THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC3050

Computer Architecture

Report for Project 3

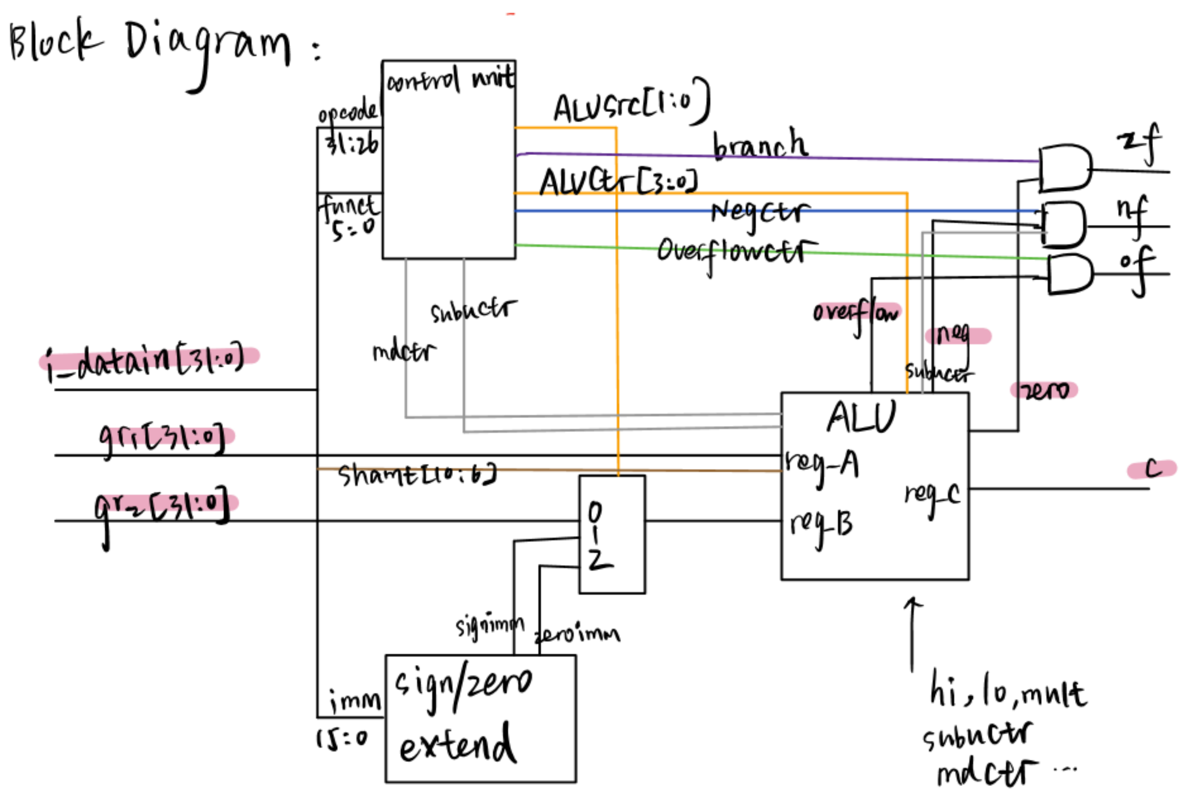
Author:
Li Jingyu 李璟瑜

Student Number:
118010141

April 13, 2020

Contents

1 Block Diagram



Data flow:

Start: Receive 32-bit instruction code--decode it into opcode and funct—decide the value of control line

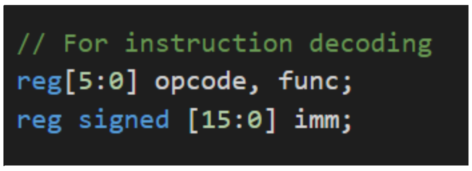
Extension: instruction—decode it into 16 bit imm, extend the imm and choose the value passed into ALU according to ALUSrc (maybe gr2 or imm).

ALU: Receive control line, perform the operation, and output the result and flags.

Flags: Output flags according to c only, and combine with the control line to get the final result (real value used in the following part)

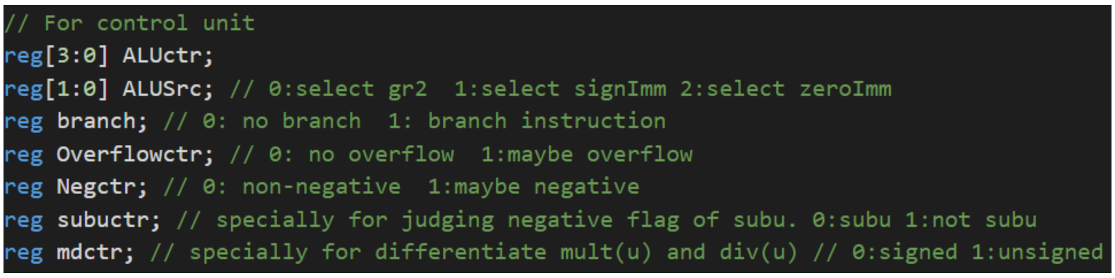
(There are also detailed comments in the code)

Registers used:



```
// For instruction decoding
reg[5:0] opcode, func;
reg signed [15:0] imm;
```

Decode the 32-bit instructions and decide the register values in control unit.

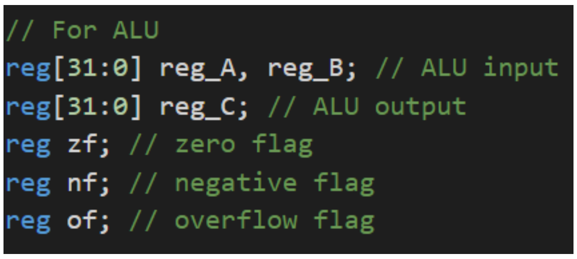


```
// For control unit
reg[3:0] ALUctr;
reg[1:0] ALUSrc; // 0:select gr2  1:select signImm 2:select zeroImm
reg branch; // 0: no branch  1: branch instruction
reg Overflowctr; // 0: no overflow  1:maybe overflow
reg Negctr; // 0: non-negative  1:maybe negative
reg subuctr; // specially for judging negative flag of subu. 0:subu 1:not subu
reg mdctr; // specially for differentiate mult(u) and div(u) // 0:signed 1:unsigned
```

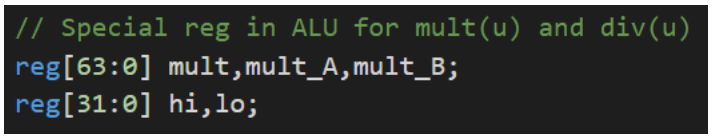
ALUCtr (Decide the function of ALU)

Code	0000	0001	0010	0011	0100	0101	0110	0111
Function	And	Or	Add	Div	Mult	Xor	Subtract	SLT
Code	1000	1001	1010	1011	1100	1101	1110	1111
Function	SLL	SRL	SRA	SRAV	Nor	SLLV	SRLV	SLTU

Other Control unit has been explained in the code.

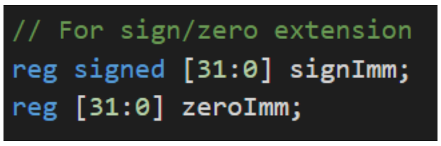


```
// For ALU
reg[31:0] reg_A, reg_B; // ALU input
reg[31:0] reg_C; // ALU output
reg zf; // zero flag
reg nf; // negative flag
reg of; // overflow flag
```



```
// Special reg in ALU for mult(u) and div(u)
reg[63:0] mult,mult_A,mult_B;
reg[31:0] hi,lo;
```

Extension part:



```
// For sign/zero extension
reg signed [31:0] signImm;
reg [31:0] zeroImm;
```

SignImm: Extend the sign bit. ZeroImm: extend 0;

Extend the 16 bits immediate (in the code)-> 32 bits.

About instruction: see part 2; About flags: see part 3;

2 Explanation of the instructions

Items displayed:

Instruction: 32-bit, expressed as 8-bit hexadecimal.

Op: 6-bit, opcode, expressed as 2-bit hexadecimal.

Func: 6-bit, expressed as 2-bit hexadecimal.

ALUCtr: 4-bit binary. ALUSrc: 1-bit binary.

gr1, gr2: values of register (input) c: value of the result

zero, overflow, neg, zf, of, nf: relative flags (Can refer to the next part)

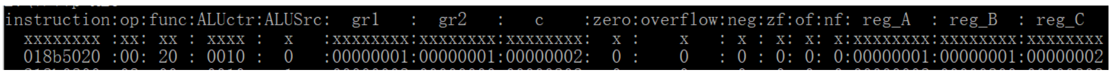
Here zero, overflow, neg only depends only on the value of c, and zf, of, nf should be the final result of the flag (AND with control unit).

Reg_A/B/C: result of the registers used in ALU

Add: Add two signed numbers in two registers.

Combine 32 1-bit adders into a full adder. Carry-in and carry-out pass.

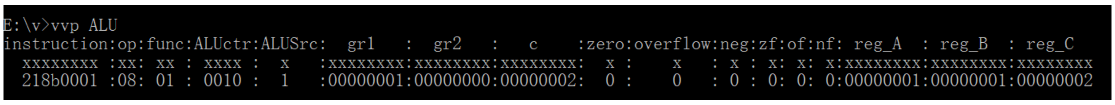
No overflow: 1+1=2


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx:  xx  :  xxxx  :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5020 :00:  20  :  0010  :   0    :00000001:00000001:00000002:   0  :     0    : 0 : 0: 0: 0:00000001:00000001:00000002
```

Overflow: See Flag below.

Addi: Add two signed numbers in register and instruction.

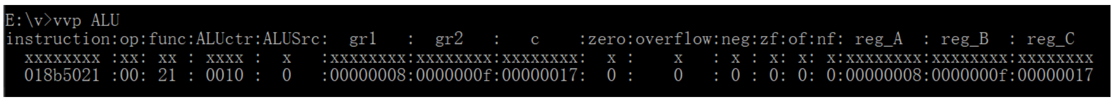
No overflow: 1+1=2:


```
E:\v>vvp ALU
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx:  xx  :  xxxx  :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  218b0001 :08:  01  :  0010  :   1    :00000001:00000000:00000002:   0  :     0    : 0 : 0: 0: 0:00000001:00000001:00000002
```

Overflow: See Flag below.

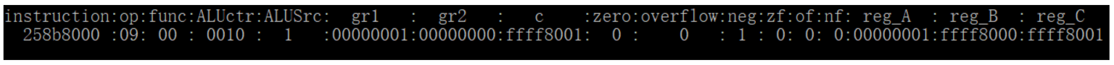
Addu: Add two unsigned numbers in two registers

No overflow: 8+f (15)=17 (23)


```
E:\v>vvp ALU
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx:  xx  :  xxxx  :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:   x  :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5021 :00:  21  :  0010  :   0    :00000008:0000000f:00000017:   0  :     0    : 0 : 0: 0: 0:00000008:0000000f:00000017
```

Addiu: Add two unsigned numbers in register and instruction.

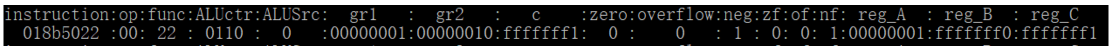
No overflow: 1+ffff8000 =ffff8001


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  258b8000 :09:  00  :  0010  :   1    :00000001:00000000:ffff8001:   0  :     0    : 1 : 0: 0: 0:00000001:ffff8000:ffff8001
```

Sub: Subtract two signed numbers in two registers

Reg_B (gr2) is changed to 2’s complement form (negative value) and add the reg_A (gr1). A-B = A+B’+1;   B’:Bnegate, bit-wise inverse.

No overflow: 1-10(16) = fffffff1 (-15) or 00…01+ff…f0 = fffffff1

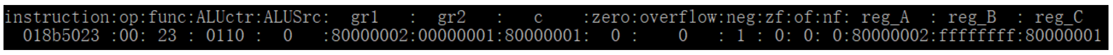


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5022 :00: 22 : 0110 :   0    :00000001:00000010:fffffff1:  0 :     0    : 1 : 0: 0: 1:00000001:fffffff0:fffffff1
```

Overflow: See Flag below.

Subu: Subtract two unsigned numbers in two registers

80…02-00..01 = 80…01 or 80…02 + fff..ff (-1)= 80..01

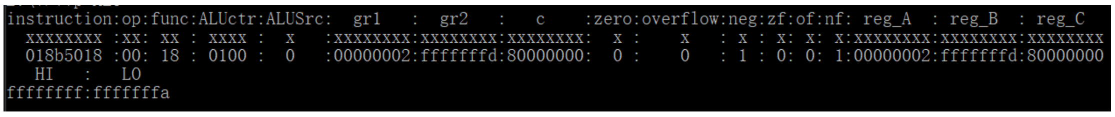


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5023 :00: 23 : 0110 :   0    :80000002:00000001:80000001:  0 :     0    : 1 : 0: 0: 0:80000002:ffffffff:80000001
```

Mult: Multiply two signed numbers in two registers. Store the higher 32-bit into register hi, and lower 32-bit into register lo.

Step: Check the last bit of reg_B, if it is 0, then shift it right (add 0000 and shift); if it is zero, then add the reg_A to left half of the product (right half is reg_B at first)(add reg_A and shift). If it has repeated 32 times, then end the program. Here reg_A and reg_B will change into 64 bits at first, and mult will store the result and split into hi and lo.

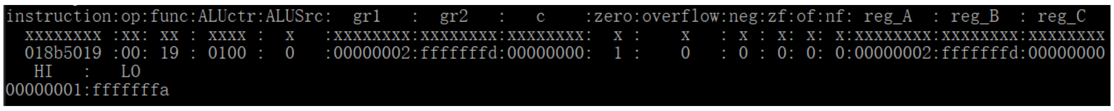
2 × (-2) = -4



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx: xx : xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:  x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5018 :00: 18 : 0100 :   0    :00000002:fffffffd:80000000:  0 :     0    : 1 : 0: 0: 1:00000002:fffffffd:80000000
   HI    :    LO
ffffffff:fffffffa
```

Multu: Multiply two unsigned numbers in two registers. Store the higher 32-bit into register hi, and lower 32-bit into register lo.

2×fffffffd = 1fffffffa (large number)   .. same input as the mult



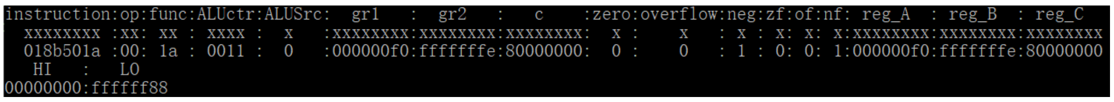
```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx: xx : xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:   x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5019 :00: 19 : 0100 :   0    :00000002:fffffffd:00000000:   1 :     0    : 0 : 0: 0: 0:00000002:fffffffd:00000000
   HI    :   LO
00000001:fffffffa
```

Div: Divide two signed numbers in two registers. Store the quotient into register lo, and remainder into register hi.

Step: Put reg_A into Remainder and shift left 1 bit. Subtract reg_A from left half of Remainder. If the result is smaller than 0 (negative), then restore the value and shift reg_A left 1 bit. Set the new bit 0; If the result is larger than or equal to 0, shift reg_A left 1 bit, and set the new bit 1. If it has repeated 32 times, then end the program. Shift the left half of remainder right 1 bit.

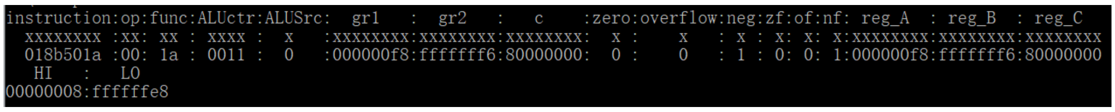
Left part: remainder, into hi. Right part: quotient, into lo.

000000f0 / -2 = ffffff88 … 0



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx: xx : xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:   x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b501a :00: 1a : 0011 :   0    :000000f0:fffffffe:80000000:   0 :     0    : 1 : 0: 0: 1:000000f0:fffffffe:80000000
   HI    :   LO
00000000:ffffff88
```

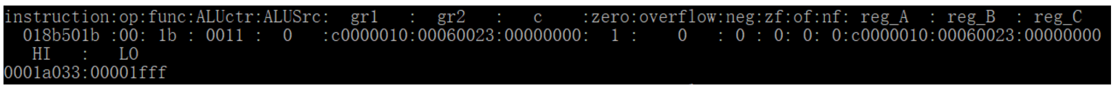
248 / -10 = -24 … 8



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx: xx : xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:   x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b501a :00: 1a : 0011 :   0    :000000f8:fffffff6:80000000:   0 :     0    : 1 : 0: 0: 1:000000f8:fffffff6:80000000
   HI    :   LO
00000008:ffffffe8
```

Divu: Divide two unsigned numbers in two registers. Store the quotient into register lo, and remainder into register hi.

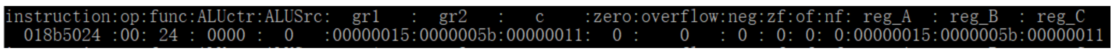
C0000010 / 00060023 = 00001fff … 0001a033 (positive)



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b501b :00: 1b : 0011 :   0    :c0000010:00060023:00000000:  1 :     0    : 0 : 0: 0: 0:c0000010:00060023:00000000
   HI    :    LO
0001a033:00001fff
```

And: logic operation (two register values)

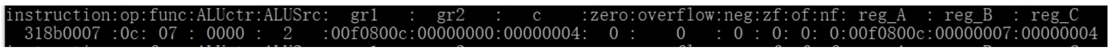
00010101 & 01011011 = 00010001



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5024 :00: 24 : 0000 :   0    :00000015:0000005b:00000011:  0 :     0    : 0 : 0: 0: 0:00000015:0000005b:00000011
```

Andi: logic operation (one register value and one imm)

f & 0 = 0   1100 & 0111 = 0100

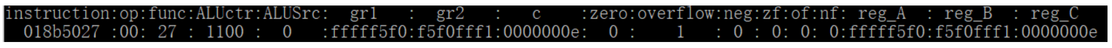


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  318b0007 :0c: 07 : 0000 :   2    :00f0800c:00000000:00000004:  0 :     0    : 0 : 0: 0: 0:00f0800c:00000007:00000004
```

Nor: logic operation (two register values)

A nor B = (not A) and (not B) = not (A or B)

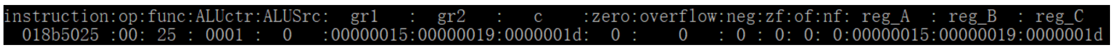
1111 nor 1111 = 0000   0000 nor 0001 = 1110



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5027 :00: 27 : 1100 :   0    :fffff5f0:f5f0fff1:0000000e:  0 :     1    : 0 : 0: 0: 0:fffff5f0:f5f0fff1:0000000e
```

Or: logic operation (two register values)

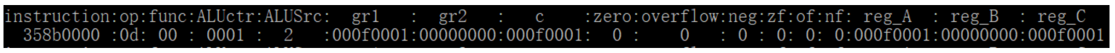
0001 | 0001 = 0001   0101 | 1001 = 1101



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5025 :00: 25 : 0001 :   0    :00000015:00000019:0000001d:  0 :     0    : 0 : 0: 0: 0:00000015:00000019:0000001d
```

Ori: logic operation (one register value and one imm)

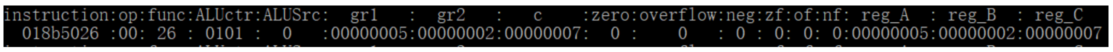
X | 0 = X



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  358b0000 :0d: 00 : 0001 :   2    :000f0001:00000000:000f0001:  0 :     0    : 0 : 0: 0: 0:000f0001:00000000:000f0001
```

Xor: logic operation (two register values)

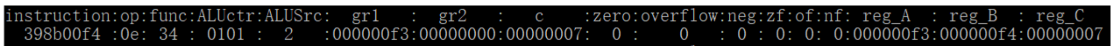
0101 xor 0010 = 0111



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5026 :00: 26 : 0101 :   0    :00000005:00000002:00000007:   0 :     0    : 0 : 0: 0: 0:00000005:00000002:00000007
```

Xori: logic operation (one register value and one imm)

1111 xor 1111 = 0000   0011 xor 0100 = 0111

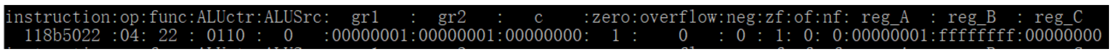


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  398b00f4 :0e: 34 : 0101 :   2    :000000f3:00000000:00000007:   0 :     0    : 0 : 0: 0: 0:000000f3:000000f4:00000007
```

Beq: If rs==rt, then branch the PC. (zero=1).

It will subtract the reg_A and reg_B and judge whether it is equal to 0.

1-1=0   zero = 1   zf = 1



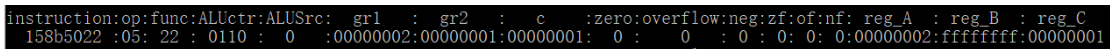
```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  118b5022 :04: 22 : 0110 :   0    :00000001:00000001:00000000:   1 :     0    : 0 : 1: 0: 0:00000001:ffffffff:00000000
```

Bne: If rs!=rt, then branch the PC. (zero=1)

It will subtract the reg_A and reg_B and judge whether it is equal to 0

(same as beq)

2-1!=0   zero=0   zf=0



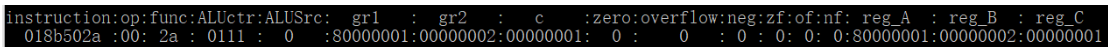
```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  158b5022 :05: 22 : 0110 :   0    :00000002:00000001:00000001:   0 :     0    : 0 : 0: 0: 0:00000002:ffffffff:00000001
```

Slt: If rs<rt, set the value of rd to be 1.

A less wire will connect into the ALU. If rs<rt in certain bit, it will be 1. Bit 0

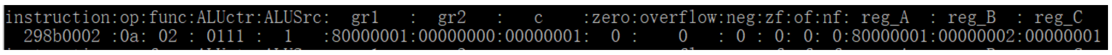
connects the set wire.

Reg_A (signed) < 0 < 2 (reg_B)   so reg_C = 1


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b502a :00: 2a : 0111 :   0    :80000001:00000002:00000001:   0 :     0    : 0 : 0: 0: 0:80000001:00000002:00000001
```

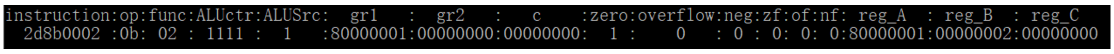
Slti: If rs<imm, set the value of rd to be 1.

Same sample as slt. Imm = gr2. Reg_C = 1


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  298b0002 :0a: 02 : 0111 :   1    :80000001:00000000:00000001:   0 :     0    : 0 : 0: 0: 0:80000001:00000002:00000001
```

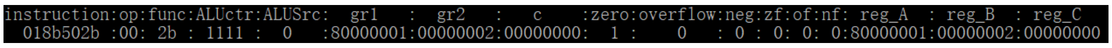
Sltiu: If unsigned rs< imm, set the value of rd to be 1

Reg_A (unsigned) > 2 (imm) so reg_C = 0;


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  2d8b0002 :0b: 02 : 1111 :   1    :80000001:00000000:00000000:   1 :     0    : 0 : 0: 0: 0:80000001:00000002:00000000
```

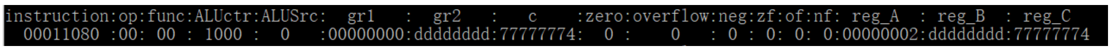
Sltu: If unsigned rs< unsigned rt, set the value of rd to be 1.

Reg_A (unsigned) > 2 (reg_B)   so reg_C = 0 (same as sltiu)


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b502b :00: 2b : 1111 :   0    :80000001:00000002:00000000:   1 :     0    : 0 : 0: 0: 0:80000001:00000002:00000000
```

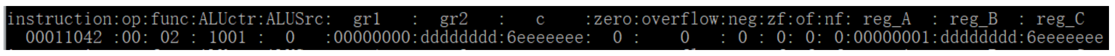
Sll: rt Shift left shamt bit (add 0).

1101…1101 << 2 = 0111…0111…0100


```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  00011080 :00: 00 : 1000 :   0    :00000000:dddddddd:77777774:   0 :     0    : 0 : 0: 0: 0:00000002:dddddddd:77777774
```

Srl: rt Shift right shamt bit (Value in rt is unsigned so it will add 0).

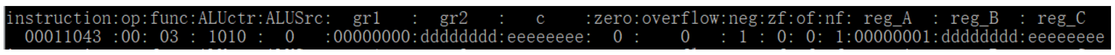
1101…1101 >> 1 = 0110 1110 … 1110



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  00011042 :00: 02 : 1001 :  0   :00000000:dddddddd:6eeeeeee:  0 :    0    : 0 : 0: 0: 0:00000001:dddddddd:6eeeeeee
```

Sra: rt Shift right shamt bit (Value in rt is signed so it will add the sign bit).

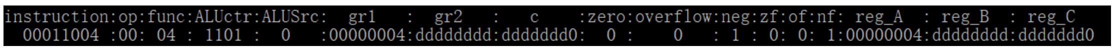
1101…1101 >> 1 = 1110 1110 … 1110



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  00011043 :00: 03 : 1010 :  0   :00000000:dddddddd:eeeeeeee:  0 :    0    : 1 : 0: 0: 1:00000001:dddddddd:eeeeeeee
```

Sllv: rt Shift left rs (value stored in rs) bit (add 0).

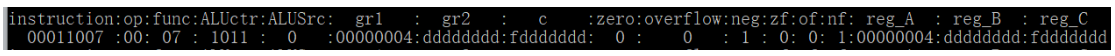
1101…1101 << 4 = 1101…1101…0000



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  00011004 :00: 04 : 1101 :  0   :00000004:dddddddd:ddddddd0:  0 :    0    : 1 : 0: 0: 1:00000004:dddddddd:ddddddd0
```

Srav: rt Shift right rs (value stored in rs) bit (Value in rt is signed so it will add the sign bit)

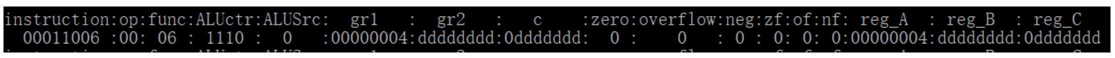
1101…1101 >> 4 = 1111…1101…1101



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  00011007 :00: 07 : 1011 :  0   :00000004:dddddddd:fddddddd:  0 :    0    : 1 : 0: 0: 1:00000004:dddddddd:fddddddd
```

Srlv: rt Shift right rs (value stored in rs) bit (Value in rt is unsigned so it will add 0).

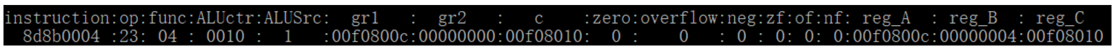
1101…1101 >> 4 = 0000…1101…1101



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  00011006 :00: 06 : 1110 :  0   :00000004:dddddddd:0ddddddd:  0 :    0    : 0 : 0: 0: 0:00000004:dddddddd:0ddddddd
```

Lw: Get the address in the register rs, plus the offset (imm), and output the result. (Use adder)

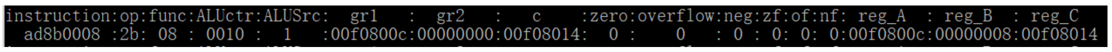
00f0800c + 4 = 00f08010



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  8d8b0004 :23: 04 : 0010 :   1    :00f0800c:00000000:00f08010:  0 :     0    : 0 : 0: 0: 0:00f0800c:00000004:00f08010
```

Sw: Get the address in the register rs, plus the offset (imm), and output the result. (Use adder)

00f0800c + 8 = 00f08014



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  ad8b0008 :2b: 08 : 0010 :   1    :00f0800c:00000000:00f08014:  0 :     0    : 0 : 0: 0: 0:00f0800c:00000008:00f08014
```

3 Explanation of the flags

In this project, we need to decide some flags which reflect the attribute of the result (zero: whether it is zero and can trigger PC branch; overflow: whether there is an overflow in some arithmetic operation; negative: whether the arithmetic result is negative).

As for the flags part, I set 3 registers and 3 wires to represent them. The wires (zero, overflow, neg) are connected with the ALU directly, whose values **only depend on the output c**. The registers (zf, nf, of) are the filtered results, which equal to the wire & control. It means that register results are the meaningful one (can be further used in the circuit) (Maybe you can only check this part and zero).

Zero: If c equals to 0, then it is 1. Otherwise it is 0.

Overflow: If reg_A = reg_B and they are different from reg_C[31], then it is 1. Otherwise it is 0.

Neg: If reg_C[31]=1, then it is 1. Otherwise it is 0

(Overflow and Neg should not be treated as the final result, they should consider the instructions.)

Of: If the instruction is **add/addi/sub**, then the value can be 1 (overflow & overflowctr), or it will be 0.

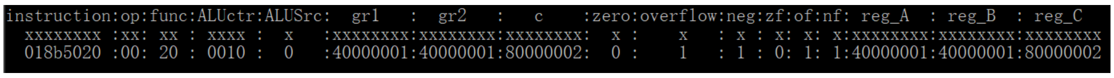
Zf: If the instruction is beq/bne, then the value can be 1 (zero & branch), or it will be 0;

Nf: If the instruction deals with signed values (or the only unsigned one: subu) and the result is negative, then it will be 1, or it will be 0.

Some special examples: (Normal example: refer to last part)

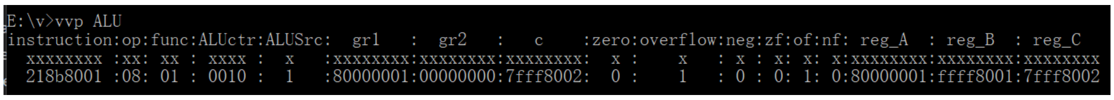
Overflow flag:

Add: Overflow happens when sum of two positive integers equals to a negative number:0100_0…0_0001 + 0100_0…0_0001 = 1000_0…0_0010   of=1



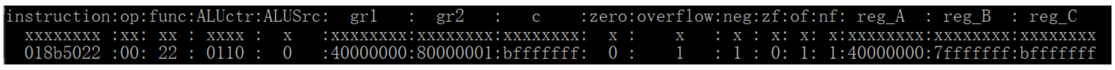
```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A    : reg_B    : reg_C
  xxxxxxxx :xx:  xx :  xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:  x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5020 :00:  20 :  0010 :   0    :40000001:40000001:80000002:  0 :     1    : 1 : 0: 1: 1:40000001:40000001:80000002
```

Addi: Overflow happens when sum of two negative integers equals to a positive number: 1000_...._0001+1111_..._0001 = 0111_..._0010   of=1



```
E:\v>vvp ALU
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A    : reg_B    : reg_C
  xxxxxxxx :xx:  xx :  xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:  x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  218b8001 :08:  01 :  0010 :   1    :80000001:00000000:7fff8002:  0 :     1    : 0 : 0: 1: 0:80000001:ffff8001:7fff8002
```

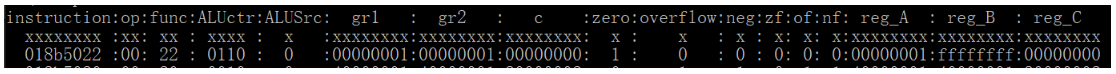
Sub: 0100…0000 -1000…0001(+ 0111…1111) = 1011…1111 of=1



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx:  xx :  xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:  x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5022 :00:  22 :  0110 :   0    :40000000:80000001:bfffffff:  0 :     1    : 1 : 0: 1: 1:40000000:7fffffff:bfffffff
```

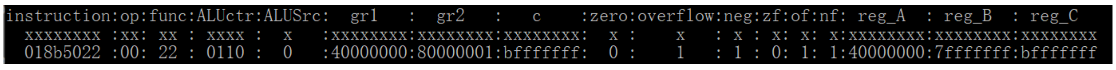
Zero flag: (sub):1-1=0. Here zero will give 1 since c==0, but it is not branch

instruction, the final result of zf is 0 (It will not trigger the branch of PC).



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx:  xx :  xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:  x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5022 :00:  22 :  0110 :   0    :00000001:00000001:00000000:  1 :     0    : 0 : 0: 0: 0:00000001:ffffffff:00000000
```

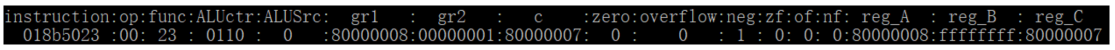
Negative flag: (sub):0100…0000-1000….0001<0



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  xxxxxxxx :xx:  xx :  xxxx :   x    :xxxxxxxx:xxxxxxxx:xxxxxxxx:  x :     x    : x : x: x: x:xxxxxxxx:xxxxxxxx:xxxxxxxx
  018b5022 :00:  22 :  0110 :   0    :40000000:80000001:bfffffff:  0 :     1    : 1 : 0: 1: 1:40000000:7fffffff:bfffffff
```

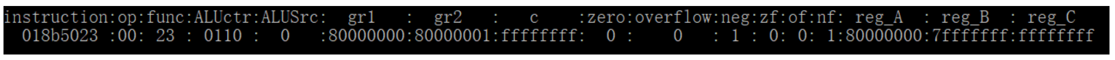
(subu) 80000008 – 00000001 = 80000007 (positive)

neg=1 (judged by c) but final result nf=0 (unsigned for subu)



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5023 :00:  23 :  0110 :   0    :80000008:00000001:80000007:  0 :     0    : 1 : 0: 0: 0:80000008:ffffffff:80000007
```

(subu) 80000000 – 80000001 = -1 (negative) neg=1 and final result nf=1



```
instruction:op:func:ALUctr:ALUSrc:   gr1    :   gr2    :    c     :zero:overflow:neg:zf:of:nf: reg_A   : reg_B   : reg_C
  018b5023 :00:  23 :  0110 :   0    :80000000:80000001:ffffffff:  0 :     0    : 1 : 0: 0: 1:80000000:7fffffff:ffffffff
```

This difference is caused by subuctr.

4 Feeling

New to Verilog, need to get familiar with the grammar…… Verilog is different from C/C++ (especially the order of executing the code), and I wrote many bugs at first.

It really cost me a large amount of time to deal with the flags.

It also cost me a great amount of time to run the code for many times, check the result and do the screenshot.

However, I really gained a lot in this project. It enhanced my understanding of ALU, control part of CPU and data flow. Also I learned how to write the hardware language Verilog code.

By the way, TA Micky MA solved many problems in the tutorial and wechat, which helped me a lot 2333. She is as nice as the TA Handsome.

That’s all.