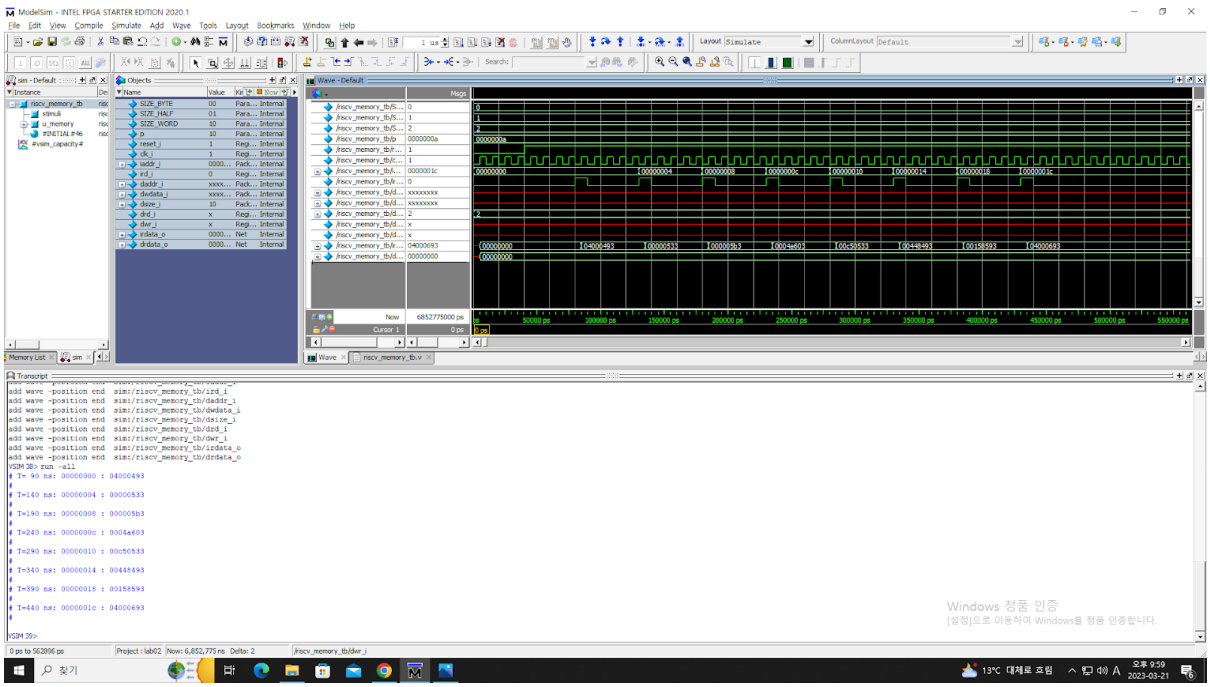
Files:

1. **Lab03/ex1_riscv_memory/b/riscv_memory_tb.v**

RISC-V simulator Venus (https://www.kvakil.me/venus/) to generate codes -


Editor
Simulator
Run
Step
Prev
Reset
Dump
Machine Code
Basic Code
Original Code
0x04000493 addi x9 x0 64 addi x9, x0, 64
0x00000533 add x10 x0 x0 add x10, x0, x0
0x000005b3 add x11 x0 x0 add x11, x0, x0
0x0004a603 lw x12 0(x9) lw x12, 0(x9)
0x00c50533 add x10 x10 x12 add x10, x10, x12
0x00448493 addi x9 x9 4 addi x9, x9, 4
0x00158593 addi x11 x11 1 addi x11, x11, 1
0x04000693 addi x13 x0 64 addi x13, x0, 64
0xfed5c6e3 blt x11 x13 -20 blt x11, x13, Loop
00158593
04000693
fed5c6e3
Registers
Memory
zero
ra (x1)
sp (x2)
gp (x3)
tp (x4)
t0 (x5)
t1 (x6)
t2 (x7)
s0 (x8)
s1 (x9)
a0 (x10)
a1 (x11)
a2 (x12)
a3 (x13)
a4 (x14)
a5 (x15)
a6 (x16)
Display Settings
Windows 정품 인증

Waveform -

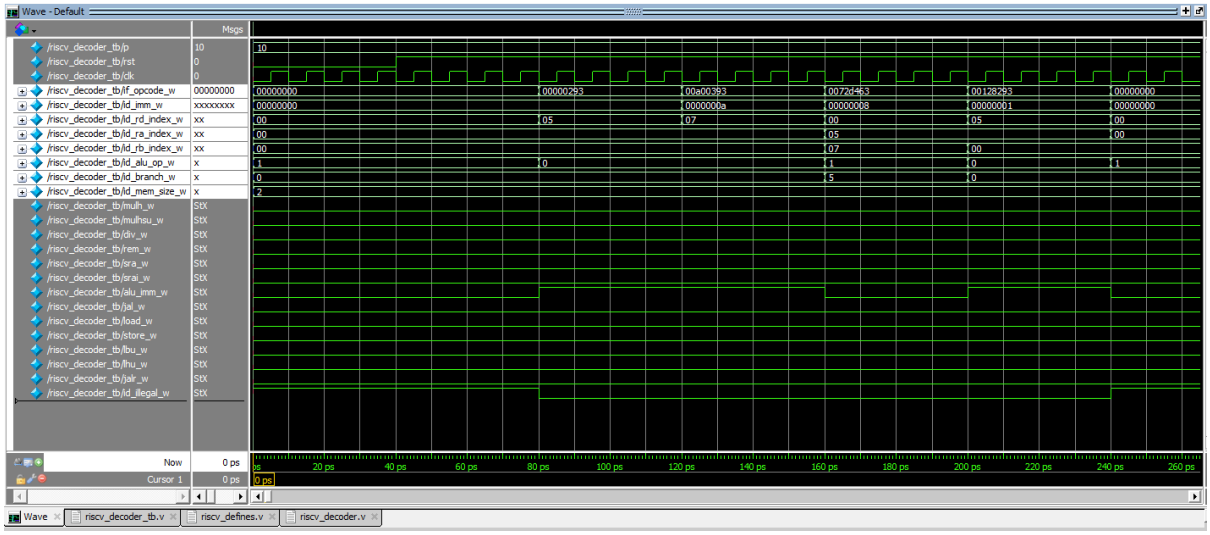

ModelSim - INTEL FPGA STARTER EDITION 2020.1
Transcript
VSIM 38> run -all
T= 90 ns: 00000000 : 04000493
T=140 ns: 00000004 : 00000533
T=190 ns: 00000008 : 000005b3
T=240 ns: 0000000c : 0004a603
T=290 ns: 00000010 : 00c50533
T=340 ns: 00000014 : 00448493
T=390 ns: 00000018 : 00158593
T=440 ns: 0000001c : 04000693
Windows 정품 인증

Problem 2 (15p): RISC-V Decoder

Files:

1. **Lab03/ex2_decoder/riscv_decoder.v**
2. **Lab03/ex2_decoder/riscv_decoder_tb.v**

Waveform -

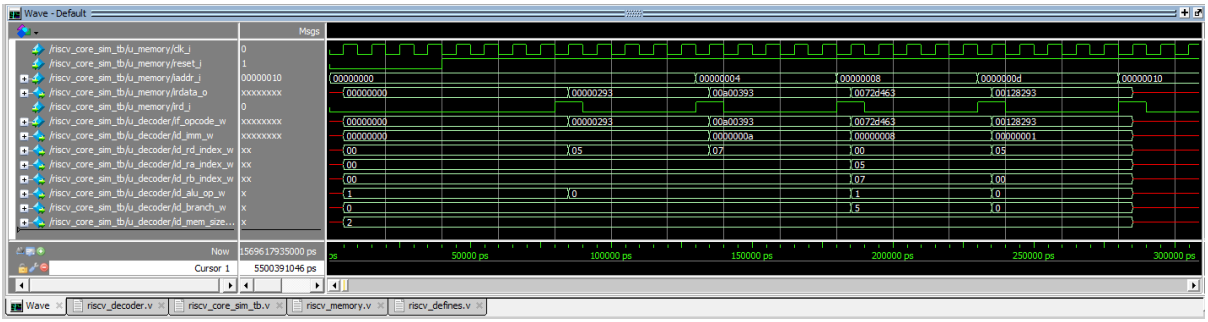


Problem 3 (5p): Memory & Decoder

Files:

1. Lab03/ex3_riscv_core_sim/a/riscv_core_sim_tb.v

Waveform -

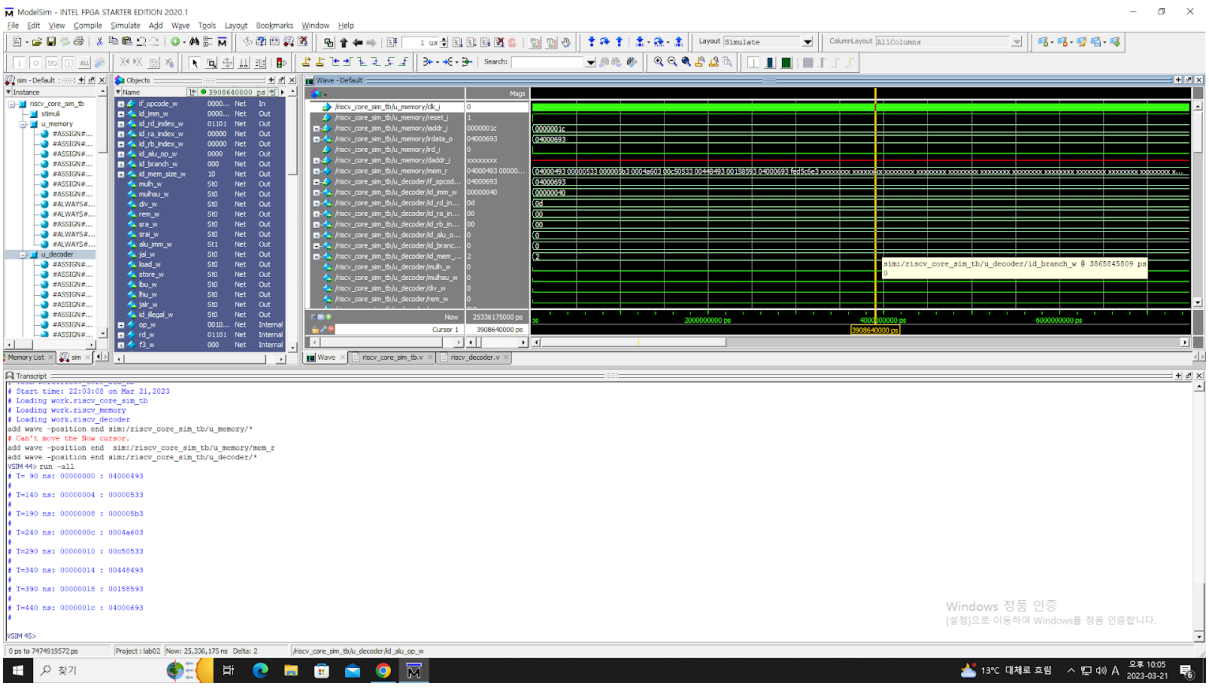


Modified test bench -

Files:

1. Lab03/ex3_riscv_core_sim/b/riscv_core_sim_tb.v

Waveform -



Problem 4 (2p): (Optional) Bonus

Briefly explain your code in Problem 1.

file : riscv_memory.v

```
//-------------------------------------------------------------------------------------------------------------
//**** module description begins with ?module modulename?
//    : this file descripts the function of module
//-------------------------------------------------------------------------------------------------------------
module riscv_memory #(
  parameter     SIZE     = 8192,
  parameter     FIRMWARE = "C:/Users/ece/Downloads/Lab03/Lab03/ex1_riscv_memory/mem.hex"
)
(
//-------------------------------------------------------------------------------------------------------------
// **** Input / output ports definition
//     : module input/output could be connected to the other module
//     : Input controlled by designer testbench for simulation.
//     : output monitored by testbench for funtional verification.
//-------------------------------------------------------------------------------------------------------------
  input         clk_i,
  input         reset_i,

  input  [31:0] iaddr_i,
  output [31:0] irdata_o,
  input         ird_i,

  input  [31:0] daddr_i,
  input  [31:0] dwdata_i,
  output [31:0] drdata_o,
  input   [1:0] dsize_i,
  input         drd_i,
  input         dwr_i
);

//-------------------------------------------------------------------------------
localparam
  SIZE_BYTE = 2'd0,
  SIZE_HALF = 2'd1,
  SIZE_WORD = 2'd2;

localparam
  DEPTH = $clog2(SIZE);

//-------------------------------------------------------------------------------
reg [1:0] daddr_r;
reg [1:0] dsize_r;
reg [31:0] irdata_r; //Internal instruction register
reg [31:0] drdata_r; //Internal data register
//-------------------------------------------------------------------------------

wire [31:0] dwdata_w =      //*** dwdata_w[31:0] reflects write data with repetition.
    (SIZE_BYTE == dsize_i) ? {4{dwdata_i[7:0]}} :  //    If Byte write, x4 repeat
    (SIZE_HALF == dsize_i) ? {2{dwdata_i[15:0]}} : dwdata_i; //    If Half Word, x2 repeat


wire [3:0] dbe_byte_w =           //*** if BYTE write,
    (2'b00 == daddr_i[1:0]) ? 4'b0001 :         //    iaddr_i[31:0]  is addressing Byte address.
        //    LSB 2bits, iaddr_i[1:0] represents one of 4 dividen slot# among 32bits
    (2'b01 == daddr_i[1:0]) ? 4'b0010 :         //    It indicates descaled address to write memory.
    (2'b10 == daddr_i[1:0]) ? 4'b0100 : 4'b1000;       //    [ #3 (1 Byte) | #2(1Byte) | #1(1Byte) | slot #0 (1Byte) ]
        //    |<----------------- 32bits------------------------------>|
        //        2'b11    2'b10    2'b01 2'b00     <----iaddr_i[1:0]



wire [3:0] dbe_half_w =   //*** if HALF WORD write,
    daddr_i[1] ? 4'b1100 : 4'b0011; //    iaddr_i[31:0]  is addressing HALFWORD address.
     //    iaddr_i[1] represents one of 2 dividen slot# among 32bits
     //    It indicates descaled address to write memory.
     //    [   #1 (1 Half Word)      |   slot #0 (1 Half Word)   ]
     //    |<----------------- 32bits------------------------------>|
     //             1'b1       1'b0            <----iaddr_i[1]


wire [3:0] dbe_w =   //*** dbe_w[3:0] indicating slot numbers to select segmented write position
(SIZE_BYTE == dsize_i) ? dbe_byte_w :         // If Byte write,
(SIZE_HALF == dsize_i) ? dbe_half_w : 4'b1111;  //  4'b0001 slot #0
      //  4'b0010 slot #1
      //  4'b0100 slot #2
      //  4'b1000 slot #3
      // If Half Word write,
      //  4'b0011 slot #0
      //  4'b1100 slot #1
      // If WORD write,
      // 4'b1111 (no slot division)


wire [7:0] rdata_byte_w =   //*** extract valid size from 32bits memory data according to read data size
    (2'b00 == daddr_r) ? drdata_r[7:0] : //    If Byte, daddr_i LSB 2bits decides slot# to read,
    (2'b01 == daddr_r) ? drdata_r[15:8] : //     [ #3 (1 Byte) | #2(1Byte) | #1(1Byte) | slot #0 (1Byte) ]
    (2'b10 == daddr_r) ? drdata_r[23:16] :      //     |<----------------- 32bits------------------------------>|
    drdata_r[31:24];    //         2'b11    2'b10     2'b01    2'b00     <----daddr_r[1:0]



wire [15:0] rdata_half_w =        //    If Half word,
    daddr_r[1] ? drdata_r[31:16] : drdata_r[15:0];   //    daddr_i[1] represents one of 2 dividen slot# among 32bits
```

```
            //    [   #1 (1 Half Word)      |   slot #0 (1 Half Word)   ]
       //    |<----------------- 32bits------------------------------>|
       //             1'b1       1'b0            <----daddr_i[1]

//-----------------------------------------------------------------------------
// Data memory: Read out
// Read data size rescale to fit 32bits module output
//-----------------------------------------------------------------------------
assign drdata_o =
    (SIZE_BYTE == dsize_r) ? { 24'b0, rdata_byte_w } :
    (SIZE_HALF == dsize_r) ? { 16'b0, rdata_half_w } : drdata_r;


always @(posedge clk_i) begin
  if (~reset_i) begin
    daddr_r <= 2'b00;
    dsize_r <= SIZE_BYTE;
  end else begin
    daddr_r <= daddr_i[1:0];
    dsize_r <= dsize_i;
  end
end

//-----------------------------------------------------------------------------
// Memory Initialization
//-----------------------------------------------------------------------------
reg [31:0] mem_r [0:SIZE/4-1];

initial begin
  $readmemh(FIRMWARE, mem_r);
end

//-----------------------------------------------------------------------------
// Instruction memory: READ ONLY
//-----------------------------------------------------------------------------
always @(posedge clk_i) begin
  if (~reset_i)
 irdata_r <= 32'h0;
  else begin
 if (ird_i)
  irdata_r <= mem_r[iaddr_i[DEPTH:2]];
  end
end
// Output ports for Instruction
assign irdata_o = irdata_r;

//-----------------------------------------------------------------------------
// Data memory: READ/WRITE
//-----------------------------------------------------------------------------
// Read operation
//-----------------------------------------------------------------------------

always @(posedge clk_i) begin
  if (~reset_i)
     drdata_r <= 32'h0;
  else begin
 if(drd_i)
  drdata_r <= mem_r[daddr_i[DEPTH:2]];
  end
end

//-----------------------------------------------------------------------------
// Write operation
//-----------------------------------------------------------------------------
always @(posedge clk_i) begin
  if (dbe_w[0] && dwr_i)    //*** According to write data size (dsize_i),  marks dbe_w[3:0]
       //    to select a segmented memory address to overwrite
    mem_r[daddr_i[DEPTH:2]][7:0] <= dwdata_w[7:0]; //    For BYTE mode, select 1 of 4 slots among 32bits
  if (dbe_w[1] && dwr_i)    //    4'b0001  : 4'b0010 : 4'b0100 : 4'b1000
    mem_r[daddr_i[DEPTH:2]][15:8] <= dwdata_w[15:8];  //     #3   #2     #1       slot#0

  if (dbe_w[2] && dwr_i)      //    For HALFWORD mode, select 1 of 2 slots among 32bits
    mem_r[daddr_i[DEPTH:2]][23:16]<= dwdata_w[23:16];   //    4'b0011         : 4'b1100
       //    lower 16 bits overwrite  higher 16 bits overwrite
  if (dbe_w[3] && dwr_i)    //    For WORD mode
    mem_r[daddr_i[DEPTH:2]][31:24]<= dwdata_w[31:24];   //    full 32bits overwrite

end

//-----------------------------------------------------------------------------
//**** module description ends with ?endmodule?
//-----------------------------------------------------------------------------
endmodule
```

ird_i → true

As last two bits will be always zero of the address (because address goes with 4 byte difference)

same as instruction address.

This code selects where next data goes depending on the length of data (if its a byte or half or word)