Chapter 5

Problem 1 through Problem 6

The code below steps through the elements of two arrays (A[] and B[]) concurrently, and for each element, it puts the larger of the two values into the corresponding element of a third array (C[]). The three arrays are of length N.

The instruction set used in this problem is as follows:

add	rd, rs, rt	rd ← rs + rt
addi	rd, rs, imm	rd ← rs + imm
lw	rd, offset(base)	rd ← MEM[offset+base] (offset = imm, base = reg)
sw	rs, offset(base)	MEM[offset+base] ← rs (offset = imm, base = reg)
bge	rs, rt, address	if (rs >= rt) PC ← address
blt	rs, rt, address	if (rs < rt) PC ← address
b	address	PC ← address

NOTE: r0 is hardwired to 0

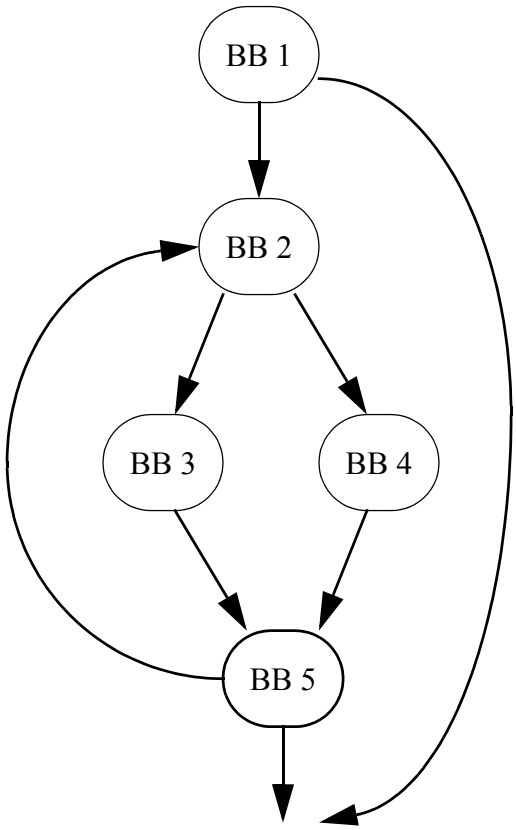
Benchmark Code:

Static Inst#	Label	Assembly_Instruction			
	main:				
1		addi	r2,	r0,	A
2		addi	r3,	r0,	B
3		addi	r4,	r0,	C
4		addi	r5,	r0,	N
5		add	r10,	r0,	r0
6		bge	r10,	r5,	end
	loop:				
7		lw	r20,	0(r2)	
8		lw	r21,	0(r3)	
9		bge	r20,	r21,	T1
10		sw	r21,	0(r4)	
11		b	T2		
	T1:				
12		sw	r20,	0(r4)	
	T2:				
13		addi	r10,	r10,	1
14		addi	r2,	r2,	4
15		addi	r3,	r3,	4
16		addi	r4,	r4,	4
17		blt	r10,	r5,	loop
	end:				

1.Identify the basic blocks of this benchmark code by listing the static instructions belonging to each basic block in the following table. Number the basic blocks based on the lexical ordering of the code. *Note*: There may be more boxes than there are basic blocks.

BB#	1	2	3	4	5	6	7	8	9
Instr. #s	1-6	7-9	10-11	12	13-17				

2. Draw the control flow graph for this benchmark.



3. Now generate the instruction execution trace (i.e., the sequence of basic blocks executed). Use the following arrays as input to the program, and trace the code execution by recording the number of each basic block that is executed.

N = 5;
A[] = {8, 3, 2, 5, 9};
B[] = {4, 9, 8, 5, 1};

TRACE: 1, 2, 4, 5, 2, 3, 5, 2, 3, 5, 2, 4, 5, 2 ,4 5

4. Fill in the following two tables based on the data you generated above.

Table 4: Instruction Mix

	Static		Dynamic	
Instr. Class	Number	%	Number	%
ALU	9	53	25	47
Load/Store	4	24	15	28
Branch	4	24	13	25

Note for Table 2: Count unconditional branches as taken branches.

Table 5: Basic Block / Branch Data

	Static	Dynamic
Average BB size (# inst.)	3.4	3.3
Number of Taken Branches		9
Number of Not-Taken Branches		4

5. Given the branch profile information you collected above, rearrange the basic blocks and reverse the sense of the branches in the program snippet to minimize the number of taken branches and to pack the code so that the frequently-executed paths are placed together. Show the new program.

   Solution not provided.

6. Given the new program you wrote in Problem 5, recompute the branch statistics in the last two rows of Table 5.

   Solution not provided.

Problem 7 through Problem 13

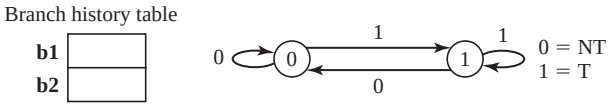
Consider the following code segment within a loop body for problems 5:

```
if (x is even) then                  ←(branch b1)
      increment a                     ←(b1 taken)
if (x is a multiple of 10) then      ←(branch b2)
      increment b                     ←(b2 taken)
```

Assume that the following list of 9 values of x is to be processed by 9 iterations of this loop.

8, 9, 10, 11, 12, 20, 29, 30, 31

Note: assume that predictor entries are updated by each dynamic branch before the next dynamic branch accesses the predictor (i.e., there is no update delay).



7. Assume that an one-bit (history bit) state machine (see above) is used as the prediction algorithm for predicting the execution of the two branches in this loop. Indicate the predicted and actual branch directions of the b1 and b2 branch instructions for each iteration of this loop. Assume initial state of 0, i.e., NT, for the predictor.

	8	9	10	11	12	20	29	30	31
b1 predicted:	N	T	N	T	N	T	T	N	T
b1 actual:	T	N	T	N	T	T	N	T	N
b2 predicted:	N	N	N	T	N	N	T	N	T
b2 actual:	N	N	T	N	N	T	N	T	N

8. What are the prediction accuracies for b1 and b2?

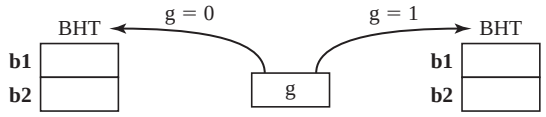
   b1: 1/9 = 11%

   b2: 3/9 = 33%

9. What is the overall prediction accuracy?

   Overall: 4/18 = 22%

10. Assume a two-level branch prediction scheme is used. In addition to the one-bit predictor, a one-bit global register (g) is used. Register g stores the direction of the last branch executed (which may not be the same branch as the branch currently being predicted) and is used to index into two separate one-bit branch history tables (BHTs) as shown below.



Depending on the value of g, one of the two BHTs is selected and used to do the normal one-bit prediction. Again, fill in the predicted and actual branch directions of b1 and b2 for nine iterations of the loop. Assume the initial value of g = 0, i.e., NT. For each prediction, depending on the current value of g, only one of the two BHTs is accessed and updated. Hence, some of the entries below should be empty.

Note: assume that predictor entries are updated by each dynamic branch before the next dynamic branch accesses the predictor (i.e. there is no update delay).

	8	9	10	11	12	20	29	30	31
For g=0									
b1 predicted:	__N	___T	___N	_____	__T	___T	_____	__T	_____
b1 actual:	__T	___N	___T	___N	___T	___T	___N	___T	___N__
b2 predicted:	____	__N	_____	__N	_____	____	__N	_____	__N__
b2 actual:	__N	___N	___T	___N	___N	___T	___N	___T	___N__
For g=1									
b1 predicted:	____	____	____	__N	_____	____	__N	_____	__N__
b1 actual:	__T	___N	___T	___N	___T	___T	___N	___T	___N__
b2 predicted:	__N	_____	__N	_____	__T	___N	_____	__T	_____
b2 actual:	__N	___N	___T	___N	___N	___T	___N	___T	___N__

11. What are the prediction accuracies for b1 and b2?

    b1: 6/9 =67%

    b2: 6/9 = 67%

12. What is the overall prediction accuracy?

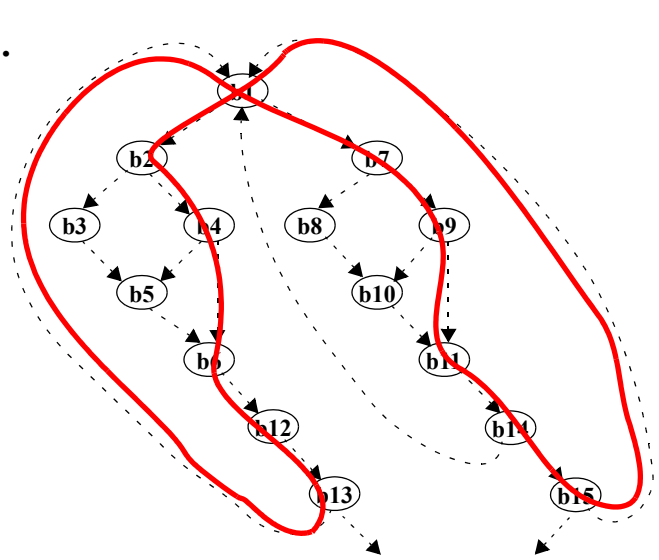
    Overall: 67%

13. What is the prediction accuracy of b2 when g = 0? Explain why.

    100%. Whenever b1 is not taken (i.e. g=0), the number being checked is odd (not even). It follows that the number is also not evenly divisible by ten. Hence, in these cases, b2 is always not taken and the predictor is able to predict b2 with high accuracy in this global context.

14. Below is the control flow graph of a simple program. The CFG is annotated with three different execution trace paths. For each execution trace circle which branch predictor (bimodal, local, or Gselect) will *best* predict the branching behavior of the given trace. More than one predictor may perform equally well on a particular trace. However, you are to use each of the three predictors *exactly once* in choosing the best predictors for the three traces. *Circle your choice* for each of the three traces and add. (*Assume each trace is executed many times and every node in the CFG is a conditional branch. The branch history register for the* local*,* global*, and* Gselect *predictors is limited to 4 bits.)*

1.



Circle one:

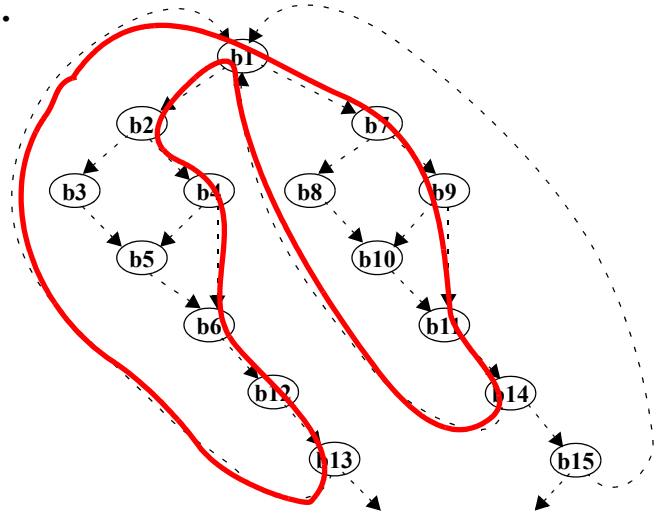
Bimodal

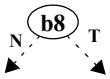
Local

Gselect

Identical global history at **b13** and **b15**, so the PC is need to differentiate them.

2.





Circle one:

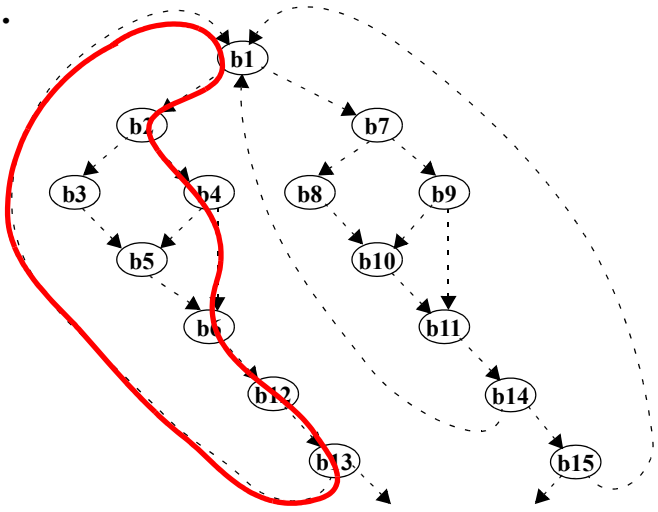
Bimodal

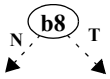
Local

Gselect

Identical global history at **b1**, so global history doesn't work. The local history of **b1** shows it alternates taken and not taken.

3.





Circle one:

Bimodal

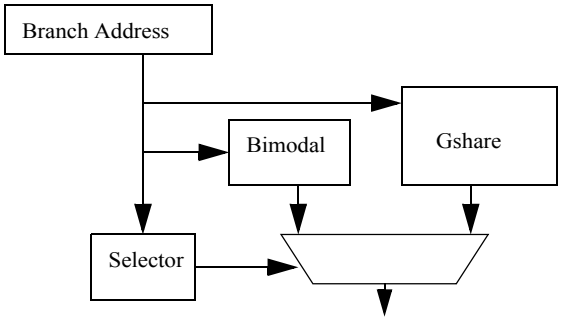
Local

Gselect

All the branches in this trace have a constant behavior, so bimodal predicts well.

Problem 15 and Problem 16: Combining Branch Prediction

Given a combining branch predictor with 2-entry direct-mapped bimodal branch direction predictor, a gshare predictor with a 1-bit BHR and 2 PHT entries, and a 2-entry selector table, simulate a sequence of taken and not-taken branches as shown in the rows of the table Problem 15, record the prediction made by the predictor before the branch is resolved as well as any change to the predictor entries after the branch resolves.



Use the following assumptions:

- Instructions are a fixed 4 bytes long; hence the two low-order bits of the branch address should be shifted out when indexing into the predictor. Use the next lowest-order bit to index into the predictor
- Each predictor and selector entry is a saturating up-down 2-bit Smith counter with the initial states shown.
- A taken branch (T) increments the predictor entry; a not-taken branch (N) decrements the predictor entry.
- A predictor entry less than 2 (0 or 1) results in a not-taken (N) prediction.
- A predictor entry greater than or equal to 2 (2 or 3) results in a taken (T) prediction.
- A selector value of 0 or 1 selects the bimodal predictor, while a selector value of 2 or 3 selects the gshare predictor.
- None of the predictors are tagged with the branch address.
- Avoid destructive interference by not updating the "wrong" predictor whenever the other predictor is right.

15. Fill in the following table with the prediction outcomes, and the predictor state following resolution of each branch.

Table 6: Combining Branch Predictor Simulation

Branch Address	Branch Outcome (TNT)	Predicted Outcome (T/N)			Predictor State After Branch is Resolved						
					Selector		Bimodal Predictor		Gshare Predictor		
		Bimodal	Gshare	Combined	PHT0	PHT1	PHT0	PHT1	BHR	PHT0	PHT1
Initial	N/A	N/A	N/A	N/A	2	0	0	2	0	2	1

Table 6: Combining Branch Predictor Simulation

Branch Address	Branch Outcome (TNT)	Predicted Outcome (T/N)			Predictor State After Branch is Resolved						
					Selector		Bimodal Predictor		Gshare Predictor		
		Bimodal	Gshare	Combined	PHT0	PHT1	PHT0	PHT1	BHR	PHT0	PHT1
0x654	N	T	N	T	2	**1**	0	2	0	2	**0**
0x780	T	N	T	T	**3**	1	0	2	**1**	**3**	0
0x78C	T	T	T	T	3	1	0	**3**	1	3	0
0x990	T	N	N	N	3	1	**1**	3	1	3	**1**
0xA04	N	T	T	T	3	1	1	**2**	0	**2**	1
0x78C	N	T	N	T	3	**2**	1	2	0	2	**0**

16. Compute the overall branch prediction rates (# of correctly predicted branches / total # of predicted branches) for the bimodal, gshare, and final (combined) predictors.

Bimodal branch prediction rate: _____1/6 = 16.7%_____

Gshare branch prediction rate: _____4/6 = 66.7%_____

Combined branch prediction rate: _____2/6 = 33%_____

17. Branch predictions are resolved when a branch instruction executes. Early out-of-order processors like the PowerPC 604 simplified branch resolution by forcing branches to execute strictly in program order. Discuss why this simplifies branch redirect logic, and explain in detail how the microarchitecture must change to accommodate out-of-order branch resolution.

    Branch misprediction recovery has to do 3 things: redirect fetch, recover rename mappings, and release resources from wrong-path instructions. The first is not really affected by OOO branch resolution; in either case, the branch execution unit provides a new fetch address and cancels fetched instructions that have not yet been inserted. Recovering rename mappings is affected; see following paragraph. The third item, releasing resources from wrong-path instructions is not substantially affected. Since these 3 tasks usually take multiple cycles, control for each of them is affected, since OOO resolution can lead to concurrent recoveries. I.e. with in-order branch resolution, once a recovery begins, the next recovery is guaranteed to not occur until after this one finishes, since the next one can't occur until a new branch has been fetched, mispredicted, and has reached the branch unit. However, with OOO resolution, another older branch can reach execute in the following cycle, leading to cascaded recoveries. Such concurrency is difficult to handle in the control logic for recovery.

More detail on map recovery: in-order branch resolution does not require random access to map checkpoints, but only FIFO access. I.e. the map checkpoints are processed in FIFO order, as branches execute, and are either discarded (if branch was correct) or are used to recover map to state at the mispredicted branch. In a rename-register machine like the Pentium Pro (that may not have map checkpoitns), further simplifications are possible: in the PPro design, a branch misprediction simply drains the out-of-order core so that all rename mappings get automatically reset to architected register (since wrong-path instructions are simply discarded and can't update the rename mapping table at commit). New instruction insertion and renaming is held off until the core has drained. This limits concurrency but simplifies implementation of the RAT.

18. Describe a scenario in which out-of-order branch resolution would be important for performance. State your hypothesis and describe a set of experiments to validate your hypothesis. Optionally, modify a timing simulator and conduct these experiments.

One case where OOO branch resolution helps performance is when an older branch is predicted correctly, but resolution of the branch is delayed since it is delayed by a cache miss. Further, assume a subsequent newer branch is mispredicted but is prevented from resolving since it is waiting for the older branch to resolve. In this case, allowing the newer branch to resolve out of order would correct the mispredicted branch sooner. One could construct a microbenchmark with this behavior, where the first branch relies on very predictable data values (e.g. all zero) but read from a very large array that does not fit in the cache. Meanwhile, a second branch depends on random data that fits in the cache. Something along the lines of:

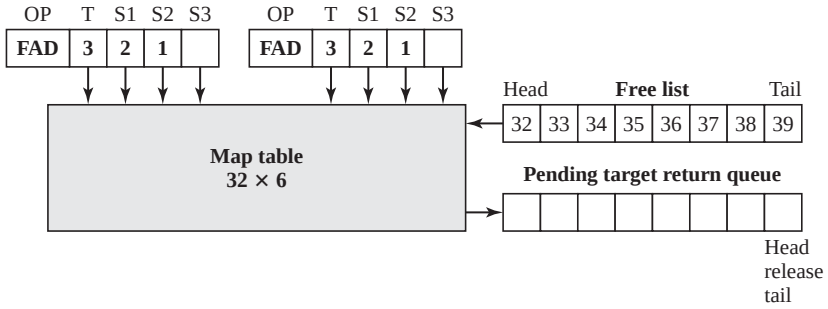
```
int branchtest() {
  int returnval=0;
  int zeroarray[1024*1024]; // initialize to zero
  int randomarray[100];     // initialize to random data
  for(int i=0;i<1024*1024/128;++i) {
    if (zeroarray[i*128] == 0)
      ++returnval;
    if (randomarray[i%100] > 100)
      returnval += 7;
  }
  return returnval;
}
```

Compile this using the toolchain from www.simplescalar.com and modify sim-outorder.c to resolve branches in order. Now compare the results between in-order and out-of-order branch resolution.

Problem 19 and Problem 20: Register Renaming

Given the DAXPY kernel shown in Figure 5-31 and the IBM RS/6000 (RIOS-I) floating-point **load** renaming scheme also discussed in class (both are shown below), simulate the execution of two iterations of the DAXPY loop and show the state of the floating-point map table, the pending target return queue, and the free list.

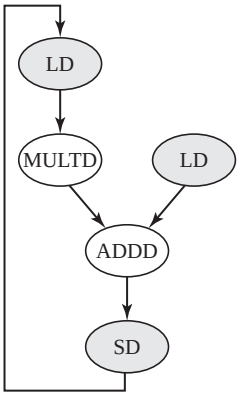
- Assume the initial state shown in the table on the next page.
- Note the table only contains columns for the registers that are referenced in the DAXPY loop.
- As in the RS/6000 implementation discussed, assume only a single load instruction is renamed per cycle and that only a single floating-point instruction can complete per cycle.
- Only floating-point load, multiply, and add instructions are shown in the table, since only these are relevant to the renaming scheme.
- Remember that only load destination registers are renamed.
- The first load from the loop prologue is filled in for you.




```
Y(i) = A * X(i) + Y(i)

    F0 ← LD,a
    R4 ← ADDI,Rx,#512     ;last address

Loop:
    F2 ← LD,0(Rx)         ; load X(i)
    F2 ← MULTD,F0,F2      ; A*X(i)
    F4 ← LD,0(Ry)         ; load Y(i)
    F4 ← ADDD,F2,F4       ; A*X(i) + Y(i)
    0(Ry) ← SD,F4         ; store into Y(i)
    Rx ← ADDI,Rx,#8       ; inc. index to X
    Ry ← ADDI,Ry,#8       ; inc. index to Y
    R20 ← SUB,R4,Rx       ; compute bound
    BNZ,R20,Loop          ; check if done
```



19. Fill in the remaining rows in the table below with the map table state and pending target return queue state after the instruction is renamed, and the free list state after the instruction completes.

Floating-point Instruction	Pending Target Return Queue after Instruction Renamed	Free List after Instruction Completes	Map Table subset		
			F0	F2	F4
Initial state		32,33,34,35,36,37,38,39	0	2	4
F0 <= LD, a	0	33,34,35,36,37,38,39	32	2	4
F2 <= LD, 0(Rx)	**2**,0	34,35,36,37,38,39	32	**33**	4
F2 <= MULTD, F0, F2	0	34,35,36,37,38,39,**2**	32	33	4
F4 <= LD, 0(Ry)	**4**,0	35,36,37,38,39,2	32	33	**34**
F4 <= ADDD, F2, F4	0	35,36,37,38,39,2,**4**	32	33	34
F2 <= LD, 0(Rx)	**33**,0	36,37,38,39,2,4	32	**35**	34
F2 <= MULTD, F0, F2	0	36,37,38,39,2,4,**33**	32	35	34
F4 <= LD, 0(Ry)	**34**,0	37,38,39,2,4,33	32	35	**36**
F4 <= ADDD, F2, F4	0	37,38,39,2,4,33,**34**	32	35	36

20. Given that the RS/6000 can rename a floating-point load in parallel with a floating-point arithmetic instruction (mult/add), and assuming the map table is a write-before-read structure, is any internal bypassing needed within the map table? Explain why or why not.

    No bypassing is needed; the load target register write occurs in first half of cycle, reads pick up new renamed values in second half of cycle. This assumes loads are always first in program order. If load is second in program order, then writes must be delayed by a cycle, effectively forcing load to be first in program order for the next rename group. Hence, assume loads are always first and no bypassing is needed.

21. Simulate the execution of the following code snippet using Tomasulo's algorithm. Show the contents of the reservation station entries, register file busy, tag (the tag is the RS ID number), and data fields for each cycle (make a copy of the table below for each cycle that you simulate). Indicate which instruction is executing in each functional unit in each cycle. Also indicate any result forwarding across a common data bus by circling the producer and consumer and connecting them with an arrow.

    i: R4 ← R0 + R8

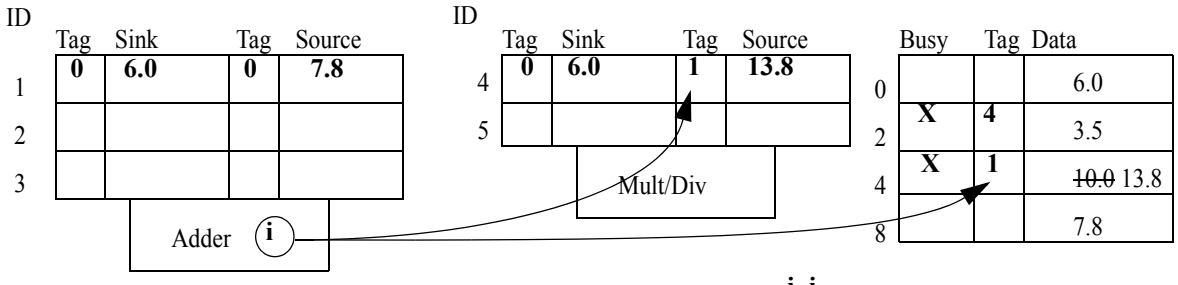
    j: R2 ← R0 * R4

    k: R4 ← R4 + R8

l: R8 ← R4 * R2

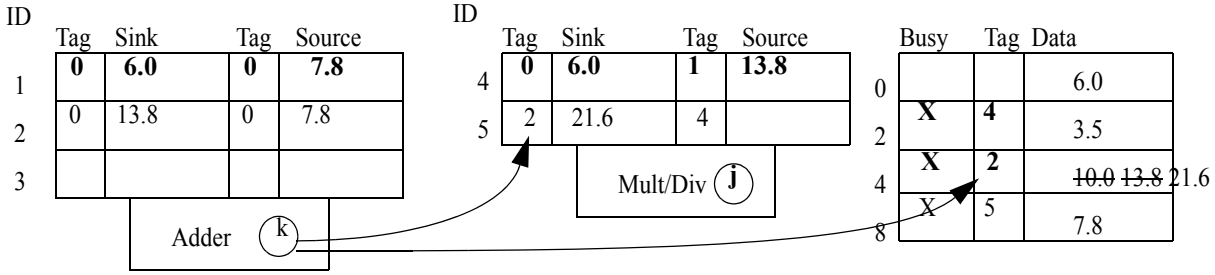
Assume dual dispatch and dual CDB (common data bus). Add latency is two cycles, and multiply latency is 3 cycles. An instruction can begin execution in the same cycle that it is dispatched, assuming all dependencies are satisfied.

CYCLE #: 0



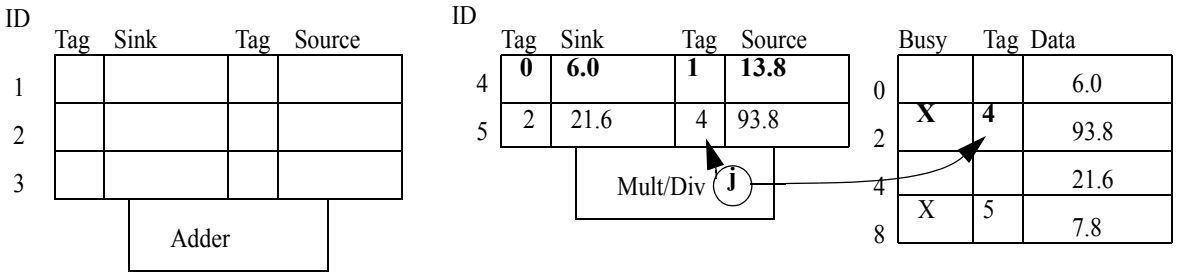
DISPATCHED INSTRUCTION(S): **i, j**

CYCLE #: 1



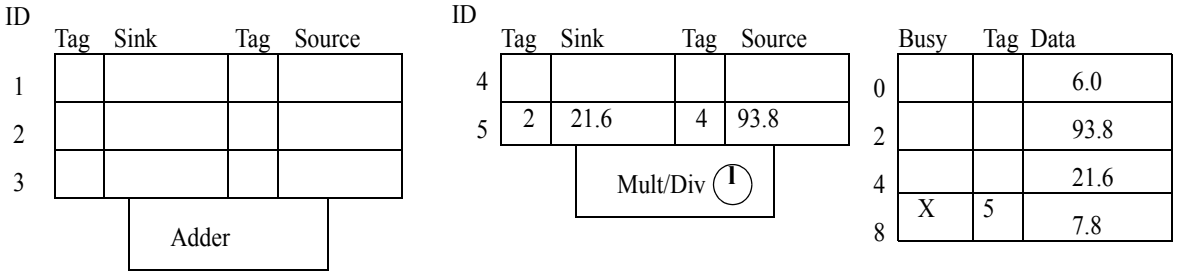
DISPATCHED INSTRUCTION(S): **k, l**

CYCLE #: 3 (skipped 2)



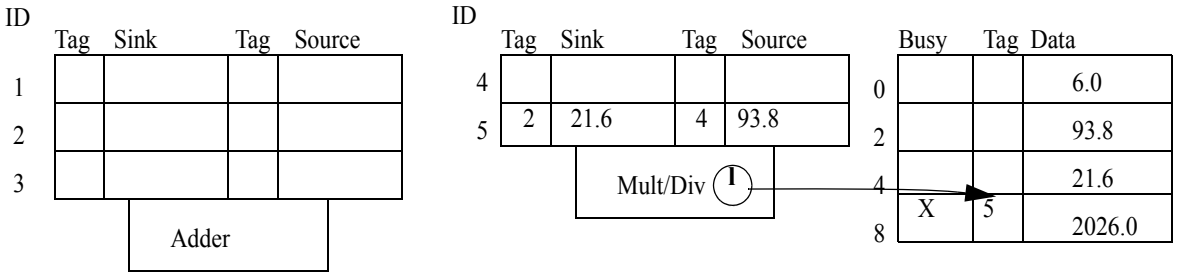
DISPATCHED INSTRUCTION(S): **k, l**

CYCLE #: 4



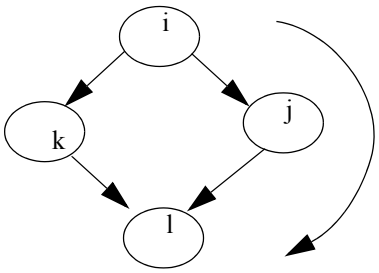
DISPATCHED INSTRUCTION(S): **k, l**

CYCLE #: 6 (skipped 5)



DISPATCHED INSTRUCTION(S): **k, l**

22. Determine whether or not the code executes at the dataflow limit for Problem 21. Explain why or why not. Show your work.



Critical path is 1+3+3=7 cycles
Prob. 18 executes in 7 cycles.
Therefore, Tomasulo executes this at the dataflow limit.

23. As presented in this chapter, load bypassing is a technique for enhancing memory data flow. With load bypassing, load instructions are allowed to jump ahead of earlier store instructions. Once address generation is done, a store instruction can be completed architecturally and can then enter the store buffer to await available bus cycle for writing to memory. Trailing loads are allowed to bypass these stores in the store buffer if there is no address aliasing.

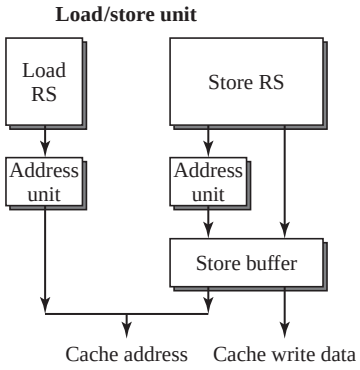
    In this problem you are to simulate such load bypassing (there is no load forwarding). You are given a sequence of load/store instructions and their addresses (symbolic). The number to the left of each instruction indicates the cycle in which that instruction is dispatched to the reservation station; it can begin execution in that same cycle. Each store instruction will have an additional number to its right, indicating the cycle in which it is ready to retire, i.e., exit the store buffer and write to the memory.

    Assumptions:

    - All operands needed for address calculation are available at dispatch.
    - One load and one store can have their addresses calculated per cycle.
    - One load OR store can be executed, i.e., allowed to access the cache, per cycle.
    - The reservation station entry is deallocated the cycle after address calculation and issue.
    - The store buffer entry is deallocated when the cache is accessed.
    - A store instruction can access the cache the cycle after it is ready to retire.
    - Instructions are issued in order from the reservation stations.
    - Assume 100% cache hits.

Example:

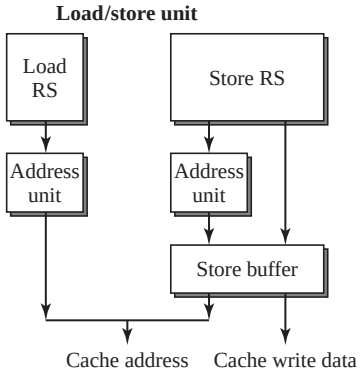
Dispatch cycle	Instruction	Retire cycle
1	Load A	
1	Load B	
1	Store C	5



Cycle	Load Reservation Station				Store Reservation Station				Store Buffer				Cache Address	Cache Write data
1	Ld A	Ld B			St C									
2	Ld B								St C				Ld A	
3									St C				Ld B	
4									St C					
5									St C					
6													St C	data

Code:

Dispatch cycle	Instruction	Retire cycle
1	Store A	6
2	Load B	
3	Load A	
4	Store D	10
4	Load E	
4	Load A	
5	Load D	



Cycle	Load Reservation Station				Store Reservation Station				Store Buffer				Cache Address	Cache Write data
1					St A									
2	Ld B								St A					
3	Ld A								St A				Ld B	

Cycle	Load Reservation Station				Store Reservation Station				Store Buffer				Cache Address	Cache Write data
4	Ld A	Ld E	Ld A		St D				St A					
5	Ld A	Ld D	Ld A						St A	St D			Ld E	
6	Ld A	Ld D	Ld A						St A	St D				
7	Ld A	Ld D	Ld A							St D			St A	data
8		Ld D	Ld A							St D			Ld A	
9		Ld D								St D			Ld A	
10		Ld D								St D				
11		Ld D											St D	data
12													Ld D	
13														
14														
15														

24. In one or two sentences compare and contrast Load Forwarding with Load Bypassing.

    Load bypassing allows independent loads to bypass earlier stores before those stores are committed, but stalls dependent loads until the aliased store commits. Load forwarding will forward values from the earlier uncommitted store by reading the value from the store queue and satisfying the load with that value.

25. Would Load Forwarding improve the performance of the above code sequence? Why or why not?

    Yes, since both the Ld A and Ld D could issue sooner, rather than having to wait for cycle 7 and cycle 11, after the respective stores had committed.

Problem 26 through Problem 28

The goal of lockup-free cache designs is to increase the amount of concurrency or parallelism in the processor by overlapping cache miss handling with other processing or other cache misses. For this problem, assume the following simple workload running on a processor with a primary cache with 64-byte lines and an 16-byte memory bus. Assume that t_{miss} is 5 cycles, and $t_{transfer}$ for each 16-byte subblock is 1 cycle. Hence, for a simple blocking cache, a miss will take t_{miss} + (64/16) x $t_{transfer}$ = 5 + 4 x 1 = 9 cycles. Here is the workload:

```
for(i=1;i<10000;++i)
        a += A[i] + B[i];
```

In RISC assembly language, assuming r3 points to A[0] and r4 points to B[0]:

```
        li r2,9999          # load iteration count into r2
loop:   lfdu  r5,8(r3)      # load A[i], incr. pointer in r3
        lfdu  r6,8(r4)      # load B[i], incr. pointer in r4
        add   r7,r7,r5      # add A[i] to a
        add   r7,r7,r6      # add B[i] to a
        bdnz  r2,loop       # decrement r2, branch if not zero
```

Here is a timing diagram for the loop body assuming neither array hits in the cache, and the cache is a simple blocking design (m = miss latency, t = transfer latency, A = load from A, B = load from B, a = add, and b = branch):

	Cycle										1										2									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A		m	m	m	m	m	t	t	t	t																				
Array B																														
											m	m	m	m	m	t	t	t	t											
Execution	A										B										a	a	b	A	B	a	a	b	A	B

For the following questions, assume that each instruction takes a single cycle to execute, and that all subsequent instructions are stalled on a cache miss until the requested data are returned by the cache (as shown in the timing diagram above). Furthermore, assume that no portion of either array (A or B) is in the cache initially, but must be fetched on demand misses. Also, assume there are enough loop iterations to fill all the table entries provided.

26. Assume a blocking cache design with critical word forwarding (i.e., the requested word is forwarded as soon as it has been transferred), but support for only a single outstanding miss. Fill in the timing diagram and explain your work.

	Cycle										1										2									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A		m	m	m	m	m	t	t	t	t																				
Array B											m	m	m	m	m	t	t	t	t											
Execution	A							B									a	a	b	A	B	a	a	b	A	B	a	a	b	A

	3										4										5									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A																										m	m	m	m	m
Array B																														
Execution	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A					

27. Now fill in the timing diagram for a lockup-free cache that supports multiple outstanding misses, and explain your work.

	Cycle										1										2									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A		m	m	m	m	m	t	t	t	t																				
Array B									m	m	m	m	m	t	t	t	t	t												
Execution	A							B								a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a

	3										4										5									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A																										m	m	m	m	m
Array B																														
Execution	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A							B

28. Instead of a 64B cache line, assume a 32B line size for the cache, and fill in the timing diagram as in Problem 24. (also explain your work.)

	Cycle										1										2									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A		m	m	m	m	m	t	t																						
Array B									m	m	m	m	m	t	t															
Execution	A							B							a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a

	3										4										5									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A				m	m	m	m	m	t	t																				
Array B											m	m	m	m	m	t	t													
Execution	a	b	A						B							a	a	b	A	B	a	a	b	A	B	a	a	b	A	

Problem 29 through Problem 30

The goal of prefetching and nonblocking cache designs is to increase the amount of concurrency or parallelism in the processor by overlapping cache miss handling with other processing or other cache misses. For this problem, assume the same simple workload from Problem 26 running on a processor with a primary cache with **16-byte** lines and an **4-byte** memory bus. Assume that t_{miss} is 2 cycles, and $t_{transfer}$ for each 4-byte subblock is 1 cycle. Hence, a miss will take $\mathbf{t_{miss} + (16/4) \times t_{transfer} = 2 + 4 \times 1 = 6}$ **cycles**.

For the remainder of this question, assume that each instruction takes a single cycle to execute, and that all subsequent instructions are stalled on a cache miss until the requested data are returned by the cache (as shown in the timing diagram below). Furthermore, assume that no portion of either array (A or B) is in the cache initially, but must be fetched on demand misses. Also, assume there are enough loop iterations to fill all the table entries provided.

Further assume a stride-based hardware prefetch mechanism that can track up to two independent strided address streams, and issues a stride prefetch the cycle after it has observed the same stride twice, from observing three strided misses (e.g. misses to A, A+32, A+64 triggers a prefetch for A+96). The prefetcher will issue its next strided prefetch in the cycle following a demand reference to its previous prefetch, but no sooner (to avoid overwhelming the memory subsystem). Assume that a demand reference will always get priority over a prefetch for any shared resource.

29. Fill in the following table to indicate miss (m) and transfer (t) cycles for both demand misses and prefetch requests for the workload above. **Annotate each prefetch with a symbolic address (e.g. A+64)**. The first 30 cycles are filled in for you.

	Cycle										**1**										**2**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Array A		m	m	t	t	t	t													m	m	t	t	t	t					
Array B							m	m	t	t	t	t													m	m	t	t	t	t
Prefetch 1																														
Prefetch 2																														
Execution	A					B					a	a	b	A	B	a	a	b	A					B					a	a

	3										**4**										**5**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Array A								m	m	t	t	t	t																	
Array B													m	m	t	t	t	t												
Prefetch 1									m	m				t					t	t	t					m	m	t	t	t
Prefetch 2														m	m							t	t	t	t		m	m		
Execution	b	A	B	a	a	b	A					B					a	a	b	A	B	a	a	b	A	B	a	a	b	A

	6										7										8									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A																														
Array B																														
Prefetch 1	t					m	m	t	t	t	t					m	m	t	t	t	t					m	m	t	t	t
Prefetch 2		t	t	t	t		m	m				t	t	t	t		m	m				t	t	t	t		m	m		
Execution	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A

	9										10										11									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Array A																														
Array B																														
Prefetch 1	t					m	m	t	t	t	t					m	m	t	t	t	t					m	m	t	t	t
Prefetch 2		t	t	t	t		m	m				t	t	t	t		m	m				t	t	t	t		m	m		
Execution	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A	B	a	a	b	A

30. Report the overall miss for the portion of execution shown above, as well as the coverage of the prefetches generated (coverage is defined as: (number of misses eliminated by prefetching) / (number of misses without prefetching)).

    Overall per-reference miss rate: _____6/39_____

    Prefetcher coverage: __(20-6)/20 = 14/20 = 70%_____

31. A victim cache is used to augment a direct-mapped cache to reduce conflict misses. For additional background on this problem, read Jouppi's paper on victim caches [Jouppi, 1990]. Please fill in the following table to reflect the state of each cache line in a 4-entry direct-mapped cache and a 2-entry fully associative victim cache following each memory reference shown. Also, record whether the reference was a cache hit or a cache miss. The reference addresses are shown in hexadecimal format. Assume the direct-mapped cache is indexed with the low-order bits above the 16-byte line offset (e.g. address 40 maps to set 0, address 50 maps to set 1, etc.). Use '-' to indicate an invalid line and the address of the line to indicate a valid line. Assume LRU policy for the victim cache and mark the LRU line as such in the table.

Reference Address	Hit/Miss	Direct-Mapped Cache				Victim Cache	
		Line 0	Line 1	Line 2	Line 3	Line 0	Line 1
[init state]		-	110	-	FF0	1F0	210/LRU
80	M	80					
A0	M			A0			
200	M	200				80	1F0/LRU
80	H-victim	80				200	1F0/LRU
B0	M				B0	FF0	200/LRU
E0	M			E0		A0	FF0/LRU
200	M	200				80	A0/LRU
80	H-victim	80				200	A0/LRU
200	H-victim	200				80	A0/LRU

32. Given your results from Problem 31, and excluding the data supplied to the processor to supply the requested instruction words, compute the total number of bytes that were transferred into and out of the direct-mapped cache array. Assume this is a read-only instruction cache--hence there are no writebacks of dirty lines. Show your work.

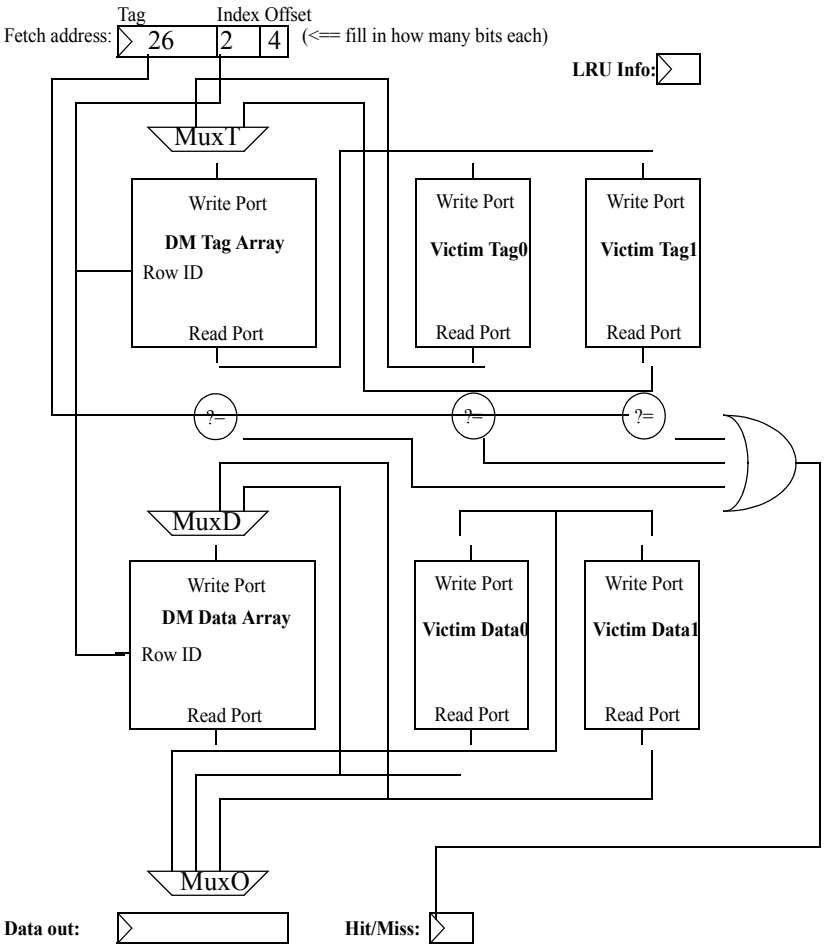
    Bytes transferred in: 6 misses x 16 bytes each + 3 VC hits x 16 bytes each = 144 bytes in

    Bytes transferred out: 7 evictions x 16 bytes each = 112 bytes out

    Total bytes = 144 + 112 = 256 bytes

33. Fill in the details for the block diagram of the read-only victim I-cache design shown below (based on the victim cache outlined in Problem 31). For additional background on this problem, read Jouppi's paper on victim caches [Jouppi, 1990]. Show data and address paths, identify which address bits are used for indexing and tag comparison, and show the logic for generating a hit/miss signal as well as control logic for any multiplexers that are included in your design. Don't forget the data paths for 'swapping' cache lines between the victim cache and the DM

cache. Assume that the arrays are read in the first half of each clock cycle and written in the second half of each clock cycle. Do not include LRU update or control or data paths for handling misses.



MuxT controlled by VTag0 and VTag1 comparators, e.g.:

MuxTctrl = VTag1

MuxD control same as MuxT

DM write enable = VTag0 + VTag1

Victim tag write enables:

wren0 = ~lru & ~DMTaghit & DMvalid

wren1 = lru & ~DMTaghit & DMvalid

MuxOctrl (one-hot):

MuxOctrl0 = DMTag

MuxOctrl1 = VTag0

MuxOctrl2 = Vtag1

Problem 34 through Problem 36: Simplescalar Simulation

These problems require you to use and modify the Simplescalar 3.0 simulator suite, available from http://www.simplescalar.com. Use the Simplescalar simulators to execute the 4 benchmarks in the instructional benchmark suite from http://www.simplescalar.com.

34. First use the sim-outorder simulator with the default machine parameters to simulate an out-of-order processor that performs both right-path and wrong-path speculative references against the instruction cache, data cache and unified level 2 cache. Report the instruction cache, data cache, and l2 cache miss rates (misses per reference) as well as the total number of references and the total number of misses for each of the caches.

    Detailed results not provided (simplescalar output).

35. Now simulate the exact same memory hierarchy as in Problem 34 but using the sim-cache simulator. Note that you will have to determine the parameters to use when you invoke sim-cache. Report the same statistics as in Problem 34, and compute the increase (or decrease) in each statistic.

Detailed results not provided (simplescalar output).

36. Now modify sim-outorder.c to inhibit cache misses caused by wrong-path references. This is very easy to do for instruction fetches in sim-outorder, since the global variable "spec_mode" is set whenever the processor begins to execute instructions from an incorrect branch path. You can use this global flag to inhibit instruction cache misses from wrong path instruction fetches. For data cache misses, you can check the "spec_mode" flag within each RUU entry, and inhibit data cache misses for any such instruction. "Inhibiting misses" means don't even bother to check the cache hierarchy; simply treat these references as if they hit the cache. You can find the places where the cache is accessed by searching for calls to the function "cache_access" within sim-outorder.c. Now recollect the statistics from Problem 34 in your modified simulator and compare your results to Problem 35. If your results still differ from Problem 35, explain why that might be the case.

    Detailed results are not provided here (simplescalar output).

    Discrepancies between Problem 35 and Problem 36 can be explained by the fact that the spec_mode flag is not set until the dispatch stage; hence, 1-2 cycles of speculative I-fetches have already occurred in the fetch stage, and some of those may have been from the wrong path. Hence the i-fetch count (and possibly number of misses) can be elevated for P36 over P35, which has no speculative i-fetches. On the data side, the reference count for P36 can actually be lower, since simplescalar does not check the cache for loads that are forwarded from the store queue. This results in fewer references to the cache, and possibly fewer misses as well. Variations in misses can also be accounted for by the fact that loads and stores issue out of order, which can affect replacement decisions in the caches LRU policy.