Last updated:
2/20/2020

Problem M4.1: Virtual Memory Bits

Problem M4.1.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 M4.1.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 M4.1.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 M4.1.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 M4.1.E

The solution for Problem M4.1.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 M4.1.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 M4.1.D will be faster.

Problem M4.1.F

Yes, the R bit can be removed in the same way we removed the V bit in 5.1.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 M4.1.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 M4.1.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 M4.2: Page Size and TLBs

Problem M4.2.A

43 — 33	32 — 22	21 — 0
L1 index	**L2 index**	**Page Offset**
11 bits	11 bits	22 bits

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 M4.2.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 M4.2.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 M4.2.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 M4.2.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 M4.3: Page Size and TLBs

Problem M4.3.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 M4.3.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 M4.3.C

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

Problem M4.3.D

4. (1 for L1 and 3 for L2)

Problem M4.4: 64-bit Virtual Memory

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

Problem M4.4.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 M4.4.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 M4.4.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 M4.4.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 M4.4.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 M4.4.F — Variable Page Sizes

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

Problem M4.4.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 5.

Problem M4.5: Cache Basics

Problem M4.5.A

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 M4.5.B

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 M4.6: Handling TLB Misses

Problem M4.6.A

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

VPN	PPN
0x0100	0x0F01
0x0003	0x00D4

TLB states

Problem M4.6.B

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

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

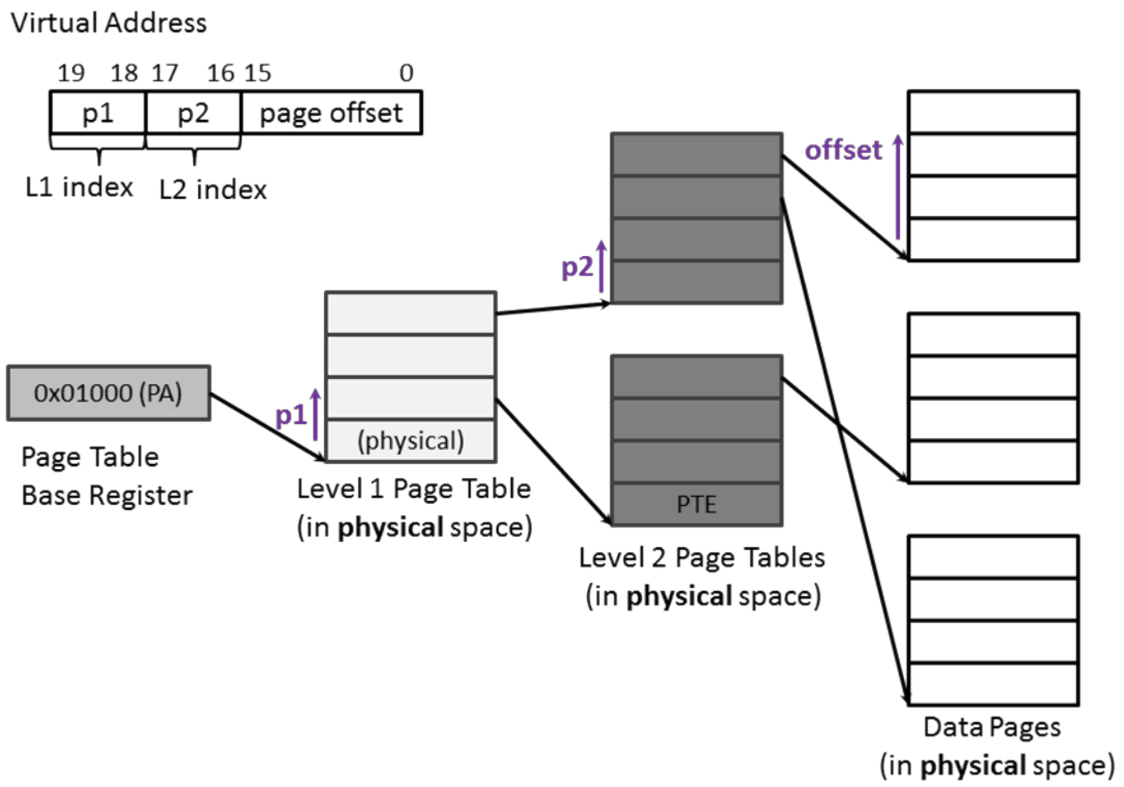
TLB states

Problem M4.6.C

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

Problem M4.7: 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 M4.7.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___________

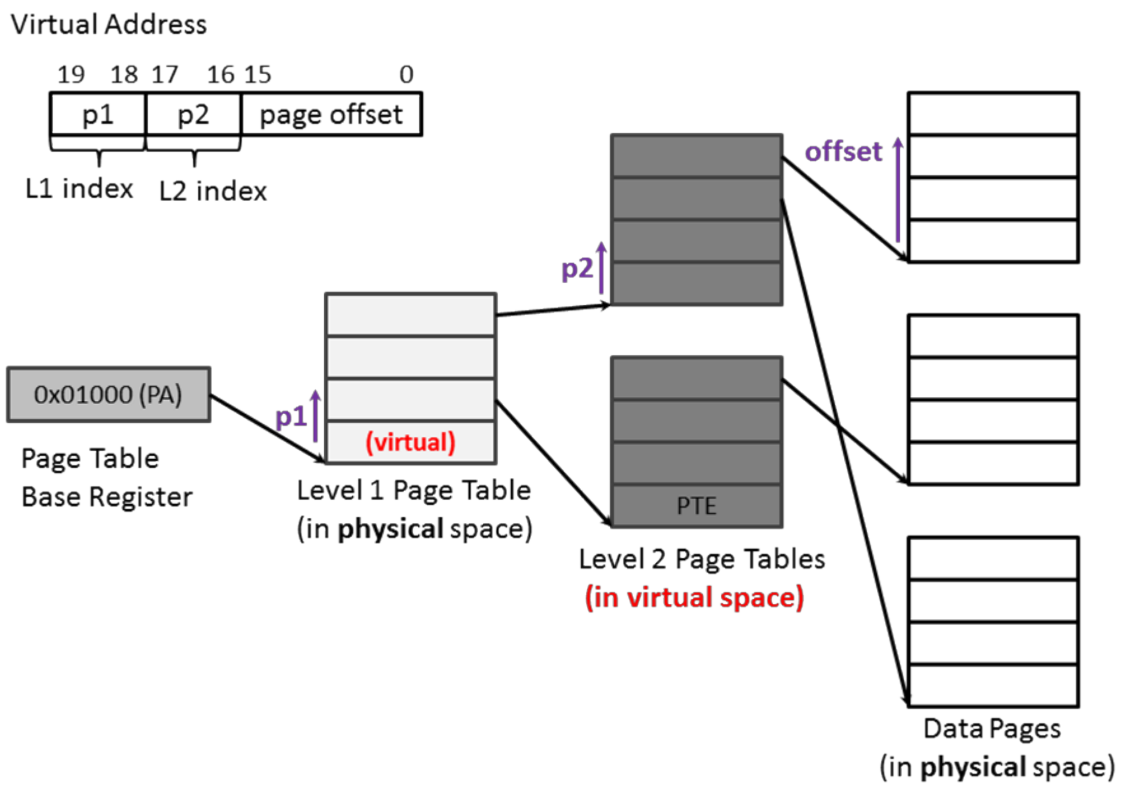
Problem M4.7.B

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 M4.7.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 M5.8.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 ":".***

.

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)

VPN	**PPN**
0x8	0x1

Initial TLB states

<u>Virtual Address</u>:

0xA:0708 (1010:0000011100001000)

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

Final TLB states

VA 0xA0708 => PA ____________0xF0708____________

Problem M4.7.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

Problem M4.8: Caches and Virtual Memory (Spring 2015 Quiz 1, Part B)

Problem M4.8.A

1.

Access	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Address	A	B	C	D	A	B	C	D	A	B	C	D
Entry 1	-	A	A	C	C	A	A	C	C	A	A	C
Entry 2	-	-	B	B	D	D	B	B	D	D	B	B
Hit?	N	N	N	N	N	N	N	N	N	N	N	N

What is the long-term miss ratio under LRU? 100%

2.

Access	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Address	A	B	C	D	A	B	C	D	A	B	C	D
Entry 1	-	A	A	A	A	A	B	C	C	C	C	C
Entry 2	-	-	B	C	D	D	D	D	D	A	B	B
Hit?	N	N	N	N	Y	N	N	Y	N	N	Y	N

What is the long-term miss ratio under optimal replacement? 66% (2/3)

3. Yes, MRU (Most Recently Used) achieves the same miss ratio since a data that has been most recently hit in the cache will be accessed the latest.

Problem M4.8.B

1. First divide the 16 address bits into the tag, index, and block offset bits. Since there are 128=2^7 bytes per block, we have 7 block offset bits. 8 sets similarly gives us 3 index bits. Thus, bits 7-9 will be our index bits, which is 111_2 .

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7	X	X	X	X

2. Now, given the physical address, we must figure out which bits of the virtual address we can know for sure. Since we have 256-byte pages, we have 8 bits of page offset. Since the page offset is the same between virtual and physical addresses, we know that the last 8 bits of our virtual address will be 0xCD, while not knowing anything about the upper 8 bits. Thus, this gives us index bits $xx1_2$, where x denotes bits we do not know. Thus, all odd rows of the cache are potential places where the data could reside.

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1	X	X	X	X
2				
3	X	X	X	X
4				
5	X	X	X	X
6				
7	X	X	X	X

3. Now, the index bits are contained entirely within the offset bits, and thus bits 7-9 of the virtual address are the same as those of the physical address (i.e. 111_2).

Index	**Cache Contents**			
	Way 0	Way 1	Way 2	Way 3
0				
1				
2				
3				
4				
5				
6				
7	X	X	X	X

Problem M4.8.C

1.

TLB entry

L	Tag	PPN
0	10101	0x47

0xABBA	= 1010 1011 1011 1010
0x47BA	= 0100 0111 1011 1010
VPN	= 1010 1011
PPN	= 0100 0111
Page offset	= 1011 1010
Index to TLB	= 011 (from VPN bit 8-10)
Tag	= 10101 (from VPN bit 11-15)

Possible locations

	Way 0	Way 1
0		
1		
2		
3	X	X
4		
5		
6		
7		

2.

TLB entry

L	Tag	PPN
1	10101	0100 01

0xABBA	= 1010 1011 1011 1010
0x47BA	= 0100 0111 1011 1010
VPN	= 1010 10
PPN	= 0100 01
Page offset	= 11 1011 1010
Index to TLB	= 000 (from VPN bit 10 + 00)
Tag	= 10101 (from VPN bit 11-15)

Possible locations

	Way 0	Way 1
0	X	X
1		
2		
3		
4		
5		
6		
7		

3. 2*2*1K page + 6*2*256 page = 7K

4. Problem: Padding zeros limits large pages to locate only in entry 0 and 4.
   Do not pad index bits with 00 for large pages but use the first 3 bits from the VPN of large pages.
   Fixed reach: 8*2*1K page = 16K

Problem M4.9: Caches and Virtual Memory (Spring 2016 Quiz 1, Part B)

Problem M4.9.A

What is the page size of Ben's machine? __________256 bytes__________

Demarcate the physical address into the following fields: Physical Page Number (PPN), Page Offset

PA	__8__ bits	__8__ bits
	PPN	Page Offset

Problem M4.9.B

In steady state, how many misses from the TLB will Ben observe per iteration of the `while` loop (lines 3, 4) on average, if (state your reasoning):

a) the TLB is direct-mapped _____4_______

The array elements accessed have VPNs 0x0, 0x4, 0x8, 0xC. In a direct-mapped TLB, these map to the same index and hence the same entry in the TLB. Every access to each array element, replaces the translation already in the TLB. Hence every access results in a TLB miss in steady state.

b) the TLB is fully-associative _____0_______
(assume LRU replacement policy)

Here translations to all 4 array elements can reside in the TLB simultaneously. Hence in steady state, there are no misses in the TLB.

Problem M4.9.C

In steady state, how many total memory accesses will Ben observe per iteration of the `while` loop (lines 3, 4) on average, if (state your reasoning):

a) the TLB is direct-mapped ____16______

Each of the 4 misses has to do a page walk: 3 memory accesses
Plus 1 memory access for the actual data

b) the TLB is fully-associative ______4______
(assume LRU replacement policy)

TLB accesses hit. So the only memory access is to fetch the data value.

Problem M4.9.D

Alyssa proposes adding a PTC with 1 entry to the processor. Does this addition benefit the code snippet in Question 2? How many total memory accesses will Ben observe now, if:

a) the TLB is direct-mapped ____8_______

The array elements have the same L1 and L2 index, but differ only in their L3 index. The PTC caches the translation from [L1 | L2] → L3. Only the very first array access does the entire page walk. The subsequent accesses (which includes all accesses in steady state) miss in the TLB, but hit in the PTC. They skip the full page walk, instead accessing only the L3 table and fetching the data value.

2 + 2 + 2+ 2 = 8

[[2 → L3 table access (1) + fetch data value (1)]]

b) the TLB is fully-associative _____4______
(assume LRU replacement policy)

TLB accesses hit. So the only memory access is to fetch the data value.

Problem M4.9.E

	Virtually Indexed	Physically Indexed
Direct-mapped (a)	G	A, C, E, G
2-way set-associative (b)	C, G	A, C, E, G

Problem M4.10: Nested paging (Spring 2017 Quiz 1, Part B)

Problem M4.10.A

Assuming the initial memory state is as shown to the right, what is the physical page number (`PPN`) of virtual address (`VA`) `0xB29A0`? What is the physical address (`PA`)? Show and explain your work for full credit. **For your convenience, we separate the page number from the offset with a colon ":".**

__Virtual Address__

```
0xB:29A0 =    0b 1011:0010100110100000
```

The L1 base `PA` is `0x02000` (shown in the figure).
The L1 index of the given `VA` is 2 (`0b10`).
Since PTP entries are 32 bits (4 bytes), the L1 PTP appears at
`0x02000 + 2x4 = 0x02008`.
Reading that `PA`, the base `PA` of the L2 table is `0x02038`.
The L2 index of the given `VA` is 3 (`0b11`).
Therefore the L2 PTE appears at `0x02038 + 3x4 = 0x02044`.
Reading that `PA`, the L2 PTE is `0x293A4`.
We ignore the 16 status bits and extract page number 2.

`VPN 0xB => PPN` _____**`0x2`**__________

The page offset from the `VA` doesn't change, so append the `VA` offset after `PPN` 2 to construct the physical address.

`VA 0xB29A0 => PA` ____**`0x229A0`**_________

Problem M4.10.B

Ben starts his foray into virtualization by thinking about `gPA=>hPA` translation.

a) Assuming Ben's host physical memory has the same snapshot as in Question 1, what is the host physical address (`hPA`) of guest physical address (`gPA) 0xB29A4`? Explain.

As stated in the Nested Paging handout, a `gPA` appears to the host system as a host virtual address. Therefore, `gPA 0xB29A4` is an address 4 bytes higher than the (host) virtual address of Question 1, and its (host) `PPN` remains the same: 2. We append page offset `0x29A4` to that `PPN`.

`gPA 0xB29A4 => hPA ____`**`0x229A4`**`__________`

b) Assuming no TLB, how many accesses to host physical memory are required to access the data associated with a `gPA` (i.e., perform the `gPA=>hPA` translation and fetch the data)? Explain.

Three. No different than accessing the data associated with a host virtual address.

One indexed access into the host L1 table
One indexed access into the host L2 table, (i.e. two for gPA=>hPA translation)
One to fetch the data at the identified host physical address.

Problem M4.10.C

Given a guest virtual address (gVA), the first step of a nested page table walk is to load the relevant guest L1 page table PTP. This provides the base gPA of the guest L2 table. Ben is shocked at how much work is required!

a) Assume host physical memory is initialized as in Question 1 and as shown to the right, the Guest Table Base Address register holds 0xB29A0 (a gPA), and the Host Table Base Address register holds 0x02000 (a hPA). During a nested page table walk of guest virtual address (gVA) 0x61EAC, what are the contents of the guest L1 page table PTP entry? **For your convenience, we separate the page number from the offset with a colon ":".**

Address (PA)	
0x0:2000	0x0:2048
0x0:2004	0x0:2010
0x0:2008	0x0:2038
0x0:200C	0x0:2028
0x0:2010	0x1:0084
...	...
0x2:2998	0xD:2E5C
0x2:299C	0x3:A000
0x2:29A0	0x6:010C
0x2:29A4	0xA:74CC
0x2:29A8	0x7:30B1
...	...
0x5:2994	0x6:11A0
0x5:2998	0xB:F149
0x5:299C	0xC:7BFF
0x5:29A0	0x4:B020
0x5:29A4	0xF:A120
...	...
0xA:299C	0x8:A624
0xA:29A0	0x9:FEED
0xA:29A4	0x2:93A4
0xA:29A8	0xA:7440
0xA:29AC	0x3:FD40

Snapshot of **host** physical memory

<u>Guest Virtual Address (gVA)</u>

0x6:1EAC = 0b 0110:0001111010101100

The guest L1 base gPA is 0xB29A0.
The L1 index of the given gVA is 1 (0b01).
PTP entries are 4 bytes, so the guest L1 PTP appears at
gPA 0xB29A0 + 4 = 0xB29A4.
From Question 2, this corresponds to hPA 0x229A4.
Reading that hPA, we find the contents of the Guest L1 Table PTP

Guest L1 Table PTP = ___**0xA74CC**_________

b) Assume no TLB. Starting from some gVA, how many accesses to host physical memory are required to determine the guest L1 PTP entry of a guest virtual address (gVA)? Explain.

Three.

We can determine the gPA of the guest L1 PTP entry with zero memory accesses: add the gVA L1 index offset to the Guest Table Base Address (which appears in a register).

Given this gPA of the L1 PTP entry, we must load its data. From Q2b, 3 accesses to host physical memory are required to fetch the data of a gPA.

Problem M4.10.D

For the 2-level nested page table in the Nested Paging handout, assuming no TLB, how many accesses to host physical memory are required to perform a guest memory access? (i.e. given a gVA, find its corresponding hPA **and fetch the data**). Explain.

Nine.

Load the guest L1 PTP (from a gPA), whose data is the gPA of the guest L2 table's base.
Load the guest L2 PTE (from a gPA), whose data is the gPA of the data to be fetched.
Load the data (from a gPA).

Each load of these three gPAs requires 3 accesses to fetch the data.

Problem M4.10.E

For an M-level hierarchical guest page table and an N-level hierarchical host page table, assuming no TLB, how many accesses to host physical memory are required to perform a guest memory access? (i.e. given a gVA, find its corresponding hPA **and fetch the data**). Explain.

$(M+1)$ $(N+1)$

Generalizing Q2b, any gPA=>hPA translation requires walking the N levels of the host page table (which consists exclusively of hPAs, and therefore accesses to host physical memory). Given the data's hPA, one more access is required to fetch the data: $(N+1)$ accesses to fetch the data of a gPA.

The gVA=>gPA translation requires walking the M levels of the guest page table. The guest page table consists of gPAs. Each level of the walk requires accessing a gPA and fetching its data (to find the gPA address of the next level's base table), so the gVA=>gPA translation requires $M(N+1)$ accesses to host physical memory.

Given the final gPA of the gVA, one more gPA access is required to fetch the data, which requires $N+1$ accesses.

$M(N+1) + N+1$