Problem 1 (exercise 5.4 in PH, 20pts)

(a)

If the access hits the cache, the following pertains:

- On a read hit, the cache contains the data in question and therefore does not need to generate a memory system access.
- On a write hit, a write through cache updates both the value in the cache and the value in main memory. Thus, the cache must generate a memory access to update the word in memory being written to cache.

If the access misses the cache, the following is true:

- On a read miss, the cache fills the appropriate cache block from memory, which requires two memory system reads as each block is two words.
- On a write miss the memory system reads. Next, the cache writes the word of data to main memory as the cache is write through.

Summarizing the behaviors described above along with specific information from the exercise statement leads to Table 1. The frequency column is given by the product of the hit or miss frequency with the read or write frequency. As a cache block is two words and the memory system can read one word at a time, cache block fills require two accesses.

The average number of accesses to the memory system from the cache can be computed by summing the product of the frequency and the number of accesses from Table 1 across all possible situations. Doing so yields:
Thus, every access to cache results in an average of 0.35 accesses to main memory. We can determine the bandwidth used in this system by computing the number of accesses generated by system and dividing the result by the number of accesses the memory system can support:

\[ \text{Accesses}_{\text{Avg}} = 71.3\% \times 0 + 23.8\% \times 1 + 3.8\% \times 2 + 1.3\% \times 3 = 0.35 \]

\[
\text{Bandwidth}_{\text{used}} = \frac{0.35 \times 10^9}{10^9} = 35.0\%
\]

The numerator is the product of the average number of accesses to the memory system generated by each cache access and the number of accesses the processor makes to the cache. The denominator is the total bandwidth the memory system can support. The result indicates that 35\% of the available bandwidth to the memory system is used to support the cache in this scenario.

(b)
This exercise requires an approach similar to that used in (a), that is, to determine the amount of bandwidth currently in use we determine how many accesses are presented to the memory system on average.

If the access hits the cache:
- On a read hit, the cache contains the data in question and therefore does not need to generate a memory system access.
- On a write hit, the cache contains the data in question and therefore does not need to generate a memory system access. Because the cache is write back, a modified cache block is written to main memory only when it is replaced.

If the access misses the cache (implying a line within the cache is replaced):
- If the line being replaced is clean, the cache fills the appropriate cache block from memory requiring two memory system reads as each block is two words.
- If the line being replaced is dirty, the cache first writes the modified contents of the block to memory, requiring two memory system writes and then fills the appropriate cache block from memory also requiring two memory system reads.
The behavior with respect to misses depends only on the cleanliness of the block being replaced and is independent of whether a read or write is being done by the processor.

Summarizing the behaviors described above along with specific information from the exercise statement leads to Table 2, which provides an overview of how processor accesses map onto memory system accesses in the presence of a write back cache. As a cache block is two words and the memory system can read one word at a time, cache block fills and write backs require two accesses. The frequencies are the product of the hit or miss frequencies and the clean or dirty block frequencies.

The average number of accesses to the memory system from the cache can be computed by summing the product of the frequency and the number of accesses from Table 2 across all possible situations. Doing so yields:

\[
Accesses_{avg} = 66.5\% \times 0 + 28.5\% \times 0 + 3.5\% \times 2 + 1.5\% \times 4 = 0.13
\]

Thus, every access to cache results in an average of 0.13 accesses to main memory. We can compute the bandwidth used in this system by computing the number of accesses generated by the system and dividing this result by the number of accesses the memory system can support:

\[
Bandwidth_{used} = \frac{0.13 \times 10^9}{10^9} = 13\%
\]

The numerator is the product of the average number of accesses to the memory system generated by each cache access and the number of accesses the processor makes to the cache. The denominator is the total bandwidth the memory system can support. The result indicates that 13% of the available bandwidth to the memory system is used to support the cache in this scenario.

**Problem 2 (exercise 5.6 in PH, 20pts)**

There are two things that can be done to improve execution time: eliminate extraneous prefetches

<table>
<thead>
<tr>
<th>Access Hits Cache?</th>
<th>Is Block Dirty</th>
<th>Frequency</th>
<th>Memory Accesses Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>95%*70%=66.5%</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>95%*30%=28.5%</td>
<td>0</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>5%*70%=3.5%</td>
<td>2</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>5%*30%=1.5%</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Summary of Memory Accesses Generated by a Write Back Cache
reduce the number of non-prefetched cache misses. For the remainder of this solution, our remarks on the example refer to the code at the top of page 404 (the solution to the example). Also, from the discussion in the text, one can gather that the cost to execute a prefetch instruction is one cycle.

Eliminating extraneous prefetches can be done easily by splitting loops so that the final iterations of the loop that do not need to prefetch are separated from those iterations that do need to prefetch. Eliminating all cache misses is not possible. Also, without completely rewriting the code, some misses to “a” in the first loop are unavoidable.

Thus, our approach will be as follows: split loops where necessary to eliminate extraneous prefetch instructions and add prefetches during the first loop nest to avoid the misses to “a” in the second loop nest.

/* loop with last 7 iterations split to avoid extraneous prefetches */

for (j=0; j<93; j++) {
    prefetch(b[j+7][0]);                  /* prefetchs elements 7-99 */
    prefetch(a[0][j+7]);                  /* prefetchs elements 7-99 */
    prefetch(a[1][j]);                      /* prefetchs elements 0-92 */
    a[0][j]=b[j][0]*b[j+1][0];
}

/* last iterations with prefetches */

for (j=93; j<100; j++) {
    prefetch(a[1][j]);                      /* prefetchs elements 93-99 */
    a[0][j]=b[j][0]*b[j+1][0];
}

Figure 1: Alternations to First Loop Nest to Avoid Unnecessary Prefetches

We start with the first loop nest and split it so the last iterations are in a loop that do not prefetching for the current iterations at all. We also add prefetches for the elements of a[1][]. When the code shown in Figure 1 finishes execution, all of “b” and all of a[0][] and a[1][] will be in cache.

Moving on to the second loop nest from the original code, since all of a[1][] is in cache (due to earlier prefetches), we only need to prefetch a[2][] during the i=1 loop. We also split off the i=2 loop, so we can write it with no prefetches (since that is the last loop iteration and there is nothing to prefetch). Since we have now split the entire i-loop into separate code segments, there is no need to have any i-loop at all. Making these changes result in the code shown in Figure 2. Figuring out the cost of all this is easy:

- Each iteration of the first new loop nest costs 10 cycles per iteration and there are 93 iterations.
- Each iteration of the second new loop nest costs 8 cycles per iteration and there are 7 iterations.
- Each iteration of the third new loop nest costs 8 cycles per iteration and there are 100 iterat-
Each iteration of the fourth new loop nest costs 7 cycles per iteration and there are 100 iterations.

/* This is the code that would have been executed in the i=1 loop */

for (j=0; j<100; j++) {
  prefetch(a[2][j]); /* prefetch all elements for next loop nest */
  a[1][j]=b[j][0]*b[j+1][0];
}

/* This is the code that would have been executed in the i=2 loop */

for (j=0; j<100; j++)
  a[2][j]=b[j][0]*b[j+1][0];

Figure 2: Altered Second Loop Nest That Avoids to Unnecessary Prefetches and Cache Misses

There are still 11 cache misses in the first loop nest, which will add 550 cycles to the execution time. However, there are no cache misses in any of the remaining loops, so we can just add up all the numbers:

\[
\text{Time} = 10 \times 93 + 8 \times 7 + 8 \times 100 + 550 = 3036
\]

Thus, our result is 3036 cycles for the new and improved code compared to the 3450 cycles for the example in the text.

Problem 3 (exercise 5.8 in PH, 35pts).

This exercise evaluates the performance of a system with a cache and TLB. The first two parts of the exercise explore the CPI of the system with real caches and TLBs while the third part discusses some performance issues related to TLBs and caches. Our solutions for the first two parts are based on the following expression for CPI:

\[
\text{CPI} = \text{CPI}_{\text{execution}} + \frac{\text{Memory Stall Cycles}}{\text{Instruction}}
\]

Equation 3.1

We begin by examining the CPI due to execution, CPI\text{execution}, which accounts for the cycles taken to execute an instruction in the absence of memory stalls. The execution CPI does not take into account three sources of stalls:

- Those caused by fetching instructions from memory.
• Those caused by load or store instructions when they access data.
• Those caused by the TLB.

These stall are accounted for in the memory stalls per instruction term of Equation 3.1.

To determine the appropriate number of memory stalls per instruction, we break the stall term into three parts based on the sources of stalls listed above:

\[
\frac{Memory\ Stalls}{Instruction} = \frac{Instruction\ Stalls}{Instruction} + \frac{Data\ Stalls}{Instruction} + \frac{TLB\ Stalls}{Instruction} \quad \text{Equation 3.2}
\]

Where the fetch and data stall terms are of the form given in Section 5.2 of the text.

\[
\frac{Stall\ Cycles}{Instruction} = \frac{Memory\ Accesses}{Instruction} \times Miss\ Rate \times Miss\ Penalty \quad \text{Equation 3.3}
\]

For fetch stalls, the number of memory accesses per instruction is one, as all instructions require one memory access to fetch each instruction. In the case of data stalls, only data access instructions have a data memory access, which implies that the number memory accesses per instruction for data accesses is simply the frequency of data access instructions. Finally, the TLB term will be left in symbolic form until later in the solution. Substituting this information into Equation 3.2 yields:

\[
\frac{Memory\ Stalls}{Instruction} = R_i \times P_i + F_d \times R_d \times P_d + \frac{TLB\ Stalls}{Instruction} \quad \text{Equation 3.4}
\]

Where \( R \) is a miss rate, \( P \) is a miss penalty, and \( F_d \) is the frequency of data access instructions. The subscripts indicate the specific miss rate, miss penalty, or frequency (e.g., \( i \) implies instruction fetches, \( d \) implies data accesses, etc.). With Equations 3.1 and 3.4 we can evaluate the performance of the caches for (a) and (b).

(a)

In the Equation 3.4, the term for TLB stalls is zero as the TLB in this exercise is ideal and therefore can not stall the CPU. Also, all three cache configurations are unified, which implies that the miss rate and penalties are independent of the type of access (i.e., data versus instruction). In terms of Equation 3.4 this observation implies that \( R_i = R_d \) and \( P_i = P_d \).

Substituting this information along with the frequency of the data accesses (20% from the exercise statement) and the execution CPI (assumed to be one as per the discussion above) into Equa-
tion 3.1 and 3.4, leads to

\[ CPI = 1.5 + [R \times P + (R \times P \times 20\%) + 0] = 1.5 + 1.2 \times R \times P \quad \text{Equation 3.5} \]

where \( R \) is the miss rate and \( P \) is the miss penalty for the unified cache. The miss rate, \( R \), depends on the specific cache under consideration and can be found from the results presented in Figure 5.9 of the text. For a write back cache, the miss penalty depends on the time required to flush and fill a cache block. Block flushes and fills both require 48 cycles to complete, according to the exercise statement (40 cycles for memory latency and 8 cycles to fill the 32-byte block at 4 byte per cycle). While a block fill must always occur on a miss, a block flush only occurs if the block being replaced is dirty, which happens 50% of the time, according to the exercise. Thus, the miss penalty is 72 cycles (from the expression 48+(50%*48)=72). Keep in mind that caches do not have a write buffer and thus the miss penalty needs to account for the time required to do the flush. Plugging these values into Equation 3.5 leads to the results shown in Table 3. These numbers should help you appreciate how much the memory hierarchy can impact performance. Even with miss rate down around 2%, the memory hierarchy accounts for around 60% of the total CPI because of the huge penalty for the missing the cache.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Cache Miss Rate</th>
<th>Miss Penalty</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>16KB Direct</td>
<td>0.029</td>
<td>72</td>
<td>4.0056</td>
</tr>
<tr>
<td>16KB Two-Way</td>
<td>0.022</td>
<td>72</td>
<td>3.4008</td>
</tr>
<tr>
<td>32KB Direct</td>
<td>0.020</td>
<td>72</td>
<td>3.2280</td>
</tr>
</tbody>
</table>

\underline{Table 3: Summary of the Performance of Several Caches with an Ideal TLB}

(b)
To solve this exercise, we repeat the same process as (a). But this time we can not assume the value of the TLB stalls is zero as the TLB is no longer ideal.

For the purposes of this exercise, we assume the size of the pages mapped by the TLB is a multiple of the block size of the cache (32 bytes in this exercise).

By making this assumption, we can assert that a cache block can only reside on one page, and therefore filling a block from memory can generate at most one TLB access. In addition, from the exercise statement we can assume that the TLB only slows the processor down when both the TLB and cache miss.

To determine the number of TLB stall cycles per instruction we apply an equation that is similar to the equation for the number of memory stall cycles per instruction presented in Section 5.2 of
In this equation we have simply taken the memory stalls per instruction equation and modified the first term to account for the number of TLB stalls per instruction. If we let $F_d$ be the frequency of data access instructions, $R_t$ and $P_t$ be the miss rate and penalty of the TLB, and $R_c$ and $R_w$ be the miss rate and frequency of writebacks of the cache then this equation becomes:

\[
\frac{TLB\ Stalls}{Instruction} = \frac{Memory\ Accesses}{Instruction} \times \frac{TLB\ Accesses}{Instruction} \times TLB\ Miss\ Rate \times TLB\ Miss\ Penalty
\]

The first term of Equation 3.6, $1+F_d$, corresponds to the number of memory accesses per instruction. The one in this term represents the single access required by each instruction to fetch the instruction word while the $F_d$ represents the data access generated by data access instructions. The second term of Equation 3.6, $R_c \times (1+R_w)$, corresponds to the number of TLB accesses per memory access. Because the TLB can only potentially slow down the system on a cache miss, we multiply the second term by the miss rate of the cache, $R_c$. On a cache miss, the TLB must translate the address of the block to be read from memory and also may have to translate the address of the block to be flushed to memory if a writeback is necessary.

Thus, the second part of the second term is $1+R_w$ and accounts for two TLB accesses: one that always occurs to fill the block and another for the block flush 3 which occurs with frequency of $R_w$.

Plugging in the values for $F_d$, $R_w$, $R_t$ and $P_t$ that are given in the exercise statement leads to a final form of Equation 3.6

\[
\frac{TLB\ Stalls}{Instruction} = \left\{\left[1 + 20\%\right] \times \left[R_c \times (1 + 50\%\right)\right\} \times 0.2\% \times 20
\]

In this expression $R_c$ depends on the specific cache being evaluated and can be found in Figure 5.9 of the text. Substituting the appropriate miss rates for the three cache configurations we are considering leads to the results presented in Table 4. From these results we can see that the TLB increases the CPI by around 0.05% in each case. Although the penalty for missing the TLB is fairly severe (20 cycles), it does not happen very frequently.
The addressing of the cache impacts TLB performance by changing the performance demands required to keep the hit time small. In large physically addressed caches, some address translation needs to occur before the cache access can be completed. As a result, the TLB must be able to provide the translation quickly if hit time is to be kept low (essentially, in this case the TLB and cache are “in series”). For virtually addressed caches, the TLB can be accessed in parallel with the cache. This reduces the pressure on TLB performance as the cache does not need information generated by the TLB to operate. Because the TLB must be accessed prior to any cache access in a physically addressed cache, there is also the danger of additional wasted cycles if the address translation is not in the TLB but the desired data is in the cache. The extra time spent servicing the TLB miss would not be seen in a virtually addressed cache if the desired information is resident in the cache.

Problem 4 (exercise 5.14 (a) in PH, 20pts)

(a)
A small direct-mapped cache can potentially outperform a fully associative cache if the system is executing a loop that does not fit entirely in cache. To see how this can happen, consider a loop that accesses three unique addresses: A, B and C, and then repeats the sequence by looping back to A. The reference stream for such a program would look like this: ABCABCABC..., where each letter corresponds to an address in the reference stream. To simplify the discussion, we assume that our caches have only two blocks of storage space with addresses A and C mapping into the first block and address B mapping into the second block. For a fully associative version of the two-block cache, our reference stream always misses the cache if the replacement policy is LRU. Such behavior does not occur in a direct-mapped cache using LRU replacement because each address maps into a specific location of the cache. In the direct-mapped version, we only miss on an access to A or C but we always hit on an access to B (ignoring initial compulsory misses). Essentially, what happens is that the direct-mapped cache allows only certain addresses to be placed into a given block. This allows blocks to be “protected” in the sense that they can not be replaced by just any access, but rather only by accesses whose addresses also map into the same block. In our two-block caches, let us consider the state of the direct-mapped cache right before address C is accessed.

<table>
<thead>
<tr>
<th>Cache Configuration</th>
<th>Cache Miss Rate</th>
<th>Ideal TLB CPI</th>
<th>Real TLB Stalls</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>16KB Direct</td>
<td>0.029</td>
<td>4.0056</td>
<td>0.0020</td>
<td>4.0076</td>
</tr>
<tr>
<td>16KB Two-Way</td>
<td>0.022</td>
<td>3.4008</td>
<td>0.0015</td>
<td>3.4023</td>
</tr>
<tr>
<td>32KB Direct</td>
<td>0.020</td>
<td>3.2280</td>
<td>0.00144</td>
<td>3.22944</td>
</tr>
</tbody>
</table>

Table 4: Summary of the Performance of Several Caches with a Real TLB

(c)

The addressing of the cache impacts TLB performance by changing the performance demands required to keep the hit time small. In large physically addressed caches, some address translation needs to occur before the cache access can be completed. As a result, the TLB must be able to provide the translation quickly if hit time is to be kept low (essentially, in this case the TLB and cache are “in series”). For virtually addressed caches, the TLB can be accessed in parallel with the cache. This reduces the pressure on TLB performance as the cache does not need information generated by the TLB to operate. Because the TLB must be accessed prior to any cache access in a physically addressed cache, there is also the danger of additional wasted cycles if the address translation is not in the TLB but the desired data is in the cache. The extra time spent servicing the TLB miss would not be seen in a virtually addressed cache if the desired information is resident in the cache.
Problem 5 (28pts)

The following parameters apply to a system employing a 32-bit virtual address and 32-bit data CPU.

PART 1: Two-level Page Table Organization

- Page size: 1 KBytes
- Entries in the Page Directory and the Page Tables are 32 bits.
- Physical memory size: 128 MBytes (27 bits of physical address)
- Word size: 4 bytes. Addresses point to bytes.

Answer the following questions directly on Figure 3.

5.1 (4pts) Complete Figure 3 to show how the entries in the page directory and in the page table are found, using the appropriate bits of the virtual address (4 numbers are needed).

5.2 (6pts) For a virtual address equal to \texttt{435241H}, compute the memory addresses in hexadecimal of the page directory entry and of the page table entry (marked \texttt{?} in Figure 3). The entries of the page directory and of the page table for virtual address \texttt{435241H} are shown in Figure 3.

5.3 (2pts) Find the physical address and the page offset in hexadecimal for virtual address \texttt{435241H} given the entries in the tables of Figure 3. Answer on Figure 3.

PART 2: TLB Access

- TLB size: 512 entries
- TLB organization: Direct mapped
- The TLB access is as in a direct-mapped cache. First the TLB is indexed meaning that the TLB entry is fetched in parallel. Second the tag from the TLB is matched with the most significant bits of the virtual page number.

Answer the following questions immediately below each question

5.4 (2pts) Which bits of the virtual address are used to index the TLB:
Answer: bits A18 to A10

5.5 (2pts) Which bits of the virtual address are used as tags in the TLB?
Answer: bits A31 to A19

5.6 (2pt) For the virtual address \texttt{435241H} indicate the set number accessed in the TLB (in binary)
Answer: Set Number = \texttt{011010100}

PART 3: Cache Access (Refer to Figure 4)

- Cache size: 96 KB
- Block size: 128 bytes
• Set size: 3 blocks per set

The cache is organized in three columns. Columns 0, 1, and 2 contain block 0, 1, and 2 (respectively) of all sets. The directory is made of 3 Tag RAMs, one for each column and each Tag RAM contains the tag plus 4 state bits, one of them being the valid bit. The tag and the valid bit are matched in a comparator to the tag from the physical address.

Please answer the following directly on Figure 4 (10 numbers are needed)

5.7 (4pt) Which bits of the physical address are used to access each tag RAM of the directory? Which bits of the physical address are matched against the tags in the Tag RAMs?

5.8 (2pts) Which bits of the physical address are used to access the cache data RAM?

5.9 (2 pts) What are the width and height of each Tag RAM in the directory (including state bits)?

5.10 (2pts) What is the height of each bank of the cache data RAM?
Physical Address for VA=435241H=516F241H

Page Offset for VA=435241H=241H

NOTE: PPN = Physical Page Number