Problem M2.8: Virtual Memory Bits

Problem M2.8.A

The answer depends on certain assumptions in the OS. Here we assume that the OS does everything that is reasonable to keep the TLB and page table coherent. Thus, any change that OS software makes is made to both the TLB and the page table.

However, the hardware can change the U bit (whenever a hit occurs this bit will be set) and the M bit (whenever a page is modified this bit will be set). Thus, these are the only bits that need to be written back. Note that the system will function correctly even if the U bit is not written back. In this case the performance would just decrease.

It is also important to note, that if the entry is laid out properly in memory, all the hardware-modified bits in the TLB can be written back to memory with a single memory write instruction. Thus it makes no difference whether one or two bits have been modified in the TLB, because writing back one bit or two bits still requires writing back a whole word.

Problem M2.8.B

An advantage of this scheme is that we do not need the TLB Entry Valid bit in the TLB anymore. One bit savings is not very much.

A disadvantage of this scheme is that the kernel needs to ensure that all TLB entries always are valid. During a context switch, all TLB entries would need to be restored (this is time-consuming). And, in general, whenever a TLB entry is invalidated, it will have to be replaced with another entry.

Problem M2.8.C

Changes to exceptions: "Page Table Entry Invalid" and "TLB Miss" exceptions are replaced with exceptions "TLB Entry Invalid" and "TLB No Match"

The TLB Entry Invalid exception will be raised if the VPN matches the TLB tag but the (combined) valid bit is false. When this exception is raised the kernel will need to consult the page table entry to see if this is a TLB miss (valid bit in page table entry is true), or an access of an invalid page table entry (valid bit in page table entry is false). Depending on what the cause of the exception was, it will then have to perform the necessary operations to recover.

The TLB No Match exception will be raised if the VPN does not match any of the TLB tags. If this exception is raised the kernel will do the same thing it did when a TLB Miss occurred in the previous design.

Problem M2.8.D

When loading a page table entry into the TLB, the kernel will first check to see if the page table entry is valid or not. If it is valid, then the entry can safely be loaded into the TLB. If the page table entry is not valid, then the Page Table Entry Invalid exception handler needs to be called to create a valid entry before loading it into the TLB. Thus we only keep valid page table entries in the TLB. If a page table entry is to be invalidated, the TLB entry needs to be invalidated.

Changes to exceptions: Page Table Entry Invalid exception is not raised by the TLB anymore.

Problem M2.8.E

The solution for Problem M2.8.C ends up taking two exceptions, if the PTE has the combined valid bit set to invalid. The first exception will be the TLB No Match exception, which will call a handler. The handler will load the corresponding PTE into the TLB and restart the instruction. The instruction will cause **another** exception right away, because the valid bit will be set to invalid. The exception will be the TLB Entry Invalid exception.

The solution for Problem M2.8.D will only take one exception, because the handler for Page Table Entry Invalid exception will get called by the TLB Miss handler. When the instruction that caused the exception is restarted, it will execute correctly, because the handler will have created a valid PTE and put it in the TLB.

Thus Bud Jet's solution in M2.8.D will be faster.

Problem M2.8.F

Yes, the R bit can be removed in the same way we removed the V bit in 8.D. When loading a page table entry into the TLB we check if the data page is resident or not. If it is resident, we can write the entry into the TLB. If it is not resident, we go to the nonresident page handler, loading the page into memory before loading the entry into the TLB. Thus, we only keep page table entries of resident pages in the TLB. In order to preserve this invariant, the kernel will have to invalidate the TLB entry corresponding to any page that gets swapped out. There's no performance penalty since the page was going to be loaded in from disk anyway to service the access that triggered the fault.

Problem M2.8.G

The OS needs to check the permissions before loading the entry into the TLB. If permissions were violated, then the Protection Fault handler is called. Thus, we only keep page table entries of pages that the process has permissions to access.

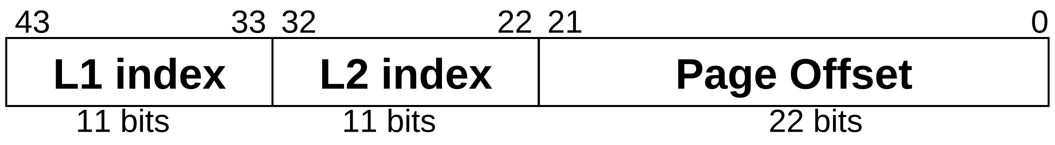
Problem M2.8.H

Whenever a page table entry is loaded into the TLB the U bit in the page table PTE can be set. Thus, we do not need the U bit in the TLB entry anymore.

Whenever a Write Fault happens (store and W bit is 0) the kernel will check the page table PTE to see if the W bit is set there. If it is not set the old Write Fault handler will be called. If the W bit is set, then the kernel will set the M bit in the PTE, set the W bit in the TLB entry to 1, and restart the store instruction. Thus, the M bit is not needed in the TLB either, and hence, TLB entries do not need to be written back to the page table anymore.

Problem M2.9: Page Size and TLBs

Problem M2.9.A



The L1 index and L2 index fields are the same, but the Page Offset field subsumes the L3 index and increases to 22 bits.

Problem M2.9.B — Page Table Overhead

$$\mathbf{PTO_{4KB}} = \frac{16\text{ KB} + 16\text{ KB} + 8\text{ KB}}{3\text{ MB}} = \frac{40\text{ KB}}{3\text{ MB}} = \mathbf{1.3\%}$$

$$\mathbf{PTO_{4MB}} = \frac{16\text{ KB} + 16\text{ KB}}{4\text{ MB}} = \frac{32\text{ KB}}{4\text{ MB}} = \mathbf{0.8\%}$$

For the 4KB page mapping, one L3 table is sufficient to map the 768 pages since each contains 1024 PTEs. Thus, the page table consists of one L1 table (16KB), one L2 table (16KB), and one L3 table (8KB), for a total of 40 KB. The 768 4KB data pages consume exactly 3MB. The total overhead is 1.3%..

The page table for the 4MB page mapping, requires only one L1 table (16KB) and one L2 table (16KB), for a total of 32 KB. A single 4MB data pages is used, and the total overhead is 0.8%.

Problem M2.9.C — Page Fragmentation Overhead

$$\mathbf{PFO_{4KB}} = \frac{0}{3\text{ MB}} = \mathbf{0\%}$$

$$\mathbf{PFO_{4MB}} = \frac{1\text{ MB}}{3\text{ MB}} = \mathbf{33\%}$$

With the 4KB page mapping, all 3MB of the allocated data is accessed. With the 4MB page mapping, only 3MB is accessed and 1MB is unused. The overhead is 33%.

Problem M2.9.D

	Data TLB misses	Page table memory references (per miss)
4KB:	**768**	**3**
4MB:	**1**	**2**

The program sequentially accesses all the bytes in each page. With the 4KB page mapping, a TLB miss occurs each time a new page of the input or output data is accessed for the first time. Since the TLB has more than 3 entries (it has 64), there are no misses during the subsequent accesses within each page. The total number of misses is 768. With the 4MB page mapping, all of the input and output data is mapped using a single page, so only one TLB miss occurs.

For either page size, a TLB miss requires loading an L1 page table entry and then loading an L2 page table entry. The 4KB page mapping additionally requires loading an L3 page table entry.

Problem M2.9.E

[✓ **1.01×**] **10×** **1,000×** **1,000,000×**

Although the 4KB page mapping incurs many more TLB misses, with either mapping the program executes 2M loads, 1M adds, and 1M stores (where M = 2^{20}). With the 4MB mapping, the single TLB miss is essentially zero overhead. With the 4KB mapping, there is one TLB miss for every 4K loads or stores. Each TLB miss requires 3 page table memory references, so the overhead is less than 1 page table memory reference for every 1000 data memory references. Since the TLB misses likely cause additional overhead by disrupting the processor pipeline, a 1% slowdown is a reasonable but probably conservative estimate.

Problem M2.10: Page Size and TLBs

Problem M2.10.A

If all data pages are 4KB

Address translation cycles = 100 + 100 +100 (for L1, L2 and L3 PTE)

*Data access cycles = 4K * 100*
(there is no cache, this assumes that memory access is byte-wise)

If all data pages are 1MB

Address translation cycles = 100 + 100 (for L1, L2 PTE)

*Data access cycles = 1M * 100*
(there is no cache, this assumes that memory access is byte-wise)

Problem M2.10.B

*Address translation cycles = (256*3 + 3 + 1) * 100*
*(Note that the arrays are contiguous and share some PTE entries. 256 L3 PTEs per array * 3 arrays, 1 L2 PTE per array * 3 arrays, 1 L1 PTE)*

*Data access cycles = 3M*100*

Problem M2.10.C

No. For the sample program given, all L3 PTEs are used only once.

Problem M2.10.D

4. (1 for L1 and 3 for L2)

Problem M2.10.E

This question was poorly worded. It was intended that the use of data cache will not be very helpful since the 4K pages keep conflicting with each other in the cache.

Problem M2.11: 64-bit Virtual Memory

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

Problem M2.11.A — Single level page tables

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. $2^{52} * 2^3 = 2^{55}$, or 32 petabytes!

Problem M2.11.B — Let's be practical

2^2 segments * $2^{(44-12)}$ virtual pages = 2^{34} PTEs. 2^3 (bytes/PTE) * 2^{34} PTEs = $\mathbf{2^{37}}$ bytes.

It is possible to interpret the question as there being 3 segments of 2^{44} bytes. Thus we'd need:

3 segments * $2^{(44-12)}$ virtual pages = $2^{33} + 2^{32}$ PTEs. $2^3*(2^{33}+ 2^{32}) = \mathbf{2^{36} + 2^{35}}$ bytes.

Problem M2.11.C — Page table overhead

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

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

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 M2.11.D — PTE Overhead

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

There are 31 wasted bits in a 64 bit page table entry. It turns out that some of the "wasted" space is recovered by the OS to do bookkeeping, but not much.

Problem M2.11.E — Page table implementation

The top level has 1024 = 2^{10} entries. Next level also has 1024 = 2^{10} entries. The 3rd level has 512 = 2^9 entries. So the table is as follows:

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

Problem M2.11.F — Variable Page Sizes

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

Problem M2.11.G — Virtual Memory and Caches

Alyssa's suggestion solves the homonym problem. If we add a PID as a part of the cache tag, we can ensure that two same virtual addresses from different processes can be distinghuished in the cache, because their PIDs will be different.

Putting a PID in the tag of a cache does not solve the synonym problem. This is because the synonym problem already deals with different virtual addresses, which presumably would have different tags in the cache. In fact, those two virtual addresses would usually belong to different processes, which would have different PIDs.

Ben is wrong in thinking that changing the cache to be direct mapped helps in any way. The homonym problem still happens, because same virtual addresses still receive the same tags. The synonym problem still happens because two different virtual addresses still receive different tags.

One way to solve both these problems is to make the cache physically tagged, as described in Lecture 10.

Problem M2.12: Cache Basics

Problem M2.12.A

	Virtually indexed	Physically indexed
Direct-mapped (A)	G	A, C, E, G
2-way Set-associative (B)	C, G	A, C, E, G

Problem M2.12.B

Index	V	Tags (way0)	V	Tags (way1)
0	**1**	0x45	**0**	
1	**1**	0x3D	**0**	
2	**1**	0x2D	**1**	0x25
3	**1**	0x1D	**0**	

Problem M2.12.C

0x34 (hit: index 2)
-> 0x38 (miss: index 3)
-> 0x50 (miss: index 2)
-> 0x54 (hit: index 2)
-> 0x208 (hit: index 1)
-> 0x20C (hit: index 1)
-> 0x74 (miss: index 2)
-> 0x54 (hit: index 2)

Because there are 5 hits and 3 misses,
Average memory access time = 1 + 3 / 8 * 16 = 7 cycles

Problem M2.13: Cache Basics

Problem M2.13.A

Virtual address 0x00030 -> Physical address (0x00D40)

VPN	PPN
0x0100	0x0F01
0x0003	0x00D4

TLB states

Problem M2.13.B

Virtual address 0x00050 -> Physical address (0x00E20)

VPN	PPN
0x0100	0x0F01
0x0101	0x0F02
0x0005	0x00E2

TLB states

Problem M2.13.C

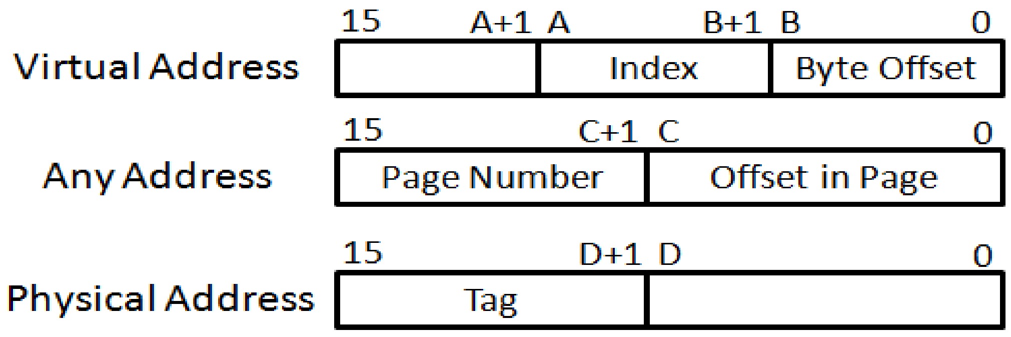
New CPI = 2 + (0.01+0.02)*20 = 2.6

Problem M2.14: Cache Basics (Fall 2010 Part A)

Problem M2.14.A

Suppose we have a **virtually-indexed, physically-tagged** 2-way set associative cache. Each way has 8 cache blocks (i.e., there are 8 sets) and the block size is 8 bytes. The page size is 256 bytes, and the byte-addressed machine uses 16-bit virtual address and 16-bit physical address. Do not worry about aliasing problems for this question.

The next diagram shows the corresponding breakdown of a virtual address and a physical address in this system (index, tag, and byte offset). Replace "A", "B", "C" and "D" with bit indexes showing the size of each field. Note that tags should contain the minimum number of bits required to provide the information needed to check whether the cache hits.



A: __5__ B: __2__ C: __7__ D: __5__

Problem M2.14.B

Now, we test the cache by accessing the following **virtual address**. We provide the corresponding binary number for the virtual address.

0x0151 (0000000101010001)

The table below shows the current TLB states. After each address is accessed, complete tables on the next page and show the progression of cache states (in the tables, inv = invalid, and the column shows tags). Assume that the **Least Recently Used (LRU)** replacement policy is used. "**LRU way**" bit in the cache represents the way that is least recently used. (Note that this bit should also be updated if necessary.) ***You may only fill in the elements in the table when a value changes from the previous table. Write tags in hexadecimal numbers***. ***If the memory access is a cache hit, mark with "V" where the hit occurred.*** Note that the cache uses **physical tags**.

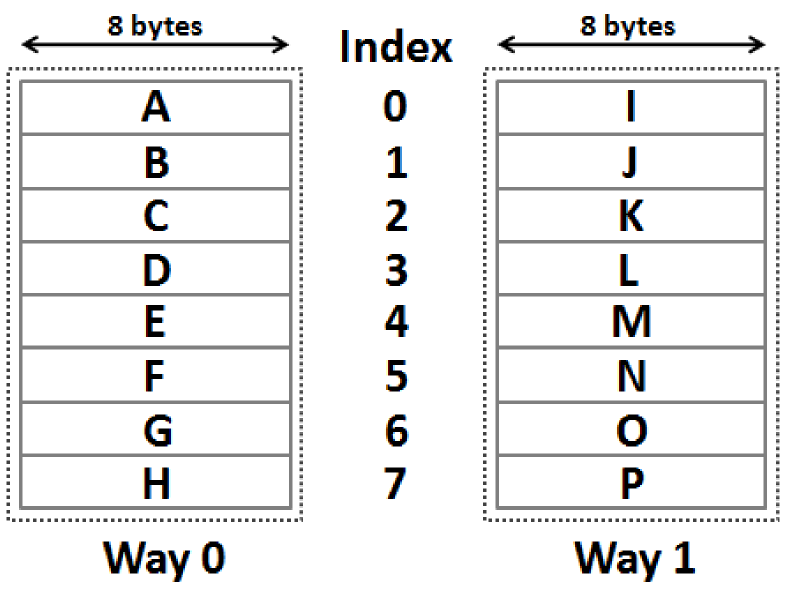
VPN	PPN
0x00	0x0A
0x01	0x1A
0x02	0x2A
0x03	0x3A

0. Initial State			
idx	LRU Way?	Tags (Way 0)	Tags (Way 1)
0	0	inv	0x40
1	0	inv	inv
2	0	0x8B	0x14
3	0	inv	inv
4	0	inv	inv
5	1	0x3F	0xAA
6	0	inv	inv
7	1	0xC3	0x1F

1. State after access 0x0151			
idx	LRU Way?	Tags (Way 0)	Tags (Way 1)
0			
1			
2	1	0x69	
3			
4			
5			
6			
7			

Problem M2.14.C

Suppose we have a 2-way set-associative cache. Each way has 8 cache blocks (i.e., there are 8 sets) and the block size is 8 bytes/block, so the total is 128 bytes. The following figure shows each cache block in this cache configuration.



Cache Configuration

Suppose this cache is **virtually indexed**. A program wants to read from a **virtual address** 0x6E, and the physical address of this virtual address 0x6E is unknown (could be arbitrary). Enumerate **all** blocks (A,B,C,D,…) that can possibly hold the content of the virtual address 0x6E, for each given page size in the next table.

Page Size	Block(s) which can be mapped to
16 bytes	F, N
32 bytes	F, N
64 bytes	F, N

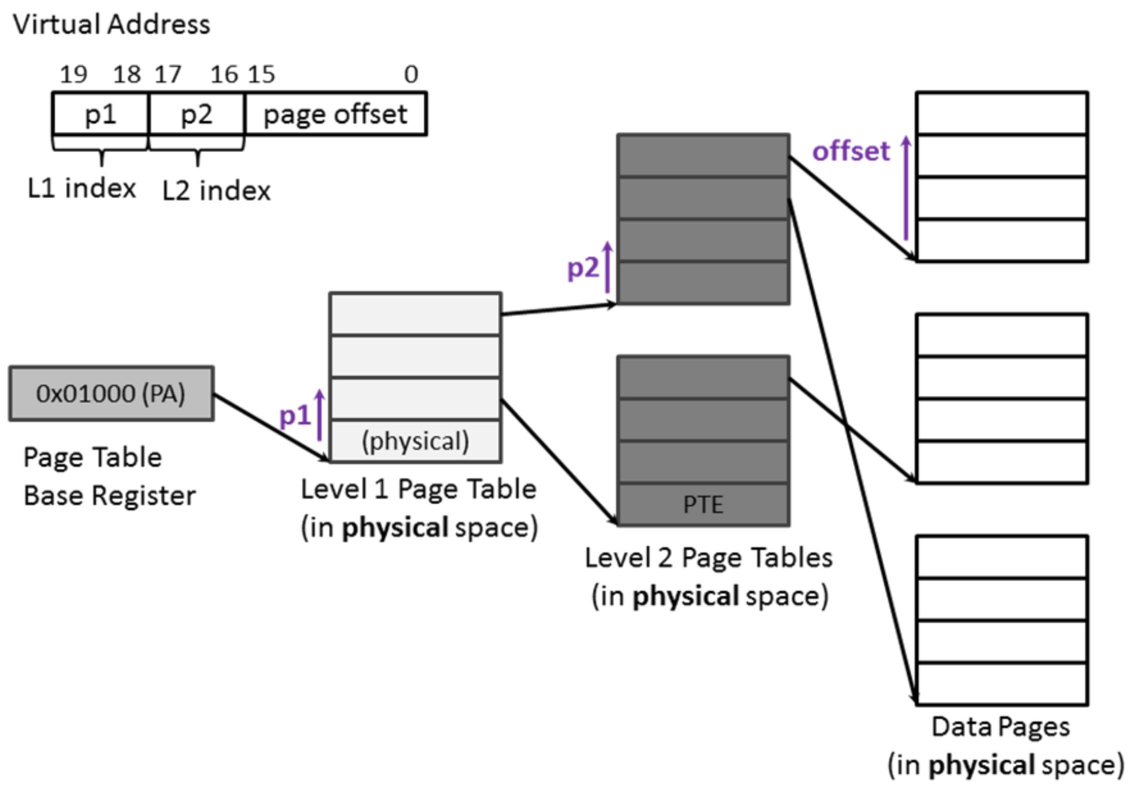
Problem M2.14.D

Now, suppose the cache is **physically indexed**. A program wants to read from the same **virtual address** 0x6E, and the physical address of this virtual address 0x6E is unknown (could be arbitrary). Enumerate **all** blocks (A,B,C,D,…) that can possibly hold the content of the virtual address 0x6E, for each given page size in the next table.

Page Size	Block(s) which can be mapped to
16 bytes	B, D, F, H, J, L, N, P
32 bytes	B, F, J, N
64 bytes	F, N

Problem M2.15: Hierarchical Page Table & TLB (Fall 2010 Part B)

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).

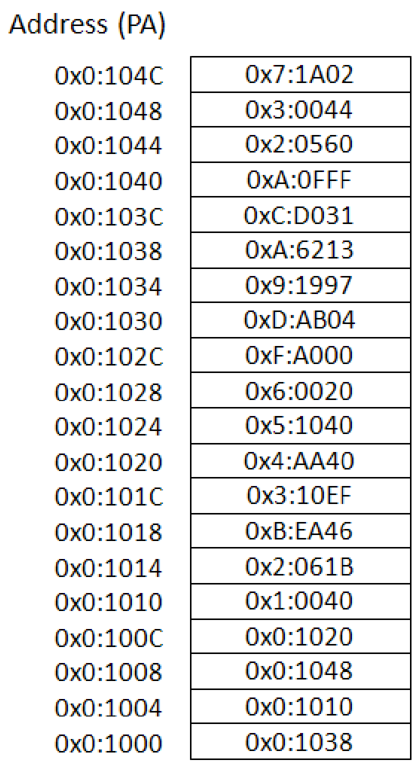
31 20	19 16	15 0
0	Physical Page Number (PPN)	Status Bits

Problem M2.15.A

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)

<u>**Virtual Address**</u>:

0xE:17B0 (1110:0001011110110000)

VPN	PPN
0x8	0x3
0xE	0x6

Final TLB states

VA 0xE17B0 => PA __________0x617B0___________

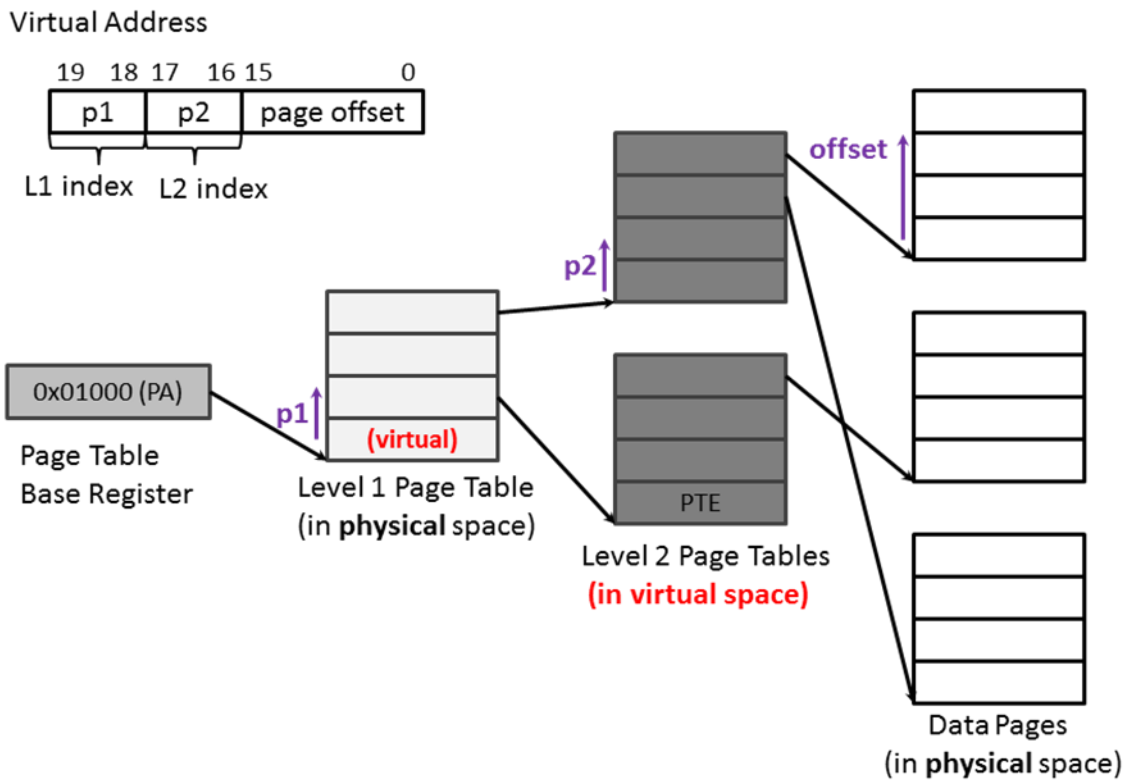
Problem M2.15.B

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

4 * 4 (level 1) + 4 * 4* 4 (level 2) = 80 bytes

Problem M2.15.C

Ben Bitdiddle 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.**)



Ben'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 M2.15.A), 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)

<u>**Virtual Address**</u>:

0xA:0708 (1010:0000011100001000)

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

Final TLB states

VA 0xA0708 => PA ____________0xF0708___________

Problem M2.15.D

Alice P. Hacker examines Ben’s design and points out that his scheme can result in infinite loops. Describe the scenario where the memory access falls into infinite loops.

1. When the TLB is empty
2. When the VPN of the virtual address and the VPN of the level 1 page table entry are the same