Chapter 8
Memory and Storage

640K [of main memory] ought to be enough for anybody.

– W.H. Gates [alleged, almost certainly misquoted, but still amusing]

Abstract In Chapter 5 we skirted around the idea of long-term memory much like we did with the idea of computation and the ALU. This was an attractive option at the time because it meant we could focus on the processor itself, but the concept of memory and storage is of course central to the operation of the processor: everything it does is oriented toward computation on (large numbers of) values we must store somewhere if the program being executed is not trivial. To examine these issue in more detail, in this chapter we investigate the design and implementation of a range of memory and storage devices focusing on those most pertinent to processor design. Specifically, we include registers and main memory already discussed previously, describing in basic terms how they can be modelled in Verilog. Ultimately, our goal is to build a memory hierarchy capable of supplying data and instructions to our processor data-path via a standard mechanism; to achieve this efficiently and robustly we include an investigation of cache memory and issues such as error correction.

8.1 Introduction

A **memory**, or more generally a **storage device**, enables us to store items of data for a period of time. Such devices represent a crucial part of any computer system since, for example, we need to store values somewhere while the processor performs computation on them. However, even though the speed of processors has increased roughly in line with the increase in transistor performance, memory access time has improved much more slowly; only a few percent each year, about 30% in ten years. This is a massive problem: for example, it does not matter how fast we can perform computation in our ALU if loading and storing operands is slow. Furthermore, and perhaps more importantly, we have already seen that the fetch-decode-execute cycle

requires the processor to fetch instructions; we perform at least one load from the memory where instructions are stored in each processor cycle. If this fetch cannot be completed quickly, the entire fetch-decode-execute cycle will be dominated by the cost of accessing memory. This problem is often called the **memory wall** [68] or **von Neumann bottleneck** and it demands we, as computer architects, think carefully about all aspects of memory and storage device design.

Throughout this chapter, we try to refer only to memories rather than the more general case of storage devices which might include magnetic and optical disks for example. This might seem a bit odd but very roughly, only when discussing their implementation are the two different: abstractly we can follow Chapter 5 and describe either as an infinite array MEM in which we can store n-bit binary vectors. Two main functions are required to access the structure:

$$\begin{array}{ll} \textsc{Load}(x) \quad : \{0,1\}^* \to \{0,1\}^n, & \textsc{Load}(x) = MEM[x] \\ \textsc{Store}(x,y) : \{0,1\}^* \times \{0,1\}^n \to \perp, & \textsc{Store}(x,y) = MEM[x] \leftarrow y \end{array}$$

In this context x is an **address** or **index**, a name or label used to identify a particular **element** or **location**. Given an address x, the LOAD function retrieves an element while the STORE function updates an element with the value y. Essentially, the main differences between what we might normally call memories and storage devices are just how we implement the LOAD and STORE functions in a concrete way, and exactly how addresses refer to particular elements.

We can categorise our memory or storage devices according to a number of key characteristics:

- A **short-term** or **volatile** memory retains stored data while the device is powered; when it is powered off, the content is lost. A **long-term** or **non-volatile** memory solves this problem by retaining stored data even after the device is powered off.
- A **Read-Write (RW)** or **mutable** memory can be used to overwrite or store new data items and retrieve existing items. The content of a **Read-Only (RO)** or **immutable** memory cannot be altered; one can only retrieve existing data items. Read-Only memories are often abbreviated **ROMs**.
- When using a **random-access** memory, one can store or retrieve *any* data item; one is not restricted in terms of the order the data items can be accessed in. A **sequential-access** memory imposes restrictions on the *order* data items can be accessed in; one must access them in sequence. Random-access memories are often abbreviated **RAMs**.
- A **primary** memory is connected directly to the processor; usually such a device will be used to store active items of data which are required for the current task. A **secondary** memory does not usually have a direct connection to the processor, it is accessed as a peripheral via some other interface.

Thus far we have talked about memory being used to store and retrieve operands for the ALU. In order for the ALU to operate however, we need both operands and operators: the values that decide which arithmetic function the ALU performs. Thus we can place stored content into one of two classes: data, which are the values we

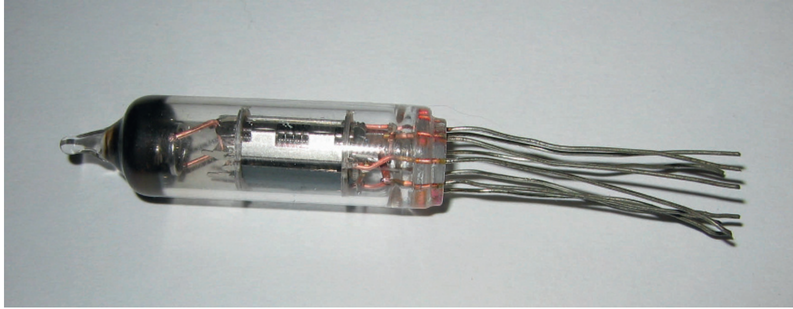


Figure 8.1 A military quality sub-miniature vacuum tube. (Reproduced by permission under the GNU Free Documentation License, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:CV4501.JPG`)

operate on, and instructions, which tell us which operation to perform. Given this definition, and recapping for completeness on the discussion in Chapter 4, one can reasonably construct two distinct forms of memory organisation:

Harvard Architecture A Harvard architecture treats data and instructions as separate entities. One would expect two separate memory hierarchies, and hence two interfaces to said hierarchies, dedicated to storing exclusively *either* data *or* instructions.

von Neumann Architecture A von Neumann architecture takes the opposite approach. There is no inherent difference between data and instructions; we have already seen that both are just numbers, it depends on how you interpret those numbers. For example, the number 12 might represent the integer value twelve if used as an ALU operand or represent the code used to invoke the addition function if used as an ALU operator. Thus a von Neumann architecture has a single, unified memory hierarchy that is used to store *both* data *and* instructions in the same way.

8.1.1 Historical Memory and Storage

In Chapter 4 we introduced EDVAC, the first practical stored program computer, which was based on the stored program architecture. EDVAC is typical of even modern computers in that it used three main forms of storage device: a **vacuum tube**-based accumulator to store very short term data that resulted from computation by the ALU, a **mercury delay line**-based volatile memory for longer term data and instructions, and a **paper tape** reader for non-volatile, long-term storage.

The vacuum tube was the technology replaced by smaller, more reliable devices such as transistors. Essentially they perform the same function, but are realised us-

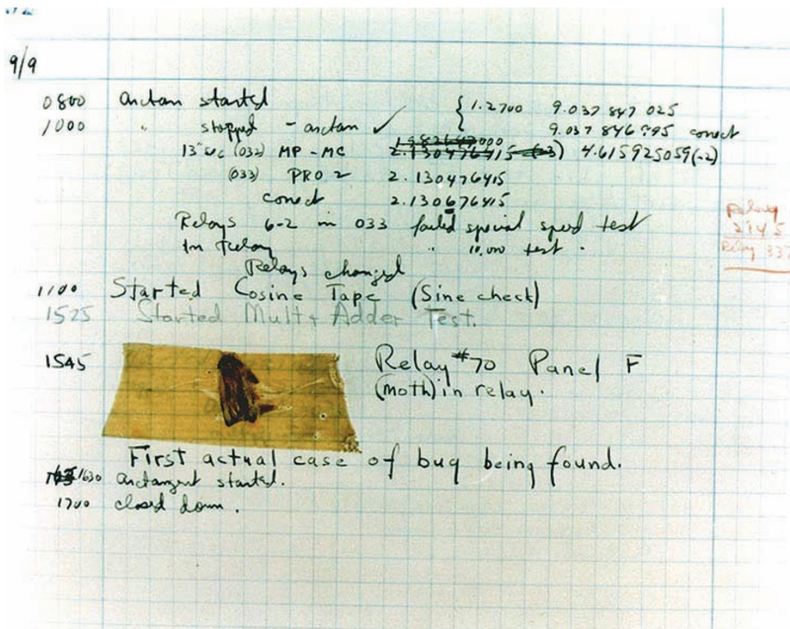


Figure 8.2 The infamous Harvard Mark II "bug" ! (A U.S. Navy photograph now in public domain, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:H96566k.jpg`)

ing a different technology; as such they can be used to construct logic and storage devices. A typical vacuum tube, or thermionic valve, looks very much like a light bulb: it uses a glass or ceramic envelope to hold a vacuum that surrounds an electron-producing filament (or cathode) and a metal plate (or anode). When the filament is heated, electrons are produced into the vacuum which are attracted by the plate resulting in a current between the two. In simple terms, this implements a basic switch: when the filament is heated the switch is on, when it is cooled the switch is off. Although similar logic and storage devices can be constructed using a vacuum tube as one would construct using a transistor, the scale of such devices is limited by reliability. Specifically, the filament slowly degrades due to impurities in the tube; the vacuum in the envelope becomes less pure as time progresses. Furthermore, the filament is stressed during the heating process which will eventually lead to it burning out; early users realised that this happened particularly often during powering up or down of a vacuum tube-based device. Although cooling devices eased this problem, tube failure was one of the most common causes of early computer crashes.

In fact, the terms "bug" and "debug" in relation to programming both stem (at least in part) from failure of this sort. In 1945, programmers of the Harvard Mark II computer developed by Howard Aiken, discovered a moth inside one of the com-

ponents; the unfortunate insect had shorted the component which had resulted in the computer malfunctioning. Although the terms had been used in various areas of engineering previously, Grace Hopper and her programmers are often cited as introducing them in the context of computer science. Certainly this real-life bug is so famous that it is still on display in the Smithsonian Museum of American History !

A mercury delay line is essentially a tube filled with liquid mercury with a pair of microphones and a speaker at each end. To store a value, the speaker at one end sends a signal representing the value into the mercury tube; due to the high speed of sound in mercury, the signal then travels along the tube faster than in a medium such as air. Upon receiving the signal at the other end via the microphone, it was relayed to the speaker and sent down in the opposite direction. This process was repeated, effectively storing the value as a perpetually regenerated looping signal within the tube. To read the value, one simply intercepted the signal as it was received by one of the microphones. To update the value, one again intercepted the signal as it was received by one of the microphones but replaced it with the new value before passing it on to the speaker for reinsertion into the tube. EDSAC stored 512 words of memory within thirty two delay lines. As one can imagine, the cost and toxicity of mercury presented a problem in terms of their use. Other problems were presented by the speed of sound in mercury, which limited the number of bits stored by each tube to around 576, and the fact that this speed changed as the mercury temperature altered ! Even so, the devices were much more reliable than other technologies of the time and have been used in computers as recently as the 1960s.

Although Babbage had proposed to program his Analytical Engine via a system of paper cards, perhaps the first use of punched card storage was by inventor Joseph Marie Jacquard in the early 1800s. Jacquard developed a weaving machine or loom whose operation, i.e., the pattern of weaving, was determined by a primitive program encoded as punched holes in a card. Roughly speaking, each hole in the card controlled a needle in the machine, each card represented one row of the intended design. The weaving process was therefore simplified from manual configuration of the machine, to semi-automatic weaving whereby the operator simply swapped in the right card at the right time. Later, in 1890 Herman Hollerith used a similar storage medium to encode US census data. He developed a mechanical tabulating machine which was able to process the cards, each of which represented the details of an individual, and produce statistics at an unprecedented speed: the 1890 census was processed in less than a quarter of the time the 1880 census had taken to complete by hand. In 1896 Hollerith founded the Tabulating Machine Company, an early incarnation of the company we now know as IBM, to exploit his invention.

Paper tape storage is a fairly natural progression from punched cards and the two mediums share much of the same technology. Instead of individual cards, paper tape is a continuous ream of paper which is notionally divided up into rows. Each row allowed for a number of holes to be punched; early tape systems allowed five holes which allowed for the use of Baudot character encodings, later systems allowed up to eight which allowed for more modern encodings such as ASCII. The tape would have perforated sprocket holes on the edge which allowed it to be fed through a tape reader which, using either a mechanical or optical mechanism, was able to translate

Figure 8.3 Maurice Wilkes with an EDSAC mercury delay line. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/delay_lines.jpg`)

Figure 8.4 Experimental EDSAC memory tank. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/edsac99.23.jpg`)

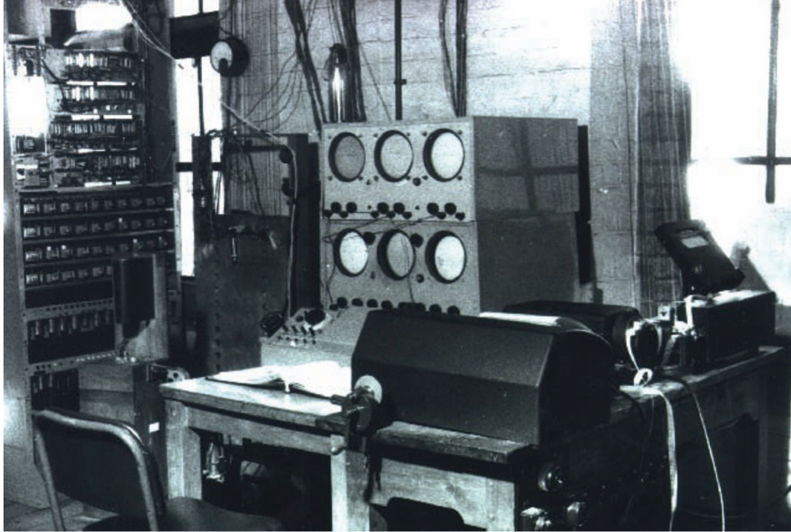


Figure 8.5 Experimental EDSAC magnetic tape rack. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/edsac99.19.jpg`)

Figure 8.6 A fanfold DEC paper tape containing a PDP-11 test program. (Reproduced by permission under the GNU Free Documentation License, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:Papertape2.jpg`)

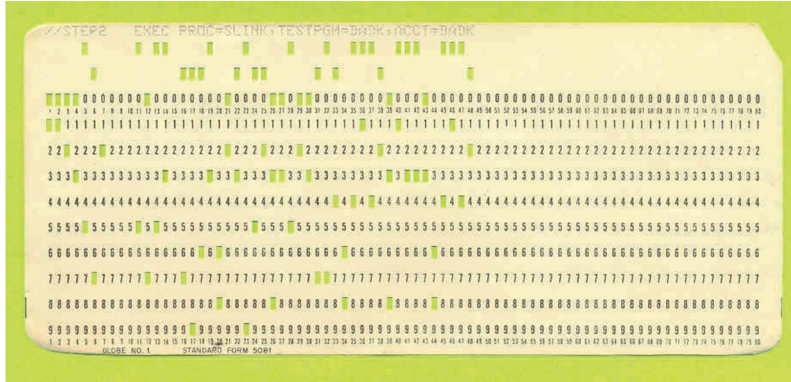


Figure 8.7 An IBM 5081 data processing card containing a line of DOS JCL program. (Reproduced by permission under the GNU Free Documentation License, photographer: Dick Kutz, source: `http://en.wikipedia.org/wiki/Image:Punch-card-5081.jpg`)

each row of holes into a binary word representing a character. Although the tape could be prepared off-line by the operator (encoding either a program of instructions or simply data values) and read at high speed, two main problems existed. Firstly, the tape was somewhat unreliable in the sense that it was prone to damage and for holes to be unreadable due to inaccuracies in the punching mechanism. Secondly, the tape mechanism was difficult to rewind; the paper tape storage device could thus be described as sequential-access only.

8.1.2 A Modern Memory Hierarchy

The previous discussion of historical memory and storage devices within the EDVAC computer was no accident; the three different levels of device live on today in modern computers although implemented using more modern technologies. The different levels form what is commonly termed the **memory hierarchy** or **storage hierarchy**, a tiered description of how the different levels compare to and interact with each other. The different levels of the memory hierarchy are managed by different parts of the system; access to the registers is performed directly by a program executing on the processor, access to a disk is usually performed by the operating system on behalf of the program during execution.

Ideally we would like very large, very fast memories but because of physical constraints this is usually impossible. To approximate the ideal, we use a combination of technologies. Roughly speaking, as one reads from top to bottom of the pyramid in Figure 8.8 the devices get farther away from the processor, are able to store more elements, are slower to access, and are less expensive to build but larger

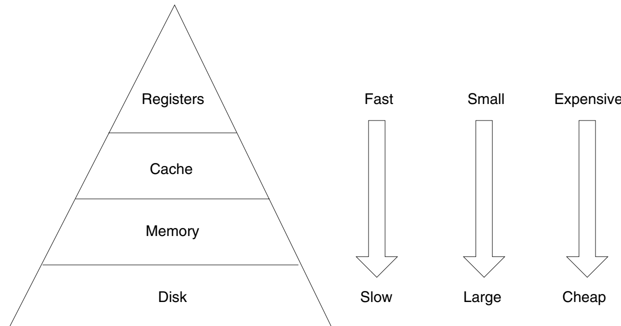


Figure 8.8 A modern memory hierarchy.

Level	1	2	3	4
Name	Registers	Cache	Memory	Disk
Size	$O(1\text{kB})$	$O(1\text{MB})$	$O(1\text{GB})$	$O(1\text{GB})$
Access Time	$O(1\text{ns})$	$O(10\text{ns})$	$O(100\text{ns})$	$O(1000\text{ns})$
Bandwidth	$O(10000\text{MB/s})$	$O(1000\text{MB/s})$	$O(100\text{MB/s})$	$O(10\text{MB/s})$
Cost	$O(100\$/\text{MB})$	$O(100\$/\text{MB})$	$O(10\$/\text{MB})$	$O(0.1\$/\text{MB})$
Managed By	Programmer	Processor	OS	OS

Table 8.1 A tabular description of typical price/performance characteristics for memory hierarchy components.

in physical size. The registers are at the top: they hold small-sized elements but are very quick to access. The next levels are the caches and main memory, which are larger but slower to access due to the fact that they are often implemented as a separate device to the processor. The next levels incorporate longer-term storage in the shape of magnetic and optical disks and tapes; these can hold the largest amount of information but are the slowest to access. The idea of course is that we should use a combination of all these devices, using the right one for the right job. Where we need to store information that need to be accessed quickly so we can feed operands to the ALU, we use a register. Where we need to store less transient information in larger quantities for the lifetime of the program, we use the main memory. Where we want to store vast amounts for information for periods of time which span when the device is powered, we use a disk or tape.

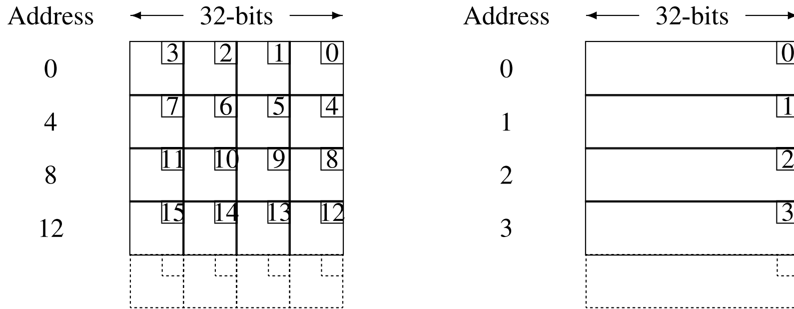


Figure 8.9 8-bit versus 32-bit memory resolution.

8.1.3 Basic Organisation and Implementation

8.1.3.1 Resolution

In reality we cannot build an infinitely large memory so the size of MEM must be set to say m elements; addresses used to access MEM are then bit-vectors of length $\lceil \log_2(m) \rceil$. Clearly there is a trade-off here: the larger the addresses we have, the more elements we can address for example. More exactly, if we have larger addresses we can either address many elements, each of which store elements of a small size, or fewer elements of a larger size. This relates to the **resolution** of the memory. Essentially the resolution dictates the size of each addressable element.

Consider an example using 10-bit addresses. We can clearly address 1024 byte sized elements in the memory, however we could also address 1024 word sized elements if we want. In the former case we have that the total memory size is 1024 bytes but, given 32-bit words, in the latter case the total size is 4096 bytes. Figure 8.9 attempts to demonstrate the difference; in each case memory elements are drawn as boxes with the address in the top-right corner, the elements are arranged in 32-bit groups with the starting address of the group to the left. In Figure 8.9a each 8-bit element has a distinct address and is individually accessible; in Figure 8.9b one can only access each 32-bit element.

Typical processors use **byte addressed** memory where each element is a byte; each byte has a unique address. So the operation $\textsc{Load}(x)$ retrieves the byte at address x and $\textsc{Store}(x,y)$ updates the byte at address x with value y.

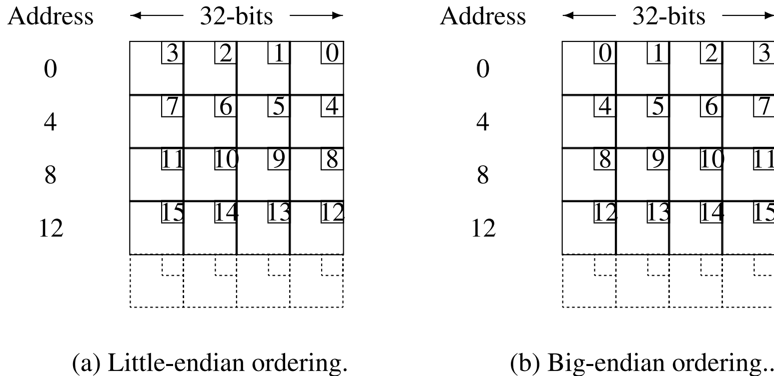


Figure 8.10 Little-endian versus big-endian memory ordering.

8.1.3.2 Endianness

Chapter 1 touched briefly on the problem of **endianness**, now we can see the problem in a more concrete setting. Suppose the memory *MEM* is byte addressable, i.e., holds 8-bit values, and we want to load a 32-bit word that starts at address 0. Clearly we would have to transfer four 8-bit bytes to construct this word; the question is, which are the most and least-significant bytes ? We could legitimately load the byte at address 0 and make this the least-significant continuing so that the byte at address 3 is the most-significant. As long as we were consistent, it would be equally as valid to do exactly the opposite: load the byte at address 0 and make this the most-significant continuing so that the byte at address 3 is the least-significant. The solution, which must be consistently enforced between load and store operations, is dictated by the endianness of the processor. The first choice means the processor is little-endian, the second choice means the processor is big-endian. Some processors can operate in both modes, one at a time, with their endian semantics selected by setting a flag in the status register when powered on. MIPS32 is an example of such a design; we will assume a little-endian system except where noted otherwise.

Again, Figure 8.10 attempts to demonstrate the difference; in each case memory elements are drawn as boxes with the address in the top-right corner, the elements are arranged in 32-bit groups with the starting address of the group to the left. In this case, Figure 8.10a has the 8-bit elements grouped in a little-endian order (the least-significant element is to the right, the most-significant to the left) while Figure 8.10b groups them in a big-endian order.

One can easily write a C function to test if a processor uses a little-endian or big-endian byte ordering; Listing 8.1 demonstrates this technique. The function places a 32-bit constant into a value shadowed by an array of four 8-bit bytes; we can then test which of the bytes in the value have been placed in which locations in the array and deduce the endian semantics as a result.

```c
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char* argv[] )
{
  unsigned char x[4];
  unsigned int  y;

  *( unsigned int* )( x ) = 0x03020100;

  y   = ( ( unsigned int )( x[ 0 ] ) <<  0 ) |
        ( ( unsigned int )( x[ 1 ] ) <<  8 ) |
        ( ( unsigned int )( x[ 2 ] ) << 16 ) |
        ( ( unsigned int )( x[ 3 ] ) << 24 ) ;

  if( y == 0x03020100 )
    printf( "little endian\n" );
  else
    printf( "big endian\n" );
}
```

Listing 8.1 A short C program to test the endianness of a processor.

8.1.3.3 Alignment

Again suppose the memory *MEM* is byte addressable, i.e., holds 8-bit values, and that we want to load a 32-bit word. In theory, we can load the four bytes that would be used to construct our word from any address. For example, we might load $0,1,2$ and 3 meaning the address of the word was 0, or load $1,2,3$ and 4 meaning the address of the word was 1. However, in practise there is some difference between these two cases.

Where an address $x \equiv 0 \pmod{w}$ we say that it is w-aligned. Less formally, in our case above if x is a multiple of four, then we say that x is word-aligned. The address 0 is word-aligned as are 4, 8, 12 and so on; the address 1 is not word-aligned. Given that the word size is typically a power-of-two, and more exactly a power-of-two number of bytes, processors often make some distinction between **aligned** and **unaligned** addresses when used in load and store operations. An aligned address can usually be accessed more quickly than an unaligned address. Firstly, this is because the byte address of an aligned word address can more easily be calculated and vice versa: one simply adds or removes bits from the least-significant end of the address. For example, when we load a 32-bit word we can add two zero bits to the least-significant end of the word address to get us the byte address; word address 0 becomes byte address 0, word address 1 becomes byte address 4 and so on. Secondly, as we will see later, if the addresses are aligned it makes it much easier to design and build other elements of the memory hierarchy such as caches.

Sometimes the distinction is explicit; an ISA will include different instructions for aligned and unaligned memory access and cause an error if, for example, an unaligned address is used with an instruction expecting an aligned address. Sometimes the issue is implicit; the ISA only has a single set of access instructions that a micro-

```
module hilevel_mem( input  wire       clk,
                    input  wire       rst,

                    input  wire [7:0] addr,
                    input  wire [7:0] data_w,
                    output wire [7:0] data_r,
                    input  wire       mrq,
                    output reg        mrs,
                    input  wire       re,
                    input  wire       we );

  reg  [7:0] data[0:3];

  always @ ( posedge mrq )
  begin
    if     ( re )
      data_r = data[ addr ];
    else if( we )
      data[ addr ] = data_w;

    #6 mrs = 1;
       @( negedge mrq );
    #6 mrs = 0;
  end

endmodule
```

Listing 8.2 A Verilog module that implements a high-level, 4-element byte-addressable memory.

architecture may execute quicker or slower depending on the alignment properties of the address.

8.1.3.4 Implementation

Accesses by the processor to the memory via the address and data buses that connect them are carried out according to a protocol we will call a **memory transaction**; this is really a special case of a more general asynchronous bus protocol. Because it is asynchronous, the protocol cannot rely on the clock to keep the processor and memory in step with each other. Instead, they use several signals to implement a **handshake**, the memory side of which is implemented in Listing 8.2, whereby each device only progresses once the other is ready:

1. The processor sets the `addr` signal to the address in memory it wants to access. If it is performing a LOAD operation, it sets the read enable or `re` signal to high; if it is performing a STORE operation, it sets the write enable or `we` signal to high. Finally, it sets the memory request or `mrq` signal to high indicating to the memory that it wants to actually perform the access.
2. As soon as the memory notices the `mrq` signal is high, it starts the access. If `re` is high, it retrieves the value at address `addr` and drives this onto the output `data_r`; if `we` is high it sets the value at address `addr` to equal the input `data_w`. Completion of the actual operation will take some time; this is termed the **memory latency**. Once the operation is completed, the memory sets

the memory response signal or mrs to high to indicate to the processor it has finished.
3. As soon as the processor notices the mrs signal is high, it sets mrq to low to complete that side of the handshake. In the case of a LOAD operation, it stores the value passed to it on data_r into wherever it needs to.
4. As soon as the memory notices that the mrq signal is low, it sets mrs to low to complete that side of the handshake. Both processor and memory are now ready to continue performing more memory accesses or other operations.

The Verilog module uses a 2-dimensional register (confusingly also called a memory), to store the values. When the memory request signal mrq is raised, the access process is invoked: if the read enable or re signal is set, the data_r output is assigned to the value in memory addressed by addr; if the write enable or we signal is set, the value in memory addressed by addr is set to the input data_w. Once the operation is completed, the module introduces an artificial delay, to model the memory latency, before raising the memory response signal mrs to indicate the end of the transaction. The behaviour of the module is shown in Figure 8.11. In this case, the test stimulus performs two STORE operations to addresses 1 and 2 with the values 10 and 20. Notice how the mrq and mrs signals are used to synchronise and enforce an order to the transaction. Two LOAD operations are then used to read addresses 0 and 1; reading address 0 gives an undefined result since we have not yet stored anything in it, reading address 1 gives the value 10 which was previously stored.

Because the access speed, or latency, of the devices lower down the memory hierarchy is slower than those at the top, the **unit of transfer** is typically bigger as well. Note that this is not strictly related to the bus width: one might have to make several bus transactions to transfer a chunk of data, the unit of transfer relates to the size of the chunks requested at one time. For example, one might expect to read a single 32-bit value at a time from a register file, but a 4kB page at a time from a disk. By reading larger chunks at a time, the number of accesses will be smaller: this somewhat compensates for the overhead associated with a single access. Although we do not cover it in any depth, this issue is related to the concept of access via a **burst mode**. The idea is that several accesses close together in time, i.e., straight after each other, can be performed quicker than the same number of accesses performed at arbitrary points in time. Again, this compensates somewhat for the overhead associated with a single access, this time by sort of treating several accesses in a row as one.

The problem with this basic module is that it is a little too abstract in the sense that we have glossed over the issue of how the 2-dimensional register is actually constructed using known low-level components: a scalable design needs more careful consideration. Figure 8.13 details, in a rough sense, a way of constructing larger memories; it shows a 4-element memory which holds n-bit words (given the dashed boxes represent the logic equivalent of continuation dots). The idea is that each row of the main body of the diagram on the right-hand side is a group of n **memory cells** that each store a 1-bit value. Since we have already described how they work, here we use D-type latches for our memory cells; we will see later how memory

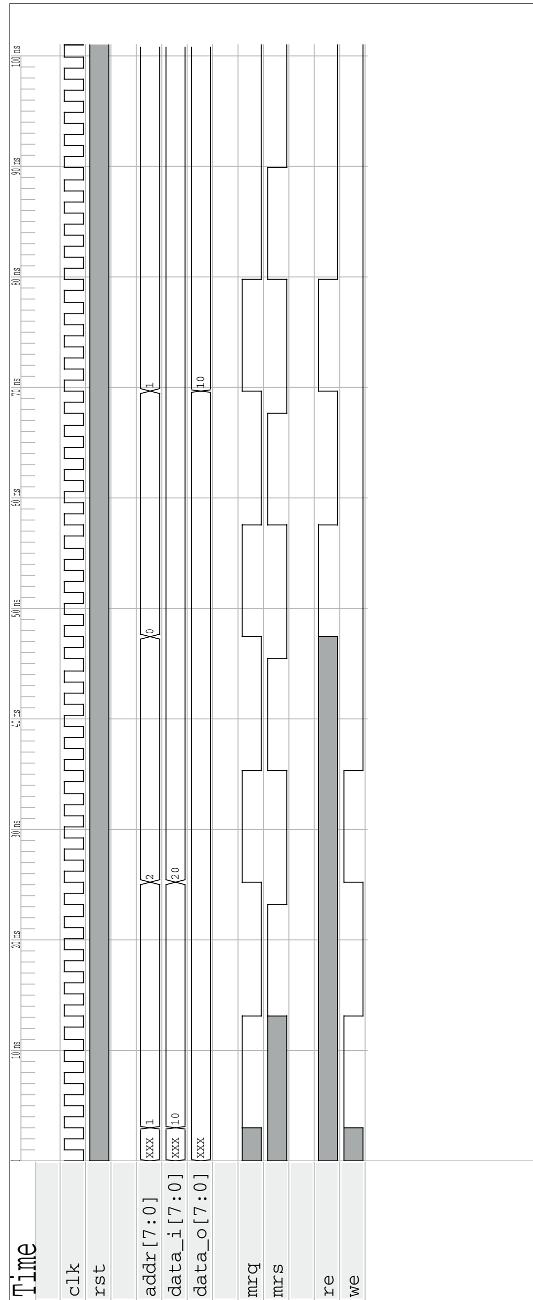


Figure 8.11 Behaviour of the high-level, 4-element, byte-addressable memory.

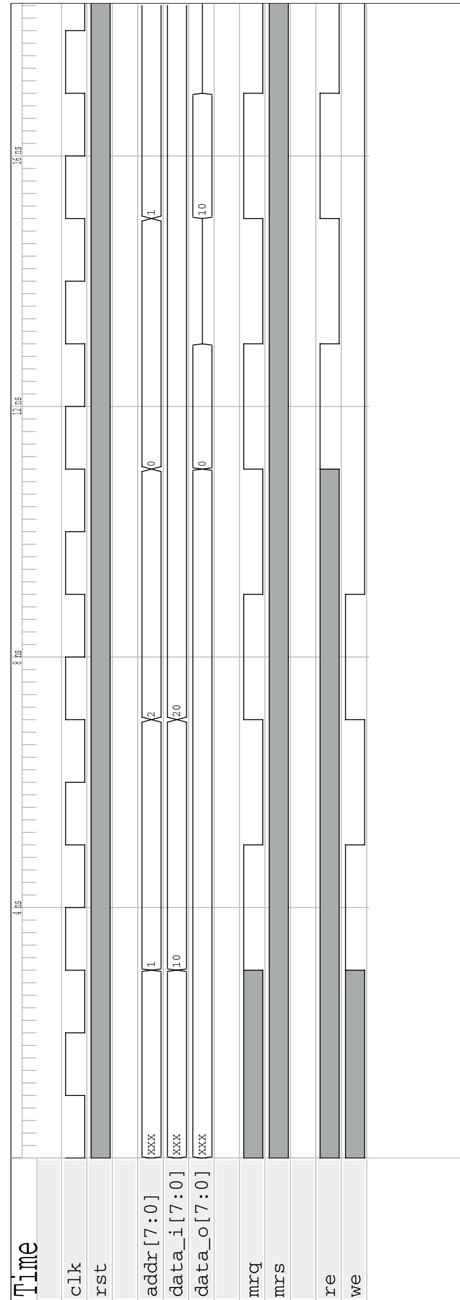


Figure 8.12 Behaviour of the low-level, 4-element, byte-addressable memory.

```
module lolevel_mem( input  wire       clk,
                    input  wire       rst,

                    input  wire [7:0] addr,
                    input  wire [7:0] data_w,
                    output wire [7:0] data_r,
                    input  wire       mrq,
                    output reg        mrs,
                    input  wire       re,
                    input  wire       we );

  wire [3:0] w0;
  wire [3:0] w1;
  wire [7:0] w2[0:3];
  wire [3:0] w3[0:7];
  wire [7:0] w4;

  assign w0[0] = ( ~addr[0] & ~addr[1] );
  assign w0[1] = (  addr[0] & ~addr[1] );
  assign w0[2] = ( ~addr[0] &  addr[1] );
  assign w0[3] = (  addr[0] &  addr[1] );

  genvar i;
  genvar j;
  genvar k;
  generate
    for( i = 0; i < 4; i = i + 1 )
    begin:gen_smem_i
      assign w1[i] = we & w0[i];

      for( j = 0; j < 8; j = j + 1 )
      begin:gen_smem_j
        dtype_ff t( .D ( data_w[j] ),
                    .Q (  w2[i][j] ),
                    .en(  w1[i]    ) );

        assign w3[j][i] = w2[i][j] & w0[i];
        assign w4[j]    =          | w3[j];
      end
    end

    for( k = 0; k < 8; k = k + 1 )
    begin:gen_smem_k
      assign data_r[k] = re ? w4[k] : 8'bZ;
    end
  endgenerate

endmodule
```

Listing 8.3 A Verilog module that implements a low-level, 4-element byte-addressable memory.

technologies replace these with more efficient cells for this specific context. The left-hand side of the diagram implements an **address decoder**: the address *addr* is used to select which row we are dealing with using a similar approach as we have seen when constructing a demultiplexer. For example, if $addr_0 = 1$ and $addr_1 = 0$, we are dealing with row 1; the horizontal **word select** signal for that row is set to 1. In this example, the memory is said to have one **read port** and one **write port** in the sense that one value can be read or one value written at a time. For some applications, it might be advantageous to have more than one read or write port: for

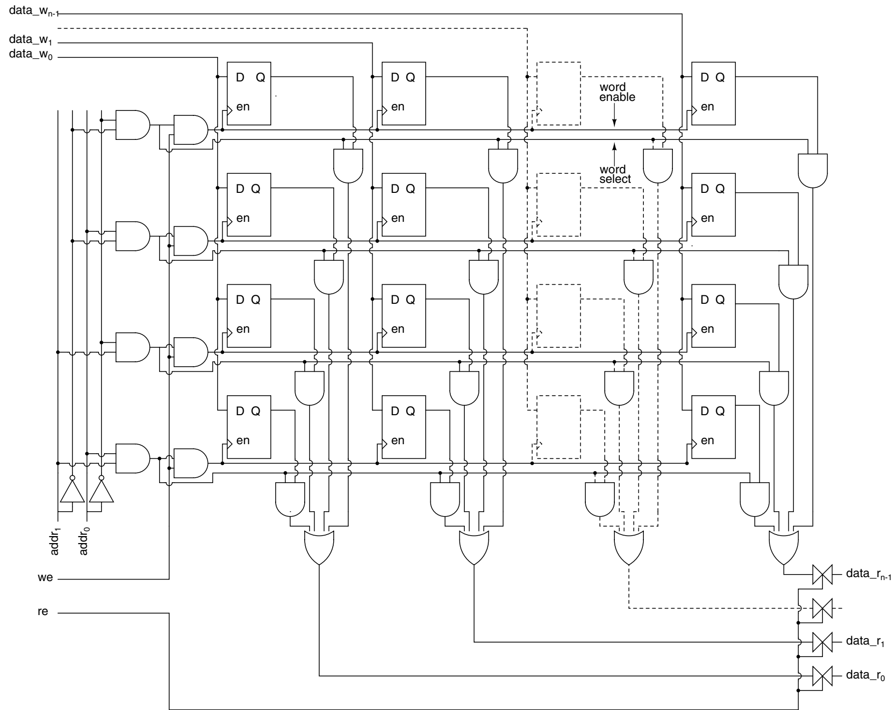


Figure 8.13 A diagram of a low-level, 4-element, *n*-bit memory device.

example, in a memory used as a register file it is common to have two read ports so two operands can be read at the same time. In such a case we can simply replicate portions of the design that are associated with reading or writing and address decoding. The key thing to realise is that this design is very regular; there is a lot of repetition of basic blocks of logic which means fabrication will be somewhat easier. The knock on effect is that the design is also scalable in the sense that we can easily add more elements or ports to the memory if we are willing to pay the associated cost in space.

When we want to read a value from a row, i.e., an address in the memory, we set the **read enable** signal to $re = 1$. Setting the **write enable** signal to $we = 0$ causes the **word enable** signal for the row we are interested in to 0 meaning that the latches do not receive a signal on their *en* inputs and a new value is not stored in the latch. Instead, the Q outputs of each latch in the row are ANDed with the word select signal for that row, so that only the output of that row is selected, and fed through to the memory output $data_r$ to complete the read. Notice that we have fed the output through 3-state enable gates: only when $re = 1$, i.e., we are actually performing a read operation, are the output wires driven with a value. This allows

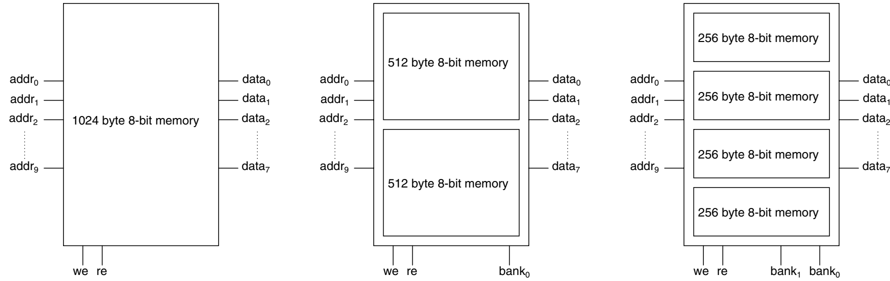


Figure 8.14 Three different organisations for a 1024 byte memory device.

the output wires (or the data bus more generally) to be shared with other devices or perhaps be used for input and output. When we want to write a value in a row we set the read enable $re = 0$ and write enable $we = 1$ which, when combined with the word select for that row, causes a pulse to be sent along the associated word enable signal into the *en* ports for the latches in that row. This causes them all to store the value passed into their D input from $data_w$. Notice that there is no handshake logic associated with this implementation: access occurs instantaneously aside from the effects of propagation delay.

The design is implemented in Listing 8.3 using the same interface as previously. The wire vector `w0` and the associated continuous assignments implement the address decoder; one can view components of `w0` as word select signals for the associated row. The rest of the design is built using a generate statement whose main job is to instantiate the D-type flip-flops that actually store values. In addition, components of `w1` act as the word enable signals for the associated row and `w4` collects values from a given column into a single signal that is fed through to the output: we use a final continuous assignment to implement the 3-state style behaviour such that when the `re` signal is low, the memory output drives no value. The associated behaviour when the device is asked to perform the same accesses as the abstract memory is shown in Figure 8.12.

8.1.4 Memory Banking

Implementing a simple memory device as described above is attractive in that increasing the size of memory simply means replicating central parts of the basic design. This is excellent from the point of view of manufacture since replication of similar basic components makes the process much easier. However, if for example you were tasked with building a byte addressable memory device capable of storing 1kB of content, there are several other possible approaches. Instead of designing a single monolithic device which stores 1024 bytes, one might for exam-

ple opt to build the device from smaller memories, say four memories each storing 256 bytes or two memories storing 512 bytes each. This concept is called **banking**, the collection of smaller memories is called a **memory bank**. To allow the user to specify which memory within the bank a particular access is targeting, extra inputs to the memory are required; Figure 8.14 describes three choices for organisation of the 1024-byte memory and adds bank selector signals $bank_0$ and $bank_1$ to address the two and four memories in the respective cases of having a banks of 512 and 256 bytes. The internal logic which uses these signals to access the component memories is not shown; clearly one would expect it to resemble some type of demultipliexer and multiplexer whereby the input and output signals are connected to the component memories depending on the values of the bank selectors.

There are several reasons why this latter approach might be better. Firstly it prevents scale from becoming a problem: although our previous design was easy to add to, a limit emerges where doing so starts to become cumbersome from the point of view of issues such as wire length and so on. Secondly, it allows designers to focus their attention more. That is, one might decide to specialise in designing and building memory devices for use by others in larger systems. By using memory banking the memory designer can concentrate on an efficient design for the basic building block and the system designer can simply use this rather than having to perform any alteration. Introduction of standard memory sizes and interfaces makes this an even more attractive proposition.

Finally, and perhaps most importantly, the use of banked memory allows us to somewhat improve memory access performance. Consider the use of our imaginary 1kB, byte addressed memory within a 32-bit processor design. Since the word size of the processor is 32 bits, loading or storing a word to or from the memory requires four accesses each loading or storing an 8-bit value. With a single monolithic memory, these accesses will probably need to be performed sequentially since the address decoder can only support one access at a time (unless there is more than one read or write port). With the banked design where we use four 256-byte memories, we can split our word into four and store 8-bits of it in each memory in the bank. As a result, since the four 8-bit parts of the word are scattered amongst different memories, they can all be accessed in parallel reducing the time required. This is sometimes called **interleaved access**. To instrument this, we need to feed the bank selector inputs with appropriate values so the right address maps to the right bank. In the case of using four 256-byte memories to build the 1024-byte device, one would drive $bank_0$ and $bank_1$ with address inputs $addr_0$ and $addr_1$ and then disregard these when using them as the actual address fed to the memory (say $addr'$). As such, this would result in the following mapping:

```
int main( int argc, char* argv[] )
{
  int A[ 1024 ];
  int B[ 1024 ];
  int C[ 1024 ];

  for( int i = 0; i < 1024; i++ ) {
    A[ i ] = B[ i ] + C[ i ];
  }
}
```

Listing 8.4 An example program to demonstrate the effects of temporal and spatial locality.

$$\begin{array}{lcll}
addr = 0000_{(2)} & \rightarrow & bank = 00_{(2)} & addr\prime = 00_{(2)} \\
addr = 0001_{(2)} & \rightarrow & bank = 01_{(2)} & addr\prime = 00_{(2)} \\
addr = 0010_{(2)} & \rightarrow & bank = 10_{(2)} & addr\prime = 00_{(2)} \\
addr = 0011_{(2)} & \rightarrow & bank = 11_{(2)} & addr\prime = 00_{(2)} \\
addr = 0100_{(2)} & \rightarrow & bank = 00_{(2)} & addr\prime = 01_{(2)} \\
addr = 0101_{(2)} & \rightarrow & bank = 01_{(2)} & addr\prime = 01_{(2)} \\
addr = 0110_{(2)} & \rightarrow & bank = 10_{(2)} & addr\prime = 01_{(2)} \\
addr = 0111_{(2)} & \rightarrow & bank = 11_{(2)} & addr\prime = 01_{(2)} \\
\vdots & & \vdots & \vdots
\end{array}$$

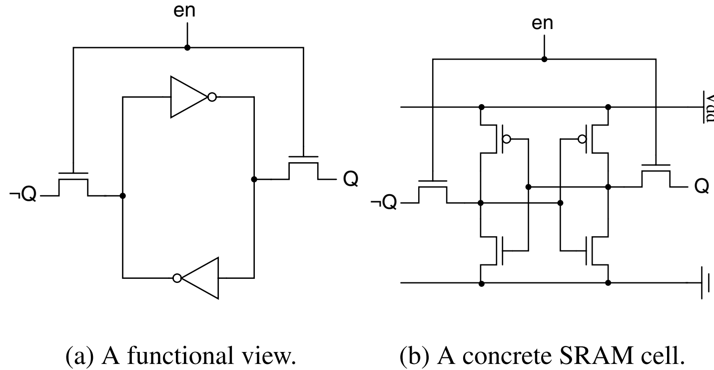
Or, in simpler terms, address 0 maps to address 0 in bank 0 while address 1 maps to address 0 in bank 1 and so on. As a result, we can access all four bytes held in address 0 within banks $0 \ldots 3$ in parallel rather than in sequence.

8.1.5 Access Locality

The principle of **locality** is used to characterise when and where accesses to a storage device are performed. The idea is that given a sequence of accesses, the device will exhibit two different forms of locality:

Definition 34. Temporal locality means that recently accessed locations are likely to be accessed again in the near future. **Spatial locality** means that two locations with addresses that are close to each other will be accessed close together in time.

Listing 8.4 demonstrates both forms of locality. Consider the instructions that make up the function; the loop performs 1024 iterations over instructions that add elements of arrays `B` and `C` together and store the result in `A`. Given one access to the instruction that performs an addition of the *i*-th elements of `B` and `C` it is likely that the instruction will be accessed again. Only when the loop terminates will this be false so 1023 times out of 1024 one access correctly predicts another, i.e., there is temporal locality in the instruction accesses. Now consider the data items in array `A` which we access sequentially. If we access the *i*-th element of `A` there is a good



(a) A functional view. (b) A concrete SRAM cell.

Figure 8.15 A diagram showing transistor level SRAM cell design.

chance we will access the $(i+1)$-th element as well; again this prediction is correct 1023 times out of 1024. Similar arguments can be made for the $(i+2)$-th or $(i+3)$-th elements although access to these given an access to the i-th element becomes increasingly unlikely; either way we clearly have spatial locality in the data accesses.

It turns out that most programs we run on a processor will exhibit both forms of access locality, they are not confined to contrived examples like the above. Thus, given knowledge of the recent past we can, with reasonable accuracy, predict which instructions are likely to be required in the near future. The idea then, is to use the memory hierarchy to improve performance by exploiting the principle of locality. We try to keep often used data and instructions, the **working set**, in the faster parts of the memory hierarchy which are typically smaller and closer to the processor.

8.2 Memory and Storage Specifics

8.2.1 Static RAM (SRAM) and Dynamic RAM (DRAM)

Static Random Access Memory (SRAM) is a method of implementing memory that uses roughly the same technique as using latches or flip-flops; we saw a small-scale example of such an approach earlier on. However, for specific use in our memory design, SRAM cells can be more efficient than a general-purpose latch or flip-flop. Figure 8.15a shows a functional view of an SRAM cell: in concept, it is built from two NOT gates which hold the value in a loop (note that this very roughly matches the approach used by the mercury delay line). The gate behaviour means that one can read the value Q on one side before it is negated so we can read $\neg Q$ on the opposite side. This functional view is implemented in Figure 8.15b using a

total of six transistors, four of which store the 1-bit value and two of which are used to access the value. Notice that the four storage transistors are arranged so that the resulting circuit has two stable states which are used to represent the values 0 and 1 (one can think of these as similar to the stable states in a latch). Access to the cell is performed via a control signal labelled *en*; this signal controls the two access transistors. To read the value stored by the cell, one pre-charges the Q and $\neg Q$ wires to 1. When the *en* signal is set to 1, either $\neg Q$ or Q is discharged depending on the stored value; if 1 is stored then $\neg Q$ is discharged, but if 0 is stored then Q is discharged. To write a value into the cell we discharge Q or $\neg Q$ and pre-charge the other, which then forces the state to be stored as required when the *en* signal is set to 1. Since SRAM cells are based on a similar design to a latch, they only retain their value for as long as they are powered. The cells have a fast access speed, in the order of 10ns, but are quite expensive in terms of the space they require. As a result SRAM is used in situations where performance is more important than size; examples include cache memory and register files.

From a simplistic point of view, **Dynamic Random Access Memory (DRAM)** makes the opposite trade-off to SRAM. Each cell is implemented using a small capacitor that is charged appropriately to store a 1-bit value. Since capacitors leak charge over time, the DRAM cell needs to be periodically refreshed; this dictates some extra, external control logic that implements the refresh cycle at appropriate intervals so that the content is only lost if the device is powered off. Where the refresh logic is built into the memory itself, we sometimes term the result **Pseudo-Static RAM** since it can be used as if it were SRAM even though internally it is DRAM.

Access to the capacitor in a DRAM cell is slow (in the order of 50ns) when compared to an SRAM cell. However, the DRAM cell has a major advantage in terms of size however: it is significantly smaller than an SRAM cell, up to a quarter of the size in fact. This means DRAM cells can be closely packed and result in larger practical memories. Examples of use include main memory which is typically orders of magnitude larger (and slower) than cache and registers. However, there is a large taxonomy of DRAM-based memory designs; all operate in roughly the same way but are typically specialised or improved to suit a particular application:

Video DRAM (VRAM) As described, our normal DRAM architecture has one read and one write port but only one of those can be used at once; one can perform a load or a store but not both simultaneously. This is unattractive in some settings, the obvious example being memory that stores data ready to be sent to some display device. VRAM solves this problem by offering two ports, one for reading data onto the display device and one for the processor to update the data read for display.

Extended Data Out (EDO) DRAM The innovation of EDO DRAM was to use pipelining techniques in respect to memory accesses. That is, an EDO DRAM would allow access to the memory to start before the previous one had entirely completed. Although we discuss pipelining in Chapter 6, this roughly means that the time taken to perform one access (the latency) is unchanged but the number of accesses completed per unit of time (the throughput) is increased.

Synchronous Dynamic RAM (SDRAM) In normal DRAM, an enable signal has to be presented to the memory which provokes it to perform some action; it has to react as quickly as it can when this signal is raised. SDRAM does away with this mode of operation by clocking the memory so that load and store operations can only be initiated on positive clock edges. Thus, the processor is able to control how quickly the memory operates and the added synchronisation allows for more complex pipelining than in simpler designs such as EDO.

Double Data Rate (DDR) SDRAM DDR SDRAM extends basic SDRAM by allowing load and store operations to be initiated on both positive and negative clock edges, thus doubling the rate at which such operations can be performed.

8.2.2 Non-volatile RAM and ROM

Most slower, secondary storage devices such as magnetic and optical disks retain their content even after they are powered off: they are non-volatile. However most faster, primary memories, implemented as SRAM or DRAM for example, only retain their content when powered. If one requires a memory which is non-volatile and yet faster than typical secondary storage devices, some form of specialist device is used. Roughly speaking, non-volatile memories are useful in applications where it is unattractive to use secondary storage devices, due to power or size for example in mobile or embedded computers, or the access latency for a secondary storage device is unacceptably long.

Consider for example the time taken for computer to cold-boot, i.e., load the operating system ready to accept input from the user after being powered on. This delay is a constant source of annoyance; Raskin [57] describes that the Canon CAT computer went to the lengths of saving an image of your screen upon logout and re-displaying it during the boot process to pretend it had completed faster than it had ! Without such trickery, the delay itself is caused largely by the time taken to access the slow secondary storage; if the operating system were stored on a non-volatile primary memory, the (legitimate) boot time could be significantly less.

Of the different approaches to non-volatile memories, major differences exist in how one approaches the construction of ROM and RAM type devices. Although they are potentially much more useful, non-volatile RAM devices are hard to construct. Although there is some recent progress in the design of devices that are close to being fast enough, the typical approach is to couple a conventional low-power RAM with a battery backup which retains the RAM content after the main power source is turned off. Clearly this is not ideal since the battery will eventually run out as well. Even so, this method is used extensively in computers to store critical information: if the clock on your computer suddenly stops working, there is a chance that the battery for this memory has failed ! Devices that cannot be written to, i.e., are read only, are easier to construct; there are roughly four evolutions of device available:

Mask-Programmed ROM The first approach essentially hard-codes the memory content in roughly the same way that a logic circuit would be created. That is, one creates a mask representing the binary content of the memory and uses the photo-lithography process to etch the image onto some substrate, coupled to some logic for accessing it, as described in Chapter 2. Once the memory has been created, the only way to alter it is to build a new one; the process is quite costly and inflexible as a result.

Programmable ROM (PROM) To avoid the problem of having to reconstruct a memory produced by the masking approach, the idea of being able to program a blank device with some image after manufacture was introduced. This process works in a similar way to the PLA device described in Chapter 2 whereby fuses are burnt out or left alone so as to produce the binary content of the memory. Once the image is created, changes are not possible.

Erasable PROM (EPROM) Even PROM type devices are somewhat inflexible in that as well as being read only when being used, they are read only forever. This leads to a slightly different definition of ROM: we are interested in a device that might be read only while in use as a memory, but is writable (or at least erasable to form a blank device) at other times. This is the goal of EPROM memory.

The first EPROM devices, invented by engineer Dov Frohman in 1971, were erasable via exposure to strong ultraviolet light due to their use of special types of transistor. The memory chip was equipped with a transparent window through which a special-purpose machine performed the exposure: the erasure process took about half an hour, the device was then able to retain the memory content for about ten years. It was common for chips to be equipped with a foil cover to stop erasure by sunlight !

Electrically Erasable PROM (EEPROM) The process of erasure in EPROM designs is somewhat cumbersome given the chip needs be taken out of the surrounding circuit and operated on using extra equipment. EEPROM developed on these basic ideas by allowing erasure via an electrical interface and selective erasure so that the memory could be partly updated rather than totally blanked. The trade-off for this is that unlike EPROM that can typically be erased an unlimited number of times most EEPROMs have a limit due to degradation of storage medium, typically of ten thousand or so times.

Flash memory is perhaps the most modern example of the EEPROM; they are fast enough and can be manufactured with enough capacity to replace a read only magnetic disk for example. These types of memory device are commonly used in digital cameras for example.

8.2.3 Magnetic Disks

A magnetic disk is a collection of circular **platters** each of whose surface is coated with a magnetisable material. The idea in terms of storing information on the plat-

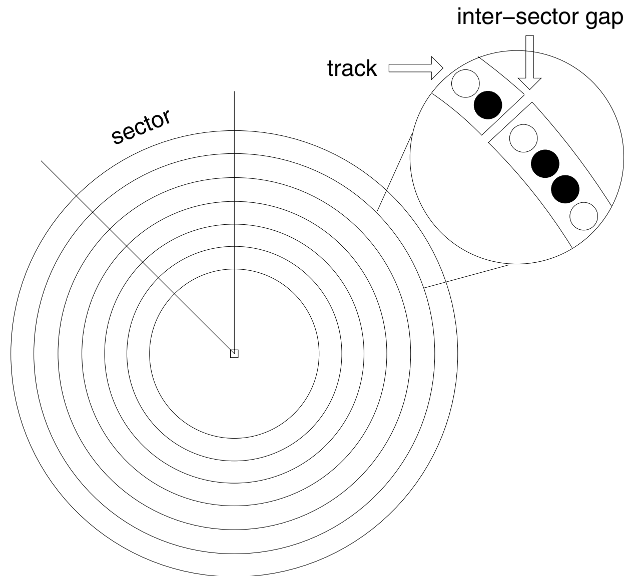


Figure 8.16 A diagram showing tracks and sectors in a typical hard disk format.

ters is that a **disk head** is positioned at a given point above the platter and used to magnetise the surrounding region to encode a single binary digit: the magnetic field is made to align one way to represent 1 and the other way to represent 0. When the information is to be read back, the disk head is positioned in the same place. The magnetic field on the platter induces a current in the disk head which is relayed back as binary data.

To provide some consistency and interoperability between disks and the devices that control them, a standard way of sectioning up the platters is employed. This is the **low-level format** of the disk. Firstly, a disk may consist of a number of platters; each platter has a dedicated disk head. A given platter is divided up into a number of **tracks** which are arranged in concentric circles on the platter surface; tracks in the middle of the platter are shorter than those at the edge. The set of tracks, over all platters, at a given position is called a **cylinder**. Each track is divided up into a number of **sectors** which roughly look like arcs of each concentric circle. Between each sector is a small blank space called the **inter-sector gap**. The end of each sector is punctuated by error correction information that is used to ensure imperfections in the process can be catered for somewhat invisibly for whoever is using the device. Figure 8.16 details these features diagrammatically.

In this context, one can think of some function of platter, track and sector as being the address we are accessing. The access latency is determined by two aspects: the **seek time** which determines how quickly the disk head can be moved to the right

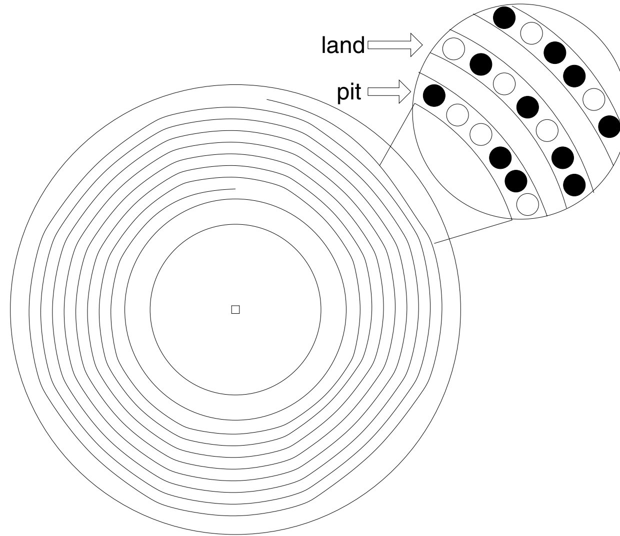


Figure 8.17 A diagram showing tracks and sectors in a typical CDROM format.

track, and the **rotational latency** which determines how quickly the right sector arrives under the disk head as the disk spins. With so-called hard disks the disk head floats above the platter on a cushion of air and can move very quickly, with floppy disks the disk head is actually in contact with the platter. This can cause the disk head and platter to degrade over time; the reliability of floppy disks is questionable as a result.

8.2.4 Optical Disks

Optical disks take a somewhat similar approach to magnetic disks in the sense that information is encoded on the surface of a (single) platter. Significant differences exist, however, in the way the information is encoded and organised. Instead of magnetic regions information is encoded, as binary digits, using a series of physical indentations in the surface; the indentations are termed **pits**, the untouched regions between pits are termed **lands**. These features are used to encode 1 and 0, although not directly but as part of some more complicated scheme (i.e., a pit or land does not directly encode a 1 or 0). Information is read by positioning a laser, the **read head**, over the relevant region and firing it at the platter. The nature of the surface is such that although both pit and land will reflect the laser, they are such that the reflection has its phase shifted depending on what it is fired at: this change in phase

is detected by a device that monitors the reflected beam and is converted into binary data.

The information is encoded as a series of **sectors** divided into **frames**; the information forms a single spiral track on the platter. Header and error correction information are appended to each frame. Reading the frames is complicated by the fact that the spiral is tighter at the middle of the platter than at the edge. Using a **Constant Linear Velocity (CLV)** scheme means the platter speed is variable but the pits pass at a constant speed: however, the drive mechanism needs to be quite complex as a result. In contrast, using a **Constant Angular Velocity (CAV)** scheme means the platter speed is constant but the pits pass at a variable rate which leads to faster transfer when reading from the edge than the middle.

Common forms of optical disks include the **Compact Disc Read Only Memory (CDROM)** and **Digital Versatile Disc (DVD)** both of which are read-only devices; their content is created using a one-off process called **mastering** whereby the pits are permanently etched on the platter. The so-called **Rainbow Book** standards dictate how information in the frames is interpreted: **Red Book** specifies the format for audio CDs, **White Book** specifies the format of **Video CDs (VCDs)**, and **Yellow Book** specifies the general CDROM format. Read-write versions, for example **Compact Disc Read-Write (CDRW)**, are possible using a coating on the platter. The coating can be selectively heated using an infra-red laser to crystallise or anneal the platter and create or remove pits. This process can repeated around a thousand times before the platter surface degrades and the encoding is rendered permanent.

8.2.5 Error Correction

Digital components sometimes fail; unlike failures in a processor which would typically result in catastrophic failure, failures in memory or a storage device can be somewhat more subtle. For example, consider if a single memory cell were to fail: this would produce errors so that values transmitted from memory to processor would be erroneous in just one bit. This might not cause a program to crash, it might just produce slightly odd results. In addition, both memory and storage devices are typically implemented as separate devices to the processor so errors might also exist in transmission rather than storage of values. We can categorise either type of error as either **permanent** or **transient**. Permanent errors are always present, they behave in the same way at all times. Transient errors only occur occasionally so that, for example, a read from memory might be completed without error at one point in time but produce an erroneous value at another. Sometimes these are termed **hard errors** and **soft errors** respectively.

Techniques for **error detection** and **error correction** partly accommodate the inevitable occurrence of errors in storage and communication devices. In our discussion we will refer to transmission of values; it is important to see that this is as relevant to retrieval of values from a storage device by the processor as it is to communication of values along some medium. Error detection hopes to identify when

an error has occurred. This can either result in the processor being signalled so that for example it can repeat an access in hope of a subsequent success, or simply to highlight erroneous memory so that it can be replaced. Error correction takes this one step further so that errors can be corrected and execution can continue without further intervention; this helps to accommodate the erroneous memory without need for immediate replacement. Roughly speaking both methods add redundant information to the value held in memory so that errors in the value can be detected and potentially corrected; the more redundant information that is added, the more errors can be detected or corrected.

8.2.5.1 Parity Codes

Probably the most basic form of error detection is the **parity code**. The basic idea is that one should count the number of bits set to 1 in x, that is, the Hamming weight of x, and call this total t. We then append a bit p to x based on the value of t to get a new value $x' = x : p$ which we call the **code word**. To use **even parity** the bit p should be 1 if t is odd, thus making the total number of bits set to 1 in the code word x' even. To use **odd parity** the reverse is true; the bit p should be 1 if t is even, thus making the total number of bits set to 1 in x' odd. As an example, consider the value $x = 1001_{(2)} = 9_{(10)}$, since there are two bits set to 1 in x, we compute x' using even and odd parity as

$$\begin{aligned} \textsc{Even-Parity}(x) &= 10010 \\ \textsc{Odd-Parity}(x) &= 10011 \end{aligned}$$

with the parity bit appended in the least-significant position in this case.

Regardless of whether even or odd parity was used, the code word x' can now be checked for errors using the extra bit. Specifically, if an odd number of errors occur in transmitting x' then we can detect this because the parity bit will differ from that expected. For example, imagine we construct the even parity-based encoding of $x = 1001_{(2)}$ as $x' = 10010$. Now imagine that instead of transmitting the code word $x' = 10010$ there is some error which flips one bit and causes us to transmit $y' = 11010$ instead. After extracting the parity bit $p' = 0$ we can count the number of bits set to 1 in y'; this time there are three. Since we are using an even parity scheme we would expect the parity bit to be 1 so that the overall number of bits set to 1 is even. But $p' = 0$ so there must have been some error.

The basic parity code is easy and inexpensive to implement; it just requires a few extra XOR gates to compute and check the parity bit. However, there are two main problems. Firstly we cannot tell where the error occurred or how to correct it, the value must be resent and we must hope that the error is transient; secondly if an even number of bits are flipped, then a single parity bit can no longer detect the error. One can address the second problem by using a more involved scheme such as the **Cyclic Redundancy Check (CRC)**; the first problem requires some more ingenuity.

8.2.5.2 Hamming Codes

The **Hamming Code**, named after their inventor Richard Hamming, improves on simple error detection mechanisms by allowing the correction of errors once they are detected. This required that more than one bit be appended to the value; Hamming used the notation (n,m)-code to denote a scheme where there were n bits in the code word of which m where actually the original value and $n-m$ were added afterwards to instrument error detection and correction.

The general Hamming Code is somewhat involved to describe, so we concentrate on the specific and common example of a $(7,4)$-code introduced in 1950. The basic idea is to have several parity bits, in this case three. In a code word under this scheme, all bits in the code word whose index i implies $i+1$ is a power-of-two are parity bits; this means bits x'_0, x'_1 and x'_3, denoted p_0, p_1 and p_3, are parity bits since $0+1=2^0$, $1+1=2^1$ and $3+1=2^2$. The four other bits represent the original value. Each parity bit deals with a sub-set of the code word: p_0 deals with the parity of x'_2,x'_4 and x'_6; p_1 deals with the parity of x'_2,x'_5 and x'_6; p_4 deals with the parity of x'_4,x'_5 and x'_6. Adopting a vector notation, our code word is thus

$$x' = (p_0, p_1, x_0, p_3, x_1, x_2, x_3)$$

where x_i are from the original value and p_j are our parity bits as described above and calculated as

$$\begin{aligned} p_0 &= x'_2 \oplus x'_4 \oplus x'_6 \\ &= x_0 \oplus x_1 \oplus x_3 \\ p_1 &= x'_2 \oplus x'_5 \oplus x'_6 \\ &= x_0 \oplus x_2 \oplus x_3 \\ p_3 &= x'_4 \oplus x'_5 \oplus x'_6 \\ &= x_1 \oplus x_2 \oplus x_3. \end{aligned}$$

Encoding the value $x = 1001_{(2)} = 9_{(10)}$ using even parity would result in the bits $p_0 = 0$, $p_1 = 0$ and $p_3 = 1$ so that the code would be computed as $x' = 1001100$. An error in any one bit of the original value embedded in this code would affect the parity bits; in fact, which bits they affect will unique in the sense that by checking all three we can determine the location of the error. Consider flipping x_0 which corresponds to x'_2 so that instead of transmitting the code word $x' = 1001100$ we transmit $y' = 1001000$. Recalculating the parity bits we find that

$$\begin{aligned} p'_0 &= y'_2 \oplus y'_4 \oplus y'_6 \\ &= 1 \\ p'_1 &= y'_2 \oplus y'_5 \oplus y'_6 \\ &= 1 \\ p'_3 &= y'_4 \oplus y'_5 \oplus y'_6 \\ &= 1. \end{aligned}$$

Since $p'_3 = p_3$ but $p'_0 \neq p_0$ and $p'_1 \neq p_1$ there is an error. To find where the error is, we form a vector from the recalculated parity bits

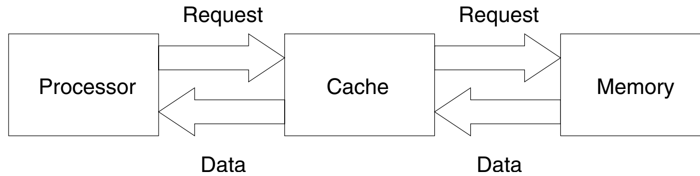


Figure 8.18 A basic block diagram of cache operation.

$$k = (p'_0 \neq p_0, p'_1 \neq p_1, p'_3 \neq p_3)$$

and consider this as an integer; the error is then at bit $k-1$. That is,

$$k = (1,1,0)$$

so we have $k = 011_{(2)} = 3_{(10)}$ and the error is detected in bit $k-1=2$. This error can now be corrected by flipping y'_2 to get the original, error-free code word x' back again. Consider another example, this time flipping bit x_3 which corresponds to x'_6 so that we transmit $y' = 0001100$. We compute

$$\begin{aligned} p'_0 &= y'_2 \oplus y'_4 \oplus y'_6 \\ &= 1 \\ p'_1 &= y'_2 \oplus y'_5 \oplus y'_6 \\ &= 1 \\ p'_3 &= y'_4 \oplus y'_5 \oplus y'_6 \\ &= 0 \end{aligned}$$

and find that $p'_0 \neq p_0$, $p'_1 \neq p_1$ and $p'_3 = p_3$. Writing the vector of parity bits out we find that $k = 111$ so that the error is in bit $k-1=6$. This can now be corrected by flipping bit y'_6 to get back to compute the correct code word x'.

One can show that this $(7,4)$-code can correct 1-bit errors, or detect but not correct 2-bit errors. Although the length of the code word is larger we do not need to retransmit very often because unless there are very many errors, they can be fixed automatically.

8.3 Basic Cache Memories

A **cache** is a small area of fast RAM and associated control logic which is placed between the processor and main memory as shown in Figure 8.18. The cache is typically invisible to the programmer; the interface between processor and main memory is unchanged and from a functional point of view, and accesses are serviced as if the cache was not there. The basic idea is that since the cache is smaller

than main memory, it stores a sub-set of the memory content. However, since the cache can be implemented using a faster technology than main memory, accesses to content in the cache are faster than those which are not. As a result of locality in the incoming address stream, the cache reduces the load on the rest of the memory hierarchy because it can be managed to hold the current working set of data and instructions.

The cache is typically organised as a number of **cache lines**, each of which comprises a number of **sub-words** that are used to store contiguous addresses from main memory:

Definition 35. A cache C is constructed from C_{lines} cache lines. The i-th cache line, which we write as $C[i]$, has three fields:

- $C[i]_{tag}$ is a **tag** used as an identifier for the content within the cache line.
- $C[i]_{valid}$ indicates whether the content is valid or not (i.e., if there is content in the cache line).
- $C[i]_{data}$ is the actual cache line content which consists of C_{words} sub-words each of whose size matches the resolution of the memory. Where appropriate we refer to the j-th sub-word within the i-th cache line as $C[i]_{data[j]}$.

To make life easier, we constrain C_{lines} and C_{words} so they can only take values which are powers-of-two; given a total cache size of 2^n sub-words for some n and a line size of $C_{words} = 2^m$ for some m, the number of lines is $C_{lines} = 2^{n-m}$. For byte addressed memory and a total cache size of $2^{10} = 1024$ bytes so that $n = 10$ for example, there are many different ways to organise things. We might select any one of the following three options:

$$
\begin{aligned}
C_{words} &= 2^2 & C_{words} &= 2^5 & C_{words} &= 2^8 \\
&= 4 & &= 32 & &= 256 \\
C_{lines} &= 2^{10-2} & C_{lines} &= 2^{10-5} & C_{lines} &= 2^{10-8} \\
&= 256 & &= 32 & &= 4
\end{aligned}
$$

Accesses that are serviced by the cache are termed **cache-hits** and are completed very quickly; accesses that are not held by the cache are termed **cache-misses** and take much longer to complete since main memory must be accessed. When the cache accesses main memory, it is said to have **fetched** data: typically an entire line is fetched in one go. Selecting the best cache organisation is therefore a trade-off: more lines typically mean we can better exploit temporal locality and are quicker to fetch, longer lines mean we can better exploit spatial locality but take more time to fetch. Standard literature categorises cache misses into three classes [29]:

Definition 36. Compulsory misses are unavoidable misses caused by the first access to an element in memory. **Capacity misses** are misses that are independent of all factors aside from the size of the cache. **Conflict misses** are misses where one element replaces another in the cache.

Since the principle of locality guarantees we should get more cache-hits than cache-misses, performance of the average case program is improved. Common measures

of cache effectiveness are the **hit ratio** and **miss ratio** which are calculated as

$$\begin{aligned}\textsc{Hit-Ratio} &= \frac{\textsc{Total-Hits}}{\textsc{Total-Accesses}}\\ \textsc{Miss-Ratio} &= \frac{\textsc{Total-Misses}}{\textsc{Total-Accesses}}\\ &= 1 - \textsc{Hit-Ratio}\end{aligned}$$

It is quite important to stress that performance of only average case programs is improved. That is, if we have a program that behaves somewhat oddly, for example it accesses data in a manner which does not respect the principle of locality, the cache might not be effectual at all.

Beyond the hit and miss ratios, an important performance metric is the idea of **mean access time**, the average length of time for a memory access to complete. If T_{cache} and T_{memory} denote the time required to perform an access to the cache and memory respectively, we calculate mean access time as

$$T_{cache} + T_{memory} \cdot \textsc{Miss-Ratio}$$

This is easy to see: to access the memory we first check in the cache and if the value is not present, we need to access the memory. So we pay the price of accessing the cache plus the price of access to memory for those proportion of accesses that provoke cache misses. As the miss ratio tends to 0, mean access time tends to T_{cache}. As the miss ratio tends to 1, mean access time tends to $T_{cache} + T_{memory}$. The principle of locality should ensure the miss ratio is close to 0; if it does not then the addition of a cache can actually slow down the system ! To get any meaningful speed-up, we need that

$$\textsc{Hit-Ratio} > \frac{T_{cache}}{T_{memory}}.$$

To design a real cache using all this theory, we need to consider several key issues:

Addressing Policy Given an access by the processor to an address in memory, how do we decide where this address should map to in the cache given that it only stores a sub-set of the memory content ? Application of this mapping is usually termed **address translation**.

Fetch Policy Given an access by the processor to an address in memory such that the location is not resident in the cache, how and where does one load data from main memory into the cache to satisfy the request ? Furthermore, how does one cope with the fact that data may already be resident in cache location we want to fetch new data into ?

Write Policy Given an access by the processor to an address in memory, how does the cache interact with the memory so that the processor's view of memory is consistent when the access is a store ?

In the next sections we explain different fetch and write policies that are common between most cache architectures. We then introduce three different cache archi-

tectures that use different addressing policies to implement different performance trade-offs. Although there can be some differences (e.g., we typically only load instructions rather than load and store them), since data and instructions are essentially the same data and instruction caches are largely the same as well: to make things easier we concentrate only on data caches.

8.3.1 Fetch Policy

When the processor tries to access an address in memory through the cache and the cache does not contain the corresponding data, it is **fetched** from memory into the cache; once it is located in a cache line it is said to be **resident** in the cache. In order to take advantage of the relative speeds and transfer sizes, the cache typically fetches an entire cache line at once. That is, if the processor tries to access address x, then the addresses around x that map into the same line will be fetched into the cache as well. The idea here is to take advantage of any locality: if x has been accessed then it is probable that the addresses close to x will be accessed soon; if they are, then the line will already be resident and a cache hit will occur rather than the cache miss that resulted from the access to x.

The address x is mapped to a cache line by the address translation mechanism. Once we know which line to fetch the corresponding data into, we need to check if there is already data resident in that line. Remember that the cache only holds a sub-set of the memory content so the resident data need not be that relating to x: if it were, we would have found it, signalled a cache hit and not needed to fetch the line for x at all ! So if there is already data resident, we need to **evict** it from the cache and replace it with the data we do want. If the data has not been altered by a write, this process is simply a case of overwriting what is already there. If it has been altered, we have a number of options which are discussed later when examining write policy.

8.3.1.1 Fetch Ordering

Consider the following example situation where we have a 1024-byte cache with 256 lines and 4 sub-words per-line. The processor issues the cache a load request for address 2. Suppose the address translation maps address 2 onto cache line 0. The line will hold data for addresses $0, 1, 2$ and 3 but does not currently hold this data, we need to fetch it; the cache needs to load data at addresses $0, 1, 2$ and 3 in main memory to fill line 0. The question is which order should we load these addresses; there are two main choices:

Critical Word First In this scheme we would load the words in the order $2, 3, 0, 1$. That is, the address we are looking for is loaded first, then the others in cyclic order.

Natural Word First In this scheme we would load the words in the order $0, 1, 2, 3$. That is, the first address in the cache line is loaded first followed by all the others.

Clearly each has some advantages and disadvantages to both schemes. Loading the critical word first means the cache can complete the request from the processor faster since it does not need to wait for other loads from main memory. That is, once it loads the value at address 2 it can return this to the processor and carry on with the others behind the scenes; the processor can hence continue execution earlier. On the other hand, this requires a more complex operational mechanism; loading the natural word first is much easier and leads to a simpler design.

8.3.1.2 Pre-fetching

We have already seen that a cache-hit can only occur if the corresponding data is already resident in the cache; if it is not, a cache-miss occurs and we have to load it from main memory. One method of trying to reduce the number of cache-misses is, given previous a list of previous accesses, to guess which address will be accessed next and fetch this into the cache speculatively. If the guess is right, then a cache-hit occurs and execution continues without the penalty associated with a cache-miss. If the guess is wrong, we do no worse than the cache-miss that would have occurred anyway. This technique is called pre-fetching and is typically realised by a second hardware device that works alongside the cache.

This seems an ideal solution and, if we had a good enough guessing strategy, it is indeed ideal in theory. In practise, such a guessing strategy is hard to construct for general programs even given their access locality patterns. Furthermore, aggressive pre-fetching introduces the new problem of **cache pollution**. For example, consider the case where we pre-fetch some data based on our guess and this fetch evicts data that would have been useful in the near future. We are essentially polluting the cache with useless data. Pre-fetch systems need to carefully balance their operation as a result; generally it is only a good idea to employ pre-fetch if the access pattern is very regular and hence easy to predict with high accuracy.

8.3.2 Write Policy

So far we have ignored what happens when the processor wants to store data into memory: the processor issues the store operation as usual, but what should the cache do ? There are two main problems we need to tackle. Firstly, if we update the data in the cache, what happens to the data in memory ? Secondly, what happens if the data we want to update is not even in the cache at all ? No matter what solution is employed the *key* requirement is that we maintain **consistency** between what is stored in the cache and what is stored in memory. That is, it should be impossible to get confused about the value held in a particular memory location.

There are two main options as regards solving the problem of maintaining consistency between the cache and memory content:

Write-Through When a store is issued by the processor, a write-through cache updates the data in the cache, potentially fetching data into the cache if it is not already present, but also sends the write through to the main memory. This is the simplest choice of policy: since the cache and memory content is always the same, it is always consistent. However, one needs to be careful that the cache still improves performance by eliminating unnecessary accesses to main memory. For example, if we perform a number of writes to the same address, there is no need to keep updating the memory.

Write-Back When a store is issued by the processor, a write-back cache updates the data in the cache but does not immediately write the data through to main memory. The hardware for a write-back cache is more complex, it has to remember which lines are inconsistent with main memory; these are said to be **dirty**. The cache needs to include an additional field for each line, say $C[i]_{dirty}$, which determines whether the content is dirty or not. When a line is evicted, the associated dirty flag is checked to see if it should be written to main memory. This scheme removes unnecessary memory traffic since now we only write to main memory when we really have to.

When the processor performs a load operation and the corresponding cache line is not resident, it is read into the cache. When a store operation is executed, this implication is not necessarily true. If the cache line is resident the store can update local data in the cache; however, if the cache line is not resident we cannot do this: to preserve consistency we need to take into account that there will be other memory locations in the same line. For example, say an example cache has four sub-words per-line and addresses $0, 1, 2$ and 3 map to line 0. If a store is executed using address 2 and the line is not resident, we cannot just update the cache line because what values would the sub-words for addresses $0, 1$ and 3 hold: the cache and memory would be inconsistent. Hence we have two main choices:

Allocate-on-Write Follow the implication of a read, that the data will be accessed again in the near future, and fetch the cache line into the cache before completing the write operation on the resident line.

No Allocate-on-Write Avoid fetching the cache line into the cache and pass the store directly onto main memory; thus cache lines are only fetched into the cache by load operations.

8.3.3 Direct-Mapped Caches

To construct a cache, we need an address translation mechanism that takes an incoming address that results from a load or store operation, and maps it onto the specific sub-word within a cache line which holds the associated data. A **direct-mapped cache** takes a simple approach to this by operating such that each 2^n-th

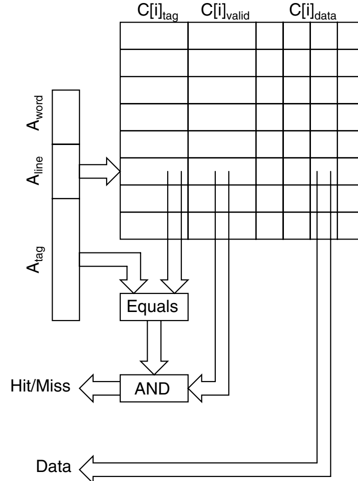


Figure 8.19 A block diagram of a direct-mapped cache.

address in memory maps to the same sub-word in the cache. The address translation mechanism calculates two components which are used to access the cache content:

$$\begin{aligned} A_{word} &= A \bmod C_{words} \\ A_{line} &= \lfloor A/C_{words} \rfloor \bmod C_{lines} \end{aligned}$$

The value A_{line} identifies the cache line for address A, while the value A_{word} identifies the sub-word within that cache line. Since both C_{lines} and C_{words} are selected to be powers-of-two, calculating these components is not costly. Given a 32-bit address and taking $C_{lines} = 2^3 = 8$ and $C_{words} = 2^2 = 4$ so the total cache size is 32 bytes, this amounts to taking the least-significant two bits of A as A_{word} and the next least-significant three bits as A_{line}. For example, given the address

$$A = 000000000000000000000000001\underbrace{000}_{A_{line}}\underbrace{10}_{A_{word}}{}_{(2)} = 34_{(10)}$$

we calculate $A_{line} = 000_{(2)} = 0_{(10)}$ and $A_{word} = 10_{(2)} = 2_{(10)}$.

However, since a given cache can only hold a sub-set of memory content, it follows that there is a many-to-one mapping of memory addresses to cache sub-words. For example, setting $A = 34_{(10)}$ or $A = 66_{(10)}$ gives the same values for A_{word} and A_{line} in both cases: we need a way to distinguish the cases from each other. To achieve this, each cache line holds a tag; we need to calculate

$$A_{tag} = \lfloor \lfloor A/C_{words} \rfloor / C_{lines} \rfloor$$

	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$		0				
$i=1$		0				
$i=2$		0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a) Cache is initially empty.

	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3
$i=1$		0				
$i=2$		0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b) Access to address 1 misses, line 0 is filled.

	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	1	1	32	33	34	35
$i=1$		0				
$i=2$		0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(c) Access to address 34 misses, line 0 evicted then re-filled; access to address 35 then hits in line 0.

	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	1	1	32	33	34	35
$i=1$	1	1	36	37	38	39
$i=2$		0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(d) Access to address 36 misses, line 1 is filled; access to address 37 then hits in line 1.

	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3
$i=1$	1	1	36	37	38	39
$i=2$		0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(e) Access to address 1 misses, line 0 evicted then re-filled; access to addresses 38 and 39 then hit in line 1.

	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3
$i=1$	1	1	36	37	38	39
$i=2$	1	1	40	41	42	43
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(f) Access to address 40 misses, line 2 is filled; access to address 41 then hits in line 2.

Figure 8.20 An example of direct-mapped cache operation.

to check that the cache line content is actually what we expected. Again, since C_{lines} and C_{words} are selected to be powers-of-two, calculating A_{tag} amounts to simply taking the remaining twenty seven bits of the 32-bit address, i.e.,

$$A = \underbrace{000000000000000000000000001}_{A_{tag}}\underbrace{000}_{A_{line}}\underbrace{10}_{A_{word}}{}_{(2)} = 34_{(10)}.$$

Now we find that setting $A = 34_{(10)}$ results in

$$A_{tag} = 000000000000000000000000001_{(2)} = 1_{(10)},$$

but if we instead set $A = 66_{(10)}$, then

$$A_{tag} = 000000000000000000000000010_{(2)} = 2_{(10)},$$

i.e., we can tell when the cache line contains content relating to one address versus the other. Note that in theory we could use the entire address A as the tag since this would obviously permit distinguishability. However, this is wasteful: the least-significant bits are implicit, or the same, for a given line so by not storing this portion we reduce the amount of logic required to implement the cache.

Figure 8.19 shows a block diagram of a typical direct-mapped cache design. The basic idea is that given an address A, the cache calculates A_{word}, A_{line} and A_{tag}. It then inspects the value

$$C[A_{line}]_{valid}.$$

If the cache line does not contain valid content, the sub-word we want is not resident and a cache-miss occurs. However, if the cache line does contain valid content, the cache inspects the value

$$C[A_{line}]_{tag}$$

and compares this to A_{tag}. If the tags do not match, the sub-word we want is not resident and a cache-miss occurs. However, if the tags do match, then a cache-hit occurs and the sub-word we want is resident in

$$C[A_{line}]_{data[A_{word}]}.$$

Imagine we continue to consider the example cache from above, and feed it a stream of memory accesses whose addresses are

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

With our address translation scheme, this produces the cache accesses

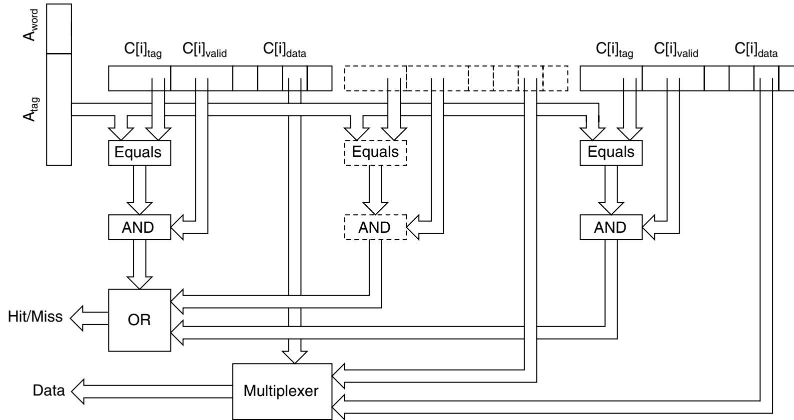


Figure 8.21 A block diagram of a fully-associative cache.

$$
\begin{array}{lllll}
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 38 & A_{word} = 2, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{hit}\,.
\end{array}
$$

The sequence produces five cache hits and five cache misses; Figure 8.20 details the sequence graphically. The first access to address 1 is a compulsory cache-miss since this is the first time we have used this address; the line for address 1 is then fetched into the cache and the correct value at offset 1 is returned as the result. The second access to address 34 maps to the same line, i.e., line 0, as the address 1 whose corresponding line is already resident. This is a conflict cache-miss so the line for address 1 is evicted and replaced by the line for address 34 that is fetched from memory; notice that these are distinguishable since they have different tags. The third access to address 35 provokes a cache-hit since the corresponding line, fetched when we accessed address 34, is resident.

8.3.4 Fully-Associative Caches

A direct-mapped cache can only ever have one choice of line per address; this can lead to what is called **cache interference** or **cache contention**. This was evident in a previous example where addresses 1 and 34 interfered when they were both mapped to line 0. When the resulting eviction happens repeatedly, we say that the access stream thrashes the cache, a situation where performance is significantly degraded.

One approach to solving this problem is to use the idea of **associativity**. An associative cache allows any memory location to map into one of many cache lines; a **fully-associative cache** allows a memory location to map to any cache line. Since the required data could be anywhere in the cache, searching for it is more complex and the hardware realisation is typically larger and more power hungry. However, associative caches can exhibit a lower miss ratio than a direct-mapped cache. Again considering the previous example, if addresses 1 and 34 were mapped to different lines rather than the same line, then the number of cache-hits would be increased.

Since any memory address can map to any cache line, the problem remains how to construct an address translation scheme. Firstly we consider how to actually decide which line, i.e., the value A_{line} to use when we fetch data into the cache. Given searching for an empty line is typically a bad idea since it is just too costly, we have a number of options. For example, we might select one of the following:

Random Using a pseudo-random number generator, a line number is chosen at random regardless of the address and what is already resident in the cache. This is a low-cost option but can suffer from problems in that "unlucky" random sequences still provoke cache contention; the hope is that on average the randomisation will give a good distribution of line usage and hence low levels of contention.

Least Recently Used (LRU) Each time a cache line is accessed, it is moved to the head of a recently used (LRU) list. When the cache is required to load new data, the line at the end of the recently used list is selected; the rationale is that this is the least recently used so is most probable never to be used again. Normally, the cache will be quite large so maintaining the LRU list is usually too costly an option.

The word offset A_{word} is simply given by the least-significant m bits of the address A, i.e., $A_{word} = A \bmod C_{words}$, in both cases; the tag A_{tag} is given by the remaining bits.

Now, given we have translated the address to a line when fetching the data we need to consider how to find it again when we want to use it. Essentially, since a fully-associative cache allows a memory address to map into any cache line we need to search the entire cache to see if it is resident. Since cache access needs to be as fast as possible, a linear (or even binary) search is out of the question; we must search all cache lines in parallel. This feature contributes massively to the cost of a hardware realisation; Figure 8.21 details a block diagram of the design.

Again consider an example fully-associative cache for a byte-addressed memory where we have that $C_{lines} = 2^3 = 8$ and $C_{words} = 2^2 = 4$ so the total cache size is 32

bytes. Imagine we feed it the same stream of memory accesses as before:

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

It is not difficult to see that using a random line selector, we can achieve optimal performance. For example, if our pseudo-random number generator happens, very fortuitously, to give the sequence

$$0, 1, 2, 3, 4, \ldots$$

then our access pattern is

$$\begin{array}{llll}
A = 1 & A_{word} = 1,\ A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 34 & A_{word} = 2,\ A_{tag} = 8 & A_{line} = 1 & \rightarrow \text{ miss} \\
A = 35 & A_{word} = 3,\ A_{tag} = 8 & A_{line} = 1 & \rightarrow \text{ hit} \\
A = 36 & A_{word} = 0,\ A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ miss} \\
A = 37 & A_{word} = 1,\ A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ hit} \\
A = 1 & A_{word} = 1,\ A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 38 & A_{word} = 2,\ A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ hit} \\
A = 39 & A_{word} = 3,\ A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ hit} \\
A = 40 & A_{word} = 0,\ A_{tag} = 10 & A_{line} = 4 & \rightarrow \text{ miss} \\
A = 41 & A_{word} = 1,\ A_{tag} = 10 & A_{line} = 4 & \rightarrow \text{ hit .}
\end{array}$$

The only misses that remain are compulsory. However, the pseudo-random number generator is equally likely, but rather less fortuitously, to produce the sequence

$$0, 0, 0, 0, 0, \ldots$$

in which case the cache performs much worse:

$$\begin{array}{llll}
A = 1 & A_{word} = 1,\ A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 34 & A_{word} = 2,\ A_{tag} = 8 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 35 & A_{word} = 3,\ A_{tag} = 8 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 36 & A_{word} = 0,\ A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 37 & A_{word} = 1,\ A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 1 & A_{word} = 1,\ A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 38 & A_{word} = 2,\ A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 39 & A_{word} = 3,\ A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 40 & A_{word} = 0,\ A_{tag} = 10 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 41 & A_{word} = 1,\ A_{tag} = 10 & A_{line} = 0 & \rightarrow \text{ hit .}
\end{array}$$

For the average case we are neither especially lucky nor unlucky, and as a result can typically expect better hit ratios than a direct-mapped cache.

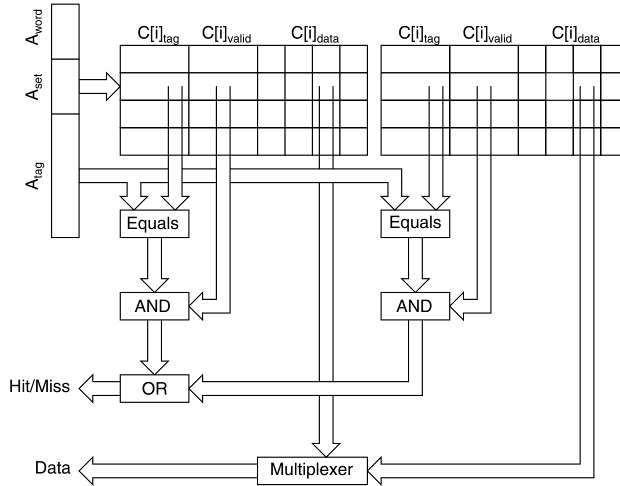


Figure 8.22 A block diagram of a set-associative cache.

8.3.5 Set-Associative Caches

On one hand, direct-mapped caches are simple to build but *can* exhibit relatively poor performance. On the other hand, fully-associative caches *can* improve performance but are more costly to realise. What would be ideal is to find a trade-off between the two designs; the **set-associative cache** does just this.

The basic idea is that the total cache size is sectioned into C_{set} **sets**, each of which groups $N = C_{lines}/C_{set}$ cache lines together. Instead of mapping an address to a cache line we instead map to a set, within each set, searching for a particular line is performed in a fully-associative manner. Usually we say such a cache is N-way set-associative since there are N places a cache line can reside in each set. Figure 8.22 gives a rough block diagram of set-associative cache design where $C_{set} = 2$. In a similar way to before, we define our address translation scheme as

$$\begin{aligned} A_{word} &= A \bmod C_{words} \\ A_{set} &= \lfloor A/C_{words} \rfloor \bmod C_{set} \\ A_{tag} &= \lfloor \lfloor A/C_{words} \rfloor / C_{set} \rfloor \end{aligned}$$

so that the word offset is taken as the bottom m bits of the address A and the set number A_{set} is the next $\lceil \log_2(C_{set}) \rceil$ bits; since we typically constrain C_{set} to be a power-of-two this value is an integer and the address translation scheme is easy to implement efficiently. To decide which line within the set is used, we just select a policy that follows our original choices for the fully-associative cache. For example, we might select a random or LRU selection policy. Since C_{set} is usually small (say

	Set 0						Set 1					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$		0						0				
$i=1$		0						0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a) Cache is initially empty.

	Set 0						Set 1					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3		0				
$i=1$		0						0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b) Access to address 1 misses, line 0 in set 0 is filled.

	Set 0						Set 1					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3		0				
$i=1$	4	1	32	33	34	35		0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(c) Access to address 34 misses, line 1 in set 0 is filled; access to address 35 then hits in line 1 of set 0.

	Set 0						Set 1					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3	4	1	36	37	38	39
$i=1$	4	1	32	33	34	35		0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(d) Access to address 36 misses, line 0 in set 1 is filled; access to address 37 then hits in line 0 of set 1; access to address 1 then hits in line 0 of set 0; access to address 38 then hits in line 0 of set 1; access to address 39 then hits in line 0 of set 1;

	Set 0						Set 1					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3	4	1	36	37	38	39
$i=1$	4	1	32	33	34	35		0				
$i=2$	5	1	40	41	42	43	44	0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(e) Access to address 40 misses, line 2 in set 0 is filled; access to address 41 then hits in line 2 of set 0.

Figure 8.23 An example of set-associative cache operation.

2, 4 or 8) it is feasible to choose a very effective selection policy such as LRU while limiting the cost this implies in terms of searching in parallel for a given line amongst the C_{set} possibilities. This option was moot with the fully-associative cache since there were too many possibilities to allow an efficient and cost-effective realisation.

Again consider an example set-associative cache for a byte-addressed memory where we have that $C_{lines} = 2^3 = 8$, $C_{set} = 2^1 = 2$ and $C_{words} = 2^2 = 4$ so the total cache size is 32 bytes. There are four sets each of two lines, suppose set 0 groups together lines 0 and 1, set 1 groups together lines 2 and 3 and so on. Imagine we feed it the same stream of memory accesses as before:

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41$$

and use an LRU selection policy to determine the line number within each set when data is fetched. With this architecture, we would expect an access pattern such as

$$\begin{array}{llll}
A = 1 & A_{word} = 1,\ A_{set} = 0,\ A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{miss} \\
A = 34 & A_{word} = 2,\ A_{set} = 0,\ A_{tag} = 4 & A_{line} = 1 & \rightarrow \text{miss} \\
A = 35 & A_{word} = 3,\ A_{set} = 0,\ A_{tag} = 4 & A_{line} = 1 & \rightarrow \text{hit} \\
A = 36 & A_{word} = 0,\ A_{set} = 1,\ A_{tag} = 4 & A_{line} = 0 & \rightarrow \text{miss} \\
A = 37 & A_{word} = 1,\ A_{set} = 1,\ A_{tag} = 4 & A_{line} = 0 & \rightarrow \text{hit} \\
A = 1 & A_{word} = 1,\ A_{set} = 0,\ A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{hit} \\
A = 38 & A_{word} = 2,\ A_{set} = 1,\ A_{tag} = 4 & A_{line} = 0 & \rightarrow \text{hit} \\
A = 39 & A_{word} = 3,\ A_{set} = 1,\ A_{tag} = 4 & A_{line} = 0 & \rightarrow \text{hit} \\
A = 40 & A_{word} = 0,\ A_{set} = 0,\ A_{tag} = 5 & A_{line} = 2 & \rightarrow \text{miss} \\
A = 41 & A_{word} = 1,\ A_{set} = 0,\ A_{tag} = 5 & A_{line} = 2 & \rightarrow \text{hit}\ .
\end{array}$$

This matches the fortuitous case of our fully-associative cache example; we have clearly used a less costly design however. The main difference is the use of LRU to determine the line numbers within each set; the fact that each set contains 2 lines means, for example, that addresses 1 and 34 no longer conflict in this case even though before they might have mapped to the same cache line. When address 1 is accessed, none of the lines in set 0 have been used so we simply select line 0 and put this at the front of the LRU list. When address 34 is accessed, although it maps to the same set as address 0, we have (at least) one line within the set which has been used less often than line 0; so we simply select line 1 for example. As the access pattern continues in this way: consider what happens when address 35 is accessed. Translation of the address gives set 0 which contains two resident lines that were loaded when we accessed addresses 0 and 34. Parallel search of the two lines in that set matches the line loaded for 34 which is the one used for the access thus producing a cache-hit.

8.3.6 Cache Organisation

8.3.6.1 Cache Bypass

Sometimes it is attractive to perform uncached access to memory. In this case, as designers of a cache or processor, we might take one of two options. Firstly, we could provide explicit instructions (or a processor mode of some sort) that perform the access without going through the cache; this is called **cache bypass**. The second option is to provide a mechanism to turn off the cache somehow. Some processors allow this to be performed as a one-off configuration by setting a flag in the status register when powered on, other processors allow dynamic turning on and off of the cache during execution. Either way, bypassing use of the cache means that maintaining consistency between the cache and memory is difficult; it is common to leave this up to the programmer using it.

8.3.6.2 Split-Use Caches

In terms of caches, we have so far avoided the issue that they might be used to store instructions as well as data. The first issue here is that one can view the caches, like the rest of the memory hierarchy, as either a Harvard or stored program architecture. That is, one either regards access to instructions and data as separate, using separate caches for each, or as unified and hence sharing one common cache. Either approach is valid but it is common, to suit the pipelining technique described in Chapter 6, that separate caches are employed. Even from a more naive standpoint it might be attractive since separating the accesses might lessen the probability of cache pollution and contention.

Either way, this affords us some opportunity to optimise our implementation. For example, an instruction cache will not really require logic to perform writes: we do not usually write into the instructions, only updating data values once execution has begun, so the instruction cache can be significantly simpler. In addition, accesses to data and instructions exhibit different forms of locality; one might optimise the cache policies to take advantage of this knowledge so that better hit-ratios can be achieved in each case.

8.3.6.3 Split-Level Caches

It is possible to split the cache into a hierarchy just as we did with the more general memory hierarchy. In particular, it is attractive to construct the hierarchy to make a distinction between caches on the same chip as the processor, so-called **on-chip caches**, and those which are separate devices or **off-chip caches**. On-chip caches are typically faster because there is less of an interface to deal with between devices. But one can only fit so much logic on-chip, therefore off-chip caches can typically be larger. One can construct multiple levels of these on-chip and off-chip caches:

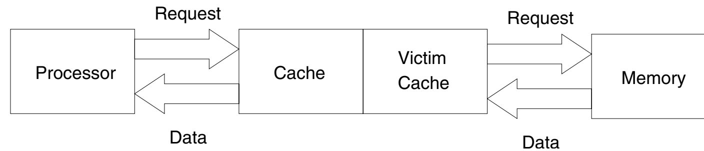


Figure 8.24 A basic block diagram of victim cache operation.

the closest level to the processor is level-one (the cache is an L1 cache), the next level in the hierarchy is level-two (the cache is an L2 cache) and so on.

This hierarchy represents a trade-off in terms of size and speed; the usual solution is to have a small on-chip cache and a larger off-chip. The hope is that the L1 cache will be able to cope with most accesses very quickly, and that it cannot deal with will mostly be dealt with by the L2 cache rather than main memory. This combined approach relies on the principle of locality to hold in order to be effective.

8.4 Advanced Cache Memories

8.4.1 Victim Caches

Considering only data caches once again, an important aspect of cache memory design is the trade-off between hit-ratio and access time. On one hand, more involved replacement policies and address translation, for example, improve hit-ratio; an improvement in hit-ratio typically means a decrease in accesses to main memory. On the other hand, as implementation of the cache design becomes more complex, the critical path of access logic becomes longer; this increases the time taken for cache hits and cache misses alike. A direct-mapped cache makes the trade-off in favour of lower access latency at the cost of lower hit-ratio; a set-associative cache improves hit-ratio using a more complex design but typically increases the access latency as a result.

Consider, for the sake of argument, a cache architecture with one level; there is a direct-mapped L1 cache between the processor and main memory. The **victim cache** design [30] makes a slightly different trade-off between hit-ratio and access time than a set-associative cache. Both hope to reduce the number of conflict misses, i.e., misses which are caused by two addresses mapping to the same cache line of the L1 cache and require access to main memory. Instead of altering the L1 cache design per se, the victim cache is placed between the two; roughly speaking, one could consider this level-one-and-a-half of the memory hierarchy. The victim cache is a small (only a few lines in size), fully-associative cache; problems with design complexity and access time are limited due to the small size. The aim is to capitalise

on the fact that empirically, it is common for cache lines that have recently been evicted from the L1 cache to be accessed again some time shortly after eviction; one can think of this as the L1 cache not quite being able to capitalise on access locality. By placing the victim cache between the L1 cache and main memory, i.e., on the fetch path of the L1 cache, it can store these victim lines temporarily and allow subsequent access to them without the need for a more expensive access to main memory. In short, the penalty associated with eviction from the L1 cache and subsequent reloading is minimised.

Figure 8.24 shows a basic block diagram of where the victim cache sits in relation to our previous discussion of cache memory. The L1 cache and the victim cache are placed close together on-chip; their access times can thus be as fast as a single processor cycle. For a given access, the L1 cache and the victim cache are searched at the same time. When a miss occurs in both caches, an access to main memory is required with the resulting cache line stored in the L1 cache only; any evicted lines are stored in the victim cache which operates according to an LRU policy. When a miss occurs in the L1 cache but the required data is located in the victim cache, the access can be completed without access to main memory; the value is taken from the victim cache by the processor and then placed in the L1 cache.

Notice that fast access time is guaranteed: since both the L1 cache *and* victim cache are on-chip and accessed in parallel, there is little difference between a hit in the L1 cache and a hit in the victim cache. Certainly they are still both at least an order of magnitude less than access to main memory. Furthermore, as long as the original empirical statement about locality holds, i.e., that recently evicted lines are often accessed again, hit-ratio moves closer to that of a set-associated cache and is certainly improved over the basic direct-mapped design. Of course this depends on the program being executed and the resulting stream of accesses to memory, but studies in the original proposal [30] show that anywhere between 10% and 70% of all misses can be removed by a sixteen line victim cache.

To give a concrete example of victim cache operation, recall where we demonstrated cache behaviours using a stream of memory accesses whose addresses were

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

We previously used an example direct-mapped cache for a byte-addressed memory where $C_{lines} = 2^3 = 8$ and $C_{words} = 2^2 = 4$ so the total cache size is 32 bytes. Using this cache, we obtained the following behaviour:

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$		0						0				
$i=1$		0						0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a) Cache is initially empty.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3		0				
$i=1$		0						0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(b) Access to address 1 misses, line 0 is filled.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	1	1	32	33	34	35	0	1	0	1	2	3
$i=1$		0						0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(c) Access to address 34 misses, line 0 evicted then re-filled via victim cache; access to address 35 then hits in line 0.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	1	1	32	33	34	35	0	1	0	1	2	3
$i=1$	1	1	36	37	38	39		0				
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(d) Access to address 36 misses, line 1 is filled; access to address 37 then hits in line 1.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3	0	0	0	1	2	3
$i=1$	1	1	36	37	38	39	1	1	32	33	34	35
$i=2$		0						0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(e) Access to address 1 misses, line 0 evicted then re-filled via victim cache; access to addresses 38 and 39 then hit in line 1.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data[0]}$	$C[i]_{data[1]}$	$C[i]_{data[2]}$	$C[i]_{data[3]}$
$i=0$	0	1	0	1	2	3	0	0	0	1	2	3
$i=1$	1	1	36	37	38	39	1	1	32	33	34	35
$i=2$	1	1	40	41	42	43		0				
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

(f) Access to address 40 misses, line 2 is filled; access to address 41 then hits in line 2.

Figure 8.25 An example of victim cache operation.

$$
\begin{array}{llll}
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 38 & A_{word} = 2, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{hit}\ .
\end{array}
$$

Notice that addresses $0 \ldots 3$ exactly match our definition of a victim above. That is, address 1 is initially loaded into line 0 in access one before being evicted by access two and then reloaded by access six. Figure 8.25 diagrammatically details the cache behaviour when a victim cache with four lines is added; the victim cache content is shown below the L1 cache content for each step. The behaviour changes to the following:

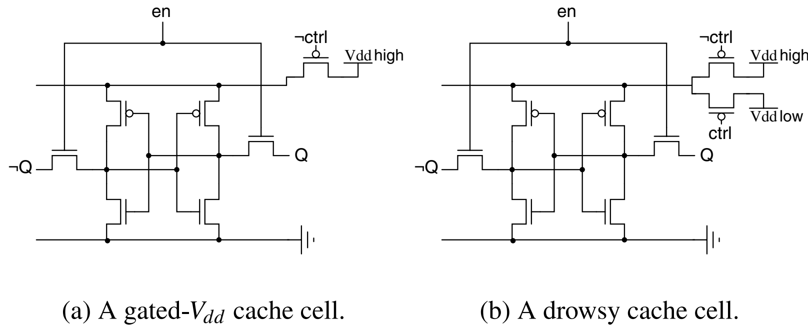
$$
\begin{array}{llll}
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{hit} \\
A = 38 & A_{word} = 2, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{hit}\ .
\end{array}
$$

Essentially, the penalty caused by reloading the line for addresses $0 \ldots 3$ in access six is eliminated: the reload is performed via the victim cache rather than main memory and hence we categorise the access a cache-hit rather than a cache-miss.

8.4.2 Gated and Drowsy Caches

Power consumption plays an important role in the design of embedded processors. Within a mobile telephone, lower power consumption is a good selling point; within a device such as a sensor node it is a vital operational characteristic. This has led to significant research into low-power design and manufacturing techniques. The design of low-power memory hierarchies is a specific area of interest given that memory, cache memories in particular, form a significant component in overall power consumption.

Like any transistor-based circuit, the overall power consumption in caches that use standard SRAM cells can be divided into dynamic and static components. Dy-



(a) A gated-V_{dd} cache cell. (b) A drowsy cache cell.

Figure 8.26 Two low-power memory cell designs.

namic power dissipation occurs when a logic gate switches from one state to another only; static power is dissipated by every logic gate independent of the switching activity. Originally the dynamic component dominated the overall power consumption. However, as transistors have become smaller, and hence their density has increased, static power dissipation has started to have a major impact on the overall power consumption.

There are many approaches to reduce power dissipation in cache memories; the **gated-V_{dd} cache** design [54] and the **drowsy cache** design [22, 31] are both attractive. Roughly speaking, both designs offer a mechanism to turn cache lines into a low-power state when they are not serving a useful purpose, i.e., not being regularly accessed and hence not improving memory access performance. Both designs achieve this by altering the basic SRAM cell design; the resulting cells are shown in Figure 8.26; the $ctrl$ signal is used to control the cell. Memory cells in a gated-V_{dd} cache structure can be powered off using a transistor that disconnects the power supply to the cell. While this technique clearly reduces overall power dissipation, the information stored in a cell is not preserved; the design is referred to as a non-state-preserving cache. The drowsy cache offers an alternative, state-preserving technique. Memory cells in a drowsy cache structure are capable of operating in a normal mode, where they can be freely read or written to, and a drowsy, low-power mode where they retain their state but need to be woken before any access can occur. This low-power mode is facilitated by switching from being powered by the normal $V_{dd\ high}$ input to an alternative low-power voltage labelled $V_{dd\ low}$. The trade-off between the two designs should be clear: either totally power off the cells (as in the gated-V_{dd} cache design) and incur a potential performance penalty if cells need to be reloaded having been emptied of content, or retain the content (as in the drowsy cache design) and incur a power dissipation penalty because cells are only in a low-power state rather than powered off. This trade-off requires careful thought since cache misses that result from, for example, powering down gated-V_{dd} cache lines require more power than a cache hit; therefore any short-term saving from powering down such lines might be lost if they are accessed again in the longer term.

Having replaced the memory cells within a cache design, it remains to control those cells somehow so they are forced into low-power mode according to some reasonable heuristic or policy. That is, we need some control logic which detects when we could benefit from switching a given cache line into low-power mode: the more accurate this detection (in relation to the associated penalty for getting it wrong), the better our trade-off between power dissipation and performance becomes. Two fairly basic approaches can be identified; see for example [22, Section 2]:

"Simple" Maintain a global counter which is incremented after each processor cycle; when the global counter reaches some threshold, place all cache lines into low-power mode.

"No-Access" Maintain a global counter which is incremented after each processor cycle and a single bit associated with each cache line which is set when the line is accessed; when the global counter reaches some threshold, place all cache lines with their access bit not set into in low-power mode.

The "simple" policy is clearly easier to implement and probably requires less control and state logic; however, it is less accurate at powering down cache lines than the "no access" policy in the sense that it is wrong more often about lines which are not being used. More advantages and disadvantages of each policy depend on the type of cache being used, the program being executed and the resulting stream of accesses to memory, and the threshold alluded to which triggers updates.

8.5 Putting It All Together

To flesh out the components in the basic Verilog processor model developed at the end of Chapter 5, we need two main devices which are not obviously implementable using low-level logic components: a general-purpose register file and a memory system (including a cache). Both devices were initially implemented as 2-dimensional Verilog registers; we now know that there is more involved under the surface. To try to make it more obvious that the devices can be realised using techniques and components we know about, we will try to write modules that more closely model their real operational behaviour. However, we will still rely on a bit of help from Verilog: we have already showed how a 2-dimensional register can be implemented using D-type flip-flops, and continue to rely on the 2-dimensional register to store data as a result. That is, to make things simple enough to read and explain, the difference is that our devices will implement associated control logic and assume we can swap the register for the flip-flop-based memory with additional work.

```
module gpr( input  wire [ 4:0] addr_w0,
            input  wire [31:0] data_w0,
            input  wire        we0,

            input  wire [ 4:0] addr_r0,
            output wire [31:0] data_r0,
            input  wire        re0,

            input  wire [ 4:0] addr_r1,
            output wire [31:0] data_r1,
            input  wire        re1 );

  reg [31:0] data[1:31];

  assign data_r0 = re0 ? ( addr_r0 == 0 ? 32'b0 : data[ addr_r0 ] ) : 32'bZ;
  assign data_r1 = re1 ? ( addr_r1 == 0 ? 32'b0 : data[ addr_r1 ] ) : 32'bZ;

  always @( posedge we0 )
  begin
    if( addr_w0 != 0 )
      data[ addr_w0 ] = data_w0;
  end

endmodule
```

Listing 8.5 A Verilog module that models the general-purpose register file.

8.5.1 Register File

The register file is perhaps the most simple of the devices we require; it is simply a memory with two read ports and one write port. That is, we can read two values and write one value at once. In addition, we required that it implements the special behaviour associated with register number 0 which always produces the value 0 when read and discards any writes.

Listing 8.5 details a simple Verilog module for such a device. The interface includes three ports which supply an address and an enable signal to indicate that port is in operation; for example, the write port accepts the register address on `addr_w0` and is enabled by `we0`. Each interface also includes an input or output signal to convey data over depending whether the port is used to read or write data, for example the write port has `data_w0` as input. The register values themselves are held in a two dimensional Verilog register called `data`. Notice that the indexing excludes register number 0 which is not required. Two continuous assignments implement the read ports. Each assignment tests if the associated port is enabled; if it is not, the assignment writes `32'bZ` onto the output to implement 3-state style behaviour. If the port is enabled, the assignment firsts tests if register number 0 is being accessed, and outputs the value 0 as a result, or simply looks up the register value if not. Writes to the register file are edge triggered. A short process, triggered by `we0`, first checks that the write is not to register number 0 and updates the register value with the input `data_w0` if not.

```
module mem( input  wire        clk,
            input  wire        rst,

            input  wire [31:0] prev_addr,
            input  wire [ 7:0] prev_data_w,
            output reg  [ 7:0] prev_data_r,
            input  wire        prev_mrq,
            output reg         prev_mrs,
            input  wire        prev_re,
            input  wire        prev_we );

  reg [7:0] data[0:1023];

  always @ ( posedge rst )
  begin
    $readmemh( "memory/mem.bin", data, 0 );
  end

  always @ ( posedge prev_mrq )
  begin
    if     ( prev_re )
      prev_data_r = data[ prev_addr ];
    else if( prev_we )
      data[ prev_addr ] = prev_data_w;

    #6 prev_mrs = 1;
       @( negedge prev_mrq );
    #6 prev_mrs = 0;
  end

endmodule
```

Listing 8.6 A Verilog module that models main memory.

8.5.2 Main Memory

Listing 8.6 details the main memory module which closely resembles the one we developed previously: remember that we claimed that we could replace the 2-dimensional Verilog register with a concrete memory design based on D-type flip-flops. The only real difference is that we include a process that can initialise the memory content.

To initialise the memory, as triggered by a positive edge on the reset signal, we use the `$readmemh` system task. This has the effect of reading the file specified as the first argument into the 2-dimensional register specified as the second argument at an offset specified by the third argument. In this case, the file `mem.bin` is filled with 256 lines which look like

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
.. .. .. ..
```

Each value represents one byte encoded as two hexadecimal digits; using the file essentially just zeros the memory content.

```
module dmc( input  wire        clk,
            input  wire        rst,

            input  wire [31:0] prev_addr,
            input  wire [ 7:0] prev_data_w,
            output reg  [ 7:0] prev_data_r,
            input  wire        prev_mrq,
            output reg         prev_mrs,
            input  wire        prev_re,
            input  wire        prev_we,

            output reg  [31:0] next_addr,
            output reg  [ 7:0] next_data_w,
            input  wire [ 7:0] next_data_r,
            output reg         next_mrq,
            input  wire        next_mrs,
            output reg         next_re,
            output reg         next_we );

  ...

endmodule
```

Listing 8.7 The interface to a Verilog module that models a direct-mapped cache.

8.5.3 Cache Memory

To describe the cache memory, we again opt for a high-level modelling approach. Of the three designs introduced, direct-mapped caches are the most conceptually simple. To follow the example presented previously, we model such a cache parametrised to house eight lines each holding four 8-bit sub-words. The Verilog code is too long to present on one page but the interface is shown in Listing 8.7. The cache module has one interface, whose signals are prefixed with `prev`, for incoming transactions with the processor and one, whose signals are prefixed with `next`, for outgoing transactions with the memory. To operate the memory via the outgoing interface, two tasks are used; these are shown in Listing 8.8. The `ld_byte` and `st_byte` tasks implement the handshake protocol which allows the cache to load and store bytes from and into the memory. Their operation is fairly straightforward but consider the `ld_byte` task as an example. It starts by waiting for the next positive clock edge to synchronise the operation; once this is encountered it sets the address, read enable and memory request signals to initiate the transaction. The task then waits for a positive edge on the memory response signal which indicates that the memory has completed the operation; the task clears the read enable and memory request signals and reads the incoming value from the data bus. Finally it waits for the memory response signal to be cleared by the memory before continuing.

The cache operation is implemented using two processes shown in Listing 8.9. This code does not show the definitions of internal cache structures: `tag` holds the tags for the cache lines, `valid` holds the valid flags for the cache lines and `data` holds the cache lines themselves. The first process simply initialises the cache content to empty by setting the valid flags for all lines to false; this is triggered

```
task ld_byte( input  [31:0] t_addr,
              output [ 7:0] t_data );
begin
  @ ( posedge clk )
  begin
    next_addr = t_addr;

    next_re  = 1;
    next_mrq = 1;
  end

  @ ( posedge next_mrs )
  begin
    next_re  = 0;
    next_mrq = 0;

    t_data = next_data_r;
  end
  @ ( negedge next_mrs );
end
endtask

task st_byte( input  [31:0] t_addr,
              input  [ 7:0] t_data );
begin
  @ ( posedge clk )
  begin
    next_addr   = t_addr;
    next_data_w = t_data;

    next_we  = 1;
    next_mrq = 1;
  end

  @ ( posedge next_mrs )
  begin
    next_we  = 0;
    next_mrq = 0;
  end
  @ ( negedge next_mrs );
end
endtask
```

Listing 8.8 Verilog tasks to access memory via the outgoing cache interface.

by a positive edge on the reset signal. The second process is triggered every time there is a memory request via the incoming interface, i.e., a memory request from the processor. It starts by calculating the tag, line number and word offset from the incoming address. It can then check the associated line to see if it contains the required data: if the line is not valid or the stored and computed tags do not match, the line needs to be fetched from memory using the `ld_byte` task. Once the data is definitely resident in the line, we can fulfil the request, writing the data through via the outgoing interface to memory if the request was a write. Finally we instrument a small artificial delay, to model the cache latency.

It is interesting to see how the cache behaves when connected to the main memory module described above. Listing 8.10 demonstrates how the two can be instantiated to form a composite cached memory system; essentially one just has to wire

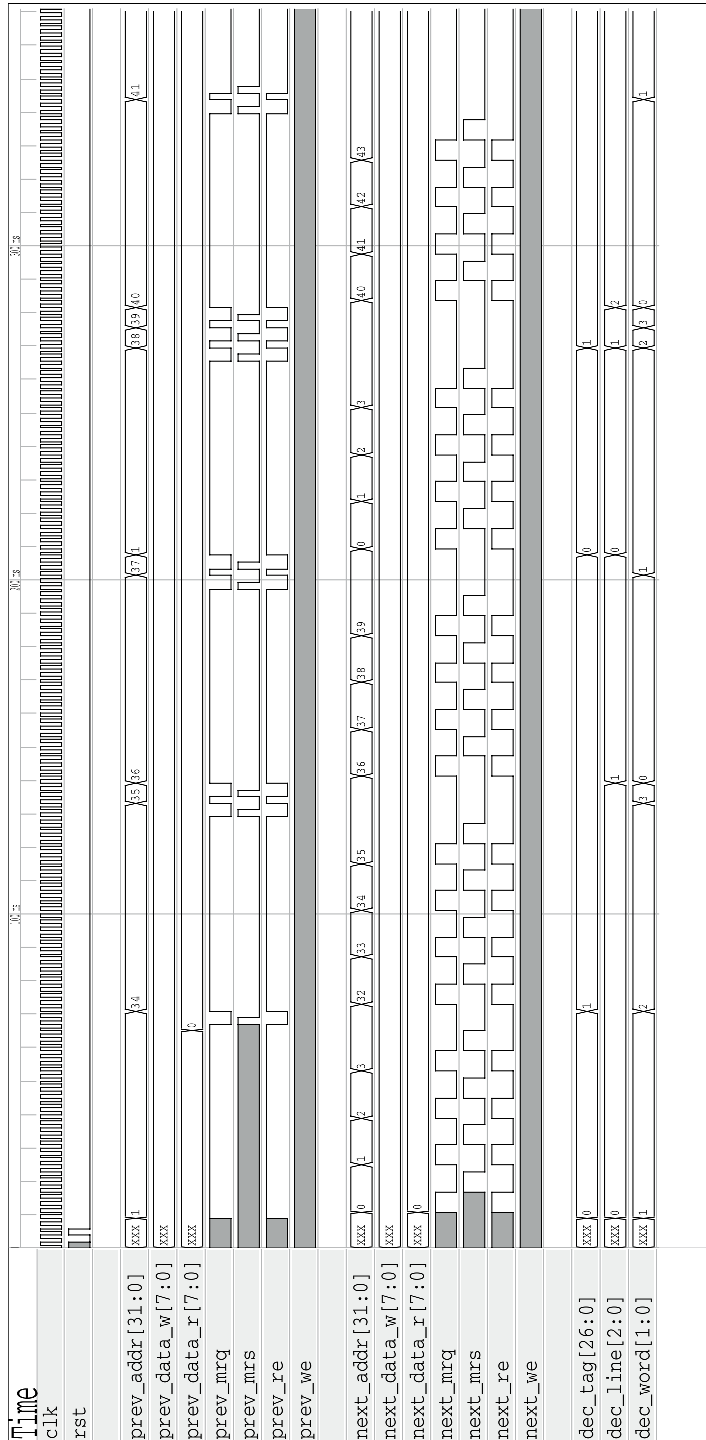


Figure 8.27 Behaviour of the composite cached memory system.

```
always @( posedge rst )
begin
  for( i = 0; i < 8; i = i + 1 )
    valid[i] = 0;
end

always @ ( posedge prev_mrq )
begin
  dec_tag  = prev_addr[31:5];
  dec_line = prev_addr[ 4:2];
  dec_word = prev_addr[ 1:0];

  if( ( !valid[dec_line] ) || ( dec_tag != tag[dec_line] ) )
  begin
    tag  [dec_line] = dec_tag;
    valid[dec_line] =       1;

    ld_byte( {dec_tag,dec_line,2'b00}, data[{dec_line,2'b00}] );
    ld_byte( {dec_tag,dec_line,2'b01}, data[{dec_line,2'b01}] );
    ld_byte( {dec_tag,dec_line,2'b10}, data[{dec_line,2'b10}] );
    ld_byte( {dec_tag,dec_line,2'b11}, data[{dec_line,2'b11}] );
  end

  if     ( prev_re )
    prev_data_r = data[{dec_line,dec_word}];
  else if( prev_we )
    data[{dec_line,dec_word}] = prev_data_w;

  if( prev_we )
    st_byte( {dec_tag,dec_line,dec_word}, data[{dec_line,dec_word}] );

  #2 prev_mrs = 1;
     @( negedge prev_mrq );
  #2 prev_mrs = 0;
end
```

Listing 8.9 Verilog processes that determine the cache behaviour.

the two devices so that they can talk to each other. To test the combined system and inspect the behaviour of the cache, we fed the same sequence of loads to it as in our more theoretical outline:

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

Figure 8.27 shows the cache behaviour during these accesses. Remember that the memory is zeroed during initialisation by the reset signal; the values read are hence all 0. The actual cache behaviour is much more interesting. We start by loading the value at address 1. The cache computes the tag as 0, the line number as 0 and the word offset as 1. Since all the cache lines are initially invalid, this causes a cache miss and the cache is forced to load the associated line, i.e., addresses 0, 1, 2 and 3, from main memory. The same pattern occurs when we try to load the value at address 34. However, when we try to access the value at address 35 the access completes much more quickly; there is no interaction with main memory at all since the value we require is already resident in line 0. The access pattern continues as

```
module cached_mem( input  wire        clk,
                   input  wire        rst,

                   input  wire [31:0] prev_addr,
                   input  wire [ 7:0] prev_data_w,
                   output wire [ 7:0] prev_data_r,
                   input  wire        prev_mrq,
                   output wire        prev_mrs,
                   input  wire        prev_re,
                   input  wire        prev_we );

  wire [31:0] next_addr;
  wire [ 7:0] next_data_w;
  wire [ 7:0] next_data_r;
  wire        next_mrq;
  wire        next_mrs;
  wire        next_re;
  wire        next_we;

  dmc t0( .clk        (clk        ),
          .rst        (rst        ),

          .prev_addr  (prev_addr  ),
          .prev_data_w(prev_data_w),
          .prev_data_r(prev_data_r),
          .prev_mrq   (prev_mrq   ),
          .prev_mrs   (prev_mrs   ),
          .prev_re    (prev_re    ),
          .prev_we    (prev_we    ),

          .next_addr  (next_addr  ),
          .next_data_w(next_data_w),
          .next_data_r(next_data_r),
          .next_mrq   (next_mrq   ),
          .next_mrs   (next_mrs   ),
          .next_re    (next_re    ),
          .next_we    (next_we    ) );

  mem t1( .clk        (clk        ),
          .rst        (rst        ),

          .prev_addr  (next_addr  ),
          .prev_data_w(next_data_w),
          .prev_data_r(next_data_r),
          .prev_mrq   (next_mrq   ),
          .prev_mrs   (next_mrs   ),
          .prev_re    (next_re    ),
          .prev_we    (next_we    ) );

endmodule
```

Listing 8.10 Instantiating the main memory and direct-mapped cache to form a composite cached memory system.

one would expect with cache-misses producing long access latencies and cache-hits completing much more quickly.

8.6 Further Reading

- J.L. Hennessy and D.A. Patterson.
  Computer Architecture: A Quantitative Approach.
  Morgan-Kaufmann, 2002. ISBN: 1-558-60724-2.
- M.D. Hill, N.P. Jouppi and G.S. Sohi.
  Readings in Computer Architecture.
  Morgan-Kaufmann, 2000. ISBN: 1-558-60539-8.
- D.A. Patterson and J.L. Hennessy.
  Computer Organization and Design: The Hardware/software Interface.
  Morgan-Kaufmann, 2004. ISBN: 1-558-60604-1.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.
- A.S. Tanenbaum.
  Structured Computer Organisation.
  Prentice-Hall, 2005. ISBN: 0-131-48521-0.

8.7 Example Questions

33. Draw a diagram of a typical memory hierarchy explaining the types of device one would expect to find in each level, their key characteristics and the principles that allow the memory hierarchy to improve system performance.

34. A given processor accesses memory using 32-bit addresses. Suppose there is a cache between the processor and a byte-addressable main memory of 512 bytes total size and with a line size of 8 bytes. For each of the following cache architectures specify how the 32-bit address would be used by the address translation mechanism in the cache:

a. Direct-mapped cache.
b. 2-way set-associative cache.
c. 4-way set-associative cache.
d. Fully-associative cache.

For example, in the direct-mapped case you would need to specify which bits of the address are used for the tag, the line number and the word offset.

35. Cache misses can be classified as either compulsory misses, capacity misses or conflict misses. For **each** class of cache miss, explain why it might occur and how changes in the cache architecture can reduce how often it occurs.

36. A given 32-bit processor has a unified level-one cache. The cache is 1 kilobyte in size, holds 4 bytes per-line and is direct-mapped. A programmer writes, compiles and executes the following program on the processor:

```
char A[1024], B[1024], C[1024];

int main( int argc, char* argv[] )
{
  for( int i = 0; i < 1024; i++ ) {
    A[i] = B[i] + C[i];
  }
}
```

a. Calculate the number of bits of an address required for each of the following quantities required to locate items in the cache:

   i. cache word address,
   ii. cache line address,
   iii. cache tag.

   and show how they are mapped onto the address.

b. Explain the following terms in relation to the cache hardware using the above program as an example:

   i. spacial locality,
   ii. temporal locality,
   iii. cache interference or contention.

c. Describe the sequence of cache hits and misses caused by accesses to `A[i]`, `B[i]` and `C[i]` as the loop progresses.

d. The definition of arrays `A`, `B` and `C` are changed to read:

```
char A[1025], B[1025], C[1025];
```

   i. Explain how this changes the sequence of cache hits and misses described above and why it improves on the original program.
   ii. Outline a different cache architecture that could increase the performance of the original program without changing.