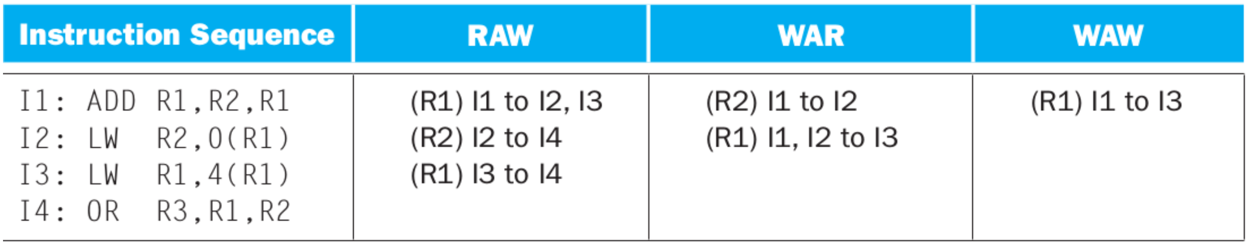
CENG 3420 Homework 3 Solutions

Due: Apr. 11, 2016

Question 1

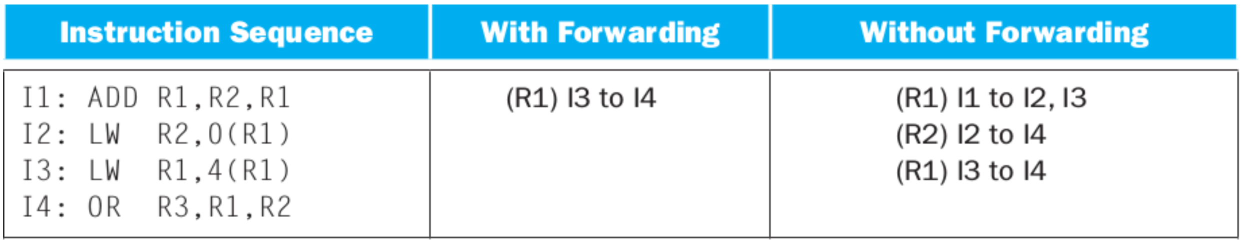
1. Dependency between instructions happens due to sharing common registers.



Instruction Sequence	RAW	WAR	WAW
I1: ADD R1,R2,R1 I2: LW R2,0(R1) I3: LW R1,4(R1) I4: OR R3,R1,R2	(R1) I1 to I2, I3 (R2) I2 to I4 (R1) I3 to I4	(R2) I1 to I2 (R1) I1, I2 to I3	(R1) I1 to I3

Figure 1: Dependency of instructions

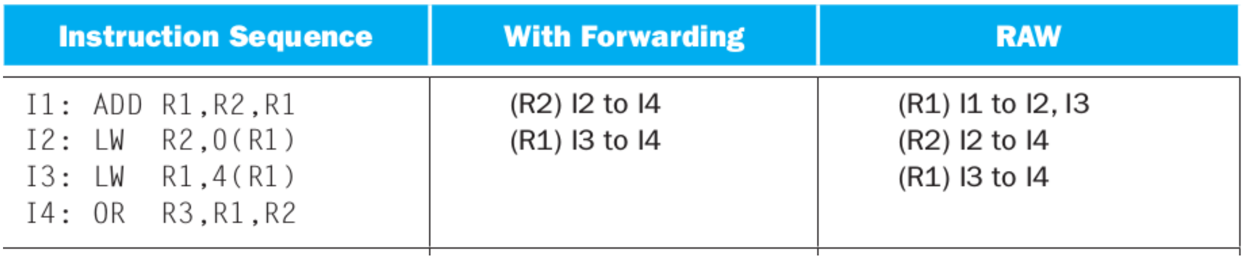
2. Only RAW dependences can become data hazards. With forwarding, only RAW dependences from a load to the very next instruction become hazards. Without forwarding, any RAW dependence from an instruction to one of the following 3 instructions becomes a hazard:



Instruction Sequence	With Forwarding	Without Forwarding
I1: ADD R1,R2,R1 I2: LW R2,0(R1) I3: LW R1,4(R1) I4: OR R3,R1,R2	(R1) I3 to I4	(R1) I1 to I2, I3 (R2) I2 to I4 (R1) I3 to I4

Figure 2: Data hazards

3. With forwarding, only RAW dependences from a load to the next two instructions become hazards because the load produces its data at the end of the second MEM stage. Without forwarding, any RAW dependence from an instruc- tion to one of the following 4 instructions becomes a hazard:



Instruction Sequence	With Forwarding	RAW
I1: ADD R1,R2,R1 I2: LW R2,0(R1) I3: LW R1,4(R1) I4: OR R3,R1,R2	(R2) I2 to I4 (R1) I3 to I4	(R1) I1 to I2, I3 (R2) I2 to I4 (R1) I3 to I4

Figure 3: Data hazards of 6-stage pipeline

Question 2

1. Each transaction requires 10000 * 5 = 50000 instructions.
   System A: CPU limit: 400M / 50K = 8000 transactions/second.
   The I/O limit for A is 1500/5 = 300 transactions/second.
   System B: CPU limit: 500M / 50K = 10000 transactions/second.
   The I/O limit for B is 1000/5 = 200 transactions/second.

Question 3

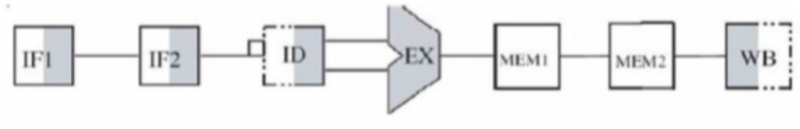
1. **For a**: The asynchronous bus should be selected. Mouse inputs are relatively infrequent in comparison to other inputs. The mouse device is electrically distant from the CPU.
   For b: we choose synchronous bus. The memory controller is electrically close to the CPU and throughput to memory must be high.
2. For all devices in the table, problems with long, synchronous buses are the same. Specifically, long synchronous buses typically use parallel cables that are subject to noise and clock skew. The longer a parallel bus is, the more susceptible it is to environmental noise. Balanced cables can prevent some of these issues, but not without significant expense. Clock skew is also a problem with the clock at the end of a long bus being delayed due to transmission distance or distorted due to noise and transmission issues. If a bus is electrically long, then an asynchronous bus is usually best.
3. The only real drawback to an asynchronous bus is the time required to transmit bulk data. Usually, asynchronous buses are serial. Thus, for large data sets, transmission can be quite high. If a device is time sensitive, then an asynchronous bus may not be the right choice. There are certainly exceptions to this rule-of-thumb such as FireWire, an asynchronous bus that has excellent timing properties.

Question 4

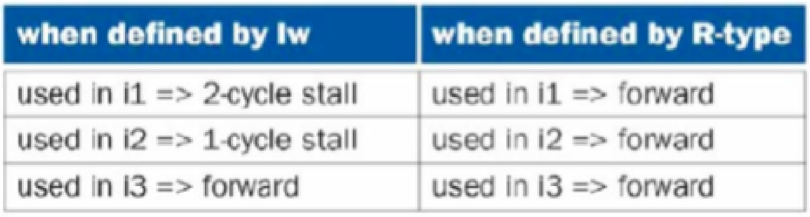
1. Yes. The CPU initiates the data transfer, but once the data transfer starts, the device and memory communicate directly with no intervention from the CPU.
2. **For a**: No. The mouse controller does not write back to system memory.
   For b: No. The ethernet controller does not write back to system memory.
   Basically, any device that writes to memory directly can cause the data in memory to differ from what is stored in cache.
3. Virtual memory swaps memory pages in and out of physical memory based on locations being addressed. If a page is not in memory when an address associated with it is accessed, the page must be loaded, potentially displacing another page. Virtual memory works because of the principle of locality. Specifically, when memory is accessed, the likelihood of the next access being nearby is high. Thus, pulling a page from disk to memory due to a memory access not only retrieves the memory be accessed, but likely the next memory element being access. Any of the devices listed in the table could cause potential problems if it causes virtual memory to thrash, continuously swapping in and out pages from physical memory. This would happen if the locality principle is violated by the device. Careful design and sufficient physical memory will almost always solve this problem.

Question 5

1.



2.



when defined by lw	when defined by R-type
used in i1 => 2-cycle stall	used in i1 => forward
used in i2 => 1-cycle stall	used in i2 => forward
used in i3 => forward	used in i3 => forward

Question 6

1. Option 1, because Option 2 results in a *race condition*. If the *race condition* was not an issue, Option 1 would still be better because we would pay the overhead of forking and joining multiple threads only once, instead of each time within the outer loop (as in Option 2).

Question 7

1. This problem is a divide and conquer problem, but utilizes recursion to produce a very compact piece of code. When the number of cores is small, we spawn a thread for the computation of left in the MergeSort code, and spawn a thread for the computation of the right. If we consider this recursively, for m initial elements in the array, we can utilize 1 + 2 + 4 + 8 + 16 + ... $log_2(m)$ processors to obtain speed-up.
2. $log_2(m)$ is the largest value of Y for which we can obtain any speed-up without restructuring. But if we had m cores, we could perform sorting using a very different algorithm. For instance, if we have greater than m/2 cores, we can compare all pairs of data elements, swap the elements if the left element is greater than the right element, and then repeat this step m times. So this is one possible answer for the question. It is known as parallel comparison sort. Various comparison sort algorithms include odd-even sort and cocktail sort.

Question 8

1. Possible results

```
x = 2, y = 2, w = 1, z = 0
x = 2, y = 2, w = 3, z = 0
```

```
x = 2, y = 2, w = 5, z = 0
x = 2, y = 2, w = 1, z = 2
x = 2, y = 2, w = 3, z = 2
x = 2, y = 2, w = 5, z = 2
x = 2, y = 2, w = 1, z = 4
x = 2, y = 2, w = 3, z = 4
x = 3, y = 2, w = 5, z = 4
```

2. We could set synchronization instructions after each operation so that all cores see the same value on all nodes.