First Trimester Examination

Name: ___Mootaz Elnozahy_____

Student Id: ____xxxxxx_____

This is an open book examination. You are allowed to make use of the Internet, your class notes, etc. But you are not allowed to consult any human being, either in person or remotely. Attempt all questions.

Exam is graded out of 20, maximum grade points is 25.

Question 1 (2 points)

Congratulations, you have graduated, and you were hired as a manager at Cloudy Clouds, a company that specializes in cloud computing. You received for approval the following contract:

The company, Foggy Systems, is contracting a Web service with Cloudy Clouds with the parameters:

- Web requests arrive at random according to an exponential distribution, with average inter-arrival time of 50 msec.
- Each request is guaranteed not to exceed 5msec in processing time.
- A Service Level Agreement requires a request's response time cannot exceed 10 msec, with a very heavy penalty for violating this condition in addition of surrendering back any payment made by Foggy Systems.

Satoshi Nakamoto, an engineer in your working group, argues that this is a wonderful opportunity. Since requests arrive at such a low frequency, it is guaranteed that the response time requirements will be met. He provided you with a spreadsheet model showing his calculations and recommended approval. What is your decision? Explain.

A spreadsheet model can only capture the aggregate demands on resources, but it cannot account for the queuing delays and effects that are inevitable when the arrival rate is random. And notwithstanding Mr. Nakamoto's plausible hypothesis that the arrival rate being very low, the reality is that there will be situations when such requests arrive too close to one another since the arrival rate is random, at which time the queuing delays will prevent the system from meeting its response time obligations.

Question 2 (3 points)

A processor chip consists of 16 cores. Each core runs at 2GHz and has two floating point units. The memory bandwidth of the entire chip is 4GB/sec. Answer the following questions:

1) What is the peak rate of floating-point operations in GF/sec? (GF: Gigaflop)?
2) Under what condition can the peak rate that you computed in (1) be realized? Be precise!
3) Considering your answer to (2), argue about the suitability or the lack thereof of the peak floating-point operations as a metric to characterize the floating-point performance of the processor.

- The peak rate of floating-point operations occurs when all the resources are busy and allocated. In the chip we have, this will be:

  Number of cores x number of floating-point units x frequency

  16 x 2 x 2 = 64GF/sec

- The peak rate would occur under the unlikely situation when the pipeline is full and no stalls occur, either due to cache misses, data dependence conflicts, control or structural hazards. It is likely that to get to the peak rate, the data have to be resident in the cache.

- The peak rate has been a standard way to rate the performance of processors in data sheets (used by salespeople and trade magazines). It is quite misleading as pipeline hazards and cache misses occur frequently in more realistic applications. The Linpack standard has attempted to ameliorate the situation by providing some useful code that still has some excellent locality of references. This benchmark is a representative of some linear algebra codes. A well-designed *system* should be able to reach between 75% to 90% of peak performance on the Linpack benchmark.

Question 3 (5 points)

The Unnecessarily Complex Architecture (UCA) processor specifies a fully pipelined processor that executes out-of-order instructions and also allows speculative executions in a way to reduce the penalty of branches. The load-store (LS) unit of the UCA contains a queue of the memory writes that are waiting to be written to the cache (call it Q1). It also contains a separate queue of the memory writes belonging to instructions that have not committed yet (either because they ran out of their order or because they are speculative). Call the latter Q2. For either Q1 or Q2, each entry consists of an address and the data to be written.

1) Consider a memory address A. Is it possible for both Q1 and Q2 to contain an entry corresponding to A simultaneously?
2) Suppose a load instruction is issued to the LS unit requesting the data item at address A. There are three places to search (the cache, Q1 or Q2). Explain how the load instruction is to be handled, considering all possibilities.

- The answer to the first question is yes. For example, the code

```
A = 3;
if(x == 0)
        A = 4;
else
        A = 5;
```

In this example, (A, 3) could be queued in Q1 waiting to be written to the cache, while the pipeline already has executed speculatively along both branches with (A, 4) and (A, 5) being present as entries in Q2.

There are two cases to consider:

- The load instruction is not speculative:
  - Then, we must first search Q1 in case if a previous update is still pending in the queue and has yet to be written in the cache. If the data item is absent from Q1, then we issue the request to search the cache. We do not approach Q2 at all in this case.

- The load instruction is speculative:
  - This case is tricky, because we have to rely on the instruction scheduler to ensure that the data hazards are taken care of even in speculative execution. In this case, we search Q2 first, then if not found, we revert to the previous case.

Question 4 (5 points)

Consider the following code:

```
double a[N], b[N], c[N], d[N], e[N];// N is a constant
int i;

for(i = 0; i < N; i++) {
        e[i] = a[i] / b[i];
        e[i]+= c[i]/ d[i];
}
```

Rewrite this code so that it can be more pipeline-friendly.

As discussed in class, division operations are very expensive and are very difficult to pipeline. The loop contains two divisions. A rewrite of this code could be

```
for(i = 0; i < N; i++)
        e[i] = (a[i] * d[i] + c[i] * b[i]) / (b[i] * d[i]);
```

On an M2 processor, the running time of the second version of the loop is 37% faster than the original one! Even if we have added three floating-point multiplications, the code still ran faster than with two divisions! Note that this case benefits from the M2 having more than one floating-point unit which masks the penalty of the added multiplications.

Note that the incorrect answer is to rewrite the code so that the loop is unrolled by hand. Please remember that loop unrolling is performed by the compiler. Forcing the programmers to rewrite the code to unroll the loops by hand will provide ugly code that is difficult to maintain. Keep in mind performance is importance, but programmer's time and code maintainability should never be sacrificed for performance. Elegant solutions usually perform best!

Question 5 (10 points)

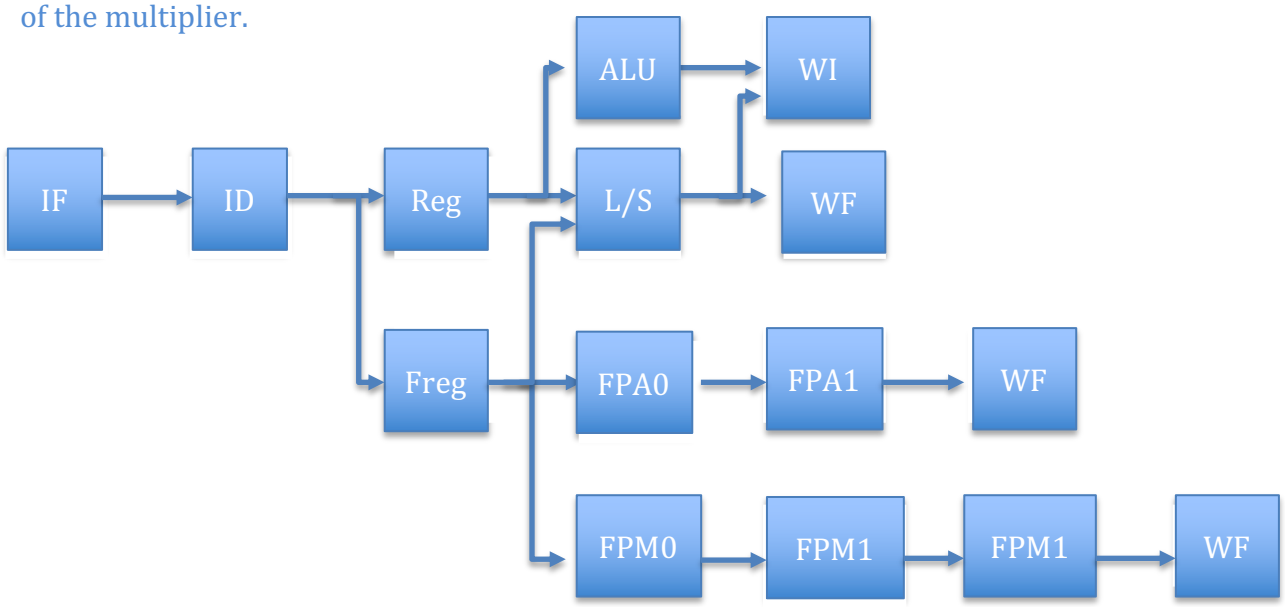
The computation Y = a * X + Y has long been used as a benchmark for processor performance (known as DAXPY). The typical assembly code for this function is:

```
start:  ld      f1, (r1)        # the machine has two register sets
        mul     f4, f2, f1      # a * X[i], f4 holds the results, a is in f2, and f1 holds X[i]
        ld      f6, (r2)        # Y[i],
        add     f6, f4, f6      # a*X[i] + Y[i]
        st      f6, (r2)        # Updating Y[i]
        add     r1, r1, 8       # increment X[i]'s index
        add     r2, r2, 8       # increment Y[i]'s index
        add     r3, -1          # the loop index
        bnz     r3, start
```

Assume the following components are available to you: an instruction fetch stage that can fetch up to two instructions in a cycle, an instruction decode that can issue up to two instructions in a cycle, an integer register file and a floating-point register file, each with four read ports and two write ports. A load-store unit can load a data item from the cache in two cycles and stores in one cycle. Assume that the integer ALU computes in one cycle, a floating-point multiplier that is pipelined over 3 stages, and a floating-point adder that is pipelined over 2 stages.

Design a pipeline showing how all the stages are connected. Draw a picture of the pipeline. Show how the code will execute on the pipeline. You may use out of order execution, register renaming, and branch prediction. Compute the IPC.

IF: Instruction fetch, ID: Instruction Decode, Reg: Reading integer register file
L/S: Load store, WI: write result into integer register file
Freg: Reading floating point register file, FPA0: Stage 0 of the adder, FPM0: Stage 0 of the multiplier.



IF	ID	Reg	L/S	ALU	Freg	FPA0	FPA1	FPM0	FPM1	FPM2
L0, L1										
L2, L3	L0, L1									
L4, L5	L2, L3	L0								
L6, L7	L4, L5		L0							
L8	L6, L7	L2, L5	L0							
	L8		L2	L5	L0					
		L6	L2		L1					
		L5		L6	L2			L1		
		L6, L7							L1	
				L7						L1
		L7			L1					
		L8			L3					
				L8		L3				
							L3			
		L4			L3					
					L4					
			L4							

```
L0:   ld    f1, (r1)      # the machine has two register sets
L1:   mul   f4, f2, f1    # a * X[i], f4 holds the results, a is in f1, and f2 holds X[i]
L2:   ld    f6, (r2)      # Y[i],
L3:   add   f6, f4, f6    # a*X[i] + Y[i]
L4:   st    f6, (r2)      # Updating Y[i]
L5:   add   r1, r1, 8     # increment X[i]'s index
L6:   add   r2, r2, 8     # increment Y[i]'s index
L7:   add   r3, -1        # the loop index
L8:   bnz   r3, start
```