CS252
Solutions to Midterm 1 Practice Problems

3 October, 2007

Problem 1: Out-of-Order Scheduling

Problem 1.A

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I_1	-1	0	1	2	L.D	T0	R2	-
I_2	0	2	12	13	MUL.D	T1	T0	F0
I_3	1	**13**	**15**	**16**	ADD.D	**T2**	**T1**	**F0**
I_4	**2**	**3**	**4**	**17**	ADDI	**T3**	**R2**	**-**
I_5	**3**	**4**	**5**	**18**	L.D	**T4**	**T3**	**-**
I_6	**4**	**6**	**16**	**19**	MUL.D	**T5**	**T4**	**T4**
I_7	**5**	**17**	**19**	**20**	ADD.D	**T6**	**T5**	**T2**

Table 1.1

Problem 1.B

	Time				OP	Dest	Src1	Src2
	Decode → ROB	Issued	WB	Committed				
I_1	-1	0	1	2	L.D	T0	R2	-
I_2	0	2	12	13	MUL.D	T1	T0	F0
I_3	3	**13**	**15**	**16**	ADD.D	**T0**	**T1**	**F0**
I_4	**14**	**15**	**16**	**17**	ADDI	**T1**	**R2**	**-**
I_5	**17**	**18**	**19**	**20**	L.D	**T0**	**T1**	**-**
I_6	**18**	**20**	**30**	**31**	MUL.D	**T1**	**T0**	**T0**
I_7	**21**	**31**	**33**	**34**	ADD.D	**T0**	**T1**	**F3**

Table 1.2

Problem 2: Branch Prediction

This problem will investigate the effects of adding global history bits to a standard branch prediction mechanism. **In this problem assume that the MIPS ISA has no delay slots.**

Throughout this problem we will be working with the following program:

```
loop:
     LW   R4, 0(R3)
     ADDI R3, R3, #4
     SUBI R1, R1, #1
b1:
     BEQZ R4, b2
     ADDI R2, R2, #1
b2:
     BNEZ R1, loop
```

Assume the initial value of R1 is n (n>0).
Assume the initial value of R2 is 0 (R2 holds the result of the program).
Assume the initial value of R3 is p (a pointer to the beginning of an array of 32-bit integers).

All branch prediction schemes in this problem will be based on those covered in lecture. We will be using a 2-bit predictor state machine, as shown below.

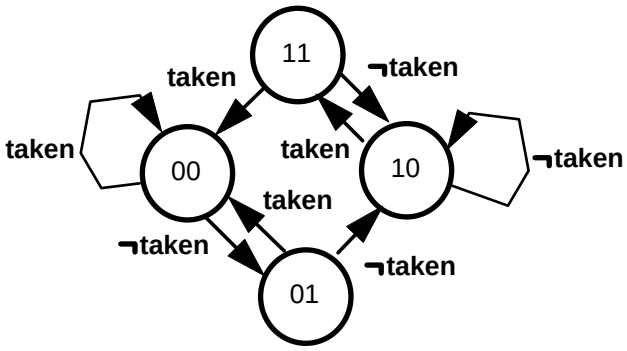

Figure 1: BP bits state diagram

In state 1X we will guess not taken. In state 0X we will guess taken.

Assume that b1 and b2 do not conflict in the BHT.

Problem 2.A **Program**

What does the program compute? That is, what does R2 contain when we exit the loop?

R2 contains the number of non-zero entries in the first n elements of array p.

Problem 2.B — 2-bit branch prediction

Now we will investigate how well our standard 2-bit history branch predictor performs. Assume the inputs to the program are n=8 and p[0] = 1, p[1] = 0, p[2] = 1, p[3] = 0,… etc.; i.e. the array elements exhibit an alternating pattern of 1's and 0's. Fill out Table 2.1 (note that the first few lines are filled out for you). What is the number of mispredicts?

Table 2.1 contains an entry for every time a branch (either b1 or b2) is executed. The Branch Prediction (BP) bits in the table are the bits from the BHT. If b1 is being executed, then the b1 bits from the BHT are to be filled in. If b2 is being executed, then the b2 bits from the BHT are to be filled in.

There are 7 mispredicts (shown in Table 2.1 in italics).

System State		Branch Predictor	Behavior		Updated Values
PC	R3/R4	BP bits	Predicted Behavior	Actual Behavior	New BP bits
b1	4/1	**10**	**N**	**N**	**10**
b2	4/1	**10**	***N***	***T***	**11**
b1	8/0	**10**	***N***	***T***	**11**
b2	8/0	**11**	***N***	***T***	**00**
b1	12/1	11	N	N	10
b2	12/1	00	T	T	00
b1	16/0	10	***N***	***T***	11
b2	16/0	00	T	T	00
b1	20/1	11	N	N	10
b2	20/1	00	T	T	00
b1	24/0	10	***N***	***T***	11
b2	24/0	00	T	T	00
b1	28/1	11	N	N	10
b2	28/1	00	T	T	00
b1	32/0	10	***N***	***T***	11
b2	32/0	00	***T***	***N***	01

Table 2.1: Behavior of branch prediction (Problem 2.B)

Problem 2.C — Branch prediction with one global history bit

Now we add a global history bit to the branch predictor. Fill out Table 2.2, and again give the total number of mispredicts you get when running the program with the same inputs.

There are 9 mispredicts (shown in Table 2.2 in italics).

System State			Branch Predictor		Behavior		Updated Values		
PC	R3/R4	history bit	BP bits		Predicted Behavior	Actual Behavior	New BP bits		New history
			set 0	set 1			set 0	set 1	
b1	4/1	**1**	**10**	**10**	**N**	**N**	**10**	**10**	**0**
b2	4/1	**0**	**10**	**10**	***N***	***T***	**11**	**10**	**1**
b1	8/0	1	10	10	***N***	***T***	10	11	1
b2	8/0	1	11	10	***N***	***T***	11	11	1
b1	12/1	1	10	11	N	N	10	10	0
b2	12/1	0	11	11	***N***	***T***	00	11	1
b1	16/0	1	10	10	***N***	***T***	10	11	1
b2	16/0	1	00	11	***N***	***T***	00	00	1
b1	20/1	1	10	11	N	N	10	10	0
b2	20/1	0	00	00	T	T	00	00	1
b1	24/0	1	10	10	***N***	***T***	10	11	1
b2	24/0	1	00	00	T	T	00	00	1
b1	28/1	1	10	11	N	N	10	10	0
b2	28/1	0	00	00	T	T	00	00	1
b1	32/0	1	10	10	***N***	***T***	10	11	1
b2	32/0	1	00	00	***T***	***N***	00	01	0

Table 2.2: Behavior of branch prediction with one history bit (Problem 2.C)

Problem 2.D **Branch prediction with two global history bits**

Now we add a second global history bit. Fill out Table 2.3. Again, compute the number of mispredicts you get for the same input.

There are 7 mispredicts (shown in Table 2.3 in italics).

System State			Branch Predictor				Behavior		Updated Values				
PC	R3/R4	history bits	BP bits				Predicted Behavior	Actual Behavior	New BP bits				New Hist.
			set 0	set 1	set 2	set 3			set 0	set 1	set 2	set 3	
b1	4/1	**11**	**10**	**10**	**10**	**10**	**N**	**N**	**10**	**10**	**10**	**10**	**01**
b2	4/1	**01**	**10**	**10**	**10**	**10**	***N***	***T***	**10**	**11**	**10**	**10**	**10**
b1	8/0	10	10	10	10	10	***N***	***T***	10	10	11	10	11
b2	8/0	11	10	11	10	10	***N***	***T***	10	11	10	11	11
b1	12/1	11	10	10	11	10	N	N	10	10	11	10	01
b2	12/1	01	10	11	10	11	***N***	***T***	10	00	10	11	10
b1	16/0	10	10	10	11	10	***N***	***T***	10	10	00	10	11
b2	16/0	11	10	00	10	11	***N***	***T***	10	00	10	00	11
b1	20/1	11	10	10	00	10	N	N	10	10	00	10	01
b2	20/1	01	10	00	10	00	T	T	10	00	10	00	10
b1	24/0	10	10	10	00	10	T	T	10	10	00	10	11
b2	24/0	11	10	00	10	00	T	T	10	00	10	00	11
b1	28/1	11	10	10	00	10	N	N	10	10	00	10	01
b2	28/1	01	10	00	10	00	T	T	10	00	10	00	10
b1	32/0	10	10	10	00	10	T	T	10	10	00	10	11
b2	32/0	11	10	00	10	00	***T***	***N***	10	00	10	01	01

Table 2.3: Behavior of branch prediction with two history bits (Problem 2.D)

Problem 2.E **Analysis I**

Compare your results from problems 2.B, 2.C, and 2.D. When do most of the mispredicts occur in each case (at the beginning, periodically, at the end, etc.)? What does this tell you about global history bits in general? For large n, what prediction scheme will work best? Explain briefly.

The first thing to notice is that the more history bits we have, the longer it takes to get any correct prediction since we have to "train" the predictor. These start-up costs go up as the number of history bits increase.

Another thing to notice is that the single history bit does not help at all (even after we get into a steady-state phase). In both the single history bit and no history cases, the b2 branch is predicted correctly once we get past the start-up phase (since b2 is always taken). The single bit of history does not help since this history is too "nearsighted". The second history bit captures the alternating pattern of the b1 branch, and hence does not mispredict once it gets past the start-up phase. For large n then, the 2-bit history predictor is the best.

The final point of observation is that all the predictors mispredict the fall-through case (last b2 branch).

The input we worked with in this problem is quite regular. How would you expect things to change if the input were random (each array element were equally probable 0 or 1). Of the three branch predictors we looked at in this problem, which one will perform best for this type of input? Is your answer the same for large and small n?

What does this tell you about when additional history bits are useful and when they hurt you?

When the input is random, no prediction scheme will help predict whether b1 is taken or not. All three schemes will eventually predict b2 as always taken. However, the more history bits are used, the more sets need to be trained to predict the always taken for b2. Thus, the more history bits used, the more mispredicts of branch b2 will occur initially. The answer does not depend on the size of n. However, as n gets large, the start-up costs become insignificant among the three schemes.

The moral of the problem is: history bits are useful if there is a pattern among a sequence of branches. The longer this pattern is, the more history bits are needed to be able to recognize this pattern. If the pattern is not recognized, then global history bits can hurt because it take longer to train the branches that can be predicted correctly.

Problem 3: Cache Access-Time & Performance

Here is the completed Table 3.1 for problems 3.A and 3.B.

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	200×(# of index bits) + 1000	Tag	3400	3000
		Data	3400	3000
Memory array	200×$\log_2$(# of rows) + 200×$\log_2$(# of bits in a row) + 1000	Tag	4217	4250
		Data	5000	5000
Comparator	200×(# of tag bits) + 1000		4000	4400
N-to-1 MUX	500×$\log_2$ N + 1000		2500	2500
Buffer driver	2000			2000
Data output driver	500×(associativity) + 1000		1500	3000
Valid output driver	1000		1000	1000

Table 3.1: Delay of each Cache Component

Problem 3.A **Access time: DM**

To use the delay equations, we need to know how many bits are in the tag and how many are in the index. We are given that the cache is addressed by word, and that input addresses are 32-bit byte addresses; the two low bits of the address are not used.

Since there are 8 (2^3) words in the cache line, 3 bits are needed to select the correct word from the cache line.

In a 128 KB direct-mapped cache with 8 word (32 byte) cache lines, there are 2^{12} cache lines (128KB/32B). 12 bits are needed to address 2^{12} cache lines, so the number of index bits is 12. The remaining 15 bits (32 – 2 – 3 – 12) are the tag bits.

We also need the number of rows and the number of bits in a row in the tag and data memories. The number of rows is simply the number of cache lines (2^{12}), which is the same for both the tag and the data memory. The number of bits in a row for the tag memory is the sum of the number of tag bits (15) and the number of status bits (2), 17 bits total. The number of bits in a row for the data memory is the number of bits in a cache line, which is 256 (32 bytes × 8 bits/byte).

With 8 words in the cache line, we need an 8-to-1 MUX. Since there is only one data output driver, its associativity is 1.

Decoder (Tag) = 200 × (# of index bits) + 1000 = 200 × 12 + 1000 = 3400 ps
Decoder (Data) = 200 × (# of index bits) + 1000 = 200 × 12 + 1000 = 3400 ps

Memory array (Tag) = 200 × $\log_2$(# of rows) + 200 × $\log_2$(# bits in a row) + 1000

	$= 200 \times \log_2(2^{12}) + 200 \times \log_2(17) + 1000$		≈ 4217 ps
Memory array (Data)	$= 200 \times \log_2$(# of rows) $+ 200 \times \log_2$(# bits in a row) $+ 1000$		
	$= 200 \times \log_2(2^{12}) + 200 \times \log_2(256) + 1000$		$= 5000$ ps
Comparator	$= 200 \times$ (# of tag bits) $+ 1000$	$= 200 \times 15 + 1000$	$= 4000$ ps
N-to-1 MUX	$= 500 \times \log_2(N) + 1000$	$= 500 \times \log_2(8) + 1000$	$= 2500$ ps
Data output driver	$= 500 \times$ (associativity) $+ 1000$	$= 500 \times 1 + 1000$	$= 1500$ ps

To determine the critical path for a cache read, we need to compute the time it takes to go through each path in hardware, and find the maximum.

Time to tag output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (valid output driver time)
≈ 3400 + 4217 + 4000 + 500 + 1000 = 13117 ps

Time to data output driver
= (data decode time) + (data memory access time) + (mux time) + (data output driver time)
= 3400 + 5000 + 2500 + 1500 = 12400 ps

The critical path is therefore the tag read going through the comparator. The access time is 13117 ps. At 150 MHz, it takes 0.013117×150, or 2 cycles, to do a cache access.

Problem 3.B **Access time: SA**

As in 3.A, the low two bits of the address are not used, and 3 bits are needed to select the appropriate word from a cache line. However, now we have a 128 KB 4-way set associative cache. Since each way is 32 KB and cache lines are 32 bytes, there are 2^{10} lines in a way (32KB/32B) that are addressed by 10 index bits. The number of tag bits is then (32 – 2 – 3 – 10), or 17.

The number of rows in the tag and data memory is 2^{10}, or the number of sets. The number of bits in a row for the tag memory is now quadruple the sum of the number of tag bits (17) and the number of status bits (2), 76 bits total. The number of bits in a row for the data memory is twice the number of bits in a cache line, which is 1024 (4×32 bytes $\times$ 8 bits/byte).

As in 3.A, we need an 8-to-1 MUX. However, since there are now four data output drivers, the associativity is 4.

Decoder (Tag)	$= 200 \times$ (# of index bits) $+ 1000$	$= 200 \times 10 + 1000$	$= 3000$ ps
Decoder (Data)	$= 200 \times$ (# of index bits) $+ 1000$	$= 200 \times 10 + 1000$	$= 3000$ ps

Memory array (Tag) $= 200 \times \log_2$(# of rows) $+ 200 \times \log_2$(# bits in a row) $+ 1000$

	$= 200 \times \log_2(2^{10}) + 200 \times \log_2(76) + 1000$		≈ 4250 ps
Memory array (Data)	$= 200 \times \log_2(\#\text{ of rows}) + 200 \times \log_2(\#\text{ bits in a row}) + 1000$		
	$= 200 \times \log_2(2^{10}) + 200 \times \log_2(1024) + 1000$		$= 5000$ ps
Comparator	$= 200 \times (\#\text{ of tag bits}) + 1000$	$= 200 \times 17 + 1000$	$= 4400$ ps
N-to-1 MUX	$= 500 \times \log_2(N) + 1000$	$= 500 \times \log_2(8) + 1000$	$= 2500$ ps
Data output driver	$= 500 \times (\text{associativity}) + 1000$	$= 500 \times 4 + 1000$	$= 3000$ ps

Time to valid output driver
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (OR gate time) + (valid output driver time)
= 3000 + 4250 + 4400 + 500 + 1000 + 1000 = 14150 ps

There are two paths to the data output drivers, one from the tag side, and one from the data side. Either may determine the critical path to the data output drivers.

Time to get through data output driver through tag side
= (tag decode time) + (tag memory access time) + (comparator time) + (AND gate time)
+ (buffer driver time) + (data output driver)
= 3000 + 4250 + 4400 + 500 + 2000 + 3000 = 17150 ps

Time to get through data output driver through data side
= (data decode time) + (data memory access time) + (mux time) + (data output driver)
= 3000 + 5000 + 2500 + 3000 = 13500 ps

From the above calculations, it's clear that the critical path leading to the data output driver goes through the tag side.

The critical path for a read therefore goes through the tag side comparators, then through the buffer and data output drivers. The access time is 17150 ps. The main reason that the 4-way set associative cache is slower than the direct-mapped cache is that the data output drivers need the results of the tag comparison to determine which, if any, of the data output drivers should be putting a value on the bus. At 150 MHz, it takes 0.0175×150, or 3 cycles, to do a cache access.

D-map									
	line in cache								hit?
Address	L0	L1	L2	L3	L4	L5	L6	L7	
110	inv	11	inv	inv	inv	inv	inv	inv	no
136				13					no
202	20								no
1A3			1A						no
102	10								no
361							36		no
204	20								no
114									yes
1A4									yes
177								17	no
301	30								no
206	20								no
135									yes

	D-map
Total Misses	10
Total Accesses	13

4-way									LRU
	line in cache								hit?
Address	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	inv	inv	inv	11	inv	inv	inv	no
136					11	13			no
202	20								no
1A3	20	1A							no
102	20	1A	10						no
361	20	1A	10	36					no
204									yes
114									yes
1A4									yes
177					11	13	17		no
301	20	1A	30	36					no
206									yes
135									yes

	4-way LRU
Total Misses	8
Total Accesses	13

4-way									FIFO
Address	line in cache								hit?
	Set 0				Set 1				
	way0	**way1**	**way2**	**way3**	**way0**	**way1**	**way2**	**way3**	
110	inv	inv	inv	inv	11	inv	inv	inv	no
136						13			no
202	20								no
1A3		1A							no
102			10						no
361				36					no
204									yes
114									yes
1A4									yes
177							17		no
301	30								no
206		20							no
135									yes

	4-way FIFO
Total Misses	9
Total Accesses	13

Problem 3.D **Average latency**

The miss rate for the direct-mapped cache is 10/13. The miss rate for the 4-way set associative cache is 8/13.

The average memory access latency is (hit time) + (miss rate) × (miss time).

For the direct-mapped cache, the average memory access latency would be (2 cycles) + (10/13) × (20 cycles) = 17.38 ≈ 18 cycles.

For the set associative cache, the average memory access latency would be (3 cycles) + (8/13) × (20 cycles) = 15.31 ≈ 16 cycles.

The set associative cache is better in terms of average memory access latency.

For the above example, LRU has a slightly smaller miss rate. This is because the FIFO policy replaced the {20} block instead of the {10} block during the 12th access, because the {20} block

has been in the cache longer even though the {10} was least recently used, whereas the LRU policy took advantage of temporal/spatial locality.

However, the LRU policy does not always yield a lower miss rate. In the last example, if the {10} block was accessed instead of the {20} block, then the FIFO policy would've done better.

Problem 4: VLIW & Vector Coding

Ben Bitdiddle has the following C loop, which takes the absolute value of elements within a vector.

```
for (i = 0; i < N; i++) {
    if (A[i] < 0)
        A[i] = -A[i];
}
```

Problem 4.A

Ben is working with an in-order VLIW processor, which issues two MIPS-like operations per instruction cycle. Assume a five-stage pipeline with two single-cycle ALUs, memory with one read and one write port, and a register file with four read ports and two write ports. Also assume that there are no branch delay slots, and loads and stores only take one cycle to complete. Turn Ben's loop into VLIW code. A and N are 32-bit signed integers. Initially, `R1` contains N and `R2` points to `A[0]`. You do not have to preserve the register values. Optimize your code to improve performance but do not use loop unrolling or software pipelining. What is the average number of cycles per element for this loop, assuming data elements are equally likely to be negative and non-negative?

```
; Solution 1
; Initial Conditions:
;     R1 = N
;     R2 = &A[0]

      SGT R3, R1, R0
      BEQZ R3, end                          ; R3 = (N > 0) | special case N ≤ 0
loop: LW R4, 0(R2)     | SUBI R1, R1, #1   ; R4 = A[i] | N--
      SLT R5, R4, R0   | ADDI R2, R2, #4   ; R5 = (A[i] < 0) | R2 = &A[i+1]
      BEQZ R5, next    |                   ; skip if (A[i]≥0)
      SUB R4, R0, R4   |                   ; A[i] = -A[i]
      SW R4, -4(R2)    |                   ; store updated value of A[i]
next: BNEZ R1, loop    |                   ; continue if N > 0
end:
```

Average Number of Cycles: ½ × (6 + 4) = 5

```
; SOLUTION #2

      SGT R3, R1, R0
      BEQZ R3, end                          ; R3 = (N > 0) | special case N ≤ 0
```

```
loop: LW R4, 0(R2)     | SUBI R1, R1, #1  ; R4 = A[i] | N--
      SLT R5, R4, R0   | ADDI R2, R2, #4  ; R5 = (A[i] < 0) | R2 = &A[i+1]
      BEQZ R5, next    | SUB R4, R0, R4   ; skip if (A[i]≥0) | A[i] = -A[i]
      SW R4, -4(R2)    |                  ; store updated value of A[i]
next: BNEZ R1, loop    |                  ; continue if N > 0
end:
```

Average Number of Cycles: ½ × (5 + 4) = 4.5

NOTE: Although this solution minimizes code size and average number of cycles per element for this loop, it causes extra work because it subtracts regardless of whether it has to or not.

Problem 4.B

Ben wants to remove the data-dependent branches in the assembly code by using predication. He proposes a new set of predicated instructions as follows:

1) Augment the ISA with a set of 32 predicate bits `P0-P31`.
2) Every standard non-control instruction now has a predicated counterpart, with the following syntax:

```
(pbit1) OPERATION1  ; (pbit2) OPERATION2
```

(Execute the first operation of the VLIW instruction if pbit1 is set and execute the second operation of the VLIW instruction if pbit2 is set.)

3) Include a set of compare operations that conditionally set a predicate bit:

```
CMPLTZ pbit,reg     ; set pbit if reg < 0
CMPGEZ pbit,reg     ; set pbit if reg >= 0
CMPEQZ pbit,reg     ; set pbit if reg == 0
CMPNEZ pbit,reg     ; set pbit if reg != 0
```

Eliminate all forward branches from Part A with the new predicated operations. Try to optimize your code but do not use software pipelining or loop unrolling.

What is the average number of cycles per element for this new loop? Assume that the predicate-setting compares have single cycle latency (i.e., behave similarly to a regular ALU instruction including full bypassing of the predicate bit).

```
      SGT R3, R1, R0
      BEQZ R3, end                          ; R3 = (N > 0) | if N ≤ 0
loop: LW R4, 0(R2)          | SUBI R1, R1, #1 ; R4 = A[i] | N--
      CMPLTZ P0, R4         | ADDI R2, R2, #4 ; P0 = (A[i]<0) | R2 = &A[i+1]
```

```
      (P0) SUB R4, R0, R4     |                   ; A[i] = -A[i]
      (P0) SW R4, -4(R2)      | BNEZ R1, loop     ; store updated value of A[i]
end:                                              ; continue if N > 0
```

Average Number of Cycles: ½ × (4 + 4) = 4 Cycles

Problem 4.C

Unroll the predicated VLIW code to perform two iterations of the original loop before each backwards branch. You should use software pipelining to optimize the code for both performance and code density. What is the average number of cycles per element for large N?

```
; Initial Conditions:
;     R1 = N
;     R2 = &A[i]

R3 = N > 0
R4 = A[i]
R5 = N odd
R6 = A[i+1]

      SGT R3, R1, R0
      BEQZ R3, end            | ANDI R5, R1, #1
      BEQZ R5, loop           | LW R4, 0(R2)
      CMPLTZ P0, R4           | SUBI R1, R1, #1
      ADDI R2, R2, #4         | (P0) SUB R4, R0, R4
      (P0) SW R4, -4(R2)      | BEZ R1, end

loop: LW R4, 0(R2)            | SUBI R1, R1, #2
      CMPLTZ P0, R4           | LW R6, 4(R2)
      (P0) SUB R4, R0, R4     | CMPLTZ P1, R6
      (P0) SW R4, 0(R2)       | (P1) SUB R6 R0, R6
                              | (P1) SW R6 4(R2)
      ADDI R2, R2, #8         | BNEZ R1, loop
end:
```

Average Number of Cycles: 6 for 2 elements = 3 cycles per element

Problem 4.D

Now Ben wants to work with a vector processor with two lanes, each of which has a single-cycle ALU and a vector load-store unit. Write-back to the vector register file takes a single cycle. Assume for this part that each vector register has exactly N elements.

Ben can also eliminate branches from his code by using vector masks. He wants to introduce a vector mask register as follows:

1) Augment the ISA with a vector mask register, `VM`.
2) Every vector instruction now writes back only to elements which have the corresponding bit in the mask register set
3) Include a set of compare operations that conditionally set a mask register:

`S--V`	`V1,V2`	Compare the elements (`EQ`,`NE`,`GT`,`LT`,`GE`,`LE`) in `V1` and `V2`. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (`VM`). The instruction `S--SV` performs the same compare but using a scalar value as one operand.
`S--SV`	`F0,V1`	

Vectorize Ben's C loop, and replace all branches using vector masks. What is the average number of cycles per element for this loop in the steady state for a very large value of N?

```
; Initial Conditions:
;     R1 = N
;     R2 = &A[i]

      L.D F0, #0
      MTC1 VLR R1             # operate on all N elements
      LV V1, R2               # load A
      SLTSV.D V1, F0          # setup the mask vector
      SUBSV.D V1, F0, V1      # negate appropriate elements
      SV R2, V1               # store back changes
```

Average Number of Cycles: ≈ (N/2 + N/2) cycles / N elements ≈ 1 cycle per element (assuming chaining). Because there is only one ALU per lane, only the LV and the SLTSV can be chained together, and the SUBSV and the SV can be chained together. Execution time (per element) of the other instructions is negligible when N is large.

Problem 4.E

Modify the code from Part D to handle the case when each vector register has *m* elements, where *m* may be less than N and is not necessarily a factor of N.

```
; assume N = known array length
; Initial Conditions:
;     R1 = N
;     R2 = &A[i]

      L.D F0, #0
      ANDI R3, R1, (m-1)      # get N%m - assume m is a power of 2
      MTC1 VLR R3             # operate on first N%m elements
      LV V1, R2               # load A
      SLTVS.D V1, F0          # setup the mask vector
      SUBSV.D V1, F0, V1      # negate appropriate elements
      SV R2, V1               # store back changes

      SUB R1, R1, R3          # decrease i by N%m (i is divisible by m now)
      SLLI R3, R3, #2         # R3=R3*4 (we're counting i down)
      ADDI R2, R2, R3         # advance A pointer
      BEQZ R1, end            # i == 0 -> done
      ADDI R3, R0, m
      MTC1 VLR R3             # operate on all elements

loop:
      CVM
      LV V1, R2               # load A
      SLTVS.D V1, F0          # setup the mask vector
      SUBSV.D V1, F0, V1      # negate appropriate elements
      SV R2, V1               # store back changes
      ADDI R2, R2, (m*4)      # advance A pointer
      SUB R1, R1, m           # decrease i by m
      BNEZ R1, loop           # done?

end:
      CVM
```

Problem 5: 64-bit Virtual Memory

This problem examines page tables in the context of processors with a 64-bit addressing.

Problem 5.A **Single level page tables**

For a computer with 64-bit virtual addresses, how large is the page table if only a single-level page table is used? Assume that each page is 4KB, that each page table entry is 8 bytes, and that the processor is byte-addressable.

12 bits are needed to represent the 4KB page. There are 64-12=52 bits in a VPN. Thus, there are 2^{52} PTEs. Each is 8 bytes. $2^{52} * 2^3 = 2^{55}$.

Problem 5.B **Let's be practical**

The MIPS R10000 does something similar. Because a 64-bit address is unnecessarily large, only the low 44 address bits are translated. This also reduces the cost of TLB and cache tag arrays. The high two virtual address bits (bits 63:62) select between user, supervisor, and kernel address spaces. The intermediate address bits (61:44) must either be all zeros or all ones, depending on the address region.

How large is a single-level page table that would support MIPS R10000 addresses? Assume that each page is 4KB, that each page table entry is 8 bytes, and that the processor is byte-addressable.

2^2 segments * $2^{(44-12)}$ virtual pages = 2^{34} PTEs. $2^3*2^{34} = \mathbf{2^{37}}$ bytes.
We can also accept $\mathbf{2^{38}}$ since the problem didn't clearly state that bits 61:44 are just copies of bit 43. Or even 3 segments * $2^{(44-12)}$ virtual pages = $2^{33} + 2^{32}$ PTEs. $2^3*(2^{33}+ 2^{32}) = \mathbf{2^{36} + 2^{35}}$ bytes since we only specified 3 valid segments. Or $\mathbf{2*(2^{36} + 2^{35})}$ if you allow 61:44 to differ from bit 43.

Problem 5.C **Page table overhead**

A three-level hierarchical page table can be used to reduce the page table size. Suppose we break up the 44-bit virtual address (VA) as follows:

VA[43:33]	VA[32:22]	VA[21:12]	VA[11:0]
1st level index	2nd level index	3rd level index	Page offset

If page table overhead is defined as (in bytes):

$$\frac{\text{PHYSICAL MEMORY USED BY PAGE TABLES FOR A USER PROCESS}}{\text{PHYSICAL MEMORY USED BY THE USER CODE, HEAP, AND STACK}}$$

Remember that a complete page table (1024 or 2048 PTEs) is allocated even if only one PTE is used. Assume a large enough physical memory that no pages are ever swapped to disk. Use 64-

bit PTEs. What is the smallest possible page table overhead for the three-level hierarchical scheme?

The smallest possible page table overhead occurs when all pages are resident in memory. In this case, the overhead is

$8(2^{11} + 2^{11}*2^{11} + 2^{11}*2^{11}*2^{10}) / 2^{44} \approx 2^{35} / 2^{44} \approx 1 / 2^{9}$

Assume that once a user page is allocated in memory, the whole page is considered to be useful. What is the largest possible page table overhead for the three-level hierarchical scheme?

The largest possible page table overhead occurs when only one data page is resident in memory. In this case, we need 1 L0 page table, 1 L1 page table, 1 L2 page table in order to get data page. Thus the overhead is:

$8(2^{11} + 2^{11} + 2^{10}) / 2^{12} = 10$

Problem 5.D **PTE Overhead**

The MIPS R10000 uses a 40 bit physical address. The physical translation section of the TLB contains the physical page number (also known as PFN), one "valid," one "dirty," and three "cache status" bits.

What is the minimum size of a PTE assuming all pages are 4KB?

PPN is 40-12=28 bits. 28+1+1+3=33 bits.

MIPS/Linux stores each PTE in a 64 bit word. How many bits are wasted if it uses the minimum size you have just calculated?

31. It turns out that some of the "wasted" space is recovered by the OS to do bookkeeping, but not much.

Problem 5.E **Page table implementation**

The following comment is from the source code of MIPS/Linux and, despite its cryptic terminology, describes a three-level page table.

```
/*
 * Each address space has 2 4K pages as its page directory, giving 1024
 * 8 byte pointers to pmd tables. Each pmd table is a pair of 4K pages,
 * giving 1024 8 byte pointers to page tables. Each (3rd level) page
 * table is a single 4K page, giving 512 8 byte ptes.
 *
 * /
```

Assuming 4K pages, how long is each index?

Index	Length (bits)
Top-level (“page directory”)	*10*
2nd-level	*10*
3rd-level	*9*

Problem 5.F **Variable Page Sizes**

A TLB may have a *page mask* field that allows an entry to map a page size of any power of four between 4KB and 16MB. The page mask specifies which bits of the virtual address represent the page offset (and should therefore not be included in translation). What are the maximum and minimum reach of a 64-entry TLB using such a mask?

*Minimum = 4KB * 64 = 256KB*
*Maximum = 16MB * 64 = 1GB*