Project Report

CSC 3050 Simplified ALU

Wei Wu

118010335

April 10, 2020

The School of Science and Engineering

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

1. Understandings of the Project

This project is about writing a simplified MIPS ALU using hardware description language Verilog. This project is divided into two modules. The first module is an instruction decoder. It decodes opcode (and funct for R-type instructions) of the MIPS instruction, and then generates aluctr and alusrc. The second module is an ALU. It takes aluctr, gr1, and operand2 (gr2 or sign-extended imm) as input. After corresponding calculation, it outputs the result and flags, including a zero flag, a negative flag, and an overflow flag.

2. The Implementations of the Project

2.1.The Big Picture Idea

At the very beginning, the hardware needs to generate control signals for the ALU from the input 32-bit MIPS instruction code. To be specific, the control signals include a 5-bit aluctr signal and a 1-bit alusrc signal. The aluctr signal determines the behavior of the ALU (add, sub, mult, …), while the alusrc signal determines the second operand (gr2 or sign-extended imm). These signals can be generated according to the opcode (and funct for R-type instructions) of the instruction.

Then, according to the control signals, the ALU starts to handle the operation. Firstly, it calculates the result. After that, it sets the zero flag, the negative flag, and the overflow flag. In the end, it outputs both the result and the flags.

2.2. Implementation Details

2.2.0. Load and store

For most instructions, three registers are used: reg_A, reg_B, and reg_C.

The ALU loads the input gr1 to reg_A. If alusrc=0, the ALU loads the input gr2 to reg_B; else if alusrc=1, the ALU loads the sign-extended immediate to reg_B.

After calculation, the ALU stores the result in reg_C.

2.2.1. Special operations with other registers

For mult and multu, the register hi stores the higher 32 bits of the result, while the register lo stores the lower 32 bits of the result.

For div and divu, the register hi stores the remainder, while the register lo stores the quotient.

For addiu, andi, ori, xori, and sltiu, the register imm0 stores the zero-extended immediate.

2.2.2. The control part

The control part (module control_unit in the code) takes both opcode and funct as input. It generates 2 control signals, namely aluctr and alusrc. The aluctr signal is 5-bit long. It determines the behavior of the ALU (add, sub, mult, …). While the alusrc signal is 1-bit long. It determines the source of the second operand (gr2 or sign-extended imm).

If opcode is 0 (R-type instructions and BEQ, BNE), the alusrc signal will be set to 0, which indicates that the second operand (reg_B) should be gr2. Else, if the opcode is not 0 (other I-type or J-type instructions), the alusrc signal will be set

to 1, which indicates that the second operand (reg_B) should be the sign-extended immediate.

Furthermore, the required instructions are sorted by their ALU behaviors. Each distinct ALU behavior is mapped to an aluctr code. For example, both sub and beq correspond to ALU subtraction. Thus, they should share the same aluctr code.

In this project, we just made up all the aluctr codes. The correspondence between aluctr and ALU behaviors are illustrated in the following table.

0: add / addi / lw / sw	1: addu / addiu	2: sub / beq / bne	3: subu
4: mult	5: multu	6: div	7: divu
8: and	9: nor	10: or	11: xor
12: slt / slti	13: sltu / sltiu	14: sll	15: sllv
16: srl	17: srlv	18: sra	19: srav
20: andi	21: ori	22: xori	

2.2.3. Sign-extension and zero-extension

According to the MIPS ISA, the immediate is sign-extended and then transferred to the ALU. The sign-extended immediate can be used directly when the instruction is addi, beq, bne, … However, when it comes to instuctions like andi, ori, xori, …, the zero-extended immediate is needed. Therefore, inside the ALU, we need to generate the zero-extended immediate from the input signed-extended immediate. Also, an additional register imm0 is needed to store the

zero-extended immediate.

For zero-extension, we only need to concatenate 16 0's with the immediate. The Verilog code is like:

```
{{16{1'b0}}, reg_B[15:0]}
```

For sign-extension, we need to extend the immediate with its sign bit (MSB). The Verilog code is like:

```
{{16{imm[15]}}, imm}
```

2.2.4. The negative flag

For signed operations, the negative flag indicates whether the result is negative. It simply equals the sign bit (MSB) of the result. For unsigned operations, the negative flag is not important. We just set it to be 0.

2.2.5. The zero flag

The zero flag indicates whether the result is equal to zero. It is a signal for branch instructions.

2.2.6. The overflow flag

For instructions like add, sub, addi, …, the overflow flag indicates whether an overflow exception happens. For other instructions, it is not important. We just set it to be 0.

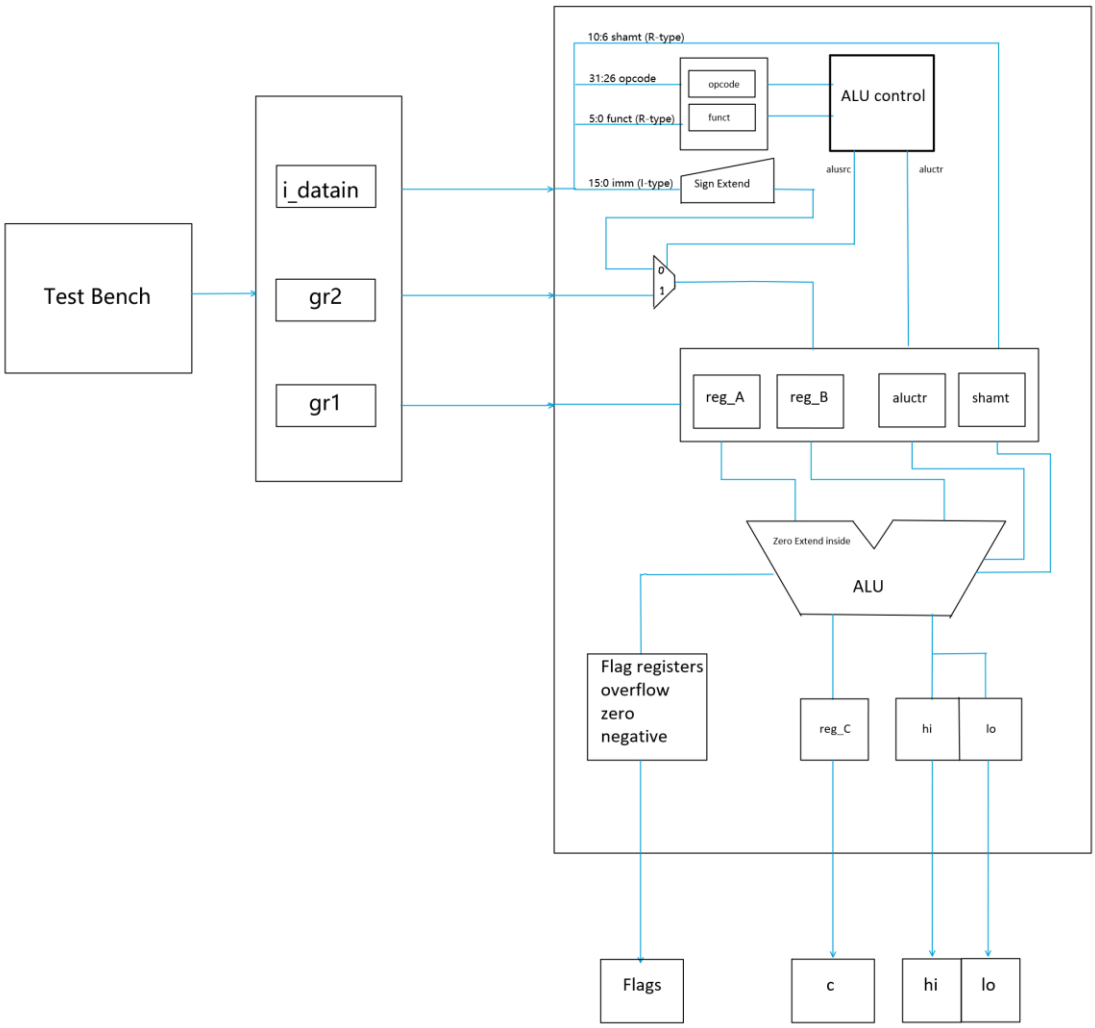
In the MIPS ALU, the 32-bit full adder consists of 32 1-bit full adders. To detect overflow, we can simply compare the 30^{th} carry-out and the 31^{st} carry-out (In other words, carry into MSB and carry out of MSB). If the 30^{th} carry-out is not equal to the 31^{st} carry-out, an overflow occurs.

In my program, I did not use a 32-bit full adder. Instead, I just use “reg_A + reg_B”. Therefore, I used an additional 1-bit register called bit32 to detect overflow. The code is

```
{bit32, reg_C} = {reg_A[31], reg_A} + {reg_B[31], reg_B};
overflow = bit32 ^ reg_C[31];
```

The mechanism is very similar. First, concatenate reg_A and reg_B to their MSB, respectively. Then, add them up. Store the result in the reg_C concatenated to the extra 1-bit register bit32. If bit32 is not equal to reg_C[31], an overflow occurs.

2.3. Block Diagram



Registers:

In main module:

```
//Instruction Decoding
reg [5:0] opcode, funct;

//shamt
reg unsigned [4:0] shamt;

//Sign-extended immediate
reg [31:0] imm;

//When alusrc = 0, operand2 = gr2
//When alusrc = 1, operand2 = imm
reg [31:0] operand2;
```

In ALU Control module:

```
//Registers for operand1, operand2, and result
reg signed [31:0] reg_A, reg_B, reg_C;

//Registers for the flags
reg zero, negative, overflow;

//Register hi and lo.
//They are used for MULT, MULTU, DIV, DIVU
reg [31:0] hi, lo;

//Unsigned registers for MULTU, DIVU, SLTU, SLTIU
reg [31:0] reg_B_u, reg_A_u;

//Zero-extended imm
reg [31:0] imm0;

//Register which is used to dectect overflow
reg bit32;
```

In the control_unit module

```
/*
aluctr is made up
It starts from 0b0_0000
*/
reg [5:0] aluctr;

//alusrc = 0 : from register
//alusrc = 1 : from extended immediate
reg alusrc;
```

2.4. Explanation of Instructions

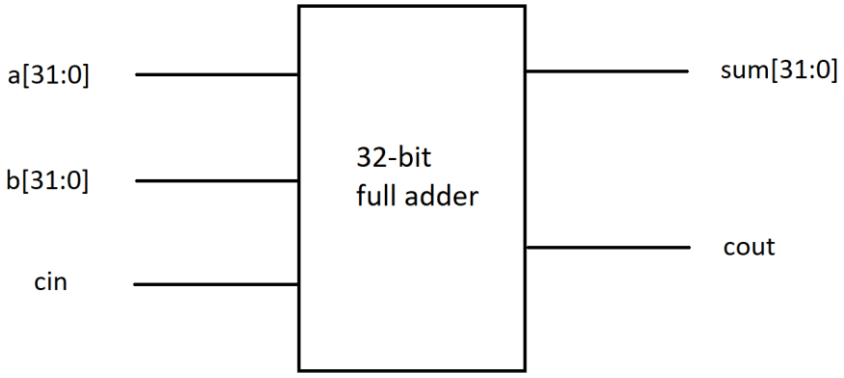
2.4.0. Testing Environment

Windows 10 x64

iVerilog v11-20190809 x64

2.4.1. ADD / ADDI

Hardware: 32-bit full adder



Theory: The 32-bit full adder is composed of 32 1-bit full adders. The inputs a[31:0] and b[31:0] are reg_A and reg_B. For addition, cin is 0.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 + gr2 / c = gr1 + [sign-extended] imm

Overflow flag: Set if an overflow occurs.

Zero flag: Set if reg_C = 0.

Negative flag: reg_C[31]

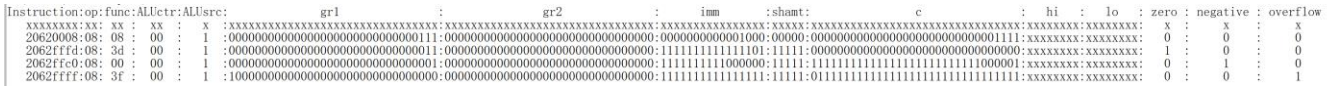
Test Result:

```
//ADD: 1 + 8 = 9
#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

```
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_1000;

//ADD: 2 + -2 = 0 zero
#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1110;

//ADD: -2 + -1 = -3 negative
#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b1111_1111_1111_1111_1111_1111_1111_1110;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

//ADD: -1 + -2147483648 = overflow
#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
gr2<=32'b1000_0000_0000_0000_0000_0000_0000_0000;

//ADD: 2147483647 + 8 = overflow
#10 i_datain<=32'b000000_00011_00010_00001_00000_100000;
gr1<=32'b0111_1111_1111_1111_1111_1111_1111_1111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_1000;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1                :              gr2                :      imm       :shamt:              c                :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :    x    :     x
   00620820:00: 20 :  00  :   0  :00000000000000000000000000000001:00000000000000000000000000001000:00001000001000000:00000:00000000000000000000000000001001:xxxxxxxx:xxxxxxxx:   0  :    0    :     0
   00620820:00: 20 :  00  :   0  :00000000000000000000000000000010:11111111111111111111111111111110:00001000001000000:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:   1  :    0    :     0
   00620820:00: 20 :  00  :   0  :11111111111111111111111111111110:11111111111111111111111111111111:00001000001000000:00000:11111111111111111111111111111101:xxxxxxxx:xxxxxxxx:   0  :    1    :     0
   00620820:00: 20 :  00  :   0  :11111111111111111111111111111111:10000000000000000000000000000000:00001000001000000:00000:11111111111111111111111111111111:xxxxxxxx:xxxxxxxx:   0  :    0    :     1
   00620820:00: 20 :  00  :   0  :01111111111111111111111111111111:00000000000000000000000000001000:00001000001000000:00000:10000000000000000000000000000111:xxxxxxxx:xxxxxxxx:   0  :    1    :     1
```

```
//ADDI: 7 + 8 = 15
#10 i_datain<=32'b001000_00011_00010_0000000000001000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

//ADDI: 3 + -3 = 0 zero
#10 i_datain<=32'b001000_00011_00010_1111111111111101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

//ADDI: 1 + -64 = -63 negative
#10 i_datain<=32'b001000_00011_00010_1111111111000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

//ADDI: -2147483648 + -1 = overflow
#10 i_datain<=32'b001000_00011_00010_1111111111111111;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```



Instruction	op	func	ALUctr	ALUsrc	gr1	gr2	imm	shamt	c	hi	lo	zero	negative	overflow
xxxxxxxx	xx	xx	xx	x	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx	xxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxx	xxxxxxxx	x	x	x
20620008	08	00	01	1	00000000000000000000000000000111	00000000000000000000000000000000	0000000000001000	00000	00000000000000000000000000001111	xxxxxxxx	xxxxxxxx	0	0	0
2062fffd	3d	00	01	1	00000000000000000000000000000011	00000000000000000000000000000000	1111111111111101	11111	00000000000000000000000000000000	xxxxxxxx	xxxxxxxx	1	0	0
2062ffc0	08	00	01	1	00000000000000000000000000000001	00000000000000000000000000000000	1111111111000000	11111	11111111111111111111111111000001	xxxxxxxx	xxxxxxxx	0	1	0
2062ffff	3f	00	01	1	10000000000000000000000000000000	00000000000000000000000000000000	1111111111111111	11111	01111111111111111111111111111111	xxxxxxxx	xxxxxxxx	0	0	1

2.4.2. ADDU / ADDIU

Hardware: 32-bit full adder

Registers: reg_A, reg_B, reg_C / c = gr1 + [sign-extended] imm

Result: c = gr1 + gr2

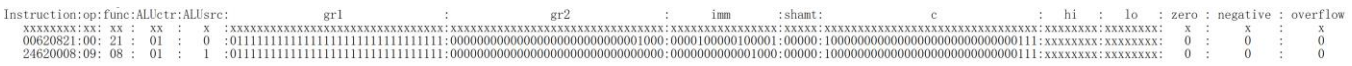
Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//ADDU: 2147483647 + 8 = 2147483655
#10 i_datain<=32'b000000_00011_00010_00001_00000_100001;
gr1<=32'b0111_1111_1111_1111_1111_1111_1111_1111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_1000;

//ADDIU: 2147483647 + 8 = 2147483655
#10 i_datain<=32'b001001_00011_00010_0000000000001000;
gr1<=32'b0111_1111_1111_1111_1111_1111_1111_1111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```



Instruction	op	func	ALUctr	ALUsrc	gr1	gr2	imm	shamt	c	hi	lo	zero	negative	overflow
xxxxxxxx	xx	xx	xx	x	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx	xxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxx	xxxxxxxx	x	x	x
00620821	00	21	01	0	01111111111111111111111111111111	00000000000000000000000000001000	0000100000100001	00000	10000000000000000000000000000111	xxxxxxxx	xxxxxxxx	0	0	0
24620008	09	08	01	1	01111111111111111111111111111111	00000000000000000000000000000000	0000000000001000	00000	10000000000000000000000000000111	xxxxxxxx	xxxxxxxx	0	0	0

2.4.3. SUB

Hardware: 32-bit full adder, invertor

Theory: Subtraction can be done by adding the 2's complement of the subtrahend. First, use the 32-bit invertor to get the 1's complement of reg_B. Then, the inputs a[31:0] and b[31:0] are reg_A and 1's complment of reg_B. For subtraction, cin is 1. This is because adding 1 to the 1's complement gives the 2's complement.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 + 1's complement of gr2 + 1

Overflow flag: Set if an overflow occurs.

Zero flag: Set if reg_C = 0.

Negative flag: reg_C[31]

Test Result:

```
//SUB: 7 - 3 = 4
#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;

//SUB: 3 - 3 = 0 zero
#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;

//SUB: 3 - 7 = -4 negative
#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0111;

//SUB: -2147483648 - 1 = overflow
#10 i_datain<=32'b000000_00011_00010_00001_00000_100010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

```
Instruction:op:func:ALUctr:ALUsrc:                gr1                :                gr2                :      imm       :shamt:                c                :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx : xx  :  x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :    x    :    x
00620822:00: 22 :  02 :  0  :00000000000000000000000000000111:00000000000000000000000000000011:00001000001000010:00000:00000000000000000000000000000100:xxxxxxxx:xxxxxxxx:   0  :    0    :    0
00620822:00: 22 :  02 :  0  :00000000000000000000000000000011:00000000000000000000000000000011:00001000001000010:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:   1  :    0    :    0
00620822:00: 22 :  02 :  0  :00000000000000000000000000000011:00000000000000000000000000000111:00001000001000010:00000:11111111111111111111111111111100:xxxxxxxx:xxxxxxxx:   0  :    1    :    0
00620822:00: 22 :  02 :  0  :10000000000000000000000000000000:00000000000000000000000000000001:00001000001000010:00000:01111111111111111111111111111111:xxxxxxxx:xxxxxxxx:   0  :    0    :    1
```

2.4.4. SUBU

Hardware: 32-bit full adder, invertor

Registers: reg_A, reg_B, reg_C

Result: c = gr1 + 1's complement of [zero-extended] imm + 1

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

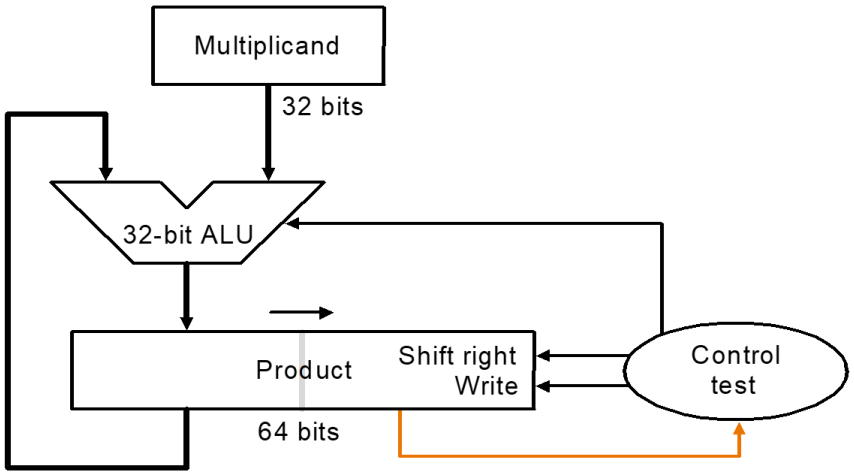
Test Result:

```
//SUBU: 2147483648 - 1 = 2147483647
#10 i_datain<=32'b000000_00011_00010_00001_00000_100011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1              :              gr2              :      imm       :shamt:              c              :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    :     x
00620823:00: 23 :  03  :   0  :10000000000000000000000000000000:00000000000000000000000000000001:0000100000100011:00000:01111111111111111111111111111111:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

2.4.5. MULT

Hardware: 32-bit x 32-bit multiplier



Theory:

Since a 32-bit x 32-bit multiplication gets a 64-bit result, we need two 32-bit registers hi and lo to store the result.

Algorithm
For positive numbers: 1. Put the multiplier (reg_A) into register lo. Set register hi to be 0. Repeat the following steps for 32 times: 2. If lo[0] = 1, add the multiplicand (reg_B) to register hi. Place the result in register hi.

3. Shift {hi, lo} right by 1 bit

For negative numbers:

1. Remember the sign of each operand.
2. Convert any negative number to its 2's complement (absolute value).
3. Do the multiplication as the positive numbers.
4. If one operand is negative and the other is positive, convert the result to its 2's complement.

Registers: reg_A, reg_B, hi, lo

Result: {hi, lo} = gr1 * gr2

Overflow flag: 0

Zero flag: Set if {hi, lo} = 0.

Negative flag: hi[31]

Test Result:

```
//MULT: 2 * 4 = 8
#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//MULT: 2 * 0 = 0 zero
#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;

//MULT: -2 * 4 = -8 negative
#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b1111_1111_1111_1111_1111_1111_1111_1110;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//MULT: 16777215 * 16777215 = 281474943156225
#10 i_datain<=32'b000000_00011_00010_00001_00000_011000;
gr1<=32'b0000_0000_1111_1111_1111_1111_1111_1111;
```

```
gr2<=32'b0000_0000_1111_1111_1111_1111_1111_1111;
```

```
|Instruction:op:func:ALUctr:ALUsrc:                gr1                :                gr2                :      imm       :shamt:                c                :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :    x    :    x
   00620818:00: 18 :  04  :   0  :00000000000000000000000000000010:00000000000000000000000000000100:0000100000011000:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:00000008:   0  :    0    :    0
   00620818:00: 18 :  04  :   0  :00000000000000000000000000000010:00000000000000000000000000000000:0000100000011000:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:00000000:   1  :    0    :    0
   00620818:00: 18 :  04  :   0  :11111111111111111111111111111110:00000000000000000000000000000100:0000100000011000:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:ffffffff:fffffff8:   0  :    1    :    0
   00620818:00: 18 :  04  :   0  :00000000111111111111111111111111:00000000111111111111111111111111:0000100000011000:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:0000ffff:fe000001:   0  :    0    :    0
```

2.4.6. MULTU

Hardware: 32-bit x 32-bit multiplier

Theory: Since the operands are unsigned, use unsigned registers for them.

Registers: reg_A_u, reg_B_u, hi, lo

Result: {hi, lo} = gr1 * gr2

Overflow flag: 0

Zero flag: Set if {hi, lo} = 0.

Negative flag: 0

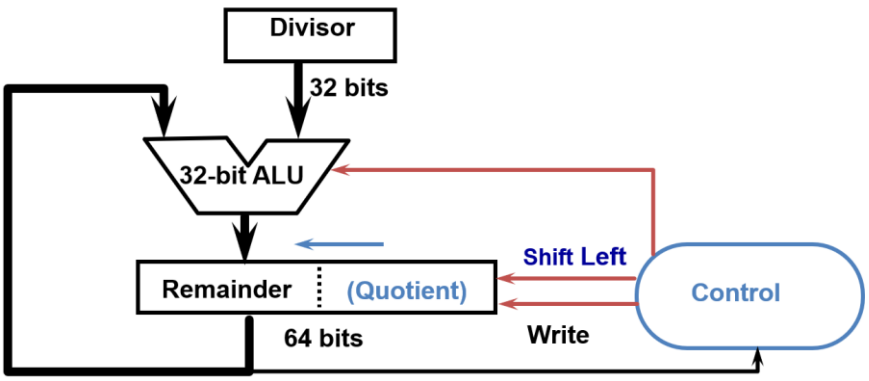
Test Result:

```
//MULTU: 2147483649 * 1 = 2147483649
#10 i_datain<=32'b000000_00011_00010_00001_00000_011001;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

```
|Instruction:op:func:ALUctr:ALUsrc:                gr1                :                gr2                :      imm       :shamt:                c                :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :    x    :    x
   00620819:00: 19 :  05  :   0  :10000000000000000000000000000001:00000000000000000000000000000001:0000100000011001:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:80000001:   0  :    0    :    0
```

2.4.7. DIV

Hardware: 32-bit / 32-bit divisor



Theory:

Since a 32-bit / 32-bit division gets a 32-bit quotient and a 32-bit remainder, we need two 32-bit registers hi and lo to store the result.

Algorithm

For positive numbers:

1. Place the dividend (reg_A) into register lo. Set register hi to be 0. Shift register {hi, lo} left by 1 bit.

Repeat the following steps for 32 times:

2. Subtract divisor (reg_B) from the register hi. Place the result in register hi.
3. If register hi >= 0, shift register {hi, lo} left by 1 bit. Set lo[0] to be 1. Else if register hi < 0, restore original value by adding Divisor divisor (reg_B) to register hi. Place sum in register hi. Also shift register {hi, lo} left by 1 bit. Set lo[0] to be 0.
4. Shift register hi right by 1 bit.

For negative numbers, do the following steps:

1. Remember the sign of each operand.
2. Convert any negative number to its 2's complement (absolute value).
3. Do the division as the positive numbers.
4. If one operand is negative and the other is positive, convert the result to its 2's complement.

Registers: reg_A, reg_B, hi, lo

Result: lo = gr1 / gr2, hi = gr1 % gr2

Overflow flag: Set if an overflow occurs.

Zero flag: Set if lo = 0.

Negative flag: lo[31]

Test Result:

```
//DIV: 8 / 3 = 2 ... 2
#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_1000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;

//DIV: 8 / -3 = -2 ... 2 negative
#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_1000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1101;

//DIV: 2 / -3 = 0 ... 2 zero
#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1101;

//DIV: -2147483648 / -1 = overflow
#10 i_datain<=32'b000000_00011_00010_00001_00000_011010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1                :              gr2                :      imm      :shamt:              c              :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :  x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    :     x
0062081a:00: 1a :  06  :  0  :00000000000000000000000000001000:00000000000000000000000000000011:0000100000011010:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000002:00000002:   0  :     0    :     0
0062081a:00: 1a :  06  :  0  :00000000000000000000000000001000:11111111111111111111111111111101:0000100000011010:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000002:fffffffe:   0  :     1    :     0
0062081a:00: 1a :  06  :  0  :00000000000000000000000000000010:11111111111111111111111111111101:0000100000011010:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000002:00000000:   1  :     0    :     0
0062081a:00: 1a :  06  :  0  :10000000000000000000000000000000:11111111111111111111111111111111:0000100000011010:00000:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:00000000:80000000:   0  :     1    :     1
```

2.4.8. DIVU

Hardware: 32-bit / 32-bit divisor

Theory: Since the operands are unsigned, use unsigned registers for them.

Registers: reg_A_u, reg_B_u, hi, lo

Result: lo = gr1 / gr2, hi = gr1 % gr2

Overflow flag: 0

Zero flag: Set if lo = 0.

Negative flag: 0

Test Result:

```
//DIVU: 2147483648 / 1 = 2147483648 ... 0
#10 i_datain<=32'b000000_00011_00010_00001_00000_011011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

2.4.9. AND / ANDI

Hardware: AND gates

Theory: AND can be done by bitwise operation. imm should be zero-extended.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 & gr2 / c = gr1 & [zero-extended] imm

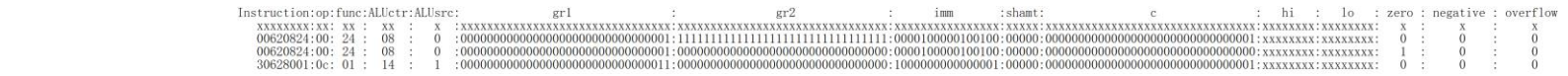
Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//AND: 1 & 2**32-1 = 1
#10 i_datain<=32'b000000_00011_00010_00001_00000_100100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

//AND: 1 & 0 = 0
#10 i_datain<=32'b000000_00011_00010_00001_00000_100100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
//ANDI: 3 & 2**15+1 = 1
#10 i_datain<=32'b001100_00011_00010_1000000000000001;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```



2.4.10. NOR

Hardware: NOR gates

Theory: NOR can be done by bitwise operation.

Registers: reg_A, reg_B, reg_C

Result: c = ~ (gr1 | gr2)

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//NOR: 4294967041 nor 241 = 14
#10 i_datain<=32'b000000_00011_00010_00001_00000_100111;
gr1<=32'b1111_1111_1111_1111_1111_1111_0000_0001;
gr2<=32'b0000_0000_0000_0000_0000_0000_1111_0001;

//NOR: 1 nor 2**32-1 = 0
#10 i_datain<=32'b000000_00011_00010_00001_00000_100111;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

2.4.11. OR / ORI

Hardware: OR gates

Theory: OR can be done by bitwise operation. imm should be zero-extended.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 | gr2 / c = gr1 | [zero-extended] imm

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//OR: 11 | 4 = 15
#10 i_datain<=32'b000000_00011_00010_00001_00000_100101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_1011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//OR: 8 | 0 = 8
#10 i_datain<=32'b000000_00011_00010_00001_00000_100101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_1000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
//ORI: 3 & 2**15+5 = 2**15+7
#10 i_datain<=32'b001101_00011_00010_1000000000000101;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0000;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1               :               gr2                :      imm        :shamt:                c                :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x   :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:  x   :    x     :    x
   00620825:00: 25 :  0a  :   0   :00000000000000000000000000001011:00000000000000000000000000000100:0000100000100101:00000:00000000000000000000000000001111:xxxxxxxx:xxxxxxxx:  0   :    0     :    0
   00620825:00: 25 :  0a  :   0   :00000000000000000000000000001000:00000000000000000000000000000000:0000100000100101:00000:00000000000000000000000000001000:xxxxxxxx:xxxxxxxx:  0   :    0     :    0
   34628005:0d: 05 :  15  :   1   :00000000000000000000000000000011:00000000000000000000000000000000:1000000000000101:00000:00000000000000001000000000000111:xxxxxxxx:xxxxxxxx:  0   :    0     :    0
```

2.4.12. XOR / XORI

Hardware: XOR gates

Theory: XOR can be done by bitwise operation. imm should be zero-extended.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 ^ gr2 / c = gr1 ^ [zero-extended] imm

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//XOR: 2**15 ^ 2**32-1 = 2*31-1
#10 i_datain<=32'b000000_00011_00010_00001_00000_100110;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

//XOR: 3 ^ 7 = 4
#10 i_datain<=32'b000000_00011_00010_00001_00000_100110;
```

```
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0111;

//XORI: 1 ^ 6 = 7
#10 i_datain<=32'b001110_00011_00010_0000000000000110;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1              :              gr2              :      imm       :shamt:               c               :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    :     x
   00620826:00: 26 :  0b  :   0  :10000000000000000000000000000000:11111111111111111111111111111111:0000100000100110:00000:01111111111111111111111111111111:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620826:00: 26 :  0b  :   0  :00000000000000000000000000000011:00000000000000000000000000000111:0000100000100110:00000:00000000000000000000000000000100:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   38620006:0e: 06 :  16  :   1  :00000000000000000000000000000001:11111111111111111111111111111111:0000000000000110:00000:00000000000000000000000000000111:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

2.4.13. BEQ / BNE

Hardware: 32-bit full adder, invertor

Theory: For the ALU, BEQ / BNE is the same as SUB. If the zero flag is 1 / 0,

PC will branch.

Registers: reg_A, reg_B, imm0, reg_C

Result: c = gr1 – gr2

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

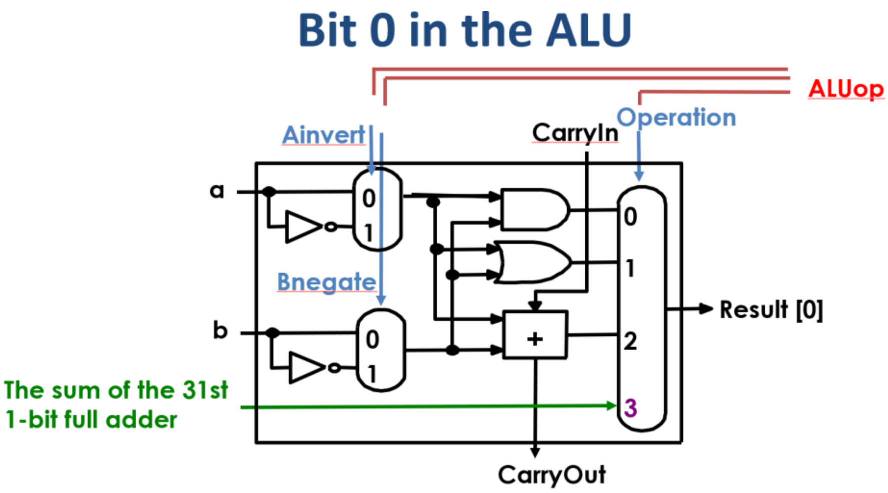
```
//BEQ: 3 - 3 = 0 zero
#10 i_datain<=32'b000100_00011_00010_0000000000000111;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;

//BEQ: 4 - 3 != 0
#10 i_datain<=32'b000100_00011_00010_0000000000000111;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0100;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;

//BNE: 3 - 3 = 0 zero
#10 i_datain<=32'b000101_00011_00010_0000000000000111;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
//BNE: 4 - 3 != 0
#10 i_datain<=32'b000101_00011_00010_0000000000000111;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0100;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1               :               gr2               :      imm       :shamt:                c               :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:  x   :    x     :    x
   10620007:04: 07 :  02  :   0  :00000000000000000000000000000011:00000000000000000000000000000011:0000000000000111:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:  1   :    0     :    0
   10620007:04: 07 :  02  :   0  :00000000000000000000000000000100:00000000000000000000000000000011:0000000000000111:00000:00000000000000000000000000000001:xxxxxxxx:xxxxxxxx:  0   :    0     :    0
   14620007:05: 07 :  02  :   0  :00000000000000000000000000000011:00000000000000000000000000000011:0000000000000111:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:  1   :    0     :    0
   14620007:05: 07 :  02  :   0  :00000000000000000000000000000100:00000000000000000000000000000011:0000000000000111:00000:00000000000000000000000000000001:xxxxxxxx:xxxxxxxx:  0   :    0     :    0
```

2.4.14. SLT / SLTI

Hardware: 32-bit full adder, invertor



Theory:

For the ALU, SLT is based on reg_A – reg_B. If the first operand (reg_A) is less than the second operand (reg_B), then reg_A – reg_B will be negative. In other words, the sum of the 31^{st} 1-bit full adder will be 1. Otherwise, it will be 0.

As for the result of SLT, the first 31 bits are always 0, while the 32^{nd} bit is the same as the sum of the 31^{st} 1-bit full adder.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 < gr2 / c = gr1 < [sign-extended] imm

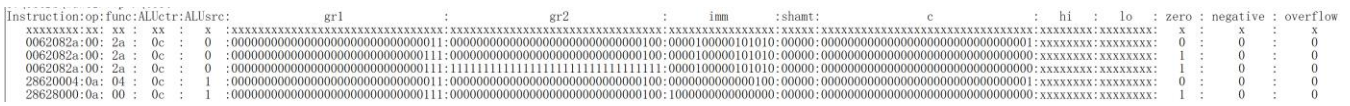
Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//SLT: 3 < 4 : 1
#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//SLT: 7 < 4 : 0
#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//SLT: 7 < -1 : 0
#10 i_datain<=32'b000000_00011_00010_00001_00000_101010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

//SLTI: 3 < 4 : 1
#10 i_datain<=32'b001010_00011_00010_0000000000000100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//SLTI: 7 < -32768 : 0
#10 i_datain<=32'b001010_00011_00010_1000000000000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;
```



```
Instruction:op:func:ALUctr:ALUsrc:              gr1                :              gr2                :      imm       :shamt:              c                :  hi  :  lo  : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :  x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    :     x
   0062082a:00: 2a :  0c  :  0  :00000000000000000000000000000011:00000000000000000000000000000100:0000100000101010:00000:00000000000000000000000000000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   0062082a:00: 2a :  0c  :  0  :00000000000000000000000000000111:00000000000000000000000000000100:0000100000101010:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   0062082a:00: 2a :  0c  :  0  :00000000000000000000000000000111:11111111111111111111111111111111:0000100000101010:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   28620004:0a: 04 :  0c  :  1  :00000000000000000000000000000011:00000000000000000000000000000100:0000000000000100:00000:00000000000000000000000000000001:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   28628000:0a: 00 :  0c  :  1  :00000000000000000000000000000111:00000000000000000000000000000100:1000000000000000:00000:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
```

2.4.15. SLTU / SLTIU

Hardware: 32-bit full adder, invertor

Registers: reg_A_u, reg_B_u, reg_C

Result: c = gr1 < gr2 / c = gr1 < [sign-extended] imm

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: 0

Test Result:

```
//SLTIU: 7 < 2**15 : 1
#10 i_datain<=32'b001011_00011_00010_1000000000000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;

//SLTU: 7 < 2**32-1 : 1
#10 i_datain<=32'b000000_00011_00010_00001_00000_101011;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0111;
gr2<=32'b1111_1111_1111_1111_1111_1111_1111_1111;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1              :              gr2              :      imm       :shamt:              c              :   hi   :   lo   : zero : negative : o
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :      x    :
   2c628000:0b: 00 :  0d  :   1  :00000000000000000000000000000111:11111111111111111111111111111111:1000000000000000:00000:00000000000000000000000000000001:xxxxxxxx:xxxxxxxx:   0  :      0    :
   0062082b:00: 2b :  0d  :   0  :00000000000000000000000000000111:11111111111111111111111111111111:0000100000101011:00000:00000000000000000000000000000001:xxxxxxxx:xxxxxxxx:   0  :      0    :
```

2.4.16. LW / SW

Hardware: 32-bit full adder

Theory: For the ALU, LW and SW is the same as ADDI

Registers: reg_A, reg_B, reg_C

Result: c = gr1 + [sign-extended] imm

Overflow flag: Set if an overflow occurs.

Zero flag: Set if reg_C = 0.

Negative flag: reg_C[31]

Test Result:

```
//LW: 8 + 4 = 12
#10 i_datain<=32'b100011_00011_00010_0000000000000100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_1000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;

//SW: 2147483648 + -4 = 2147483644
#10 i_datain<=32'b100011_00011_00010_1111111111111100;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0100;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1              :              gr2              :      imm       :shamt:              c              :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :      x    :      x
   8c620004:23: 04 :  00  :   1  :00000000000000000000000000001000:00000000000000000000000000000100:0000000000000100:00000:00000000000000000000000000001100:xxxxxxxx:xxxxxxxx:   0  :      0    :      0
   8c62fffc:23: 3c :  00  :   1  :10000000000000000000000000000000:00000000000000000000000000000100:1111111111111100:11111:01111111111111111111111111111100:xxxxxxxx:xxxxxxxx:   0  :      0    :      1
```

2.4.17. SLL / SLLV

Hardware: 32-bit shift register

Theory: The data-in of the shift register is always 0.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 << shamt / c = gr1 << gr2

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: reg_C[31]

Test Result:

```
//SLL: 3 << 1 = 6
#10 i_datain<=32'b000000_00011_00010_00001_00001_000000;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0111;

//SLL: 2**31 << 1 = 0
#10 i_datain<=32'b000000_00011_00010_00001_00001_000000;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0111;

//SLLV: 3 << 1 = 6
#10 i_datain<=32'b000000_00011_00010_00001_00111_000100;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0011;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0001;
```

Instruction	op	func	ALUctr	ALUsrc	gr1	gr2	imm	shamt	c	hi	lo	zero	negative	overflow
xxxxxxxx	xx	xx	xx	x	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx	xxxxx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxx	xxxxxxxx	x	x	x
00620840	00	00	0e	0	00000000000000000000000000000011	00000000000000000000000000000111	0000100001000000	00001	00000000000000000000000000000110	xxxxxxxx	xxxxxxxx	0	0	0
00620840	00	00	0e	0	10000000000000000000000000000000	00000000000000000000000000000111	0000100001000000	00001	00000000000000000000000000000000	xxxxxxxx	xxxxxxxx	1	0	0
006209c4	00	04	0f	0	00000000000000000000000000000011	00000000000000000000000000000001	0000100111000100	00111	00000000000000000000000000000110	xxxxxxxx	xxxxxxxx	0	0	0

2.4.18. SRL / SRLV

Hardware: 32-bit shift register

Theory: The data-in of the shift register is always 0.

Registers: reg_A, reg_B, reg_C

Result: c = gr1 >> shamt / c = gr1 >> gr2

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: reg_C[31]

Test Result:

```
//SRL: 2**31 >> 21 = 2**10 = 1024
#10 i_datain<=32'b000000_00011_00010_00001_10101_000010;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0111;

//SRL: 2 >> 21 = 0
#10 i_datain<=32'b000000_00011_00010_00001_10101_000010;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0111;

//SRLV: 9 >> 2 = 2
#10 i_datain<=32'b000000_00011_00010_00001_10101_000110;
gr1<=32'b0000_0000_0000_0000_0000_0000_0000_1001;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0010;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1              :              gr2              :      imm       :shamt:              c              :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    :     x
   00620d42:00: 02 :  10  :   0  :10000000000000000000000000000000:00000000000000000000000000000111:00001101010000010:10101:00000000000000000000000010000000:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
   00620d42:00: 02 :  10  :   0  :00000000000000000000000000000010:00000000000000000000000000000111:00001101010000010:10101:00000000000000000000000000000000:xxxxxxxx:xxxxxxxx:   1  :     0    :     0
   00620d46:00: 06 :  11  :   0  :00000000000000000000000000001001:00000000000000000000000000000010:00001101010001110:10101:00000000000000000000000000000010:xxxxxxxx:xxxxxxxx:   0  :     0    :     0
```

2.4.19. SRA / SRAV

Hardware: 32-bit shift register

Theory: The data-in of the shift register is equal to reg_A[31].

Registers: reg_A, reg_B, reg_C

Result: c = gr1 >>> shamt / c = gr1 >>> gr2

Overflow flag: 0

Zero flag: Set if reg_C = 0.

Negative flag: reg_C[31]

Test Result:

```
//SRA: -2147483648 >> 21 = -1024
#10 i_datain<=32'b000000_00011_00010_00001_10101_000011;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0000_0010;

//SRAV: -2147483648 >> 21 = -1024
```

```
#10 i_datain<=32'b000000_00011_00010_00001_00000_000111;
gr1<=32'b1000_0000_0000_0000_0000_0000_0000_0000;
gr2<=32'b0000_0000_0000_0000_0000_0000_0001_0101;
```

```
Instruction:op:func:ALUctr:ALUsrc:              gr1                :              gr2                :      imm       :shamt:              c              :   hi   :   lo   : zero : negative : overflow
   xxxxxxxx:xx: xx :  xx  :   x  :xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxxxxxxxxxx:xxxxx:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :      x     :      x
   00620d43:00: 03 :  12  :   0  :10000000000000000000000000000000:00000000000000000000000000000010:0000110101000011:10101:11111111111111111111111110000000:xxxxxxxx:xxxxxxxx:   0  :      1     :      0
   00620807:00: 07 :  13  :   0  :10000000000000000000000000000000:00000000000000000000000000010101:0000100000000111:00000:11111111111111111111111110000000:xxxxxxxx:xxxxxxxx:   0  :      1     :      0
```

2.5.Discussion & Further Improvement

In the testbench, three pieces of data are updated each time:

i_datain, gr1, gr2

Then, they are processed in the following order:

1. The main module decodes i_datain into opcode, funct, and imm.
2. The control module generates alusrc and aluctr according to funct and opcode.
3. The main module assigns gr2 or imm to register operand2.
4. The main module sends aluctr, gr1, and operand2 to the ALU.

In the ALU module, I used the following statement:

always @(aluctr, gr1, operand2)

where aluctr is the control signal, gr1 is general register 1, and operand is general register 2 or sign-extended immediate.

When testing my Verilog code, I found out that the four ALU inputs were not synchronized. In the ALU, gr1 is updated earlier, while aluctr and operand2 are updated a little bit later. Hence, for an instant, the ALU will generate the result from the current gr1 and the previous aluctr and operand2. However, this instant is so short that the monitor will automatically ignore the result. So, this asynchronous issue will not affect the result.

In the project 4, a clock signal will be added to the ALU. Naturally, this problem will be completely fixed.