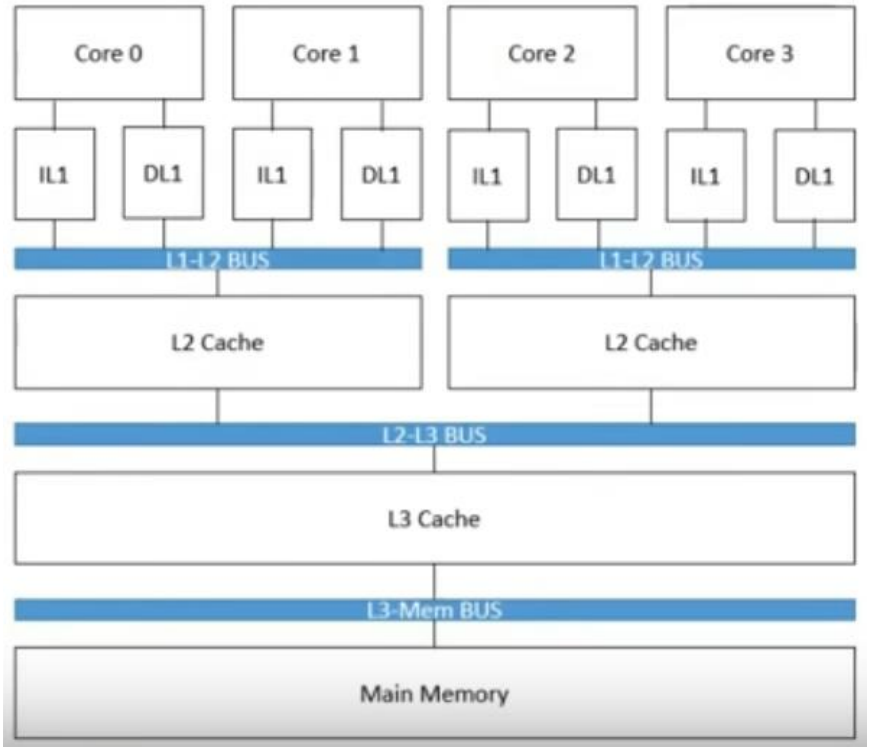
Sidharth Gilela
CMPE 110
12/4/15

Homework #4

1)



I used this diagram from Matt's section online to help me understand the flow of the states of each core for each request made. I realize that the L2 caches are different than the one compared in the homework assignment. All I need to know about the L2 cache in this assignment is that it is shared between all the cores, is 512KB, 8-way-set associative and a 64 bytes cacheline.

The specs of my L1 caches are:

- 32 KB each
- 2-way-set associative for each cache
- Uses write-back scheme
- 64 byte cacheline.

Other information that was useful for me is included here:

- Since each cacheline is 64 bytes and each word is 4 bytes, 64bytes/4bytes = 16 words per cacheline

For example if the starting address is 0x00ffabc0, then the remaining words in the cacheline are:

0x00ffabc0, 0x00ffabc4, 0x00ffabc8, 0x00ffabcc,
0x00ffabd0, 0x00ffabd4, 0x00ffabd8, 0x00ffabdc,

0x00ffabe0, 0x00ffabe4, 0x00ffabe8, 0x00ffabec
0x00ffabf0, 0x00ffabf4, 0x00ffabf8, 0x00ffabfc

- As shown above, these would be the 16 words in the cacheline that would be put into the level one cache of the core that it is requested for, if that starting address was read.
- Load is a read and write is a store
- First the tag where there is f is put into the cacheline, and then the tag with a. When the tag with b gets puts in the tag with f goes away because these first level caches are only 2-way-set associative.

From all of the above information and after watching the video about cache coherence that Matt posted, I was able to come up with this table:

		Request		C0 L1	C1 L1	C2 L1	C3 L1
	Core	Type	Address	State	State	State	State
1	0	Load	0x00ffabc0	E	-	-	-
2	0	Store	0x00ffabc8	M	-	-	-
3	1	Load	0x00ffabd4	S	S	-	-
4	1	Store	0x00ffabd8	I	M	-	-
5	1	Load	0x00afabc0	-	E	-	-
6	2	Load	0x00afabc8	-	S	S	-
7	1	Load	0x00bfabf0	-	E	-	-
8	0	Load	0x00ffabc0	S	-	-	-

Additionally, here are some descriptions for each of the steps that were done:

Step1 - Core 0 is exclusive because that is the only core that has read in this address for far.

Step2 – Core 0 is modified because the address is written into core 0. When an address is written into the core, and the other cores use the same cacheline, the other cores will become invalidated.

Step3 – Core 1 is shared because is now shares the same cacheline as core 0

Step4 - Core 1 is modified because the address is written into core 1. When an address is written into the core, and the other cores use the same cacheline, the other cores will become invalidated.

Step5 – Core 1 is exclusive because it reads in a cacheline with a different tag. Thus, for this new cacheline in this core, it is exclusive, invalidating, the other cores that use this cacheline since at this point, no other cores do.

Step6 – Core 2 shares this cacheline with core 1 because both these cores use the same cacheline with the tag that has a

Step7 – Core 1 is exclusive because it reads in a cacheline with a different tag. Thus, for this new cacheline in this core, it is exclusive, invalidating, the other cores that use this cacheline since at this point, no other cores do.

Step8 – Core 0 is exclusive because it reads in a cacheline with a tag that has already been invalidated. Thus, for this new cacheline in this core, it is exclusive, invalidating, the other cores that use this cacheline since at this point, no other cores do.

2) 2A) This is the table I have come up with using the knowledge of how cores work. Below the table, I have listed my logic as to why I believe each step causes each core to have a certain state. Here is the table:

		Request	C0 Cache	C1 Cache	C2 Cache	C3 Cache
	Core	Type	Line State	Line State	Line State	Line State
1	0	Read x	V	-	-	-
2	1	Read x	I	V	-	-
3	2	Read x	I	I	V	-
4	3	Write x	I	I	I	V
5	1	Read x	I	V	I	I

Here is my analysis for each step:
Information that makes the analysis pretty short:

- In VI protocol each core cacheline can either be valid or invalid. When a read or write is done, the cacheline state would become valid if it was either previously invalid or not set yet to either.
- Also, all the level one core cachelines can only have one valid cacheline making the rest of the cachelines of the other cores to invalid. This is on the assumption that the same tag is used for one cacheline of each core.

Step1) Since a read is done onto a cold cache, the cacheline state would be validated making it a V

Step2) The core 1 cacheline is read making this cacheline in core 1 valid. Additionally, since only one core can have a valid cacheline, this will invalidate the cacheline in core 0.

Step3) The core 2 cacheline is read making this cacheline in core 2 valid. Additionally, since only one core can have a valid cachline, this will invalidate the cachelines in core 0 and core 1.

Step4) The core 3 cacheline is being written making this cacheline in core 3 valid. Additionally, since only one core can have a valid cacheline, this will invalidate the cachelines in core 0, core 1 and in core 2.

Step5) The core 1 cacheline is being read making this cacheline in core 1 valid. Additionally, since only one core can have a valid cachline, this will invalidate the cachelines in core 0, core 2 and in core 3.

2B) This is the table I have come up with using the knowledge of how cores work. Below the table, I have listed my logic as to why I believe each step causes each core to have a certain state. Here is the table:

		Request	C0 Cache	C1 Cache	C2 Cache	C3 Cache
	Core	Type	Line State	Line State	Line State	Line State
1	0	Read x	S	-	-	-
2	1	Read x	S	S	-	-
3	2	Read x	S	S	S	-
4	3	Write x	I	I	I	M
5	1	Read x	I	S	I	S

Here is my analysis for each step:

Step1) In MSI protocol when a read is done to a cold cache, the state of that cacheline would be shared. Part of the reason why is because the other states of the MSI protocol wouldn't make sense for this case. If there was an option for exclusive this would be exclusive but we do not have that option for this protocol.

Step2) The state of the cacheline for core 1 would be shared because there is a cold cache and a new cacheline is being read. The cacheline that is being read however already exists in another core and is thus the reason why this new cacheline is shared. The cacheline in core 0 stays the same because both core 0 and core 1 have the same cacheline that hasn't been modified. Thus, core 0 stays as shared and now core 1 is shared as well.

Step3) The state of the cacheline for core 2 would be shared because there is a cold cache and a new cacheline is being read. The cacheline that is being read however already exists in another core and is thus the reason why this new cacheline is shared. The cacheline in core 0 and 1 stays the same because core 0, core 1, and core 2 have the same cacheline that hasn't been modified.

Step4) When a write is done to cache of core 3, the cacheline will be M for modified. Whenever a write is done in an MSI or an MESI protocol, the state of that cacheline will be modified. Additionally, since this cacheline gets modified and the other cores have this same cacheline, this cacheline in the other cores, gets invalidated making their states to an I. This is because when writing the address to the cacheline, the value could change.

Step5) When core 1 tries to now read this new cacheline, this cacheline changes to the state of S which is shared. This is because the core 3 already has the modified cacheline and core 1 is reading from that same cacheline. This in return would change the state of the cacheline in core 3 to shared as well because it shares the same cacheline as core 1.

2C) This is the table I have come up with using the knowledge of how cores work. Below the table, I have listed my logic as to why I believe each step causes each core to have a certain state. Here is the table:

		Request	C0 Cache	C1 Cache	C2 Cache	C3 Cache
	Core	Type	Line State	Line State	Line State	Line State
1	0	Read x	E	-	-	-
2	1	Read x	S	S	-	-
3	2	Read x	S	S	S	-
4	3	Write x	I	I	I	M
5	1	Read x	I	S	I	S

Here is my analysis for each step:

Step1) In MESI protocol since there is a cold cache, and the first address is read, the cacheline is brought from main memory. Since, no other caches have this same cacheline, the state of the cacheline in core 0 would be exclusive.

Step2) The state of the cacheline for core 1 would be shared because there is a cold cache and a new cacheline is being read. The cacheline that is being read however already exists in another core and is thus the reason why this new cacheline is shared. The cacheline in core 0 will now change to shared because both this core and core 1 have the same cacheline that hasn't been modified or changed in any way.

Step3) The state of the cacheline for core 2 would be shared because there is a cold cache and a new cacheline is being read. The cacheline that is being read however already exists in another core and is thus the reason why this new cacheline is shared. The cacheline in core 0 and 1 stays the same because core 0, core 1, and core 2 have the same cacheline that hasn't been modified.

Step4) When a write is done to cache of core 3, the cacheline will be M for modified. Whenever a write is done in an MSI or an MESI protocol, the state of that cacheline will be modified. Additionally, since this cacheline gets modified and the other cores have this same cacheline, this cacheline in the other cores, gets invalidated making their states to an I. This is because when writing the address to the cacheline, the value could change.

Step5) When core 1 tries to now read this new cacheline, this cacheline changes to the state of S which is shared. This is because the core 3 already has the modified cacheline and core 1 is reading from that same cacheline. This in return would change the state of the cacheline in core 3 to shared as well because it shares the same cacheline as core 1.

3) Number of pages in virtual memory

Info: 40-bit virtual address with 16KB pages and 4 bytes per page table entry
40 bit address can address => 2^40 bytes
Each page = 16KB => 2^14 bytes
(2^40 bytes)/(2^14 bytes) = 2^26 addressable pages

Maximum size of addressable physical memory in system

4 bytes entries => 2^32 pages able to be referenced
Each page = 2^14 bytes
Thus, (2^14 bytes) * (2^32 pages) = 2^46 = physical page number size

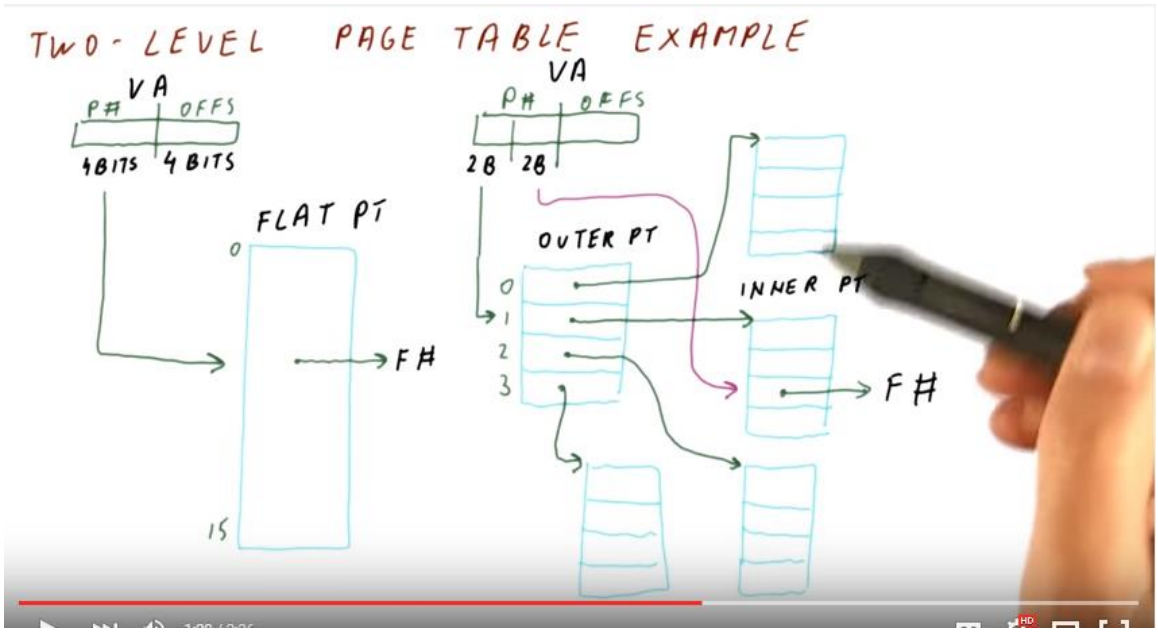
Size of one-level page table

2^26 pages in each virtual address space

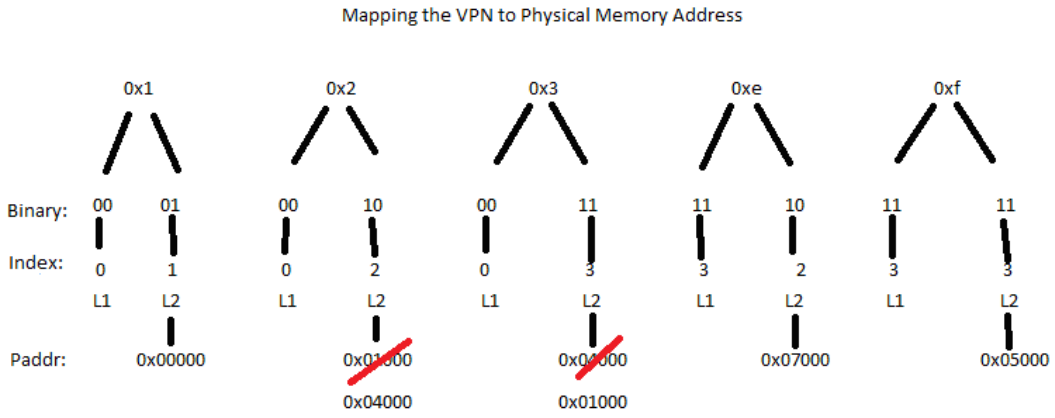
4 bytes => 2^2 bytes per page table entry
Size => (2^26 pages) * (2^2 bytes) = 2^28
Notice: 1/(2^5) of the average process size is 2^28
Thus, 1/(2^5) * (8GB) = 0.25GB = size of level-one page table

Size of two-level page table
First, need to split the number of pages in half since this is a two-level page table
40 – 14 = 26 which then => 26/2 = 13 bits for each level
Other info: Each page is 2^14 bytes
Some more info: 8GB => 2^33 bytes = process size
Process accesses => (2^33 bytes)/(2^14 bytes) = 2^19 pages
Thus, the bottom level of page holds 2^19 references
(2^19 bytes)/(2^13 bytes) = 2^6 needed for the bottom level parts
Size => 1 * (2^13 bytes) * 4 + (2^6) * (2^13 bytes) * 4
=> 2.12992 MB

4) 4A) Here is the information summed up from this part of the problem:
4 KB pages, 2-level page table, each page table entry is 4 bytes
We also know that the virtual addresses are 16 bits each and that the physical addresses are 20 bits each.
Info: VPN stands for virtual page number
Next:
4 KB pages => 4096 bytes = 2^12 => 12 bits for the offset
Since 12 bits for the offset, 16-12 = 4 bits for VPN
Thus:
VPN: 4 bits and Offset: 12 bits
However:
We also need to take into consideration that this is a two-level page table so, we need to separate out the VPN bits to be able to index through the first level page table and second level page table. Since, on our diagram, there appears to be 4 addresses for the first level page table, we need two bits to address through the first level since 2^2 = 4. We can also see from the diagram provided that each level 2 page table has 4 addresses as well. Thus, we need two bits to address through each level 2 page table since 2^2 = 4.
Thus:
The first two bits of the address of the virtual address will be used to index through the first level page table and the next two bits will be used the index through a level two page table. The combination of these bits which equals 4 is the VPN and thus, the offset is 16-4 = 12 bits.
An image from a youtube video has better enabled me to understand how indexing through a two-leveled page table is done. Here is the picture from the video I watched:



After figuring out how to index through a two-leveled page table, I made a diagram that I redrew on paint that has helped me figure out what the where the physical memory address will be put into the two-leveled page table. Here is the diagram that I created that has helped me immensely:



From this diagram and all the information I have found and from analyzing matt's video, I was able to come up with all this information in the table:

Level	Index		Page-Table	Entry
		Paddr of PTE	Valid	Paddr
L1	3	0xffffc	1	0xfffe0
L1	2	0xffff8	0	0xfffd0
L1	1	0xffff4	0	0xfffc0
L1	0	0xffff0	1	0xfffb0
L2	3	0xfffec	1	0x05000
L2	2	0xfffe8	1	0x07000
L2	1	0xfffe4	0	disk
L2	0	0xfffe0	0	disk
L2	3	0xfffdc	0	disk
L2	2	0xfffd8	0	disk
L2	1	0xfffd4	0	disk
L2	0	0xfffd0	0	disk
L2	3	0xfffcc	0	disk
L2	2	0xfffc8	0	disk
L2	1	0xfffc4	0	disk
L2	0	0xfffc0	0	disk
L2	3	0xfffbc	1	0x01000
L2	2	0xfffb8	1	0x04000
L2	1	0xfffb4	1	0x00000
L2	0	0xfffb0	0	disk

I added some extra parts to this table to help me have an easier time in completing this table.

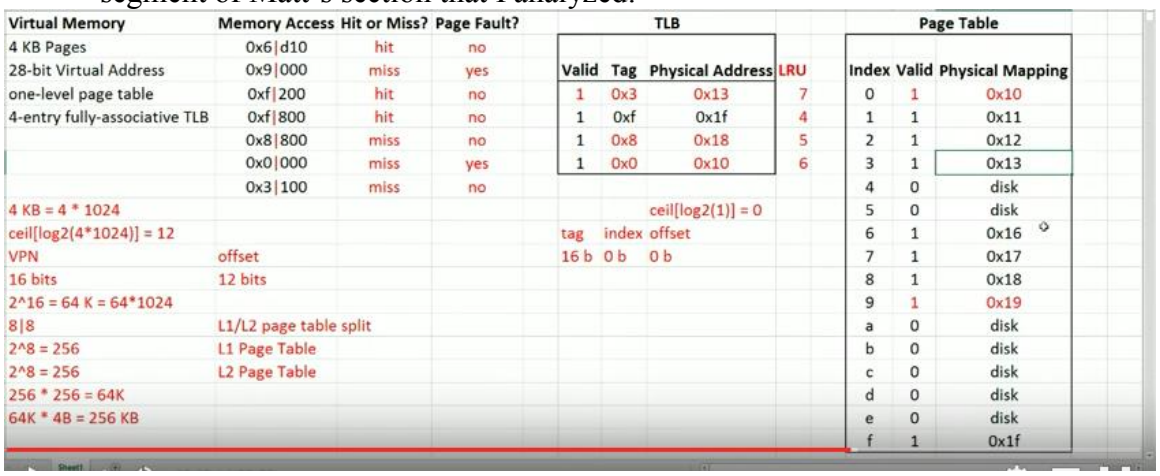
4B) In order to approach this problem, I first analyzed the specs of this TLB. Some information about this TLB is that it is a fully associative 2-entry TLB.

The next thing I did was map all the VPN's being used to their corresponding PPN's which can be figured out by looking at the diagram posted under the description of number 4. The arrow on that diagram from the virtual address space to the physical address space helps to figure out the PPN for each VPN. As figured out in 4A, the VPN is the first four bits of the virtual address. The PPN for each physical address is the first 8 bits because the total physical address is 20 bits and 12 of those bits are used as the offset. Thus, 20 – 12 = 8 bits for the PPN.

The table below shows my thought process of how the VPN has changed and how the LRU has been changing:

VPN => PPN					
0xe => 0x07			TLB		
0x2 => 0x04		Valid	VPN	PPN	LRU
0xf => 0x05		~~0~~ 1	~~e~~ f	…	1 3 5 7 9
0x3 => 0x01		~~0~~ 1	~~2~~ 3	…	2 4 6 8 10

Other information that has helped me complete the table includes the idea that there is a miss when the LRU is at 7 and 10 because that is when this TLB gets different VPN's of 0xf and 0x3.
Additionally, Matt's section online, has helped me gain an understanding with how a fully associative TLB functions. This picture below shows a segment of Matt's section that I analyzed:



Virtual Memory	Memory Access	Hit or Miss?	Page Fault?
4 KB Pages	0x6|d10	hit	no
28-bit Virtual Address	0x9|000	miss	yes
one-level page table	0xf|200	hit	no
4-entry fully-associative TLB	0xf|800	hit	no
	0x8|800	miss	no
	0x0|000	miss	yes
	0x3|100	miss	no
4 KB = 4 * 1024			
ceil[log2(4*1024)] = 12			
VPN	offset		
16 bits	12 bits		
2^16 = 64 K = 64*1024			
8|8	L1/L2 page table split		
2^8 = 256	L1 Page Table		
2^8 = 256	L2 Page Table		
256 * 256 = 64K			
64K * 4B = 256 KB			

TLB

Valid	Tag	Physical Address	LRU
1	0x3	0x13	7
1	0xf	0x1f	4
1	0x8	0x18	5
1	0x0	0x10	6

ceil[log2(1)] = 0

tag index offset

16 b 0 b 0 b

Page Table

Index	Valid	Physical Mapping
0	1	0x10
1	1	0x11
2	1	0x12
3	1	0x13
4	0	disk
5	0	disk
6	1	0x16
7	1	0x17
8	1	0x18
9	1	0x19
a	0	disk
b	0	disk
c	0	disk
d	0	disk
e	0	disk
f	1	0x1f

From all of this information I was able to come up with this table:

Transaction Address	VPN	Page Offset	m/h	Total Num Mem Accesses	TLB Way 0 VPN	TLB Way 0 PPN	TLB Way 1 VPN	TLB Way 1 PPN
0xeff4	0xe	0xff4	m	3	-	-	-	-
0x2ff0	0x2	0xff0	m	3	0xe	0x07	-	-
0xeff8	0xe	0xff8	h	1	0xe	0x07	0x2	0x04
0x2ff4	0x2	0xff4	h	1	0xe	0x07	0x2	0x04
0xeffc	0xe	0xffc	h	1	0xe	0x07	0x2	0x04
0x2ff8	0x2	0xff8	h	1	0xe	0x07	0x2	0x04
0xf000	0xf	0x000	m	3	0xe	0x07	0x2	0x04
0x2ffc	0x2	0xffc	h	1	0xf	0x05	0x2	0x04
0xf004	0xf	0x004	h	1	0xf	0x05	0x2	0x04
0x3000	0x3	0x000	m	3	0xf	0x05	0x2	0x04
0xf008	0xf	0x008	h	1	0xf	0x05	0x3	0x01
0x3004	0x3	0x004	h	1	0xf	0x05	0x3	0x01
0xf00c	0xf	0x00c	h	1	0xf	0x05	0x3	0x01
0x3008	0x3	0x008	h	1	0xf	0x05	0x3	0x01
Number of	Misses =	4						
Miss Rate =	(4)/(14) =	28.57%						

From this table I able to figure out the miss rate and the number of misses:
Number of Misses = 4 misses
Miss Rate = 4/14 = 28.57%
Total Memory Accesses = 4 misses * 3 accesses + 10 hits * 1 access
=> 22 Memory Accesses