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 and 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. Rename this instruction sequence:

mul r4, r5 $\rightarrow$ r1
add r1, r2 $\rightarrow$ r3

Map table	
r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Free-list
p6
p7
p8
p9
p10

mul p4, p5 $\rightarrow$ p6
add r1, r2 $\rightarrow$ r3

Map table	
r1	p6
r2	p2
r3	p3
r4	p4
r5	p5

mul p4, p5 $\rightarrow$ p6
add p6, p2 $\rightarrow$ p7

Map table	
r1	p6
r2	p2
r3	p7
r4	p4
r5	p5

Free-list
p8
p9
p10

3. Dispatch this instruction:

div p7, p6 → p1

Insn	**Inp1**	**R**	**Inp2**	**R**	**Dst**	**Age**

Ready bits	
p1	y
p2	y
p3	y
p4	y
p5	y
p6	n
p7	y
p8	y
p9	y

Insn	**Inp1**	**R**	**Inp2**	**R**	**Dst**	**Age**
div	p7	y	p6	n	p1	0

4. Determine which of the following instructions are ready.

Insn	Inp1	R	Inp2	R	Dst	Age
add	p3	y	p1	y	p2	0
mul	p2	n	p4	y	p5	1
div	p1	y	p5	n	p6	2
xor	p4	y	p1	y	p9	3

The add and xor instruction are ready to issue.

(f) Which will be issued on a 1-wide machine?

The add instruction will issue, because it has the smallest age (i.e., it is the oldest).

(g) Which will be issued on a 2-wide machine?

Both the add and the xor will issue on a 2-wide machine.

(h) What information will change if we issue the instruction from part (a)?

After the add instruction issues, the table will be revised to look as follows:

Insn	Inp1	R	Inp2	R	Dst	Age
mul	p2	y	p4	y	p5	1
div	p1	y	p5	n	p6	2
xor	p4	y	p1	y	p9	3

and the mul instruction is ready to issue (this assumes the add latency is just one clock).

5. Using the revised pipeline diagrams presented in class, show the execution of the following instruction sequence:

div	r2 ← r3, r5
add	r1 ← r2, r4
mul	r4 ← r6, r6

You should assume that the execution units for the three instructions are as follows:

4 clocks for an add
10 clocks for a multiply
20 clocks for a divide

Start by showing the instructions after renaming, and then show the pipeline diagram for a dual-issue processor.

Assuming the map table starts out as below:

Map table	
r1	p1
r2	p2
r3	p3
r4	p4
r5	p5
r6	p6

The renamed registers are as follows:

div	p7 $\leftarrow$ p3, p5
add	p8 $\leftarrow$ p7, p4
mul	p9 $\leftarrow$ p6, p6

and the pipeline diagram becomes:

Instruction	**Disp**	**Issue**	**WB**	**Commit**
div p7 $\leftarrow$ p3, p5	1	2	22	23
add p8 $\leftarrow$ p7, p4	1	22	26	27
mul p9 $\leftarrow$ p6, p6	2	3	13	27