University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 1998
Solution Set #2

J.D. Kubiatowicz

Solution Set #2
CS252 Graduate Computer Architecture

$\boxed{5.1}$ **(8 points)**

This problem explores unfair memory-system benchmarks. We have two cache configurations to compare: cache A is 2-way set associative, 128 sets, 32-byte blocks, write-through, no-write allocate; cache B is 1-way associative, 256 sets, 32-byte blocks, write-back, write-allocate. A word on the terminology in this problem: we are told that the miss time is 10 times the hit time. Miss time here is the total time spent processing a miss, not just the miss penalty; you can think of it as miss time = miss penalty + hit time.

(a) We want a program that makes A as much faster than B as possible. The basic idea is to write a program that generates a large amount of conflict misses in the direct-mapped cache B, but generates no misses in cache A. One way you might think about doing this is writing a program that repeatedly reads from two alternating addresses α and β such that $\alpha \bmod (256 \times 32) = \beta \bmod (256 \times 32)$ (so that α and β map to the same line in B, and to different parts of the same set in A). There is a subtlety, however. Both cache A and B are unified caches, so for every memory read, there must also be an instruction fetch from the cache. If there are no conflicts in the instruction fetches, the instruction fetch times will be the same for both caches, and we will not see as much of a performance difference between A and B than if there are instruction conflicts. One way to guarantee maximum performance difference between A and B is to only use instruction conflicts. Consider the following code:

```
a:   j  b   ; absolute jump to b
      .
      .
      .
b:   j  a   ; absolute jump to a
```

where `a` and `b` refer to addresses that collide in the direct-mapped cache but not in the 2-way cache (for example, `a` mod 8192 = `b` mod 8192). For cache A, the instruction fetches map to different entries in a set, so each instruction fetch is a hit. For cache B, the fetches map to the same set, so they cause conflict misses on each instruction fetch.

(b) To make B run faster than A, we need to take advantage of its write-back property. We basically want a program that writes repeatedly to one cache line (that has never been read before and is thus not initially in the cache). Cache B can absorb all those writes as cache hits, whereas cache A is forced to propagate them through to memory. Again there is a complication due to instruction accesses, but this time the most we can do is assume that instruction fetches don't conflict in the cache (since we can't actually perform writes via instruction fetches). Thus, a simple program that does what we want is:

```
l1:   sw  0(r1), 0
      sw  0(r1), 0
      .
      .              ; we assume the loop is unrolled n times
      .
      b   l1
```

We assume that the loop can be unrolled n times without causing code conflicts in the cache; n must be less than 2039 (the number of 4-byte instructions that fit in the cache minus a slot for the branch and an entire block (32 bytes) for the data being stored).

(c) The program is 10 times faster on cache A than on cache B, if we assume single-cycle jumps. The analysis is simple: the only cache accesses are for the instruction fetches; all accesses in cache A hit; all accesses in cache B miss; miss time is 10x hit time. Thus, machine A runs 10x faster than B, assuming that the machines have the same clock rate.

(d) The analysis is more complicated than in part (c), since we must take into account data accesses and instruction fetches. We'll assume the loop is unrolled $n \leq 2039$ times, and the value of `r1` is carefully chosen so that no conflict misses occur during execution of the loop. We'll also assume a DLX-like in-order pipeline, but with single-cycle branches (e.g., perfect branch target prediction) and a unified, single-ported instruction/data cache.

Each store instruction on cache B takes two cache cycles (and thus two real cycles since the cache is single-ported): one to fetch the instruction and one to write the data. Let t_{hit} be the time for a cache hit. There are n of the store instructions, each taking $2t_{\text{hit}}$ cycles, and one branch instruction, taking t_{hit} cycles. So the total time for n iterations on cache B is simply $t_{\text{hit}}(2n+1)$.

The analysis for cache A is similar, except that each store instruction requires $6t_{\text{hit}}$ cycles, 5 for the write and one for the instruction fetch (we're assuming the pipeline stalls until the write completes (no write buffer)). So the total time for n iterations on cache A is $t_{\text{hit}}(6n+1)$.

The performance ratio of cache B to cache A is:

$$\frac{\text{Time}_{\text{A}}}{\text{Time}_{\text{B}}} = \frac{t_{\text{hit}}(6n+1)}{t_{\text{hit}}(2n+1)} = \frac{6n+1}{2n+1}.$$

If we unroll $n = 2039$ times, then the ratio is 2.9995, and so cache B is about 3 times faster than cache A.

$\boxed{5.4}$ (8 points)

We want to calculate the utilization of memory bus bandwidth for two different cache configurations. The basic approach in each case is to list all of the different types of memory reference, and, for each, to compute how many bus transactions it takes and how frequently it occurs.

(a) We have a write-through cache. There are four types of memory reference to consider: read hit (in the cache), read miss, write hit, and write miss. All blocks in the cache are always clean, so it is not necessary to distinguish between dirty and clean blocks. Read hits are absorbed by the cache and generate no memory system traffic; these comprise a $0.95 \times 0.75 = 0.7125$ fraction of the total references (since there's a 95% hit rate, and 75% of accesses are reads. Read misses require two accesses, since both blocks of the cache line must be loaded; these comprise a $0.05 \times 0.75 = 0.0375$ fraction of the total time. Write hits require a single memory reference to write the new word back to memory (only the modified word is written, not the entire block), and comprise $0.95 \times 0.25 = 0.2375$ of the references. Finally, since the cache is write-allocate, a write miss first fetches the entire old block from memory (two references), then modifies one word of the block and writes the modified data through to memory (one reference). So these take three memory references, and occur with a frequency of $0.05 \times 0.25 = 0.0125$. These results are summarized in the table below.

Access type	# Memory Refs	Frequency
Read hit	0	$0.95 \times 0.75 = 0.7125$
Read miss	2	$0.05 \times 0.75 = 0.0375$
Write hit	1	$0.95 \times 0.25 = 0.2375$
Write miss	3	$0.05 \times 0.25 = 0.0125$

We can now compute the bandwidth used. Using the data above, we first compute the average number of main memory references per processor access to be:

$$0 \times 0.7125 + 2 \times 0.0375 + 1 \times 0.2375 + 3 \times 0.0125 = 0.35$$

Accesses are generated by the processor at a rate of 10^9 words per second, and on average 35% of these will reach the memory system. The memory system can sustain 10^9 words per second, so the utilization of the memory system is simply:

$$\frac{0.35 \times 10^9}{10^9} = 0.35$$

or 35%.

(b) Now we have a write-back cache. Our breakdown of memory references is more complicated, since we must consider the cases when a miss is to a block in the cache that is dirty. We are told that 30% of the blocks are dirty on average. If we assume random replacement, then on average 30% of misses will force a writeback of a dirty line. The following table shows the breakdown of processor accesses, the number of references generated by each, and the frequency of each:

Access type	# Memory Refs	Frequency
Read hit	0	$0.95 \times 0.75 = 0.7125$
Read miss to clean block	2	$0.05 \times 0.75 \times 0.70 = 0.02625$
Read miss to dirty block	4	$0.05 \times 0.75 \times 0.30 = 0.01125$
Write hit	0	$0.95 \times 0.25 = 0.2375$
Write miss to clean block	2	$0.05 \times 0.25 \times 0.70 = 0.00875$
Write miss to dirty block	4	$0.05 \times 0.25 \times 0.30 = 0.00375$

Both read and write hits are absorbed by the cache, so they generate no memory system references. Read and write misses to clean blocks each generate two references to read in the old block from the memory system (a write miss behaves like a read miss plus a write hit). Finally, read and write misses to dirty blocks each generate four references: two to write back the dirty block, and two to read in the new block from memory.

We can again compute the average number of main memory references per processor access to be:

$$\begin{aligned} & 0 \times 0.7125 + 2 \times 0.02625 + 4 \times 0.01125 + 0 \times 0.2375 + 2 \times 0.00875 + 4 \times 0.00375 \\ = \; & 0.13 \end{aligned}$$

Again, accesses are generated by the processor at a rate of 10^9 per second, and the memory system bandwidth is 10^9 accesses per second, so the utilization is simply:

$$\frac{0.13 \times 10^9}{10^9} = 0.13$$

or 13%. Notice that a write-back cache significantly reduces the memory bandwidth required as compared to a write-through cache. This is often why processors are built with write-back L2 caches, especially when they are intended for use on a shared-memory bus where bandwidth is a precious resource.

$\boxed{5.6}$ **(10 points)**

We want to try to remove all extraneous prefetches, and also eliminate all non-prefetched data accesses. We start with some assumptions:

- the computational instruction (`a[i][j] = b[j][0] * b[j+1][0]`) takes 7 cycles to execute

- the loop overhead is 0
- the memory system is fully pipelined and can handle as many outstanding requests as we need
- prefetches take 1 cycle to execute
- the cache miss penalty is 50 cycles
- a prefetch to one word of a cache block also prefetches all other words in that block

One simple way to answer this problem is to just prefetch all data in advance. The code for this is shown below; note that we unroll the first loop in order to only prefetch one word of each cache block containing data from the `a[]` array, but that we can't do this for the `b[]` array since it is accessing noncontiguous data.

```
/* prefetch b array */
for (j = 0; j < 100; j = j + 1)
    prefetch(b[j][0]);
/* prefetch a array */
for (i = 0; i < 3; i = i + 1)
    /* prefetch first element in case array is not cache-block aligned */
    prefetch(a[i][0]);
    /* prefetch remaining elements */
    for (j = 1; j < 100; j = j + 2) {
        prefetch(a[i][j]);
    }
/* do computation */
for (i = 0; i < 3; i = i + 1)
    for (j = 0; j < 100; j = j + 1)
        a[i][j] = b[j][0] * b[j+1][0];
```

This is not a good general solution, however, since it requires that the cache be big enough to hold the entire `a[]` and `b[]` arrays. Even if the arrays fit into the cache, this scheme significantly increases the cache working set (amount of data that must stay in the cache to be useful) over what it could be.

A better approach is to prefetch the data as the loops walk over the array. In order to do this, we have to split the loops into several parts. We begin with an initial loop that prefetches the first eight elements needed, `a[0][0...7]` and `b[0...7][0]`:

```
/* SOLUTION PART 1 */
prefetch(a[0][0]);
for (j = 0; j < 8; j += 2) {
    prefetch(b[j][0]);
    prefetch(b[j+1][0]
    prefetch(a[0][j+1]);
}
```

A few notes about this loop: first, we begin by prefetching `a[0][0]` manually so that the loop will fetch the odd elements of `a`; the loop is then unrolled to only prefetch the odd elements of `a`. If we knew the array was aligned with a cache block, this wouldn't be necessary. However, if the array is not aligned, this guarantees that we actually prefetch all needed elements. Note that we unroll the loop rather than using mod (`%`) to prefetch every other element of `a`; this is because the mod operation can be very expensive compared to unrolling the loop. Also, we put the prefetches in a loop to reduce the code footprint of the program; this would be particularly important if the cache was unified. Next, notice that the prefetches are scheduled to prefetch the `b` array first. This is because the values of the `b` array are needed first in the computational instruction. One might wonder why we bother prefetching the first few iterations worth of data, rather than just allowing them to cache-miss. The key is that the prefetches are non-blocking, and, since we assumed that the memory system is pipelined with a large number of outstanding requests, the prefetches can all run in parallel. So, when the first computational instruction tries to access b[0][0], the prefetch will not have completed (since only about 12 cycles have elapsed) and the load will block until the data is ready. However, all the other prefetches will continue (separated in time by just one cycle), so, by the time the first load is ready and its value consumed, the remaining data will be ready, and thus there will be no cache misses on any of the initial computation iterations.

Now we consider the main part of the loop. It must do the computation and also prefetch for subsequent iterations. Again, we unroll the loop to avoid prefetching twice for each cache block. Finally, note that the loop only runs until `j == 90`; this is to avoid extra unnecessary prefetches.

```
/* SOLUTION PART 2 */
for (j = 0; j < 92; j += 2) {
    prefetch(b[j+8][0]);
    prefetch(a[0][j+8];
    a[0][j] = b[j][0] * b[j+1][0];
    prefetch(b[j+9][0]);
    a[0][j+1] = b[j+1][0] * b[j+2][0];
}
prefetch(b[100][0]);
```

Finally, we have a last loop that finishes the computation without prefetching past the bounds of the array. Note, however, that up to this point we've only been working on the `i == 0` iteration of the *outer* loop. Since there are two more outer iterations to do, we begin prefetching for the next outer iteration in this last part of the `i == 0` loop:

```
/* SOLUTION PART 3 */
prefetch(a[1][0]);
for (j = 92; j < 100; j += 2) {
    prefetch(a[1][j-91]);
    a[0][j] = b[j][0] * b[j+1][0];
    a[0][j+1] = b[j+1][0] * b[j+2][0];
}
```

Note that we don't need to prefetch the `b` array, since it is already in the cache and is used unchanged in the next iteration.

OK, now we've got the first outer loop iteration done. The next iteration is similar to the previous one; it begins with a main loop that prefetches as it computes, and ends with a loop that prefetches for the last (`i == 2`) iteration:

```
/* SOLUTION PART 4 */
for (j = 0; j < 92; j += 2) {
    prefetch(a[1][j+8];
    a[1][j] = b[j][0] * b[j+1][0];
    a[1][j+1] = b[j+1][0] * b[j+2][0];
}
prefetch(a[2][0]);
for (j = 92; j < 100; j += 2) {
    prefetch(a[2][j-91]);
    a[1][j] = b[j][0] * b[j+1][0];
    a[1][j+1] = b[j+1][0] * b[j+2][0];
}
```

And to complete the problem, we do the last outer iteration, prefetching only in the main loop. We don't need to prefetch in the last part of the loop since there are no more iterations to do.

```
/* SOLUTION PART 5 */
for (j = 0; j < 92; j += 2) {
    prefetch(a[2][j+8]);
    a[2][j] = b[j][0] * b[j+1][0];
```

```
        a[2][j+1] = b[j+1][0] * b[j+2][0];
    }
    for (j = 92; j < 100; j += 1) {
        a[2][j] = b[j][0] * b[j+1][0];
    }
```

Lastly, let's compute the performance of this solution. The first code fragment runs in 13 cycles. The second code fragment would take 783 cycles to execute, but it has to stall in cycle 16 on the first access to b[0][0]; this data was prefetched in cycle 2 and is thus available in cycle 52, causing 36 stall cycles. There are no further stalls in the code, so we can continue adding up cycles. The third fragment takes 61 cycles; the fourth fragment takes 751, and the fifth fragment takes 746. Adding these together, we get 2390 cycles in total, or 44% faster than the simple prefetch code on page 404, and 6.13× faster than the non-prefetched code.

One other note on this problem. The original (unprefetched) loop is rather poorly written. It can be simplified significantly by performing a loop interchange optimization. Performing this optimization also simplifies the prefetch problem. Here's the code after loop interchange:

```
    /* after loop interchange, no prefetch */
    for (j = 0; j < 100; j++) {
        t1 = b[j][0] * b[j+1][0];
        a[2][j] = a[1][j] = a[0][j] = t1;
    }
```

Since we've eliminated the outer loop, the prefetched version is simpler as well, and has a much smaller cache working set (since the B array doesn't have to hang around in the cache):

```
    /* Solution with loop interchange */
    prefetch(a[0][0]);
    prefetch(a[1][0]);
    prefetch(a[2][0]);
    for (j = 0; j < 8; j += 2) {
        prefetch(b[j][0]);
        prefetch(b[j+1][0]
        prefetch(a[0][j+1]);
        prefetch(a[1][j+1]);
        prefetch(a[2][j+1]);
    }
    for (j = 0; j < 92; j += 2) {
        prefetch(b[j+8][0]);
        prefetch(a[0][j+8];
        prefetch(a[1][j+8];
        prefetch(a[2][j+8];
        t1 = b[j][0] * b[j+1][0];
        a[2][j] = a[1][j] = a[0][j] = t1;
        prefetch(b[j+9][0]);
        t1 = b[j+1][0] * b[j+2][0];
        a[2][j+1] = a[1][j+1] = a[0][j+1] = t1;
    }
    for (j = 92; j < 100; j += 2) {
        t1 = b[j][0] * b[j+1][0];
        a[2][j] = a[1][j] = a[0][j] = t1;
        t1 = b[j+1][0] * b[j+2][0];
        a[2][j+1] = a[1][j+1] = a[0][j+1] = t1;
    }
```

$\boxed{5.7}$ **(8 points)**

In this problem, we consider a pseudo-associative cache in which we do not swap blocks on a hit to the "slow" part of the cache. We also assume that a hit to the slow part takes only one extra cycle, not two (since we don't have to swap). Finally, we are assuming a random replacement policy.

(a) We first want to derive the AMAT for this cache in terms of miss rates and miss penalties of 1-way (direct-mapped) and 2-way caches. We use a slightly different formulation of AMAT than the usual one:

$$\text{AMAT}_{\text{pseudo}} = \text{Hit rate}_{\text{pseudo}} \times \text{Hit time}_{\text{pseudo}} + \text{Miss rate}_{\text{pseudo}} \times \text{Miss time}_{\text{pseudo}}$$

The reason for using this formulation (rather than the traditional AMAT = Hit time + Miss rate × Miss penalty) is that the miss *time* in a pseudo-associative cache is different than the hit time plus the miss penalty. Here's why: in a traditional cache, the hit time is a constant. When there's a miss in a traditional cache, that same constant amount of time is spent checking for a hit before the miss penalty is paid. In a pseudo-associative cache, the hit time for a given hit can be one of two values, and thus the overall hit time is just a statistical measure. On a miss in a pseudo-associative cache, it is the *slow* hit time that must be paid before starting to process a miss, not the statistical average hit time.

Given that, we can say now examine the terms of the equation. The miss/hit rates are the same as a standard 2-way associative cache of the same size, since the two types of caches behave the same in terms of hits and misses (since there are two possible locations for each datum):

$$\text{Miss rate}_{\text{pseudo}} = \text{Miss rate}_{\text{2-way}} = 1 - \text{Hit rate}_{\text{pseudo}}.$$

The miss time is just the miss penalty of a 1-way cache (since the underlying memory system and fill path is the same) plus the slow hit time in the associative cache. So at this point our AMAT equation looks like:

$$\text{AMAT}_{\text{pseudo}} = (1 - \text{Miss rate}_{\text{2-way}}) \times \text{Hit time}_{\text{pseudo}} + \text{Miss rate}_{\text{2-way}} \times (\text{Hit time}_{\text{pseudo-slow}} + \text{Miss penalty}_{\text{1-way}})$$

The only remaining term in the AMAT formula is the overall hit time for the cache. We start by expressing the overall hit time in terms of the percentage of accesses and the access time to the fast and slow blocks, as follows:

$$\text{Hit time}_{\text{pseudo}} = \text{Hit fraction}_{\text{fast block}} \times \text{Hit time}_{\text{fast block}} + \text{Hit slow}_{\text{slow block}} \times \text{Hit time}_{\text{slow block}}.$$

Because we are using random replacement, and because we are not swapping blocks, the likelihood of a given block being in the fast position is the same as the it being in the slow position, so:

$$\text{Hit fraction}_{\text{fast block}} = \text{Hit fraction}_{\text{slow block}} = \frac{1}{2}.$$

So:

$$\text{Hit time}_{\text{pseudo}} = \frac{1}{2}\left(\text{Hit time}_{\text{fast block}} + \text{Hit time}_{\text{slow block}}\right).$$

Putting this together with the AMAT formula and simplifying gives:

$$\begin{aligned}\text{AMAT}_{\text{pseudo}} = {} & \frac{\text{Hit time}_{\text{pseudo-fast}} + \text{Hit time}_{\text{pseudo-slow}}}{2} + \\ & \text{Miss rate}_{\text{2-way}}\Bigg(\text{Hit time}_{\text{pseudo-slow}} + \text{Miss penalty}_{\text{1-way}} - \\ & \frac{\text{Hit time}_{\text{pseudo-fast}} + \text{Hit time}_{\text{pseudo-slow}}}{2}\Bigg)\end{aligned}$$

Now, recalling that $\text{Hit time}_{\text{fast block}} = 1$, $\text{Hit time}_{\text{slow block}} = 2$, and $\text{Miss penalty}_{\text{1-way}} = 50$, we can compute the final AMAT formula as:

$$\text{AMAT}_{\text{pseudo}} = 1.5 + \text{Miss rate}_{\text{2-way}} \times 50.5.$$

(b) We assume that the values in Figure 5.9 in the text still apply for a cache with random replacement. We assume a 50-cycle miss penalty. For the 2KB cache:

$$\begin{aligned}\text{AMAT}_{\text{pseudo-2KB}} &= 1.5 + 0.076(50.5) \\ &= 5.338.\end{aligned}$$

And for the 128KB cache:

$$\begin{aligned}\text{AMAT}_{\text{pseudo-2KB}} &= 1.5 + 0.007(50.5) \\ &= 1.854.\end{aligned}$$

$\boxed{5.10}$ **(8 points)**

In this problem we consider how to implement a memory system that uses a prime number of memory banks.

(a) This problem effectively asks us to produce a formula for the number of bits needed as input to a ROM in order to look up the result of a mod operation. We are given that there are $2^N - 1$ memory banks, and an input address that is M bits wide. We will assume that N is chosen such that $2^N - 1$ is prime. According to the problem in the text, after step 3, an input address A will be represented in the form:

$$A = \sum_{i=0}^{N-1} \text{Term}_i$$

where the i'th term Term_i is the sum of the binary digits a_i, a_{i+N}, a_{i+2N}, ... There will be $\lceil \frac{M}{N} \rceil - 1$ of these terms, since the last bit we have is a_{M-1} and the interval between bits in the term is N. So:

$$\text{Term}_i = 2^i \times \sum_{j=0}^{\lceil \frac{M}{N} \rceil - 1} a_{i+N \cdot j}$$

A simple upper bound on this sum for Term_i is to treat the upper sum bound as just $\lceil \frac{M}{N} \rceil$ and all bits a_k as 1. Then,

$$\text{Term}_i \leq 2^i \cdot \left\lceil \frac{M}{N} \right\rceil$$

and

$$A \leq \sum_{i=0}^{N-1} 2^i \left\lceil \frac{M}{N} \right\rceil = \left\lceil \frac{M}{N} \right\rceil \sum_{i=0}^{N-1} 2^i = \left\lceil \frac{M}{N} \right\rceil \cdot (2^N - 1)$$

Finally, we want to compute the number of bits needed to represent this maximum value (which is equal to the number of input lines to the mod lookup ROM). This is just the ceiling of the base-2 log of the maximum value:

$$\begin{aligned} nbits &= \left\lceil \log_2 \left(\left\lceil \frac{M}{N} \right\rceil \cdot (2^N - 1) \right) \right\rceil \\ &= \left\lceil \log_2 \left\lceil \frac{M}{N} \right\rceil + \log_2 (2^N - 1) \right\rceil \\ &\leq \left\lceil N + \log_2 \left\lceil \frac{M}{N} \right\rceil \right\rceil . \end{aligned}$$

Finally, the reduction in the size of the address is given by:

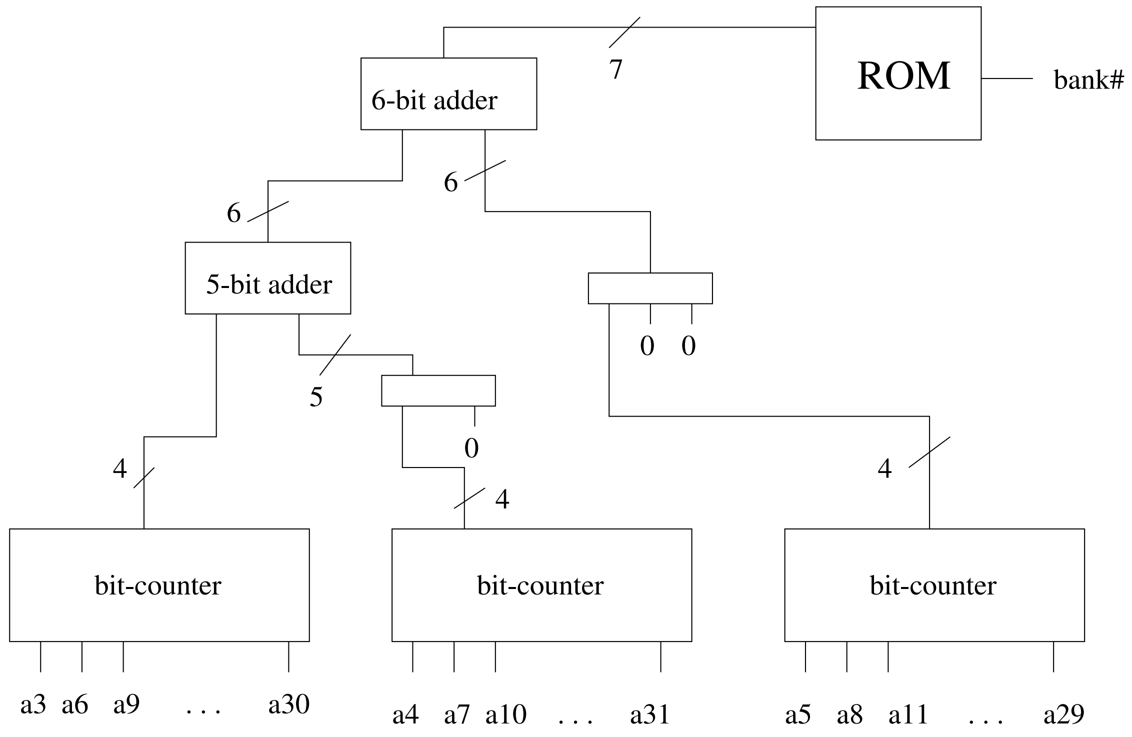
$$\frac{M}{\lceil N + \log_2 \lceil \frac{M}{N} \rceil \rceil}$$

For the example with $M = 32$ and $N = 3$, an upper bound on the maximum size of the address is

$$\left\lceil 3 + \log_2 \left\lceil \frac{32}{3} \right\rceil \right\rceil = \lceil 3 + 3.46 \rceil = 7 \text{ bits.}$$

(b) In this part we want to come up with simple hardware to pick the correct bank out of 7 given a 32-bit address. We are told to assume that the bank width is 8 bytes, and thus the lower 3 bits of the address are used to select a byte within a row of a bank. Thus, we effectively have a $M = 29$ bit bank address. Again, $N = 3$.

The following diagram shows a simple way of implementing the bank number generator. Since the ROM has 7 input lines and returns a 3-bit value (the mod-7 of the input), it must have $2^7 = 128$ 3-bit entries, or 384 total bits. The blocks labeled as "bit-counters" return (as a 4-bit number) the number of set (high) input lines. These could be implemented as LUTs, a rippled chain of 2-bit adders, a tree of wider adders, or something more clever.



$\boxed{5.20}$ **(8 points)**

In answering this question, we must consider the four functions of a virtual memory system as presented in class:

1. to provide a larger effective memory space by using a paging disk to back physical memory
2. to provide memory protection
3. to provide translation between multiple virtual address spaces and one physical address space
4. to provide a means of sharing memory between virtual address spaces

The argument presented in the problem is that memory capacity is growing quickly, so we no longer really need virtual memory to provide a larger effective memory space (since most programs that people want to run will fit directly in physical memory). While true, this argument only addresses the first function of a virtual memory system. The final three points above are critical to the functioning of a multiprogrammed computer system. Protection allows multiple untrusted user programs to be running at once, without worries about memory corruption. Translation allows all programs to be loaded at the same virtual address (and allows shared code such as kernel entry points to be at a static location in each process's address space), simplifying linkers, loaders, and the kernel. Sharing allows multiple processes to synchronize and share data with low overhead. Of course, protection, translation, and sharing could be achieved without virtual memory, but at the cost of increased software complexity and programmer burden. For example, protection could be provided by running all code in a Java virtual machine; the need for translation can be sidestepped by using relocating dynamic linkers; and sharing could be provided by cooperation between running JVMs or via some OS IPC mechanism. The existence of virtual memory makes all of these problems much simpler.

Perhaps a more compelling argument would be that future systems should no longer include operating system support for paging/swapping to disk. This allows the system to retain the protection, translation, and sharing mechanisms provided by VM, yet eliminates the complexity of managing disk swapping. However, since most OS's reuse the paging code to support such useful features as memory-mapped files and demand-paged executables, it is unlikely that a significant reduction in complexity would be achieved by removing support for paging to disk.