VISHNU VARMA VENKATA

ECE4100/ECE6100/CS4290/CS6290
Advanced Computer Architecture
Homework 2

Part A: Caches

Direct-mapped Cache

The following diagram shows how a direct-mapped cache is organized. To read a word from the cache, the input address is set by the processor. Then the index portion of the address is decoded to access the proper row in the tag memory array and in the data memory array. The selected tag is compared to the tag portion of the input address to determine if the access is a hit or not. At the same time, the corresponding cache block is read and the proper line is selected through a MUX.

$2^b = 2^5 \Rightarrow b = 5$

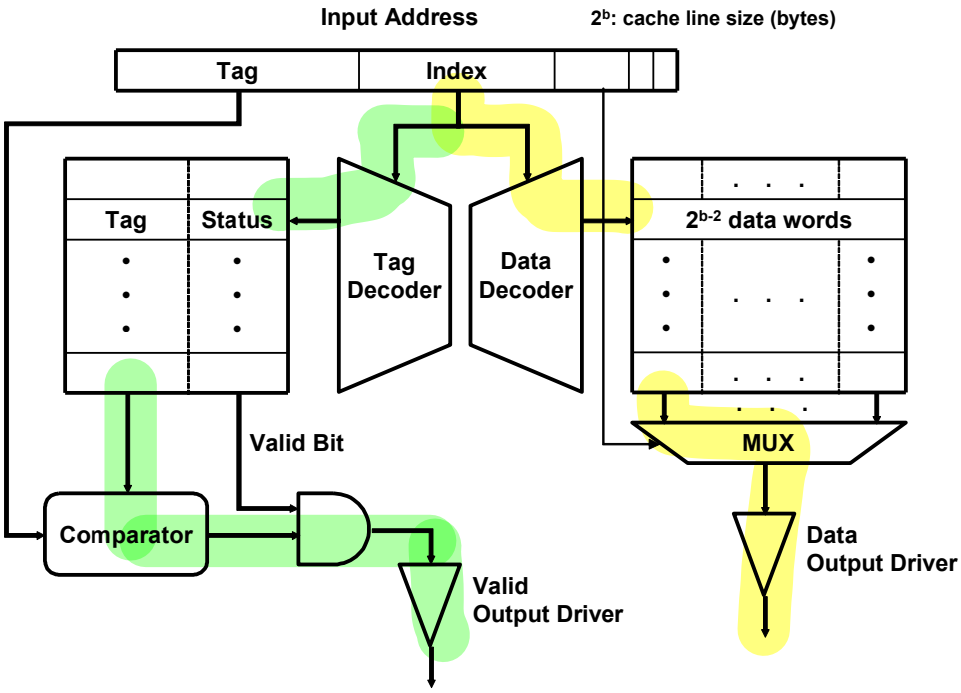


Figure A.1: A direct-mapped cache implementation

In the tag and data array, each row corresponds to a line in the cache. For example, a row in the tag memory array contains **one tag** and **two status bits** (valid and dirty) for the cache line. For direct-mapped caches, a row in the data array holds one cache line.

Four-way Set-associative Cache

The implementation of a 4-way set-associative cache is shown in the following diagram. (An *n*-way set-associative cache can be implemented in a similar manner.) The index part of the input address is used to find the proper row in the data memory array and the tag memory array. In this case, however, each row (set) corresponds to four cache lines (four ways). A row in the data memory holds four cache lines (for 32-bytes cache lines, 128 bytes), and a row in the tag memory array contains four tags and status bits for those tags (2 bits per cache line). The tag memory and the data memory are accessed in parallel, but the output data driver is enabled only if there is a cache hit.

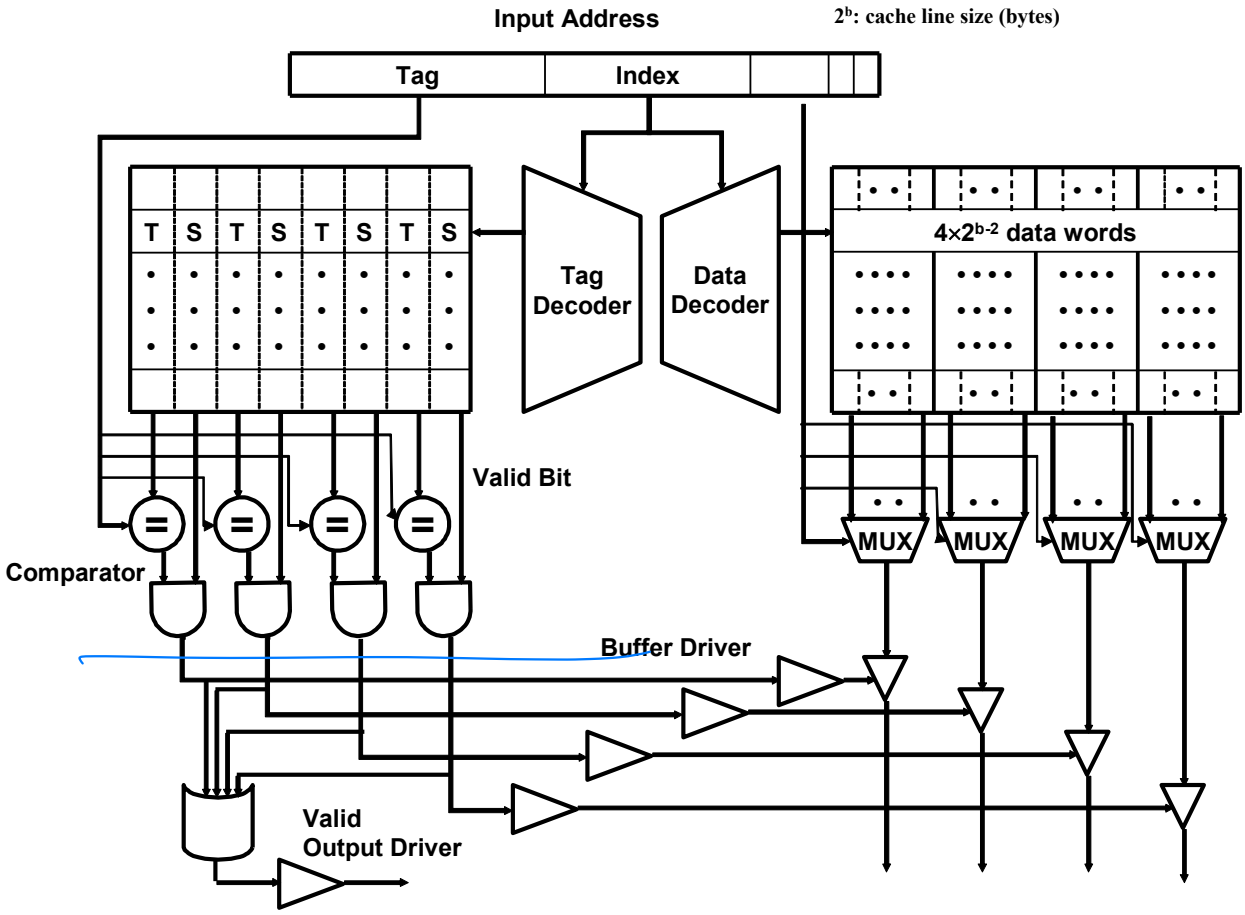


Figure A.2: A 4-way set-associative cache implementation

Problem A.1:

# of cache lines = $\frac{128K}{32} = \frac{2^{17}}{2^5} = 2^{12}$ cache lines ⇒ 12 bits to index

tag bits = 32 − 12 − 5 = 15 bits

2^3 data words ⇒ b = 5

$2^3 \times 2^5 = 2^8$ bits / cache line

We want to compute the access time of the direct-mapped (DM) cache. Assume a 128-KB cache with 8-word (32-byte) cache lines. 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 A.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×(# of index bits) + 1000	Tag	3400	3000
		Data	3400	3000
Memory array	200×log2 (# of rows) + 200×log2 (# 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×log2 N + 1000		2500	2500
Buffer driver	2000			2000
Data output driver	500×(associativity) + 1000		1500	3000
Valid output driver	1000		1000	1000

N = 8

Table A.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?

Data side = 3400 + 5000 + 2500 + 1500 = 12400 ps

Tag side = 3400 + 4217 + 4000 + 500 + 1000 = 13117 ps

⇒ CPU cycles = $13117 \times 10^{-12} \times 150 \times 10^6$

= 1.96755 cycles = 2 cycles!

4-way:- Tag side = 3000 + 4250 + 4400 + 500 = 12150

+ 2000 + 3000 | + 1000 + 1000

⇒ critical = 12150 + 5000 = 17150

⇒ $17150 \times 150 \times 10^{-6}$

= 2.5725 cycles = 3 cycles!

Problem A.2:

We also want to investigate the access time of a set-associative (SA) cache using the 4-way set-associative cache. 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 A.1. Compute the delay of each component, and fill in the column for a 4-way set-associative cache in Table A.1.

$\frac{128KB}{128B} = 2^{10}$ cache lines

$\Rightarrow$ indexing = 10 bits

$b = 5$ tag = 32 − 10 − 5 = 17 bits

# of bits in a row $= 128 \times 2^3 = 2^{10}$ bits

each way = 32KB 4 ways — 128 KB

each cache line = 32 B

$\frac{32KB}{32B} = 1K$ sets : 10 bits to index

8 X 16 B = 2^7 Bytes cache size

Problem A.3:

Now George P. Burdell 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, George 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, George tries out two replacement policies – least recently used (LRU) and round robin (FIFO).

George 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.*

index: 3 bits
b = 4 bits
9 = 1001
A = 1010

001001010000

110

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

set bit = 1
b = 4 bit

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		1A							no
102			10						no
361				36					no
204									yes
114									yes
1A4									yes
177							17		no
301			30						no
206									yes
135									yes

	4-way LRU
Total Misses	8
Total Accesses	13

4-way									FIFO
	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						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 A.4:

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 Problems A.1 and A.2. What would be the average memory access latency in CPU cycles for each cache (assume that a cache miss takes 20 cycles)?

cache hit takes = 3 cycles

DM AMAT = $2 + (10/13)(20) = 17.3846$ cycles @ 150 MHz
$= 18$ cycles

4-way LRU AMAT = $3 + (8/13)(20) = 15.3076$ cycles @ 150 MHz
$= 16$ cycles

Which one is better?

4-way LRU set associative cache 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 Problem A.3? Explain why. Is that replacement policy always going to yield better miss rates? If not, give a counter example using an address stream.

LRU > FIFO for the given address stream!

counter example!

In the above example instead of 20H-11H-1AH in the address stream if we had 10H-361-1AH then 20H would be evicted in LRU but if the initial entry would be 102-1A3-202 then 20 would stay in FIFO.
for this stream FIFO would be better than LRU so it is not fixed which is better

110-136-102-1A3-202-361-10H-361-1AH-177-301-206-135

↳ this is the counter example.

Part B: Multi-Level Caches

Suppose we have a physically-indexed, physically-tagged L1 cache.
All physical **addresses are 12 bits**. The L1 cache is **direct-mapped**, and has **4 sets**.
The **block size is 16 bytes**.

b = 4

i = 2

tag = 12 - 2 - 4 = 6 bits

Problem B.1:

L1 Cache Size = 64 bytes
Index Size = 2 bits
Tag Size = 6 bits

16 B x 4 = 64 B

Problem B.2:

Table B-1 tracks the contents of the L1 cache for a sequence of memory accesses.
The first entry shows the initial state of the cache.
"X" represents an invalid/empty entry.
To simplify things for you, **valid entries in the cache are represented by the {Tag, Index}.**

Complete the Table for the given sequence of addresses. The first two have been filled for you.
You only need to fill the elements in the Table when a value changes, except for the "L1 Hit?" column which needs to be filled out for all accesses.

Access Address	**L1 Hit?** (Yes/No)	**L1 State after Access** {Tag, Index}			
		Set0	**Set1**	**Set2**	**Set3**
...	…	X	45	F6	X
0x7B0	No				7B
0xDC4	No	DC			
0xDCF	Yes				
0xF3C	No				F3
0x8CB	No	8C			
0xB8B	No	B8			

1011
1100
1100
0011
1100
1000

Table B-1: Contents of L1 Cac

Problem B.3:

In the current design, all misses from L1 need to go to Memory which takes hundreds of cycles. To reduce the penalty of a L1 miss, we add an L2 cache after the L1 cache.

Now every L1 miss first performs a L2 lookup. If the line is found in L2, it is sent to L1. If not, it is brought in from Memory.

We follow an Exclusive L2 policy: All lines in L1 CANNOT be present in L2 (i.e., L1 fills do not result in L2 fills).

Assume that the L2 cache is **4-way set-associative, with 2 sets.**
The block size is the same (16 bytes).

Fill out the L1 and L2 cache contents for the given access pattern.
The initial state of the L1 and L2 in each case is given to you.
"X" represents an invalid/empty entry.
Valid entries are represented by the {Tag, Index}.

Each L2 entry also tracks its insertion time/last access time in order to implement LRU replacement within each set.
The LRU way is the candidate for replacement only if all ways have valid data, otherwise any of the ways with invalid data is replaced.

You only need to fill the elements in the Table when a value changes, except for the "L1 Hit?" and "L2 Hit" columns which needs to be filled out for all accesses.

Exclusive L2

All lines in L1 should NOT be present in L2

Time Unit	Access Addr	L1 Hit? (Yes/ No)	L1 State after access {Tag, Index}				L2 Hit? (Yes/ No/ NA)	Exclusive L2 State after access {Tag, Index} (Access Time)							
			Set0	Set1	Set2	Set3		Set0				Set1			
								Way0	Way1	Way2	Way3	Way0	Way1	Way2	Way3
…			X	45	F6	X		90 (7)	8C (3)	X	X	47(4)	4B(6)	E0(7)	X
10	0x7B0	No				7B	No								
11	0xDC4	No	DC				No								
12	0xDCF	Yes													
13	0xF3C	No				F3	No							7B(13)	
14	0x8CB	No	8C				Yes		X	DC(14)					
15	0xB8B	No	B8				No		8C(15)						

Table B.2 : Contents of L1 and Excusive L2 Caches

Part C: Memory Hierarchy

[All addresses in this Part at Physical Addresses].
Suppose we are running the following code:

```
#define ARRAY_SIZE 4
for (int i = 0; i < ARRAY_SIZE; i++) {
        S[i] = P[i] + Q[i]
}
```

The arrays S, P and Q have 4 entries each, and hold integer values (4 bytes at each entry). The memory addresses for each are shown below:

0x1A20	40
0x1A24	42
0x1A28	44
0x1A2C	48

Array P

0x1C60	5
0x1C64	1
0x1C68	7
0x1C6C	8

Array Q

0x2BA0	
0x2BA4	
0x28A8	
0x2BAC	

Array S

Suppose this code translates to the following assembly instruction sequence.

```
// Initial values:
// Rp = 0x1A20, Rq = 0x1C60, Rs = 0x2BA0
// Ri = 4

loop:     LD   R1, (Rp)        // R1 ← Mem[Rp]
          LD   R2, (Rq)        // R2 ← Mem[Rq]
          ADD  R3, R1, R2      // R3 = R1 + R2
          ST   R3, (Rs)        // Mem[Rs] ← R3
          ADDI Rp, 4           // Rp ← Rp + 0x4
          ADDI Rq, 4           // Rq ← Rq + 0x4
          ADDI Rs, 4           // Rs ← Rs + 0x4
          SUBI Ri, 1           // R4 ← R4 – 1
          BNEZ Ri, loop        // if R4 != 0, branch to loop
```

This code produces the following 12 memory accesses to the cache hierarchy:

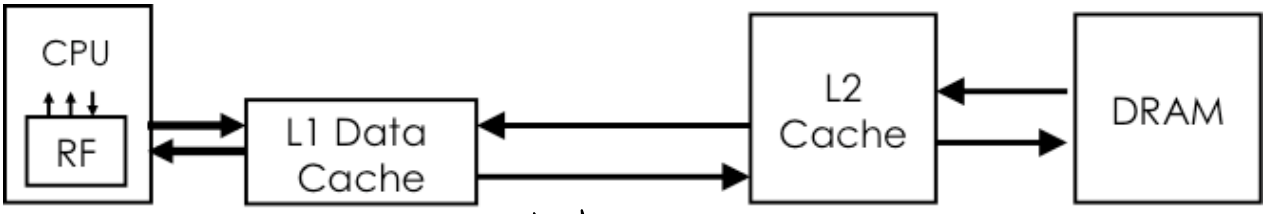
```
LD  0x1A20
LD  0x1C60
ST  0x2BA0
LD  0x1A24
LD  0x1C64
ST  0x2BA4
```

```
LD  0x1A28
LD  0x1C68
ST  0x2BA8
LD  0x1A2C
LD  0x1C6C
ST  0x2BAC
```

We will construct multiple memory hierarchies and analyze the miss rates for each.

Question C-1

Consider the following cache hierarchy with a L1 connected to a L2, which is connected to DRAM.



All cache lines are 16B.
The L1 is Direct Mapped, with 4 sets.
The L2 is 2-way Set Associative, with 4 sets.
The L2 is Non-Inclusive with L1.
The Non-Inclusion policy is as follows:

- All L1 allocations also result in L2 allocations
- All L1 evictions result in L2 allocations
- L2 evictions do not result in L1 evictions

L1 is write-through with respect to L2 → i.e., any **writes to addresses in L1 propagate data to L2 at the same time. Thus only the data in L2 is considered dirty.**
L2 is a write-back cache with respect to DRAM → **dirty data from L2 has to be written back to DRAM when the line gets evicted.**

Question C-1.1

On the next page, the initial state of the L1 and L2 are given. **Dirty data is circled in L2.**
Update the state of the L1 and L2 for each of the accesses. For each entry you can write the {tag, index} for simplicity and circle the dirty data. The writeback column specifies the cache line whose data is being written back to DRAM.

0010 index

00
01
10
11

110

1010

Access	L1 State After Access {tag, index}					Non-Inclusive L2 State After Access {tag, index}									Write-back
	Hit? (Yes / No)	Set 0	Set 1	Set 2	Set 3	Hit? (Yes/ No/ NA)	Set 0		Set 1		Set 2		Set 3		
							Way 0	Way 1	Way 0	Way 1	Way 0	Way 1	Way 0	Way 1	
		0x110	0x1A5	0x11E	0x1BB		0x110				0x11E			0x1BB	
LD 0x1A20	No			0x1A2		No						0x1A2			
LD 0x1C60	No			0x1C6		No					0x1C6				0x11E
ST 0x2BA0	No			0x2BA		No						0x2BA			
LD 0x1A24	No			0x1A2		No					0x1A2				
LD 0x1C64	No			0x1C6		No						0x1C6			0x2BA
ST 0x2BA4	No			0x2BA		No					0x2BA				
LD 0x1A28	No			0x1A2		No						0x1A2			
LD 0x1C68	No			0x1C6		No					0x1C6				0x2BA
ST 0x2BA8	No			0x2BA		No						0x2BA			
LD 0x1A2C	No			0x1A2		No					0x1A2				
LD 0x1C6C	No			0x1C6		No						0x1C6			0x2BA
ST 0x2BA0C	No			0x2BA		No					0x2BA				

Question C-1.2

What is the L1 Miss Rate (L1 Misses/L1 Accesses) ?

100%.

Question C-1.3

What is the L2 Miss Rate (L2 Misses/L2 Accesses) ?

100%.

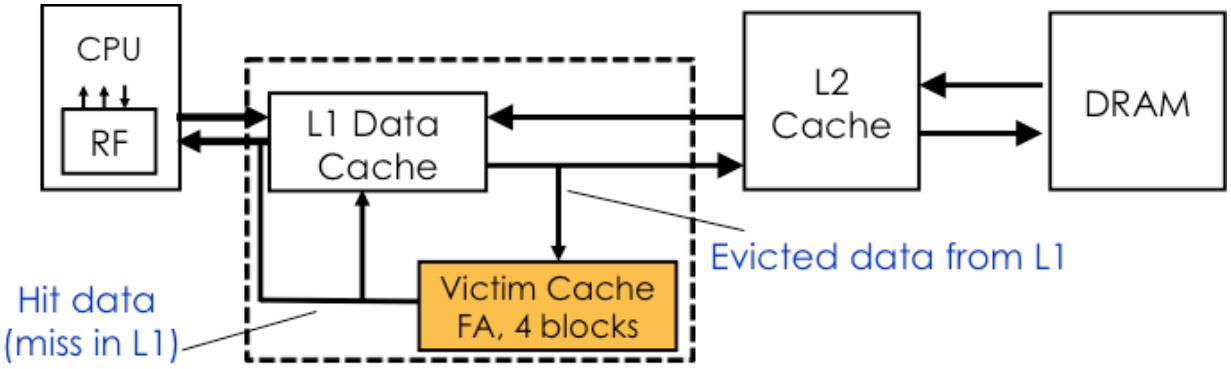
Question C-1.4

How many times does a write back from the L2 to the DRAM take place?

4 times

Question C-2

We add a Victim Cache next to the L1 cache:



All evicted data from L1 goes to L2 as before, but a copy is also retained in the Victim Cache. The victim cache has 4 entries, and is fully associative.
Upon a L1 miss, first the Victim Cache is checked, before going to L2.
IF THE DATA IS FOUND IN EITHER THE DIRECT MAPPED CACHE OR THE VICTIM CACHE, IT IS CONSIDERED A L1 HIT.
The line is brought into L1 and the evicted line from L1 is added to the Victim Cache. Victim Caches are Exclusive with respect to L1 – i.e., either L1 or the Victim Cache will have a cache line, never both.

Question C-2.1

Suppose L1 has an overall hit rate of 90%. Of these hits, 70% hit in the direct mapped cache and take 1-cycle, while 30% hit in the victim cache and take 4 cycles. A L1 miss takes 50 cycles to bring the data into L1. What is the average memory access time?

$$AMAT = 0.9 \times 0.7\,(1) + 0.9 \times 0.3\,(1+4) + 0.1 \times 50$$

$$= 6.98 \text{ cycles}$$

Question C-2.2

Update the state of the L1 and Victim Caches **for the same set of memory accesses.**

Access	L1 State After Access {tag, index}					Victim Cache State After Access {tag, index}				
	Hit? (Yes/ No)	**Set 0**	**Set 1**	**Set 2**	**Set 3**	**Hit? (Yes/ No/ NA)**	**Way 0**	**Way 1**	**Way 2**	**Way 3**
		0x110	0x1A5	0x11E	0x1BB					
LD 0x1A20	No			0x1A2		No	0x11E			
LD 0x1C60	No			0x1C6		No		0x1A2		
ST 0x2BA0	No			0x2BA		No			0x1C6	
LD 0x1A24	No			0x1A2		Yes		X		0x2BA
LD 0x1C64	No			0x1C6		Yes		0x1A2	X	
ST 0x2BA4	No			0x2BA		Yes			0x1C6	X
LD 0x1A28	No			0x1A2		Yes		X		0x2BA
LD 0x1C68	No			0x1C6		Yes		0x1A2	X	
ST 0x2BA8	No			0x2BA		Yes			0x1C6	X
LD 0x1A2C	No			0x1A2		Yes		X		0x2BA
LD 0x1C6C	No			0x1C6		Yes		0x1A2	X	
ST 0x2BA0C	No			0x2BA		Yes			0x1C6	X

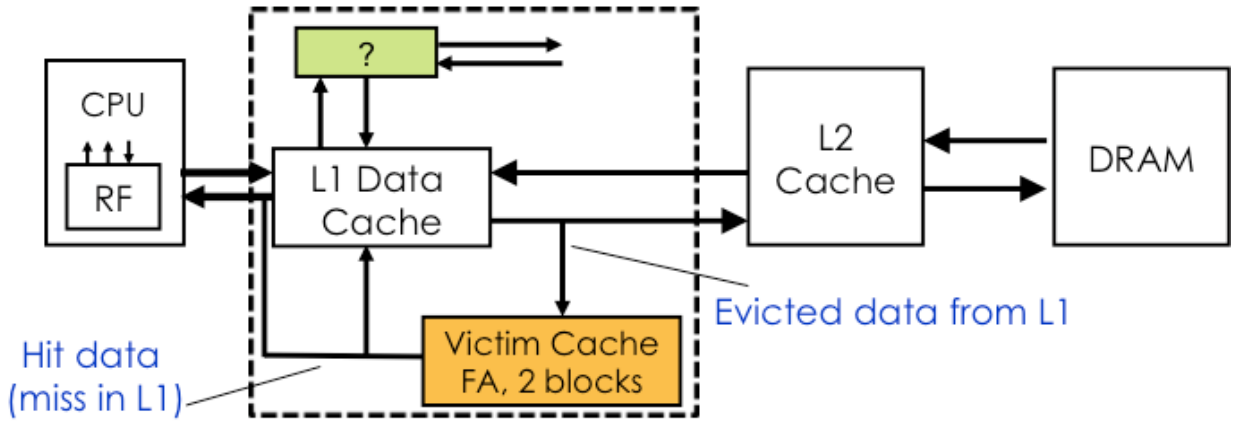
Question C-2.3

What is the L1 Miss Rate (L1 Misses/L1 Accesses) ?

3/12 = 25%.

Question C-2.4

George P. Burdell is interested in designing a memory hierarchy optimized for this code. He looks at the memory access pattern and the L1 Miss Rate and believes that he can reduce it even further without increasing the size of any of the caches. His solution is to reduce the size of the Victim Cache to two entries, and use the remaining two entries for some other structure connected to L1, as shown below.

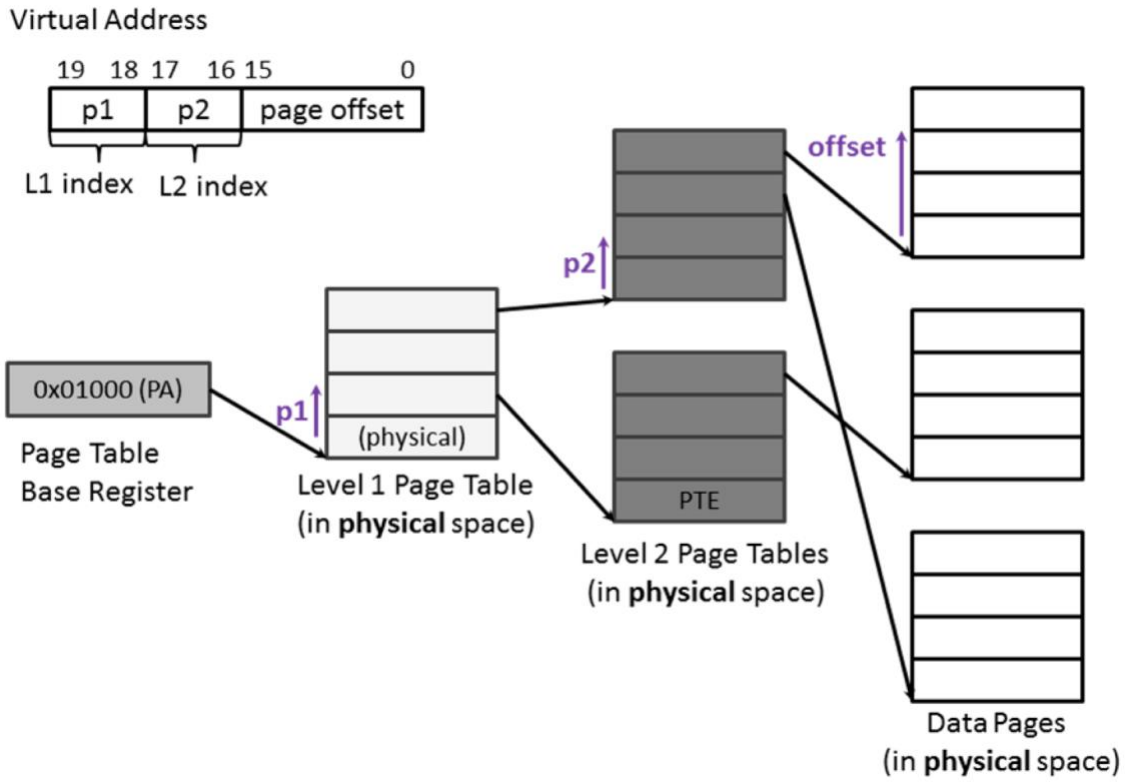


What structure do you think George has in mind? Briefly describe the function of this structure, and a recipe to allocate it and access it.

From the previous part only the initial compulsory misses were seen all others were hits; in order to reduce the compulsory misses we can add a prefetcher to access the initial required addresses. Even with this method there will be one miss but it is a better design.

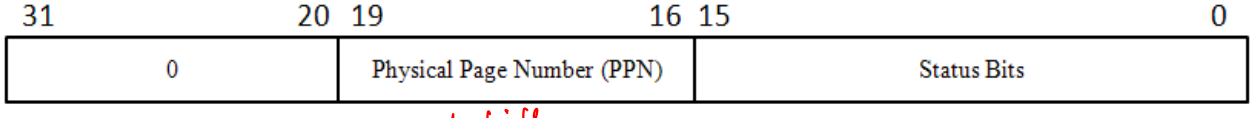
Part D: Hierarchical Page Table & TLB

Suppose there is a virtual memory system with 64KB page which has 2-level hierarchical page table. The **physical address** of the base of the level 1 page table (**0x01000**) is stored in a special register named Page Table Base Register. The system uses **20-bit** virtual address and **20-bit** physical address. The following figure summarizes the page table structure and shows the breakdown of a virtual address in this system. The size of both level 1 and level 2 page table entries is **4 bytes** and the memory is byte-addressed. Assume that all pages and all page tables are loaded in the main memory. Each entry of the level 1 page table contains the **physical address** of the base of each level 2 page tables, and each of the level 2 page table entries holds the **PTE** of the data page (the following diagram is not drawn to scale). As described in the following diagram, L1 index and L2 index are used as an index to locate the corresponding ***4-byte entry*** in Level 1 and Level 2 page tables.



2-level hierarchical page table

A PTE in level 2 page tables can be broken into the following fields (Don't worry about status bits for the entire part).



4 bits

Problem D.1:

Assuming the TLB is initially at the state given below and the initial memory state is to the right, what will be the final TLB states after accessing the virtual address given below? Please fill out the table with the final TLB states. You only need to write VPN and PPN fields of the TLB. The TLB has 4 slots and is fully associative and if there are empty lines they are taken first for new entries. Also, translate the virtual address (VA) to the physical address (PA). ***For your convenience, we separated the page number from the rest with the colon ":".***

VPN	PPN
0x8	0x3

Initial TLB states

Address (PA)	
0x0:104C	0x7:1A02
0x0:1048	0x3:0044
0x0:1044	0x2:0560
0x0:1040	0xA:0FFF
0x0:103C	0xC:D031
0x0:1038	0xA:6213
0x0:1034	0x9:1997
0x0:1030	0xD:AB04
0x0:102C	0xF:A000
→ 0x0:1028	0x6:0020
0x0:1024	0x5:1040
→ 0x0:1020	0x4:AA40
0x0:101C	0x3:10EF
0x0:1018	0xB:EA46
0x0:1014	0x2:061B
0x0:1010	0x1:0040
→ 0x0:100C	0x0:1020
0x0:1008	0x0:1048
0x0:1004	0x0:1010
0x0:1000	0x0:1038

The part of the memory (in physical space)

18 bits - anything = 2^{18}

Virtual Address:

0xE:17B0 (1110:0001011110110000)

P1 P2

11 << 2

3 positions

100C → 0x0:1020

11 10 01 00

0000

0x6:0020

0x6:17B0

VPN	PPN
0x8	0x3
0xE	0x6

Final TLB states

VA 0xE17B0 => PA 0x6:17B0

Problem D.2:

What is the total size of memory required to store both the level 1 and 2 page tables?

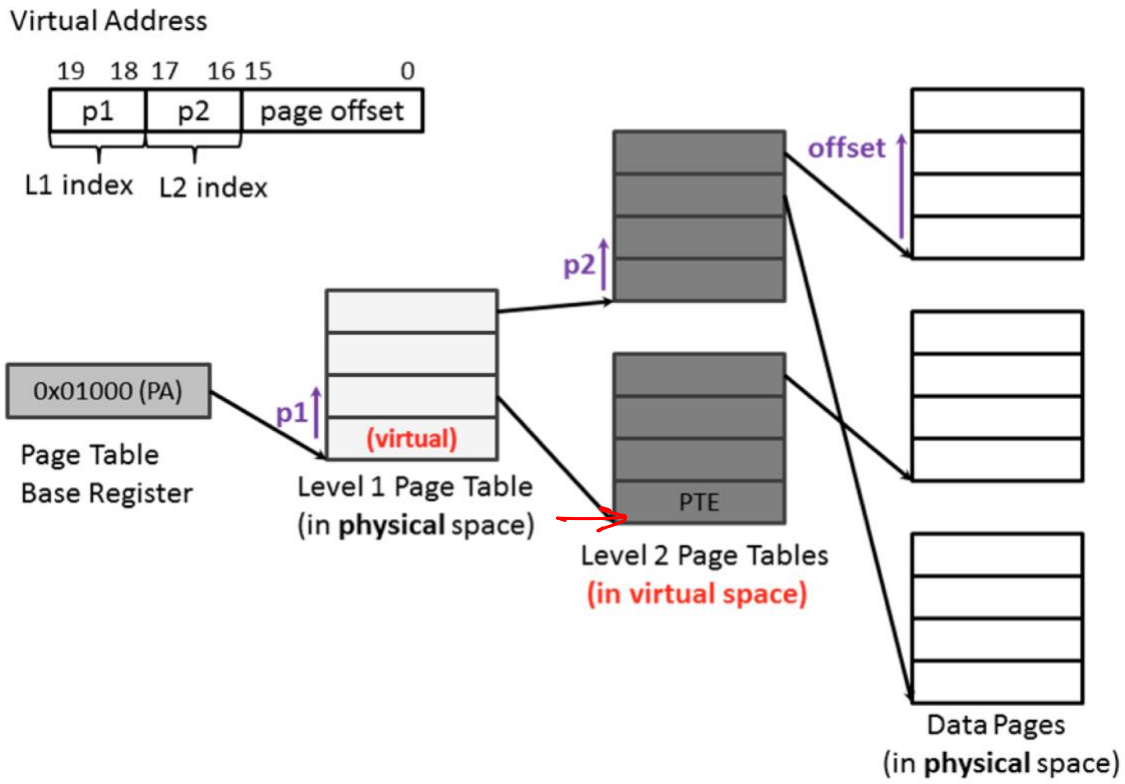
L1 : 4 lines each 4B
⇒ 16 B

L2: 4 x 4 x 4B = 64B

⇒ Total = 16 + 64 = 80B

Problem D.3:

George Burdell wanted to reduce the amount of physical memory required to store the page table, so he decided to only put the level 1 page table in the physical memory and use the virtual memory to store level 2 page tables. Now, each entry of the level 1 page table contains the **virtual address** of the base of each level 2 page tables, and each of the level 2 page table entries contains the **PTE** of the data page (the following diagram is not drawn to scale). Other system specifications remain the same. (The size of both level 1 and level 2 page table entries is **4 bytes.**)



George's design with 2-level hierarchical page table

Assuming the TLB is initially at the state given below and the initial memory state is to the right (**different** from Problem C.1), what will be the final TLB states after accessing the virtual address given below? Please fill out the table with the final TLB states. You only need to write VPN and PPN fields of the TLB. The TLB has 4 slots and it is fully associative and if there are empty lines they are taken first for new entries. Also, translate the virtual address to the physical address. ***Again, we separated the page number from the rest with the colon ":".***

.

VPN	PPN
0x8	0x1

Initial TLB states

Address (PA)	
.....	.....
0x1:1048	0x3:0044
0x1:1044	0x2:0560
0x1:1040	0x1:0FFF
0x1:103C	0x1:D031
0x1:1038	0xA:6213
0x1:1034	0x9:1997
.....	.....
0x1:0018	0xF:A000
0x1:0014	0x6:0020
0x1:0010	0x1:1040
0x1:000C	0x4:AA40
0x1:0008	0x3:10EF
0x1:0004	0xB:EA46
.....	.....
0x0:1010	0x1:0040
0x0:100C	0x0:1020
0x0:1008	0x2:0010
0x0:1004	0x8:0010
0x0:1000	0x8:1038

The part of the memory (in physical space)

0x2:0010
virtual address

Virtual Address:

0xA:0708 (1010:0000011100001000)
p1 p2

VPN	PPN
0x8	0x1
0x2	0x1
0xA	0xF

Final TLB states

0x 0010
p1 p2
0x8:1038 ->
0x1:1038 physical address

12 base
p2: 10
0x1:1038 -> 0x1:1040
0x1:0FFF

⇒ physical address
= 0x1:0010

VA 0xA0708 => PA 0xF:0708

now, 0x1:0010, p2 = 2
0x1:0018
0xF:A000
PPN

Problem D.4

Buzz examines George's design and points out that his scheme can result in infinite loops. Describe the scenario where the memory access falls into infinite loops.

when the VPN to PPN translation from the VA obtained at L1 to PA to be found at L2 is not a TLB hit then this access pattern will turn into an infinite loop iterating over and over.
if the VPN of both the initial address and L1 o/p VA is the same there won't be a TLB hit ⇒ so ∞ loop.

Problem D.5

Suppose we design a physically tagged, virtually indexed 2-way set associative cache that we want to access *in parallel* to the TLB. Suppose the cache line size is 64B. What is the largest possible cache we can design, and how many sets would it have?

64B ⇒ b = 6
p = 16 offset bits!

k + b ≤ p
⇒ k + 6 ≤ 16
k ≤ 10

k = 10 ← # of index bits

it can have 2^{10} sets
1K sets
⇒ largest possible cache
$= 2^{10} \times 2 \times 64\,B$
$= 2^{17}\,B$
= 128KB cache