CS152
Computer Architecture and Engineering
SOLUTIONS
Caches and the Memory Hierarchy
Problem Set #3 Solutions

Assigned 10/11/16 *Due October 18*

http://inst.eecs.berkeley.edu/~cs152/fa16

Problem 1: Superscalar Processor

Problem 1.A

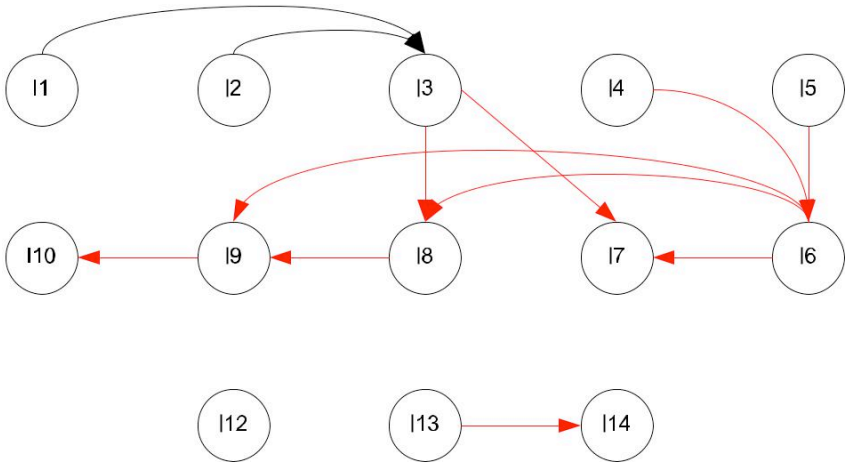
Fill in the renaming tags in the following two tables for the execution of instructions I1 to I10

Instr #	Instruction	Dest	Src1	Src2
I1	LD F2, 0(x2)	T1	x2	0
I2	LD F3, 0(x3)	T2	x3	0
I3	FMUL F4, F2, F3	T3	T1	T2
I4	LD F2, 4(x2)	T4	x2	4
I5	LD F3, 4(x3)	T5	x3	4
I6	FMUL F5, F2, F3	T6	T4	T5
I7	FMUL F6, F4, F5	T7	T3	T6
I8	FADD F4, F4, F5	T8	T3	T6
I9	FMUL F6, F4, F5	T9	T8	T6
I10	FADD F1, F1, F6	T10	F1	T9

Renaming table

	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10
x2										
x3										
F1										T10
F2	T1			T4						
F3		T2			T5					
F4			T3					T8		
F5						T6				
F6							T7		T9	

Problem 1.B



Problem 1.C

See the following table.

Slot	Instruction	Cycle instruction entered ROB	Argument 1		Argument 2		dst	*Cycle dispatched*	*Cycle written back to ROB*
			src1	cycle available	Src2	cycle available	dst reg		
T1	`LD F2, 0(x2)`	1	C	1	x2	1	F2	2	6
T2	`LD F3, 0(x3)`	1	C	1	x3	1	F3	3	7
T3	`FMUL F4, F2, F3`	2	F2	6	F3	7	F4	8	12
T4	`LD F2, 4(x2)`	2	C	2	x2	2	F2	4	8
T5	`LD F3, 4(x3)`	3	C	3	x3	3	F3	5	9
T6	`FMUL F5, F2, F3`	3	F2	8	F3	9	F5	10	14
T7	`FMUL F6, F4, F5`	4	F4	12	F5	14	F6	15	19
T8	`FADD F4, F4, F5`	4	F4	12	F5	14	F4	15	18
T9	`FMUL F6, F4, F5`	5	F4	18	F5	14	F6	19	23
T10	`FADD F1, F1, F6`	5	F1	5	F6	23	F1	24	27
T11	`ADD x2, x2, 8`	6	x2	6	C	6	x2	7	9
T12	`ADD x3, x3, 8`	6	x3	6	C	6	x3	8	10
T13	`ADD x4, x4, -1`	7	x4	7	C	7	x4	9	11
T14	`BNE x4, x0, loop`	7	x4	11	C	Loop			
T15	`LD F2, 0(x2)`	8	C	8	x2	9	F2	10	14
T16	`LD F3, 0(x3)`	8	C	8	x3	10	F3	11	15
T17	`FMUL F4, F2, F3`	9	F2	14	F3	15	F4	16	20
T18	`LD F2, 4(x2)`	9	C	9	x2	9	F2	12	16
T19	`LD F3, 4(x3)`	10	C	10	x3	10	F3	13	17
T20	`FMUL F5, F2, F3`	10	F2	16	F3	17	F5	18	22
T21	`FMUL F6, F4, F5`	11	F4	20	F5	22	F6	23	27
T22	`FADD F4, F4, F5`	11	F4	20	F5	22	F4	23	26
T23	`FMUL F6, F4, F5`	12	F4	26	F5	22	F6	27	31
T24	`FADD F1, F1, F6`	12	F1	27	F6	31	F1	32	35
T25	`ADD x2, x2, 8`	13	x2	13	C	13	x2	14	16
T26	`ADD x3, x3, 8`	13	x3	13	C	13	x3	15	17
T27	`ADD x4, x4, -1`	14	x4	14	C	14	x4	16	18
T28	`BNE x4, x0, loop`	14			C	Loop			
T29									

Problem 1.D

I5, I6, I7, I8, I9, I10 (see registers in blue in previous table)

27 cycles.

Problem 1.E

The behavior should repeat- should be obvious from the dependency graph (DAG) in Problem 3.1.D.

Problem 1.F

Yes/No ----------

An extra FP multiplier does not really help, because All FMUL instructions execute as soon as operands are ready. But an extra memory port helps, because dispatch of I4, I5 was delayed waiting for memory port.

Problem 1.G

The answer is 4 cycles.

Since the integer index/counter additions are relatively short, they can proceed to generate values for different loop iterations and load all values from memory saving them to renamed registers. After a large number of iterations, many iterations of the loop will be running in parallel. With each iteration depending only on the previous iteration for the FADD on F1 value. Hence, the number of cycles is the latency of FADD (3 + 1 cycle for writeback). At steady state, one iteration can complete every 4 cycles.

Problem 2: Register Renaming and Static vs. Dynamic Scheduling

Problem 2.A **Simple Pipeline**

The following table shows the cycles in which instructions are decoded, issued, and written back. It starts with cycle 0 in which the first load has been decoded (and thus has just entered the issue stage). It is assumed that all instructions prior to the first load have already been completed. Although not shown below, there is a buffer that holds instructions that are waiting in the issue stage. Since there is no bypassing, an instruction must complete the write-back stage before a dependent instruction can issue. For example, as shown in the table, the second load is issued in cycle 2, executes for 2 cycles, and is written back in cycle 4. Thus, any instruction that depends on the load can issue no earlier than cycle 5.

Cycle	Decoded Instruction (Enters Issue)	Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
0	`L.S F0, 0(x1)`	Stall	
1	`L.S F1, 0(x2)`	`L.S F0, 0(x1)`	3
2	`MUL.S F0, F0, F1`	`L.S F1, 0(x2)`	4
3	`L.S F2, 0(x3)`	Stall	
4	`L.S F3, 0(x4)`	Stall	
5	`MUL.S F2, F2, F3`	`MUL.S F0, F0, F1`	9
6	`ADD.S F0, F0, F2`	`L.S F2, 0(x3)`	8
7	`S.S F0, 0(x5)`	`L.S F3, 0(x4)`	9
8		Stall	
9		Stall	
10		`MUL.S F2, F2, F3`	14
11		Stall	
12		Stall	
13		Stall	
14		Stall	
15		`ADD.S F0, F0, F2`	17
16		Stall	
17		Stall	
18		`S.S F0, 0(x5)`	

The number of cycles from the issue of the first load instruction (in cycle 1) until the issue of the final store instruction (in cycle 18) is 18 cycles, inclusive.

Problem 2.B **Static Scheduling**

The new code sequence is given below. Originally there were two stall cycles after the second load instruction. Now these cycles will be filled by the third and fourth load instructions. The remaining instructions cannot be reordered due to data dependencies (except for the two multiply instructions, although doing that would hurt performance).

```
L.S       F0, 0(R1)
L.S       F1, 0(R2)
L.S       F2, 0(R3)
L.S       F3, 0(R4)
MUL.S     F0, F0, F1
MUL.S     F2, F2, F3
ADD.S     F0, F0, F2
S.S       F0, 0(R5)
```

The following table shows the cycles in which the instructions in the above sequence are decoded, issued, and written back.

Cycle	Decoded Instruction (Enters Issue)	Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
0	L.S F0, 0(R1)	Stall	
1	L.S F1, 0(R2)	L.S F0, 0(R1)	3
2	L.S F2, 0(R3	L.S F1, 0(R2)	4
3	L.S F3, 0(R4)	L.S F2, 0(R3)	5
4	MUL.S F0, F0, F1	L.S F3, 0(R4)	6
5	MUL.S F2, F2, F3	MUL.S F0, F0, F1	9
6	ADD.S F0, F0, F2	Stall	
7	S.S F0, 0(R5)	MUL.S F2, F2, F3	11
8		Stall	
9		Stall	
10		Stall	
11		Stall	
12		ADD.S F0, F0, F2	14
13		Stall	
14		Stall	
15		S.S F0, 0(R5)	

The number of cycles from the issue of the first load instruction (in cycle 1) to the issue of the final store instruction (in cycle 15) is 15 cycles, inclusive. Using static scheduling has enabled us to reduce the execution time of the sequence by 17%.

Problem 2.C **Fewer Registers**

The new code sequence using only two floating-point registers is shown below. It is assumed that R6 holds the address of a memory location that can be used to store temporary values.

```
L.S       F0, 0(R1)
L.S       F1, 0(R2)
MUL.S     F0, F0, F1
L.S       F1, 0(R3)
S.S       F0, 0(R6)
L.S       F0, 0(R4)
MUL.S     F0, F0, F1
L.S       F1, 0(R6)
ADD.S     F0, F0, F1
S.S       F0, 0(R5)
```

The following table shows the cycles in which the instructions in the above sequence are decoded, issued, and written back. For this problem, a store instruction is needed in the middle of the instruction sequence in order to spill a register. Although not explicitly stated in the problem, stores have the same latency as loads (two cycles), since they use the same functional unit. However, if you assume a different latency in your solutions, that is acceptable as long as you state it. Because the result of the store is not needed for several cycles after it completes (when the load restores the spilled value), it would take a very long latency for store instructions in order to delay the last load. We don't have to worry about WAR hazards in the above sequence because instructions are issued in-order. Note that we can no longer execute the four original loads in sequence as in 3.2.B because of the lack of available registers. We can, however, execute the third load before saving the intermediate value from the first MUL instruction.

Cycle	Decoded Instruction (Enters Issue)	Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
0	L.S F0, 0(R1)	Stall	
1	L.S F1, 0(R2)	L.S F0, 0(R1)	3
2	MUL.S F0, F0, F1	L.S F1, 0(R2)	4
3	L.S F1, 0(R3)	Stall	
4	S.S F0, 0(R6)	Stall	
5	L.S F0, 0(R4)	MUL.S F0, F0, F1	9
6	MUL.S F0, F0, F1	L.S F1, 0(R3)	8
7	L.S F1, 0(R6)	Stall	
8	ADD.S F0, F0, F1	Stall	
9	S.S F0, 0(R5)	Stall	
10		S.S F0, 0(R6)	
11		L.S F0, 0(R4)	13
12		Stall	
13		Stall	
14		MUL.S F0, F0, F1	18
15		L.S F1, 0(R6)	17
16		Stall	
17		Stall	
18		Stall	
19		ADD.S F0, F0, F1	21
20		Stall	
21		Stall	
22		S.S F0, 0(R5)	

The number of cycles from the issue of the first load instruction (in cycle 1) to the issue of the final store instruction (in cycle 22) is 22 cycles, inclusive. Using only two floating-point registers results in a severe performance hit.

Problem 2.D **Register renaming and dynamic scheduling**

The table below shows the cycles in which the instructions in the original code sequence are decoded, issued, and written back on the single-issue machine with register renaming and out-of-order issue. The table also contains the rename table for the architectural registers.

Cycle	Decoded/Renamed Instruction (Enters Issue)	Rename				Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
		F0	F1	F2	F3		
0	L.S T0, 0(R1)	T0				Stall	
1	L.S T1, 0(R2)	T0	T1			L.S T0, 0(R1)	3
2	MUL.S T2, T0, T1	T2	T1			L.S T1, 0(R2)	4
3	L.S T3, 0(R3)	T2	T1	T3		Stall	
4	L.S T4, 0(R4)	T2	T1	T3	T4	L.S T3, 0(R3)	6
5	MUL.S T5, T3, T4	T2	T1	T5	T4	MUL.S T2, T0, T1	9
6	ADD.S T6, T2, T5	T6	T1	T5	T4	L.S T4, 0(R4)	8
7	S.S T6, 0(R5)	T6	T1	T5	T4	Stall	
8						Stall	
9						MUL.S T5, T3, T4	13
10						Stall	
11						Stall	
12						Stall	
13						Stall	
14						ADD.S T6, T2, T5	16
15						Stall	
16						Stall	
17						S.S T6, 0(R5)	

The number of cycles from the issue of the first load instruction (in cycle 1) to the issue of the final store instruction (in cycle 17) is 17 cycles, inclusive. This is one cycle better than executing this code on an in-order machine but not quite as good as the performance of the optimized code in 3.2.B, which only required 15 cycles. The difference in performance between the statically scheduled code and the dynamically scheduled code can be attributed to the fact that only a single instruction can be decoded at a time, which limits the hardware's ability to find independent instructions to issue. The optimized version of the code from 3.2.B executing on this machine would not improve in performance over executing on an in-order machine – it would still take 15 cycles.

Note, that in cycle 5, we would get better performance if we issued the final load instruction rather than the MUL instruction. The machine doesn't know that, so it issues the instruction that entered the ROB first.

Problem 2.E **Effect of Register Spills**

The table below shows the cycles in which the instructions in the original code sequence are decoded, issued, and written back on the single-issue machine with register renaming and out-of-order issue.

Cycle	Decoded/Renamed Instruction (Enters Issue)	Rename		Issued Instruction (Enters Execute)	WB Cycle For Issued Instruction
		F0	F1		
0	L.S T0, 0(R1)	T0		Stall	
1	L.S T1, 0(R2)	T0	T1	L.S T0, 0(R1)	3
2	MUL.S T2, T0, T1	T2	T1	L.S T1, 0(R2)	4
3	L.S T3, 0(R3)	T2	T3	Stall	
4	S.S T2, 0(R6)	T2	T3	L.S T3, 0(R3)	6
5	L.S T4, 0(R4)	T4	T3	MUL.S T2, T0, T1	9
6	MUL.S T5, T4, T3	T5	T3	Stall	
7	L.S T6, 0(R6)	T5	T6	Stall	
8	ADD.S T7, T5, T6	T7	T6	Stall	
9	S.S T7, 0(R5)	T7	T6	Stall	
10				S.S T2, 0(R6)	12
11				L.S T4, 0(R4)	13
12				L.S T6, 0(R6)	14
13				Stall	
14				MUL.S T5, T4, T3	18
15				Stall	
16				Stall	
17				Stall	
18				Stall	
19				ADD.S T7, T5, T6	21
20				Stall	
21				Stall	
22				S.S T7, 0(R5)	24

It now takes 22 cycles between issue of the first load instruction and issue of the last store instruction. That is the same performance as 3.2.C, and much worse than 3.2.D.

We managed to execute two instructions out of order, but we still couldn't beat the in-order performance. The problem lies with the fact that we had to wait for the first store to issue before we could continue with the program. This is directly linked to having only two registers, thus having to store intermediate values.

Problem 3: Importance of Features

For the following snippets of code, select the single architectural feature that will *most* improve the performance of the code. Explain your choice, including description of why the other features will not improve performance as much and your assumptions about the machine design. The features you have to choose from are: out-of-order issue with renaming, branch prediction, and superscalar execution. Loads are marked whether they hit or miss in the cache.

Problem 3.A

```
ADD.D F0, F1, F8
ADD.D F2, F3, F8
ADD.D F4, F5, F8

ADD.D F6, F7, F8
```

Circle one:

- ☐ Out-of-Order Issue with Renaming
- ☐ Branch Prediction
- ☐ Superscalar

Superscalar because the instructions have no dependencies so they could be issued in parallel (superscalar could deliver speedup).

The other two techniques will waste hardware because they will offer no speedup. Out-of-Order with renaming will not help because there are no dependencies. Branch prediction is useless since there are no branches.

Problem 3.B

```
loop: ADD R3 R4 R0
      LD R4, 8(R4)

      BNEQZ R4, LOOP
```

cache hit

Circle one:

- ☐ Out-of-Order Issue with Renaming
- ☐ Branch Prediction
- ☐ Superscalar

Branch prediction is necessary with a tight loop to prevent bubbles in the pipeline.

Superscalars will be limited not only by the branches, but also the WAR and RAW hazards. They will limit how much ILP it can achieve.

Out-of-order with renaming will be limited by the branches. Renaming will take care of the WAR hazard, but the RAW hazard will still limit how much improvement is possible.

Problem 3.C

```
LD   R1   0(R2)  # cache miss
ADD  R2  R1   R1
LD   R1   0(R3)  # cache hit
LD   R3   0(R4)  # cache hit
ADD  R3  R1   R3
ADD  R1  R2   R3
```

Out-of-order with renaming will let the third, fourth, and fifth instruction run while the cache miss is being handled. When the miss completes it only needs to do the second and sixth instruction.

Branch prediction won't help with getting around the cache miss and there are enough hazards to limit a superscalar.

Circle one:

- Out-of-Order Issue with Renaming
- Branch Prediction
- Superscalar

Problem 4: Out-of-order Machine Design

An out-of-order superscalar processor uses a unified physical register file for register renaming and also separates the reorder buffer from the instruction window. During decode, instructions are allocated a slot in the reorder buffer, have their registers renamed, and then are placed in the instruction window to await issue into execution.

Part A) Should the number of **reorder buffer entries** be greater than, equal to, or less than the number of **instruction window entries**? Explain.

GREATER THAN
the ROB is tracking all in-flight instructions, whereas the instruction window is holding only instructions that have been decoded/renamed but not issued.

Part B) Should the number of **reorder buffer entries** be greater than, equal to, or less than the number of **physical registers**? Explain.

GREATER THAN
the ROB has an entry for all in-flight instructions, but not all instructions write to a destination (stores, branches), thus not all entries need a physical register allocated for them, so you can get away with having fewer physical registers than ROB entries.

Part C) Should the number of **instruction window entries** be greater than, equal to, or less than the number of **physical registers**? Explain.

LESS THAN
Since this is a unified physical register file design, the instruction window is holding instructions that have been decoded/renamed but not issued. However, nearly all instructions moving through the machine need a physical register to write to, so you would need fewer IW entries as you would need physical registers