CSE 560 – Practice Problem Set 5 Solution

1. In this question, you will investigate how the compiler can increase the amount of ILP via the scheduling of instructions on a single-issue, in-order pipeline. Our code uses a simple loop that adds a scalar value to an array in memory. The source code (in C) looks like this:

```
for (i=1000; i > 0; i=i-1)
        x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. The first step is to translate the above code segment into assembly language. In the following code segment, r1 is initially the address of the element of the array with the highest address, and f2 contains the scalar value, s. Register r2 is pre-computed, so that 8(r2) is the last element to operate on. Straightforward assembly language code, not scheduled for the pipeline, looks like this:

```
(1)     loop:   load    f0, 0(r1)       ;f0 ← array element
(2)             addf    f4 ← f0, f2     ;add scalar in f2
(3)             store   0(r1), f4       ;store result
(4)             addi    r1 ← r1, #-8    ;decrement pointer 8 bytes (sizeof double)
(5)             bneq    r1, r2, loop    ;branch if r1 != r2
```

Assume floating point additions take 4 cycles, and the 5-stage pipeline has full bypassing paths available. Assume the branch predicts "not-taken" and miss-predicted branches flush the pipeline.

(a) Show the timing of this instruction sequence (i.e., draw a pipeline diagram) without any code transformations. How many clock cycles are required per iteration? For the entire code snippet?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
(1)	F	D	X	M	W													
(2)		F	D	d*	X1	X2	X3	X4	M	W								
(3)			F	p*	d*	d*	D	X	M	W								
(4)							F	D	X	M	W							
(5)								F	D	X	M	W						
(6)									F	D	f*							
(1)											F	D	X	M	W			

This code takes 10 cycles per iteration, or 10,000 cycles total (ignoring the pipeline fill time). Note that there isn't a structural hazard in clocks 9 and 10 because instruction (2) doesn't use the M stage and instruction (3) doesn't use the W stage. (This typically isn't allowed, however, as the pipeline registers would need to be doubled to support it.)

(b) Re-schedule the code (make sure it still performs the required computation) to diminish the time required per iteration. Show the timing of this revised instruction sequence. How many clock cycles are required per iteration? For the entire code snippet?

```
(1)     loop:   load    f0, 0(r1)       ;f0 ← array element
(2)             addi    r1 ← r1, #-8    ;decrement pointer 8 bytes (sizeof double)
(3)             addf    f4 ← f0, f2     ;add scalar in f2
(4)             store   8(r1), f4       ;store result (note address transformation)
(5)             bneq    r1, r2, loop    ;branch if r1 != r2
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
(1)	F	D	X	M	W													
(2)		F	D	X	M	W												
(3)			F	D	X1	X2	X3	X4	M	W								
(4)				F	D	d*	d*	X	M	W								
(5)					F	p*	p*	D	X	M	W							
(6)								F	D	f*								
(1)										F	D	X	M	W				

This code takes 9 cycles per iteration, or 9000 cycles total (ignoring the pipeline fill time).

(c) Unroll the loop 4 times (i.e., 4 copies of the original loop are computed each iteration). You may assume r1 is initially a multiple of 32, which means that the number of original loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the floating point registers (you may use additional registers as needed).

```
(1)     loop:   load    f0, 0(r1)       ;f0 ← array element
(2)             addf    f4 ← f0, f2     ;add scalar in f2
(3)             store   0(r1), f4       ;store result
(4)             load    f5, 8(r1)       ;f5 ← array element
(5)             addf    f6 ← f5, f2     ;add scalar in f2
(6)             store   8(r1), f6       ;store result
(7)             load    f7, 16(r1)      ;f7 ← array element
(8)             addf    f8 ← f7, f2     ;add scalar in f2
(9)             store   16(r1), f8      ;store result
(10)            load    f9, 24(r1)      ;f9 ← array element
(11)            addf    f10 ← f9, f2    ;add scalar in f2
(12)            store   24(r1), f10     ;store result
(13)            addi    r1 ← r1, #-32   ;decrement pointer 32 bytes (4 doubles)
(14)            bneq    r1, r2, loop    ;branch if r1 != r2
```

(d) Show the timing of the unrolled loop. How many clock cycles are required per iteration? For the entire code snippet? (Note: you can skip some columns in the middle of the diagram if they are simply repeating an earlier pattern.)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		25	26	27	28	29	30
(1)	F	D	X	M	W																		
(2)		F	D	d*	X1	X2	X3	X4	M	W													
(3)			F	p*	d*	d*	D	X	M	W													
(4)							F	D	X	M	W												
(5)								F	D	d*	X1	X2	X3	X4	M	W							
(6)									F	p*	d*	d*	D	X	M	W							
(12)																		D	X	M	W		
(13)																		F	D	X	M	W	
(14)																			F	D	X	M	W
(15)																				F	D	f*	
(1)																						F	D

Notice that instruction (3) finishes at time 10 and instruction (6) finishes at time 16, implying instruction (9) will finish at time 22 and instruction (12) will finish at time 28 (i.e., there are 6 cycles per store instruction within the iteration). Since instruction (1) starts in the second iteration at time 29, this loop requires 28 clock cycles per iteration, for a total time of 28 clocks/iteration X 250 iterations = 7000 clocks.

(e) Re-schedule the unrolled loop, show the timing of this re-scheduled loop. How many clock cycles are required per iteration? For the entire code snippet?

```
(1)     loop:   load    f0, 0(r1)       ;f0 ← array element
(2)             load    f5, 8(r1)       ;f5 ← array element
(3)             load    f7, 16(r1)      ;f7 ← array element
(4)             load    f9, 24(r1)      ;f9 ← array element
(5)             addf    f4 ← f0, f2     ;add scalar in f2 to f0
(6)             addf    f6 ← f5, f2     ;add scalar in f2 to f5
(7)             addf    f8 ← f7, f2     ;add scalar in f2 to f7
(8)             addf    f10 ← f9, f2    ;add scalar in f2 to f9
(9)             store   0(r1), f4       ;store result from f4
(10)            store   8(r1), f6       ;store result from f6
(11)            store   16(r1), f8      ;store result from f8
(12)            store   24(r1), f10     ;store result from f10
(13)            addi    r1 ← r1, #-32   ;decrement pointer 32 bytes (4 doubles)
(14)            bneq    r1, r2, loop    ;branch if r1 != r2
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(1)	F	D	X	M	W																			
(2)		F	D	X	M	W																		
(3)			F	D	X	M	W																	
(4)				F	D	X	M	W																
(5)					F	D	X1	X2	X3	X4	M	W												
(6)						F	D	X1	X2	X3	X4	M	W											
(7)							F	D	X1	X2	X3	X4	M	W										
(8)								F	D	X1	X2	X3	X4	M	W									
(9)									F	D	d*	d*	X	M	W									
(10)										F	p*	p*	D	X	M	W								
(11)													F	D	X	M	W							
(12)														F	D	X	M	W						
(13)															F	D	X	M	W					
(14)																F	D	X	M	W				
(15)																	F	D	f*					
(1)																			F	D	X	M	W	

Since instruction (1) starts in the second iteration at time 19, this loop requires 18 clock cycles per iteration, for a total time of 18 clocks/iteration X 250 iterations = 4500 clocks. The total time has decreased to 45% of its original value.

With 14 instructions, and a single-issue pipeline, the minimum iteration time possible is at least 14 clocks. Then, consider that 2 clocks are used in the miss-predicted branch, and therefore we are only 2 clocks longer than the absolute minimum. Note, however, that there is a 2 cycle stall in instruction (9) that causes (11) to be fetched 3 cycles later than (10). Maybe we can eliminate this by exploiting the ability to share the M and W stages during a clock cycle when one instruction is a floating point add and the other is a store.

Let's look at one more code transformation:

```
(1)     loop:   load    f0, 0(r1)       ;f0 ← array element
(2)             load    f5, 8(r1)       ;f5 ← array element
(3)             load    f7, 16(r1)      ;f7 ← array element
(4)             load    f9, 24(r1)      ;f9 ← array element
(5)             addf    f4 ← f0, f2     ;add scalar in f2 to f0
(6)             store   0(r1), f4       ;store result from f4
(7)             addf    f6 ← f5, f2     ;add scalar in f2 to f5
(8)             store   8(r1), f6       ;store result from f6
(9)             addf    f8 ← f7, f2     ;add scalar in f2 to f7
(10)            store   24(r1), f10     ;store result from f10
(11)            addf    f10 ← f9, f2    ;add scalar in f2 to f9
(12)            store   16(r1), f8      ;store result from f8
(13)            addi    r1 ← r1, #-32   ;decrement pointer 32 bytes (4 doubles)
(14)            bneq    r1, r2, loop    ;branch if r1 != r2
```

This results in the following diagram:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(1)	F	D	X	M	W																			
(2)		F	D	X	M	W																		
(3)			F	D	X	M	W																	
(4)				F	D	X	M	W																
(5)					F	D	X1	X2	X3	X4	M	W												
(6)						F	D	d*	d*	X	M	W												
(7)							F	D	X1	X2	X3	X4	M	W										
(8)								F	D	d*	d*	X	M	W										
(9)									F	D	X1	X2	X3	X4	M	W								
(10)										F	D	d*	d*	X	M	W								
(11)											F	D	X1	X2	X3	X4	M	W						
(12)												F	D	d*	d*	X	M	W						
(13)													F	p*	p*	D	X	M	W					
(14)																F	D	X	M	W				
(15)																	F	D	f*					
(1)																			F	D	X	M	W	

What we observe is that while we are able to overlap the use of the M and W stages, it doesn't save us any cycles. Completing instructions (6), (8), and (10) one clock cycle earlier because the structural hazard isn't really a hazard didn't save us anything, because delaying their completion by one clock wouldn't slow down the pipeline (the iteration took the same number of clocks due to the need to keep instructions retiring in order later in the iteration.

2. You are building a system around a processor with in-order execution that runs at 2.66 GHz and has a CPI of 0.7 excluding memory accesses. The only instructions that that read or write data from memory are loads (20% of all instructions) and stores (5% of all instructions).

   The memory system for this computer is composed of a split L1 cache that imposes no penalty on hits. Both the I-cache and the D-cache are direct mapped and hold 32 KB each. The I-cache has a 2% miss rate and 32-byte blocks, and the D-cache is write through with a 5% miss rate and 16-byte blocks. There is a write buffer on the D-cache that eliminates stalls for 95% of all writes (i.e., there is no stall for a hit, even though the cache is write through).

   The 512 KB write-back, unified L2 cache has 64-byte blocks and an access time of 15 ns. Of all memory references sent to the L2 cache in this system, 80% are satisfied without going to main memory.

   The main memory has an access latency of 60 ns, after which any number of bus words may be transferred at the rate of one per cycle on the 128-bit-wide 133 MHz main memory bus.

   (a) What is the average memory access time for instruction accesses?

   So as to not get lost in the notation, let's first define a few symbols.

Symbol	Meaning	Value (if given in problem statement)
$t_{HIT,I\$}$	Time (penalty) for hit in I$	0 clocks
$t_{MISS,I\$}$	Time for miss in I$	
$t_{HIT,D\$}$	Time (penalty) for hit in D$	0 clocks
$t_{MISS,D\$}$	Time for miss in D$	
$t_{HIT,L2}$	Time for hit in L2	15 ns = 40 clocks
$t_{MISS,L2}$	Time for miss in L2	
$t_{HIT,M}$	Time for hit in main memory	
$\%miss_{I\$}$	Miss rate for I$	0.02
$\%miss_{D\$}$	Miss rate for D$	0.05
$\%miss_{L2}$	Miss rate for L2	0.20
$t_{AVG,I\$}$	Average time for I$	
$t_{AVG,D\$}$	Average time for D$	
$t_{AVG,L2}$	Average time for L2	

   This problem is all about repeated use of the general form $t_{AVG} = t_{HIT} + \%miss \times t_{MISS}$.

   The values needed to compute $t_{AVG,I\$}$ and that aren't in the table above are: $t_{MISS,I\$}$, $t_{MISS,L2}$, and $t_{HIT,M}$. Starting at the bottom of the memory hierarchy and moving up, we can compute the value of $t_{HIT,M}$ by adding the access time to the transfer time:

   60 ns = 160 clocks

   64 bytes X 8 bits/byte / 128 bits/transfer = 4 bus transfers/access

   2.66 GHz processor freq. / 133 MHz bus freq. = 20 processor clocks per transfer

   4 transfers X 20 clocks/transfer = 80 clocks to transfer data from main memory

$t_{HIT,M}$ = access time + transfer time = 160 + 80 = 240 clocks

and

$t_{MISS,L2} = t_{HIT,M}$ = 240 clocks

We can now apply the general expression for t_{AVG} on L2.

$t_{AVG,L2} = t_{HIT,L2} + \%miss_{L2}$ X $t_{MISS,L2}$

= 40 clocks + 0.20 X 240 clocks

= 88 clocks

Next, we observe that $t_{MISS,I\$} = t_{AVG,L2}$, and we are ready to apply the general expression again, this time for t_{AVG} on the I$.

$t_{AVG,I\$} = t_{HIT,I\$} + \%miss_{I\$}$ X $t_{MISS,I\$}$

= 0 + 0.02 X 88 clocks

= 1.76 clocks

(b) What is the average memory access time for data reads?

We have everything we need to directly apply the general expression for t_{AVG} on the D$.

$t_{AVG,D\$} = t_{HIT,D\$} + \%miss_{D\$}$ X $t_{MISS,D\$}$

$= t_{HIT,D\$} + \%miss_{D\$}$ X $t_{AVG,L2}$

= 0 + 0.05 X 88 clocks

= 4.4 clocks

(c) What is the average memory access time for data writes?

There is no reason given for writes to have a different access time than reads, given the statements made in the problem description.

(d) What is the overall CPI, including memory accesses?

The overall CPI is the base CPI + instruction cache CPI + data cache CPI due to loads + data cache CPI due to stores.

CPI = 0.7 + 1.76 + (0.20)(4.4) + (0.05)(4.4) = 3.56

Memory access took a 0.7 CPI system and turned it into a 3.56 CPI system. Clearly, some improvement in the memory system is warranted here, maybe an L3 cache?

3. Let's try to show how one can make *unfair* benchmarks. Here are two machines with the same processor and main memory but different cache organizations. Assume that both processors run at 2 GHz, have a CPI of 1, and have a cache (read) miss time of 100 ns. Further, assume that writing a 32-bit word to main memory requires 100 ns (for the write-through cache), and that writing a 32-byte block requires 200 ns (for the write-back cache). The caches are unified – they contain both instructions and data, and each cache has a total capacity of 64 KB, not including tags and status bits.

   The cache on system A is two-way set associative and has 32-byte blocks. It is write through and does not allocate a block on a write miss.

   The cache on system B is direct mapped and has 32-byte blocks. It is write back and allocates a block on a write miss.

   (f) Describe a program that makes system A run as fast as possible relative to system B's speed.

   Consider the code blurb below:

```
foo:    beqz    r0, bar   ; branch if r0 == 0
        .
        .
        .
bar:    beqz    r0, foo   ; branch if r0 == 0
```

   We make two assumptions in this code. First, the value of r0 is zero; second, locations foo and bar map into the same set in both caches.

   On cache A, this code only causes two compulsory misses to load the two instructions into the cache. After that, all accesses generated by the code hit in the cache. For cache B, all the accesses miss the cache because a direct-mapped cache can only store one block in each set.

   This is a good example of where a victim cache could help. Keep in mind that in this example, the information that cache B misses on is always recently resident.

   (g) Describe a program that makes system B run as fast as possible relative to system A's speed.

   Consider the code blurb below:

```
baz:    store   0(r1), r0       ; store r0
qux:    beqz    r0, baz         ; branch if r0 == 0
```

   We make two assumptions here. First, locations baz and qux and the location pointed to by 0(r1) map to different sets within the caches and are all initially resident; second r0 is zero.

This code illustrates the main trust of a program that makes a system with cache B outperform a system with cache A, that is, one that repeatedly writes to a location that is resident in cache. Each time the store instruction executes on cache A, the data stored are written to memory because cache A is write through. For cache B, the store instruction always finds the appropriate block in the cache (as we assume the data at location 0(r1) are resident in the cache) and updates only the cache block, as the cache is write back; the block is not written to main memory until it is replaced.

(h) How much faster is the program in part (a) on system A as compared to system B?

With all access hits, cache A allows the processor to maintain a CPI of 1. Cache B misses each access at a cost of 100 ns, or 200 clock cycles. Thus cache B allows its processor to achieve CPI = 200. Cache A offers a speedup of 200 over cache B.

(i) How much faster is the program in part (b) on system B as compared to system A?

In the steady state, cache B hits on every write, and so maintains CPI = 1. Cache A writes to memory on each store, consuming an extra 100 ns each time. Cache B allows the processor to complete one iteration in 2 clocks. With cache A the processor needs 202 clocks per iteration. Cache B offers a speedup of 101 over cache A.

If we alter cache A so that it is a lockup free cache, then cache A's processor can overlap the writes to memory with further processing. This gives 200 clocks per iteration, and a speedup of 100 for cache B over cache A. Clearly, a lockup free cache doesn't do much for us here.