2022 計算機組織
Computer Organization

Lab 2 Report

系級	電機 114
學號	E24106220
姓名	簡誌加

Question List

Q1 : Why ***IALIGN*** of RV32I is 32 bits ?

Because the rules say so.

Why ***IALIGN*** need to support 16 bits ?

Because there's a function call "compression" which make the size of the instruction 16 bits, hence IALIGN has to keep the compatibility of the 16 bits instruction.

Why ***IALIGN*** have no greater than 32 bits (e.g. 64 / 128 bits) ?

Because if the alignment of instructions is separated for every 64 bits, the instruction may not be perfectly divided, there would be too much blank in the memory just to follow the alignment of 64 bits separation.

Q2 : Why temporary registers and saved registers are not numbered sequentially ?

Because embedded devices exist, the arrangement of the location of registers have to be compatible with both I and E devices at the same time, so the design of it may not be perfectly sequential.

Q3 : Why return value needs 2 registers (a0, a1) ?

Because the return value may not always be smaller than 32 bits, for instance, there might be a return value whose data type is double or float, whose sizes are 64 bits.

Q4 : Why there is no lwu in RV32I?

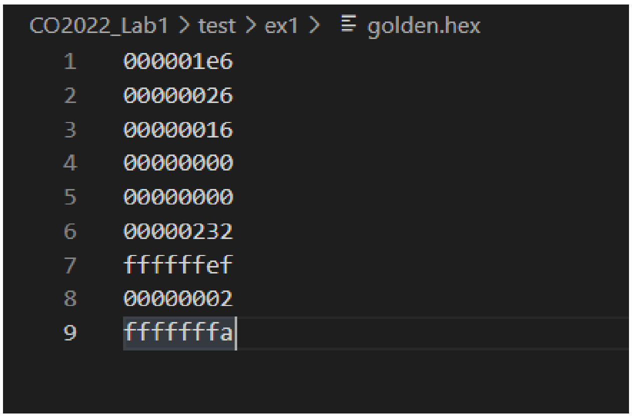
Because a word's size is 32 bits, the original purpose of lbu and lhu is to extend the size of halfword and byte with two different ways depending on the situation at the time. Since there's no need of extension of word, there's no need of unsigned loading.

Q5 : What is the addressing mode of Load & Store ?

The addressing mode of Load is I-type, and the addressing mode of Store is S-type

Exercise 1

1. Please screenshot your golden.hex (Need 9 answers)



CO2022_Lab1 > test > ex1 > golden.hex

```
000001e6
00000026
00000016
00000000
00000000
00000232
ffffffef
00000002
fffffffa
```

2. Please explain how you implement the following C code with RISC-V assembly code we learned this time

(1) Variable * -3:

Var * 3 * (-1)

Var * 4 (slli 2)

(Var * 4) – Var (sub)

Make a (-1) number in a register

xor (Var *3) with (-1), which is ffffffff.

(2) abs(Variable):

I use slti 0 to verify whether the Var is less than 0,

I save the result in a register, and then make some amendment,

Let's say, t0 is the result of (Var < 0), I then modify it as

t1 = t0 – 2t0, let's see what happens depending on what t0 is,

if t0 > 0, t1 = 0; however, if t0 < 0, t1 = -1, which ffffffff
we now xor Var with t1, let's see what happens,
if Var > 0, Var xor t1 = Var xor 0, which won't change Var's sign,
if Var < 0, Var xor t1 = Var xor 1, which change the sign of Var.
and that's what I came up with behind abs(Var).

(3) Variable % 4:
Simple,
I first check if they are smaller than 0,
I get the absolute number of them,
I % them by ANDing them with 00000003, which is 0….011
I use the result of slti Var 0 and Var andi 000003 and get the final result of Var % 4 whether Var > 0 of not.

(4) (int) (Variable / 8):
Just srai it with 3, those floating number will be eaten and keep those integers.

(5) (int) (100 * 5.625):
5.625 = 45/8
I tried two ways,
45 * 100 / 8, which is easier I think,
li x, 45, 100x = {[(4x + x) * 4] + 5x} * 4, and then srai 3__#
the other way is: (I implemented in this way at last)
100 * 45/8,
li x, 100, (32x + 8x + 5x) / 8 __#

(6) (int) (-5 * 3.5):
I figured out that if srai is used on negative number, it will be 1 less than it should have been, so I added it back after then,
3.5 = 7/2 = (8 – 1)/2, after all these, I think I don't have to explain how I implement those progress already.

(7) (int) (3 * 0.75):

0.75 = 3/4 = (4 – 1)/4.

(8) (int) (Variable * 0.75):

Same as (7).

3. Please explain if just using the assembly code we learned this time is enough to do Variable * Variable, how or why not?

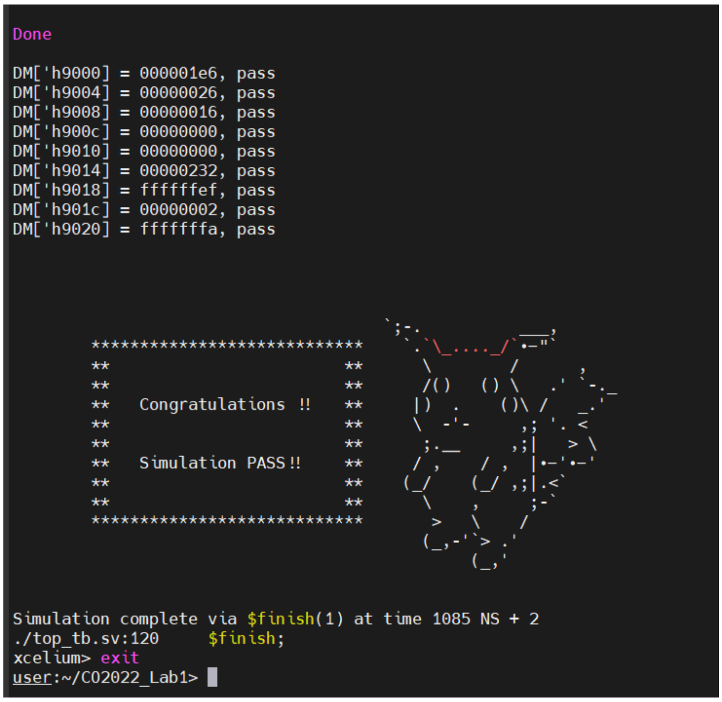
   NO, because basically the ways we use this time are usually first slli it with a certain number that we already know and then add the rest if it.

   However, we can't do this if we are not knowing either of the number.

4. Please compare the differences between main.s & main.dump (e.g. How Pseudo Instructions correspond to actual instructions & other differences you found)

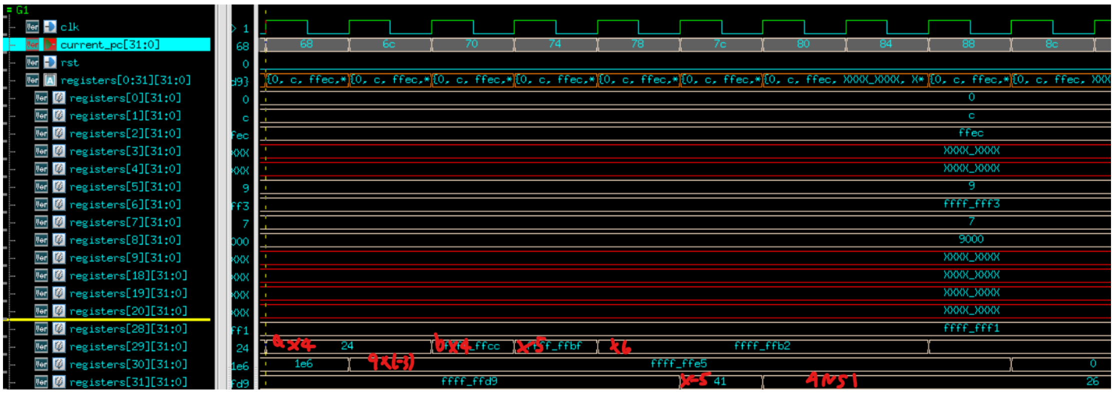
   As what I saw between main.s and main.dump, maybe it's because that the asm instructions I used are all very simple, and I didn't use any pseudo code, so the difference between main.s and main.dump is small. The only difference I found was that the indication of numbers in main.dump are sometimes 0x..... rather than just the number of itself, that's it.

5. Please screenshot the pass information



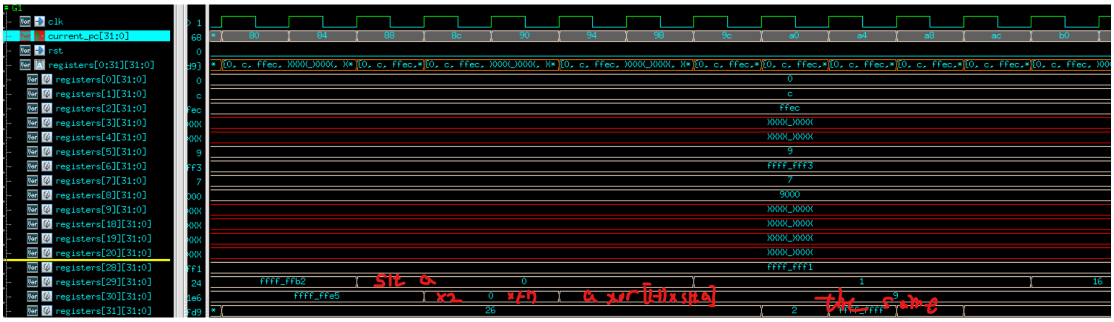
6. Please explain the waveform at the 8 locations listed above with "main.dump"

(1) Variable * -3:



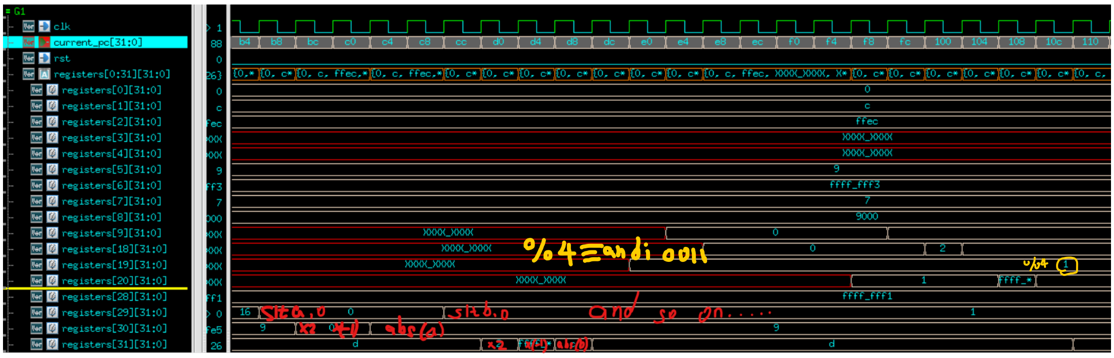

```
64: 00229e93          slli  t4,t0,0x2
68: 41d28f33          sub t5,t0,t4
6c: 00231e93          slli  t4,t1,0x2
70: 006e8eb3          add t4,t4,t1
74: 006e8eb3          add t4,t4,t1
78: 41d30fb3          sub t6,t1,t4
7c: 01ef8fb3          add t6,t6,t5
80: 01f42223          sw  t6,4(s0)
```

(2) abs(Variable):



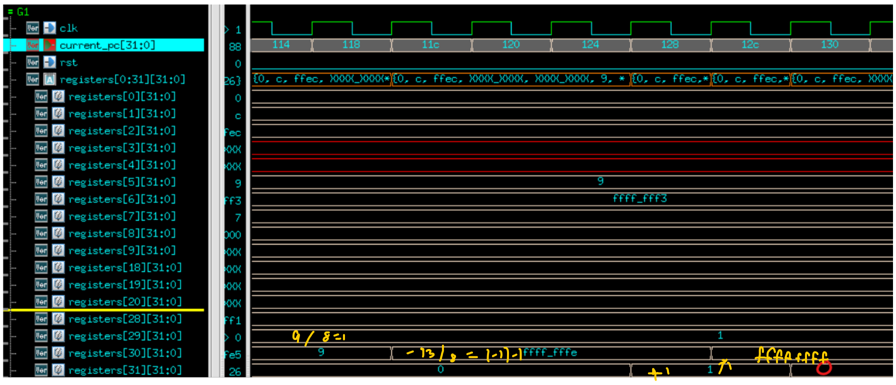

```
84:  0002ae93        slti   t4,t0,0
88:  001e9f13        slli   t5,t4,0x1
8c:  41ee8f33        sub t5,t4,t5
90:  005f4f33        xor t5,t5,t0
94:  01df0f33        add t5,t5,t4
98:  00032e93        slti   t4,t1,0
9c:  001e9f93        slli   t6,t4,0x1
a0:  41fe8fb3        sub t6,t4,t6
a4:  006fcfb3        xor t6,t6,t1
a8:  01df8fb3        add t6,t6,t4
ac:  01ff0eb3        add t4,t5,t6
b0:  01d42423        sw   t4,8(s0)
```

(3) Variable % 4:



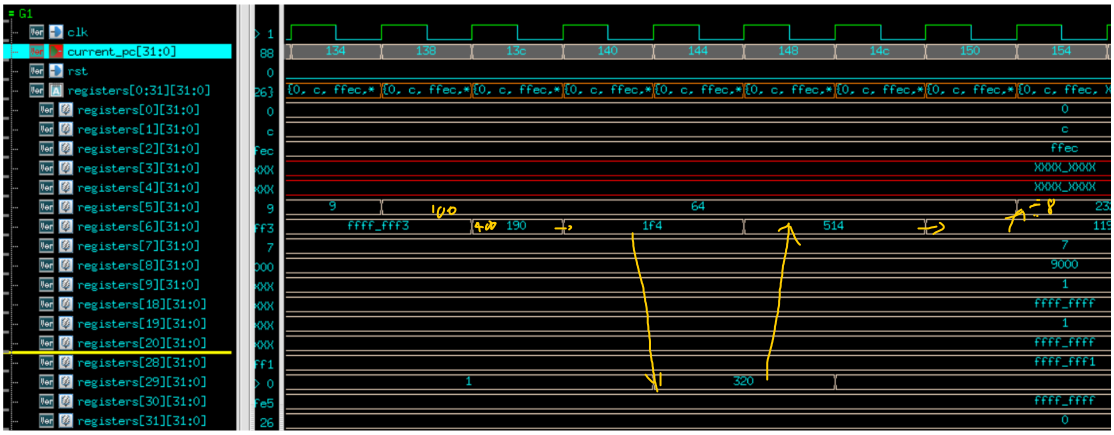
```
 b4: 0002ae93          slti  t4,t0,0
 b8: 001e9f13          slli  t5,t4,0x1
 bc: 41ee8f33          sub t5,t4,t5
 c0: 005f4f33          xor t5,t5,t0
 c4: 01df0f33          add t5,t5,t4
 c8: 00032e93          slti  t4,t1,0
 cc: 001e9f93          slli  t6,t4,0x1
 d0: 41fe8fb3          sub t6,t4,t6
 d4: 006fcfb3          xor t6,t6,t1
 d8: 01df8fb3          add t6,t6,t4
 dc: 003f7993          andi  s3,t5,3
 e0: 0002a493          slti  s1,t0,0
 e4: 00149913          slli  s2,s1,0x1
 e8: 41248933          sub s2,s1,s2
 ec: 0129c9b3          xor s3,s3,s2
 f0: 009989b3          add s3,s3,s1
 f4: 003ffa13          andi  s4,t6,3
 f8: 00032493          slti  s1,t1,0
 fc: 00149913          slli  s2,s1,0x1
100: 41248933          sub s2,s1,s2
104: 012a4a33          xor s4,s4,s2
108: 009a0a33          add s4,s4,s1
10c: 01498fb3          add t6,s3,s4
110: 01f42623          sw  t6,12(s0)
```

(4) (int) (Variable / 8):



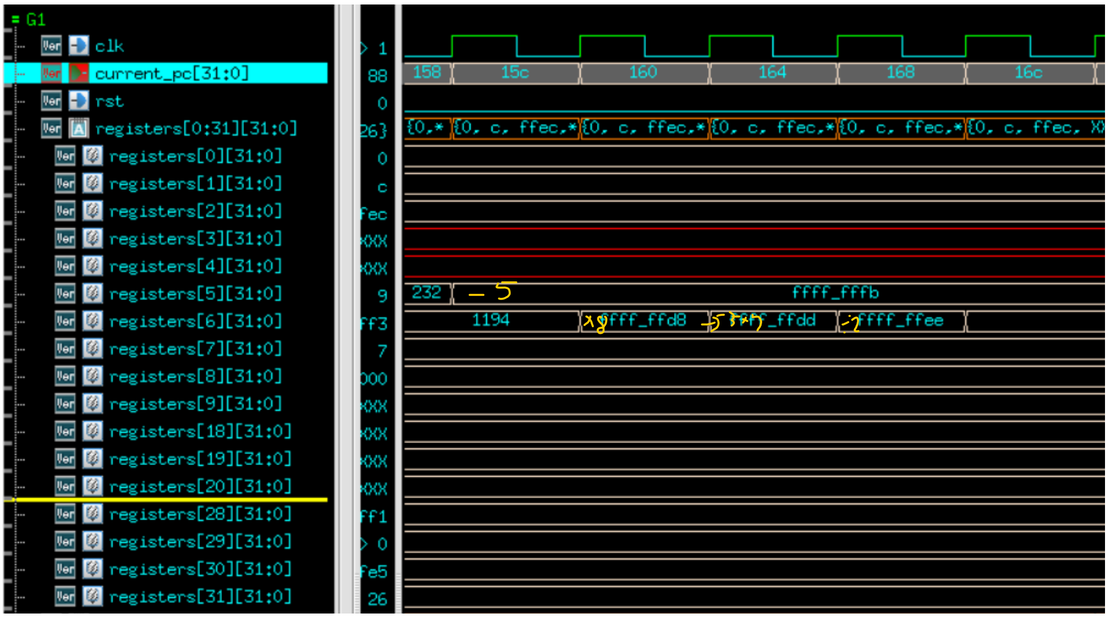
```
114: 4032de93          srai  t4,t0,0x3
118: 40335f13          srai  t5,t1,0x3
11c: 000eaf93          slti  t6,t4,0
120: 01fe8eb3          add t4,t4,t6
124: 000f2f93          slti  t6,t5,0
128: 01ff0f33          add t5,t5,t6
12c: 01ee8fb3          add t6,t4,t5
130: 01f42823          sw  t6,16(s0)
```

(5) (int) (100 * 5.625):



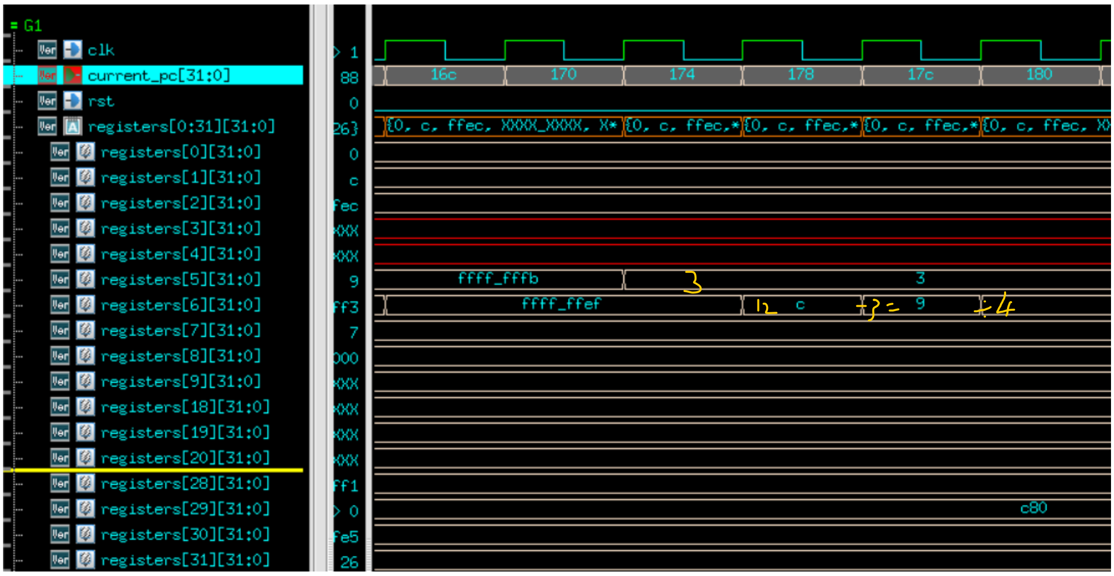

```
134: 06400293        li   t0,100
138: 00229313        slli  t1,t0,0x2
13c: 00530333        add t1,t1,t0
140: 00329e93        slli  t4,t0,0x3
144: 01d30333        add t1,t1,t4
148: 00529e93        slli  t4,t0,0x5
14c: 01d30333        add t1,t1,t4
150: 00335293        srli  t0,t1,0x3
154: 00542a23        sw  t0,20(s0)
```

(6) (int) (-5 * 3.5):



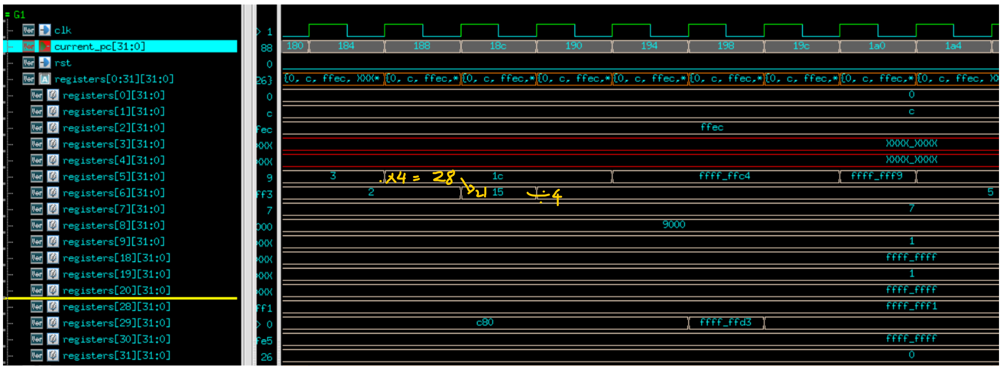

```
158: ffb00293        li   t0,-5
15c: 00329313        slli  t1,t0,0x3
160: 40530333        sub t1,t1,t0
164: 40135313        srai  t1,t1,0x1
168: 00130313        addi  t1,t1,1
16c: 00642c23        sw  t1,24(s0)
```

(7) (int) (3 * 0.75):




```
158: ffb00293        li   t0,-5
15c: 00329313        slli  t1,t0,0x3
160: 40530333        sub t1,t1,t0
164: 40135313        srai  t1,t1,0x1
168: 00130313        addi  t1,t1,1
16c: 00642c23        sw   t1,24(s0)
170: 00300293        li   t0,3
174: 00229313        slli  t1,t0,0x2
178: 40530333        sub t1,t1,t0
17c: 00235313        srli  t1,t1,0x2
180: 00642e23        sw   t1,28(s0)
```

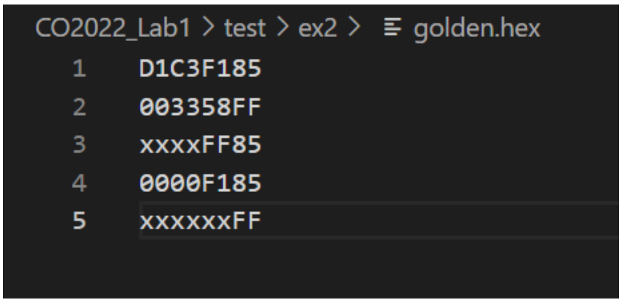
(8) (int) (Variable * 0.75):




```
180: 00642e23        sw  t1,28(s0)
184: 00239293        slli  t0,t2,0x2
188: 40728333        sub t1,t0,t2
18c: 40235313        srai  t1,t1,0x2
190: 002e1293        slli  t0,t3,0x2
194: 41c28eb3        sub t4,t0,t3
198: 402ede93        srai  t4,t4,0x2
19c: 01d302b3        add t0,t1,t4
1a0: 00128293        addi  t0,t0,1
1a4: 02542023        sw  t0,32(s0)
```

Exercise 2

1. Please screenshot your golden.hex (Need 5 answers)



2. Please explain how you put long integers (0xD1C3F185, 0x003358FF) into registers

   Just as usual, use the data type "word", which is the biggest so far.

3. Please explain how you figured out the answers at 0x9008, 0x900c, 0x9010

   I read the asm code, and then I figured out that saving those data with halfword or byte will lose some of the information, hence the lost information will be illustrated with "x".

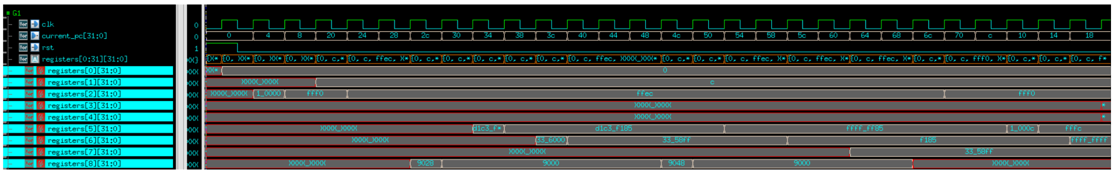
4. Please compare the differences between main.s & main.dump (e.g. How Pseudo Instructions correspond to actual instructions & other differences you found)

   In main.dump, the instruction "li t0, 0xD1C3F185" is divided into two steps, which is to lui 0xD1C3F first and then add it with the remaining 185, and so does the "li t1, 0x003358FF".

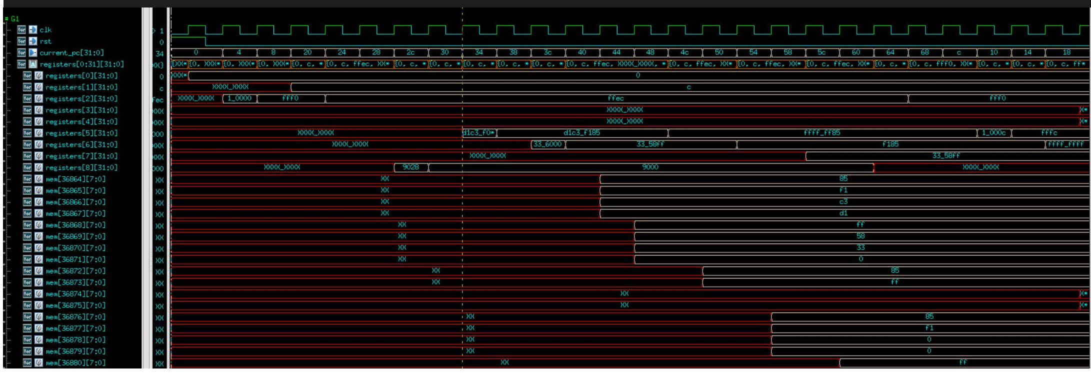
5. Please screenshot the pass information

6. Please use the waveform & "main.dump" to explain and verify your

calculations are correct



```
30: d1c3f2b7              lui t0,0xd1c3f
34: 18528293              addi  t0,t0,389 # d1c3f185 <_sim_end+0xd1c2f189>
38: 00336337              lui t1,0x336
3c: 8ff30313              addi  t1,t1,-1793 # 3358ff <_sim_end+0x325903>
40: 00542023              sw  t0,0(s0)
44: 00642223              sw  t1,4(s0)
48: 00009417              auipc s0,0x9
4c: fb840413              addi  s0,s0,-72 # 9000 <_answer>
50: 00040283              lb  t0,0(s0)
54: 00541423              sh  t0,8(s0)
58: 00045303              lhu t1,0(s0)
5c: 00642623              sw  t1,12(s0)
60: 00442383              lw  t2,4(s0)
64: 00740823              sb  t2,16(s0)
```



In main.dump, the instruction of loading 0xd1c3f185 and 003358ff has been divided into two steps, for instance, li t0, 0xd1c3f185 has been divided into lui t0, 0xd1c3f in 30's and addi t0, t0, 389 in 34's

The left most numbers indicate pc if the wave form, and as long as you can find the location of the instruction matching in the wave form, you can read the wave form easily.