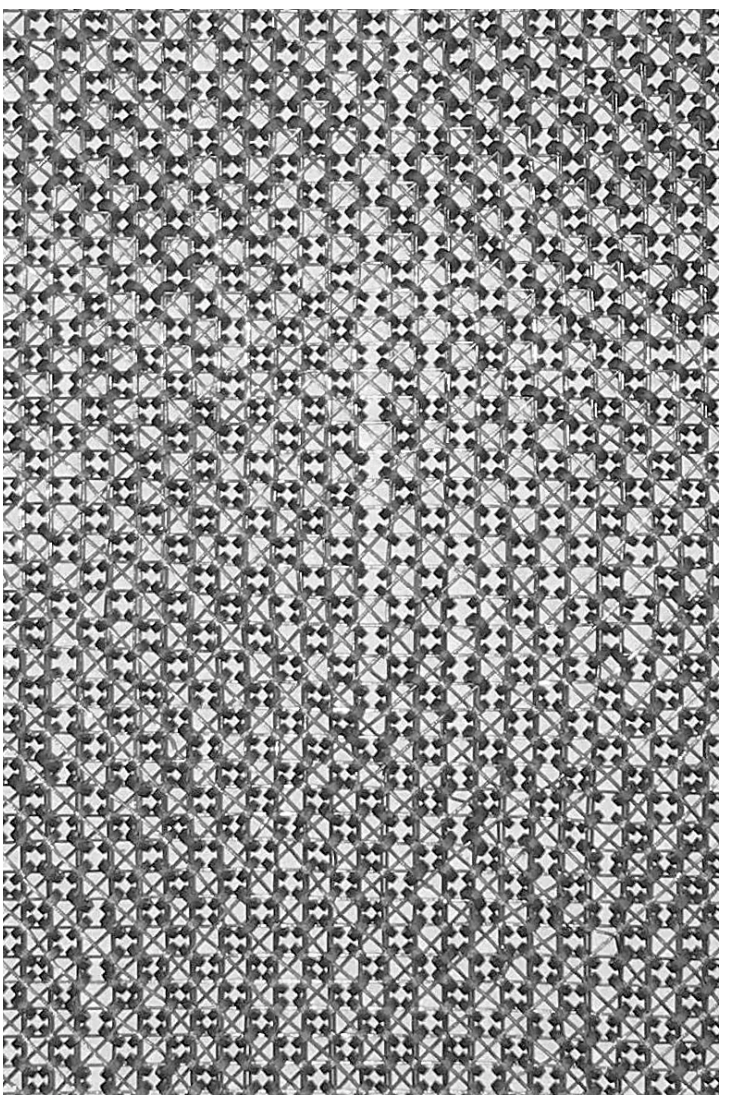
Making Memories 5



Close-up image of Magnetic core-store Memory array (1960s). Photo: Chris Crispin-Bailey.

5.1 Memories, past and present

Computer memory systems have evolved very significantly in the past 60 years. In some ways, they have changed beyond recognition, but in other aspects, for example, the way memory is organised and used, they still owe much to the earliest days of computer system design and construction.

In the simplest definition, a computer memory is a component that **retains information** (data) for a period of time. This should not be confused with the concept of a storage device, which we will examine later. Storage devices are secondary data storage mediums, often (but not always) capable of being removed from the computer system, and are very typically **non-volatile**. By non-volatile, we mean that data is retained even when the system is disconnected from the computer and has no power source attached. A computer memory, on the other hand, is typically a permanent part of the computer system, and can be either volatile or non-volatile.

Early computer systems used some fairly exotic methods to retain data in the form of a data memory. In the days when computer systems were still being built from radio valves, it was of course possible to build logic gates and perhaps also to build storage elements by using radio valves in a similar way to transistors. However, the size of a single bit of storage would then be quite large, a whole byte even larger, and a 'reasonable' amount of computer memory, let us say a puny 256 bytes, would fill shelves full of such electronics. Clearly, if memories were to get any larger, some new technologies were going to be required.

One system, the **mercury delay line**, used sound waves travelling through a tube of mercury to represent binary zeros and ones as pulses of acoustic energy. By capturing the sound waves at the other end and then feeding them back into the start, the pulses could travel along the delay line indefinitely, at least in theory. This meant that data bits were held within the system as long as power was maintained, and nothing interfered with normal operation. One delay line could hold of the order of 500 bits of data. Though more reliable than radio valve logic, the downside was the need for 20kg of mercury for each delay line. Be glad that computers do not still use this technology... a modern laptop with even a small 4Gigabyte memory would weigh over 1.3 million metric tonnes!

Another system, known as **magnetic core-store**, used very small magnetic rings threaded onto a **row and column matrix** of wires, to form a

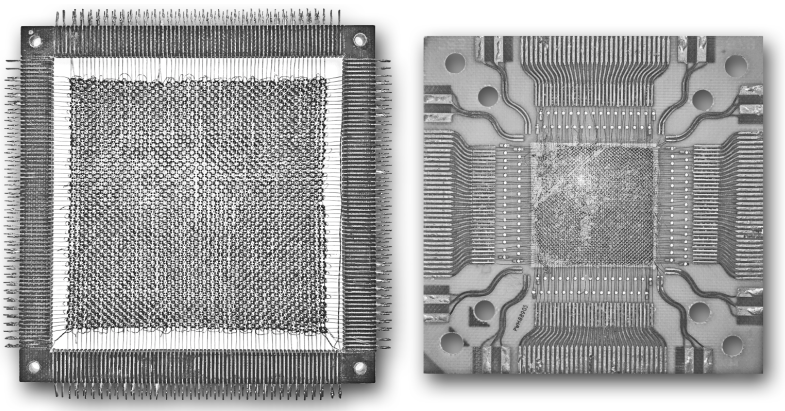


Figure 5.1: core-store Memory Modules. Showing a 1kbit module (left) and a later, more compact, double-sided card module (right). The term 'core-dump' is still used to refer to dumping memory content to a diagnostic-file when an error occurs.

grid of binary storage elements. Indeed, we still refer to memory organisation as rows and columns internally. An example of core-store is given in Figure 5.1. Core-store was relatively compact. A 1 kilobit module might be 10 x 10cm across and 1cm deep, and weigh 40 or 50 grams.

This memory system was still terribly slow by modern standards. Even so, several variations of core-store memory were used to build the Apollo guidance computers[52], and managed to get men to the moon and back. Without core-store it simply would not have been possible. One small step for man, one giant leap for computer memory!

[52] The Apollo-11 Guidance computer (AGC) used both erasable core-store arrays, and a permanent memory version of core-store known as **rope memory**, in which the guidance programs were stored.

Needless to say, computer systems engineers were not satisfied with the memory technologies they had to make use of, and it was not long before all-electronic memories were starting to be developed.

As we learned briefly in Chapter 2, transistors can be combined into logic gates, and logic gates can be combined into simple data latches with the capability of storing a single binary bit. However, as 6 transistors are needed for one storage cell (often referred to as a **6T bit-cell**), this was a bulky form of memory. One bit could easily occupy a few square centimetres of circuit board space. Later, the advent of **integrated circuits** allowed many transistors to be squeezed onto a single chip of similar size, and before long, computer memory chips with many kilobytes of data were beginning to become available.

If we fast-forward to modern day computer systems, we find that individual chips, of the same size as those early memory chips, have capacities as high as 10's of giga-bits. And contain billions of transistors. When a few of these chips are combined on a small circuit board, we get a

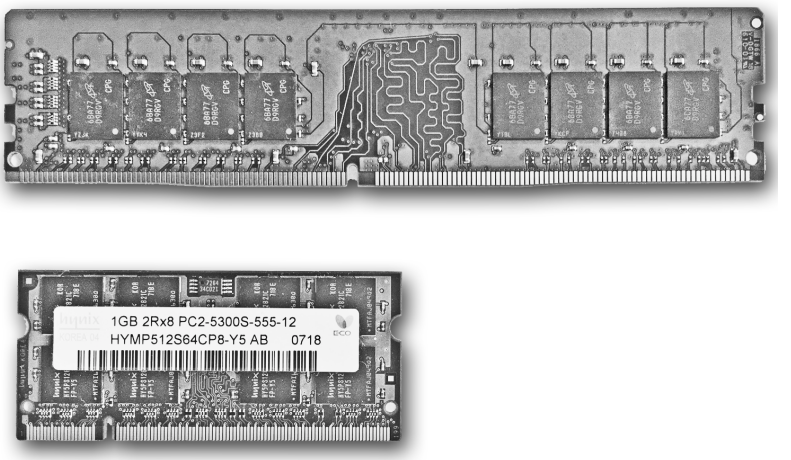


Figure 5.2: Examples of DIMM modules: Dual Inline Memory Module. **DDR DRAM** (top), and **SDRAM** (bottom).

module known as a **SIMM** or **DIMM**. Some DIMMs are shown in Figure 5.2. These modules typically come in capacities of 8, 16, 32 or even 64 Gigabytes. Their cost is very low considering how much data capacity they have.

5.1.1 Getting physical

[53] Small? Bill Gates, cofounder of MICROSOFT , and inventor of MS-DOS, was once claimed to have said 640 Kilobytes would be more than enough for most users. Never make predictions about computer technology!

It is noticeable that a computer system that perhaps has only 4 Gigabytes of memory can still run a large number of programs or applications simultaneously, and each of those programs potentially seems to have a lot of data in play: perhaps much more than the memory capacity. So how come we can fit so much into a relatively small memory?[53] The answer is that the hardware of the system, the actual memory chips, are only part of the equation.

The electronic memory chips that make up the system memory are sometimes referred to as **physical memory**. This is to reflect the fact that for every byte of physical memory, there is some hardware, some transistor, some chip, where that data is physically held whilst the computer operates. These are known as the physical memory. It also helps to differentiate that memory from a concept known as **virtual or logical memory**. We will say more about this later, as this is actually more of an operating systems concept, and not so much to do with the hardware of the system. For now we can just say that virtual memory is an illusion of having a much bigger physical memory, when in reality most of that data is stored on a storage device such as a disk unit and moved into

the real physical memory only when it is needed. The result is that a system with 4 Gigabyte of physical memory may appear just like one with 32 Gigabyte of physical memory. The only noticeable difference is that things may run much slower. We will come back to this point later.

5.2 Non-volatile memories

In actual fact, there is not one kind of computer memory in current use, but a wide variety of them. We can divide these into categories that help us to understand what their properties and uses are. One major category is **non-volatile memory**, and there are many sub-types of this technology:

Non-Programmable (Read-Only) Memory:

- Permanently manufactured with a particular data content.
- Manufactured as an Integrated circuit (IC) with fixed content.
- Expensive to design, and to update (needs redesign).
- Cheap when mass produced.
- Sometimes referred to as Read-Only Memory (ROM).
- Data content is never lost.

One-Time Programmable Memory

- Manufactured with an array preset to a blank state.
- Can then be programmed to a particular data content.
- Once data is programmed it can never be erased.
- Cheaper to manufacture.
- Changes to content just require changes to the data being 'burned' into the memory.
- Data content is never lost.
- Often known as PROM (Programmable Read Only Memory).

Programmable Non-Volatile Memory.

- Can be erased and reprogrammed many times.
- Data is retained even when no power is present.
- Erasure and reprogramming can be done in several ways:
  - **EPROM:** Erasable Programmable ROM, is erased by UV light, and programmed by electrical signals.
  - **EEROM:** Electrically Erasable ROM, erased by an electrical signal, programmed by electrical signal.

- **EAROM**: Electrically Alterable ROM, is reprogrammable by electrical signals.
- **Flash Memory**: a variation of EEROM, has large capacity, cheap, relatively fast to read and write data.

These represent the main types of non-volatile electronic memory in common use currently. The obvious question is, why are there so many kinds of memory? There are many reasons, but a key one is the requirements of the application for which the memory is to be used.

A true **ROM** (i.e. one that is manufactured as an IC with a fixed content) is used where a permanent data content is required, and this is to be mass produced in very large numbers. ROM is used where content must never be altered. For example, a program in an implanted medical device is safety critical and must never be able to be corrupted or deliberately hacked. ROM is a safe solution here.

However, it is very expensive to manufacture a ROM from start to finish. The chips may only cost $5 but the design cost could be $250,000. To avoid such large up-front costs, we often use PROM instead.

A **PROM** is very similar to a ROM in use, but comes from the factory **un-programmed** (or, in other words, blank). The customer can then buy thousands of chips and only program a few hundred chips at a time, thus allowing for revisions of the data content at a later date. Once the data is **programmed** into the chip it is as permanent as ROM and can never again be modified.

For a range of less safety critical applications, the desire to have memory chips that act as if they are permanent, but can be altered under tightly controlled conditions, has led to the development of **EPROM**. An EPROM is like a ROM that is supplied blank, can be programmed, and then acts very much the same. However, at some later date, the chip can be placed in a **UV light-box** and **erased**, ready to be reprogrammed. This allows for modifications of devices 'in the field', provided an engineer can be physically there. It is not possible for EPROM to be erased by a remote hacker simply by sending a virus or a rogue command, so it is considered relatively secure.

Other types of non-volatile memory have different categories of use. All of the remaining types can be erased by some form of **electrical signal**, and therefore are less secure in some respects, if the system they are built into allows control of those signals to be accessed. This is often the case, since the purpose of these type of memories is to allow **semi-permanent** data storage but also to permit firmware updates to be possible.

For instance, a product might be configured to have certain factory settings, and during its use after being deployed to a customer, those settings might be updated. So-called 'Smart-TV's' are a good example. They contain non-volatile memories, where their software (effectively firmware) is stored. Data broadcast over the TV radio spectrum or via internet connection is received by smart TVs and used to overwrite this firmware as and when updates are released by the manufacturer. Of course this can be done entirely remotely, and therefore presents a theoretical risk of a hacker doing this in an unauthorised fashion.

Most **electrically erasable** non-volatile memories are relatively slow to read and write. However, one particular type of memory, known as **flash-memory**, is able to perform at reasonably high speeds (though nowhere near as fast as volatile memories). These memory chips are used in **USB flash drives**, also known as **memory sticks**, because they are relatively fast and have high capacities.

However, we have strayed into the domain of storage devices! Yes, it is true that a flash memory stick is a memory, but it is also a type of storage device (it is removable, non-volatile). As we shall see later, the way storage devices are built is beginning to overlap with how memory operates, but they should still be viewed as two distinct and different things in practical use.

[54] This is not entirely true. EPROMS can lose their programmed data over very long time periods, of the order of decades. Some other types of memory can become corrupted over very long time scales. Often we see fit to ignore this problem; after all computers are often obsolete so quickly that this problem may never be observed. However, it is one that cannot be forgotten entirely.

5.3 Volatile memories

Just as non-volatile memories can retain data indefinitely, even without power[54], the **volatile memory** can only retain data when powered up and operated correctly. There are currently two major types of volatile memory: the **Static Randomly Access Memory** (**SRAM**) and the **Dynamic Random Access Memory** (**DRAM**)[55]. Both of these memory technologies are found in most general purpose computers, and the reasons for having both will become clearer as we discover more about them.

Certainly **RAM**, **ROM**, **PROM**, **EPROM**, all appear to be very similar, if not identical, in terms of how data can be read from them. Other non-volatile memories may have certain read and write constraints, which we will not dig into at this point.

[55] Incidentally, the idea of RAM being **randomly accessible** is a bit of a 'red herring'. Almost all modern memory devices allow data to be accessed in a random fashion. In early systems such as mercury delay lines, data had to be accessed in a sequential pattern. The term 'randomly addressable' came about to highlight the new non-sequential capabilities of newer and (then) more advanced memories.

5.3.1 Static Random Access Memory (SRAM)

The SRAM, as we mentioned earlier, is based upon an arrangement of transistors, the 6T bit-cell. There are other variations which have specialised properties, but the standard SRAM storage element is based upon this idea.

As this is not an electronics or a digital engineering course specifically, we will not delve into precisely how six transistors can store one bit of data, but this can easily be researched by interested readers.

One of the main principles of static RAM, is that it operates using purely digital circuitry, and the 6T bit-cell will remain in a zero or one state for as long as power is provided. However, even a momentary loss of power is enough to lose some or all of the memory content, and leave it **corrupted**[56] . For this reason, most robust computer systems have power supplies that maintain a smooth delivery of power, even if momentary glitches might occur on the supply to the building.

[56] Corruption creates random ones and zeros. This is much worse than simply zeroing the memory as it could cause unlimited variations of unpredictable behaviours.

SRAM is typically very fast, but because of the circuit design, it occupies a lot of chip space per bit, at least when compared to DRAM. Another way to express this is to say that SRAM has lower **storage density** than DRAM. This means it is more expensive per bit for a given piece of silicon containing an array of bit-cells, and one megabyte of SRAM may be many times more expensive than one megabyte of DRAM. Therefore, if you wish to have a huge memory capacity, then DRAM appears to be attractive in terms of cost and space occupied on the system motherboard.

5.3.2 Dynamic Random Access Memory (DRAM)

Because the 6T SRAM cell occupies a relatively large amount of silicon space, another competing technology has also become widely used. DRAM uses a different way of retaining a binary one or zero. By using a small **chargeable component** known as a **capacitor**[57], an electrical charge corresponding to a binary '1' can be represented when the capacitor is charged, and when discharged this represents a zero. Such **capacitive bit-cells** are smaller and therefore more bit-cells can be packed into a DRAM chip than an SRAM chip, leading to higher storage density and higher overall capacity. Of course there must be a downside, otherwise SRAM would have become obsolete long ago. DRAM is relatively slow compared to SRAM, and has certain additional circuit operating requirements, known as a **refresh cycle**[58]. These factors add to the

[57] A capacitor is almost like a mini-battery, but it loses its charge quickly.

[58] This is due to the capacitive bit-cells they use, which lose their charges over a period of milliseconds. Thus they must be topped up (refreshed) frequently in order to retain their data values.

complications of engineering a system using DRAM. But because it is very inexpensive, it is widely used for what is known as **bulk memory**; in other words, the main provider of physical memory in the computer system.

DRAM's come in a number of formats, the most popular of which are small circuit boards known as **SIMM** and **DIMM modules**. These can be slotted into a socket on a motherboard of a computer system, and just as easily removed, making the system easily configurable and upgradeable.

However, because DRAM **access time** (the time taken to read a data value) is slower, this presents a problem for designers. Memory speed is typically slower than CPU speed in terms of technology progression over time. DRAM access speeds are often anywhere between five and twenty times slower than contemporary processor clock frequencies, and SRAM can be one to three times slower. A one-to-one speed rating is good because it means the CPU never has to wait more than one clock cycle for SRAM to respond, thereby maximising performance.

This potential slow relative speed of memory means that a CPU could spend a lot of time simply waiting for DRAM to respond to a request for data instead of working on it, and as a result the CPU will not deliver its peak performance capability. To get around this problem there are multiple solutions, with various cost implications. We will explore some of these in the next section.

5.4 Memory performance strategies

As we have just highlighted, memory speed, which so far we have defined only in terms of access time, is a critical factor for computer performance, and unfortunately for us, large and inexpensive memories are relatively slow. Consider a CPU operating at 3.9 GHz, and a DRAM operating at a relatively fast 1330MHz. There is a three-to-one ratio between CPU speed and memory speed if we optimistically assume that one clock cycle yields one memory access. This already seems to be a significant problem, but there are more factors to consider to get the whole picture. We probably need to learn a little more about memory before we can see the whole picture from an informed viewpoint.

5.4.1 Access time and cycle time

Both SRAM and DRAM (indeed any IC-based memory device) have several potentially different timing properties: **Access time** is the time it takes to read a data value onto the system bus after completing the process of requesting that data from the memory. Ideally this will require fewer nanoseconds than the length of one CPU clock cycle period, such that the CPU can request data during one clock cycle and receive it before the start of the next clock cycle (when it wants to use it). Any delay longer than that results in whole extra clock cycles having to be spent waiting for memory to catch up[59] . These are known as **wait-states** or **wait-cycles**. If we have a **zero-wait-state memory** system, then we are in good stead, because the CPU gets the data within one clock cycle.

[59] We can have 1, 2, 3 clock cycles, but we cannot have fractions of clocks. If access time is even a picosecond too slow, then an entire extra clock cycle will be needed.

Cycle time is another critical property of memory. Whereas it may take a particular amount of time to access data, the memory cycle time represents the time required for that memory chip to set up a request, access the data, and then also recover internally from its effort and be ready for the next access. The cycle time will therefore typically be much longer than the access time, and even more so after a write operation.

5.4.2 Memory access protocol

Taking a read operation as an example, in order for a memory device to provide a data value, it must first be told what address to read. This process requires the CPU to place an address on address lines of the system bus, and to assert the correct timing signals to tell the memory device to perform a read operation. This required set of actions is called the **memory access protocol**. We should be able to determine from this that a standard read operation must therefore require at least two clock cycles, but there are reasons why this is likely to be more, as we will see shortly.

Consider the following scenario:

So, let us suppose the CPU requests data from DRAM, and gets a response in three clock cycles. But, on further investigation, it is found that the DRAM has a full cycle time of 5 clock cycles. Now we can see that the CPU cannot request another data item until a further two clock cycles have elapsed after data access. This means that the CPU can only read a new data item once in every 5 clock cycles in this case.

The lesson here is that we cannot always rely upon access time to determine system performance. It is cycle time that ultimately limits the use of memory bandwidth.

Suppose that a CPU has a clock frequency of 1000MHz, and it encounters the memory system we have just described, with a memory cycle time of 5 clock cycles, this means it can only access 200 million data items in memory per second. For peak performance we might want this to be 1000 million data access per second. Fortunately some techniques exist to try to augment this disparity.

[60] Unless specifically stated we will use decimal megabytes here, and for bandwidth and data rate calculations generally, as we are taking about data transfer on a clocked system. Memory capacity, on the other hand, is almost always specified in binary megabytes.

5.4.3 Memory width

One thing we have not yet considered is **memory bus width**. If a memory location contains 8 bits, in other words, 1 byte, then 200 million memory accesses per second means 200 Million bytes of data can be accessed per second (or 190 Binary Megabytes/sec). We refer to this measure as **memory bandwidth**[60].

It might be obvious therefore that to increase memory bandwidth, we could simply make the memory bus wider. Why not have a 16-bit or 32-bit wide memory instead of 8 bits?[61] A 32-bit memory with a cycle time of 5 clocks would deliver 800 Million bytes/sec, instead of 200 Million bytes/sec.

[61] Just as adopting a four-lane highway rather than a one-lane road will potentially increase traffic flow.

In recent years we have seen computer systems moving toward 64bit system architectures, partly for the reason that it delivers wider memory busses and increased memory bandwidth. In-fact, all devices connected to the system bus will enjoy improved bandwidth, and we sometimes refer to the total bandwidth available as the **system bandwidth**. However, there are limits: simply increasing memory width to 256 bits, for example, won't necessarily give the gains that might be expected[62]. Such schemes are also likely to complicate the circuit board design of the system considerably, demand more expensive system components, and so on, and so it is not an unlimited panacea.

[62] One of the reasons is that, as data width gets larger, the probability that all of the data bits will be useful for each and every read starts to diminish, and the system may then simply end up frequently reading unwanted data.

5.4.4 Single-cycle versus Burst-mode access

In the single-read DRAM **access policy**, as observed in Figure 5.3, the procedure required to perform a memory access begins when the CPU sends an address as a **row address** and a **column address**, taking one clock cycle for each transfer to the DRAM. The DRAM then spends

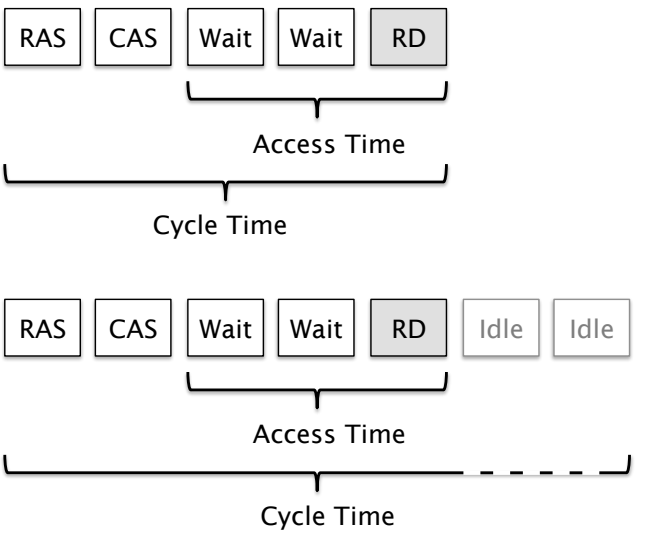


Figure 5.3: DRAM single read operation. Showing actions performed in successive clock cycles, assuming an access time of 3 clock cycles. The top case represents the simple scenario, whilst the second case shows the possibility of idle cycles being added at the end of a memory transaction.

time looking up its data (internal readout), and then provides the result. There is sometimes also a period of extra 'idle' time whilst the DRAM completes its additional cycle time requirements[63]. Only then can the CPU start a new request.

[63] There are various reasons why this may be the case. The CPU may simply not be ready to start a new read, there may be additional circuit effects to consider, and the memory itself may have constraints. Manufacturer datasheets, for example, often have several complex design parameters to consider when building a real system. In particular, the requirement to issue an 'RAS precharge' command, when access switches to another row address, can create delays of multiple cycles before the DRAM is able to properly start a new memory access.

However, given the structure of programs and the way data is organised in memory, we typically find that a CPU may be accessing multiple memory locations in a linear sequence. Therefore, sending a new and updated address for every one of these successive accesses, and then also waiting for the DRAM to complete its cycle time, is very wasteful.

Burst mode is a very useful innovation in DRAM technology that boosts memory performance, in part overcoming the penalties of cycle time, and certain other factors that limit performance.

In burst mode, as shown in Figure 5.4, a DRAM can be instructed to read a number of **consecutive** memory contents out onto the bus one after another. It is then only necessary to provide the first row/col address, since the other data items follow directly afterward in a linear sequence, and because the DRAM is still actively reading the same block of memory during this burst sequence, any additional end of cycle delays are only encountered one time, at the end of the burst.

Now let us compare these examples, making some assumptions such that, for both cases, the access time assumes two wait states before a read,

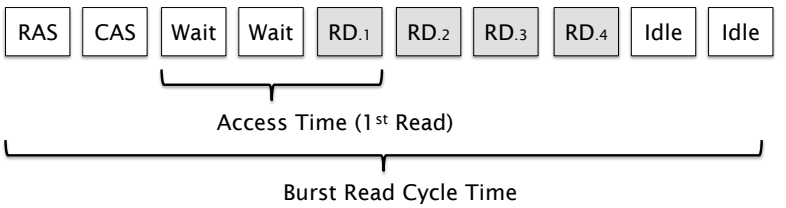


Figure 5.4: DRAM burst read operation. Showing actions performed in successive clock cycles, assuming a burst length of 4.

and also assuming that the cycle time includes two idle clock cycles after the read.

It can be observed that in the first case, with the assumptions as shown in Figure 5.3, and using the single-read DRAM memory access protocol, then one random read operation requires a total of 7 clock cycles if idle cycles are assumed present (or 5 without any). We would therefore require 28 cycles to read four consecutive values, repeating the single-read process four times. The figure of 7 clocks per access represents a rather poor level of performance, especially as a processor might desirably want a new memory value available in every clock cycle.

However, the second example, shown in Figure 5.4, uses a burst-mode access protocol, and can read all four locations in only 10 clock cycles, with the same assumptions. This translates to an average of 2.5 clocks per access, and clearly much improved performance. Indeed, this speed is approaching three times faster than the simple random access case. There are several reasons for this. We may note first of all that only the first read access requires the longer delay with wait states, and successive reads in the burst are then single-cycle operations. Secondly, if (as in this case) the idle cycles are assumed to be needed at the end of a read, they are incurred only once, at the end of the burst, rather than potentially four times[64].

Useful though burst-mode is, it cannot be used continuously in real programs however. This is because there will be sections of the program where burst mode is heavily used: primarily linear code sequences, and linear accesses to data sequences, in successive memory locations, but then the memory addressing pattern will reach a point where it will jump to a different part of memory, forcing the current burst to terminate early, and a new memory access to begin. If this happens frequently then the average length of a burst access may be less than the maximum supported burst length. In practice, what we will get is a mixture of the multiple cases, linear and non-linear patterns, based upon the kind of program and the kind of data we are dealing with.

[64] Why 'potentially'? If you appreciated the earlier comments relating to RAS Precharge, you might see that four consecutive reads could remain within the same row, and therefore the idle cycles are incurred only once in the simple case too. That would mean only 22 cycles for four reads, an average of 5.5 cycles per read. However, taking this into consideration requires a much deeper knowledge of the memory system and specifications than we wish to pursue at this point in our study.

Now let us evaluate a burst mode scenario with the same access protocol just discussed, but in this case we will assume that the memory access pattern is a mixture of single cycle access mode and burst mode accesses, as and when they are possible:

A Mixed Burst Mode Scenario

Suppose we find that burst mode can be used 40% of the time on average, with our memory system as already discussed. What will be the average data rate of the memory if it is 32 bits wide and the system clock rate is 1000 MHz?

Average rate of access:

Burst = (40% x 2.5) = 1.00
Simple = (60% x 7.0) = 4.2
Total average = 1.0 + 4.2 = 5.2

Memory width = 4 bytes, therefore **average per byte = 5.2/4.0**
Therefore, we can access one byte every 1.3 clock cycles.

There are 1000 million clock cycles per second in this system, so we can access 1000 million divided by 1.3 = **769 Million bytes/sec.**

Thus, the achievable memory data rate of this scenario is about **733 MB/sec (using binary megabytes per second).**

A further problem with using burst-mode is that the von Neumann architecture uses a single bus for both instructions and data. Therefore, we will find that the system bus is flitting between at least two sets of addresses at very different locations in memory. When we introduce the idea of instruction fetch, and overlapped instruction fetch, the estimates of memory behaviour become quite complex, and predicting exact performance is a challenge.

5.4.5 Page mode access

Page mode access, also known as fast page mode (FPM) DRAM, is another way of speeding up the slow standard DRAM access protocol, and as with burst-mode DRAM, this attempts to exploit the program and data addressing behaviour of the system favourably.

With burst-mode access, the system supplies one row and one column address, and then the DRAM will read consecutive locations. So the CPU

Figure 5.5: DRAM Fast Page Mode read operation. Showing each successive column address being provided in order to read the word within the row already activated by the first access within the page.

might request Row 6524, Column 122, and then it will receive data values at column 122,123,124, and 125 if the burst is four items in length.

A saving is achieved there by not having to send the same row and col values four times when the CPU knows it is not changing, and also by not having to complete a cycle time period after every read, just at the end of the burst sequence. However, if the whole burst sequence is not usefully exploited, then this efficiency starts to reduce quickly.

Fast-Page-Mode DRAM uses another variation of this idea, whereby the row address is transmitted the first time a new row address is encountered, and then only the column addresses are subsequently provided by the CPU, with the assumption that all of the columns in this page sequence are in the same row. Duplicating the scenario above, the CPU would request Row 6524, but then send column address 122, column address 123, column address 124, and column address 125. So a series of four reads, as in the previous examples, would look like Figure 5.5, with a total of 13 cycles for four reads (an average of 3.25 cycles per read).

This may initially seem to be just a less efficient version of burst-mode access. However, there is an advantage: the CPU can send **any column** address, in **any order**. So it could instead send Row address 6524, then column address 55, 12, 77, 78, and so on. It might be apparent that this is a more random ordering of addresses, yet within the same row (or page as it is known here), which is more in keeping with the semi-random nature of program code and data access patterns often encountered.

It would seem that FPM DRAM access mode is likely to be more frequently able to be used than burst-mode for many programs, but at the cost of reduced efficiency that can result from having to send a unique column address for every data item accessed.

There are several other variations of DRAM access protocols, indeed new schemes are being invented every year or two as DRAM design concepts

evolve. You may wish to research these and familiarise yourself with their mode of operation, but we will not discuss them further at this point. We will instead investigate the concept of cache memory, which has a lot to offer in smoothing over many of the problems just discussed.

5.5 Cache memory

Since we have established that a memory system composed of DRAM is far from perfect, even though it is relatively inexpensive, we are fortunate to have a further card to play in delivering a suitably high performance from our memory system. This is the concept of **cache**.

As we noted earlier, SRAM is smaller, more expensive, but much faster than DRAM. But DRAM is low cost, and allows us to have very large memories, even in everyday computers. Computer system engineers play a very clever trick to deliver both high performance **and** low cost from these memory systems, and as in the previous section, it exploits the behaviour of programs and data in the way they access memory.

It has been observed over many years of experience, that in very general terms at least, there is concept that 10% of program code is accessed 90% of the time[65] . This is very much due to the way programs are structured: subroutines and loops being a major contributor. Often many of these program accesses relate to near or adjacent locations (**spatial locality**), or are used multiple times within short periods of time (**temporal locality**).

[65] This is of course quite dependant upon the actual program or programs being studied, it might be 20/80, 15/85, or another ratio, but the principle that a small **subset** of the total program is used the majority of the time is still very prevalent.

A similar story can be told for data held in memory, but this is much more dependant upon which kind of data it is, what is being done with it, and how the programmer has chosen (wisely or not) to organise the data in memory[66] .

[66] For example, a programmer who knows much about the underlying hardware might organise data accesses to exploit the system in ways that allow more optimal memory traffic capacities.

For example, let us suppose that a program searches through a huge list of perhaps 10 million data values to find one particular item. Then, it may be the case that the CPU needs to access every data item exactly once. There is no opportunity to reuse these data values, so storing them in cache would be wasted effort.

On the other hand, a program performing mathematical manipulation of an image consisting of 10 million pixels (each one a data item) may need to access 10% of the pixels 10 times, and the rest of the pixels only twice.

The behaviour of these two cases is quite different. The second case is much more amenable to exploiting cache than the first case. Indeed, the first scenario may actually make system performance worse by wasting cache resources without any benefit.

Let us keep the story simple, and assume that data and program code both appear to follow the case where 10% of the memory content is accessed 90% of the time. Incidentally, we usually refer to the 10% portion of memory content that we wish to access frequently (and therefore efficiently) as the **working set**. So how does cache exploit this principle?

The concept of cache is to provide a small amount of fast SRAM alongside a large amount of slow DRAM. Indeed, we included cache as a system block in our von Neumann system diagram (Figure 2.1). When considered together, the cost of the whole memory system is still relatively low. However, we shall see that this can boost performance considerably for that small extra investment.

Imagine we have 512 Mbyte of SRAM-based cache, and 4 Gigabyte of DRAM, a good ratio (8 to 1) if we expect cache to keep a copy of 10% of memory contents. The SRAM has single-cycle access time. The DRAM has an initial access time of 3 clock cycles, and both memories also require row and column address cycles[67] .

[67] In practice, this means that the SRAM read time is 3 cycles (2+1) and the DRAM read time is 5 cycles (2+3).

Each time memory is read, SRAM is accessed. If the data is present, a cache hit then delivers a read in one clock cycle in this case. If a cache miss occurs, the DRAM also then needs to be read, a further 5 cycles. During a miss, the cache controller (a special control circuit) copies that data into SRAM so that next time it is accessed it can just get the data from SRAM (a cache hit).

Now, suppose that we have a hit-rate of 90% based on our assumptions of program behaviour as mentioned earlier: what will be the average read time?

Cache Example

90% of Memory Access with a cache Hit:
This uses SRAM, therefore read time **=3 clock x 90%**
10% of memory access with a cache miss:
This uses DRAM, therefore read time = **5 clock x10%**
But a cache miss also reads SRAM = **3 clock x 10%**
Total average read time:
= (90%x3)+(5x10%)+(3x10%) **= 3.5 clocks**

So we see that whilst DRAM requires at least 5 clock cycles to deliver every data item, the SRAM/DRAM cache example delivers the same data in only 3.5 clock cycles on average[68].

[68] As always, *if* we have more detailed information than that known in our simpler examples, we can refine our evaluations. The miss penalty of a cache might well require fewer cycles than a cache hit. Consider what the same calculation would derive if the miss penalty was only two cycles for instance. Likewise, in the absence of such information we should assume the pessimistic case of miss and hit being of equal cost (as we did in this case).

What we have just confirmed is that small amounts of cache memory can speed up memory access by significant amounts. When a complex memory system employs wider memory busses, burst-mode and page-mode access, and cache together, the memory bandwidth can reach quite high levels of average performance.

Now, what about cost? Let us also make some assumptions here: We could get suitable data from researching the actual components if we wished, but some arbitrary values will do for this example.

So let us suppose that 1 Gigabyte of SRAM costs $200 and 1 Gigabyte of DRAM costs $50. There are now three possibilities for our 4 Gigabyte memory system, each with a different cost:

Option 1: Only Use DRAM (least expensive option)

DRAM $50 per Gigabyte, so total cost = 4 x $50 = $200

So in this case we get a system where a memory access takes 5 clocks. This is the lowest cost option, but also the slowest.

Option 2: Only use SRAM (most expensive option)

SRAM is $200 per Gigabyte, so total cost = 4 x $200 = $800

This option is very expensive, but we get the fastest memory accesses which means optimal memory speed.

Option 3: Use SRAM and DRAM

SRAM is $200 per gigabyte, so cost for 512MB = $100
DRAM is $50 per gigabyte, so cost for 4GB = $200
Therefore, Total cost = $300
This option is near to the less expensive option but performs very close to the fastest. It is therefore very cost effective.

It should be clear now that cache provides a very useful compromise: we get most of the performance benefit of an optimal memory configuration,

but at a cost that is almost the same as the least expensive option. This is an example of what we mean by a strong cost-performance tradeoff.

5.5.1 Multilevel cache

Another aspect of cache that you may come across in computer systems architecture is the idea that there can actually be **multiple caches** in the system. This enhances the benefits we have already highlighted for immediate memory read times, but also helps to alleviate some of the issues relating to the von Neumann bottleneck, as we shall see.

The concept of **multi-level cache** is the most common case we might encounter. Put in simple terms, a processor may have an external bus and external DRAM memory chips, and alongside those DRAM chips there will be an external SRAM cache. However, at the same time that this is happening, the CPU chip can also have cache built into its internal circuit layout (known as **on-chip cache**). And furthermore, this cache may be split into two parts (**a dual cache**), these being **instruction cache** and **data cache**. An example microarchitecture is shown in Figure 5.6.

We can see that the von Neumann bottleneck is reduced in this machine, since both instructions and data can be fetched at the same time, provided at least one of the dual caches scores a hit. Indeed, the bottleneck is only incurred here when both caches miss simultaneously and then both have to compete for access to the main system bus and external memory. If the miss rate for each cache was 10%, then the probability of both caches missing simultaneously would only be 10% x 10% , which is only 1%.

When this example processor is used in a system where the external memory already has a cache, it would be referred to as a **two-level cache hierarchy**, where the on-chip cache represents level one and the external cache represents level two.

Some processors take this a step further, having the split instruction and data cache on chip, then also a combined cache (the **unified cache**) also on chip, and finally the main memory cache[69].

[69] The INTEL Core i7 processor for example, has such a hierarchy. The i-cache and d-cache operate in parallel at **level-1**, and connect to the unified cache at **level-2**, both on chip. Finally, the external memory bus will likely have a cache of its own, forming **level-3 cache**.

There are several benefits of using on-chip cache:

- The on-chip cache can be optimised to work with the CPU circuitry to hide the address setup clock cycles, using overlapping activities (pipelining) to make these appear to take zero time. This means that the on-chip cache can deliver a new data item for every clock cycle.

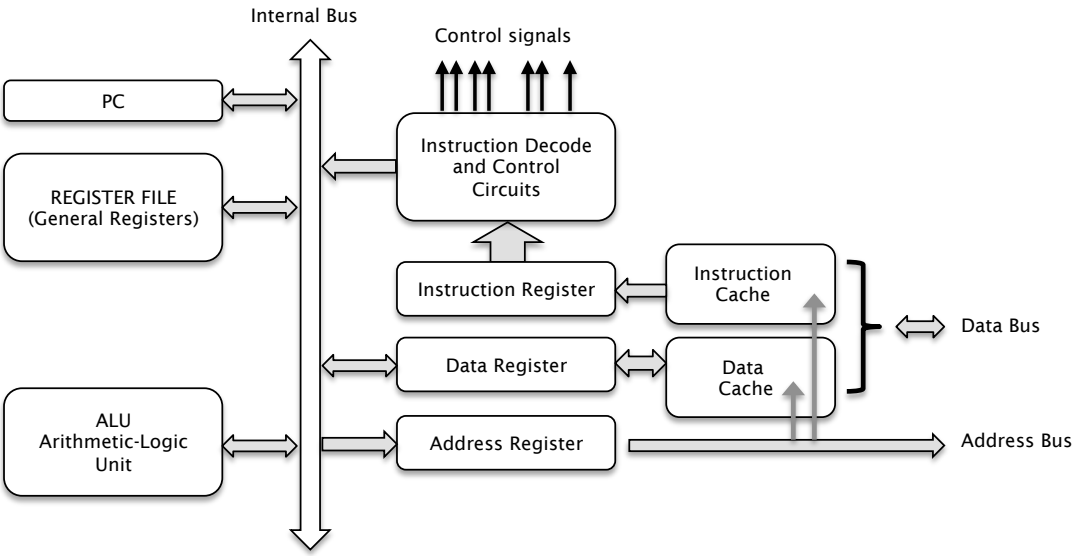


Figure 5.6: A scalar microarchitecture with dual on-chip cache hierarchy. Showing separate data and instruction caches on chip, providing independent memory channels.

- The width of the on-chip cache can be fairly arbitrary, matched to the internals of the CPU rather than external memory, and potentially organised in ways that maximise cache performance for that specific processor under specific conditions.
- The instruction cache and data cache can be accessed simultaneously, and it is rare that both caches have a miss at the same time, meaning that even if one cache misses, it can use the external memory bus whilst the other cache continues as if nothing has happened.
- The split cache gives the impression that the memory bus no longer has the von Neumann bottleneck. On-chip at least, it is much closer to a Harvard architecture.

Importantly, all of these advantages come as part of the CPU chip, and do not need radical changes to the design of the system as a whole.

Now consider our previous cache example again. Remember that we discovered that we were able to reduce memory read time from 5 clock cycles to 3.5 clock cycles. Now let us consider what happens if the on-chip cache can also achieve a 90% hit rate, and requires only one clock cycle per access:

Multilevel Cache Calculation

90% hit rate: Internal (on-chip) access:
Average main memory read time = 90% x 1 clock cycle
10% miss rate: External read time:
Average main memory read time = 10% x 3.5 clock cycles
Cache miss penalty = 10% x 1 clock
Total average read time:
(90%x1)+(10%x3.5)+(10%x1) = 1.35 clock cycles.

Let us reflect upon what we have now achieved: The average memory read time in this multi-level cache system is now getting reasonably close to 1 clock cycle; in other words, very close to optimal performance.

But, there is much better news hidden in the data: Actually, this result may be duplicated for each cache, operating in parallel. So now, with the ability to operate the CPU such that it can access both instruction and data cache simultaneously, we could reasonably argue that **average access time** is actually half this amount, or 0.68 clock cycles[70] .

If we were again to assume the processor/system clock frequency is 1000 MHz, then we can also say that instruction bandwidth for the multilevel system with split cache is 1000/0.68 = 1470 Million reads/sec, and the same for data bandwidth. Compared to the original figure of 200 Million reads/sec, from our first memory example, in Section 5.4.1, this is over seven times the performance of that original case, but with very small investment[71] .

[70] This is evident, because if we can read two values at a time with an average access time of 1.35 clocks, then the average per single value is 1.35 divided by 2, giving 0.68 clocks per access.

[71] For a 32-bit internal/external memory width assumption, this would imply nearly 6 billion bytes per second.

5.6 Cache management policies

In the simplest terms, cache is merely a much faster memory, where the most useful parts of the main memory content are kept for quick access. The definition of what is useful and how this should be determined, to exploit temporal locality, is the basis of a concept called **cache replacement policy**. Meanwhile, the way the data is organised within the cache, to enhance spatial locality, is known as the **cache mapping policy**.

5.6.1 Mapping policies

For a cache to maintain copies of selected memory locations within its faster memory, it needs to keep track of which locations it has cached,

as well as their content. In the simplest possible scheme, a cache could therefore be organised as a table of values, where the first **n** bits designate the whole address of the item being cached, and the remaining **m** bits represent the copied data. If a memory system is 32-bits wide, then **m** must be 32 bits. If the address space is also 32 bits, then **n** will also be 32 bits, and thus the total size of each cache entry would need to be **n+m=64** bits. We also typically need at least one bit for each entry to determine if the data is actually in cache[72], and another to indicate if the cache content is also resident in memory[73].

[72] This is known as the Valid bit, and indicates if the cache entry has been read in from memory already.

This simplest case (known as a **fully associative cache**[74]) results in a lot of circuitry and bit storage flip-flops. Since multiple adjacent locations are often within the same working set (spatial locality), then having a separate address field for every single location is very inefficient. Consequently, caches are usually organised into **cache lines**, **cache blocks**, and **cache sets**, in an attempt to optimise hardware cost and performance.

[73] This is known as the Dirty bit, and indicates if data in the cache needs to be written back to memory at some point.

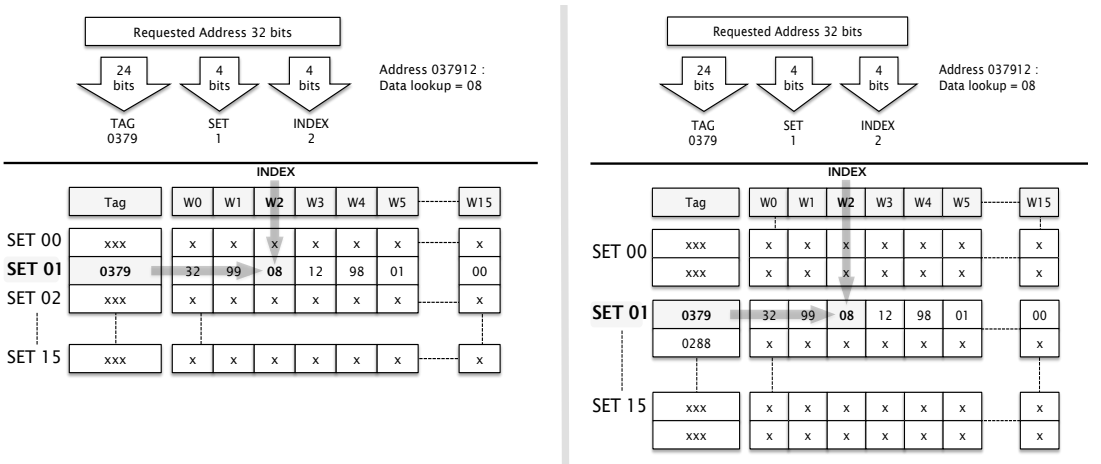
In cache terminology, a **tag** or **tag field** is simply an address or more typically a portion of an address. A **line** is simply a tag and one or more data words related to that address or **locality**. If a line contains more than one data word then those words represent a block. A **block** might be 4, 8, 16, 32, or any useful group of consecutive data items relating to a 2^n numerical word count.

[74] The simple case is a fully associative cache with block size of one

Thus, to improve our simple case, we might decide that each address relates to a block of sixteen words, or memory locations. We would then have sixteen words per address field, resulting in the address **tag** shrinking by 4 bits and becoming a **partial address**. In hardware terms, we end up with a line consisting of sixteen 32-bit words plus a single shared address tag of 28 bits, a total of 540 bits, or about 34 bits per word[75], requiring much fewer storage bits, and much less circuitry and significant reductions in power consumption. Adding a valid and dirty bit for each word would take the total number of bits to 572 and an average of around 36 bits per word, which still compares very favourably to the 64 bits per word in the simplest case given earlier.

[75] This would be described as a fully associative cache with block size 16.

Now what about **sets**? The idea of a **set** is that a fixed part of the address range has a set of lines associated with it. If we designed our cache to have 16 sets, with a single line in each set, then the already smaller tag field can again be reduced by a further four bits (since 2^4=16). By doing this we would end up with the **directed mapped cache** scheme shown in Figure 5.7(a). Here there are 16 sets and each set has a size of one line. Note that if we had the opposite (only one set but lots of lines



(a) Direct Mapped Cache

(b) Set Associative Cache

Figure 5.7: Cache Mapping Schemes. Item (a) shows a **Direct Mapped** cache, which has multiple sets and single lines per set. Item (b) shows a **Set Associative** cache, which has multiple sets and multiple lines per set. Compare these cases to a fully associative cache, where there is only one set but multiple lines.

in that set), then we have ended up back with a fully-associative cache once again.

From this point it is easy to extend the concept of direct mapped cache to **set associative cache**. Actually, directed mapped cache is simply a special case of set associative cache where the set size is one line per set. So adding **multiple lines to each set** gives us set-associative cache mapping as shown in Figure 5.7(b). This final scheme has some advantages beyond simply saving on cache storage hardware cost, but this is beyond the scope of this introductory text. Advanced readers might wish to read further on the subject however.

5.6.2 Replacement policies

A cache **replacement policy** is an algorithm used by a cache controller circuit to decide what should be in the cache and what should not. Remember that cache is typically small compared to main memory, so it can only hold a subset of the whole memory content. For cache to be successful at speeding up memory access and memory bandwidth, it needs to make sure it contains the most useful content. As a consequence, there will come a time when, to bring something new into cache, some

existing content must be discarded to make way. Out with the old, in with the new. Deciding what should be replaced is the basis of **cache replacement policies**.

There are a number of policies that can be used. These are worth reviewing briefly, if not in advanced detail.

Least Recently Used (LRU) replacement policy: In this case, the data item in the cache that was least recently used (not necessarily the oldest item) is replaced by the new item. The cache must keep track of when items were added to its content, to support this policy, which exploits temporal locality.

Least Frequently Used (LFU) replacement policy: In this case, the data item in the cache that was least often used is replaced by the new item. The cache must keep a tally of the use of each item in cache in order to support this policy.

First In First Out (FIFO): replaces the oldest item in the cache with the new item regardless of how often that item has been used. Circuits are required to track the age of the items in the cache.

Random Replacement Policy: In this case, the data item in the cache to be replaced is chosen at random. This has the advantage that there is no need for circuits to keep track of the use of items in the cache.

There are many other algorithms, many of which are variations on those just mentioned. These are rather specialised and need more advanced knowledge of computer architecture to fully appreciate. Also, it should be noted that when we talk about replacing an item, this may well be a single word, or a line, or even a set, depending upon the cache design and replacement policy.

5.6.3 Write-back policies

There is one further important consideration for cache memory, which relates particularly to writing data to memory rather than reading data from it. This is the concept of cache write-back policy. This is also important when understanding cache design. There are two significant variations of this concept:

Delayed write-back: This will allow the cache to capture any memory writes generated by the CPU, without involving main memory. Repeated writes will go to the cache, which is much faster. Only when an item

is finally discarded from the cache, during a replacement action, will its latest new value be written back to main memory.

In computer architecture, the idea of a **dead store**, an item written to memory and then overwritten again before it is ever read, indicates that those memory writes are redundant. Delayed write-back captures this optimisation very well.

However, this policy also means that discarding a cache item necessitates a main memory write operation (but nowhere near as many repeated writes as are being directed to the cache for that same memory location). This potentially has a speed penalty, as well as advantages.

Cache Write-Through policy: In contrast, a **write-through** policy ensures that every write to a location held in cache is also written at the same time to main memory. This means that every cache write has a main memory speed penalty associated with it.

This model is less efficient, but it does ensure that main memory content is always as up-to-date as possible. Nothing can reside in cache that is not also in main memory.

You may wonder why there are two policies. Delayed write-back seems more efficient than write-through, so why not just use that? The answer is to do with a concept known as **memory coherency**. When the memory content is identical to the content the processor sees, when 'viewing memory through the cache' so to speak, we say that the memory is **fully coherent** (or in sync with cache content if you like). This is always the case with write-through cache, since the memory is always updated by a write that also updates the cache. On the other hand, write-back cache does not immediately update main memory when it receives a write, and therefore the memory coherence in such a system will not be 100% all of the time.

Does memory coherence matter? This very much depends upon the system. Often the answer is no. However, where a system has a number of devices connected to the system bus, and these devices have access to parts of the main memory, then the fact that they may be looking at out-of-date memory content might well be important.

Likewise, if a device in the system changes the content of main memory, and the cache is not aware of this, then it could keep providing an out of date value to the CPU when it performs a read (known as a stale cache item). This is one aspect of the related problem of cache coherency.

Although it is beyond the scope of this text to get into more detailed discussions of replacement and write-back policies, it is clear that they affect the way the CPU sees the content of memory, and for some applications and some systems designs, this could be a problem if not taken into account.

These hierarchical internal cache schemes, and the choice of replacement and write-back policy, will dictate performance for a system, and can sometimes be chosen specifically for certain kinds of computational workload. A weather simulation is very different from a video editing package, even though both may deal with large volumes of data. Where a system is built with a specific application in mind, the cache and memory architecture can be customised to maximise performance based upon knowledge of how the system behaves in terms of patterns of code and data usage[76].

[76] The interested reader can read further on this topic: in particular the concept of associative cache policies.

5.7 Avoiding errors

Whilst most memory systems are utilised without **error detection** and **error correction** in mind, there are some memory systems where this is a major consideration. A bit error in a large memory system could be caused by a number of things, including faulty transistors in the memory chips themselves, potentially damaged due to voltage surges, or excessive heat due to insufficient cooling, momentary electrical interference, or even radiation. There are situations where this cannot be tolerated, in particular in safety-critical systems as mentioned earlier.

There is thus sometimes a need for memory systems that detect and even correct errors. One mechanism is to use a specialised memory controller, and another is to use a slightly modified version of RAM known as **ECC memory**. ECC is an abbreviation of Error Correcting Code, and there are potentially several algorithms that are used to implement ECC capabilities, typically derived from a mathematical technique known as Hamming codes. Simpler techniques include parity checking, a topic which we will look at further in Section 14.7.

Power-on self test is another mechanism that helps to reduce memory errors. When a computer turns on, it may perform a startup checklist, including writing test data patterns to every memory location in its physical memory. If the correct response does not occur when reading back the data it has written, then it knows that this location of memory is damaged (or potentially the connections to the chip). A lot of systems

will, at this point, just stop with an error message, and request that you resolve the problem. In some cases, however, it may be possible for the operating system or BIOS to record the location of faulty memory and mark this as unusable during system operation. This is particularly common where flash memory is used.

5.8 Programming considerations

We have focused up to this point on the hardware aspects of memory systems. At this point it is worth introducing some additional points, particularly relating to memory architecture in the programming domain.

The concept of a **memory map** is often used to represent the structure of the memory as it may be accessed by the programmer. For example, a 16-bit ranged memory map may show some of the following aspects:

0000-07FFh : BIOS ROM
1000-37FFh : FLASH RAM
4000-7FFFh : RAM
8000-EFFFh : UNUSED
F000-FFFFh : IO Mapping

This can be translated into a diagram as shown in Figure 5.8. Most of these items should make sense as blocks of memory. However, the final block of memory (**IO Mapping**) is unusual. What this memory map suggests is that **IO devices** are mapped into memory addresses. This **memory mapped IO** makes some sense if we remember that IO devices (or at least their interface circuits) are often connected to the system bus alongside memory and CPU and are therefore potentially visible to the CPU as if they are memory spaces. In practice, what happens is that a given IO device will occupy a small number of uniquely allocated locations, relating to individual control addresses within that IO device. For example, a keyboard and mouse might have the following address designations:

F000h : Main Key Value
F001h : Status of Caps Lock key (on/off)
F002h : Caps Lock Light Control (on/off)
- - - -
F020h : Left Button Status
F021h : Right Button Status
F022h : Motion Register

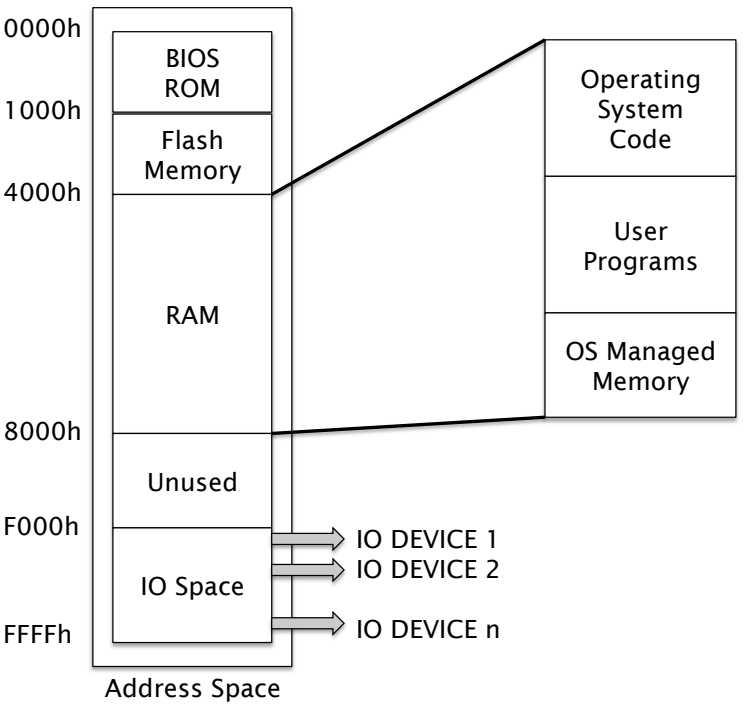


Figure 5.8: Memory Map Example. Showing BIOS in low memory, flash memory for reconfigurable firmware, RAM, and IO device mapping space.

These are typically known as **IO addressable registers**. Each IO device will have a set of such register addresses, relating to internal functions and data. The assignment of IO addresses is normally controlled by the operating system in a modern computer, either via the low-level (BIOS) or the installed OS[77].

One important thing that we might note is that if IO registers look like memory addresses, then they could, in theory, end up in cache. In the case of our keyboard, if we keep reading a key value from cache, we will not see new key-presses (only the first one to go into cache). This is a case where memory content should not be cached.

Another scenario is where two processors share access to the same memory area (a shared memory system). Clearly, if either processor is caching the content of the shared memory block, it will be unaware of any changes made by the other processor, which defeats the point of having shared memory. This cache coherency issue is a well known problem.

To combat some of the problems just highlighted, some operating systems permit areas of memory to be designated as **non-cacheable**, and

[77] In practice, mapping an IO device with 'n' IO registers to a very specific set of n corresponding memory addresses would require lots of logic circuitry, with little benefit. Consequently it is normal to map, for instance, eight IO registers to a larger memory block. Thus, our keyboard example might repeat the IO register addresses across the range F000h to F01Fh, and the mouse register addresses might occupy FO20h to F03FH. Registers simply appear to repeat within this block. For the programmer, there is no disadvantage.

configure the memory controller and/or cache controller appropriately.

Many processors also support the idea of IO having a separate address space (**Direct IO Addressing**), which allows it to exist in a memory space outside of the scope of the caching hierarchy. Memory maps can be very useful for illustrating these kinds of systems and making such problems and solutions visible in the design process, and in the system documentation.

Finally, memory maps also provide a useful way to visualise how memory is utilised by the software and operating system components. In Figure 5.8 for example, we see that the RAM is organised by the operating system into different functionalities.

5.9 The future of memory devices

Memory devices have progressed almost incomprehensibly over the past 40 years. The first memory chips, manufactured in the 1970s, had capacities measured in kilobytes. Modern memory chips have capacities which are vastly superior. This advance has been driven primarily by the advances in silicon chip manufacture. Early chips had component dimensions (**feature size**) measured in **microns** (millionths of a metre), whereas today's technologies have components measured in the **nanometre** (billionths of a metre) scale.

Even in the past ten years, chip technology has progressed from 90nm down to 10nm, an almost ten-fold reduction in feature size, and potentially a 100-fold reduction in silicon area per bit-cell. This translates to memory chip capacities increasing by over 80 times in a decade. As of 2019, 16 Gigabytes on a single DRAM chip is considered to be fairly advanced, especially if combined with high-speed data access techniques. Figure 5.9 shows recent memory capacity trends over recent years.

It would be fortuitous if feature size continued to shrink every year, far into the future, resulting in ever increasing memory chip capacities. Unfortunately, **VLSI**[78] technology is hitting a number of roadblocks at the present time. It was once considered doubtful that circuits with feature sizes of 10nm were capable of being reliably manufactured, yet this has been achieved, and it is not inconceivable that this may continue a little further. However, there are basic laws of physics that prevent traditional electronics circuits from operating properly at ever smaller sizes. We are rapidly reaching the point where transistor devices are so small that effects of individual atoms and electrons become significant enough to

[78] VLSI: Very Large Scale Integration, representing the current chip fabrication era.

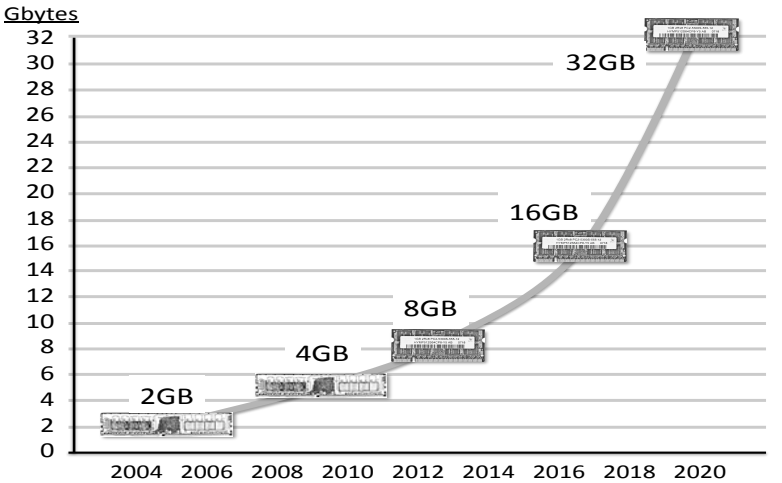


Figure 5.9: DRAM memory module capacity trends. Showing plug-in memory module growth in storage capacity over the past 15 years.

spoil the party. It is notable that memory chip capacities were doubling roughly every two or so years, but more recently this has slowed, as these effects become harder to design around.

A further problem is the concept of **yield**. When chips are manufactured, some of them are faulty. With one billion transistors on a chip, there is a small probability that at least one of these might be substandard or just non-functional. This could mean the whole chip is useless and has to be discarded[79]. The remaining chips, with no faults, are the ones that can be sold. If 70% of chips in a batch are good, then the yield is 70%. This impacts upon cost, since the manufacturer still has to make 100 chips in order to get 70 that work. It then follows that every time the number of transistors on a chip increases, the risk of a fault increases too (and thus yield drops, then costs potentially go up). This problem only gets worse as we head down toward ever smaller feature sizes.

[79] There are a few tricks that can be played to reduce this impact. For example, if a 4 Gbyte DRAM chip is manufactured with four identical blocks of 1 Gigabyte inside its chip layout, and one block is found to be faulty, then that chip might be reconfigured to disable two blocks (including the faulty one) and then be sold as a 2Gbyte chip. This helps to keep costs lower by making effective yield better across several product lines.

To bypass the worst of the yield problem, and side-step the physics of very small components, manufacturers have recently taken a slightly different route. Rather than making a RAM chip that is twice as complex, they take two chips manufactured with existing technologies and stack them one on top of the other (di stacking), using specialised micro-manufacturing techniques. We can quite happily stack four or perhaps even eight chips into a stack, and as a result, the size of the chip does not increase much (since most of the physical chip size is related to the external package and not the small piece of silicon inside).

A limiting factor here is power and heat. As more chips are stacked and

packed into a small chip package, they consume more power in a smaller volume (the **power density** increases). This in turn leads to higher heat concentration within the package (thermal density).

Stacking also makes it harder for heat to escape from the silicon chips. If chips get too hot then they become unreliable, and ultimately get damaged. It is not unusual to see high performance memory modules having their own heatsinks to help cool the modules down when running at peak activity. Ultimately, there is a point where it is not possible to pack more chips into a given module, unless feature size, power, and heat are all reduced further. This barrier to progress is sometimes referred to as the **power wall**.

Another area of interest, apart from data capacity, is of course speed. By organising internal memory structures in the internal memory chip design, there are ways to accelerate the access of data within the chip. In terms of external data transfer from chip to processor, there are also new methods of speeding up data transfer, by altering bus protocols, and methods of clocking data from one device to another.

Various **DDR** (**Double Data Rate**) schemes already exist; for example, combining dual clock-edge data transfer and burst mode data transfers (for example **DDR2** and **DDR3**), and these have helped to push data transfer speeds across busses to high levels.

In the more distant future, new electronics and physics discoveries will help to provide yet more memory advancement. The recent discovery and implementation of the **memristor**, a new kind of electronic component, has promised to allow new innovations in memory technology. It may take a decade for these leading-edge technology concepts to mature to mainstream products, so we shall see how things develop.

5.10 Summary

Memory is a fundamental and extremely important component of the working computer system. In this chapter we have found that rather than memory being a single entity, it is actually an entire family of related but differing component technologies, each with specialist capabilities.

A particular observation made in this chapter relates to performance: memory is imperfect and creates delays. Delays make processors wait for data when they should be working. Therefore, memory is a limiting factor on both processor and system performance.

Finally, we discovered that to overcome these limitations, it is necessary to make carefully considered compromises in memory system design, balancing cost against ideal performance. As with most solutions in computer architectures, introducing a solution, such as cache for example, brings new problems. The designer must be aware of these, at the hardware level, software level, user programming level, and also appreciate the consequences for the operating system too.

5.11 Terminology introduced in this chapter

Burst-Mode cache coherency	Cache hit
Cache line	Cache memory
Cache miss	Cache replacement policy
Cache write-back	Cache Write-Through
Capacitive bit cell	Column
Column address	Computer memory
Cycle time	Data bandwidth
Data cache	DDR Double-Data Rate
DIMM	DRAM
Dynamic Ram	EAROM
ECC memory	EEROM
Effective memory bandwidth	EPROM
Erasable Memory	Error correction
Error detection	Fast-Page-Mode
Feature size	Flash Drive
Flash Memory	Heatsink
Hit-rate	Instruction bandwidth
Instruction cache	Internal readout
Least Frequently used replacement	Least recently Used replacement
Linear sequence	Magnetic core-store
Memory bandwidth	Memory bus width
Memory capacity	Memory coherency
Memory data rate	Memory module
Memory stick	Memristor
Mercury delay line	Micron multilevel cache nanometre
Non-linear sequence	Non-programmable memory
Non-volatile	One-Time Programmable
Physical memory	Power density
Power wall	Power-on self-test
PROM	Random Replacement
Read-Only memory	Refresh cycle

List continued overleaf ...

ROM	Row
Row address	Safety-critical system
SIMM	SRAM
Stale cache item	Static Ram
Storage density	Storage device
System bandwidth	Thermal density
Unified cache	Virtual memory
Volatile	Working set
Yield (chip)	

These terms are defined in the glossary, Appendix A.1.