Please write down the MIPS assembly code for the C statement (a).

NOTE: Assume that the variable, C[x+y*N], A[x+z*N], B[z+y*N], have already been loaded into the CPU registers, $f4, $f6, and $f8 respectively.

(a)

```
for (int x=0 ; x<N ; x++)
    for(int y=0; y<N ; y++)
        for(int z=0; z<N ; z++)
            C[x+y*N] += A[x+z*N] * B[z+y*N];
```

Ans.

Suppose that the register $s0, $s1, and $s2 are assigned to the variable x, y, and z, respectively. The constant N is contained in the register $s3.

MIPS:

```
1. add   $s0, $zero, $zero    # x = 0
2. L1:   slt   $t0, $s0, $s3  # label "L1", If (x<N) Then return 1
3.                            # Else return return 0.
4. beq   $t0, $zero, ExitX    # if (x≥N) goto "ExitX"
5. add   $s1, $zero, $zero    # y = 0
6. L2:   slt   $t0, $s1, $s3  # If (y<N) Then return 1
7.                            # Else return 0
```

```
8.  beq   $t0, $zero, ExitY   # if(y≥N) Then goto "ExitY"
9.  add   $s2, $zero, $zero   # z=0
10. L3:  slt   $t0, $s2, $s3  # If(z<N) Then return 1
11.                           # Else return 0
12.  beq  $t0, $zero, ExitZ   # If(z≥N) Then goto "ExitZ"
13.  mul.d $f2, $f6, $f8      # temp=f2 = A[]xB[]
14.  add.d $f4, $f4, $f2      # C[] = C[]+temp
15.  addi  $s2, $s2, 1        # z++
16.  j     L3                 # goto "L3"
17. ExitZ: addi, $s1, $s1, 1  # y++
18.  j     L2                 # goto "L2"
19. ExitY: addi, $s0, $s0, 1  # x++
20.  j     L1                 # goto "L1"
21. ExitX:                    # all done.
```

Which datapath (s) of the target CPU should be optimized if we want to accelerate the computation speed of above operations?

Ans.

The given code requires massive transmission of operands between registers and memory. Since memory is very slow relative to CPU, to optimize the speed of memory system will accelerate the computation speed of the operations.

Question:

Sometime, embeded processors may not have the hardware support of floating-point operations. On such embedded platforms, the floating-point operations could be emulated by a software implementation of the floating-point operations (as library routines) or by a integer operations. What is the technique of the floating-point emulation with integer operations?

1. Floating-point unit emulator (a floating-point library):

In systems without any floating-point hardware, the CPU emulates it using a series of simpler fixed-point arithmetic operations that run on the integer arithmetic logic unit. The software that lists the necessary series of operations to emulate floating-point operations is often packaged in a floating-point library.

2. Integrated FPU:

In some cases, FPUs may be specialized, and divided between simpler floating-point operations (mainly addition and multiplication) and more complicated operations, like division. In some cases, only the simple operations may be implemented in hardware or microcode, while the more complex operations are implemented as software.

Question:

Continue with the above question. Which one of the two emulation techniques is better for computaion efficiency?

Ans.

Integrated FPU is better for computation efficiency since simpler floating-point operations are still performed by hardware.

Question:

On a single-core embeded processor, which type of instructions is often supported by the embedded processor to take advantage of the data-level parallelism found in the above example code? Please rewrite the above code into the version with such type of instructions.

Ans.

The kernel of the given code is to multiply two array variables and accumulate the result to another array variable. So, if a multiply-add instruction (for example, madd.d $f4, $f6, $f8, this instruction multiply operands in registers $f6 and $f8, respectively, and then accumulate the result to register $f4) is added to the ISA, the data-level parallelism of the given code could be increased since two operations, multiply and add, are performed within a single instruction.

The following code rewrite the code into the version with multiply-add instructions.

```
.
.
.
.

L3: slt   $t0, $s2, $s3
    beq   $t0, $zero, ExitZ
    madd.d $f4, $f6, $f8   # Merged mul.d and add.d
    addi  $s2, $s2, 1
    j     L3
    :
```

Convolution neural networks (CNNs) have been widely adopted in many different application domains. Nevertheless, as CNNs incur high computation overhead and large memory footprint (占用), it is an important task to customize the designs of embedded processors so as to meet the timing constrains required by target applications. Among other types of computations, matrix operations are time-consuming functions in CNNs and are important targets for hardware and software optimizations to facilitate the CNN computations on such platforms. Given the partial C code for the convolution operations, there are three N-by-N matrices (A, B, and C), and they keep double-precision floating-point numbers (each number is eight bytes). Please answer the following questions related to processor designs and software optimizations.

```
for(int x=0; x<N; x++)
    for(int y=0; y<N; y++)
        for(int z=0; z<N; z++)
            C[x+y*N] += A[x+z*N] * B[z+y*N];
```

Question:

When N is 64, what is the smallest L1 data cache size, which can accommodate the data of all three matrices and provide no cache conflicts? NOTE: The cache should be sized in powers of 2.

Ans.

Each matrix is 64×64 = 4k elements and each elements is 8 bytes, so the three matrices occupy 4K × 8 bytes × 3 = 96 Kbytes.

The smallest L1 data cache size = 128 Kbyte (since the cache size is required to be in powers of 2).

Question:

Usually, to ensure the matrix elements can fit in the cache enjoying spatial and temporal locality, it is a common pratice to rewrite the code into the blocked version so that the computations are done within submatrices. Please determine the maximum matrix size, M, so that the three submatrices can fit in a 16 KB data cache.

Ans.

There are at most (16 Kbytes / 8 bytes) / 4 = 512 = 2^9 elements in each matrix. Since $2^4 \times 2^4 < 2^9 < 2^5 \times 2^5$

→ the maximum matrix size (M) = $2^4 \times 2^4 = 16 \times 16$.

Question:

Continue with the above question. Please rewrite the above C code into the blocked version with the calculated M.

Ans.

The following rewrites the given C code into the blocked version with blocksize = 8.

```
// C code
1. #define blocksize 8
2. void do_block (int si, int sj, int sk, double *A, double *B, double *C)
3. {
4.     for (int i=si; i<si+blocksize; i++)
5.         for(int j=sj; j<sj+blocksize; j++){
6.             double cij = C[i+j*16];
7.             for(int k=sk; k<sk+blocksize; k++)
8.                 cij += A[i+k*16] * B[k+j*16];
9.             C[i+j*16] = cij;
10.        }
11. }
```

```
12. void dgemm (double *A, double *B, double *C ){
13.     for(int sj=0; sj<16; sj+= blocksize)
14.         for(int si=0; si<16; si+= blocksize)
15.             for(int sk=0; sk<16; sk+= blocksize)
16.                 do_block(si, sj, sk, A, B, C);
17. }
```