CENG 3420 Homework 3

Due: Apr. 11, 2016

Q1 Problems in this exercise refer to the following instruction sequences:

```
I1: ADD R1, R2, R1
I2: LW  R2, 0(R1)
I3: LW  R1, 4(R1)
I4: OR  R3, R1, R2
```

1. Find all data dependences in this instruction sequence.
2. Find all hazards in this instruction sequence for a 5-stage pipeline with and then without forwarding.
3. To reduce clock cycle time, we are considering a split of the MEM stage into two stages. Find all hazards in this instruction sequence for this 6-stage pipeline with and then without forwarding.

Q2 Here are two different I/O systems intended for use in transaction proceeding:

System A can support 1500 I/O operations per second and use the processor that executes 400 million instructions per second.

System B can support 1000 I/O operations per second and use the processor that executes 500 million instructions per second.

Assume that each transaction requires 5 I/O operations and that each I/O operation requires 10,000 instructions. Ignoring response time and assuming that transactions may be arbitrarily overlapped, what is the maximum transactionpersecond rate that each machine can sustain?

Q3 I/O can be performed either synchronously or asynchronously. Explore the differences by answering performance questions about the following peripherals.

a. Mouse Controller

b. Memory Controller

1. What would be the most appropriate bus type (synchronous or asynchronous) for handling communications between a CPU and the peripherals listed in the table?
2. What problems would long, synchronous busses cause for connections between a CPU and the peripherals listed in the table?
3. What problems would asynchronous busses cause for connections between a CPU and the peripherals listed in the table?

Q4 Direct Memory Access (DMA) allows devices to access memory directly rather than working through the CPU. This can dramatically speed up the performance of peripherals, but adds complexity to memory system implementations. Explore DMA implications by answering the questions about the following peripherals.

a. Mouse Controller

b. Ethernet Controller

1. Does the CPU relinquish control of memory when DMA is active? For example, can a peripheral simply communicate with memory directly, avoiding the CPU completely?
2. Of the peripherals listed in the table,which could cause coherency problems with cache contents? What criteria determine if coherency issues must be addressed?
3. Describe what problems could occur when mixing DMA and virtual memory. Which of the peripherals in the table could introduce such problems? How can they be avoided?

Q5 For a multi-cycle processor, assume that the operation times for the major function units are as following:

1. Memory units: 200 ps;
2. ALU and adders: 100 ps;
3. Register file (read or write): 50 ps.

Assume that the multiplexors, control unit, PC accesses, sign extension unit and wires have no delay. Assume that instruction frequencies are as following:

1. 25% loads
2. 10% stores;
3. 15% branches;
4. 5% jumps;
5. 45% ALU instructions.

For pipelined execution, assume that half of the load instructions are immediately followed by and instruction that uses the result, that the branch delay on misprediction is 1 clock cycle, and that one-quarter of the branches are mispredicted. Assume that jumps always pay 1 full clock cycle of delay, so their average time is 2 clock cycles. Ignore any other hazards. Now suppose the memory access became 2 clock cycles long.

1. Draw the modified pipeline.
2. List all the possible new forwarding situations and all possible new hazards and their length.

Q6 For the following code:

```
for (int i = 0; i < N; ++i) {
    sum[i] = 0;
    for (int j = 0; j < i; ++j) {
        sum[i] = (sum[i] + array[j]) % N;
    }
}
```

Clearly the code takes O(N^2) time we would like to improve the actual running time.

Which of these strategies would you recommend. Why?

```
// Option1
parallel_for(int i = 0; i < N; ++i) {
    sum[i] = 0;
    for (int j = 0; j < i; ++j) {
        sum[i] = (sum[i] + array[j]) % N;
    }
}

// Option2
for(int i = 0; i < N; ++i) {
    sum[i] = 0;
    parallel_for(int j = 0; j < i; ++j) {
        sum[i] = (sum[i] + array[j]) % N;
    }
}
```

Q7 Consider the following recursive mergesort algorithm (another classic divide and conquer algorithm). Mergesort was first described by John Von Neumann in 1945. The basic idea is to divide an unsorted list x of m elements into two sublists of about half the size of the original list. Repeat this operation on each sublist, and continue until we have lists of size 1 in length. Then starting with sublists of length 1, merge the two sublists into a single sorted list.

```
Mergesort()
{
    var left, right, result
    if length(m) <= 1 {
        return m
    }
    else {
        var middle = length(m) / 2
    }
    for each x in m up to middle {
        add x to left
    }
    for each x in m after middle {
        add x to right
    }
    left   = Mergesort(left)
    right  = Mergesort(right)
    result = Merge(left, right)
    return result
}
```

The merge step is carried out by the following code:

```
Merge(left,right)
{
    var list, result
    while (length(left) > 0 and length(right) > 0)
    {
        if first(left) <= first(right) {
            append first(left) to result
            left = rest(left)
        }
        else {
            append first(right) to result
            right = rest(right)
        }
    }
    if length(left) > 0 {
        append rest(left) to result
    }
    if length(right) > 0 {
        append rest(right) to result
    }
    return result
}
```

1. Assume that you have Y cores on a multi-core processor to run MergeSort. Assuming that Y is much smaller than length(m), express the speedup factor you might expect to obtain for values of Y and length(m). Plot these on a graph.
2. Next, assume that Y is equal to length(m). How would this affect your conclusions your previous answer? If you were tasked with obtaining the best speedup factor possible (i.e., strong scaling), explain how you might change this code to obtain it.

Q8 Consider the following portions of two different programs running at the same time on four processors in a symmetric multi-core processor (SMP). Assume that before this code is run, both x and y are 0.

Core1: x = 2;

Core2: y = 2;

Core3: w = x + y + 1;

Core4: z = x + y;

1. What are all the possible resulting values of w, x, y, and z? For each possible outcome, explain how we might arrive at those values. You will need to examine all possible interleavings of instructions.
2. How could you make the execution more deterministic so that only one set of values is possible?