

**ELEC373 Assignment 3**

**MIPS Processor**

**Yiyang Zhou**

**ID: 201601355**

**March 2023**

**Abstract**

Assignment 3 is divided into two parts. The objective of the first part was mainly to introduce the synthesised MIPS single-cycle processor, and try to write some simple programs to control the processor. The second part focuses on extending the processor to let it implement additional instructions.

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own words except where I have clearly indicated and acknowledged that I have quoted or used figures from published or unpublished sources (including the web).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

Contents

[1 Part A 3](#_Toc129635498)

[2 Part B 5](#_Toc129635499)

# Part A

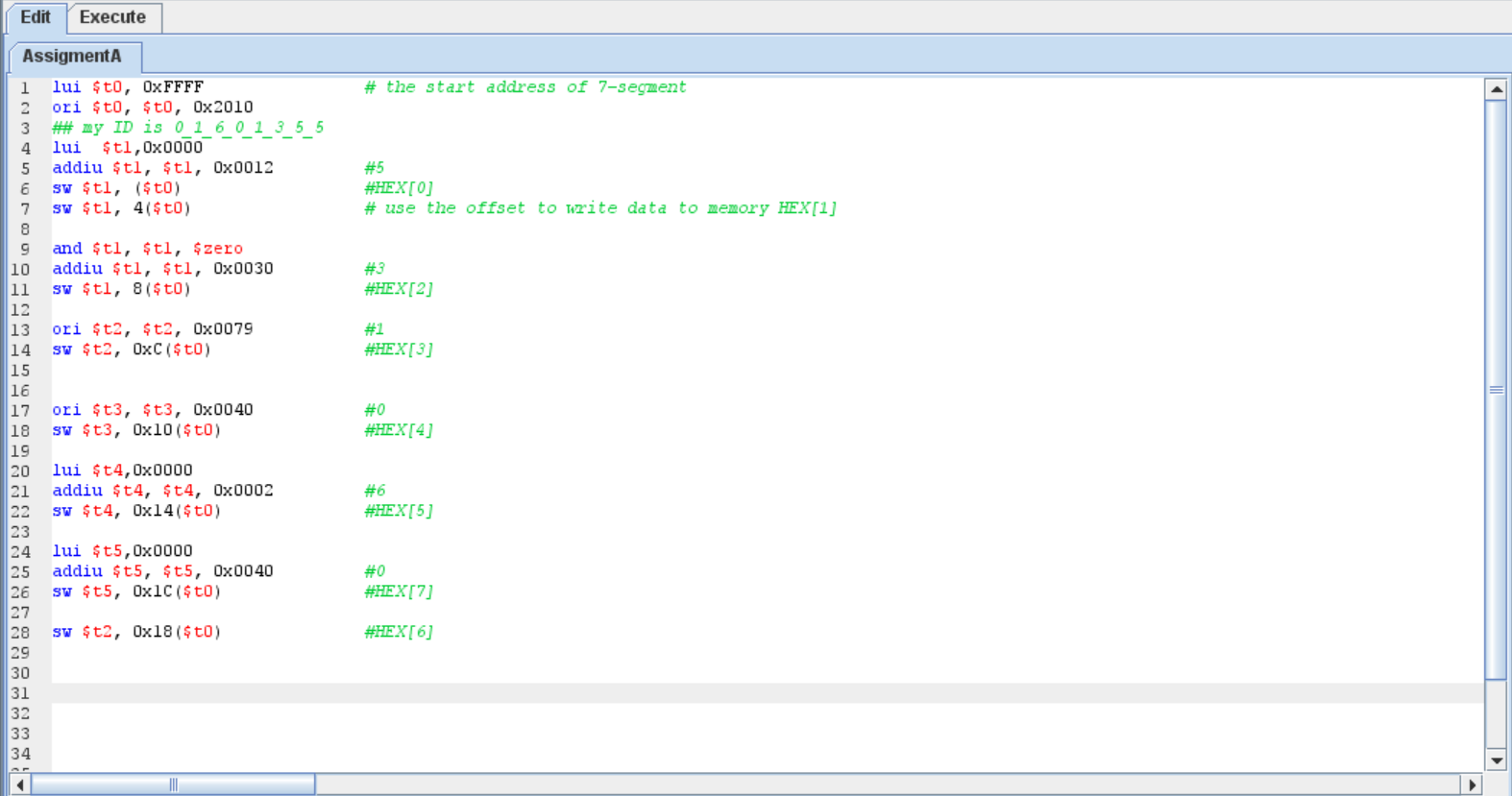


Fig. 1. MIPS assembly code for part A

Fig. 1 shows the assembly language code for part A. And we code for this part by using t0 to store the address value of the HEX[0], because in this part of the work we want to control the 7-segment displays. Then we store the seven-segment display control value to register and use the ‘SW’ instruction to store the value to corresponding address. We also use offset to store value and it could help us avoid repeatedly writing the address value to register then store the value. The number and corresponding 7-segment code used in this part are shown in the table below.

Table 1: The number used and the corresponding 7-segment code

|  |  |
| --- | --- |
| Number | 7-segment code |
| 5 | 0x12 |
| 3 | 0x30 |
| 1 | 0x79 |
| 0 | 0x40 |
| 6 | 0x02 |

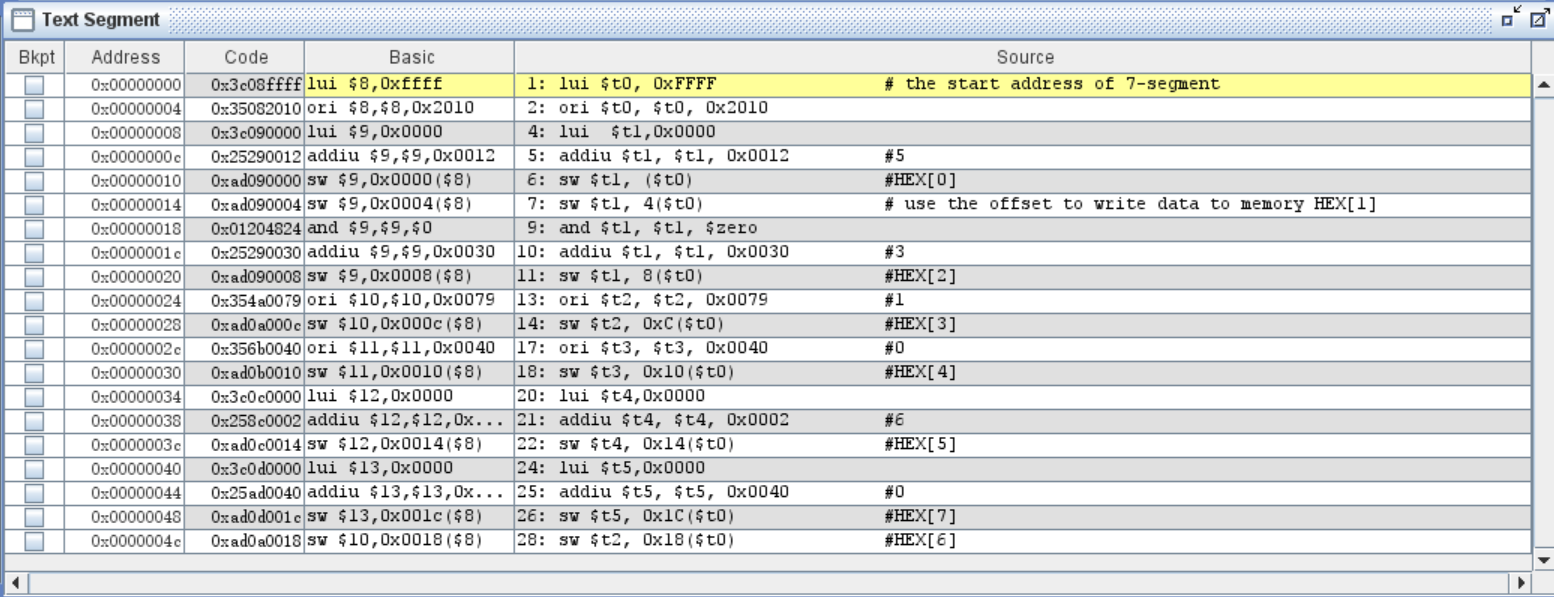


Fig. 2. Display the assembly language code and instruction code

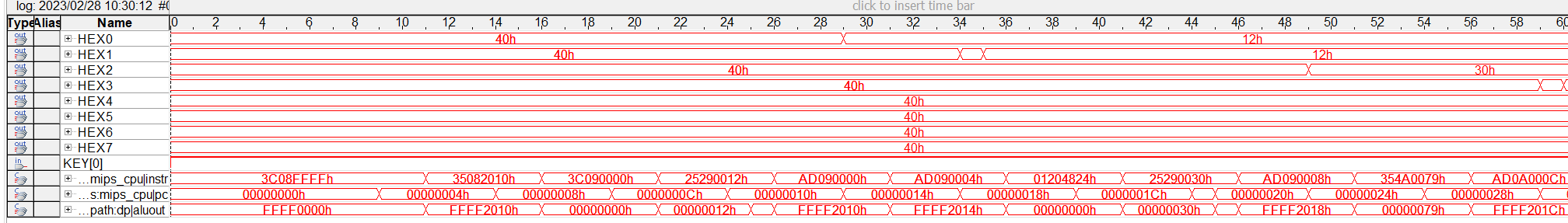


Fig. 3. Simulation result (1)

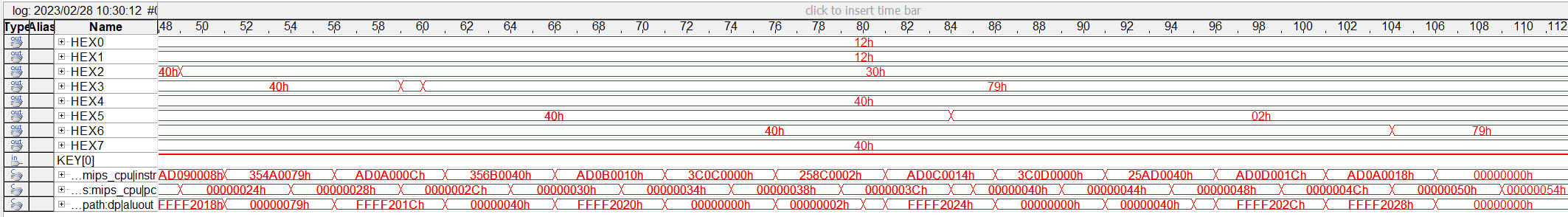


Fig. 4. Simulation result (2)

According to the assembly code and instruction code shown in Fig. 2, and the simulation results in SignalTap Logic analyser as shown in Fig. 3 and Fig. 4. We could know whether the instruction is executed as we want. For example, for the instruction to write HEX[0], the instruction code is 0xAD090000 and the instructions before it set the value of registers. Then we find the store instruction code in Fig.3, and we could find after some delay the value store in HEX0 will become 0x12 which is the number 5.

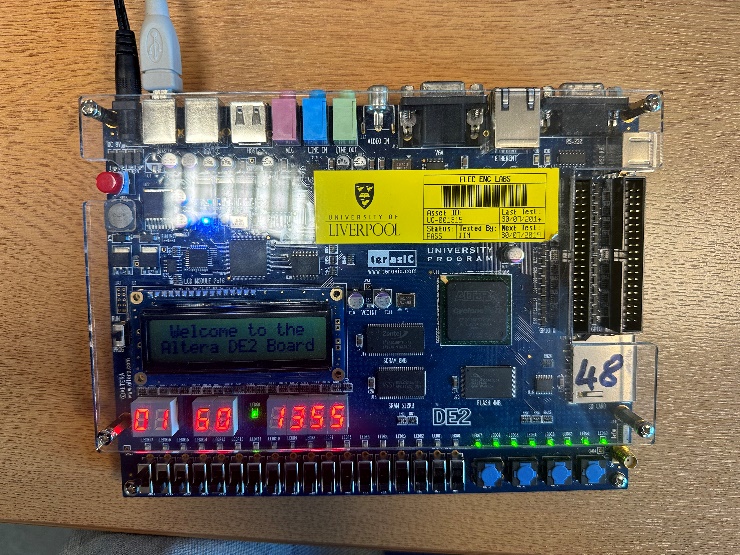


Fig. 5. 7-segment displays on DE2 board

Finally, the lowest 8 digits of my ID could display on the DE2 board as shown in Fig. 5.

# Part B

* 1. **Modified Verilog code**

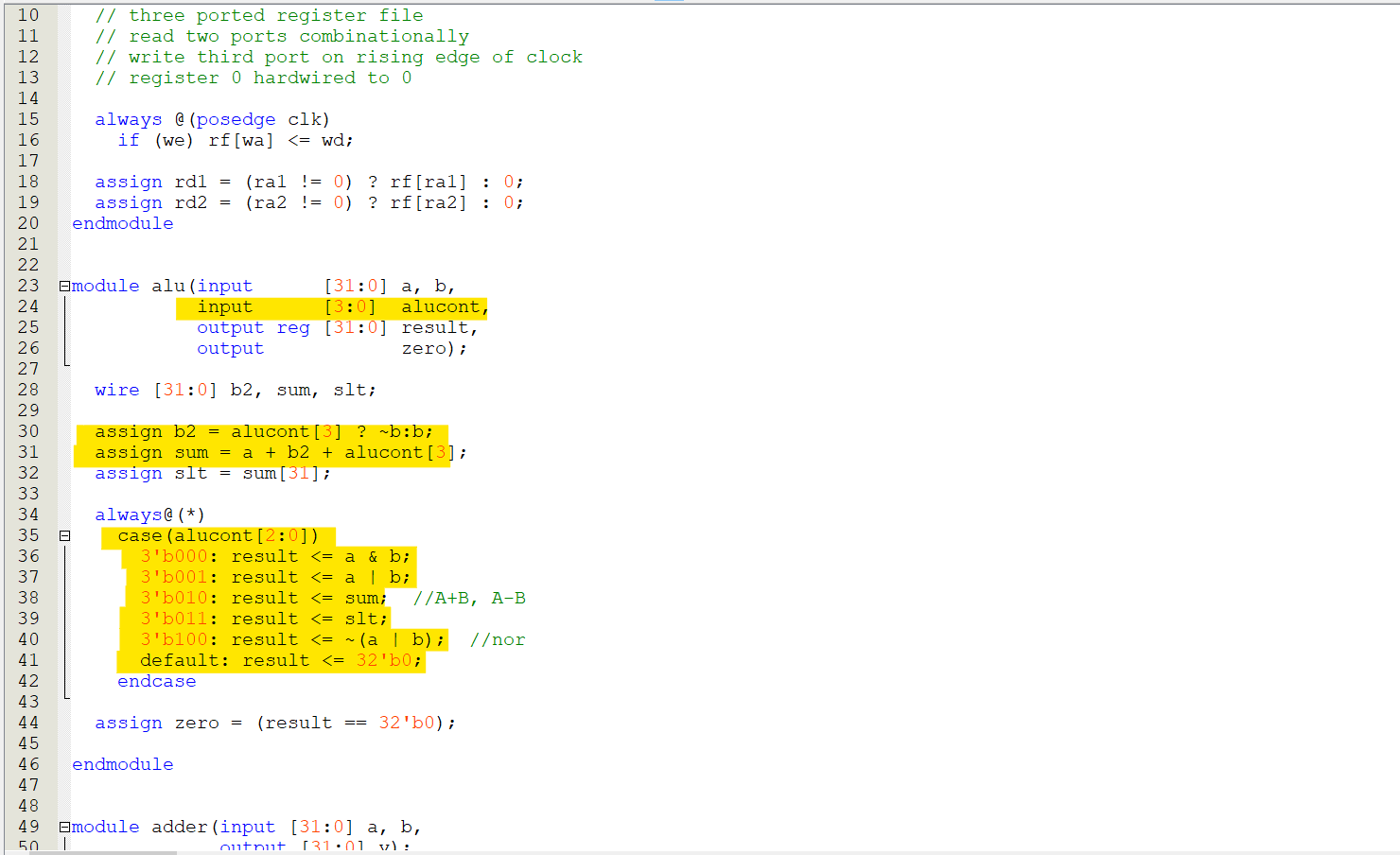
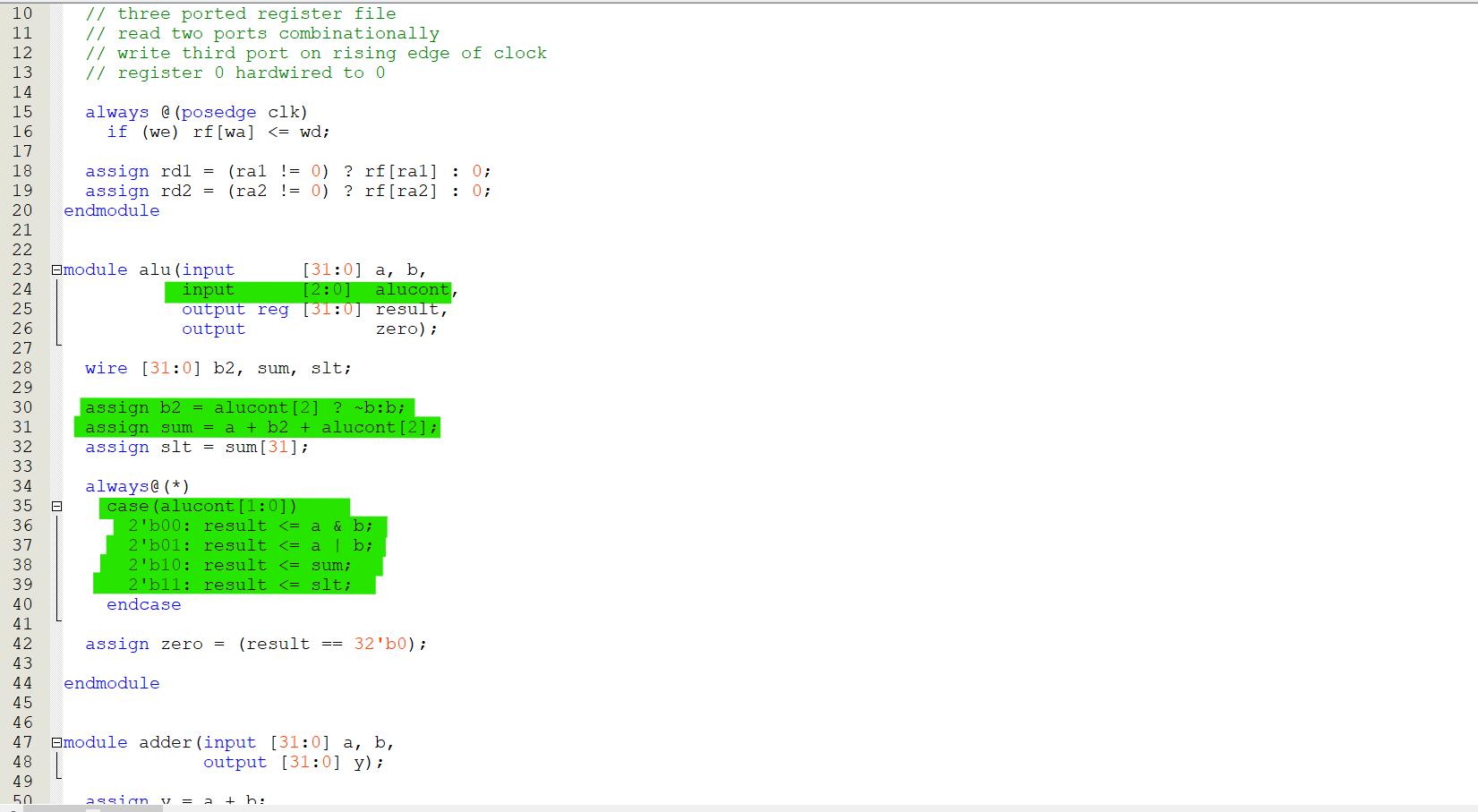


Fig. 6. mipsparts.v

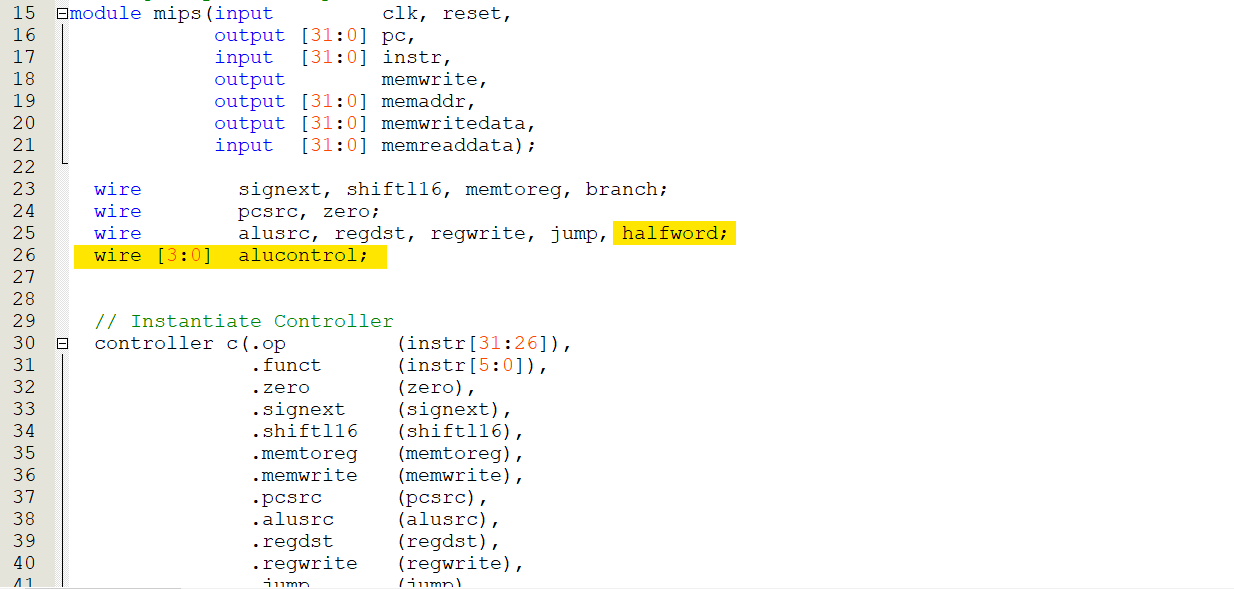
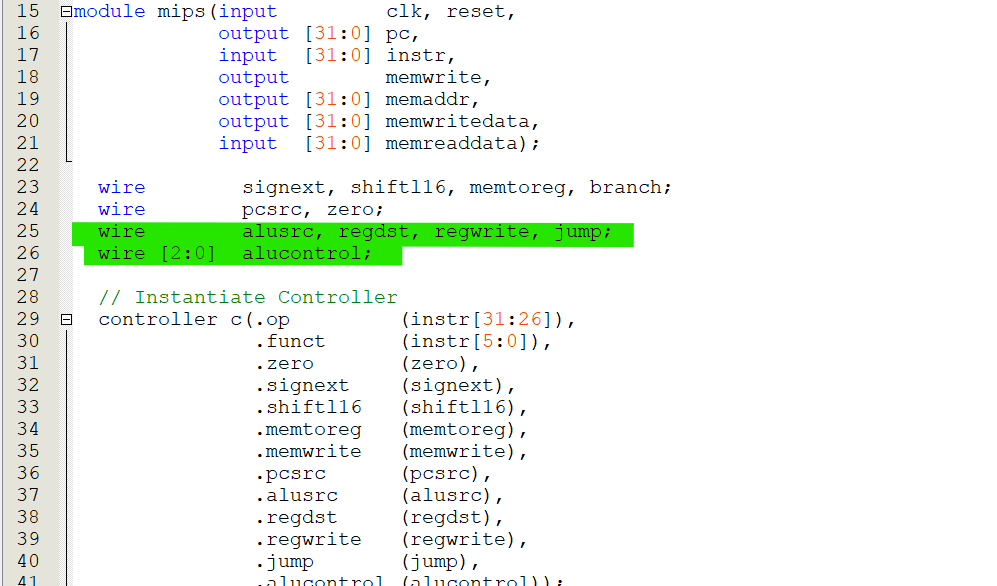


Fig. 7. mips.v\_mips(1)

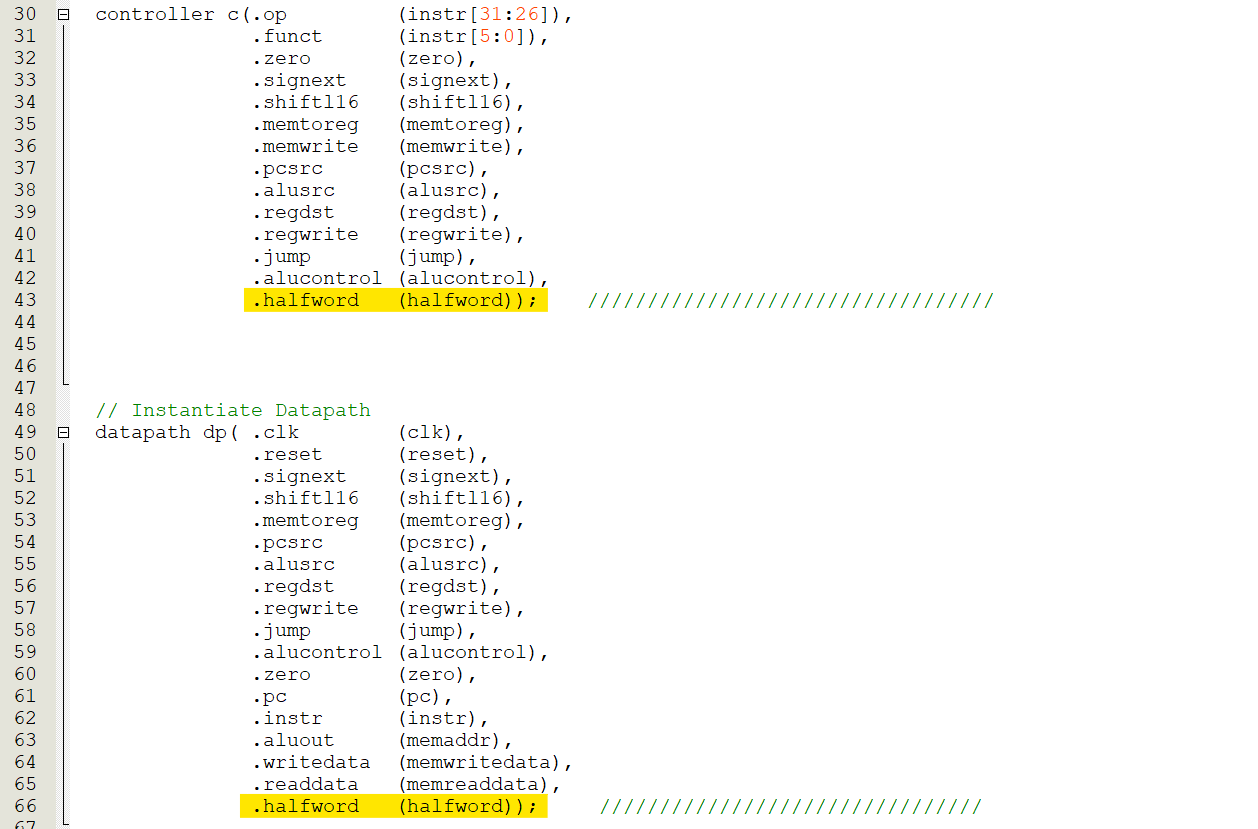
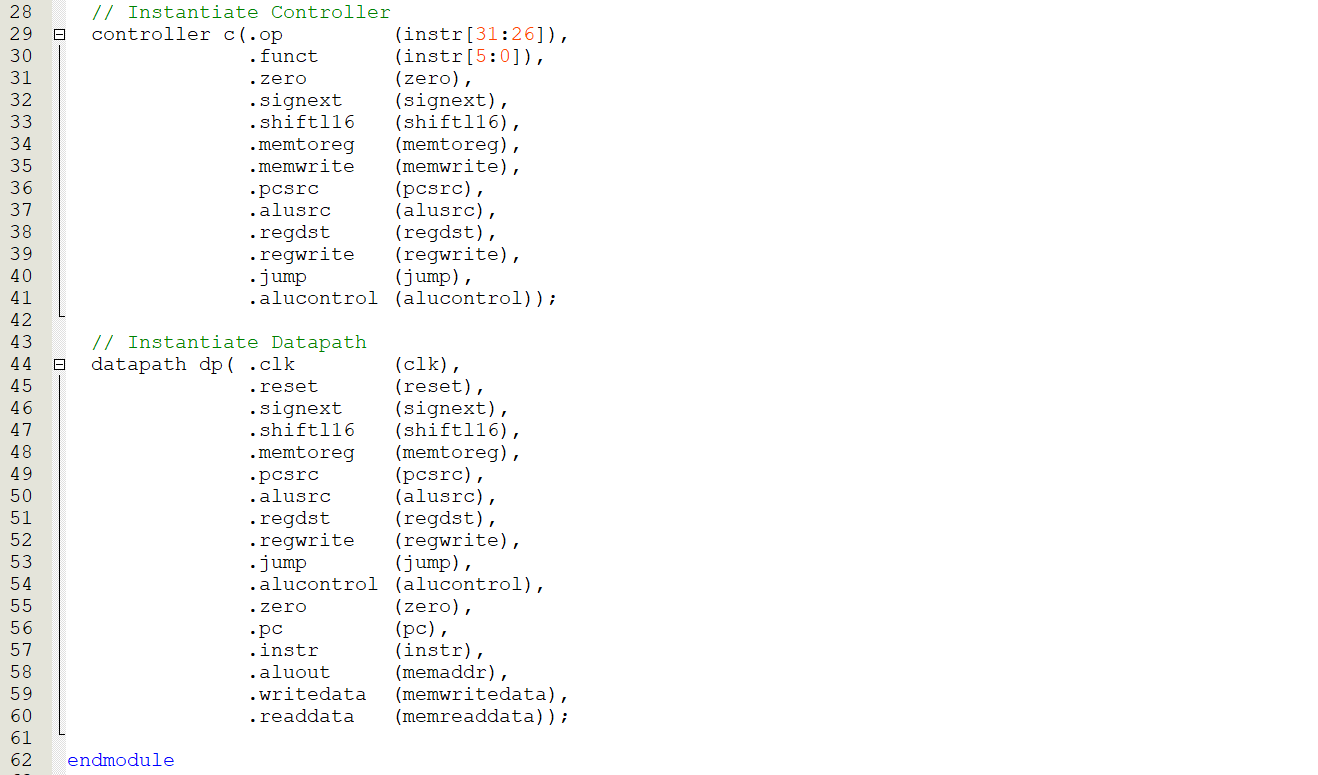


Fig. 8. mips.v\_mips(2)

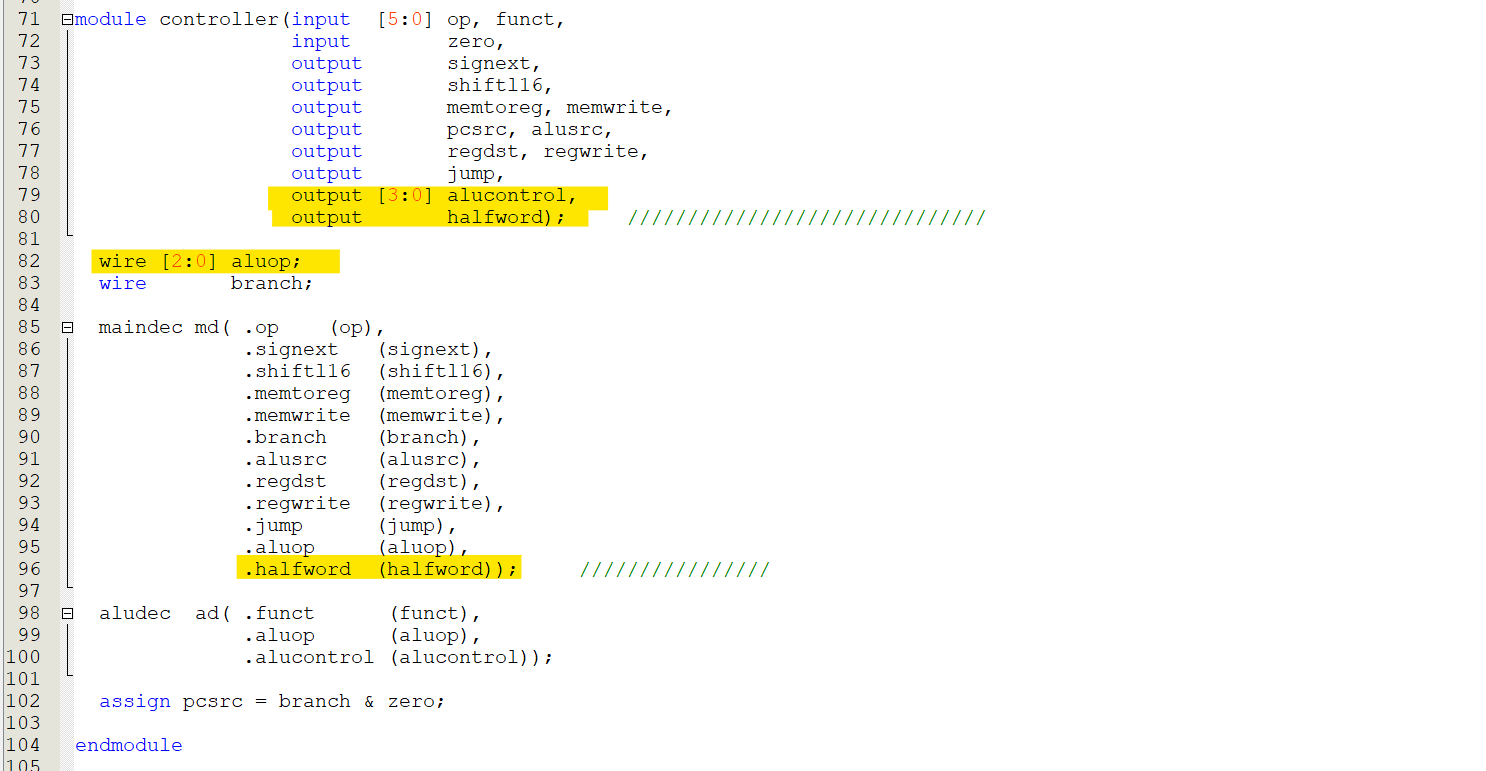


Fig. 9. mips.v\_controller

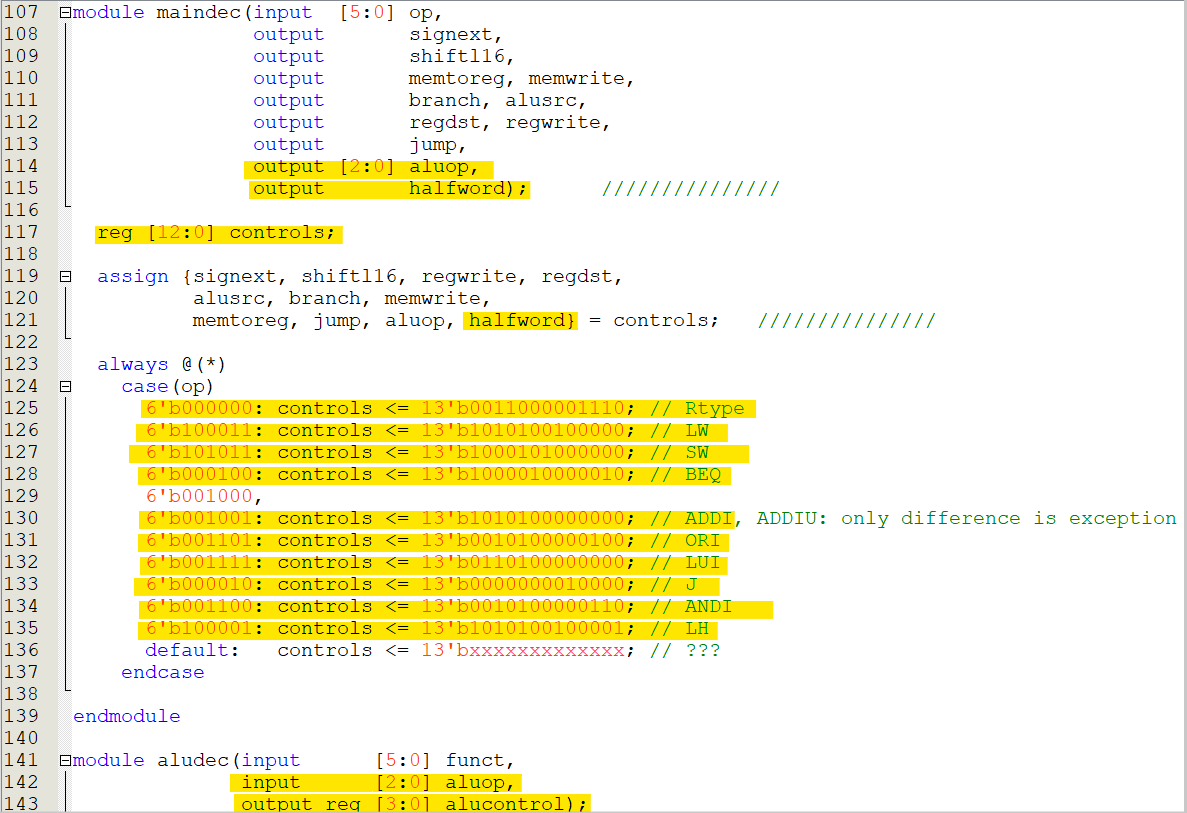
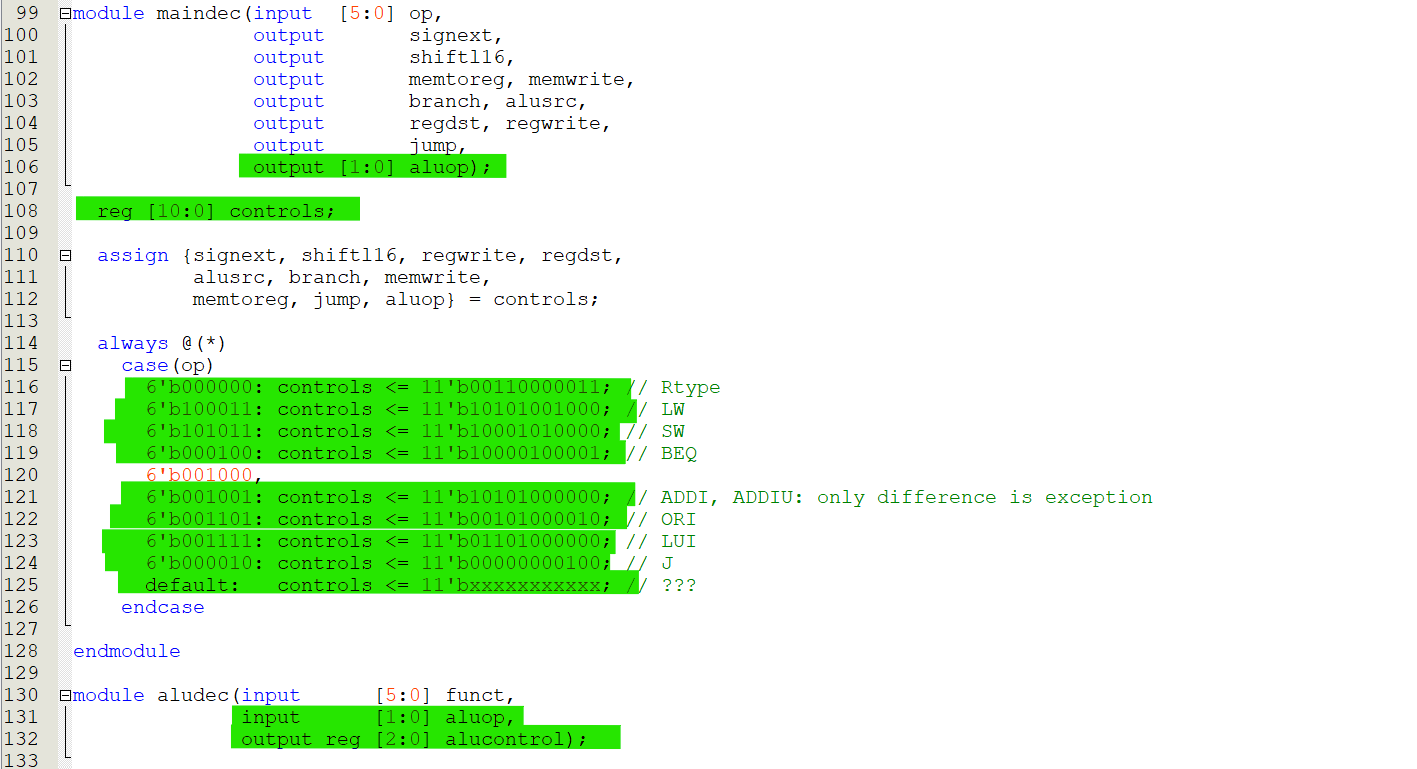


Fig. 10. mips.v\_maindec

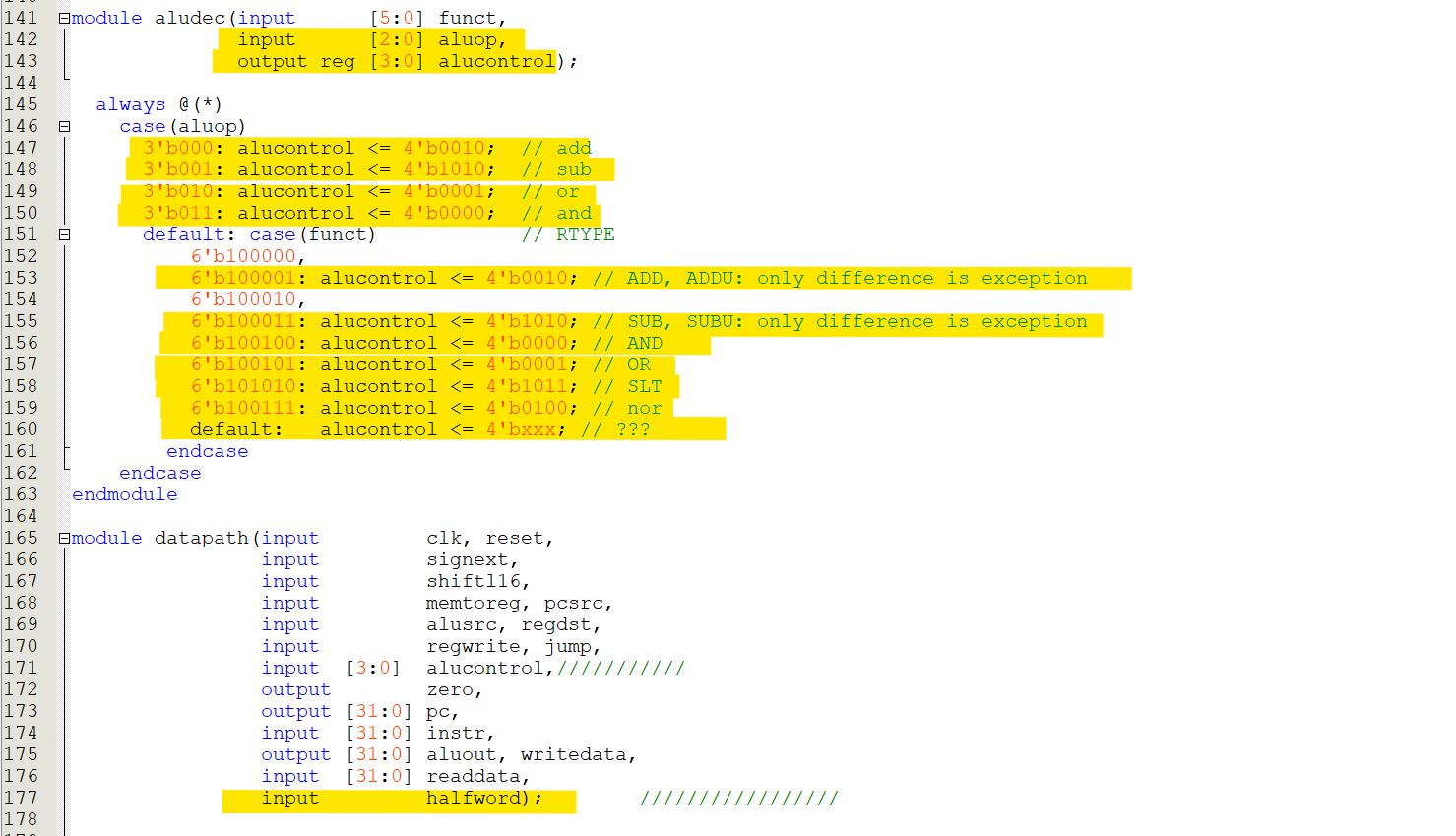
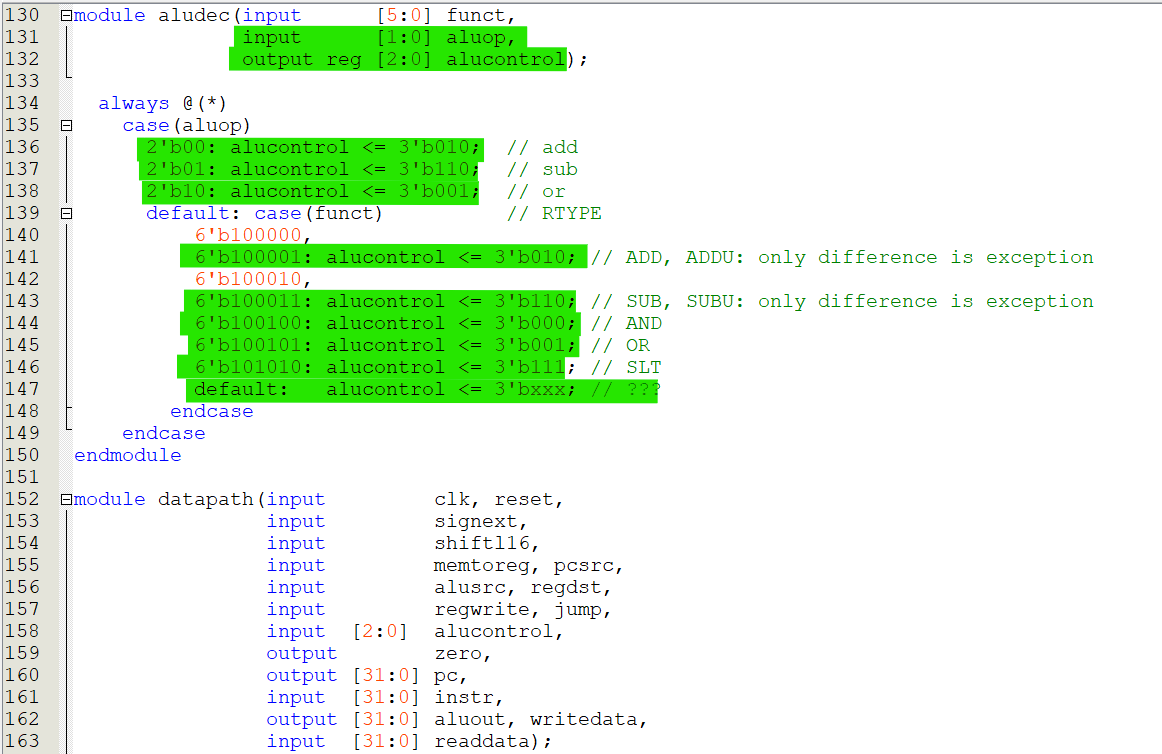


Fig. 11. mips.v\_aludec

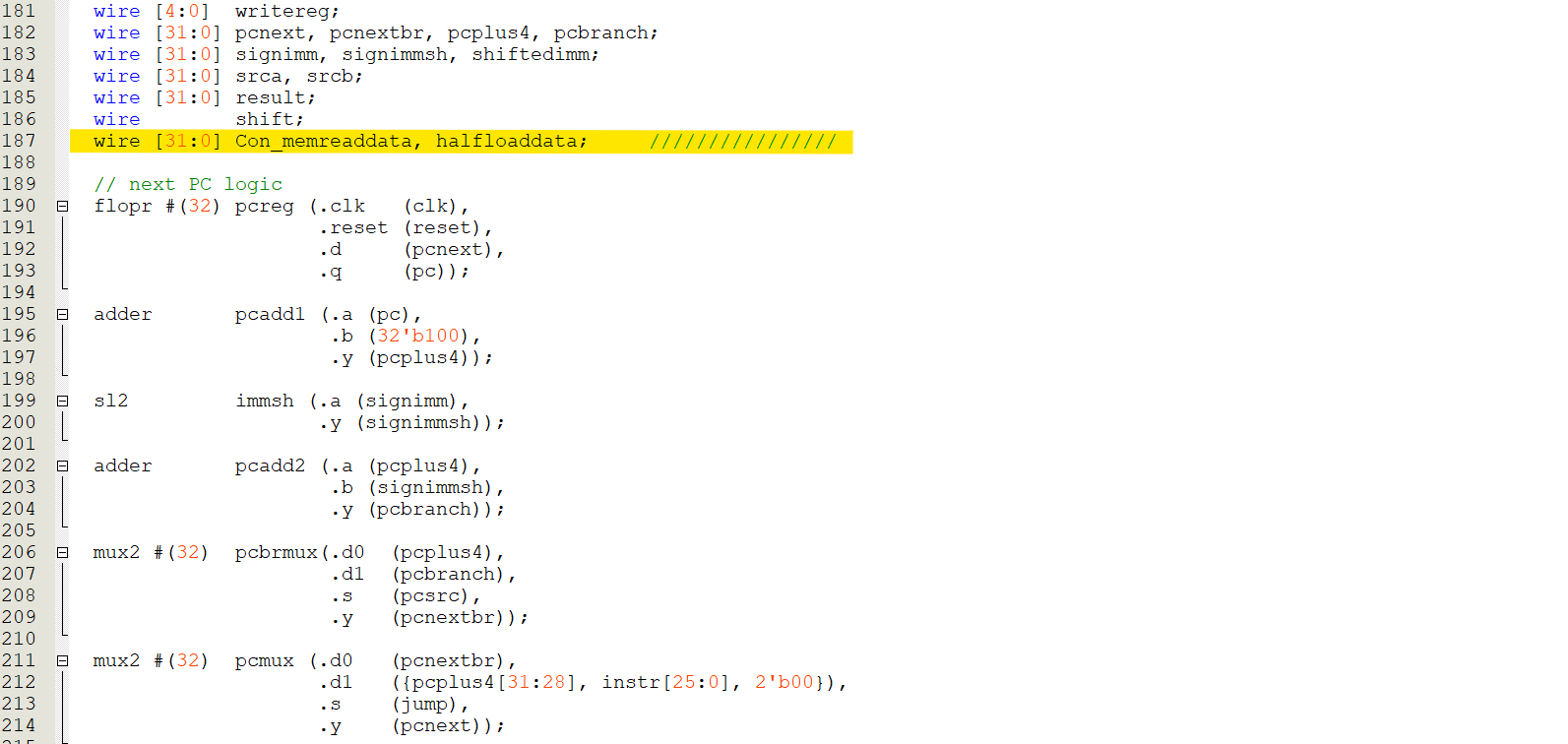
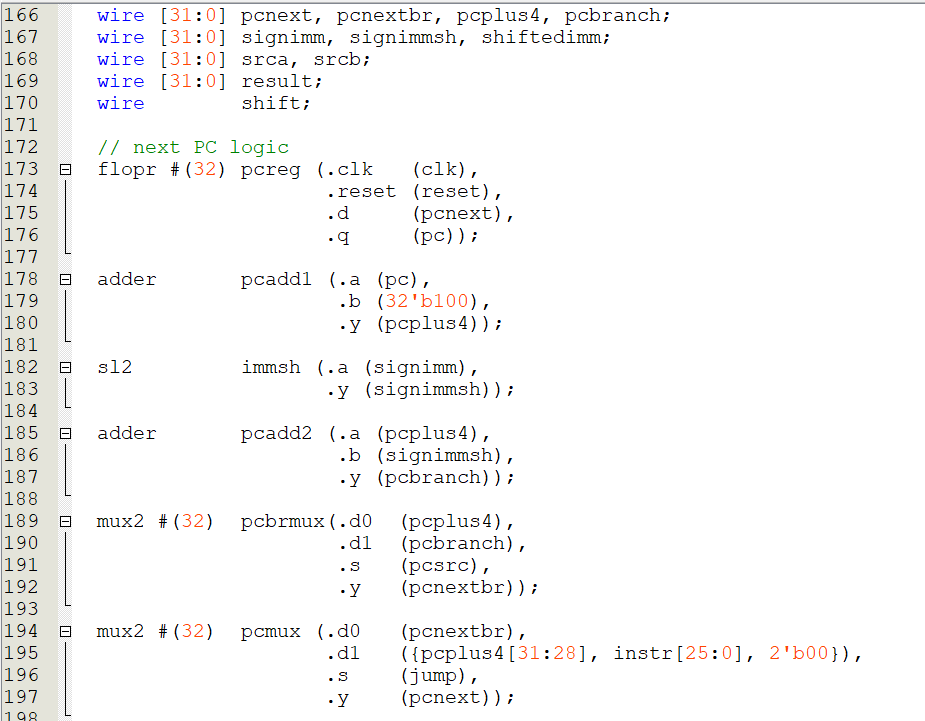


Fig. 12. mips.v\_datapath

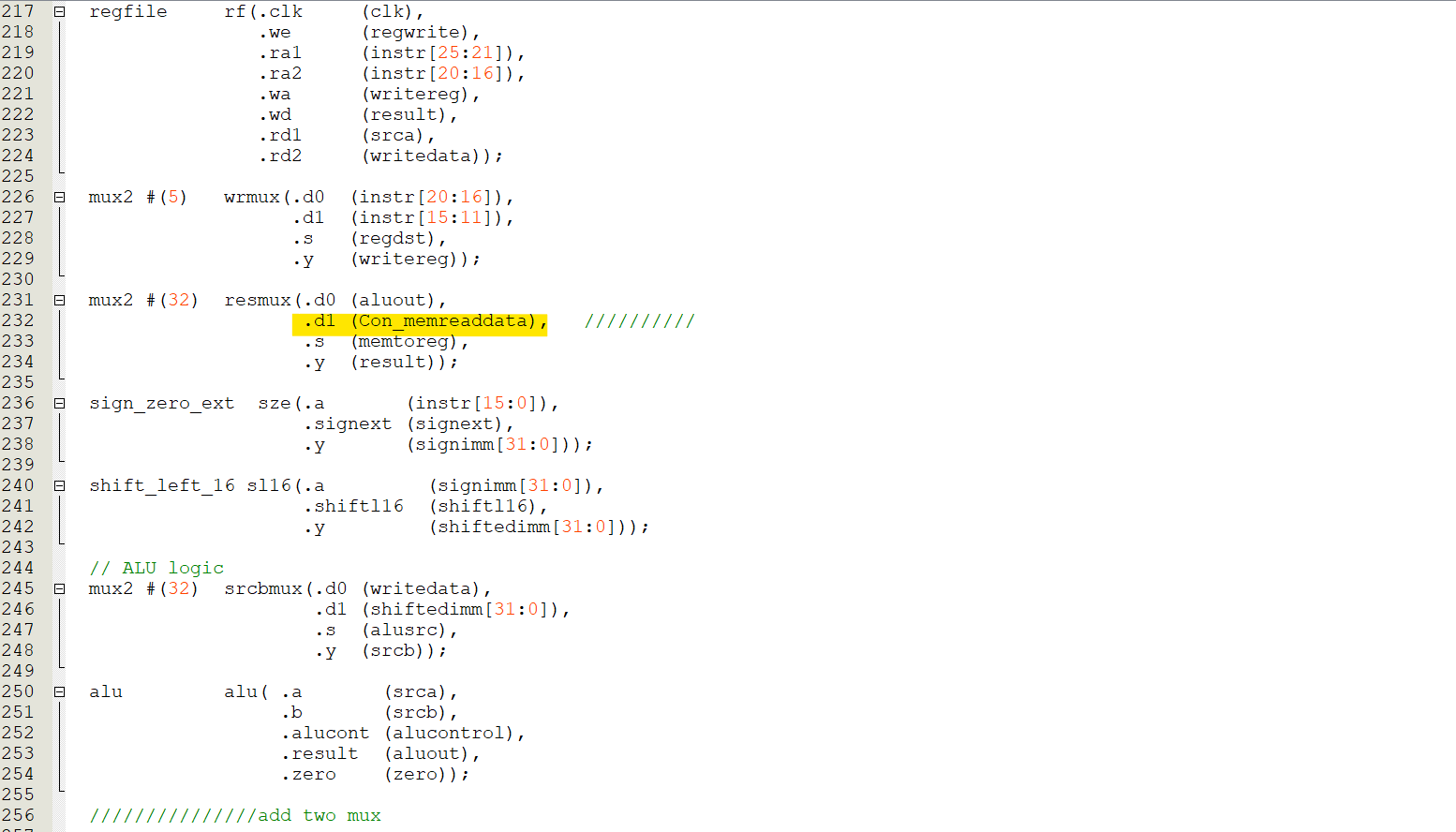
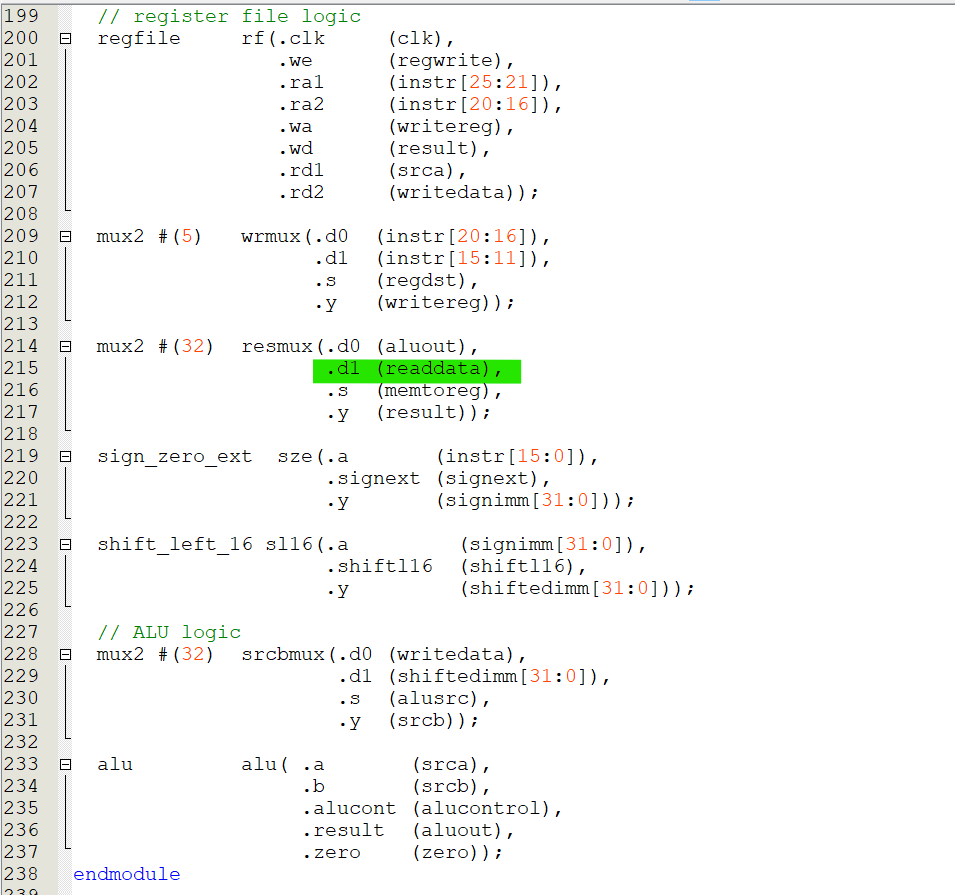


Fig. 13. mips.v\_datapath2

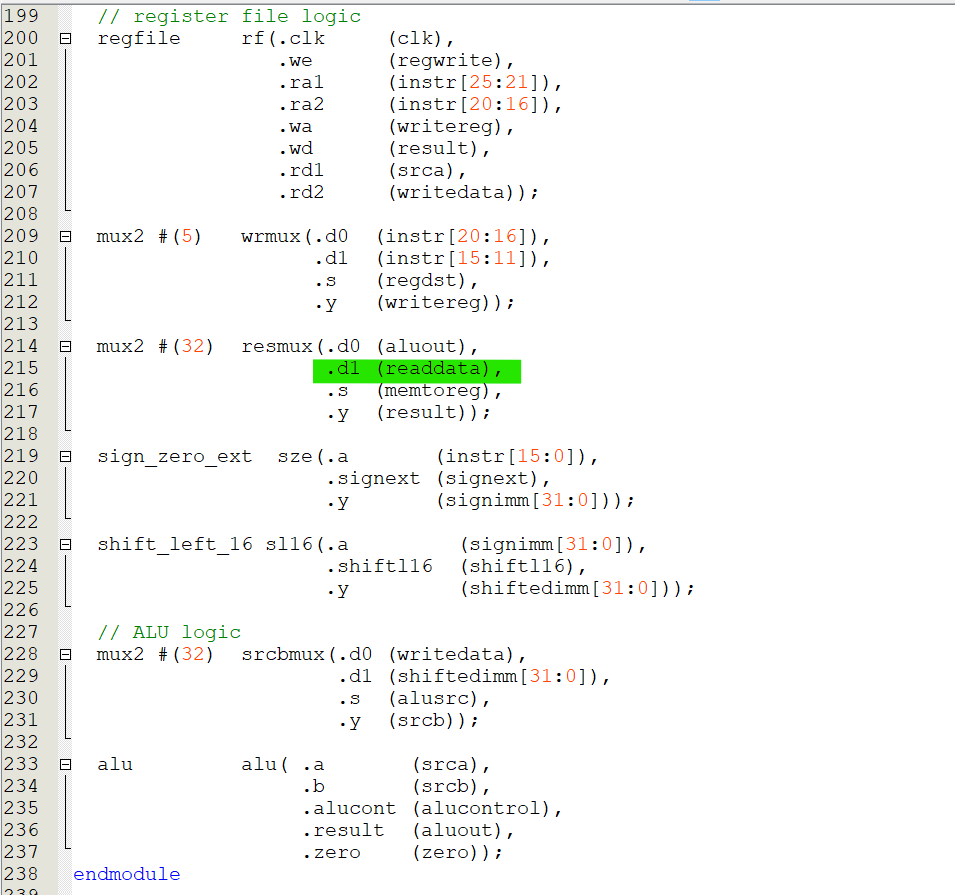


Fig. 14. mips.v\_datapath3

* 1. **NOR instruction**

For the R-type instruction, it depends on the function code to set the ‘alucontrol’ value. So, we start from the conditional output with ‘funct’ as input to draw the ASM chart.

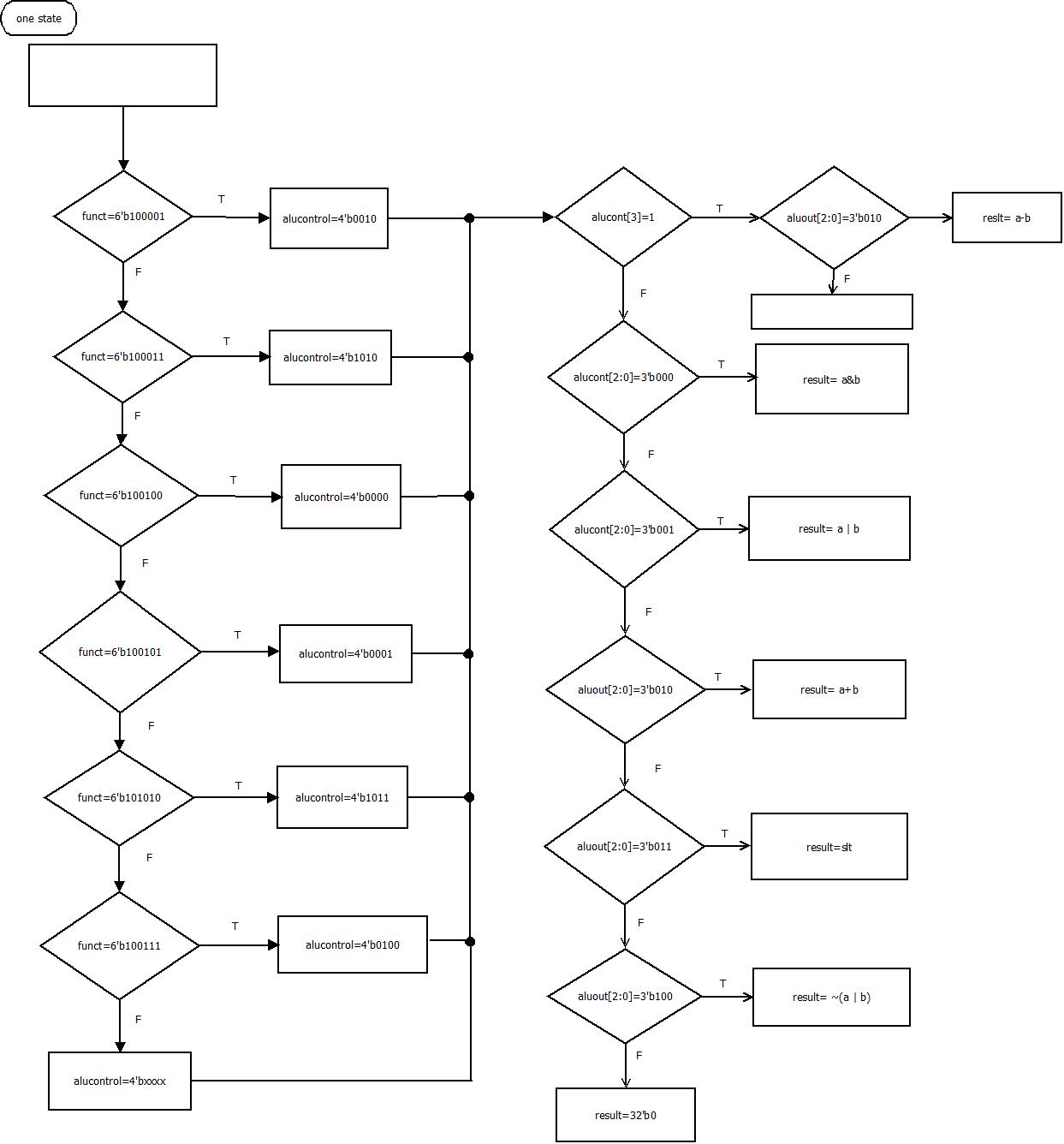


Fig. 15. ASM chart of nor instruction

* 1. **ANDI instruction**

Andi is an I-type of instruction, we need to add the corresponding opcode for ‘andi’ to the program and set the other value in the main decoder which will be used to execute the instruction. Besides, we need to add a conditional output with ‘aluop’ to let the ALU could execute ‘and’ algorithm by decoding ‘aluop’.

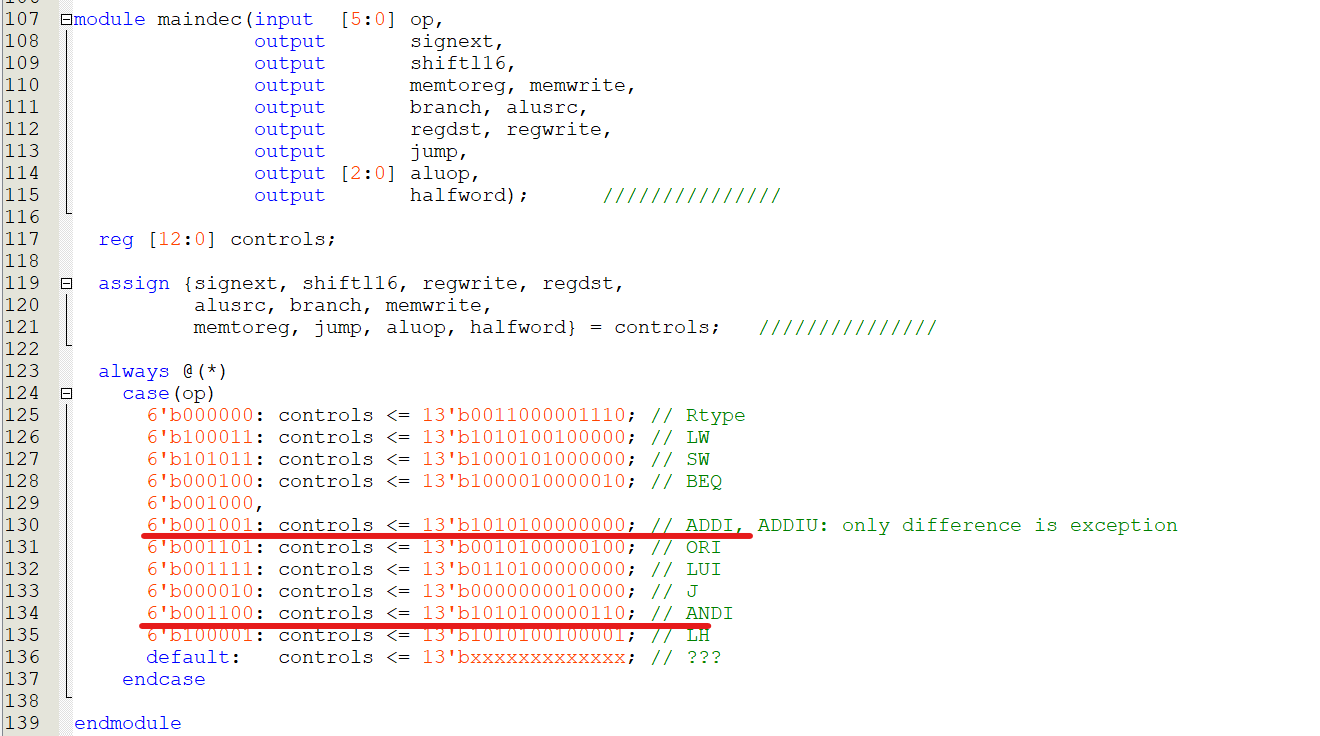


Fig. 16. How to set the controls value andi instruction

As shown in Fig. 16, the way we set the ‘controls’ value from the ‘andi’ instruction, is similar to the ‘add’ instruction, and the only difference is the ‘aluop’ value. Below, we will draw the ASM chart for the ani instruction starting from the conditional output of ‘controls’, and we do not include the result part since it had been mentioned in Fig. 15.

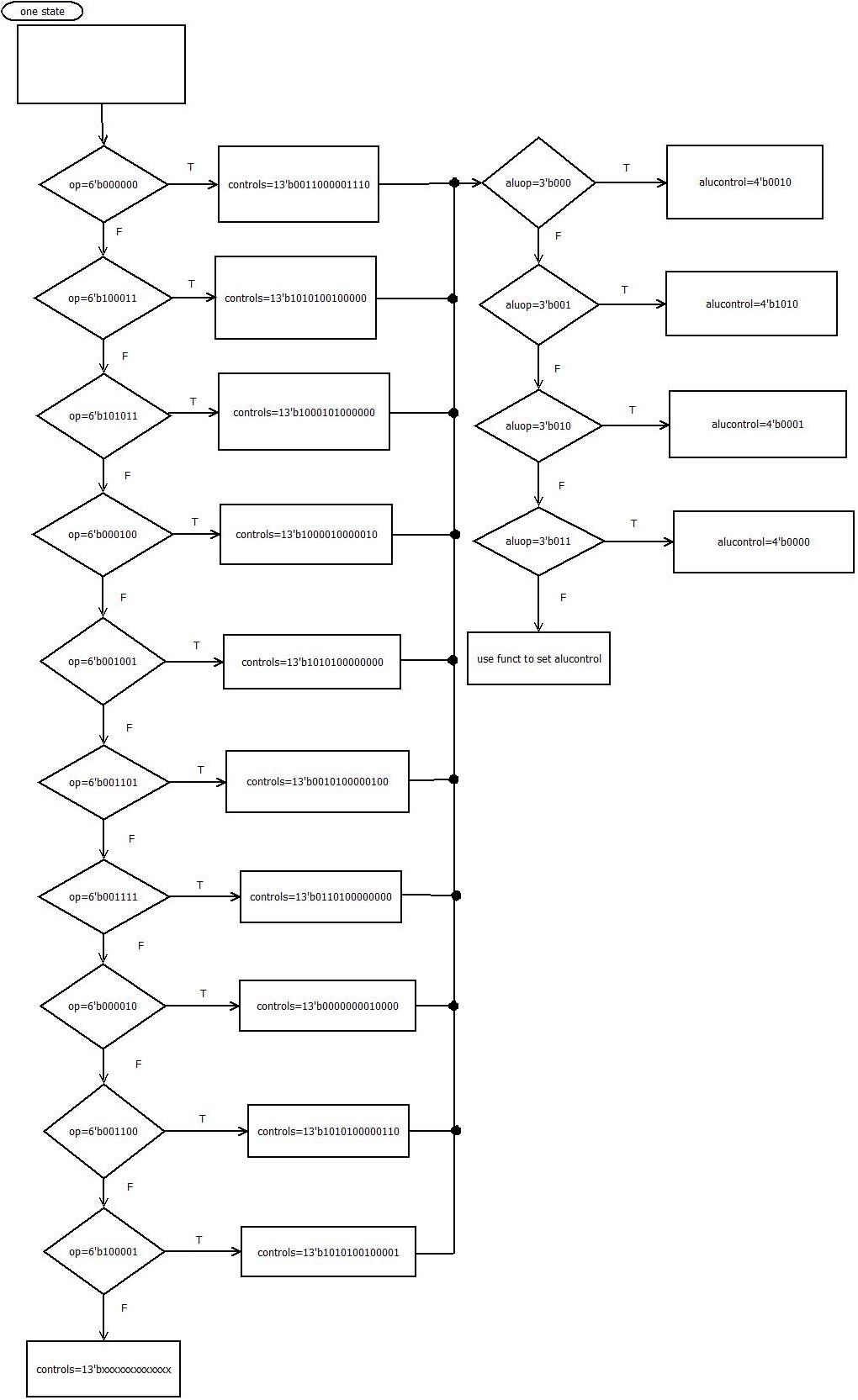


Fig. 17. ASM chart of andi instruction

* 1. **LH instruction**

LH instructions are the I-type of instruction, so we need to add the opcode of lh to the control unit, and we need to add an extra output signal in the controller to decide whether we load the upper half-word or the lower half-word. Besides, in the data path we need two 32-bit wires to represent the loaded half-word data and the loaded data (load word or load half). The block diagram of this part is shown below.

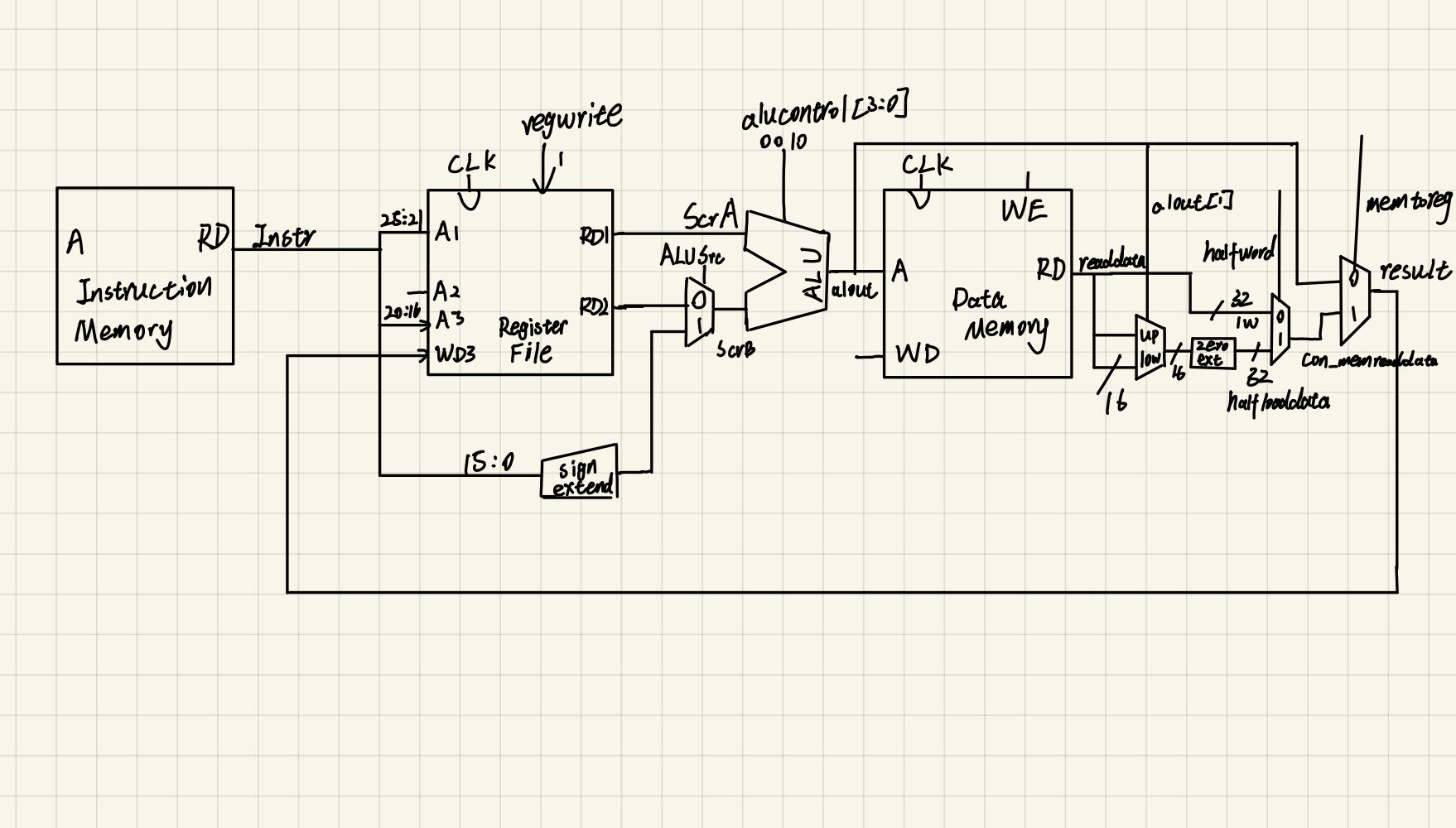


Fig. 18. Block diagram of lh

* 1. **Assembly code for testing**

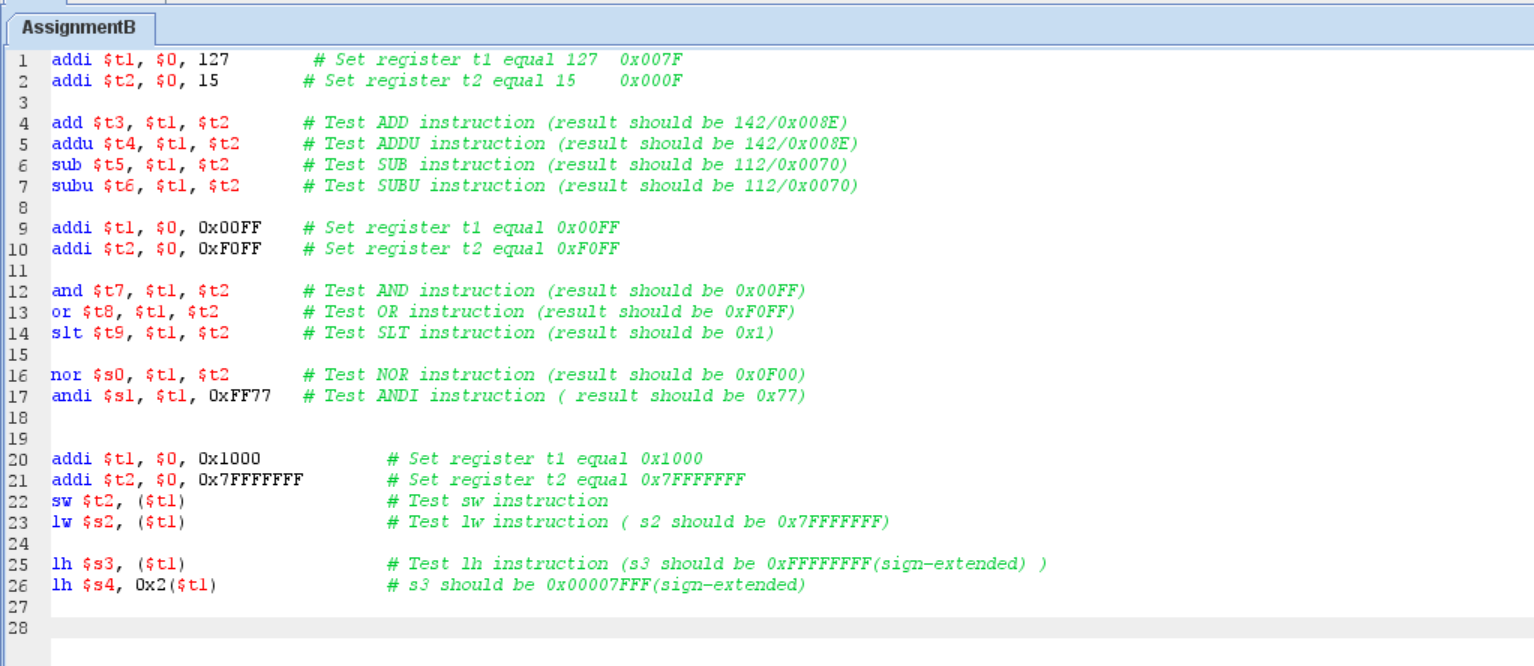


Fig. 19. MIPS assembly code for part B

* 1. **SignalTap logic analyser result**

**2.6.1 ADD, ADDU, SUB, SUBU, AND, OR, SLT, NOR and ANDI**

For the instructions we need to examine, some of them could directly be checked by observing ‘aluout’. Therefore, we put them together to check whether we could correctly implement the instructions. Besides, observing the ‘wd’ value could examine whether the data had been written back to the register file.

1. **Set t1=0x7F, t2=0x0F**

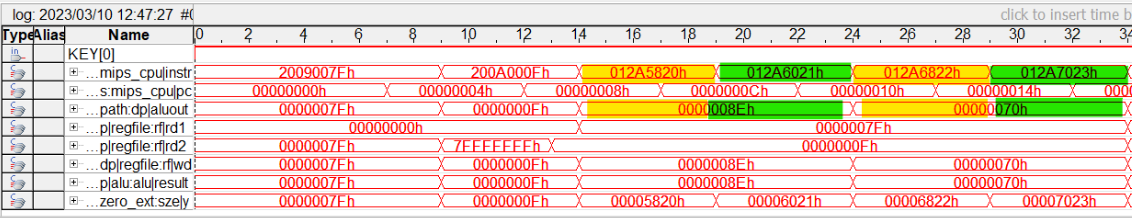


Fig. 20. ADD, ADDU, SUB and SUBU

Instruction 0x012A5820: ADD

Implement: t1+t2=0x8E

Instruction 0x012A6021: ADDU

Implement: t1+t2=0x8E

Instruction 0x012A6822: SUB

Implement: t1-t2=0x70

Instruction 0x012A7023: SUBU

Implement: t1-t2=0x70

1. **Set t1=0x00FF, t2=0xF0FF**

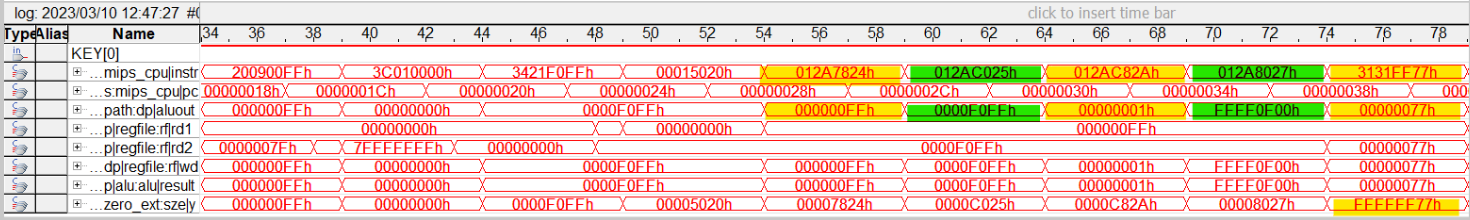


Fig. 21. AND, OR, SLT, NOR and ANDI

Instruction 0x012A7824: AND

Implement: t1&t2=0xFF

Instruction 0x012AC025: OR

Implement: t1|t2=0xF0FF

Instruction 0x012AC82A: SLT

Implement: (t1<t2) ?= 0x1

Instruction 0x 3131FF77: ANDI

Implement: t1(0x00FF) & 0xFF77=0x77

There is a zero extension with the immediate value so we could find the value with ‘zero\_ext’ port equal to 0x00000077. This is because we do not use sign extensions for ‘andi’ instruction. And we used the sign-extended value to do the logic operation.

**2.6.2 SW, LW and LH**

Firstly, set t1 equal to 0x1000 and t2 equal to 0x7FFFFFF.

1. **SW**

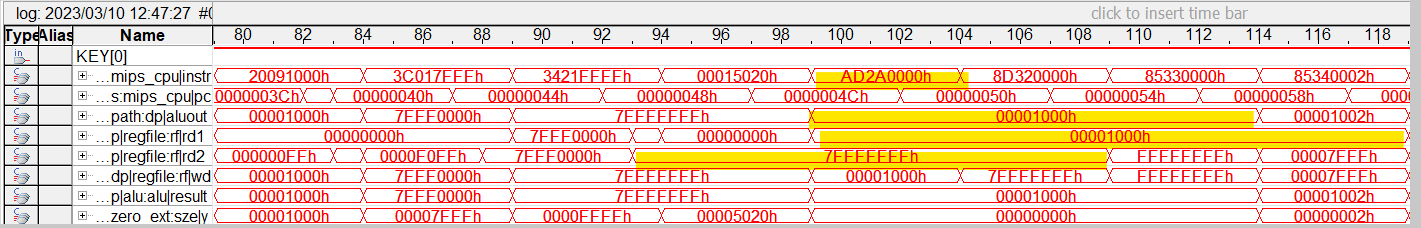


Fig. 22. SW instruction

As shown in Fig. 22, according to checking the value of rd1 and rd2 in the register file we could know the instruction will store the value in rd2(0x7FFFFFFF) to the memory location which is the value f rd2(0x00001000). Besides, observing the ‘aluout’ value we could find there is no offset value with the SW instruction. To verify whether successfully store the value, we will use the load instruction to check it.

1. **LW**

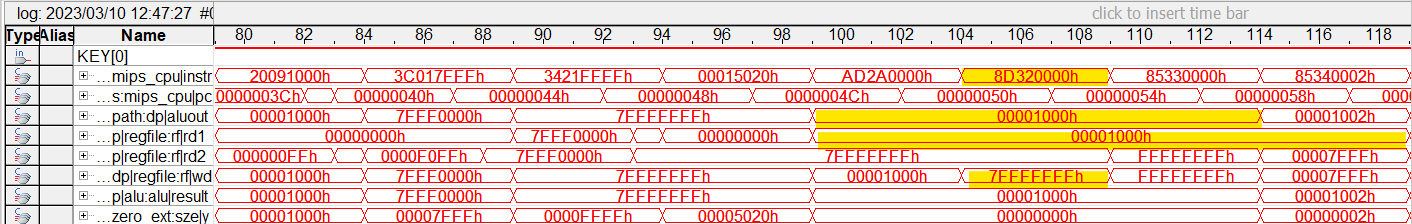


Fig. 23. LW instruction

For the load instruction, observe the value of ‘rd1’ we could know which memory address data we want to load, and the value of ‘aluout’ shows that the loaded value does not have an offset value. The critical point to know whether the data had been read from memory and written back to the register file is by observing the ‘wd’ value. And in this part, we also verified the value we read from the memory address of r1 are store the value we store by the SW instruction.

1. **LH**

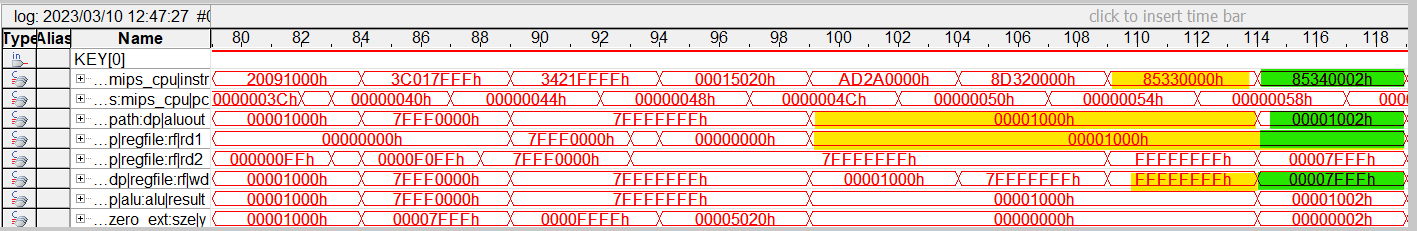


Fig. 24. LH instruction

From the previous result, we know the data value with the memory address of rd1 (0x00001000) is equal to 0x7FFFFFFF.

For the first instruction, aluout[1] is 0 so the LH instruction will load the sign extended lower half word. Therefore, from the value of ‘wed’ we could find the data written back the register file equal to 0xFFFFFFFF.

For the second instruction, aluout[1] is 1 so the LH instruction will load the sign extended upper half word. And from the value of ‘wd’ we could verify the write-back value equal to 0x00007FFF.