UNIVERSITY OF LIVERPOOL

Digital Systems Design

Assignment 3: MIPS Processor

Abstract

This study aims to delve into the functionality and expansion of synthesized MIPS single-cycle processors. Initially, by crafting simple programs to control the processor, the display of student identification numbers was achieved. Subsequently, by extending the processor to execute additional instructions, including nor, xori, and lh instructions, and testing them using SignalTap. The findings indicate a significant enhancement in the functionality and flexibility of the MIPS processor through these expansions. This research provides profound insights into the understanding and extension of MIPS processors, offering valuable references for future processor designs and optimizations.

Declaration

Contents

1 Introduction

The MIPS (Microprocessor without Interlocked Pipeline Stages) architecture stands as a prominent figure in the realm of computer processors, renowned for its efficiency, simplicity, and versatility. Since its inception, MIPS has undergone significant developments, evolving from its initial 32-bit Reduced Instruction Set Computing (RISC) architecture to encompass a diverse array of applications spanning from embedded systems to high-performance computing.

Part A of this study involves familiarizing oneself with the synthesized MIPS single-cycle processor and its underlying architecture. The single-cycle processor, characterized by its straightforward execution of each instruction in a single clock cycle, serves as an ideal starting point for understanding the fundamentals of MIPS architecture and instruction principles. By crafting simple programs, such as those to display student identification numbers, students gain hands-on experience in controlling the processor through assembly language instructions.

Part B extends the exploration by augmenting the processor's capabilities to execute additional instructions beyond the standard MIPS instruction set. This expansion involves incorporating instructions like nor (bitwise NOT OR), xori (exclusive OR immediate), and lh (load halfword) into the processor's repertoire. Furthermore, SignalTap is employed to conduct thorough testing, ensuring the seamless integration and proper functioning of these augmented instructions within the MIPS processor framework.

Throughout this study, a comprehensive understanding of MIPS architecture, instruction principles, and processor design is cultivated. By delving into both Part A and Part B, students not only gain insights into the foundational concepts of MIPS processors but also acquire the skills necessary to extend their functionalities, thereby contributing to the ongoing advancements in processor design and optimization.

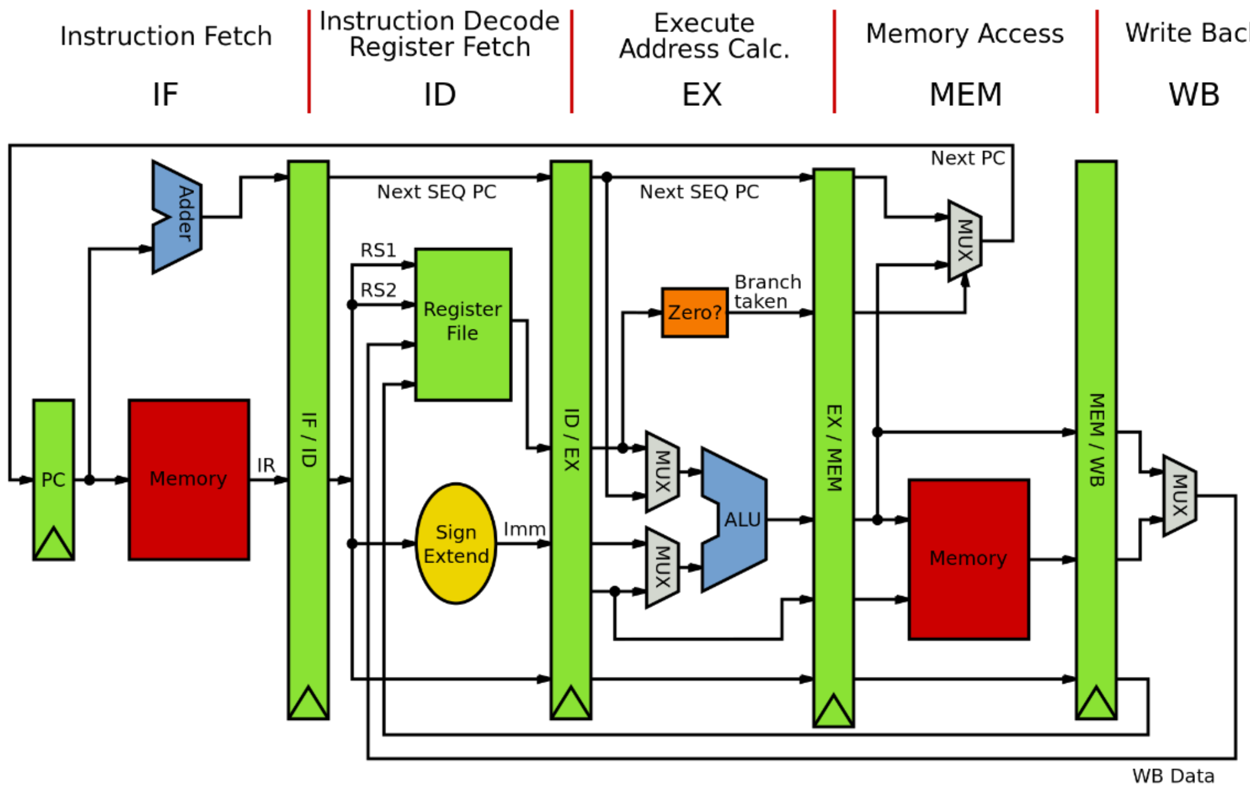
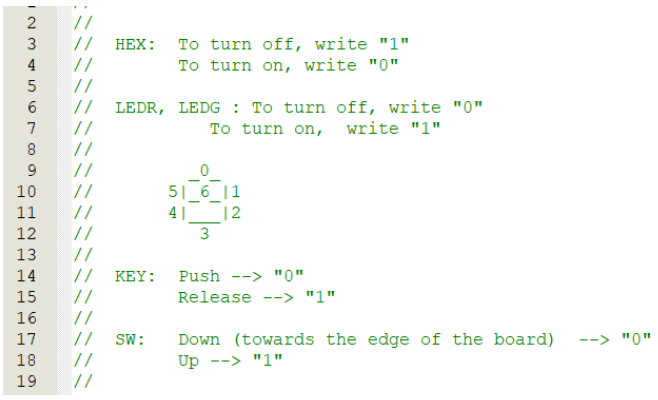


Fig.1. Pipeline MIPS Architecture [1]

2 Part A

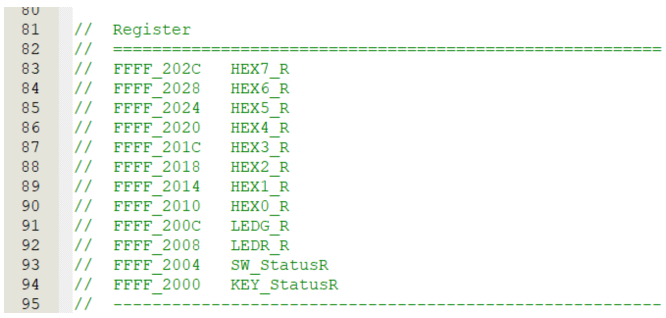
2.1 Task

The task of this part is to control eight 7-segment displays using MIPS assembly instructions and to display the school number on the DE2 board. We begin by observing how the seven-segment display is displayed in the GPIO interface and the address that controls the seven-segment display. They are shown below.



```
//
//  HEX:  To turn off, write "1"
//        To turn on, write "0"
//
//  LEDR, LEDG : To turn off, write "0"
//            To turn on,  write "1"
//
//          _0_
//        5|_6_|1
//        4|___|2
//           3
//
//  KEY:  Push --> "0"
//        Release --> "1"
//
//  SW:   Down (towards the edge of the board)  --> "0"
//        Up --> "1"
//
```

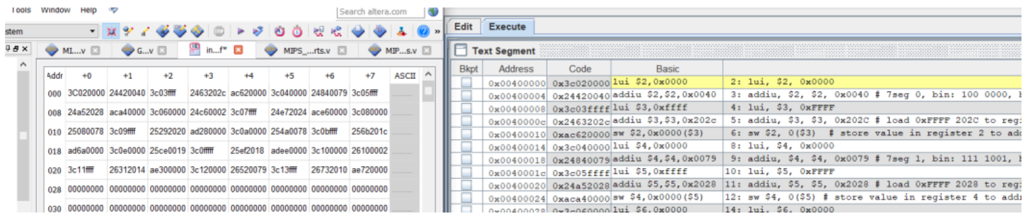
Fig.2. 7-segment



```
//  Register
//  ======================================================
//  FFFF_202C   HEX7_R
//  FFFF_2028   HEX6_R
//  FFFF_2024   HEX5_R
//  FFFF_2020   HEX4_R
//  FFFF_201C   HEX3_R
//  FFFF_2018   HEX2_R
//  FFFF_2014   HEX1_R
//  FFFF_2010   HEX0_R
//  FFFF_200C   LEDG_R
//  FFFF_2008   LEDR_R
//  FFFF_2004   SW_StatusR
//  FFFF_2000   KEY_StatusR
//  ------------------------------------------------------
```

Fig.3. Register address of 7 segment

Now all we need to do is enter the instructions to control the 7-segment display into the register memory of each 7-segment display to control the 7-segment display to display the specified content. For example, if we want the 7-segment display to show the number 7, first, observe how the 7-segment display shows the number 7, which means that the number 0, 1, and 2 LEDs are on, and all other LEDs are off. In binary, it is 111 1000, that is, 78 in hexadecimal it is arranged from MSB to LSB, with 1 representing off and 0 representing on. 7-segment display [6:0], we only need to let the specified led light up to display the number we need. And so on, we can make 8 7-segment displays to show our school number. Therefore, we can illuminate the 7-segment display by storing the specified hexadecimal number into the memory address of the specified 7-segment display in the register. We can use MIPS assembly to do this step. the MIPS assembly code is explained in detail in the picture below. In a MIPS processor design or simulation, a MIF file can be used to initialize memory, such as instruction memory. So, we can convert the instructions we need from MIPS assembly to hexadecimal instruction code.



```
Addr  +0        +1        +2        +3        +4        +5        +6        +7        ASCII
000   3C020000  24420040  3c03ffff  2463202c  ac620000  3c040000  24840079  3c05ffff
008   24a52028  aca40000  3c060000  24c60002  3c07ffff  24e72024  ace60000  3c080000
010   25080078  3c09ffff  25292020  ad280000  3c0a0000  254a0078  3c0bffff  256b201c
018   ad6a0000  3c0e0000  25ce0019  3c0fffff  25ef2018  adee0000  3c100000  26100002
020   3c11ffff  26312014  ae300000  3c120000  26520079  3c13ffff  26732010  ae720000
028   00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
030   00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
```

```
Text Segment
Bkpt  Address     Code        Basic
      0x00400000  0x3c020000  lui $2,0x0000           2: lui, $2, 0x0000
      0x00400004  0x24420040  addiu $2,$2,0x0040      3: addiu, $2, $2, 0x0040 # 7seg 0, bin: 100 0000, h
      0x00400008  0x3c03ffff  lui $3,0xffff           4: lui, $3, 0xFFFF
      0x0040000c  0x2463202c  addiu $3,$3,0x202c      5: addiu, $3, $3, 0x202C # load 0xFFFF 202C to regi
      0x00400010  0xac620000  sw $2,0x0000($3)        6: sw $2, 0($3)  # store value in register 2 to add
      0x00400014  0x3c040000  lui $4,0x0000           8: lui, $4, 0x0000
      0x00400018  0x24840079  addiu $4,$4,0x0079      9: addiu, $4, $4, 0x0079 # 7seg 1, bin: 111 1001, h
      0x0040001c  0x3c05ffff  lui $5,0xffff          10: lui, $5, 0xFFFF
      0x00400020  0x24a52028  addiu $5,$5,0x2028     11: addiu, $5, $5, 0x2028 # load 0xFFFF 2028 to regi
      0x00400024  0xaca40000  sw $4,0x0000($5)       12: sw $4, 0($5) # store value in register 4 to addr
      0x00400028  0x3c060000  lui $6,0x0000          14: lui, $6, 0x0000
```

Fig.4. Instruction in .mif file

2.2 Assembly

```
# ID: 201677461, Jinsong Zhang, last 8 bits: 0167 7461
lui, $2, 0x0000
addiu, $2, $2, 0x0040 # 7seg 0, bin: 100 0000, hex: 40, load 0x0000 0040 to register 2
lui, $3, 0xFFFF
addiu, $3, $3, 0x202C # load 0xFFFF 202C to register 3
sw $2, 0($3)  # store value in register 2 to address in register 3

lui, $4, 0x0000
addiu, $4, $4, 0x0079 # 7seg 1, bin: 111 1001, hex: 79, load 0x0000 0079 to register 4
lui, $5, 0xFFFF
addiu, $5, $5, 0x2028 # load 0xFFFF 2028 to register 5
sw $4, 0($5) # store value in register 4 to address in register 5

lui, $6, 0x0000
addiu, $6, $6, 0x0002 # 7seg 6, bin: 000 0010, hex: 2, load 0x0000 0002 to register 6
lui, $7, 0xFFFF
addiu, $7, $7, 0x2024 # load 0xFFFF 2024 to register 7
sw $6, 0($7) # store value in register 6 to address in register 7

lui, $8, 0x0000
addiu, $8, $8, 0x0078 # 7seg 7, bin: 111 1000, hex: 78, load 0x0000 0078 to register 8
lui, $9, 0xFFFF
addiu, $9, $9, 0x2020 # load 0xFFFF 2020 to register 9
sw $8, 0($9) # store value in register 8 to address in register 8

lui, $10, 0x0000
addiu, $10, $10, 0x0078 # 7seg 7, bin: 111 1000, hex: 78, load 0x0000 0078 to register 10
lui, $11, 0xFFFF
addiu, $11, $11, 0x201C # load 0xFFFF 201c to register 11
sw $10, 0($11) # store value in register 10 to address in register 11

lui, $14, 0x0000
addiu, $14, $14, 0x0019 # 7seg 4, bin: 001 1001, hex: 19, load 0x0000 0019 to register 14
lui, $15, 0xFFFF
addiu, $15, $15, 0x2018 # load 0xFFFF 2018 to register 15
sw $14, 0($15) # store value in register 14 to address in register 15

lui, $16, 0x0000
addiu, $16, $16, 0x0002 # 7seg 6, bin: 000 0010, hex: 2, load 0x0000 0002 to register 16
lui, $17, 0xFFFF
addiu, $17, $17, 0x2014 # load 0xFFFF 2014 to register 17
sw $16, 0($17) # store value in register 16 to address in register 17

```

```

lui, $18, 0x0000
addiu, $18, $18, 0x0079 # 7seg 1, bin: 111 1001, hex: 79, load 0x0000 0079 to register 18
lui, $19, 0xFFFF
addiu, $19, $19, 0x2010 # load 0xFFFF 2010 to register 19
sw $18, 0($19) # store value in register 18 to address in register 19
```

2.3 Signal Tap

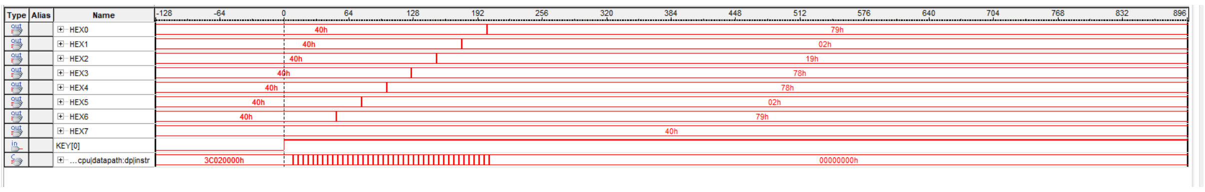


Fig.5. Signal Tap of 7-segment in 01677461

We can find that the seven 7 segments show exactly what we expected. After executing the instruction, the number we need is stored in the corresponding address. That is, 01677461. The 7-segment display is: 0x40 (0), 0x79 (1), 0x2 (6), 0x78 (7), 0x78 (7), 0x19 (4), 0x2 (6), 0x79 (1). This is also exactly what we expected earlier.

2.4 Screenshot

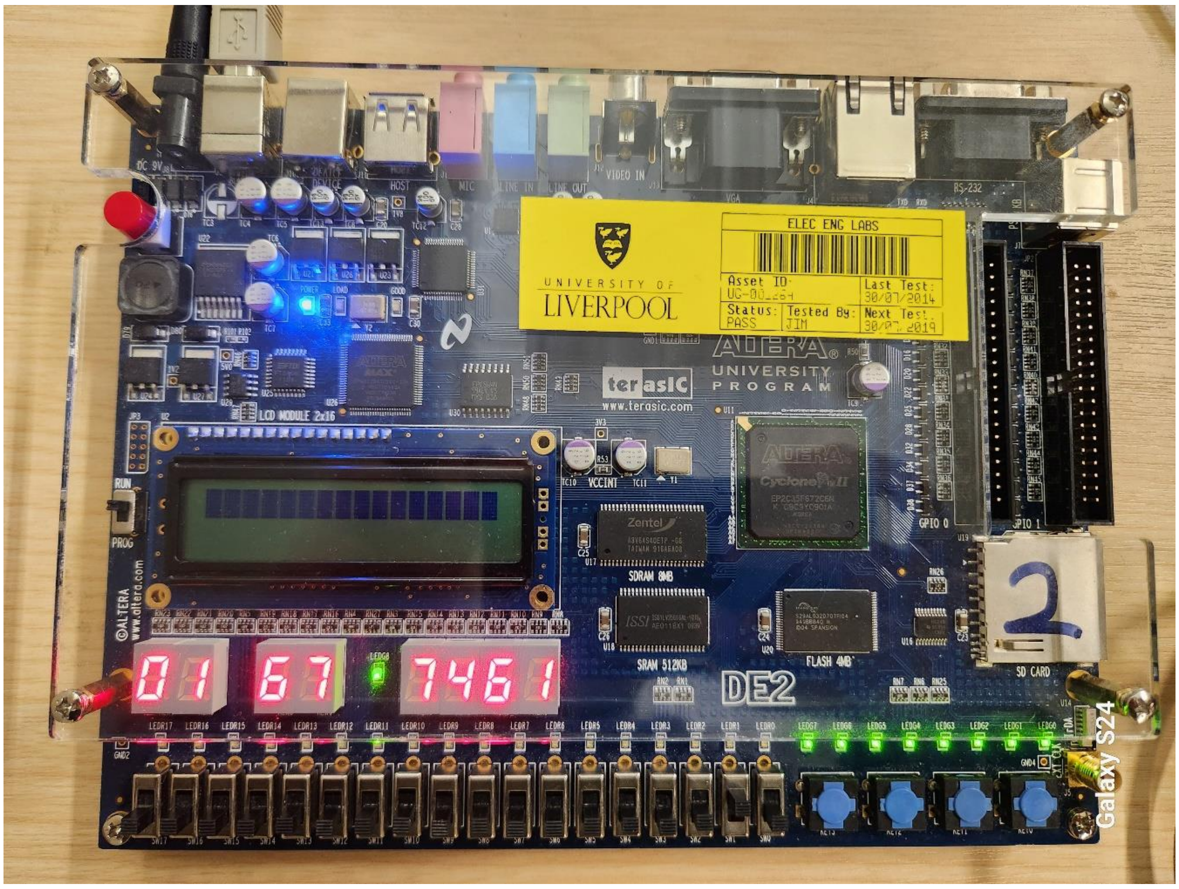
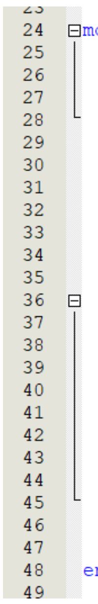


Fig.6. ID: 01677461 on DE 2 board

3 Part B

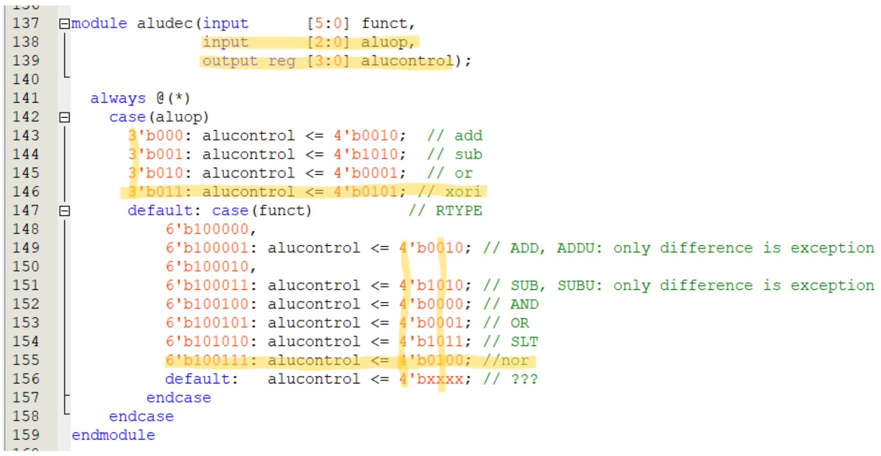
3.1 NOR Instruction

3.1.1 Verilog Code



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

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

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

  always@(*)
    case(alucont[2:0])
      3'b000: result <= a & b;
      3'b001: result <= a | b;
      3'b010: result <= sum;
      3'b011: result <= slt;
      3'b100: result <= ~(a | b); // nor
      3'b101: result <= a ^ b; // xor
      default: result <= 32'b0;
    endcase

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

endmodule
```

Fig.7. ALU module

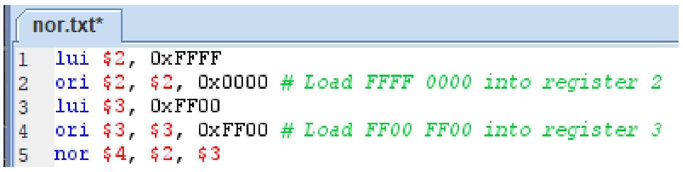


```verilog
module aludec(input       [5:0] funct,
              input       [2:0] aluop,
              output reg [3:0] alucontrol);

  always @(*)
    case(aluop)
      3'b000: alucontrol <= 4'b0010;  // add
      3'b001: alucontrol <= 4'b1010;  // sub
      3'b010: alucontrol <= 4'b0001;  // or
      3'b011: alucontrol <= 4'b0101; // xori
      default: case(funct)           // RTYPE
          6'b100000,
          6'b100001: alucontrol <= 4'b0010; // ADD, ADDU: only difference is exception
          6'b100010,
          6'b100011: alucontrol <= 4'b1010; // SUB, SUBU: only difference is exception
          6'b100100: alucontrol <= 4'b0000; // AND
          6'b100101: alucontrol <= 4'b0001; // OR
          6'b101010: alucontrol <= 4'b1011; // SLT
          6'b100111: alucontrol <= 4'b0100; //nor
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

Fig.8. ALU dec module

For the nor instruction, we only need to change the alu module and the alu decoder module. In the alu module, alucont has only 3 bits, which means that only 4 instructions can be represented. But we need more instructions. So we add one bit to make it 4 bits so that we can add as many nor instructions as we need. Here we set the nor instruction to 3'b100. when we come to the alu decoder module, we also need to add a nor instruction to the R type case to decode the function. By looking at the function code of the MIPS instruction, we can see that the function code of the nor instruction is 100111, so we can just add it. Similarly, in the alu module, we set the nor instruction to 100, so it should be 100 after decoding here.

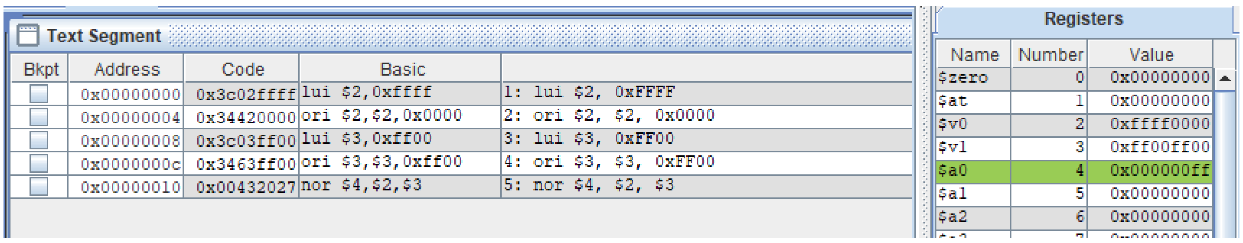
3.1.2 Assembly Code



nor.txt*

```
lui $2, 0xFFFF
ori $2, $2, 0x0000 # Load FFFF 0000 into register 2
lui $3, 0xFF00
ori $3, $3, 0xFF00 # Load FF00 FF00 into register 3
nor $4, $2, $3
```

Fig.9. MIPS assembly code for nor



Text Segment

Bkpt	Address	Code	Basic	
☐	0x00000000	0x3c02ffff	lui $2,0xffff	1: lui $2, 0xFFFF
☐	0x00000004	0x34420000	ori $2,$2,0x0000	2: ori $2, $2, 0x0000
☐	0x00000008	0x3c03ff00	lui $3,0xff00	3: lui $3, 0xFF00
☐	0x0000000c	0x3463ff00	ori $3,$3,0xff00	4: ori $3, $3, 0xFF00
☐	0x00000010	0x00432027	nor $4,$2,$3	5: nor $4, $2, $3

Registers

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0xffff0000
$v1	3	0xff00ff00
$a0	4	0x000000ff
$a1	5	0x00000000
$a2	6	0x00000000

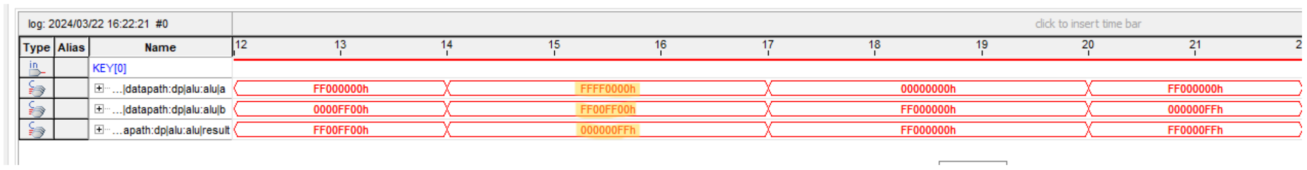
Fig.10. MIPS assembly code for nor (execute)

A	B	Result
0	0	1
0	1	0
1	0	0
1	1	0

Table.1. Ture table for nor gate

All we need to do to assign a value to a register is to use lui to give a value to the high bit of the register and ori to give a value to the low bit of the register. Then we use the nor instruction to perform a logical nor operation on the values in these two registers and store them in register 4. We can see that in the compiler, after executing these instructions, the value in register 4 will be 0x0000 00FF, which is logical. Because 0xFFFF 0000 and 0xFF00 FF00 after the nor operation the correct result is 0x0000 00FF.

3.1.3 Signal Tap



```
log: 2024/03/22 16:22:21  #0                                                                   click to insert time bar
Type Alias  Name                          12        13        14        15        16        17        18        19        20        21        2
            KEY[0]
            ...|datapath:dp|alu:alu|a        FF000000h             FFFF0000h              00000000h             FF000000h
            ...|datapath:dp|alu:alu|b        0000FF00h             FF00FF00h              FF000000h             000000FFh
            ...apath:dp|alu:alu|result       FF00FF00h             000000FFh              FF000000h             FF0000FFh
```

Fig.11. Signal Tap for nor instruction

We can see that at the end of the assignment to the register we get the correct result. I.e. 0xFFFF 0000 and 0xFF00 FF00 after the nor operation the correct result is 0x0000 00FF. so Signal Tap is also as we expected.

3.1.4 ASM

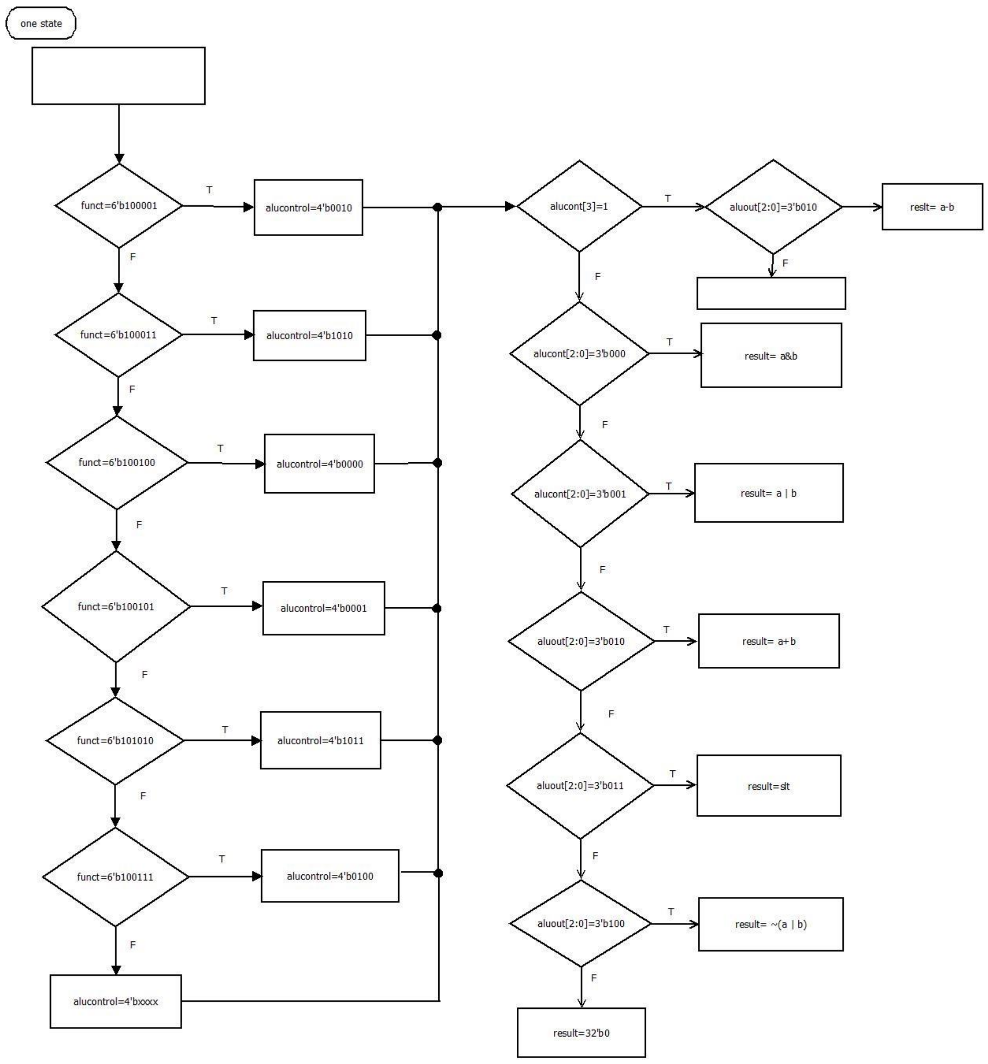
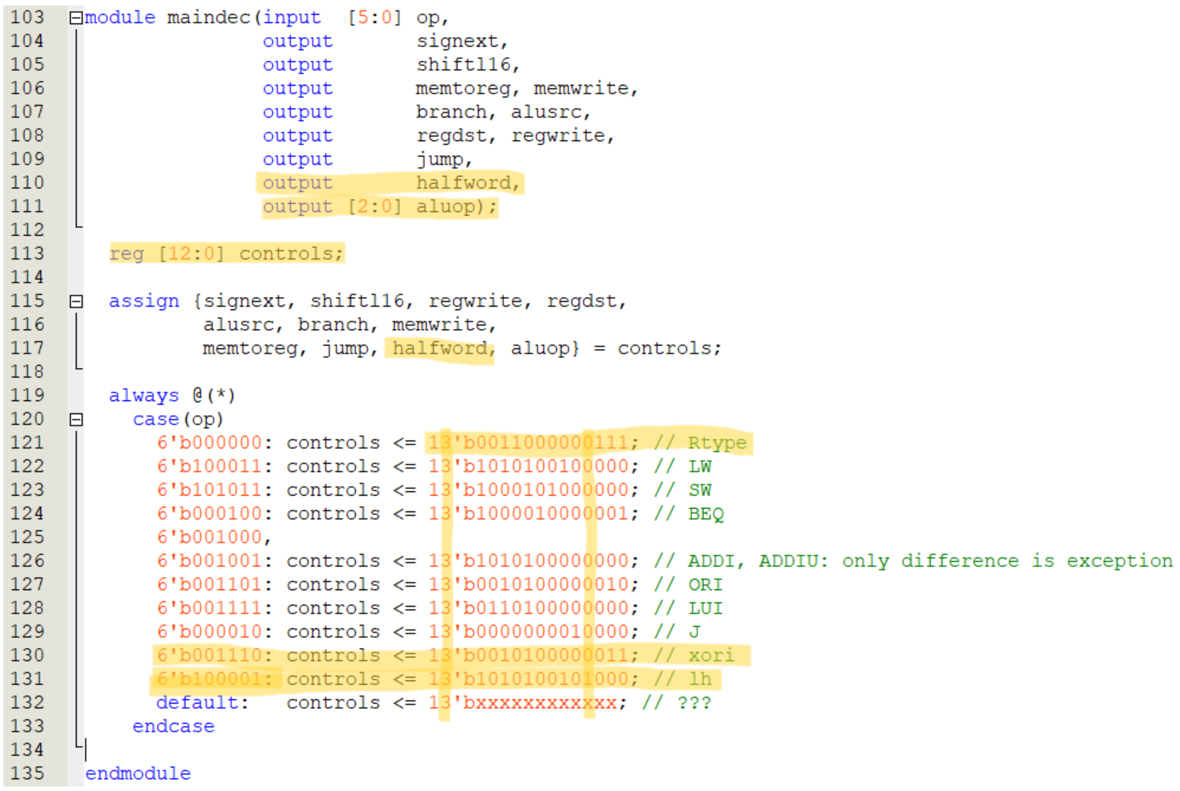


Fig.12. ASM for nor instruction

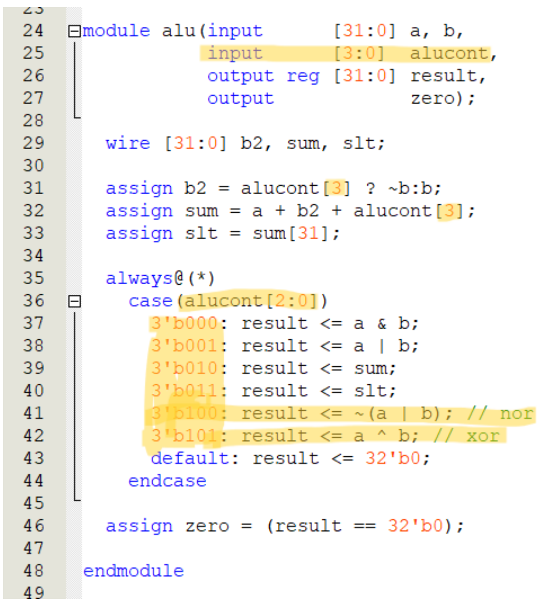
3.2 XORI Instruction

3.2.1 Verilog Code



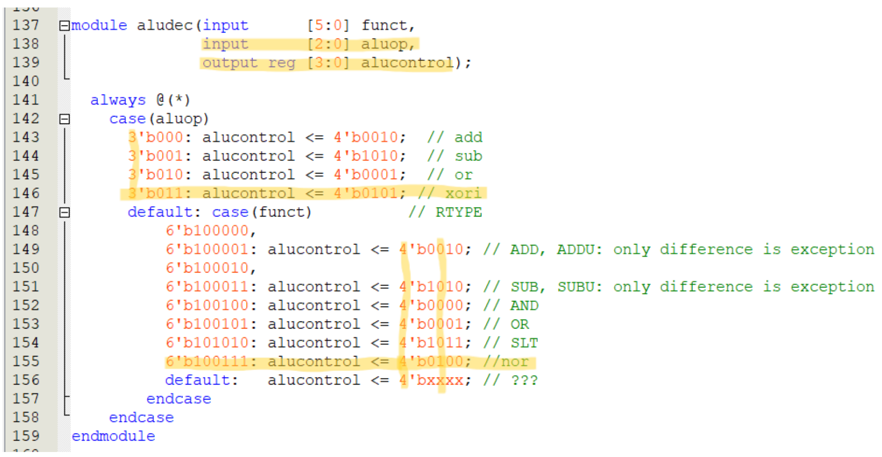
```
module maindec(input  [5:0] op,
               output       signext,
               output       shiftl16,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
               output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

  reg [12:0] controls;

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

  always @(*)
    case(op)
      6'b000000: controls <= 13'b0011000000111; // Rtype
      6'b100011: controls <= 13'b1010100100000; // LW
      6'b101011: controls <= 13'b1000101000000; // SW
      6'b000100: controls <= 13'b1000010000001; // BEQ
      6'b001000,
      6'b001001: controls <= 13'b1010100000000; // ADDI, ADDIU: only difference is exception
      6'b001101: controls <= 13'b0010100000010; // ORI
      6'b001111: controls <= 13'b0110100000000; // LUI
      6'b000010: controls <= 13'b0000000010000; // J
      6'b001110: controls <= 13'b0010100000011; // xori
      6'b100001: controls <= 13'b1010100101000; // lh
      default:   controls <= 13'bxxxxxxxxxxxxx; // ???
    endcase

endmodule
```

Fig.13. Main decoder module



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

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

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

  always@(*)
    case(alucont[2:0])
      3'b000: result <= a & b;
      3'b001: result <= a | b;
      3'b010: result <= sum;
      3'b011: result <= slt;
      3'b100: result <= ~(a | b); // nor
      3'b101: result <= a ^ b; // xor
      default: result <= 32'b0;
    endcase

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

endmodule
```

Fig.14. ALU module



```
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'b1010;  // sub
      3'b010: alucontrol <= 4'b0001;  // or
      3'b011: alucontrol <= 4'b0101; // xori
      default: case(funct)          // RTYPE
          6'b100000,
          6'b100001: alucontrol <= 4'b0010; // ADD, ADDU: only difference is exception
          6'b100010,
          6'b100011: alucontrol <= 4'b1010; // SUB, SUBU: only difference is exception
          6'b100100: alucontrol <= 4'b0000; // AND
          6'b100101: alucontrol <= 4'b0001; // OR
          6'b101010: alucontrol <= 4'b1011; // SLT
          6'b100111: alucontrol <= 4'b0100; //nor
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

Fig.15. ALU decoder module

For the xori instruction, we need to add a new instruction in the ALU module and include logic for the xor operation. We'll set its opcode as 3b'101. Then, in the ALU decoder, we'll set it as 3'b011. Similarly, after decoding, it should be 101 in the ALU module. Since we don't need the operation from the first bit, we set the highest bit as 0. That means after decoding, it's 4'b0101. Additionally, because xori is an I-type instruction, we need to add decoding for xori in the main decoder. By consulting the opcode for xori in the I-type instructions, we know it's 001110, so in the main decoder, we use 001110 as its opcode. Then, we decode it as 13'b0010100000011. The interpretation of this is explained below:

signext: Sign-extend the immediate value.

shiftl16: Shift the immediate value left by 16 bits.

regwrite: Write to registers.

regdst: Register destination selection.

alusrc: ALU's second operand selection.

branch: Branching.

memwrite: Write to memory.

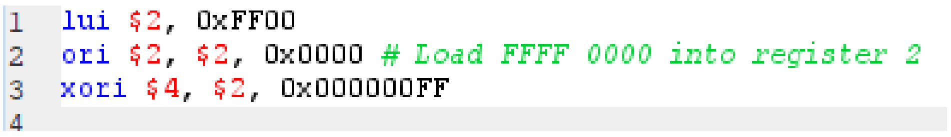
memtoreg: Memory to register.

jump: Jump.

halfword: Load half-word.

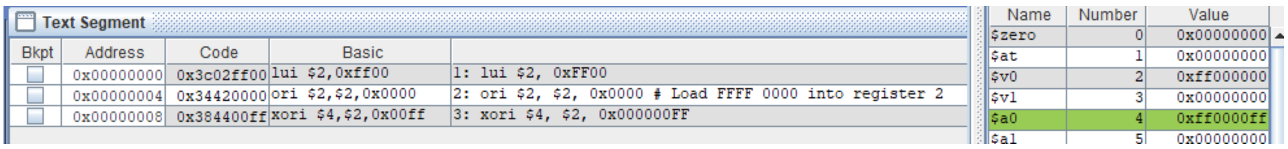
aluop: ALU operation code.

3.2.2 Assembly Code



```
lui $2, 0xFF00
ori $2, $2, 0x0000 # Load FFFF 0000 into register 2
xori $4, $2, 0x000000FF
```

Fig.16. Assembly code for xori



Text Segment

Bkpt	Address	Code	Basic	
☐	0x00000000	0x3c02ff00	lui $2,0xff00	1: lui $2, 0xFF00
☐	0x00000004	0x34420000	ori $2,$2,0x0000	2: ori $2, $2, 0x0000 # Load FFFF 0000 into register 2
☐	0x00000008	0x384400ff	xori $4,$2,0x00ff	3: xori $4, $2, 0x000000FF

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0xff000000
$v1	3	0x00000000
$a0	4	0xff0000ff
$a1	5	0x00000000

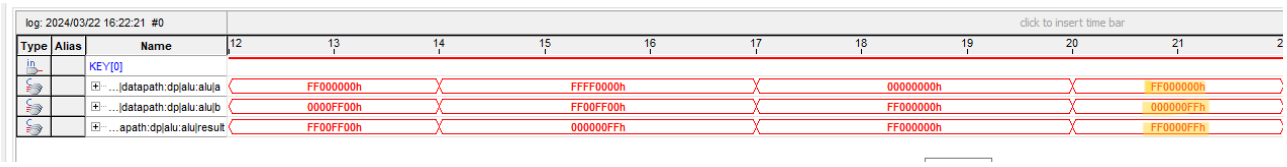
Fig.17. Assembly code for xori (execute)

A	B	Result
0	0	0
0	1	1
1	0	1
1	1	0

Table.2. Ture table for xor gate

Because xori is an operation on immediate numbers. So for assembly code we just load a value into a register and do the xori operation with an immediate number against the value in the register to test it. We start by storing 0xFF00 0000 into register 2. Then we can do the xori operation with the immediate number 0x0000 00FF. According to the above truth table and the execution result, we can find that the result of the operation is 0xFF00 00FF.

3.2.3 Signal Tap

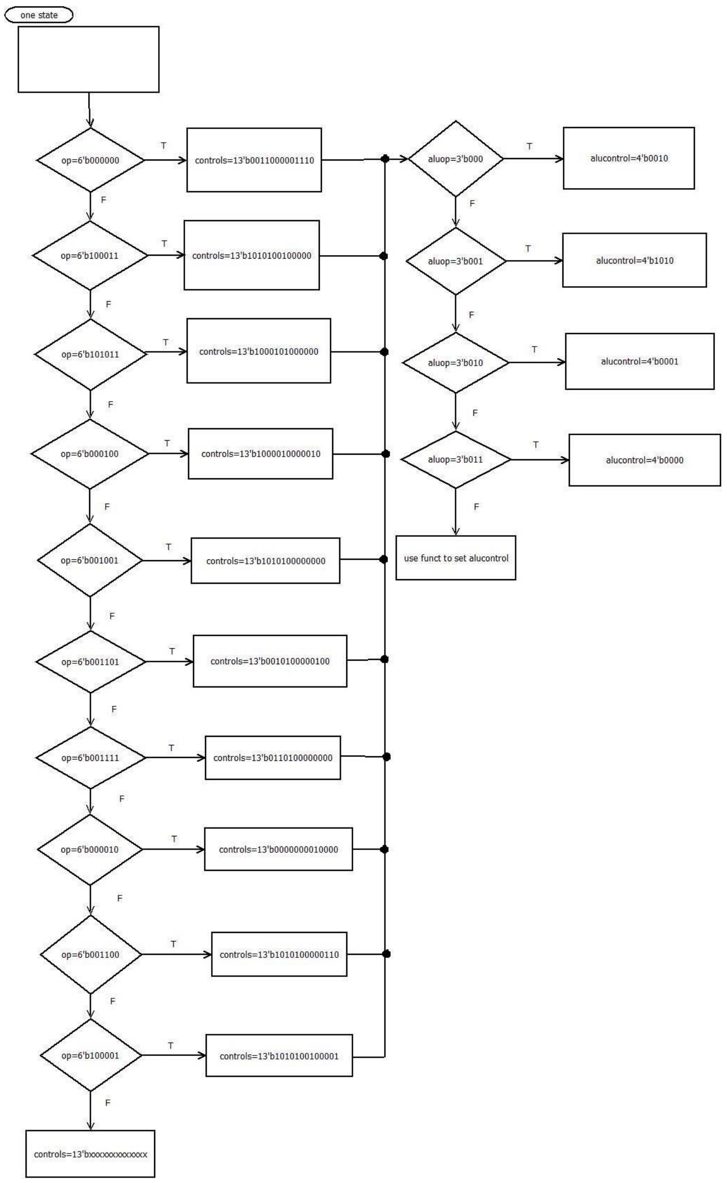


```
log: 2024/03/22 16:22:21  #0                                                             click to insert time bar
Type Alias Name                      12        13        14        15        16        17        18        19        20        21        2
           KEY[0]
           ...|datapath:dp|alu:alu|a        FF000000h             FFFF0000h                  00000000h                 FF000000h
           ...|datapath:dp|alu:alu|b        0000FF00h             FF00FF00h                  FF000000h                 000000FFh
           ...apath:dp|alu:alu|result       FF00FF00h             000000FFh                  FF000000h                 FF0000FFh
```

Fig.18. Signal Tap for xori instruction

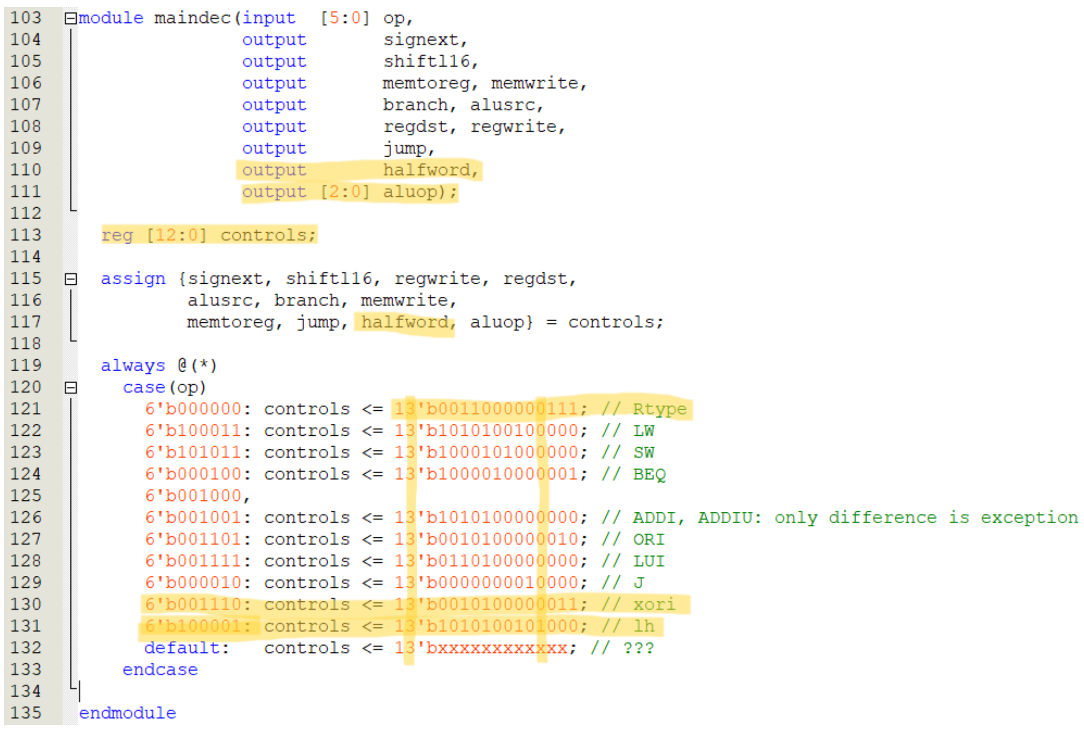
In the signal Tap, we can observe that the computation result meets our expectations. Specifically, the result of the operation between 0xFF00 0000 and 0x0000 00FF is indeed 0xFF00 00FF. This result also aligns with our previous analysis.

3.2.4 ASM


one state
op=6'b000000
T
controls=13'b0011000001110
F
op=6'b100011
T
controls=13'b1010100100000
F
op=6'b101011
T
controls=13'b1000101000000
F
op=6'b000100
T
controls=13'b1000010000010
F
op=6'b001001
T
controls=13'b1010100000000
F
op=6'b001101
T
controls=13'b0010100000100
F
op=6'b001111
T
controls=13'b0110100000000
F
op=6'b000010
T
controls=13'b0000000010000
F
op=6'b001100
T
controls=13'b1010100000110
F
op=6'b100001
T
controls=13'b1010100100001
F
controls=13'bxxxxxxxxxxxxx
aluop=3'b000
T
alucontrol=4'b0010
F
aluop=3'b001
T
alucontrol=4'b1010
F
aluop=3'b010
T
alucontrol=4'b0001
F
aluop=3'b011
T
alucontrol=4'b0000
F
use funct to set alucontrol

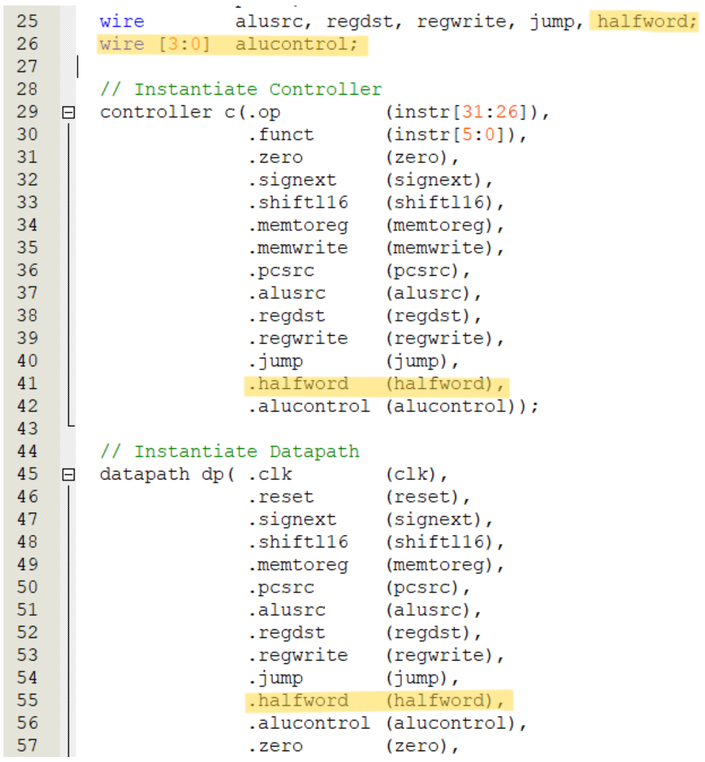
3.3 LH Instruction

3.3.1 Verilog Code

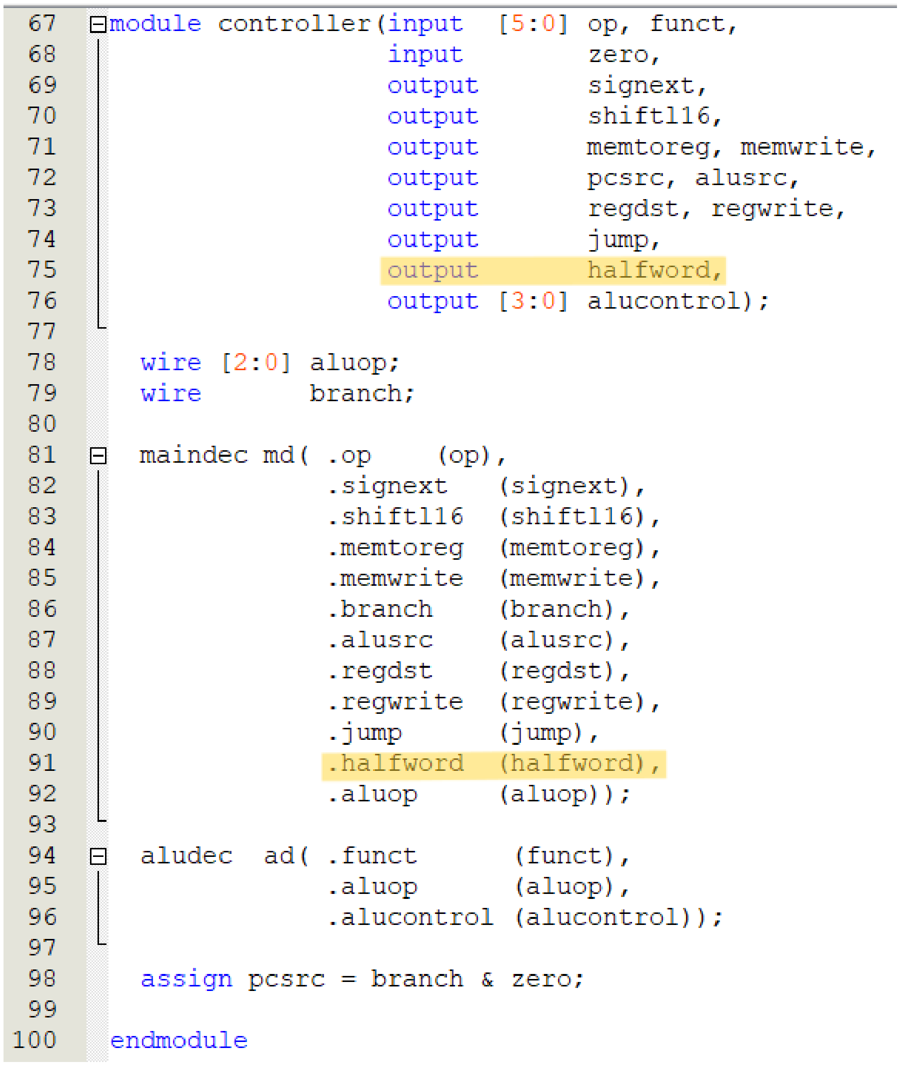


```verilog
module maindec(input  [5:0] op,
               output       signext,
               output       shiftl16,
               output       memtoreg, memwrite,
               output       branch, alusrc,
               output       regdst, regwrite,
               output       jump,
          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

  reg [12:0] controls;

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

  always @(*)
    case(op)
      6'b000000: controls <= 13'b0011000000111; // Rtype
      6'b100011: controls <= 13'b1010100100000; // LW
      6'b101011: controls <= 13'b1000101000000; // SW
      6'b000100: controls <= 13'b1000010000001; // BEQ
      6'b001000,
      6'b001001: controls <= 13'b1010100000000; // ADDI, ADDIU: only difference is exception
      6'b001101: controls <= 13'b0010100000010; // ORI
      6'b001111: controls <= 13'b0110100000000; // LUI
      6'b000010: controls <= 13'b0000000010000; // J
      6'b001110: controls <= 13'b0010100000011; // xori
      6'b100001: controls <= 13'b1010100101000; // lh
      default:   controls <= 13'bxxxxxxxxxxxxx; // ???
    endcase

endmodule
```

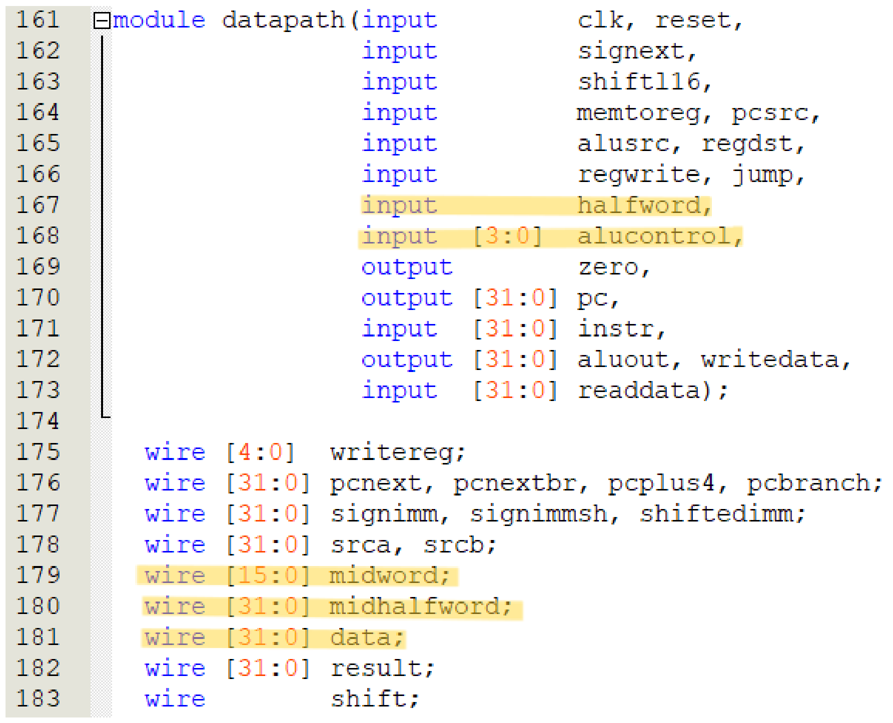
Fig.20. Main decoder module



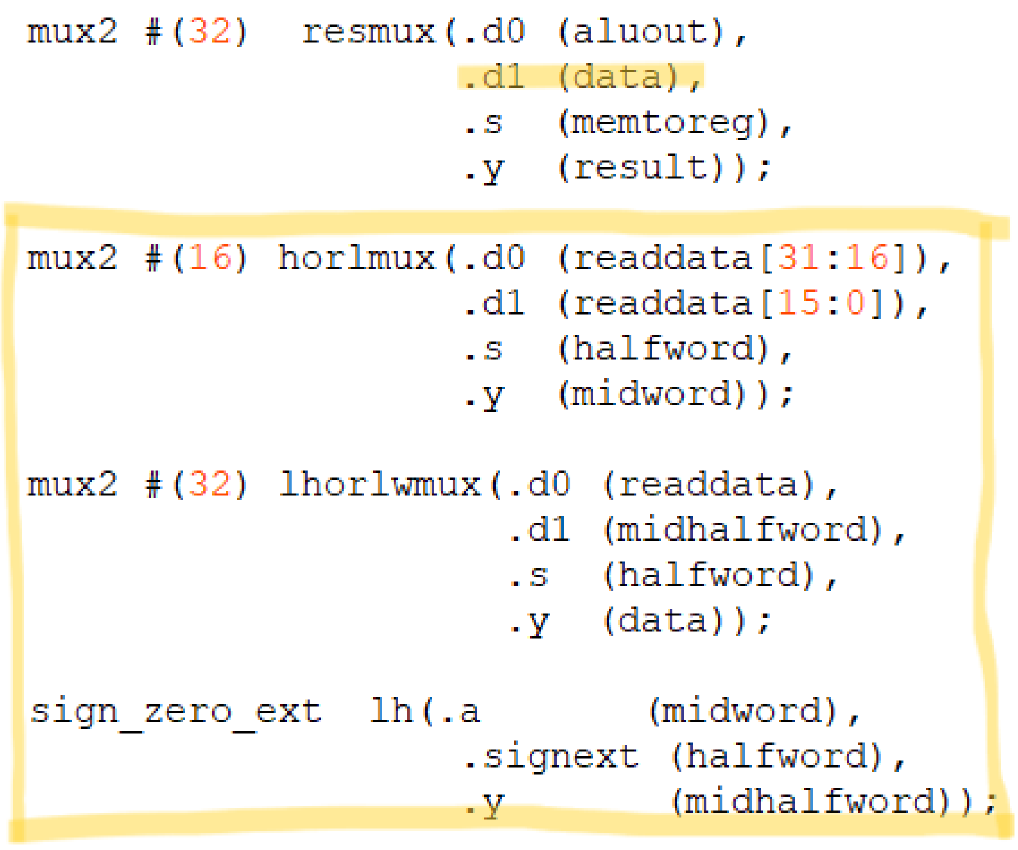
```verilog
wire        alusrc, regdst, regwrite, jump, halfword;
wire [3:0]  alucontrol;

// 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),
             .jump       (jump),
             .halfword   (halfword),
             .alucontrol (alucontrol));

// Instantiate Datapath
datapath dp( .clk        (clk),
             .reset      (reset),
             .signext    (signext),
             .shiftl16   (shiftl16),
             .memtoreg   (memtoreg),
             .pcsrc      (pcsrc),
             .alusrc     (alusrc),
             .regdst     (regdst),
             .regwrite   (regwrite),
             .jump       (jump),
             .halfword   (halfword),
             .alucontrol (alucontrol),
             .zero       (zero),
```



```verilog
module controller(input   [5:0] op, funct,
                  input         zero,
                  output        signext,
                  output        shiftl16,
                  output        memtoreg, memwrite,
                  output        pcsrc, alusrc,
                  output        regdst, regwrite,
                  output        jump,
                  output        halfword,
                  output  [3:0] alucontrol);

  wire [2:0] aluop;
  wire       branch;

  maindec md( .op      (op),
              .signext   (signext),
              .shiftl16  (shiftl16),
              .memtoreg  (memtoreg),
              .memwrite  (memwrite),
              .branch    (branch),
              .alusrc    (alusrc),
              .regdst    (regdst),
              .regwrite  (regwrite),
              .jump      (jump),
              .halfword  (halfword),
              .aluop     (aluop));

  aludec  ad( .funct      (funct),
              .aluop      (aluop),
              .alucontrol (alucontrol));

  assign pcsrc = branch & zero;

endmodule
```



```verilog
module datapath(input          clk, reset,
                input          signext,
                input          shiftl16,
                input          memtoreg, pcsrc,
                input          alusrc, regdst,
                input          regwrite, jump,
                input          halfword,
                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 [15:0] midword;
  wire [31:0] midhalfword;
  wire [31:0] data;
  wire [31:0] result;
  wire        shift;
```

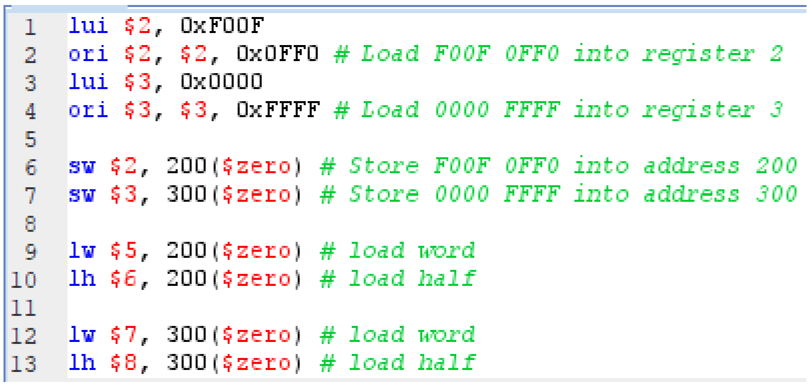


```
mux2 #(32)  resmux(.d0 (aluout),
                   .d1 (data),
                   .s  (memtoreg),
                   .y  (result));

mux2 #(16) horlmux(.d0 (readdata[31:16]),
                   .d1 (readdata[15:0]),
                   .s  (halfword),
                   .y  (midword));

mux2 #(32) lhorlwmux(.d0 (readdata),
                     .d1 (midhalfword),
                     .s  (halfword),
                     .y  (data));

sign_zero_ext  lh(.a       (midword),
                  .signext (halfword),
                  .y       (midhalfword));
```

Fig.21. Two Multiplexer and a Sign or Zero Extender

Here, we first need to extend the signals in the control module with an extra signal called halfword to control the loading of halfwords. Similarly, we need to change the instantiated wiring. Add this new signal to each place where it is needed. Finally, we need to add Two Multiplexer and a Sign or Zero Extender to the datapath module, where the first Multiplexer is used to select the high halfword or low halfword. The second Multiplexer is used to select whether to load a word or a half word and the Sign or Zero Extender is used to extend the sign bits.

3.3.2 Assembly Code

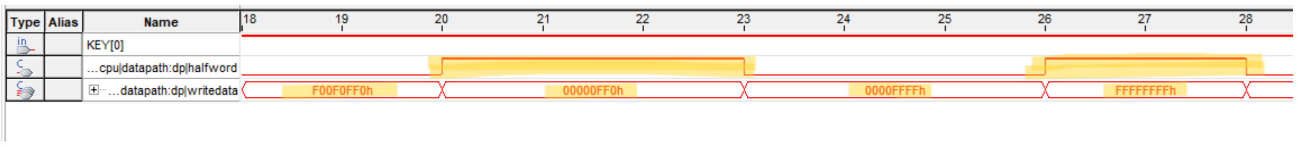


```
lui $2, 0xF00F
ori $2, $2, 0x0FF0 # Load F00F 0FF0 into register 2
lui $3, 0x0000
ori $3, $3, 0xFFFF # Load 0000 FFFF into register 3

sw $2, 200($zero) # Store F00F 0FF0 into address 200
sw $3, 300($zero) # Store 0000 FFFF into address 300

lw $5, 200($zero) # load word
lh $6, 200($zero) # load half

lw $7, 300($zero) # load word
lh $8, 300($zero) # load half
```

Fig.22. Assembly code for lh instruction

We need to test this instruction twice, firstly, we test it once for sign bit position 0, that is to say, expanding the sign bit followed by a complement of 0. And then we'll test one with an extended symbol bit of 1. Here, we'll start by loading two numbers 0xF00F 0FF0 and 0x0000 FFFF. we can then use the lw instruction we already have to make sure we loaded them correctly. We use the sw instruction to store these two values into memory at addresses 200 and 300, respectively. The lw and lh instructions are then used to load them. This way we are able to test if the results are correct.

3.3.3 Signal Tap



```
Type  Alias  Name                          18   19   20   21   22   23   24   25   26   27   28
             KEY[0]
             ...cpu|datapath:dp|halfword
             ...datapath:dp|writedata      F00F0FF0h | 00000FF0h | 0000FFFFh | FFFFFFFFh
```

Fig.23. Signal Tap for lh instruction

Here we can see that the first lw instruction loads 0xF00F 0FF0 in, and then we use the lh instruction to activate the halfword signal. Again, our data becomes the low halfword of the value above. And after completing the sign bit it is 0x0000 0FF0. This shows that our lh instruction is working correctly. Then we look at the second part of the lw instruction, where we loaded a 0x0000 FFFF in, and then used lh to load the halfword of that word, and after completing the sign bits, we get 0xFFFF FFFF, which is what we would expect, since the second word has a 1 in the sign bit of the halfword, which is the [15] bit, which is a negative number. So completing all the ones before that gives us 0xFFFF FFFF.

3.3.4 Block Diagram

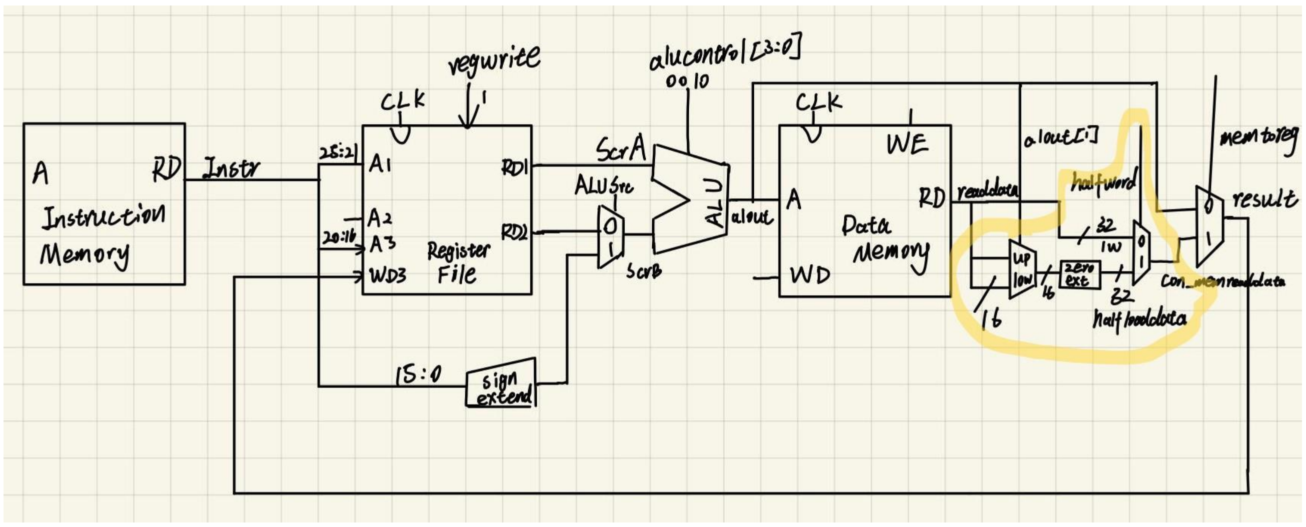


Fig.24. Block diagram for lh instruction

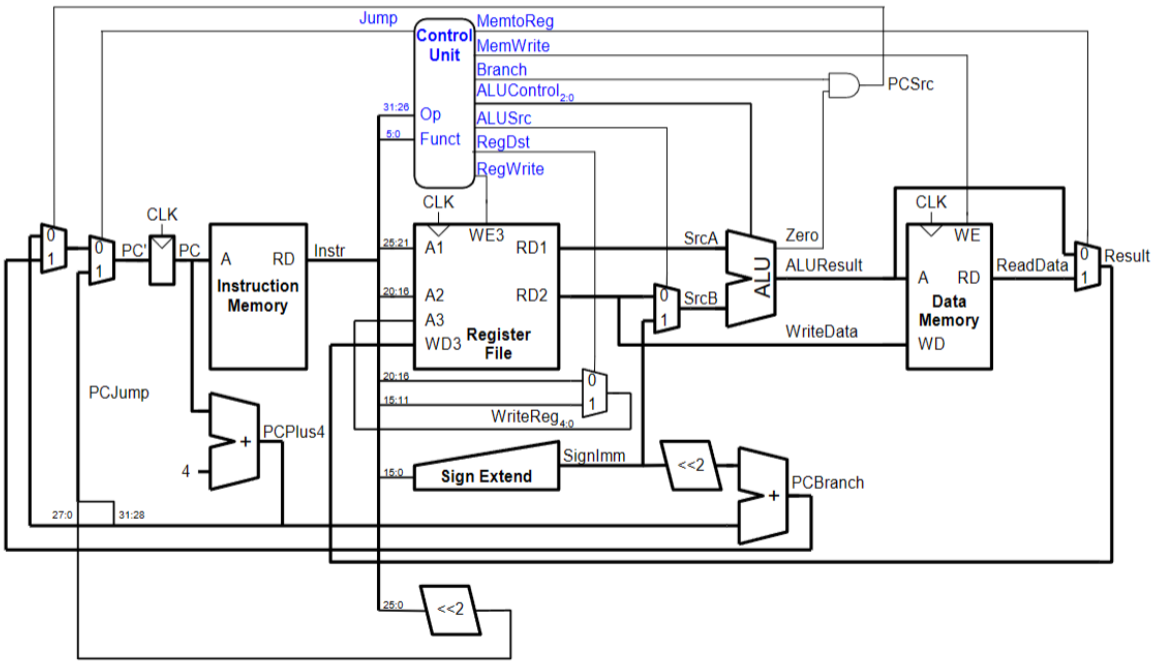


Fig.25. Block diagram for MIPS [2]

We can notice that the circled ones are our newly added modules, that is, two multiplexers as well as a symbol expander. We can see the difference between our added modules and the original modules by comparing them with the previous block diagram.

4 Discussion/Conclusions

In conclusion, the integration of additional instructions such as nor (bitwise NOT OR), xori (exclusive OR immediate), and lh (load halfword) into the MIPS (Microprocessor without Interlocked Pipeline Stages) architecture. By incorporating these instructions, along with utilizing multiplexers for selecting high and low bits, and incorporating load halfword and load word operations, as well as a sign extender, the MIPS processor's capabilities have been expanded. Also plotted ASM with block diagrams. Through thorough testing utilizing SignalTap, the seamless integration and proper functioning of these augmented instructions have been ensured within the MIPS processor framework.

5 References

[1] “MIPS architecture processors,” From Wikipedia, the free encyclopedia, Wikipedia, https://en.wikipedia.org/wiki/MIPS_architecture_processors (accessed Mar. 24, 2024).

[2] “Lecture 20-Tutorial-MIPS-Assignment-3-v4.pdf,” Studocu, https://liverpool.instructure.com/courses/67906/pages/lecture-notes-as-pdfs (accessed Mar. 24, 2024).