7 Caches

In the second part of this book, we shall focus on the design of the memory system. To sustain a high performance pipeline, we need a high performance memory system. Otherwise, we will not be able to realize the gains of having a high performance pipeline. It is like having a strong body and a strong mind. Unless we have a strong body, we cannot have a strong mind, and vice versa.

The most important element in the on-chip memory system is the notion of a cache that stores a subset of the memory space, and the hierarchy of caches. In this section, we assume that the reader is well aware of the basics of caches, and is also aware of the notion of virtual memory. We shall provide a very brief introduction to these topics in this section for the sake of recapitulation. However, this might be woefully insufficient for readers with no prior background. Hence, readers are requested to take a look at some basic texts such as [Sarangi, 2015] to refresh their basics.

In line with this thinking, we shall provide a very quick overview of cache design in Section 7.1 and virtual memory in Section 7.2. Then, we shall move on to discuss methods to analytically estimate the area, timing, and power consumption of caches in Section 7.3. This will give us a practical understanding of the issues involved in designing caches. We shall then extend this section to consider advanced cache design techniques in Section 7.4.

We will then proceed to look at a very unconventional design – Intel's trace cache – in Section 7.5. It is designed to store sequences of instructions, whereas conventional caches are designed to store just blocks of bytes. Storing traces gives us a lot of benefits. In most cases we can completely skip the fetch and decode stages.

The next half of the chapter focuses on using methods to improve the efficiency of the memory system by using complicated logic that resides outside the caching structures. In Sections 7.6 and 7.7, we shall focus on prefetching mechanisms where we try to predict the memory blocks that are required in the near future and try to fetch them in advance. This reduces the average memory latency. Prefetching techniques are highly effective for both instructions and data.

7.1 Memory Hierarchy and the Notion of Caches

Let us provide a quick introduction to the memory systems in today's processors. Let us start by noting that we need memory to store instructions and data. At the level of the memory system it does not matter if we are storing instructions or data. We treat everything as data, and the memory system with some exceptions is oblivious of the contents of the bytes that it stores. In fact storing instructions

as data was one of the most revolutionary ideas in the history of computing. The credit goes to early pioneers such as Alan Turing and John von Neumann.

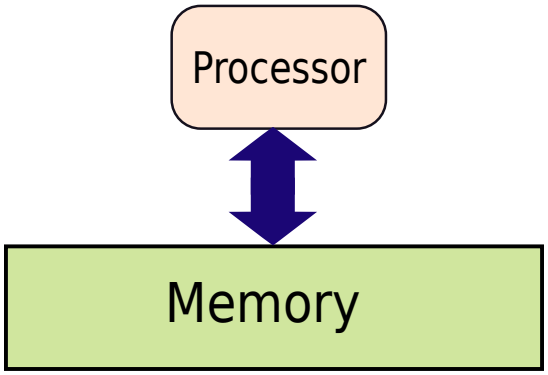

Figure 7.1: A simple processor-memory system (Von Neumann architecture)

In a simplistic model we have the processor connected to the memory system as shown in Figure 7.1: the memory system stores both instructions and data. This is the Von Neumann architecture. The main problem with this organization is that a single unified memory is too large and too slow. If every memory access takes 100s of cycles, the IPC will be less than 0.01. A ready fix to this issue is to use registers, which are named storage locations within the processor. Each register takes less than a cycle to access and this is why we use registers for most instructions. However, to keep the register file fast, we need to keep it small. Hence, we have a limited number of on-chip registers. The number is typically limited to 8 or 16.

Compilers often run out of registers while mapping variables to registers. It is thus necessary to spill the values of some registers to memory to free them. The spilled values can then be read back from memory, whenever we need to use them. Additionally, it is also necessary to store the registers to memory before calling a function. This is because the function may overwrite some registers, and their original contents will be lost. Finally, we need to restore their values once the function returns. Along with local variables in functions, most programs also use large arrays and data structures that need to be stored in the memory system. Because of multiple such reasons, memory accesses are an integral part of program execution. In fact, memory instructions account for roughly a third of all the instructions in most programs.

We have till now discussed only data accesses. However, instructions are also stored in the memory system, and every cycle we need to fetch them from the memory system. The part of the memory system that stores the instructions is traditionally known as the instruction memory. Having one single memory for both instructions and data is an inefficient solution because it needs to simultaneously provide both instructions and data. This increases the overheads significantly. Hence, a more practical method is to split the unified memory into separate instruction and data memories as shown in Figure 7.2. This is known as the Harvard architecture.

To store all the instructions and data in a modern program, we need large memories. Large memories are slow, have large area, and consume a lot of power. Another direct consequence of a large memory size is that such memories cannot be fit within the CPU. They need to reside outside the CPU (off-chip). Accessing off-chip memory (also referred to as *main memory*) for both instructions and data is simply not practical, neither feasible. In most processors, it takes 200-300 cycles to get data back from off-chip memory. This will decrease our IPC significantly.

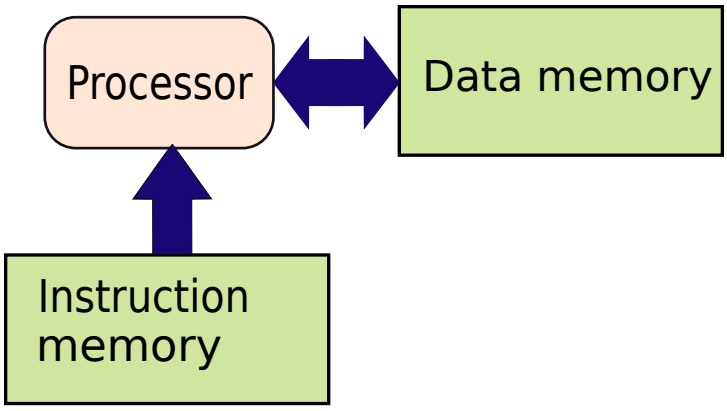

Figure 7.2: The Harvard architecture

7.1.1 Temporal and Spatial Locality

There is thus a dire need to fix this situation. Let us make use of some common patterns in programs to suggest an organization of the memory system. Most programs contain loops, and also spend most of their time executing loops. Loops are typically small pieces of code, particularly when we consider the entire program. Most programs typically move from function to function. In each function they do some intense activity within a hierarchy of loops, and then move to another function where they do the same. Such kind of patterns are very common in programming languages, and are often referred to using two very special names: *spatial locality* and *temporal locality*.

Spatial Locality

Spatial locality refers to a pattern where we access objects that are in some sense proximate (close by) in a small interval of time. Before looking at deeper aspects of this definition, let us explain with an example. Consider the instructions in a loop. In terms of PC addresses, we access instructions that have addresses that are close by. Thus, we have spatial locality. Similarly, when we are accessing an array, we also have spatial locality if we are accessing it sequentially – from indices 0 to N. Spatial locality is an inherent property of most programs that we use in our everyday life. As a result, most computer architects take spatial locality in programs for granted.

Note that there is some vagueness in the definition. We have not precisely defined what exactly we mean by "a small interval of time". Is it in nanoseconds, microseconds, or hours? This is subjective, and depends on the scenario.

Consider a loop. If in every iteration of a loop we access an array location in sequence, we have spatial locality because of the way we are accessing the array. Now assume that we take the same loop and in every iteration we insert a function call that takes a few thousand cycles to complete. Do we still have spatial locality? The answer is, no. This is because it is true that we are accessing nearby addresses in the array; however, this is being done across thousands of cycles. This is by no means a small interval of time as compared to the time it takes to access a single location in an array. Hence, we do not have spatial locality.

Let us further augment the code of the loop to include an access to the hard disk that takes a million cycles. Let the hard disk accesses be to consecutive sectors (blocks of 512 bytes on the disk). Do we have spatial locality? We do not have spatial locality for the array accesses; however, we do have spatial locality for the disk accesses. This is because in the time scale of a disk access (a few million cycles), the instructions of the loop qualify as a "small interval of time". The summary of this discussion is that we need to deduce spatial locality on a case by case basis. The accesses that we are considering to be

"spatially local" should occupy a non-trivial fraction of the interval under consideration.

A related concept is the notion of the *working set*. It is defined as the set of memory addresses that a program accesses repeatedly in a short time interval. Here again, the definition of *short* is on the same lines as the definition for spatial locality – there is a degree of subjectivity. This subjectivity can be reduced if we realize that program execution can typically be divided into phases: in each phase a program executes the same piece of code and accesses the same region of data over and over again, and then moves to another region of the code – the next phase begins. We typically have spatial locality for accesses within a phase, and the set of addresses accessed repeatedly in a phase comprise the working set at that point of time.

Definition 38
We can divide program execution into phases, where each phase has a distinct pattern in terms of instruction and data accesses. Within a phase, we typically access data that is proximate in terms of memory addresses – this is known as spatial locality. The set of addresses accessed repeatedly in a phase comprises the working set of the program at that point of time.

Temporal Locality

Let us consider a program with loops once again. There are some variables and regions of memory that we tend to access frequently in a small interval of time. For example, in a loop we access the loop variables frequently, and also we execute the instructions in the loop in every iteration. Even while walking through an array we access the base register of the array on every access. Such patterns, where we keep accessing the same memory locations over and over again is referred to as *temporal locality*. Note that temporal locality has been found to be a general property in most programs. Temporal locality is observed while accessing instructions or data. In fact, we can see temporal locality in almost all memory and storage structures inside the chip.

Most schemes in computer architecture are designed to make use of temporal locality. For example, a branch predictor uses this fact to keep a small table of saturating counters. The expectation is that the hit rate (probability of finding an entry) in this table will be high; this is guaranteed by temporal locality. The branch target buffer operates on a similar principle. Even predictors such as value predictors or dependence predictors rely on the same phenomenon. Had we not had temporal locality, most of our architectural structures would have never come into being. In the case of the memory system as well, we shall explicitly rely on temporal locality in our memory access patterns.

Definition 39
Temporal locality refers to an access pattern where we repeatedly access the same locations over and over again in a small interval of time.

7.1.2 Notion of a Cache

We now know that a typical memory access stream has both temporal locality and spatial locality. We also know that we cannot afford large memories. What do we do?

We can take inspiration from registers, and create a structure that can be conceptually thought of as a larger register file. This structure can keep the data corresponding to the most frequently accessed memory addresses close to the processor. It will also be small, fast, and power efficient. Since we have temporal locality, we can expect to find most of our data in this structure. Let us call such a structure

a *cache*. Formally, a cache is a memory structure that stores a subset of the memory values used by a program. We can also say that a cache stores a subset of the program's address space, where the address space is defined as the set of addresses that a program uses. Finally, note that this is a non-contiguous subset. For example, we can have addresses 8, 100, and 400 in the cache, and not have a lot of intervening addresses. The bottom line is that caches store frequently accessed data and instructions. Hence, due to temporal locality, we are guaranteed to see a high cache hit rate (probability of successfully finding a value).

Definition 40
If we find a value in the cache, then this event is known as a cache hit*, otherwise it is known as a cache* miss*.*

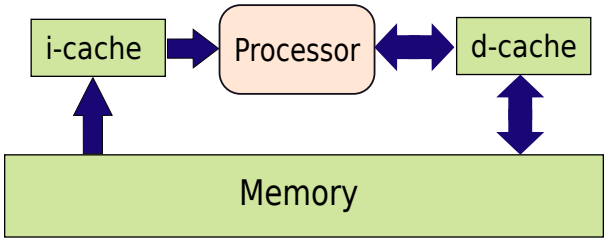

Figure 7.3: The i-cache and d-cache

Taking inspiration from the Harvard and Von Neumann architectures, we arrive at the design in Figure 7.3, where the pipeline reads in data from the instruction cache (i-cache), and reads or writes data to the data cache (d-cache). These are small caches, which as of 2020 are from 16 KB to 64 KB in size. They are referred to as the level 1 or L1 caches. Often when people use the word "L1 cache" they refer to the data cache. We shall sometimes use this terminology. The usage will be clear from the context.

Observe that in Figure 7.3, we have the processor, two caches, and a combined memory that stores instructions and data: we have successfully combined the Harvard and Von Neumann paradigms. The access protocol is as follows. For both instructions and data, we first access the respective caches. If we find the instruction or data bytes, then we use them. This event is known as a *cache hit*. Otherwise, we have a *cache miss*. In this case, we go down to the lower level, which is a large memory that contains all the code and data used by the program. It is guaranteed to contain everything (all code and data bytes). Recall that we store instructions as data in structures that store both kinds of information. Finally, note that this is a very simplistic picture. We shall keep on refining it, and adding more detail in the subsequent sections.

Up till now we have not taken advantage of spatial locality. Let us take advantage of it by grouping bytes into blocks. Instead of operating on small groups of bytes at a time, let us instead create blocks of 32 or 64 bytes. Blocks are atomic units in a cache. We always fetch or evict an entire block in one go – not in smaller units. One advantage of this is that we automatically leverage spatial locality. Let's say we are accessing an array, and the array elements are stored in contiguous memory locations as is most often the case. We access the first element, which is 4 bytes wide. If the block containing the element is not in the cache, then we fetch the entire block from the memory to the cache. If this element was at the beginning of the 32-byte block, then the remaining 28 bytes are automatically fetched because they are a part of the same block. This means that for the next 7 accesses (28 bytes = 7 accesses * 4 bytes/access), we have the elements in the cache. They can be quickly accessed. In this case, by creating

blocks, we have taken advantage of spatial locality, and consequently reduced the time it takes to access the array elements.

Way Point 7
We now know that memory access patterns exhibit both temporal and spatial locality. Most modern memory systems take advantage of these properties. This is done as follows:

- *We create a small structure called a cache that stores the values corresponding to a subset of the memory addresses used by the program. Because of temporal locality, we are guaranteed to find our data or instructions in the caches most of the time. Two such structures that most processors typically use are the instruction cache (i-cache) and the data cache (d-cache).*
- *To take advantage of spatial locality, we group consecutive bytes into blocks. Furthermore, we treat blocks atomically, and fetch or evict data at the granularity of blocks within the memory system. The advantage of fetching 32-64 bytes at once is conspicuously visible when we are accessing a sequence of contiguous instructions or accessing an array. If we read a memory word (4 bytes), then with a very high probability we shall find the next memory word in the same block. Since the block has already been fetched from memory, and kept in the cache, the access time for other memory words in the same block will get reduced significantly. In other words, if our access pattern has spatial locality, then we will find many memory words in the cache because other words in the same blocks would have already been fetched. This will reduce our overall memory access time.*

7.1.3 Hierarchy of Caches

Till now, we have discussed the need and use of an i-cache and a d-cache. However, we can stretch our line of argument further and have another larger cache between them and the off-chip main memory. This cache can contain both data as well as instructions – at this level we do not differentiate between data and instructions, they are both treated as data. This is typically known as the L2 (level 2) cache, and as of 2020, it is roughly between 256 KB and 8 MB. The L2 cache is larger and slower than L1 caches.

The access protocol is as follows. The processor first accesses the L1 caches (i-cache and d-cache). Because of temporal locality, we expect most of the accesses to be hits. Just in case we miss in the L1 caches, then we need to fetch the data from the lower level in the memory system, which will be an L2 cache. The L1 caches are typically very fast, and often provide the data in 1-3 clock cycles. In comparison, L2 caches are much slower and can take anywhere from 10-50 cycles depending on how much time it takes to send a message to the cache, execute the operation, and get the data back. However, this is still better than main memory that typically takes 200-400 cycles to access. Owing to temporal and spatial locality, we expect a good hit rate at this level as well.

Most embedded and laptop processors do not have any more levels in the cache hierarchy. If there is a miss in the L2 cache, then they send the request to the main memory. However, in most modern processors that are used in servers and high-end desktops, we have an additional L3 (level 3) cache, which is much larger (2-16 MB), and consequently slower as well (access time: 20-80 cycles). Because of similar reasons, the L3 level can particularly be beneficial if we are accessing a large amount of data. As of 2020 some servers have started including an L4 cache as well.

We thus have a memory system that consists of a hierarchy of caches as shown in Figure 7.4.

It is important to mention at this stage that the caches are inclusive. This means that if a block exists in the L1 cache, it has to be present in the caches at all the lower levels: L2, L3, and main memory. This property is known as *inclusion*, and caches with this property are known as inclusive caches. The

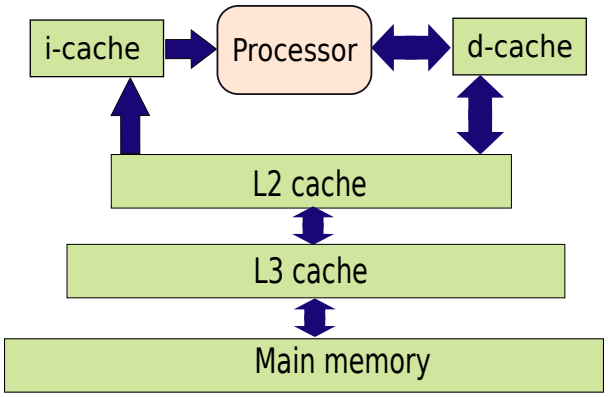

Figure 7.4: Hierarchy of caches

same holds true for the L2 cache as well. If we have a block in the L2 cache, then it has to be present in the L3 cache and main memory.

Inclusion simplifies things significantly. If we evict a block in the L1 cache, then the changes can be seamlessly written to the possibly older copy of the block present in the L2 cache. However, if the block is not present in the L2 cache, and we have modified its contents, the situation becomes tricky. We need to search in the L3 cache, and main memory to find if they contain the block. This increases the complexity of the cache logic significantly. However, inclusion has its costs. If the processor is fetching a block for the first time, then it needs to come from main memory and be stored in each of the levels of the cache hierarchy.

Example 3
A natural question that might arise is why do we stop at 2 or 3 cache levels. Why do we not have 7 or 8 levels of caches?

Answer: *Caches are not particularly free. They have costs in terms of the transistor area. Given that we cannot synthesize very large silicon dies, we have to limit the number of transistors that we place on the chip. The typical silicon die size is about 200 mm^2 for desktop processors and 400-500 mm^2 for server class processors. In larger dies, we can place more transistors; however, we cannot place enough to create additional caching levels.*

Also note that as we go to lower and lower cache levels the miss rates typically become high and saturate. The incremental benefit of having more cache levels goes down.

Finally, additional layers of caches introduce additional overheads in terms of the area and power consumed by the caches. Moreover, the miss penalty increases for a block that is being accessed for the first time, because now the request has to pass through multiple caches.

7.1.4 Organization of a Cache

Let us provide an overview of how a cache is organized. A cache stores a subset of the blocks used in the memory system. For example, if a program requires 10,000 blocks, we may store 1024 blocks in the cache. Their addresses are not necessarily contiguous. Given a block address, we need to find out if it exists within a cache or not.

Let us consider a running example. Assume a 32-bit memory system (memory addresses are 32 bits)

for the sake of simplicity. We want to design a 64 KB cache with 64-byte blocks. Let us first separate the block address from the memory address. Given that a block consists of consecutive bytes, we can easily conclude that the last 6 bits of the memory address give the address of the byte within the block. This is because $2^6 = 64$, and thus we require 6 bits to uniquely index a byte within a block. This is shown in Figure 7.5, where we divide a 32-bit memory address into two parts: 26 bits for the block address, and 6 bits for the offset (required for uniquely indexing a byte within a block).

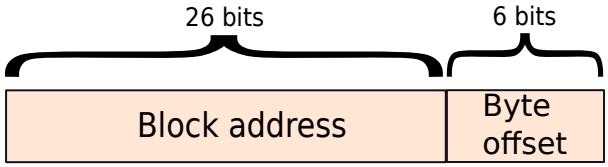

Figure 7.5: Splitting a memory address into two parts: block address and byte offset

Let us now consider the 26-bit block address, and the fact that we have 1024 blocks in the cache (64 KB / 64 B). We need to search these 1024 entries, and find out if the given block address is present in the cache. Conceptually, we can think of this cache as a 1024-entry array. Each entry is also known as a *cache line*. Typically, the terms "cache line" and "cache block" are used synonymously and interchangeably. However, we shall use the term "cache line" to refer to the entire entry in the cache and the term "cache block" to refer to the actual, usable contents: 64 bytes in the case of our running example. A cache line thus contains the cache block along with some additional information. Typically, this subtle distinction does not matter in most cases; nevertheless, if there are two terms, it is a wise idea to precisely define them and use them carefully.

Now, we are clearly not going to search all 1024 entries for a given block address. This is too slow, and too inefficient. Let us take a cue from a course in computer algorithms, and design a method based on the well known technique called *hashing* [Cormen et al., 2009]. Hashing is a technique where we map a large set of numbers to a much smaller set of numbers. This is a many-to-one mapping. Here, we need to map a 26-bit space of block addresses to a 10-bit space of cache lines ($1024 = 2^{10}$).

The simplest solution is to extract the 10 LSB (least significant) bits from the block address, and use them to access the corresponding entry in the cache, which we are currently assuming to be a simple one-dimensional table. Each entry is identified by its row number: 0 to 1023. This is a very fast scheme, and is very efficient. Extracting the 10 LSB bits, and on the basis of that accessing a hardware table is a very quick and power efficient operation. Recall that this is similar to how we were accessing the branch predictor.

However, there is a problem. We can have aliasing, which means that two block addresses can map to the same entry. We need to have a method to disambiguate this process. Furthermore, we can do what was suggested way back in Section 3.2, which is to store some additional information with each entry. Consequently, we can divide a 32-bit address into three parts: 16-bit tag, 10-bit index, and 6-bit offset.

The 10-bit index is used to access the corresponding line in the cache, which for us is a one-dimensional table (or an array). The 6-bit offset will be used to fetch the byte within the block. And, finally the 16-bit tag will be used to uniquely identify a block. Even if two separate blocks have the last 10 bits of the block address (index) in common, they will have different tags. Otherwise, the block addresses are the same, and the blocks are not different (all 26 bits of the block address are common). This is pictorially shown in Figure 7.6.

Let us explain this differently. Out of a 32-bit memory address, the upper 26 bits (more significant) comprise the block address. Out of this, the lower 10 bits form the index that we use to access our cache. The upper 16 bits can thus vary between two block addresses that map to the same line in the cache. However, if this information is also stored along with each line in the cache, then while accessing the cache we can compare these 16 bits with the tag part of the memory address, and decide if we have

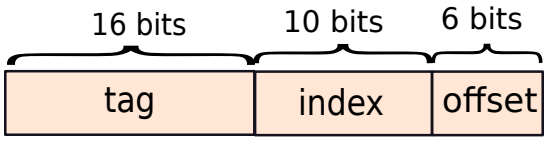

Figure 7.6: The concept of the tag

a cache hit or miss. We are thus comparing and using all the information that is present in a memory address. There is no question of aliasing here. Figure 7.7 explains this concept graphically.

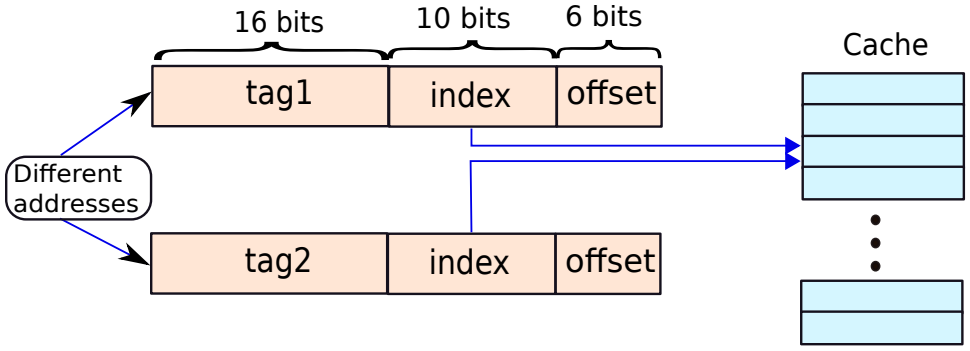

Figure 7.7: The concept of the tag explained along with aliasing

Let us now take a look at one of the simplest cache designs that uses this concept.

Direct Mapped Cache

A direct mapped cache contains two arrays: a tag array and a data array. The tag array in the case of our running example stores 1024 tags, and the data array stores the contents of 1024 blocks as shown in Figure 7.8.

In the case of a direct mapped cache, we index the tag and data array bits using the n least significant bits of the block address, where 2^n is equal to the number of entries in either array. Thus, in this case, we extract the 10 least significant bits of the block address, and use them to access both the arrays. Assume that we want to access the cache and fetch the block, whose block address is A. We proceed as discussed by extracting the 10 LSB bits, and use them to index the data and tag arrays. Assume that the entry that is present in the arrays is for a block address A'. Now, if $A = A'$, we have a hit, otherwise we have a miss.

Let us summarize the process with a set of bullet points. Assume that the function $tag(A)$ provides the tag part (upper 16 bits in this case) of the block address. We will use the block addresses A and A' (as defined in the previous paragraph) in the following description.

- Every byte is uniquely identified by a 32-bit memory address.
- Out of these 32 bits, we have taken out the 6 least significant bits because they specify the offset of a byte within a block. We are left with 26 bits.
- Out of these 26 bits, we have used the 10 least significant bits to index the tag and data arrays. By this point, we have used 16 bits out of the 32-bit address. This means that between addresses A and A', we have the lower 16 bits in common. We are not sure of the upper (more significant) 16 bits. If they match, then the addresses are the same.

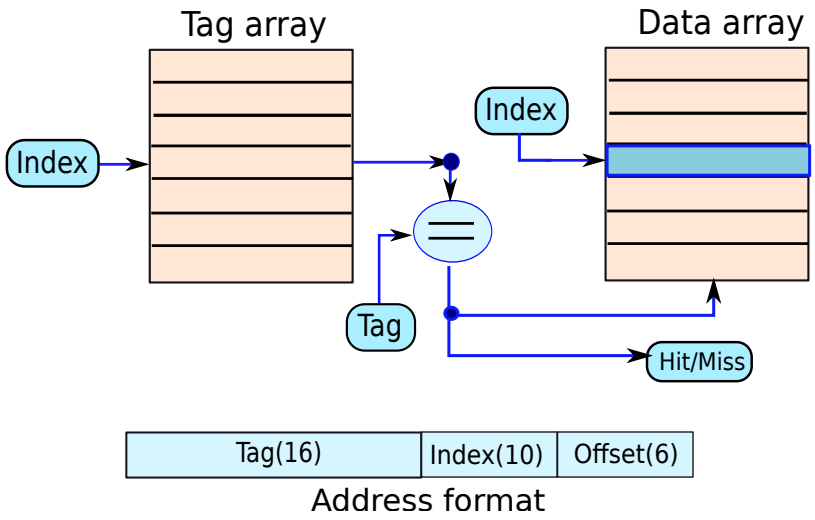

Figure 7.8: A direct mapped cache

- The upper 16 bits are the tag part of the address. Thus, the problem gets reduced to a simpler problem: check if $tag(A) = tag(A')$.
- The upper 16 bits of A' are stored in the tag array. We need to compare them with the upper 16 bits of A, which is $tag(A)$. These bits can be easily extracted from the address A and sent to a comparator for comparison, as shown in Figure 7.8.

Thus, accessing a direct mapped cache is per se rather simple. We access the data and tag arrays in parallel. Simultaneously, we compute the tag part of the address. If there is a tag match – the tag part of the address matches the corresponding contents of the tag array – then we declare a hit, and use the contents of the corresponding entry in the data array. Otherwise, we declare a miss.

The logic for having separate tag and data arrays will be clear in Section 7.3. Let us proceed to look at other variants of caches.

Fully Associative Cache

A direct mapped cache is a good idea. However, it suffers from a major flaw. Assume we access two memory locations very frequently, and their corresponding addresses map to the same entry in the tag and data arrays. The hit rate in the case of direct mapped caches will be very low, because these accesses will constantly displace each other's data. Let us instead design a cache where an entry can reside anywhere in the cache. This means that for our running example, we have 1024 possible locations for a given block. Performance considerations aside, this can completely mitigate the problem of aliasing. In theory, we can have 1024 blocks that map to the same location reside in the cache without causing any misses.

The idea definitely does look very appealing; however, the devil is in the details. We need to appreciate that we never get anything for free in life, and this rule is particularly important in computer architecture.

We shall discuss the details of this design in Section 7.3 including a description of the hardware. In this section, we shall offer a superficial treatment of this area.

As we shall see in Section 7.3, such caches use a different kind of basic memory cell known as the CAM (content addressable memory) cell. It is typically made of 10 transistors. In comparison, regular

SRAM (static RAM) cells have 6 transistors. The main advantage of CAM cells is that it is possible to compare a set of bits pairwise with the bits stored in an array of CAM cells. If all of them match, then the match line in Figure 7.9 is set to 1.

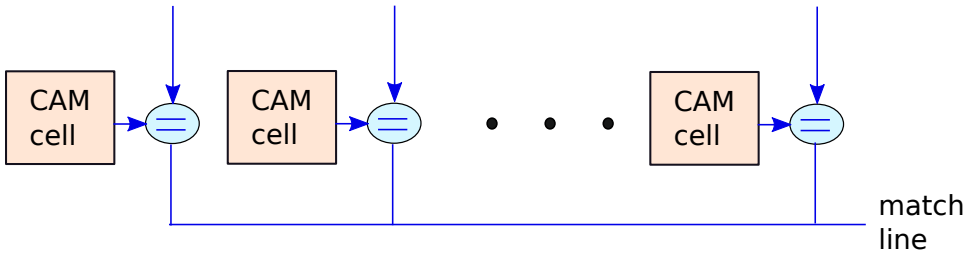

Figure 7.9: A set of CAM cells. The match line is a wired-AND bus (computes a logical AND of all the inputs).

Using such CAM cells, we can create a content addressable memory (CAM memory). This term deserves some more explanation. In a normal memory such as the data and tag arrays of a direct mapped cache, we access each entry based on the index. The index is the number of the entry. It starts from 0, and monotonically increases by 1 for each subsequent entry. This is similar to an array in programming languages. However, a CAM array or CAM memory is more like a hash table in C++ or Java. We typically access an entry based on its contents. Let us explain with an example. Consider an array of numbers: *vals[] = {5, 6, 2, 10, 3, 1 }*. The expression *vals[3]* refers to accessing the array *vals* by its index, which is 3. In this case, the result of this expression is 10.

However, we can also access the array by the contents of an array element, and get the index if the array contains the value. For example, we can issue the statement *get(vals, 10)*. In this case, the answer will be 3 because the value 10 exists at the array index 3 (we start counting from 0). CAM arrays can be used to see if a given value exists within an array, and for finding the index of the row that contains the value. If there are multiple copies of the same value, then we can either return the lowest index, or any of the indices at random. The behavior in such cases is often undefined.

Now let us use the CAM array to build a cache (refer to Figure 7.10). We show a simple example with only 4 entries (can be scaled for larger designs). For our problem at hand, the CAM array is the best structure to create the tag array. The input is the tag part of the memory address; we need to quickly search if it is contained within any row of the tag array. In this case, we do not have an index. Instead, the address is split into two parts: 6-bit offset and 26-bit tag. The tag is large because we are not dedicating any bits to index the tag array. The output will be the index of the entry that contains the same tag or a miss signal (tag not present). Once we get the index, we can access the data array with that index, and fetch the contents of the block. Note that there is a one-to-one correspondence between the entries of the tag array and the data array.

The main technological innovation that allowed us to build this cache, which is called a *fully associative cache*, is the CAM array. It allows for a quick comparison with the contents of each entry, and thus we gain the flexibility of storing an entry anywhere in the array. Even though such an approach can reduce the miss rate by taking care of aliasing, it has its share of pitfalls.

The CAM cell is large and slow. In addition, the process of comparing with each entry in the CAM array is extremely inefficient when it comes to power. This approach is not scalable, and it is often very difficult to construct CAM arrays of any practical significance beyond 64 entries. It is a very good structure when we do not have many entries (≤ 64).

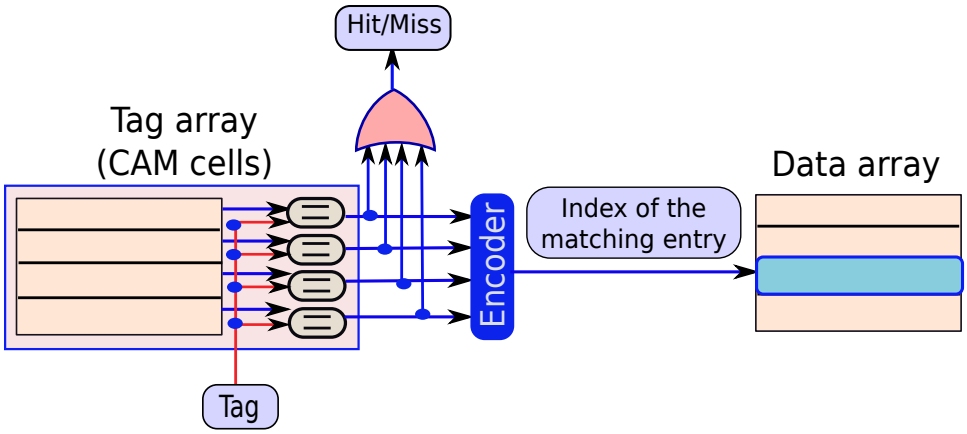

Figure 7.10: Fully associative cache

Set Associative Cache

We have seen a direct mapped cache and a fully associative cache. The former has a low access time, and the latter has a low miss rate at the cost of access time and power. Is a compromise possible?

Indeed, it is possible. Such a design is called a *set associative cache*. Here, we offer limited flexibility, and do not significantly compromise on the access time and power. The idea is as follows. Instead of offering the kind of flexibility that the fully associative cache provides, let us restrict a given block address to a few locations instead of all the locations. For example, we can restrict a given address to 2 or 4 locations in the tag and data arrays.

Let us proceed as follows. Let us divide the tag array into groups of equal sized sets. Sets often contain 2, 4, or 8 blocks. Let us implement our running example by creating a 4-way set associative cache, where each set contains 4 entries. In general, if a set contains k entries, we call it a k-way set associative cache. Each entry in the set is called a *way*. In other words, a way corresponds to an entry in the tag array and data array that belongs to a given set. In a k-way set associative cache we have k ways per set.

Definition 41
If a set contains k entries, we call it a k-way set associative cache. Each entry in the set is called a way.

Recall that we needed to create a 64 KB cache with a 64-byte block size in a 32-bit memory system. The number of blocks in the data array and the number of entries in the tag array is equal to 1024 (64 KB/ 64 B). As we had done with direct mapped caches, we can split the 32-bit memory address into a 6-bit offset and 26-bit block address.

Let us now perform a different kind of calculation. Each set has 4 entries. We thus have $1024/4 = 256$ sets. The number of bits required to index each set is $log_2(256)$, which is 8. We can thus use the 8 LSB bits of the block address to find the number or index of the set. Once we get the number of the set we can access all the entries in the set. We can trivially map the entries in the tag array to sets as follows. For a 4-way set associative cache, we can assume that entries 0 to 3 belong to set 1, entries 4 to 7 belong to set 2, and so on. To find the starting entry of each set, we just need to multiply the set index by 4, which is very easy to do in binary (left shift by 2 positions).

Till now, we have used 6 bits for the offset within the block and 8 bits for the set id. We are left with 18 (32 - 14) bits. This is the part of the address that is not common to different addresses that map to the same set, and thus by definition it is the *tag*. The size of the tag is thus 18 bits. The breakup of an address for a 4-way set associative cache in our running example is shown in Figure 7.11.

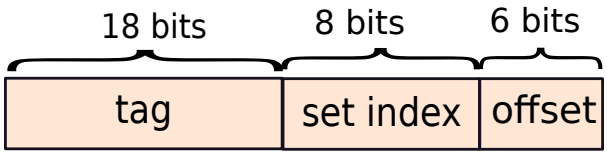

Figure 7.11: Breakup of the address in a 4-way set associative cache

To access (read/write) a data block, we first send the address to the tag array. We compute the set index, and access all the entries belonging to that set in parallel. In this case, we read 4 tags (corresponding to each way) from the tag array, and compare each of them with the tag part of the address using a set of comparators. If there is no match, then we have a miss, otherwise we have a hit. If the tags match, then we can easily compute the index in the tag array that contains the matching tag with an encoder. Refer to Figure 7.12 for a representative design of a set associative cache.

We subsequently index the data array with the index and read out the data block. Sometimes for the sake of efficiency we can read out the contents of the 4 corresponding data blocks in parallel. After computing the tag match, we can choose one of the blocks as the final output. This is faster because it creates an overlap between accessing the tag array and reading the data blocks; however, it is inefficient in terms of power because it involves extra work.

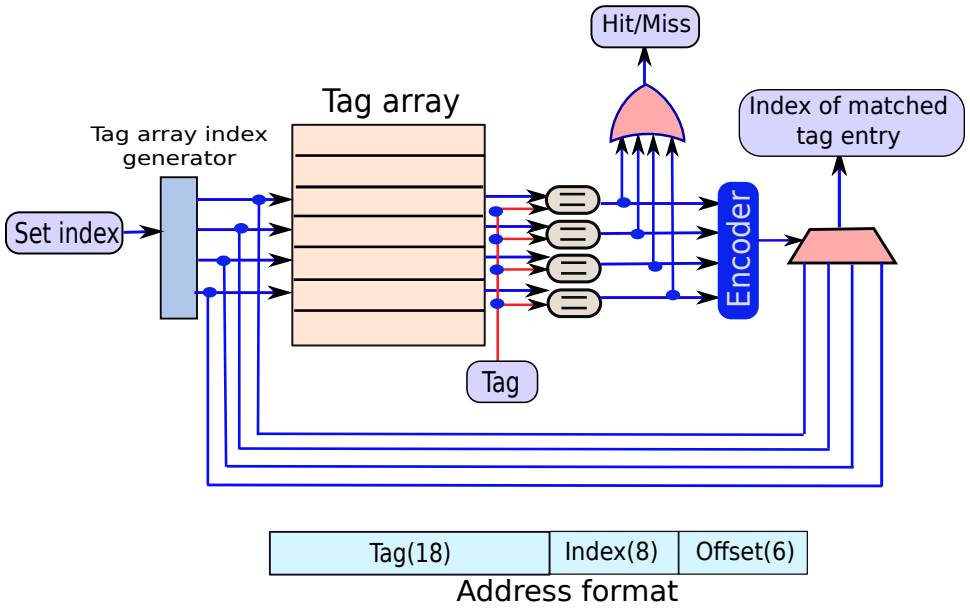

Figure 7.12: A set associative cache

Important Point 11
If we think about it, a direct mapped cache is also a set associative cache. Here, the size of each set is 1. A fully associative cache is also a set associative cache, where the size of the set is equal to the number of blocks in the cache.

The set associative cache represents an equitable trade-off between direct mapped and fully associative caches. It is not as slow as fully associative caches, because it still uses the faster SRAM cells. Additionally, its miss rate is not as low as direct mapped caches. This is because there is some degree of immunity against aliasing. If two addresses conflict (map to the same block) in a direct mapped cache, they can be placed in different ways of a set in a set associative cache. In many such cases, we can avoid misses altogether as compared to a direct mapped cache.

7.1.5 Basic Operations in a Cache

Let us quickly look at the basic operations that a cache supports. The most basic primitive that the cache needs to support is the lookup operation. The lookup operation tells us if the cache contains the block corresponding to a given memory address or not. If it does, then it returns a hit/miss signal and the location of the block if it exists in the cache. Subsequently, we can either read or modify the contents of the block.

Let us consider the situation in which a lookup operation indicates that we have a cache miss. In this case, we need to fetch the contents of the block from the lower levels of the memory system. For example, if we have a cache miss in the L1 cache, then we will try to fetch the block from the L2 cache. If we do not find the block in the L2 cache, then we need to fetch it from an even lower level such as the L3 cache (if it exists) or the main memory. After we fetch the contents of the block, we need to write it to all the levels of the memory system that did not have a copy of it. This is because we wish to support the property of inclusiveness in a cache.

Subsequently, we can read and write to the block in the L1 cache. Reading is a very easy operation. Writing is tricky. Let us thus look at the process of writing in some more detail. After completing a store operation, the L1 cache will have the most up-to-date copy of the block. The other caches such as L2 and L3 will have possibly *stale* copies of the block. Nevertheless, to support the property of inclusive caches we need to maintain such stale copies. We shall discuss this issue in some more detail in Section 7.1.5.

Important Point 12
Why is it necessary to first fetch a block to the L1 cache before writing to it?

Answer: *A block is typically 32 or 64 bytes long. However, most of the time we write 4 bytes or 8 bytes at a time. If the block is already present in the cache, then there is no problem. However, if there is a cache miss, then the question that we need to answer is, "Do we wait for the entire contents of the block to arrive from the lower levels of the memory hierarchy?" A naive and incorrect way of doing so will be to go ahead and write the data at the appropriate positions within an empty cache block, even though the rest of the contents of the actual block are not present in the cache. However, this method is fraught with difficulties. We need to keep track of the bytes within a block that we have updated. Later on, these bytes have to be merged with the rest of the contents of the block (after they arrive). The process of merging is complicated because some bytes would have been updated, and the rest would be the same. We need to maintain information at the byte level with regard to whether a given byte has been updated or not. Keeping these complexities in mind, we treat a block as an atomic unit, and do not keep track of any information at the intra-block level. Hence, before writing to a block, we ensure that it is present in the cache first.*

Replacement and Eviction

Consider a set associative cache with k ways. It is possible that $k+1$ block addresses might map to the set. In this case, we cannot store all of them in the set. Whenever we need to add a block to the set, one of the blocks in the set has to be evicted (thrown out) to make room for the new block.

We have k possible options. Any of these blocks may be *replaced* with the new block. There are many algorithms to choose the best candidate for replacement. The most effective algorithm is typically a variant of the classic LRU (least recently used) algorithm. Here we try to find the block that has been used the least in the recent past. Based on this we predict that the probability of this block being used in the future is the least among all the blocks in the set. Implementing an accurate LRU algorithm is impractical. Most systems thus implement a pseudo-LRU algorithm. In this algorithm, we associate a saturating counter with each way in the set. Every time we access the way, we increment the counter. This gives us a measure of how many times the block has been used.

This is sadly not enough because over time the counters will tend to saturate, and all the blocks within a set will hold the maximum count. To ensure that this does not happen, and we capture recent history, we need to periodically decrement the values of all the counters such that the counts of unused blocks move towards 0. We can thus deduce that higher the count, higher is the probability of accesses in the recent past. The least recently used block is expected to have the least count. Note that periodically decrementing counters is absolutely necessary, otherwise we cannot maintain information about the recent past.

Since this approach approximates the LRU scheme, it does not always identify the least recently used block; however, in all cases it identifies one of the least used blocks in the recent past. This is a good candidate for replacement.

To replace a block, we evict the block that has the least count, and in its place we bring in the new block. The process of eviction is not easy, and there are various complexities. Let us quickly understand the trade-offs between different approaches.

Write-Through and Write-Back Schemes

There are two paradigms: write-through and write-back. In a write-through cache, whenever we perform a write operation, we write the data to the lower level immediately. Because our memory system is inclusive, we are guaranteed to find the block in the lower level as well. Also recall that we never write to all the bytes of a typical 64-byte block together. Instead, we write at a much finer granularity – 2 to 8 bytes at a time. These small writes are sent to the lower level cache, where the corresponding block is updated. Thus, we never have stale copies while using a write-through cache. In this case, eviction is very easy. We do not have the concept of fresh and stale copies. Since all the writes have been forwarded to the lower level, the block is up-to-date in the lower level cache. We can thus seamlessly evict the block and not do anything more.

However, a write-back cache is more complicated: here we do not send writes automatically to the lower level. We keep a bit known as the *modified bit* in each cache line that indicates if the line has been *modified*. If it is modified, then we need to perform additional steps, when this line is going to be evicted. Note that in this case, the write-back cache will contain the up-to-date copy of the data, whereas the cache at the immediately lower level will contain a stale copy because it has not received any updates. Hence, whenever we observe the modified bit to be 1 during eviction, we write the block to the lower level. This ensures that the lower level cache contains all the modifications made to the contents of the block. However, if the modified bit is 0, then it means that the block has not been written to. There is thus no need to write back the block to the lower level. It can be seamlessly evicted.

The write-back cache basically defers writing back the block till the point of eviction. In comparison, a write-through cache writes back the block every time it is written. In a system with high temporal locality and a lot of writes, a write-through cache is clearly very inefficient. First, because of temporal locality we expect the same block to be accessed over and over again. Additionally, because of the high write traffic in some workloads, we will need to write a lot of blocks back to the lower level – every

time they are modified at the upper level. These writes are unnecessary and can be avoided if we use a write-back cache. However, on the flip side, write-through caches are simple and support seamless eviction.

There is one more subtle advantage of write-through caches. Assume we have a three level cache hierarchy. Because of the property of inclusiveness, we will have three entries for a given block in all the three caches: L1, L2, and L3. Now, assume that there is an eviction in L3, or an I/O device wishes to write to the block. Many I/O devices write directly to main memory, and not to the caches. In this case, it is necessary to evict the block from all three caches. In the case of write-through caches, this is simple. The blocks can be seamlessly evicted. However, in the case of a write-back cache, we need to check the modified bits at each level, and perform a write back to main memory if necessary. This has high overheads.

Given the nature of the requirements, we need to make a judicious choice.

7.1.6 Mathematical Analysis

Let us compute the formula for the average memory access time (AMAT) of a given memory request. Let us start with the L1 cache. It is equal to:

$$AMAT = L1_{hit\_time} + L1_{miss\_rate} \times L1_{miss\_penalty} \tag{7.1}$$

Irrespective of a hit or a miss, we need to spend some time in accessing the cache. If we get back the data, we declare a hit, else we declare a miss. We can consider this to be the $L1_{hit\_time}$. Note that this is a simplistic model. We shall see in Section 7.3 that the equations can get more elaborate. However, for the sake of a simplistic model, this is sufficient. If we have a miss with a probability equal to $L1_{miss\_rate}$, then we need to pay the miss penalty in terms of time, which is denoted by $L1_{miss\_penalty}$. This is a fairly simple equation that requires little explanation. It can be directly derived from the formula of the expected value of a random variable.

Let us now compute $L1_{miss\_penalty}$. This is nothing but the average memory access time for the L2 cache. This is because once we have a miss in the L1 cache, we go to the L2 cache, and we initiate a fresh cache access. Similar to Equation 7.1, the average access time of the L2 cache ($L2_{access\_time}$) is given by

$$\begin{aligned} L1_{miss\_penalty} &= L2_{access\_time} \\ &= L2_{hit\_time} + L2_{miss\_rate} \times L2_{miss\_penalty} \end{aligned} \tag{7.2}$$

We can write similar equations for the rest of the levels of the memory system. This process will terminate when we reach the main memory, or the hard disk. At this stage, the miss rate will be 0, and thus this recursive process will terminate. The set of equations for a memory system with three levels of caches and a main memory that contains all the data are as follows:

$$\begin{aligned} AMAT &= L1_{hit\_time} + L1_{miss\_rate} \times L1_{miss\_penalty} \\ L1_{miss\_penalty} &= L2_{hit\_time} + L2_{miss\_rate} \times L2_{miss\_penalty} \\ L2_{miss\_penalty} &= L3_{hit\_time} + L3_{miss\_rate} \times L3_{miss\_penalty} \\ L3_{miss\_penalty} &= MainMem_{access\_time} \end{aligned} \tag{7.3}$$

Note that the term, "miss rate", refers to the local miss rate, which is defined as follows. For example, the L3 miss rate (or local miss rate) is defined as the number of L3 misses divided by the number of L3 accesses. In comparison, the L3 global miss rate is defined as then number of L3 misses divided by the total number of memory accesses.

Finally, let us put the pieces together. The AMAT can be used to compute the CPI (cycles per instruction) of an in-order machine (also see Equation 2.1). For an OOO machine, the formula does not

hold exactly; however, it can be calibrated with a real system to provide numbers that are suggestive of broad trends. We have

$$CPI = CPI_{base} + f_{mem} * (AMAT - L1_{hit\_time}) \tag{7.4}$$

Here, CPI_{base} is the baseline CPI with a perfect memory system and f_{mem} is the fraction of memory instructions. In this equation, we account for the additional delay of the memory instructions. The additional delay is equal to $(AMAT - L1_{hit\_time})$. We multiply this with the fraction of memory instructions and add it to the baseline CPI. Note that we subtract $L1_{hit\_time}$ from $AMAT$ because we assume that the L1 hit time is already accounted for while computing CPI_{base}. The formula is derived by using the fact that the expected CPI is a sum of the expected values of its components.

7.1.7 Optimizing the Cache Design

The formula for the average memory access time does teach us something. It teaches us that to reduce the average memory access time we can either reduce the hit time, the miss rate, or the miss penalty. Let us discuss a few basic methods to reduce each of these factors.

Reducing the Hit Time

The hit time can be reduced by designing a smaller and faster cache. We can also reduce the associativity. However, we need to pay a price in terms of an increased miss rate. If the AMAT (average memory access time) decreases for the programs we are interested in, then such trade-offs are justified, otherwise such design choices are not justified.

Reducing the Miss Rate

Now that we know about the design of caches, particularly, set associative caches, let us look at the types of cache misses that we can have. These are typically known as the three Cs; misses can be classified into three categories.

Compulsory or Cold Misses These misses happen when we read in instructions or data for the first time. In this case, misses are inevitable, unless we can design a predictor that can predict future accesses and prefetch them in advance. Such mechanisms are known as prefetchers. We shall discuss prefetching schemes in detail in Sections 7.6 and 7.7. Prefetching is a general technique and is in fact known to reduce all kinds of misses.

Capacity Misses Assume we have a 16 KB L1 cache. However, we wish to access 32 KB of data on a frequent basis. Then we shall inevitably have a lot of misses, because the amount of data that the program wants to access will not fit in the cache. Other than generic schemes such as prefetching, better compiler algorithms or optimizations at the level of the code are more effective. For example, if we are multiplying two large matrices, we shall have capacity misses. It is possible to reduce this by reorganizing the code such that we always consider small blocks of data and operate on them (we shall look at such compiler driven schemes in Section 7.4.5).

Conflict Misses This is an artifact of having finite sized sets. Assume that we have a 4-way set associative cache, and we have 5 blocks in our access pattern that map to the same set. Since we cannot fit more than 4 blocks in the same set, the moment the 5^{th} block arrives, we shall have a miss. Such misses can be reduced by increasing the associativity. This will increase the hit time and thus may not always be desirable.

The standard way to reduce the miss rate is to have better prefetching schemes, increase the cache size or the associativity. However, they increase the hardware overheads. Here is a low-overhead scheme that is very effective.

Victim Cache A victim cache is a small cache that is normally added between the L1 and L2 caches. The insight is that sometimes we have some sets in the L1 cache that see a disproportionate number of accesses, and we thus have conflict misses. For example, in a 4-way set associative cache, we might have 5 frequently used blocks mapping to the same set. In this case, one of the blocks will frequently get evicted from the cache. Instead of going to the lower level, which is a slow process, we can add a small cache between L1 and L2 that contains such *victim blocks*. It can have 8-64 entries making it very small and very fast. For many programs, victim caches prove to be extremely beneficial in spite of their small size. It is often necessary to wisely choose which blocks need to be added to a victim cache. We would not like to add blocks that have a very low probability of being accessed in the future. We can track the usage of sets with counters and only store evicted blocks of those sets that are frequently accessed.

Reducing the Miss Penalty

Reducing the miss penalty is equivalent to reducing the memory access time at the lower level. This includes reducing the hit time or the miss rate at the lower level. However, there are schemes that are particularly tailored towards reducing the miss penalty.

One such scheme is *critical word first and early restart*. The insight is as follows. Almost always reads are on the critical path because we have waiting instructions and writes are not on the critical path. Furthermore, we often read data at the granularity of 4 bytes or 8 bytes, whereas a block is 32-128 bytes. This means that we often read a very small part of every block. We thus need not pay the penalty of fetching the entire block. Consider a 64-byte block. In most on-chip networks we can only transfer 8 bytes in a single cycle, and it thus takes 8 cycles to transfer a full block. The 8-byte packets are ordered as per their addresses.

We can instead transfer them in a different order. We can transfer the memory word (4 or 8 bytes) that is immediately needed by the processor first, and transfer the rest of the words in the block in later cycles. This optimization is known as fetching the *critical word first*. Subsequently, the processor can process the memory word, and start executing the consumer instructions. This is known as *early restart* since we are not waiting to fetch the rest of the bytes in the block.

This optimization helps reduce the miss penalty because we are prioritizing the data that needs to be sent first. This is easy to implement, and is heavily used. Furthermore, this technique is the most useful when applied between the L1 and L2 caches. It fails to show appreciable benefits when the miss penalty is large. The savings in the miss penalty do not significantly affect the AMAT in this case.

Next, let us discuss the write buffer that reduces the latency of writes to the lower level. It temporarily buffers the write and allows other accesses to go through.

Write Buffer A write buffer is a small fully associative cache for writes that we add between levels in the memory hierarchy, or attach with the store unit of the pipeline. It typically contains 4-8 entries where each entry stores a block. The insight is as follows. Due to spatial locality we tend to write to multiple words in a block one after the other. If we have repeated misses for different words in the same block, we do not want to send separate write-miss requests to the lower level. We should ideally send a single write-miss message for the block, and *merge* all the writes. This is the job of the write buffer's entry. For every write operation, we allocate a write buffer entry, which absorbs the flurry of writes to the same block. It is managed like a regular cache, where older entries are purged out and written to the lower level of the memory hierarchy. Subsequent read operations to the same block get their value from the write buffer entry. Other read operations that do not find their blocks in the write buffer can bypass the write buffer and can directly be sent to the lower levels of the memory system. To summarize, write buffers allow the processor to move ahead with other memory requests, reduce the pressure on the memory system by merging write operations to the same block, and can quickly serve read requests if their data is found in the write buffer.

Now that we have gained a good high level understanding of caches in modern processors, let us proceed to understand the internals of modern caches in Section 7.3. Before that we need to understand the concept of virtual memory.

7.2 Virtual Memory

This section provides a quick introduction to virtual memory. The treatment is cursory. For a detailed explanation the reader is requested to consult a basic textbook in architecture or operating systems.

Up till now we have conceptually assumed that the memory space is one large contiguous array of bytes. Each index of this array is the memory address. Figure 7.13 shows this conceptual view, where we have the processor and a large physical array of bytes. Of course, we have created a cache hierarchy to store parts of the address space closer to the processor; the key abstraction is that the entire memory space is nevertheless just one large array of bytes.

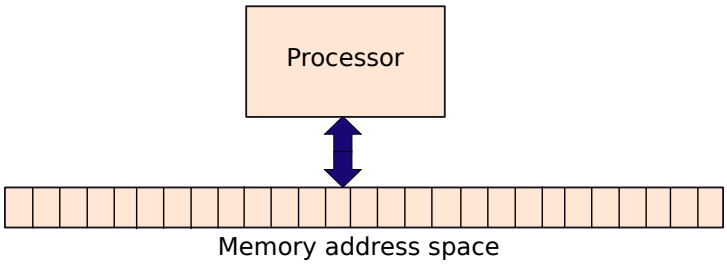

Figure 7.13: Conceptual view of the address space

Even though this abstraction suffices for a programmer and the compiler, it is not practical. Hence, as we have argued in Section 7.1, we need to create a memory hierarchy that consists of a set of caches. We have layers of caches that increasingly store larger and larger subsets of the memory address space. The reason that the memory hierarchy works is because of temporal locality. In the rare instances, where we cannot find some data in the caches, we need to access the off-chip main memory, which we assume contains all the data and instructions for the program. We never have a miss in the main memory.

Let us now systematically take the peel off all of these assumptions, and argue about the problems that we shall face in a real situation. As described in the introductory text by Sarangi [Sarangi, 2015], there are two problems in such implementations: the *overlap problem* and the *size problem*.

7.2.1 Overlap and Size Problems

The overlap problem is elaborated as follows. Even though we have been assuming that a single program is running on the processor, this is seldom the case. If we have multiple cores (processors on chip), then each core runs a separate program. Even if we have a single core, then also we share the core among different processes. Here a *process* is defined as the running instance of a program. The processes time-share the CPU. This means that after one process runs for some time, an external device called a timer signals an interrupt. The CPU loads the operating system (OS). The OS finds a process that is ready to execute and starts executing it on the core. In this manner a plurality of processes share the same CPU and operate in a time-multiplexed fashion.

Definition 42
A process is defined as a running instance of a program. Note that for one program we can create any number of processes. All of them are independent of each other, unless we use sophisticated mechanisms to pass messages between them. A process has its own memory space, and it further assumes that it has exclusive access to all the regions within its memory space.

The reader can click Ctrl-Alt-Del on her Microsoft® Windows® machine and see the list of processes that are currently active. She will see that there will be tens of processes that are active even if she has just one core on her laptop. This is because the processor is being time shared across all the processes. Furthermore, these programs are switching so quickly (tens of times a second) that the human brain is not able to perceive this. This is why we can have an editor, web browser, and video player running at the same time.

Note that these programs running on the processor have been compiled at different places by different compilers. All of them assume that they have complete and exclusive control over the entire memory space. This means that they can write to any memory location that they please. The problem arises if these sets of memory addresses overlap across processes. It is very much possible that process A and process B access the same memory address. In this case, one process may end up overwriting the other process's data, and this will lead to incorrect execution. Even worse, one process can steal secret data such as credit card numbers from another process. Such overlaps fortunately do not happen in real systems. This is because additional steps are taken to ensure that processes do not inadvertently or maliciously corrupt each other's memory addresses. This means that even if we maintain the abstraction that each process's memory space belongs to it exclusively, we somehow ensure that two processes do not corrupt each other's data. We need to somehow ensure that in reality the memory spaces do not unintentionally overlap.

Let us now look at the *size problem*. Assume that the size of the main memory is 1 GB. We have been assuming till now that all the accesses find their data in the main memory (miss rate is 0). This means that the maximum amount of memory that any process is allowed to use is limited to 1 GB. This is too restrictive in practice. It should be possible to run larger programs. In this case, we need to treat the main memory as the cache, and create a lower level beneath it. This is exactly how modern systems are organized as shown in Figure 7.14. The level beneath the main memory is the hard disk, which is a large magnetic storage device that typically has 10-100 times more capacity than main memory. We dedicate a part of the hard disk known as the *swap space* to store data that does not fit in the main memory.

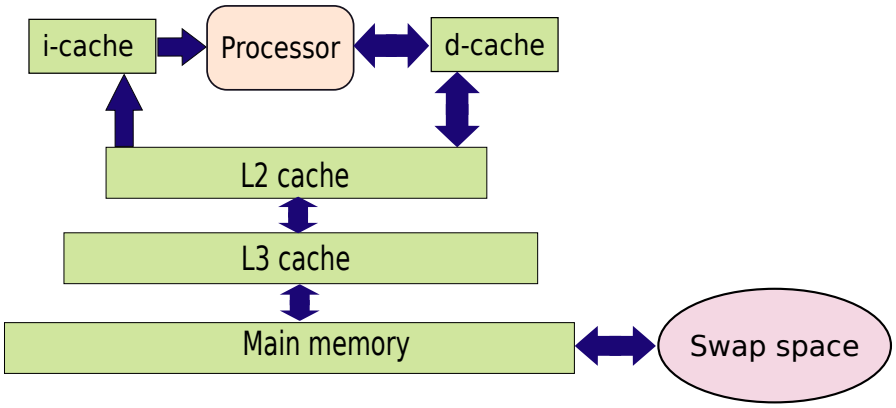

Figure 7.14: The memory hierarchy with the swap space

This should happen seamlessly, and the programmer or the compiler should not be able to know about the movement of data between main memory and the swap space. It should thus be possible for the programmer to use more memory than the capacity of the off-chip main memory. In this specific case, it should for example be possible to run a program that uses 3 GB of memory. Some data blocks will be in main memory, and the rest need to be in the swap space. The relationship is typically not *inclusive*.

Such a pristine view of the memory space is known as *virtual memory*. Every process has a *virtual*

view of memory where it assumes that the size of the memory that it can access is 2^N bytes, where we assume that valid memory addresses are N bits wide. For example, in a 32-bit machine the size of the virtual address space is 2^{32} bytes and on a 64-bit machine the size of the virtual address space is 2^{64} bytes. Furthermore, the process can unreservedly write to any location within its memory space without any fear of interference from other programs and the total amount of memory that a process can use is limited by the size of the main memory and the swap space. This memory space is known as the *virtual address space*.

Definition 43
Virtual memory is defined as a view of the memory space that is seen by each process. A process assumes its memory space (known as the virtual address space) is as large as the range of valid memory addresses. The process has complete and exclusive control over the virtual address space, and it can write to any location in the virtual address space at will without any fear of interference from other programs.

Note that at this point of time, *virtual memory* is just a concept. We are yet to provide a physical realization for it. Any physical implementation has to be consistent with the memory hierarchy that we have defined. Before proceeding further, let us enumerate the advantages of virtual memory:

1. It automatically solves the overlap problem. A process cannot unknowingly or even maliciously write in the memory space of another process.

2. It also automatically solves the size problem. We can store data in an area that is as large as the swap space and main memory combined. We are not limited by the size of the main memory.

3. Since we can write to any location at will within the virtual address space, the job of the programmer and the compiler become very easy. They can create code that is not constrained by a restrictive set of allowed memory addresses.

Important Point 13
Many students often argue that virtual memory is not required. We can always ask a process to use a region of memory that is currently unused, or we can force different programs at run time to use a different set of memory addresses. All of these approaches that seek to avoid the use of virtual memory have problems.

A program is compiled once, and run millions of times. It is not possible for the compiler to know about the set of memory addresses that a program needs to use in a target system to avoid interference from other programs. What happens if we run two copies of the same program? They will have clashing addresses.

Another school of thought is to express all addresses in a program as an offset from a base address, which can be set at runtime. The sad part is that this still does not manage to solve the overlap problem completely. This will work if the set of memory addresses in a program are somewhat contiguous. Again, if the memory footprint grows with time, we need to ensure that irrespective of how much it grows it will never encroach into the memory space of another process. This is fairly hard to ensure in practice.

7.2.2 Implementation of Virtual Memory

Virtual memory is extremely alluring as a concept. However, it needs a physical implementation and this physical implementation has to be reconciled with our existing hierarchy of caches. Let us make the following assumptions. All programs are written and compiled with virtual memory in mind. The compiler generates code that uses virtual addresses. In fact the programmer and compiler are only aware of the virtual address space. They are not aware of the physical memory hierarchy and the details of the caches.

Moreover, the address that is computed in the execute stage by the processor is also the virtual address because it is computed based on values supplied by the programmer. However, this address is not presented to the memory system. This is because the virtual addresses of two processes can be the same even though we are not referring to the same piece of data. Hence, it is necessary for the addresses to undergo a process of translation where virtual addresses are converted to physical addresses. Physical addresses are the same addresses that we use to access the memory hierarchy: this includes all the caches and the main memory itself.

Definition 44
A physical address refers to an actual location of a byte or a set of bytes in the on-chip or off-chip memory structures such as the caches, main memory, and swap space. The available range of physical addresses is known as the physical address space.

This process is shown in Figure 7.15. If we assume a 32-bit address then the input to the translator is a 32-bit address, and let's say the output is also a 32-bit address[1]. The only distinction is that the former is a virtual address, and the latter is a physical address, which can be used to access the memory system. Let us delve into this further.

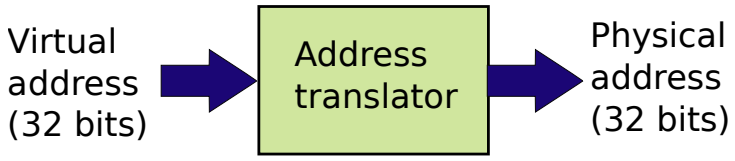

Figure 7.15: Virtual to physical address translation

Pages and Frames

Let us divide the virtual address space into fixed size chunks called pages. Typically, each page is 4 KB in size. Thus, to address a byte within a page, it takes 12 bits ($2^{12} = 4096$). If we remove these bits that specify the offset of a byte in a page (similar to the offset of a byte within a cache block), we are left with 20 bits. Let us refer to these 20 bits as the page id.

On similar lines let us break the physical address space into 4 KB chunks, and call them frames. *The main aim of the translation process is to map a page to a frame.* Note that we map a page to only one frame, and almost all the time we map a frame to only one page of a given process. Unlike the page, which is just a programmatic concept, a frame is associated with 4096 physical locations that store 1 byte each. Let us now understand why using pages and frames solves all the problems for us.

Consider the mapping shown in Figure 7.16. Here, we map the pages of two processes to frames. Note that even though the pages are contiguous, the frames are not. The mapping looks haphazard,

[1]The virtual and physical addresses need not have the same size.

because we genuinely have the flexibility of mapping a page to any frame, and there is no restriction on the order of the frames. They need not necessarily be arranged sequentially in the physical address space.

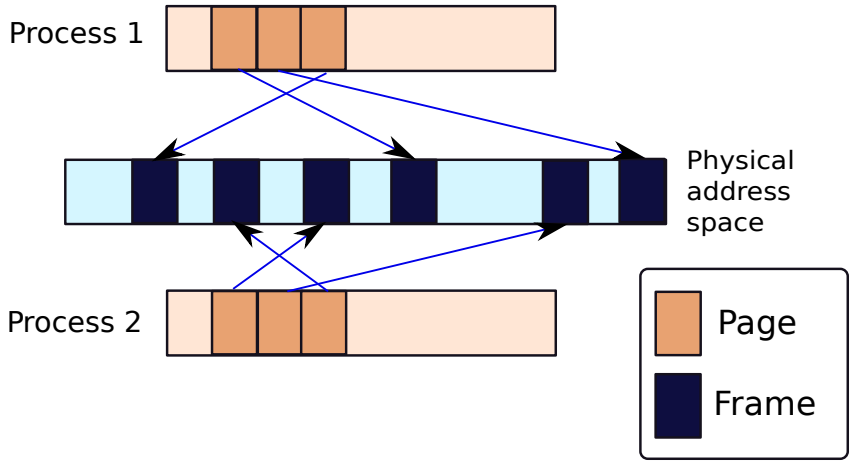

Figure 7.16: Mapping of pages to frames

It is easy to realize that we are solving the overlap problem seamlessly. If we never map the same frame to two different pages, then there is no way that writes from one process will be visible to another process. The virtual memory system will never translate addresses such that this can happen.

Solving the size problem is also straightforward (refer to Figure 7.17). Here, we map some pages to frames in memory and some to frames in the swap space. Even if the virtual address space is much larger than the physical address space, this will not pose a problem.

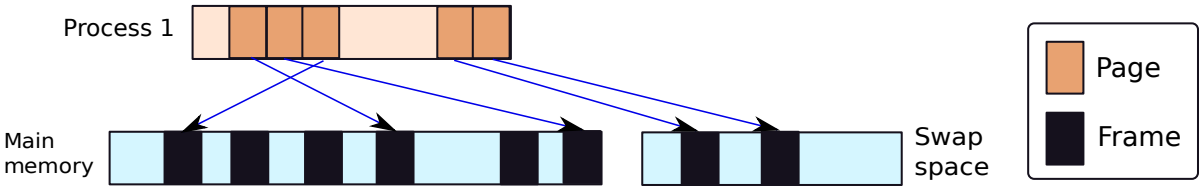

Figure 7.17: Mapping to the main memory and swap space

Mapping between Pages and Frames

In a 32-bit memory system with 4-KB pages, each page id is 20 bits. This is because we require 12 bits to address a byte in a page, and the remaining 20 bits specify the id of the page. Further, assume that we have 1 GB of main memory, and 1 GB of swap space. The total physical space thus available to a process is 2 GB. To address 2 GB of physical memory we require 31 bits (2 GB = 2^{31} bytes). Given that the frame size is equal to the page size, we require 19 (=31 - 12) bits to address each frame. We thus need to create a mapping that converts a 20-bit page id to a 19-bit frame id. Note that it is possible that we might later on add memory into the system by inserting additional memory chips in the motherboard, or we can dynamically increase the size of the swap space. In this case, we will need more bits for the frame ids. In such cases, 19 bits will not be enough. Since the size of a frame id can never be more than the size of a page id, we thus conservatively set the size of a frame id to the maximum value: 20

bits. Even if we actually require 19 bits in this case, we still use 20 bits keeping in mind that we might add more physical memory or swap space later, even at runtime. This is a standard assumption that is made in all practical systems. If some MSB bits are unused, they can be set to 0. Hence, to summarize, our mapping process needs to convert a 20-bit page id to a 20-bit frame id.

We need to maintain a data structure in software to store such mappings for each process. Such a data structure is known as the page table. There are different ways of efficiently implementing page tables. The reader can refer to books on operating systems [Silberschatz et al., 2018] or the background text by Sarangi [Sarangi, 2015] for a deeper discussion on page tables. A naive approach is to have a single level page table with 2^{20} entries, where each entry stores the id of a frame: 20 bits. The total space requirement is 2.5 MB per process, which is prohibitive.

To save space, most page tables are organized as 2-level tables, where we use the upper 10 bits to access a primary page table. Each entry of the primary page table points to a secondary page table. We use the subsequent 10 bits of the page id to access the secondary page table that contains the mapping. Note that all the primary page table entries will not point to valid secondary page tables. This is because most of the entries will correspond to portions of the address space that are unused. As a result, we can save a lot of space by creating only those secondary page tables that are required. For a 64-bit address space we can design a 3-level or 4-level page table. The reader is invited to study the space requirements of different page tables; she needs to convince herself that having multi-level page tables is a very good idea for large virtual address spaces.

Sadly, with such page tables we require multiple accesses to the memory system to read a single entry. Some of these entries maybe there in the cache; however, in the worst case we might need to make several accesses to main memory. Given that each main memory access takes roughly 200 to 400 cycles depending on the technology, this is a very expensive operation. Now if we look at these numbers in the light of the fact that we need to perform a virtual-to-physical mapping for every single memory access, the performance is expected to be abysmally poor. We clearly cannot afford such costly operations for every single memory access. This will offset all the gains that we have made in creating a sophisticated out-of-order pipeline and an advanced memory system.

Thankfully, a simple solution exists. We can use the same ideas that we used in caching: temporal locality and spatial locality. We keep a small hardware cache known as the Translation Lookaside Buffer (TLB) with each core. This keeps a set of the most recently used mappings from virtual pages to physical frame ids. Given the property of temporal locality, we expect to have a very high hit rate in the TLB. This is because most programs tend to access the same set of pages repeatedly. In addition, we can also exploit spatial locality. Since most accesses will be to nearby addresses, they are expected to be within the same page. Hence, saving a mapping at the level of pages is expected to be very beneficial because it has a potential for significant reuse.

Definition 45

- *The page table is a data structure that maintains a mapping between page ids and their corresponding frame ids. In most cases this data structure is maintained in a dedicated memory region by software. Specialized modules of the operating system maintain a separate page table for each process. However, in some architectures, notably the latest Intel processors, the process of looking up a mapping (also referred to as a* page walk*) is performed by hardware.*
- *To reduce the overheads of address translation, we use a small cache of mappings between pages and frames. This is known as the Translation Lookaside Buffer or the TLB. It typically contains 32-128 entries, and can be accessed in less than 1 clock cycle. It is necessary to access the TLB every time we issue a load/store request to the memory system.*

The process of memory address translation is thus as follows. The virtual address first goes to the TLB where it is translated to the physical address. Since the TLB is typically a very small structure that contains 32-128 entries its access time is typically limited to a single cycle. Once we have the translated address, it can be sent to the memory system, which is either the i-cache (for instructions) or the d-cache (for data).

TLB Misses and Page Faults

We expect a very high hit rate in the TLB (more than 99%). In rare cases, when we have a miss in the TLB, we need to fetch the mapping from the page table. This is an expensive operation because it may involve accesses to main memory, which take hundreds of cycles. If the mapping does not exist, yet the virtual address is valid, then we create a new mapping and proceed to allocate an empty frame in main memory. If the mapping exists, there are two cases: the frame is either present in memory, or it is present in the swap space on the hard disk. In the former case, nothing needs to be done other than simply updating the TLB with the mapping. However, in the latter case, we additionally need to allocate a frame in memory, and read in its contents from the disk. To summarize, whenever we do not find a frame in main memory, we need to perform some costly operations in terms of creating space in main memory, and possibly reading the data of the frame from the swap space. This event is known as a *page fault*.

The first step in servicing a page fault is to allocate a frame in main memory. If free space is available, then we can choose an empty frame, and use it. However, if such a frame is not available, there is a need to evict a frame from main memory by writing its contents to the swap space. This requires a method for page (or frame) replacement. There are many common algorithms to achieve this such as FIFO (first in first out) and LRU (least recently used). Once a frame is allocated, we either need to initialize it with all zeros (for security reasons) if we are creating a new mapping, or read in its contents from the disk. The latter is a slow operation. Hard disk access times are of the order of milliseconds. This translates to several million cycles for a single page fault.

The flowchart for the entire process is shown in Figure 7.18.

Definition 46
A page fault is defined as an event where a page's corresponding frame is not found in main memory. It either needs to be initialized in main memory, or its contents need to be read from the hard disk and stored in the main memory.

7.3 Modeling and Designing a Cache

Let us now delve deeper into the components of a cache's performance, which are area, power, and latency. The most widely used tool to model a cache and compute its parameters is the Cacti tool [Wilton and Jouppi, 1993]. It has been around for almost the last 25 years, and is still the tool of choice for modeling a cache. In this section, we shall heavily cite concepts from the Cacti research reports (versions 1 to 6). Note that we shall take liberties to simplify and generalize concepts wherever possible.

7.3.1 Memory Technologies used in a Cache: SRAM and CAM Arrays

Let us briefly explain how a cache is designed. Note that this is a superficial introduction to this topic. Interested readers can consult textbooks on the design of memory systems for more in-depth coverage.

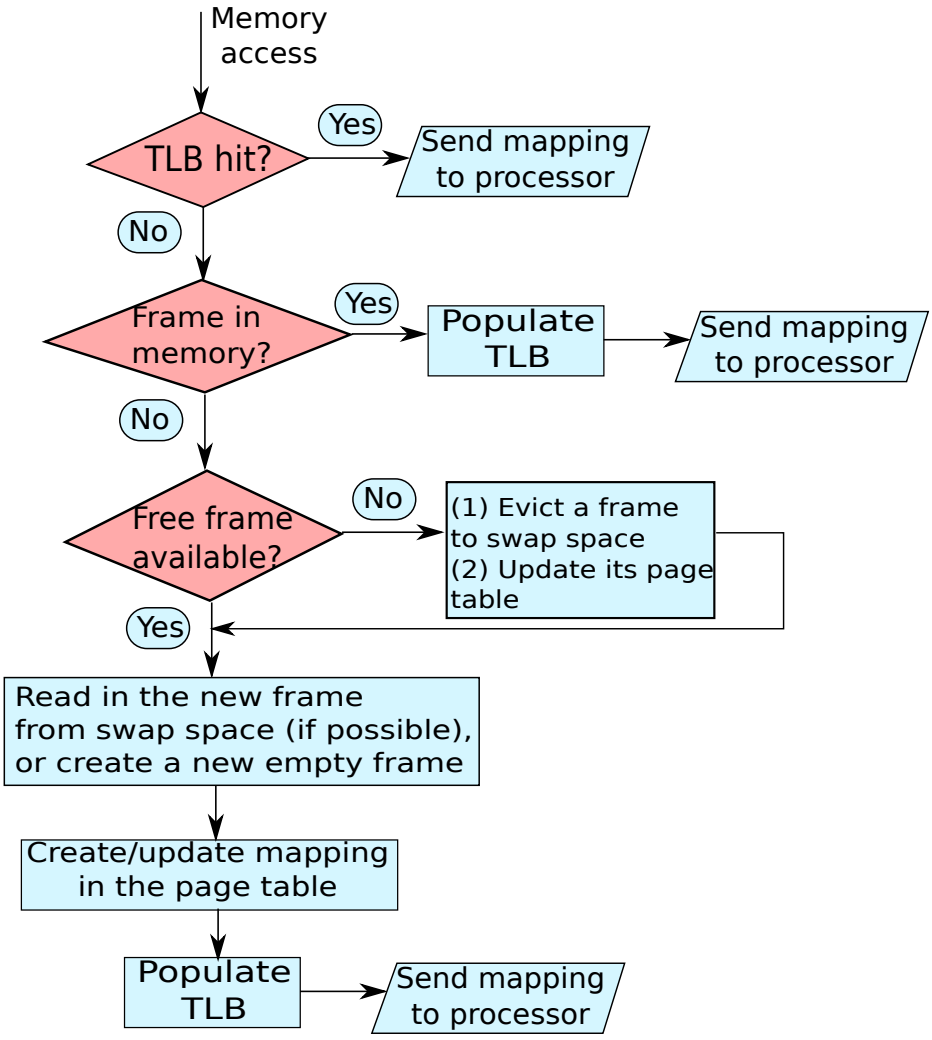

Figure 7.18: Flowchart for memory accesses

The basic element of any memory structure is a memory cell that stores 1 bit of information. There are many kinds of memory cells. In this book, we shall mainly use the 6-transistor SRAM cell, and the 10-transistor CAM cell. Let us quickly describe these memory technologies.

SRAM Cell

The question that we wish to answer is how do we store 1 bit of information? We can always use latches and flip-flops. However, these are area intensive structures and cannot be used to store thousands of bits. We need a structure that is far smaller.

Let us extend the idea of a typical SR latch as shown in Figure 7.19. An SR latch can store a single bit. If we set S to 1 and R to 0, then we store a 1. Conversely, if we set $S = 0$ and $R = 1$, we store a 0.

Let us appreciate the basic structure of this circuit. We have a cross-coupled pair of NAND gates.

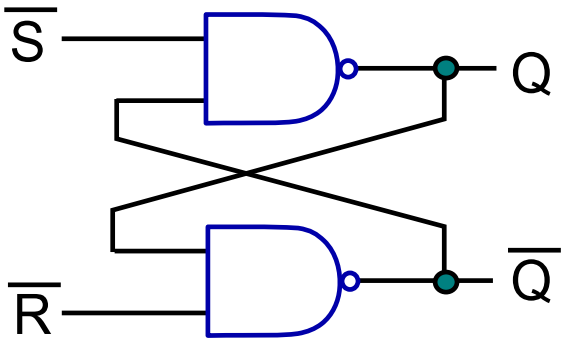

Figure 7.19: A basic SR latch

By cross-coupling, we mean that the output of one NAND gate is connected to the input of the other and likewise for the other gate. Unfortunately, a NAND gate has four transistors, and thus this circuit is not area efficient. Let us take this idea, and build a circuit that has two cross-coupled inverters as shown in Figure 7.20.

This structure can also store a bit. It just uses four transistors, because one CMOS inverter can be made out of one NMOS transistor and one PMOS transistor. To write a value, we can simply set the value of node Q to the value that we want to write. The other node will always have $\overline{Q}$ because of the two inverters. Reading a value is also easy. It is the output at node Q.

However, this circuit has a drawback. There is no way to enable or disable the circuit. Enabling and disabling a memory cell is very important because we are not reading or writing to the cell in every cycle. We want to maintain the value in the cell when we are not accessing it. During this period, its access should be disabled. Otherwise, whenever there is a change in the voltages of nodes Q or $\overline{Q}$, the value stored in the cell will change. If we can effectively disconnect the cell from the outside world, then we are sure that it will maintain its value.

This is easy to do. We can use the most basic property of a transistor, which is that it works like a switch. We can connect two transistors to both the nodes of the cross-coupled inverter. This design in shown in Figure 7.21. We add two transistors – W_1 and W_2 – at the terminals Q and $\overline{Q}$, respectively. These are called *word line transistors*; they connect the inverter pair to two bit lines on either side. The gates of W_1 and W_2 are connected to a single wire called the *word line*. If the word line is set to 1, both the transistors get enabled (the switch closes). In this case, the bit lines get connected to the terminals Q and $\overline{Q}$ respectively. We can then read the value stored in the inverter pair and also write to it. If we set the word line to 0, then the switch gets disconnected, and the 4-transistor inverter pair is disconnected from the bit lines. We cannot read the value stored in it, or write to it. Thus, we have a 6-transistor memory cell; this is known as an SRAM (static random access memory) cell. This is a big improvement as compared to the SR latch in terms of the number of transistors that are used.

SRAM Array

Now that we have designed a memory cell that can store a single bit, let us create an array of SRAM cells. We can use this array to store data.

To start with, let us create a matrix of cells as shown in Figure 7.22. We divide the address used to access the SRAM array into two parts: row address, and column address. We send the row address to the row decoder. Assume that it contains r bits. The decoder takes in these r bits, and it sets one out of 2^r output lines that are the word lines to 1. Each word line is identified by the binary number represented by the row address. For example, if the row address is 0101, then the fifth output of the

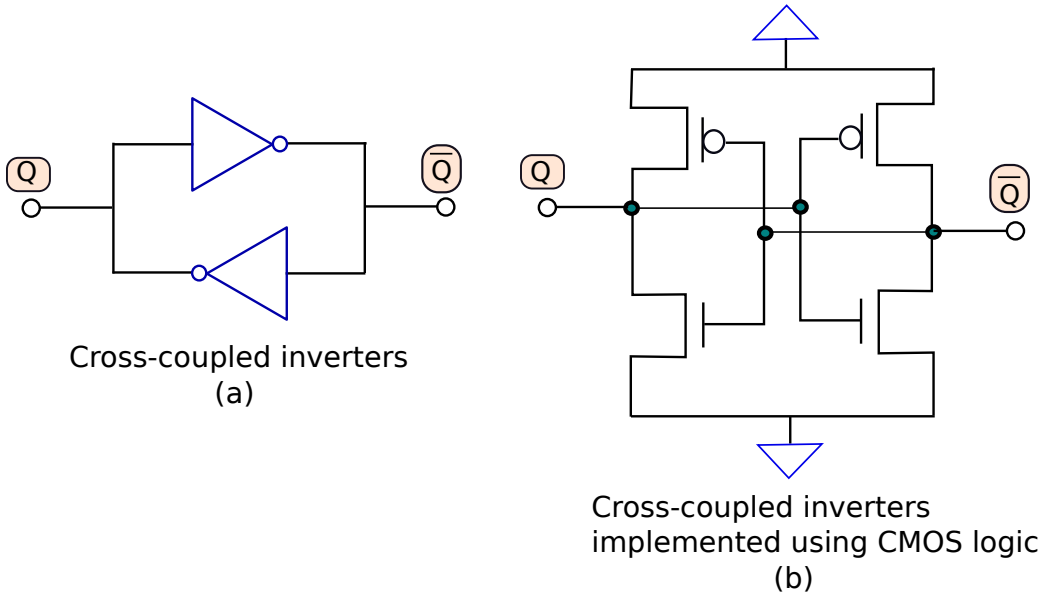

Figure 7.20: Cross-coupled inverters

decoder is set to 1 or the fifth word line is set to 1. The rest of the output lines (word lines) are set to 0.

The decoder is a standard electronic circuit, and can be easily constructed out of logic gates. The benefit of using the decoder is that it enables only one of the word lines. This word line subsequently enables a row of cells in the 2D array (matrix) of memory cells. All the memory cells in a row can then be accessed.

Let us consider the process of writing first. Every cell is connected to two wires that are called bit lines. Each bit line is connected to a node of the memory cell (Q or $\overline{Q}$) via a transistor switch that is enabled by a word line. The bit lines carry complementary logic values. Let us refer to the left bit line as BL and the right bit line as $\overline{BL}$. To write values we use fast write drivers that can quickly set the voltage of the bit lines. If we want to write a 1 to a given cell, then we set its bit lines as follows: BL to a logical 1 and $\overline{BL}$ to a logical 0. We do the reverse, if we wish to write a logical 0. The pair of inverters get reprogrammed once we set the voltages of BL and $\overline{BL}$.

The more difficult operation is reading the SRAM array. In this case, once we enable the memory cell, the bit lines get charged to the values that are contained in the memory cell. For example, if the value of a node of the memory cell is a logical 1 (assume a logical 1 is 1 V), then the voltage on the corresponding bit line increases towards 1 V. Similarly, the voltage on the other bit line starts moving towards 0 V. This situation is not the same as writing a value. While writing a value we could use large

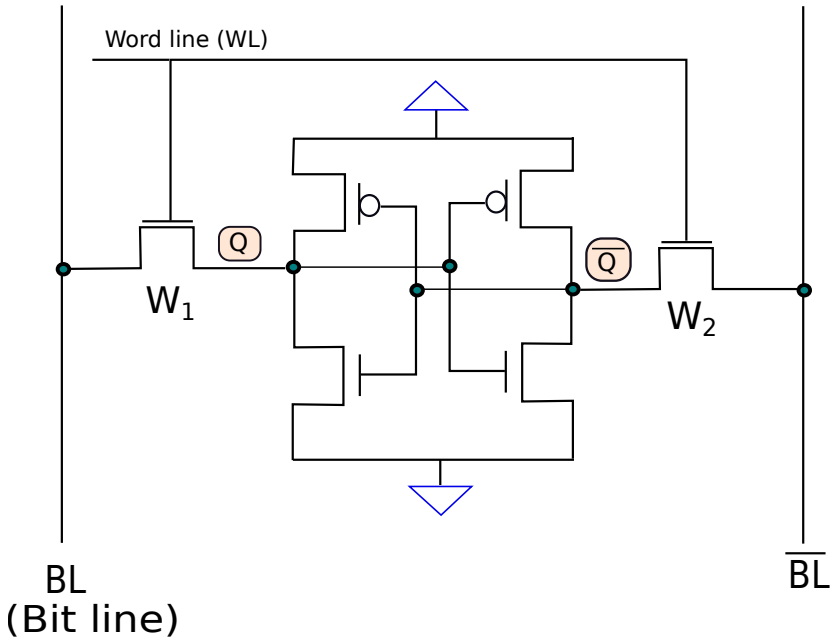

Figure 7.21: Cross-coupled inverters with enabling transistors

write drivers that can pump in a lot of current into the bit lines. In this case, we only have a small 6-transistor cell that is charging the bit lines. It is far weaker than the powerful write drivers. As a result, the process of charging (towards a logical 1) or discharging (towards a logical 0) is significantly slower. Note that the process of reading is crucial. It is often on the critical path because there are instructions waiting for the value read by a load instruction. Hence, we need to find ways to accelerate the process.

A standard method of doing this is called *precharging*. We set the value of both the bit lines to a value that is midway between the voltages between logical 0 and 1. Since we have assumed that the voltage corresponding to a logical 1 is 1 V, the precharge voltage is 0.5 V. We use strong precharge drivers to precharge both the bit lines to 0.5 V. Akin to the case with write drivers, this process can be done very quickly. Once the lines are precharged, we enable a row of memory cells. Each memory cell starts setting the values of the bit lines that it is connected to. For one bit line the voltage starts moving to 1 V and for the other the voltage starts moving towards 0 V. We monitor the difference in voltage between the two bit lines.

Assume the cell stores a logical 1. In this case, the voltage on the bit line BL will try to move towards 1 V, and the voltage on the bit line $\overline{BL}$ will try to move towards 0 V. The difference between the voltages of BL and $\overline{BL}$ will start at 0 V and gradually increase to 1 V. However, the key idea is that we do not have to wait till the difference reaches 1 V or -1 V (when we store a logical 0). Once the difference crosses a certain threshold, we can infer the final direction in which the voltages on both the bit lines are expected to progress. Thus, much before the voltages on the bit lines reach their final values, we can come to the correct conclusion.

Let us represent this symbolically. Let us define the function V to represent the voltage. For example, $V(BL)$ represents the instantaneous voltage of the bit line BL. Here are the rules for inferring a logical 0 or 1 after we enable the cell for reading.

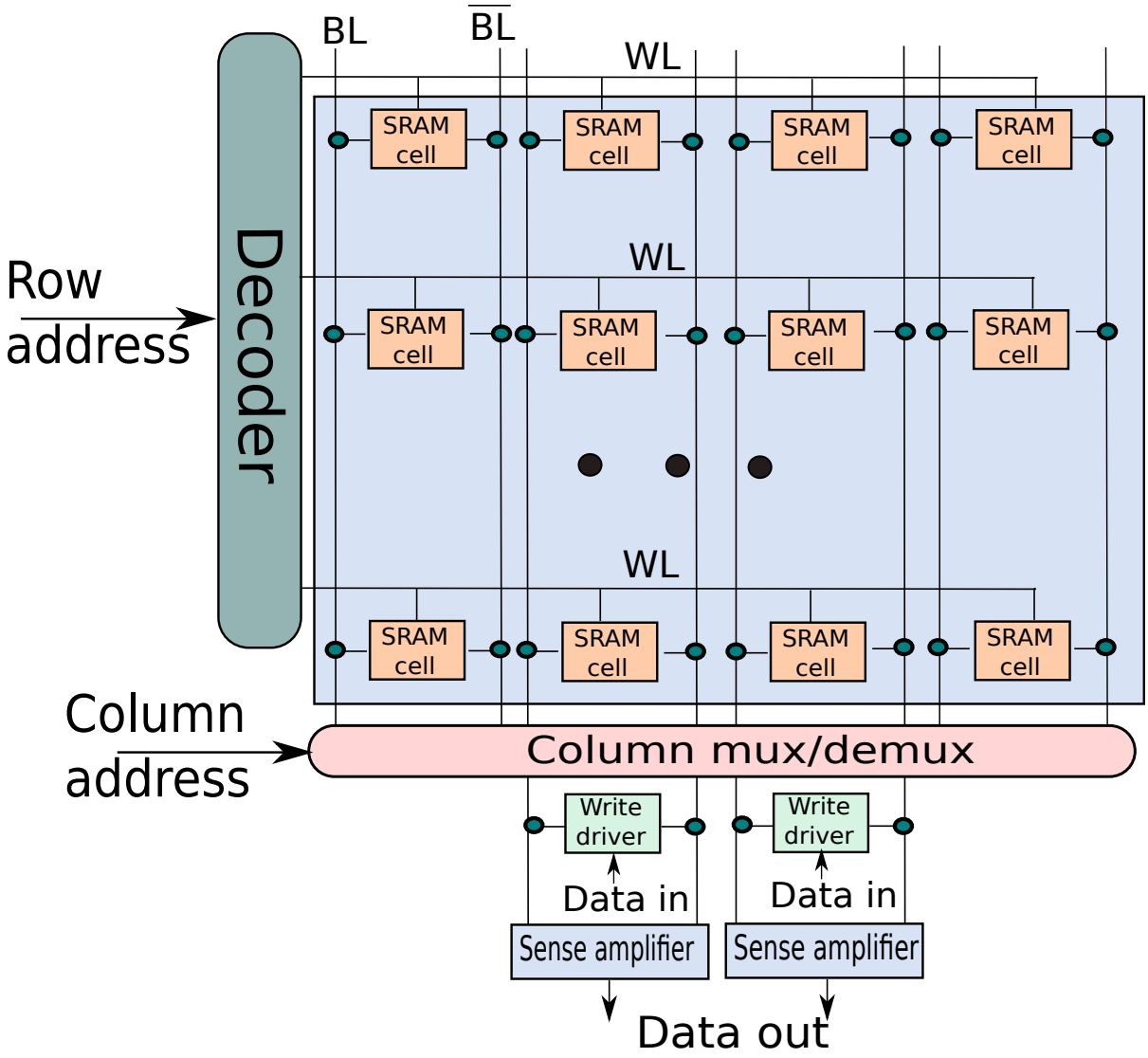

Figure 7.22: An SRAM array

$$value = \begin{cases} 1 & V(BL) - V(\overline{BL}) > \Delta \\ 0 & V(\overline{BL}) - V(BL) > \Delta \end{cases}$$

In this case, we define a threshold Δ that is typically of the order of tens of millivolts. An astute reader might ask a question about the need for the threshold, Δ. One of the reasons is that long copper wires such as bit lines can often accumulate an EMF due to impinging electromagnetic radiation. In fact, unbeknownst to us a long copper wire can act as a miniature antenna and pick up electric fields. This might cause a potential to build up along the copper wire. In addition, we might have crosstalk between copper wires where because of some degree of inductive and capacitive coupling adjacent wires might get charged. Due to such types of *noise*, it is possible that we might initially see the voltage on a bit line swaying in a certain direction. To completely discount such effects, we need to wait till the absolute value of the voltage difference exceeds a threshold. This threshold is large enough to make us

sure that the voltage difference between the bit lines is not arising because of transient effects such as picking up random electric fields. We should be sure that the difference in voltages is because one bit line is moving towards logical 1 and the other towards logical 0.

At this point, we can confidently declare the value contained in the memory cell. We do not have to wait for the voltages to reach their final values. This is thus a much faster process. The lower we set Δ, faster is our circuit. We are limited by the amount of noise.

Note that there is one hidden advantage of SRAM arrays. Both BL and $\overline{BL}$ are spaced close together. Given their spatial proximity, the effects of noise will be similar, and thus if we consider the difference in voltages, we shall see that the effects of electromagnetic or crosstalk noise will mostly get canceled out. This is known as *common mode rejection* and works in our favor. If we had a single bit line, this would not have happened.

In our SRAM array (shown in Figure 7.22) we enable the entire row of cells. However, we might not be interested in all of this data. For example, if the entire row contains 64 bytes of data, and we are only interested in 16 bytes, then we need to choose the component of the row that we are interested in. This is done using the column multiplexers that read in only those columns that are we are interested in. The column address is the input to these column multiplexers. For example, in this case since there are four 16 byte chunks in a 64 byte row, there are four possible choices for the set of columns. We thus need 2 bits to encode this set of choices. Hence, the column address is 2 bits wide.

Column Multiplexers

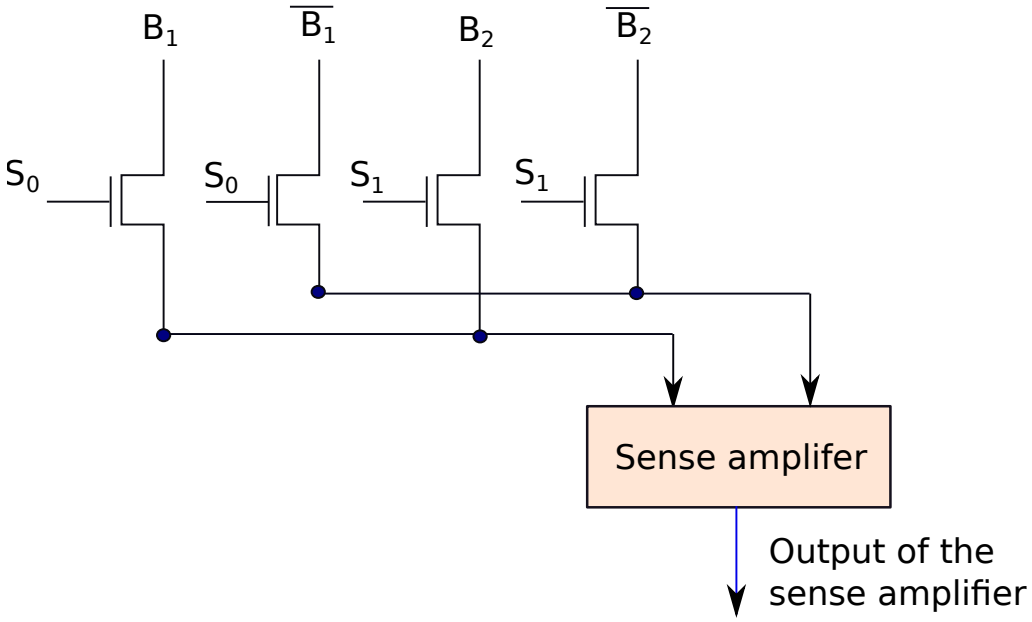

Figure 7.23: A column multiplexer

Let us look at the design of the column multiplexers. Figure 7.23 shows a simple example. We have two pairs of bit lines: $(B_1, \overline{B_1})$ and $(B_2, \overline{B_2})$. We need to choose one of the pairs, and connect each wire to an NMOS transistor, which is known as a *pass transistor*. If the input at its gate is a logical 1, then the transistor conducts, and the voltage at the drain is reflected at the source. However, if the voltage at the gate is a logical 0, then the transistor behaves as an open circuit.

In this case, we need a single column address bit because there are two choices. We send this bit to a decoder and derive two outputs: S_0 and S_1. Only one of them can be true. Depending upon the output

that is true, the corresponding pass transistors get enabled. We connect B_1 and B_2 to the same wire, and we do the same with $\overline{B_1}$ and $\overline{B_2}$. Only one bit line from each pair will get enabled. The enabled signals are then sent to the sense amplifier that senses the difference in the voltage and determines the logic level of the bit stored in the SRAM cell.

Sense Amplifiers

After we have chosen the columns that we are interested in, we need to compare BL and $\overline{BL}$ for each cell, and then ascertain which way the difference is going (positive or negative). We use a specialized circuit called a sense amplifier for this purpose. The circuit diagram of a typical sense amplifier is shown in Figure 7.24.

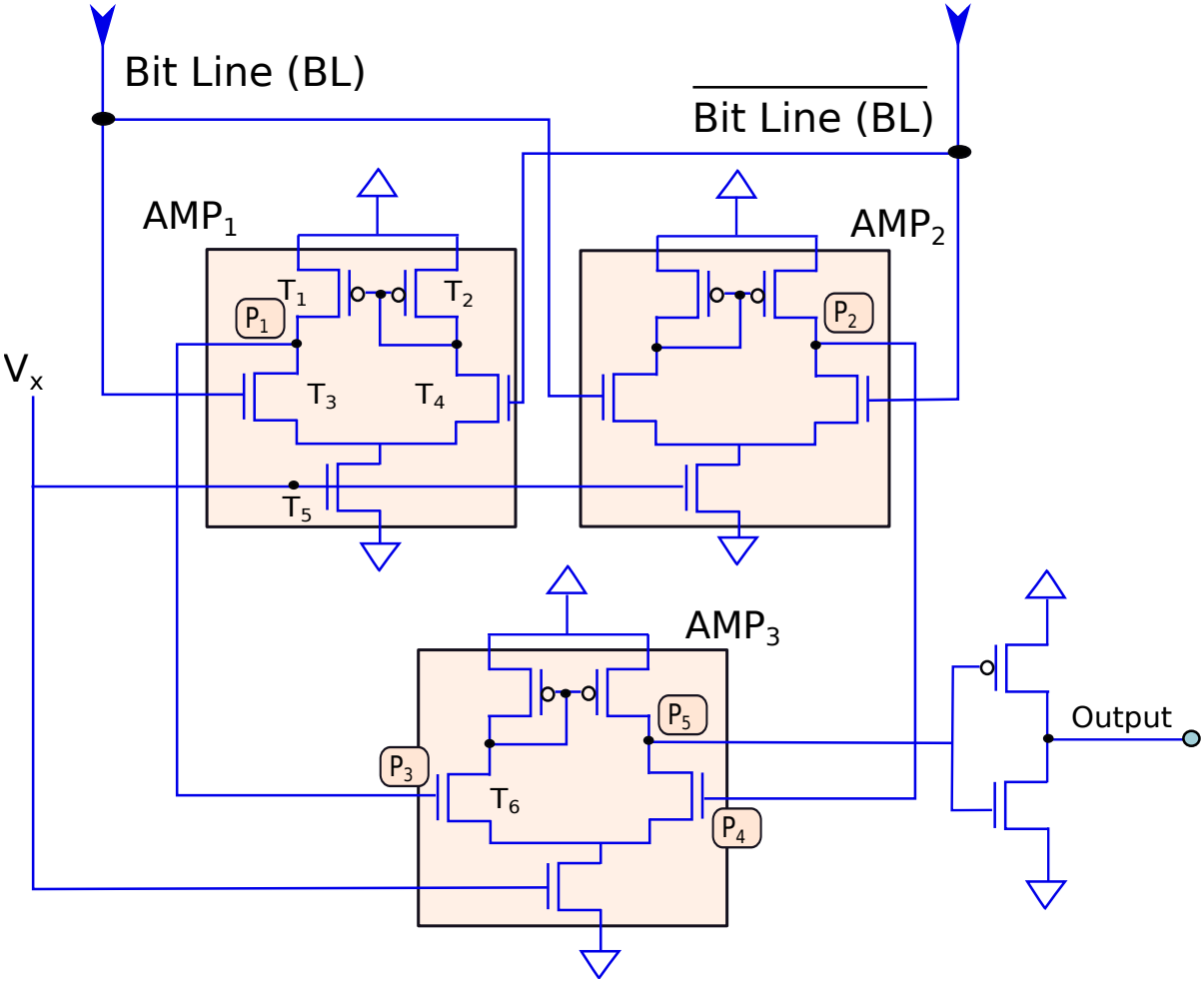

Figure 7.24: A sense amplifier

Let us describe the operation of a sense amplifier as shown in Figure 7.24. A sense amplifier is made up of three differential amplifiers. In Figure 7.24, each shaded box represents a differential amplifier. Let us consider the amplifier at the left top and analyze it. We shall provide an informal treatment in this book. For a better understanding of this circuit, readers can perform circuit simulation or analyze the circuit mathematically. First, assume that $V(BL) = V(\overline{BL})$. Transistors T_1 and T_2 form a circuit known

as a current mirror, where the current flowing through both the transistors is the same. Furthermore, transistor T_2 is in saturation ($V_{SD} > V_{SG} - V_T$), which fixes the current flowing through the transistor.

Now assume that $V(BL)$ is slightly lower than $V(\overline{BL})$, specifically $V(\overline{BL}) - V(BL) = \Delta$, which is the difference threshold for detecting a logic level. In this case transistor T_4 will draw slightly more current as compared to the equilibrium state. Let this current be I_d mA. This means an additional current of I_d mA will pass through T_2. Because we have a current mirror, the same additional current I_d will also pass through T_1. However, the current through T_5 will remain constant because it is set to operate in the saturation region (the reader should verify this by considering possible values of V_x). This means that since the current through T_4 has increased by I_d, the current through T_3 needs to decrease by I_d to ensure that the sum (flowing through T_5) remains constant.

The summary of this discussion is that an additional I_d mA flows through T_1 and the current in T_3 decreases by I_d mA. There is thus a total shortfall of $2I_d$ mA, which must flow from terminal P_1 to P_3. Terminal P_3 is the gate of transistor T_6. This current will serve the purpose of increasing its gate voltage. Ultimately, the current will fall of to zero once the operating conditions of transistors $T_1 \ldots T_5$ change. Thus, the net effect of a small change in the voltage between the bit lines is that the voltage at terminal P_3 increases significantly. Consider the reverse situation where $V(\overline{BL})$ decreases as compared to $V(BL)$. It is easy to argue that we shall see a reverse effect at terminal P_3. Hence, we can conclude that transistors $T_1 \ldots T_5$ make up a differential amplifier (AMP_1 in the figure).

Similarly, we have another parallel differential AMP_2 where the bit lines are connected to the amplifier in a reverse fashion. Let us convince ourselves that the directions in which the voltages increase at terminals P_1 and P_2 are opposite. When one decreases, the other increases and vice versa. The role of the parallel differential amplifiers(AMP_1 and AMP_2) is to amplify the difference between the voltages at the bit lines; this shows up at terminals P_3 and P_4.

Terminals P_3 and P_4 are the inputs to another differential amplifier AMP_3, which further amplifies the voltage difference between the terminals. We thus have a two-stage differential amplifier. Note that as $V(\overline{BL})$ increases, the voltage at terminal P_3 increases and this increases the voltage at terminal P_5 (using a similar logic). However, with an increase in $V(\overline{BL})$ we expect the output to become 0, and vice versa. To ensure that this happens, we need to connect an inverter to terminal P_5. The final output of the sense amplifier is the output of the inverter.

Sense amplifiers are typically connected to long copper wires that route the output to other functional units. Sense amplifiers are typically very small circuits and are not powerful enough to charge long copper wires. Hence, we need another circuit called the *output driver* that takes the output of a sense amplifier, and stabilizes it such that it can provide enough charge to set the potential of long copper wires to a logical 1 if there is a need. Note that this is a basic design. There are many modern power-efficient designs. We shall discuss another variant of sense amplifiers that are used in DRAMs in Chapter 10.

CAM Cell

In Section 7.1.4 we had discussed the notion of a CAM (content-addressable memory) array, where we address a row in the matrix of memory cells based on its contents and not on the basis of its index. Let us now proceed to design such an array. The basic component of a CAM array is the CAM cell (defined on the same lines as a 6-transistor SRAM cell).

Figure 7.25 shows the design of a CAM cell with 10 transistors. Let us divide the diagram into two halves: above and below the match line. The top half looks the same as an SRAM cell. It contains 6 transistors and stores a bit using a pair of inverters. The extra part is the 4 extra transistors at the bottom. The two output nodes of the inverter pair are labeled Q and $\overline{Q}$ respectively. In addition, we have two inputs, A and $\overline{A}$. Our job is to find out if the input bit A matches the value stored in the inverter pair, Q.

The four transistors at the bottom have two pairs of two NMOS transistors each connected in series. Let us name the two transistors in the first pair T_1 and T_2 respectively. The drain terminal of T_1 is connected to the *match line*, which is initially precharged to the supply voltage. The first pair of NMOS

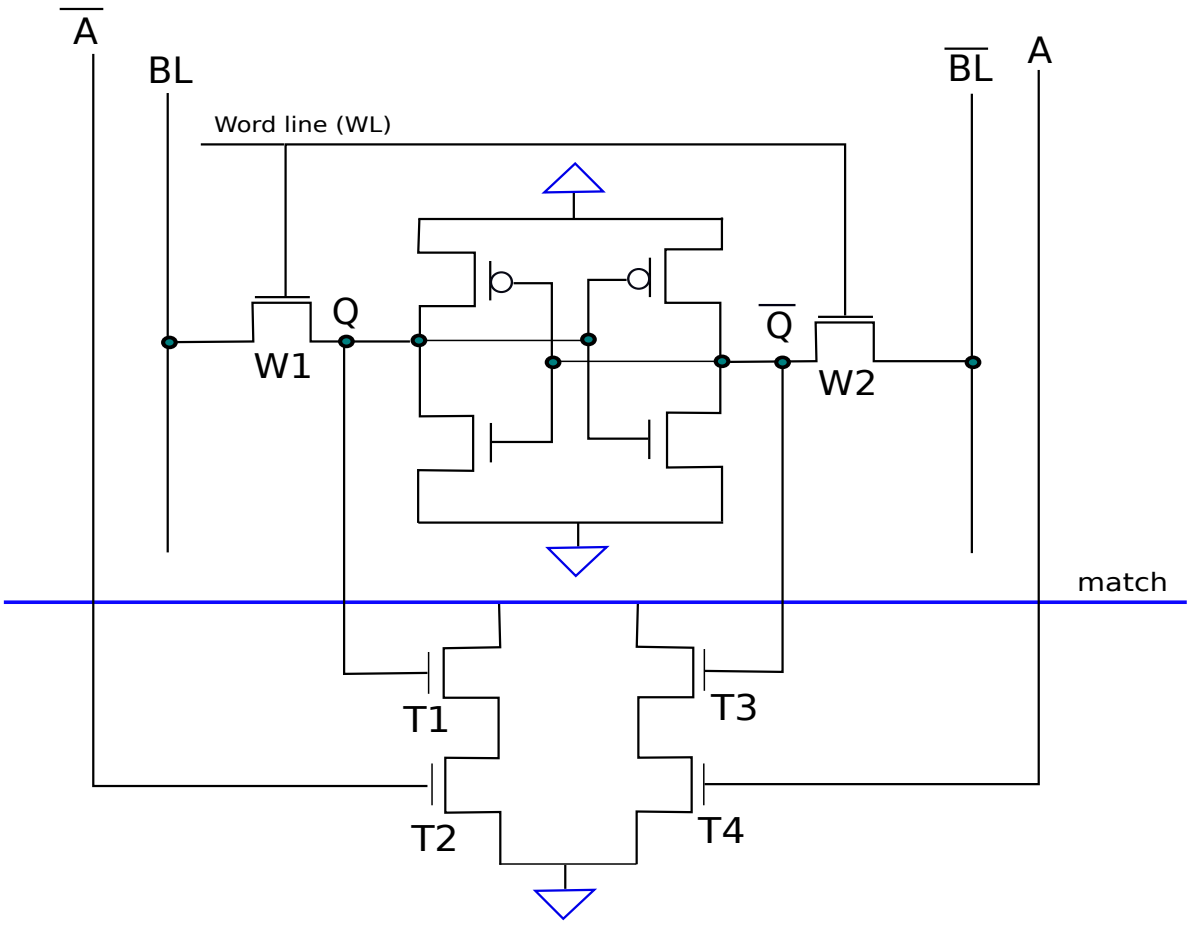

Figure 7.25: A CAM cell

transistors is connected to Q and $\overline{A}$ respectively. Let us create a truth table based on the inputs and the states of the transistors T_1 and T_2.

Q	$\overline{A}$	T_1	T_2
0	0	off	off
0	1	off	on
1	0	on	off
1	1	on	on

If the inputs Q and $\overline{A}$ are both 1, then only do both the transistors conduct. Otherwise, at least one of the transistors does not conduct. In other words if $Q = 1$ and $A = 0$, there is a straight conducting path between the match line and the ground node. Thus, the voltage of the match line becomes zero. Otherwise, the voltage of the match line remains the same as its precharged value because there is no conducting path to ground.

Let us now look at the other pair of transistors, T_3 and T_4, that are connected in the same way albeit to different inputs: $\overline{Q}$ and A (complements of the inputs to transistors T_1 and T_2). Let us build a similar truth table.

$\overline{Q}$	A	T_3	T_4
0	0	off	off
0	1	off	on
1	0	on	off
1	1	on	on

Here also, the only condition for a conducting path is $\overline{Q} = A = 1$. If we combine the results of both the truth tables, then we have the following conditions for the match line to get set to 0. Either $Q = \overline{A} = 1$, or $\overline{Q} = A = 1$. We should convince ourselves that this will only happen if $A \neq Q$. If $A = Q$, then both of these conditions will always be false. One of the values will be 1 and the other will be 0. However, if $A \neq Q$, then only this is possible.

Let us thus summarize. A CAM cell stores a value Q. In addition, it takes as input the bit A that is to be compared with Q. If the values are equal, then the match line maintains its precharged value. However, if they are not equal then a direct conducting path forms between the match line and ground. Its value gets set to a logical 0. Thus, by looking at the potential of the match line, we can infer if there has been a match or not.

CAM Array

Let us build a CAM array the same way we built the SRAM array. Figure 7.26 shows the design.

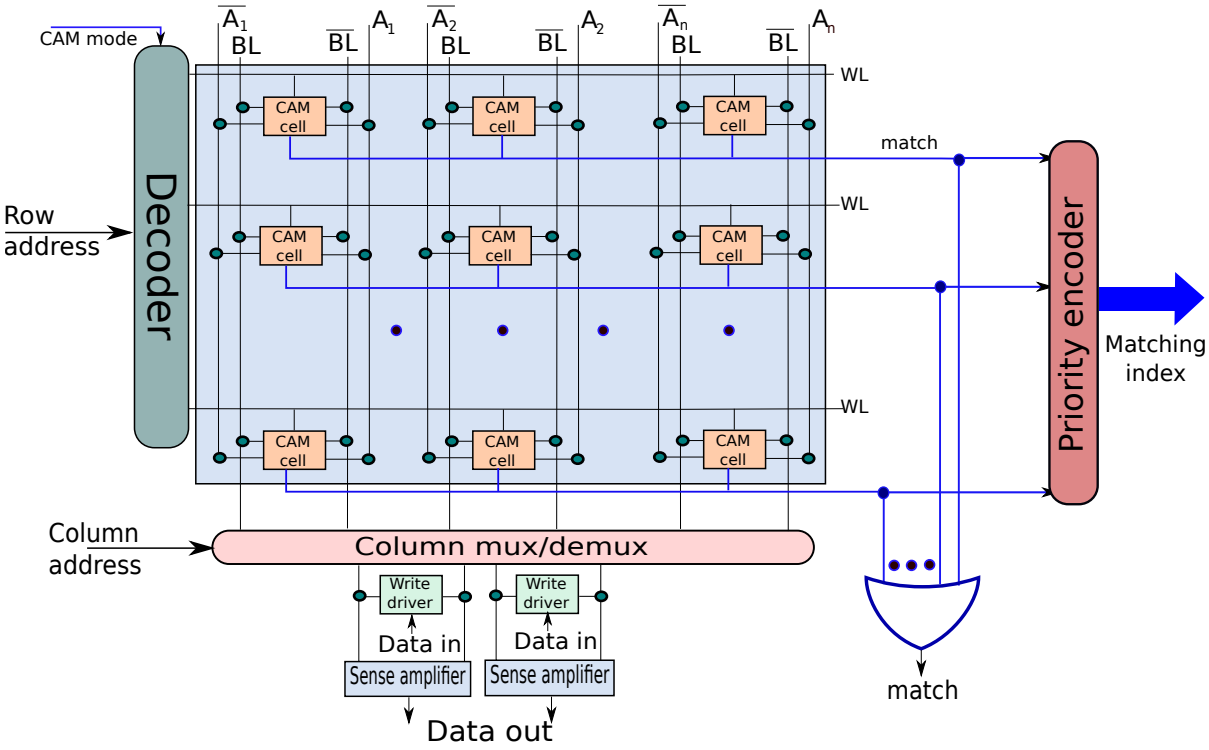

Figure 7.26: A CAM array

Typically, a CAM array has all the features of an SRAM array, and in addition it has more features such as content-based addressability. The row address decoder, column multiplexers, sense amplifiers, precharge circuits, write and output drivers are the parts that the CAM array inherits from the SRAM

array. To use the CAM array like a regular SRAM array, we just simply enable the row decoder and proceed with a regular array access. In this case, we do not set the match line or read its value.

Let us now focus on the extra components that are useful when we are searching for an entry based on its contents. To keep the discussion simple, we assume that we wish to match an entire row. Extending this idea to cases where we wish to match a part of the row is trivial. Now, observe that all the transistors in the same row are connected to the same match line (see Figure 7.26). We provide a vector V as input such that it can be compared with each row bit by bit. Thus, the i^{th} bit in the vector is compared with the value stored in the i^{th} CAM cell in the row. In other words, at each cell we compare a pair of bits: one from the vector V (bit A_i in the figure) and the value stored in the CAM cell. First, assume that all the bits match pairwise. In this case, we observe the bits to be equal at each cell, and thus a conducting path between the match line and ground does not form. This happens for all the cells in the row. Thus, the match line continues to maintain its precharged value: a logical 1.

Now, consider the other case, where at least one pair of bits does not match. In this case at that CAM cell, a conducting path forms to the ground, and the match line gets discharged; the voltage gets set to a logical 0.

Thus, to summarize, we can infer if the entire row has matched the input vector V or not by basically looking at the voltage of the match line after the compare operation. If it is the same as its precharged value (logical 1), then there has been a full match. Otherwise, if the match line has been discharged, then we can be sure that at least one pair of bits has not matched.

Hence, the process of addressing a CAM memory based on the contents of its cells is as follows. We first enable all the word lines. Then, we set the bits of the input vector (bits $A_1 \ldots A_n$ in the figure) and then allow the comparisons to proceed. We always assume that we do not have any duplicates. There are thus two possible choices: none of the match lines are at a logical 1 or only one of the match lines is set to 1. This is easy to check. We can create an OR circuit that checks if any of the match lines is a logical 1 or not. If the output of this circuit is 1, then it means that there is a match, otherwise there is no match. Note that it is impractical to create a large OR gate using NMOS transistors. We can either create a tree of OR gates with a limited fan-in (number of inputs), or we can use wired-OR logic, where all the match lines are connected to a single output via diodes (as shown in Figure 7.27). If any of the match lines is 1, it sets the output to 1. Because of the diodes current cannot flow from the output terminal to the match lines.

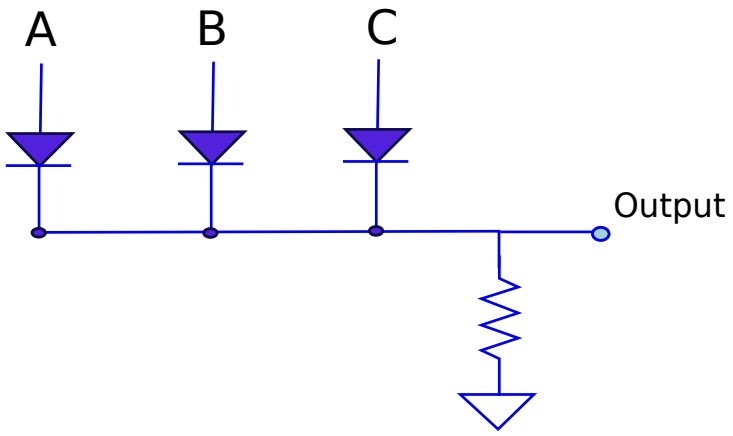

Figure 7.27: Wired-or logic

Now, if we know that one of the match lines is set (equal to 1), we need to find the number of the row that matches the contents. Note that our count starts from 0 in this case (similar to arrays in programming languages). We can use an encoder for this that takes N inputs and has $log_2(N)$ outputs.

The output gives the id (Boolean encoding) of the input that is set to 1. For example, if the 9^{th} input out of a set of 16 inputs is set to 1 its encoding will be 1001 (assume that the count starts at 0).

In a fully associative cache whose tag array is implemented as a CAM array, once we get the id of the row whose match line is set to 1, we can access the corresponding entry of the data array and read or write to the corresponding block. A CAM array is thus an efficient way of creating a hash table in hardware. The tag array of a fully associative cache is typically implemented as a CAM array, whereas the data array is implemented as a regular SRAM array.

7.3.2 Designing a Cache

To design a real cache using our basic technologies, we need to consider the inputs that a typical cache modeling tool such as Cacti takes ([Wilton and Jouppi, 1993]).

Field	Description
A	Associativity
B	Block size (in bytes)
C	Cache size (in bytes)
b_o	Width of the output (in bits)
b_{addr}	Width of the input address (in bits)

These are the most basic parameters for any set associative cache. Note that direct mapped caches and fully associative caches are specific instances of set associative caches. Hence, we use the set associative cache as a basis for all our subsequent discussion. Using the ABC parameters (associativity, block size, and cache size), we can compute the size of the set index and the size of the tag (refer to Example 4).

Example 4
Given A (associativity), B (block size in bytes), W (width of the input address in bits), and C (cache size), compute the number of bits in the set index and the size of the tag.

Answer: *The number of blocks that can be stored in the cache is equal to C/B. The size of each set is A. Thus, the number of sets is equal to $C/(BA)$. Therefore, the number of bits required to index each set is $log_2(C/(BA))$.*

Let us now compute the size of the tag. We know that the number of bits in the memory address is W. Furthermore, the block size is B bytes, and we thus require $log_2(B)$ bits to specify the index of a byte within a block (block address bits).

Hence, the number of bits left for the tag is as follows:

$$\begin{aligned} \textit{tag bits} &= \textit{address size} - \#\textit{set index bits} - \#\textit{block address bits} \\ &= W - log_2(C/(BA)) - log_2(B) \\ &= W - log_2(C) + log_2(A) \end{aligned}$$

A naive organization with the ABC parameters might result in a very skewed design. For example, if we have a large cache, then we might end up with a lot of rows and very few columns. In this case, the load on the row decoder will be very large and this will become a bottleneck. Additionally, the number of devices connected to each bit line would increase and this will increase its capacitance, thus making it slower because of the increased RC delay.

On the other hand, if we have a lot of columns then the load on the column multiplexers and the word lines will increase. They will then become a bottleneck. In addition, placing a highly skewed structure

on the chip is difficult. It conflicts with other structures. Having an aspect ratio (width/length) that is close to that of a square is almost always the best idea from the point of view of placing a component on the chip. Hence, having a balance between the number of rows and columns is a desirable attribute.

It is thus necessary to break down a large array of SRAM transistors into smaller arrays such that they are faster and more manageable. We refer to the large original array as the *undivided array* and the smaller arrays as *subarrays*. Cacti thus introduces two additional parameters: N_{dwl} and N_{dbl}. N_{dwl} indicates the number of segments that we create by splitting each word line or alternatively the number of partitions that we create by splitting the set of columns of the undivided array. On similar lines, N_{dbl} indicates the number of segments that we create by splitting each bit line or the set of rows. After splitting, we create a set of subarrays. In this case, the number of subarrays is equal to $N_{dwl} \times N_{dbl}$.

Additionally, the Cacti tool introduces another parameter called N_{spd}, which basically sets the aspect ratio of the undivided array. It indicates the number of sets that are mapped to a single word line. Let us do some math using our ABC parameters. The size of a block is B, and if the associativity is A, then the size of a set in bytes is $A \times B$. Thus, the number of bytes that are stored in a row (for a single word line) is $A \times B \times N_{spd}$.

Example 5 *Compute the number of rows and columns in a subarray using Cacti's parameters.*

Answer: *Let us compute the number of columns as follows. We have $A \times B \times N_{spd}$ bytes per row in the undivided cache. This is equal to $8 \times A \times B \times N_{spd}$ bits. Now, if we divide the set of columns into N_{dwl} parts, the number of columns in each subarray is equal to $\frac{8 \times A \times B \times N_{spd}}{N_{dwl}}$.*

Let us now compute the number of rows in a subarray. The number of bytes in each row of the undivided cache is equal to $A \times B \times N_{spd}$. Thus, the number of rows is equal to the size of the cache C divided by this number, which is equal to $\frac{C}{A \times B \times N_{spd}}$. Now, if we divide this into N_{dbl} segments, we get the number of rows in each subarray as $\frac{C}{A \times B \times N_{spd} \times N_{dbl}}$.

Thus, given a cache, the task is to compute these three parameters – N_{dwl}, N_{dbl}, and N_{spd}. We need to first figure out a goal such as minimizing the access time or the energy per access. Then, we need to compute the optimal values of these parameters. These parameters were for the data array (d in the subscript). We can define similar parameters for the tag array: N_{twl}, N_{tbl}, and N_{tspd} respectively.

Let us now summarize our discussion. We started out with an array or rather a matrix of SRAM cells. We quickly realized that we cannot have a skewed ratio (disproportionate number of rows or columns). In one case, we will have very slow word lines, and in the other case we will have very slow bit lines. Both are undesirable. Hence, to strike a balance we divide an array of memory cells into a series of subarrays. This is graphically shown in Figure 7.28.

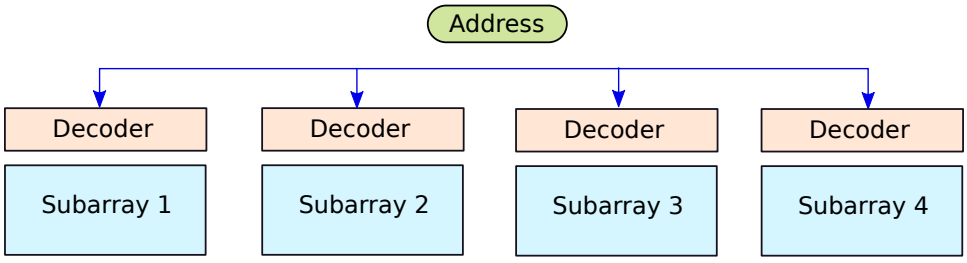

Figure 7.28: An array broken into several subarrays

Each subarray has its own decoder. Recall that the input to the decoder is the set index. If we have 4

subarrays then we can use the last two bits of the set index to index the proper subarray. Subsequently, we expect to find a full set in the row of the subarray. However, in theory it is possible that the set of blocks maybe split across multiple subarrays. In this case, we need to read all the subarrays that contain the blocks in the set. Given that we can divide a large SRAM array in this manner, we will end up accessing subarrays, which are much smaller and faster.

Multiple Parallel Accesses

Let us now complicate the model of our cache. Assume that we need to send multiple read and write requests each cycle. In this case, we need to define a multi-ported structure. A *port* is defined as an interface for accepting a read or write request. We can have a read port, a write port, or a read/write port.

Definition 47
A port *is defined as an interface for accepting a read or write request. We can have a read port, a write port, or a read/write port.*

The traditional approach for creating a multi-ported structure is to connect each SRAM cell to an additional pair of bit lines as shown in Figure 7.29. We thus introduce two additional word line transistors W_3 and W_4 that are enabled by a different word line – this creates a 2-ported structure. Now, since we have two pairs of bit lines, it means that we can make two parallel accesses to the SRAM array. One access will use all the bit lines with subscript 1, and the other access will use all the bit lines with subscript 2. Each pair of bit lines needs its separate set of column multiplexers, sense amplifiers, write, precharge, and output drivers. Additionally, we need two decoders – one for each address. This increases the area of each array significantly. A common thumb rule that is used is that the area of an array increases as the square of the number of ports – proportional increase in the number of word/bit lines in both the axes.

A better solution is a multi-banked cache. A bank is defined as an independent array with its own subarrays and decoders. If a cache has 4 banks, then we split the physical address space between the 4 banks. This can be done by choosing 2 bits in the physical address and then using them to access the right bank. Each bank may be organized as a cache with its own tag and data arrays, alternatively we can divide the data and tag arrays into banks separately. For performance reasons, each bank typically has a single port.

The advantage of dividing a cache into banks is that we can seamlessly support concurrent accesses to different banks. Now, for 4 banks there is a 25% chance of two simultaneous accesses accessing the same bank – assuming a uniformly random distribution of accesses across the banks. This is known as a *bank conflict*, and in this case we need to serialize the accesses. This means that one memory access needs to wait for the other. There is an associated performance penalty. However, this is often outweighed by the fast access time of banks. Finally, note that each bank has its own set of subarrays. However, subarrays cannot be accessed independently.

Hierarchical Organization of Caches

As caches increase in size, the simple organization that we have presented does not remain efficient. Recall that we have shown how to divide a cache into multiple banks and then divide each bank into multiple subarrays.

Let us now look at even larger caches, where we need to further optimize the data and tag arrays [Thoziyoor et al., 2007] separately. Let us focus on the data array, which is much larger than the

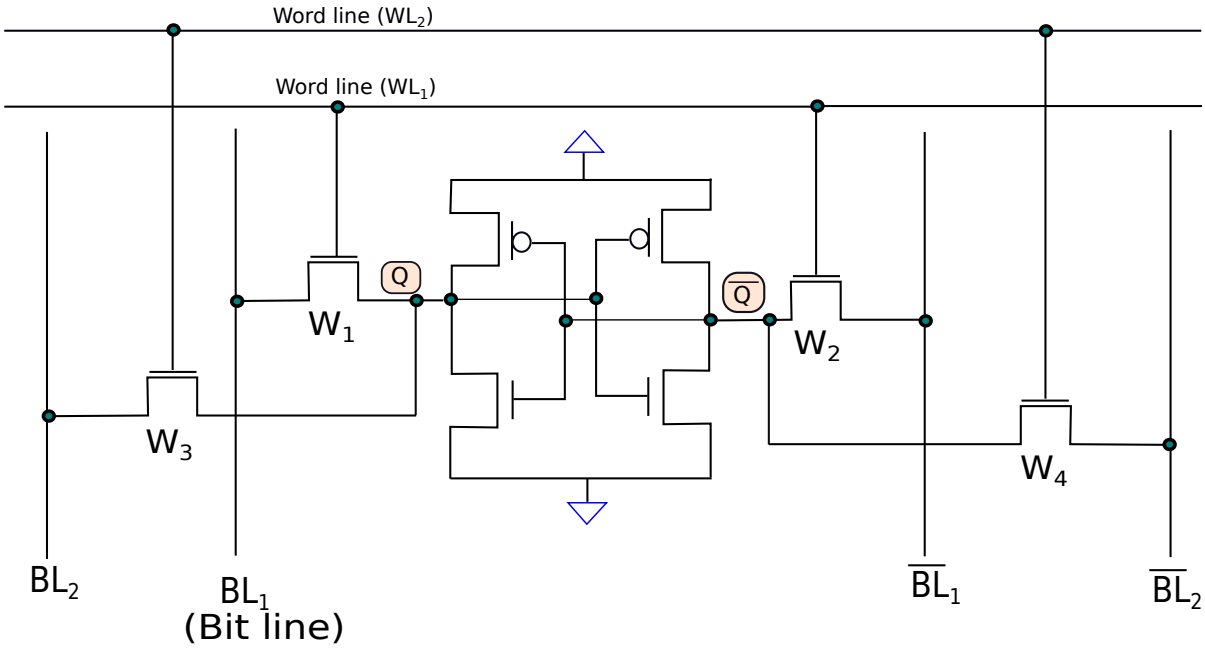

Figure 7.29: SRAM cell with 2 ports

tag array. In Cacti 5.0, the authors propose to divide an array into multiple banks, then further subdivide a bank into subbanks. Banks can be accessed independently by concurrent requests. However, one bank can process only one memory request at any given point of time. Each bank consists of multiple subbanks, and only one of the subbanks can be enabled for a memory request. A subbank contains an entire data block, which is typically either 64 bytes or 128 bytes. Following the maxim, "smaller is faster", we divide a subbank into multiple mats, and each mat into 4 subarrays. The structure is shown in Figure 7.30. The hierarchy is $Array \rightarrow Bank \rightarrow Subbank \rightarrow Mat \rightarrow Subarray$.

The logic for such a deep hierarchy is as follows. If a subbank is one large array, it will be very big and very slow. Hence, we divide a subbank into multiple mats, and we divide each mat into multiple subarrays. We store a part of each block in each mat. For a read operation, each mat supplies the part of the block that it contains and at the subbank level we join all the parts to get the block. We do the same for the subarrays within mats. This process ensures that the mats and their constituent subarrays are small and hence fast. This also parallelizes the process of reading and writing. We can thus quickly read or write an entire data block. Another advantage of this design is that different subarrays within a mat share their decoding logic, which increases the overall speed of operation and minimizes the area.

Routing messages between the cache controller, which is a small piece of logic in each bank, and the subarrays can be complicated in large caches that have long wire delays. Let us outline an approach to solve this problem.

H-Trees

The memory address needs to be sent to all the mats and subarrays. Because wire delays in large caches can be of the order of a few cycles, it is possible that the request might not reach all the mats and subarrays at the same time, particularly if there is a mismatch in the length of wires. If this happens, the responses will not arrive at the same time. To ensure that all the requests reach the subarrays at the same time, we create a network that looks like an H-tree as shown in Figure 7.31. The sender is at the center of the figure. Observe that it is located at the center of the middle bar of a large 'H' shaped

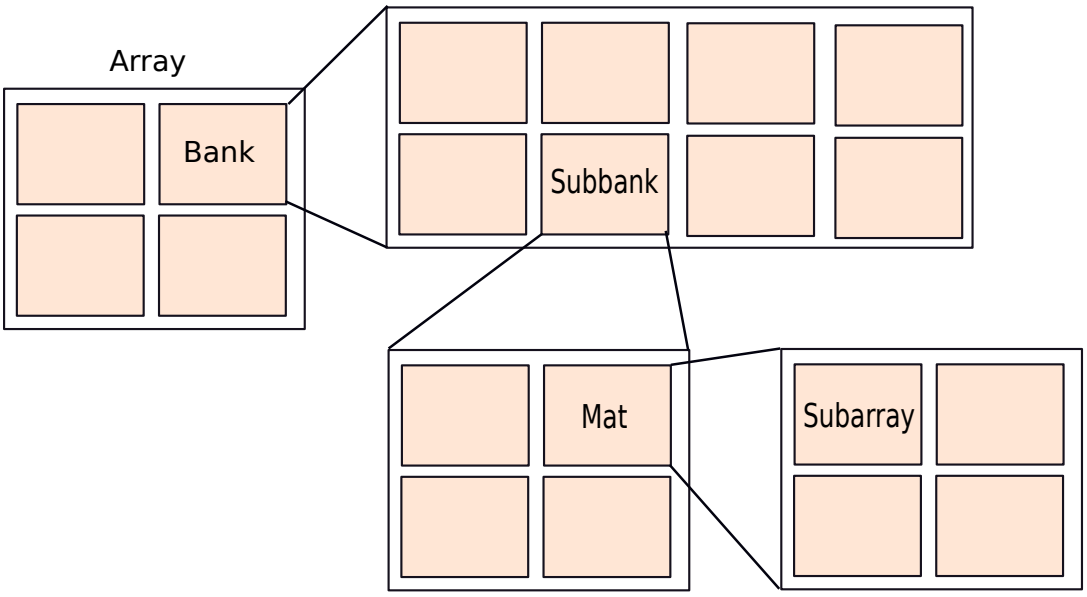

Figure 7.30: Bank → Subbank → Mat → Subarray

subnetwork. Each corner of the large 'H' shaped subnetwork is the center of another smaller 'H' shaped network. This process continues till all the receivers are connected. The reader needs to convince herself that the distance from the center to each receiving node (dark circle) is the same. The address and data are sent along the edges of the H-tree. They reach each subarray at exactly the same time.

7.3.3 Circuit Level Modeling of a Cache: Elmore Delay Model

The challenge now is to model the delay and power consumption of an array of SRAM and CAM cells. There are many components in these cells: transistors in the memory cell, pass transistors, long wires, and complex circuits such as sense amplifiers, decoders, and driver circuits. In general, to simulate such circuits, we use circuit simulation tools. However, using such tools for a large structure such as a cache will take a lot of time. We need simpler and faster models. We are willing to trade accuracy for simulation speed.

Let us make our analysis simpler. Let us replace every circuit element with an equivalent RC circuit made of just resistors and capacitors, which is small and simple. For example, we can replace a long wire with a sequence of resistors and capacitors as shown in Figure 7.32. The logic is that every segment of a wire will have a resistance because it has a finite resistivity, length, and cross-sectional area. In addition, it will also have a capacitance associated with each segment. This is because whenever we have two metallic conductors in proximity, they will act like a classic capacitor – two parallel plates separated by a dielectric. Such a pair of conductors will store some charge across a potential difference. We can model this as a capacitor between the segment of the wire and ground. A wire can thus be visualized as a ladder network of such R and C elements (see Figure 7.32).

Similarly, we can replace a transistor with a set of resistors and capacitors as shown in Figure 7.33. We have a gate capacitance because the gate consists of a conducting plate that is set to a given potential. Thus, it is going to store charge, and this can be modeled as the gate capacitance. Using a similar argument, both the drain and source nodes will also have a capacitance associated with them. When the transistor is in the linear region, the resistance across the channel will not be 0. For a given drain-source and gate voltage, the drain current is given by the Id-Vg curve of the transistor. This relationship can be

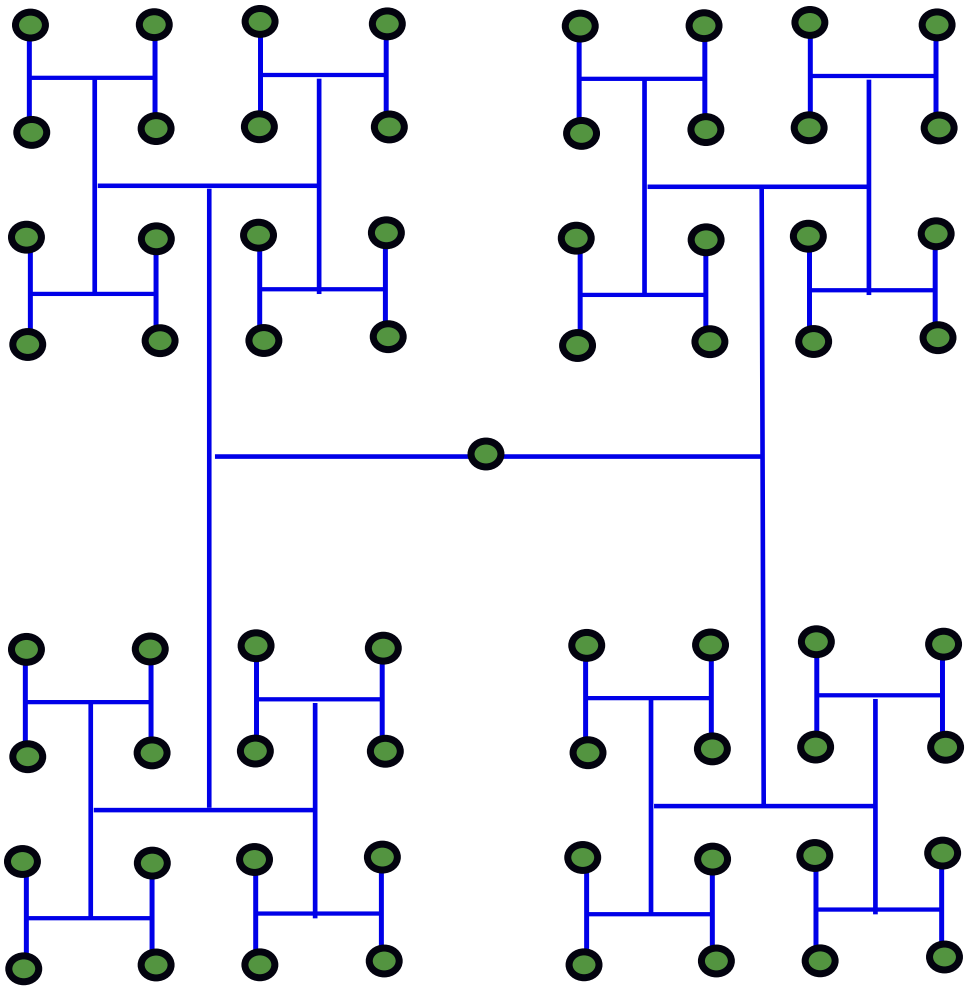

Figure 7.31: H-tree network

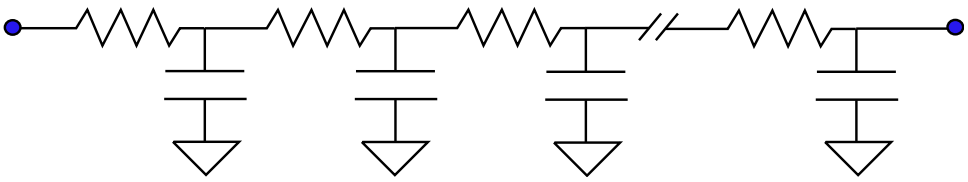

Figure 7.32: Equivalent RC circuit for a wire

modeled by placing a resistor between the drain and the source. When the transistor is in saturation it behaves as a current source and the drain-source resistor can be replaced with a regular current source.

Using such RC networks is a standard approach in the analysis of electronic circuits, particularly when we want to leverage the power of fast circuit simulation tools to compute the voltage at a few given nodes in the circuit. We sometimes need to add an inductance term if long wires are involved. Subsequently, to compute the voltages and currents, it is necessary to perform circuit simulation on these simplified RC networks.

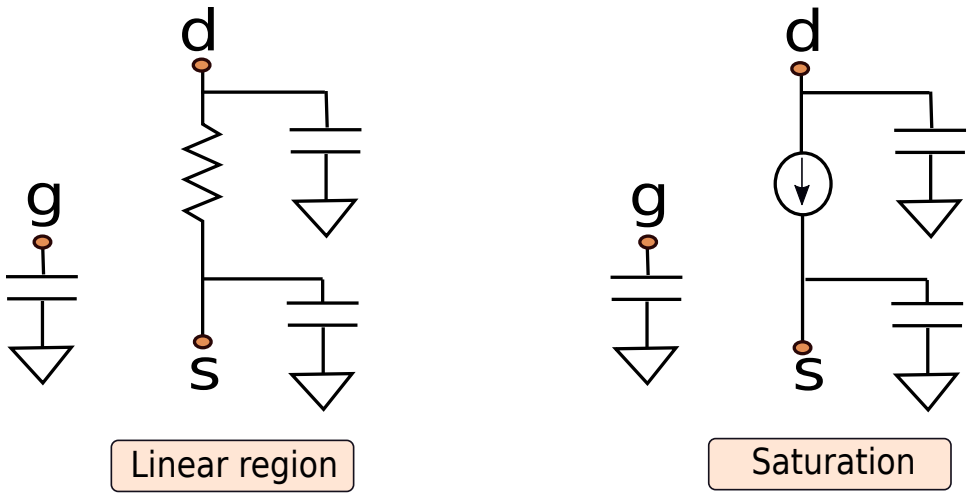

Figure 7.33: Equivalent RC circuit for an NMOS transistor

The Cacti 1.0 [Wilton and Jouppi, 1993] model proposes to replace all the elements in a cache inclusive of the wires, transistors, and specialized circuits with simple RC circuits. Once we have a circuit consisting of just voltage sources, current sources, and RC elements, we can then use quick approximations to compute the voltage at points of interest. In this section, we shall mainly present the results from Horowitz's paper [Horowitz, 1983] on modeling the delay of MOS circuits. This paper in turn bases its key assumptions on Elmore's classic paper [Elmore, 1948] published in 1948. This approach is often referred to as the Elmore delay model.

RC Trees

Let us consider an RC network, and try to compute the time it takes for a given output to either rise to a certain voltage (rise time), or fall to a certain voltage (fall time). For example, our model should allow us to compute how long it will take for the input of the sense amplifier to register a certain voltage after we enable the word lines.

Let us make two assumptions. The first is that we consider an RC tree and not a general RC network. This means that there are no cycles in our network. Most circuits can be modeled as RC trees and only in rare cases where we have a feedback mechanism, we have cycles in our network. Hence, we are not losing much by assuming only RC trees.

The second assumption that we make is that we consider only a single type of voltage sources that provide a step input (see Figure 7.34).

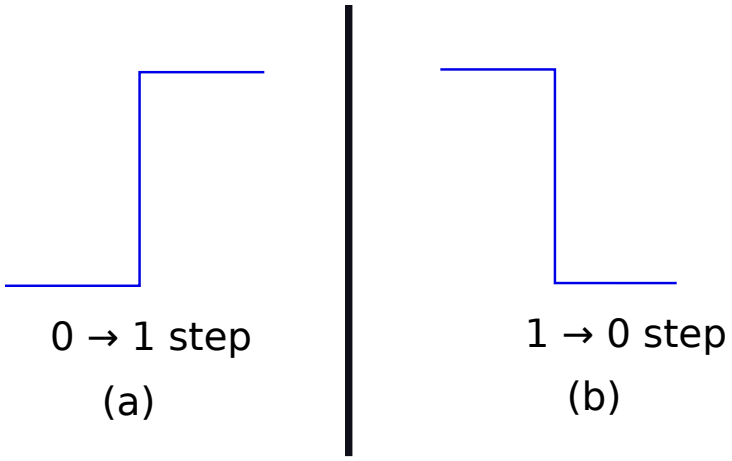

Figure 7.34: A step input

We consider two kinds of such inputs: a $0 \rightarrow 1$ transition and a $1 \rightarrow 0$ transition. In digital circuits we typically have such transitions. We do not transition to any intermediate values. Thus, the usage of the step function is considered to be standard practice. For the sake of simplicity, we assume that a logical 0 is at 0 V and a logical 1 is at 1 V.

Analysis of an RC Tree

Let us consider a generic RC tree as described by Horowitz [Horowitz, 1983]. Consider a single voltage source that can be treated as the input. As discussed, it is a step input that can either make a $0 \rightarrow 1$ transition or a $1 \rightarrow 0$ transition. Let us assume that it makes a $1 \rightarrow 0$ transition (the reverse case is analogous).

Let us draw an RC tree and number the resistors and capacitors (see Figure 7.35). Note that between an output node and the voltage sources we only have a series of resistors, we do not have any capacitors. All the capacitors are between a node and ground.

Each capacitor can be represented as a current source. For a capacitor with capacitance C, the charge that it stores is $V(t)C$, where $V(t)$ is the voltage at time t. We assume that the input voltage makes a transition at $t = 0$. Now, the current leaving the capacitor is equal to $-CdV(t)/dt$. Let us draw an equivalent figure where our capacitors are replaced by current sources. This is shown in Figure 7.36.

The goal is to compute $V_x(t)$, where x is the number of the output node (shown in an oval shaped box in the figure). Let us show how to compute the voltage at node 3 using the principle of superposition. If we have n current sources, we consider one at a time. When we are considering the k^{th} current source we disconnect (replace with an open circuit) the rest of the $n-1$ current sources. This reduced circuit has just one current source. We then proceed to compute the voltage at node 3.

In this RC tree, only node 0 is connected to a voltage source, which makes a $1 \rightarrow 0$ transition at $t = 0$. The rest of the nodes are floating. As a result the current will flow towards node 0 via a path consisting exclusively of resistors.

Now, assume that the current source at node 4 is connected, and the rest of the current sources are replaced with open circuits. The current produced by the current source is equal to $-C_4 dV_4^4/dt$. The term V_i^j refers to the voltage at terminal i because of the effect of the current source placed at terminal j using our methodology. The voltage at node 1 is therefore $-R_1 C_4 dV_4^4/dt$. Since the rest of the nodes are floating, this is also equal to the voltage at node 3. We thus have:

$$V_3^4 = -R_1 C_4 \frac{dV_4^4}{dt} \tag{7.5}$$

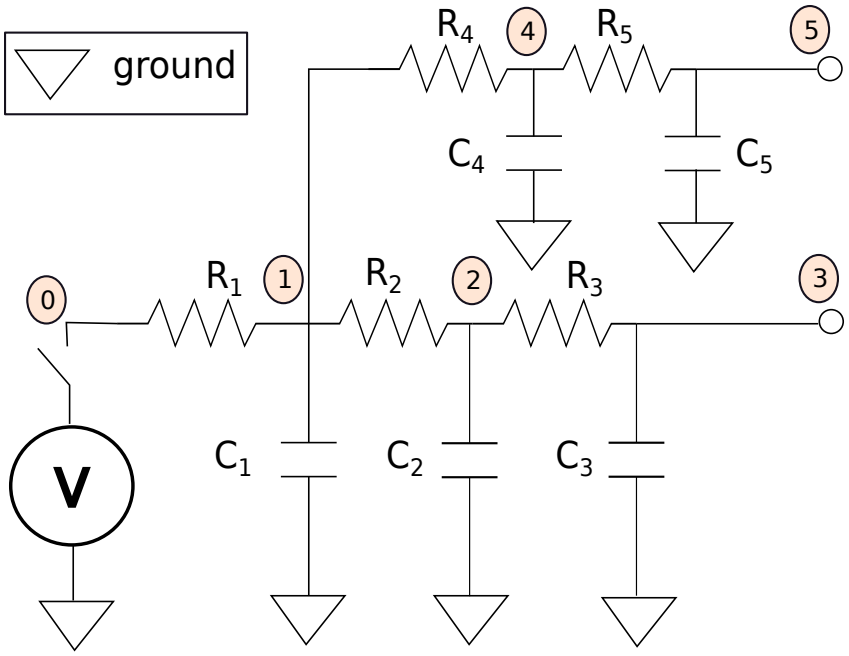

Figure 7.35: RC tree

Let us do a similar analysis when the current source attached to node 2 is connected. In this case, the voltage at node 2 is equal to the voltage at node 3. The voltage at node 2 or 3 is given by the following equation:

$$V_2^2 = V_3^2 = -(R_1 + R_2)C_2\frac{dV_2^2}{dt} \tag{7.6}$$

Let us generalize these observations. Assume we want to compute the voltage at node i, when the current source attached to node j is connected. Now consider the path between node 0 (voltage source) and node i. Let the set of resistors on this path be P_{0i}. Similarly, let the set of resistors in the path from node 0 to node j be P_{0j}. Now, let us consider the intersection of these paths and find all the resistors that are in common. These resistors are given by

$$P_{ij} = P_{0i} \cap P_{0j} \tag{7.7}$$

We can easily verify that $P_{34} = \{R_1\}$, and $P_{23} = \{R_1, R_2\}$. Let R_{ij} be equal to the sum of all the resistors in P_{ij}. Formally,

$$R_{ij} = \sum_{R \in P_{ij}} R \tag{7.8}$$

Now, please convince yourself that Equations 7.5 and 7.6 are special cases of the following equation. Assume that we only consider the current source at node j.

$$V_i^j = -R_{ij}C_j\frac{dV_j^j}{dt} \tag{7.9}$$

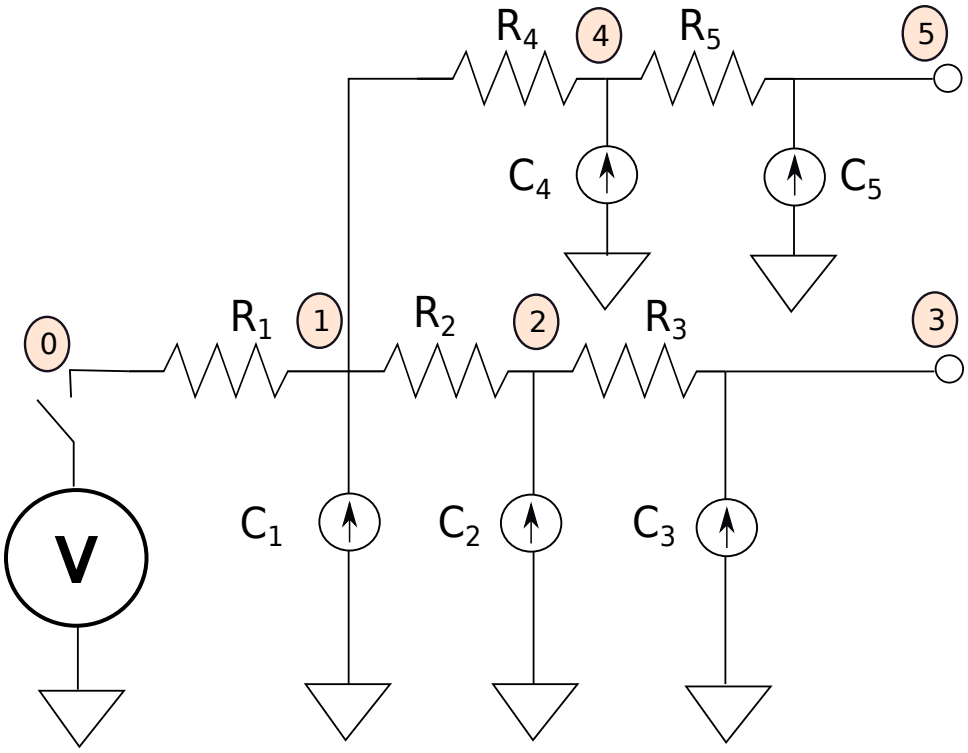

Figure 7.36: Equivalent RC tree (replacing capacitors with current sources)

Now, if we consider all the capacitors one by one and use the principle of superposition, we compute V_i to be a sum of the voltages at i computed by replacing each capacitor with a current source.

$$V_i = \sum_j V_i^j = \sum_j -R_{ij} C_j \frac{dV_j^j}{dt} \tag{7.10}$$

Unfortunately, it is hard to solve a system of simultaneous differential equations, that too quickly and accurately. It is therefore imperative that we make some approximations.

This is exactly where Elmore [Elmore, 1948] proposed his famous approximation. Let us assume that $dV_j^j/dt = \alpha dV_i/dt$, where α is a constant. This is also referred to as the single pole approximation (refer to the concept of poles and zeros in electrical networks). Using this approximation, we can compute the voltage at node i to be

$$V_i^* = \sum_j -\alpha R_{ij} C_j \frac{dV_i}{dt} \tag{7.11}$$

Here, V_i^* is the voltage at node i computed using our approximation. Let us now consider the error

$\int (V_i - V_i^*)dt$. Assume $\alpha = 1$ in the subsequent equation. We have

$$\begin{aligned} V_i - V_i^* &= \sum_j -R_{ij}C_j \frac{dV_j^j}{dt} - \sum_j -R_{ij}C_j \frac{dV_i}{dt} \\ &= \sum_j -R_{ij}C_j (\frac{dV_j^j}{dt} - \frac{dV_i}{dt}) \\ \int (V_i - V_i^*)dt &= \sum_j -R_{ij}C_j \int \left(\frac{dV_j^j}{dt} - \frac{dV_i}{dt}\right) dt \\ &= \sum_j -R_{ij}C_j (V_j^j - V_i)\Big|_0^\infty \\ &= 0 \end{aligned} \tag{7.12}$$

Note that the expression, $(V_j^j - V_i)\Big|_0^\infty = 0$, because both V_i and V_j^j start from the same voltage (1 V in this case) and end at the same voltage (0 V in this case). Thus, we can conclude that the error $\int (V_i - V_i^*)dt = 0$, when we assume that $dV_j^j/dt = dV_i/dt$ for all i and j. This is the least possible error, and thus we can conclude that our approximation with $\alpha = 1$ minimizes the error as we have defined it (difference of the two functions). Let us now try to solve the equations for any V_i using our approximation. Let us henceforth not use the term V_i^*. We shall use the term V_i (voltage as a function of time) to refer to the voltage at node i computed using Elmore's approximations.

We thus have

$$\begin{aligned} V_i &= \sum_j -R_{ij}C_j \frac{dV_i}{dt} \\ &= -\tau_i \frac{dV_i}{dt} \qquad (\tau_i = \sum_j R_{ij}C_j) \\ \Rightarrow &\frac{dt}{\tau_i} = -\frac{dV_i}{V_i} \\ \Rightarrow &\frac{t}{\tau_i} - ln(k) = -ln(V_i) \qquad \text{ln(k) is the constant of integration} \\ \Rightarrow &V_i = ke^{-\frac{t}{\tau_i}} \\ \Rightarrow &V_i = V_0 e^{-\frac{t}{\tau_i}} \qquad \text{at } t = 0,\ V_i = V_0 = k \end{aligned} \tag{7.13}$$

V_i thus reduces exponentially with time constant τ_i. Recall that this equation is similar to a capacitor discharging in a simple RC network consisting of a single resistor and capacitor. Let us now use this formula to compute the time it takes to discharge a long copper wire (see Example 6).

Example 6
Compute the delay of a long copper wire.

Answer: *Let us divide a long copper wire into n short line segments. Each segment has an associated resistance and capacitance, which are assumed to be the same for all the segments.*

Let the total resistance of the wire be R and the total capacitance be C. Then the resistance and capacitance of each line segment is R/n and C/n respectively. Let terminal i be the end point of segment i. The time constant measured at terminal n is given by Equation 7.13. It is equal to

$$V_n = \sum_j -R_{nj} C_j \frac{dV_n}{dt} \tag{7.14}$$

Any R_{nj} in this network is equal to $\sum_{i=1}^{j} R_i$. R_i and C_i correspond to the resistance and capacitance of the i^{th} line segment respectively. We can assume that $\forall i, R_i = R/n$ and $\forall i, C_i = C/n$. Hence, $R_{nj} = jR/n$. The time constant of the wire is therefore equal to

$$\begin{aligned} \tau &= \sum_{j=1}^{n} R_{nj} C_j = \frac{C}{n} \sum_{j=1}^{n} \frac{R}{n} \times j \\ &= \frac{C}{n} \times \frac{R}{n} \times \frac{n(n+1)}{2} = \frac{C}{n} \times R \times \frac{n+1}{2} \\ &= RC \times \frac{n+1}{2n} \end{aligned} \tag{7.15}$$

As $n \to \infty$, $\tau \to \frac{RC}{2}$. We can assume that the time constant of a wire is equivalent to that of a simple RC circuit that has the same capacitance and half the resistance of the wire (or vice versa).

We can draw some interesting conclusions from Example 6. The first is that the time constant of a wire is equal to $RC/2$, where R and C are the resistance and capacitance of the entire wire respectively. Let the resistance and capacitance for a small segment of the wire be r and c respectively. Then, we have the following relations.

$$\begin{aligned} R &= nr \\ C &= nc \end{aligned} \tag{7.16}$$

Hence, the time constant, τ, is equal to $rcn^2/2$. Recall that the time constant is the time it takes for the input to rise to 63% of its final value $(1 - 1/e)$, or the output to fall to 37% of the maximum value $(1/e)$. We can extend this further. A typical RC circuit charges or discharges by 98% after 4τ units of time. After 5τ units of time, the final voltage is within 0.7% of its final value. If we set a given threshold for the voltage for deciding whether it is a logical 0 or 1, then the time it takes to reach that threshold can be expressed in terms of time constants. It is common to refer to the time a circuit takes to respond to an input in terms of time constants.

Now, given that $\tau = rcn^2/2$ for a long wire, we can quickly deduce that **the delay is proportional to the square of the wire's length**. This is bad news for us because it means that long wires are not scalable and thus should not be used.

Dealing with Slow Inputs

Up till now we have been making two critical assumptions. First, the input is a step function, and second, the devices can be replaced with linear elements such as resistors and capacitors. Unfortunately, both these assumptions do not exactly hold with real MOS transistors, and sometimes the errors can be unacceptably large. Let us thus replace step sources with arbitrary sources, and call them *slow inputs* (same terminology used by Horowitz [Horowitz, 1983]).

To consider circuits that do not have step inputs and have a non-linear response, the authors of the Cacti tool use the Horowitz approximation for non-linear circuits. This is the equation of the rise time for an inverter.

$$delay_{rise} = \tau\sqrt{(log(v_{th})^2 + 2t_{rise}b(1 - v_{th})/\tau} \tag{7.17}$$

τ is the time constant assuming a step input, v_{th} is the threshold voltage as a fraction of the supply voltage, t_{rise} is the rise time of the input, and b is the fraction of the input's swing at which the output changes (Cacti 1 uses a value of $b = 0.5$). We have a similar equation for the time it takes for an input to fall.

$$delay_{fall} = \tau\sqrt{(log(v_{th})^2 + 2t_{fall}b(1 - v_{th})/\tau} \tag{7.18}$$

Equations 7.17 and 7.18 are primarily based on empirical models that describe the behavior of transistors in the linear and saturation regions. These equations can change with the transistor technology and are thus not fundamental principles. Hence, it is necessary to change these equations appropriately if we are trying to use a different kind of transistors.

Example: Delay of a Bit Line

Let us show the calculation of the delay of a bit line in the Cacti 1 model when the bit line goes low (towards 0 V). The equivalent circuit used by the Cacti model is shown in Figure 7.37.

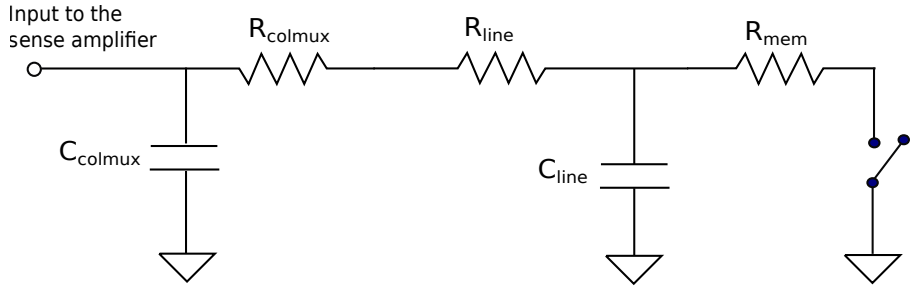

Figure 7.37: Equivalent circuit of a bit line(adapted from [Wilton and Jouppi, 1993])

R_{mem} is the combined resistance of the word line transistor and the NMOS transistor in the memory cell (via which the bit line discharges). These transistors connect the bit line to the ground. C_{line} is the effective capacitance of the entire bit line. This includes the drain capacitance of the pass transistors (controlled by the word lines), the capacitance that arises due to the metallic portion of the bit line, the drain capacitances of the precharge circuit and the column multiplexer.

R_{colmux} and C_{colmux} represent the resistance of the pass transistor in the column multiplexer and the output capacitance of the column multiplexer respectively.

R_{line} needs some explanation. Refer to Example 6, where we had computed the time constant of a long wire to be approximately $RC/2$, where R and C are its resistance and capacitance respectively. A model that treats a large object as a small object with well defined parameters is known as a lumped model. In this case, the lumped resistance of the entire bit line is computed as follows:

$$R_{line} = \frac{\#rows}{2} \times R_{segment} \tag{7.19}$$

Here, $R_{segment}$ is the resistance of the segment of a bit line corresponding to one row of SRAM cells. We divide it by 2 because the time constant in the lumped model of a wire is $RC/2$. We need to divide either the total resistance or capacitance by 2.

Using the Elmore delay model the time constant (τ) is equal to $R_{mem}C_{line}$ + $(R_{mem} + R_{line} + R_{colmux})C_{colmux}$.

Example: Delay of a Word Line

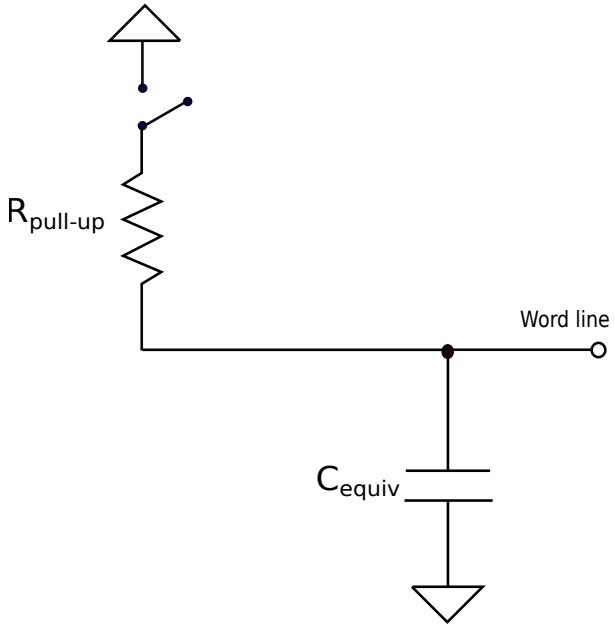

Figure 7.38: Equivalent circuit for a word line (adapted from [Wilton and Jouppi, 1993])

Figure 7.38 shows the equivalent circuit for a word line. In this circuit $R_{pull-up}$ is the pull-up resistance. The internal resistance of the word line drivers determine the value of this parameter.

C_{equiv} is the equivalent capacitance of the word line. It is given by the following equation.

$$C_{equiv} = \#cols \times (2 \times C_{gatewl} + C_{metal}) \tag{7.20}$$

The parameter $\#cols$ refers to the number of columns in a row. C_{gatewl} is the gate capacitance of each pass transistor that is controlled by the word line. Since there are two such transistors per memory cell, we need to multiply this value by 2. Finally, C_{metal} is the capacitance of just the metallic portion of the word line. C_{equiv} is a sum of all these parameters.

The time constant in this case is simply $R_{pull-up} \times C_{equiv}$.

7.4 Advanced Cache Design

In this section we shall look at some common methods to increase the performance of a cache.

7.4.1 Pipelined Caches

Assume that the L1 d-cache (data cache) has a hit latency of 3 cycles. In this case, there are two options. The first is that if a cache is processing a request, then it does not process any other requests till it is done. This is not efficient in terms of performance. The other scheme is that the cache itself is pipelined. This means that it can accept a new request every cycle akin to normal processor pipelines. The pipelined design is needless to say more efficient because it provides a greater throughput. In other words, if a cache can start the processing of a new set of requests every cycle, it can match the rate at which the processor produces memory requests in the worst case. Thus, memory requests will not queue up at the side of the processor.

Let us look at the set of steps that need to be performed to pipeline a cache. The first task is to decompose the work of a cache into multiple subtasks. Figure 7.39 shows a representative example for

a read operation. We first perform a data array access and a tag array access in parallel. This is done for performance reasons. At the outset we don't know which way of a set will match. Hence, we read all the data blocks in parallel, and choose one of them later if there is a match. The advantage in this case is that the process of reading the data blocks is off the critical path. It can be overlapped with reading the tags. Subsequently, we start the process of tag comparison, and immediately after that we choose the right data block based on the results of the tag comparison. Since we have read all the data blocks in the set in advance, we need not access the data array again. We simply choose one of the blocks that has been read.

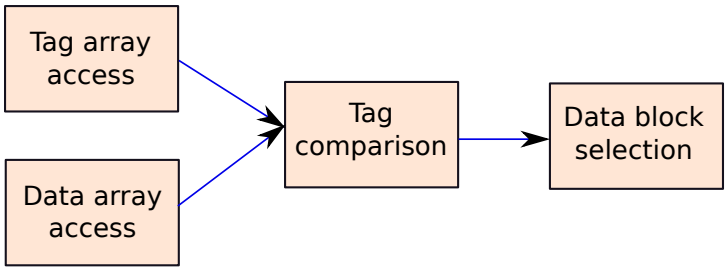

Figure 7.39: Subtasks in a read operation

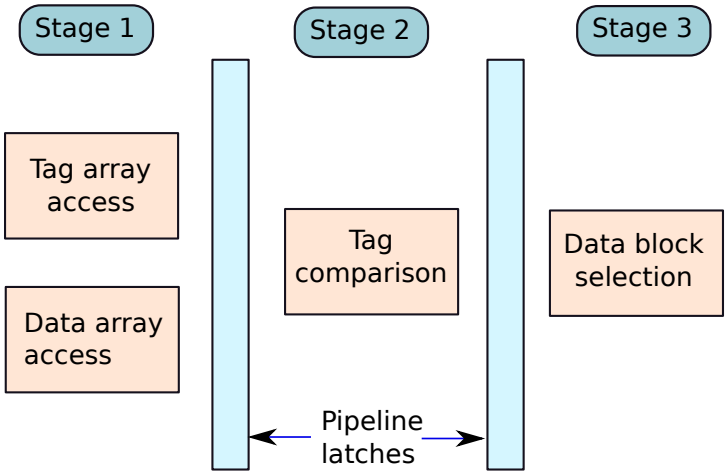

Figure 7.40: Design of a pipelined cache (for the read operation)

We need to add pipeline latches or buffers between these stages (similar to what we had done in an in-order pipeline) to create a pipelined cache. The resulting structure is shown in Figure 7.40. Of course, we are making many simplistic assumptions in this process, notably that the time it takes to complete each subtask (stage) is roughly the same. Sometimes we might wish to create more pipeline stages.

In pursuance of this goal, it is possible to break the SRAM array access process into two stages: address decode and row access. If the decoder has N outputs, then we can create a large N-bit pipeline latch to temporarily store its output. In the next stage we can access the target row of the SRAM array. This will increase the depth of the pipeline to 4 stages. It is typically not possible to pipeline the row access process because it is basically an analog circuit.

Even though the exact nature of pipelining may differ, the key idea here is that we need to pipeline the cache to ensure that it does not lock up while processing a request. The process of pipelining ensures

a much larger throughput, which is mandatory in high performance systems.

Now, if we consider write accesses as well, then they can be pipelined similarly: first access the tag array, then compare the tag part of the address with all the tags in the set, and finally perform the write. If the subsequent access is a read, then we require forwarding; therefore forwarding paths similar to in-order processors are required here as well. Working out the details of these forwarding paths is left as an exercise for the reader.

7.4.2 Non-blocking Caches

Let us consider the interface between the core and the memory system. After we issue a load or a store to the memory system, we access the i-cache for instructions and the d-cache for data. If there is a hit, then we can immediately supply the data to the processor. In Section 7.4.1, we discussed how to increase the throughput of a cache by pipelining it. However, all of this was done to make cache hits faster. We did not talk about misses. Let us now see what happens in the case of misses.

Whenever there is a miss in the cache, we need to fetch the block from the lower level, irrespective of whether we need to perform a read or a write operation. The key question is, "Do we block the cache during this time?" or do we proceed with other requests. In modern processors, we always choose the latter option, where we proceed with other requests. Even if there is a miss, or there are several outstanding misses that are waiting to get their data, we proceed with other requests. This is also known as the, "access under multiple misses," assumption. Such a cache that never blocks is known as a non-blocking cache. It is an essential part of all high performance processors as of 2020.

Note that even if this sounds as a very worthy goal, implementing such a cache is difficult. Assume that we suffer a miss because we wanted to access the fourth word (4 or 8 bytes) in a block. In a non-blocking cache, we do not lock the cache till the miss returns. Instead, we let other accesses go through. Assume that at a later point in time we suffer a miss for the fifth word in the same block. In this case, if we also allow this miss to go through then we are in a sense doubling the requests made to the lower level as compared to a blocking cache. Since we have spatial locality, we are expected to access many more words in the block before the original miss returns with the data. The unwanted side effect of a non-blocking cache is that we significantly increase the requests made to the lower level cache. We stand to lose all the gains we wanted to make by having non-blocking caches.

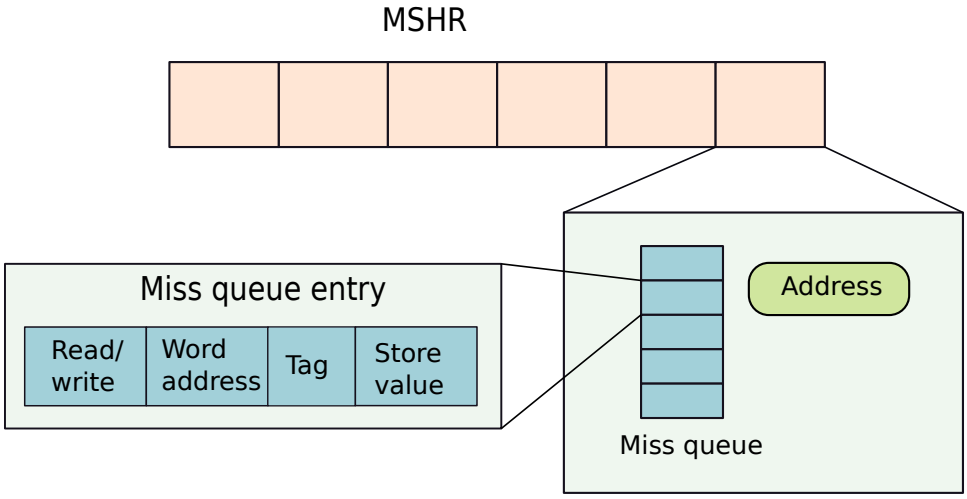

Figure 7.41: Design of an MSHR

Let us thus propose a new structure called an MSHR (miss status holding register) [Kroft, 1981,

Scheurich and Dubois, 1988]. We associate an MSHR with each cache. The structure of an MSHR is shown in Figure 7.41. It consists of a set of arrays. Whenever we miss in the cache, and there are no other pending misses for that block, we refer to such a miss as a *primary miss*. Upon a primary miss, we allocate an empty array in the MSHR to this miss. In the MSHR entry, we store the block address. Then, we initialize the first array entry that stores the type of the access (read or write), address of the word within the block (word address) that the current access refers to, and the destination register (*tag*) of a load instruction or the *value* that a store instruction writes to memory. The miss request is subsequently dispatched to the lower level. Let us refer to each such array as a miss queue. The reason for calling it a miss queue will be clear after we have described the operation of the MSHR.

Before the miss has returned, we might have several additional misses for other words in the block. These are called *secondary misses*. The method to handle secondary misses is conceptually similar to the way we handled memory requests in the LSQ. Assume that a secondary miss is a write. We create an entry at the tail of the miss queue, which contains the value that is to be written along with the address of the word within the block. Now, assume that the secondary miss is a read. In this case, if we are reading a single memory word, then we first check the earlier entries in the miss queue to see if there is a corresponding write. If this is the case, then we can directly forward the value from the write to the read. There is no need to queue the entry for the read. However, if such forwarding is not possible, then we create a new entry at the tail of the miss queue and add the parameters of the read request to it. This includes the details of the memory request such as the id of the destination register (in the case of the L1 cache) or the id of the requesting cache (at other levels) – referred to as the *tag* in Figure 7.41.

The advantage of an MSHR is that instead of sending multiple miss requests to the lower level, we send just one. In the time being the cache continues to serve other requests. Then, when the primary miss returns with the data, we need to take a look at all the entries in the miss queue, and start applying them in order. After this process, we can write the modified block to the cache, and return all the read/write requests to the upper level.

Now let us account for the corner cases. We might have a lot of outstanding memory requests, which might exhaust the number of entries in a miss queue, or we might run out of miss queues. In such a scenario, the cache needs to lock up and stop accepting new requests.

7.4.3 Skewed Associative Caches

The basic philosophy in a regular k-way set associative cache is that given the address of a block, there are k locations in the cache that might potentially keep its data. We check all the k locations and compare the tag part of the address with the tags that are stored in these locations. The problem with this approach is as follows. Assume there are $k+1$ blocks that map to the same set, and are frequently accessed. They will continue to create conflict misses. Some of these penalties can be absorbed by using a victim cache (see Section 7.1.7); however, if there are many such sets, then a victim cache is useless.

This is not a contrived scenario. Even if we consider a workload with a small working set (see Section 7.1.1), there will always be some sets that see a lot of contention. There is no way to offload this contention to sets that are relatively less contended. This is where a skewed associative cache [Seznec, 1993] can help. It introduces a different kind of thinking.

To start the discussion, let us consider a traditional 2-way set associative cache first. Given a block, we first find its set index, which is a subset of the bits in the block address. Let it set index be I. Then, the two locations that can potentially contain a copy of the block are $2I$ and $2I+1$. If two blocks with addresses A_1 and A_2 have the same set index, then both of them will vie for the same locations on the cache: $2I$ and $2I+1$. If we have three such blocks that are accessed frequently, then we will have a large number of misses. Let us do something to minimize the contention by designing a skewed associative cache.

Let us partition a cache into two subcaches, and use two different functions – f_1 and f_2 – to map a block to lines in the different subcaches. This means that if a block has address A, we need to search for it in the lines $f_1(A)$ and $f_2(A)$ in the two subcaches, respectively. Let function f_k refer to a line in the k^{th}

subcache. The crux of the idea is to ensure that for two block addresses, A_1 and A_2, if $f_1(A_1) = f_1(A_2)$ (in subcache 1), then $f_2(A_1) \neq f_2(A_2)$ (in subcache 2). In simple terms, if two blocks have a conflict in one subcache, they should have a very high likelihood of not conflicting in the other subcache. We can easily extend this idea to a cache with k subcaches. We can create separate mapping functions for each subcache to ensure that even if a set of blocks have conflicts in a few subcaches, they do not conflict in the rest of the subcaches. To implement such a scheme, we can treat each bank as a subcache.

The operative part of the design is the choice of functions to map block addresses to lines in subcaches. The main principle that needs to be followed is that if two blocks map to the same line in subcache 1, then their probability of mapping to the same line in subcache 2 will be $1/N$, where N is the number of lines in a subcache. The functions $f_1()$ and $f_2()$ are known as *skewing functions*.

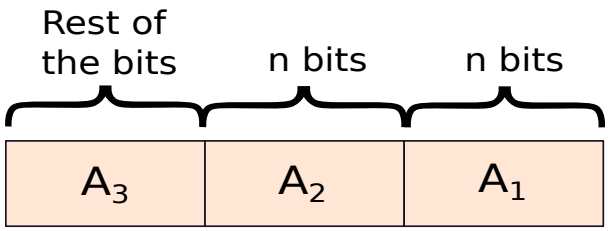

Figure 7.42: Breakup of a memory address in a skewed associative cache

Let us discuss the skewing functions described by Bodin and Seznec [Bodin and Seznec, 1997]. We divide a block address into three parts as shown in Figure 7.42. Assume each subcache has 2^n cache lines. We create three chunks of bits: A_1 (lowest n bits), A_2 (n bits after the bits in A_1), and A_3 (rest of the MSB bits).

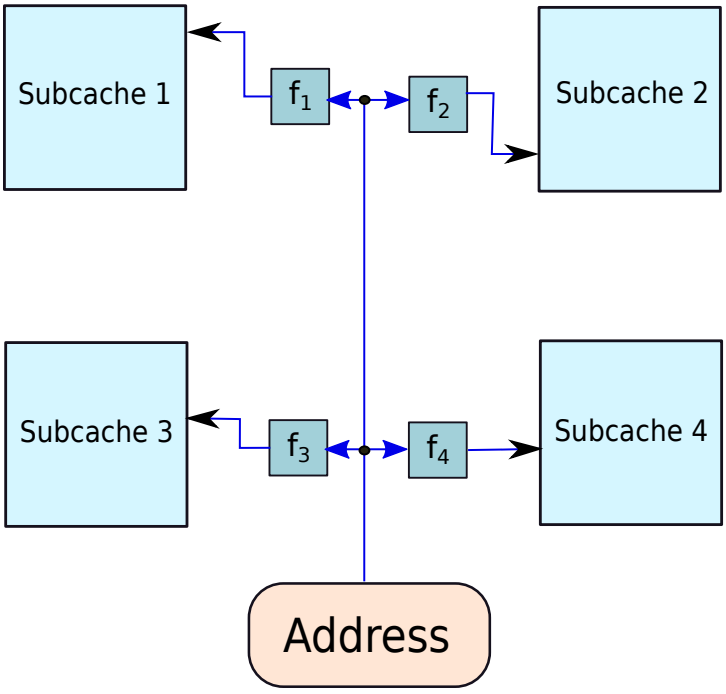

Figure 7.43: Skewed associative cache

The authors use a function σ that shuffles the bits similar to shuffling a deck of playing cards. There are several fast algorithms in hardware to shuffle a set of bits. Discussing them is out of the scope of the book. For a deeper understanding of this process, readers can refer to the seminal paper by Diaconis [Diaconis et al., 1983]. For a 4-way skewed associative cache (see Figure 7.43), where each logical subcache is mapped to a separate bank, the four mapping functions for block address A are as follows. The $\oplus$ sign refers to the XOR function.

Bank 1 $f_1(A) = A_1 \oplus A_2$

Bank 2 $f_2(A) = \sigma(A_1) \oplus A_2$

Bank 3 $f_3(A) = \sigma(\sigma(A_1)) \oplus A_2$

Bank 4 $f_4(A) = \sigma(\sigma(\sigma(A_1))) \oplus A_2$

These functions can be computed easily in hardware, and we can thus reduce the probability of conflicts to a large extent as observed by Bodin and Seznec. In a skewed associative cache, let us refer to the locations at which a block can be stored as its *set*. A set is distributed across subcaches.

The last piece remaining is the replacement policy. Unlike a traditional cache, where we can keep LRU timestamps for each set, here we need a different mechanism. We can opt for a very simple pseudo-LRU policy. Assume we want to insert a new block, and all the cache lines in its set are non-empty. We ideally need to find the line that has been accessed the least. One approach to almost achieve this is to have a bit along with each cache line. When the cache line is accessed this bit is set to 1. Periodically, we clear all such bits. Now, when we need to evict a line out of the set of k lines in a k-way skewed associative cache, we can choose that line whose bit is 0. This means that it has not been accessed in the recent past. However, if the bits for all the lines where the given block can be inserted are set to 1, then we can randomly pick a block as a candidate for replacement.

We can always do something more sophisticated. This includes associating a counter with each line, which is decremented periodically, and incremented when the line is accessed (similar to classical pseudo-LRU). We can also implement *Cuckoo hashing*. Assume that for address A, all the lines in its set are non-empty. Let a block with address A' be present in its set (in subcache 2). It is possible that the line $f_1(A')$ in subcache 1 is empty. Then the block with address A' can be moved to subcache 1. This will create an empty line in the set, and the new block with address A can be inserted there. This process can be made a cascaded process, where we remove one block, move it to one of its alternative locations, remove the existing block in the alternative location, and try to place it in another location in its set, until we find an empty line.

7.4.4 Way Prediction

We have discussed two kinds of cache addressing schemes: a regular set-index based approach, and the skewed associative approach. In both the cases there are multiple ways associated with each set, and we unfortunately need to read the tags in all the ways to compute a tag match. This is wasteful in terms of both time and energy. It would be a much better idea to predict the way in the set that most likely contains the data and access it first. If there is a miss, then we access the rest of the ways using the regular method. The efficacy of this approach completely depends on the accuracy of the predictor.

Let us describe two simple way predictors (see [Powell et al., 2001]). The main idea is to predict the way in which we expect to find the contents of a block before we proceed to access the cache. If after computing the memory address, it will take a few cycles to translate it and check the LSQ for a potential forwarding opportunity, then we can use the memory address itself for way prediction. Let us thus consider the more difficult case where we can access the cache almost immediately because address translation and LSQ accesses are not on the critical path either because of speculation or because they are very fast. In this case, we need to use information other than the memory address to predict the way.

For predicting the way in advance, the only piece of information that we have at our disposal is the PC of the load or the store. This is known well in advance, and thus we can use a similar table as we had used for branch prediction or value prediction to predict the way. For a k-way set associative cache, we need to store $log_2(k)$ bits per entry. Whenever we access the cache, we first access the predicted way. If we do not find the entry there, then we check the rest of the ways using the conventional approach. Let us take a look at the best case and the worst case. The best case is that we have a 100% hit rate with the way predictor. In this case, our k-way set associative cache behaves as a direct mapped cache. We access only a single way. The energy is also commensurately lower.

Let us consider the worst case at the other end of the spectrum, where the hit rate with the predicted way is 0%. In this case, we first access the predicted way, and then realize that the block is not contained in that way. Subsequently, we proceed to access the rest of the ways using the conventional cache access mechanism. This is both a waste of time and a waste of energy. We unnecessarily lost a few cycles in accessing the predicted way, which proved to be absolutely futile. The decision of whether to use a way predictor or not is thus dependent on its accuracy and the resultant performance gains (or penalties).

XOR based Approach

Let us now discuss another method that has the potential to be more accurate than the PC based predictor. The main problem with the PC based predictor is that the computed memory addresses can keep changing. For example, with an array access, the memory address keeps getting incremented, and thus predicting the way becomes more difficult. It is much easier to do this with the memory address of the load or store. However, we do not want to introduce another way prediction stage between the stage that computes the value of the memory address and the cache access because this is on the critical path.

Let us find a solution in the middle, where we can compute the prediction just before the cache access. Recall that after we read the registers, the values are sent to the execution unit. There is thus at least a single cycle gap between the time at which the value of the base register is available, and the time at which the address reaches the L1 cache. We have the execute stage in the middle that adds the offset to the address stored in the base register. Let us use this time productively. Once we have the value of the base register, let us compute a XOR between this value and the offset.

Consider an example: a load instruction *ld r2, 12[r1]*. In this case, the base register is $r1$. We read the value V of $r1$ in the register read stage. Subsequently, we add 12 to V and this becomes the memory address of the load instruction. Next, we need to update the corresponding entry in the LSQ and check for store→load forwarding. Meanwhile, let us compute the XOR of V and 12. We compute $R = V \oplus 12$. Similar to what we had done in the GShare predictor, the result R of the XOR instruction in some sense captures the value of the base register and the offset. Computing a XOR is much faster than doing an addition and possibly checking for forwarding in the LSQ. Thus, in the remaining part of the clock cycle we can access a 2^n-entry way prediction cache that is organized in a manner similar to a last value predictor (see Section 5.1.5) using n bits from the result, R. Once the address has been computed, we can access the cache using the predicted way. This approach uses a different source of information, which is not the memory address, but is a quantity that is similar to it in terms of information content.

7.4.5 Loop Tiling

We have looked at several designs based exclusively in hardware. Let us now look at software based approaches, and consider the most popular optimization technique in this space. It is known as *loop tiling* or *blocking*. It is a compiler based technique to improve the temporal locality of data accesses in matrices.

Let us provide a brief background. Linear algebra operations using matrices are integral to many scientific and numerical computations. They are also the core operations in image processing and computer graphics. As a result, there is a lot of demand for processors that execute matrix operations

very quickly. Since such operations form the kernel of most numerical algorithms, a lot of end-user applications will gain from speeding up such operations.

One of the most important matrix based operations is matrix multiplication. Let us look at a naive implementation of matrix multiplication as shown in Listing 7.1. Here, we are multiplying two $N \times N$ matrices referred to as A and B to produce a matrix C.

Listing 7.1: Matrix multiplication

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        sum = 0;
        for (k=0; k<N; k++)
                sum +=  A[i][k] * B[k][j];
        C[i][j] = sum;
    }
}
```

This is the classic matrix multiplication algorithm – nice and simple. However, this code is not efficient from the point of view of cache accesses for large values of N, which is most often the case. Let us understand why. Assume that N is a very large number; hence, none of the matrices fit within the L1 cache. In the case of this algorithm, we multiply a row in A with a column in B, element by element. Subsequently, we move to the next column in B till we reach the end of the matrix. Even though we have temporal locality for the elements of the row in A, we essentially touch all the elements in B column by column. The N^2 accesses to elements in B do not exhibit any temporal locality, and if the size of N is large, we shall have a lot of capacity misses. Thus, the cache performance of this code is expected to be very poor.

Let us now see what happens in the subsequent iteration of the outermost loop. We choose the next row of A and then again scan through the entire matrix B. We do not expect any elements of B to have remained in the cache after the last iteration because these entries would have been displaced from the cache given B's size. Therefore, there is a need to read the elements of the entire matrix (B in this case) again. We can thus conclude that the main reason for poor temporal locality and consequently poor cache hit rates is because in every iteration of the outermost loop, we need to read the entire matrix B from the lowest levels of memory. This is because it does not fit in the higher level caches. If we can somehow increase the degree of temporal locality, then we can improve the cache hit rates as well as the overall performance.

The key insight here is to not read the entire matrix B in every iteration. We need to consider small regions of A and small regions of B, process them, and then move on to other regions. We do not have the luxury of reading large amounts of data every iteration. Instead, we need to look at small regions of both the matrices simultaneously. Such regions are also called *tiles*, and thus the name of our algorithm is called *loop tiling*.

Let us start by looking at matrix multiplication graphically as depicted in Figure 7.44. In traditional matrix multiplication (Figure 7.44(a)), we take a row of matrix A and multiply it with a column of matrix B. If the size of each row or column is large, then we shall have a lot of cache misses. In comparison, the approach with tiling is significantly different. We consider a $b \times b$ tile in matrix A, and a same-sized tile in matrix B. Then we multiply them using our conventional matrix multiplication technique to produce a $b \times b$ tile (see Figure 7.44(b)). The advantage of this approach is that at any point of time, we are only considering three matrices of b^2 elements each. Thus, the total amount of working memory that we require is $3b^2$. If this data fits in the cache, then we can have a great degree of temporal locality in our computations.

Now, that we have looked at the insight, let us look at the code of an algorithm that uses such kind of loop tiling or blocking (refer to Listing 7.2). Assume that the result matrix C is initialized to all zeros. Additionally, assume that both the input matrices, A and B, are $N \times N$ matrices, where N is divisible by the tile size b.

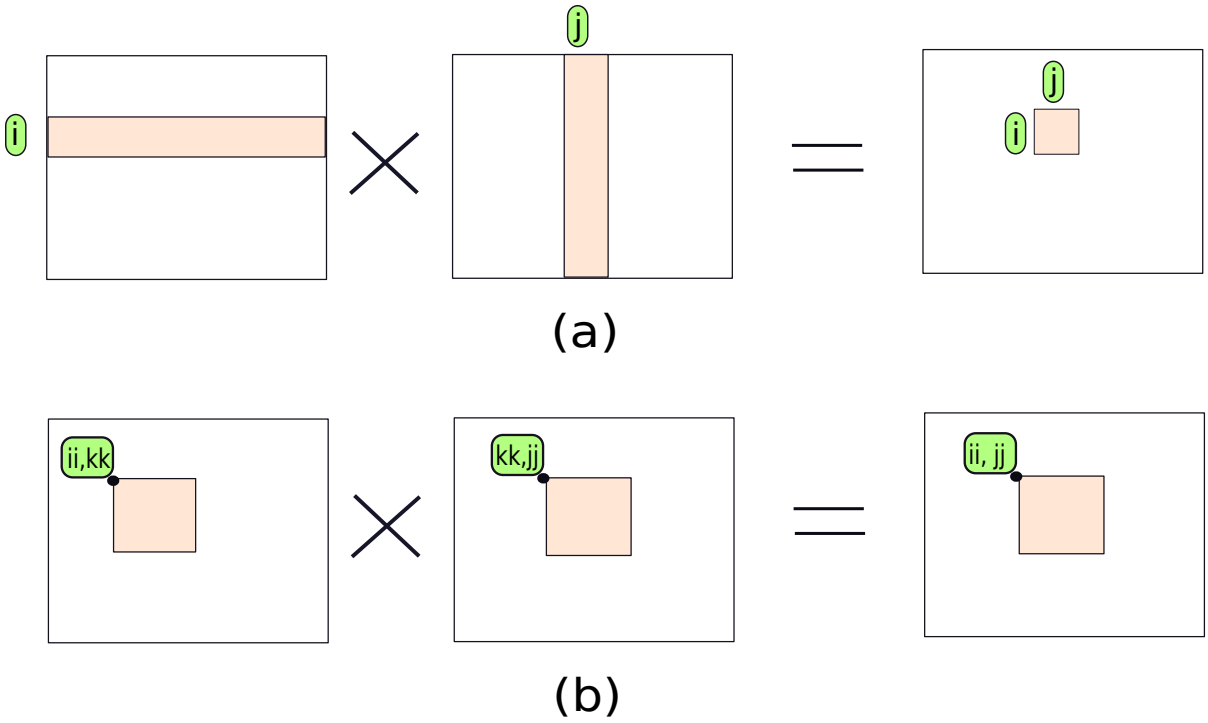

Figure 7.44: Matrix multiplication: (a) normal (b) with tiling

Listing 7.2: Matrix multiplication using tiling or blocking

```
/* Iterate through the tiles */
for (ii=0; ii<N; ii+=b) {
    for (jj=0; jj<N; jj+=b) {
        for (kk=0; kk<N; kk+=b) {

/* iterate within a tile */
            for (i=ii; i<(ii+b); i++) {
                for (j=jj; j<(jj+b); j++) {
                    for (k=kk; k<(kk+b); k++) {
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
```

There are many implementations of tiling. There are variants that have 5 nested loops. We show a simpler implementation with 6 nested loops. First we consider the three matrices consisting of arrays of tiles, where each tile has $b \times b$ elements. Similar to traditional matrix multiplication, we iterate through all combinations of tiles in the three outermost loops. We essentially choose two tiles from the matrices A and B in the three outermost loops. The first tile starts at (ii, kk); it is b elements deep and b elements wide. Similarly, the second tile starts at (kk, jj) – its dimensions are also $b \times b$.

Next, let us move our attention to the three innermost loops. This is similar to traditional matrix multiplication where we iterate through each and every individual element in these tiles, multiply the

corresponding elements from the input matrices, and add the product to the result element's value – value of $C[i][j]$. Let us convince ourselves that this algorithm is correct, and it is equivalent to the traditional matrix multiplication algorithm.

This is easy to prove. Consider the traditional matrix multiplication algorithm. We consider all combinations of i, j, and k. For each combination we multiply $A[i][k]$ and $B[k][j]$, and add the result to the current value of the result element $C[i][j]$. There are N^3 possible values of such combinations and that's the reason we need three loops.

In this case, we simply need to prove that the same thing is happening. We need to show that we are considering all combinations of i, j, and k, and the result is being computed in the same manner. To prove this, let us start out by observing that the three outermost loops ensure that we consider all combinations of $b \times b$ tiles across matrices A and B. The three innermost loops ensure that for each pair of input tiles, we consider all the values that the 3-tuple (i, j, k) can take. Combining both of these observations, we can conclude that all the combinations of i, j, and k are being considered. Furthermore, the reader should also convince herself that no combination is being considered twice. Finally, we perform the multiplication between elements in the same way, and also compute the result matrix C in the same way. A formal proof of correctness is left as an exercise for the reader.

The advantage of such techniques is that it confines the execution to small sets of tiles. Thus, we can take advantage of temporal locality, and consequently reduce cache miss rates. Such techniques have a very rich history, and are considered vitally important for designing commercial implementations of linear algebra subroutines.

7.4.6 Virtually Indexed Physically Tagged (VIPT) Caches

Up till now we have blissfully ignored the fact that we actually need to translate virtual to physical addresses before accessing the cache. The reason was that we assumed that the LSQ stores physical addresses, and the translation happens prior to accessing the LSQ. However, we can have an alternative design where the LSQ stores virtual addresses, and no two virtual pages map to the same physical frame. In this case, we can defer address translation till we access the cache. This introduces an additional overhead in terms of time. It lengthens the critical path by introducing an additional stage – virtual to physical address translation.

Let us try to reduce this overhead by creating a virtually indexed physically tagged (VIPT) cache, which is indexed with the virtual address, and the tag comes from the physical address. Let us first discuss the design of this cache, and then we shall discuss its benefits. Consider the way we access a cache in a 32-bit machine. We divide a memory address into three parts as shown in Figure 7.45(a). The three parts are the bits required to address the byte within the block (block offset), the bits for indexing the set (index), and the tag (rest of the bits). Note that in this section, we will use two terms: block offset (bits used to address a byte within a cache block), and page offset (bits used to address a byte within a page).

Just below Figure 7.45(a), we have Figure 7.45(b) that shows the way that we split a virtual memory address into the 12-bit offset (address of a byte within a page) and the 20-bit page number (see Section 7.2). Immediately below Figure 7.45(b), we have Figure 7.45(c) that shows the breakup of a physical address into two components: offset of a byte within the frame (or page) and the frame address. Let us now notice a pattern that is staring at us. If the size of the block offset and the set index is less than 12 bits, then there is an opportunity for an interesting optimization. Consider an example. Assume we have a 64-byte block size, and 64 sets in a cache. In this case, we need 6 bits to specify the address of a byte within the block, and we need 6 bits to uniquely index a set. Thus, in total, we need 12 bits.

How do we connect this to the fact that we need the same 12 bits to specify the address of a byte within a page or a frame? Here, the key insight is that in the process of address translation, the 12 LSB bits do not change. They remain the same because the size of a page or a frame is 4 KB (as per our assumptions). However, the remaining 20 MSB bits change according to the mapping between pages and frames. The crucial insight is that the **12 LSB bits** are the same in the virtual and physical addresses.

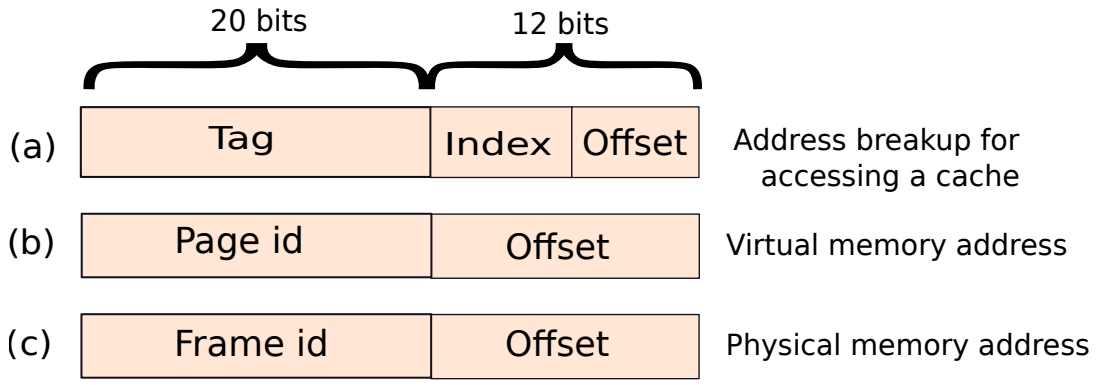

Figure 7.45: Breakup of a memory address for accessing a set associative cache. This example is for a 32-bit memory system with 4 KB pages.

If we can find the set index using these 12 bits, then it does not matter if we are using the physical address or the virtual address. We can index the correct set before translating the address.

In this particular case, where we are using 12 bits to find the set index and block offset, we can use the virtual address to access the set. Refer to the VIPT cache in Figure 7.46. When the memory address is ready, and we are sure that there are no chances of store→load forwarding in the LSQ, we can proceed to access the L1 cache. We first extract the 6 set index bits, and read out all the tags in the set. Simultaneously (see the timing diagram in Figure 7.46) we perform the virtual to physical address translation by accessing the TLB. The greatness of the VIPT cache is that it allows us to overlap the tag accesses with the process of translation. In the next stage of the access, we have the physical address with us, and then we can extract its tag and compare the tag portion of the address with the tags stored in the ways. The rest of the access (read or write) proceeds as usual.

Note that the VIPT scheme has its limitations. If we have many sets, then it is possible that the set index bits are split between the page offset, and the page/frame number. Then, this approach is not feasible. The only reason this approach works is because it is possible to access the set and read out all of its constituent ways in parallel without translating the address.

Let us now extend this idea. Assume that the number of block offset bits and set index bits adds up to 14. Since a page is 4 KB (12 bits), our VIPT scheme will not work. This is because we have two extra bits, and they will not be the same after the mapping process. This is where we can get some help from either software or hardware.

Let us discuss the software approach first. Assume that our process of translation is such that the least significant 14 bits are always the same between physical and virtual addresses. This requires minimal changes to the OS's page mapping algorithms. However, the advantage is that we can then use the 14 LSB bits to read out all the tags from the set (similar to the original VIPT scheme). We shall thus have all the advantages of a virtually indexed physically tagged cache. However, there is a flip side to this. In this case, we are creating a super-page (larger than a page) that is 16 KB (2^{14} $bytes = 16\,KB$). Frames in memory need to be reserved at the granularity of 16 KBs. This might cause wastage of memory space. Assume that in a frame, we are only using 8 KB; the remaining 8 KB will get wasted. We thus have a trade-off between memory usage and performance. For some programs, such a trade-off might be justified. This needs to be evaluated on a case-by-case basis.

The other method of dealing with such cases is with a hardware trick. We start out by noting that most of the time we have a good amount of temporal and spatial locality. Most consecutive accesses are expected to be to the same page. We can thus keep the corresponding frame number in a small register. We can speculatively read the translation from the register, create a physical address, and start accessing the cache. This is a very fast operation as compared to a full TLB access. In parallel, we need to access

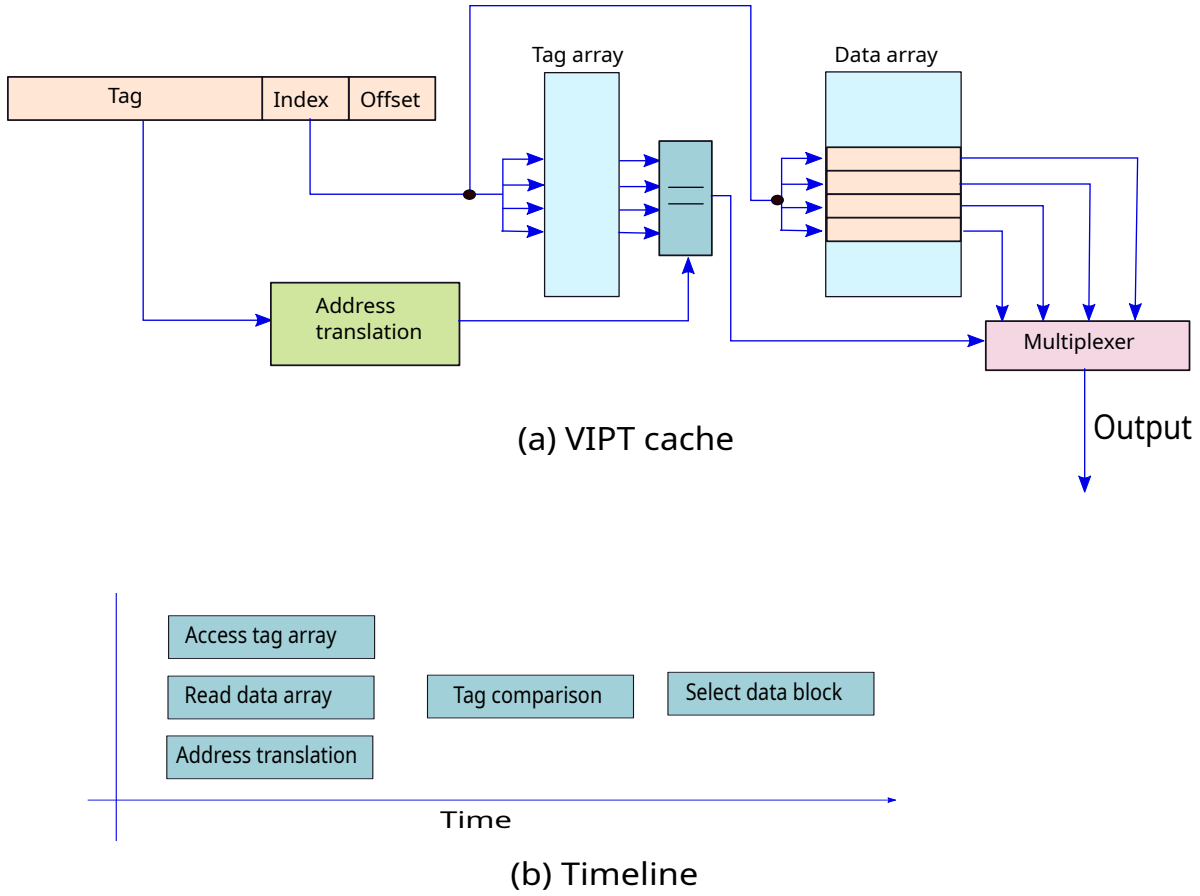

Figure 7.46: Virtually indexed physically tagged (VIPT) cache

the TLB, and verify if the speculation is correct or not. If we have a high chance of success, then we have effectively minimized the address translation overhead. The accuracy of this process can further be enhanced by having a more sophisticated predictor that uses the PC, and possibly the memory address of the load. The prediction however must be done before the request is ready to be sent to the L1 cache.

7.5 Trace Caches

Let us sit back for a minute, and think about what we really want. We need a stream of instructions on the correct path at a rate that is more than the issue or commit width of the processor. This would ensure that the fetch stage is not a bottleneck. In CISC processors, we need a high-bandwidth decode unit as well. In most CISC ISAs as of 2020 such as Intel x86, instructions are converted from CISC to RISC micro-instructions inside the processor such that they can easily be processed in the pipeline. They are then sent to the decode unit. Sometimes this process can become very slow, and as a result, the decode stage can also become a bottleneck. To create a high-throughput decode stage, processors typically have multiple decoders that operate in parallel.

All this circuitry consumes a fair amount of chip area, and consumes a disproportionate amount of power. If we can somehow get rid of all of this circuitry, and manage to get decoded RISC micro-instructions directly from the caches, then we can radically decrease the power consumption and improve

performance. In other words, we can completely skip the fetch and decode stages of the pipeline.

This does sound too good to be true. However, it is possible to get close. Designers at Intel tried to realize this goal in early 2000, when they designed a novel structure called a *trace cache* for the Intel® Pentium® 4 processor. In this section, we shall present the ideas contained in the patent filed by Krick et al. [Krick et al., 2000]. We shall simplify some sections for the sake of readability. For all the details please refer to the original patent.

Let us first explain the concept of a trace. A trace is a sequence of dynamic instructions that subsumes branches and loop iterations. Let us explain with an example. Consider the following piece of C code.

```
int sum = 0;

for (i=0; i<3; i++){
        if (i == 1)
                continue;
        sum = sum + arr[i];
}
```

We have a loop with 3 iterations and an *if* statement that skips the body of the second loop iteration. Let us look at the sequence of instructions that the processor will execute. For the sake of readability we show C statements instead of x86 assembly code. Let the label *.loop* point to the beginning of the loop, and the label *.exit* point to the statement immediately after the loop. Note that we are not showing a well-formed assembly program, we are instead just showing a dynamic sequence of instructions that the processor will see (simplifications made to increase readability).

```
/* initial part */
sum = 0;
i = 0;

/* first iteration */
if (i >= 3) goto .exit;
if (i == 1) goto .temp;
sum = sum + arr[i]; // i = 0
.temp: i = i + 1;
goto .loop;

/* second iteration: starts at .loop */
if (i >= 3) goto .exit;
if (i == 1) goto .temp;  /* next iteration */
.temp: i = i + 1;
goto .loop;


/* third iteration: starts at .loop */
if (i >= 3) goto .exit;
if (i == 1) goto .temp;
sum = sum + arr[i]; // i = 2
.temp: i = i + 1;
goto .loop;


/* fourth iteration */
if (i >= 3) goto .exit;  /* exit the for loop */
```

The instructions in this unrolled loop form a trace. It is the sequence of instructions that the processor is going to fetch to execute the code. If we can store the instructions corresponding to the

entire trace in a trace cache, then all that we need to do is to simply fetch the instructions in the trace and process them. Furthermore, if we can also store them in their decoded format, then we can skip the power-hungry decode stage. In the case of CISC processors, it is desirable if the trace cache stores micro-instructions instead of full CISC instructions. If we observe a good hit rate in the trace cache, then we can save all the energy that would have been consumed in the fetch and decode stages. In addition, the trace cache is also serving as a branch predictor. We are using the information about subsequent trace segments as branch predictions. Finally, note that we still need an i-cache in a system with a trace cache. We always prefer reading instructions from the trace cache; however, if we do not find an entry, we need to access the conventional i-cache.

7.5.1 Design of the Trace Cache

The basic structure of a trace cache is shown in Figure 7.47. Akin to a normal cache, we have a tag array, a data array, and a cache controller. The only extra structure that we see is the fill buffer, which is used while constructing a trace.

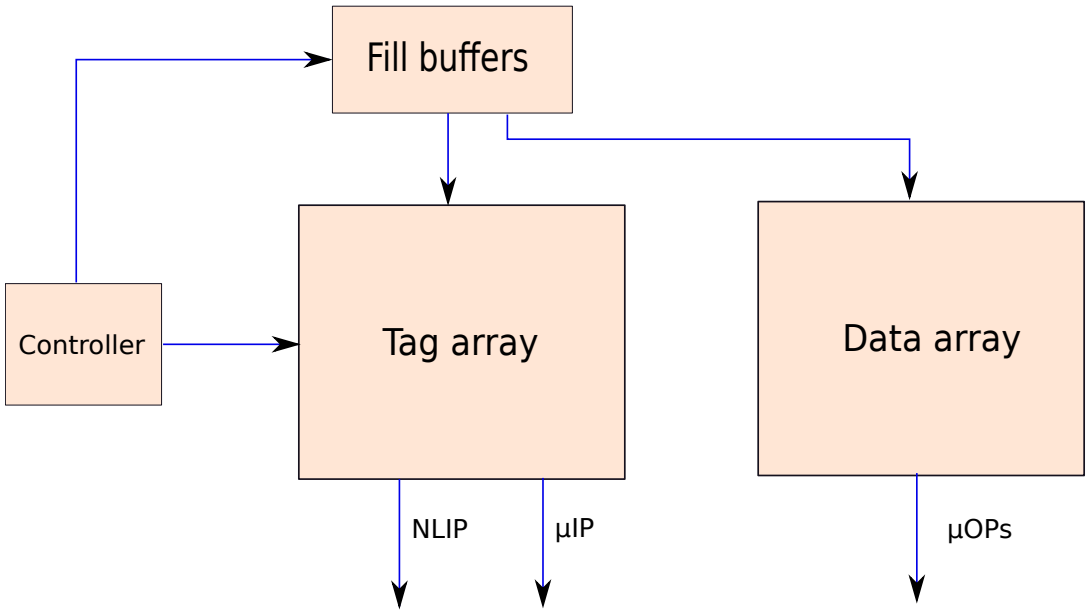

Figure 7.47: Overview of the trace cache

The data array is a regular k-way set associative cache. Let us assume that it is a 4-way set associative cache with 4 ways per set. Instead of defining traces at the granularity of instructions, let us define a trace as a sequence of cache lines. For example, it is possible that a trace many contain 5 cache lines. Each such line is known as a *trace segment*. We can have three kinds of segments: head, body, and tail. Every trace is organized as a linked list as shown in Figure 7.48. It starts with the head segment, and ends with the tail segment.

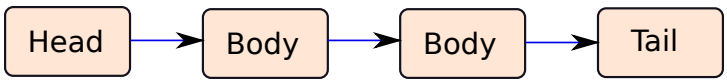

Figure 7.48: A trace consisting of multiple segments

Next, we need to store the trace in the trace cache. We start with the head of the trace. We create a tag array entry with the head of the trace. The rest of the segments in the trace are organized as a linked list (see Figure 7.48). Each segment is stored in a separate data line, and has a dedicated entry in the tag array. The standard way of representing a linked list is by storing a pointer to the next node within the current node. However, this is not space efficient. If a data block is 64 bytes, and a pointer is 64 bits (8 bytes), then the space overhead is equal to 12.5%. This is significant. Hence, let us restrict the way a trace is stored.

Let us store a trace in contiguous cache sets. For example, if a trace has 5 segments, we can store the segments in sets s, $s+1$, ..., $s+4$, where s is the index of the set that stores the trace head. Consider a 4-way set associative cache. Each trace segment can be stored in any of the 4 ways of a set. Given that we have stored a trace segment in a given set, we know that the next trace segment is stored in the next set. The only information that we need to store is the index of the way in that set. This requires just 2 bits, and thus the additional storage overhead is minimal. Figure 7.49 shows how we store multiple traces in the data array. In this figure, each column is a way in a set. A trace cache can be visualized as a packet of noodles, where each individual strand represents a trace.

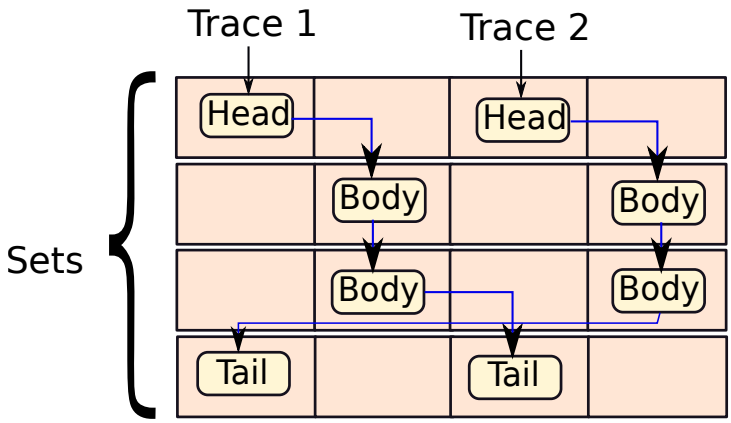

Figure 7.49: Visualization of traces stored in the data array

Traces cannot be arbitrarily long. Their length is limited by the number of sets in the cache. However, there are a few additional conditions that govern the length of a trace.

Let us first look at conditions for terminating the creation of a trace segment, we shall then move on to the rules for terminating the creation of a trace. Let us henceforth refer to a microinstruction as a μOP (micro-op).

1. If we encounter a complex CISC instruction that translates to many more μOPs than what a single data line can store, then we store all the μOPs that can be stored in the data line, and then terminate the trace segment. The remaining μOPs need to be generated by the decode unit by reading the microcode memory. The microcode memory contains all the microinstructions for complex CISC instructions.

2. We allow a limited number of branch instructions per trace segment. If we encounter more than that, we terminate the trace segment. This is to avoid structural hazards.

3. We never distribute the μOPs of a CISC instruction across trace segments. We terminate the segment if we do not have enough space to store all the μOPs of the next CISC instruction.

Let us now look at the criteria to terminate the process of trace creation.

1. In an indirect branch, a call, or a return statement, the branch's target may be stored in a register. Since the address is not based on a fixed PC-relative offset, the next CISC instruction tends to change for every trace. Consider a function return statement. Depending on the caller function, we may return to a possibly different address in each invocation. This is hard to capture in a trace. Hence, it is better to terminate a trace after we encounter such instructions.

2. If we receive a branch misprediction or an interrupt alert, then we terminate the trace. This is because the subsequent instructions will be discarded from the pipeline and thus will not be checked for correctness.

3. The length of every trace is limited by the number of sets in the cache, and this is thus a hard limit on the length of the trace.

Tag Array

Let us now look at the trace cache in greater detail. Each entry in the tag array contains the following fields: address tag, valid bit, type (head or body or tail), next way, previous way, NLIP (next line's instruction pointer), and μIP. Let us describe them in sequence (also see Figure 7.50).

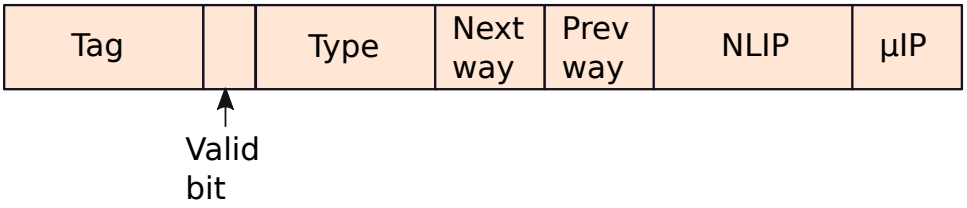

Figure 7.50: Entry in the tag array of the trace cache

When we are performing a lookup in the tag array to locate the head of the trace, we send the address to the tag array. This is similar to a regular cache access, where the tag portion of the address needs to be compared with the tag stored in the entry. Hence, the first entry that we store is the tag. Subsequently, we have a customary valid bit that indicates the validity of the entry.

For each data line that stores one trace segment, we need to store whether it is a trace head, body or tail. This requires 2 bits (type field). Since we store consecutive trace segments in consecutive sets, the only information that we need to store is the id of the next and previous ways such that we can create a doubly linked list comprising trace segments. Previous pointers are required to delete the trace at a later point of time starting from a body or a tail segment. The next field is NLIP, which stores the address of the next CISC instruction. This is only required for the tail segment such that we can locate the address of the next CISC instruction. The last field, μIP, is used to read microinstructions for a complex CISC instruction. We use it to index a table of microinstructions known as the *microcode memory*.

Data Array

Each line in the data array can store up to a maximum of 6 microinstructions (μOPs). We have a valid bit for each μOP. To skip the decode stage, we store the μOPs in a decoded format such that the microinstruction does not have to be decoded in the pipeline. In addition, each μOP also stores a branch target such that it does not have to be computed. This saves us an addition operation for every instruction that has a PC-relative branch.

7.5.2 Operation

To fetch a trace, the trace cache runs a state machine as shown in Figure 7.51. During execution, we run a state machine that starts in the *Head lookup* state. Given the program counter of an instruction, we search for it in the trace cache.

Assume that we find an entry. We then transition to the *Body lookup* state, where we keep reading all the trace segments in the body of the trace and supplying them to the pipeline. Once we reach the tail, we transition to the *Tail* state. In the *Body lookup* state, if there is a branch misprediction, or we receive an interrupt, we abort reading the trace, and move to the *Head lookup* state to start anew. Furthermore, if at any point, we encounter a complex macroinstruction, we read all its constituent microinstructions from a dedicated microcode memory (*Read microinstructions* state), and then continue executing the trace. If at any point, we do not find a trace segment in the trace cache, we transition to the *body miss* state, which means that our trace has snapped in the middle. This is because while building another trace we evicted a data block in the current trace. Whenever we terminate executing a trace either because of an unanticipated event such as an external interrupt, we reached the *Tail* state, or we reached the *Body miss* state, we start from the *Head lookup* state once again.

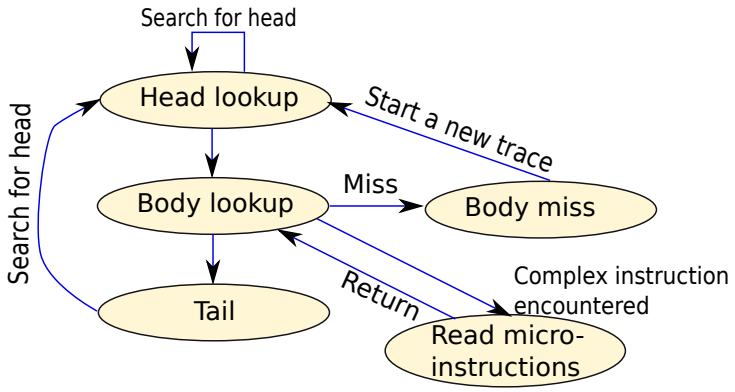

Figure 7.51: FSM (finite state machine) used for reading a trace

Let us now look at the process of creating a trace. The flowchart is shown in Figure 7.52. We trigger such an operation when we find that a fetched instruction is not a part of any trace. We treat it as the head of a trace, and try to build a trace. The first step is to issue a fetch request to the i-cache. Then the state changes to the *wait for μOPs* state, where we wait for the decoder to produce a list of decoded μOPs. Once the instruction is decoded, the μOPs are sent to the fill buffer. Then we transition to the *bypass μOPs* state, where we send the μOPs to the rest of the pipeline. This continues till we encounter a trace segment terminating condition. There are two common cases that can disrupt the flow of events.

The first is that we encounter a complex instruction, where we need a list of microinstructions from microcode memory. In this case, we transition to the *Read microinstructions* state. The second is a normal trace segment terminating condition. In this case, we move to the *Transfer* state where the data line created in the fill buffer is transferred to the tag and data arrays. Then we transition back to the *Wait for μOPs* state, if we have not reached the end of the trace. However, if we have encountered a condition to end the trace, then we mark the data line as the tail of the trace, and finish the process of creating the trace.

We subsequently fetch the next instruction, and check if it is the head of a trace. If it is not the head of any trace, then we start building a new trace.

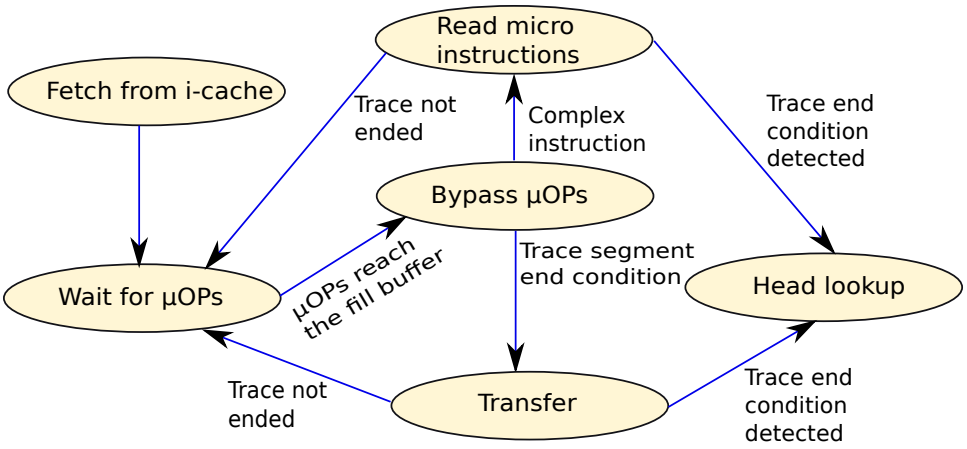

Figure 7.52: FSM used for building a trace

7.6 Instruction Prefetching

If we go back to our discussion in Section 7.1.7, we shall recall that prefetching can solve most of the problems that lead to frequent cache misses. They can reduce all three types of misses: cold misses, capacity misses, and conflict misses. Predicting memory accesses in advance is a very powerful mechanism, and in theory, it can hide the memory access latency almost completely. We shall discuss instruction prefetching in this section; we will discuss data prefetching in the next section.

The basic idea of prefetching is very simple. We predict the misses before hand, and issue memory requests to "prefetch" data to higher levels in the memory hierarchy. If we are able to predict misses with a reasonable accuracy, then we are effectively increasing the hit rate of caches at higher levels of the memory hierarchy. This increases the performance, and reduces the traffic to the lower levels of the memory system such as the LLC (last level cache) and off-chip memory.

The two key questions that need to be asked are (1) "When to prefetch," and (2) "What to prefetch." Regarding the first question, we can neither prefetch too soon nor too late. If we prefetch too soon, then the prefetched block needs to wait for a long time till it is used. During this time, some other accesses can evict this block. Additionally, prefetching may displace other blocks with useful data from the cache. These displaced blocks might be used in the near future, and thus we might actually be increasing the miss rate by prefetching a lot of blocks too soon. It is possible that we might be displacing useful data with useless data.

In comparison, if we prefetch a block too late, then it will not be available in the caches when the core needs it. It is true that the time a memory request needs to wait might get reduced because of an outstanding prefetch request, however this situation is not optimal. Hence, the golden rule for prefetching is that we need to prefetch the right set of addresses at exactly the right time. If we prefetch too soon or too late, too little or too much, there is an associated performance penalty.

Keeping these broad principles in mind, let us look at some of the most common prefetching strategies.

7.6.1 Next Line Prefetching

Let us consider the simplest method of prefetching known as *next line prefetching*. The basic insight is as follows. If we have accessed a cache block with block address X, then there is a very high probability that the next few blocks that we shall access will have their block addresses equal to $X+1$, $X+2$, and so on. This means that after we access address X, we need to prefetch the subsequent blocks. Even though the high-level idea looks simple, there is an important point to note here.

Most prefetchers actually base their decisions on the miss sequence and not the access sequence. This means that a prefetcher for the L1 cache takes a look at the L1 misses, but not at the L1 accesses. This is because we are primarily concerned with L1 misses, and there is no point in considering accesses for blocks that are already there in the cache. It is not power efficient, and this information is not particularly useful. Hence, we shall assume from now on that all our prefetchers consider the miss sequence only while computing their prefetching decisions. Furthermore, we shall only consider block addresses while discussing prefetchers – the bits that specify the addresses of words in a block are not important while considering cache misses.

Important Point 14
Prefetchers operate on the miss sequence of a cache, and not on the access sequence. This means that if a prefetcher is associated with a cache, it only takes a look at the misses that are happening in the cache; it does not look at all the accesses. This is because most cache accesses are typically hits, and thus there is no need to consider them for the sake of prefetching data/instructions into the cache. We only need to prefetch data/instructions for those block addresses that may record misses in the cache. Additionally, operating on the access sequence will consume a lot of power.

Hence, in the case of a next line or next block prefetcher, we look at misses. If we record a miss for block address X, we predict a subsequent *miss* for block address $X + 1$ – we thus prefetch it.

This is a simple approach for instruction prefetching and often works well in practice. Even if we have branches, this method can still work. Note that most branches such as *if-else* statements, or the branches in *for* loops have targets that are nearby (in terms of memory addresses). Hence, fetching additional cache blocks, as we are doing in this scheme, is helpful.

There can be an issue with fetching blocks too late. This can be fixed by prefetching the block with address $X + k$, when we record a miss for block address X. If we record a miss for a new block every n cycles, and the latency to access the lower level memory is L cycles, k should be equal to L/n. This means that the block $X + k$ will arrive just before we need to access it.

7.6.2 Markov Prefetching

Let us now design something more sophisticated. The main problem with next line prefetching is that it relies on an assumption that if an access with address X misses in the cache, then there is a high likelihood of an access with address $X + 1$ also suffering from a miss in the near future (note that these are block addresses). This need not necessarily be the case all the time. We can have branches to blocks that are far away. Next line prefetching will not be able to cover such cases. Let us thus use a different algorithm called *Markov prefetching*, which is inspired from the concept of Markov chains. A Markov chain is a type of random process that moves from state to state. We can make predictions about future states solely based on the current state. This is also known as the *Markov property*.

We can use the same property here. Consider two cache blocks A and B. Further, assume that every time we record a miss for cache block A, the next miss that we observe is for cache block B. Can this pattern be used to design a good prefetcher? Such a pattern does indeed follow the Markov property. If we record a miss for block A, we can immediately deduce that we need to prefetch cache block B. We can thus immediately issue a prefetch request to the memory system to fetch cache block B. This is the basic idea of a Markov prefetcher.

Let us thus create a miss table as shown in Figure 7.53, where we record the pattern of misses. Each row of the miss table corresponds to a cache block. Consider the row that corresponds to the cache block with address X. In each row we have multiple columns that record the access frequencies of other cache blocks that have recorded a miss just after a miss was recorded for block X. Let us explain in the context of the example shown in Figure 7.53. Here, we keep data for two other blocks that suffered from

a cache miss, after we failed to find X in the cache. Let them be Y and Z. For each block, we store the corresponding miss count. We associate a saturating counter with each column that indicates the miss count. To ensure that the information stays fresh, we periodically decrement these counters.

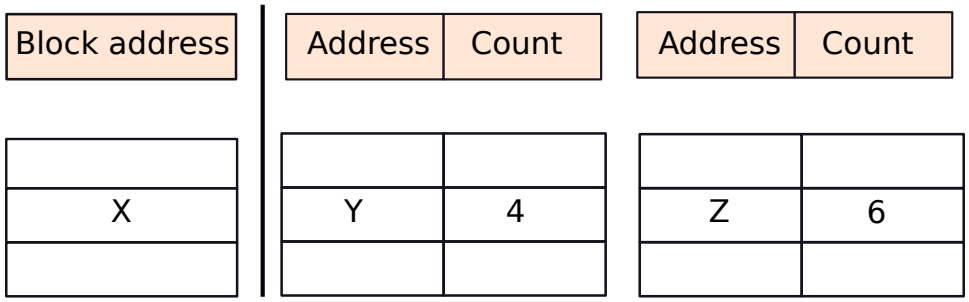

Figure 7.53: The miss table

Now, when we have a cache miss, we look up the address in this table. Assume we suffer a miss for block X. In the table we find two entries: one each for blocks Y and Z respectively. There are several choices. The most trivial option is to issue prefetch requests for both Y and Z. However, this is not power efficient and might increase the pressure on the lower level cache. Sometimes it might be necessary to adopt a better solution to conserve bandwidth. We can compare the miss counts of Y and Z and choose the block that has a higher miss count.

After issuing the prefetch request, we continue with normal operation. Assume that for some reason the prefetch request suffers from an error. This could be because the virtual memory region corresponding to the address in the request has not been allocated. Then we will get an "illegal address" error. Such errors for prefetch requests can be ignored. Now, for the next cache miss, we need to record its block address. If the miss happened for block Y or block Z, we increment the corresponding count in the table. For a new block we have several options.

The most intuitive option is to find the entry with the lowest miss count in the table among the blocks in the row corresponding to the cache miss, and replace that entry with an entry for the new block. However, this can prove to be a bad choice, particularly, if we are disturbing a stable pattern. In such cases, we can replace the entry probabilistically. This provides some hysteresis to entries that are already there in the table; however, it also allows a new entry to come into the miss table with a finite probability. The choice of the probability depends on the nature of the target workload and the architecture.

7.6.3 Call Graph Prefetching

Till now, we had been looking at instructions at the granularity of cache blocks. However, let us now consider some modern approaches for prefetching, where we consider instructions at the granularity of functions. In such approaches, we directly prefetch the entire function. If the function is large, we only prefetch the first N lines of the function. Let us discuss one of the early approaches in this space. It is known as call graph prefetching (CGP) [Annavaram et al., 2003]. The basic idea is that when we are executing one function, we predict the functions that it will call and prefetch their instructions. This reduces the number of instruction cache misses.

There are two approaches: one in software and one in hardware. Let us discuss the software approach first.

Software Approach

We first start out by creating a *call graph* of a program. A call graph is created as follows. We run the program in profiling mode, which is defined as a *test run* of the program before the actual run, where we collect important statistics regarding the program's execution. These statistics are known as the program's *profile*, and this process is known as *profiling*. In the profiling phase, we create a graph (defined in Section 2.3.2) in which each node represents a function. If function A calls function B, then we add an edge between the nodes representing A and B respectively. Note that it is possible for node A to call different functions across its invocations. One option is to only consider the first invocation of function A; in this case, we do not collect any data for subsequent invocations. Based on this information, we can create a graph of function calls, which is referred to as the *call graph*. From the call graph, we can create a list of $\langle caller, callee \rangle$ function pairs, and write them to a file. Note that if function A calls function B, then A is the caller and B is the callee.

Let us explain this process with an example. Consider the set of function invocations shown in Figure 7.54(a). The associated call graph is shown in Figure 7.54(b). In addition, we label the edges based on the order in which the parent function invokes the child functions. For example, in Figure 7.54(a), *foo2* is called after *foo1*, and thus we have labeled the edges to indicate this fact.

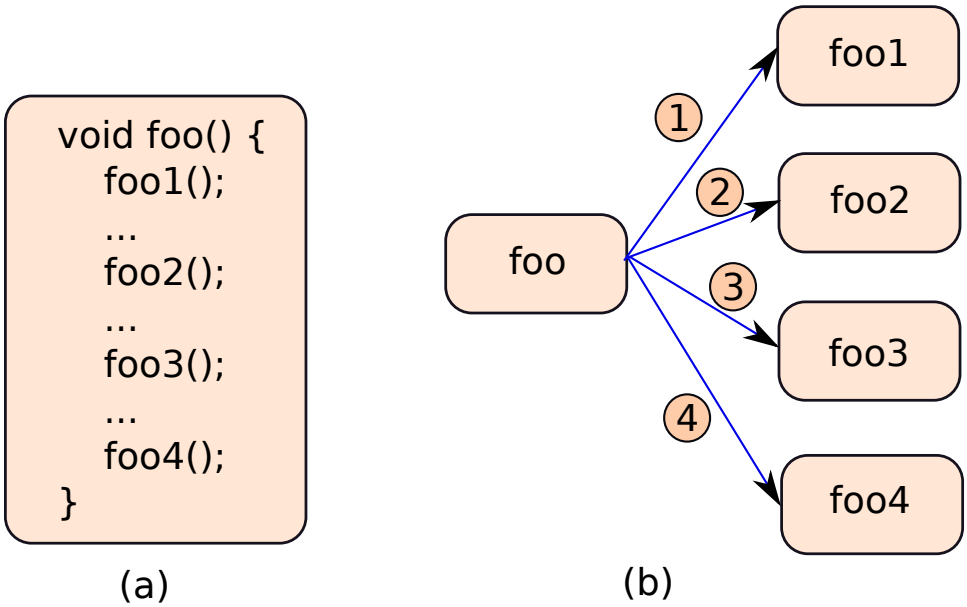

Figure 7.54: Example of a call graph

Subsequently, we can use a binary instrumentation engine, or even a compiler to generate code that prefetches instructions. The algorithm to insert prefetch statements is as follows. Assume function A calls function B and then function C. We insert prefetch code for function B at the beginning of function A. We assume that it will take some time to set up the arguments for B, and then we shall invoke the function. During this time, the memory system can fetch the instructions for function B.

After the call instruction that calls function B, we insert prefetch code for function C. Again the logic is the same. We need some time to prepare the arguments for function C after B returns. During this time, the memory system can in parallel prefetch the instructions for C. If after C, we had invoked another function D, then we would have continued the same process.

The software approach is effective and is generic. However, it necessitates a profiling run. This is an additional overhead. In addition, it is not necessary that the inputs remain the same for every run

of the program. Whenever, the input changes substantially, we need to perform the process of profiling once again. In addition, for large programs, we can end up generating large files to store these profiles. This represents a large storage overhead also. Finally, the profiling run need not be representative. A function might be called many times and its behavior might vary significantly, and all of these might not be effectively captured in the profile. Hence, whenever we have the luxury, a hardware based approach is preferable.

Hardware based Approach

Let us create a call graph history cache (CGHC) as originally proposed by [Annavaram et al., 2003]. In this case, we maintain a dynamic subset of the call graph. Unlike the software approach, the execution is not preceded by a profiling phase.

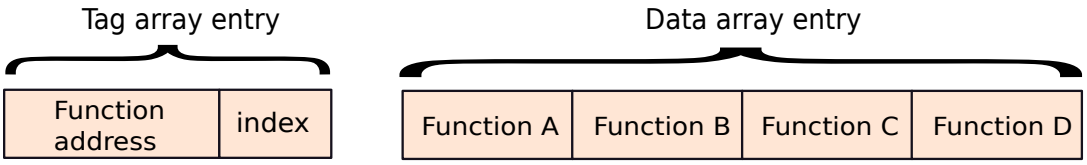

Figure 7.55: Tag and data array entries in the CGHC

Figure 7.55 shows the structure of a call graph history cache. It consists of a tag array and data array similar to a normal cache. The tag array stores the tags corresponding to the PCs of the first (starting) instructions of the functions. Along with each entry in the tag array, we store an integer called an index, which is initialized to 1. The data array can contain up to N entries, where each entry is the starting PC of a function that can be invoked by the current function.

Assume function A invokes the functions B, C, and D, in sequence. We store the starting PCs of B, C, and D, respectively, in the data array row. Note that we store the starting PCs of functions in the same order in which they are invoked. This order is also captured by the *index*. If the *index* is k, then it means that we are referring to the k^{th} function in the data array row.

Similar to software prefetching, whenever we invoke function A, we start prefetching the instructions of function B. When B returns, we prefetch the instructions of function C, and so on. We use the *index* field in the tag array for this purpose. Initially, the index is 1, hence, we prefetch the first function in the data array row. On every subsequent return, we increment the *index* field, and this is how we identify the next function to prefetch: if the *index* is i, then we prefetch the instructions for the i^{th} function in the row. When A returns, we reset its *index* field to 1.

Whenever A invokes a new function, we have two cases. If the function is invoked after the last function in the data array row, then we need to just create a new entry in the row, and store the address of the first instruction of the newly invoked function. However, if we invoke a new function that needs to be in the middle of the row, then there is a problem: it means that the control flow path has changed. We have two options. The first is that we adopt a more complicated structure for the data array row. In the first few bytes, we can store a mapping between the function's index and its position in the data array row. This table can be updated at run time such that it accurately tracks the control flow. The other option is to discard the rest of the entries in the data array row, and then insert the new entry. The former is a better approach because it allows changes to propagate faster; however, the latter is less complex.

In addition, we can borrow many ideas from the notion of saturated counters to keep track of the functions that are a part of the call sequence in the current phase of the program. Whenever A calls B, we can increment the saturating counter for B in A's row of the CGHC. This indicates that the entry is still fresh. Periodically, we can decrement the counters to indicate the fact that the information that we have stored is ageing. Once a counter becomes zero, we can remove the entry from the row.

7.6.4 Other Approaches

Instruction prefetching per se is a very rich area, and there are many recent research proposals in this area. Most of the advanced proposals prefetch instructions at the granularity of functions, and in addition try to group the functions into larger units.

Recent works [Ferdman et al., 2011, Kolli et al., 2013, Kallurkar and Sarangi, 2017] create instruction groups called streams, tasks, or super-functions, where the idea is that we group together functions that are typically executed in sequence. Such approaches deduce a high level structure in the program. For example, if we are executing a large program, we divide it into large and distinct subtasks. If we are executing subtask 1, we can predict that we will execute subtask 2 once it ends. We can then start prefetching the starting functions of subtask 2.

7.7 Data Prefetching

In Section 7.6, we looked at instruction prefetching algorithms. In this section, let us look at data prefetching algorithms. Instruction prefetching and data prefetching have similarities, as well as many differences. The first is that instructions have much more temporal and spatial locality as compared to data. The most common access patterns for instructions are as follows: we access them in sequence, or we access instructions as a part of loops. Both of these access patterns have a significant amount of temporal and spatial locality.

However, data access has a different kind of pattern. The primary sources of data are the stack, heap, and data sections (specific to Linux) in the memory map of a process. The stack is used to store function arguments, return addresses, and local variables. Ideally we would like to keep all of these variables in registers. However, because of a paucity of registers, we often face the need to spill these registers into memory, and this requires data accesses. Local variables and function arguments do not take up a lot of space. Even if something is spilled to memory, its likelihood of still being in the L1 cache is fairly high. Hence, prefetching such data that resides in the stack is not very beneficial.

However, we often allocate large data structures in the heap and sometimes in the *data* section of the memory map, which contain dynamically allocated and global variables, respectively. Prefetching these data structures can prove to be extremely beneficial.

Let us classify data memory accesses as either *regular* or *irregular*. A regular access is a traversal of an array or a matrix element-by-element. In comparison, we perform irregular accesses when we traverse a tree or a linked list. These are based on pointers and the sequence of memory addresses that we access does not follow a simple and well defined pattern. We can access an array using irregular accesses as well. Consider two arrays A and B. An access pattern of the form $A[B[i]]$ is an irregular access pattern for A because the array indices are not necessarily consecutive.

On the lines of our discussion in the section on instruction prefetching, a good data prefetching algorithm should be accurate, timely (not issue prefetches too soon or too fast), and additionally be able to cater to all kinds of data accesses: regular and irregular.

7.7.1 Stride based Prefetching

Notion of a Stride

The most common regular access is an array access, when we traverse the array in the sequence of increasing indices. Most programs often sequentially access array elements. Hence, such instructions should be first targeted for the purpose of prefetching. Let us create a predictor to detect this pattern.

Let us start with the definition of the term *stride* (also defined in Section 5.1.2). Consider the following piece of code.

```
for (i=0; i<N; i++){
    ...
    sum += A[i];
    ...
}
```

Here, A is an array. Depending on the data type of the array, the memory address of the array is calculated for the statement, *sum += A[i];* . If A is an array of integers, then in each iteration we increment i by 4. If it is an array of double precision numbers, then we increment i by 8. Let us now look at the assembly code for the statement *sum += A[i];* .

```
1 /* r0 contains the base address of A */
2 /* r1 contains the index i */
3 /* r2 contains the sum */
4 add r3, r1, r0
5 ld  r4, 0[r3]     /* r4 contains A[i] */
6 add r2, r2, r4    /* sum = sum + A[i] */
```

Consider the load instruction (Line 5). Every time that it is invoked, it will have a different address. Let us define the difference in the memory addresses between consecutive calls to a memory instruction (load or store) as the *stride*. In this case , the stride depends on the data type stored in array A. Alternatively, it is possible that the enclosing *for* loop does not visit array locations consecutively, instead it only traverses the even indices within the array. The value of the stride will double in this case.

Definition 48
Let us define the difference in the memory addresses between consecutive calls to a memory instruction (load or store) as the stride.

We thus observe that the value of the stride is dependent on two factors: the data type and the array access pattern. However, the only thing that we require is to know if a given stride is relatively stable and predictable. This means that the stride, irrespective of its value, should not change often. Otherwise, we will not be able to predict the addresses of future memory accesses. Let us design a stride predictor, and a stride based prefetcher.

Stride Predictor

Let us design a stride predictor on the lines of the predictors that we have been seeing up till now. We create an array indexed by the least significant bits of the PC. In each row, we can optionally have a tag for increasing the accuracy. In addition, we have the following fields (also see Figure 7.56): last address, stride, confidence bits.

The *last address* field stores the value of the memory address that was computed by the last invocation of the memory instruction. The *stride* field stores the current stride, and the confidence bits (implemented using a saturating counter) show the confidence that we have in the value of the stride that is stored in the entry.

Let us now discuss the logic. Whenever, we record a miss in a cache, we access the stride predictor. First, we subtract the last address from the current address to compute the current stride. If this is equal to the value of the stride stored in the table, then we increment the saturating counter that represents

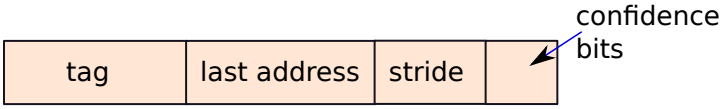

Figure 7.56: Contents of each row in the stride predictor

the confidence bits. If the strides do not match, then we decrement the confidence bits. The confidence bits provide a degree of hysteresis to the stride. Even if there is one irregular stride, we still maintain the old value till we make a sufficient number of such observations. At the end, we set the last address field to the memory address of the current instruction.

As long as the stride is being observed to be the same, this is good news. We keep incrementing the confidence bits till they saturate. However, if the value of the stride changes, then ultimately the saturating counter will reach 0, and once it does so, we replace the value of the stride stored in the entry with the current stride. In other words, our predictor can dynamically learn the current stride, and adapt to changes over time.

Process of Prefetching

Now, the process of prefetching is as follows. Whenever, we record a miss for a given block, we access its corresponding entry in the stride predictor. If the value of the confidence is high (saturating counter above a certain threshold), we decide to prefetch. If the value of the stride is S and the current address is A, we issue a prefetch instruction for address $A' = A + \kappa S$. Note that κ in this case is a constant whose value can either be set dynamically or at design time. The basic insight behind this parameter is that the prefetched data should arrive just before it is actually required. It should not arrive too soon nor too late. If we do not expect a huge variance in the nature of the workloads, then κ can be set at design time.

However, if we have a lot of variation, and we are not sure where the prefetched values are coming from, then we are not in a position to predict how long it will take to prefetch the values. If they are coming from the immediately lower level of memory, they will quickly arrive; however, if the values are coming from main memory, then they will take hundreds of cycles. This can be dynamically estimated by maintaining a set of counters. For example, we can have a counter that starts counting after a prefetch request for block A' is sent to the memory system. The counter increments every cycle. Once the data arrives, we note the value of the counter. Let the count be $T_{prefetch}$ – this gives us an estimate of the time it takes to prefetch a block.

We can have another counter to keep track of the duration between prefetching the block A' and accessing it. This counter starts when a prefetch request is sent to the memory system. The counter stops when subsequently the first memory request for a word in the block A' is sent to the memory system. Let this duration be referred to as T_{access}.

Ideally, $T_{prefetch}$ should be equal to T_{access}. If $T_{prefetch} < T_{access}$, then it means that the data has arrived too soon. We could have possibly prefetched later. This can be done by decreasing κ. Similarly, $T_{prefetch} > T_{access}$ means that the data arrived too late. In this case, we need to increase κ. We can dynamically learn the relationship between κ and T_{access} by dynamically changing the value of κ for different blocks and measuring the corresponding values of T_{access}. This approximate relationship can be used to tune κ accordingly.

As of today, stride based prediction is the norm in almost all high-end processors. This is a very simple prefetching technique, and is very useful in codes that use arrays and matrices.

7.7.2 Pointer Chasing

Let us now look at irregular accesses. A typical example of an irregular access is traversing a linked list. Consider a representative code snippet.

```
struct node_t {
    int val;
    struct node_t *next;
};

typedef struct node_t node;

void foo() {
    ...
    /* traverse the linked list */
    node* temp = start_node;
    while (temp != NULL) {
        process(temp);
        temp = temp->next;
    }
    ...
}
```

In this piece of code, we define a linked list node (*struct node_t*). To traverse the linked list, we keep reading the next pointer of the linked list, which gives us the address of the next node in the linked list. The addresses of subsequent nodes need not be arranged contiguously in memory, and thus standard prefetching algorithms do not work. A simple way of solving this problem is to insert a prefetch instruction in software for the linked list traversal code.

```
    ...
    /* traverse the linked list */
    node* temp = start_node;
    while (temp != NULL) {
        prefetch(temp->next);  /* prefetch the next node */
        process(temp);
        temp = temp->next;
    }
    ...
```

We add a set of prefetch instructions for fetching the next node in the linked list before we process the current node. If the code to process the current node is large enough, it gives us enough time to prefetch the next node, and this will reduce the time we need to stall for data to come from memory. Such a prefetching strategy is known as *pointer chasing*. We are literally chasing the next pointer and trying to prefetch it. We can extend this scheme by traversing a few more nodes in the linked list and prefetching them. Since prefetch instructions do not lead to exceptions, there is no possibility of having null pointer exceptions or illegal memory access issues in such code.

The compiler and memory allocator can definitely help in this regard. If on a best-effort basis, the memory allocator tries to allocate new nodes in the linked list in contiguous cache lines, then traditional prefetchers will still work. Of course the situation can get very complicated if we have many insertions or deletions in the linked list. However, parts of the linked list that are untouched will still maintain a fair amount of spatial locality.

7.7.3 Runahead Execution and Helper Threads

Let us now do something similar to pointer chasing in hardware. Unlike software, where we have a lot of visibility, and with explicit support from the programmer we can add prefetch instructions at the right places, hardware has very limited capabilities. It is not possible for it to make sense of the billions of instructions that pass through it every second. Hence, our hardware interventions will in a sense be less intelligent; however, they can still be effective. Let us discuss a space of such ideas.

Consider a miss in the L2 cache. Upon an L2 miss, the request needs to go to main memory, which often has a very high access latency. This is typically between 200-400 clock cycles. During this time the processor pipeline will fill up, and stall the fetch process. The processor will just wait for the data to come back from memory, and in the meanwhile it will not do anything. Hence, such L2 or L3 misses are very expensive – they lead to a lot of stalled cycles.

Keeping this in mind, let us try to do something when the processor is stalled. There are many similar ideas in this space. We shall discuss some major proposals in this area. They are collectively known as *pre-execution techniques*.

Runahead Execution

Let us discuss one of the earliest ideas in this space known as runahead execution [Mutlu et al., 2003]. In this case, whenever we have a high-latency L2 miss, we let the processor proceed with a possibly predicted value of the data. This is known as the *runahead mode*. In this mode, we do not change the architectural state. Once the data from the miss comes back, the processor exits the runahead mode, and enters the normal mode of operation. All the changes made in the runahead mode are discarded. The advantage of the runahead mode is that we still execute a lot of instructions with correct values, and in specific, we execute many memory instructions with correct addresses. This effectively prefetches the data for those instructions from the memory system. When we restart normal execution, we shall find many much-needed blocks in the caches, and thus the overall performance is expected to increase. Let us elaborate further.

Whenever we have an L2 miss, we enter runahead mode. We take a checkpoint of the architectural register file and the branch predictors. Similar to setting the poison bit in the delayed selective replay scheme (see Section 5.2.4), we set the invalid bit for the destination register of the load that missed in the L2 cache. We subsequently propagate this poison bit to all the instructions in the load's forward slice. Recall that the forward slice of an instruction consists of its consumers, its consumers' consumers and so on. The invalid bit is propagated via the bypass paths, the LSQ, and the register file. This ensures that all the consumers of an instruction receive an operand marked as invalid. If any of the sources are invalid, the entire instruction including its result is marked as invalid. An instruction that is not marked invalid, is deemed to be valid.

We execute instructions in the runahead mode as we execute them in the normal mode. The only difference is that we always keep track of the valid/invalid status of instructions. Second, we do not update the branch predictor when we resolve the direction of a branch that is invalid.

Runahead execution introduces the notion of *pseudo-retirement*, which means retirement in runahead mode. Once an instruction reaches the head of the ROB, we inspect it. If it is invalid, we can remove it immediately, otherwise we wait for it to complete. We never let stores in the runahead mode write their values to the normal cache. Instead, we keep a small runahead L1 cache, where the stores in runahead mode write their data. All the loads in the runahead mode first access the runahead cache, and if there is a miss, they are sent to the normal cache. Furthermore, whenever we evict a line from the runahead cache, we never write it to the lower level.

Once the value of the load that missed in the L2 cache arrives, we exit the runahead mode. This is accompanied by flushing the pipeline and cleaning up the runahead cache. We reset all the invalid bits, and restore the state to the checkpointed state that was collected before we entered the runahead mode.

There are many advantages of this scheme. The first is that we keep track of the forward slice of the load that has missed in the L2 cache. The entire forward slice is marked as invalid, and we do not allow

instructions in the forward slice to corrupt the state of the branch predictor and other predictors that are used in a pipeline with aggressive speculation. This ensures that the branch prediction accuracy does not drop after we resume normal execution. The other advantage is that we use the addresses of valid memory instructions in runahead mode to fetch data from the memory system. This in effect prefetches data for the normal mode, which is what we want.

Helper Threads

Let us now look at a different method of doing what runahead execution does for us. This method uses *helper threads*. Recall that a thread is defined as a lightweight process. A process can typically create many threads, where each thread is a subprocess. Two threads share the virtual memory address space, and can thus communicate using virtual memory. They however have a separate stack and program counter. Programs that use multiple threads are known as multithreaded programs.

Definition 49
A thread is a lightweight process. A parent process typically creates many threads that are themselves instances of small running programs. However, in this case, the threads can communicate amongst each other via their shared virtual address space. Each thread has its dedicated stack, architectural register state, and program counter.

The basic idea of a helper thread is as follows. We have the original program, which is the parent process. In parallel, we run a set of threads known as helper threads that can run on other cores of a multicore processor. Their job is to prefetch data for the parent process. They typically run small programs that compute the values of memory addresses that will be used in the future. Then they issue prefetch requests to memory. In this manner, we try to ensure that the data that the parent process will access in the future is already there in the memory system. Let us elaborate.

First, let us define a *backward slice*. It is the set of all the instructions that determine the value of the source operands of an instruction. Consider the following set of instructions.

```
1 add r1, r2, r3
2 add r4, r1, r1
3 add r5, r6, r7
4 add r8, r4, r9
```

The backward slice of instruction *4* comprises instructions *1* and *2*. It does not include instruction *3*. Of course, the backward slice of an instruction can be very large. Nevertheless, if we consider the backward slice in a small window of instructions, it is limited to a few instructions.

Definition 50
The backward slice of an instruction comprises all those instructions that determine the values of its source operands. It consists of the producer instruction of each operand, the producers of the operands of those instructions, and so on.

We can create small subprograms for loads that may most likely miss in the L2 cache, and launch them as helper threads way before the load gets executed. To figure out which loads have a high likelihood of missing in the L2 cache, we can use an approach based on profiling, or prediction based on misses in the past. Each helper thread runs the backward slice of such a load instruction, computes its address, and sends it to the memory system for prefetching the data.

7.8 Summary and Further Reading

7.8.1 Summary

Summary 6

1. *Most programs exhibit temporal and spatial locality.*
2. *Given that large memories are too slow and too inefficient in terms of power consumption, we need to create a memory hierarchy.*
   (a) *A typical memory hierarchy has 3-5 levels: L1 caches (i-cache and d-cache), L2 cache, L3 and L4 caches (optional), and the main memory.*
   (b) *The cache hierarchy is typically inclusive. This means that all the blocks in the L1 cache are also contained in the L2 cache, and so on.*
   (c) *The performance of a cache depends on its size, latency, and replacement policy.*
3. *The salient features of a cache are as follows.*
   (a) *A cache has a tag array and a data array.*
   (b) *We first search for an entry in the tag array, and only if there is a tag match, we get an index for the entry in the data array.*
   (c) *A direct mapped cache maps a block address to only one cache line. However, a k-way set associative cache maps each block address to a set of k lines (a set). In a fully associative cache, a block can be stored in any cache line.*
   (d) *A direct mapped cache and a set associative cache are made of 6-transistor SRAM cells, whereas the tag array in a fully associative cache uses 10-transistor CAM cells.*
   (e) *Every memory cell is connected to two bit lines. Before reading the value, we precharge the bit lines and then monitor the difference in the voltages using a sense amplifier. This is done to increase performance, and reduce the susceptibility to noise.*
4. *It is necessary to have support for virtual memory.*
   (a) *Solves the* size *and* overlap *problems.*
   (b) *The virtual to physical address mapping is stored in the page tables.*
   (c) *The TLB (Translation Lookaside Buffer) acts as a small and fast hardware cache for the page tables. The TLB is accessed before sending a memory request to the cache.*
5. *We can use the Cacti tool to model and design a cache.*
   (a) *We replace each element in the cache by an equivalent RC circuit. This helps us estimate the latency and power.*
   (b) *We calculate the time constant of each element using the Elmore delay model.*
   (c) *Subsequently, we divide large arrays into banks, subbanks, mats, and subarrays based on the objective function: we can either minimize power, minimize latency, or minimize area.*

6. *Modern caches are pipelined and are non-blocking. They have miss status handling registers (MSHRs) that do not allow secondary misses to be sent to the lower levels of the memory hierarchy. We record a secondary miss in the cache, when at the point of detecting a miss, we find that we have already sent a request to the lower level for a copy of the block. Such misses are queued in the MSHR.*

7. *We can use skewed associative caches, way prediction, loop tiling, and VIPT (virtually indexed, physically tagged) caches to further increase the performance of a cache.*

8. *The trace cache stores traces, which are commonly executed sequences of code. We can read decoded instructions directly from it, and skip the fetch and decode stages altogether.*

9. *There are three kinds of misses: compulsory, conflict, and capacity. For all these types of misses, prefetching is helpful.*

10. *We can prefetch either instructions or data. We learn patterns from the miss sequence of a cache, and then leverage them to prefetch data or code blocks.*

11. *For instruction prefetching, next line prefetching is often very effective. However, modern approaches prefetch at the level of functions or groups of functions. They take the high-level structure of the code into account.*

12. *For prefetching data, we studied stride based prefetching for regular memory accesses and pre-execution based methods for irregular memory accesses. The latter class of techniques is very important for code that uses linked lists and trees.*

7.8.2 Further Reading

For a basic introduction to caches and virtual memory, we recommend the textbook by Sarangi [Sarangi, 2015]. For more advanced concepts, readers can refer to the book titled, "Multi-core Cache Hierarchies", [Balasubramonian et al., 2011] by Balasubramonian, Jouppi, and Muralimanohar.

For cache modeling, arguably the best references are Cacti's design manuals: Cacti 1.0 [Wilton and Jouppi, 1993], Cacti 2.0 [Reinman and Jouppi, 2000], Cacti 3.0 [Shivakumar and Jouppi, 2001], Cacti 4.0 [Tarjan et al., 2006], Cacti 5.0 [Thoziyoor et al., 2007], and Cacti 6.0 [Muralimanohar et al., 2009]. For additional details, we recommend the book by Jacob et al. [Jacob et al., 2007]. Readers interested in analytical models can consult the papers by Guo et al. [Guo and Solihin, 2006] and the Roofline model [Williams et al., 2009]. The Roofline model is a general model that correlates compute and memory performance. We shall look at it separately in Chapter 10. For a better understanding of the popular LRU and FIFO replacement schemes including their mathematical underpinnings, we recommend the book by Dan and Towsely [Dan and Towsley, 1990]. In this chapter, we have only explained inclusive caches; however, making caches inclusive is not necessary. Gaur et al. [Gaur et al., 2011] present an alternative design where it is possible to bypass the last level cache in the memory hierarchy.

For additional details on prefetching, readers can refer to the survey paper by Mittal [Mittal, 2016b] and Callahan et al.'s survey paper on software prefetching [Callahan et al., 1991].

Exercises

Ex. 1 — A cache has block size b, associativity k, and size n (in bytes). What is the size of the tag in bits? Assume a 64-bit memory system.

Ex. 2 — Does pseudo-LRU approximate the LRU replacement scheme all the time?

Ex. 3 — From the point of view of performance, is an i-cache miss more important or is a d-cache miss more important? Justify your answer.

Ex. 4 — Why do write buffers and victim caches work?

Ex. 5 — Is it a good idea to have an L4 and L5 cache as well?

Ex. 6 — Why is it necessary to read in the entire block before even writing a single byte to it?

* **Ex. 7** — Assume the following scenario. An array can store 10 integers. The user deliberately enters 15 integers, and the program (without checking) tries to write them to successive locations in the array. It will cross the bounds of the array and overwrite other memory locations. It is possible that if the array is stored on the stack, then the return address (stored on the stack) might get overwritten. Is it possible to hack a program using this trick? In other words, can we direct the program counter to a region of code that it should not be executing? How can we stop this attack using virtual memory based techniques?

Ex. 8 — Assume we have an unlimited amount of physical memory, do we still need virtual memory?

* **Ex. 9** — Does the load-store queue store physical addresses or virtual addresses? What are the trade-offs. Explain your answer, and describe how the load-store queue needs to take this fact (physical vs virtual addresses) into account.

Ex. 10 — In a set associative cache, why do we read the tags of all the lines in a set?

Ex. 11 — What is the key approximation in the Elmore delay model? Why do we need to make this assumption?

Ex. 12 — Show the design of an MSHR where every load checks all the previous stores. If there is a match, then it immediately returns with the store value (similar to an LSQ).

Ex. 13 — Does the VIPT scheme place limits on the size of the cache?

Ex. 14 — Why was it necessary to store a trace in consecutive sets? What did we gain by doing this?

** **Ex. 15** — Consider an application that makes a lot of system calls. A typical execution is as follows. The application executes for some time, then it makes a system call. Subsequently, the OS kernel starts to execute, and then after some time, it switches back to the application. This causes a lot of i-cache misses. Suggest some optimizations to reduce the i-cache miss rate.

* **Ex. 16** — Can you design a piece of hardware that can detect if an OOO processor is traversing a linked list?

** **Ex. 17** — Suggest an efficient hardware mechanism to prefetch a linked list. Justify your answer. Extend the mechanism to also prefetch binary trees.

Design Questions

Ex. 18 — Understand the working of the CACTI tool. Create a web interface for it.

Ex. 19 — Implement way prediction in an architectural simulator such as the Tejas Simulator.

Ex. 20 — Implement a trace cache in an architectural simulator.

Ex. 21 — Implement different prefetching techniques and compare their performance.