Homework 9

1)configuration of the two L1 caches

1. Total number of bytes = 2^a where a is 14

We get this by looking at the 16KB size. 16 is 2^4 and K is 2^10. 2^4 x 2^10 = 2^14

2.bytes/line = 2 ^b where b is 5

We get this by looking at the block size which is also known as lines size. This is 32 bytes. 32 is 2^5

3.number of lines = 2^c where c is 9

I got this by doing the total number of bytes divided by the line size. 2^14 / 2^5 is 2^9

4.What bits of the address will be used as the tag?

Bits 14 – 31 are used for the tag. These are the remaining bits after removing the 5 needed for the offset and 9 needed for the index.

5.What bits of the address are used as the index?

Bits 5-13 are used for the index. We know it is 9 long because the number of lines is 2^9.

6. What control bits are required for I-cache?

The instruction cache only needs a valid bit. An instruction cache does not have the ability to write, so we would not need the dirty bit. In this type of cache, the replacement is dictated by the index since it is a direct mapped cache.

7. What control bits are required for D-cache?

The D-cache or data cache needs the valid bit and the dirty bit. A data cache can write so we need the dirty bit in this case. This cache is specified as a write back cache which creates the need for a dirty bit to let us know to write it back to main memory when we replace it in the cache. The replacement for this cache is directly mapped, so it is dictated by the index which removes the need for any other control bits.

2. determine the configuration of the L2 cache

1.Total bytes = 2^a where a is 18

We can figure this out by looking at the size of the cache. It is 256KB. 256 is 2^8 and K is 2^10. The total number of bytes would then be 2^8 * 2^10 which is 2^18

2. Number of sets = 2^b b in this case is 2

I got this because the cache is 4 way set associative. 4 is 2^2.

3. Number of bytes/set = 2^c c in this case is 16

I got this by looking at the total byes which was 2^18 and dividing this by the number of sets which is 2^2. This results in 2^16

4. Bytes/line = 2^d d in this case is 5

We get this by looking at the block size, otherwise known as the line size. In this case it was specified to be a 32-byte block size which is 2^5.

5.The number of lines per set = 2^e where e is 11

I got this by dividing the total number of lines by the number of sets. The number of lines are found by doing the cache size divided by the block size. In this case that is 2^18/ 2^5 which is 2^13. Then I took this and divided it by the number of set we have which is 4 way set associative or 2^2. 2^13/2^2 is 2^11.

6. What bits of the address will be used as the tag?

The bits used for the tag will be 16 – 32.

I got this by looking at how large the offset and the index were. The offset is 5 because the block size is 32 bits which is 2^5. This means 0-4 are used for offset. The index needs eleven bits because we have 2^11 lines per set. This takes up the next 11 bits which are 5-15. This leaves 16-32 for the tag.

7. What bits of the address will be used as the index?

The bits used for the index will be 5-15.

I got this by looking at how large the offset and the index were. The offset is 5 because the block size is 32 bits which is 2^5. This means 0-4 are used for offset. The index needs eleven bits because we have 2^11 lines per set. This takes up the next 11 bits which are 5-15. This leaves 16-32 for the tag.

8.What are the control bits required for the cache?

For this cache we need the valid bit and the dirty bit. Since the replacement is random; we do not need any extra control bits to handle the replacement algorithm. This would change if it was using something like least frequently used to replace data in the cache.

3. Write your detailed algorithm for processing an access on HW10 diagram (10 points)

I am not quite sure what you want here specifically but I put in some pseudocode and then a lot of writing on my plan. The paragraphs are to explain my plan of action in the same way I would explain lines of code when I am building this.

1. Test L1 (either L1I or L1D) depending on whether the access is an instruction of data to see if there is a hit (tag matches & V bit set).

To determine if there is a hit the code would look at the tags of the items in the caches at the index that matches the address we are looking for. These caches are directly mapped so the index should show us what is in that spot and we know to make sure that the valid bit is set for both the instruction and data cache. The tags need to match for a hit as well. The struct for this address has the l1tag variable. This will be the value we compare to the tag of the memory at that index already. The cache struct holds the valid/present bit variable that we will use to see if the cache has valid memory in it. We access these like the structs from last week. (currAcc.l1Tag would grab the tag of the current memory we are trying to match tags with)

(L1.p will grab the present or valid bit)

Once we have these values for the current access, we would need to grab the data in the cache at the same index that we got from the memory access. We can get this by using the L1I or L1D caches that are instantiated on line 40 of the hw10.c.

Something like :

L1I.tag[index] to get the tag for an instruction cache

L1D.tag[index] to get the tag for the data cache

Check if the tags match and if the valid bit is set.

Putting this all together:

If(L1I.tag[index] == currAcc.l1Tag && (L1.p == 1){

There has been a hit

//handle the hit

//increment hit and try stats

//make sure to update number transfers

}

Note: (L1I is for instructions and L1D is for data)

1. If there is a hit ….

If there is a hit that means, we have found the data we are looking for. It also means that we do not have to write this memory back to another place (either L2 or main memory). This comes from the idea of the truth table we looked at for caches where we said if there was a hit, we didn't have to write the data back anywhere. In this homework, the write up says we have stats for trys, hits and misses for all three caches. In this case we would increase the trys and the hit stats because we tried to find the data and had a successful hit. We also need to make sure that the number of clocks and transfers reflect how many accesses this takes. For the number of transfers, we need to get the correct mask for a 32-byte bus. I am not sure what the size of this mask is, but I know I will need a logical shift left to mask these off and then a bitwise and operation to isolate those bits. The masked off bits are added to the bytes of the access to get the span. It looks like most of the clocks are taken care of in the file given to us. For 4 bytes we needed two bits, and 4 bytes is 2^2 so I am assuming we might need 5 bits in this case since 32 bytes would be 2^5. If that is the case the code would look something like

```
mask = (1 << 5)-1;
```

i. If write …

If there is a write into this cache when we access the data we were looking for (there was a hit, so we found what we are looking for and now we are performing a write), then we need to make sure that the dirty bit is set for that line in the cache. If this line gets replaced in the cache, we do not want to lose what we wrote here so we need to make sure that it is marked as dirty. In the event it is replaced, the write back implementation will write this memory back if it is marked dirty right before it is replaced in the cache. Note we can only write to a data cache and the data cache is the one with the dirty bit while an instruction cache does not have that.

(cacheline.d will grab the dirty bit and we would set to one)

2. If there is a miss in L1 …..

   If there is a miss in L1, we need to go look for the data in the L2 cache. If it is not there, then we need to go out to the main memory to grab the thing we are looking for. If there is a miss in a L1D cache and the dirty bit is also set, we write back to the L2 cache. Either the tag doesn't match, or the valid bit isn't set. These can be checked the same way as we did for a hit, or we assume no hit = miss. I showed some rough pseudocode for this in hit.

   (currAcc.l1Tag would grab the tag of the current memory we are trying to match tags with)

   (L1.p will grab the present or valid bit)

   Once we have these values for the current access, we would need to grab the data in the cache at the same index that we got from the memory access.

   Check if the tags match and if the valid bit is set.

   If they match it is a hit

   Else miss

   ```
   If(L1I.tag[index] == currAcc.l1Tag && (L1.p == 1){
       There has been a hit
       //handle the hit
       //increment hit and try stats
       //make sure to update number transfers
   ```

}

Else{

Miss check the L2 cache

Example of going through L2 cache found in l2UpdateDirty()

For I = 0 to size of sets{

If the index in that set has a present bit of 1 and a tag that matches the tag of the data, we are looking for

Then we will mark that as the data found, grab it and place it in the index in L1

This would check all the sets at that index for the data we are looking for/ trying to fetch

}

i. if line is dirty …..

If the line is dirty and there is a miss, we need to write the memory back to L2. This was shown in the truth table for a data cache in the quiz. If we look at the code for hw10 in this case, it will call l2UpdateDirty. Which will update the L2 cache and set the clocks this takes.

Make sure that when it is written back it increases the number of writebacks for L1. Code that shows this increase can be found in line 375 of hw10.c.

Code: M.l1wbfound++;

ii. so, what needs to happen….

If there is a miss in the cache, we need to look at if the data is also marked dirty. If it is marked dirty and there is a miss, the data needs to be written back to L2. We also need to make sure to keep up to date with the stats for trys, hits, and misses. When we encounter a miss, we need to make sure that trys and misses are incremented in this case. If there is a hit and a write, then we need to make sure that the dirty bit is set for that data.

3. Make access to L2 to fetch data ….

If we need to make access to L2 to fetch data, this means there was a miss, and we are searching for the data. We would need to check each of the sets in the L2 cache to look for this data.

This would look something like the l2UpdateDirty I think. Something like the following rough pseudocode:

For I = 0 to size of sets

If the index in that set has a present bit of 1 and a tag that matches the tag of the data, we are looking for

Then we will mark that as the data found, grab it and place it in the index in L1

This would check all the sets at that index for the data we are looking for/ trying to fetch

2. Test L2 (all 4 sets) to see if there is a hit (how do we know?)

We know there is a hit if in one of the sets the tag matches and the valid bit is set. We could loop through and check each of the sets at this index the same way you did in l2UpdateDirty.

For I = 0 to size of sets

If the index in that set has a present bit of 1 and a tag that matches the tag of the data, we are looking for and it is marked a hit

If L2[i][index].p == 1 and currAcc.l1Tag (if present and the tag matches the tag we are finding){

//we found the data

}

If we make it through this for loop, then we didn't find it and it's a miss.

Up above I defined the mask we are using for transfers. Once that is set in Xtransfers it should not be special for an L2 cache.

We need to make sure to update stats variables appropriately. In the case of both a hit and miss the trys stats should be incremented while hit should be incremented if there is a hit and miss incremented only if there is a miss.

1. If there is a hit….

If there is a hit that means, we have found the data we are looking for. It also means that we do not have to write this memory back to another place. We do not have to write back to main memory or look for the data elsewhere. Increment trys and hit stats. Double check that transfers and number of clocks are working as intended.

i. . If a write ….

If there is a write to this cache, we need to set the dirty bit. (L2.d will grab the dirty bit and we would set to one). If this line gets replaced in the cache, we do not want to lose what we wrote here so we need to make sure that it is marked as dirty. In the event it is replaced, the write back implementation will write this memory back if it is marked dirty right before it is replaced in the cache. Note we can only write to a data cache and the data cache is the one with the dirty bit while an instruction cache does not have that.

2. If there is a miss in L2, ….
   i. Options ….

   We could look for the data in main memory. If we make it to L2 we know it's not in L1 so we would look in the main memory which takes more time but could result in finding it. I am not 100 percent on the other options you have since the only other place it could be is in main memory. If the cache is full, we would have to run a replacement algorithm. Otherwise, we could just place the just add that data to the cache. If we must replace something, the data there would need to be written to main memory if it is marked dirty since we are running a write back cache.

   ii. What replacement algorithm would you use? Why?

   It was determined for us that we are using a random replacement in this homework. This is good for this homework because it simplifies the model we are working with. We do not need as many control bits and the algorithm itself will be simpler since it is a random choice of what gets replaced.

   If I was working on this without the hw10 model, I think the algorithm that makes most sense is the least recently used. This algorithm makes sense to me because of the principle of locality. If it is least recently used there is a low chance it will be used again soon if there is a strong locality in the accesses, you are making.

   iii. If all sets have D&V bits set, what do we need to do?

   If all dirty and present bits are set, we need to replace a line with the data we were looking for. Since all of them are dirty we would need to write whatever

was replaced into main memory before it was replaced. Since there was a miss, the data isn't in the cache, so we need to make space for it using replacement.

3. Once the set is selected what needs to happen?

   Once a set is selected, we would need to update the tag and control info. The tag gets the tag of the new data in this area of the set. The present bit needs to be set to one. The dirty bit should be set to one as well.

4. Explain the steps required if you are going to replace a dirty line.

To replace a dirty line, the main concept is that you write this line back to main memory before it is replaced and then replace it with the new information you want in the cache.

We replace the line using the writeback method as defined in the hw10 outline. In the write back method, we writeback only when there is a miss, and the dirty bit is set. The way hw10 is defined it looks as though replacing a dirty line in L1D creates a writeback to L2 before the data line is replaced by the new information we want in the cache. The definition in hw10 shows that a replacement of a dirty line in L2 writeback to main memory.

The image below shows where I found information on replacing a dirty line in L1 cache.

```
345     /************************************
346      * If L1D p&d set but tag miss, need to write
347      * back to L2 & set dirty bit.
348      ************************************/

365         // reparse old address to L2 index & tag
366         index = oldAddr >> OFFSETL2I;
367         index = index & MASKL2I;
368         tag = oldAddr >> OFFSETL2T;
369         tag = tag & MASKL2T;
370
```

This image shows where I found information on the L2 cache dirty line replacement.

```
453     else {
454         // if miss, determine if replacement set is dirty
455         // l2setPtr points to next set to be replaced on a miss
456         if((L2[nextSet][index].p == 1) && (L2[nextSet][index].d == 1)) {
457             // need to write old line to memory
458             sM.l2wb++;
459             sM.memAccess++;
460             sM.clock = sM.clock + MEMCLOCKS;
461         } // end write dirty line
```

Steps for replacing dirty line in L1:

Take the address you are replacing and convert it into an address for the L2 cache.

Then we check for this address in the L2 cache to see if the line we are replacing already exists in the L2 cache. If it is found and doesn't need to be written to memory.

If it is not found, we write it to the memory and set the dirty bit.

Once it is written back to L2, we can replace this line with the new line we want in the cache.

Note:(The replacement here is determined by the index)

Steps to replace dirty line in L2:

Write the old line to the main memory.

Use random replacement to replace the line with the one we now want in the cache.

Note:(The replacement here is a random replacement algorithm)