Team Members: Pratik Mistry, Aditya Singh Thakur, Vikhyat Dhamija

Solution 1:

A von Neumann system consists of a central processor with an arithmetic/logic unit and a control unit, a memory, mass storage, input and output. The addition of cache and virtual memory to a von Neumann system will not change its designation as a SISD system. This is because the CPU cache is defined as a collection of locations in memory and is considered as a single monolithic structure. Further, cache makes it more convenient to have a quick access to data and instructions by the CPU in the main memory. Virtual memory keeps only the active parts of the currently running programs in main memory and the other parts are kept in secondary storage called swap places.

Pipelining is a technique in which instructions are lined one after the other for execution and although pipelining is a type of parallel hardware, it depends on the pipelining technique which affects the designation of an SISD system.

Multiple issues surely changes the designation of a von Neumann architecture as an SISD system to either of MISD or MIMD systems. This is because Multiple Issues try to simultaneously execute different instructions at the same time period on the basis of speculation.

Hardware Multithreading also changes the designation of a von Neumann architecture from SISD system to either SIMD or MIMD system. This is because hardware multithreading aims to provide a quick switch between threads during execution.

Solution 2:

In order to analyze loop-level parallelism, we need to determine whether there is a “loop-carried dependence” — i.e., whether data accesses in iterations are independent of each other.

Then there is instruction level parallelism, when all the instructions within the loop are independent so that ILP with multiple data can be applied in order to maximise the parallelism.

A loop is parallel if it can be written without a cycle in the dependencies: the absence of a cycle means that the dependencies give a partial ordering on the statements. We do have to make this loop conform to the parallel ordering to expose the parallelism.

For Loop 1 –

```
 Loop 1: for (i=0;i<100;i++)
{
        A[i] = A[i] * B[i]; /* S1 */
        B[i] = A[i] + c; /* S2 */
        A[i] = C[i] * c; /* S3 */
        C[i] = D[i] * A[i]; /* S4 */
}
```

There are true dependencies from S1 to S4 because of A[i] within the loop. But these aren't loop carried, so they do not prevent the loop from being parallelized. They will, however, force the later statements to wait until it completes the earlier ones first.

So in order to maximise the parallelism what we can do is :-

Renaming LHS A[i] with any variable name let say x. Which will make the statements 1 and 2 independent and hence parallel.

But since s3 is changing A[i] so it must execute after 1 and 2 being executed and then the 4th statement which is dependent on A[i] must be executed.

So we converted 2 statements independent and hence can be parallelized.

For Loop 2 –

As can be seen below, Statement S1 uses the value assigned to the current index of B in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1.

```
for (i=0;i < 100;i++) {

    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */

}
```

However, this can be made parallel, because the dependence is not circular — neither statement depends on itself, and only S1 depends on S2, not the other way around.

1. There is no dependence from S1 to S2. If there were, then this would be a cycle in the dependencies, and the loop would not be parallel. Since this other dependency is absent, we can exchange the order of the statements, which won't affect the execution of S2.
2. In each iteration of the loop, S1 depends on the value of B at the current index computed prior to the loop. With these two observations, we can replace the loop above with the following code:

```
A[0]=A[0]+B[0]
for (i=1 ; i < 100 ; i++)
{
        B[i] = C[i-1] + D[i-1]; /* S2 */
        A[i] = A[i] + B[i]; /* S1 */
}
```

So Here Value of B at the current index is getting calculated from the previous values and then A at the current index is calculated from the value of B and A.

The dependencies are no longer loop carried, so we can now parallelize the loop. We do need to ensure that the two statements in the loop remain in order, but taken together, they are independent of all the other iterations of the loop.

Solution 3:

Part 1 :

C pseudo code :

```
for ( I = 1; I < 1000000 +1 ; I++ )
{
        Y[I] = X[I-1] + Y[I] + Z[I+1] ;
}
```

Yes, we can execute safely in a data-parallel model language using loop parallelism. Reason being that there is no write read dependency between various iterations of the loop as we are modifying the value of Y with index I in Ith iteration, so all iterations are independent. Thus, using **forall** which is a data parallel expression, Compiler automatically understands that loop iterations are independent, and that the same loop body will be executed on a large number of data elements.

Part 2:

```
nsize = sizeof(a);
#pragma concurrent
for (i=0; i< nsize; i++)
    a[i] = a[i+m];
```

Just looking at this without thinking about the various possibilities it simply appears that answer to the above is No because of the fact that any i indexed array value is dependent on i+m th index element and in case we will be applying the loop level parallelism then it may be that part of the loop starting with higher indexes may execute first and hence the result will be different as we expected out of this program. Suppose considering m = 4 :

a[1]=a[5]

a[5]=a[9]

Then if the second statement executes first then the value of a[1] will be different from what is expected out of the program.

Yes, in order to make it parallel what we can do is that we may execute the loop not for all values of the array but with index less than m so in that case other parts of the array will not change hence no write read dependencies in the loop and then it will work fine.

Part 3:

```
for (i=0; i< nsize; i++)
        printf ("element A[%d] = %f\n",i,a[i]);
```

The answer to this will be no and the reason for this is that we want the program to print in order then in case we perform this task in loop parallel mode then the order of the printing will not be sequential and can be anything. So , we can execute them using loop parallelism when we do not care about the order of printing.

Solution 4:

Given -

single-core single-processor system with CPU speed of 4GHz.

L1 size - 48KB

DRAM size - 418 MB

The latency to L1 is 5 cycles and the latency to main memory is 150 cycles

ANS -

Assuming that the memory bandwidth is so high that we can fill both a and b vectors in the L1 cache in one main memory cycle, and as per question we can fetch the cache line with a size of 8 words.

1 cycle takes = 1/ (4x10^9) secs

One access to memory takes = 150/ (4x10^9) secs for 150 cycles

With Dim as size of array , time to get the two lines of 8 words to perform one operation is
5/ (4x10^9) * 2

So total operations per second for complete vector multiplication are :
(dim/8)* 5/ (4x10^9) * 2 + 150 / (4x10^9)

Solution 5:

Assume the following instruction stream or kernel:

```
for (int x=0; x<dimX-1; x++) {
for (int y=0; y<dimY-1; y++) {
for (int z=0; z<dimZ-1; z++) {
        int index = x*dimY*dimZ + y*dimZ + z;
        if (y>0 && x >0) {
                solid = idx[index];
                dH1 = (Hz[index] – Hz[index-incY])/dy[y];
                dH2 = (Hy[index] – Hy[index-incZ])/dz[z];
                Ex[index] = Ca[solid]*Ex[index]+Cb[solid]*(dH2-dH1);
}}}}
```

It is given that

dH1, dH2, Hy, Hz, dy, dz, Ca, Cb, Ex are single-precision floating-point arrays and idx is an unsigned int array.

Part 1:

Let's look at the number of math and data access operations in this single instruction stream.

int index = x*dimY*dimZ + y*dimZ + z; - *Math operation = 5, Data access = 0*

solid = idx[index]; - *Math operation = 0, Data access = 1*

dH1 = (Hz[index] – Hz[index-incY])/dy[y]; - *Math operation = 3, Data access = 3*
dH2 = (Hy[index] – Hy[index-incZ])/dz[z]; - *Math operation = 3, Data access = 3*
Ex[index] = Ca[solid]*Ex[index]+Cb[solid]*(dH2-dH1); - *Math operation = 4, Data access = 5*

Thus, the arithmetic intensity which is given by ratio of number of math operations to data access operations for 5 clock cycles (since we run 5 instructions in one instruction stream) is:
Arithmetic intensity = Total Math Operations/Total Data Access = (16/12) = 1.33 per 5 clock cycles

Part 2:

We believe that the instruction stream/kernel is amenable to vector based execution because we have many instructions in this single instruction stream and not all of them are independent and neither they run on the same data. Also, vector based can help us execute this instruction stream having many instructions depending on each other in a completely parallel way.

Part 3:

In this case, it will be compute bound because if we assume this kernel to be executed on a processor that has 20Gb/sec of memory bandwidth, then the number of data access operations would be very fast. Thus, the performance of the kernel given will solely depend on the number of math operations and time it takes to complete it. Also, we have seen in Part 1 of this question where we found the arithmetic intensity that comes out to be greater than 1 because the number of math operations is higher than data access operations, we can say that this kernel is compute bound.

Solution 6:

Part 1.a:

1. **<u>Adding more ALU's:</u>**

For heavy uniform data calculations, we assume that the scientific applications would run the same calculations (instructions) for huge data (heavy data/multiple data). Thus, we would need more computing units to do these similar calculations on multiple data i.e. we would just need to add more units of ALU's in each core to achieve maximum efficiency.

For example, if we add 8 ALU's in each core then in each clock cycle, each core will take up one instruction and execute that operation on 8 different data. Thus, since we have 4 cores we will have 32 independent pieces of work running in parallel. This is classic example of 8-way SIMD (single instruction multiple data)

2. **<u>Adding ALU's and Execution Contexts:</u>**

We can also add 4 Execution Contexts along with 8 ALU's because adding execution contexts would allow us to execute more threads i.e. interleaved multithreading can be achieved. Thus, it would run 16 concurrent instruction streams and in total 128 independent pieces of work.

But this would increase complexity to manage the execution specially if there is only one type of operation needed for execution. This is because context switching also takes up some clock cycles and hence there is no point if we have only one type of operation needed to perform on multiple data.

Part 1.b:

1. **Adding Fetch and Decode Units:**

In this case since we have multiple diverse programs from various users, we would have a lot of independent instruction streams as well while the data might be huge or small. And to achieve maximum efficiency where we have multiple independent instructions streams, we need to add Fetch and Decode units in each core. This will help fetch different independent instruction streams in each core for each user running diverse programs.

For example, we have 4 users who are running their own distinct programs and have various instructions in each of the programs. Assuming each user executes its program in each individual core and we add 2 Fetch and Decode units in each core, we can run two independent instructions for each of the programs in each core. Thus, it is the same as 2-way ILP (instruction level parallelism) in each core. Now, since we have 4 cores we can run a total 4 independent instruction streams with total 8 instructions in parallel.

2. **Adding Fetch and Decode units and ALU's:**

In the above point we discussed adding F/D units, but if the program that each user runs in each core has similar instructions and multiple data. In that case, we add more ALU's to each core so that the same instruction can be executed for multiple data in the program that the user runs.

For example, if we add 2 F/D and 8 ALU's in each core, then we can execute 2 independent instructions and 16 total independent pieces of work for each program running in each core. Thus, as we have 8 different instructions across 4 different cores and in total 64 pieces of work.

3. **Adding F/D units, ALU's and Execution Contexts:**

This is the most complex architecture where we are adding F/D units to achieve ILP in each core, ALU's to maximize the computing and Execution contexts to achieve simultaneous multithreading. Thus, if we add execution contexts as well to each core, then the program that is run by each user in its core can achieve interleaved multithreading for its own program.

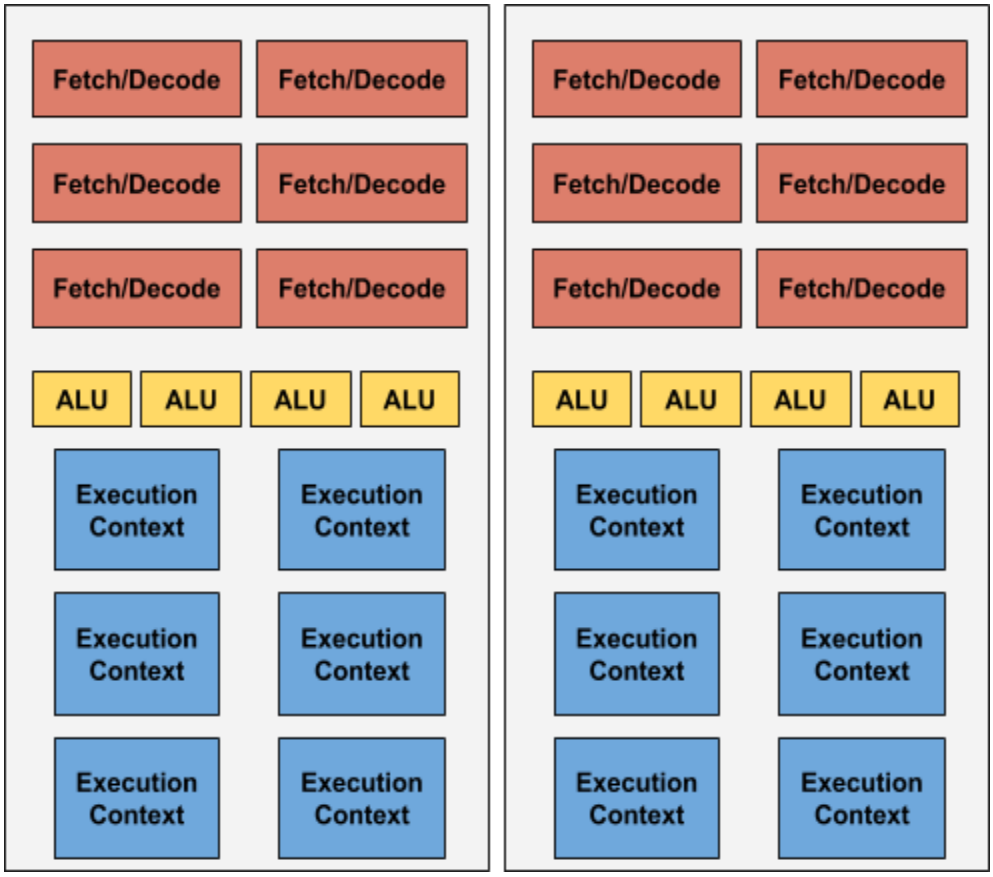
For example, if we add 2 F/D, 8 ALU's and 4 Execution contexts in each core then we can have a total 8 threads per core (2 F/D and 4 Execution contexts) and thus 64 pieces of work per core. Thus in total, we can have 32 concurrent instruction streams (4 cores and 8 threads per core) and 256 pieces of work running in parallel.

Part 2:

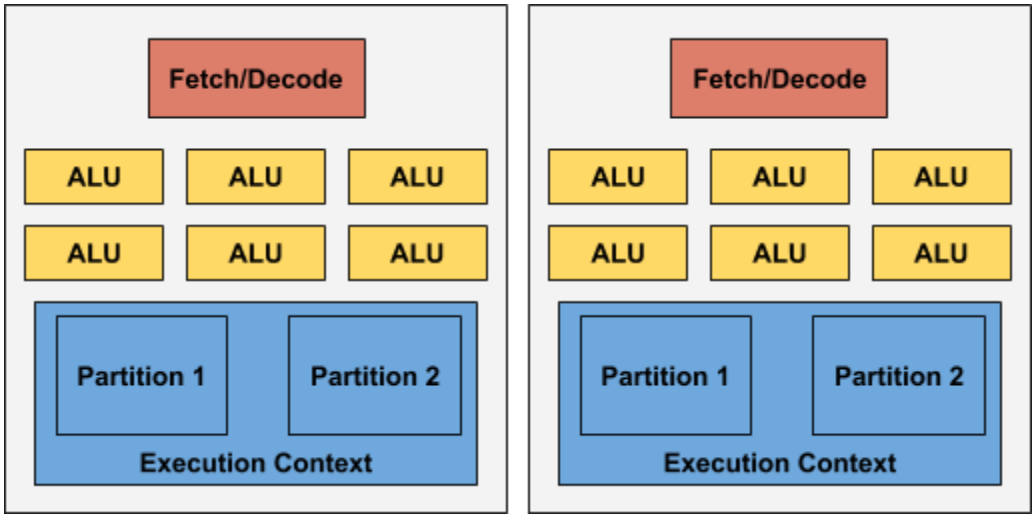
If we talk about the entire quad-core system then it is surely MIMD because each core can run multiple instructions in parallel i.e. SMT is achievable within each core and also across cores we can have simultaneous independent instruction streams. Further, to achieve maximal utilization of resources within each core, we must have an instruction stream that has two independent instructions which can be made run by the two different threads. One of the instructions can be a SISD based instruction while the other instruction can be SIMD based instruction because we have a combination of 1 F/D, 1 ALU and 1 Execution Context and also a combination of 1 F/D, 8 ALU's and 1 Execution context. Thus, within each core we can have SMT with SISD and SIMD both.

Solution 7:

Part 1.a: Below is the picture of 2 core, 6 SMT, 4-wide SIMD capability



Part 1.b: Below is the picture of 2 core, 2 interleaved/temporal Multithreading, 6-wide SIMD capability



Part 2:

Solution 1:

In architecture 1, we can run 12 independent instructions simultaneously because we have 2 cores and each core can execute 6 independent instructions because we have 6 Fetch/Decode units. While in architecture 2, we can run 2 independent instructions simultaneously because we have 2 cores only each with 1 Fetch/Decode unit.

Also, we can run 12 instructions concurrently in architecture 1 because we have 6 Execution Units and also 6 Fetch/Decode units in each code. Thus, since only one thread can be running at a time, each core runs 6 instructions (same or different) concurrently i.e. 12 across both the cores. While in architecture 2, we can run only 4 concurrent instructions streams because we have 2 execution contexts partitioned per core.

Solution 2:

Architecture 1 offers high memory latency reduction because we have 6 independent instructions running parallelly in each thread and if one thread is stalled on memory there are additional threads utilizing the processor at the same time (same clock cycle). However, architecture 2 offers low memory latency reduction because we have 2 interleaved threads running the same instruction on different elements/data and thus if a instruction stream pointed by the current thread runs until it either hits a memory stall (e.g. load and store operation) or until it finishes, only then the next thread is triggered.

Solution 3:

In architecture 1, we would need 48 independent pieces of work (data) to run the chip with maximum latency hiding ability. While in architecture 2, we would need 24 independent pieces of data (work) to run the chip with maximum latency hiding ability. Thus, chip with architecture 1 offers maximum latency hiding ability as it can process 48 independent data per clock cycle concurrently and also it can run different independent instructions in each core as well. This increases the efficiency of the chip mainly because there would always be nearly full utilization of all the resources.

Solution 4:

For heavy computation bound SIMD applications, we would strongly recommend the processor with architecture 2 because it can load one instruction in each clock cycle while working on 6 different sets of data in parallel for each core. Thus, if we have a huge dataset on which a particular operation needs to be performed, it would be very efficient if we use architecture 2. Moreover, since we can do interleaved threading in each core of architecture 2 and execute the same operation for multiple data, we can achieve very high efficiency as the processor is capable to perform heavy computation.

Now for distributed multi-user, multi-purpose based systems, we suggest to go with architecture 1 because it is capable of 6 simultaneous multi threading in each core where it can run 6 same or different instructions in one clock in each core. Thus, as the distributed multi-user and multi-purpose based system is bound to have different independent instructions, architecture 1 would serve the purpose of parallelism making the system more efficient.