台科大

HW1

EE3005301 計算機組織

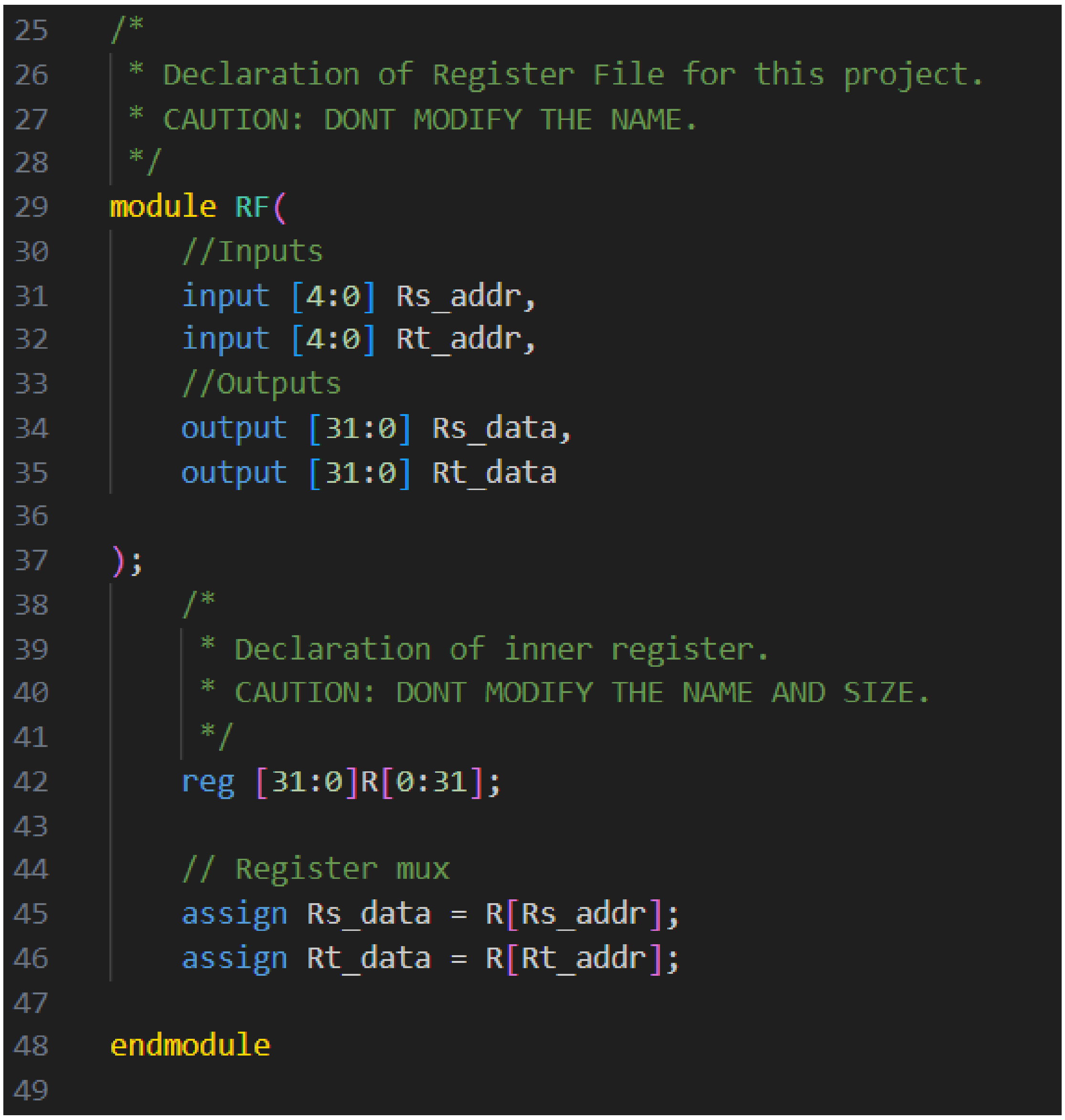
B11032006 周 柏宇

2024/3/13

内容

Modules

Register File



```
25  /*
26   * Declaration of Register File for this project.
27   * CAUTION: DONT MODIFY THE NAME.
28   */
29  module RF(
30      //Inputs
31      input [4:0] Rs_addr,
32      input [4:0] Rt_addr,
33      //Outputs
34      output [31:0] Rs_data,
35      output [31:0] Rt_data
36
37  );
38      /*
39       * Declaration of inner register.
40       * CAUTION: DONT MODIFY THE NAME AND SIZE.
41       */
42      reg [31:0]R[0:31];
43
44      // Register mux
45      assign Rs_data = R[Rs_addr];
46      assign Rt_data = R[Rt_addr];
47
48  endmodule
49
```

Line 29~37

Declaration and I/O.

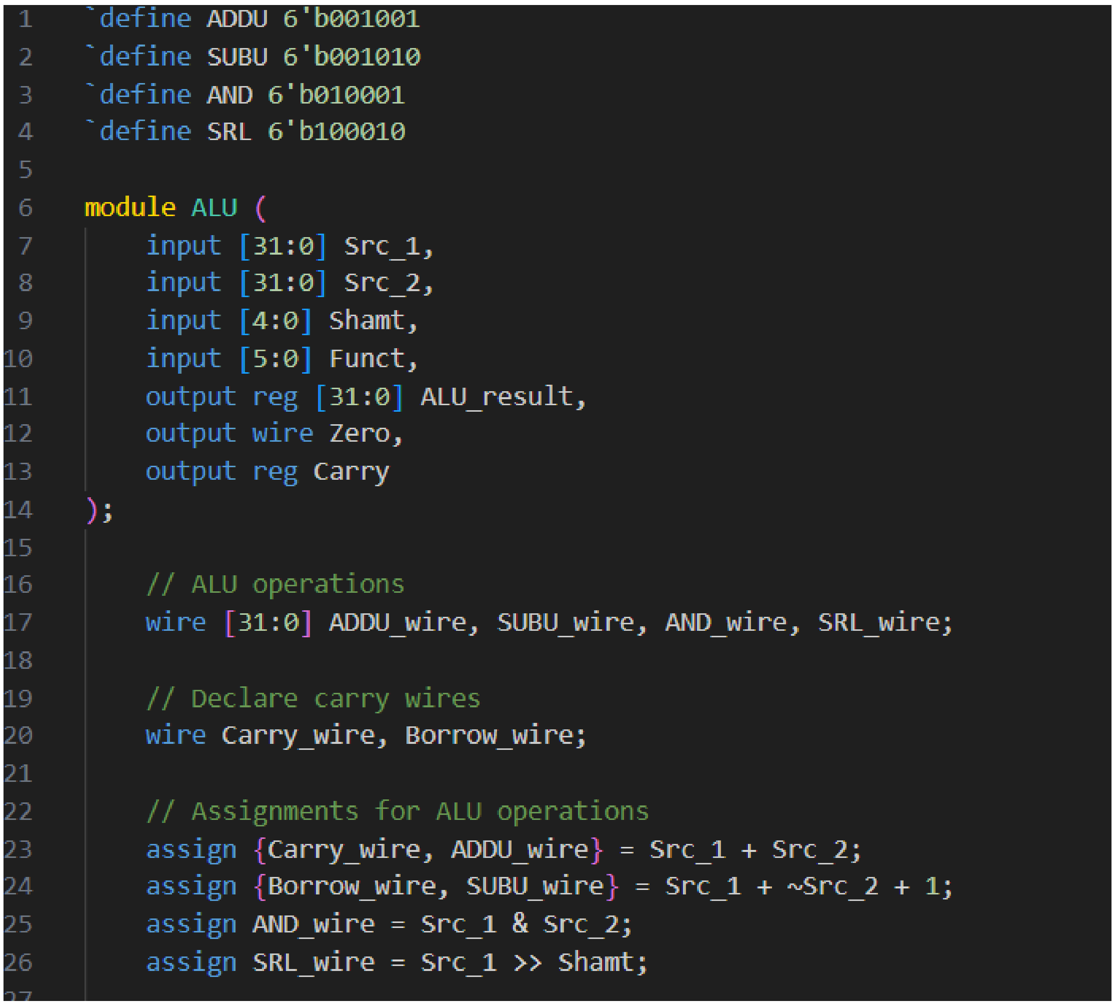
Line 42

Allocate 32 x 32-bits sram with named R.

Line 45~46

Rs_addr, Rt_addr are registers' index (5 bits = [0, 31]). Use square brackets to fetch data (32 bits = [0, 2^31]).

Arithmetic_Logic_Unit



```
 1  `define ADDU 6'b001001
 2  `define SUBU 6'b001010
 3  `define AND 6'b010001
 4  `define SRL 6'b100010
 5
 6  module ALU (
 7      input [31:0] Src_1,
 8      input [31:0] Src_2,
 9      input [4:0] Shamt,
10      input [5:0] Funct,
11      output reg [31:0] ALU_result,
12      output wire Zero,
13      output reg Carry
14  );
15
16      // ALU operations
17      wire [31:0] ADDU_wire, SUBU_wire, AND_wire, SRL_wire;
18
19      // Declare carry wires
20      wire Carry_wire, Borrow_wire;
21
22      // Assignments for ALU operations
23      assign {Carry_wire, ADDU_wire} = Src_1 + Src_2;
24      assign {Borrow_wire, SUBU_wire} = Src_1 + ~Src_2 + 1;
25      assign AND_wire = Src_1 & Src_2;
26      assign SRL_wire = Src_1 >> Shamt;
27
```

Line 1~4

Define constants.

Line 17, 20

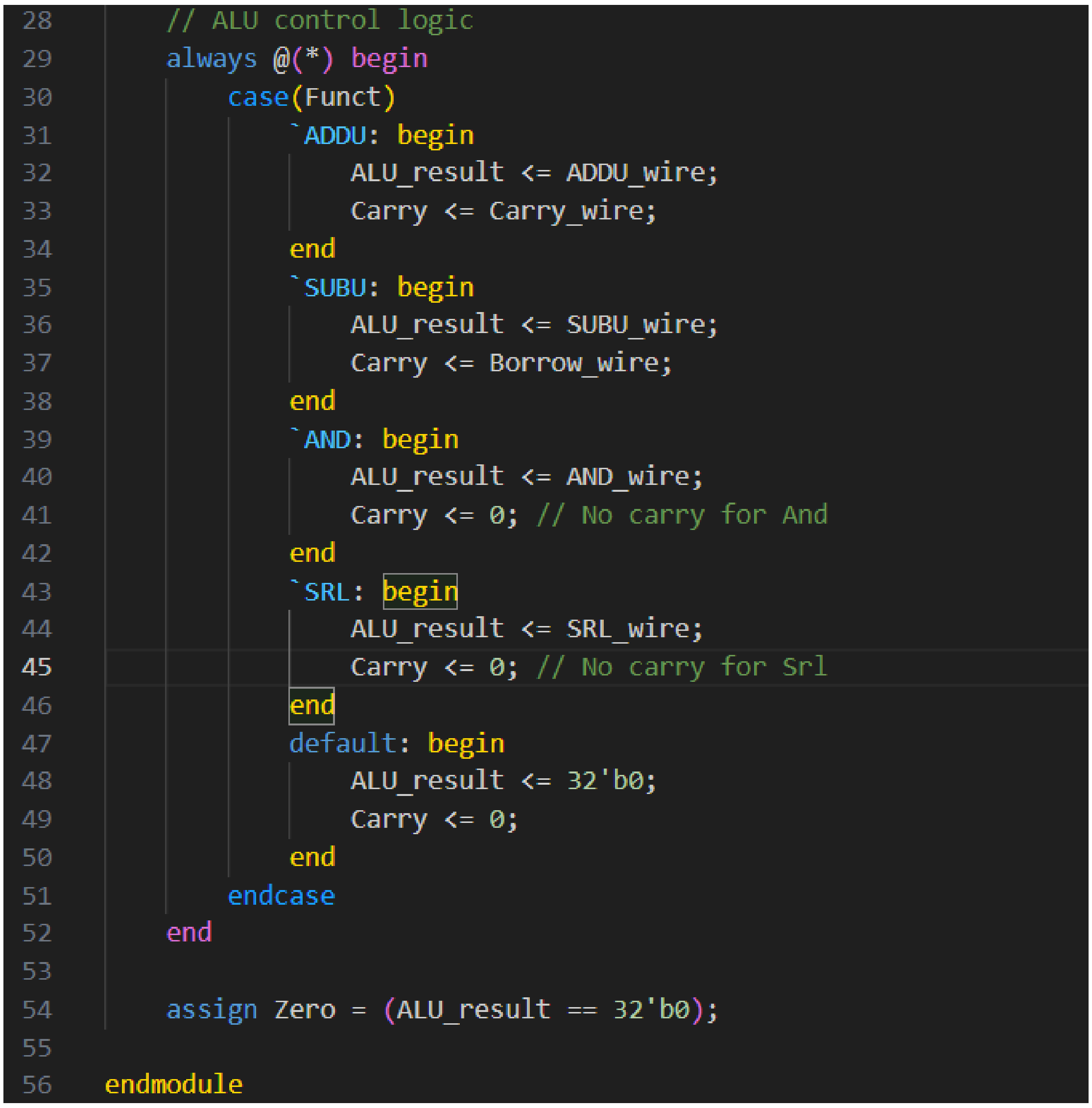
Use wires to store results.

Line 24

Use 2's complement to implement subtraction in order to get borrow bit.

Line 23~26

Use "assign" to get all the corresponding results simultaneously.



```
28      // ALU control logic
29      always @(*) begin
30          case(Funct)
31              `ADDU: begin
32                  ALU_result <= ADDU_wire;
33                  Carry <= Carry_wire;
34              end
35              `SUBU: begin
36                  ALU_result <= SUBU_wire;
37                  Carry <= Borrow_wire;
38              end
39              `AND: begin
40                  ALU_result <= AND_wire;
41                  Carry <= 0; // No carry for And
42              end
43              `SRL: begin
44                  ALU_result <= SRL_wire;
45                  Carry <= 0; // No carry for Srl
46              end
47              default: begin
48                  ALU_result <= 32'b0;
49                  Carry <= 0;
50              end
51          endcase
52      end
53
54      assign Zero = (ALU_result == 32'b0);
55
56  endmodule
```

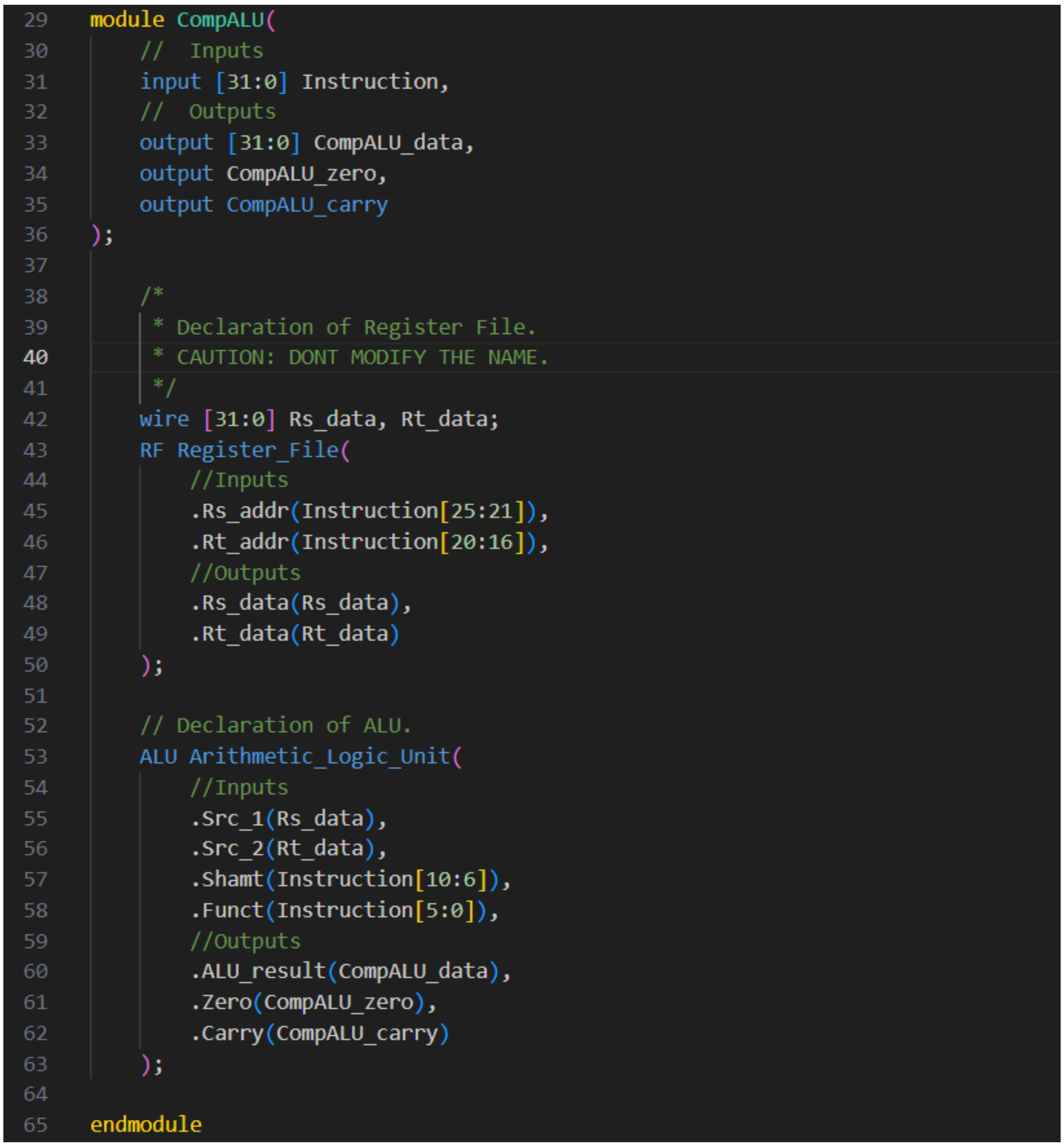
Line 30 ~ 51

Use conditional statement to decide which data should be output.

Line 54

Set Zero flag.

Complete ALU



```
29  module CompALU(
30      //  Inputs
31      input [31:0] Instruction,
32      //  Outputs
33      output [31:0] CompALU_data,
34      output CompALU_zero,
35      output CompALU_carry
36  );
37
38      /*
39       * Declaration of Register File.
40       * CAUTION: DONT MODIFY THE NAME.
41       */
42      wire [31:0] Rs_data, Rt_data;
43      RF Register_File(
44          //Inputs
45          .Rs_addr(Instruction[25:21]),
46          .Rt_addr(Instruction[20:16]),
47          //Outputs
48          .Rs_data(Rs_data),
49          .Rt_data(Rt_data)
50      );
51
52      // Declaration of ALU.
53      ALU Arithmetic_Logic_Unit(
54          //Inputs
55          .Src_1(Rs_data),
56          .Src_2(Rt_data),
57          .Shamt(Instruction[10:6]),
58          .Funct(Instruction[5:0]),
59          //Outputs
60          .ALU_result(CompALU_data),
61          .Zero(CompALU_zero),
62          .Carry(CompALU_carry)
63      );
64
65  endmodule
```

Line 42

Use wire to communicate between multiple components.

Line 43~50

Declare an RF instance with corresponding I/O.

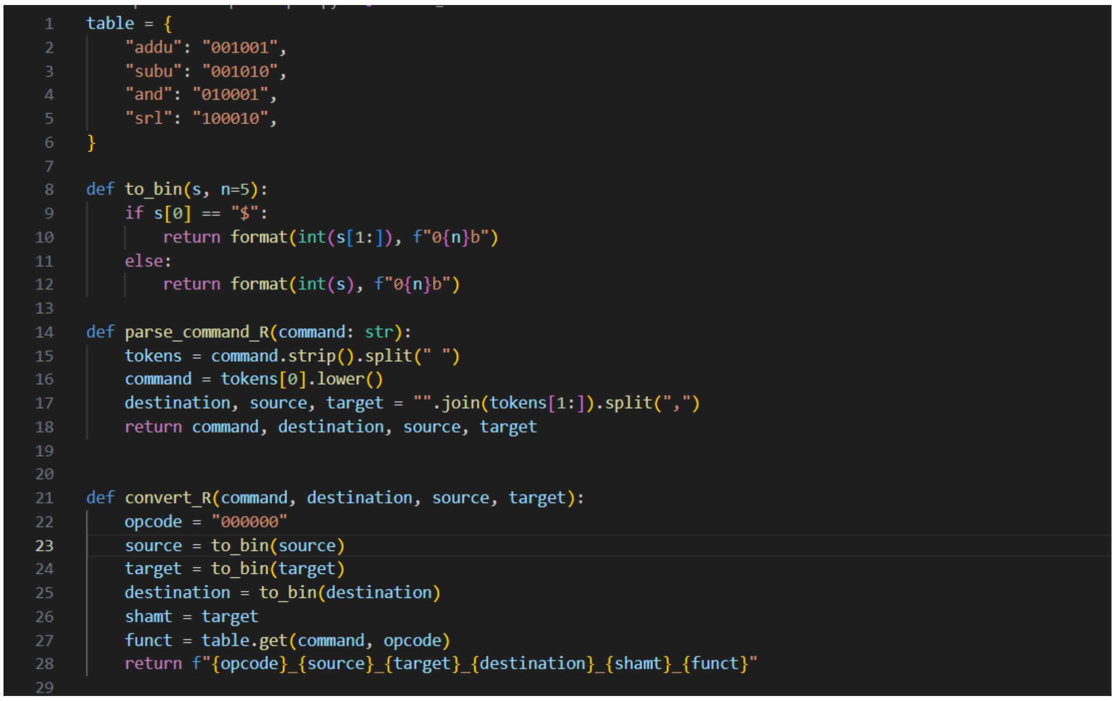
Line 53~63

Declare an ALU instance with corresponding I/O.

Test Commands

Helper Program

Simple Compiler



```
1   table = {
2       "addu": "001001",
3       "subu": "001010",
4       "and": "010001",
5       "srl": "100010",
6   }
7
8   def to_bin(s, n=5):
9       if s[0] == "$":
10          return format(int(s[1:]), f"0{n}b")
11      else:
12          return format(int(s), f"0{n}b")
13
14  def parse_command_R(command: str):
15      tokens = command.strip().split(" ")
16      command = tokens[0].lower()
17      destination, source, target = "".join(tokens[1:]).split(",")
18      return command, destination, source, target
19
20
21  def convert_R(command, destination, source, target):
22      opcode = "000000"
23      source = to_bin(source)
24      target = to_bin(target)
25      destination = to_bin(destination)
26      shamt = target
27      funct = table.get(command, opcode)
28      return f"{opcode}_{source}_{target}_{destination}_{shamt}_{funct}"
29
```

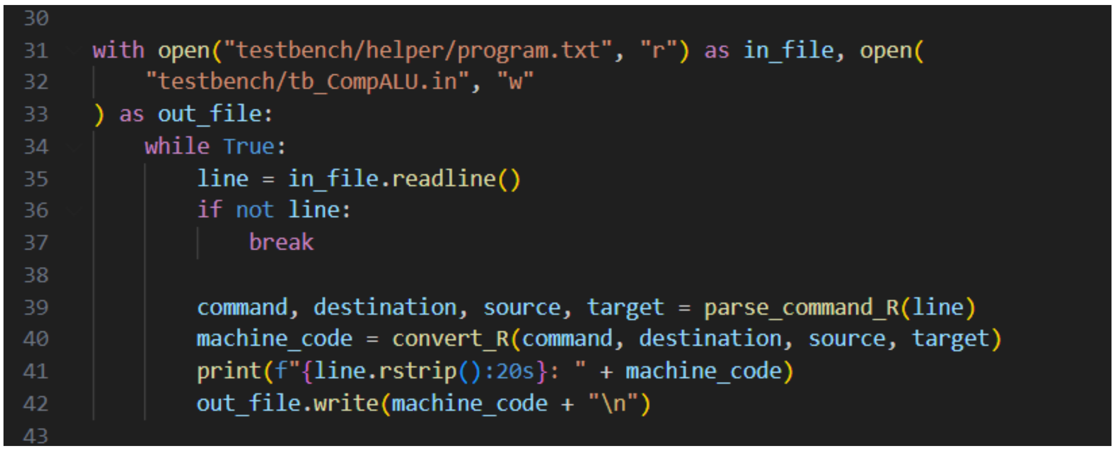
# This program convert assembly to machine code.

Line 1~6

Machine code look-up table.

Line 8~28

Simple parser. Since target and shamt will not be used at the same time, I set them to be the same value.

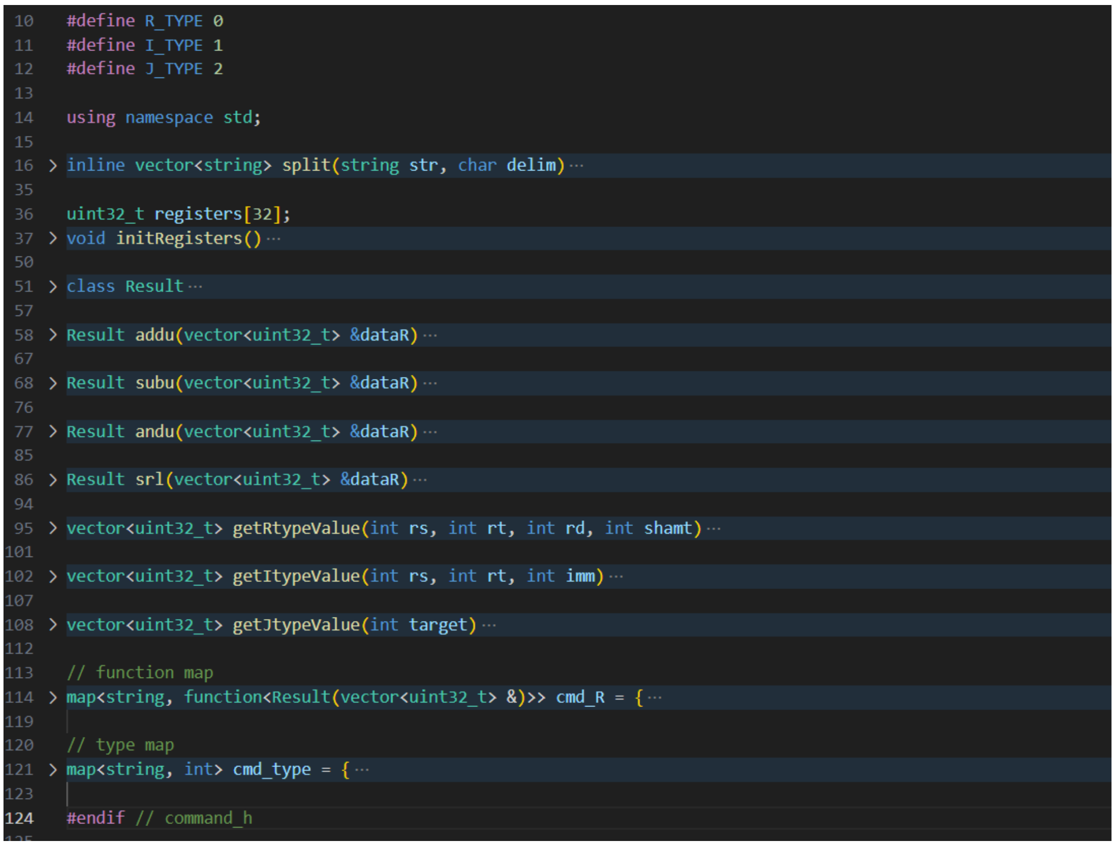


```
30
31  with open("testbench/helper/program.txt", "r") as in_file, open(
32      "testbench/tb_CompALU.in", "w"
33  ) as out_file:
34      while True:
35          line = in_file.readline()
36          if not line:
37              break
38
39          command, destination, source, target = parse_command_R(line)
40          machine_code = convert_R(command, destination, source, target)
41          print(f"{line.rstrip():20s}: " + machine_code)
42          out_file.write(machine_code + "\n")
43
```

Line 31~42

Procedure.

Verify Program



```
10    #define R_TYPE 0
11    #define I_TYPE 1
12    #define J_TYPE 2
13
14    using namespace std;
15
16  > inline vector<string> split(string str, char delim)···
35
36    uint32_t registers[32];
37  > void initRegisters()···
50
51  > class Result···
57
58  > Result addu(vector<uint32_t> &dataR)···
67
68  > Result subu(vector<uint32_t> &dataR)···
76
77  > Result andu(vector<uint32_t> &dataR)···
85
86  > Result srl(vector<uint32_t> &dataR)···
94
95  > vector<uint32_t> getRtypeValue(int rs, int rt, int rd, int shamt)···
101
102 > vector<uint32_t> getItypeValue(int rs, int rt, int imm)···
107
108 > vector<uint32_t> getJtypeValue(int target)···
112
113   // function map
114 > map<string, function<Result(vector<uint32_t> &)>> cmd_R = {···
119
120   // type map
121 > map<string, int> cmd_type = {···
123
124   #endif // command_h
```

# This program get program from machine code and output from modelsim to verify ans.

Line 16~49

Utility function. There is no need for 2 functions to create a new file, so they're here.

Line 51~56

Wrapper for value, zero, and carry.

Line 58~93

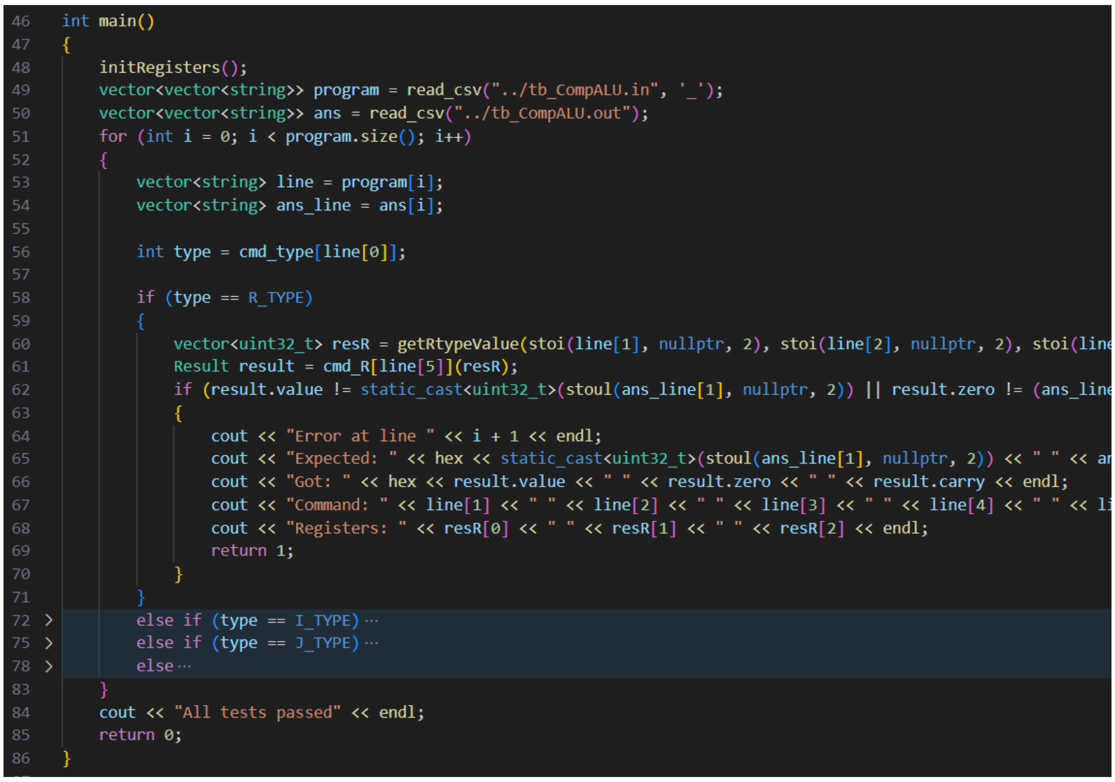
Functions implemented in this homework.

Line 95~111

Instruction's type to value decomposer. For simplicity, all types return the same data type.

Line 114~122

Maps.

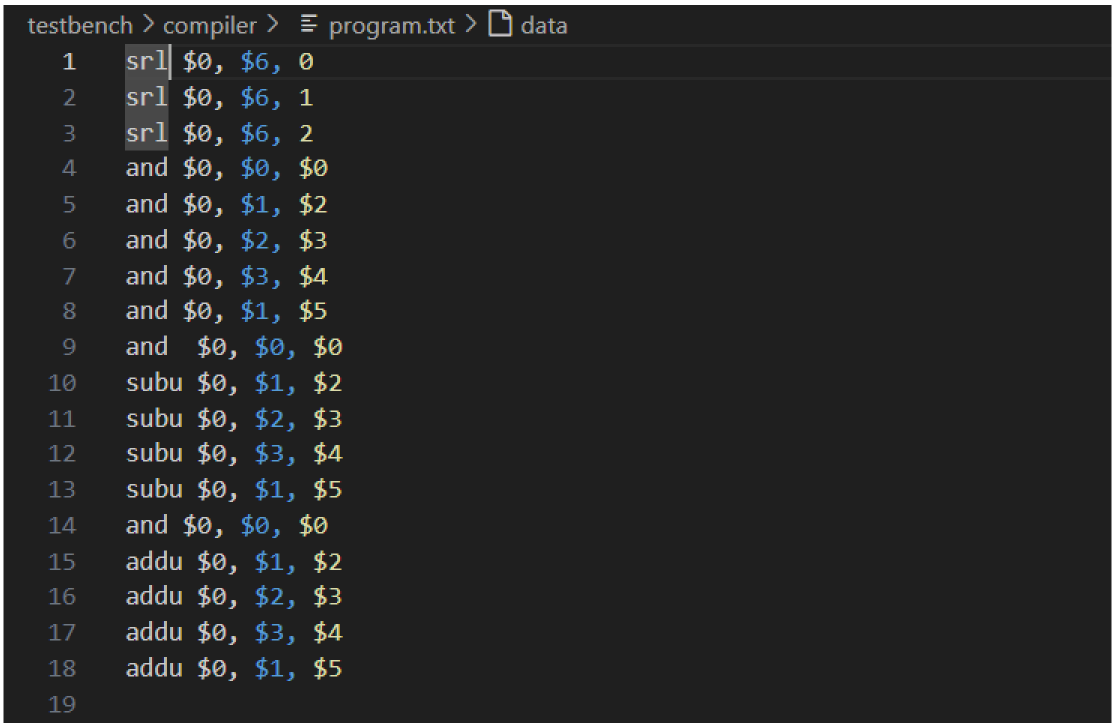


```cpp
46  int main()
47  {
48      initRegisters();
49      vector<vector<string>> program = read_csv("../tb_CompALU.in", '_');
50      vector<vector<string>> ans = read_csv("../tb_CompALU.out");
51      for (int i = 0; i < program.size(); i++)
52      {
53          vector<string> line = program[i];
54          vector<string> ans_line = ans[i];
55
56          int type = cmd_type[line[0]];
57
58          if (type == R_TYPE)
59          {
60              vector<uint32_t> resR = getRtypeValue(stoi(line[1], nullptr, 2), stoi(line[2], nullptr, 2), stoi(line
61              Result result = cmd_R[line[5]](resR);
62              if (result.value != static_cast<uint32_t>(stoul(ans_line[1], nullptr, 2)) || result.zero != (ans_line
63              {
64                  cout << "Error at line " << i + 1 << endl;
65                  cout << "Expected: " << hex << static_cast<uint32_t>(stoul(ans_line[1], nullptr, 2)) << " " << an
66                  cout << "Got: " << hex << result.value << " " << result.zero << " " << result.carry << endl;
67                  cout << "Command: " << line[1] << " " << line[2] << " " << line[3] << " " << line[4] << " " << li
68                  cout << "Registers: " << resR[0] << " " << resR[1] << " " << resR[2] << endl;
69                  return 1;
70              }
71          }
72 >        else if (type == I_TYPE) ···
75 >        else if (type == J_TYPE) ···
78 >        else ···
83      }
84      cout << "All tests passed" << endl;
85      return 0;
86  }
```

Line 48~84

Verify.

Test Commands



```
 1  srl $0, $6, 0
 2  srl $0, $6, 1
 3  srl $0, $6, 2
 4  and $0, $0, $0
 5  and $0, $1, $2
 6  and $0, $2, $3
 7  and $0, $3, $4
 8  and $0, $1, $5
 9  and  $0, $0, $0
10  subu $0, $1, $2
11  subu $0, $2, $3
12  subu $0, $3, $4
13  subu $0, $1, $5
14  and $0, $0, $0
15  addu $0, $1, $2
16  addu $0, $2, $3
17  addu $0, $3, $4
18  addu $0, $1, $5
19
```

#Note: it’s colored because I have csv extension.

Line 4, 9, 14

“and $0, $0, $0” is to zero the output acting as separator.

Line 1~3

Tests for srl.

Line 5~8

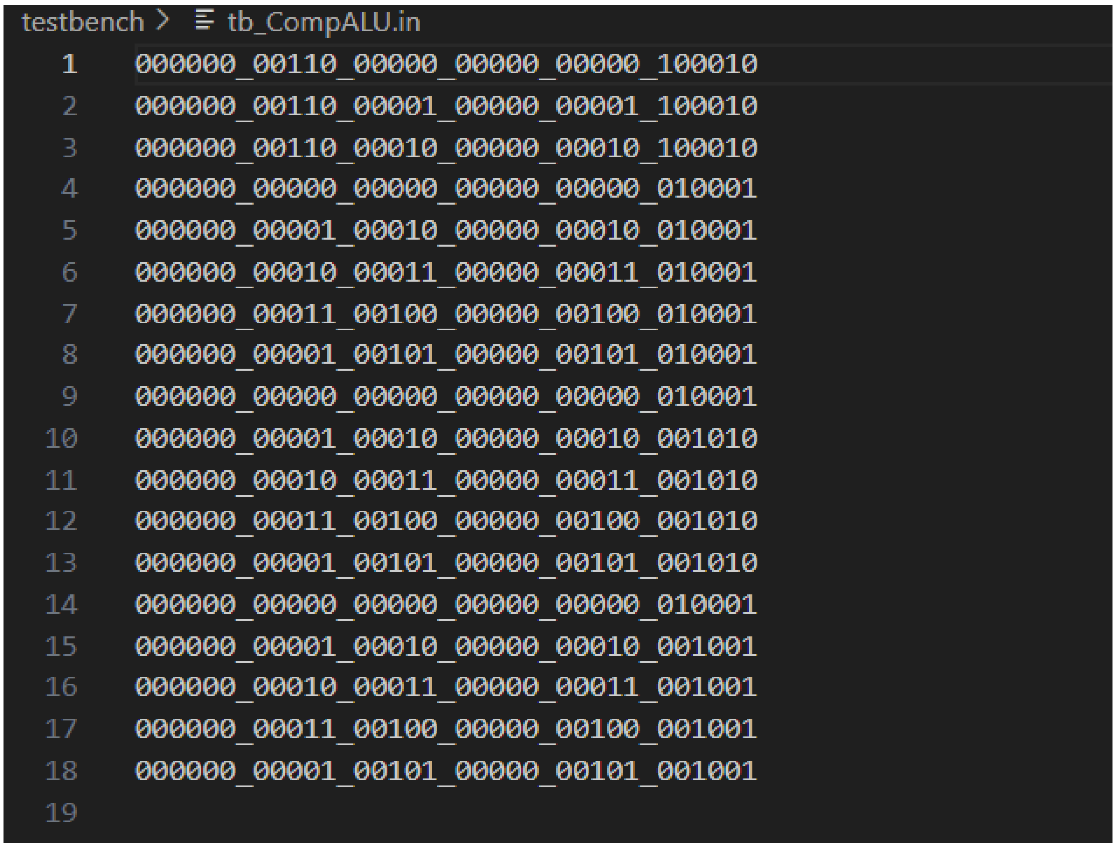
Tests for and.

Line 10~13

Tests for subu. Including both having borrow or not.

Line 15~18

Tests for addu. Including both overflowing or not.



testbench > tb_CompALU.in

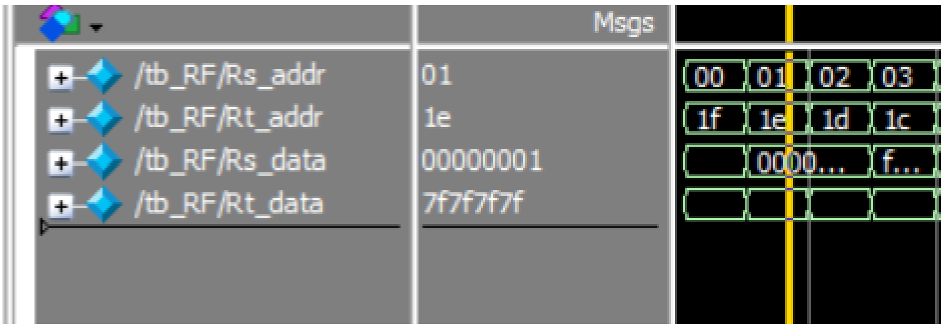
```
000000_00110_00000_00000_00000_100010
000000_00110_00001_00000_00001_100010
000000_00110_00010_00000_00010_100010
000000_00000_00000_00000_00000_010001
000000_00001_00010_00000_00010_010001
000000_00010_00011_00000_00011_010001
000000_00011_00100_00000_00100_010001
000000_00001_00101_00000_00101_010001
000000_00000_00000_00000_00000_010001
000000_00001_00010_00000_00010_001010
000000_00010_00011_00000_00011_001010
000000_00011_00100_00000_00100_001010
000000_00001_00101_00000_00101_001010
000000_00000_00000_00000_00000_010001
000000_00001_00010_00000_00010_001001
000000_00010_00011_00000_00011_001001
000000_00011_00100_00000_00100_001001
000000_00001_00101_00000_00101_001001

```

Corresponding machine code.

Test Results

RF

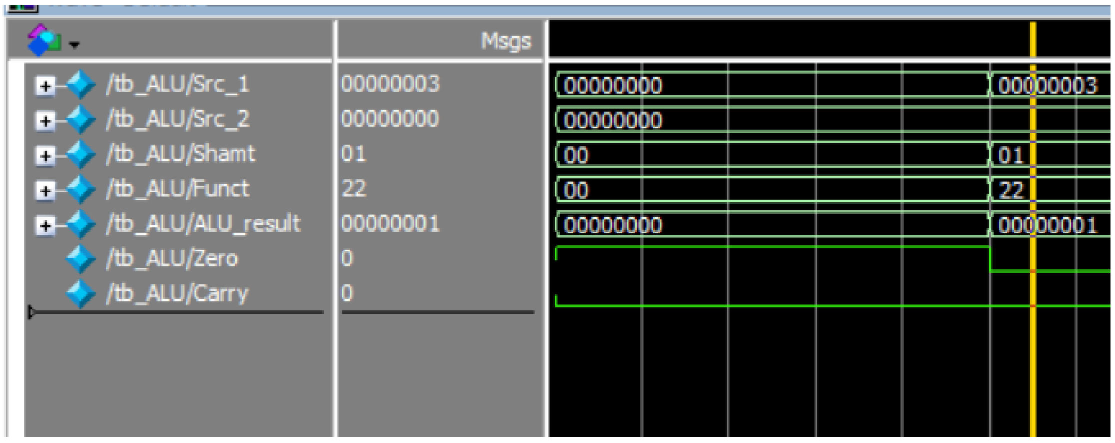


Rs_addr: 0x01 = 1_{10}, Rs_data: 0x000001

Rt_addr: 0x1e = 30_{10}, Rt_data: 0x7f7f7f7f

Initial register data is in ./testbench/RF.dat

ALU

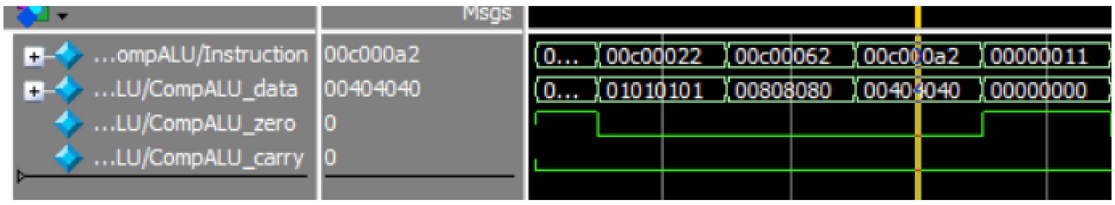


Funct = 100010 in this hw means srl

Shift right logically perform on Src_1 by Shamt bits: 0b11 >> 1 = 0b01 = 1_{10}.

CmpALU

Tests for srl



```
                            Msgs
...ompALU/Instruction   00c000a2   0...  00c00022  00c00062  00c000a2  00000011
...LU/CompALU_data      00404040   0...  01010101  00808080  00404040  00000000
...LU/CompALU_zero      0
...LU/CompALU_carry     0
```

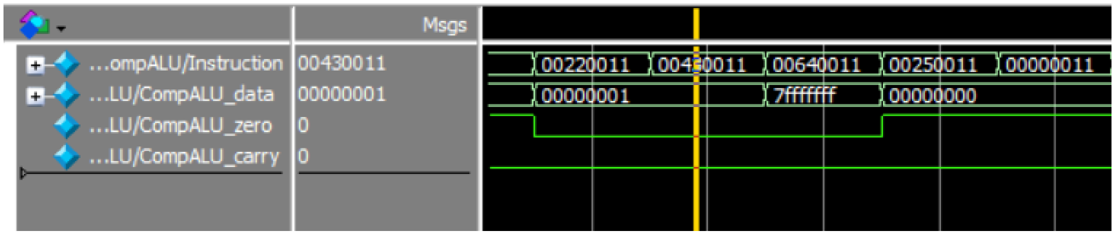
#Note and $0, $0, $0 in machine code is 0x00000011. It's acted as sperator.

srl $0, $6, 0: 0x0101_0101 >> 0x0 = 0x0101_0101 (equivalent as $6 / 2^0)

srl $0, $6, 1: 0x0101_0101 >> 0x1 = 0x0080_8080 (equivalent as $6 / 2^1)

srl $0, $6, 2: 0x0101_0101 >> 0x2 = 0x0040_4040 (equivalent as $6 / 2^2)

Tests for and



```
                            Msgs
...ompALU/Instruction   00430011   00220011  00430011  00640011  00250011  00000011
...LU/CompALU_data      00000001   00000001            7fffffff  00000000
...LU/CompALU_zero      0
...LU/CompALU_carry     0
```

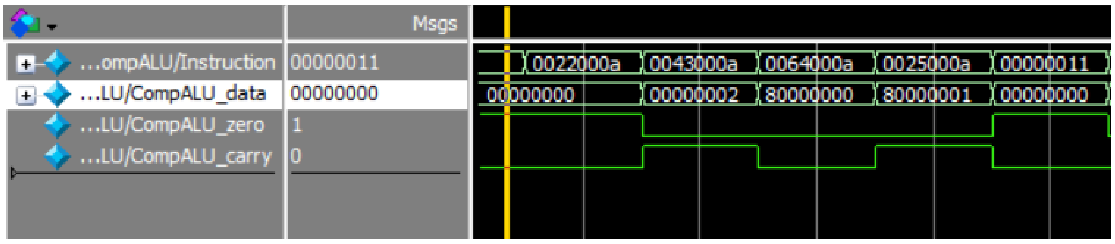
and $0, $1, $2: 0x0000_0001 & 0x0000_0001 = 0x0000_0001

and $0, $2, $3: 0x0000_0001 & 0xFFFF_FFFF = 0x0000_0001

and $0, $3, $4: 0xFFFF_FFFF & 0x7FFF_FFFF = 0x7FFF_FFFF

and $0, $1, $5: 0x0000_0001 & 0x8000_0000 = 0x0000_0000

Tests for subu



```
                            Msgs
...ompALU/Instruction   00000011   0022000a  0043000a  0064000a  0025000a  00000011
...LU/CompALU_data      00000000   00000000  00000002  80000000  80000001  00000000
...LU/CompALU_zero      1
...LU/CompALU_carry     0
```

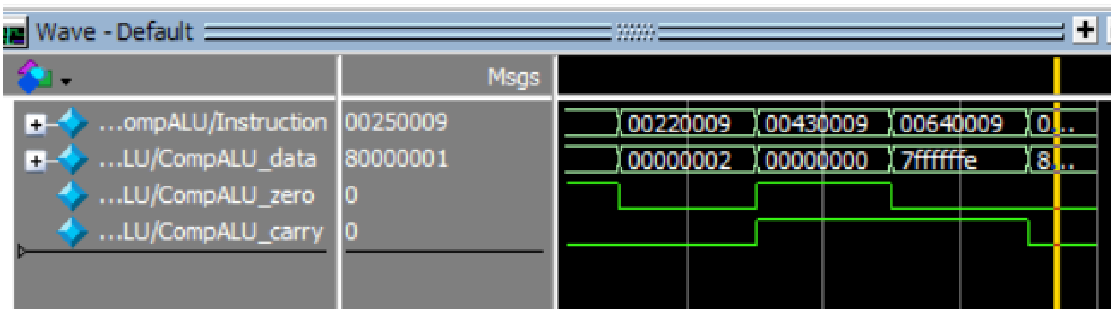
subu $0, $1, $2:0x0000_0001 - 0x0000_0001 = 0x0000_0000, Z = 1, C = 0

subu $0, $2, $3:0x0000_0001 - 0xFFFF_FFFF = 0x0000_0002, Z = 0, C = 1

subu $0, $3, $4:0xFFFF_FFFF - 0x7FFF_FFFF = 0x8000_0000, Z = 0, C = 0

subu $0, $1, $5:0x0000_0001 - 0x8000_0000 = 0x8000_0001, Z = 0, C = 1

Tests for addu



```
...ompALU/Instruction   00250009   00220009   00430009   00640009   0...
...LU/CompALU_data      80000001   00000002   00000000   7ffffffe   8...
...LU/CompALU_zero      0
...LU/CompALU_carry     0
```

addu $0, $1, $2:0x0000_0001 + 0x0000_0001 = 0x0000_0002

addu $0, $2, $3:0x0000_0001 + 0xFFFF_FFFF = 0x0000_0000,   Z = 1, C = 1

addu $0, $3, $4:0xFFFF_FFFF + 0x7FFF_FFFF = 0x7FFF_FFFE,   Z = 0, C = 1

addu $0, $1, $5:0x8000_0000 + 0x0000_0001 = 0x8000_0001

Conclusion

This homework is to make students have more understand about the basic principles of MIPS.

By employing distinct modules to implement various functionalities, akin to the top-down design method prevalent in programming. Breaking down the complex architecture into smaller, manageable structures enable students to tackle problems step by step.

Through the process, students not only gain insight into MIPS but also develop essential skills in modular design and problem-solving.