COL 380	January 26, 2018
Homework 1	
Instructor: Subodh Sharma	**Due: February 2, 23:55 hrs**

NOTE: Answer submissions must be made either in word document or pdfs. No hand-written assignments would be accepted. All assignment submissions will be checked for plagiarism.

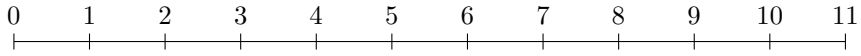
Problem 1: Pipelining

Consider the execution of the following code:

```
float x[1000], y[1000], z[1000];
...
for(i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

In each iteration of the loop, we are adding two floats which require 7 operations (Fetch operands, Compare exponents, Normalize, Add, Normalize result, Round result, Store result). Assuming "Fetch operands" and "Store result" take 2ns each and every other operation takes 1 ns to complete, answer the following:

- How long does it take to execute the loop in an unpipelined processor? (2 marks)
- How long does it take to execute the loop in a 2-way pipelined processor? Show the pipeline state after each cycle for 11 cycles. The format is shown below. (4 marks)

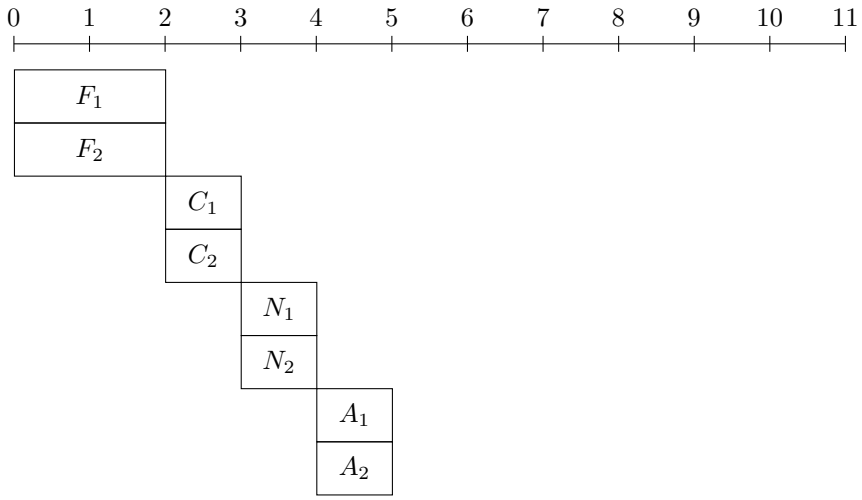


For each subproblem, show your working.

Solution 1: Pipelining

- For a single operation, it would take 9 cycles or 9 ns (2+1+ 1+1+1+1+2). So for executing 1000 times, it would take 9000ns.
- After first 9ns, two iterations of the loop will be executed, *i.e.* , 2 add operations will be discharged. After which, every 2ns we will witness completion of 2 add operations. This means, that total time taken for the loop to execute = 9 + 998 * (2/2) = 1007ns.

  A partial pipeline snapshot is shown below.



Problem 2: Caches

Consider a memory system with L1 cache of 32 KB and DRAM of 1 GB with the processor operating at 1GHz. The latency to L1 cache is 1 ns and that to DRAM is 100 ns. In each fetch cycle, 4 memory words (a cache line) are transfered from DRAM to CPU.

- Consider now the problem of multiplying two dense matrices (4K $\times$ 4K) as shown below; the matrices are laid out in a row-major fashion in memory. What is the peak achievable performance for the sought multiplication? (4 marks)

```
for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j++)
    for(k = 0; k < SIZE; k++)
      z[i][j]  += x[i][k] * y[k][j];
```

Solution 2: Caches

NOTE: We have not accounted for cache access latency (because it is very small) and the time to to store the results back to DRAM . If students do take it in account, mark accordingly.

- Here, assuming the multiplication algorithm is the following:

```
for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j++)
    for(k = 0; k < SIZE; k++)
    z[i][j]  += x[i][k] * y[k][j];
```

Then, 5 cache lines will have to fetched, one for the matrix X and 4 for the matrix Y (since the access is performed in a column-major fashion). Thus, the peak performance = 8 FLOPs/500 ns = 16 MFLOPS. Of course, one can assume a different and an optimized matrix-multiply algorithm, in which case cache line fetches may drop. As long as the explanation is consistent with the algorithm used, allot marks!

Problem 3: Mutual Exclusion & Deadlocks

Barrier is a synchronization construct where a set of processes synchronizes globally i.e. each process in the set arrives at the barrier and waits for all others to arrive and then all processes leave the barrier. Let the number of processes in the set be three and S be a binary variable (also called a *binary semaphore* used to implement lock (resp. unlock) functions L (resp. UL). Consider the following **C** implementation of a barrier with line numbers shown on left.

```
1  void barrier (void) {
2   L(S);
3     process_arrived ++;
4   UL(S);
5   while (process_arrived !=3);
6   L(S);
7     process_left++;
8     if (process_left==3) {
9       process_arrived = 0;
10      process_left = 0;
11    }
12  UL(S);
13 }
```

The variables process arrived and process left are shared among all processes and are initialized to zero. In a concurrent program all the three processes call the barrier function when they need to synchronize globally. The above implementation of barrier is incorrect. Which one of the following is true? (2 marks)

1. The barrier implementation is wrong due to the use of binary variable S.
2. The barrier implementation may lead to a deadlock if two barrier in invocations are used in immediate succession.
3. Lines 6 to 10 need **not** be inside a critical section.
4. The barrier implementation is correct if there are only two processes instead of three.

How would one rectify the problem? (4 marks)

Solution 3: Mutual Exclusion & Deadlocks

The answer is (2). The arrival-update code-section of the second call to barrier can interleave with the left-update code-section of the first call to barrier leading to a deadlock.

The problem could be rectified in many ways: simplest way (without creating too many changes to the code) is to use three binary semaphores to appropriately coordinate the update of process_arrived with the update of process_left.

Problem 4: Mutual Exclusion & Deadlocks

Consider following algorithm for realizing mutual exclusion between two concurrent processes i and j.

```
1 while(true) {
2   flag [i] = true;
3   turn = j;
4   while ( P ) ;
5    /* enter critical section, perform actions, exit critical section */
6   flag [i] = false;
7 }
```

For the program to guarantee mutual exclusion, the predicate P in the while loop should be (3 marks):

1. flag[j] = true and turn = i
2. flag[j] = true and turn = j
3. flag[i] = true and turn = j
4. flag[i] = true and turn = i

Solution 4: Mutual Exclusion & Deadlocks

it is option (2). All other options lead to ME violation.

Problem 5: Performance

- Assume $T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$ where p is the number of cores. Show that keeping p fixed, if we were to increase the input problem size, then *efficiency* will increase. All suitable assumptions must be clearly specified. (3 marks)
- Suppose $T_{serial} = n$ and $T_{parallel} = \frac{n}{p} + log_2 p$. Show whether or not the program is scalable. (4 marks)
- Design a cost-optimal version of the parallel-sum algorithm for n numbers on p processing cores such that $p < n$. Supposing that addition of two numbers takes 1 unit of time while communication among cores takes 20 units of time, derive expressions for $T_{parallel}$, S, E, cost, and the isoefficiency function. (6 marks)

Solution 5: Performance

- $E = \frac{T_{serial}}{p \times T_{parallel}}$ which becomes $E = \frac{1}{1 + p \times \frac{T_{overhead}}{T_{serial}}}$. Assuming $T_{overhead}$ is constant OR grows slowly in comparison to T_{serial} then increasing the problem size will make the denominator smaller, thereby increasing E.
- Lets increase p by a factor of k. If we can find an increase in n to keep the efficiency constant, then the program is scalable. Assume that increase is r. Then:

$$E = \frac{n}{n + plog(p)} \quad = \frac{rn}{rn + kplog(kp)} \tag{1}$$

$$r = \frac{klog(kp)}{log(p)} \tag{2}$$

  Since we have found an r, the program is scalable.

- The cost-optimal algorithm for adding numbers is when $\frac{n}{p}$ numbers on each core are added locally in $\Theta(\frac{n}{p})$ time and then parallel sum of the results on p cores proceed and finish in $\Theta(log\, p)$ steps. Therefore the runtime of the algorithm is $\Theta(\frac{n}{p} + log\, p)$. Lets now work with more operation details:

  1. each core performs $\frac{n}{p} - 1$ additions. And there are $log\, p$ steps.
  2. Thus, $T_{parallel} = (\frac{n}{p} - 1) \times t_{add} + (t_{add} + t_{communicate}) \times log\, p$

3. $i.e, T_{parallel} = (\frac{n}{p} - 1) + 21 \times log\, p$
4. $S = \frac{n}{\frac{n}{p} - 1 + 21 log\, p}$
5. $E = \frac{S}{p} = \frac{n}{n - p + 21 \times p \times log\, p}$
6. $Cost = p \times T_{parallel} = n - p + 21 \times p \times log\, p$
7. The asymptotic isoefficiency function is $\Theta(p \times log\, p)$