32 INSTRUCTION-PHASE DEPENDENCIES

The advantage of partitioning a CPU into stages with pipeline registers is most significant because it enables the CPU to process more then one instruction at a time. With pipeline registers holding the key values necessary to execute a given stage, each stage can be executed independently of the other. This permits the processing of different instructions at different stages within the CPU. To visualize better what part of each instruction a CPU is currently processing, it is helpful to construct a "instruction-phase" diagram. This diagram plots the stages of execution of each instruction of a program as a function of time, with time expressed as a succession of "phases" according to the multi-stage model.

Some examples:

3-stage CPU : The original UDF diagram of the add instruction suggests a 3-stage CPU; that is, the retrieval and execution of an instruction consists of an instruction fetch (IF) stage, an instruction decode (DEC) stage, and an execute (EX) stage. For a four-instruction programming segment, the "ideal" instruction-phase diagram is:

	PHASE									
INST	–1–	–2–	–3–	–4–	–5–	–6–	–7–	–8–	–9–	-10–
1.	IF	DEC	EX							
2.		IF	DEC	EX						
3.			IF	DEC	EX					
4.				IF	DEC	EX				

The instructions are not explicitly identified except by their order in the program. However, for any given phase, one can observe which component of the 3-stage CPU is currently processing which instruction.

This diagram can be used to compute the average execution time of an instruction by the 3-stage CPU. The total execution time is given by the previously determined minimum clock period (76 ns) times the number of phases used in the execution of the program, divided by the number of instructions in the program. In this example:

$$\frac{76 \text{ ns} \times 6 \text{ phases}}{4 \text{ instructions}} = 456/4 = 114 \text{ ns}$$

5 stage CPU : The UDF diagram of the add instruction was partitioned into 5 stages with pipeline registers. The stages were identified as instruction fetch (IF), instruction decode (DEC), execute (EX), data memory access (MEM), and write back (WB). A 4-instruction program, expressed as an "ideal" instruction-phase diagram is shown on the next page.

	PHASE									
INST	–1–	–2–	–3–	–4–	–5–	–6–	–7–	–8–	–9–	-10–
1.	IF	DEC	EX	MEM	WB					
2.		IF	DEC	EX	MEM	WB				
3.			IF	DEC	EX	MEM	WB			
4.				IF	DEC	EX	MEM	WB		

Using the previously determined clock period of 46 ns, the average execution time per instruction is:

$$\frac{46 \text{ ns} \times 8 \text{ phases}}{4 \text{ instructions}} = 368/4 = 92 \text{ ns}$$

32.1 Resource Conflicts

The previous instruction-phase diagrams were "ideal." That is, it was assumed that there were no "resource conflicts." A resource conflict occurs when two instructions require the same component at the same time. The following example illustrates how one can arise. Consider the following program:

```
1.  add  r0,r1 || R[1] ← R[1] + R[0]
2.  add  r1,r3 || R[3] ← R[3] + R[1]
```

The "ideal" instruction-phase diagram is as follows, where the stage name is followed by a brief decryption of what part of the instruction is being processed:

	PHASE							
INST	–1–	–2–	–3–	–4–	–5–	–6–	–7–	–8–
1.	IF	DEC:add	EX:R1+R0	MEM:NA	WB:R1			
2.		IF	DEC:add	EX:R3+R1	MEM:NA	WB:R3		

The problem is that instruction 2 requires the value of R1 computed by instruction 1 in phase 4, but the CPU will not determine this value until phase 5. The solution is to "stall" the execution of instruction 2 in the DEC stage until a result has been stored in R1. Called a "pipeline stall" this delay is indicated in the instruction phase diagram by a "Φ":

	PHASE							
INST	–1–	–2–	–3–	–4–	–5–	–6–	–7–	–8–
1.	IF	DEC:add	EX:R1+R0	MEM:NA	WB:R1			
2.		IF	DEC:add	Φ	Φ	EX:R3+R1	MEM:NA	WB:R3

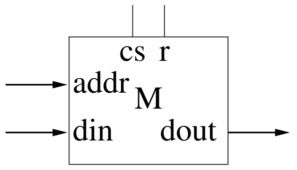
The CPU can now correctly execute the program. The average execution time is:

$$\frac{46 \text{ ns} \times 8 \text{ phases}}{2 \text{ instructions}} = 368/2 = 184 \text{ ns}$$

Because all the necessary information to detect a potential resource conflict can be determined from the values in the pipeline registers, it is possible to design the CPU, or more specifically the controller of the CPU to detect these problems and insert pipeline stalls when necessary.

33 MEMORY ORGANIZATION

External memory, or more specifically "random access external memory" is defined by the following behavioral description: As suggested by the function select table, the control



cs	r	Behavior
0	X	None. dout = Z
1	0	M[addr] <– din, dout = Z
1	1	dout <– M[addr]

input, `cs`, is a "master enable" that enables the memory to function, while the control input, —tt r, selects which function to perform: retrieve (when `r = 1`) or store)when `r = 0`.

The physical design of a memory system can employ one or more of the following technologies:

TYPE	SPEED	COST/GB
SRAM	5 ns	\$2000
DRAM	100 ns	\$20
FLASH	10^4 ns	\$4
DISK	10^7 ns	\$0.20

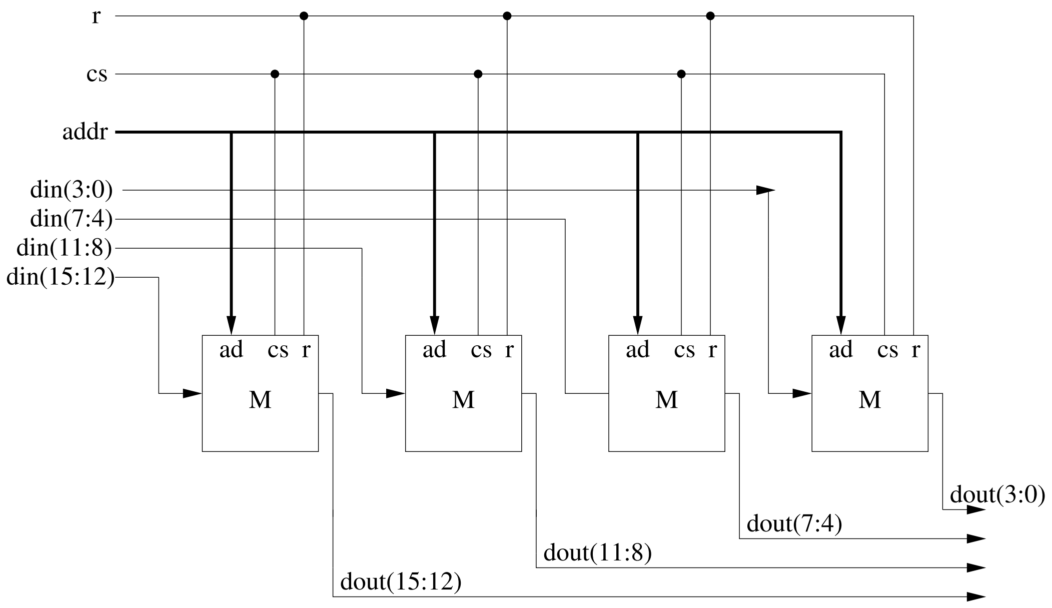
The goal of memory design is to provide large size memory with fast access in an economical architecture. For physical random access memory, DRAM, provides the best combination of economy, storage capacity, and performance. However, "virtual memory" can provide the illusion of random access while employing all of these technologies.

33.1 Expanding the Size of Memory

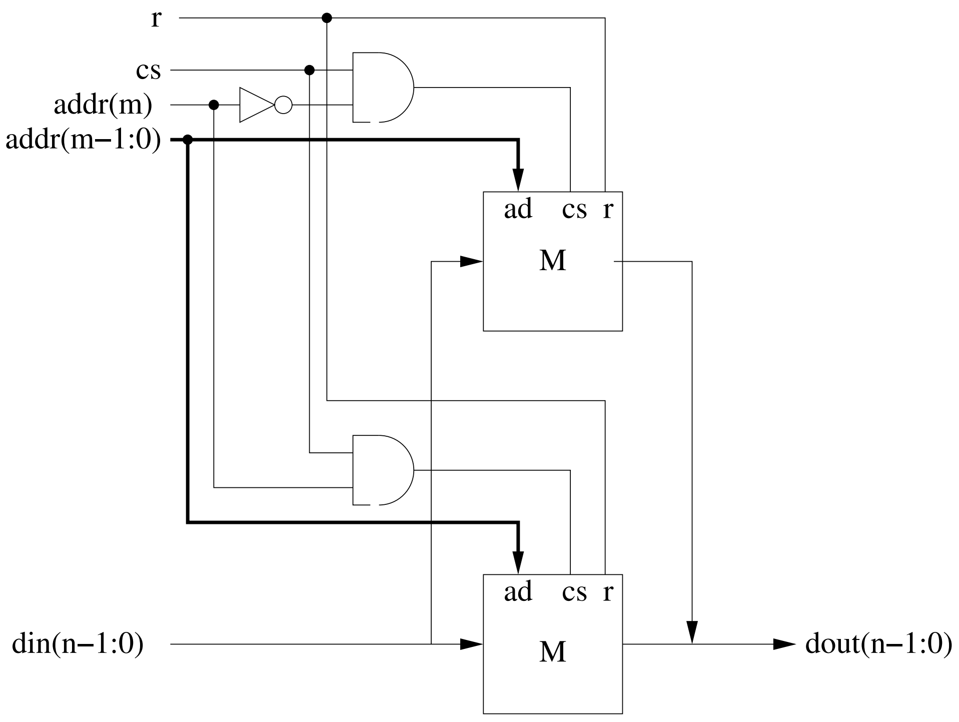
Larger memories are constructed from repeated use of a smaller "basic" memory chip. Given a $2^m \times 4$ RAM component as the basic building block, the size of memory can be increased by:

- Lengthening the word size;
- Increasing the address space; that is, the number of memory locations;
- Both of the above.

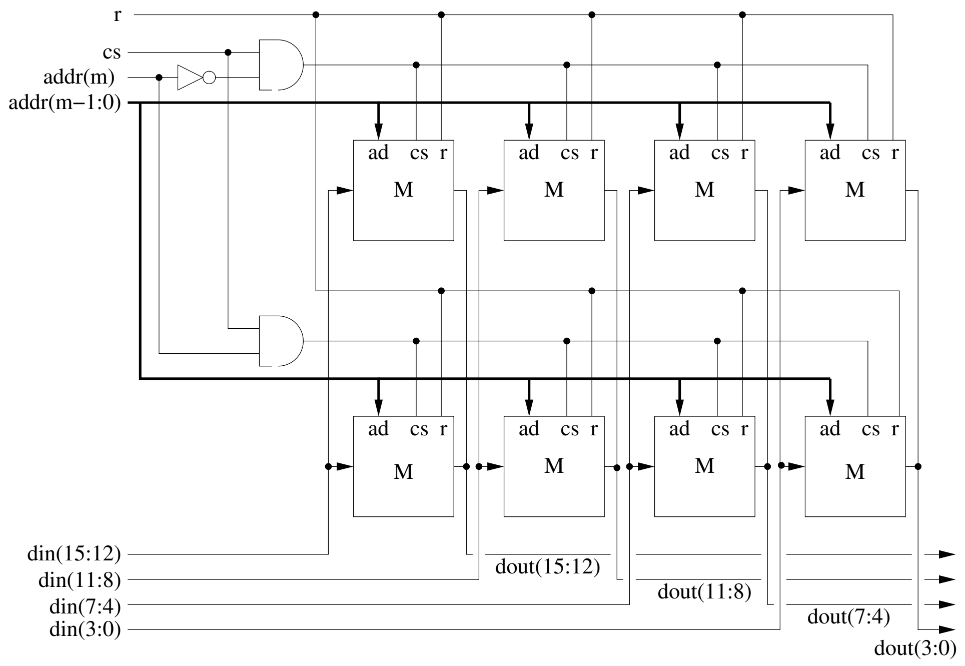
Lengthening the Word Size : A $2^m \times 16$ RAM using $2^m \times 4$ RAM's.



Expanding the Address Space : A $2^{m+1} \times 4$ RAM using $2^m \times 4$ RAM's.

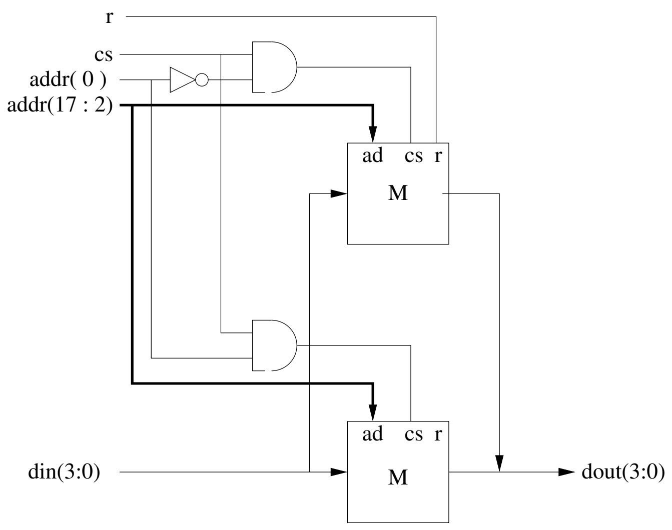


Expanding Word Size and Address Space : A $2^{m+1} \times 16$ RAM using $2^m \times 4$ RAM's.

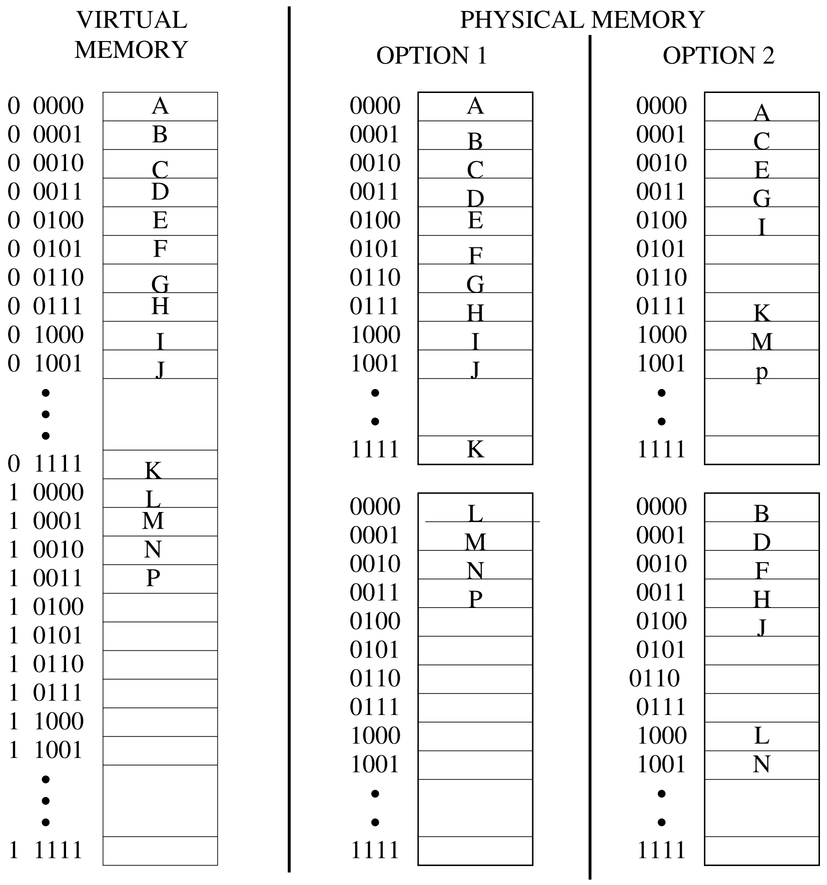


33.2 Interleaved Memory

An alternate way of accessing a $2^{m+1} \times 4$ RAM is as follows:



The effect of this is to distribute consecutive locations of virtual memory between the two DRAM's rather than storing the first half of virtual memory in the first DRAM chip and the last half of memory in the second DRAM chip. The effect is as follows:



34 PHYSICAL MEMORY ORGANIZATION

It is important to understand the distinction between "virtual memory" and its "physical implementation." "Virtual memory" refers to the programmer or CPU viewpoint of memory; that is, each address can be viewed as an index into the contents of a large one dimensional array, called the "virtual address space". Specifically:

Virtual memory viewpoint :

- Memory is a 1-d array
- Locations are accessed with a subscript called a "virtual address"

Physical Memory Viewpoint :

- The structural description of the Memory
- The means by which values are accessed:
  - The organization of DRAM memory
  - The address mapping format
  - The role and provision of supporting SRAM memory

34.1 Block Structured Memory

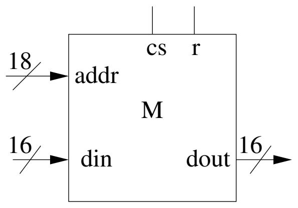
A "block structured memory" is a physical memory organized as follows:

1. Virtual memory is partitioned into "blocks" of k consecutive locations.
2. Each word of a block of virtual memory is mapped to a different DRAM.
3. The value " k " is called the "blocking factor."

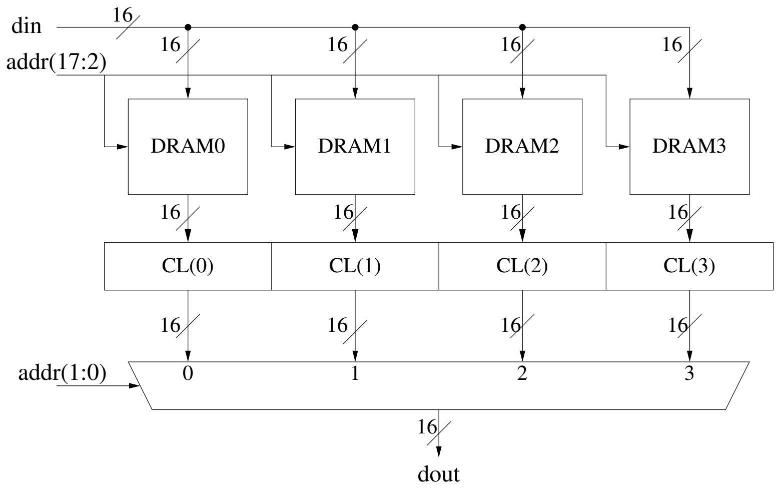
Block-structured memories are designed to address performance limitations. They are designed so that the cost of retrieval of one block is no more than the cost to retrieve one location in the block. So if t is the time to retrieve a block of size k then t/k is the average retrieval time per location.

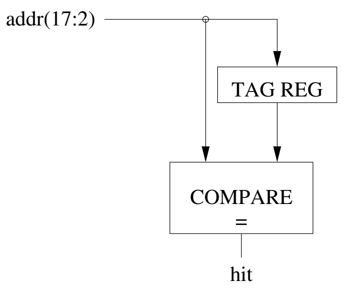
The performance "benefit" is realized when the memory access time of a memory module is slow compared to the propagation delays of combinational components and the access time of data stored in a register. For example, a set of four DRAM's can be organized in a block-structured memory as follows:

Entity Definition:



Datapath:





We view the four individual registers shown as a single large register, CL, called a "cache liner", and its purpose is to hold an entire block of DRAM.

Each word in every block of four words of memory is located at the same address but in a different one of the four DRAM chips, called the "block address". All four words of a block are retrieved at the same time by specifying a single block address. Since the propagation delay of the block register and the select logic is significantly less than the memory access time of the DRAM components, a benefit is gained when the values within the block are retrieved in succession, since all but the first retrieval can be addressed by accessing the appropriate input port of the multiplexer.

Since any block can be copied to CL, it is important to keep track of which block occupies CL by storring the block address in the register "TAG REG". When a virtual address, `addr`, is used to retrieve a value, addr(17:2) specifies the block address and therefore is compared with the block address stored in TAG REG. If they match, then a copy of the block containing the value to be retrieved is known to be in CL. This is indicated by a value of '1' on the output `hit`.

To illustrate the retrieval process, suppose we wish to retrieve the value stored at virtual address 0x000A1 = "00 0000 0000 1010 0001". The block address is given by the first 16 bits of the 18 bit virtual address; that is, "0000 0000 0010 10 00" = 0x0028. So the values retrieved from the four chips, each accessed using the same physical address X"0028" correspond to the four values that were stored in memory beginning at logical address 0x000A0; That is the values stored at locations 0x00A0, 0x00A1, 0x00A2, and 0x0013

The address bits `addr(1:0)` can now be used to select the value retrieved from any one of these four locations. Since the value of `addr(1:0)` is "01" in the example, port 1 of the MUX is selected and the value retrieved from register CL(1) is the content of virtual address 0x00A1 (that is, the contents of physical address 0x0028 of chip 1) is placed on the output bus, `dout`.

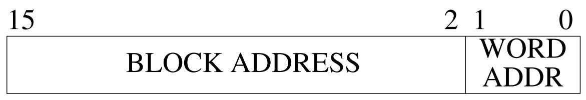
For a given value of `addr(17:2)`, by changing only the values on `addr(1)` and `addr(0)`, the other values in the block can be retrieved from the CL register without another memory access request to the DRAM chips, thus saving any further memory access time to access these locations.

The retrieval time of this architecture may be an improvement over a "DRAM only" memory architecture if the next few memory references access the block that is currently in the block register. That this is in fact often the case is due to an empirical property of most programs called "spatial locality of reference."

One of the tasks of the memory hardware is to "map" the virtual address to a physical location in the memory system. To do so, the virtual address is partitioned into fields and this partition is called an "address mapping format".

34.2 Address Mapping Format

In the block-structured memory above, the memory components map the virtual address into one that accesses the DRAM by block and then the register by field. This mapping is defined by an address mapping format as follows:



However, since a copy of the block that contains the value to be retrieved may in fact already be in the cache line, the block address can be used to confirm this by comparing it with the block address stored in the TAG REG..

As an example, consider the $2^5 \times 16$ Memory using two $2^4 \times 16$ RAM's described previously.

To caculate the average access time, observe that every access retrieves a value from the CL register, either because the required block is already in CL or because a block was copied to the CL first. The DRAM's are only accessed when `hit = 0`. If the CPU access memory n times in order to execute a program, and if $m < n$ times the value can be retrieved from CL, then the "hit ratio" given by:

$$\textbf{hit-ratio} = \frac{\text{Number of accesses made from cache only}}{\text{The total number of accesses}}$$

specifies the proportion of times that a memory access will be found in CR. So the average access time is given by:

$$\mathrm{t}_{avg} = \text{cache access time} + (1 - \text{hit ratio}) * \text{main memory access time}$$

35 LOCALITY OF REFERENCE

The goal of memory design is to create an architecture that provides a large amount of storage economically and at the same time can be accessed quickly and efficiently. The following observations are important in understanding the design strategies that will be described shortly:

1. Most programs only require a small amount of the available large memory;
2. Most programs have their instructions and data stored in contiguous locations of memory;
3. Many programs include loops. This results in a small number of instructions being retrieved many times during the fetch cycle over a small period of time.

Observations (2) and (3) capture a property that is important to the performance of hierarchical memories and is exhibited by many programs: "locality of reference". Locality of reference can be exhibited in two ways:

1. **Spatial Locality of Reference**:

   Spatial locality of reference is a measure of whether two locations A_1 and A_2 satisfy the following:

   (a) The time between accessing location A_1 and accessing location A_2 is short.

   (b) Locations A_1 and A_2 are physically near each other.

   Example: Instructions loaded in consecutive locations of memory.

   Storing a copy of a block in a cache line register takes advantage of the spatial locality of reference present in a program.

2. **Temporal Locality of Reference**:

   Temporal locality of reference is a measure of whether two locations A_1 and A_2 satisfy the following:

   (a) The time between accessing location A_1 and accessing location A_2 is short.

   (b) Locations A_1 and A_2 are not physically near each other.

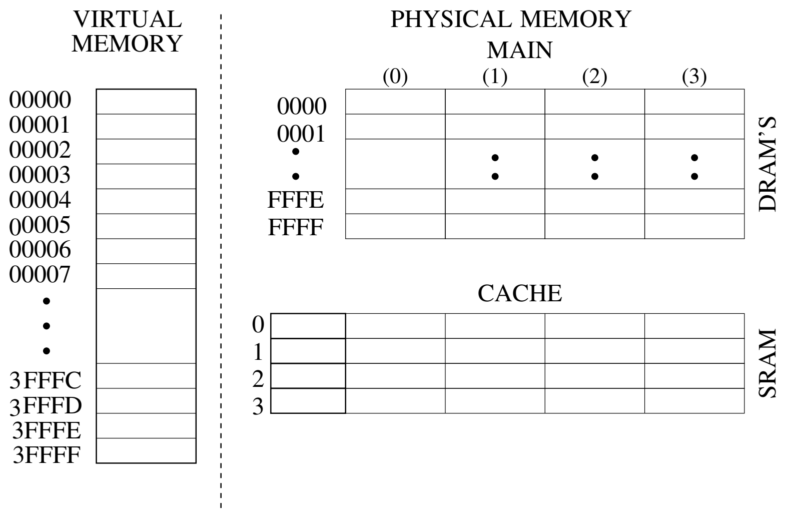
   Example: A subprogram called inside the loop of a main program.

   Using multiple cache line registers takes advantage of temporal locality of reference present in a program.

36 CACHE MEMORY ORGANIZATION

Rather than using individual registers to hold each cache line, an SRAM memory is used. Thus the he term "cache" refers to the SRAM memory components of the memory system that possess fast retrieval times. At each location of cache a copy of some block of main memory can be stored along with an identifier that uniquely determines which block of DRAM memory was copied there. The DRAM provides the primary storage for the memory system. That is, there is one storage location in the DRAM for each virtual address provided by the CPU to the memory system. However, even though a virtual address specifies the contents of but one location in the DRAM or SRAM, when a memory access occurs, the contents of an entire block of consecutive addresses is retrieved. The contents of the specified virtual address is then accessed from the retrieved block already in or copied to the SRAM.

The diagram on the next page illustrates a physical memory with an SRAM of four locations. Each location holds a block of data from DRAM memory plus the tag that uniquely identifies the block currently in the given location of cache memory.



36.1 Efficient Retrieval from Cache

Two major design issues that need to be addressed when designing cache memory systems are:

1. **Taking advantage of locality of reference**. Increasing the blocking factor improves the chance that locations will be loaded into memory that are spatially ”close” to the location actually retrieved. This is also true if the number of cache lines is increased. Increasing the number of cache lines also provides a means of accommodating temporal locality of reference, particularly when data areas and instruction areas of memory are physically far apart.
2. **Tag searching**. A means must be provided for comparing the tag from the address with that stored in SRAM. Searching is undesirable when trying to determine if a desired location to be retrieved is already in fast memory, since it adds to the overhead of the retrieval. Both addressing mapping techniques and hardware design can be used to address this issue.

37 DIRECT MAPPING

“Direct Mapping” is a strategy that can be employed to eliminate the need to search locations of the SRAM for a matching “tag.” A tag is a binary sequence used to distinguish one block of memory in the DRAM from another. In the previous examples, the block address was used as the tag. However, a shorter tag can be used in direct mapping.

1. Blocks of virtual memory are divided into s sets, numbered 0 to $s - 1$, each with an equal number of virtual memory blocks.
2. Each block in a given set is given a unique identifier called its “TAG.”
3. All blocks in a given set, i are only copied to cache line i of the SRAM. The tag for a block is copied into the corresponding TAG field of the cache line.

As indicated, nly one block at a time can occupy cache memory, and if it belongs to set i, then it can only be copied to SRAM[i]. Therefore, the tag only needs to distinguish one block from another within a given set.

To distinguish one block from another in a given set, the number of bits required to specify a tag uniquely is determined as follows:

1. Let 2^m be the number of locations of virtual memory and let 2^k be the blocking factor.
2. Then the number of virtual blocks is given by $2^m \ / \ 2^k = 2^{m-k}$.
3. Now, if SRAM provides 2^r locations of cache memory, then the blocks of virtual memory are partitioned into 2^r sets.
4. Therefore the number of blocks per set is $2^{m-k} \ / \ 2^r = 2^{m-k-r}$.

For example, consider a $2^{12} \times 8$ hierarchical memory with a blocking factor of 4 and a cache size of 4 cache lines. In this case, $m = 12$, $k = 2$, and $r = 2$. So virtual memory is partitioned into $2^2 = 4$ sets. So the number of blocks in each set is $2^{12-2} \ / \ 2^2 = 2^8$.

37.1 Address Mapping Format

When a block is copied to a cache line, its corresponding tag is also stored in the cache.

To locate a value in a cache memory, the virtual address is used to determine physical addresses for both the cache and the main memory. The virtual address is partitioned into two fields:

- The *block address* field specifies the unique block address within slow memory. All locations within a given block have the same "block id"; that is, the same value in the block field of their respective logical addresses. Thus the block id is really the address of one block in block structured memory.
- The *word address* field specifies the unique location of the given logical address within a block.

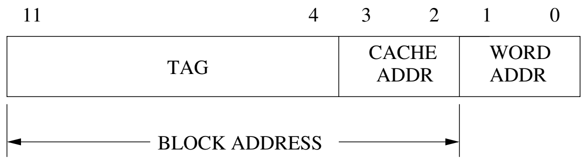
With the direct mapping strategy, the block address field is also partitioned into two fields:

- The *cache line address* (or *set*) field specifies the physical address in cache memory (i.e., the cache line) where the block that is being accessed "might" be found if it is in cache at all.
- The *tag* field specifies the tag of the block that contains the location specified by the virtuall address. This tag uniquely distinguishes it from other blocks within the same set. That is, it distinguishes a block from the other blocks that could be copied to the same cache line.

For the $2^{12} \times 8$ cache memory with a blocking factor of 4 and a cache size of 4 cache lines, using a direct mapping strategy, the size of each field within the virtual address is determined as follows:

- The blocking factor determines the size of the word field: $4 = 2^2$ implies that 2 bits are required to specify one of 4 locations within a block. That is, the word field size is 2 bits.
- The cache size determines the cache line field: Four cache lines implies that 2 bits are required to specify one of 4 cache lines. So, the size of the cache line address is 2 bits.

- The number of blocks of DRAM memory is obtained by dividing the number of locations of virtual memory by the blocking factor. In the example, this is $2^{12}/4 = 2^{10}$ blocks.
- The *set size* is $2^{10}/4 = 2^8$ blocks per set, as shown previously. Therefore 8 bits are required to specify a tag, and so the TAG size is 8 bits.

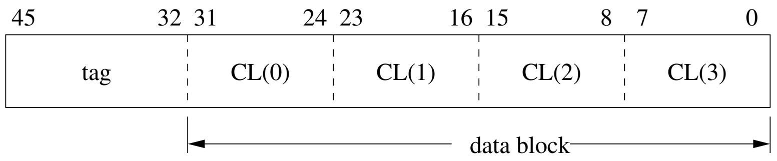


37.2 Cache Line Format

Each row of cache memory must be large enough to store the following:

- **One block of data** from main memory. In the example above, each word of a block is 8 bits and there are 4 words per block, so 32 bits are required to store the data;
- **The tag** associated with the block that uniquely identifies it among the other blocks in the set associated with the cache line. In the example above there are 2^8 blocks per set, so 8 bits will be needed to store the tag.

Thus the word format for a cache line of SRAM for the example above is:



37.3 Retrieval Algorithm

1. Set cache_line = SRAM[CACHE ADDR].
2. If TAG matches cache_line(tag) then
   . dout = cache_line(CL(WORD ADDR)).
3. If not:
   (a) SRAM[CACHE ADDR] = TAG : DRAM[BLOCK ADDRESS]
   (b) Set cache_line = SRAM[CACHE ADDR].
   (c) dout = cache_line(CL(WORD ADDR)).

NOTE "TAG" refers to the tag field of the address mapping format, while "tag" refers to the tag field of the cache line format. Refer to the diagrams above that define the formats.

The following example serves to illustrate the process of retrieval:

Given the $2^{12} \times 8$ memory with blocking factor of 4, determine the address of the cache line in a 4-line SRAM, the tag, and location of data within the cache line if a retrieval request is made for logical address `0x0AE`. Also determine what block of DRAM must be retrieved if a miss occurs:

1. Convert to binary: `0x0AE = 0000 1010 1110`.
2. Apply the address mapping format to identify the values of the TAG, CACHE ADDR, WORD ADDR, and BLOCK ADDRESS:

Field	binary	hex
TAG	0000 1010	0A
CACHE ADDR	11	3
WORD ADDR	10	2
BLOCK ADDRESS	00 0010 1011	02B

Therefore the contents of virtual address `0x0AF` may be found in SRAM[3], in `CL(2)` provided the tag field of SRAM[3] equals `0x0A`. Otherwise a block of data must be retrieved from DRAM[02B] and placed in SRAM[3] first.

NOTE: Storing a value in a cache memory is more complicated than retrieval because two memories, SRAM and DRAM, both need to be updated to preserve consistency. This topic will not be explored here, but the interested person should refer to the textbook.