Consider this 2-way set associative cache to be used with our beta processor.

Each line holds a single 32 bit data word together with a valid bit and a tag.

Our beta uses 32 bit addresses.

We want to determine which address bits should be used for the cache index and which should be used for the tag so as to ensure best cache performance.

The bottom two bits of the address are always assumed to be 00 for word alignment since are addresses are in bytes but our data is in words of 32 bits or 4 bytes.

Our cache has 8 lines in it, so this means that our index must be 3 bits wide.

The bits that we want to use for the index are the next least significant bits which are address bits [4:2].

The reason that we want to make these bits be part of the index instead of the tag is because of locality.

The idea is that instructions or data that are near each other in memory are more likely to be accessed around the same time than instructions or data that reside in a different part of memory.

So if for example, our instruction comes from address 0x1000, it is fairly likely that we will also access the next instruction which is at address 0x1004.

With this scheme the index of the first instruction would map to line 0 of the cache while the next instruction would map to line 1 of the cache so they would not cause a collision or a miss in the cache.

This leaves the higher order bits for the tag.

We need to use all of the remaining bits for the tag in order to be able to uniquely identify each distinct address.

Since many addresses will map to the same line in the cache, we must compare the tag of the data in the cache to see if we have in fact found the data that we are looking for.

So we use address bits [31:5] for the tag.

Suppose our beta executes a read of address 0x5678.

We would like to identify which locations in the cache will need to be checked to determine if our data is already present in the cache or not.

In order to determine this, we need to identify the portion of the address that corresponds to the index.

The index bits are bits [4:2] which correspond to 110 in binary for this address.

That means that this address would map to cache line 6 in our cache.

Since this is a 2-way set associative cache, there are two possible locations that our data could be located, either in 6A or 6B.

So we would need to compare both tags to determine whether or not the data we are trying to read is already in the cache.

Assuming that checking the cache on a read takes 1 cycle, and that refilling the cache on a miss takes an additional 8 cycles, this means that the time it takes on a miss is 9 cycles, 1 to first check if the value is in the cache, plus another 8 to bring the value into the cache if it wasn't already there.

Now suppose that we want to achieve an average read access time of 1.1 cycles.

What is the minimum hit ratio required to achieve this average access time over many reads?

We know that average access time = (hit time * hit rate) + (miss time * miss rate).

If we call 'a' our hit rate, then our miss rate is (1-a).

So our desired average access time of 1.1 must equal 1 * a plus 9 * (1-a).

This reduces to 1.1 = 9-8a, which means that 8a = 7.9 or a = 7.9/8.

So to achieve a 1.1 cycle average access time our hit rate must be at least 7.9/8.

We are provided with this benchmark program for testing our 2-way set associative cache.

The cache is initially empty before execution begins.

In other words, all the valid bits of the cache are 0.

Assuming that an LRU, or least recently used, replacement strategy is used, we would like to determine the approximate cache hit ratio for this program.

Let's begin by understanding what this benchmark does.

The program begins at address 0.

It first performs some initialization of registers using three CMOVE operations.

The first, initializes R0 to hold source which is the address in memory where our data will be stored.

The second initializes R1 to 0, and the third initializes R2 to 0x1000 which is the number of words that this benchmark will work with.

We then enter the loop which is shown in the yellow rectangle.

The loop loads the first element of our data from location source + 0 into register R3.

It then increments R0 to point to the next piece of data.

Since our data is 32 bits wide, this requires the addition of the constant 4 representing the number of bytes between consecutive data words.

It then takes the value that was just loaded and adds it to R1 which holds a running sum of all the data seen so far.

R2 is then decremented by 1 to indicate that we have one fewer piece of data to handle.

Finally, as long as R2 is not equal to 0 it repeats the loop.

At the very end of the benchmark the final sum is stored at address source, and the program halts.

When trying to determine the approximate hit ratio, the instructions that occur only once because they live outside of the loop can basically be ignored.

So looking only at what happens over and over again in the loop, each time through the loop, we have 5 instruction fetches and one data fetch.

The first time through the loop, we miss on the instruction fetches and then bring them into the cache.

We also miss on the data load from address 0x100.

When this data is brought into the cache, instead of replacing the recently loaded instructions, it loads the data into the 2nd set of the cache.

The loop is then repeated.

This time through the loop, all the instruction fetches result in hits.

However, the data that we now need is a new piece of data so that will result in a cache miss and once again load the new data word into the 2nd set of the cache.

This behavior then repeats itself every time through the loop.

Since the loop is executed many times, we can also ignore the initial instruction fetches on the first iteration of the loop.

So in steady state, we get 5 instruction cache hits, and 1 data cache miss every time through the loop.

This means that our approximate hit ratio is 5/6.

The last question we want to consider is what is stored in the cache after execution of this benchmark is completed.

As we saw earlier, because we have a 2-way set associative cache, the instructions and data don't conflict with each other because they can each go in a separate set.

We want to determine which instruction and which piece of data will end up in line 4 of the cache after execution is completed.

We'll begin by identifying the mapping of instructions to cache lines.

Since our program begins at address 0, the first CMOVE instruction is at address 0, and it's index is equal to 0b000, or 0 in binary.

This means that it will map to cache line 0.

Since at this point, nothing is in the cache, it will be loaded into line 0 of set A.

In a similar manner the next 2 CMOVE instructions and the LD instruction will be loaded into lines 1-3 of set A. At this point, we begin loading data.

Since the cache is 2-way set associative, the data will be loaded into set B instead of removing the instructions that were loaded into set A.

The instructions that are outside the loop will end up getting taken out of set A in favor of loading a data item into those cache locations, but the instructions that make up the loop will not be displaced because every time something maps to cache lines 3-7, the least recently used location will correspond to a data value not to the instructions which are used over and over again.

This means that at the end of execution of the benchmark, the instruction that will be in line 4 of the cache is the ADDC instruction from address 0x10 of the program.

The loop instructions which will remain in the cache are shown here.

Now let's consider what happens to the data used in this benchmark.

We expect the loop instructions to remain in lines 3-7 of set A of the cache.

The data will use all of set B plus locations 0-2 of set A as needed.

The data begins at address 0x100.

The index portion of address 0x100 is 0b000 so this data element maps to cache line 0.

Since the least recently used set for line 0 is set B, it will go into set B leaving the instructions in tact in set A. The next data element is at address 0x104.

Since the bottom two bits are used for word alignment, the index portion of this address is 0b001, so this data element maps to line 1 of set B, and so on through element 0x7.

Data element 0x8 is at address 0x120.

The index portion of this address is once again 0b000.

So this element maps to cache line 0 just like element 0x0 did.

However, at this point line 0 of set B was accessed more recently than line 0 of set A, so the CMOVE instruction which is executed only once will get replaced by a data element that maps to line 0.

As we mentioned earlier, all the instructions in the loop will not be displaced because they are accessed over and over again so they never end up being the least recently used item in a cache line.

After executing the loop 16 times meaning that data elements 0 through 0xF have been accessed, the cache will look like this.

Note that the loop instructions continue in their original location in the cache.

The state of the cache continues to look like this with more recently accessed data elements replacing the earlier data elements.

Since there are 0x1000 elements of data, this continues until just before address 0x4100.

The last element is actually located 1 word before that at address 0x40FC.

The last 8 elements of the data and their mapping to cache lines is shown here.

The data element that ends up in line 4 of the cache, when the benchmark is done executing, is element 0x0FFC of the data which comes from address 0x40F0.