

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**

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own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web).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.

**Contents**

[1 Introduction 1](#_Toc162216506)

[2 Part A 2](#_Toc162216507)

[2.1 Task 2](#_Toc162216508)

[2.2 Assembly 4](#_Toc162216509)

[2.3 Signal Tap 5](#_Toc162216510)

[2.4 Screenshot 5](#_Toc162216511)

[3 Part B 6](#_Toc162216512)

[3.1 NOR Instruction 6](#_Toc162216513)

[3.1.1 Verilog Code 6](#_Toc162216514)

[3.1.2 Assembly Code 7](#_Toc162216515)

[3.1.3 Signal Tap 8](#_Toc162216516)

[3.1.4 ASM 8](#_Toc162216517)

[3.2 XORI Instruction 9](#_Toc162216518)

[3.2.1 Verilog Code 9](#_Toc162216519)

[3.2.2 Assembly Code 11](#_Toc162216520)

[3.2.3 Signal Tap 12](#_Toc162216521)

[3.2.4 ASM 12](#_Toc162216522)

[3.3 LH Instruction 13](#_Toc162216523)

[3.3.1 Verilog Code 13](#_Toc162216524)

[3.3.2 Assembly Code 16](#_Toc162216525)

[3.3.3 Signal Tap 16](#_Toc162216526)

[3.3.4 Block Diagram 17](#_Toc162216527)

[4 Discussion/Conclusions 18](#_Toc162216528)

[5 References 18](#_Toc162216529)

# 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.

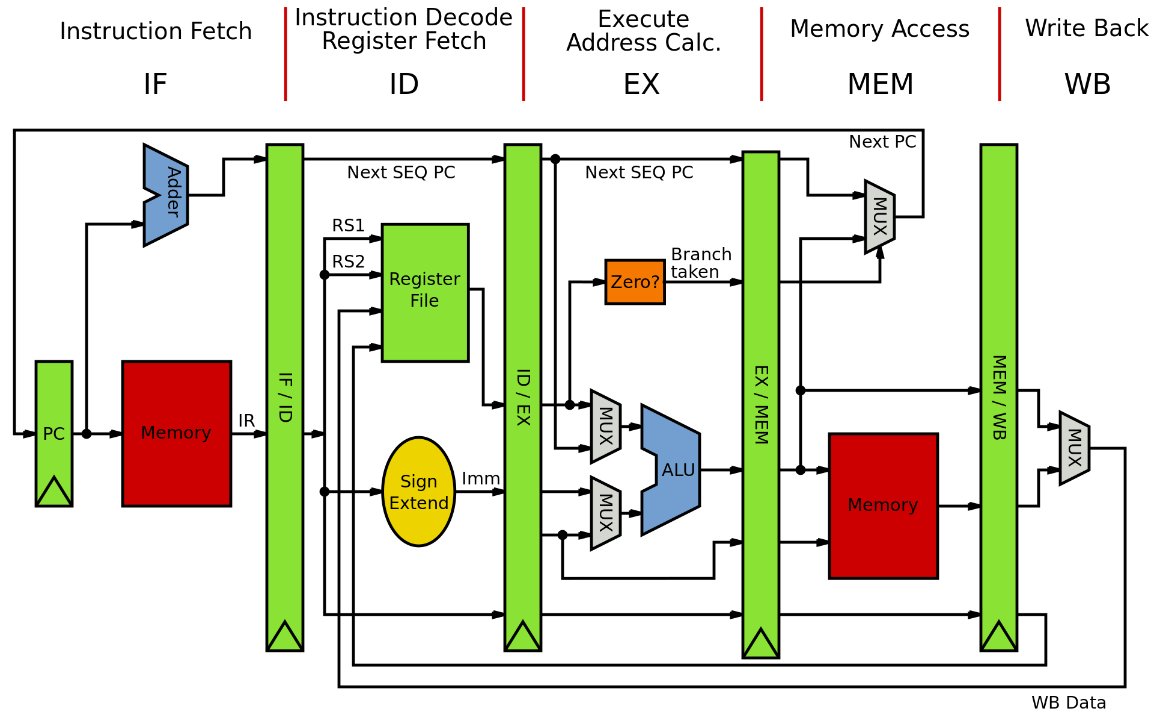


Fig.1. Pipeline MIPS Architecture [1]

# 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.

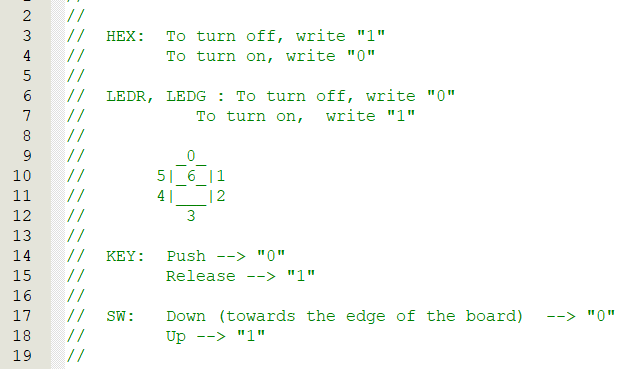


Fig.2. 7-segment

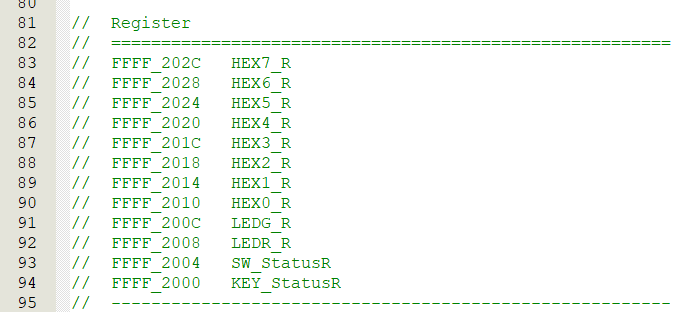


Fig.3. Register address of 7 segment

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.

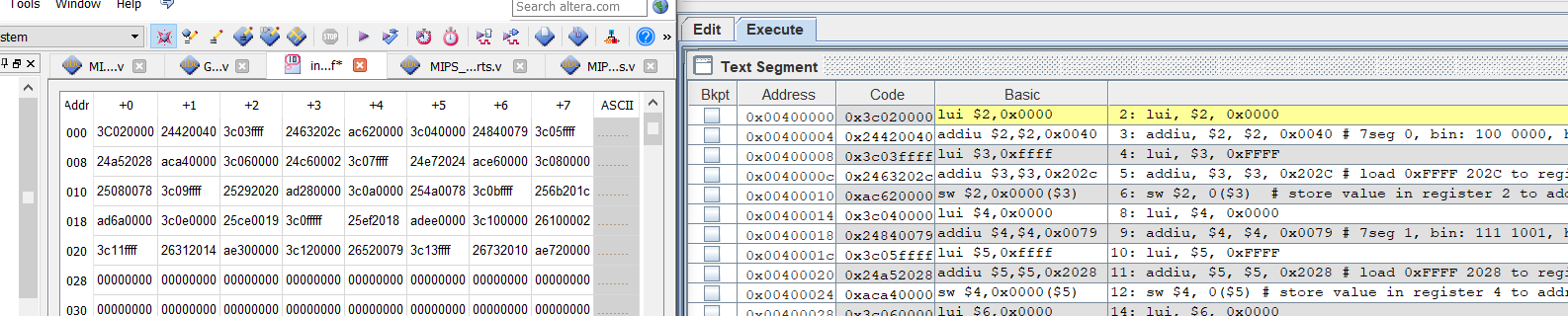
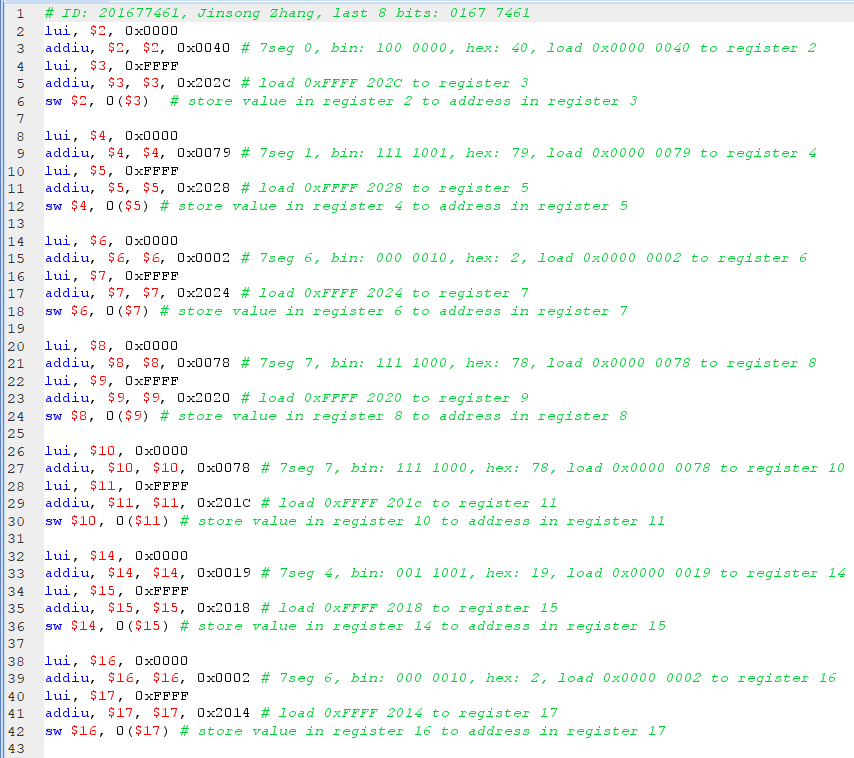
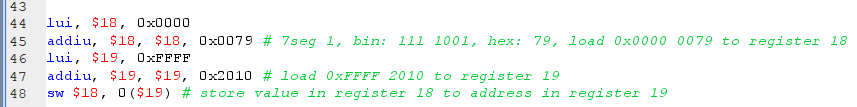


Fig.4. Instruction in .mif file

## 2.2 Assembly





## 2.3 Signal Tap

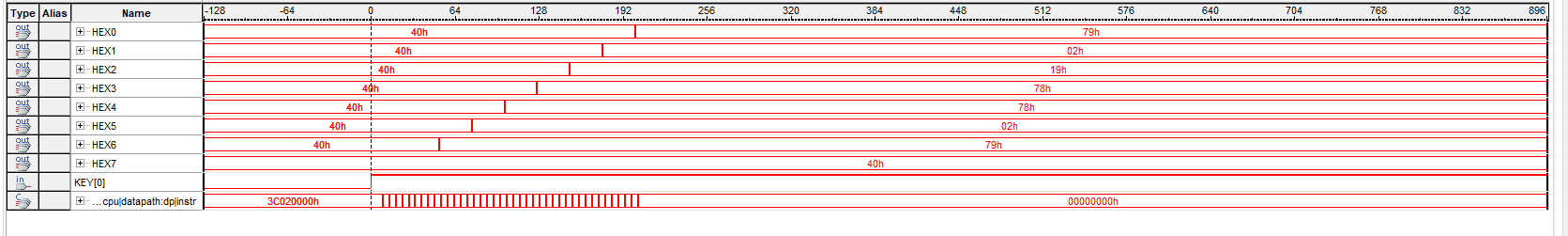


Fig.5. Signal Tap of 7-segment in 01677461

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

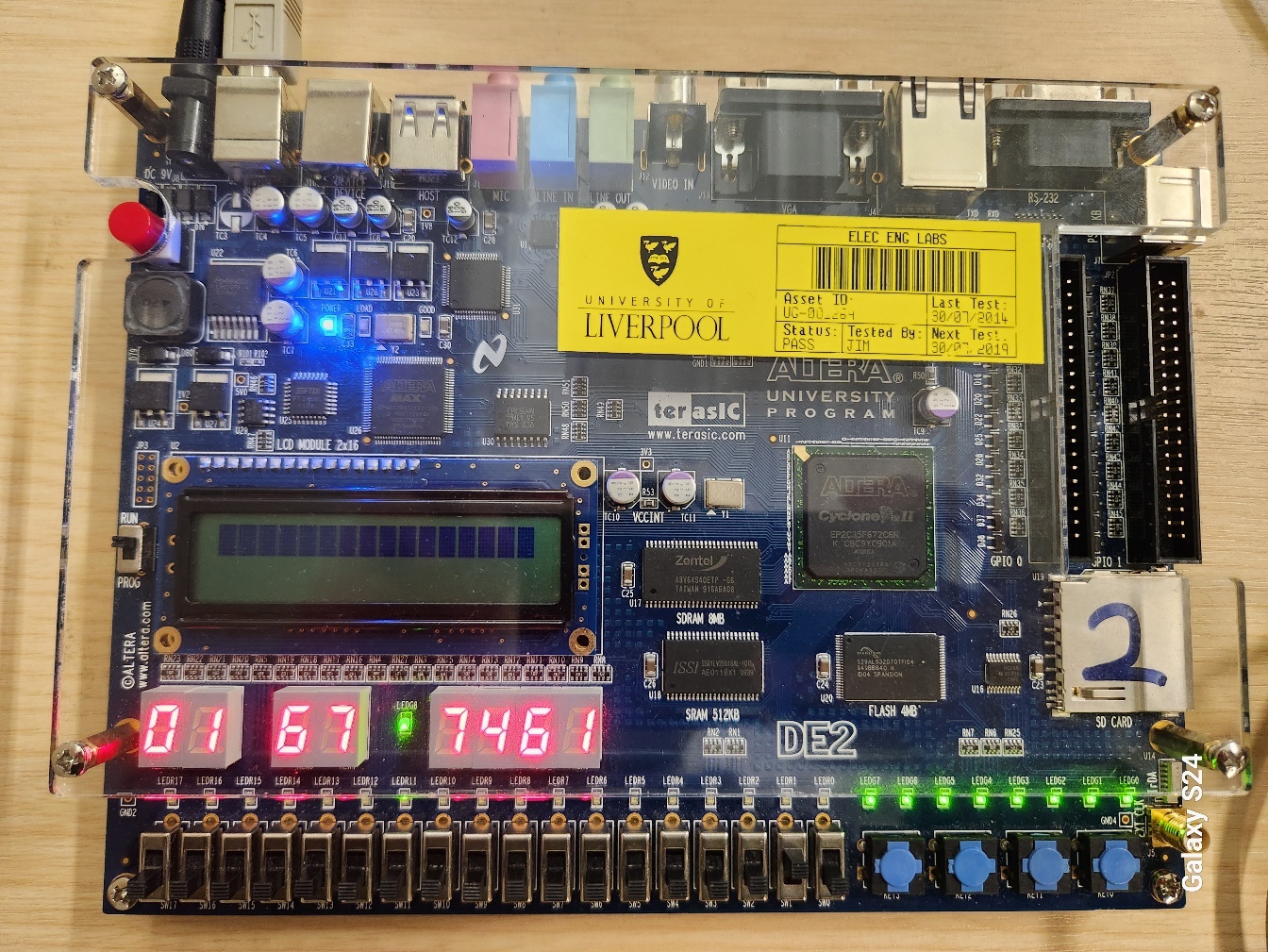


Fig.6. ID: 01677461 on DE 2 board

# 3 Part B

## 3.1 NOR Instruction

### 3.1.1 Verilog Code

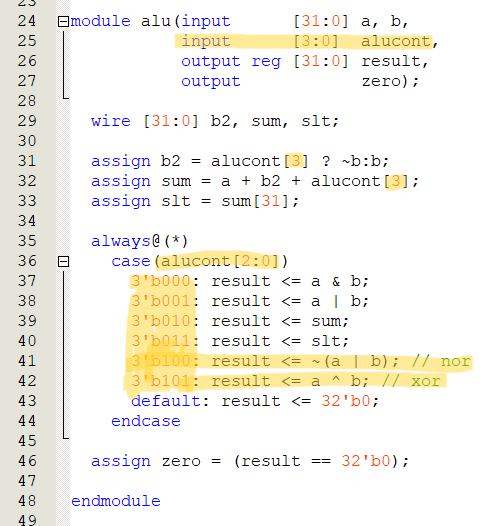


Fig.7. ALU module

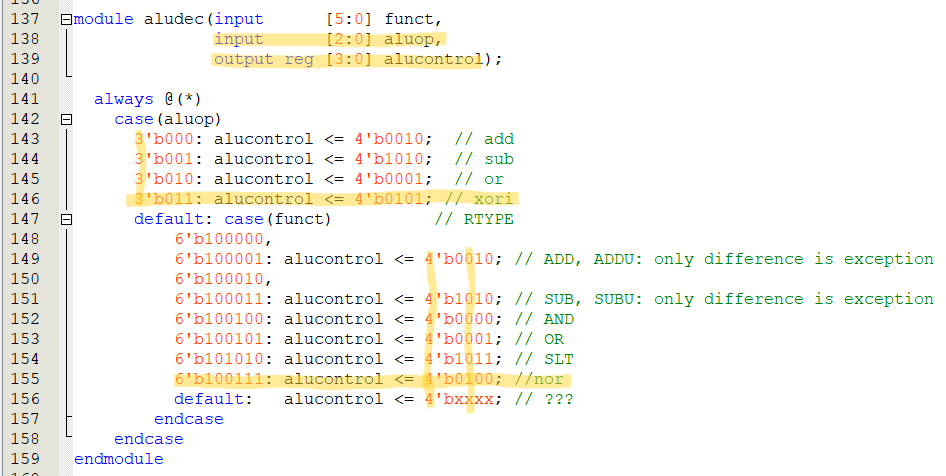


Fig.8. ALU dec module

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

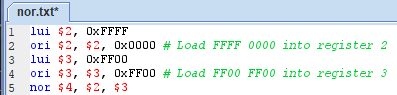


Fig.9. MIPS assembly code for nor

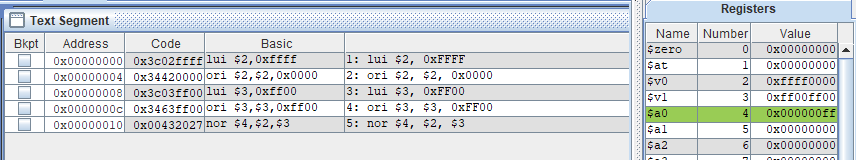


Fig.10. MIPS assembly code for nor (execute)

|  |  |  |
| --- | --- | --- |
| **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

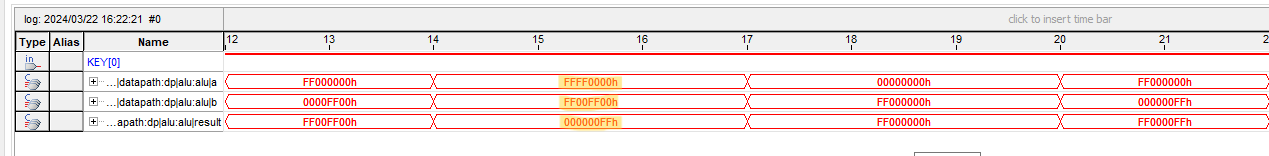


Fig.11. Signal Tap for nor instruction

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

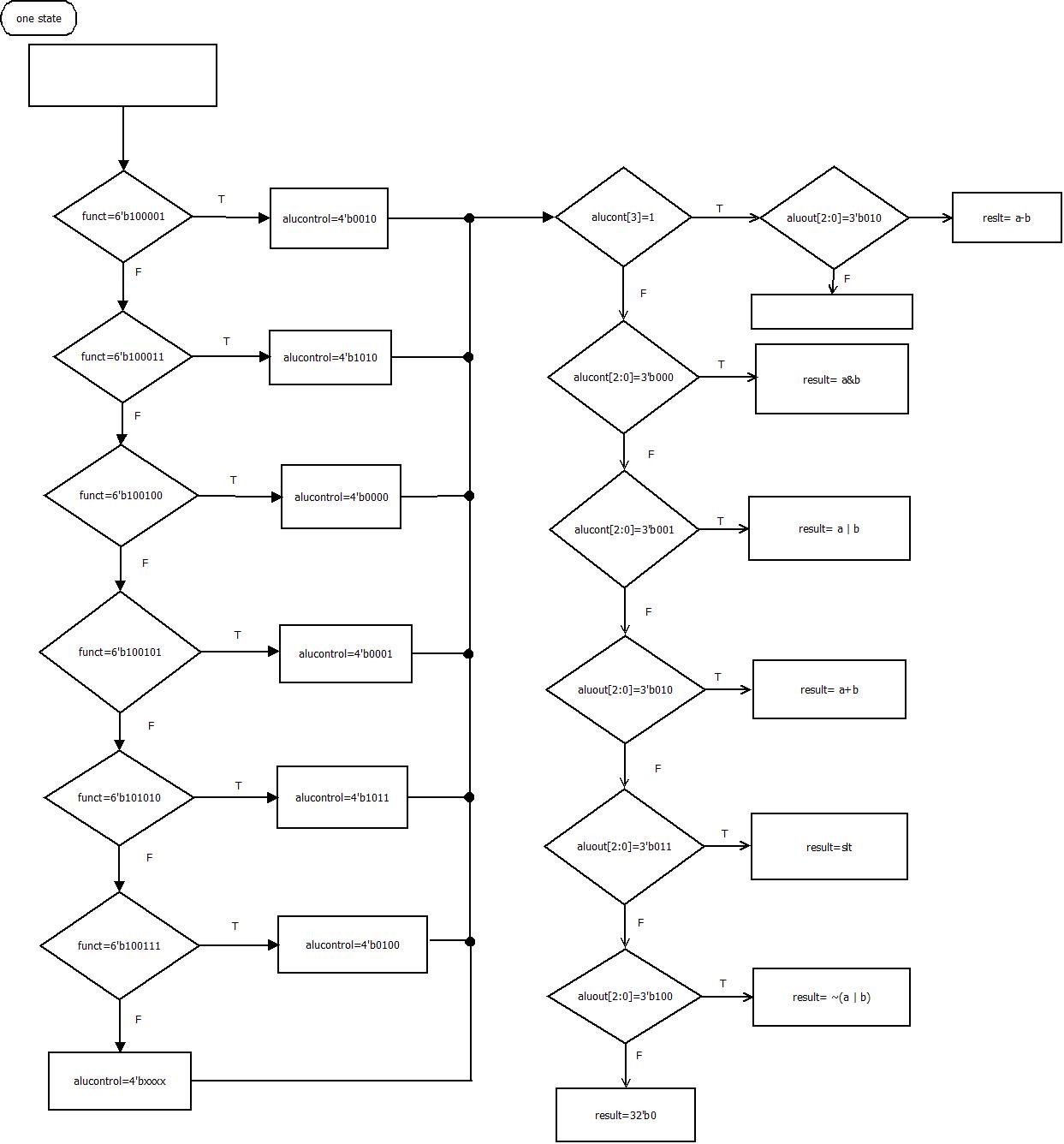


Fig.12. ASM for nor instruction

## 3.2 XORI Instruction

### 3.2.1 Verilog Code

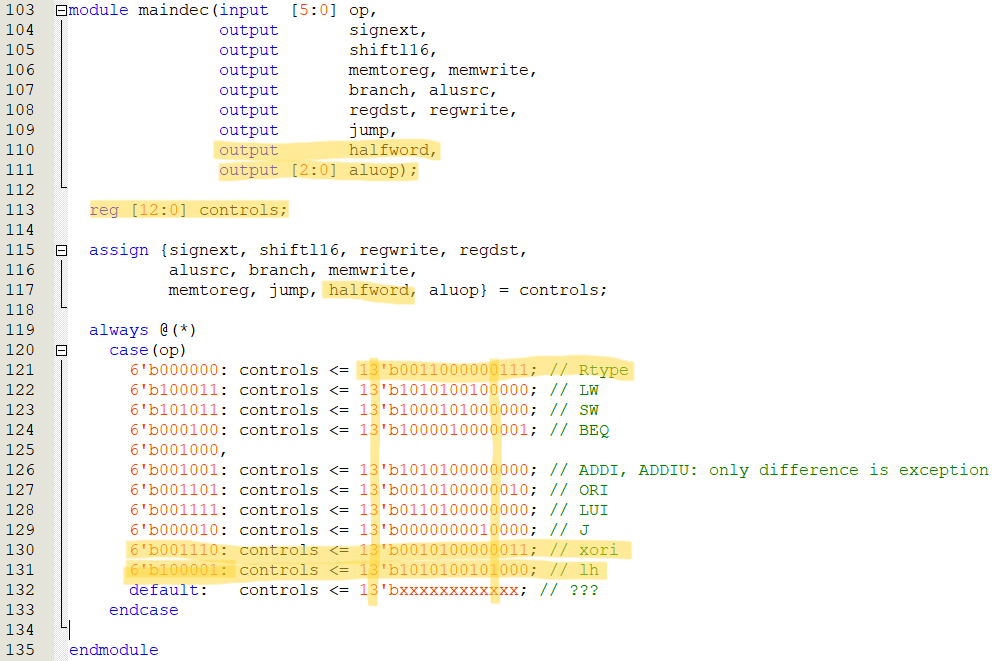


Fig.13. Main decoder module

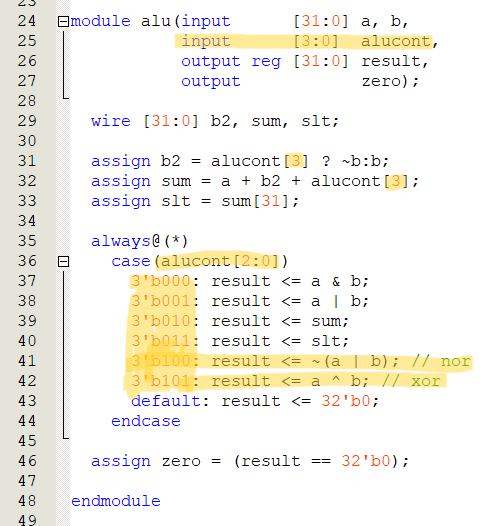


Fig.14. ALU module

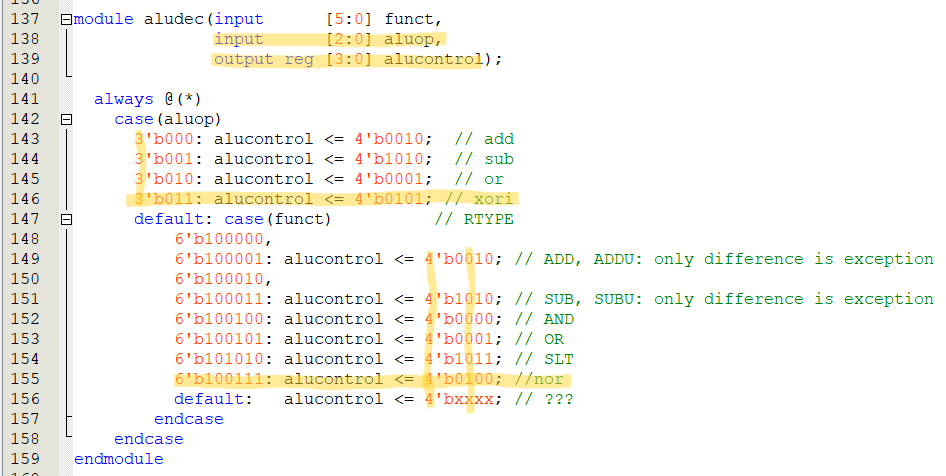


Fig.15. ALU decoder module

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



Fig.16. Assembly code for xori

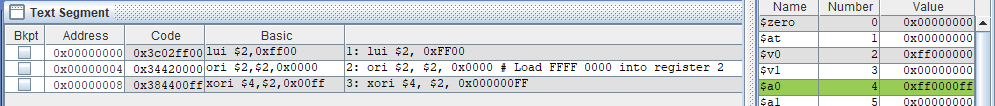


Fig.17. Assembly code for xori (execute)

|  |  |  |
| --- | --- | --- |
| **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

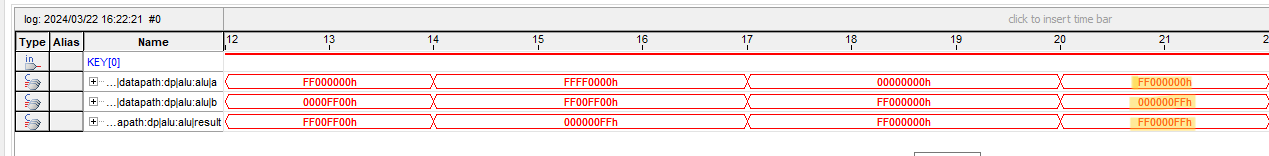
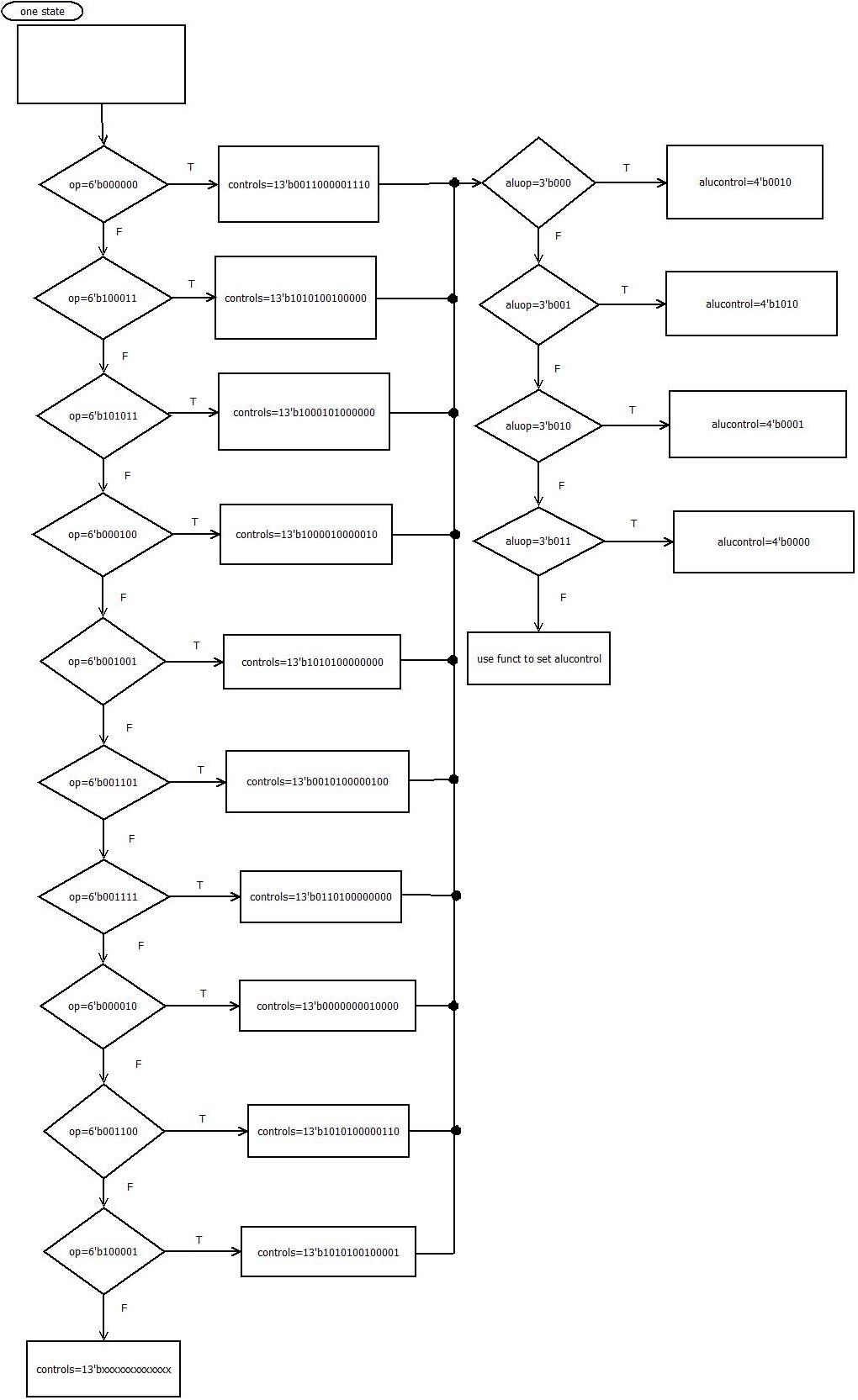


Fig.18. Signal Tap for xori instruction

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



## 3.3 LH Instruction

### 3.3.1 Verilog Code

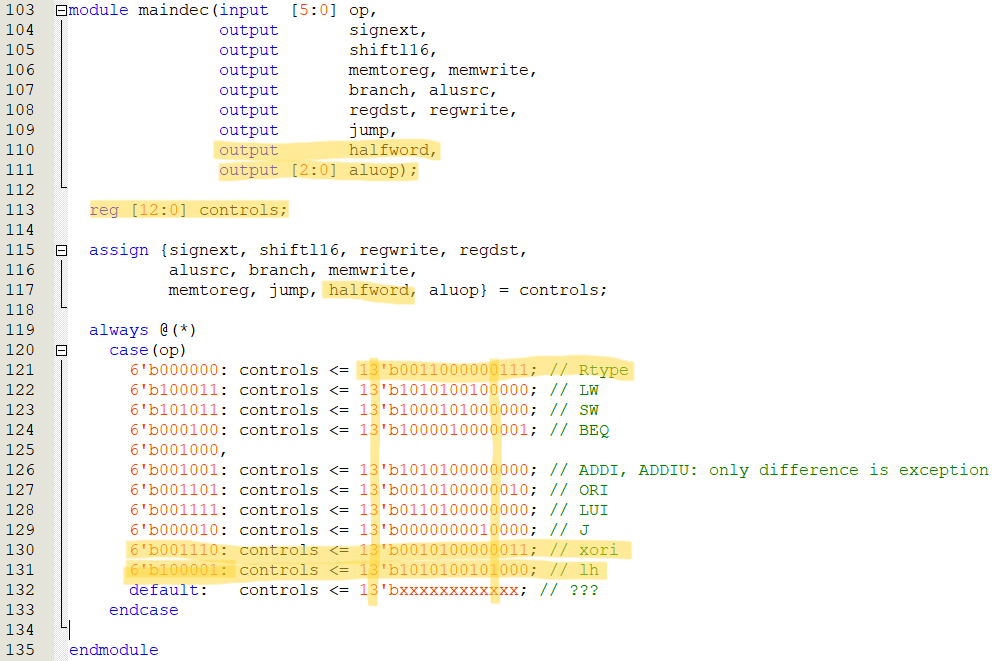
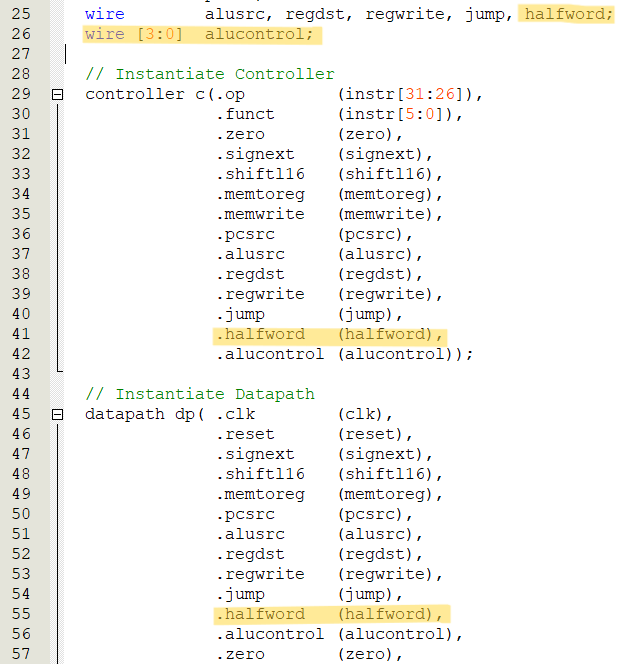
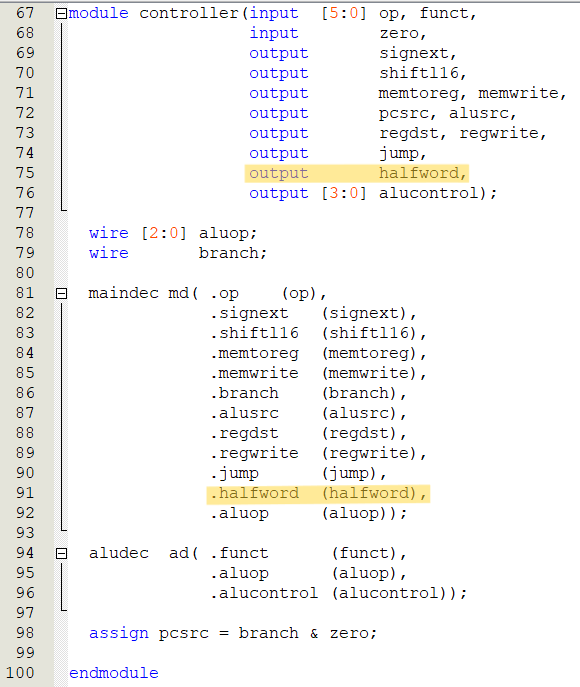
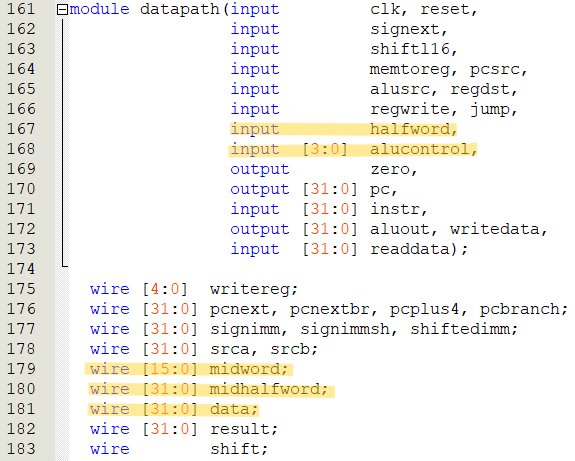


Fig.20. Main decoder module







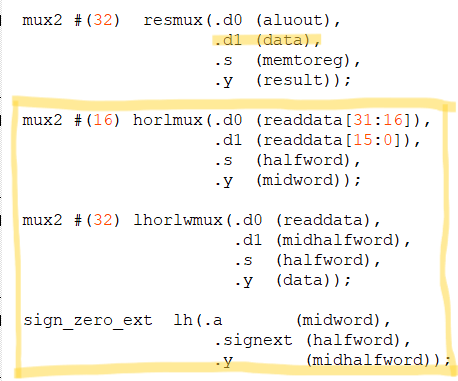


Fig.21. Two Multiplexer and a Sign or Zero Extender

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

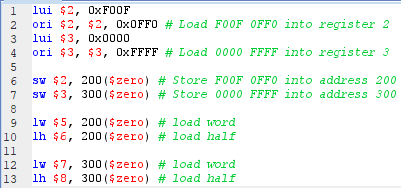


Fig.22. Assembly code for lh instruction

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

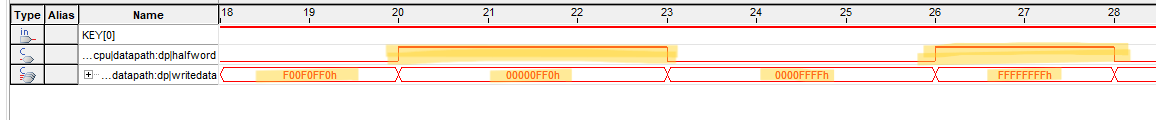


Fig.23. Signal Tap for lh instruction

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

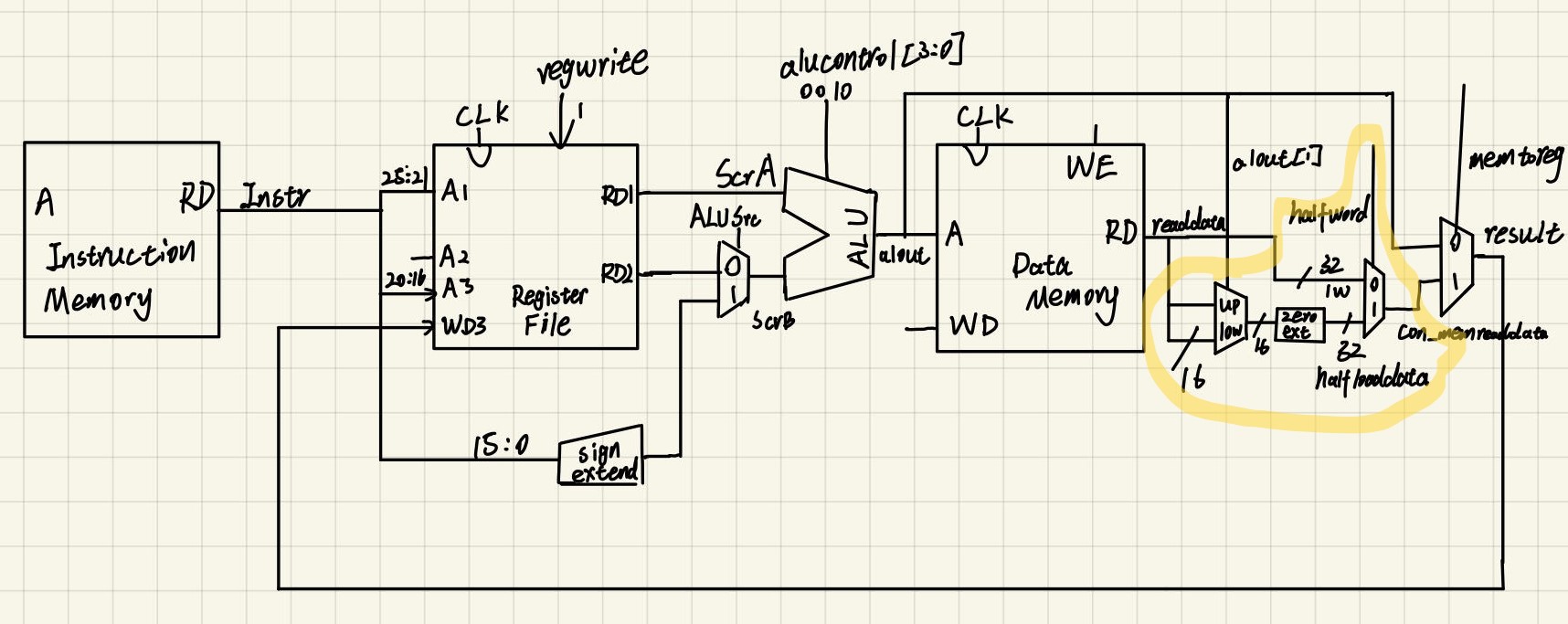


Fig.24. Block diagram for lh instruction

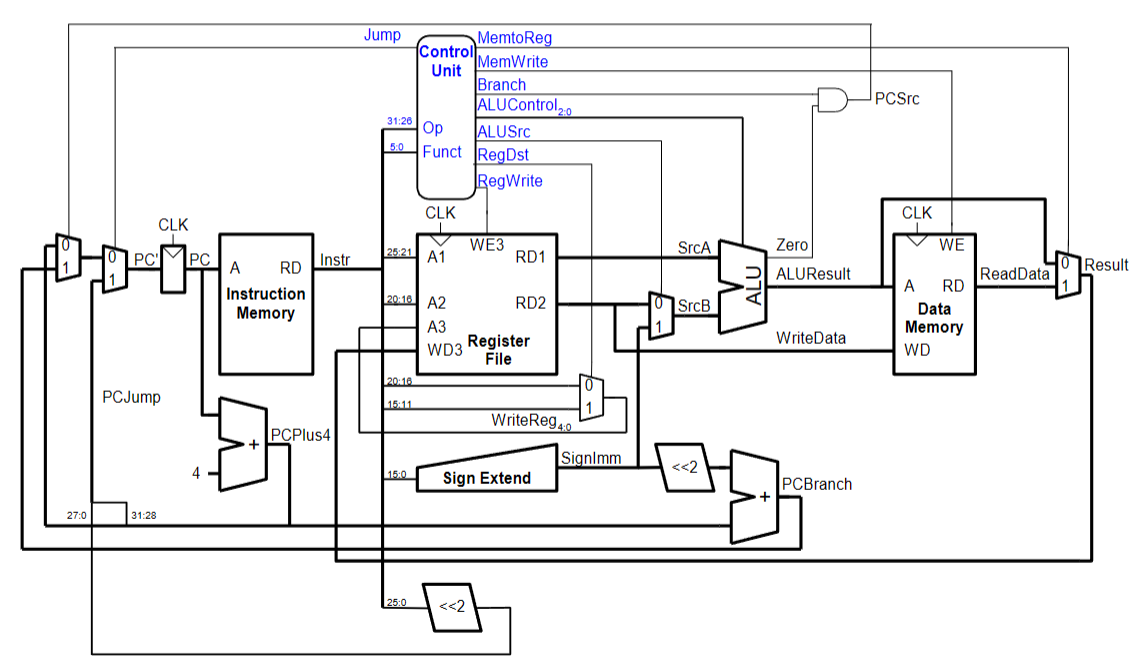


Fig.25. Block diagram for MIPS [2]

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).