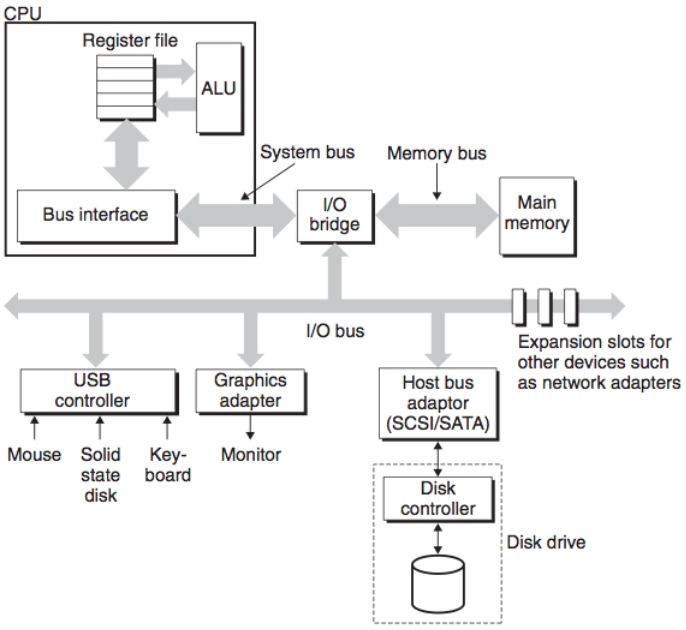
We will now turn our focus towards the various memory technologies available in a computer system. As programmers, knowledge of the various memory technologies is useful because each level of memory has it own performance characteristics and limitations that can impact the performance of our code. Each unit in the memory system has a uniqiue access time, cost and storage capacity. For example, y86 has 8 registers that require mere picoseconds to access, however, increasing the number of registers in an effective manner requires redesigning significant portions of the chip. If we care about backwards compatibility, this can become quite expensive due to the engineering task of designing a chip with these new properties. It may be quite cheaper, to just add more RAM or cacheing to the system.



Storage Technologies

RAM

RAM stands for Random-Access Memory which means that storage blocks may be accessed in any arbitrary order. This is contrasted to Sequential-Access Memory devices such as a CD-ROM or hard disk which read (or write) data in sequence and must "seek" to the appropriate location if out-of-order access is requested. There are many types of RAM on the market today, but they fall into two categories: *static (SRAM)* and *dynamic (DRAM)*

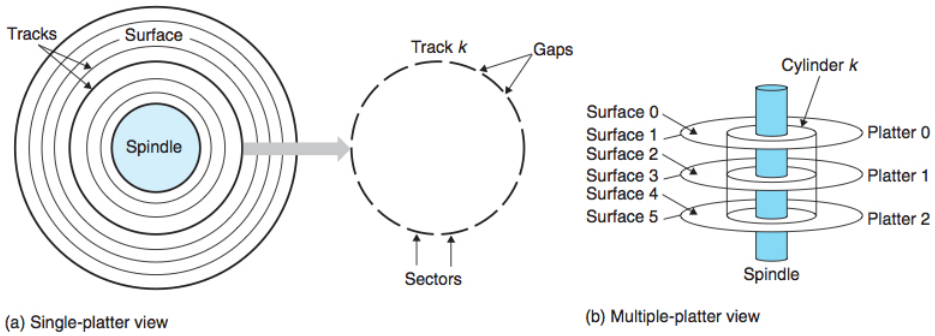
SRAM vs. DRAM

Both DRAM and SRAM are examples of *volatile* memory. Voltage is required to maintain state. However, the stability and time-to-live for bits that are set in each time of memory differ due to the way they are constructed. SRAM is cpmprised of *bistable* memory cells. Each has two stable states and an unstable intermediary state. Once an SRAM cells has been set to one of the stable states, it can retain this value indefinitely however when it is in the unstable state, even the smallest power disturbance will throw it off balance. A good analogy is to look at a pendulum turned upside down. When the pendulum is vertical it is in the ubstable state. DRAM uses a very small capacitor to hold a charge. This creates a very dense memory unit that is sensitive to electrical disturbances and depending on conditions, each unit will lose its charge within the range of 10ms to 100ms.

Memory Type	Transistors per bit	Relative Access Time	Persistent	Sensitive	Relative Cost	Typical Usage
SRAM	6	1x	Yes	No	100x	Cache
DRAM	1	10x	No	Yes	1x	Main memory, sensors in digital cameras, frame buffers

Disk Storage

Disks are comprised of two-sided platters coated with a magenetic material and a rotating spindle in the center that operates with a fixed rotational rate typically between 5400 and 15000 RPMS (revolutions per minute). Each platter surface consists of a cocentric ring of tracks which is partitioned into sectors. Each sector contains an equal # of bytes (typically 512).



Example:Given a disk drive with 5 platters, 512 byte sectors, 20,000 tracks per surface and an average of 300 sectors per track. What is the total capacity of the drive?

```
Capacity = [ bytes / sector ] * [ sector / track ] * [ track / surface ] * [surface / platter ] * [ platter / disk ]

         =  (512)*(300) * ( 20000) * (2) * (5)

         =  30, 720, 000, 000 bytes
```

```
=  30.72 GB
```

Access Time
Acces time on disk storage is determined by the seek time, rotational latency and transfer time.

- **Seek Time**: is the time is takes for the read/write head of the disk to be positioned over the track that contains the target sector.
- **Rotational latency**: is the time it takes for the first bit of the target sector to pass underneath the read/write head. The performance will vary based on the position of the surface when the head arrives an the rotational speed of the disk. The worst case scenario is when the head arrives just after the target sector passes and must wait for a full rotation.
- **Transfer Time**: is the time it takes to read/write the data to disk.

Example: Estimate the average time in milliseconds to access a sector on a disk with the follwing properties:

Property	Value
Rotational Rate	15,000 RPM
Average Seek Time	8ms
Average #sectors/track	500

```
Average access Time = Average Rotational Latency + Average Seek Time  + Average Transfer Time

We will first compute the average rotational latency which is just 1/2 the rotation rate of the disk.

        R_avg = .5 * R_max
                R_max = 1 / (15000 RPM) * (60000 mseconds/minute) = 4ms
                = .5 * (4)
                    = 2ms

Now we  compute the average transfer time:

        T_avg  = (1/RPM) * (1/avg. sectors/track) * 60seconds/minute
               = (1/15000) * (1/500) * (60)
               = .008 ms

        AT = R_avg + T_avg + T_seek
              = (2ms) + (.008ms) + (8ms)
              = ~10ms
```

Modern computer systems make use of a *disk controller* which abstracts away the specifics of disk geometry and presents the system with a simple view of disk memory as a sequence of B *logical blocks* each the size of one sector.

Example:Given a disk with a rotational rate of 10,000 RPMs, and average seek time of 5ms, an average of 1000 sectors per track, 2 platters and a sector size of 512 bytes. Suppose a program reads 512 byte logical blocks sequentially and the time to position the head over the first block is average seek time + average rotation time, answer the following about a 1MB file:

1. Best case scenario: logical blocks are mapped to sequential locations on disk.

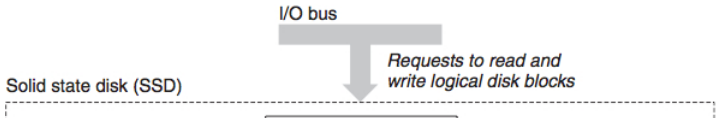
```
    A 1MB file will consist of ~2000 (2048) 512-byte logical blocks.  Average seek time is 5ms, Max rotation= 6 ms and average rotation = 3ms.

    In the best case we only need to seek once since the logical blocks are mapped to a contiguous sector. We'll need two rotations to read all ~2000 sectors:

        3ms + 5ms + 2 * 6ms
      = 3ms + 5ms + 12ms
     = 20ms
```

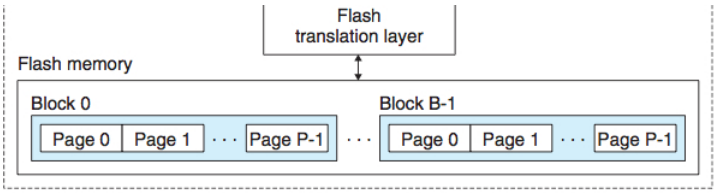
2. Random case: logical blocks are mapped to random disk sectors.

```
    With random locations we will have to incur the full seek cost with each logical block we read.  This demonstrates why it is helpful to defragment hard dri


2000 * (8ms + T_avg)

    T_avg = (1/RPM) * (1/sectors/track) * (60seconds/minute)
                = (1/10000) * (1/1000) * 60
            = .006ms

2000 * (8.006) = 16012ms
                            ~ 16000ms
```

Solid State Disk (SSD)

Solid State Disks are a type of *flash memory* which is an *electrically eraseable and programmable read-only memory (EEPROM)*





SSDs in general provide an improvement on read/write access over disk, however random writes may bottleneck due to the underlying properties of SSD drives which require an entire block to be erased before a write can take place. This can be problematic if only a single page in the block needs to be written because the remaining pages will also have to be rewritten as a consequence of the write. Additionally, a block wears out after approximiately 100,000 repeated writes. Random access times are much faster due to the lack of moving parts.

How long will an SSD last?Example: MEGASUPER SSD &tm; guarantees 1 petabyte (10^9 MB) of random writes befire their SSDs will wear out. What is the estimated lifetime of a MEGASUPER SSD for the following workloads, given the following performance characteristics:
Sequential read throughput: 250 MB/s
Random read throughput:: 140 MB/s
Random read access time:: 30 microseconds
Sequential write throughput: 170 MB/s
Random write throughput: 14 MB/s
Random write access time: 300 microseconds

1. SSD is written to continuosly at a rate of 170 MB/s.

```
We can use dimensional analysis and unit conversion to arrive at a solution:

   [MB/s] * [s] =  MB
    rate * time = lifespan

   time = lifespan/rate
    = (10^9 MB )/(170 MB/s)
    = (1 000 000 000) / 170
    = 5 882 352.94 * (1 year / (365 * 24 * 60 * 60 ) seconds)
  = 5 882 352.94 / 31 536 000 seconds
 =  .19 years
```

2. SSD is written to continuously at a rate of 14 MB/s.

```
      time = lifespan/rate
   = (10^9 MB )/(14 MB/s)
   = (1 000 000 000) / 170
   = 71 428 571.43 / (1 year / (365 * 24 * 60 * 60 ) seconds)
  = 71 428 571.43  / 31 536 000 seconds
 =  2.26 years
```

3. SSD is written to at a rate of 20 GB/day

```
   time = lifespan/rate
    = (10^9 MB )/(20 GB/day)
   = (10^9MB)(1 GB/1000MB)/ ( 20 GB/day )
   = (1 000 000 GB) / (20 GB/day)
  =  50000 days
  =  (50000 days) (1 year/365 days)
 = ~ 137 years
```

Locality

Locality is a concept that refers to referencing data items that are near other recently references data items or items that were themselves recently referenced. This *principle of locality* can have a big impact on the design and performance of comptuer systems. There are two disctinct forms of locality: *temporal* and *spatial*

- A program is said to have good **temporal locality** if a memory location that is referenced once is likely to be referenced again multiple times in the near future.
- A program is said to have good **spatial locality** if a memory location is references once and the program is likely to reference a nearby memory location in the near future.

Consider the following C code for computing the sum of the elements in a vector:

```
int vector_sum(int v[N]) {

  int i,sum=0;

  for(i=0;i<N;i++)
    sum+=v[i];
  return sum;

}
```

If we look at variables ***i*** and ***sum*** we can see that they both have good temporal locality since they are accessed in each iteration of the loop. Since they are both scalar values, there is no spatial locality with respect to ***sum*** or ***i***.

```
for N=8, the reference pattern for our vector, v:

Contents                v0      v1      v2      v3      v4      v5      v6      v7

Address                 +0      +4      +8      +12     +16     +20     +24     +28

Access Order            1       2       3       4       5       6       7       8
```

Since we read the values of our vector sequentially, variable ***v*** enjoys good spatial locality but poor temporal locality since we only access each element once. Since each variable in the function has either good spatial or temporal locality, we can classify the function has good locality.

Stride-k referencing A function like vector_sum is said to have a *sequential reference pattern* (with respect to its element size) because it visits each element of the vector sequentially. Visiting every kth element in contiguous memory block is called a *stride-k reference pattern*. As a general principle, spatial locality and the stride width have an inverse relationship.

Question: Why do we benefit from having a function with good temporal locality?

One key reason: registers! We only have a limited number of registers. If our code exhibits poor temporal locality, then these registers will have to either be saved to memory or pushed onto the stack for retrieval later. Good temporal locality means our registers can maintain their values and we can minimize which ones must be written to memory or pushed onto the stack.

Example: Modify the following C function so that the 3-dimensional array is scanned with a sequential reference pattern.

```
int sum_array_3D(int a[N][N][N]) {

  int i,j,k, sum=0;

  for(i=0; i<N; i++) {
    for(j=0;j<N;j++) {
      for(k=0;k<N;k++) {
        sum+= a[k][i][j];
      }
    }
   }
}
```

To ensure we have a sequential reference pattern, we want the rightmost index to change most often:

```
int sum_array_3D(int a[N][N][N]) {

  int i,j,k, sum=0;

  for(k=0; k<N; k++) {
    for(i=0;i<N;i++) {
      for(j=0;j<N;j++) {
        sum+= a[k][i][j];
      }
    }
   }
}
```

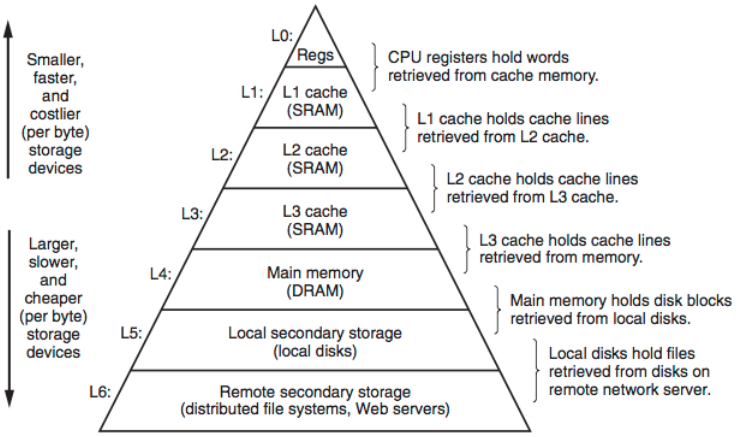
Example: For the following C functions, rank-order the functions with respect to spatial locality. Explain the ranking.

```
  #define N 1000

  typedef struct {
    int vel[3];
    int acc[3];
  } point;

  point p[N];

  void clear1(point *p, int n) {

    int i, j;

    for(i=0; i<n; i++) {
      for(j=0; j<3 j++)
        p[i].vel[j]=0;
      for(j=0; j<3 j++)
        p[i].acc[j]=0;
    }
  }


 void clear2(point *p, int n) {
    int i, j;

    for(i=0;i<N;i++) {
      for(j=0;j<3;j++) {
        p[i].vel[j]=0;
        p[i].acc[j]=0;
      }
    }
 }


  void clear3(point *p, int n) {
    int i,j;

    for(j=0; j<3;j++) {
      for(i=0;<n;i++)
        p[i].vel[j]=0;
      for(i=0;<n;i++)
        p[i].acc[j]=0;
    }
  }
```

Recall how structs are laid out in memory. Our point data type would look something like this:

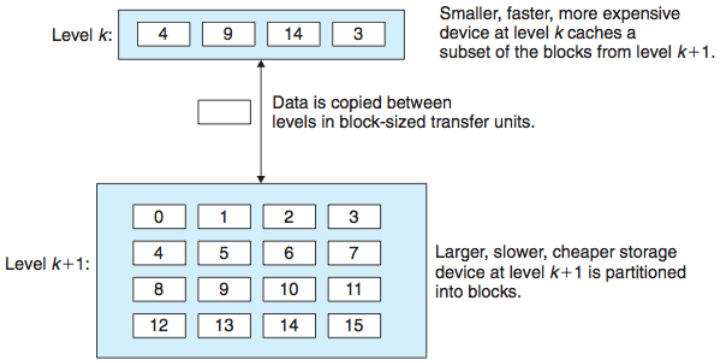
```
  +0    +4      +8      +12     +14     +16
vel[0]  vel[1]  vel[2]  acc[0]  acc[1]  acc[2]

so to achieve a sequential reference pattern, we want to accees each element of vel followed by each element of acc for each point.

clear1 accomplishes this and so is the best in terms of spatial locality.
clear2 achieves a sequential reference pattern between each point but has a stride-3 reference pattern within each struct.
clear3 is our worst performer, not only does it hop around within each struct but also between each struct.
```

The Memory Hierarchy and Caching



In general, a ***cache*** is a small, fast storage device that acts as a staging area for data stored in larger, slower devices. The central idea is that for a given memory hierarchy, the smaller, faster storage device at level *k* serves as a cache for level *k+1*

Cache Organization



Cache memory is organized as an array of S *cache sets*, each consisting of E *cache lines* made up of B *data blocks* plus indexing information. The size of a cache is determined by:

```
C = S x E x B
    = 2^s x E x 2^b
```

Each cache line contains indexing bits comprised of a *valid bit* that determines if the line contains meaningful data and a set of *tag bits* that uniquely identify the block stored in a cache line. In general, we describe a cache using a four-tuple where *m* is the number of bits that form M=2^m unique memory addressess in a computer system.

Cache Parameters:

Parameter	Description
S = 2^s	Number of sets
E	Number of lines per set
B = 2^b	Block size(bytes)
m = log_2(M)	Number of physical address bits

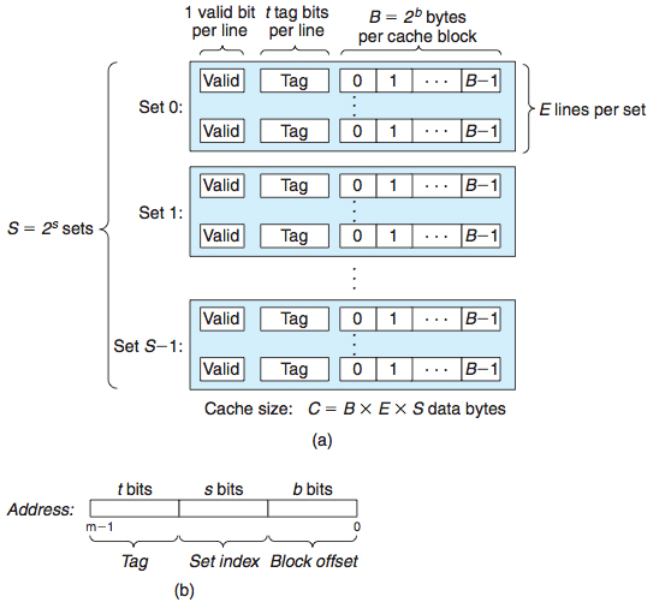
Derived Quantities

Parameter	Description
M = 2^m	Maximum # of unique memory addresses supported

s = log_2(S) Number of set index bits

b = log_2(B) Number of block offset bits

t = m - (s+b) Number of tag bits



How does the computer know if a particular word is cached? The memory addresses is used to imply the set bits, tag bits and block offset.

1. The set index bits designate the cache set
2. The tag index bits designate the cache line
3. The line holds the word if and only if the valid bit of the line is set to 1 (true) AND the tag bits of the line match the tag bits in the address.
4. The block offset bits, tell us which word to use from the block of B-byte words.

Example: For a cache of size 1024 on a 32-bit system, determine the set size and number of tag, set and block offset bits given a block size of 4 bytes and 4 lines per set.

```
For a 32-bit system we have 2^32 possible physical memory addresses, so m= 32.

for the set size

S =  C / E* B
S = 1024 / (4)*(4)
S = 64

We will need 2 bits to represent the block size of 4, so b=2.

we need 6 bits to represent the set bits, so s= 6
log_2(S) = (log_2(64)) = 6


for tag bits, t = m - (s+b)

   t = 32 - ((6) + (2))
```

```
t =24
```

We determine if a paritcular word is in the cache by taking the memory addresses and extracting the set bits, tag bits and block offset. The set index bits designate the cache set. The tag index bits designate the cache line. The line is determined to hold the word if and only if the valid bit is set to true (1) AND the tag bits stored in the line match the tag bits from the memory address.

Cache-Mapping
Caches are grouped into classes based on the number of caches lines per set (E).

- **Direct-Mapped Cache:** In a direct-mapped cache there is only one cache line per set (E=1). This is the simplest implementation.
- **Set-Associative Cache:** In a set-associative cache, each set holds more than one cache line such that 1 < E < C/B. This is known as an *E-Way set associative cache*. Set associative caches help address conflicts between memory addresses that share the set bits.
- **Fully-Associative Cache:** In a fully associative cache, there is only one set (S=1) and this set contains all of the cache lines. Fully associative caches are only appropriate for very small caches.

Cache Matching

Cache matching is the process that determines if a particular word is in the cache, known as a *cache hit* or if it is not in the cache, known as a *cache miss*. The process for determining if a word is in the cache or not is accomplished in three general steps:

1. Set selection: use set bits to select the correct set.
2. Line matching: determing which cache line in the set matche (if any)
3. Word extraction: use the block offset to determine which word to extract. Think of the block as just an array of bytes and the block offset as an index into the array

Cache Misses

When there is a cache miss, the value must be loaded into the cache from the memory unit that is the next level down in the hierarchy. How we handle cache misses depends on the type of cache. If all of the cache lines are valid or at the very least the cache line the word maps to is used, then there must be an evicition policy to determine how the cache line is updated.

For direct-mapped caches, a cache miss will cause the current cache line value to be replaced with the newly fetched value.

For set-associative caches, an eviction policy must be in place since each set has multiple lines. If there is a cache miss, the policy must determine which line to use. If a line is avaialble, it will use it but if all lines have valid cached data it must choose which to evict. Unfortunately for programmers, there is little we can do in our code to take advantage of eviction policy. The most common eviction policies are:

1. Least-Frequently Used (LFU): the line that has been referenced the fewest times over a time window is evicted.
2. Least-Recently Used (LRU): the line that was accessed the furthest in the past is evicted
3. At Random: as its name implies, the evicted cache line is chosen at random

Example:Given a direct-mapped cache: (S,E,B,m) = (4,1,2,4). Let's look at what happens as we request a set of values from the cache assuming a word size of 1 byte.

Our cache contains four sets, so s=2 bits. We have two bytes per block (B), so b=1 and we have a 4-bit address space: m=4.

so t = m - (s+b)
t = 4 - ((2) + (1))
t = 1

Cache size:
S*B*E = (4) * (2) * (1) = 8 bytes

Our address space looks something like this:

Address(decimal)	Tag bits (t=1)	Set index bits (s=2)	Offset bits (b=1)
0	0	00	0
1	0	00	1
2	0	01	0
3	0	01	1
4	0	10	0
5	0	10	1
6	0	11	0
7	0	11	1
8	1	00	0
9	1	00	1
10	1	01	0
11	1	01	1
12	1	10	0
13	1	10	1
14	1	11	0
15	1	11	1

Here is what the cache looks like initially:

Set	Valid	Tag	block[0]	block[1]
0	0			
1	0			
2	0			
3	0			

1. Read word at address 0x0

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

2. Read word at address 0x1

This is a cache hit, so the cache remains unchanged.

3. Read word at address 0xC

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

4. Read word at address 0x8. Maps to set 0, but tag is 1 so this will be a cache miss

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

5. Read word at address 0x1. Unfortunately another cache miss, since we just replaced it with the read at 0x8.

Set	Valid	Tag	block[0]	block[1]
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Example: Now, lets read the same sequence of words but change our cache to be defined by C = (2,2,2,4)

Notice that this is a *set associative* cache. Let's assume it uses a LFU eviction policy. 1. Read word at address 0x0

Set,Line	Valid	Tag	block[0]	block[1]
0,0	1	00	m[0]	m[1]
0,1	0			
1,0	0			
1,1	0			

2. Read word at address 0x1

This is a cache hit, so the cache remains unchanged.

3. Read word at address 0xC. This now maps to set 0.

Set,Line	Valid	Tag	block[0]	block[1]
0,0	1	00	m[0]	m[1]
0,1	1	11	m[12]	m[13]
1,0	0			
1,1	0			

4. Read word at address 0x8. Maps to set 0, but is a cache miss. all lines are in use, so eviction policy must be used. line 1 will be evicted since it has only been accessed once and line 0 has been accessed twice.

Set, Line	Valid	Tag	block[0]	block[1]
0,0	1	00	m[0]	m[1]
0,1	1	10	m[8]	m[9]
1,0	0			
1,1	0			

5. Read word at address 0x1. This time, we still get a cache hit!

Set, Line	Valid	Tag	block[0]	block[1]
0,0	1	00	m[0]	m[1]
0,1	1	10	m[8]	m[9]
1,0	0			
1,1	0			

Question: Why do we use the middle bits of a memory address for the set index?

If we used the high order bits then contiguous memory blocks will map to the same cache set. If a program has good spatial locality, then the cache can only hold block-sized chunks of the array in the cache at any given time. With middle-order bit indexing, the cache can hold a C-size chunk of the array (C= cache size) at any given time.

Cache Write Policy
It is simple to read values from a cache, but what happens when we update a value. How do we make sure the cache has the latest value?

Write-Hits A *write-hit* occurs when the word being written is already cached. After the cache updates its copy of w, what does it do for the next level in the hierarchy. There are two main approached:

1. **write-through**: After updating the cached value, the new value is immediately written to the next level in the memory hierarchy. This is a simple implementation but it creates significant bus traffic.
2. **write-back**: After updating the cached value, a *dirty bit* is set to indicate the cache value needs to be written back to lower levels in the hierarchy. The update is defered until the cache line is evicted by its replacement policy. This reduces bus traffic but adds complexity to the hardware. (read expense).

Write-misses A *write-miss* occurs when a value is written to an address that is not currently in the cache. The two main strategies for write-misses are:

1. **write-allocate**: the corresponding block from the next lower level into the us loaded and then the cache block is written with the updated value.
2. **no-write-allocate**:this approaches bypasses the cache all-together and writes the word directly to the next lower-level.

Write-miss strategy operates independent of write-hit strategy but in general write-through caches are usually no-write-allocate and write-back caches are typically write-allocate.

Cache-friendly Coding

Writing cache-friendly code is not an *exact* science, but a couple of good principles to follow:

- Focus optimizing the most common task: Long running programs usually spend most of their time in a few core functions. Focus on those functions and address things like temporal and spatial locality. If the functions have loops, focus on the innermost loops and ignore much of the test.
- Try to minimize the number of cache misses inside inner loops.

Example: Assume a cache with blocks of 4 words on a system with 4 byte words and block aligned memory allocatoin (variables, allocated memory, etc will point to the first memory address in a block). Trace the cache hits and misses for the following C function:

```
int sumarraycols(int a[M][N]) {

  int i,j,sum=0;

  for(j=0;j<N;j++)
    for(i=0;i<M;i++)
      sum += a[i][j];

  return sum;
}
```

a[i][j]	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
i = 0	1 **[m]**	5 **[m]**	9 **[m]**	13 **[m]**	17 **[m]**	21 **[m]**	25 **[m]**	29 **[m]**
i = 1	2 **[m]**	6 **[m]**	10 **[m]**	14 **[m]**	18 **[m]**	22 **[m]**	26 **[m]**	30 **[m]**
i = 2	3 **[m]**	7 **[m]**	11 **[m]**	15 **[m]**	19 **[m]**	23 **[m]**	27 **[m]**	31 **[m]**
i = 3	4 **[m]**	8 **[m]**	12 **[m]**	16 **[m]**	20 **[m]**	24 **[m]**	28 **[m]**	32 **[m]**

Our best scenario is if the cache size is greater than or equal to the array size. In this case, our miss rate is 1 out of every 4 accesses. If, however, our array is larger than the cache then every single access will be a cache miss!By swapping the loops we can get much better peformance due to the fact that 2-d arrays are stored in row-major order in C:

```
int sumarraycols(int a[M][N]) {

  int i,j,sum=0;

  for(i=0;i<M;i++)
    for(j=0;j<N;j++)
      sum += a[i][j];
  return sum;
}
```

a[i][j]	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
i = 0	1 **[m]**	2 [h]	3 [h]	4 [h]	5 **[m]**	6 [h]	7 [h]	8 [h]
i = 1	9 **[m]**	10 [h]	11 [h]	12 [h]	13 **[m]**	14 [h]	15 [h]	16 [h]
i = 2	17 **[m]**	18 [h]	19 [h]	20 [h]	21 **[m]**	22 [h]	23 [h]	24 [h]
i = 3	25 **[m]**	26 [h]	27 [h]	28 [h]	29 **[m]**	30 [h]	31 [h]	32 [h]

Cache Performance
The metrics for cache performance are:

1. Miss rate: the fraction of references that cause a cache miss.
   Miss rate = # misses / # of references
2. Hit rate: the fraction of references that are a cache hit.
   Hit rate = 1 - Miss rate.
3. Hit time: the time to deliver the word in the cache to the CPU. Usually on the order of several clock cycles for on-chip caches (i.e. L1).
4. Miss penalty: Any additional time required because of a cache miss.

Example Given a 1024-byte direct-mapped cache with 16 byte block sizes on machine with word size=4. Using the following C code and assuming that variables i,j,

total_x and total_y are stored in registers, the cache is initially empty and grid begins at memory address 0, answer the following:

```
typedef struct point {
  int x;
  int y;
}

...

for(i=0;i<16;i++) {
  for(j=0;j<16;j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
  }
}
```

- What is the total # of reads?

  512 reads (16 * 16 * 2)

- What is the total # of reads that miss in the cache

  Because we have a nice stride-1 access pattern, our initial cold misses are the only cache misses. Each miss loads 16 byte blocks into the cache, so our # of misses is 128.

- What is the miss rate

  Miss rate = 128/512 = 25%

- What would be the miss rates if the cache were twice as big?

  The miss rate would be the same if the cache were twice as big since we cannot avoid cold misses.

- What would be the miss rate if the cache were only 1/4th the size?

  Because we can't avoid the cold misses and we have a sequential access pattern in our loop, the cache miss rate would be the same.