Name _____(answer key)_____

Computer Architecture and Engineering
CS152 Quiz #2
March 7th, 2011
Professor Krste Asanović

Name:___<ANSWER KEY>___

This is a closed book, closed notes exam.
80 Minutes
14 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get **no** credit for selecting multiple-choice answers without giving explanations if the instruction ask you to explain your choice.

Writing name on each sheet	_____	2 Points
Question 1	_____	30 Points
Question 2	_____	24 Points
Question 3	_____	24 Points
TOTAL	_____	**80 Points**

Question 1: Column-Associative Caches (30 points)

In this problem we analyze two different proposals for a cache to be used in a high-performance processor. We care about both the cycle time (which is limited by the cache hit access time) *and* the miss rate.

The first proposal is for a two-way set-associative cache (see Appendix B). The second design comes from Agarwal and Pudar, who proposed a *column-associative cache* in 1993. They recognized the main advantage of direct-mapped caches is their fast hit time, but their main disadvantage is the large number of conflict misses. A set-associative cache can improve the miss rates over the direct-mapped cache, but its hit time is worse because the expensive tag checks are placed on the critical path.

The column-associative cache aims to achieve the hit time of a direct-mapped cache, but with the lower miss rate of a two-way set-associative cache. The design of this cache is shown below in Figure 2. It's probably best to think of it as a two-way cache, but that data is always read out of `Way0`.

If the memory access is a hit in `Way0`, then the processor continues as normal (a "fast hit"). If it misses in `Way0`, then `Way1` is checked for the memory line. If there is a hit in `Way1`, then the data from `Way1` is returned to the processor (the "slow hit") and the cache lines in `Way0` and `Way1` are swapped (so that the next access to the line will be a fast hit in `Way0`). If both ways miss, `Way1` is evicted, `Way0`'s cache line is moved to `Way1`, and the accessed memory is allocated to `Way0`. In short, the most-recently-used line is always moved to `Way0`.

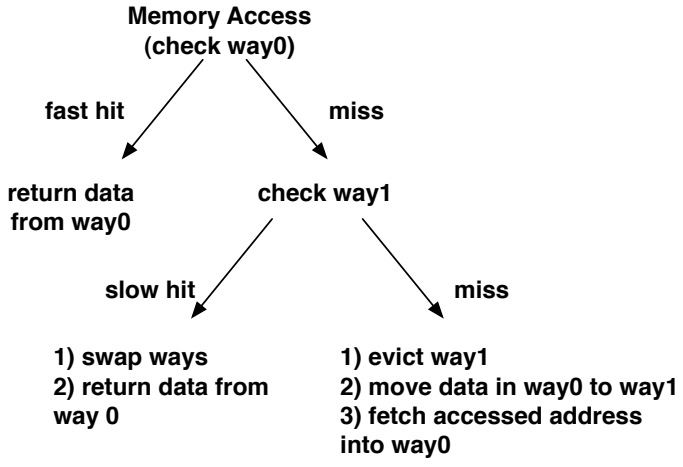

Figure 1. Column-Associative Cache Behavior

A "fast hit" returns data in a single cycle (hit time = 1 cycle). A "slow hit" takes 4 cycles:

1st cycle - detect miss in Way0
2nd cycle - check `Way1`'s tag, write `Way0` and `Way1` to the swap registers
3rd cycle - swap ways (on tag hit for `Way1`)
4th cycle - read out the data (now located in `Way0`)

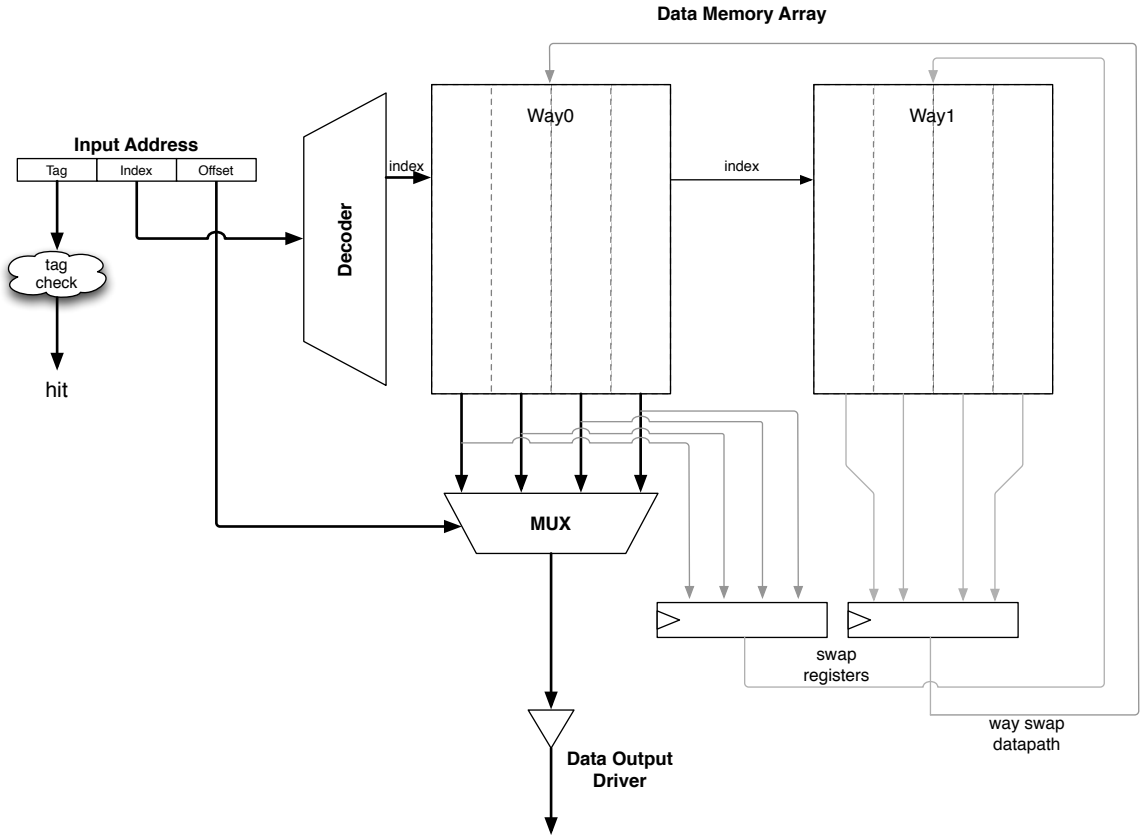

Figure 2. Column-Associative Cache Datapath

Q1.A: Cache parameters (2 points)

Fill out the following table of cache parameters. The address size is 32 bits, the index size is 8 bits, and the block offset size is 4 bits (*note*: these parameters will hold for the rest of Question 1).

	2-way set-associative	column-associative
tag size (bits)	20 bits	20 bits
line size (bytes)	2^(blk_offset_sz) = 16 bytes	16 bytes
number of sets	2^(index_sz) = 256 sets	256 sets
cache capacity (KB)	256*16*2=8192 bytes (8KB)	256*16*2=8192 bytes (8KB)

Table 1. Cache Parameters

Q1.B: Critical Paths (8 points)

Using Figure 1, Figure B-1 (found in Appendix B), Table 1, and Table 2, determine and *explain* what the critical path is for both the two-way set-associative cache (SA) and the column-associative cache (CA). Also, determine the cache access time (ps) for both configurations (i.e., the delay through the critical path).

Component	Delay equation (ps)		2-way set associative	column-associative
Decoder	20×(# of index bits) + 60	Tag	220	220
		Data	220	220
Memory array	$20\times\log_2$ (# of rows) + $20\times\lfloor\log_2$ (# of bits in a row)$\rfloor$+100	Tag	360	340*
		Data	420	400*
Comparator	20×(# of tag bits) + 50	Tag	450	450
2-input AND	40		40	40
2-input OR	100		100	100
N-to-1 MUX	$50\times\log_2$ N + 200		300	300
Buffer driver	190		190	190
Data output driver	90×(associativity) + 100		280	190
Valid output driver	50		50	50

Table 2. Delay of each cache component

**Note*: For the column-associative cache, you only need to consider the fast-hit scenario. In this case, only Way0 needs to be accessed, allowing it to have a faster memory array access time. You can also ignore the tag check path in the column-associative cache. We promise it's not on the critical path.

SA Critical Path:

tag decoder->tag array->comparator->AND->buffer driver->data output driver
= 220+360+450+40+190+280= 1540ps

CA Critical Path:

decoder->memory array (only way0)-> mux->data output driver
= 220+400+300+190=1110ps

SA Access Time : 1540 ps
CA Access Time : 1110 ps

The following questions evaluate the cache performance of the following C code, which performs an in-place vector-vector add. A vector **A** and a vector **B** are added together. The result is this written back to vector **A**. Pseudo-disassembly of the inner loop is shown on the right.

```
#define N 4096
int A[N], B[N];
int i;

for(i = 0; i < N; i++)
  A[i] = A[i] + B[i];
```

```
# rA holds the addr to A[i]
# rB holds the addr to B[i]
LD  r2,0(rB)
LD  r1,0(rA)
ADD r1,r1,r2
ST  r1,0(rA)
```

Assume **A** and **B** are cache aligned to a 4KB boundary and are contiguous in memory. **int**s are 32 bits (4 bytes).

Also assume that the caches will use the parameters found in Table 1: Cache Parameters.

Q1.C: Miss-rates? (2 points)

What is the miss-rate for the two-way set-associative (using LRU) and column-associative caches running the above code? (percentage of memory accesses that completely miss in the cache and require fetching the data from main memory?).

By the stated definition of miss-rate, both the SA and CA caches will look identical.

cache line size found in Q1.A = 16 bytes, or 4 elements.

This is a unit-stride walk through memory, so the first access to each array will be a compulsory miss, but spatial locality will allow us to hit on the next 3 elements.

Each inner loop is 3 accesses, and when i%4==0 that's two misses, but the next 3*3+1 accesses will hit (i.e., the next 3 iterations, plus the store to **A** in the first iteration).

So miss_rate = 2 misses / 12 accesses = 16.67%.

(both **A**[i] and **B**[i] will map to the same index, but because it's two-way, they won't kick each other out).

SA cache: _16.67%

CA cache: _16.67%

Q1.D: Hit-rates? (2 points)

What is the hit-rate running the above code for the two-way set-associative (using LRU) and the column-associative cache?

For the SA cache: hit-rate = (100% - miss_rate), so 83.33%.

For the CA cache:

Both **A**[i] and **B**[i] will map to the same index, but because it's two-way, they won't kick each other out. However, if **B**[i] is in Way0, an access to **A**[i] will kick **B**[i] to Way1 (and vice versa).

For every 12 accesses (4 loop iterations, 3 inner accesses per loop), first iteration is 2 cache misses, 1 fast hit. the next three iterations will all slow hit the loads, and fast hit the store.
so fast_hit = 4/12 = 33.33%
and slow_hit = 6/12 = 50.%

-1 for people who said 2/3 slow hits, 1/3 fast hits, and didn't factor in the first iteration.

SA cache:__*83.33%*
CA cache:__*33.33%* (fast hit rate)
__*50.00%* (slow hit rate)

Q1.E: AMAT (4 points)

What is the Average Memory Access Time from running the above code for the two-way set-associative and the column-associative cache? Assume the miss penalty is 100 ns. Also assume that the processor's clock speed is limited by the cache access time, which was computed in Q1.B.

For the SA cache:
AMAT_sa = hit_time + miss_rate*miss_penalty
AMAT_sa = 1 cycle (1.54ns/cyc) + (.1667)*100ns = 1.54ns+16.67ns = 18.21ns

Note that the fast_hit_time is always paid for, because you *always* have to eat a cycle to detect if a miss occurred before you can initiate the miss sequence.

An other way to think of this is:

AMAT_sa = hit_rate*hit_time + miss_rate*miss_time

Where: miss_time = miss_penalty + time_to_detect_miss

In the SA cache, time_to_detect_miss is 1 cycle (i.e., the hit_time, because that's when you check both ways for a miss).

For the CA cache, perhaps the clearest equation is [Equation #1]:

AMAT = fast_hit_rate * fast_hit_time
+ slow_hit_rate * slow_hit_time
+ miss_rate * (miss_penalty + time_to_verify_miss)

A more common, but confusing way to think about this is [Equation #2]:

AMAT= fast_hit_time
+ slow_hit_rate*(slow_hit_time - fast_hit_time)
+ miss_rate*(miss_penalty+(slow_hit_time-3))

Note: It was stated that "slow_hit_time" is 4 cycles BUT that's double-counting the fast_hit_time (i.e., double-counting the time to detect a miss in Way0). If the fast_hit_time was weighted 100%, then the student needed to be careful to subtract the fast_hit_time from the slow_hit_time (i.e., slow_hit_penalty = slow_hit_time - fast_hit_time).

Note#2: In the same vein, the student must also account for the fact that the miss_penalty doesn't start until *after* we've discovered the memory access is not a slow-hit, so pure misses must also add the additional cycle spent checking Way1's tag (basically one cycle to check Way0 for a miss, then another cycle to check Way1 for a miss). Thus "time_to_verify_miss" = 2 cycles, meaning the "miss_time" = "miss_penalty + 2 cycles".

Note#3: Again, Note#2 does *not* imply that you must add *all* of the slow_hit_time to the miss_time. You only need to check Way0 and Way1 to verify the miss before you activate the memory request, and start the miss_penalty countdown (however, full credit was given if you assumed you had to go through the entire slow-hit process).

The following equation is *wrong* because it fails to account for the cycles spent checking for misses, which must always occur first before you can start paying *miss_penalty* cycles on a memory request.

~~AMAT_ca=fast_hit_rate*fast_hit_time + slow_hit_rate*slow_hit_rate+miss_rate*miss_penalty~~

Most credit was given for this equation however, because it only short-changes the time spent servicing a miss by a few nanoseconds (out of the >100ns).

So using Equation #1 we get, and filling in from previous questions:

AMAT_ca = (0.33)*(1 cycle)(1.11ns/cycle)+(0.5)(4 cycles *1.11ns/cycle) + (.1667)*(100ns +2cycles*1.11ns/cycles) = 0.37ns+2.22ns+17.04ns = 19.63ns

Final note: the CA's poor fast hit rate hurts AMAT performance (slightly), but the cpu can be clocked quite a bit faster! (900MHz vs 650MHz)

SA cache: _18.21ns
CA cache: _19.63ns

Q1.F: Crossover Point in Performance (3 points)

The performance of the column-associative cache is highly dependent on the fast-hit rate. What is the ratio of fast-hits to slow-hits for which the AMAT is the same for the two-way set-associative and the column-associative caches?

This question is asking when is (AMAT_sa == AMAT_ca). From Q1.E, we get:

AMAT_sa = hit_rate*1.54ns + miss_rate*(1cycle+miss_penalty) =
AMAT_ca = fast_hit_rate*1.11ns+slow_hit_rate*(4*1.11ns) + miss_rate*(2cycles +miss_penalty)

(both have the same miss_rates, and the actual miss_times only differ by about a nanosecond out of 100ns), so for this problem we can ignore the miss_rate and miss_times and focus on the hits. This leaves us with:

AMAT_sa = AMAT_ca =
hit_rate*1.54ns = fast_hit_rate*1.11ns + slow_hit_rate*4.44n

if fast_hit_rate=x, slow_hit_rate=y then:

(x+y)*1.54 = 1.11x + 4.44y

1.54x + 1.54y = 1.11x + 4.44y

0.43x = 2.90y

the problem is asking for the ratio of x/y (fast hits to slow hits), so if r=x/y then:

x/y = r = 2.90/0.43 = 6.74

so we need >6 times as many fast hits as slow hits.

This makes sense because we eat 4 cycles for a slow hit, so we'd want >4 one-cycle fast-hits for every 4-cycle slow-hit.

Q1.G: Crossover Point in Energy (3 points)

Assume that the energy usage of the cache is dominated by the memory array access, in which a single way access is 1 nJ (both read and writes operations take 1nJ). *Hint:* notice that a column-associative cache does not look at `Way1` unless it misses in `Way0`.

How much energy is used during the following scenarios? *Explain* what accesses are occurring in each scenario.

A fast-hit for a column-associative cache?
1nJ (only read way0)

A hit for a two-way set-associative cache?
2nJ (must read both ways and both sets of tags to see which has the data)

A slow-hit for a column-associative cache?
6 nJ
(**cycle1**: read way0, verify miss - 1nJ
cycle2: read out both ways to the swap registers and check way1's tags - 2nJ
cycle3: write to both ways to finish swap -2nJ
cycle4: read out way0's data - 1nJ)

Q1.H: Crossover Point in Energy (3 points)

What is the ratio of fast-hits to slow-hits for a column-associative cache for it to be more energy efficient than a two-way set-associative cache?
For this problem, we're only consider energy used during the memory array accesses. Therefore, we can ignore the miss scenario because it happens sufficiently rare that it won't be more than a 2nd order effect on the energy, with regards to accessing the memory arrays within the cache.

So with this assumption made of only considering accesses hits in the cache we get the following equation (using Q1.G):

SA_energy = CA_energy
hit_rate*(2nJ) = fast_hit_rate*(1nJ) + slow_hit_rate*(6nJ)

if x = fat_hit_rate, y = slow_hit_rate....
(x+y)2 = x + y6
2x+2y = x + 6y
x = 4y

x/y = 4, or .... 4 fast hits + one slow hit == 5 SA hits

Q1.I: Aliasing (3 points)

Assuming that the page size is 4KB for this machine, and the cache parameters from Table 1 still hold, and the cache is physically-tagged, virtually-indexed, does the column-associative cache run in to problems with aliasing? *Explain* your answer.

Aliasing is not a problem. index_sz+offset_sz=page_offset_sz, this behaves like a regular SA cache.

-2 points if said “yes” because aliases can fit into both ways. This is technically true, but in practice can never occur. Consider VA1 and VA2, which both alias to PA. VA1 accesses the cache and brings PA, with physical tag Pt, into Way0. VA2 then accesses the cache. It maps to the same set as VA1 (because index_sz+offset_sz <= page_offset_sz). It also maps to the same tag (Pt). Therefore, VA2 hits in the cache and returns the data pointed to by PA. Thus there are no problems because the second alias will never be brought into the cache and written to an other way - it will simply use the first alias’s copy.

Name _____(answer key)_____

Question 2: Three C's of Cache Misses (24 points)

	Compulsory Misses	Conflict Misses	Capacity Misses
Changing from two-way set-associative to column associative (See Question 1 of this Quiz for details)	no change nothing changes because they look identical. only change is slow-hit/fast-hit semantics	no change (same reason) -1 for increase. while the number of conflict cache misses stays constant, there is an increase in average hit time access due to less "fast" ways	no change (same reason)
Add a sub-blocking scheme (Divide block into sub-blocks with a valid bit for each sub-block) (line size constant)	increases we now bring in less of the block (sub-blocking saves you on memory bandwidth, and this miss_penalties, by only bringing in a sub-block when needed).	no change associativity constant	no change capacity constant
Double line size to *2n* bytes, by combining two lines of size *n* bytes with a valid bit for each block (again, sub-blocking) (capacity constant)	no change because of sub-blocking, we still bring in the same amount of data to exploit for spatial locality	increases less sets so more addresses will conflict with one another	no change capacity constant
Adding prefetching	decreases (it's kind of the point of the prefetcher)	increases prefetch data could pollute the cache -1 for saying no change	increases prefetch data could pollute the cache. -1 for saying no change

Name _____(answer key)_____

Question 3: TLB Performance (24 points)

	TLB contribution to the CPI	TLB reach	TLB capacity misses
Increase page size	decreases, because TLB can hold translations for a greater region of memory, so there will be fewer misses across a given program's accesses	increases each TLB entry maps to a larger space	decreases, since TLB has greater reach -1 for saying "no change if capacity viewed as # of entries".
Increase number of TLB entries	decreases, because fewer TLB misses (conflict and capacity)	increases (pagesize*# of entires)	decreases (TLB is now bigger and so provides more capacity)
Increase the number of levels in the virtual-memory hierarchy ---------- some ambiguity here, intended to refer to levels in page table hierarchy, but correct answers given if assumed that the question asked about # levels in the TLB hierarchy	increases (more levels in the page table to walk over on each miss, which translates directly to more memory accesses required to compute the path to the next level). ---------- decreases because of more "caching" provided by more TLB levels	no change ---------- increase if stated that outer levels of the TLB provide more capacity	no change ---------- it was okay to say "decreases if the outer levels in the TLB hierarchy captured the inner level misses"
Increase virtual address size from 32-bits to 64-bits This forces more levels in the VM page table hierarchy to be added	increases (more levels in the page table to walk over on each miss) -1 for stating the # of levels in the hierarchy would remain unchanged	no change unchanged, as page size and # of TLB entries remains constant	no change unchanged, as page size and # of TLB entries remains constant

END OF QUIZ

Appendix A. Direct-mapped Cache

The following diagram shows how a direct-mapped cache is organized. To read a word from the cache, the input address is set by the processor. Then the index portion of the address is decoded to access the proper row in the tag memory array and in the data memory array. The selected tag is compared to the tag portion of the input address to determine if the access is a hit or not. At the same time, the corresponding cache block is read and the proper line is selected through a MUX.

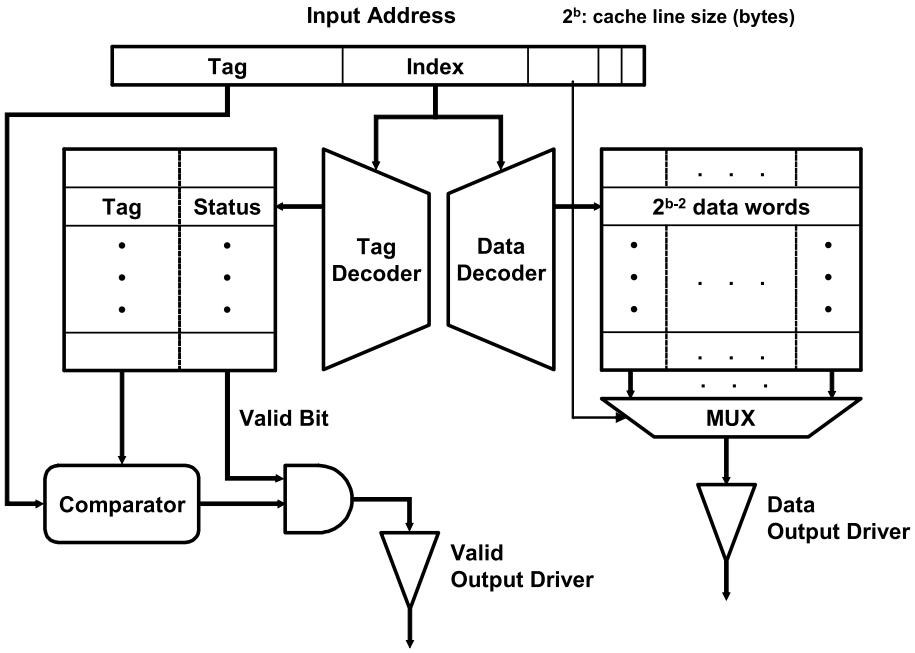

Figure A-1: A direct-mapped cache implementation

In the tag and data array, each row corresponds to a line in the cache. For example, a row in the tag memory array contains one tag and two status bits (valid and dirty) for the cache line. For direct-mapped caches, a row in the data array holds one cache line.

Appendix B. Two-way Set-associative Cache

The implementation of a 2-way set-associative cache is shown in the following diagram. (An *n*-way set-associative cache can be implemented in a similar manner.) The index part of the input address is used to find the proper row in the data memory array and the tag memory array. In this case, however, each row (set) corresponds to two cache lines (two ways). A row in the data memory holds two cache lines (for 32-bytes cache lines, 64 bytes), and a row in the tag memory array contains two tags and status bits for those tags (2 bits per cache line). The tag memory and the data memory are accessed in parallel, but the output data driver is enabled only if there is a cache hit.

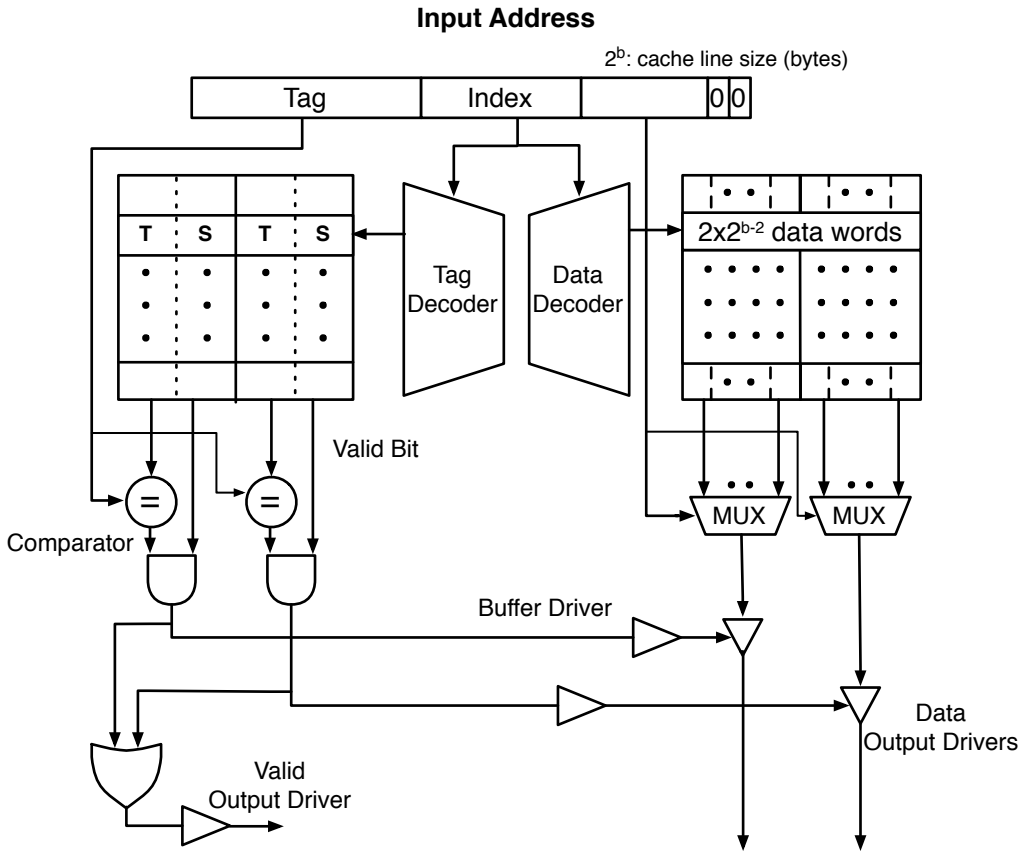

Figure B-1: A 2-way set-associative cache implementation