

## Problem M2.1: Cache Access-Time & Performance

*This problem requires the knowledge of Handout #6 (Cache Implementations) and Lecture 7. Please, read these materials before answering the following questions.*

Ben is trying to determine the best cache configuration for a new processor. He knows how to build two kinds of caches: direct-mapped caches and 4-way set-associative caches. The goal is to find the better cache configuration with the given building blocks. He wants to know how these two different configurations affect the clock speed and the cache miss-rate, and choose the one that provides better performance in terms of average latency for a load.

### Problem M2.1.A

### Access Time: Direct-Mapped

Now we want to compute the access time of a direct-mapped cache. We use the implementation shown in Figure H6-A in Handout #6. Assume a 128-KB cache with 8-word (32-byte) cache lines. The address is 32 bits, and the two least significant bits of the address are ignored since a cache access is word-aligned. The data output is also 32 bits, and the MUX selects one word out of the eight words in a cache line. Using the delay equations given in Table M2.1-1, fill in the column for the direct-mapped (DM) cache in the table. *In the equation for the data output driver, 'associativity' refers to the associativity of the cache (1 for direct-mapped caches, A for A-way set-associative caches).*

Component	Delay equation (ps)		DM (ps)	SA (ps)
Decoder	$200 \times (\# \text{ of index bits}) + 1000$	Tag		
		Data		
Memory array	$200 \times \log_2 (\# \text{ of rows}) + 200 \times \log_2 (\# \text{ of bits in a row}) + 1000$	Tag		
		Data		
Comparator	$200 \times (\# \text{ of tag bits}) + 1000$			
N-to-1 MUX	$500 \times \log_2 N + 1000$			
Buffer driver	2000			
Data output driver	$500 \times (\text{associativity}) + 1000$			
Valid output driver	1000			

Table M2.1-1: Delay of each Cache Component

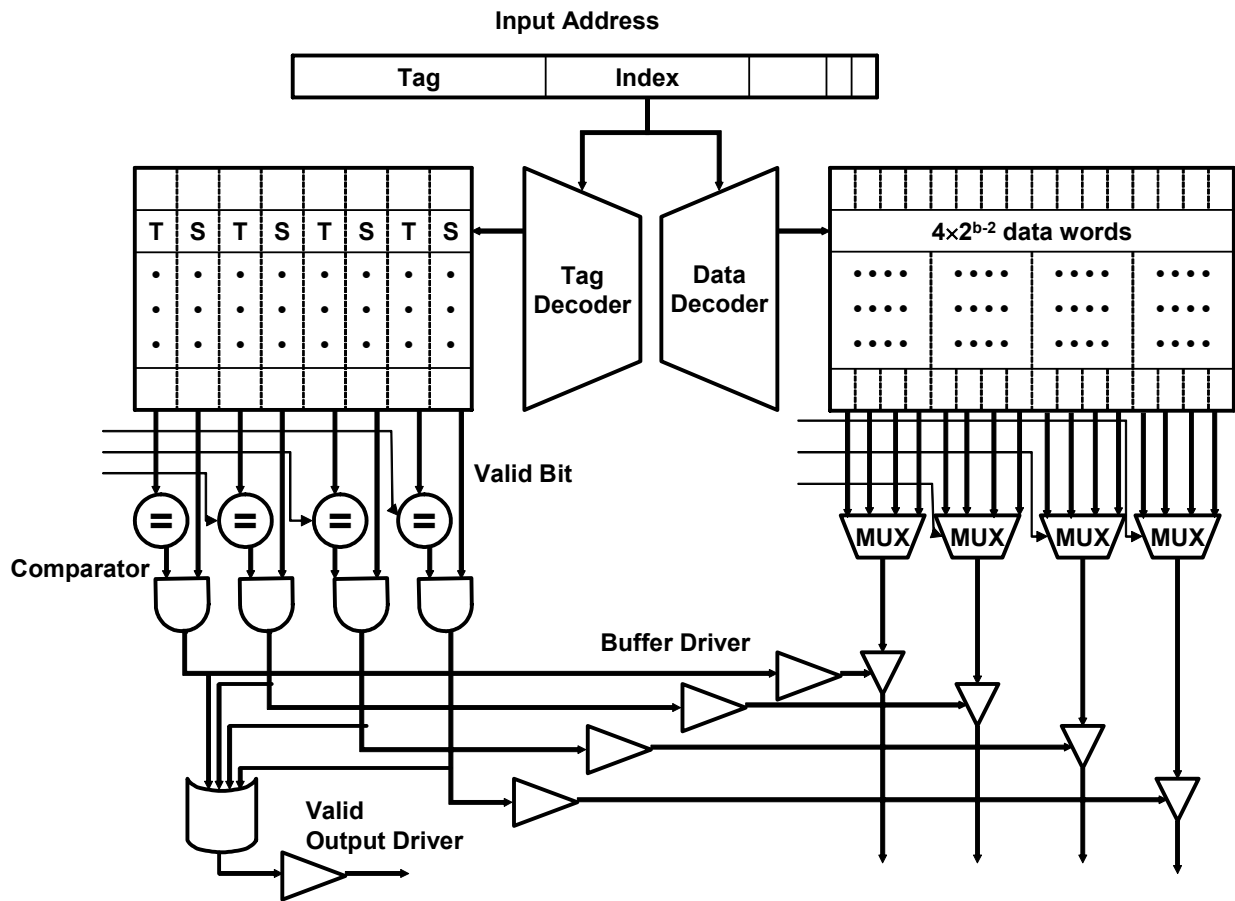
What is the critical path of this direct-mapped cache for a cache read? What is the access time of the cache (the delay of the critical path)? To compute the access time, assume that a 2-input gate (AND, OR) delay is 500 ps. If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?

**Problem M2.1.B**

**Access Time: Set-Associative**

We also want to investigate the access time of a set-associative cache using the 4-way set-associative cache in Figure H6-B in Handout #6. Assume the total cache size is still 128-KB (each way is 32-KB), a 4-input gate delay is 1000 ps, and all other parameters (such as the input address, cache line, etc.) are the same as part M2.1.A. Compute the delay of each component, and fill in the column for a 4-way set-associative cache in Table M2.1-1.

What is the critical path of the 4-way set-associative cache? What is the access time of the cache (the delay of the critical path)? What is the main reason that the 4-way set-associative cache is slower than the direct-mapped cache? If the CPU clock is 150 MHz, how many CPU cycles does a cache access take?



**Problem M2.1.C**

**Miss-rate analysis**

Now Ben is studying the effect of set-associativity on the cache performance. Since he now knows the access time of each configuration, he wants to know the miss-rate of each one. For the miss-rate analysis, Ben is considering two small caches: a direct-mapped cache with 8 lines with 16 bytes/line, and a 4-way set-associative cache of the same size. For the set-associative cache, Ben tries out two replacement policies – least recently used (LRU) and round robin (FIFO).

Ben tests the cache by accessing the following sequence of hexadecimal byte addresses, starting with empty caches. For simplicity, assume that the addresses are only 12 bits. Complete the following tables for the direct-mapped cache and both types of 4-way set-associative caches showing the progression of cache contents as accesses occur (in the tables, ‘inv’ = invalid, and the column of a particular cache line contains the {tag,index} contents of that line). *You only need to fill in elements in the table when a value changes.*

<b>D-map</b> Address	line in cache								hit?
	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									
102									
361									
204									
114									
1A4									
177									
301									
206									
135									

<b>D-map</b>	
<b>Total Misses</b>	
<b>Total Accesses</b>	

Address	4-way LRU								LRU	
	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					11	13			no	
202	20								no	
1A3										
102										
361										
204										
114										
1A4										
177										
301										
206										
135										

4-way LRU	
Total Misses	
Total Accesses	

Address	4-way FIFO								FIFO	
	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										
102										
361										
204										
114										
1A4										
177										
301										
206										
135										

4-way FIFO	
Total Misses	
Total Accesses	

**Problem M2.1.D**

**Average Latency**

---

Assume that the results of the above analysis can represent the average miss-rates of the direct-mapped and the 4-way LRU 128-KB caches studied in M2.1.A and M2.1.B. What would be the average memory access latency in CPU cycles for each cache (assume that a cache miss takes 20 cycles)? Which one is better? For the different replacement policies for the set-associative cache, which one has a smaller cache miss rate for the address stream in M2.1.C? Explain why. Is that replacement policy always going to yield better miss rates? If not, give a counter example using an address stream.

## Problem M2.2: Pipelined Cache Access

*This problem requires the knowledge of Lecture 7 and 8. Please, read these materials before answering the following questions. You may also want to take a look at pipeline lectures (Lecture 5 and 6) if you do not feel comfortable with the topic.*

### **Problem M2.2.A**

---

Ben Bitdiddle is designing a five-stage pipelined MIPS processor with separate 32 KB direct-mapped primary instruction and data caches. He runs simulations on his preliminary design, and he discovers that a cache access is on the critical path in his machine. After remembering that pipelining his processor helped to improve the machine's performance, he decides to try applying the same idea to caches. Ben breaks each cache access into three stages in order to reduce his cycle time. In the first stage the address is decoded. In the second stage the tag and data memory arrays are accessed; for cache reads, the data is available by the end of this stage. However, the tag still has to be checked—this is done in the third stage.

After pipelining the instruction and data caches, Ben's datapath design looks as follows:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check	Instruction Decode & Register Fetch	Execute	D- Cache Address Decode	D- Cache Array Access	D- Cache Tag Check	Write- back
------------------------------	----------------------------	-------------------------	--	---------	----------------------------------	--------------------------------	-----------------------------	----------------

Alyssa P. Hacker examines Ben's design and points out that the third and fourth stages can be combined, so that the instruction cache tag check occurs in parallel with instruction decoding and register file read access. If Ben implements her suggestion, what must the processor do in the event of an instruction cache tag mismatch? Can Ben do the same thing with load instructions by combining the data cache tag check stage with the write-back stage? Why or why not?

### **Problem M2.2.B**

---

Alyssa also notes that Ben's current design is flawed, as using three stages for a data cache access won't allow writes to memory to be handled correctly. She argues that Ben either needs to add a fourth stage or figure out another way to handle writes. What problem would be encountered on a data write? What can Ben do to keep a three-stage pipeline for the data cache?

### **Problem M2.2.C**

---

With help from Alyssa, Ben streamlines his design to consist of eight stages (the handling of data writes is not shown):

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back
------------------------	----------------------	--	---------	------------------------	----------------------	-------------------	------------

Both the instruction and data caches are still direct-mapped. Would this scheme still work with a set-associative instruction cache? Why or why not? Would it work with a set-associative data cache? Why or why not?

### **Problem M2.2.D**

---

After running additional simulations, Ben realizes that pipelining the caches was not entirely beneficial, as now the cache access latency has increased. If conditional branch instructions resolve in the Execute stage, how many cycles is the processor's branch delay?

### **Problem M2.2.E**

---

Assume that Ben's datapath is fully-bypassed. When a load is executed, the data becomes available at the end of the D-cache Array Access stage. However, the tag has not yet been checked, so it is unknown whether the data is correct. If the load data is bypassed immediately, before the tag check occurs, then the instruction that depends on the load may execute with incorrect data. How can an interlock in the Instruction Decode stage solve this problem? How many cycles is the load delay using this scheme (assuming a cache hit)?

### **Problem M2.2.F**

---

Alyssa proposes an alternative to using an interlock. She tells Ben to allow the load data to be bypassed from the end of the D-Cache Array Access stage, so that the dependent instruction can execute while the tag check is being performed. If there is a tag mismatch, the processor will wait for the correct data to be brought into the cache; then it will re-execute the load and all of the instructions behind it in the pipeline before continuing with the rest of the program. What processor state needs to be saved in order to implement this scheme? What additional steps need to be taken in the pipeline? Assume that a **DataReady** signal is asserted when the load data is available in the cache, and is set to 0 when the processor restarts its execution (you don't have to worry about the control logic details of this signal). How many cycles is the load delay using this scheme (assuming a cache hit)?

**Problem M2.2.G**

---

Ben is worried about the increased latency of the caches, particularly the data cache, so Alyssa suggests that he add a small, unpipelined cache in parallel with the D-cache. This “fast-path” cache can be considered as another level in the memory hierarchy, with the exception that it will be accessed simultaneously with the “slow-path” three-stage pipelined cache. Thus, the slow-path cache will contain a superset of the data found in the fast-path cache. A read hit in the fast-path cache will result in the requested data being available after one cycle. In this situation, the simultaneous read request to the slow-path cache will be ignored. A write hit in the fast-path cache will result in the data being written in one cycle. The simultaneous write to the slow-path cache will proceed as normal, so that the data will be written to both caches. If a read miss occurs in the fast-path cache, then the simultaneous read request to the slow-path cache will continue to be processed—if a read miss occurs in the slow-path cache, then the next level of the memory hierarchy will be accessed. The requested data will be placed in both the fast-path and slow-path caches. If a write miss occurs in the fast-path cache, then the simultaneous write to the slow-path cache will continue to be processed as normal. The fast-path cache uses a no-write allocate policy, meaning that on a write miss, the cache will remain unchanged—only the slow-path cache will be modified.

Ben’s new pipeline design looks as follows after implementing Alyssa’s suggestion:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	Fast-Path D-Cache Access and Tag Check & Slow Path D-Cache Address Decode	Slow-Path D-Cache Array Access	Slow-Path D-Cache Tag Check	Write-Back
------------------------	----------------------	--	---------	---	--------------------------------	-----------------------------	------------

The number of processor pipeline stages is still eight, even with the addition of the fast-path cache. Since the processor pipeline is still eight stages, what is the benefit of using a fast-path cache? Give an example of an instruction sequence and state how many cycles are saved if the fast-path cache always hits.



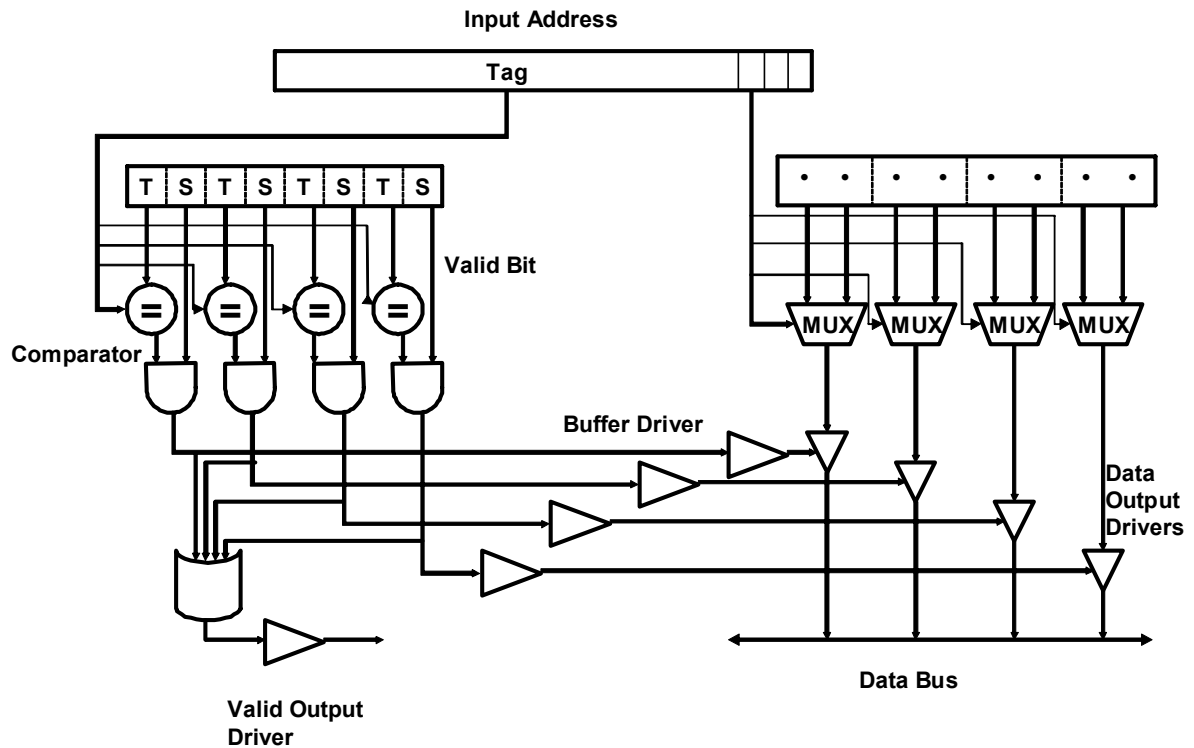
### Problem M2.3: Victim Cache Evaluation

*This problem requires the knowledge of Handout #7 (Victim Cache) and Lecture 7. Please, read these materials before answering the following questions.*

#### Problem M2.3.A

#### Baseline Cache Design

The diagram below shows a 32-Byte fully associative cache with four 8-Byte cache lines. Each line contains two 4-Byte words and has an associated tag and two status bits (valid and dirty). The Input Address is 32-bits and the two least significant bits are assumed to be zero. The output of the cache is a 32-bit word.



Please complete Table M2.3-1 below with delays across each element of the cache. Using the data you compute in Table M2.3-1, calculate the critical path delay through this cache (from when the Input Address is set to when both Valid Output Driver and the appropriate Data Output Driver are outputting valid data).

Component	Delay equation (ps)	FA (ps)
Comparator	$200 \times (\# \text{ of tag bits}) + 1000$	
N-to-1 MUX	$500 \times \log_2 N + 1000$	
Buffer driver	2000	
AND gate	1000	
OR gate	500	
Data output driver	$500 \times (\text{associativity}) + 1000$	
Valid output driver	1000	

**Table M2.3-1**

Critical Path Cache Delay: \_\_\_\_\_

**Problem M2.3.B**

**Victim Cache Behavior**

---

Now we will study the impact of a victim cache on a cache hit rate. Our main L1 cache is a 128 byte, direct mapped cache with 16 bytes per cache line. The cache is word (4-bytes) addressable. The victim cache in Figure H7-A (in Handout #7) is a 32 byte fully associative cache with 16 bytes per cache line, and is also word-addressable. The victim cache uses the first in first out (FIFO) replacement policy.

Please complete Table M2.3-2 on the next page showing a trace of memory accesses. In the table, each entry contains the {tag,index} contents of that line, or “inv”, if no data is present. You should only fill in elements in the table when a value changes. For simplicity, the addresses are only 8 bits.

The first 3 lines of the table have been filled in for you.

For your convenience, the address breakdown for access to the main cache is depicted below.

7	6	4	3	2	1	0
TAG	INDEX		WORD SELECT		BYTE SELECT	

**Problem M2.3.C**

**Average Memory Access Time**

---

Assume **15%** of memory accesses are resolved in the victim cache. If retrieving data from the victim cache takes **5 cycles** and retrieving data from main memory takes **55 cycles**, by how many cycles does the victim cache improve the average memory access time?

Input Address	Main Cache									Victim Cache		
	L0	L1	L2	L3	L4	L5	L6	L7	Hit?	Way0	Way1	Hit?
	inv	inv	inv	inv	inv	inv	inv	inv	-	inv	inv	-
00	0								N			N
80	8								N	0		N
04	0								N	8		Y
A0												
10												
C0												
18												
20												
8C												
28												
AC												
38												
C4												
3C												
48												
0C												
24												

Table M2.3-2

## Problem M2.4: Loop Ordering

*This problem requires the knowledge of Lecture 7. Please, read it before answering the following questions.*

This problem evaluates the cache performances for different loop orderings. You are asked to consider the following two loops, written in C, which calculate the sum of the entries in a 128 by 64 matrix of 32-bit integers:

<b><i>Loop A</i></b>	<b><i>Loop B</i></b>
<pre>sum = 0; for (i = 0; i &lt; 128; i++)   for (j = 0; j &lt; 64; j++)     sum += A[i][j];</pre>	<pre>sum = 0; for (j = 0; j &lt; 64; j++)   for (i = 0; i &lt; 128; i++)     sum += A[i][j];</pre>

The matrix A is stored contiguously in memory in row-major order. Row major order means that elements in the same row of the matrix are adjacent in memory as shown in the following memory layout:

$A[i][j]$  resides in memory location  $[4 * (64 * i + j)]$

Memory Location:

0	4		252	256		$4 * (64 * 127 + 63)$
A[0][0]	A[0][1]	...	A[0][63]	A[1][0]	...	A[127][63]

For *Problem M2.4.A* to *Problem M2.4.C*, assume that the caches are initially empty. Also, assume that only accesses to matrix A cause memory references and all other necessary variables are stored in registers. Instructions are in a separate instruction cache.

**Problem M2.4.A**

---

Consider a 4KB direct-mapped data cache with 8-word (32-byte) cache lines.  
Calculate the number of cache misses that will occur when running Loop A.  
Calculate the number of cache misses that will occur when running Loop B.

**The number of cache misses for Loop A:** \_\_\_\_\_

**The number of cache misses for Loop B:** \_\_\_\_\_

**Problem M2.4.B**

---

Consider a direct-mapped data cache with 8-word (32-byte) cache lines. Calculate the minimum number of cache lines required for the data cache if Loop A is to run without any cache misses other than compulsory misses. Calculate the minimum number of cache lines required for the data cache if Loop B is to run without any cache misses other than compulsory misses.

**Data-cache size required for Loop A:** \_\_\_\_\_  
**cache line(s)**

**Data-cache size required for Loop B:** \_\_\_\_\_  
**cache line(s)**

**Problem M2.4.C**

---

Consider a 4KB fully-associative data cache with 8-word (32-byte) cache lines. This data cache uses a first-in/first-out (FIFO) replacement policy.

Calculate the number of cache misses that will occur when running Loop A.

Calculate the number of cache misses that will occur when running Loop B.

**The number of cache misses for Loop  
A:** \_\_\_\_\_

**The number of cache misses for Loop  
B:** \_\_\_\_\_



## **Problem M2.5: Cache Parameters**

For each of the following statements about making a change to a cache design, circle **True** or **False** and provide a one sentence explanation of your choice. Assume all cache parameters (capacity, associativity, line size) remain fixed except for the single change described in each question. Please provide a one sentence explanation of your answer.

### **Problem M2.5.A**

---

Doubling the line size halves the number of tags in the cache

**True / False**

### **Problem M2.5.B**

---

Doubling the associativity doubles the number of tags in the cache.

**True / False**

### **Problem M2.5.C**

---

Doubling cache capacity of a direct-mapped cache usually reduces conflict misses.

**True / False**

**Problem M2.5.D**

---

Doubling cache capacity of a direct-mapped cache usually reduces compulsory misses.

**True / False**

**Problem M2.5.E**

---

Doubling the line size usually reduces compulsory misses.

**True / False**

## Problem M2.6: Microtags

### Problem M2.6.A

---

Explain in one or two sentences why direct-mapped caches have much lower hit latency (as measured in picoseconds) than set-associative caches of the same capacity.

### Problem M2.6.B

---

A 32-bit byte-addressed machine has an 8KB, 4-way set-associative data cache with 32-byte lines. The following figure shows how the address is divided into tag, index and offset fields. Give the number of bits in each field.

tag	Index	offset
-----	-------	--------

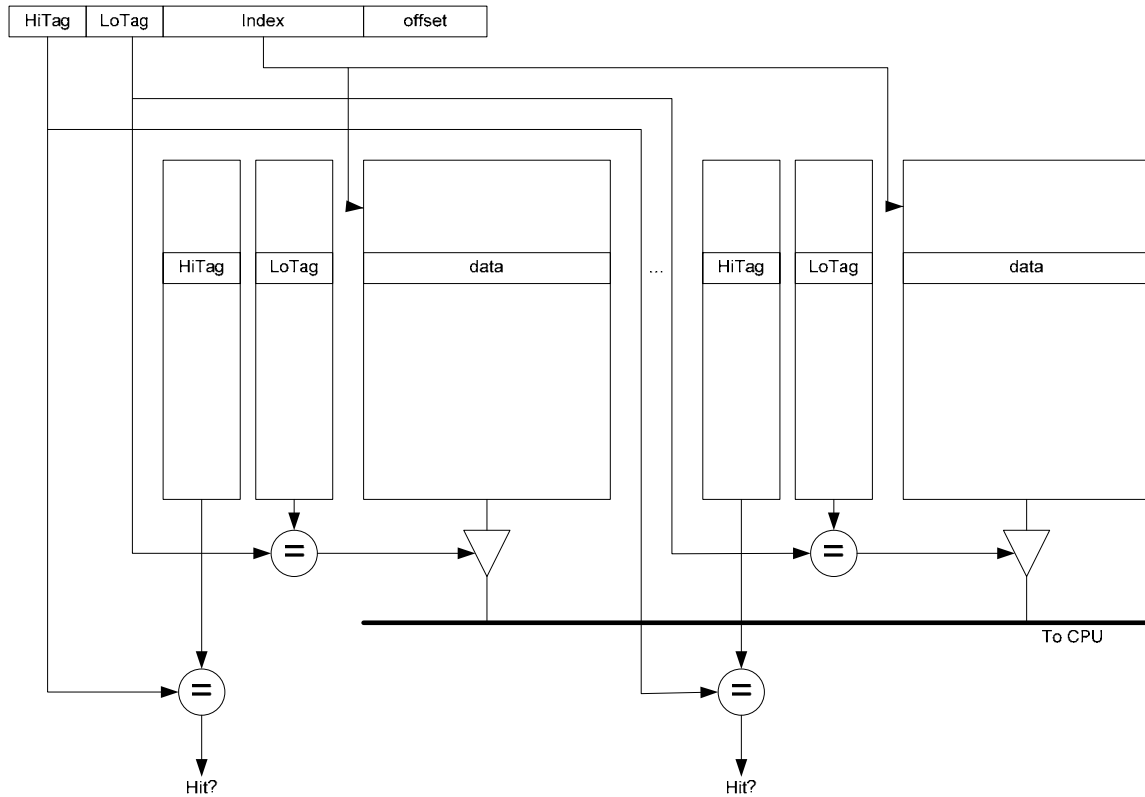
# of bits in the tag: \_\_\_\_\_

# of bits in the index: \_\_\_\_\_

# of bits in the offset: \_\_\_\_\_

### Microtags (for questions M2.6.C – M2.6.H)

Several commercial processors (including the UltraSPARC-III and the Pentium-4) reduce the hit latency of a set-associative cache by using only a subset of the tag bits (a “microtag”) to select the matching way before speculatively forwarding data to the CPU. The remaining tag bits are checked in a subsequent clock cycle to determine if the access was actually a hit. The figure below illustrates the structure of a cache using this scheme.



**Problem M2.6.C**

---

The tag field is sub-divided into a **loTag** field used to select a way and a **hiTag** field used for subsequent hit/miss checks, as shown below.

tag			
<b>hiTag</b>	<b>loTag</b>	index	offset

The cache design requires that all lines within a set have unique loTag fields. In one or two sentences, explain why this is necessary.

**Problem M2.6.D**

---

If the **loTag** field is exactly two bits long, will the cache have greater, fewer, or an equal number of conflict misses as a direct-mapped cache of the same capacity? State any assumptions made about replacement policy.

### **Problem M2.6.E**

---

If the **loTag** field is greater than two bits long, are there any additional constraints on replacement policy beyond those in a conventional 4-way set-associative cache?

### **Problem M2.6.F**

---

Does this scheme reduce the time required to complete a write to the cache? Explain in one or two sentences.

### **Problem M2.6.G**

---

In practice, microtags hold virtual address bits to remove address translation from the critical path, while the full tag check is performed on translated physical addresses. If the **loTag** bits can only hold untranslated bits of the virtual address, what is the largest number of **loTag** bits possible if the machine has a 16KB virtual memory page size? (Assume 8KB 4-way set-associative cache as in Question M2.6.B)

### **Problem M2.6.H**

---

Describe how microtags can be made much larger, to also include virtual address bits subject to address translation. Your design should not require address translation before speculatively forwarding data to the CPU. Your explanation should describe the replacement policy and any additional state the machine must maintain.

## **Problem M2.7: Virtual Memory Bits**

*This problem requires the knowledge of Handout #8 (Virtual Memory Implementation) and Lecture 9. Please, read these materials before answering the following questions.*

In this problem we consider simple virtual memory enhancements.

### **Problem M2.7.A**

---

Whenever a TLB entry is replaced we write the entire entry back to the page table. Ben thinks this is a waste of memory bandwidth. He thinks only a few of the bits need to be written back. For each of the bits explain why or why not they need to be written back to the page table.

With this in mind, we will see how we can minimize the number of bits we actually need in each TLB entry throughout the rest of the problem.

### **Problem M2.7.B**

---

Ben does not like the TLB design. He thinks the TLB Entry Valid bit should be dropped and the kernel software should be changed to ensure that all TLB entries are always valid. Is this a good idea? Explain the advantages and disadvantages of such a design.

### **Problem M2.7.C**

---

Alyssa got wind of Ben's idea and suggests a different scheme to eliminate one of the valid bits. She thinks the page table entry valid and TLB Entry Valid bits can be combined into a single bit.

On a refill this combined valid bit will take the value that the page table entry valid bit had. A TLB entry is invalidated by writing it back to the page table and setting the combined valid bit in the TLB entry to invalid.

How does the kernel software need to change to make such a scheme work? How do the exceptions that the TLB produces change?

### **Problem M2.7.D**

---

Now, Bud Jet jumps into the game. He wants to keep the TLB Entry Valid bit. However, there is no way he is going to have two valid bits in each TLB entry (one for the TLB entry one for the page table entry). Thus, he decides to drop the page table entry valid bit from the TLB entry.



How does the kernel software need to change to make this work well? How do the exceptions that the TLB produces change?

---

**Problem M2.7.E**

---

Compare your answers to Problem M2.7.C and M2.7.D. What scheme will lead to better performance?

---

**Problem M2.7.F**

---

How about the R bit? Can we remove them from the TLB entry without significantly impacting performance? Explain briefly.

---

**Problem M2.7.G**

---

The processor has a kernel (supervisor) mode bit. Whenever kernel software executes the bit is set. When user code executes the bit is not set. Parts of the user's virtual address space are only accessible to the kernel. The supervisor bit in the page table is used to protect this region—an exception is raised if the user tries to access a page that has the supervisor bit set.

Bud Jet is on a roll and he decides to eliminate the supervisor bit from each TLB entry. Explain how the kernel software needs to change so that we still have the protection mechanism and the kernel can still access these pages through the virtual memory system.

---

**Problem M2.7.H**

---

Alyssa P. Hacker thinks Ben and Bud are being a little picky about these bits, but has devised a scheme where the TLB entry does not need the M bit or the U bit. It works as follows. If a TLB miss occurs due to a load, then the page table entry is read from memory and placed in the TLB. However, in this case the W bit will always be set to 0. Provide the details of how the rest of the scheme works (what happens during a store, when do the entries need to be written back to memory, when are the U and M bits modified in the page table, etc.).

## Problem M2.8: Page Size and TLBs

*This problem requires the knowledge of Handout #8 (Virtual Memory Implementation) and Lecture 9. Please, read these materials before answering the following questions.*

Assume that we use a hierarchical page table described in Handout #8.

The processor has a data TLB with 64 entries, and each entry can map either a 4KB page or a 4MB page. After a TLB miss, a hardware engine walks the page table to reload the TLB. The TLB uses a first-in/first-out (FIFO) replacement policy.

We will evaluate the memory usage and execution of the following program which adds the elements from two 1MB arrays and stores the results in a third 1MB array (note that, 1MB = 1,048,576 Bytes):

```
byte A[1048576]; // 1MB array
byte B[1048576]; // 1MB array
byte C[1048576]; // 1MB array

for(int i=0; i<1048576; i++)
    C[i] = A[i] + B[i];
```

We assume the A, B, and C arrays are allocated in a contiguous 3MB region of physical memory. **We will consider two possible virtual memory mappings:**

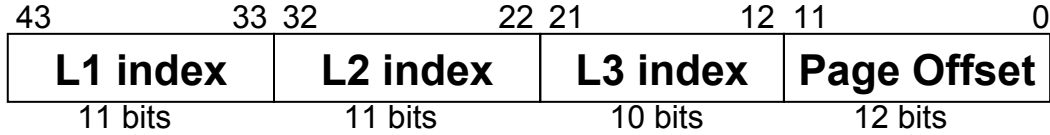
- **4KB:** the arrays are mapped using 768 4KB pages (each array uses 256 pages).
- **4MB:** the arrays are mapped using a single 4MB page.

For the following questions, assume that the above program is the only process in the system, and ignore any instruction memory or operating system overheads. Assume that the arrays are aligned in memory to minimize the number of page table entries needed.

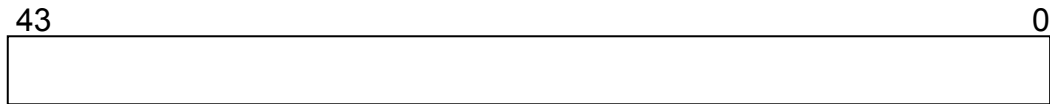
**Problem M2.8.A**

---

This is the breakdown of a virtual address which maps to a 4KB page:



Show the corresponding breakdown of a virtual address which maps to a 4MB page. Include the field names and bit ranges in your answer.



**Problem M2.8.B**

**Page Table Overhead**

---

We define page table overhead (PTO) as:

<b>PTO =</b>	Physical memory that is allocated to page tables
	Physical memory that is allocated to data pages

For the given program, what is the PTO for each of the two mappings?

<b>PTO<sub>4KB</sub> =</b>	
----------------------------	--

<b>PTO<sub>4MB</sub> =</b>	
----------------------------	--

**Problem M2.8.C**

**Page Fragmentation Overhead**

We define page fragmentation overhead (PFO) as:

<b>PFO =</b>	Physical memory that is allocated to data pages but is never accessed
	Physical memory that is allocated to data pages and is accessed

For the given program, what is the PFO for each of the two mappings?

<b>PFO<sub>4KB</sub> =</b>	
----------------------------	--

<b>PFO<sub>4MB</sub> =</b>	
----------------------------	--

**Problem M2.8.D**

Consider the execution of the given program, assuming that the data TLB is initially empty. For each of the two mappings, how many TLB misses occur, and how many page table memory references are required per miss to reload the TLB?

	Data TLB misses	Page table memory references (per miss)
<b>4KB:</b>		
<b>4MB:</b>		

**Problem M2.8.E**

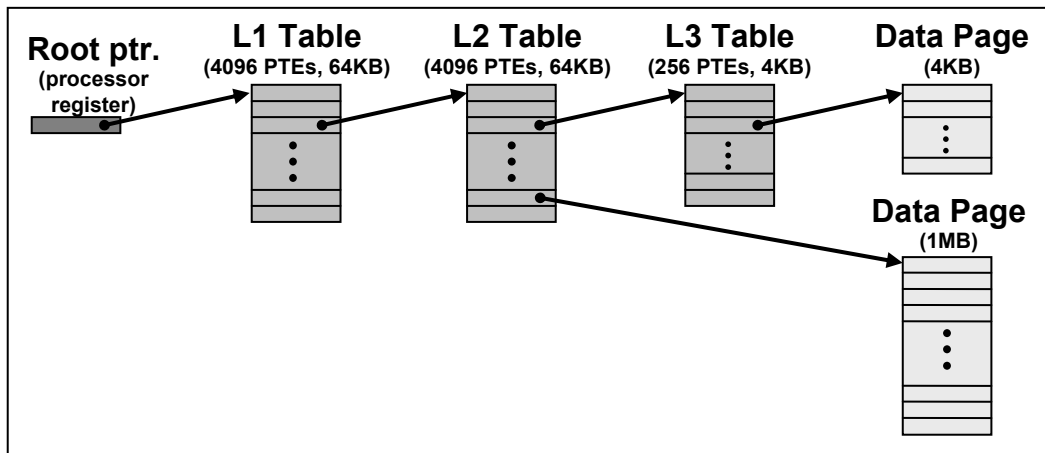
Which of the following is the best estimate for how much longer the program takes to execute with the 4KB page mapping compared to the 4MB page mapping? Circle one choice and **briefly explain** your answer (about one sentence).

<b>1.01×</b>	<b>10×</b>	<b>1,000×</b>	<b>1,000,000×</b>
--------------	------------	---------------	-------------------

## Problem M2.9: Page Size and TLBs

*This problem requires the knowledge of Handout #8 (Virtual Memory Implementation) and Lecture 9. Please, read these materials before answering the following questions.*

The configuration of the hierarchical page table in this problem is similar to the one in Handout #8, but we modify two parameters: 1) this problem evaluates a virtual memory system with two page sizes, 4KB and 1MB (instead of 4 MB), and 2) all PTEs are 16 Bytes (instead of 8 Bytes). The following figure summarizes the page table structure and indicates the sizes of the page tables and data pages (not drawn to scale):



The processor has a data TLB with 64 entries, and each entry can map either a 4KB page or a 1MB page. After a TLB miss, a hardware engine walks the page table to reload the TLB. The TLB uses a first-in/first-out (FIFO) replacement policy.

We will evaluate the execution of the following program which adds the elements from two 1MB arrays and stores the results in a third 1MB array (note that, 1MB = 1,048,576 Bytes, the starting address of the arrays are given below):

```
byte A[1048576]; // 1MB array 0x00001000000
byte B[1048576]; // 1MB array 0x00001100000
byte C[1048576]; // 1MB array 0x00001200000

for(int i=0; i<1048576; i++)
    C[i] = A[i] + B[i];
```

Assume that the above program is the only process in the system, and ignore any instruction memory or operating system overheads. The data TLB is initially empty.

### **Problem M2.9.A**

---

Consider the execution of the program. There is no cache and each memory lookup has 100 cycle latency.

If all data pages are 4KB, compute the ratio of cycles for address translation to cycles for data access.

If all data pages are 1MB, compute the ratio of cycles for address translation to cycles for data access.

### **Problem M2.9.B**

---

For this question, assume that in addition, we have a PTE cache with one cycle latency. A PTE cache contains page table entries. If this PTE cache has unlimited capacity, compute the ratio of cycles for address translation to cycles for data access for the 4KB data page case.

### **Problem M2.9.C**

---

With the use of a PTE cache, is there any benefit to caching L3 PTE entries? Explain.

### **Problem M2.9.D**

---

What is the minimum capacity (number of entries) needed in the PTE cache to get the same performance as an unlimited PTE cache? (Assume that the PTE cache does not cache L3 PTE entries and all data pages are 4KB)

### **Problem M2.9.E**

---

Instead of a PTE cache, we allow the data cache to cache data as well as PTEs. The data cache is a 4KB direct-mapped with 16 byte lines. Compute the ratio of cycles for address translation to cycles for data access for the 4KB data page case.

## Problem M2.10: 64-bit Virtual Memory

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

### Problem M2.10.A

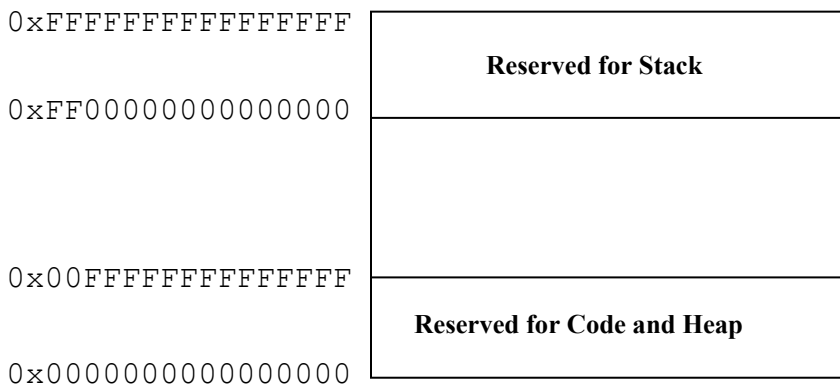
### Single level page tables

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.

### Problem M2.10.B

### Let's be practical

Many current implementations of 64-bit ISAs implement only part of the large virtual address space. One way to do this is to segment the virtual address space into three parts as shown below: one used for stack, one used for code and heap data, and the third one unused.



A special circuit is used to detect whether the top eight bits of an address are all zeros or all ones before the address is sent to the virtual memory system. If they are not all equal, an invalid virtual memory address trap is raised. This scheme in effect removes the top seven bits from the virtual memory address, but retains a memory layout that will be compatible with future designs that implement a larger virtual address space.

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.

**Problem M2.10.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]
1 <sup>st</sup> level index	2 <sup>nd</sup> level index	3 <sup>rd</sup> level index	Page offset

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

PHYSICAL MEMORY USED BY PAGE TABLES FOR A USER PROCESS

---

PHYSICAL MEMORY USED BY THE USER CODE, HEAP, AND STACK

Remember that a complete page table page (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?

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?

**Problem M2.10.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?

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

### Problem M2.10.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”)	
2 <sup>nd</sup> -level	
3 <sup>rd</sup> -level	

### Problem M2.10.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? The R10000 actually doubles this reach with little overhead by having each TLB entry map *two* physical pages, but don't worry about that here.

### Problem M2.10.G

### Virtual Memory and Caches

Ben Bitdiddle is designing a 4-way set associative cache that is virtually indexed and virtually tagged. He realizes that such a cache suffers from a *homonym* aliasing problem. The *homonym* problem happens when two processes use the same virtual address to access different physical locations. Ben asks Alyssa P. Hacker for help with solving this problem. She suggests that Ben should add a PID (Process ID) to the virtual tag. Does this solve the *homonym* problem?

Another problem with virtually indexed and virtually tagged caches is called *synonym* problem. *Synonym* problem happens when distinct virtual addresses refer to the same physical location. Does Alyssa's idea solve this problem?

Ben thinks that a different way of solving *synonym* and *homonym* problems is to have a direct mapped cache, rather than a set associative cache. Is he right?