CSC3050 Project 2 Report

118010045 Cui Yuncong

March 2019

Contents

1 Background

1.1 MIPS Architecture

MIPS is a load/store architecture (also known as a register-register architecture); except for the load/store instructions used to access memory, all instructions operate on the registers. There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later.

1.2 MIPS Instructions

MIPS I has thirty-two 32-bit general-purpose registers (GPR). Register $0 is hardwired to zero and writes to it are discarded. Register $31 is the link register. For integer multiplication and division instructions, which run asynchronously from other instructions, a pair of 32-bit registers, HI and LO, are provided. There is a small set of instructions for copying data between the general-purpose registers and the HI/LO registers.

The program counter has 32 bits. The two low-order bits always contain zero since MIPS I instructions are 32 bits long and are aligned to their natural word boundaries.

Instructions are divided into three types: R, I and J. Every instruction starts with a 6-bit opcode. In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field; I-type instructions specify two registers and a 16-bit immediate value; J-type instructions follow the opcode with a 26-bit jump target.

The following are the three formats used for the core instruction set:

Type	-31-	format (bits)				-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Figure 1: MIPS Instruction Format

1.3 Verilog

Hardware description languages such as Verilog are similar to software programming languages because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment, and a non-blocking assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially written software programs to document and simulate electronic circuits.

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/-variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks,

and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

2 Project Description

2.1 Object

The Arithmetic and Logic Unit(ALU) is to do math co-processing. This program uses verilog to complete it.

2.2 Functions

The diagram has test bench. The CPU get information from the instruction and find the data flow of each instruction. However, in the ALU module, the input is sometimes not only rely on the instructions. Therefore, the test bench should also contain the value of relevant registers.

From test bench to the register, with different kind of instructions, send different things to the ALU module. At least show the value of these registers in the diagram. Other important value in the data flow can be also displayed.

The flag registers includes zero flag, negative flag and overflow flag. The negative flag will appear when doing subtraction with unsigned values or other arithmetic with signed values. In the additional part, there may be overflow and the overflow detection will better improve our ALU module. The zero flag can easily decide the changing of pc in the instructions like bne, beq.

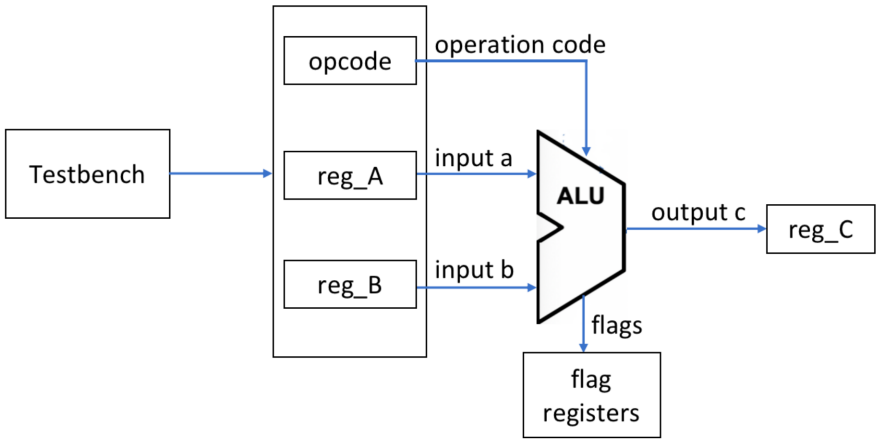


Figure 2: Block Diagram

3 Project Analysis

3.1 General Thought

Generally, MIPS assembly language has three kinds of instructions, which are R, I and J format. Different kinds instruction has different formats to write; therefore, the the general idea of the project is that separate the instructions into specific kinds and test them respectively.

3.2 Specific Steps

The ALU.v firstly preprocesses the instructions, extract the operation and function code.

Then simulate and process the instruction separately.

At last, display the test results.

4 Test

4.1 File Details

Source Code Filename: ALU.v test_ALU.v

4.2 Virtual System Configuration

VMware Workstation version: VMware Workstation 15.5 Pro

System: Linux version 5.0.0-31-generic (buildd@lcy01-amd64-010) (gcc version 8.3.0 (Ubuntu 8.3.0-6ubuntu1)) #33-Ubuntu SMP Mon Sep 30 18:51:59 UTC 2019

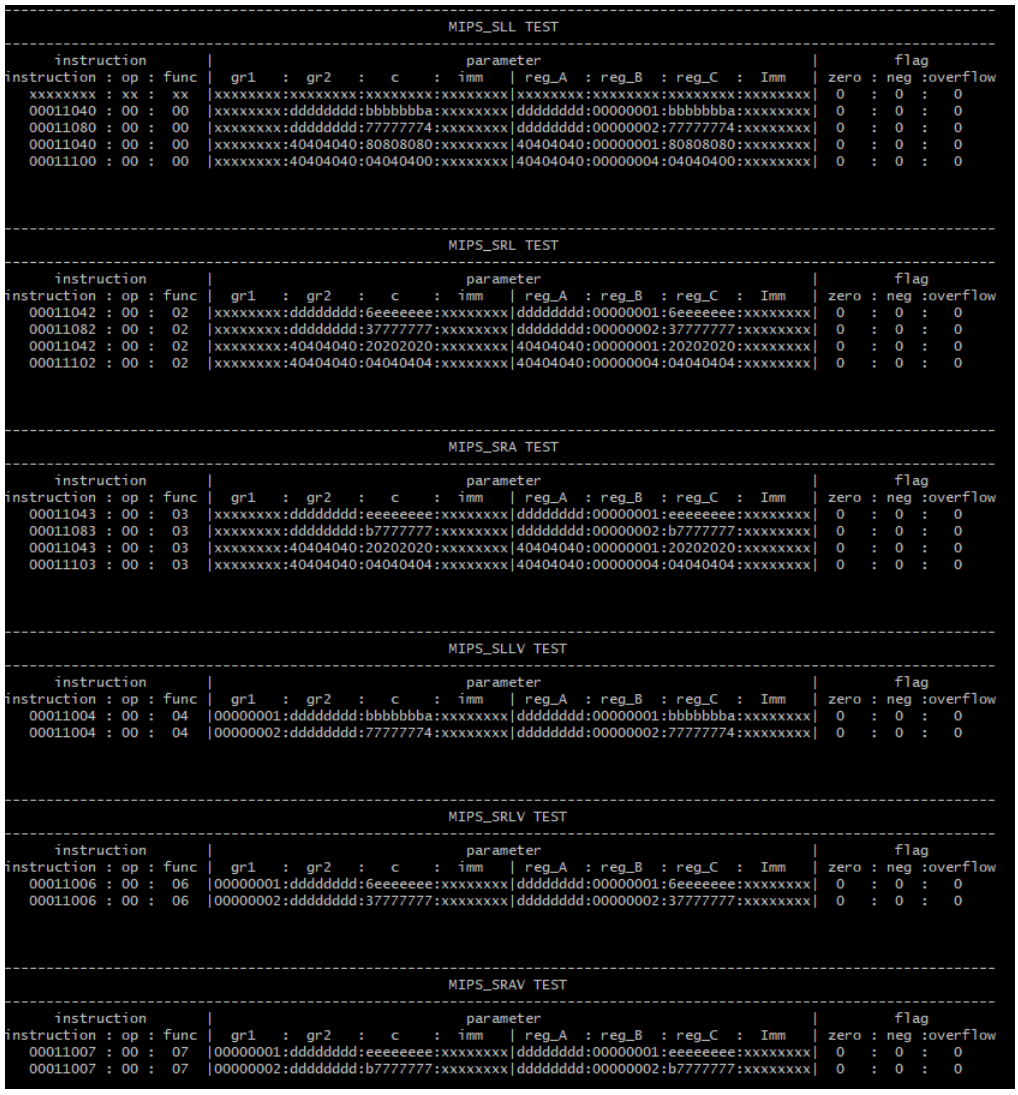
4.3 Compile and Execute

Compile: $ iverilog -o ALU ALU.v test_ALU.v

Demonstrate: $ vvp ALU

4.4 Result

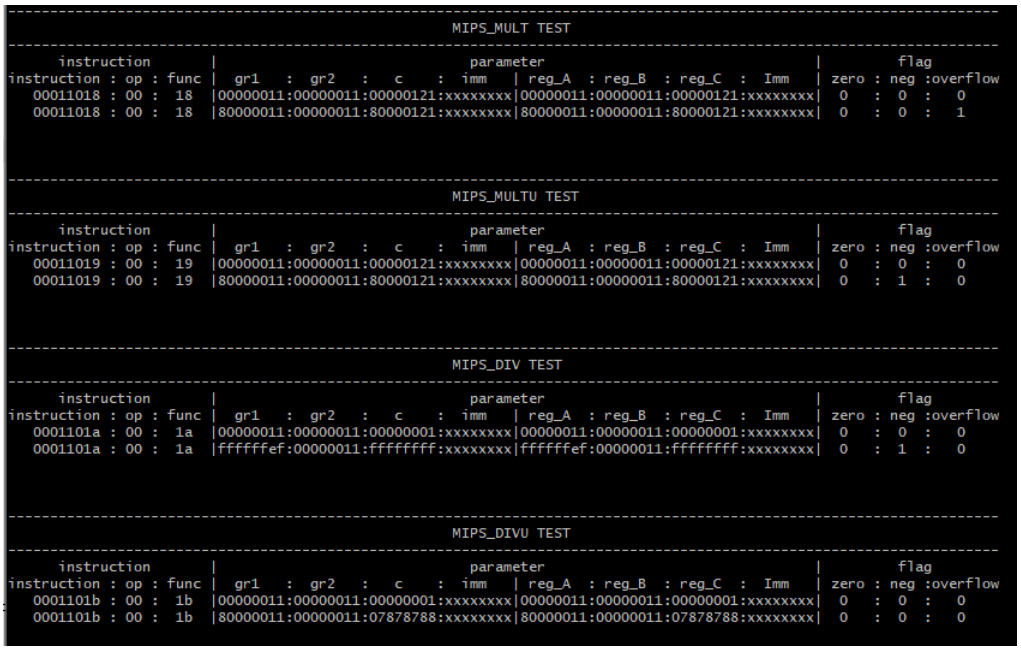
4.4.1 Shift Left and Shift Right



```
MIPS_SLL TEST
     instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    : imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   xxxxxxxx : xx :  xx  |xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx|xxxxxxxx:xxxxxxxx:xxxxxxxx:xxxxxxxx|  0   :  0  :   0
   00011040 : 00 :  00  |xxxxxxxx:ddddddddd:bbbbbbba:xxxxxxxx|ddddddddd:00000001:bbbbbbba:xxxxxxxx|  0   :  0  :   0
   00011080 : 00 :  00  |xxxxxxxx:ddddddddd:77777774:xxxxxxxx|ddddddddd:00000002:77777774:xxxxxxxx|  0   :  0  :   0
   00011040 : 00 :  00  |xxxxxxxx:40404040:80808080:xxxxxxxx|40404040:00000001:80808080:xxxxxxxx|  0   :  0  :   0
   00011100 : 00 :  00  |xxxxxxxx:40404040:04040400:xxxxxxxx|40404040:00000004:04040400:xxxxxxxx|  0   :  0  :   0

MIPS_SRL TEST
     instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    : imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011042 : 00 :  02  |xxxxxxxx:ddddddddd:6eeeeeee:xxxxxxxx|ddddddddd:00000001:6eeeeeee:xxxxxxxx|  0   :  0  :   0
   00011082 : 00 :  02  |xxxxxxxx:ddddddddd:37777777:xxxxxxxx|ddddddddd:00000002:37777777:xxxxxxxx|  0   :  0  :   0
   00011042 : 00 :  02  |xxxxxxxx:40404040:20202020:xxxxxxxx|40404040:00000001:20202020:xxxxxxxx|  0   :  0  :   0
   00011102 : 00 :  02  |xxxxxxxx:40404040:04040404:xxxxxxxx|40404040:00000004:04040404:xxxxxxxx|  0   :  0  :   0

MIPS_SRA TEST
     instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    : imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011043 : 00 :  03  |xxxxxxxx:ddddddddd:eeeeeeee:xxxxxxxx|ddddddddd:00000001:eeeeeeee:xxxxxxxx|  0   :  0  :   0
   00011083 : 00 :  03  |xxxxxxxx:ddddddddd:b7777777:xxxxxxxx|ddddddddd:00000002:b7777777:xxxxxxxx|  0   :  0  :   0
   00011043 : 00 :  03  |xxxxxxxx:40404040:20202020:xxxxxxxx|40404040:00000001:20202020:xxxxxxxx|  0   :  0  :   0
   00011103 : 00 :  03  |xxxxxxxx:40404040:04040404:xxxxxxxx|40404040:00000004:04040404:xxxxxxxx|  0   :  0  :   0

MIPS_SLLV TEST
     instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    : imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011004 : 00 :  04  |00000001:ddddddddd:bbbbbbba:xxxxxxxx|ddddddddd:00000001:bbbbbbba:xxxxxxxx|  0   :  0  :   0
   00011004 : 00 :  04  |00000002:ddddddddd:77777774:xxxxxxxx|ddddddddd:00000002:77777774:xxxxxxxx|  0   :  0  :   0

MIPS_SRLV TEST
     instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    : imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011006 : 00 :  06  |00000001:ddddddddd:6eeeeeee:xxxxxxxx|ddddddddd:00000001:6eeeeeee:xxxxxxxx|  0   :  0  :   0
   00011006 : 00 :  06  |00000002:ddddddddd:37777777:xxxxxxxx|ddddddddd:00000002:37777777:xxxxxxxx|  0   :  0  :   0

MIPS_SRAV TEST
     instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    : imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011007 : 00 :  07  |00000001:ddddddddd:eeeeeeee:xxxxxxxx|ddddddddd:00000001:eeeeeeee:xxxxxxxx|  0   :  0  :   0
   00011007 : 00 :  07  |00000002:ddddddddd:b7777777:xxxxxxxx|ddddddddd:00000002:b7777777:xxxxxxxx|  0   :  0  :   0
```

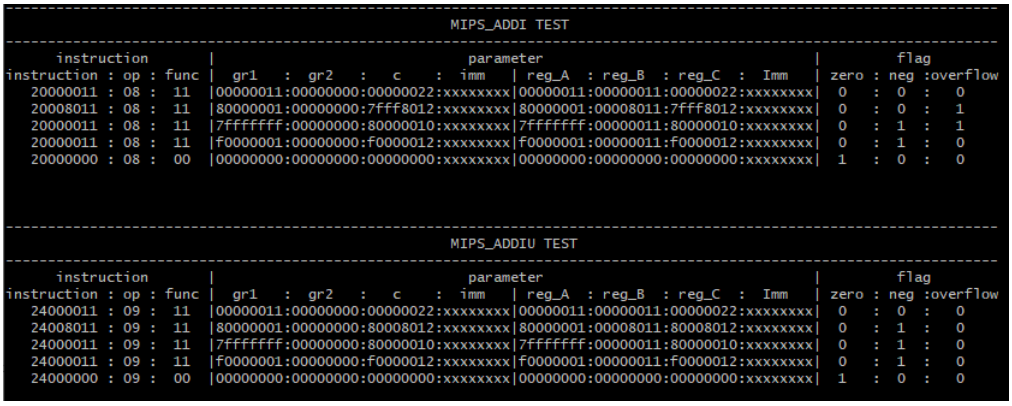
Figure 3: sll, sllv, srl, srlv, sra, srav

4.4.2 Multiply and Divide



```
MIPS_MULT TEST
     instruction      |                                parameter                                |          flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011018 : 00 :  18  |00000011:00000011:00000121:xxxxxxxx|00000011:00000011:00000121:xxxxxxxx|  0   :  0  :   0
   00011018 : 00 :  18  |80000011:00000011:80000121:xxxxxxxx|80000011:00000011:80000121:xxxxxxxx|  0   :  0  :   1

MIPS_MULTU TEST
     instruction      |                                parameter                                |          flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011019 : 00 :  19  |00000011:00000011:00000121:xxxxxxxx|00000011:00000011:00000121:xxxxxxxx|  0   :  0  :   0
   00011019 : 00 :  19  |80000011:00000011:80000121:xxxxxxxx|80000011:00000011:80000121:xxxxxxxx|  0   :  1  :   0

MIPS_DIV TEST
     instruction      |                                parameter                                |          flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   0001101a : 00 :  1a  |00000011:00000011:00000001:xxxxxxxx|00000011:00000011:00000001:xxxxxxxx|  0   :  0  :   0
   0001101a : 00 :  1a  |fffffef:00000011:ffffffff:xxxxxxxx|fffffef:00000011:ffffffff:xxxxxxxx|  0   :  1  :   0

MIPS_DIVU TEST
     instruction      |                                parameter                                |          flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   0001101b : 00 :  1b  |00000011:00000011:00000001:xxxxxxxx|00000011:00000011:00000001:xxxxxxxx|  0   :  0  :   0
   0001101b : 00 :  1b  |80000011:00000011:07878788:xxxxxxxx|80000011:00000011:07878788:xxxxxxxx|  0   :  0  :   0
```

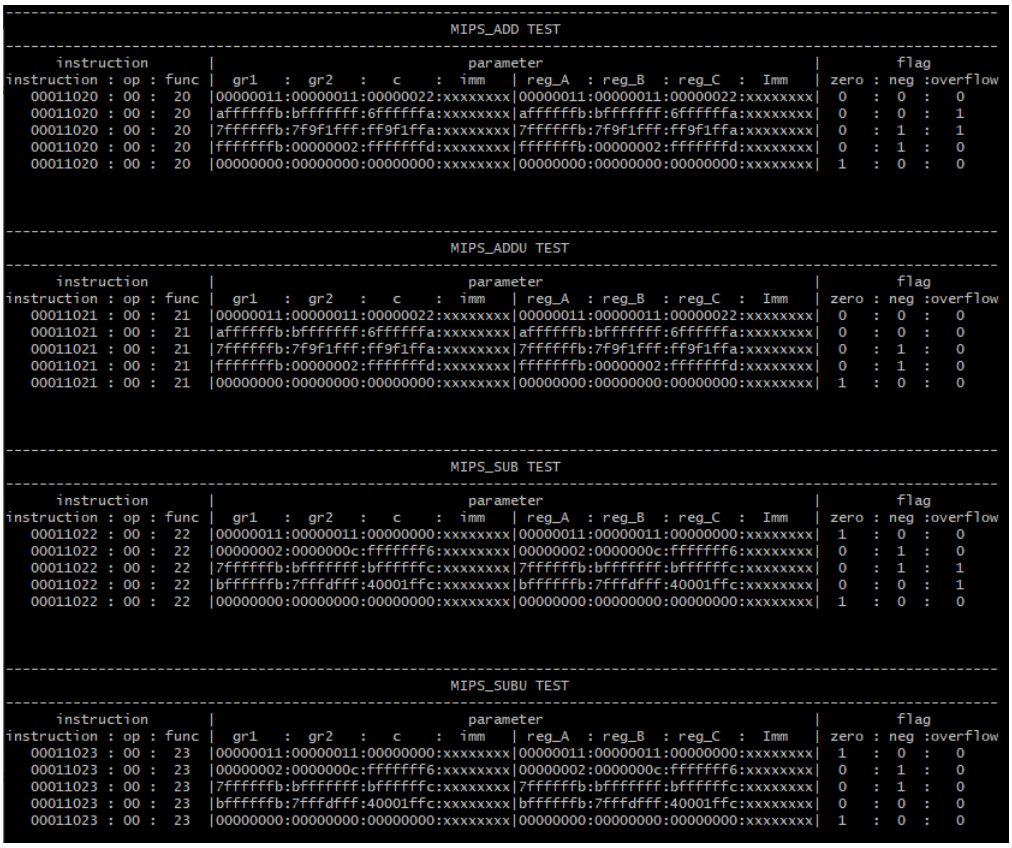
Figure 4: mult, multu, div, divu



```
MIPS_ADDI TEST
     instruction      |                                parameter                                |          flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   20000011 : 08 :  11  |00000011:00000000:00000022:xxxxxxxx|00000011:00000011:00000022:xxxxxxxx|  0   :  0  :   0
   20008011 : 08 :  11  |80000001:00000000:7fff8012:xxxxxxxx|80000001:00008011:7fff8012:xxxxxxxx|  0   :  0  :   1
   20000011 : 08 :  11  |7fffffff:00000000:80000010:xxxxxxxx|7fffffff:00000011:80000010:xxxxxxxx|  0   :  1  :   1
   20000011 : 08 :  11  |f0000001:00000000:f0000012:xxxxxxxx|f0000001:00000011:f0000012:xxxxxxxx|  0   :  1  :   0
   20000000 : 08 :  00  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0

MIPS_ADDIU TEST
     instruction      |                                parameter                                |          flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   24000011 : 09 :  11  |00000011:00000000:00000022:xxxxxxxx|00000011:00000011:00000022:xxxxxxxx|  0   :  0  :   0
   24008011 : 09 :  11  |80000001:00000000:80008012:xxxxxxxx|80000001:00008011:80008012:xxxxxxxx|  0   :  1  :   0
   24000011 : 09 :  11  |7fffffff:00000000:80000010:xxxxxxxx|7fffffff:00000011:80000010:xxxxxxxx|  0   :  1  :   0
   24000011 : 09 :  11  |f0000001:00000000:f0000012:xxxxxxxx|f0000001:00000011:f0000012:xxxxxxxx|  0   :  1  :   0
   24000000 : 09 :  00  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0
```

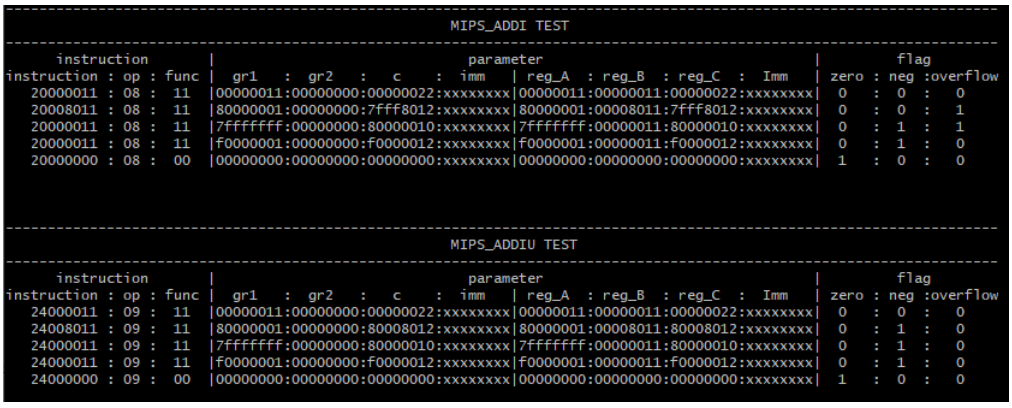
Figure 5: addi, addiu

4.4.3 Addition and Subtraction



```
------------------------------------------------------------------------------------------------------------------------
                                                MIPS_ADD TEST
------------------------------------------------------------------------------------------------------------------------
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011020 : 00 :  20  |00000011:00000011:00000022:xxxxxxxx|00000011:00000011:00000022:xxxxxxxx|  0   :  0  :   0
   00011020 : 00 :  20  |affffffb:bfffffff:6ffffffa:xxxxxxxx|affffffb:bfffffff:6ffffffa:xxxxxxxx|  0   :  0  :   1
   00011020 : 00 :  20  |7ffffffb:7f9f1fff:ff9f1ffa:xxxxxxxx|7ffffffb:7f9f1fff:ff9f1ffa:xxxxxxxx|  0   :  1  :   1
   00011020 : 00 :  20  |fffffffb:00000002:fffffffd:xxxxxxxx|fffffffb:00000002:fffffffd:xxxxxxxx|  0   :  1  :   0
   00011020 : 00 :  20  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0



------------------------------------------------------------------------------------------------------------------------
                                                MIPS_ADDU TEST
------------------------------------------------------------------------------------------------------------------------
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011021 : 00 :  21  |00000011:00000011:00000022:xxxxxxxx|00000011:00000011:00000022:xxxxxxxx|  0   :  0  :   0
   00011021 : 00 :  21  |affffffb:bfffffff:6ffffffa:xxxxxxxx|affffffb:bfffffff:6ffffffa:xxxxxxxx|  0   :  0  :   0
   00011021 : 00 :  21  |7ffffffb:7f9f1fff:ff9f1ffa:xxxxxxxx|7ffffffb:7f9f1fff:ff9f1ffa:xxxxxxxx|  0   :  1  :   0
   00011021 : 00 :  21  |fffffffb:00000002:fffffffd:xxxxxxxx|fffffffb:00000002:fffffffd:xxxxxxxx|  0   :  1  :   0
   00011021 : 00 :  21  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0



------------------------------------------------------------------------------------------------------------------------
                                                MIPS_SUB TEST
------------------------------------------------------------------------------------------------------------------------
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011022 : 00 :  22  |00000011:00000011:00000000:xxxxxxxx|00000011:00000011:00000000:xxxxxxxx|  1   :  0  :   0
   00011022 : 00 :  22  |00000002:0000000c:fffffff6:xxxxxxxx|00000002:0000000c:fffffff6:xxxxxxxx|  0   :  1  :   0
   00011022 : 00 :  22  |7ffffffb:bfffffff:bffffffc:xxxxxxxx|7ffffffb:bfffffff:bffffffc:xxxxxxxx|  0   :  1  :   1
   00011022 : 00 :  22  |bfffffb:7fffdfff:40001ffc:xxxxxxxx|bfffffb:7fffdfff:40001ffc:xxxxxxxx|  0   :  0  :   1
   00011022 : 00 :  22  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0



------------------------------------------------------------------------------------------------------------------------
                                                MIPS_SUBU TEST
------------------------------------------------------------------------------------------------------------------------
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   00011023 : 00 :  23  |00000011:00000011:00000000:xxxxxxxx|00000011:00000011:00000000:xxxxxxxx|  1   :  0  :   0
   00011023 : 00 :  23  |00000002:0000000c:fffffff6:xxxxxxxx|00000002:0000000c:fffffff6:xxxxxxxx|  0   :  1  :   0
   00011023 : 00 :  23  |7ffffffb:bfffffff:bffffffc:xxxxxxxx|7ffffffb:bfffffff:bffffffc:xxxxxxxx|  0   :  1  :   0
   00011023 : 00 :  23  |bfffffb:7fffdfff:40001ffc:xxxxxxxx|bfffffb:7fffdfff:40001ffc:xxxxxxxx|  0   :  0  :   0
   00011023 : 00 :  23  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0
```

Figure 6: add, addu, sub, subu



```
------------------------------------------------------------------------------------------------------------------------
                                                MIPS_ADDI TEST
------------------------------------------------------------------------------------------------------------------------
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   20000011 : 08 :  11  |00000011:00000000:00000022:xxxxxxxx|00000011:00000011:00000022:xxxxxxxx|  0   :  0  :   0
   20008011 : 08 :  11  |80000001:00000000:7fff8012:xxxxxxxx|80000001:00008011:7fff8012:xxxxxxxx|  0   :  0  :   1
   20000011 : 08 :  11  |7fffffff:00000000:80000010:xxxxxxxx|7fffffff:00000011:80000010:xxxxxxxx|  0   :  1  :   1
   20000011 : 08 :  11  |f0000001:00000000:f0000012:xxxxxxxx|f0000001:00000011:f0000012:xxxxxxxx|  0   :  1  :   0
   20000000 : 08 :  00  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0



------------------------------------------------------------------------------------------------------------------------
                                                MIPS_ADDIU TEST
------------------------------------------------------------------------------------------------------------------------
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   24000011 : 09 :  11  |00000011:00000000:00000022:xxxxxxxx|00000011:00000011:00000022:xxxxxxxx|  0   :  0  :   0
   24008011 : 09 :  11  |80000001:00000000:80008012:xxxxxxxx|80000001:00008011:80008012:xxxxxxxx|  0   :  1  :   0
   24000011 : 09 :  11  |7fffffff:00000000:80000010:xxxxxxxx|7fffffff:00000011:80000010:xxxxxxxx|  0   :  1  :   0
   24000011 : 09 :  11  |f0000001:00000000:f0000012:xxxxxxxx|f0000001:00000011:f0000012:xxxxxxxx|  0   :  1  :   0
   24000000 : 09 :  00  |00000000:00000000:00000000:xxxxxxxx|00000000:00000000:00000000:xxxxxxxx|  1   :  0  :   0
```

Figure 7: addi, addiu

4.4.4 Logical Operation

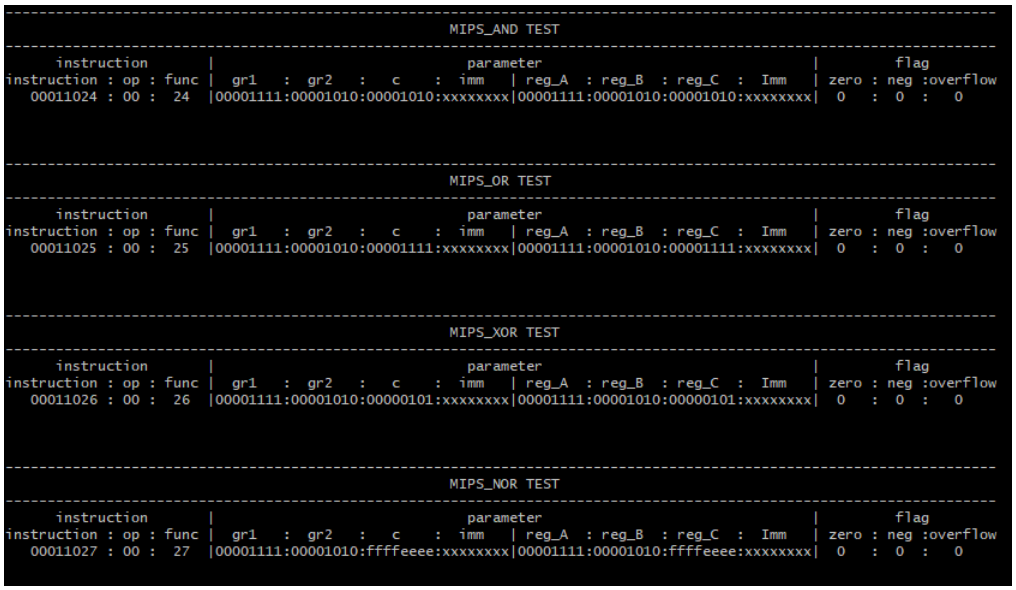


Figure 8: and, xor, or, nor

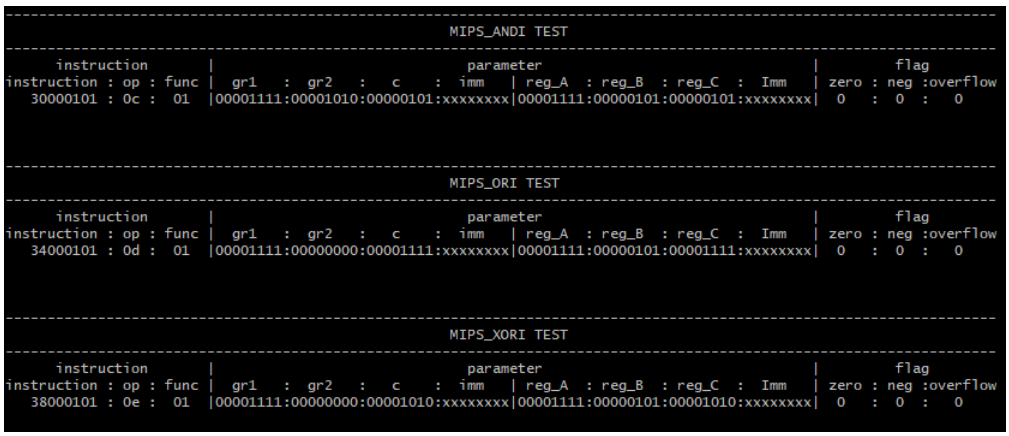
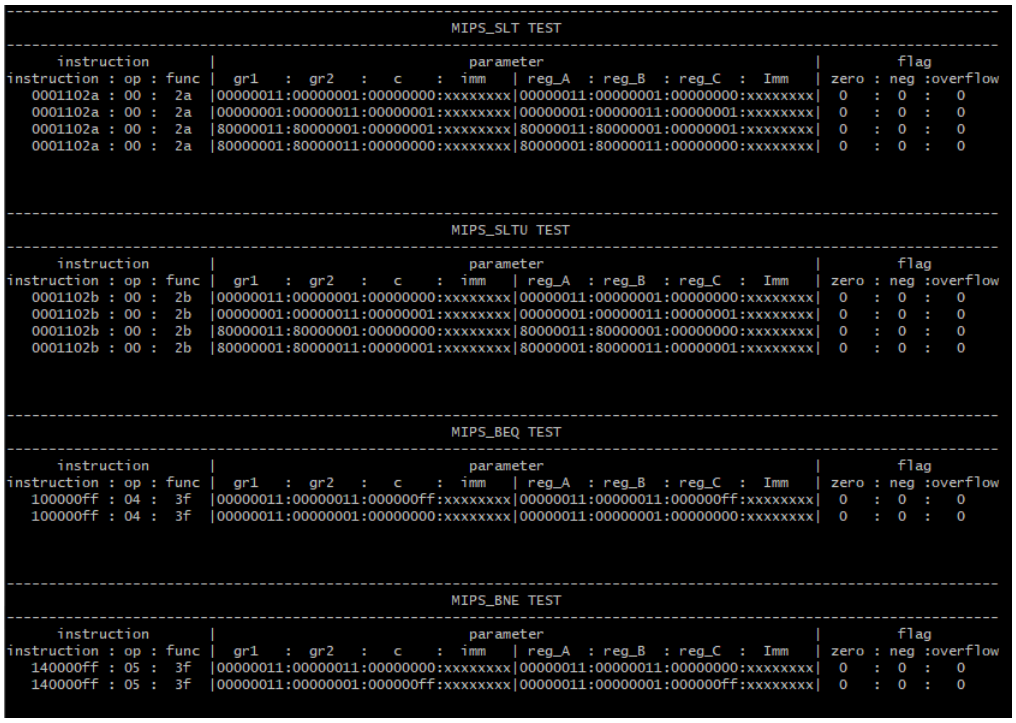


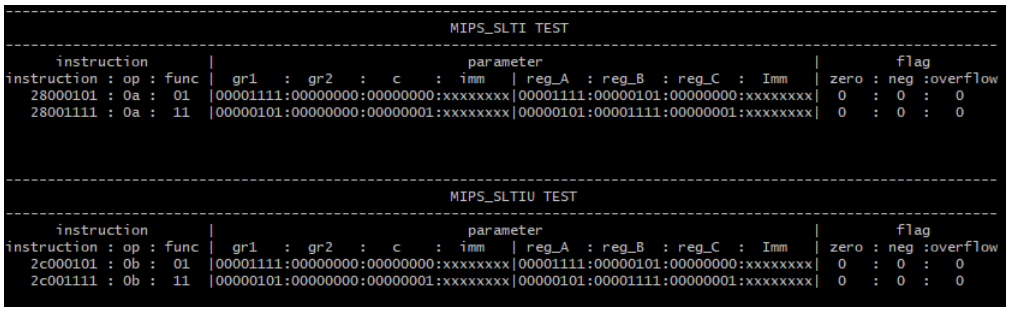
Figure 9: andi, xori, ori

4.4.5 Compare and Branch



```
MIPS_SLT TEST
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   0001102a : 00 :  2a  |00000011:00000001:00000000:xxxxxxxx|00000011:00000001:00000000:xxxxxxxx|  0   :  0  :   0
   0001102a : 00 :  2a  |00000001:00000011:00000001:xxxxxxxx|00000001:00000011:00000001:xxxxxxxx|  0   :  0  :   0
   0001102a : 00 :  2a  |80000011:80000001:00000001:xxxxxxxx|80000011:80000001:00000001:xxxxxxxx|  0   :  0  :   0
   0001102a : 00 :  2a  |80000001:80000011:00000000:xxxxxxxx|80000001:80000011:00000000:xxxxxxxx|  0   :  0  :   0

MIPS_SLTU TEST
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   0001102b : 00 :  2b  |00000011:00000001:00000000:xxxxxxxx|00000011:00000001:00000000:xxxxxxxx|  0   :  0  :   0
   0001102b : 00 :  2b  |00000001:00000011:00000001:xxxxxxxx|00000001:00000011:00000001:xxxxxxxx|  0   :  0  :   0
   0001102b : 00 :  2b  |80000011:80000001:00000000:xxxxxxxx|80000011:80000001:00000000:xxxxxxxx|  0   :  0  :   0
   0001102b : 00 :  2b  |80000001:80000011:00000001:xxxxxxxx|80000001:80000011:00000001:xxxxxxxx|  0   :  0  :   0

MIPS_BEQ TEST
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   100000ff : 04 :  3f  |00000011:00000011:000000ff:xxxxxxxx|00000011:00000011:000000ff:xxxxxxxx|  0   :  0  :   0
   100000ff : 04 :  3f  |00000011:00000001:00000000:xxxxxxxx|00000011:00000001:00000000:xxxxxxxx|  0   :  0  :   0

MIPS_BNE TEST
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   140000ff : 05 :  3f  |00000011:00000011:00000000:xxxxxxxx|00000011:00000011:00000000:xxxxxxxx|  0   :  0  :   0
   140000ff : 05 :  3f  |00000011:00000001:000000ff:xxxxxxxx|00000011:00000001:000000ff:xxxxxxxx|  0   :  0  :   0
```

Figure 10: slt, sltu, beq, bne



```
MIPS_SLTI TEST
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   28000101 : 0a :  01  |00001111:00000000:00000000:xxxxxxxx|00001111:00000101:00000000:xxxxxxxx|  0   :  0  :   0
   28001111 : 0a :  11  |00000101:00000000:00000001:xxxxxxxx|00000101:00001111:00000001:xxxxxxxx|  0   :  0  :   0

MIPS_SLTIU TEST
      instruction       |                                parameter                                |         flag
instruction : op : func |  gr1   :  gr2   :   c    :  imm   | reg_A  : reg_B  : reg_C  :  Imm   | zero : neg :overflow
   2c000101 : 0b :  01  |00001111:00000000:00000000:xxxxxxxx|00001111:00000101:00000000:xxxxxxxx|  0   :  0  :   0
   2c001111 : 0b :  11  |00000101:00000000:00000001:xxxxxxxx|00000101:00001111:00000001:xxxxxxxx|  0   :  0  :   0
```

Figure 11: slti, sltiu

4.4.6 Save and Load

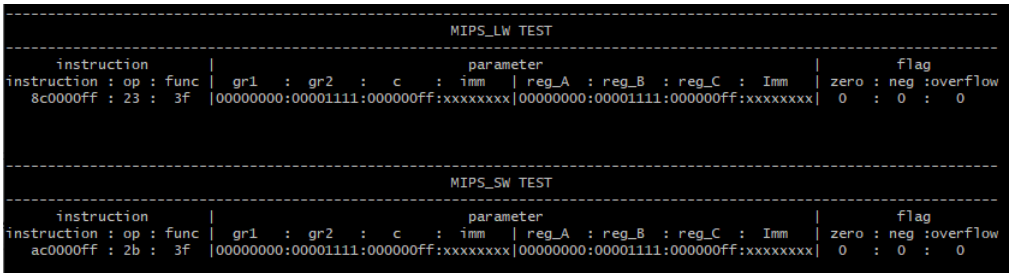


Figure 12: slw,

5 Solution

5.1 Implementation

5.1.1 Parameters (ALU.v)

output signed[31:0] c: The result for output.

output zero: The zero flag for output.

output overflow: The overflow flag for output.

output neg: The negative flag for output.

input signed[31:0] imm: Immediate number for input.

input signed[31:0] i_datain: Instruction for input.

input signed[31:0] gr1: Parameter 1 for input.

input signed[31:0] gr2: Parameter 2 for input.

reg[5:0] opcode: Operation code.

reg[5:0] func: Function code.

reg zf = 0 : The zero flag for output.

reg nf = 0 : The negative flag for output.

reg of = 0 : The overflow flag for output.

reg[31:0] reg_A: Register parameter A.

reg[31:0] reg_B: Register parameter B.

reg[31:0] reg_C: Register parameter C.

reg[31:0] Imm: Immediate number.

5.1.2 Parameters (test_ALU.v)

reg[31:0] i_datain: Instruction.

reg[31:0] gr1: Parameter 1.

reg[31:0] gr2: Parameter 2.

reg[31:0] imm: Immediate number.

wire[31:0] c: Test result.

wire zero: The zero flag.

wire overflow: The overflow flag.

wire neg: The negative flag.

5.2 Core Code

```
1 begin
2
3 opcode = i_datain[31:26];
4 func = i_datain[5:0];
```

```
5
6  case(opcode)
7  6'b001000:␣␣␣//␣addu:␣Addition␣immediate␣(with␣overflow)
8  ␣␣␣␣begin
9  ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
10 ␣␣␣␣␣␣␣␣reg_B␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
11         reg_C = $signed(reg_A) + $signed(reg_B[15:0]);
12         if (~(reg_A[31] ^ reg_B[15]) && (reg_C[31] ^ reg_A[31]))
13             of = 1;
14         else
15             of = 0;
16         if (reg_C == 0)
17             zf = 1;
18         else
19             zf = 0;
20         if (reg_C[31] == 1)
21             nf = 1;
22         else
23             nf = 0;
24     end
25 6'b001001:␣␣␣//␣addiu:␣Addition␣immediate␣(without␣overflow)
26 ␣␣␣␣begin
```

```
        reg_A = gr1;
        reg_B = {16'b0000_0000_0000_0000, i_datain[15:0]};
            reg_C = $unsigned(reg_A) + $unsigned(reg_B[15:0]);
            of = 0;
            if (reg_C == 0)
                zf = 1;
            else
                zf = 0;
            if (reg_C[31] == 1)
                nf = 1;
            else
                nf = 0;
        end
    6'b001100:    // andi: AND immediate
        begin
        reg_A = gr1;
        reg_B = {16'b0000_0000_0000_0000, i_datain[15:0]};
            reg_C = reg_A & reg_B;
            of = 0;
            zf = 0;
            nf = 0;
        end
```

```
49 6'b001101:␣␣␣//␣ori:␣OR␣immediate
50 ␣␣␣␣begin
51 ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
52 ␣␣␣␣␣␣␣␣reg_B␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
53          reg_C = reg_A | reg_B;
54          of = 0;
55          zf = 0;
56          nf = 0;
57      end
58 6'b001110:␣␣␣//␣xori:␣XOR␣immediate
59 ␣␣␣␣begin
60 ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
61 ␣␣␣␣␣␣␣␣reg_B␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
62          reg_C = reg_A ^ reg_B;
63          of = 0;
64          zf = 0;
65          nf = 0;
66      end
67 6'b000100:␣␣␣//␣beq:␣Branch␣on␣equal
68 ␣␣␣␣begin
69 ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
70 ␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
```

```
71 ␣␣␣␣␣␣␣␣if␣(reg_A␣==␣reg_B)
72 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
73          else
74              reg_C = 32'b0;
75 ␣␣␣␣␣␣␣␣of␣=␣0;
76 ␣␣␣␣␣␣␣␣zf␣=␣0;
77 ␣␣␣␣␣␣␣␣nf␣=␣0;
78 ␣␣␣␣end
79 6'b000101:   // bne: Branch on not equal
80     begin
81         reg_A = gr1;
82         reg_B = gr2;
83         if (reg_A != reg_B)
84             reg_C = {16'b0000_0000_0000_0000,␣i_datain[15:0]};
85 ␣␣␣␣␣␣␣␣else
86 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣32'b0;
87         of = 0;
88         zf = 0;
89         nf = 0;
90     end
91 6'b100011:␣␣␣//␣lw:␣Load␣word
92 ␣␣␣␣begin
```

```
93  ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
94  ␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
95  ␣␣␣␣␣␣␣␣reg_C␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
96      end
97  6'b101011:␣␣␣//␣sw:␣Store␣word
98  ␣␣␣␣begin
99  ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
100 ␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
101 ␣␣␣␣␣␣␣␣reg_C␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
102     end
103 6'b001010:␣␣␣//␣slti:␣Set␣less␣than␣immediate
104 ␣␣␣␣begin
105 ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
106 ␣␣␣␣␣␣␣␣reg_B␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
107         if (reg_A[31] && reg_B[15])
108             reg_C = (reg_A[30:0] >= reg_B[14:0]);
109         else if (~(reg_A[31] || reg_B[15]))
110             reg_C = (reg_A[30:0] < reg_B[14:0]);
111         else if (reg_A[31])
112             reg_C = 32'b1;
113 ␣␣␣␣␣␣␣␣else
114 ␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣32'b0;
```

```
115             of = 0;
116             zf = 0;
117             nf = 0;
118         end
119     6'b001011:␣␣␣//␣sltiu:␣Set␣less␣than␣unsigned␣immediate
120     ␣␣␣␣begin
121     ␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
122     ␣␣␣␣␣␣␣␣reg_B␣=␣{16'b0000_0000_0000_0000, i_datain[15:0]};
123             reg_C = $unsigned(reg_A) < $unsigned(reg_B);
124             of = 0;
125             zf = 0;
126             nf = 0;
127         end
128     6'b000000:
129     ␣␣␣␣begin
130     ␣␣␣␣␣␣␣␣case(func)
131     ␣␣␣␣␣␣␣␣6'b000000:   // sll: Shift left logical
132                 begin
133                     reg_A = gr2;
134                     reg_B = i_datain[10:6];
135                     reg_C = reg_A << reg_B;
136                     of = 0;
```

```
137                     zf = 0;
138                     nf = 0;
139                 end
140             6'b000100:␣␣␣//␣sllv:␣Shift␣left␣logical␣variable
141 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
142 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr2;
143 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr1;
144 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣reg_A␣<<␣reg_B;
145 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
146 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
147 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
148 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
149 ␣␣␣␣␣␣␣␣6'b000011:   // sra: Shift right arithmetic
150                 begin
151                     reg_A = gr2;
152                     reg_B = i_datain[10:6];
153                     reg_C = reg_A >> reg_B;
154                     reg_C = {reg_A[31:31],reg_C[30:0]};
155                     of = 0;
156                     zf = 0;
157                     nf = 0;
158                 end
```

```
159         6'b000111:␣␣␣//␣srav:␣Shift␣right␣arithmetic␣variable
160 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
161 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr2;
162 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr1;
163 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣reg_A␣>>␣reg_B;
164 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣{reg_A[31:31],reg_C[30:0]};
165 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
166 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
167 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
168 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
169 ␣␣␣␣␣␣␣␣6'b000010:   // srl: Shift right logical
170             begin
171                 reg_A = gr2;
172                 reg_B = i_datain[10:6];
173                 reg_C = reg_A >> reg_B;
174                 of = 0;
175                 zf = 0;
176                 nf = 0;
177             end
178         6'b000110:␣␣␣//␣srlv:␣Shift␣right␣logical␣variable
179 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
180 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr2;
```

```
181 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr1;
182 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣reg_A␣>>␣reg_B;
183 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
184 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
185 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
186 ␣␣␣␣␣␣␣␣␣␣␣␣end
187 ␣␣␣␣␣␣␣␣6'b100000:   // add: Addition (with overflow)
188             begin
189                 reg_A = gr1;
190                 reg_B = gr2;
191                 reg_C = $signed(reg_A) + $signed(reg_B);
192                 if ((~(reg_A[31] ^ reg_B[31])) && (reg_C[31] ^ reg_A[31]))
193                     of = 1;
194                 else
195                     of = 0;
196                 if (reg_C == 0)
197                     zf = 1;
198                 else
199                     zf = 0;
200                 if (reg_C[31] == 1)
201                     nf = 1;
202                 else
```

```
                    nf = 0;
            end
        6'b100001:␣␣␣//␣addu:␣Addition␣(without␣overflow)
␣␣␣␣␣␣␣␣␣␣␣␣begin
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣$unsigned(reg_A)␣+␣$unsigned(reg_B);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C␣==␣0)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣1;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C[31]␣==␣1)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣1;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
␣␣␣␣␣␣␣␣␣␣␣␣end
␣␣␣␣␣␣␣␣6'b100010:   // sub: Subtract (with overflow)
            begin
                reg_A = gr1;
                reg_B = gr2;
                reg_C = $signed(reg_A) - $signed(reg_B);
```

```
225                     if ((reg_A[31] ^ reg_B[31]) && (reg_A[31] ^ reg_C[31]))
226                         of = 1;
227                     else
228                         of = 0;
229                     if (reg_C == 0)
230                         zf = 1;
231                     else
232                         zf = 0;
233                     if (reg_C[31] == 1)
234                         nf = 1;
235                     else
236                         nf = 0;
237                 end
238             6'b100011:␣␣␣//␣subu:␣Subtract␣(without␣overflow)
239 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
240 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
241 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
242 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣$unsigned(reg_A)␣-␣$unsigned(reg_B);
243 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
244 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C␣==␣0)
245 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣1;
246 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
```

```
247 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
248 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C[31]␣==␣1)
249 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣1;
250 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
251 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
252 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
253 ␣␣␣␣␣␣␣␣6'b011010:   // div: Divide (with overflow)
254                 begin
255                     reg_A = gr1;
256                     reg_B = gr2;
257                     reg_C = $signed(reg_A)/ $signed(reg_B);
258                     of = 0;
259                     if (reg_C == 0)
260                         zf = 1;
261                     else
262                         zf = 0;
263                     if (reg_C[31] == 1)
264                         nf = 1;
265                     else
266                         nf = 0;
267                 end
268             6'b011011:␣␣␣//␣divu:␣Divide␣(without␣overflow)
```

```
269 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
270 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
271 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
272 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣$unsigned(reg_A)␣/␣$unsigned(reg_B);
273 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
274 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C[31]␣==␣1)
275 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣1;
276 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
277 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
278 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C␣==␣0)
279 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣1;
280 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
281 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
282 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
283 ␣␣␣␣␣␣␣␣6'b011000:   // mult: Multiply (with overflow)
284                 begin
285                     reg_A = gr1;
286                     reg_B = gr2;
287                     reg_C = reg_A * reg_B;
288                     ex = reg_A * reg_B;
289                     if (ex == reg_C)
290                     begin
```

```
291                         of = 0;
292                         if (reg_C[31] == 1)
293                             nf = 1;
294                         else
295                             nf = 0;
296                         if (reg_C == 0)
297                             zf = 1;
298                         else
299                             zf = 0;
300                     end
301                     else
302                         of=1;
303                 end
304             6'b011001:␣␣␣//␣multu:␣Multiply␣(without␣overflow)
305 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
306 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
307 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
308 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣$unsigned(reg_A)␣*␣$unsigned(reg_B);
309 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C[31]␣==␣1)
310 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣1;
311 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
312 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
```

```
313 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(reg_C␣==␣0)
314 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣1;
315 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
316 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
317 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
318 ␣␣␣␣␣␣␣␣␣␣␣␣end
319 ␣␣␣␣␣␣␣␣6'b100100:   // and: AND
320               begin
321                   reg_A = gr1;
322                   reg_B = gr2;
323                   reg_C = reg_A & reg_B;
324                   of = 0;
325                   zf = 0;
326                   nf = 0;
327               end
328           6'b100101:␣␣␣//␣or:␣OR
329 ␣␣␣␣␣␣␣␣␣␣␣␣begin
330 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
331 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
332 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣reg_A␣|␣reg_B;
333 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
334 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
```

```
335 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
336 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
337 ␣␣␣␣␣␣␣␣6'b100110:   // xor: XOR
338             begin
339                 reg_A = gr1;
340                 reg_B = gr2;
341                 reg_C = reg_A ^ reg_B;
342                 of = 0;
343                 zf = 0;
344                 nf = 0;
345             end
346         6'b100111:␣␣␣//␣nor:␣NOR
347 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
348 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
349 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
350 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣~(reg_A␣|␣reg_B);
351 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
352 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
353 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
354 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
355 ␣␣␣␣␣␣␣␣6'b101010:   // slt: Set less than
356             begin
```

```
357                     reg_A = gr1;
358                     reg_B = gr2;
359                     if (reg_A[31] && reg_B[31])
360                         reg_C = (reg_A >= reg_B);
361                     else if (~(reg_A[31] || reg_B[31]))
362                         reg_C = (reg_A < reg_B);
363                     else if (reg_A[31])
364                         reg_C = 32'b1;
365 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣else
366 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣32'b0;
367                     of = 0;
368                     zf = 0;
369                     nf = 0;
370                 end
371             6'b101011:␣␣␣//␣sltu:␣Set␣less␣than␣unsigned
372 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣begin
373 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_A␣=␣gr1;
374 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_B␣=␣gr2;
375 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reg_C␣=␣reg_A␣<␣reg_B;
376 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣=␣0;
377 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣zf␣=␣0;
378 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣nf␣=␣0;
```

```
379 ␣␣␣␣␣␣␣␣␣␣␣␣␣end
380 ␣␣␣␣␣␣␣␣endcase
381 ␣␣␣␣end
382 endcase
383
384 end
```