2.8

addi x30, x10, 8 → x30 = &A[1] x30 = address of A[1], but with value = A[0]

addi x31, x10, 0 → x31 = &A[0] x31 = address of A[0], with value = A[0]

sd x31, 0(x30) → A[1] = A[0] ⇒ in C : A[1] = A[0]

ld x30, 0(x30) → x30 = A[0] f = A[1] + A[0]

add x5, x30, x31 → x5 = x30 + x31

2.9.

instruction	type	opcode (op)	source register (rs1)	destination register (rd)	funct 3	other
addi	I	0010011	01010	11110	000	immediate : 000000001000
addi	I	0010011	01010	11111	000	immediate : 000000000000
sd	S	0100011	11110	00000	011	funct 7 : 0000000
ld	I	0000011	11110	11110	011	immediate : 000000000000
add	R	0110011	11110	00101	000	rs2 : 11111 , funct 7 : 0000000

2.16 register file from 32 to 128 ⇒ from 5 bit to 7 bit

Instruction set expands four times ⇒ opcode from 7 bit to 9 bit

2.16.1 R-type rd, rs1, rs2 will increase from 5 bit to 7 bit respectively

opcode will increase from 7 bit to 9 bit

other fields (funct 3, funct 7) share the remaining 2 bits

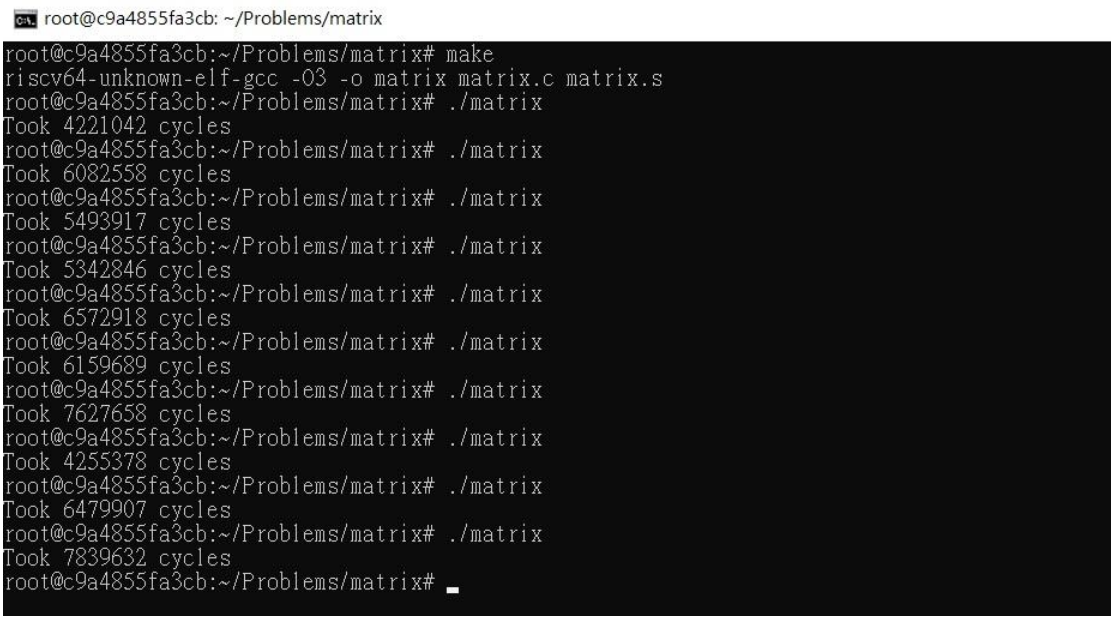
2.16.2 I-type : rd, rs1 will increase from 5 bit to 7 bit respectively

opcode will increase from 7 bit to 9 bit

other fields (funct 3, imm) share the remaining 9 bits

2.16.3 With 128 registers, an assembly program may reduce the frequency of load/store instruction ; with a more complex instruction set, the total number of instruction may decrease because some newly-added instruction may have the same effect as several old instructions do ; thus decreasing the size. But with 128 registers, some instruction such as addi will became less powerful because the range of imm decrease, and it needs to store constant to register and then add, causing more instructions ; increasing instruction set will also decrease the range of imm ; thus increasing a size of an assembly program.

計算機結構 HW2　report　　資工三　陳富春　B07902116



I run my matrix_mul in the given environment for ten times, and the average cycles is about 6007554.5.
I finished my matrix_mul by the method in 3.

1. It takes about 16,000,000 cycles by doing naïve matrix multiplication.
2. The naïve method load $a_{1,1}$, $b_{1,1}$, $a_{1,2}$, $b_{2,1}$, $a_{1,3}$, $b_{3,1}$ ……, $a_{1,128}$, $b_{128,1}$ when calculating $c_{1,1}$, and store $c_{1,1}$. When calculating $c_{1,2}$, $a_{1,1}$ is not in registers, so it must be reload to register. For an element in matrix c, it takes about 2 * 128 load instructions and 1 store instruction, so it needs about 2 * 128 * 128 * 128 load/store instructions to do the matrix multiplication.
3. I keep $a_{1,1}$ to $a_{1,8}$ in registers and let them multiply $b_{1,1}$ to $b_{8,1}$ respectively, and then sum them and add to $c_{1,1}$. The $c_{1,1}$ is not its final value now. Then, using the registers that keep $b_{1,1}$ to $b_{8,1}$ to store another 8 elements in B, that is, $b_{1,2}$ to $b_{8,2}$ to calculate $c_{1,2}$ and so on. When it finishes calculating $c_{1,128}$, $a_{1,1}$ to $a_{1,8}$ are no longer needed and can be replaced by $a_{1,9}$ to $a_{1,16}$ and doing similar thing until the matrix C is completely calculated.
   By doing so, $a_{1,1}$ to $a_{1,8}$ will only need to be loaded into registers one time during the whole matrix multiplication, and so are other elements in matrix A. It can reduce the number of load instruction to about 128^3 times, half to naïve method.
4. The whole computation can be done in three layers loop.
   It may look like:

```
for i = 0, i < SIZE, i++:
    for k = 0, k <SIZE, k += 8:
        for j = 0, j < SIZE, j++:
            C[i][j] += (A[i][k+0] * B[k+0][j] + … + A[i][k+7] * B[k+7][j])
            C[i][j] %= 1024
```