Materials

W5 Techniki zwiększania wydajności procesora. Część I: pamięć podręczna

Omawiane zagadnienia

- cache memory

Searching the speed

Several performance enhancing techniques are employed routinely in processor and system designs to achieve peak execution speed in real-world computer systems. **These techniques do not alter what the processor does in terms of program execution and data processing; they just help get it done faster.**

Modern high performance processors devote a substantial portion of their logic resources to supporting effective pipelining under a wide variety of processing conditions. The combined benefits of multicore, superscalar, and superpipelined processing provide key performance enhancements in recent generations of sophisticated processor architectures.

Cache memory

A *cache memory* is a memory region that stores program instructions or data, usually instructions or data that have been accessed recently, for future use. The primary purpose of cache memory is to **increase the speed of repeatedly accessing the same memory location or nearby memory locations**. To be effective, accessing the cached data must be significantly faster than accessing the original source of the data, refferred to as the **backing store**.

When caching is in use, each attempt to access a memory location begins with a search of the cache. If the data is present, the processor retrieves and uses it immediately. This is called a **cache hit**. If the cache search is unsuccessful (a **cache miss**), the data must be retrieved from the backing store. In the process of retrieving the requested data, a copy is added to the cache for anticipated future use.

The goal of processor cache memory is to maximize the percentage of cache hits over time, thus providing the highest sustained rate of instruction execution. To achieve this objective,

the caching logic must determine which instructions and data will be placed into the cache and retained for future use.

Cache memory improves computer performance because many algorithms executed by operating systems and applications exhibit **locality of reference**. Locality of reference refers to the reuse of data that has been accessed recently (this is referred to as **temporal locality**) and to the access of data in physical proximity to data that has been accessed previously (called **spatial locality**).

As a general rule, cache memory regions are small in comparison to the backing store. **Cache memory devices are designed for maximum speed, which generally means they are more complex and costly per bit than the data storage technology used in the backing store.** As a consequence of their limited size, cache memory devices tend to fill quickly. When a cache does not have an available location to store a new entry, an older entry must be discarded. The cache controller uses a cache replacement policy to select which cache entry will be overwritten by the new entry.

In general, a processor's caching logic does not have any assurance that a cached data item will ever be used again once it has been inserted into the cache. The logic relies on the likelihood that, due to temporal and spatial locality, there is a very good chance the cached data will be accessed in the near future. In practical implementations on modern processors, cache hits typically occur on 95 to 97 percent of memory accesses.

Multilevel processor caches

In the years since the introduction of personal computers, processors have undergone dramatic increases in the speed of instruction processing. The internal clocks of modern Intel and AMD processors are close to 1,000 times faster than the 8088 processor used in the first IBM PC. **The speed of DRAM technology, in comparison, has increased at a much slower rate over time.** Given these two trends, if a modern processor were to access DRAM directly for all of its instructions and data, it would spend the vast majority of its time simply waiting for the DRAM to respond to each request.

How fast it could be?

Consider a modern processor capable of accessing a 32-bit data value from a processor register in 1 ns. Accessing the same value from DRAM might take 100 ns. Oversimplifying things somewhat, if each instruction requires a single access to a memory location, and the execution time for each instruction is dominated by the memory access time, we can expect a processing loop that accesses the data it requires from processor registers to run 100 times faster than the same algorithm accessing main memory on each instruction.

Now, assume a cache memory is added to the system with an access time of 4 ns. By taking

advantage of cache memory, the algorithm that accesses DRAM on each instruction will suffer the 100 ns performance penalty the first time a particular address is referenced, but subsequent accesses to the same and nearby addresses will occur at the cache speed of 4 ns. Although accessing the cache is four times slower than accessing registers, it is 25 times faster than accessing DRAM. This example represents the degree of execution speedup achievable through the effective use of cache memory in modern processors.

In fact, most modern high performance processors implement three (or more) levels of cache on-chip. As with the transition from L1 to L2, the transition from L2 to L3 involves a larger block of memory with slower access speed. Similar to L2, an L3 cache usually combines instructions and data in a single memory region. On a consumer-grade PC processor, an L3 cache typically consists of a few megabytes of SRAM with an access time 3-4 times slower than an L2 cache.

Various memory technologies, various memory speed

High performance processors generally employ multiple cache levels with the goal of achieving peak instruction execution rate. **Processor cache hardware is constrained in terms of size and performance by the economics of semiconductor technology.** Selecting an optimally performing mix of processor cache types and sizes while achieving a price point acceptable to end users is a key goal of processor designers.

The two types of RAM circuits in common use as main memory and for processor internal storage are:

- **dynamic RAM (DRAM)**
- and **static RAM**.

DRAM

DRAM is **inexpensive**, but has a comparatively **slow access time**, due largely to the time required to charge and discharge the bit cell capacitors during read and write operations. DRAM designs are **optimized for density**, resulting in the largest possible number of bits stored on a single DRAM integrated circuit.

SRAM

Static RAM is **much faster than DRAM**, but is **much more costly**, resulting in its use in smaller quantities in applications where performance is critical. Static RAM designs are **optimized for speed**, minimizing the time to read or write a location. **Processor cache memory is generally implemented using SRAM.**

Static RAM Static RAM (SRAM) boasts a substantially faster access time than DRAM, albeit at the expense of significantly more complex circuitry. SRAM bit cells take up much more space on the integrated circuit die than the cells of a DRAM device capable of storing an equivalent quantity of data. A single DRAM bit cell consists of just **one MOSFET** transistor and **one capacitor**:

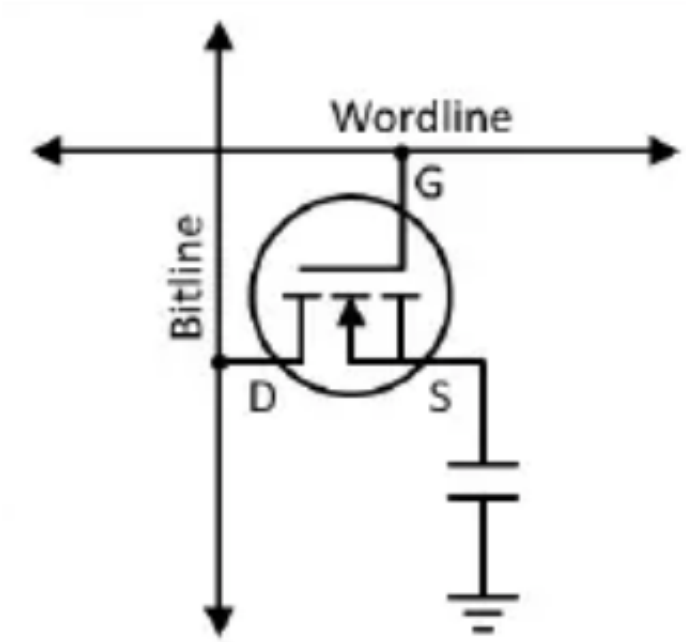

Figure: DRAM bit cell circuit

The standard circuit for a single bit of SRAM contains **six MOSFET** transistors. Four of these transistors are used to form two NOT gates. The output of each of the NOT gates is connected to the input of the other, forming a flip-flop:

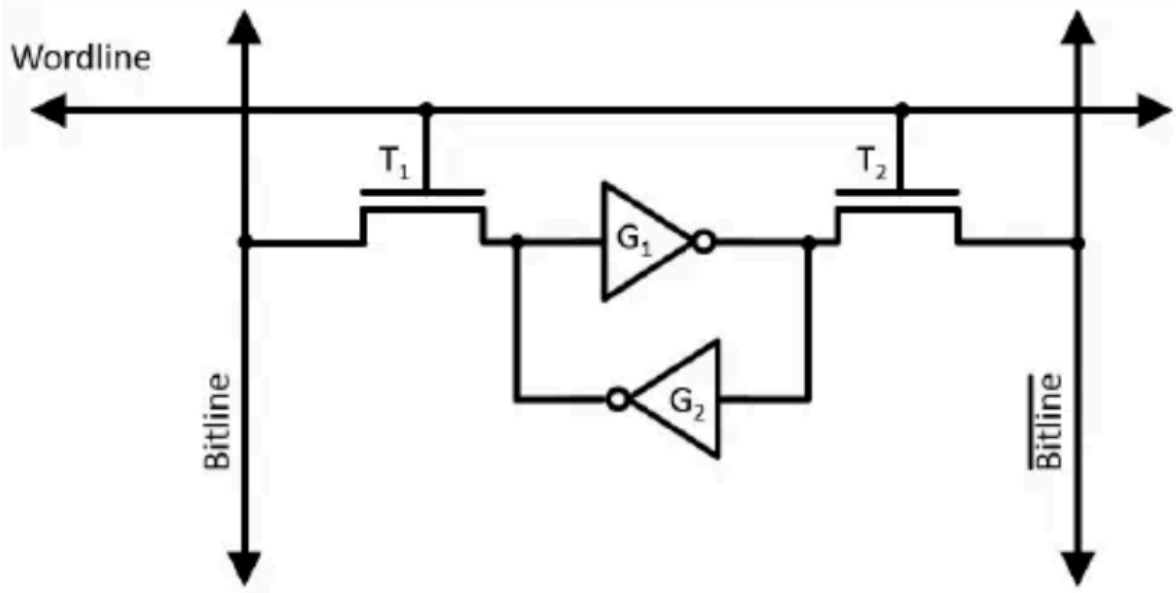

Figure: SRAM bit cell circuit

Note that, unlike DRAM, SRAM does not require periodic refreshes to retain its data content. This is why it is referred to as static RAM.

How to find what you are looking for? Direct mapped cache case

How it works

A direct mapped cache is a block of memory organized as a one-dimensional array of cache sets, where **each address in the main memory maps to a single set in the cache**. Each cache set consists of the following items:

- A **cache line**, containing a **block of data** read from main memory.
- A **tag value**, indicating the location in main memory corresponding to the cached data.
- A **valid bit**, indicating whether the cache set contains data.

There are times when the cache contains no data, such as immediately following processor power-on. The valid bit for each cache set is initially clear to indicate the absence of data in the set. When the valid bit is clear, use of the set for lookups is inhibited. Once data has been loaded into a cache set, the hardware sets the valid bit.

To explain how it works I will use two examples. The first one is with decimal numbers as we used to use these numbers every day which simplifies considerations. Second example is analogous but in binary system.

Very simple decimal example

In every decimal address in range from 0000 to 9999 you can distinguish three parts: tag, set and offset:

```
tag (specifies a range from data is coming)
| set (block of data number)
| |offset within a block
| ||
...
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
...

There are 10 different sets numbered from 0 to 9.
In every set there are 10 different offsets from 0 to 9.
Each set can be populated with data coming from the following ranges

Number  Range  From  To
  1     00xx   0000  0099
  2     01xx   0100  0199
        ...
100     99xx   9900  9999
```

Data at address `1123` are stored in set number `2` at offset `3`. This data is stored in the set numer `2` along with all other data from range `1120` to `1129`.

However there are other addresses resulting with exactly the same set reference. For example data at address `3723` should also be stored in set number `2` and at offset `3`.

To differentiate them and to avoid disambiguity you have to provide a "contex" (in the form of tag) which is in this case the range your block od data is comming. In the first case context is equal to `11` while in the second `37`.

```
Set   Tag    Offset  Data
0-9   00-99  0-9
-----+-----+-------+------
2    | 11  | 0     |
     |     +-------+------
     |     | 1     |
     |     +-------+------
     |     | 2     |
     |     +-------+------
     |     | 3     | data    <-- data at address 1123
     |     +-------+------
     |     | 4     |
     |     +-------+------
     |     | 5     |
     |     +-------+------
     |     | 6     |
     |     +-------+------
     |     | 7     |
     |     +-------+------
     |     | 8     |
     |     +-------+------
     |     | 9     |
-----+-----+-------+------
```

Figure: Cache organization (decimal case)

Remember that even if you read one datum, for example at address 1123, the whole block in cache is populated with all data lying in the same range and belonging to the same set. In case of address 1123 all data from 1120 to 1129 are loaded to a cache.

Other words, even if you want to load a single datum to processor's register a whole block containing the addressed location is loaded into cache.

More real example

As a more real example consider a small L1 cache size of 512 bytes. Because often this is a read-only instruction cache, it need not support the ability to write to memory. The **cache line** size is 64 bytes. Dividing 512 bytes by 64 bytes per set results in 8 **cache sets**. The 64 bytes in each set equals

$$2^6$$

bytes, which means the least significant 6 bits in the address select a **location within the cache line**. Tree additional bits of an address are required to select one of the eight sets in the cache (because

$$2^3 = 8$$

).

Figure: Cache organization

From this information, the following diagram shows the division of a 32-bit physical memory address into **tag**, **set number**, and **cache line byte offset** components:

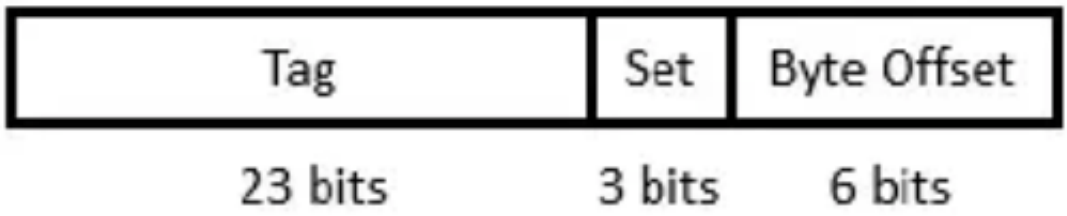

Figure: Components of a 32-bit physical memory address

Each time the processor reads an instruction from DRAM (which is necessary any time the instruction is not already present in the cache), the MMU reads the 64-byte block containing the addressed location, and stores it in the L1 cache set selected by the three set bits (the part of physical address). The upper 23 bits of the address are stored in the tag field of the cache set, and the valid bit is set, if it was not already set. As the processor fetches each subsequent instruction, the control unit uses the three set bits in the instruction's address to select a cache set for comparison. The hardware compares the upper 23 bits of the executing instruction's address with the tag value stored in the selected cache set. If the values match, a cache hit has occurred, and the processor reads the instruction from the cache line. If a cache miss occurs, the MMU reads the line from DRAM into cache and provides the instruction to the control unit for execution. This process is depicted on the set of figures below:

Figure: L1 cache in action

How it speed-up

To demonstrate why a direct-mapped cache can produce a high hit rate, assume you are running a program containing a loop starting at physical memory address 8000h and containing 256 bytes of code. The following actions will take place:

1. The loop executes instructions sequentially from the beginning to the end of the 256 bytes, and then branches back to the top of the loop for the next iteration.

2. Address 8000h contains 000b in its set field, so this address maps to the first cache set.
3. On the first pass through the loop, the MMU retrieves the 64-byte set 000b cache line from DRAM and stores it in the first set of the cache.
4. As the remaining instructions stored in the same 64-byte block execute, each will be retrieved directly from cache.
5. As execution flows into the second 64 bytes, another read from DRAM is required.
6. By the time the end of the loop is reached, sets 000b through 011b have been populated with the loop's 256 bytes of code.
7. For the remaining passes through the loop, assuming the thread runs without interruption, the processor will achieve a 100 percent cache hit rate, and will achieve maximum instruction execution speed.

Not always so good, but it still pays off

Alternatively, if the instructions in the loop happen to consume significantly more memory, the advantage of caching will be reduced. Assume the loop's instructions occupy 1024 bytes, twice the cache size. The loop performs the same sequential execution flow from top to bottom. When the instruction addresses reach the midpoint of the loop, the cache has been completely filled with the first 512 bytes of instructions. At the beginning of the next cache line beyond the midpoint, the address will be 8000h plus 512, which is 8200h. 8200h has the same set bits as 8000h, which causes the cache line for address 8000h to be overwritten by the cache line for address 8200h. Each subsequent cache line will be overwritten as the second half of the loop's code executes.

Even though all of the cached memory regions are overwritten on each pass through the loop, the caching structure continues to provide a substantial benefit because, once read from DRAM, each 64-byte line remains in the cache and is available for use as its instructions are executed. The downside in this example is the increased frequency of cache misses. This represents a substantial penalty because, as you have seen, accessing an instruction from DRAM may be 25 (or more) times slower than accessing the same instruction from the L1 cache.

Note: what to cache

The example in this section assumes that cache memory uses physical memory addresses to tag cache entries. This implies that the addresses used in cache searches are the output of the virtual-to-physical address translation process in systems using paged virtual memory. It is up to the processor designer to select whether to use virtual or physical addresses for

caching purposes.

Bibliography

1. William Stallings, *Computer Organization and Architecture*, Pearson, 11th edition (2019)
2. Jim Ledin, *Modern Computer Architecture and Organization*, Packt, April 2020