ECE 475 | PS# 5 | David Wentzlaff

Problem #1)

Note: we assume that 3 stall cycles after load means load has latency of 4.

Processor A (Other schedules are possible if we assume non-greedy schedulers)

Time		
1	T1 LD R1	T2 LD R1
2	T3 LD R1	T4 LD R1
3	T1 LD R2	T2 LD R2
4	T3 LD R2	T4 LD R2
15	T1 LD R8	T2 LD R8
16	T3 LD R8	T4 LD R8
17	T1 BEQ	T2 BEQ
18	T3 BEQ	T4 BEQ
19		
20		
21	T1 BEQ	T2 BEQ
22	T3 BEQ	T4 BEQ
23		
24		
45	T1 BEQ	T2 BEQ
46	T3 BEQ	T3 BEQ
47		
48		
49	T1 DADDIU	T2 DADDIU
50	T3 DADDIU	T4 DADDIU
51	T1 BLT	T2 BLT
52	T3 BLT	T4 BLT
53		
54		

Processor A takes 54 × 2 cycles = 108 cycles

Processor B)

time					
1	T1	LD R1	LD R2	LD R3	LD R4
2	T1	LD R5	LD R6	LD R7	LD R8
3	T1	BEQ			
4	T2	LD R1	LD	LD	LD
5	T2	LD	LD	LD	LD
6	T2	BEQ			
10	T3	LD	LD	LD	LD
11	T3	LD	LD	LD	LD
12	T3	BEQ			
13	T1	BEQ			
14	T2	BEQ			
15	T3	BEQ			
16	T4	BEQ			
37	T1	BEQ			
38	T2	BEQ			
39	T3	BEQ			
40	T4	BEQ			
41	T1	DADDIU			
42	T2	DADDIU			
43	T3	DADDIU			
44	T4	DADDIU			
45	T1	BLT			
46	T2	BLT			
47	T3	BLT			
48	T4	BLT			
49					
50					
51					

If we count no overlap of last BLT for T4 with other work, the last loop iteration takes 3 extra cycles. 48 + 51 = 99 cycles

If overlap, 2 × 48 = 96 cycles.

Processor C)

Time		
1	T1	LD LD LD LD LD LD LD LD
2		
3		
4		
5	T1	BEQ
6		
7		
8		
9	T1	BEQ
10		
11		
12		
33	T1	BEQ
34		
35		
36		
37	T1	DADDIU
38	T1	BLT
39		
40		
41		

$(41 \times 2) + \underbrace{1}_{\text{next loop iteration cache miss}} = 83$ cycles $\times 4$ threads $= 332$ cycles

But, we don't need T4's last LD therefore $332 - 4 = 331$ cycles

ELE 475 PS#5 Solution
Problem # 2:

One of the interesting things about compare and exchange is that atomically operates on a single memory address, but does not have any place to store a lock variable because the locking is implicit. Hence, if you want to build compare and exchange out of test and set, you will need another address. Therefore below, we create a struct which contains both a lock and the value.

```
typedef struct cae_t
{
   int value;
   int lock;  // 1 denotes locked
} cae_t;

// prototype of test_and_set
// old value returned
int test_and_set(int * mem);

// returns 1 if compare success
// returns 0 if compare failure
int compare_and_exchange(cae_t * mem, int compare_value, int swap_value)
{
   int ret_value = 0;
   // grab lock
   while(test_and_set(&(mem->lock)){}
   if(compare_value == mem->value)
   {
      mem->value = swap_value;
      ret_value = 1;
   }
   else
   {
      ret_value = 0;
   }
   // release lock
   mem->lock = 0;
}
```

ELE 475 PS#5 Solution

Problem # 3:

We first investigate the final value for ‘i’. Because thread T3 only has one memory operation, that memory operation is either ordered before or after the store to ‘i’ in thread T1, therefore `i` is either 30 or 100 and these outcomes are valid with all outcomes of ‘j’.

Now we investigate valid values for ‘j’.

Case 1:

```
T2 LW R6, 0(j) j = 10
T2 SW R7, 0(j) j = 109
T1 LW R2, 0(j) j = 109
T1 SW R2, 0(j) j = 109
```

Case 2:

```
T2 LW R6, 0(j) j = 10
T1 LW R2, 0(j) j = 10
T2 SW R7, 0(j) j = 109
T1 SW R2, 0(j) j = 10
```

Case 2:

```
T2 LW R6, 0(j) j = 10
T1 LW R2, 0(j) j = 10
T1 SW R2, 0(j) j = 10
T2 SW R7, 0(j) j = 109
```

Case 3:

```
T1 LW R2, 0(j) j = 10
T2 LW R6, 0(j) j = 10
T2 SW R7, 0(j) j = 109
T1 SW R2, 0(j) j = 10
```

Case 4:

```
T1 LW R2, 0(j) j = 10
T2 LW R6, 0(j) j = 10
T1 SW R2, 0(j) j = 10
T2 SW R7, 0(j) j = 109
```

Case 5:

```
T1 LW R2, 0(j) j = 10
T1 SW R2, 0(j) j = 10
T2 LW R6, 0(j) j = 10
T2 SW R7, 0(j) j = 109
```

Therefore, valid sequentially consistent outcomes are {i, j} = {30, 10}, {30, 109}, {100, 10}, and {100, 109}

ELE 475   PS#5   Solution

Problem # 4:

```
// Assumes that the input and output arrays are created
// outside of the function.  Note that the output lock array is assumed to be
// initialized to 1 (this allows for a mutex)
#include <thread.h>
#include <stdlib.h>
#define MAX_VALUE 1023
#define NUM_THREADS 100
#define INPUT_ARRAY_ELEMENTS (1024*1024*512)  // 512 million entries

struct thread_data{
   int input_array_size;
   int * input_array;
   int * output_array;
   int * output_lock_array;
};

struct thread_data thread_data_array[NUM_THREADS];

void function(int input_array_size, int * input_array, int * output_array,
int * output_lock_array)
{
   int counter;
   for(counter = 0; counter < input_array_size; counter++)
   {
      assert(input_array[counter] <= MAX_VALUE);
      assert(input_array[counter] >= 0);
      P(&output_lock_array[counter]);
      output_array[input_array[counter]]++;
      V(&output_lock_array[counter]);
   }
}

void function_starter(void * thread_args)
{
    struct thread_data * temp_data;
    temp_data = (struct thread_data *) thread_args;
    function(temp_data->input_array_size, temp_data->input_array, temp_data-
>output_array, temp_data->output_lock_array);
    pthread_exit();
}
```

```
int main()
{
   int counter;
   pthread_t threads[NUM_THREADS];
   int * input_array;
   input_array = malloc(INPUT_ARRAY_ELEMENTS*sizeof(int));
   // this would be a good place to read a file into input_array
   int * output_array = malloc((MAX_VALUE + 1) * sizeof(int));
   int * output_lock_array = malloc((MAX_VALUE + 1) * sizeof(int));
   for(counter = 0; counter <= MAX_VALUE; counter++);
   {
      output_array = 0;
      output_lock_array = 1;
   }
   for(counter = 0; counter < 100; counter++)
   {
       thread_data_array[counter].input_array_size = INPUT_ARRAY_ELEMENTS /
NUM_THREADS;
       thread_data_array[counter].input_array =
&input_array[INPUT_ARRAY_ELEMENTS / NUM_THREADS * counter];
       thread_data_array[counter].output_array = output_array;
       thread_data_array[counter].output_lock_array = output_lock_array;
       pthread_create(&threads[counter], NULL, function_starter, (void
*)thread_data_array[counter]);
   }
   for(counter = 0; counter < 100; counter++)
   {
      void * dummy;
      pthread_join(threads[counter], &dummy);
   }
}
```

No this program will not see 100x performance speedup for two reasons. First, we require more work to be done in setting and clearing the semaphores. Also, we have used a very fine grain locking scheme which minimized the probability that two threads will use the same location in the output array simultaneously, but it is still possible. When there is contention, a thread will be stalled and performance will decrease. Finally, the coherence protocol along with true and false sharing will cause cache lines to move between the caches of the processors. When a processor is waiting for a cache line to be transferred, these cycles are wasted thereby decreasing performance.

ELE 475 PS5 Problem 5 Solution

MSI Protocol

	P1 Cache		P2 Cache		P3 Cache		Notes
Time	Line Address	State	Line Address	State	Line Address	State	
1	0	Shared	All lines Invalid		All lines Invalid		
2	0	Shared	0	Shared	All lines Invalid		
3	0	Shared	0	Shared	0	Shared	
4	0	Shared	0	Shared	0	Shared	
					64	Modified	
5	0	Shared	0	Shared	0	Shared	
					64	Modified	
6	0	Shared	0	Shared	0	Shared	
			64	Shared	64	Shared	
7	0	Modified	64	Shared	64	Shared	Line at address 0 is marked invalid in P2 and P3 cache.
8	4096	Shared	64	Shared	64	Shared	Address 4100 (line starts at 4096) aliases with line 0
9	4096	Modified	64	Shared	64	Shared	
10			64	Shared	64	Shared	Line at address 4096 is invalidated in P1 cache
			4096	Modified			
11			64	Shared	64	Shared	Line at address 4096 and address 0 can exist
			4096	Modified	0	Modified	simultaneously in two different caches.

MESI Protocol

	P1 Cache		P2 Cache		P3 Cache		Notes
Time	Line Address	State	Line Address	State	Line Address	State	
1	0	Exclusive	All lines Invalid		All lines Invalid		
2	0	Shared	0	Shared	All lines Invalid		
3	0	Shared	0	Shared	0	Shared	
4	0	Shared	0	Shared	0	Shared	
					64	Modified	
5	0	Shared	0	Shared	0	Shared	
					64	Modified	
6	0	Shared	0	Shared	0	Shared	
			64	Shared	64	Shared	
7	0	Modified	64	Shared	64	Shared	Line at address 0 is marked invalid in P2 and P3 cache.
8	4096	Exclusive	64	Shared	64	Shared	Address 4100 (line starts at 4096) aliases with line 0
9	4096	Modified	64	Shared	64	Shared	
10			64	Shared	64	Shared	Line at address 4096 is invalidated in P1 cache
			4096	Modified			
11			64	Shared	64	Shared	Line at address 4096 and address 0 can exist
			4096	Modified	0	Modified	simultaneously in two different caches.

ELE 475 PS#5 Solution

Problem # 6:

We assume that each link is bidirectional even though most mesh networks are made of two sets of unidirectional links. For a 4-ary 3-cube, there are 16 links across the bisection.

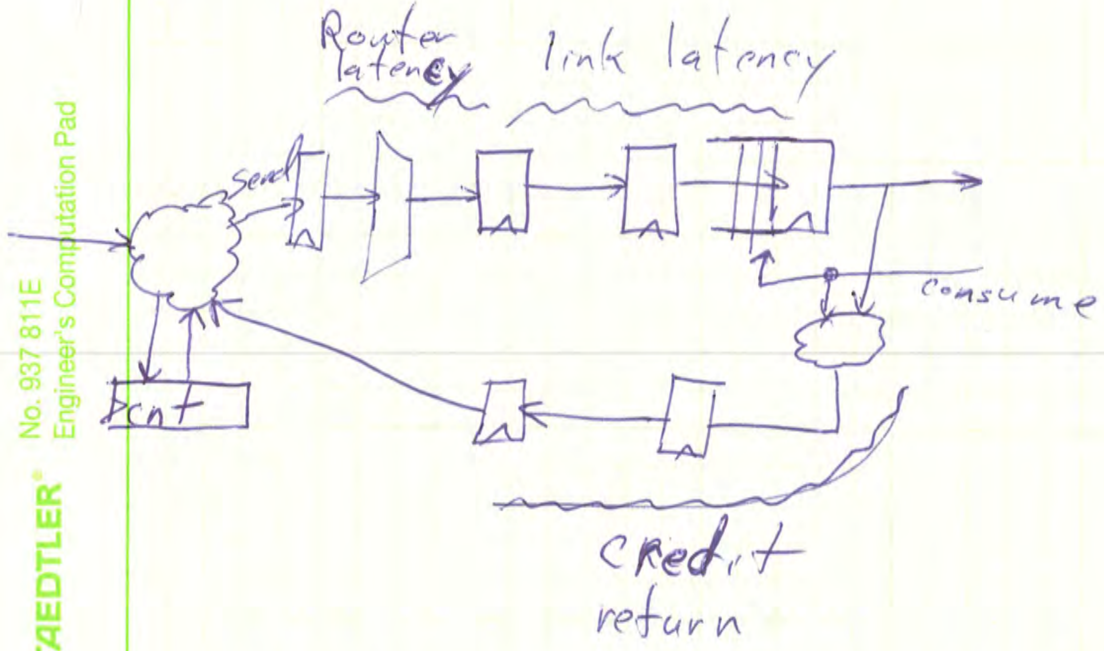
16 links * 32bits/link * 800*10^6 cycles/second = 4.096 Tbps.

Each stage of an omega network has the same number of links, but if we examine one stage where a crossover occurs, we see that in a pipelined manner that only 4 links cross the bisection cut of the machine. We assume that each switch is pipelined. Therefore on a given cycle, only 4 links cross the minimum bisection.

4 links * 64bits/link * 1.2*10^9 cycles/second = 307.2Gbps.

ECE 475 PS#5 Solution

Problem 7)



Credit counter needs to be **6** if we assume above diagram. The wording of problem is a little bit ambiguous. Note that the counter must be large enough to hold 7 values {0, 1, 2, 3, 4, 5, 6}

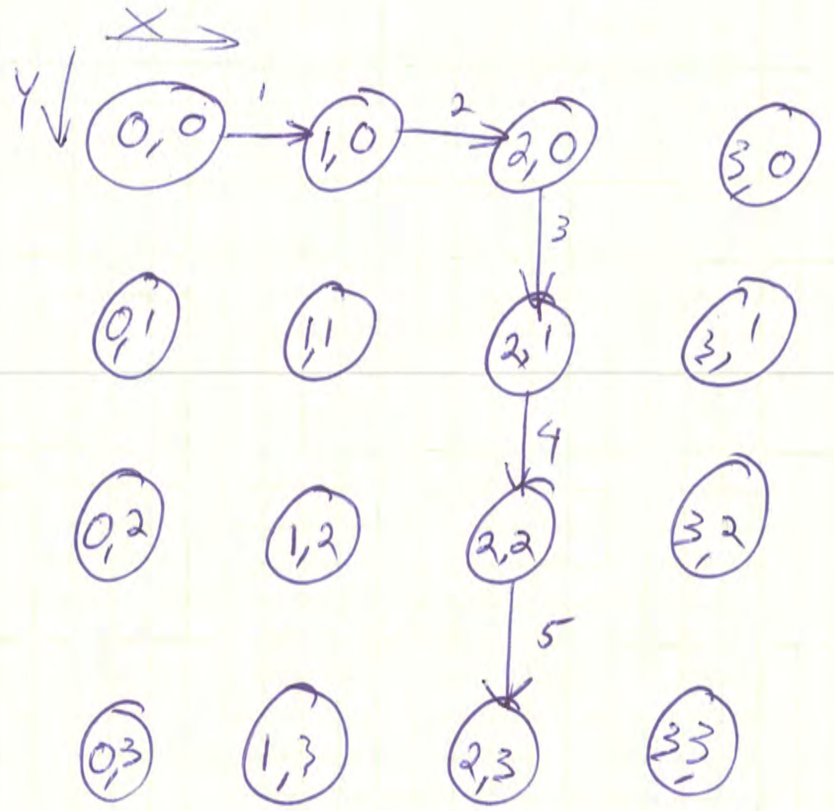
If we need 6 as the maximum number of credits and we only allow the counter to reach 4, we see scenarios such as this assuming that the consumer is reading at full bandwidth. Slower reading can interrupt sending also.

1 2 3 4 S S S 5 6 7 8 S S 9 10 11 12 . . . . .

$\frac{4}{6} = \frac{2}{3}$ of peak bandwidth

ECE 475 PS# 5 David Wentzlaff

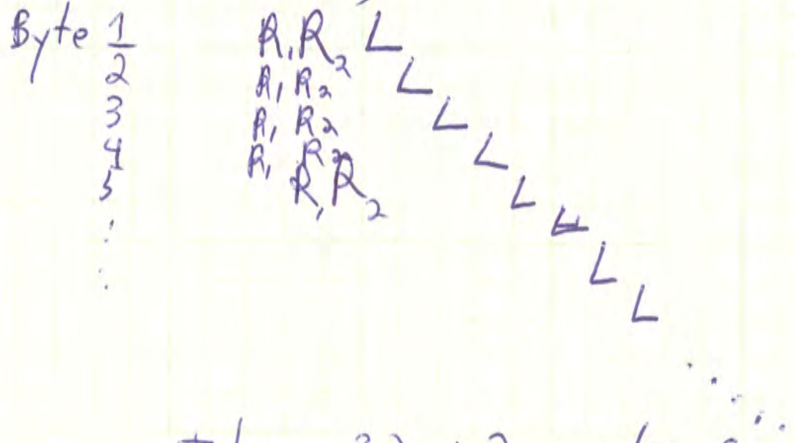
Problem 8)



Store and Forward

Note, we assume pipelined routers that are 4 bytes wide. Also, we assume the store and forward buffer is large enough to hold a whole message.

Crossing one link and router pair

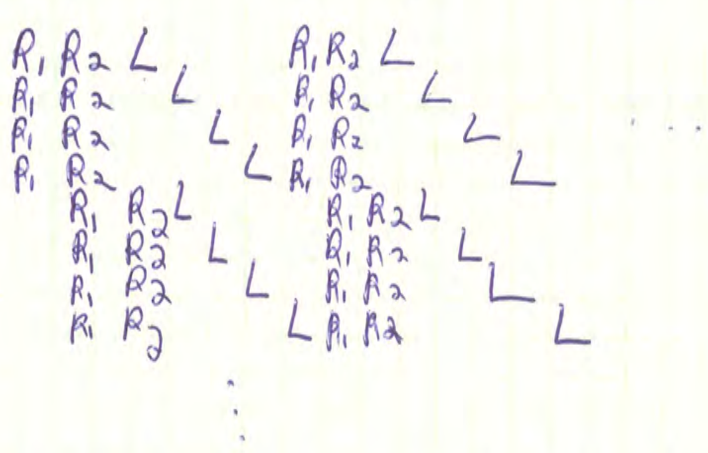


takes 32 + 2 cycles for one hop.

Because Store & Forward, no overlap is possible. We assume that there is no router delay at sender side even though this is unrealistic.

Therefore, 5 network links where each hop takes 34 cycles yields 170 cycles

Wormhole/cut-through



First Flit is recieved at destination after

6 × 5 cycles = 30 cycles

there is 4 cycles for each additional Flit (32-bits)

= 28 additional cycles

Total = 30 + 28 = 58 cycles

ELE 475 PS5 Problem 9 Solution

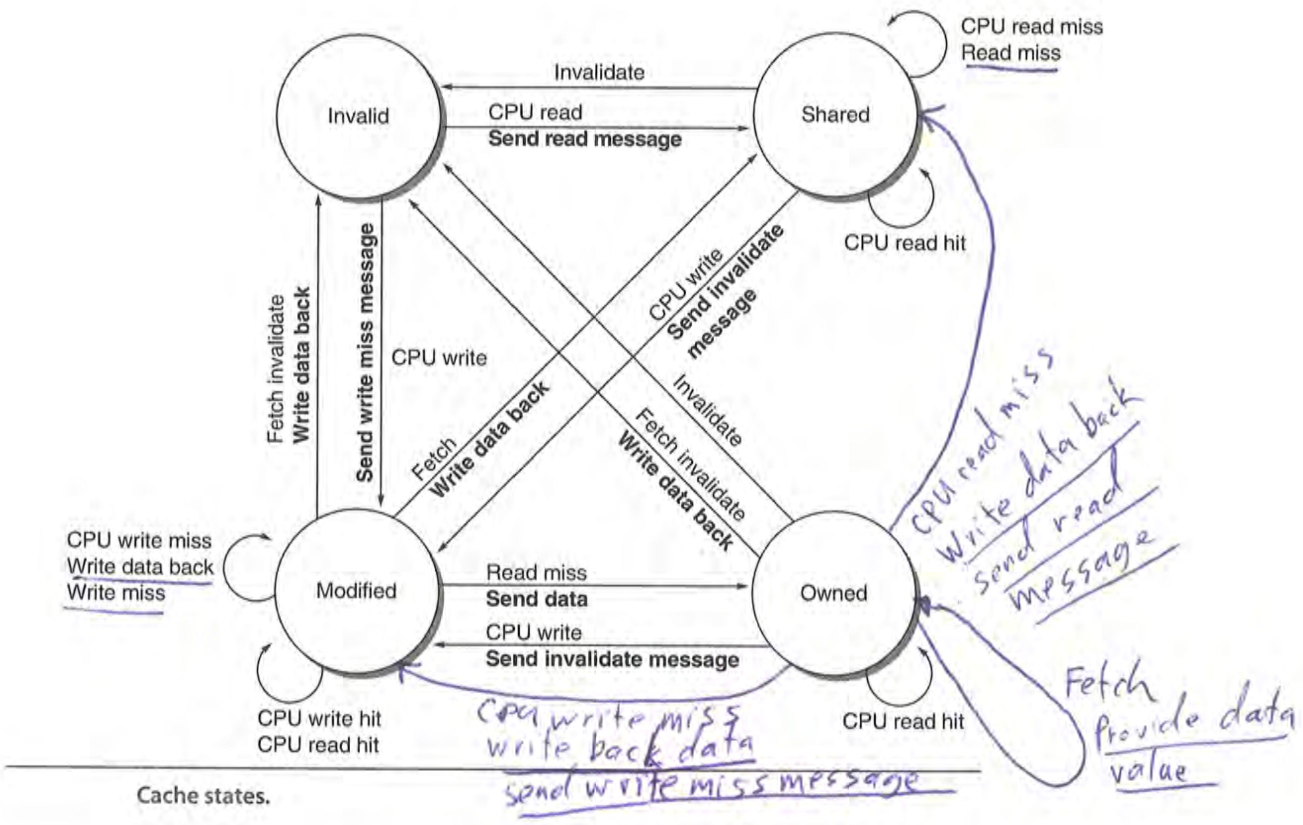
ESU Directory Protocol

Note: We assume that writeback of data notifies the directory, but invalidation due to conflict misses on shared data does not.

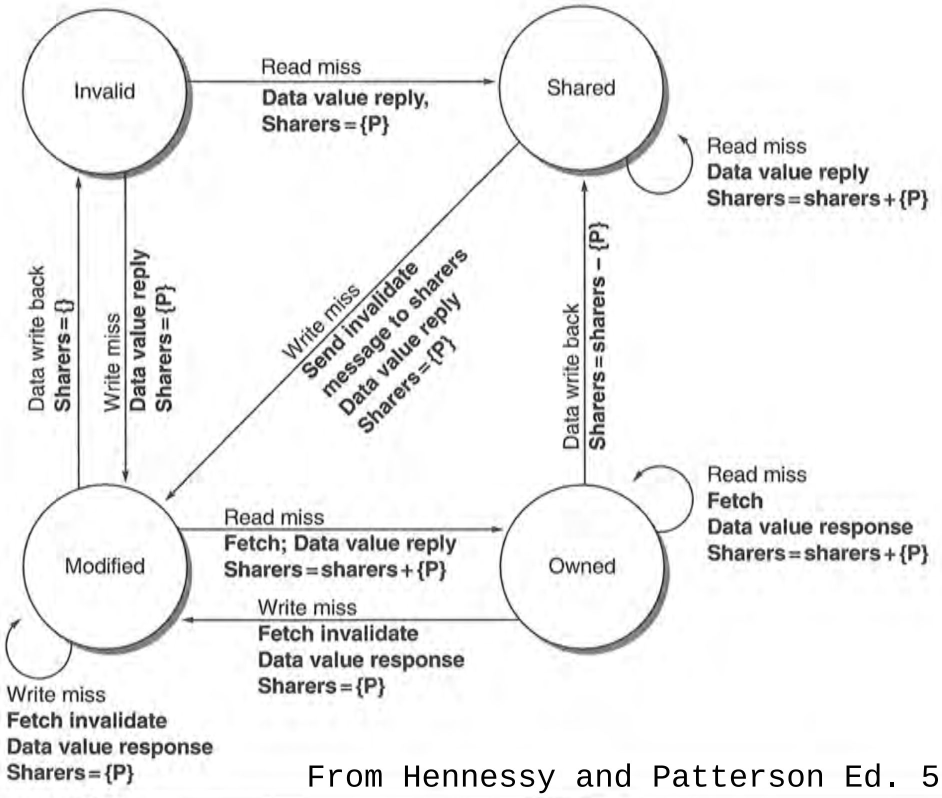
	P1 Cache		P2 Cache		P3 Cache		Directory			
Time	Line Address	State	Line Address	State	Line Address	State	Line Address	State	Share List/Owner	
1	0	Shared	All lines Invalid		All lines Invalid		0	Shared	{P1}	
							All other lines Uncached			
2	0	Shared	0	Shared	All lines Invalid		0	Shared	{P1, P2}	
							All other lines Uncached			
3	0	Shared	0	Shared	0	Shared	0	Shared	{P1, P2, P3}	
							All other lines Uncached			
4	0	Shared	0	Shared	0	Shared	0	Shared	{P1, P2, P3}	
					64	Modified	64	Exclusive	{P3}	
							All other lines Uncached			
5	0	Shared	0	Shared	0	Shared	0	Shared	{P1, P2, P3}	
					64	Modified	64	Exclusive	{P3}	
							All other lines Uncached			
6	0	Shared	0	Shared	0	Shared	0	Shared	{P1, P2, P3}	
			64	Shared	64	Shared	64	Shared	{P2, P3}	
							All other lines Uncached			
7	0	Modified	64	Shared	64	Shared	0	Exclusive	{P1}	
							64	Shared	{P2, P3}	
							All other lines Uncached			
8	4096	Shared	64	Shared	64	Shared	0	Invalid		Writeback
							64	Shared	{P2, P3}	updates
							4096	Shared	{P1}	directory
9	4096	Modified	64	Shared	64	Shared	0	Invalid		
							64	Shared	{P2, P3}	
							4096	Exclusive	{P1}	
10			64	Shared	64	Shared	0	Invalid		
			4096	Modified			64	Shared	{P2, P3}	
							4096	Exclusive	{P2}	
11			64	Shared	64	Shared	0	Exclusive	{P3}	
			4096	Modified	0	Modified	64	Shared	{P2, P3}	
							4096	Exclusive	{P2}	

ELE 475 PS# 5 David Wentzlaff

Problem #10)



Cache states.



Directory states.