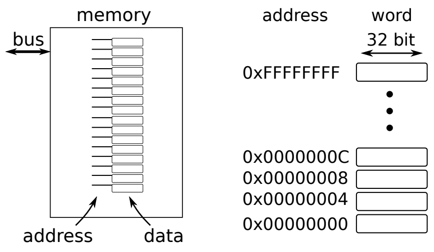
Lecture Notes on Essentials of Computer Systems - Topic 2, part 2

Table of Contents

3 Memory Hierarchy

3.1 Von Neumann model

First, we have introduced the Von Neumann model and view the memory as an one-dimensional array of memory words whereby each word is uniquely determined by an address. The figure on the right shows a more detailed view of the main memory, showing addresses (in hex) to the left of individual memory words. The memory word is the smallest number of bits with the same address. The length of the word is the number of bits it can store, say n. One word can store 2^n different values. The address is a number that uniquely defined one memory word, and its address length (in bits) m will determine the maximal memory size. For 2^m words of length n the max. size is $2^m \cdot n$.



In case of MIPS we work with $n = m = 32$, which is also why we incremented the PC by 4, to achieve alignment with 4 bytes.

The bus between the CPU and the memory carries the following: the address, the data, and control (direction: read or write). The CPU must fetch both the insns and the operands from the memory, but since the bus is shared, it can not fetch both at the same time. This limits the system performance, and the memory bus is often referred to as the Von Neumann bottleneck. We will discuss two ways to make a big and slow main memory appear like a small and fast memory:

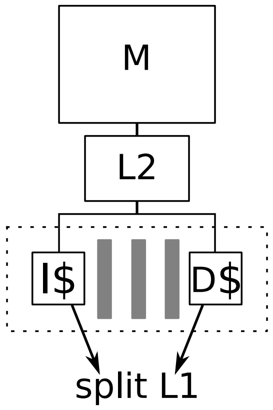
- the locality principle
- the split cache

A memory holding both operands and data is called *unified*. We have already also encountered the *split* memory when we were describing the CPU datapath, namely the insn cache (I$) and the data cache (D$). We will now explain the rationale behind this design.

3.2 The locality principle

From the view-point of the memory: in a given time-frame, certain addresses[1] are more likely then others. The sequence of addresses for a given program is namely not random. We will denote the current insn with i and the next insn with $i + 1$. Let $A(i)$ be the address of insn i, then we usually expect the next insn at the next address, namely $A(i + 1) = A(i) + 1$ (Note the abstraction: in case of MIPS $+1$ is replaced by $+4$ as the insns are 4 bytes long). Different programs will form different sequences of addresses[2], with deviations from the usual pattern caused by jumps, branches and by swapping between insn and data access. We distinguish between *temporal locality*, which is capturing the fact that recently accessed memory locations will be likely accessed in the near future, and *spatial locality*, which is reflecting that the next address will be close to the current one. Based on locality we can assume that it is enough to work with a small subset of the memory at a time. This allows us build a memory hierarchy to optimize the average access time by making a big, slow main memory look like a small, fast cache.

We show a 3 level hierarchy in the figure on the right. Top level is labeled M, this is the big and slow main memory. Below we see smaller and faster (different technology, closer to the chip or even on the same chip) L2 cache. On the bottom we see abstraction of the pipeline with the I$ and the D$: together they form the split L1 cache. They are very small and very fast, whereby the speed refers to the access[a] time. We will mention coherence later in this section. Often this entire hierarchy is considered as Von Neumann main memory.



hierarchy	L1		L2		M
coherence	L1	$\subset$	L2	$\subset$	M
size	L1	$<$	L2	$<$	M
speed	L1	$>$	L2	$>$	M

[a]for simplicity we can assume that writing and reading delays are the same, unless stated otherwise

[1]requested by the CPU
[2]which also implies different degrees of locality

As mentioned before, locality allows us to work with small subset of data, which we will casually call a block. For example, when the CPU is fetching the insn, it will access the I$ with the address stored in PC. So far, we assumed all addresses "fit" into the I$. But in reality, all addresses "fit" into the main memory, and the L2 and L1 caches merely hold subsets of data (with only selected addresses). Based on locality principle we take a small block[3] of consecutive memory words and move it into the I$. There are different mechanisms of mapping the addresses of this subset from main memory to the cache, a detail we will omit[4]. If the desired address is included in this block, then the data stored at this location is released to the pipeline - we call this a *cache hit*, and access time is very fast, say t_{L1}. If however, the address belongs to the part still in main memory, we have a *cache miss* and a *miss penalty*, that is a longer access delay, for example t_{L2}, will occur. Miss penalty is usually large, 10 to 100 clock cycles, but the miss rate[5] is small thanks to locality, for example less then 2%. In case of a cache miss, a new block of insns is transferred down the hierarchy into I$. First, L2 is checked, and if L2 cache miss occurs, the block is retrieved from M. Data movement up/down hierarchy is automatic, i.e. hardware managed by cache/memory controllers.
Access to D$ follows the same principle, however, both read and write access is possible for data (operands). As data can be modified in L1 D$, in case of a cache miss, the block currently in D$ must be first sent up the hierarchy and then new block comes down the hierarchy. Cache coherence is very important: the L1 contents must be a subset of L2 contents, which again must be a subset of M.

[3]a cache will hold more then one block, we will consider block as a unit that can be moved up/down the hierarchy and omit further details

[4]interested reader can look up the concepts of block placement, such as fully associative, set-associative and direct-mapped caches, block identification mechanisms such as tag, index, offset, block replacement strategies and writing strategies - however, these concepts will **not** be tested!

[5]probability of cache miss