HW #4 Solutions

1 a) Direct-mapped cache with 16 one-word blocks

Word Address	Hit/Miss	Type
2	Miss	Compulsory
3	Miss	Compulsory
11	Miss	Compulsory
16	Miss	Compulsory
21	Miss	Compulsory
13	Miss	Compulsory
64	Miss	Compulsory
48	Miss	Compulsory
19	Miss	Compulsory
11	Hit	
3	Miss	Conflict with 19
22	Miss	Compulsory
4	Miss	Compulsory
27	Miss	Compulsory
6	Miss	Compulsory
11	Miss	Conflict with 27

Line	Cache Word
0	~~16~~, ~~64~~ , **48**
1	
2	**2**
3	~~3~~, ~~19~~, **3**
4	**4**
5	**21**
6	~~22~~, **6**
7	
8	
9	
10	
11	~~11~~, ~~27~~, **11**
12	
13	**13**
14	
15	

b) 2-way Set Associative

Word Address	Hit/Miss	Type
2	Miss	Compulsory
3	Miss	Compulsory
11	Miss	Compulsory
16	Miss	Compulsory
21	Miss	Compulsory
13	Miss	Compulsory
64	Miss	Compulsory
48	Miss	Compulsory
19	Miss	Compulsory
11	Hit	
3	Miss	Conflict with 19
22	Miss	Compulsory
4	Miss	Compulsory
27	Miss	Compulsory
6	Miss	Compulsory
11	Miss	Conflict with 27

Line	Cache Word Set 0	Cache Word Set 1
0	~~16~~, **48**	**64**
1		
2	**2**	
3	~~3~~, ~~19~~, ~~3~~, 11	~~11~~, **27**
4	**4**	
5	**21**	**13**
6	**22**	6
7		

3. Yes, the shown form of decoding gives rise to problems. If the Index bits are in the more significant locations than the Tag bits, while accessing data spatially close to each other in memory, the Index bits might not change or show minimal change, causing data to map to the same locations in the cache. This would lead to increased number of misses.

4. As discussed in class (will not be graded).

5. Using a 32-bit virtual address and 4 KB page size, the virtual address is partitioned into a 20-bit virtual page number and a 12-bit page offset. We divide the virtual page number into two 10-bit fields. The first field is the page table number and is used as an index into the first-level page table. The size of the first-level page table in 2^{10} entries×4 bytes/entry = 2^{12} bytes = one page.

2.1.

```
char* find_letter(char letter, int wordSize, int wordsSize, char** words)
{
   for(int i = 0; i < wordsSize; i++)
   {
     for(int j = 0; j < wordSize; j++)
     {
       if(words[i][j] == letter)
       {
         return words[i];
       }
     }
   }
   return 0; //Not found
}
```

2.2. It's slower because it's going column major instead of row major. This causes there to be a greater number of cache misses as the list of strings gets longer.

2.3. It takes better advantage of spacial locality by reading all of one string before advancing to the next.

2.4. Consider two strings: 1 million 'a's and one 'b'. In this system. The old function will find the 'b' first by checking the columns, whereas the new will have to read all of the 'a's before finding the 'b'.