1.

B05202068

2.8 f : x5 , g : x6 , h : x7 , i : x28 , j : x29

&A[0] : x10 , &B[0] : x11

```
addi  x30, x10, 8    // x30 = &A[0]+8 (i.e., &A[1])
addi  x31, x10, 0    // x31 = &A[0]+0 (i.e., &A[0])
sd    x31, 0(x30)    // x30 = x31 (i.e., A[1] = &A[0])
ld    x30, 0(x30)    // x30 = A[1] = &A[0]
add   x5 , x30, x31  // f = &A + &A = 2*(&A).
```

2.9

	opcode, funct3, 7	rs1	rs2	rd	imm
addi x30, x10, 8 : I-type	0x13, 0x0, -	10	-	30	8
addi x31, x10, 0 : I-type	0x13, 0x0, -	10	-	31	0
sd x31, 0(x30) : S-type,	0x23, 0x3, -	31	30	-	0
ld x30, 0(x30) : I-type	0x3, 0x3, -	30	-	30	0
add x5, x30, x31 : R-type	0x33, 0x0, 0x0	30	31	5	-

2.16

RISC-V : 32 regs. → 128 registers

Instructions increase 4 times

① . The rs1, rs2, and rd fields : $\overset{2^5 \text{ regs.}}{\text{5 bits}} \rightarrow \overset{2^7 \text{ regs}}{\text{7 bits.}}$ (R-type)

the opcode field : 7 bits → 9 bits.

② the rs1, rd : 5 bits → 7 bits.

imm won't change by itself, but may be shorten to suit the need of overall instruction size. (I-type.

the opcode : 7 bits → 12 bits.

③ Increasing the size of each bit field → instruction longer → may increase code size. #

But, the increase in # of registers → less use among lots of registers
→ reduce the total # of instructions → may reduce code size

Report on matrix multiplication :

53195612.

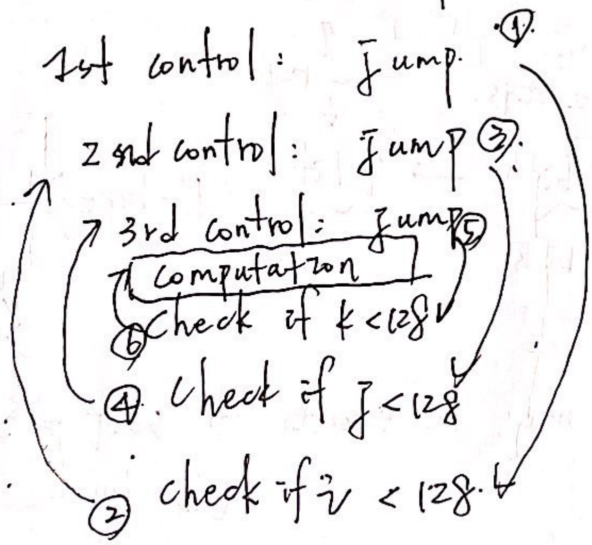
-① naive multiplication : 115558044 cycles / block multiplication :

-③ # of load & store : $\underbrace{4 \times 128}_{\text{1st loop}} + \underbrace{4 \times 128^2}_{\text{2nd loop}} + \underbrace{(4+17) \times 128^3}_{\text{3rd loop}}$
(naive compute)

$= 410112$

-④ There are three loops to do the multiplication

The # of loop control for each Loop is 2 → 6 in total.



1st control : jump ①
2nd control : jump ③
3rd control : jump ⑤
Computation
⑥ Check if $k<128$
④ Check if $j<128$
② check if $i < 128$.

- ③ The use : C[i][j] += A[i][k] * B[k][j] → one entry in the result matrix at a time

⇒ Can compute a small block of entries simultaneously

eg.
```
for (i=0 ; i< 128 ; i+=2){
      for (j=0 ; j< 128 ; j+=2 ) {
          a_00 = a_01 = a_10 = a_11 =0;
          for (k=0 ; k<128; k++) {
              a_00 += B[k][j+0]* A[i+0][k];
              a_01 += B[k][j+1] * A[i+0][k];
              a_10 += B[k][j+0] * A[i+1][k];
              a_11 += B[k][j+1] * A[i+1][k];
          }
          C[i+0][j+0] = a_00 ;
          C[i+0][j+1] = a_01 ;
          C[i+1][j+0] = a_10 ;
          C[i+1][j+1] = a_11 ;
      }
}
```

→ each load values twice

⇓

4 loads

4 multiply-adds.