Report

Block Diagram

In my ALU, the following registers are used:

Name	Size	Function
opcode	6	Store the op part ([31:26]) of given command line
func	6	Store the func ([5:0]) part of given command line
crudeimm	16	Store the im ([15:0]) part of given command line
z/n/vf	1	zero/negative/overflow flag
b/l/sc	1	Branch/load/store control signal
SignExt	1	Store the information that if the immediate number should be sign-extended
reg_A/B/C	32	Store data for ALU computation and result
hi/lo	32	Store mult(u) and div(u) result
imm	32	Store the sign-extended immediate number
mul	63	Store the crude mult(u) result and will be distributed into hi & lo
ALUop	4	Store the corresponding arithmetic operation of ALU
FlagEn	2	Store the information of the specific flag (z/nf) that should be examined
ALUsrcA	1	Store the information that if reg_A should be shamt part of instruction or gr1
ALUsrcB	1	Store the information that if reg_B should be the immediate number or gr2

The different ALUop indicates:

ALUop	Actual operation
0	+
1	-
2	*
3	/
4	| (or)
5	& (and)
6	nor
7	^ (xor)
8	<< (shift left)
9	>> (shift right)
10	A special operation for sltu

At the beginning, the opcode, func and b/l/sc registers will be firstly assigned. Then the ALU will start to examine the structure of given instruction by examining opcode. The result will be either i structure or r structure: if the instruction is i structure, the crudeimm register will be assigned. Then the ALUop, SignExt, ALUsrcA/B, FlagEn and unsign registers will be assigned according to the instruction line.

Then the imm register will be assigned. The least significant 16 digits will be the same as crudeimm; meanwhile, if the SignExt is set to 1 while the crudeimm is negative (the most significant bit is 1), the most significant 16 digits will be set to 1; otherwise, they will be set to 0.

The ALUsrc registers will also be examined. The reg_A will be assigned by the shamt part of instruction if ALUsrcA is set to 1; otherwise it will be the given number of gr1. Similarly, the reg_B will be assigned by the immediate number, or the given number of gr2.

The flags and branch control signal will be assigned during and after ALU operations. During the additive and subtract operation, the overflow signal will be assigned. All three registers will be examined if they are positive or negative, thus it can be examined if this operation overflows. If overflowed and the unsign register is set to 0, the overflow flag will be set to 1.
The zero and negative flag will be examined after ALU operations. It will only examine the reg_C: they will be set to 1 if reg_C is zero or negative. So for multiplicative and divisive operation, the flags will not be assigned.
At last, the FlagEn registers will be examined. If the FlagEn is 01, it means that the negative flag should be examined (e.g. slt and slti). In this situation, the reg_C will be the value of negative flag, that is, 1 or 0. If the FlagEn is 10 (bne) and 11 (beq), the zero flag will be examined. If the zero flag equals to the FlagEn[0], the branch control signal will be set to 1.

Explanation of all required instructions

sll
The sll function provides logical left shift with shamt part of instruction and rt.
Input:

i_datain	gr2
000000 00000 00001 00010 00001 000000	1101_1101_1101_1101_1101_1101_1101_1101
000000 00000 00001 00010 00010 000000	1101_1101_1101_1101_1101_1101_1101_1101
000000 00000 00001 00010 00001 000000	0100_0000_0100_0000_0100_0000_0100_0000
000000 00000 00001 00010 00100 000000	0100_0000_0100_0000_0100_0000_0100_0000

Output:

gr2	c	reg_A	reg_B	reg_C	zero	neg
dddddddd	bbbbbbba	00000001	dddddddd	bbbbbbba	0	1
dddddddd	77777774	00000002	dddddddd	77777774	0	0
40404040	80808080	00000001	40404040	80808080	0	1
40404040	04040400	00000004	40404040	04040400	0	0

As bbbbbbba and 80808080 are negative, the neg flag is enabled.

srl
The srl function provides logical right shift with shamt part of instruction and rt.
Input:

i_datain	gr2
000000 00000 00001 00010 00001 000010	1101_1101_1101_1101_1101_1101_1101_1101
000000 00000 00001 00010 00010 000010	1101_1101_1101_1101_1101_1101_1101_1101
000000 00000 00001 00010 00001 000010	0100_0000_0100_0000_0100_0000_0100_0000
000000 00000 00001 00010 00100 000010	0100_0000_0100_0000_0100_0000_0100_0000

Output:

gr2	c	reg_A	reg_B	reg_C	zero	overflow	neg
dddddddd	6eeeeeee	00000001	dddddddd	6eeeeeee	0	X	0
dddddddd	37777777	00000002	dddddddd	37777777	0	X	0
40404040	20202020	00000001	40404040	20202020	0	X	0
40404040	04040404	00000004	40404040	04040404	0	X	0

srl provides data right shift without sign-extension. Therefore the neg flag is always 0.

sra

The sra function provides arithmetic right shift with shamt part of instruction and rt.

Input:

i_datain	gr2
000000 00000 00001 00010 00001 000011	1101_1101_1101_1101_1101_1101_1101_1101
000000 00000 00001 00010 00010 000011	1101_1101_1101_1101_1101_1101_1101_1101
000000 00000 00001 00010 00001 000011	0100_0000_0100_0000_0100_0000_0100_0000
000000 00000 00001 00010 00100 000011	0100_0000_0100_0000_0100_0000_0100_0000

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
eeeeeeee	00000001	dddddddd	eeeeeeee	0	X	1
f7777777	00000002	dddddddd	f7777777	0	X	1
20202020	00000001	40404040	20202020	0	X	0
04040404	00000004	40404040	04040404	0	X	0

sra provides data right shift with sign-extension. Thus for first two examples, the neg flag is 1.

sllv

The sllv function provides logical left shift with rs and rt (rt<<rs).

Input:

i_datain	gr2	gr1
000000 00000 00001 00010 00000 000100	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 000100	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00010 00000 000100	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 000100	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

c	reg_A	reg_B	reg_C	zero	neg
bbbbbbba	00000001	dddddddd	bbbbbbba	0	1
77777774	00000002	dddddddd	77777774	0	0
80808080	00000001	40404040	80808080	0	1

04040400	00000004	40404040	04040400	0	0

srlv

The srlv function provides logical right shift with rs and rt (rt>>rs).

Input:

i_datain	gr2	gr1
000000 00000 00001 00010 00000 000110	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 000110	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00010 00000 000110	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 000110	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

c	reg_A	reg_B	reg_C	zero	neg
6eeeeeee	00000001	dddddddd	6eeeeeee	0	0
37777777	00000002	dddddddd	37777777	0	0
20202020	00000001	40404040	20202020	0	0
04040404	00000004	40404040	04040404	0	0

srav

The srav function provides arithmetic right shift with rs and rt (rt>>>rs).

Input:

i_datain	gr2	gr1
000000 00000 00001 00010 00000 000111	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 000111	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00010 00000 000111	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 000111	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

c	reg_A	reg_B	reg_C	zero	neg
eeeeeeee	00000001	dddddddd	eeeeeeee	0	0
f7777777	00000002	dddddddd	f7777777	0	0
20202020	00000001	40404040	20202020	0	0
04040404	00000004	40404040	04040404	0	0

The sll/srl/srav resembles sll, srl and sra.

mult

The mult instruction provides multiplication between rs and rt. The hi stores the most significant 32 bits, while the lo stores the least significant 32 bits.

The mult instruction could be processed by following steps:

1. Shift left the product for 1 bit.
2. If the least significant bit of the multiplicand is 1, add the multiplier to the product.
3. Shift right the multiplicand for 1 bit. (logical)
4. If the multiplicand has not been shifted for 32 times, go to step 1.
5. If the multiplicand and the multiplier is of different symbol, set all bits before the most significant bit to 1.

Input:

i_datain	gr2	gr1
000000 00000 00001 00010 00000 011000	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011000	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00010 00000 011000	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011000	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

reg_A	reg_B	hi	lo
00000001	dddddddd	ffffffff	dddddddd
00000002	ffffffff	ffffffff	fffffffe
0000000a	40404040	00000002	82828280
00000004	40404040	00000001	01010100

multu

The mult instruction provides multiplication between rs and rt. The hi stores the most significant 32 bits, while the lo stores the least significant 32 bits. Both rs and rt are unsigned.

The multu instruction could be processed by following steps:

1. Shift left the product for 1 bit.
2. If the least significant bit of the multiplicand is 1, add the multiplier to the product.
3. Shift right the multiplicand for 1 bit. (logical)
4. If the multiplicand has not been shifted for 32 times, go to step 1.

Input:

i_datain	gr2	gr1
000000 00000 00001 00010 00000 011001	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011001	1101_1101_1101_1101	0000_0000_0000_0000

	1101_1101_1101_1101	0000_0000_0000_0010
000000 00000 00001 00010 00000 011001	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011001	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

reg_A	reg_B	hi	lo
00000001	dddddddd	00000000	dddddddd
00000002	ffffffff	00000001	fffffffe
0000000a	40404040	00000002	82828280
00000004	40404040	00000001	01010100

div

The div instruction provides division for rs by rt (rs/rt) . The hi stores the quotient, while the lo stores the remainder. The remainder is always positive. For example, -3/2=-2, -3%2=1.

The div instruction could be processed by following steps:

1. If the dividend is negative, change it into its 2’s complement.
2. Add 32 ‘0’ bits after the divisor.
3. Shift right the divisor for 1 bit.
4. Shift left the quotient for 1 bit.
5. Subtract the divisor from the dividend.
6. If the dividend changes symbol, add back the divisor; else, set the least significant bit of the quotient to 1.
6. If the divisor has not been shifted for 32 times, go to step 2.
7. If the dividend was negative and the remainder is not 0, change the quotient into its 2’s complement, minus 1, and the new remainder is the divisor minus the old remainder.

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00000 011010	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011010	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00010 00000 011010	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_1001
000000 00000 00001 00010 00000 011010	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

reg_A	reg_B	hi	lo
dddddddd	00000001	00000000	dddddddd
dddddddd	00000002	00000001	eeeeeeee

40404040	00000009	00000004	0723955c
40404040	00000004	00000000	10101010

The hi stores the remainder, while the lo stores the quotient.

divu

The divu instruction provides division for rs by rt (rs/rt) . The hi stores the quotient, while the lo stores the remainder. Both rs and rt are unsigned. The remainder is always positive. For example, -3/2=-2, -3%2=1.

The divu instruction could be processed by following steps:

1. Add 32 '0' bits after the divisor.
2. Shift right the divisor for 1 bit.
3. Shift left the quotient for 1 bit.
4. Subtract the divisor from the dividend.
5. If the dividend changes symbol, add back the divisor; else, set the least significant bits of the quotient to 1.
6. If the divisor has not been shifted for 32 times, go to step 2.

Input:

i_datain	gr2	gr1
000000 00000 00001 00010 00000 011011	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011011	1101_1101_1101_1101 1101_1101_1101_1101	0000_0000_0000_0000 0000_0000_0000_0010
000000 00000 00001 00010 00000 011011	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00000 011011	0100_0000_0100_0000 0100_0000_0100_0000	0000_0000_0000_0000 0000_0000_0000_0100

Output:

reg_A	reg_B	hi	lo
dddddddd	00000001	00000000	dddddddd
dddddddd	00000002	00000001	6eeeeeee
40404040	00000009	00000004	0723955c
40404040	00000004	00000000	10101010

In instructions mult(u) and div(u), the flags are not enabled.

add

The add instruction provides addition with rs and rt.

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100000	1000_0000_0000_0000 0000_0000_0000_0000	1111_1111_1111_1111 0000_0000_0000_0000
000000 00000 00001 00010 00010 100000	0000_0000_0000_0000 0000_0000_0000_0001	1111_1111_1111_1111 1111_1111_1111_1110

000000 00000 00001 00010 00001 100000	0111_1111_1111_1111 1111_1111_1111_1111	0111_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00100 100000	0000_0000_0000_0000 0000_0000_0000_0001	0111_1111_1111_1111 1111_1111_1111_1110

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
7fff0000	80000000	ffff0000	7fff0000	0	1	0
ffffffff	00000001	fffffffe	ffffffff	0	0	1
fffffffd	7fffffff	7ffffffe	fffffffd	0	1	1
7fffffff	00000001	7ffffffe	7fffffff	0	0	0

The overflow flag will be enabled if the result overflows.

addu

The add instruction provides addition with rs and rt. The overflow flag will be ignored.

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100001	1000_0000_0000_0000 0000_0000_0000_0000	1111_1111_1111_1111 0000_0000_0000_0000
000000 00000 00001 00010 00010 100001	0000_0000_0000_0000 0000_0000_0000_0001	1111_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00001 100001	0111_1111_1111_1111 1111_1111_1111_1111	0111_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00100 100001	0000_0000_0000_0000 0000_0000_0000_0001	0111_1111_1111_1111 1111_1111_1111_1110

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
7fff0000	80000000	ffff0000	7fff0000	0	0	0
ffffffff	00000001	fffffffe	ffffffff	0	0	1
fffffffd	7fffffff	7ffffffe	fffffffd	0	0	1
7fffffff	00000001	7ffffffe	7fffffff	0	0	0

The overflow flag is disabled.

sub

The sub instruction provides subtraction with rs and rt(rs-rt).

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100010	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00010 100010	0000_0000_0000_0000 0000_0000_0000_0010	1111_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00001 100010	0111_1111_1111_1111	1000_1111_1111_1111

	1111_1111_1111_1111	1111_1111_1111_1110
000000 00000 00001 00010 00100 100010	1111_1111_1111_1111 1111_1111_1111_1100	0111_1111_1111_1111 1111_1111_1111_1110

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
00000000	00000001	00000001	00000000	1	0	0
00000004	00000002	fffffffe	00000004	0	0	0
00000001	7fffffff	8ffffffe	f0000001	0	1	1
7ffffffe	fffffffc	7ffffffe	7ffffffe	0	1	0

The overflow flag will be enabled if the result overflows.

subu

The sub instruction provides subtraction with rs and rt(rs-rt). The overflow flag is ignored.

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100011	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0000_0000_0000_0001
000000 00000 00001 00010 00010 100011	0000_0000_0000_0000 0000_0000_0000_0010	1111_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00001 100011	0111_1111_1111_1111 1111_1111_1111_1111	1000_1111_1111_1111 1111_1111_1111_1110
000000 00000 00001 00010 00100 100011	1111_1111_1111_1111 1111_1111_1111_1100	0111_1111_1111_1111 1111_1111_1111_1110

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
00000000	00000001	00000001	00000000	1	0	0
00000004	00000002	fffffffe	00000004	0	0	0
00000001	7fffffff	8ffffffe	f0000001	0	0	1
7ffffffe	fffffffc	7ffffffe	7ffffffe	0	0	0

The overflow flag is disabled.

and

The and instruction provides logical and operation with rs and rt(rs&rt).

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100000	0001_0010_0011_0100 1001_1010_1011_1100	0101_0110_0111_1000 1101_1110_1111_0000

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
12309ab0	12349abc	5678def0	12309ab0	0	0	0

or

The or instruction provides logical or operation with rs and rt(rs|rt).

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100001	0001_0010_0011_0100 1001_1010_1011_1100	0101_0110_0111_1000 1101_1110_1111_0000

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
567cdefc	12349abc	5678def0	567cdefc	0	0	0

xor

The xor instruction provides logical exclusive-or operation with rs and rt(rs^rt).

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100010	0001_0010_0011_0100 1001_1010_1011_1100	0101_0110_0111_1000 1101_1110_1111_0000

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
444c444c	12349abc	5678def0	444c444c	0	0	0

nor

The nor instruction provides logical nor operation with rs and rt(~(rs|rt)).

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 100011	0001_0010_0011_0100 1001_1010_1011_1100	0101_0110_0111_1000 1101_1110_1111_0000

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
a9832103	12349abc	5678def0	a9832103	0	0	1

slt

The slt instruction compares rs and rt. If rs<rt, the reg_C will be set to 1, else 0.

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 101010	1111_1111_1111_1111 1111_1111_1111_1110	1111_1111_1111_1111 1111_1111_1111_1111
000000 00000 00001 00010 00010 101010	0000_0000_0000_0000 0000_0000_0000_0001	1111_1111_1111_1111 1111_1111_1111_1111
000000 00000 00001 00010 00001 101010	0000_0000_0000_0000	0000_0000_0000_0000

	0000_0000_0000_0001	0001_0000_0000_0000
000000 00000 00001 00010 00100 101010	1111_1111_1111_1111 1111_1111_1111_1111	0000_0000_0000_0000 0000_0000_0000_0001

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
00000001	fffffffe	ffffffff	00000001	0	0	0
00000000	00000001	ffffffff	00000000	1	0	0
00000001	00000001	00001000	00000001	0	0	0
00000001	ffffffff	00000001	00000001	0	0	0

sltu

The sltu instruction compares rs and rt. If rs<rt, the reg_C will be set to 1, else 0. Both rs and rt are unsigned.

Input:

i_datain	gr1	gr2
000000 00000 00001 00010 00001 101011	1111_1111_1111_1111 1111_1111_1111_1110	1111_1111_1111_1111 1111_1111_1111_1111
000000 00000 00001 00010 00010 101011	0000_0000_0000_0000 0000_0000_0000_0001	1111_1111_1111_1111 1111_1111_1111_1111
000000 00000 00001 00010 00001 101011	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0001_0000_0000_0000
000000 00000 00001 00010 00100 101011	1111_1111_1111_1111 1111_1111_1111_1111	0000_0000_0000_0000 0000_0000_0000_0001

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg
00000001	fffffffe	ffffffff	00000001	0	0	0
00000000	00010000	ffffffff	00000001	0	0	0
00000001	00000001	00001000	00000001	0	0	0
00000001	ffffffff	00000001	00000000	1	0	0

In slt and sltu, the zero flag will be enabled if the reg_C is changed to 0.

beq

The beq instruction compares rs and rt. If rs=rt, the branch signal will be enabled and the reg_C will be set to the sign-extended immediate number, else 0.

Input:

i_datain	gr1	gr2
000100 00001 00010 1111111111111111	1111_1111_1111_1111 1111_1111_1111_1111	1111_1111_1111_1111 1111_1111_1111_1110
000100 00001 00010 1111111111111111	1111_1111_1111_1111 1111_1111_1111_1111	1111_1111_1111_1111 1111_1111_1111_1111
000100 00001 00010 1111111111111111	0000_0000_0000_0000	0000_0000_0000_0000

	0000_0000_0000_0001	0001_0000_0000_0000
000100 00001 00010 1111111111111111	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0000_0000_0000_0001

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg	branch
00000001	ffffffff	fffffffe	00000000	0	0	0	0
00000000	ffffffff	ffffffff	ffffffff	1	0	0	1
00000001	00000001	00000000	00000000	0	0	0	0
00000001	00000001	00000001	ffffffff	1	0	0	1

bne

The beq instruction compares rs and rt. If rs!=rt, the branch signal will be enabled and the reg_C will be set to the sign-extended immediate number, else 0.

Input:

i_datain	gr1	gr2
000101 00001 00010 1111111111111111	1111_1111_1111_1111 1111_1111_1111_1111	1111_1111_1111_1111 1111_1111_1111_1110
000101 00001 00010 1111111111111111	1111_1111_1111_1111 1111_1111_1111_1111	1111_1111_1111_1111 1111_1111_1111_1111
000101 00001 00010 1111111111111111	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0001_0000_0000_0000
000101 00001 00010 1111111111111111	0000_0000_0000_0000 0000_0000_0000_0001	0000_0000_0000_0000 0000_0000_0000_0001

Output:

c	reg_A	reg_B	reg_C	zero	overflow	neg	branch
00000001	ffffffff	fffffffe	ffffffff	0	0	0	1
00000000	ffffffff	ffffffff	00000000	1	0	0	0
00000001	00000001	00000000	ffffffff	0	0	0	1
00000001	00000001	00000001	00000000	1	0	0	0

In beq and bne, the zero and negative flag will not follow the reg_C: if branch signal is enabled, the reg_C stores the offset of the branch.

addi

The add instruction provides addition with rs and zero-extended immediate number.

Input:

i_datain	gr1
001000 00001 00010 1111111111111111	7fff_fffe
001000 00001 00010 1111111111111111	ffff_ffff
001000 00001 00010 1111111111111111	0000_0000
001000 00001 00010 1111111111111111	0000_0001

Output:

c	reg_A	reg_C	zero	overflow	neg
00000001	7ffffffe	8000fffd	0	1	1
00000000	ffffffff	0000fffe	0	0	0
00000001	00000000	0000ffff	0	0	0
00000001	00000001	00010000	0	0	0

addiu

The add instruction provides addition with rs and zero-extended immediate number. The overflow flag will be ignored.

Input:

i_datain	gr1
001001 00001 00010 1111111111111111	7fff_fffe
001001 00001 00010 1111111111111111	ffff_ffff
001001 00001 00010 1111111111111111	0000_0000
001001 00001 00010 1111111111111111	0000_0001

Output:

c	reg_A	reg_C	zero	overflow	neg
00000001	7ffffffe	8000fffd	0	0	1
00000000	ffffffff	0000fffe	0	0	0
00000001	00000000	0000ffff	0	0	0
00000001	00000001	00010000	0	0	0

slti

The slti instruction compares rs and and zero-extended immediate number. If rs<imm, the reg_C will be set to 1, else 0.

Input:

i_datain	gr1
001010 00001 00010 0111111111111111	0000_0ffe
001010 00001 00010 1111111111111111	ffff_ffff
001010 00001 00010 1111111111111100	0000_ffff
001010 00001 00010 1111111111111111	0000_0001

Output:

c	reg_A	reg_C	zero	overflow	neg
00000001	00000ffe	00000001	0	0	0
00000000	ffffffff	00000000	1	0	0
00000001	0000ffff	00000000	1	0	0
00000001	00000001	00000000	1	0	0

sltiu

The sltiu instruction compares rs and and zero-extended immediate number. If rs<imm, the reg_C

will be set to 1, else 0. The rs will be unsigned.

Input:

i_datain	gr1
001010 00001 00010 0111111111111111	0000_fffe
001010 00001 00010 1111111111111111	ffff_ffff
001010 00001 00010 1111111111111100	0000_ffff
001010 00001 00010 1111111111111111	0000_0001

Output:

c	reg_A	reg_C	zero	overflow	neg
00000001	0000fffe	00000001	0	0	0
00000000	ffffffff	00000000	1	0	0
00000001	0000ffff	00000000	1	0	0
00000001	00000001	00000001	1	0	0

For slti(u), the zero flag will be enabled if reg_C is assigned 0.

andi

Input:

i_datain	gr1
001100 00001 00010 1001101010111100	0101_0110_0111_1000_1101_1110_1111_0000

Output:

c	reg_A	reg_C	zero	overflow	neg
00009ab0	5678def0	00009ab0	0	0	0

ori

Input:

i_datain	gr1
001101 00001 00010 1001101010111100	0101_0110_0111_1000_1101_1110_1111_0000

Output:

c	reg_A	reg_C	zero	overflow	neg
567cdefc	5678def0	567cdefc	0	0	0

xori

Input:

i_datain	gr1
001110 00001 00010 1001101010111100	0001_0010_0011_0100_1001_1010_1011_1100

Output:

c	reg_A	reg_C	zero	overflow	neg
444c444c	5678def0	5678444c	0	0	0

lw
Input:

i_datain	gr1
100011 00001 00010 0000000000000001	08000008

Output:

c	reg_A	reg_C	load
0800000c	08000008	0800000c	1

sw
The lw and sw instructions shift left the immediate number for 2 bits then add it to rs. The load/store control signal will be enabled.
Input:

i_datain	gr1
101011 00001 00010 0000000000000001	08000008

Output:

c	reg_A	reg_C	save
0800000c	08000008	0800000c	1

According to practical operation, the outputs above are all correct.

Flags and control signals
There are 3 flags in the ALU:

nf (neg)	Enabled if the result is negative.
vf (overflow)	Enabled if the result is overflowed.
zf (zero)	Enabled if the result is zero.

nf and zf will only examine the reg_C; vf will only examine when the ALU performs addition or subtraction.
For example,

instruction	c	reg_A	reg_B	reg_C	zf	vf	nf
add	7fff0000	80000000	ffff0000	7fff0000	0	1	0
add	ffffffff	00000001	fffffffe	ffffffff	0	0	1
sra	f7777777	00000002	dddddddd	f7777777	0	X	1
bne	00000000	ffffffff	ffffffff	00000000	1	0	0
sub	00000000	00000001	00000001	00000000	1	0	0

For beq and bne, the negative flag will not follow the reg_C. It only shows whether reg_A=reg_B.

instruction	c	reg_A	reg_B	reg_C	zf	vf	nf
beq	00000001	00000001	00000001	ffffffff	1	0	0

There are 3 control signals in the ALU:

bc (branch)	Enabled if the PC is going to branch.
sc (store)	Enabled if the data store step is enabled.
lc (load)	Enabled if the data load step is enabled.

For example,

instruction	c	reg_A	reg_C	save
sw	0800000c	08000008	0800000c	1
instruction	c	reg_A	reg_C	load
lw	0800000c	08000008	0800000c	1

instruction	c	reg_A	reg_B	reg_C	zero	overflow	neg	branch
beq	00000001	ffffffff	fffffffe	00000000	0	0	0	0
beq	00000000	ffffffff	ffffffff	ffffffff	1	0	0	1
bne	00000001	ffffffff	fffffffe	ffffffff	0	0	0	1
bne	00000000	ffffffff	ffffffff	00000000	1	0	0	0

WU Dongyang
118010324