Solution to Multithreading Example

Assume you are using a processor with the following characteristics:

- 4 functional units: 2 ALUs, 1 memory port (load/store), 1 branch
- In-order execution

Given the three threads below, show how these instructions would execute using:

- Fine-grained multithreading
- Coarse-grained multithreading
  - Switch threads on any stall over 2 cycles
- Simultaneous multithreading
  - Thread 1 is preferred, followed by Thread 2 and Thread 3

You should assume any two instructions without stalls between them are independent.

Thread 1:	Thread 2:	Thread 3:
ADD.D	SUB.D	L.D
L.D	stall	stall
stall	L.D	stall
stall	S.D	stall
stall	L.D	stall
stall	stall	stall
SUB.D	ADD.D	stall
S.D	stall	ADD.D
stall	BNE	stall
BEQ		stall
		S.D
		stall
		stall
		BEQ

First, some notes:

- All threads are independent of one another.
- Each individual thread contains very few independent instructions that can actually be scheduled together. Only Thread 1, which has the ADD.D/L.D and SUB.D/S.D pairings, contains independent instructions that can be simultaneously scheduled on this machine. Thread 2 has the L.D/S.D/L.D instructions, but all need to use the single memory port. Thread 3 has no independent instructions.
- The stall cycles given indicate not only if two instructions are independent, but also the amount of time that must pass between dependent instructions.
- Once we have executed all instructions in a given thread, that thread is no longer active and should not be considered when alternating threads.

Fine-grained multithreading: In this case, we alternate threads every cycle, executing as many independent instructions as we can. If there are no available instructions in the chosen thread during a particular cycle, the machine effectively stalls, as shown in cycles 4, 6, 9, 14, 17, and 18. What determines whether instructions are available is the latency between dependent instructions—in this example, the stall cycles shown in each thread. For example, Thread 1 stalls in cycle 4 because there are four stall cycles between its L.D and SUB.D instructions. With the L.D issued in cycle 1, the SUB.D cannot issue before cycle 6.

The long latency stalls in Thread 3 lead to a number of stall cycles—five of the six stalls described above. However, cycling between threads does allow both Threads 1 and 2 to proceed with minimal stalls (only one stall for Thread 1).

Cycle	ALU1	ALU2	Memory	Branch	
1	T1: ADD.D		T1: L.D		
2	T2: SUB.D				
3			T3: L.D		
4					T1 stall
5			T2: L.D		
6					T3 stall
7	T1: SUB.D		T1: S.D		
8			T2: S.D		
9					T3 stall
10				T1: BEQ	
11			T2: L.D		
12	T3: ADD.D				
13	T2: ADD.D				
14					T3 stall
15				T2: BNE	
16			T3: S.D		
17					T3 stall
18					T3 stall
19				T3: BEQ	

Coarse-grained multithreading: One of the downsides to fine-grained multithreading, as we discussed, is that individual threads that can finish quickly have to wait for their turn even if they have instructions ready to execute. Coarse-grained multithreading attempts to fix this issue by only switching threads on long-latency stalls. The processor still rotates through all threads using round-robin scheduling, executing instructions from one thread at a time, but short stalls are tolerated.

In this example, we defined "long-latency stalls" as being longer than 2 cycles, so the shorter stalls—cycle 13 for Thread 1, 3, 7, and 9 for Thread 2, and cycles 19-20 and 21-22 for Thread 3—remain. (The stalls in cycles 15-17 for Thread 3 are part of a long-latency stall that impacts execution only because Threads 1 and 2 have finished.)

Note that this organization works extremely well for Thread 2, which has the fewest stall cycles in this example. This thread's quick finish illustrates the main benefit of coarse-grained multithreading. Note, also, that this organization allows the processor to more reasonably tolerate the long-latency stall between the ADD.D/L.D and SUB.D/S.D pairs in Thread 1.

Cycle	ALU1	ALU2	Memory	Branch	
1	T1: ADD.D		T1: L.D		
2	T2: SUB.D				
3					T2 stall
4			T2: L.D		
5			T2: S.D		
6			T2: L.D		
7					T2 stall
8	T2: ADD.D				
9					T2 stall
10				T2: BNE	
11			T3: L.D		
12	T1: SUB.D		T1: S.D		
13					T1 stall
14				T1: BEQ	
15					T3 stall
16					T3 stall
17					T3 stall
18	T3: ADD.D				
19					T3 stall
20					T3 stall
21			T3: S.D		
22					T3 stall
23					T3 stall
24				T3: BEQ	

Simultaneous multithreading: Both fine-grained and coarse-grained multithreading suffer from the limitation of only choosing instructions from a single thread at one time. Simultaneous multithreading allows a processor to check all threads for independent instructions, thus improving the utilization of the hardware as well as overall performance.

Simultaneous multithreading must have some way of choosing which threads, if any, to prioritize each cycle. Our example uses a simple preferred thread scheme, with Thread 1 as the highest-priority thread, followed by Thread 2 and then Thread 3. This method does not maximize fairness and may cause lower-priority threads to be starved, so more elaborate thread selection schemes may be used.

Cycle	ALU1	ALU2	Memory	Branch	
1	T1: ADD.D	T2: SUB.D	T1: L.D		
2			T3: L.D		
3			T2: L.D		
4			T2: S.D		
5			T2: L.D		
6	T1: SUB.D		T1: S.D		
7	T2: ADD.D				
8				T1: BEQ	
9	T3: ADD.D			T2: BNE	
10					T3 stall
11					T3 stall
12			T3: S.D		
13					T3 stall
14					T3 stall
15				T3: BEQ	