CS 152
Computer Systems Architecture
Summer Session II 2018

Homework 4

Due September 6th, 2018, EEE 11:59PM PDF

Name	*Student ID*
Cristian Gonzales	**45349991**

IMPORTANT NOTES:

1. **No late submissions.**
2. **Please show your work. Remember that bottom line answers without proper explanations are worth ZERO points.**
3. **Even if you consult with your peers, do make sure the answers herein are yours and that you do not copy or cut-and-paste from other people's work.**

For Grading Purposes Only:

Q1	Q2	Q3	Q4	Q5	Q6
10	15	10	20	10	10

Q7	Q8
5	20

Total Score
100

Problem 1 (10 points)

Consider a RISC microprocessor, like the MIPS presented in the textbook, for which we want to implement the full addressable space. Assume we have a 30GB hard disk, a 512MB main memory, a 1MB L2 Cache and a 256KB internal Cache.

Explain how programmers are given the possibility of writing programs that assume an implementation of the full addressable space. (Hint: explain the function of the management of the memory hierarchy).

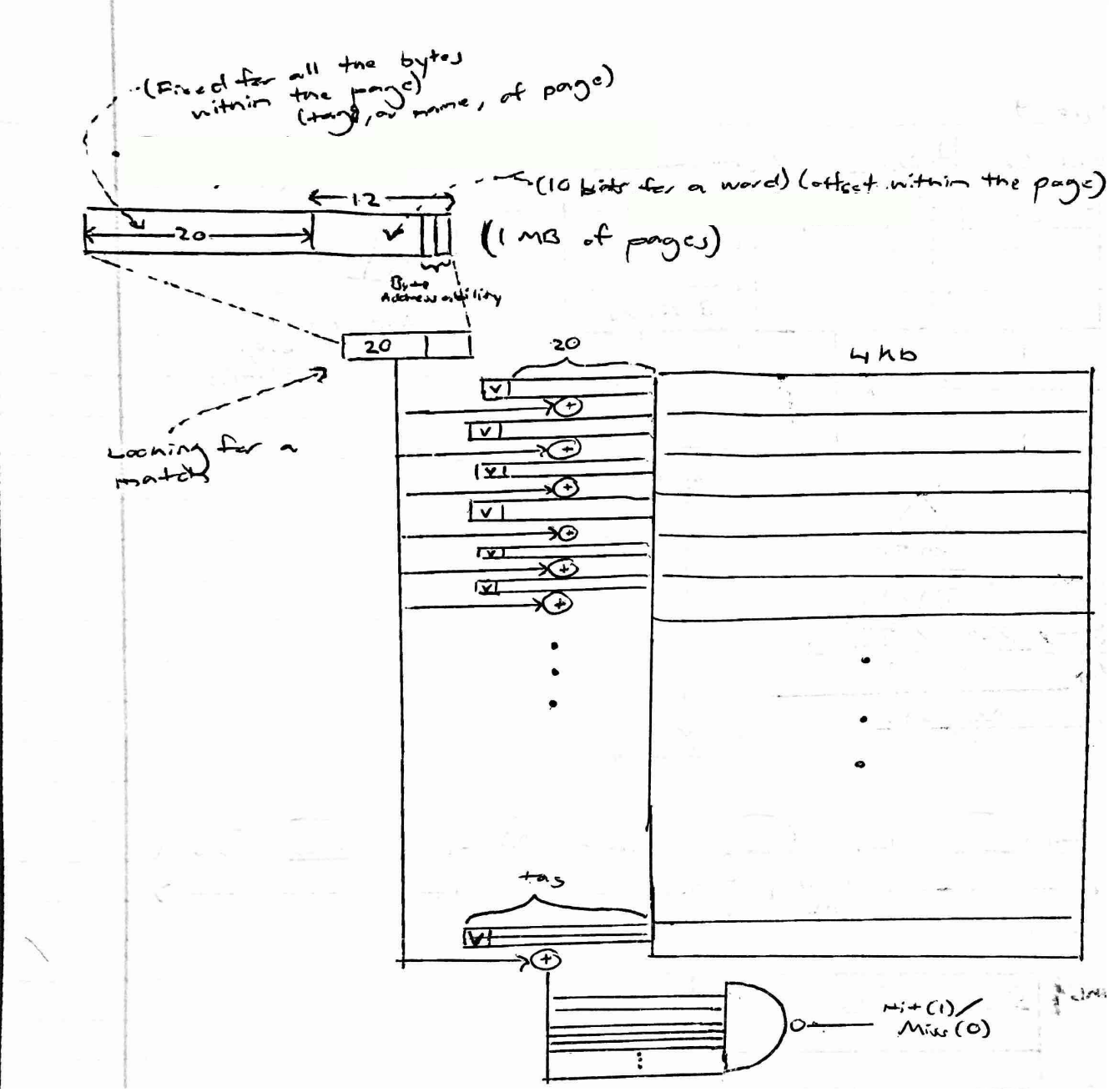
Programmers are given the possibility of writing programs that assume an implementation of the full addressable space because the CPU doesn't need access to the entire addressable space to be able to execute programs. It is enough for the CPU to be able to access parts of the program—that is, the addressable space is divided into chunks—and the CPU is provided chunks of memory (otherwise known as blocks, pages, or lines) when it needs that memory. We will see if the block is present in the L1 cache, then the CPU may continue to run in the case that it is in the L1 cache. If it isn't present in the L1 cache, then the CPU will be stopped, and the block will be brought to the L1 cache, the only place where the CPU may be able to access that block. We then let the CPU continue to run after the transfer. This is possible through the Memory Management Unit (MMU), a special purpose unit (generally a microcontroller) that looks at the addresses the CPU is issuing, and if the CPU generates an address that is not available in the proximity in the CPU, it stops the CPU and starts transferring blocks through the memory hierarchy. This is done by transferring the block(s) from the disk, to the main memory, then to the L2 cache, and finally to the L1 cache. This is to say that the L1 cache contains a proper subset of what the L2 cache contains, the L2 cache contains a proper subset of what the main memory (MM) contains, and the main memory contains a proper subset of what the disk contains (the disk being the "end-all" of all possible blocks). The transfer of information is done in "bursts" of pages at a time, those of which are consecutive memory addresses. The transfer of information is done in such a fashion whereby the information is transferred to consecutive subsets in the hierarchy until it reaches the L1 cache, where the CPU may access it (e.g. if any given sought-after block is in main memory, it will be brought to the L2 cache, and then the L1 cache, for the CPU to access the block).

Problem 2 (15 points)

Define direct mapped, fully associative and set associative caches. Explain how they relate to one another. Describe the advantages and disadvantages for each type of cache. Clearly explain your answer.

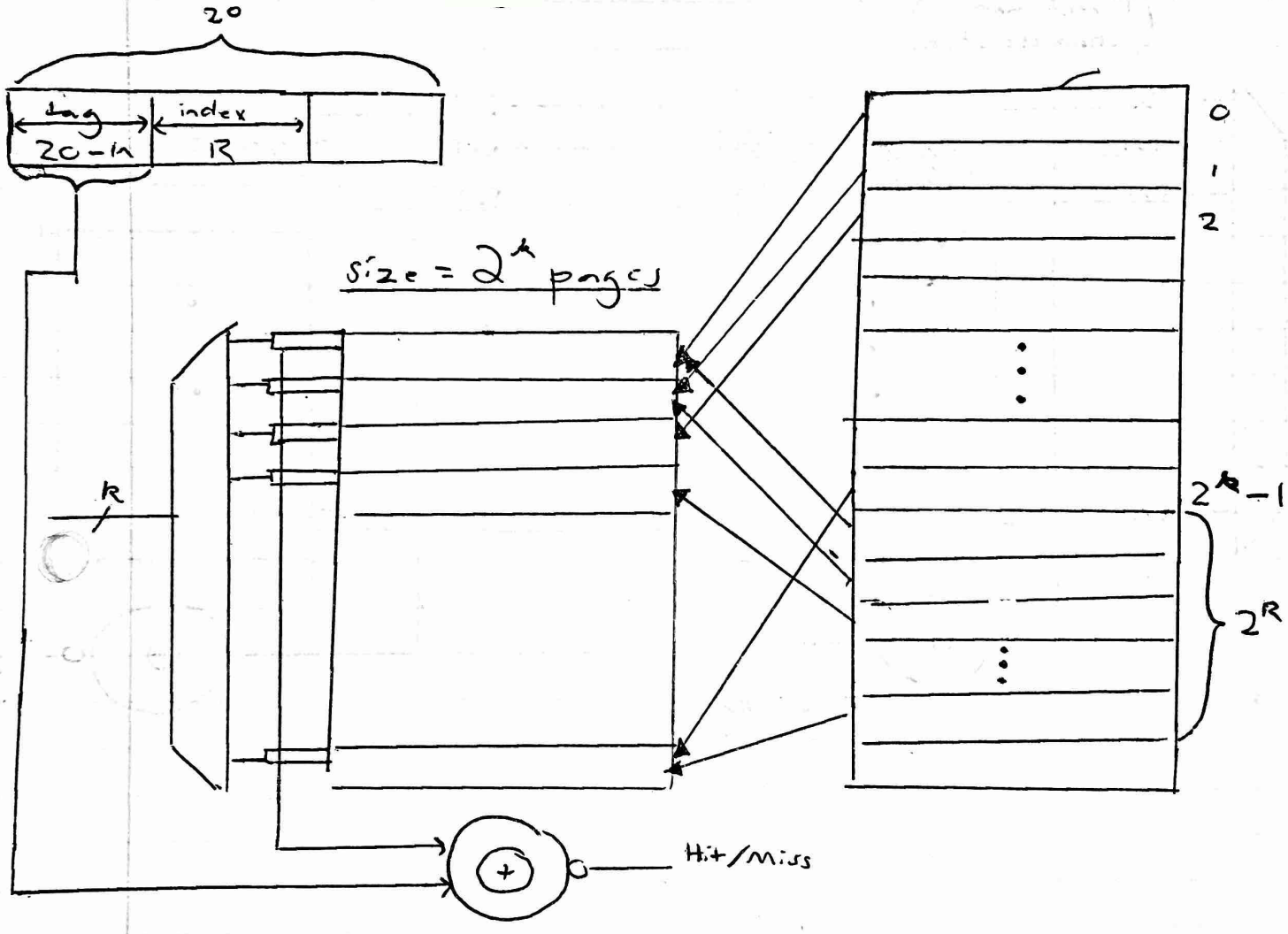
(Diagrams/explanations shown in the following pages)

Fully Associative Cache



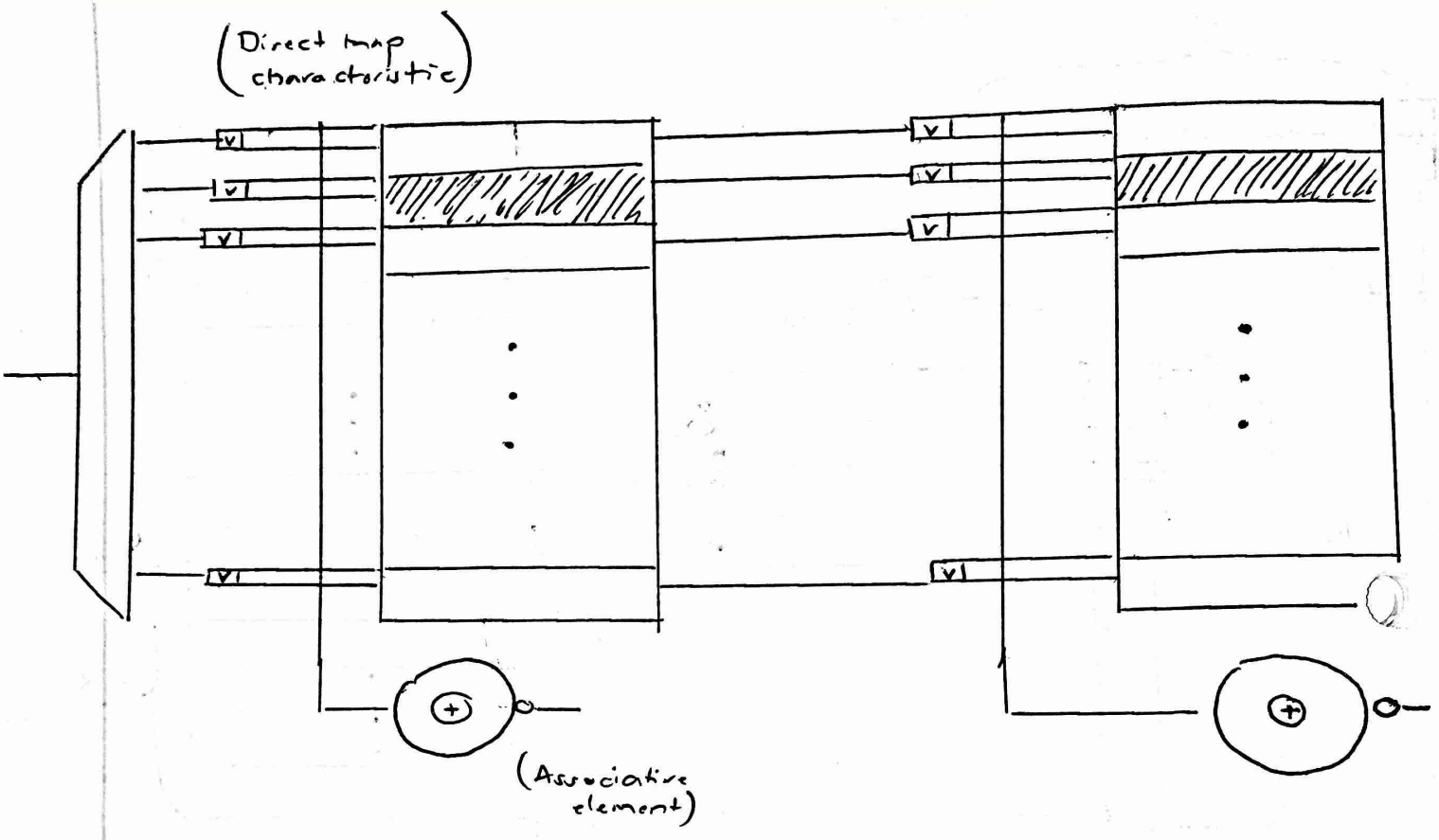
Fully Associative Caches, while expensive, give the flexibility of putting pages anywhere and finding them in a search, in hardware, that is carried out in all the pages in the L1 cache simultaneously. In one XOR delay, we would have found if there is a hit or a miss. A cache, like this one, is associative when we are looking for something within a list without having any idea of a location for an item within the list—location is disregarded (it is accessed by the content). You do the search in a location of the memory hierarchy, within the tags, regardless of which page number it is.

Direct Mapped Cache



Direct mapped caches assume a size of 2^k pages. Every page of memory will map to one page location to the cache. In the diagram above, each page (assuming we start counting at page 0) will map directly to each location until page 2^k - 1. The next page (the first location for 2^R) starts over and maps in the same fashion. This means that the page goes into the location that is the label of the page modulo 2^k (the size of the cache). The 20 bits are broken into a tag of size 20-k (the actual page), and an index of R (where in page it goes). Indexes are put through a decoder, to select one location in the cache, and says where in the cache the given page belongs. If the CPU issues a certain address, we use k bits and we know that that page will be at the issued address by comparing the rest of the bits (20-k) with the tag (we only need to compare that given one). This uses 20-k 1-bit XORs. Thus, one of the tags from the cache block and the sought-after tag are XOR-ed, which gives us a hit or a miss. The latency we pay is the price of a decoder. The drawback for direct mapped is, for a given execution, if one additional instruction reaches beyond the bound of the page which that instruction is on, the MMU detects a miss and we suffer the delay of the transfer of the size of that page multiplied by how many times that instruction is needed for subsequent executions.

(Two-Way) Set Associative Cache



Using a direct mapped memory with a cost effective implementation of the XOR, we put a copy of the same cache next to it, with exactly the same size and parameters. Suppose that we are running the program on the highlighted page shown above, and a subroutine is called down below. When the CPU issues the first address to that subroutine, the MMU will detect that the data is not in there. The valid bit determines when the space is empty or full. We have our decoder, and the decoder selects all the tags in both cache copies. They both have one XOR (in the same fashion as a direct mapped cache). We need twice the amount of XORs as the direct mapped implementation, two times 20 - k. These are direct mapped searches, but within each one of the set of pages that we have replicated, we do the search directly with XORs (in an associative manner). The name is derived from having many sets of direct mapped caches, and two-way for two copies of the cache. There must be a reason to implement more than two copies of the cache in the set associative cache implementation (the problem inherited from the direct mapped implementation), and the more caches implemented for said reason, it means that the problem arises often. The more copies implemented, the less likely the problem occurring between two pages.

Problem 3 (10 points)

Consider two processors with different cache configurations:

Cache 1: Direct-mapped with one-word blocks

Cache 2: Two-way set associative with four-word blocks

The following miss rate measurements have been made:

Cache 1: Instruction miss rate is 3%; data miss rate is 6%

Cache 2: Instruction miss rate is 2%; data miss rate is 3%

For these processors, one-half of the instructions contain a data reference. Assume that the cache miss penalty is **6 + Block size in words**. Determine which processor spends more cycles on cache misses.

For the first cache, the cache miss penalty is 6 + (2^0) = 7 clock cycle penalty. For the second cache, the cache miss penalty is 6 + (2^2) = 6 + 4 = 10 clock cycle penalty. For both processors, one-half of x instructions contain a data reference, which results in 0.5x instructions containing a data reference. We know that for all x instructions, the miss rates for caches 1 and 2 are 3% and 2% respectively, and the data miss rates for 0.5x instructions for caches 1 and 2 are 6% and 3%, respectively.

Thus, cache one sees a total penalty of
7 * (0.03(x) + 0.06(0.5x)) = 7 * 0.06x = 0.42x clock cycles for x instructions

Cache two sees a total penalty of
10 * (0.02(x) + 0.03(0.5x)) = 10 * 0.035x = 0.35x clock cycles for x instructions

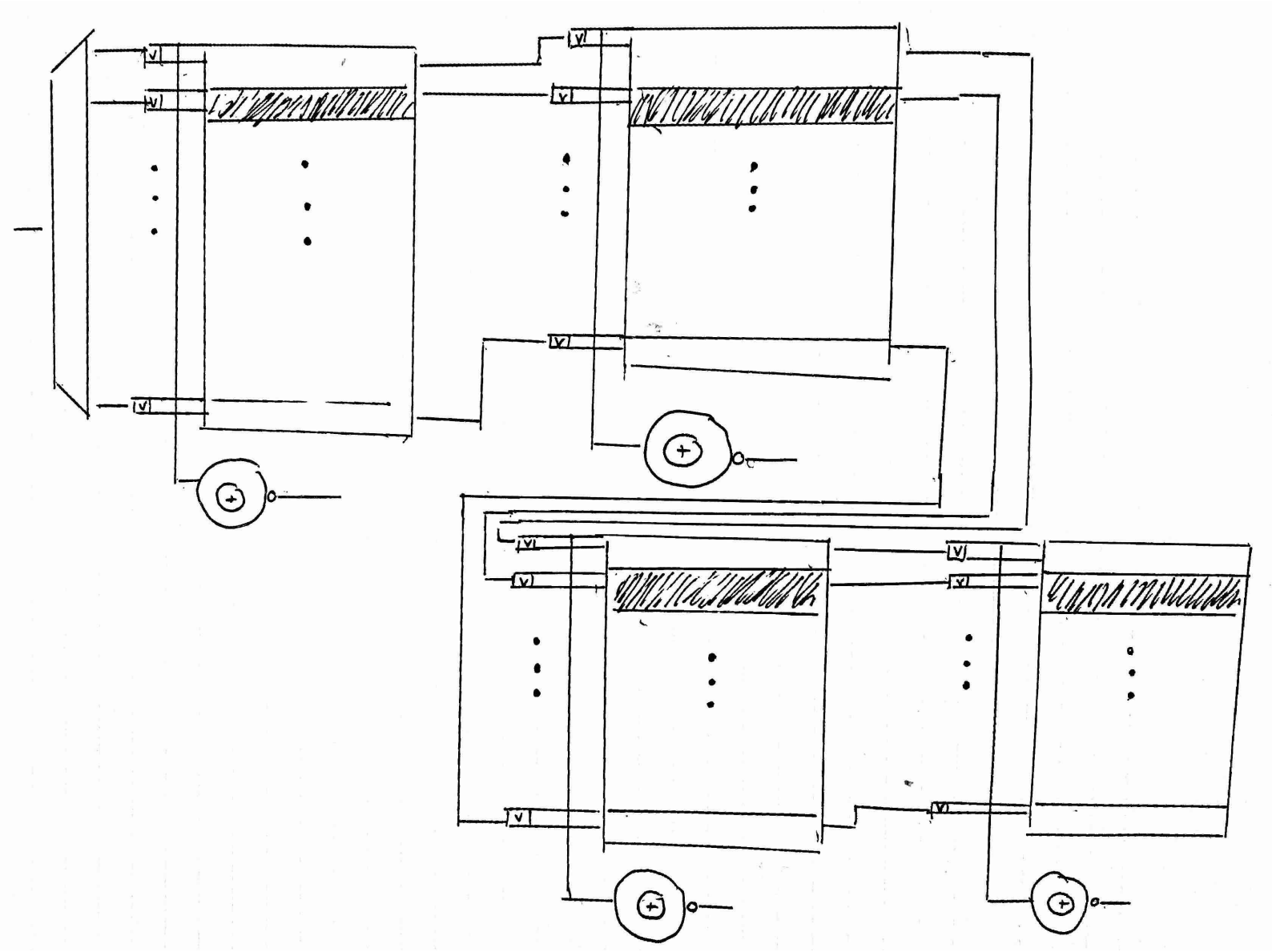
In essence, for processor 1 to processor 2, the ratio for clock cycle penalties is 0.42 : 0.35, so processor 1 spends more cycles on cache misses.

Problem 4 (20 points)

Design a 32 KB, 4-way set-associative cache that has one-word blocks and each word is 32 bits. The computer's address size is 32 bits and uses byte addressing. Assume a valid bit is used in the cache. Make sure you calculate the number of bits you will use for the index and tag. Include all necessary hardware.

Since the cache is 32KB, or 32 * 2^10 bytes (or further, 2^(5+10) bytes = 2^15 bytes) distributed amongst four sets, we can find the total number of blocks by dividing the cache size by the block size, which is given by the fact that we *only* have one word blocks that are 32 bits, so the block size is indeed 32 bits, or 4 bytes. Thus, the number of blocks is given by 2^15 bytes / 4 bytes = 2^15 bytes / 2^2 bytes = 2^13 bytes. This means that the tag is given by block size minus the number of bits needed to address all the blocks—this is 32 - 13 = 19. Because we know the tag is 19, the number of XOR gates implemented is 4 * (19 - k), where k is the number of bits needed to represent all the blocks. This is 4 * (19 - 13), or 24 XOR gates. The index, likewise, is just the number bits needed to represent all of the blocks, so the index is 13. Obviously, the valid bit is also implemented to determine if the contents of a given block is valid or not, and there are four cache copies because the specification is a four-way set associative cache. A sketch of our design is shown below.

Four-Way Set Associative Cache



Problem 5 (10 points)

The following is a sequence of address references given as word addresses.
1, 5, 8, 4, 17, 19, 20, 6, 9, 8, 43, 5, 6, 21, 9, 17
Assuming a **2-way set associative cache with 16 one-word blocks** that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache. Assume **LRU replacement** policy.

Reference	Hit or Miss	
1	Miss	**1 % 8 = 1**
5	**Miss**	**5 % 8 = 5**
8	**Miss**	**8 % 8 = 0**
4	**Miss**	**4 % 8 = 4**
17	**Miss**	**17 % 8 = 1**
19	**Miss**	**19 % 8 = 3**
20	**Miss**	**20 % 8 = 4**
6	**Miss**	**6 % 8 = 6**
9	**Miss**	**9 % 8 = 1**
8	**Hit**	**8 % 8 = 0**
43	**Miss**	**43 % 8 = 3**
5	**Hit**	**5 % 8 = 5**
6	**Hit**	**6 % 8 = 6**
21	**Miss**	**21 % 8 = 5**
9	**Hit**	**9 % 8 = 1**
17	**Hit**	**17 % 8 = 1**

Set #	Address
0	**8**
1	**1, 9**
	17
2	
3	**19**
	43
4	**20**
	4
5	**5**
	21
6	**6**
7	

Problem 6 (10 points)

Assuming a **direct-mapped cache with 4 four-word blocks** that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache.

Reference	Hit or Miss	
1	Miss	**1 % 16 = 1**
5	**Miss**	**5 % 16 = 5**
8	**Miss**	**8 % 16 = 8**
4	**Hit**	**4 % 16 = 4**
17	**Miss**	**17 % 16 = 1**
19	**Hit**	**19 % 16 = 3**
20	**Miss**	**20 % 16 = 4**
6	**Miss**	**6 % 16 = 6**
9	**Hit**	**9 % 16 = 9**
8	**Hit**	**8 % 16 = 8**
43	**Miss**	**43 % 16 = 11**
5	**Hit**	**5 % 16 = 5**
6	**Hit**	**6 % 16 = 6**
21	**Miss**	**21 % 16 = 5**
9	**Miss**	**9 % 16 = 9**
17	**Hit**	**17 % 16 = 1**

Block #	Word0	Word1	Word2	Word3
0	**~~0~~, 16**	**~~1~~, 17**	**~~2~~, 18**	**~~3~~, 19**
1	**~~4, 20, 4~~, 20**	**~~5, 21, 5~~, 21**	**~~6, 22, 6~~, 22**	**~~7, 23, 7~~, 23**
2	**~~8, 40~~, 8**	**~~9, 41~~, 9**	**~~10, 42~~, 10**	**~~11, 43~~, 11**
3	**Empty**	**Empty**	**Empty**	**Empty**

Problem 7 (5 points)

Multiple Choice Questions:

(a) What is the reason for using a TLB?

1. To reduce page fault time.
2. To reduce the clock cycle time.
3. To reduce address translation time.
4. To speed up cache access
5. Both 1 and 2

(b) The speed of the memory system affects the designer's decision on the size of the cache block. Which of the following cache designer guidelines are generally valid? **Choose two.**

1. The higher the memory bandwidth, the larger the cache block.
2. The higher the memory bandwidth, the smaller the cache block.
3. The shorter the memory latency, the larger the cache block.
4. The shorter the memory latency, the smaller the cache block.

(c) Which of the following combinations of events in the TLB, virtual memory system, and cache is **impossible**?

1. TLB hit – page table miss – cache hit
2. TLB hit – page table hit – cache hit
3. TLB miss – page table hit – cache hit
4. TLB miss – page table hit – cache miss
5. TLB miss – page table miss – cache miss

(a) The reason for using a TLB is to reduce address translation time because the page table is a directory of CPU addresses/page numbers mapped with an associated pointer to its physical memory address. The point of the TLB is that when there are pages that are constantly being requested, the page table can cache its translated address to the TLB. Thus, because the translated addresses stored in the TLB are the most frequently requested, the address does not need to be constantly translated to a physical memory address in the page table—it is already buffered for us.

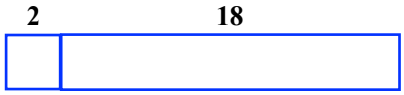
(b) Bandwidth implies the number of data elements that can be transferred per some period of time, and therefore the cache would be smaller because data can be transferred faster. In other words, if there is more bandwidth, we can afford to implement a small cache because if we need to get data from any other superset of the memory hierarchy, the transfer would be of greater size/volume, thus making the transfer "faster", in a sense. If the cache was bigger, the lookup time would increase linearly, and as a consequence the bandwidth would decrease because the rate at which data is transferred would decrease over said period of time. Latency implies the time it takes to get to a transfer. Thus, shorter latency times means that the cache is smaller. A large cache does not need to be implemented into a computer's memory system if we can make up for less cached data being closer to the CPU. We make up for this by being able to transfer data between memory units in the hierarchy, faster. Thus, we do not have to pay the price of a transfer being slow.

(c) Since the page table is a map of physical pointers to memory addresses in main memory, the cache is a subset of the page table, and the TLB is a subset of the page table. With this being said, it must be impossible for there to be a TLB hit while there is a miss in the page table, since the TLB is a subject of the page table. Also, it is impossible for there to be a cache hit and a page table miss—since the page table maps main memory directly, this is a violation of the memory hierarchy since the cache is a subset of main memory.

Problem 8 (20 points)

Consider 256Kx8bits dynamic RAM chips where the access time is 1/2 (0.5) of the cycle time.
Suggest a memory organization that will contain 4 megabytes, will have a 32-bit data bus and that will yield one word (32-bits) every access time if words are read from consecutive memory locations. Please clearly explain your answer.

We can implement a memory system, a 4-way interleaved memory system. The chip will have three inputs—the address, a read/write signal (determined at rising edges of the clock), and the chip select, which dictates what data is electronically defined/undefined. These are the building blocks to build dynamic RAMs. For this example, 4MB divided by 256KB is 16 chips that will be implemented in our memory system. Since the data width is 8 bits, we will need 4 chips to build a 32 bit bus. Each chip has 8 bits coming in and out. In order to read one word at a time, all the address lines need to be connected together. The read/write signal and the chip select are the same, such that as a row, we have 256 kilo-words. Every time we read the row, we select all the chip selects, give an address to all of them (they all read the same location), and we give them a read or write such that the chips will contribute or take in 8 bits out of the 32 bits for the row. That results in 1 megabyte in any given row, so we need to have four rows. We need [log2(256) + 10] address bits for 256KB for each one of the chips—this implies that we need 18 address bits to address each one of the bytes (since kilobytes implies a 2^10 multiple, hence the 10 bits). Thus, the address lines will be 18 bits long. This gives us 256 kilowords. The other four words receive the same control lines and address lines. If we read one word out of the first 256KB, we give an 18-bit address and we activate the chip selector to select the row. We need 2 bits to select one out of the four rows (since there are 2^2 possible rows). We map the chips in the following manner to read consecutive memory addresses:



In regards to yielding one word every access time, there will be time to spare because we read the first row in one unit of time, since the access time accounts for half of the cycle time. For each subsequent row, we need one unit of time to read as well, for a total of four units of time between the four rows. Though, the previous two rows wrote back in one unit of time each, so we need two units of time for the entire memory access time for one row, which made us a unit of time over. Though, this is okay, because it wrote back. In other words, if a read takes one unit of time for one row (since the ratio is 1:1 for reads to write backs since the write back accounts for the other 1/2 of the cycle time), we will read in one unit of time for subsequent rows.

4-way Interleaved Memory System