

Memory and Caches

Prof. Loh

CS3220 - Processor Design - Spring 2005

March 10, 2005

1 Simple SRAM Memory

On chip memory arrays are usually implemented as static memory (SRAM) cells. Off chip memory is usually dynamic memory (DRAM). A cell (one bit) of DRAM is usually little more than a capacitor that holds a charge (1) or not (0), plus a single transistor to access the bit (common DRAM cells vary from 1 to 3 or 4 transistors). Because of this, DRAM cells can be packed very densely. Unfortunately, DRAM cells are also much slower than SRAM cells. A SRAM cell is really just a latch or flip-flop, but this takes more area to implement (usually about six transistors). These are faster and we will focus our attention on on-chip SRAM structures.

Figure 1 shows a simple 8 byte read-only SRAM memory, arranged as two rows of 4 bytes per row. Each “latch” icon represents a one-byte wide storage unit (eight SRAM cells). For 8 distinct storage locations, we need $\log_2 8 = 3$ bits to uniquely address all of the locations. In this example, the most significant bit $a[2]$ is used to choose one of the two rows. All locations in the selected row will have their tri-state buffer enabled. The lower two bits of the address $a[1, 0]$ are used to select from one of these 4 bytes (we will be using the notation $x[i, j]$ to represent bits i through j (inclusive) of x). Because the outputs of all locations within the same column are connected by a shared bus, it is important that the logic for selecting a row chooses *exactly* one row.

Figure 2 shows another 8 byte read-only SRAM memory, but arranged as four rows of 2 bytes per row. This time, address bits $a[2, 1]$ are used to select one of the four rows, and the least significant address bit $a[0]$ is used to select from one of the two columns.

In general, we have have a $w2^n$ bit memory, where n is the number of address bits, and w is the number of bits per memory location (8 in our previous two examples). From the n bits of address, r bits are used to choose one of the 2^r rows, and $c = n - r$ bits are used to select from one of the 2^c columns. The logic to enable one of the 2^r rows is called the *row decoder*. The logic to select one of the 2^c columns is the *column selector*. Figure 3 shows a general read-only SRAM structure. The *access time* to read a memory location is composed of several components:

$$\text{access time} = \overbrace{d(\text{row decode}) + d(\text{SRAM cell}) + d(\text{line driving})}^{\text{majority of the latency}} + d(\text{select})$$

Note that in Figure 2, the row decoder logic used $2^r = 4$ AND gates, each with $r = 2$ inputs. In the general case, a r -input AND gate would be required. In most cases, the *fan-in* (number of inputs) of a CMOS logic gate is limited to three or four inputs, and so the row decode logic would have to be implemented with multiple levels of logic. This results in a $O(\lg r)$ gate delay to perform the row decode. Figure 4(a) shows a simple 1-of-2 decoder block. If s is high, then $y_{up} = x$; if s is low, then $y_{down} = x$. The other input is always low. Now a 1-of- k decoder block can be inductively constructed. The base case is the 1-of-2 decoder. Figure 4(b) illustrates the inductive case where a 1-of- k decoder is constructed with a 1-of-2 decoder and two 1-of- $\frac{k}{2}$ decoders. In the final 1-of- k decoder, the input x is set to one (high).

To pack in as many SRAM cells as possible, the transistors used in each cell are very small. Smaller transistors take longer to drive larger outputs. The column buses that a selected cell must drive can be quite long, thus requiring a long latency to read the data. Figure 5 shows a typical six transistor (6T) SRAM cell. The two looped inverters are the “storage” remembering the stored value. There are also two access transistors, which are controlled by the row select.

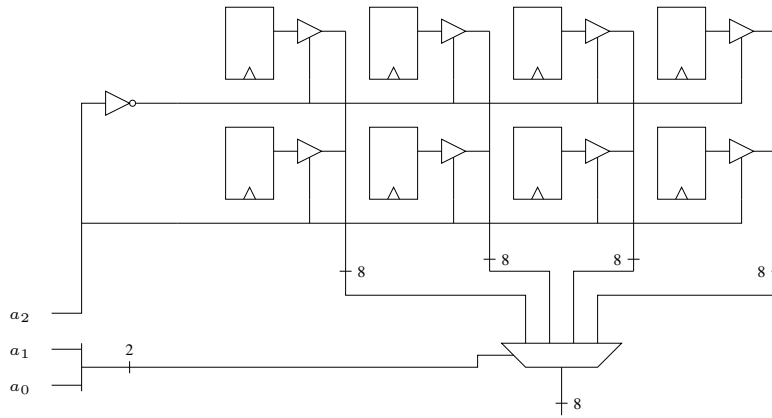


Figure 1: An 8 byte read-only SRAM memory, arranged in a 2x4 configuration.

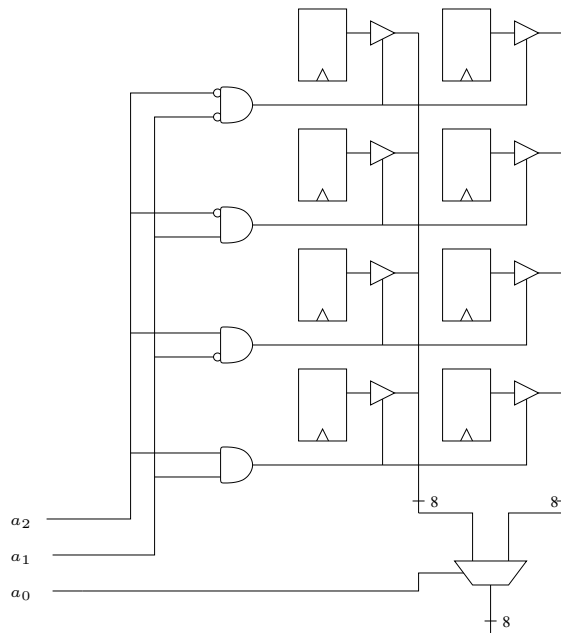


Figure 2: A 4x2 byte read-only SRAM memory.

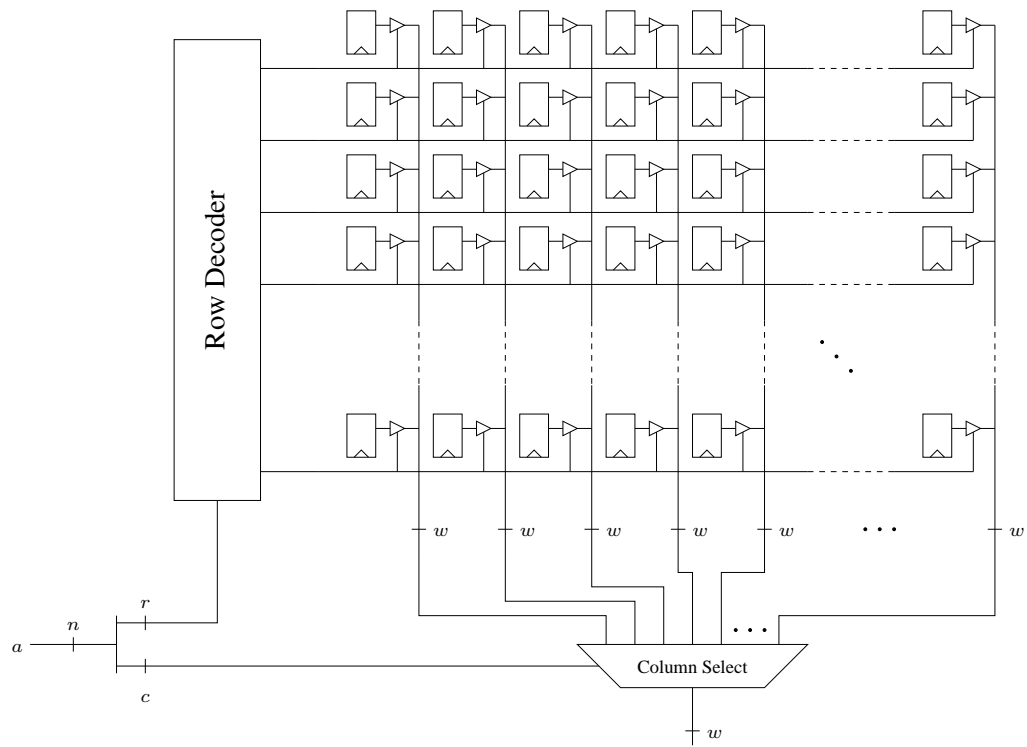


Figure 3: A generic r by c location read-only SRAM memory with w bits per memory location.

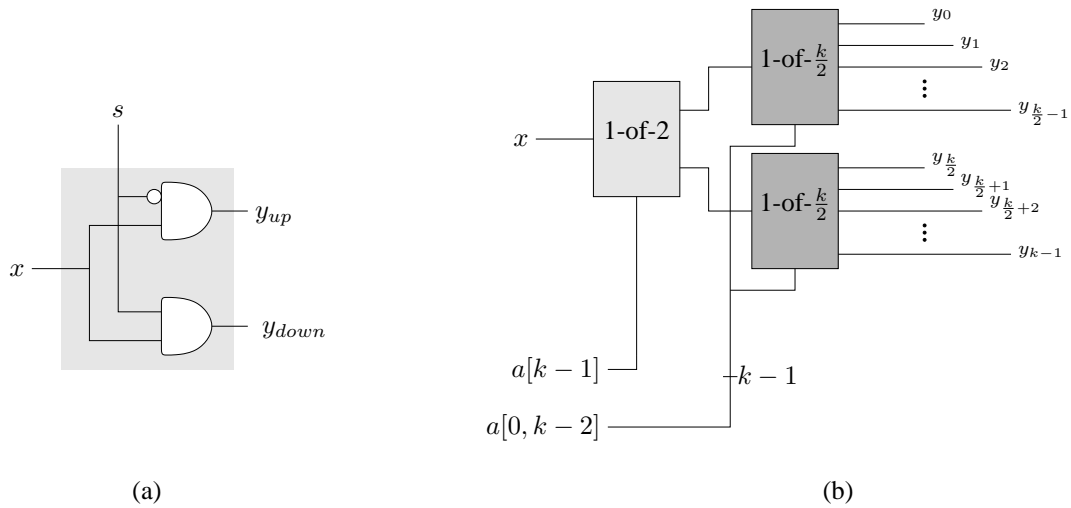


Figure 4: (a) a 1-of-2 decoder. (b) a 1-of- k decoder constructed recursively with a 1-of-2 decoder and two 1-of- $\frac{k}{2}$ decoders.

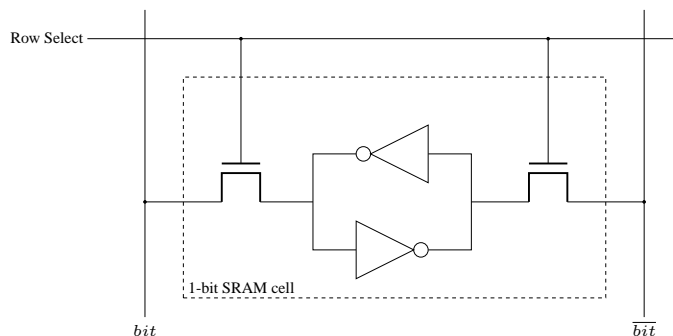


Figure 5: A simple read-write SRAM cell consisting of only six transistors (an inverter uses two transistors).

On one side of the inverter loop, the stored value x is maintained, and the other side keeps the inverted value \bar{x} . There are really two column lines, labeled bit and \bar{bit} . To perform a read, both the bit and \bar{bit} lines are precharged to V_{dd} . When the row is selected, one of the lines will start to get pulled towards V_{ss} (ground), albeit slowly. At the bottom of the column lines, a *sense amplifier* (a type of differential amplifier - it amplifies the voltage difference between its inputs which are the two bitlines) can quickly detect the slight change in voltage and drive the appropriate signal for the column select multiplexor.

To perform a write to a memory location, the value $data$ to be stored is driven on the bit line, and \overline{data} is driven on the \bar{bit} line. The drivers are stronger than the storage inverters, and so the inverters are “over-ridden” by the drivers. The bit/\bar{bit} lines are only driven in the column of the destination cell, and only the destination row is enabled for reading. All other columns are tri-stated, and so the other columns in the selected row will simply be driven by the storage inverters.

Multi-porting

With this arrangement, we can only have a single read or a single write operation occur at one time because the row decoder can only enable one row at a time. We are interested in memory structures that allow multiple simultaneous memory transactions. A simple example is that we will want to be able to load a value from memory, and at the same time load the next instruction that we will want to execute. We will look at ways to enable two read ports (i.e. being able to read/load from two distinct memory locations at the same time), although these approaches can be generalized to allow an arbitrary number of read ports.

The first approach is to simply duplicate the row decoder and column select logic to allow multiple rows to be enabled at the same time. Figure 6 shows a generic memory that has two copies of the row and column logic. This also requires the addition of two separate row select lines, and thus two sets of access transistors per SRAM cell. Figure 7 shows such a SRAM cell. The disadvantage with this approach is that the area cost of having an 8T cell with double the number of column lines make the overall memory much larger, and therefore slower. Either the cycle time or memory size must be sacrificed.

Another approach to providing multiple read ports is to simply duplicate the entire memory structure. At first this may seem wasteful, but it is sometimes the case in VLSI circuits that the total area used is not so important so long as the latency of the circuit is kept low. In the previous approaches, larger circuits meant that wires would have to travel longer distances. With full replication, each structure is the same size as the original structure, and so the wire lengths are not affected. Figure 8 shows a SRAM memory structure that provides two read ports by means of full replication.

Yet another approach sometimes called *double pumping* can be used. In double pumping, the read requests to the SRAM are issued back to back, with the first request being issued in the first *half* clock cycle, and the second request being issued in the second half of the clock cycle. The memory must be small/fast enough to allow two accesses in a single cycle. Such a technique may be useful in situations where the amount of memory is limited. There may only be enough room to provide a small memory, but a larger read bandwidth is still desired. Double pumping is used in the

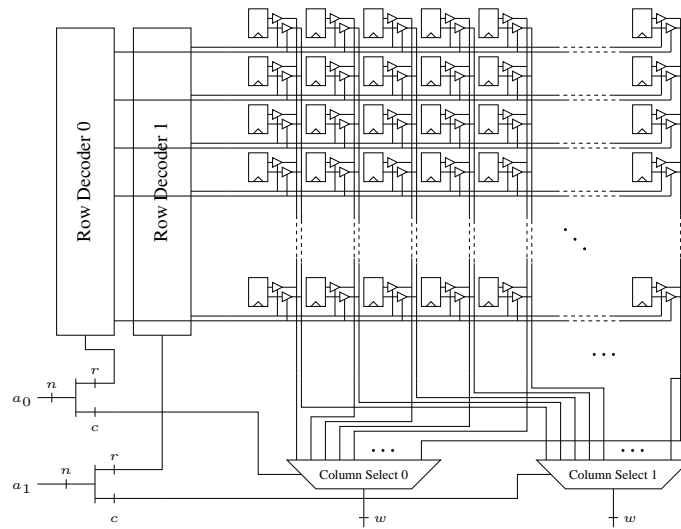


Figure 6: A memory with 2 read ports.

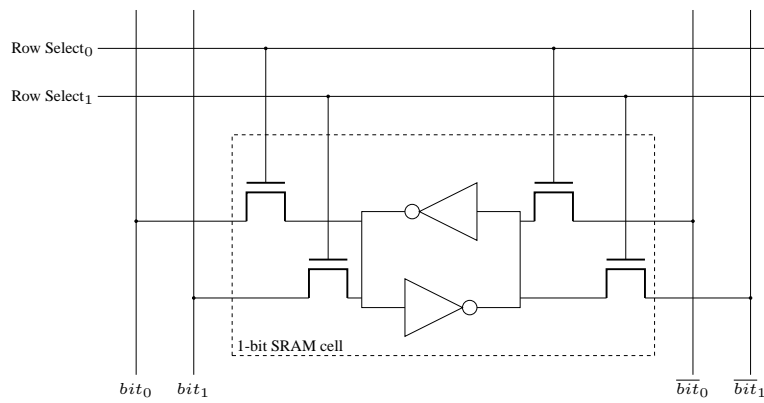


Figure 7: A SRAM cell with two separate sets of access transistors. A total of 8 transistors are needed.

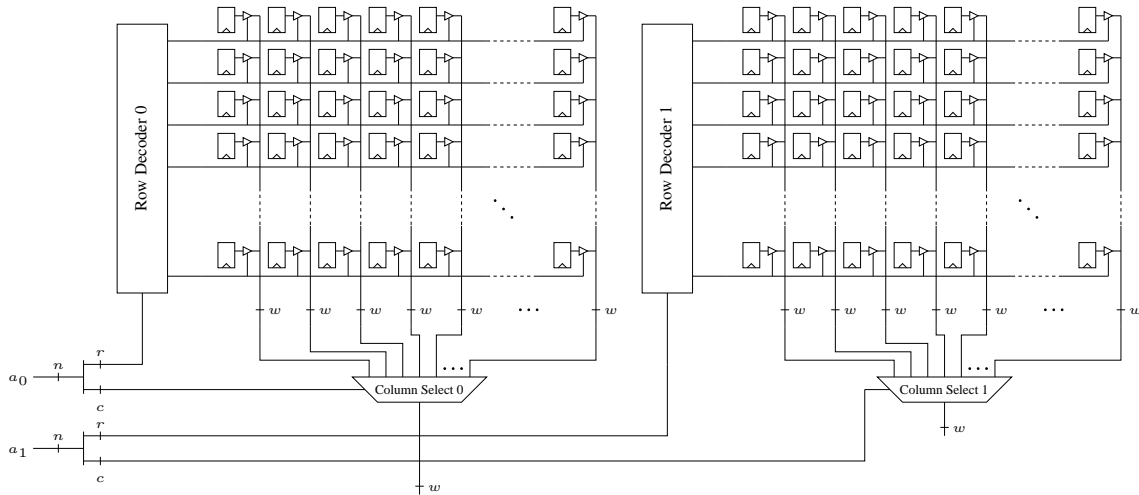


Figure 8: A two read port memory implemented by replication.

Alpha 21264 CPU's on chip data cache.

The complexity of supporting writes with multi-ported memories varies depending on the implementation strategy. With full replication, the write must occur in both copies of the memory to prevent a situation where a read from a location in one copy returns a different result than a read from the same location in another copy. In the cases where the various decoder/selection logics are duplicated, more complex routing is necessary to simultaneously route multiple results to multiple locations. There are also additional issues with handling the situation where more than one write is destined for the same column. It is not uncommon to find memories that support more simultaneous reads than writes for these reasons. The double pumped solution does not suffer from these deficits, but none of these techniques scale very well to larger read/write capacities.

Another possible solution to providing more memory bandwidth is *banking or interleaving*. A simple interleaving scheme is to divide up all of the memory locations into two disjoint sets (such as all odd addresses and all even addresses). Each set of locations is grouped in a separate memory structure. This has the advantage that each of these structures are smaller, and therefore faster. Additionally, each structure can be accessed in parallel. Figure 9 shows a memory that is 2-way interleaved which allows two memory operations per cycle *so long as both operations are to memory locations in different sets*. If there are two memory operations to odd locations at the same time, one of them will have to wait until the other is finished. Note that only $c - 1$ bits are used to select the column because the least significant bit is already being used to select between the odd and even banks (alternatively, $r - 1$ bits could be used to select from half as many rows). This has the advantages similar to full replication, except that the memory structures are also reduced in size, and writes only have to go to a single bank. The tradeoff is in possible bank conflicts, and the added hardware complexity associated with handling the conflicts.

Interleaving can be generalized to k -way interleaving. The sets are often formed by simply taking the remainder of the address when divided by k . A memory location a belongs in set i if $a \bmod k = i$. If k is a power of two, this computation is very simple since it simply involves looking at the $\lg k$ least significant bits of a . This approach is significantly more scalable than the other techniques presented, but in many applications, even a few infrequent access conflicts can have profound performance implications, or the additional conflict handling logic may simply be too expensive to implement (in chip area and/or circuit latency). There is also the additional problem of routing memory requests to the correct bank.

Modulo addressing is not the only way the memory address space can be partitioned. Another very common approach is to separate the instruction and data memory spaces. Except for the case of self-modifying code (which is not as often used anymore), it is rarely the case that an instruction is used as data, or data as an instruction. So it is very natural to place these two classes of memory locations in separate memories. Some systems go so far as to even make two distinct memory spaces, such that instructions can never be treated as data (i.e. there are no instructions

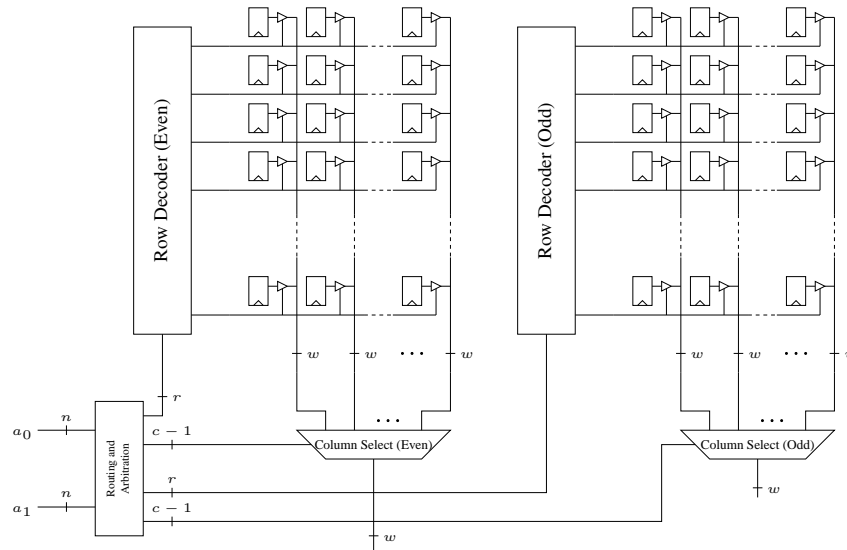


Figure 9: A 2-way interleaved memory, with the address space partitioned into odd and even addresses.

to even *read* an instruction into a general register, and the IR can never be used as an argument to an instruction). An example of such a chip is the PIC series of microprocessors. Architectures with this kind of memory partitioning are referred to as *Harvard* architectures. The other situation where there is one global address space for memory and data is a *Princeton* architecture. Although almost all CPUs for desktop/workstation/server purposes have Princeton architectures these days, the Harvard architecture concept of split instruction and data memories are still employed in modern L1 instruction and data caches (we will discuss caches shortly). This allows a memory instruction to execute during the same cycle that a new instruction is being read from memory.

2 Caching

Modern programs may require many megabytes of memory for both the code (instructions) executed and the data used. With current technologies, it's not practical to place that much memory on the same chip as the processor, so the memory is located externally. This causes a problem because an access to memory now takes a long time. The memory request must travel from the CPU to the edge of the chip, go through the I/O pads and pins, across the motherboard, through memory control logic, access the memory itself, and then make its way all the way back to the processor where the value is to be used. The amount of time it takes current processors to execute a single instruction is on the order of nanoseconds. The amount of time it takes to read a value from an off chip memory may be hundreds of times longer. Every instruction requires at least one read from memory to simply load the instruction into the CPU. In light of this, it seems silly to execute an instruction in mere nanoseconds when the processor must then sit around and wait for hundreds of cycles before it receives the next instruction to start working on. This problem would be further exacerbated by a memory instruction, since an additional access to memory must also be performed, further slowing down the system.

This has long been a problem, even back in the days of huge CISC machines that were composed of many discrete chips (i.e. before VLSI technology). The solution then was to create these huge, complex instructions that specified a lot of different things to do in very few bits. The goal was to keep the CPU as busy as possible until the next instruction was available. With the advent of RISC machines, the CPU designers took a different approach, which was to have each instruction only perform a single simple operation, but to be able to process these instructions incredibly quickly. The natural requirement is that a steady stream of instructions must be available. Clearly, going to off chip memory for every instruction is not a feasible solution.

A key observation is that even though a program may have several megabytes of code, it is often the case that only a small fraction of that is being used during a limited interval of time. Consider, for example, the short piece of code in Figure 10. Most of the time, the program will be iterating through the doubly nested loop in `main`. If we can keep a copy of the little bit of code that comprises the loops in the CPU, then we can save the long, slow trip to main memory. Keeping a small subset of a larger set of data in a fast structure is called *caching*. The benefits of caching are realized only when we repeatedly access the same items. This type of repeated-use behavior is called *temporal locality*. Temporal locality is the likelihood that a recently used item will be accessed again in the near future. In the case of our sample code, we will be using the body of the inner loop 128 times in a row before executing any other code. Now it makes sense to be able to execute instructions quickly because we can keep a steady stream of instructions coming at a fast rate.

There is another kind of locality that can be exploited in the code of Figure 10. It may be the case that we can read several bytes of memory in at a time, or perhaps we can pipeline the memory accesses. What we can do is everytime we access memory location a , we'll read it in from memory and cache it. Then while we're at it, we'll also read in locations $a + 1, a + 2, \dots, a + L$. The hope is that if we access a memory location, then it's likely that we'll also access other nearby locations. This is called *spatial locality*. In our example, whenever we use $x[j]$, we'll also use $x[j + 1]$ in the next cycle (except for the last iteration of the loop).

Figure 11 shows how exploiting spatial locality can reduce the amount of time the processor spends waiting for memory accesses to complete. On cycle 42, the CPU reaches the code that loads $x[0]$ in from memory. The processor issues the request, and on subsequent cycles, issues requests for the next three memory locations as well. The program can not make any progress until $x[0]$ is loaded from memory and the CPU stalls. 80 cycles later, the load finally completes, and the program continues its execution. On the next few cycles, $x[1], x[2]$ and $x[3]$ also arrive in the CPU, although the program has not yet asked for these locations. These values are stored on the CPU where they can be quickly accessed by future instructions. Finally, on cycle 217, the program has made it around to the next iteration of the loop and performs a load from $x[1]$. This time, the data is already present in the cache, and is quickly loaded in a single cycle. The program can now immediately continue execution without stalling.

There is a tradeoff between how many extra locations the CPU reads in when attempting to take advantage of spatial locality. If too many nearby locations are proactively cached, it is possible that only a few of them are ever actually used, thus wasting both memory bandwidth, as well as cache locations that could otherwise store other more useful data.

Cache Designs

For all of our cache designs, we'll assume that a load or store instruction operates on 32-bit data (which is typical for 32-bit machines). This also means that all memory addresses will be aligned to 4 byte boundaries (bits $a[0, 1]$ are always zeros).

Figure 12 shows a simple cache structure that can store up to eight words. In this cache, we simply use the three least significant bits (not including $a[0, 1]$ which are always zero) to choose one of the entries in the cache. The example shows $a[2, 4] = 011$, and so entry number three is where we'll look for our data. Note that *any* address with $a[2, 4] = 011$ will get mapped to this entry, so we need some way to identify what data is in the cache location. The solution is to store a *tag*. The tag consists of the upper (more significant) bits of the address that are not involved in choosing a cache entry. In this case, the tag is $a[5, 31]$. When a cache lookup is performed, the lower address bits select a cache entry, which returns a value and a tag. The tag is then compared to the upper address bits, and if they match, then we have a *cache hit*, i.e. the data we are looking for is present in the cache, and we don't have to make the long journey to main memory. Additionally, a *valid* bit is stored that specifies whether the stored data is valid or not (the 'V' bit in Figure 12). When the cache is empty perhaps at program start up, or after a context switch the data in the cache will not be valid.

This cache organization is called a *direct mapped* cache because every memory location (address) is directly mapped to only one possible entry/location in the cache. Direct mapped caches are simple and, more importantly, fast. A downside of direct mapped caches is that you can get a kind of undesirable situation called *thrashing*. Assume a program alternately reads from addresses `0xC04C` and `0xB36C`. Both of these addresses will map to the same cache entry if we use the cache from Figure 12. On the first access of `0xC04C`, we will miss in the cache simply because it's

```

#define ARRAY_SIZE 128
#define N 16

int x[ARRAY_SIZE]
int y[N][ARRAY_SIZE]

main()
{
    int i,j;

    /* ... */

    for(i=0; i < N; i++)
    {
        for(j=0; j < ARRAY_SIZE; j++)
        {
            y[i][j] = power(x[j],i);
        }
    }

    /* ... */
}

int power(int a, int b)
{
    int n;
    int r=1;

    for(n=0; n < b; n++)
    {
        r *= a;
    }
    return r;
}

```

Figure 10: A simple program that contains a loop which computes the values of array x raised to different powers.

Cycle	Instruction	Memory Action
42	load $x[0]$	issue read request for $x[0]$
43	waiting for memory	issue read request for $x[1]$
44		issue read request for $x[2]$
45		issue read request for $x[3]$
⋮	⋮	waiting for memory requests to return
122		$x[0]$ read returns, place in cache
123	call <code>power ()</code>	$x[1]$ read returns, place in cache
124	<code>r = 1;</code>	$x[2]$ read returns, place in cache
125	<code>n = 0;</code>	$x[3]$ read returns, place in cache
126	<code>n < b?</code>	
⋮	to next loop iteration...	
217	load $x[1]$	it's in the cache, no need to go to memory
218	call <code>power ()</code>	
⋮		

Figure 11: A cache can attempt to exploit spatial locality by fetching several consecutive memory locations from the off chip memory at the same time, even when only one is requested. If the subsequent locations are later used, the processor will not have to wait for the lengthy memory access.

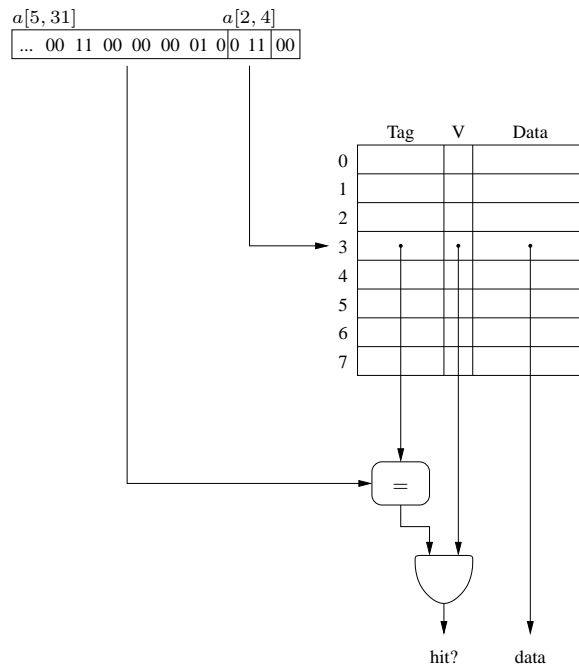


Figure 12: An example eight entry direct mapped cache.

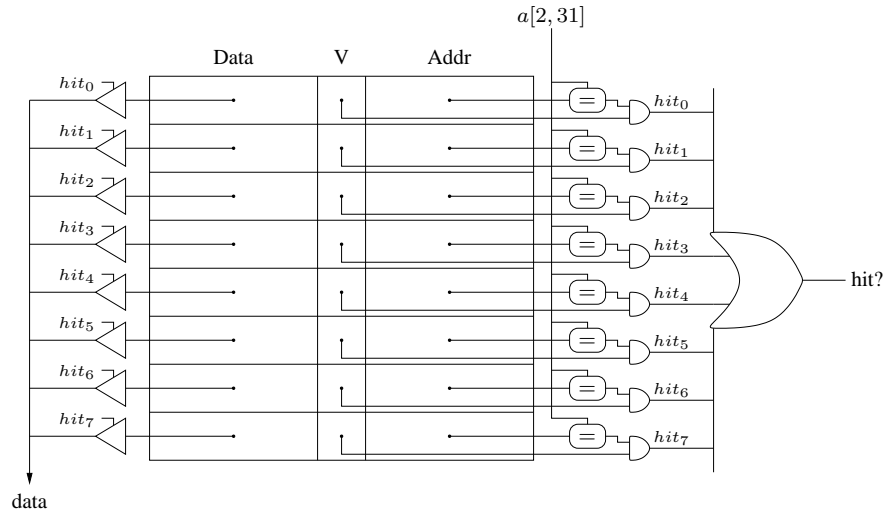


Figure 13: An example eight entry fully-associative cache.

the first time we ever touch that memory location, so it cannot possibly be in the cache yet. The value is read in from memory, and the data at address 0xC04C is stored in the cache along with the tag. Now the second memory access to 0xB36C occurs. The first step is to check the cache to see if the data we want is already around. We take a look in the cache, and find that 0xC04C is present, which is not what we want, and so we are forced to make a trip to main memory. When the value from 0xB36C returns from memory, we store it in the cache for future use, but in doing so, we overwrite, or *evict*, the data from 0xC04C. Sadly, on our next memory access, which is to 0xC04C, we'll miss in the cache because the value we wanted was just kicked out. These two values will keep bumping each other out, causing the CPU to never hit in the cache for these addresses. Two memory addresses that mapping to the same entry is called *aliasing*.

Clearly, the limitation of having each address map to only a single location can have some serious performance implications. One possible solution is to make the cache larger, so that fewer addresses are mapped to the same cache entry. Even so, pathological memory access patterns can still cause some degree of thrashing.

A second approach is to allow any memory address to map to any cache location. Figure 13 shows a cache with eight entries, where any location is free to hold any address. Now to perform a cache access, we need to check *all* of the cache entries and compare the address we want with the addresses of all cached locations. For a cache with n entries, we need n 30-bit address comparators (don't need to compare the zeros in $a[0, 1]$ since those will always match), as compared to a single $(30 - \lg n)$ -bit tag comparator for the direct mapped cache. This takes up a fair amount of area, which slows down the circuit. In the case of a cache hit, the data can not start to make its way out of the cache until we have computed which entry has hit.

This type of cache is called a *fully associative* cache, because any memory address can be associated with any cache location. The previous example of alternating accesses to 0xC04C and 0xB36C would not thrash with this cache, because the two memory locations can be placed in two separate cache entries.

After n distinct memory addresses have been accessed, the cache will become full. When the $n + 1$ st unique memory address is accessed, one of the n existing cache locations must be evicted to make room. The method used to choose an evictee is called the *replacement policy*. Ideally, we would like to choose an entry, such that when we miss on that entry, performance is not greatly impacted. One such possibility is to evict an address that is rarely used. The optimal choice is dependent on the program executing, the processor configuration, and most importantly, knowledge of the future behavior of the program. Unfortunately, the last piece of information is usually unavailable, and so optimal replacement policies are generally impossible to implement. Instead, various *heuristics* can be used. One simple policy is random replacement: simply choose one entry at random and evict it. Another more commonly used heuristic is to evict the *least recently used* (LRU) entry. This requires keeping track of the "age" of all of the

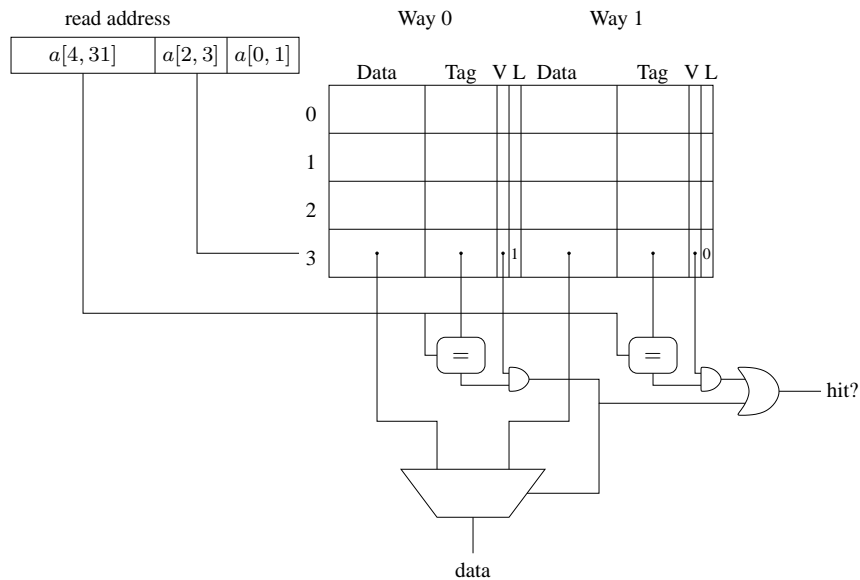


Figure 14: An example eight entry, 2-way set-associative cache.

entries, and updating the ages of all entries everytime a memory access is made. This requires a substantial amount of hardware to keep track of and update all of this state. On the other hand, LRU has been empirically shown to be a good heuristic for many applications. Because of this overhead, as well as the generally slower access times, fully associative caches tend to be much smaller than other cache organizations with similar access times.

So which cache organization should we use? Of course the answer to this depends on the situation, but in general, the direct mapped cache gives us a fast, simple solution with the possibility of thrashing and poor hit rates, while a fully associative cache is close to an “ideal” solution, but the hardware costs (in circuit area and speed) can be prohibitive. Another solution is to use a hybrid of these two. In direct mapped caches, each memory location is mapped to a single cache entry, while fully associative caches allow each memory location to be mapped to any (i.e. all) cache entries. Instead, we can allow each memory location to be mapped to only 2 cache entries. Figure 14 shows such a cache solution which is called a *2-way set associative cache*. This example only has four lines, where each line consists of two *ways*. As in previous examples, bits $a[0, 1]$ are ignored. Now, any address with $a[2, 3] = 10$ gets mapped into entry number 2, but the data may be present in either of the two ways. This means that we need two comparators to determine if we have a hit (contrast this to the n comparators needed in the fully associative cache). A final mux selects between the two ways.

Since there is more than one possible cache entry per memory location, the issue of choosing a replacement policy still needs to be addressed. With only two entries though, a LRU solution is very easy to implement. Each cache entry is simply augmented with a single bit. Each time an entry is accessed (on a hit), its LRU bit is reset, and the LRU bit in the other way is set to high (thus signifying that it is the least recently used of the two entries). When evicting, we simply choose the way with the LRU bit set high.

In general, a k -way set associative cache can be used. In a sense, a direct mapped cache is just the extreme case of a 1-way set associative cache, while a fully associative cache is a n -way set associative cache.

Writing data into the cache can present an interesting situation. When the cache is updated, it can leave the cache and memory in an incoherent state. Figure 15 shows the state of a location in cache and memory before and after a write. When the cache is updated with the new value, the two copies are now inconsistent. How we deal with this situation is called the *write policy*.

The first write policy we look at is called *write-through* (a cache that supports a write-through write policy is often called a write-through cache). The rule is simple: everytime a cache entry is written to, a write to the corresponding memory location is also sent to the main memory. The advantage of this policy is that cache and memory are always

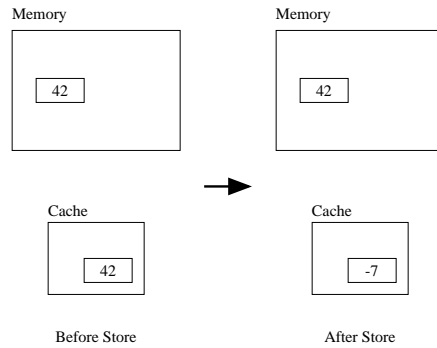


Figure 15: A write to a cache entry can leave it inconsistent with the copy in main memory.

consistent, and so the processor or memory subsystem doesn't have to do any further work to guarantee consistency. The cons are that a lot of unnecessary memory traffic can be generated, which can eat up the CPU-memory bandwidth, and cause additional contention for memory (a load instruction may be delayed because a store is busy writing through to main memory). For example, a single location may be written to 20 times before the cache entry is evicted. This generates 20 writes to main memory, but only the last one matters.

The more commonly used write policy is called *write-back*. The rule is to only send updates to the main memory when a cache line is evicted. This can reduce the amount of memory traffic when a particular memory location is written to many times, say s times, before it is evicted. With a write-through cache, this will generate s updates to main memory. With a write-back cache, the cache entry will be updated s times, but only a single update to main memory is generated when the line is finally evicted.

One approach to handling write-back caches is to always write a cache entry back into main memory when the line is evicted. But this is not always needed! It is often the case that a value is loaded from memory, used, but never stored to. In this case, the value in the cache is still the same as the copy in memory, and so there is no need to update the main memory. Instead, we can keep a *dirty bit* with each cache entry (we'll show this as a bit marked 'D' in the cache illustrations). The dirty bit specifies whether the cache entry has been modified (i.e. stored to). When a memory location is first brought into the cache from main memory, the dirty bit is cleared. On the first write to this cache entry, the bit is set, and remains set. There may be several more writes to this entry, but in any case, the dirty bit will remain high. Finally, when a cache entry is evicted, an update to main memory is only generated if the dirty bit for that entry is set. After the update to main memory has been made, the bit is cleared.

How much data do we store in a cache entry? We actually want our *cache lines* to store several consecutive memory locations. The first reason is to take advantage of spatial locality. We earlier saw how some programs often access several nearby locations, and so proactively bringing some extra memory locations can boost performance. Since the main memory system is often pipelined and due to the asymmetry in row access and column select times of DRAMs, it is faster to read in w consecutive locations than to make w separate memory requests. Figure 16 shows a direct mapped cache where each cache line stores $w = 4$ consecutive memory locations. The amount of data stored in a cache line is called the *line width*. Because all w locations are consecutive, this allows us to store $\lg w$ fewer bits in the tag, and also just a single tag for the entire line. The cache has 16 lines, and a width of 4 words, and so bits $a[8, 31]$ are used for the tag, $a[4, 7]$ are the index, $a[2, 3]$ are used to select one of the w words, and $a[0, 1]$ are zero as usual.

Cache Hierarchy

The general trend is that you can either have fast, but small caches, or large and slow memories. This is actually just one part of the overall picture. There is an entire spectrum of memory choices with varying speed/size tradeoffs. The memory systems of most modern computers are organized into a multi-level *hierarchy*. At one extreme, we have a few bytes of storage in the registers of the CPU. Locality is also necessary for making registers useful. Data is often loaded from memory into registers and then manipulated several times (temporal locality) before being stored again.

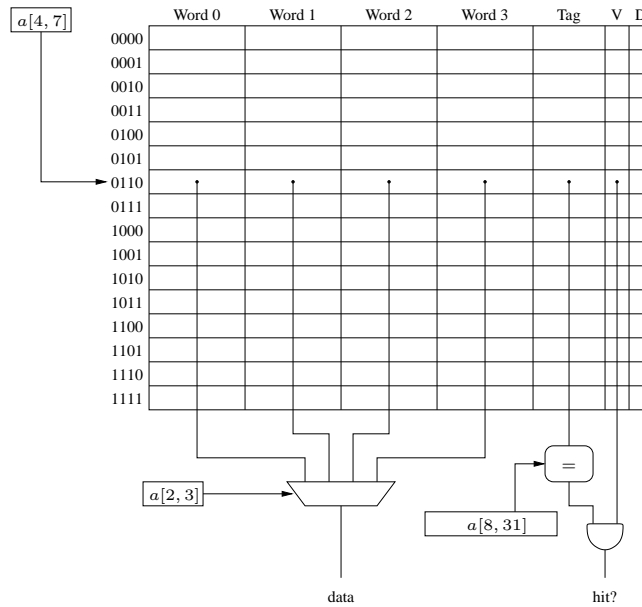


Figure 16: An example 16 line direct mapped cache with 4 word (16 byte) cache lines.

At the other extreme, we have magnetic disk drives that can store many tens or hundreds of gigabytes, but require many milliseconds to access (as opposed to fractions of nanoseconds for registers).

Figure 17 shows the different levels of a memory hierarchy. Most contemporary systems have an on-chip L1 (level one) cache which may only take a few clock cycles to access. A larger L2 cache is sometimes on chip, sometimes off chip, with a access time of anywhere from half a dozen to over a dozen cycles. On some high performance systems, there may even be a third level of cache, usually up to several megabytes in size. These levels are sometimes collectively referred to as the *cache hierarchy*. A particular system may only support a subset of these levels. Next, several megabytes to a few gigabytes of slower DRAM memory comprise the “main memory”. For 32-bit systems, there is a limit of 4GB of main memory. Beyond this is the non-volatile disk storage, which also plays an important part of the memory hierarchy.

Note that at any level of the hierarchy, each cache can have different sizes, associativities, write policies, replacement policies, etc. Another typical design is to have a Harvard styled L1 cache, and a Princeton styled organization for the rest of the hierarchy. The L1 cache is often divided into a separate L1 instruction cache and a L1 data cache. Besides the benefits that were earlier discussed with respect to Harvard organizations, there are VLSI layout reasons why separate caches make sense. Figure 18 illustrates a two-level cache hierarchy with a split L1 cache and a *unified* L2 cache. In most computer architecture literature, the names Harvard and Princeton are only used to describe (instruction set) architectures that support separate or unified memory models. Although a cache hierarchy may have separate caches for instructions and data, the view the programmer has is still one where there is only a single memory space. The split caches are usually explicitly referred to as the “L1 Data/Instruction Caches”, and the term “unified” is most often used to describe Princeton *styled* caches. The older terms are presented here for historical reasons. The instruction and data caches are sometimes abbreviated as “I\$” and “D\$” respectively (“\$” = “cash” \approx “cache”).

Cache Misses

Cache misses hamper performance in computer systems. To better understand how caches affect how fast a program will run, we will quickly look at the three reasons why a cache access might result in a cache miss.

1. *Compulsory Misses*: These are due to the fact that the data requested is being accessed for the first time, and so it is impossible to have the data in the cache at this point. These misses are also called *cold-start* misses.

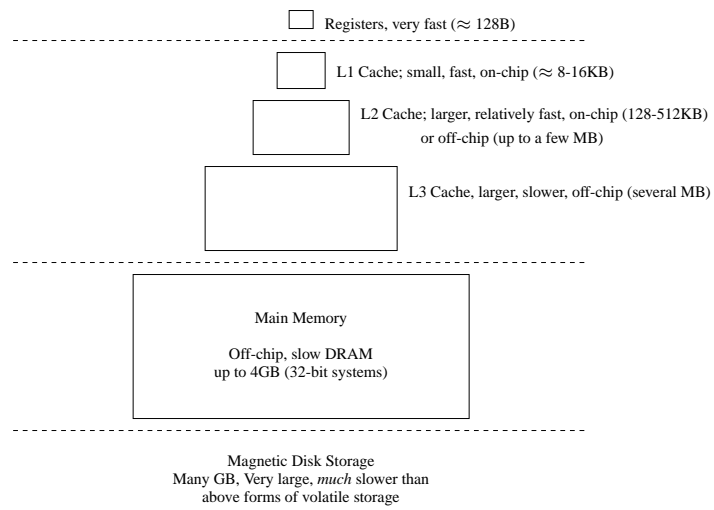


Figure 17: The different levels of the memory hierarchy.

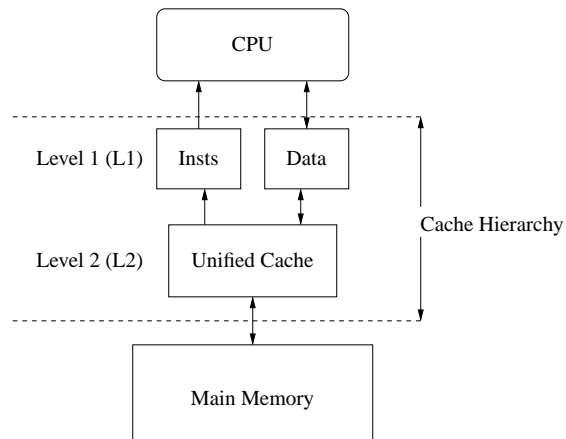


Figure 18: A two-level cache hierarchy with a split L1 cache organization and a unified L2 cache.

2. *Capacity Misses*: These are due to the cache simply not being large enough to hold all of the data the program wants.
3. *Conflict Misses*: These are due to the limited number of locations that a memory location can be mapped to in direct-mapped or set-associative caches. That is, this is a miss that is caused because a useful value was evicted when there were other locations in the cache that were available, but the memory location was not mapped to any of these cache entries. These are also called *collision* misses.

List of Concepts

This is just a summary of the terms and concepts covered this week. This can serve as a checklist to see if you understand everything that was covered.

- SRAM
- Row Decoder, Column Select
- Sense Amp
- Access Time
- Read Port, Write Port
- Multi-porting
- Interleaving/Banking
- Temporal Locality
- Spatial Locality
- Cache Hit/Miss
- Cache Tags
- Direct Mapped, Fully- and k -way Set-Associative
- Replacement policy
- Write-through, Write-back
- Valid Bit, Dirty Bit
- Instruction Cache, Data Cache, Unified Cache
- Line Size
- Memory/Cache Hierarchy
- Compulsory, Capacity and Conflict Misses