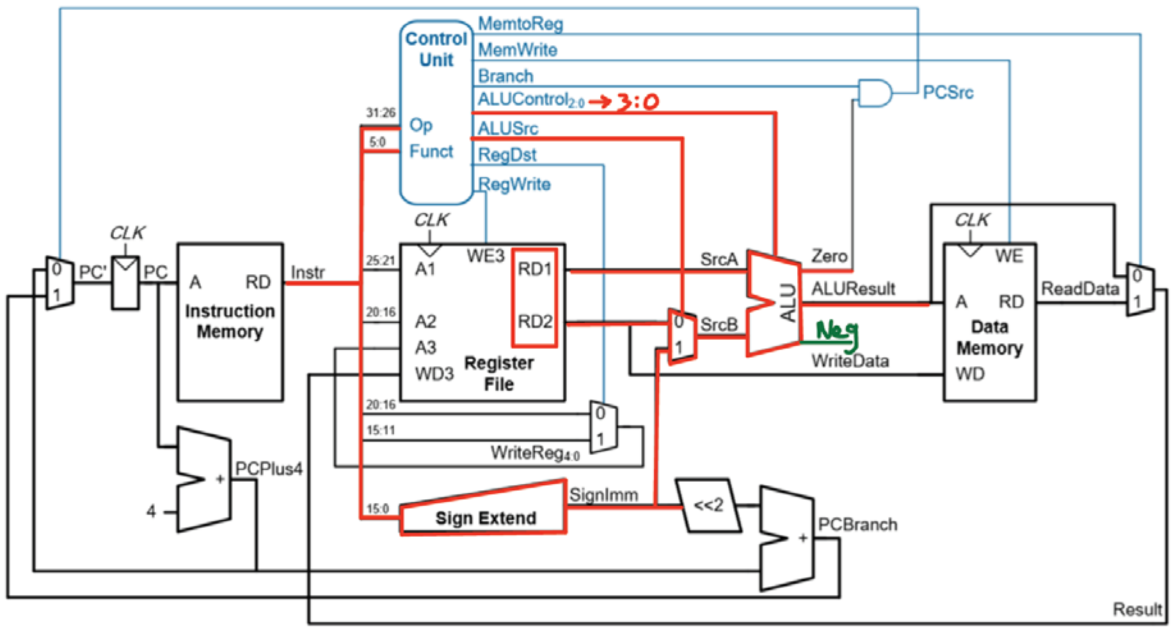
CSC3050 Project 3 Report --- 高梓骐（118010077）

1. Topic

This project asked us to design an arithmetic logic unit (ALU) with a part of the control block of the dataflow. The blocks and wires in red will be designed and tested in this project.



To achieve the goal, the hardware design should decode the instruction (*i_datain*) first, and then display the result of ALU according to the decoded instruction. In reality, the ALU should read the two inputs from the register file. However, in this project, since we do not design the register file block, the user should input the values of the two inputs (*gr1, gr2 for R-type, shamt for shift instructions, and immediate for I-type*) that are considered as the input. The ALU will give the output then.

This project also designs the sign extension block for I-type instructions, and control unit blocks to decode the instructions and control the ALU. Those two parts will also be introduced in detail later.

2. General Logic of the project

This project consists of four parts.

1. Control Unit Block:

   This block is designed in the file *control.v*. This block will decode the instruction. It will read the first 6 bits as the opcode of the instructions, and output the 2-bits *aluOp* and 1-bit *aluSrc*. These two outputs will decide the Alu control input to select which function should the ALU conduct.

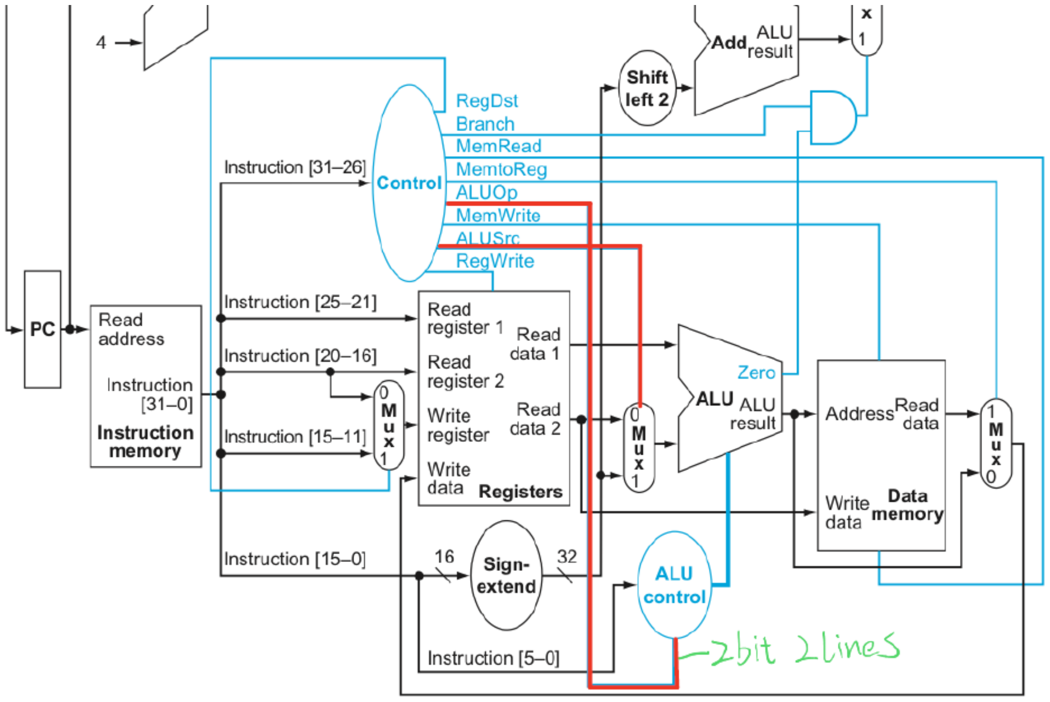


Figure 2.1: A schematic of the ALU Control Block and Control Unit Block.

Figure 2.1 shows how the instructions are decoded. When the opcode of the instruction is 6'b000000, the instruction is R-type, thus the value of the *AluSrc* should be 0, which means we do not need to read the *i_datain[15:0]* as the input of ALU but to use the input from the user. When the opcode is not 6'b000000, the instruction should be I-type. All of those instructions should set the aluSrc to be 1 in order to read the input from the *i_datain[15:0]*. Three frequently used functions *lw*, *sw*, and *beq* are listed separately to improve the efficiency a little. They have the special value of *aluOp*. The meaning of aluOp is speed up the decode of the instructions to improve the efficiency of the ALU Control Block. For example, the ALU Control will consider the instruction as R-type if the *aluOp* is 2'b10.

Instruction Opcode (i_datain[31:26])	**aluSrc**	**aluOp**
000000 (R-type)	0	10
101011(sw)	1	00
100011(lw)	1	00
000100(beq)	1	01
Other Opcode (I-type)	1	11

Figure 2.1 AluSrc and AluOp controbits for different instructions.

2. ALU Control Block:

This block takes three inputs from the Control Unit Block and the instructions (*opcode, funcode, aluctl*), and outputs the 4-bits *aluctl* signal to control the ALU. ALU will conduct its function according to the *aluctl.*

ALU Control Block will first read the *aluOp* to check which kind of instruction it is, following the rule of Figure 2.1. Then it will check whether to read the *funcode* or the *opcode* for I-type instructions and R-type instructions respectively.

In this part, I make some adjustments to the original design in the textbook and the lecture. In the lecture we only deal with *lw, sw, beq,* and few R-type functions. However, we have more than 16 different R-type functions and some I-type instructions. As a result, it is necessary to combine lots of functions together, and let the ALU Control Block to read the instruction opcode to decide the *aluctl* for the I-type functions.

One thing that is need to mention is that though we reduce the ultimate *aluctl* signal to only 4-bits, we have to add more multiplexers to select the inputs and the outputs. For example, *add* and *addu* are assigned to the same *aluctl* signal, which means we always have an overflow flag as one output, though we do not need to check it. Therefore, we need to set another multiplexer to decide whether we need the overflow flag according to the instruction.

Similarly, this multiplexer will also help to control the output of *mul, mulu, sub,* and *subu.*

Another multiplexer is needed for shift instructions. For *sll, sra,* and *srl,* one input of the ALU is the *shamt (i_datain[10:6])*. For *sllv, srav,* and *srlv*, this input should come from the register (it is input by the user in this program). This program check which input should be read according to the fourth bit of the funcode. If this bit is one, ALU read from the user. Otherwise, it read the shamt directly.

The third multiplexer is needed for the negative flag of *slt. slt* is combined with the *sub* function. Thus, we need a multiplexer to decide whether we need this flag.

In conclusion, this block will decide the value of *aluctl* according to the *aluOp, Opcode,* and *funccode.* Figure 2.2 shows the value of *aluctl* and their corresponding functions in this program. Some functions have the same *aluctl* because they are combined as one function. We need three more multiplexers to finish all kinds of Datapath with 4-bits *aluctl.*

AluOp	Opcode	Funcode	ALUctl	Function
00	lw-100011	xxxxxxx	0100	Add
00	sw-101011	xxxxxxx	0100	Add
01	Beq-000100	xxxxxxx	0101	Sub
10	and-000000	100100	0000	And
10	or-000000	100101	0001	Or
10	xor-000000	100110	0010	Xor
10	nor-000000	100111	0011	Nor
10	add-000000	100000	0100	Add
10	addu-000000	100001	0100	Add
10	sub-000000	100010	0101	Sub
10	subu-000000	100011	0101	Sub
10	mul-000000	011000	0110	Mul
10	mulu-000000	011001	0110	Mul
10	div-000000	011010	0111	Div
10	divu-000000	011011	1000	divu
10	slt-000000	101010	0101	sub
10	sltu-000000	101011	1001	sltu
10	sll-000000	000000	1011	sll
10	sllv-000000	000100	1011	sllv
10	srl-000000	000010	1100	srl
10	srlv-000000	000110	1100	srlv
10	sra-000000	000011	1101	sra
10	srav-000000	000111	1101	srav
11	addi-001000	xxxxxxx	0100	add
11	addiu-001001	xxxxxxx	0100	add
11	andi-001100	xxxxxxx	0000	and
11	ori-001101	xxxxxxx	0001	or
11	xori-001110	xxxxxxx	0010	xor
11	bne-000101	xxxxxxx	1010	bne
11	slti-001010	xxxxxxx	0101	sub
11	sltiu-001011	xxxxxxx	1001	sltu

Figure2.2 The value of *aluctl* for all the required MIPS instructions

3. Sign Extension Block:

   This block is important for the I-type instructions. We need 32bits inputs instead of only 16bits from *i_datain[15:0]*. Therefore, it is necessary to have a Sign Extension Block.

   This block is easy to build in Verilog. Just use the *$signed(i_datain[15:0])* to get the sign extended result.

4. ALU Block:

   This block is the key block of this project. It will conduct all the functions from the instruction list. The *aluctl* from Figure2.2 will choose which function will this ALU block conduct. This block will take 3 inputs *aluctl*, *reg_A*, and *reg_B* where *reg_A* and *reg_B* is decided by *i_datain[25:21]* (*$rs*), *i_datain[20:16]($rt)* or *i_datain[15:0] (imm)*. The value of *reg_A* and *reg_B* will be chosen by *aluSrc* mentioned before.

This ALU Block will output 4 possible results *result*, *overflow*, *zero*, and *neg.*

(1) *result* output will be the "ALUResult" of Fig 2.3.

(2) *zero* will be the output for instructions *beq* and *bne.* Also, *beq* is combined with *sub* function. It will check whether the two inputs equal with:

$$neg = ((reg_A - reg_B) == 0)$$

It is the same for *bne* except its result is the logic not of *beq.*

(3) *neg* will be used for instructions of *slt, sltu, slti,* and *sltiu.* Since these four functions are combined with *sub* function, *sub* will also output *neg*. If the result of *reg_A – reg_B* is less than 0, *neg will be set 1.* We can extend *neg* to 32bits to write it back to the Register File.

(4) *overflow* will be used for *mul, add, sub* which will return 1 if overflow occurs when the ALU conduct those three functions.

To check whether the overflow happens in *add* and *sub,* this project checks the overflow with the function:

$$overflow = CarryIn[N-1] \oplus CarryOut[N-1]$$

It is more difficult to check whether the overflow occurs in *mul* because the result of multiplication may range from 32bits to 64bits. This project will check whether the result divided by one input can get the other input just like:

reg_B == (result/reg_A);

If they do, the *overflow* will output 0.

3. Details of Some Functions

(1) *slt, sltv, sra, srav, sll, sllv, sltu, divu*: Those functions need unsigned inputs. As a result, this program required two more registers *u_reg_A*, *u_reg_B* to convert the signed number to unsigned number.

(2) *slt, sra, sll*: Those three functions need special inputs *shamt* from the instruction code (*i_datain[10:6]*). We need one more multiplexer to select the input just like what is mentioned above.

(3) Since I combine lots of functions to one function. Figure 3.1 shows all the instructions that are required in this project and their corresponding combined ALU functions.

Functions of ALU (*aluctl*)	**Instructions**
add (0100)	**add**
	addi
	lw
	sw
	addu
	addiu
sub(0101)	**sub**
	subu
	beq
	slt
	slti

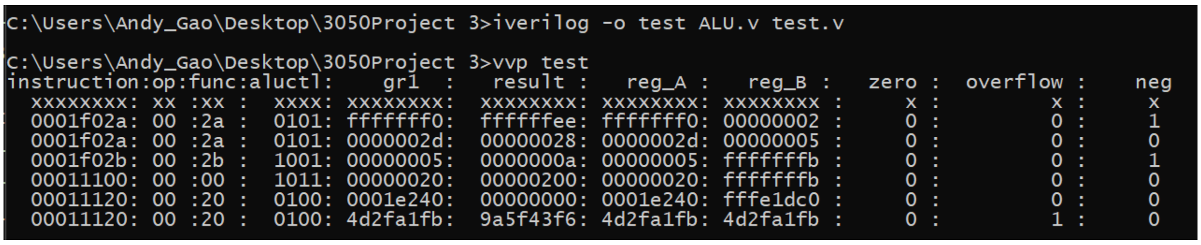
sll(1011)	**sll**
	sllv
srl(1100)	**srl**
	srlv
sra(1101)	**sra**
	srav
and(0000)	**and**
	andi
or(0001)	**or**
	ori
xor(0010)	**xor**
	xori
nor(0011)	**nor**
	nori
mul(0110)	**mul**
	mulu
sltu(1001)	**sltu**
	sltiu
div(0111)	**div**
divu(1000)	**divu**
bne(1010)	**bne**
sltu(1001)	**sltui**

Figure 3.1: Instructions and their corresponding ALU functions with *aluctl.*

4. Test

This project consists of 4 files: *test.v, ALU.v, aluControl.v, control.v*.

My project is compiled and simulated by Icarus Verilog (*iverilog*) in Windows 10. Just like Figure 4.1 shows: enter "iverilog -o test ALU.v test.v" and then enter "vvp test" can get the result of the simulation. All the outputs except *aluctl* are given in hexadecimal form.



```
C:\Users\Andy_Gao\Desktop\3050Project 3>iverilog -o test ALU.v test.v

C:\Users\Andy_Gao\Desktop\3050Project 3>vvp test
instruction:op:func:aluctl:    gr1   :   result :   reg_A :   reg_B :   zero :  overflow :    neg
  xxxxxxxx: xx :xx :  xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :         x :      x
  0001f02a: 00 :2a :  0101: fffffff0:  ffffffee: fffffff0: 00000002 :     0 :         0 :      1
  0001f02a: 00 :2a :  0101: 0000002d:  00000028: 0000002d: 00000005 :     0 :         0 :      0
  0001f02b: 00 :2b :  1001: 00000005:  0000000a: 00000005: fffffffb :     0 :         0 :      1
  00011100: 00 :00 :  1011: 00000020:  00000200: 00000020: fffffffb :     0 :         0 :      0
  00011120: 00 :20 :  0100: 0001e240:  00000000: 0001e240: fffe1dc0 :     0 :         0 :      0
  00011120: 00 :20 :  0100: 4d2fa1fb:  9a5f43f6: 4d2fa1fb: 4d2fa1fb :     0 :         1 :      0
```

Figure4.1: How to run the program with Icarus Verilog

User should enter the *i_datain*, the machine code of MIPS instructions, *gr1*, the first input of ALU, and an optional *gr2*, the second input of ALU which is optional because the program will read the value from *i_datain* for I-type and shift instructions.

I saved some instructions in "test.v" for test with comments for each kind of instructions. But of course, the users can enter the test case by themselves.

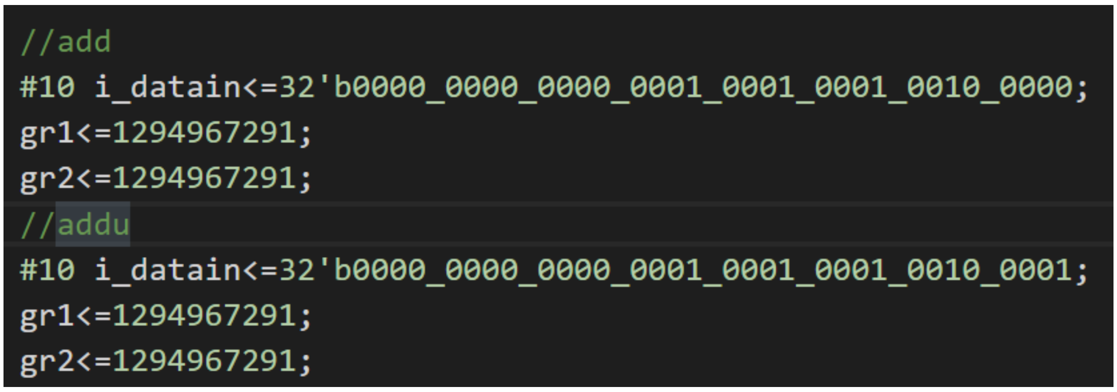
Figure 4.2 shows the meaning of every output:

Output	Meaning and Remark
instruction	Machine Code of MIPS instructions (input by the user)
op (opcode)	The first 6 bits of the instructions *i_datain[31:26]*

func(funcode)	The last 6bits of the instructions *i_datain[5:0]*
aluctl	The value of *aluctl* of this instruction (decoded by ALU Control Block)
gr1	Unsigned number input by the user.
result	ALU Output (as a signed number)
reg_A	The first input of ALU (same as gr1 but a signed number)
reg_B	The second input of ALU (same as gr2 but a signed number)
zero	Zero Flag
overflow	Overflow Flag (1 if overflow occurs)
neg	1 if the first input is less than the second input

Figure 4.2: The meaning of the output variables

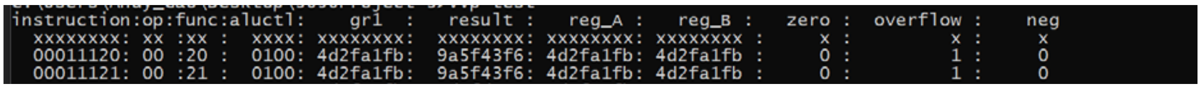
Here are some test cases for all instructions. The test cases should be input just like Figure 4.3.



```
//add
#10 i_datain<=32'b0000_0000_0000_0001_0001_0001_0010_0000;
gr1<=1294967291;
gr2<=1294967291;
//addu
#10 i_datain<=32'b0000_0000_0000_0001_0001_0001_0010_0001;
gr1<=1294967291;
gr2<=1294967291;
```

Figure 4.3: An example of test case

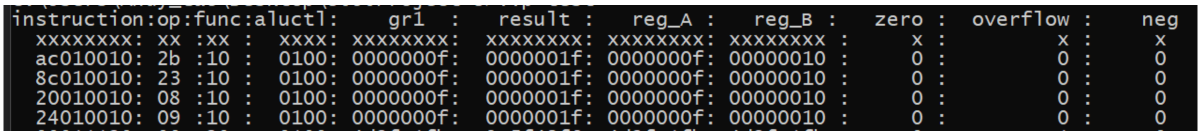
(1) *add, addu:*



```
instruction:op:func:aluctl:    gr1   :   result :   reg_A :   reg_B :   zero :  overflow :    neg
 xxxxxxxx: xx :xx :   xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :          x :     x
 00011120: 00 :20 :   0100: 4d2fa1fb:  9a5f43f6: 4d2fa1fb: 4d2fa1fb :     0 :          1 :     0
 00011121: 00 :21 :   0100: 4d2fa1fb:  9a5f43f6: 4d2fa1fb: 4d2fa1fb :     0 :          1 :     0
```

For both of these two functions, the overflow flag will have its value. It needs one more multiplexer to decide whether we need this flag. The two inputs are positive but the output is negative so the overflow is set 1.

(2) *lw*, *sw*, *addi*, *addiu*:

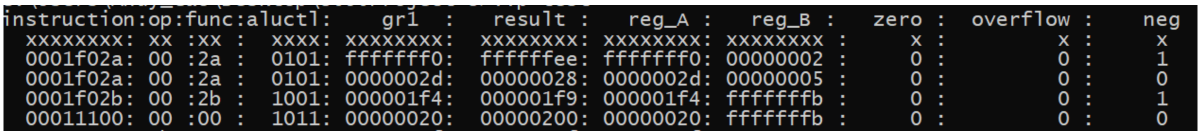


```
instruction:op:func:aluctl:    gr1   :    result :    reg_A :    reg_B :   zero :   overflow :    neg
  xxxxxxxx: xx :xx :   xxxx: xxxxxxxx:   xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :          x :      x
  ac010010: 2b :10 :   0100: 0000000f:   0000001f: 0000000f: 00000010 :     0 :          0 :      0
  8c010010: 23 :10 :   0100: 0000000f:   0000001f: 0000000f: 00000010 :     0 :          0 :      0
  20010010: 08 :10 :   0100: 0000000f:   0000001f: 0000000f: 00000010 :     0 :          0 :      0
  24010010: 09 :10 :   0100: 0000000f:   0000001f: 0000000f: 00000010 :     0 :          0 :      0
```

These four functions are very similar to *add* and *addu*. The only one difference is that *reg_B* comes from *i_datain[15:0]*. In this case *reg_B* is set to be 16.

(3) *slt*, *slti*, *sltiu*, *sltu:*

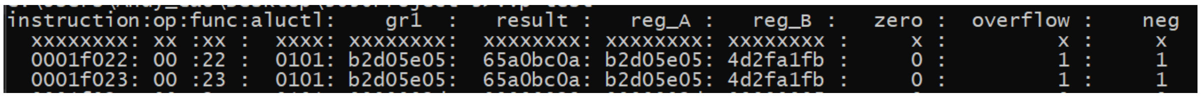
The difference between *slt* and *sltu* is that *sltu* consider the inputs as unsigned numbers. Thus negative 1 is the largest number. The *neg* flag will be used in this function. If the first input is less than the second input, the *neg* flag will be set 1. The first two cases are *slt,* the last two cases are *sltu.* It is easy to find the difference from those four cases.



```
instruction:op:func:aluctl:    gr1   :    result :    reg_A :    reg_B :   zero :   overflow :    neg
 xxxxxxxx: xx :xx :    xxxx: xxxxxxxx:   xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :          x :      x
 0001f02a: 00 :2a :    0101: fffffff0:   ffffffee: fffffff0: 00000002 :     0 :          0 :      1
 0001f02a: 00 :2a :    0101: 0000002d:   00000028: 0000002d: 00000005 :     0 :          0 :      0
 0001f02b: 00 :2b :    1001: 000001f4:   000001f9: 000001f4: fffffffb :     0 :          0 :      1
 00011100: 00 :00 :    1011: 00000020:   00000200: 00000020: fffffffb :     0 :          0 :      0
```

(4) *sub, subu:*

The difference between *sub* and *subu* is that *subu* will not check whether overflow occurs. But in this program, they both output the overflow flag.



```
instruction:op:func:aluctl:    gr1   :    result :    reg_A :    reg_B :   zero :   overflow :    neg
  xxxxxxxx: xx :xx :   xxxx: xxxxxxxx:   xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :          x :      x
  0001f022: 00 :22 :   0101: b2d05e05:   65a0bc0a: b2d05e05: 4d2fa1fb :     0 :          1 :      1
  0001f023: 00 :23 :   0101: b2d05e05:   65a0bc0a: b2d05e05: 4d2fa1fb :     0 :          1 :      1
```

(5) *mul, mulu:*

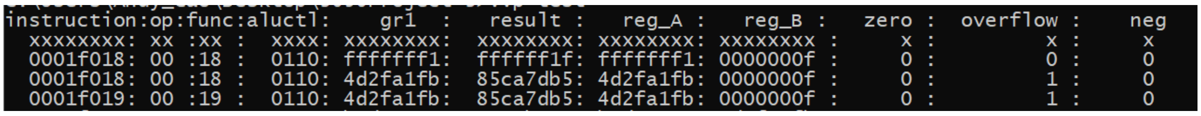
It is more difficult to check whether the overflow occurs in *mul* because the result of multiplication may range from 32bits to 64bits. We need one more register *temp* to save the result. This project will check whether the result divided by one input can get the other input just like:

$$temp = \frac{result}{reg\_A};$$

$$reg\_B == (temp);$$

If they do, the *overflow* will output 0.

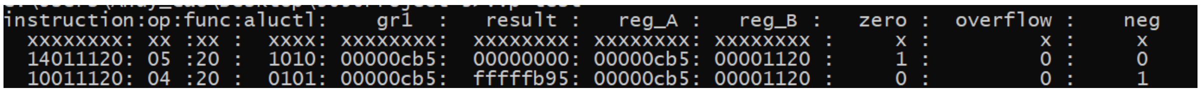
The first case is -15*15, the second and third cases are a very large number multiply 15. Thus, both of them will set the overflow to 1.



```
instruction:op:func:aluctl:    gr1   :   result :   reg_A :   reg_B :   zero :  overflow :    neg
  xxxxxxxx: xx :xx :  xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :         x :     x
  0001f018: 00 :18 :  0110: fffffff1:  ffffff1f: fffffff1: 0000000f :     0 :         0 :     0
  0001f018: 00 :18 :  0110: 4d2fa1fb:  85ca7db5: 4d2fa1fb: 0000000f :     0 :         1 :     0
  0001f019: 00 :19 :  0110: 4d2fa1fb:  85ca7db5: 4d2fa1fb: 0000000f :     0 :         1 :     0
```

(6) *beq, bne:*

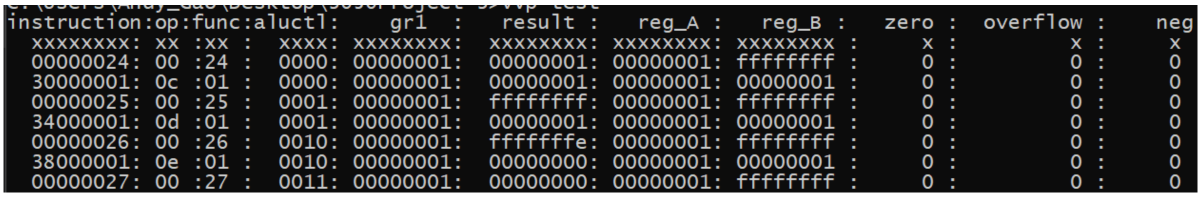
These two cases have the same inputs: 3253 and 12942. The first case is *bne,* and the second case is *beq*. Thus, zero is set to 1 in the first case but 0 in the second case.



```
instruction:op:func:aluctl:    gr1   :   result :   reg_A :   reg_B :   zero :  overflow :    neg
  xxxxxxxx: xx :xx :  xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :         x :     x
  14011120: 05 :20 :  1010: 00000cb5:  00000000: 00000cb5: 00001120 :     1 :         0 :     0
  10011120: 04 :20 :  0101: 00000cb5:  fffffb95: 00000cb5: 00001120 :     0 :         0 :     1
```

(7) *and, andi, or, ori, xor, xori, nor:*

These functions are all tested by -1 and 1. It is easy to see the difference between those functions. These examples following the sequence: *and, andi, or, ori, xor, xori, nor*.

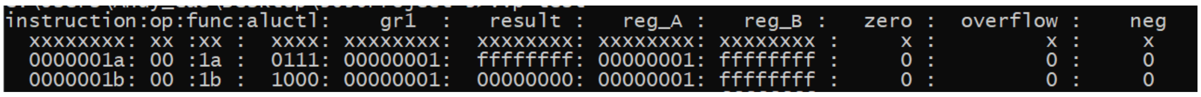


```
instruction:op:func:aluctl:    gr1   :   result :   reg_A :   reg_B :   zero :  overflow :    neg
  xxxxxxxx: xx :xx :  xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :         x :     x
  00000024: 00 :24 :  0000: 00000001:  00000001: 00000001: ffffffff :     0 :         0 :     0
  30000001: 0c :01 :  0000: 00000001:  00000001: 00000001: 00000001 :     0 :         0 :     0
  00000025: 00 :25 :  0001: 00000001:  ffffffff: 00000001: ffffffff :     0 :         0 :     0
  34000001: 0d :01 :  0001: 00000001:  00000001: 00000001: 00000001 :     0 :         0 :     0
  00000026: 00 :26 :  0010: 00000001:  fffffffe: 00000001: ffffffff :     0 :         0 :     0
  38000001: 0e :01 :  0010: 00000001:  00000000: 00000001: 00000001 :     0 :         0 :     0
  00000027: 00 :27 :  0011: 00000001:  00000000: 00000001: ffffffff :     0 :         0 :     0
```

(8) *div, divu:*

The difference between these two functions is that *divu* consider both of these two inputs as unsigned number. As a result, I set two more registers *u_reg_A, u_reg_B* to convert the signed number into unsigned number.

The inputs of these two examples are -1 and 1. *divu* will not output the correct answer because it consider the -1 as a very big positive number. Therefore, it outputs 0, instead of -1.

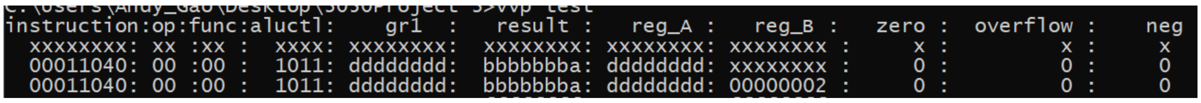


```
instruction:op:func:aluctl:    gr1   :   result :    reg_A :    reg_B :   zero :  overflow :    neg
  xxxxxxxx: xx :xx :   xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :         x :      x
  0000001a: 00 :1a :   0111: 00000001:  ffffffff: 00000001: ffffffff :     0 :         0 :      0
  0000001b: 00 :1b :   1000: 00000001:  00000000: 00000001: ffffffff :     0 :         0 :      0
```

(9) *sll, sllv:*

These two functions are almost the same except the *sll* read the second input from *i_datain[10:6].*

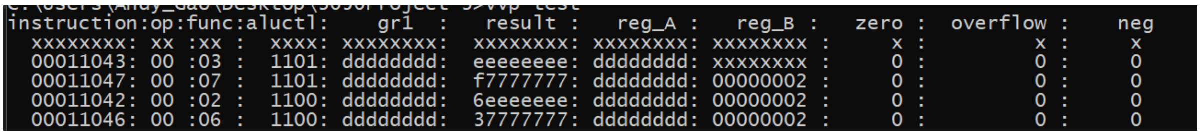
One thing to mention is that the shift logic instructions input are considered as unsigned number. Thus *reg_B* do not change the value when I display it. The second value is sav ed in *u_reg_B*.



```
instruction:op:func:aluctl:    gr1   :   result :    reg_A :    reg_B :    zero :  overflow :     neg
  xxxxxxxx: xx :xx :   xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :      x :         x :       x
  00011040: 00 :00 :   1011: dddddddd:  bbbbbbba: dddddddd: xxxxxxxx :      0 :         0 :       0
  00011040: 00 :00 :   1011: dddddddd:  bbbbbbba: dddddddd: 00000002 :      0 :         0 :       0
```

(10) sra, srav, srl, srlv:

The difference between shift right logic and shift right arithmetic is that the shift right logic instruction will not maintain the sign of the first input. A negative number may finally get a positive result.



```
instruction:op:func:aluctl:    gr1   :   result :    reg_A :    reg_B :   zero :  overflow :     neg
  xxxxxxxx: xx :xx :   xxxx: xxxxxxxx:  xxxxxxxx: xxxxxxxx: xxxxxxxx :     x :         x :       x
  00011043: 00 :03 :   1101: dddddddd:  eeeeeeee: dddddddd: xxxxxxxx :     0 :         0 :       0
  00011047: 00 :07 :   1101: dddddddd:  f7777777: dddddddd: 00000002 :     0 :         0 :       0
  00011042: 00 :02 :   1100: dddddddd:  6eeeeeee: dddddddd: 00000002 :     0 :         0 :       0
  00011046: 00 :06 :   1100: dddddddd:  37777777: dddddddd: 00000002 :     0 :         0 :       0
```

All of the input (*gr1*) of those examples are negative. Only the first two cases get the negative result (*sra* and *srav*).

4. Improvement

To build a complete Datapath, there is still a lot to add.

1. Control Unit Block: To make the ALU control only 4-bits, we combine lots of functions together like (*add* and *addu*). As a result, we have to design some more control wires and multiplexers to control the ALU to give us correct result.

2. The Register File Block and the Memory Block are not included in this project.

3. Program Counter and the Branch Control Block are not included in this project.
4. Jump instructions is not supported by this project.