ELEC373
Digital Systems Design
Assignment 3
MIP Processor

Junhao Zhang
201377244

20 March 2020

Declaration

I confirm that I have read and understood the University's definitions of plagiarism and collusion from the Code of Practice on Assessment. I confirm that I have neither committed plagiarism in the completion of this work nor have I colluded with any other party in the preparation and production of this work. The work presented here is my own and in my own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web). I understand the consequences of engaging in plagiarism and collusion as described in the Code of Practice on Assessment (Appendix L).

1 Contents

2 Introduction

In this assignment, there are two parts including developing an assembly language code to display the lower 8 digits of student ID number and implementing three additional instructions by modifying the Verilog codes. Some assembly programs are developed in order to prove that the instructions are working properly. The simulations result generated from the ModelSim demonstrate that the desired function can be achieved.

3 Part A: Displaying ID on the 7-segment LED

3.1 Assembly Language Code

```
# number 0
lui $s0, 0x0000
addiu $s0, $s0, 0x0040
# number 1
lui $s1, 0x0000
addiu $s1, $s1, 0x0079
# number 3
lui $s2, 0x0000
addiu $s2, $s2, 0x0030
# number 7
lui $s3, 0x0000
addiu $s3, $s3, 0x0078
# number 7
lui $s4, 0x0000
addiu $s4, $s4, 0x0078
# number 2
lui $s5, 0x0000
addiu $s5, $s5, 0x0024
# number 4
lui $s6, 0x0000
addiu $s6, $s6, 0x0019
# number 4
lui $s7, 0x0000
addiu $s7, $s7, 0x0019

# 7-segment HEX7
lui $t0, 0xffff
addiu $t0, $t0, 0x202C
# 7-segment HEX6
lui $t1, 0xffff
addiu $t1, $t1, 0x2028
# 7-segment HEX5
lui $t2, 0xffff
addiu $t2, $t2, 0x2024
# 7-segment HEX4
```

```
lui $t3, 0xffff
addiu $t3, $t3, 0x2020
# 7-segment HEX3
lui $t4, 0xffff
addiu $t4, $t4, 0x201C
# 7-segment HEX2
lui $t5, 0xffff
addiu $t5, $t5, 0x2018
# 7-segment HEX1
lui $t6, 0xffff
addiu $t6, $t6, 0x2014
# 7-segment HEX0
lui $t7, 0xffff
addiu $t7, $t7, 0x2010

# stores the values into the 7-segment registers
sw $s0, 0x0000($t0)
sw $s1, 0x0000($t1)
sw $s2, 0x0000($t2)
sw $s3, 0x0000($t3)
sw $s4, 0x0000($t4)
sw $s5, 0x0000($t5)
sw $s6, 0x0000($t6)
sw $s7, 0x0000($t7)

while: j while
```

3.2 ModelSim Result

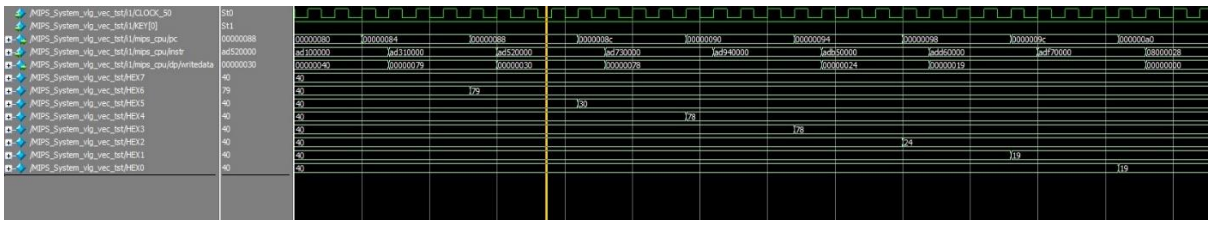

Figure 1: ModelSim Simulation Result

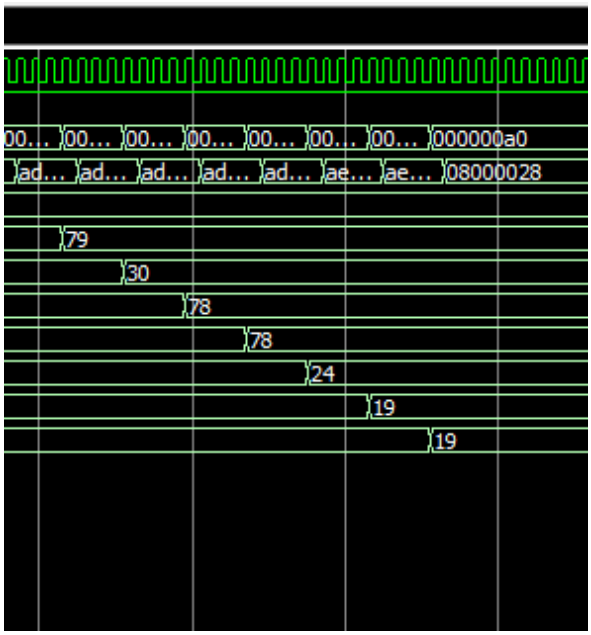

Figure 2: ModelSim Simulation Result

As demonstrated in Figure 1 and Figure 2, the simulation result shows that the 8 numbers are in succession stored into the memories with the addresses from 0xFFFF2010 to 0xFFFF202C when each sw instruction is executed. The hexadecimal numbers indicates the LEDs on the 7-segment display and the numbers are respectively 0, 1, 3, 7, 7, 2, 4, 4 displayed on the 7-segment.

3.3 A Photograph of the 7-segment Displays

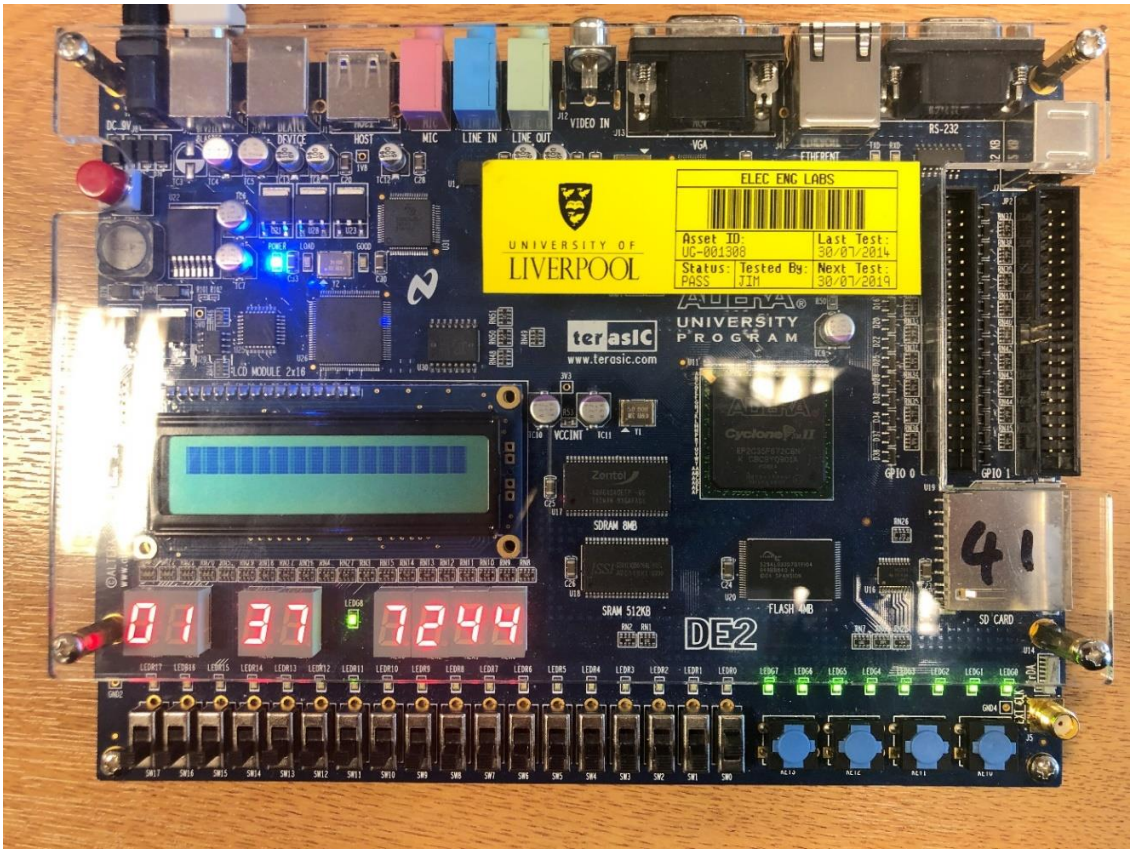

Figure 3: ID Displayed on the 7-segment

4 Part B: Implementation of Additional Instructions

4.1 Modified Modules

In the following sections, the modified Verilog code for the modules are highlighted with red underline. Also included are some extra module added into the Verilog code.

4.1.1 Byte Address Decider

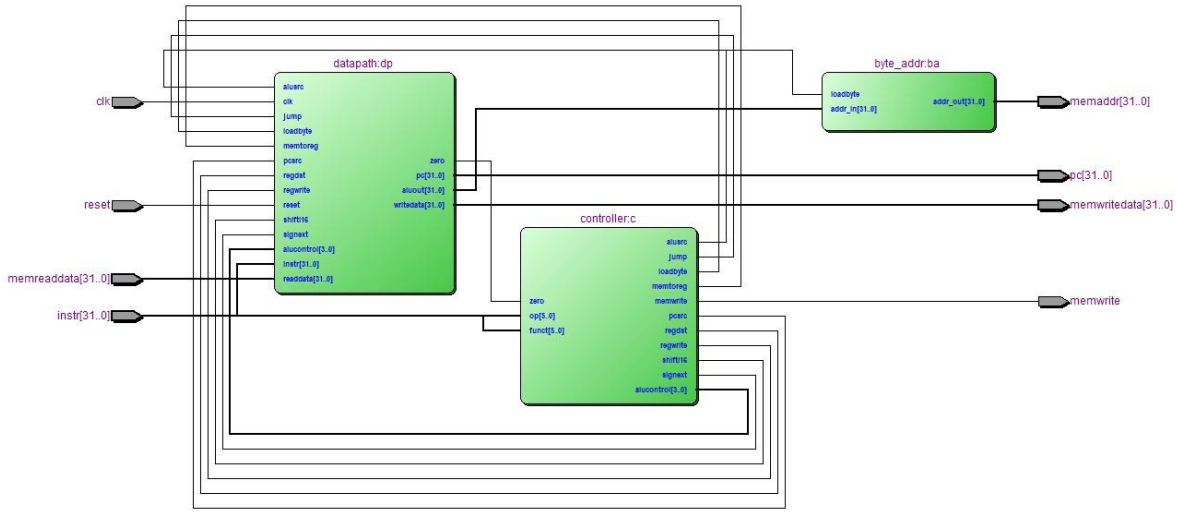

Figure 4: Modified Block Diagram

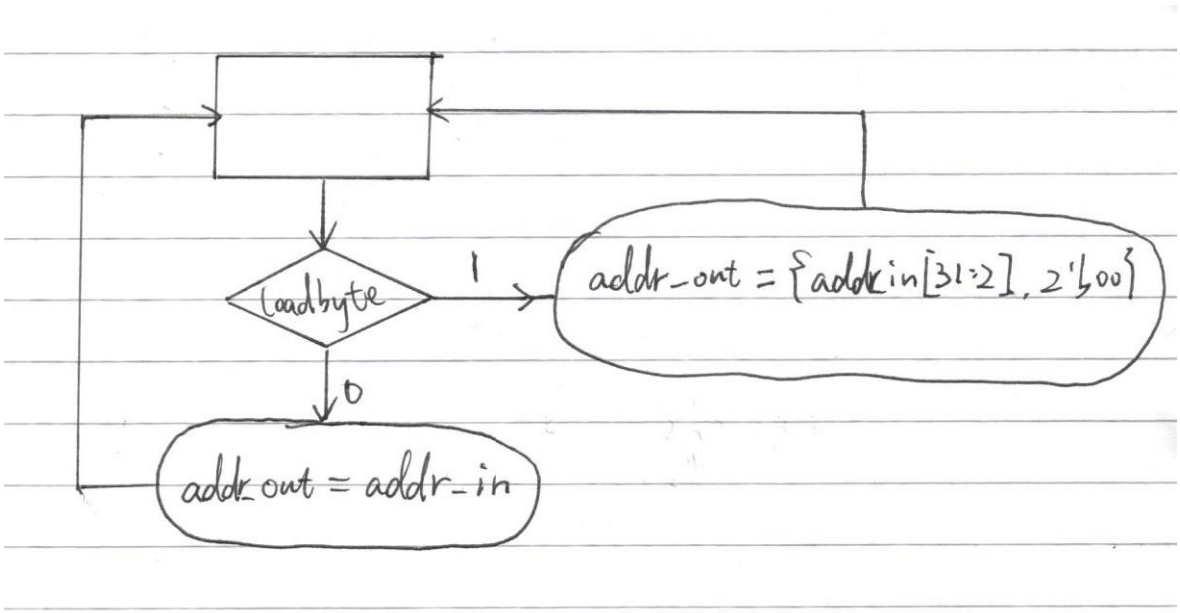

Figure 5: ASM

```
module byte_addr( input     [31:0]  addr_in,
                  input             loadbyte,
                 output reg  [31:0]  addr_out);
  always @(*)
  begin
    if(loadbyte)
      addr_out <= {addr_in[31:2], 2'b00};
    else
      addr_out <= addr_in;
  end
```

```
endmodule
```

This module is connected between the aluout signal from the data path and the memaddr signal of the memory. It determines whether a full 32 bit data should be transmitted to the memory or a 32 bit data with the least two significant bits replaced by two bits of zeros. This is depending on the loadbyte signal which indicates whether the instruction is requiring to load a byte from the memory.

4.1.2 Mips

```
module mips(input         clk, reset,
            output [31:0] pc,
            input  [31:0] instr,
            output        memwrite,
            output [31:0] memaddr,
            output [31:0] memwritedata,
            input  [31:0] memreaddata);

  wire        signext, shiftl16, memtoreg, branch;
  wire        pcsrc, zero;
  wire        alusrc, regdst, regwrite, jump;
  wire        loadbyte;
  wire [3:0]  alucontrol;
  wire [31:0] aluout;

  // Instantiate Controller
  controller c(.op         (instr[31:26]),
               .funct      (instr[5:0]),
               .zero       (zero),
               .signext    (signext),
               .shiftl16   (shiftl16),
               .memtoreg   (memtoreg),
               .memwrite   (memwrite),
               .pcsrc      (pcsrc),
               .alusrc     (alusrc),
               .regdst     (regdst),
               .regwrite   (regwrite),
               .loadbyte   (loadbyte),
               .jump       (jump),
               .alucontrol (alucontrol));

  // Instantiate Datapath
  datapath dp( .clk        (clk),
               .reset      (reset),
               .signext    (signext),
               .shiftl16   (shiftl16),
```

```
                .memtoreg   (memtoreg),
                .pcsrc      (pcsrc),
                .alusrc     (alusrc),
                .regdst     (regdst),
                .regwrite   (regwrite),
                .loadbyte   (loadbyte),
                .jump       (jump),
                .alucontrol (alucontrol),
                .zero       (zero),
                .pc         (pc),
                .instr      (instr),
                .aluout     (aluout),
                .writedata  (memwritedata),
                .readdata   (memreaddata));

   byte_addr ba(.addr_in(aluout),
                .loadbyte(loadbyte),
                .addr_out(memaddr));

endmodule
```

For the MIPS module, an additional module byte_addr is added in order to select the proper memory address from the aluout. Because the LBU instruction requires a different memory address which is modified by replacing the last two bits with zeros, the module can output different memory address format depending on the loadbyte signal.

4.1.3 Main Decoder

```
module maindec(input  [5:0] op,
               output       signext,
               output       shiftl16,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       loadbyte,
               output       jump,
               output [2:0] aluop);

  reg [13:0] controls;

  assign {signext, shiftl16, regwrite, regdst,
          alusrc, branch, memwrite,
          memtoreg, loadbyte, jump, aluop} = controls;

  always @(*)
    case(op)
      6'b000000: controls <= 13'b0011000000100; // Rtype
```

```
      6'b100011: controls <= 13'b1010100100000; // LW
      6'b100100: controls <= 13'b1010100110000; // LBU
      6'b101011: controls <= 13'b1000101000000; // SW
      6'b000100: controls <= 13'b1000010000001; // BEQ
      6'b001000,
      6'b001001: controls <= 13'b1010100000000; // ADDI, ADDIU: only differenc
e is exception
      6'b001101: controls <= 13'b0010100000010; // ORI
      6'b001100: controls <= 13'b0010100000011; // ANDI
      6'b001111: controls <= 13'b0110100000000; // LUI
      6'b000010: controls <= 13'b0000000001000; // J
      default:   controls <= 13'bxxxxxxxxxxxxx; // ???
    endcase

endmodule
```

In order to add three additional instructions, the main decoder should be modified to have the corresponding control signals for those instructions. Additionally, a loadbyte signal is added into the set of the control signals and it only becomes 1 when the opcode is detected as the LBU instruction.

4.1.4 ALU Decoder

```
module aludec(input      [5:0] funct,
              input      [2:0] aluop,
              output reg [3:0] alucontrol);

  always @(*)
    case(aluop)
      3'b000: alucontrol <= 4'b0010;  // add
      3'b001: alucontrol <= 4'b0110;  // sub
      3'b010: alucontrol <= 4'b0001;  // or
      3'b011: alucontrol <= 4'b0000;  // and
      3'b100: // RTYPE
            case(funct)
               6'b100000,
               6'b100001: alucontrol <= 4'b0010; // ADD, ADDU: only difference
 is exception
               6'b100010,
               6'b100011: alucontrol <= 4'b0110; // SUB, SUBU: only difference
 is exception
               6'b100100: alucontrol <= 4'b0000; // AND
               6'b100101: alucontrol <= 4'b0001; // OR
               6'b101010: alucontrol <= 4'b0111; // SLT
               6'b100111: alucontrol <= 4'b1100; // NOR
               default:   alucontrol <= 4'bxxxx; // ???
            endcase
```

```
      default: alucontrol <= 4'bxxxx; // ???
    endcase
endmodule
```

In order to implement the NOR instruction, an additional function code NOR is implement into the R-type instruction. This generates a unique alucontrol signal which lead the ALU to perform a NOR operation.

4.1.5 ALU

```
module alu(input      [31:0] a, b,
           input      [3:0]  alucont,
           output reg [31:0] result,
           output            zero);

  wire [31:0] a2, b2, sum, slt;

  assign a2 = alucont[3] ? ~a:a;
  assign b2 = alucont[2] ? ~b:b;
  assign sum = a2 + b2 + alucont[3] + alucont[2];
  assign slt = sum[31];

  always@(*)
    case(alucont[1:0])
      2'b00: result <= a2 & b2;
      2'b01: result <= a2 | b2;
      2'b10: result <= sum;
      2'b11: result <= slt;
    endcase

  assign zero = (result == 32'b0);

endmodule
```

As mentioned in the previous section, the alucontrol signal for the NOR instruction generated from the ALU decoder is 1100. Therefore, it leads the ALU to perform a AND operation of the two input but with a NOT in front of them. By the De Morgan's law, NOT a AND NOT b equals a NOR b, and this renders the ALU perform the NOR operation equivalently. For the ANDI instruction, the ALU just AND the two inputs without a NOT in front of them.

4.1.6 Datapath

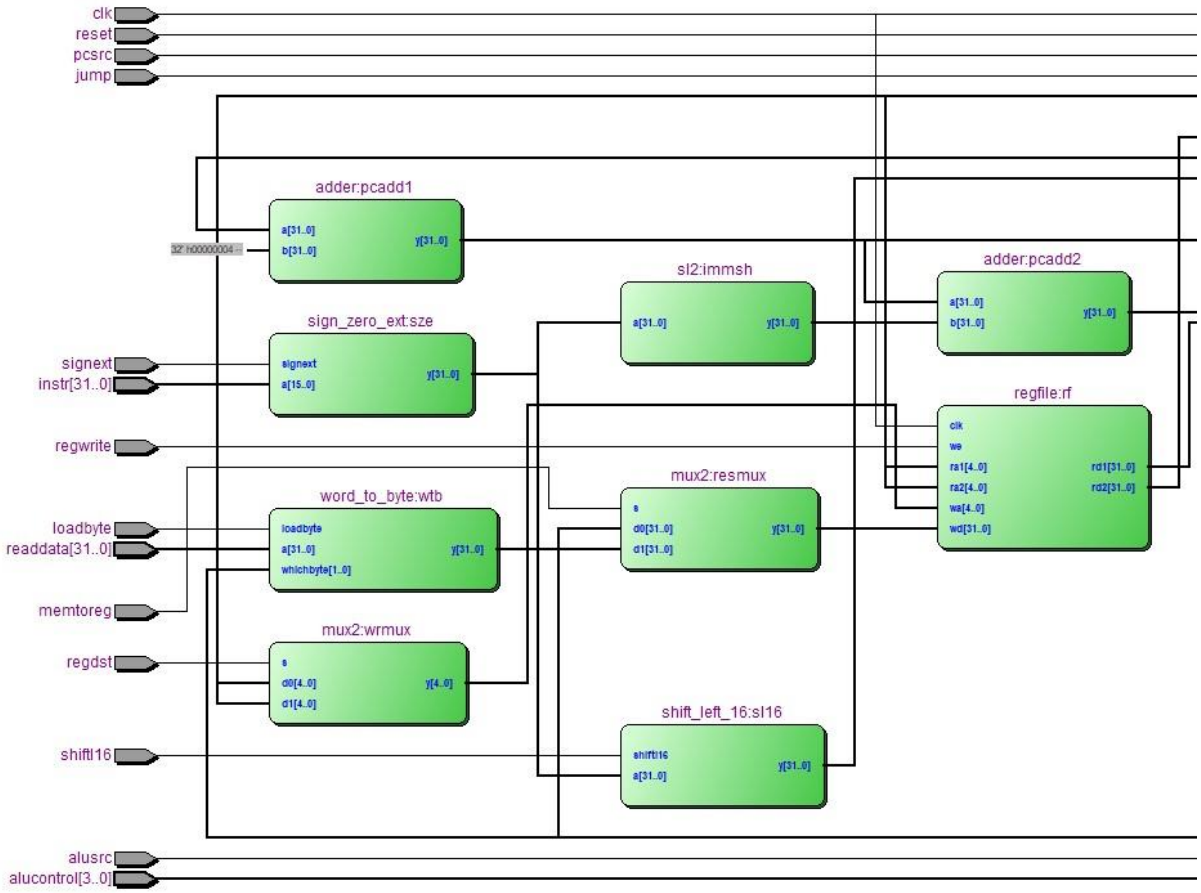

Figure 6: Modified Block Diagram

```
module datapath(input         clk, reset,
                input         signext,
                input         shiftl16,
                input         memtoreg, pcsrc,
                input         alusrc, regdst,
                input         regwrite, jump,
                input         loadbyte,
                input  [3:0]  alucontrol,
                output        zero,
                output [31:0] pc,
                input  [31:0] instr,
                output [31:0] aluout, writedata,
                input  [31:0] readdata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh, shiftedimm;
  wire [31:0] srca, srcb;
  wire [31:0] result;
```

```
wire [31:0] memdata;
wire        shift;

// next PC logic
flopr #(32) pcreg (.clk   (clk),
                   .reset (reset),
                   .d     (pcnext),
                   .q     (pc));

adder       pcadd1 (.a (pc),
                    .b (32'b100),
                    .y (pcplus4));

sl2         immsh (.a (signimm),
                   .y (signimmsh));

adder       pcadd2 (.a (pcplus4),
                    .b (signimmsh),
                    .y (pcbranch));

mux2 #(32)  pcbrmux(.d0  (pcplus4),
                    .d1  (pcbranch),
                    .s   (pcsrc),
                    .y   (pcnextbr));

mux2 #(32)  pcmux (.d0   (pcnextbr),
                   .d1   ({pcplus4[31:28], instr[25:0], 2'b00}),
                   .s    (jump),
                   .y    (pcnext));

// register file logic
regfile     rf(.clk     (clk),
               .we      (regwrite),
               .ra1     (instr[25:21]),
               .ra2     (instr[20:16]),
               .wa      (writereg),
               .wd      (result),
               .rd1     (srca),
               .rd2     (writedata));

mux2 #(5)   wrmux(.d0  (instr[20:16]),
                  .d1  (instr[15:11]),
                  .s   (regdst),
                  .y   (writereg));

mux2 #(32)  resmux(.d0 (aluout),
                   .d1 (memdata),
                   .s  (memtoreg),
```

```
                    .y  (result));

  sign_zero_ext  sze(.a       (instr[15:0]),
                     .signext (signext),
                     .y       (signimm[31:0]));


  word_to_byte   wtb(.a          (readdata),
                     .loadbyte   (loadbyte),
                     .whichbyte  (aluout[1:0]),
                     .y          (memdata));

  shift_left_16 sl16(.a         (signimm[31:0]),
                     .shiftl16  (shiftl16),
                     .y         (shiftedimm[31:0]));

  // ALU logic
  mux2 #(32)  srcbmux(.d0 (writedata),
                      .d1 (shiftedimm[31:0]),
                      .s  (alusrc),
                      .y  (srcb));

  alu         alu( .a       (srca),
                   .b       (srcb),
                   .alucont (alucontrol),
                   .result  (aluout),
                   .zero    (zero));
endmodule
```

Since the original data path is not able to implement the LBU instruction, a module called word_to_byte is added to the data path. This module receive the data from the memory and also a loadbyte signal from the control signal. If the control signal gives a loadbyte signal of 1, the module will load one byte of the data from the memory instead of the whole 32 bit data. The position of the byte will be determined by the whichbyte signal which is decided from the least two significant bits of the aluout signal which is the memory address of the data.

4.1.7 Load Byte

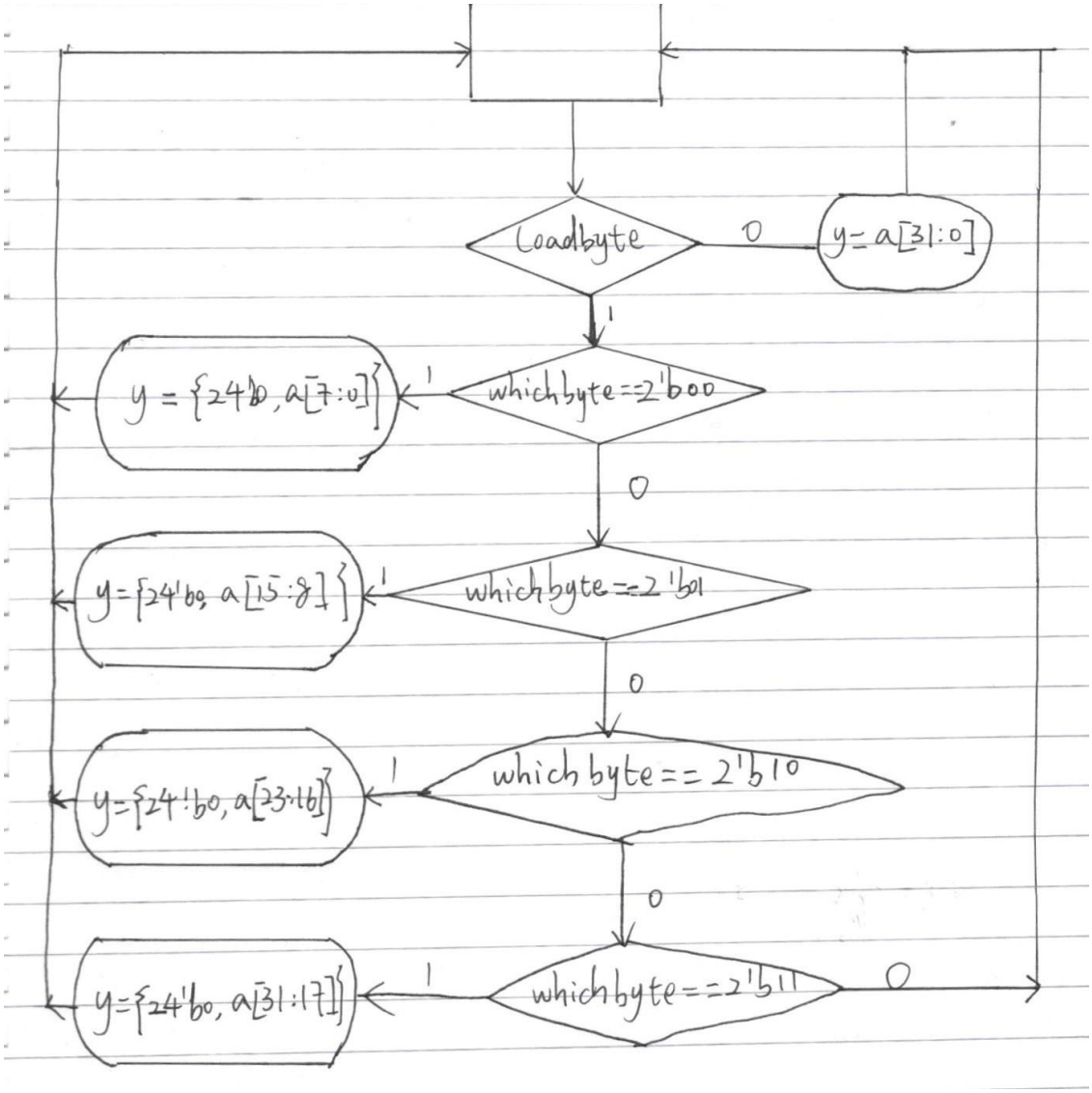

Figure 7: ASM

```
module word_to_byte(input    [31:0] a,
              input    [1:0]  whichbyte,
              input           loadbyte,
              output reg [31:0] y);

  always @(*)
  begin
    if(loadbyte)
      case(whichbyte[1:0])
        2'b00: y <= {24'b0, a[7:0]};
        2'b01: y <= {24'b0, a[15:8]};
        2'b10: y <= {24'b0, a[23:16]};
        2'b11: y <= {24'b0, a[31:24]};
      endcase
    else
```

```
      y <= a[31:0];
  end

endmodule
```

This module determines whether a byte or a work is loaded from the memory. If a loadbyte signal is detected true, the module loads a byte, and the whichbyte signal decides which of the byte within the 32-bit data is loaded.

4.2 NOR

4.2.1 Assembly Language Code

```
# loads first operand
lui $s0, 0x0000
addiu $s0, $s0, 0x0f0f
# loads second operand
lui $s1, 0x0000
addiu $s1, $s1, 0x00ff
# the address of LED_R
lui $t0, 0x0000
addiu $t0, $t0, 0x2008
# 0x0F0F nor 0x00FF
nor $s2, $s0, $s1
# LED_R to display the result
sw $s2, 0x0000($t0)
```

4.2.2 ModelSim Result

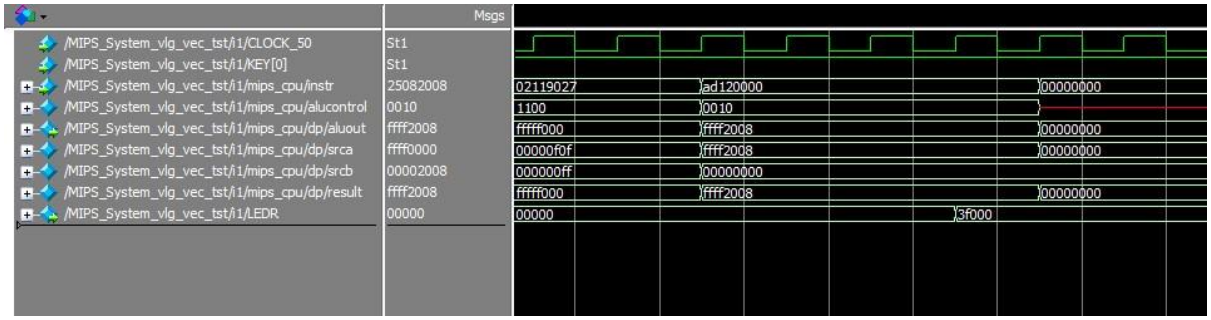

Figure 8: ModelSim Simulation Result

As shown in Figure 8, the simulation result shows that the function of the NOR instruction is working properly. It can be seen that for the clock cycle of the NOR instruction, the alucontrol signal is 1100. Meanwhile, the two inputs to the ALU is hexadecimal 0000 0F0F and 0000 00FF. Since

$$000\ 00F0F\ NOR\ 0000\ 00FF = FFFF\ F000$$

As the lower 18 bits is stored into the LEDR address, the result becomes 11 1111 0000 0000 0000, which is 3f000 as shown in the figure.

4.3 ANDI

4.3.1 Assembly Language Code

```
# loads first operand
lui $s0, 0x0000
addiu $s0, $s0, 0x0f0f
# the address of LED_R
lui $t0, 0xffff
addiu $t0, $t0, 0x2008
# 0x0F0F and 0x00FF
andi $s2, $s0, 0x00ff
# LED_R to display the result
sw $s2, 0x0000($t0)
```

4.3.2 ModelSim Result

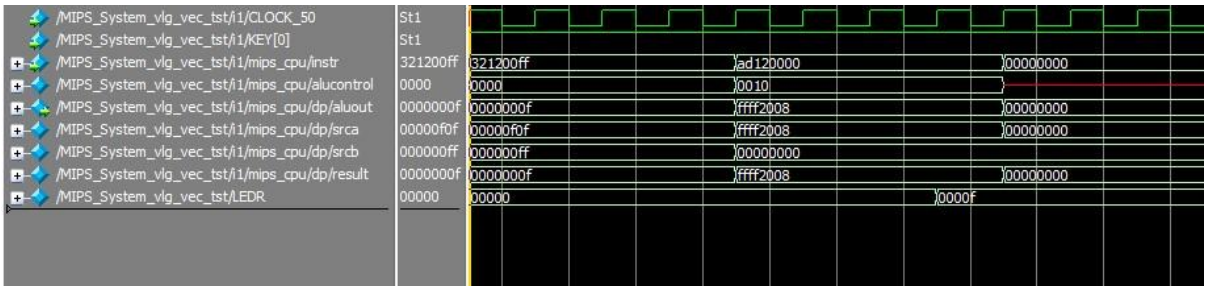

Figure 9: ModelSim Simulation Result

This instruction conducts an AND operation between 00000F0F and 000000FF. Since

$$0000\ 0F0F\ AND\ 0000\ 00FF = 0000\ 000F$$

Hence, the result shown on the LEDR is a 18 bits number which is 0000F in hexadecimal.

4.4 LBU

4.4.1 Assembly Language Code

```
# loads first operand
lui $s0, 0x0000
addiu $s0, $s0, 0x5500
# a temporary memory address
lui $t0, 0x0000
addiu $t0, $t0, 0x2000
# the address of LED_R
lui $t1, 0xffff
addiu $t1, $t1, 0x2008
```

```
# stores 0x0000 5500 to mem(0x0000 2000)
sw $s0, 0x0000($t0)
# loads the second byte(0x55) from mem(0x0000 2000)
lbu $s1, 0x0001($t0)
# stores 0x0055 to the memory of LED_R
sw $s1, 0x0000($t1)
```

4.4.2 Additional Pathways Diagram

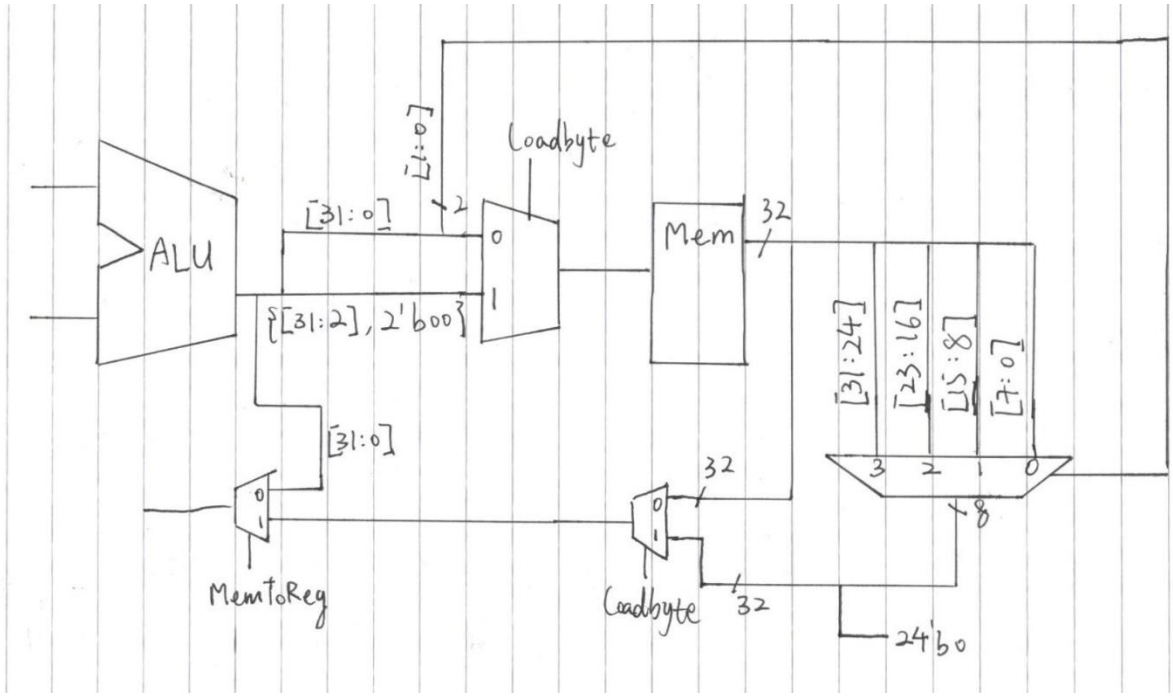

Figure 10: Block Diagram of Additional Pathways

As shown in the Block Diagram, one of the additional pathways is added between the ALU and the memory. Data will be separated as one full 32 bits data and one sliced 32 bits data with 2 bits of zeros replacing the least two significant bits. The original least two significant two bits is routed to the other additional mux to determine which byte of the data should be loaded. The loadbyte signal will determine whether the byte data is loaded to the register or rather a full 32 bit one.