CHAPTER 5: *Memory*

5.1 Introduction

Any embedded system's functionality consists of three aspects: processing, storage, and communication. Processing is the transformation of data, storage is the retention of data for later use, and communication is the transfer of data. Each of these aspects must be implemented. We use *processors* to implement processing, *memory* to implement storage, and *buses* to implement communication. The earlier chapters described common processor types: custom single-purpose processors, general-purpose processors, and standard single-purpose processors. This chapter describes memory.

Let's start by describing some basic memory concepts. A memory stores large numbers of bits. These bits can be viewed as m words of n bits each, for a total of $m * n$ bits, as illustrated in Figure 5.1(a). We refer to a memory as an $m \times n$ ("m-by-n") memory. Figure 5.1(b) shows an external view of a memory. $\mathrm{Log}_2(m)$ address input signals are required to identify a particular word. Stated another way, if a memory has k address inputs, it can have up to 2^k words. n data signals are required to output (and possibly input) a selected word. For example, a 4,096-by-8 memory can store 32,768 bits, and requires 12 address signals and eight input/output data signals. To read a memory means to retrieve the word of a particular address, while to write a memory means to store a word in a particular address. A *memory access* refers to either a read or write. A memory that can be both read and written requires an

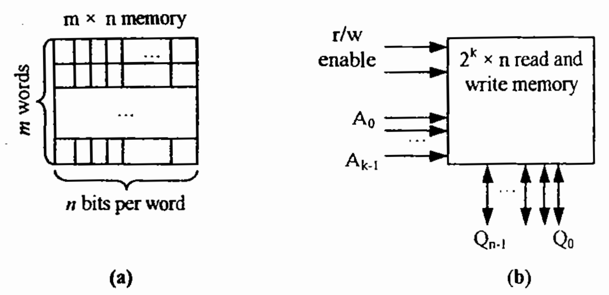


Figure 5.1: Memory: (a) words and bits per word, (b) memory block diagram.

additional control input, labeled *r/w* in Figure 5.1(b), to indicate which access to perform. Most memory types have an enable control input, which when deasserted, causes the memory to ignore the address, such that data is neither written to or read from the memory. Some types of memory, known as *multiport* memory, support multiple accesses to different locations simultaneously. Such a memory has multiple sets of control lines, address lines, and data lines, where one set of address and corresponding data and control lines is known as a *port*.

Memory has evolved very rapidly over the past few decades. The main advancement has been the trend of memory-chip bit-capacity doubling every 18 months, following Moore's Law. The importance of this trend in enabling today's sophisticated embedded systems should not be underestimated. No matter how fast and complex processors become, those processors still need memories to store programs and to store data to operate on. For example, a digital camera is possible not only because of fast A2D and compression processors but also because of memories capable of storing sufficient quantities of bits to represent quality pictures.

Further advancements to memory have blurred the distinction between the two traditional memory categories of ROM and RAM, providing designers with the benefit of more choice. Traditionally, the term *ROM* has referred to a memory that a processor can only read, and which holds its stored bits even without a power source. The term *RAM* has referred to a memory that a processor can both read and write but loses its stored bits if power is removed. However, processors can not only read, but also write to advanced ROMs, like EEPROM and Flash, although such writing may be slow compared to writing RAMs. Furthermore, advanced RAMs, like NVRAMs, can hold their bits even when power is removed.

Thus, in this chapter, we depart from the traditional ROM/RAM distinction, and instead distinguish among memories using two characteristics, namely write ability and storage permanence. We then introduce forms of memories commonly found in embedded systems. We describe techniques for the common task of composing memories to build bigger memories. We describe the use of memory hierarchy to improve memory access speed.

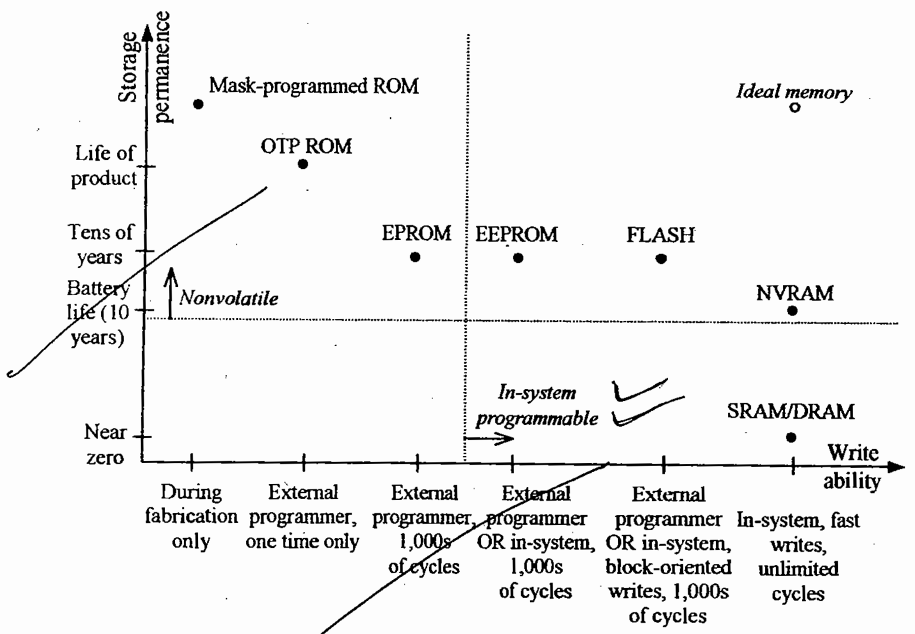


Figure 5.2: Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).

5.2 Memory Write Ability and Storage Permanence

Write Ability

We use the term *write ability* to refer to the manner and speed that a particular memory can be written. All types of memory can be read from by a processor, since otherwise their stored bits would serve little purpose in an embedded system. Likewise, all types of memory can be written, since otherwise we would have no way to store bits in such a memory. However, the manner and speed of such writing varies greatly among memory types.

At the high end of the range of write ability, we have types of memory that a processor can write to simply and quickly by setting such a memory's address lines, data input bits, and control lines appropriately. Toward the middle of the range, we have types of memory that are slower to write by a processor. Toward the lower end of the range, we have types of memory that can only be written to by a special piece of equipment called a "programmer." This device must apply special voltage levels to write to the memory, also known as "programming" or "burning" the memory. Do not confuse this use of the term *programmer* with the use referring to someone who writes software. At the low end of the range of write ability, we have types of memory that can only have their bits stored when the memory chip itself is being fabricated.

Storage Permanence

Storage permanence refers to the ability of memory to hold its stored bits after those bits have been written. At the low end of the range of storage permanence is memory that begins to lose its bits almost immediately after those bits are written, and therefore must be continually refreshed. Next is memory that will hold its bits as long as power is applied to the memory. Next comes memory that can hold its bits for days, months, or even years after the memory's power source has been turned off. At the high-end of the range is memory that will essentially never lose its bits — as long as the memory chip is not damaged, of course.

The terms *nonvolatile* and *volatile* are commonly used to divide memory types into two categories along the storage permanence axis, as shown in Figure 5.2. *Nonvolatile* memory can hold its bits even after power is no longer supplied. Conversely, volatile memory requires continual power to retain its data.

Likewise, the term *in-system programmable* is used to divide memories into two categories along the write ability axis. In-system programmable memory can be written to by a processor appearing in the embedded system that uses the memory. Conversely, a memory that is not in-system programmable must be written by some external means, rather than during normal operation of the embedded system.

Trade-offs

As described in Chapter 1, design metrics often compete with one another. Memory write ability and storage permanence are two such metrics. Ideally, we want a memory with the highest write ability and the highest storage permanence, as illustrated by the ideal memory point in Figure 5.2. Unfortunately, write ability and storage permanence tend to be inversely proportional to one another. Furthermore, highly writable memory typically requires more area and/or power than less-writable memory.

5.3 Common Memory Types

Introduction to "Read-Only" Memory — ROM

ROM, or read-only memory, is a nonvolatile memory that can be read from, but not written to, by a processor in an embedded system. Of course, there must be a mechanism for setting the bits in the memory, but we call this programming, not writing. For traditional types of ROM, such programming is done off-line, when the memory is not actively serving as a memory in an embedded system. We program such a ROM before inserting it into the embedded system. Figure 5.3(a) provides an external block diagram of a ROM.

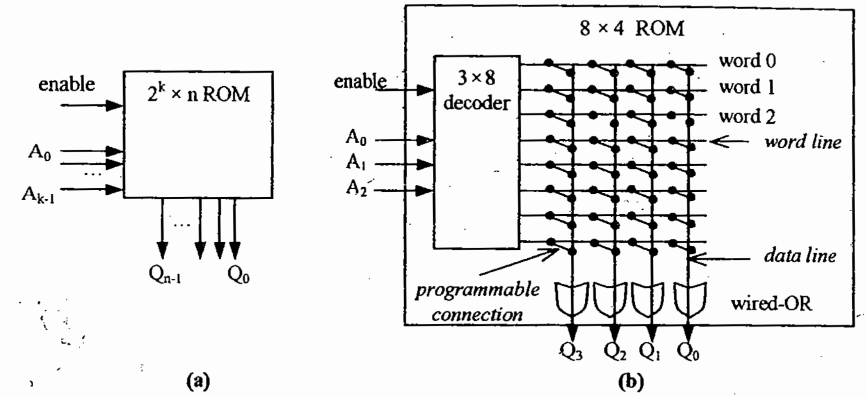


Figure 5.3: ROM: (a) external block diagram, (b) internal view of an 8×4 ROM.

We can use ROM for various purposes. One use is to store a software program for a general-purpose processor. We may write each program instruction to one ROM word. For some processors, we write each instruction to several ROM words. For other processors, we may pack several instructions into a single ROM word. A related use is to store constant data, like large lookup tables of strings or numbers.

A second common use is to store constant data needed by a system. A third, less common, use is to implement a combinational circuit. We can implement any combinational function of k variables by using a $2^k \times 1$ ROM, and we can implement n functions of the same k variables using a $2^k \times n$ ROM. We simply program the ROM to implement the truth table for the functions, as shown in Figure 5.4.

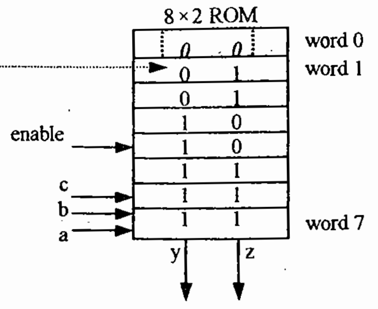
Figure 5.3(b) provides a symbolic view of the internal design of an 8×4 ROM. To the right of the 3×8 decoder in the figure is a grid of lines, with word lines running horizontally and data lines vertically; lines that cross without a circle in the figure are not connected. Thus, word lines only connect to data lines via the programmable connection lines shown. The figure shows all connection lines in place except for two connections in word 2. To see how this device acts as a read-only memory, consider an input address of 010. The decoder will thus set word 2's line to 1. Because the lines connecting this word line with data lines 2 and 0 do not exist, the ROM output will read 1010. Note that if the ROM *enable* input were 0, then no word would be read, since all decoder outputs would be 0. Also note that each data line is shown as a wired-OR, meaning that the wire itself acts to logically OR all the connections to it.

How do we program the programmable connections? The answer depends on the type of ROM being used. Common types include mask-programmed ROM, one-time programmable ROM, erasable programmable ROM, electrically erasable programmable ROM, and Flash, in order of increasing write ability. In terms of write ability, the latter two have such a high

Truth table (ROM contents)

Inputs (address)			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

(a)



(b)

Figure 5.4: Implementing combinational functions with a ROM: (a) truth table, (b) ROM contents.

degree of write ability that calling them read-only memory is not really accurate. In terms of storage permanence, all ROMs have high storage permanence, and in fact, all are nonvolatile. We now describe each ROM type briefly.

Mask-Programmed ROM

In a *mask-programmed* ROM, the connection is programmed when the chip is being fabricated by creating an appropriate set of masks. Mask-programmed ROM obviously has extremely low write ability, as illustrated in Figure 5.2, but has the highest storage permanence of any memory type, since the stored bits will never change unless the chip is damaged. Such ROM types are typically only used after a final design has been determined, and only in high-volume systems, for which the NRE costs can be amortized to result in a lower unit cost than other ROM types.

OTP ROM — One-Time Programmable ROM

Many systems use some form of user-programmable ROM device, meaning the ROM can be programmed by a designer in the lab, long after the chip has been manufactured. User-programmable ROMs are generally referred to as *programmable* ROMs, or PROMs. These devices are better suited to prototyping and to low-volume applications than are mask-programmed ROM. The most basic PROM uses a fuse for each programmable connection. To program a PROM device, the user provides a file that indicates the desired ROM contents. A piece of equipment called a ROM programmer then configures each programmable connection according to the file. Note that here the programmer is a piece of equipment, not a person who writes software. The ROM programmer blows fuses by passing a large current wherever a connection should not exist. However, once a fuse is blown, the

connection can never be reestablished. For this reason, basic PROM is often referred to as one-time-programmable ROM, or OTP ROM.

OTP ROMs have the lowest write ability of all PROMs, as illustrated in Figure 5.2, since they can only be written once, and they require a programmer device. However, they have very high storage permanence, since their stored bits won't change unless someone reconnects the device to a programmer and blows more fuses. Because of their high storage permanence, OTP ROMs are commonly used in final products, versus other PROMs, which are more susceptible to having their contents inadvertently modified from radiation, maliciousness, or just the mere passage of many years.

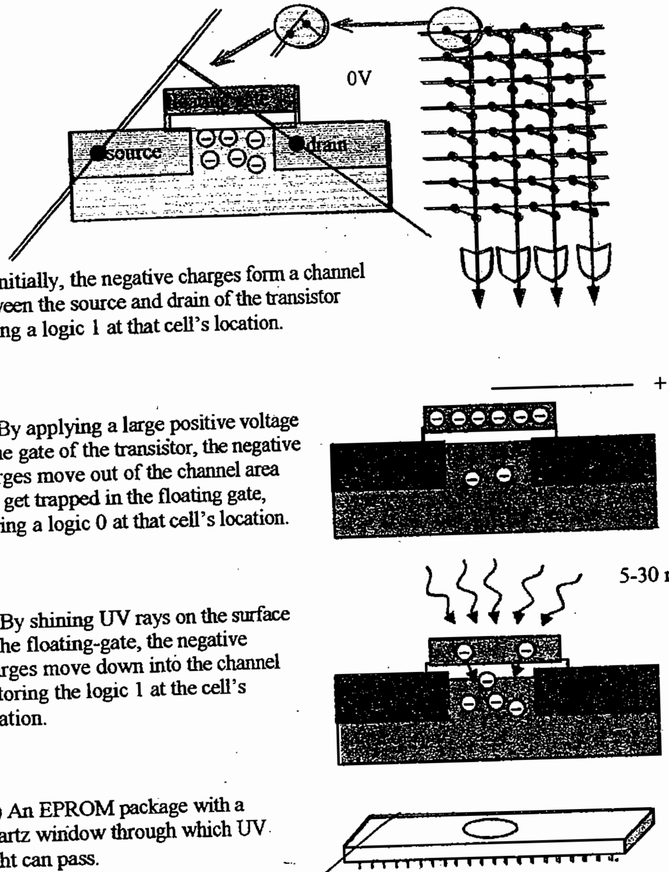
OTP ROMs are also cheaper per chip than other PROMs, often costing under a dollar each. This also makes them more attractive in final products versus other types of PROM, and also versus mask-programmed ROM when time-to-market constraints or unit costs make them a better choice. Because the chips are so cheap, some designers even use OTP ROMs during design development. Those designers simply throw away the used chips as they program new ones.

EPROM — Erasable Programmable ROM

Erase at chip level

Another type of PROM is an *erasable* PROM, or EPROM. This device uses a MOS transistor as its programmable component. The transistor has a "floating gate," shown in Figure 5.5(a), meaning the transistor's gate is not connected and is instead surrounded by insulator. An EPROM programmer injects electrons into the floating gate, using higher than normal voltage (usually 12 V to 25 V) that causes electrons to tunnel through the insulator into the gate, as in Figure 5.5(b). When that high voltage is removed, the electrons cannot escape, and hence the gate has been charged and programming has occurred. Reading an EPROM is much faster than writing, since reading doesn't require programming. To erase the program, the electrons must be excited enough to escape from the gate. Ultraviolet (UV) light is used to fulfill this role of erasing, as shown in Figure 5.5(c). The device must be placed under a UV eraser for a period of time, typically ranging from 5 to 30 minutes, after which the device can be programmed again. For the UV light to reach the chip, EPROMs come with a small quartz window in the package through which the chip can be seen, as shown in Figure 5.5(d). For this reason, EPROM is often referred to as a *windowed* ROM device. EPROMs can typically be erased and reprogrammed thousands of times, and standard EPROMs are guaranteed to hold their programs for at least 10 years.

Compared with OTP ROM, EPROMs have improved write ability, as illustrated in Figure 5.2, since they can be erased and reprogrammed thousands of times. However, they have reduced storage permanence, since they are guaranteed to hold a program only for about 10 years, and their stored bits are susceptible to undesired changes if the chip is used in environments with much electrical noise or radiation. Thus, use of EPROMs in production parts is limited. If used in production, EPROMs should have their windows covered by a sticker to reduce the likelihood of undesired changes of the memory.



(a) Initially, the negative charges form a channel between the source and drain of the transistor storing a logic 1 at that cell's location.

(b) By applying a large positive voltage at the gate of the transistor, the negative charges move out of the channel area and get trapped in the floating gate, storing a logic 0 at that cell's location.

(c) By shining UV rays on the surface of the floating-gate, the negative charges move down into the channel restoring the logic 1 at the cell's location.

(d) An EPROM package with a quartz window through which UV light can pass.

Figure 5.5: EPROM internals.

EEPROM — Electrically Erasable Programmable ROM

Electrically erasable PROM, or EEPROM, developed in the early 1980s, was designed to eliminate the time-consuming and sometimes impossible requirement of exposing an EPROM to UV light to erase the ROM. An EEPROM is not only programmed electronically, but it is also erased electronically, typically by using higher than normal voltage. Such electronic erasing typically only requires seconds, rather than the many minutes required for EPROMs. Furthermore, EEPROMs can have individual words erased and reprogrammed, whereas

EPROMs can only be erased in their entirety. EEPROMs are typically more expensive than EPROMs, but far more convenient to use. EEPROMs are often called E^{2}s, pronounced "E-squareds."

Because EEPROMs can be erased and programmed electronically, we can build the circuit providing the higher-than-normal voltage levels for such electronic erasing and programming right into the embedded system in which the EEPROM is being used. Thus, we can treat this as a memory that can be both read and written — a write to a particular word would consist of erasing that word followed by programming that word. Thus, an EEPROM is in-system programmable. We can use it to store data that an embedded system should save after power is shut off. For example, EEPROM is typically used in telephones that can store commonly dialed phone numbers in memory for speed-dialing. If you unplug the phone, thus shutting off power, and then plug it back in, the numbers will still be in memory. EEPROMs can typically hold data for 10 years and can be erased and programmed tens of thousands of times before losing their ability to store data.

In-system programming of EEPROMs has become so common that many EEPROMs come with a built-in memory controller. A *memory controller* hides internal memory-access details from the memory user, and provides a simple memory interface to the user. In this case, the memory controller would contain the circuitry and single-purpose processor necessary for erasing the word at the user-specified address, and then programming the user-specified data into that word.

While read accesses may require only tens of nanoseconds, writes may require tens of microseconds or more, because of the necessary erasing and programming. Thus, EEPROMs with built-in memory controllers will typically latch the address and data, so that the writing processor can move on to other tasks. Furthermore, such an EEPROM would have an extra "busy" pin to indicate to the processor that the EEPROM is busy writing, meaning that a processor wanting to write to the EEPROM must check the value of this busy pin before attempting to write. Some EEPROMs support read accesses even while the memory is busy writing.

A common use of EEPROM is to serve as the program memory for a microprocessor. In this case, we may want to ensure that the memory cannot be in-system programmed. Thus, EEPROM typically comes with a pin that can be used to disable programming.

EEPROMs are more writable than EPROMs, as illustrated in Figure 5.2, since EEPROMs can be programmed in-system, and they are easier to erase. EEPROM is where the distinction between ROM and RAM begins to blur, since EEPROMs are in-system programmable and thus writable directly by a processor. Thus, the term "read-only-memory" for EEPROM is really a misnomer, since the processor can in fact write to an EEPROM. Such writes are slow compared to reads and are limited in number, but nevertheless, EEPROMs can and are commonly written by a processor during normal system operation.

Flash Memory

Flash memory is an extension of EEPROM that was developed in the late 1980s. While also using the floating-gate principle of EEPROM, flash memory is designed such that large blocks of memory can be erased all at once, rather than just one word at a time as in traditional EEPROM. A block is typically several thousand bytes large. This fast erase ability can vastly improve the performance of embedded systems where large data items must be stored in nonvolatile memory, systems like digital cameras, TV set-top boxes, cell phones, and medical monitoring equipment. It can also speed manufacturing throughput, since programming the complete contents of flash may be faster than programming a similar-sized EEPROM.

Like EEPROM, each block in a flash memory can typically be erased and reprogrammed tens of thousands of times before the block loses its ability to store data, and can store its data for 10 years or more.

A drawback of flash memory is that writing to a single word in flash may be slower than writing to a single word in EEPROM, since an entire block will need to be read, the word within it updated, and than the block written back.

Introduction to Read-Write Memory — RAM

We now turn our attention to a type of memory referred to as RAM. RAM, or random-access memory, is a memory that can be both read and written easily. Writing to a RAM is about as fast as reading from a RAM, in contrast to in-system programmable ROMs where writes take much longer than reads. Furthermore, RAM is typically volatile. Unlike forms of ROM, RAM never contains data when inserted in an embedded system. Instead, the system writes data to and then reads data from the RAM during its execution. Figure 5.1(b) provides a block diagram of a RAM.

A common question is, where does the term *random-access* come from in the name random-access memory? RAM should really be called read-write memory, to contrast it from read-only memory. However, when RAM was first introduced, it was in stark contrast to the then-common sequentially accessed memory media, like magnetic tapes or drums. These media required that the particular location to be accessed be positioned under an access device (e.g., a head). To access another location not immediately adjacent to the current location on

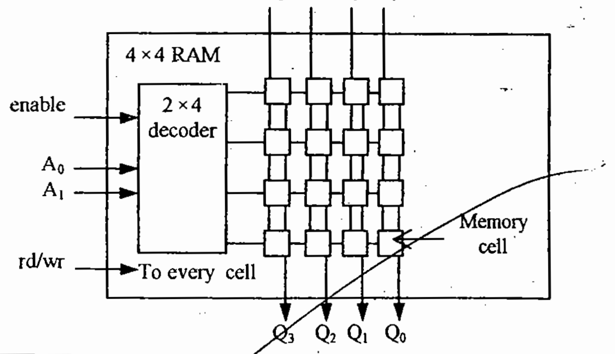


Figure 5.6: RAM internals.

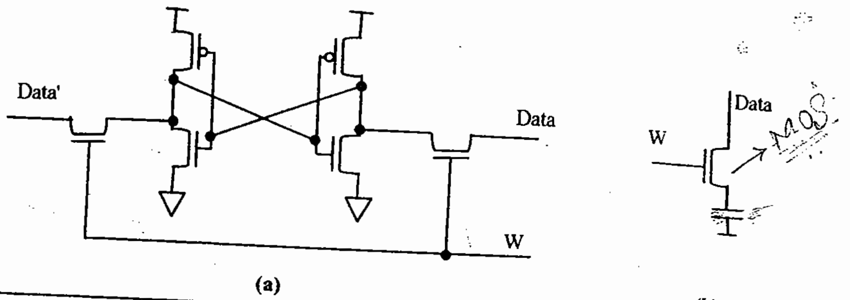


Figure 5.7: Memory cell internals: (a) SRAM, (b) DRAM.

the media, one would have to sequence through a number of other locations. For example, a tape would have to be rewound or fast-forwarded. In contrast, with RAM, any "random" memory location could be accessed in the same amount of time as any other location, regardless of the previously accessed location. This random-access feature was the key distinguishing feature of this memory type at the time of its introduction, and the name has stuck even today.

A RAM's internal structure is somewhat more complex than a ROM's, as shown in Figure 5.6, which illustrates a 4 × 4 RAM. (*Note*: RAMs typically have thousands of words, not just four as in the figure.) Each word consists of a number of memory cells, each storing 1 bit. In the figure, each input data line connects to every cell in its column. Likewise, each output data line connects to every cell in its column, with the output of a memory cell being ORed with the output data line from above. Each word enable line from the decoder connects to every cell in its row. The read/write input (*rd/wr*) is assumed to be connected to every cell. The memory cell must possess logic such that it stores the input data bit when *rd/wr* indicates write and the row is enabled, and such that it outputs this bit when *rd/wr* indicates read and the row is enabled.

There are two basic types of RAM, static and dynamic. Static RAM is faster but larger than dynamic RAM. Furthermore, static RAM is easily implemented on the same IC as processors, whereas dynamic RAM is usually implemented on a separate IC.

SRAM — Static RAM

Static RAM, or SRAM, uses a memory cell, shown in Figure 5.7(a), consisting of a fl[illegible] [illegible] to store a bit. Each bit thus requires about six transistors. This RAM type is called [illegible] because it will hold its data as long as power is supplied, in contrast to dynamic RAM. [illegible] RAM is typically used for high-performance parts of a system (e.g., cache).

[illegible]

DRAM — Dynamic RAM

Dynamic RAM, or DRAM, uses a memory cell, shown in Figure 5.7(b), consisting of a MOS transistor and capacitor to store a bit. Each bit thus requires only one transistor, resulting in more compact memory than SRAM. However, the charge stored in the capacitor leaks gradually, leading to discharge and eventually to loss of data. To prevent loss of data, each cell must regularly have its charge "refreshed." A typical DRAM cell's minimum refresh rate is once every 15.625 microseconds. Because of the way DRAMs are designed, reading a DRAM word refreshes that word's cells. In particular, accessing a DRAM word results in the word's data being stored in a buffer and then being written back to the word's cells. DRAMs tend to be slower to access than SRAMs.

PSRAM — Pseudo-Static RAM

Many RAM variations exist. *Pseudo-static* RAMs, or PSRAMs, are DRAMs with a memory refresh controller built-in. Thus, since the RAM user need not worry about refreshing, the device appears to behave much like an SRAM. However, in contrast to true SRAM, a PSRAM may be busy refreshing itself when accessed, which could slow access time and add some system complexity. Nevertheless, PSRAM is a popular low-cost high-density memory alternative to SRAM in many embedded systems.

NVRAM — Nonvolatile RAM

Nonvolatile RAM, or NVRAM, is a special RAM variation that is able to hold its data even after external power is removed. There are two common types of NVRAM.

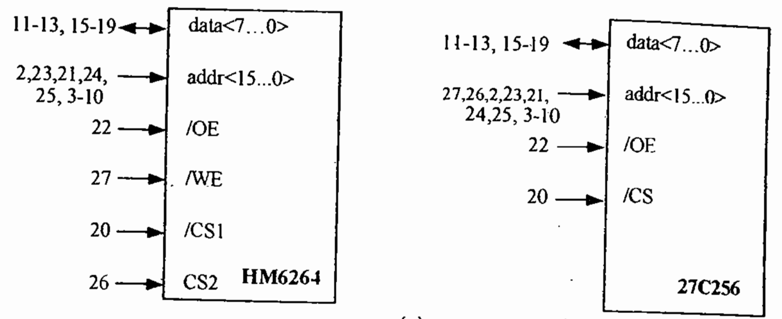
One type, often called *battery-backed RAM*, contains a static RAM along with its own permanently connected battery. When external power is removed or drops below a certain threshold, the internal battery maintains power to the SRAM, and thus the memory continues to store its bits. Compared with other forms of nonvolatile memory, battery-backed RAM is far more writable, as illustrated in Figure 5.2. Since no special programming is necessary, writes are done in nanoseconds, just like reads. Furthermore, unlike ROM-based forms of nonvolatile memory, battery-backed RAM imposes no limits on the number of times it can be written to. Storage permanence is obviously better than SRAM or DRAM, with many NVRAMs having batteries that can last for 10 years. However, NVRAMs are more susceptible to having bits changed inadvertently due to noise than are EEPROM or flash.

A second type of NVRAM contains a static RAM as well as an EEPROM or flash having the same capacity as the static RAM. This type of NVRAM stores its complete RAM contents into the EEPROM just before power is turned off, or whenever instructed to store the data, and then reloads that data from EEPROM into RAM after power is turned back on.

Example: HM6264 and 27C256 RAM/ROM Devices

In this example, we introduce a pair of low-cost low-capacity memory devices, shown in Figure 5.8(a), commonly used in 8-bit microcontroller-based embedded systems. The first two numeric digits in these devices indicate whether the device is RAM (62) or ROM (27).

PSRAM

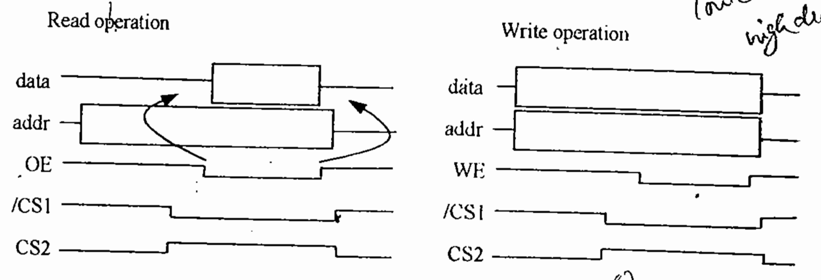


(a)

Device	Access Time (ns)	Standby Pwr. (mW)	Active Pwr. (mW)	Vcc Voltage (V)
HM6264	85-100	.01	15	5
27C256	90	.5	100	5

(b)

low cost high density

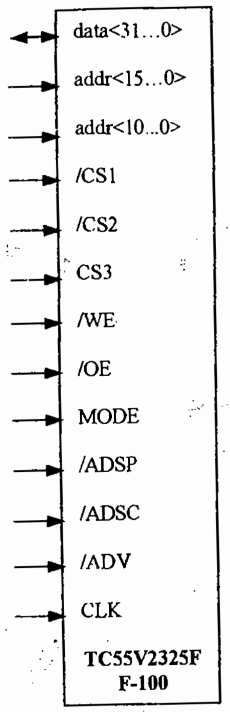


(c)

Figure 5.8: HM6264 and 27C256 RAM/ROM devices: (a) block diagram, (b) characteristics, (c) timing diagrams.

Subsequent digits give the memory capacity in kilobits. Both these devices are available in 4, 8, 16, 32, and 64 kilobytes, so the part numbers 62 or 27 would be followed by the number of kilobits, which may be 32, 64, 128, 256, or 512. Figure 5.8(b) summarizes some of the characteristics of these devices.

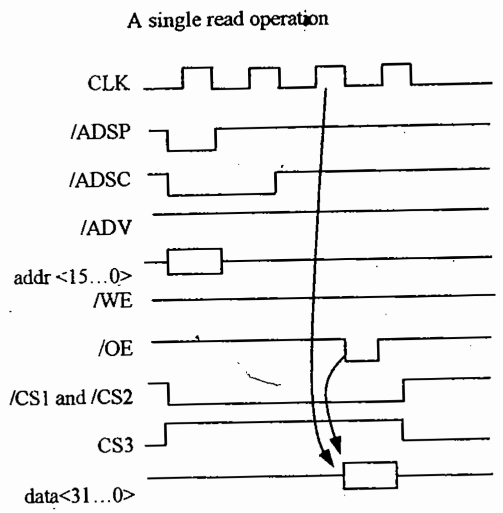
Memory access to and from these devices is performed through an 8-bit parallel protocol. Placing a memory address on the address-bus and asserting the read signal output enable (OE) performs a read operation. Placing some data and a memory address on the data and address busses and asserting the write signal write enable (WE) performs a write operation. The read and write timing is given in Figure 5.8(c).



(a)

Device	Access Time (ns)	Standby Pwr. (mW)	Active Pwr. (mW)	Vcc Voltage (V)
TC55V23 25FF-100	10	na	1200	3.3

(b)



(c)

Figure 5.9: TC55V2325FF-100 RAM devices: (a) block diagram, (b) characteristics, (c) timing diagrams.

Example: TC55V2325FF-100 Memory Device

In this example, we introduce a 2-megabit synchronous pipelined burst SRAM memory device, shown in Figure 5.9(a), designed to be interfaced with 32-bit processors. This device, made by Toshiba Inc., is organized as 64K × 32 bits. Figure 5.9(b) summarizes some of the characteristics of this device.

In Figure 5.9(c), we present the timing diagram for a single read operation. Write operation is similar. This device is capable of fast sequential reads and writes as well as single

byte I/O. The interested reader should refer to the manufacturer's datasheets for complete timing information. The read operation can be initiated with either the address status processor (*ADSP*) input or the address status controller (*ADSC*) input. Here, we have asserted both. Subsequent burst addresses can be generated internally and are controlled by the address advance (*ADV*) input. In other words, as long as *ADV* is asserted, the device will keep incrementing its address register and output the corresponding data on the next clock cycle.

5.4 Composing Memory

An embedded system designer is often faced with the situation of needing a particular-sized memory (ROM or RAM), but having readily available memories of a different size. For example, the designer may need a $2^k \times 8$ ROM, but may have $4^k \times 16$ ROMs readily available. Alternatively, the designer may need a $4^k \times 16$ ROM, but may have $2^k \times 8$ ROMs available for use. needed lower order

The case where the available memory is larger than needed is easy to deal with. We simply use the needed lower words in the memory, thus ignoring unneeded higher words and their high-order address bits, and we use the lower data input/output lines, thus ignoring unneeded higher data lines. Of course, we could use the higher data lines and ignore the lower lines instead.

The case where the available memory is smaller than needed requires more design effort. In this case, we must compose several smaller memories to behave as the larger memory we need. Suppose the available memories have the correct number of words, but each word is not wide enough. In this case, we can simply connect the available memories side-by-side. For example, Figure 5.10(a) illustrates the situation of needing a ROM three-times wider than that available. We connect three ROMs side-by-side, sharing the same address and enable lines among them, and concatenating the data lines to form the desired word width.

Suppose instead that the available memories have the correct word width, but not enough words. In this case, we can connect the available memories top to bottom. For example, Figure 5.10(b) illustrates the situation of needing a ROM with twice as many words, and hence needing one extra address line, than that available. We connect the ROMs top to bottom, ORing the corresponding data lines of each. We use the extra high-order address line to select the higher or lower ROM using a 1×2 decoder, and the remaining address lines to offset into the selected ROM. Since only one ROM will ever be enabled at a time, the ORing of the data lines never actually involves more than one nonzero data line.

If we instead needed four times as many words, and hence two extra address lines, we would instead use four ROMs. A 2×4 decoder having the two high-order address line as input would select one of the four ROMs to access.

Suppose the available memories have a smaller word width as well as fewer words than necessary. We then combine the above two techniques, first creating the number of columns of memories necessary to achieve the needed word width, and then creating the number of rows of memories necessary, along with a decoder, to achieve the needed number of words. The approach is illustrated in Figure 5.10(c).

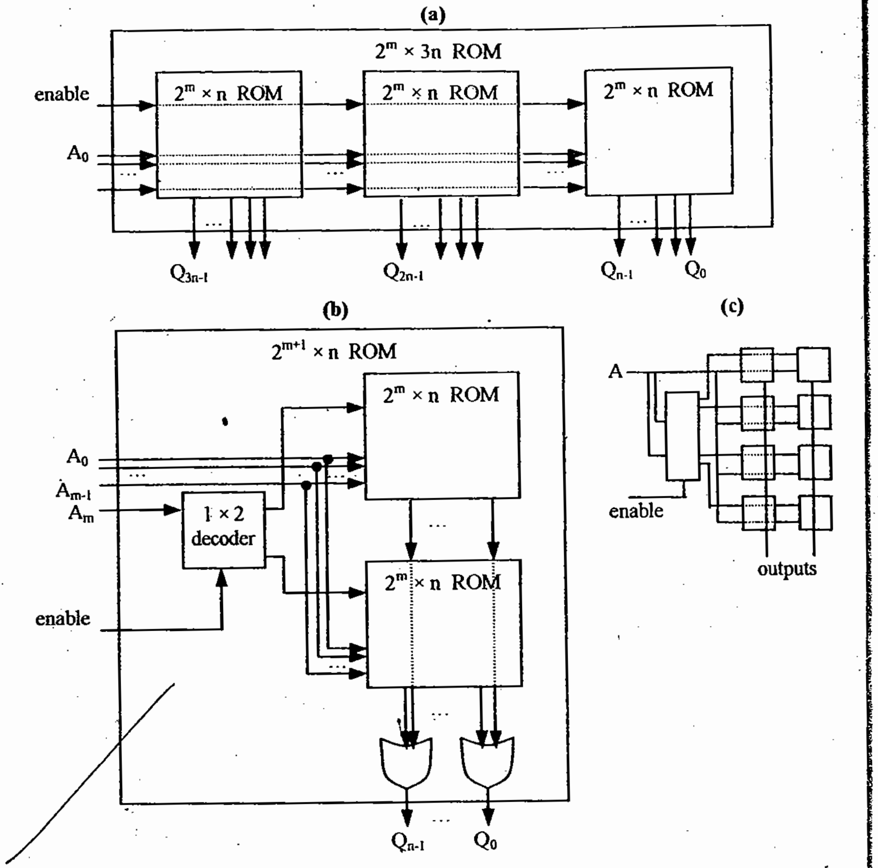
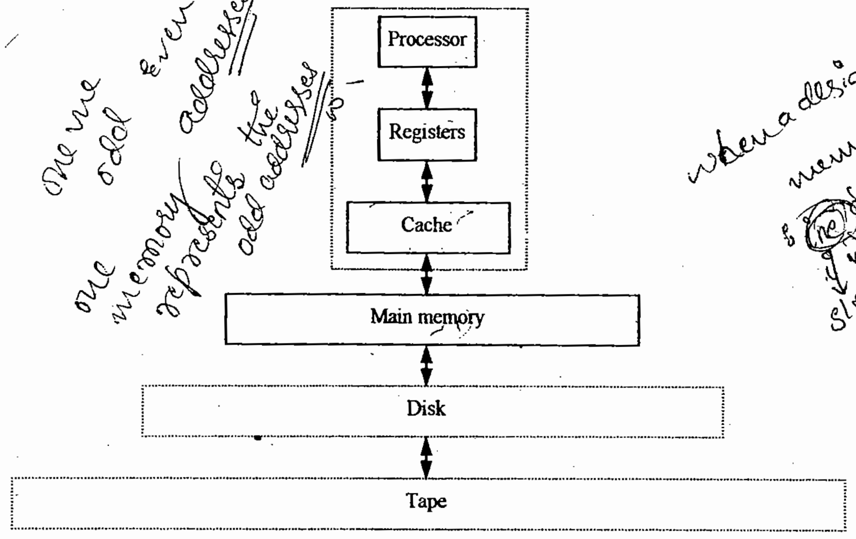


Figure 5.10: Composing smaller memory parts into larger memory.

Note that, when composing memories to increase the number of words, we don't necessarily have to use the highest-order address lines to select the appropriate memory, although these are the most logical choice. Sometimes, especially when we are composing just two memories, we use the lowest order bit to select among memories — thus, one memory represents "odd" addresses, and the other represents "even" addresses.

One we odd even addresses
one memory represents the odd addresses



when a design a memory.
[illegible] f.m.
slow, Ex.

Figure 5.11: An example memory hierarchy.

Copies of likely access

5.5 Memory Hierarchy and Cache

When we design a memory to store an embedded system's program and data, we are often faced with a dilemma: We want an inexpensive and fast memory, but inexpensive memory tends to be slow, whereas fast memory tends to be expensive. The solution to this problem is to create a memory hierarchy, as illustrated in Figure 5.11. We use an inexpensive but slow main memory to store all of the program and data. We use a small amount of fast but expensive cache memory to store copies of likely accessed parts of main memory. Using cache is analogous to posting on a wall near a telephone a short list of important phone ~~numbers rather than posting the entire phonebook.~~

~~Some systems~~ include even larger and less expensive forms of memory, such as ~~disk and~~ tape, for some of their storage needs. However, we do not consider these further as they are not especially common in embedded systems. Also, although the figure shows only one cache, we can include any number of levels of cache, those closer to the processor being smaller and faster than those closer to main memory. A two-level cache scheme is common.

Cache is usually designed using static RAM rather than dynamic RAM, which is one reason that cache is more expensive but faster than main memory. Because cache usually appears on the same chip as a processor, where space is very limited, cache size is typically only a fraction of the size of main memory. Cache access time may be as low as just one clock cycle, whereas main memory access time is typically several cycles.

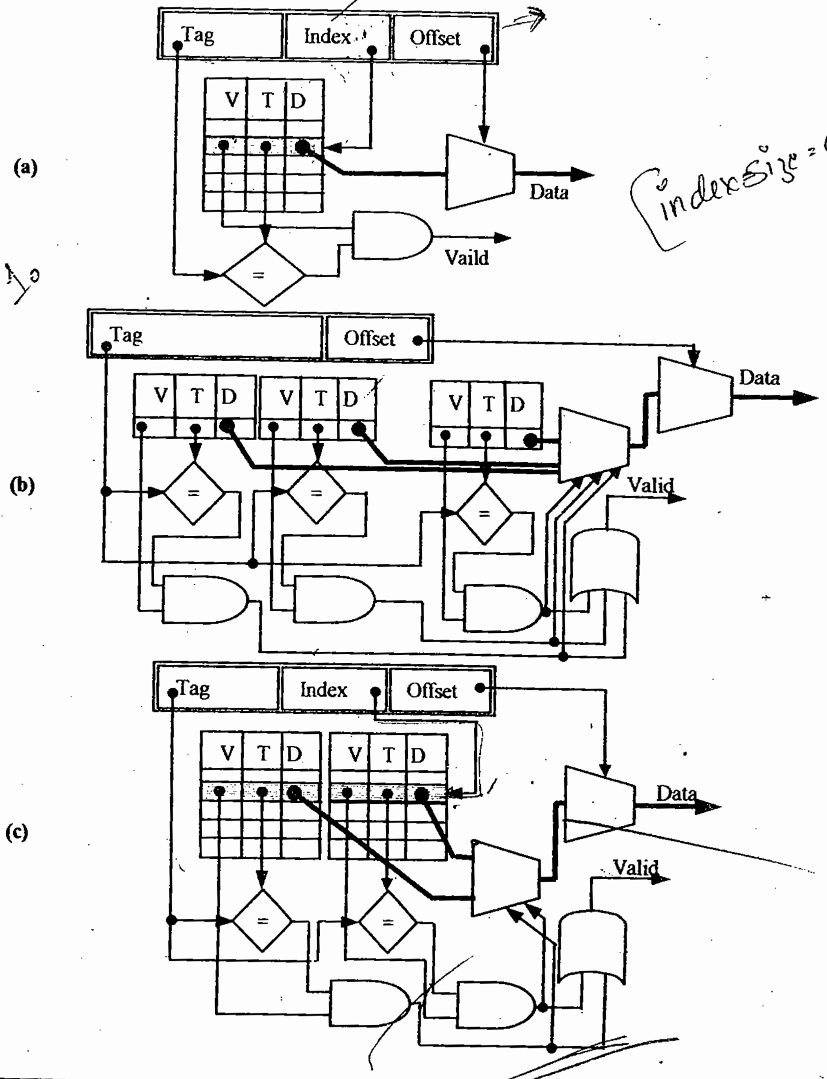
A cache operates as follows. When we want the processor to access (read or write) a main memory address, we first check for a copy of that location in cache. If the copy is in the cache, called a *cache hit*, then we can access it quickly. If the copy is not there, called a *cache miss*, then we must first read the address and perhaps some of its neighbors into the cache. This description of cache operation leads to several cache design choices: cache mapping, cache replacement policy, and cache write techniques. These design choices can have significant impact on system cost, performance, as well as power, and thus should be evaluated carefully for a given application.

Cache Mapping Techniques

Cache mapping is the method for assigning main memory addresses to the far fewer number of available cache addresses, and for determining whether a particular main memory address's contents are in the cache. Cache mapping can be accomplished using one of three basic techniques (see Figure 5.12):

1. In *direct mapping*, illustrated in Figure 5.12(a), the main memory address is divided into two fields, the index and the tag. The *index* represents the cache address, and thus the number of index bits is determined by the cache size (i.e., index size = $\log_2$(cache size)). Note that many different main memory addresses will map to the same cache address. When we store the contents of a main memory address in the cache, we also store the *tag*. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and compare the tag there with the desired tag. If the tags match, then we check the valid bit. The *valid bit* indicates whether the data stored in that cache slot has previously been loaded into the cache from the main memory. We use the *offset* portion of the memory address to grab a particular word within the cache line. A cache *line*, also known as a cache *block*, is the number of (inseparable) adjacent memory addresses loaded from or stored into main memory at a time. A typical block size is four or eight addresses.
2. In *fully associative mapping*, illustrated in Figure 5.12(b), each cache address contains not only the contents of a main memory address, but also the complete main memory address. To determine if a desired main memory address is in the cache, we simultaneously (associatively) compare all the addresses stored in the cache with the desired address.
3. In *set-associative mapping*, illustrated in Figure 5.12(c), a compromise is reached between direct and fully associative mapping. As in direct mapping, an index maps each main memory address to a cache address, ~~but now each cache address~~ contains the content and tags of two or more memory locations, namely, a set of entries. To determine if a desired main memory address is in the cache, ~~we go to the~~ cache address indicated by the index, and we then simultaneously (associatively) compare all the tags at that location (i.e., of that set) with the desired tag. A cache with a set of size N is called an N-way set-associative cache; 2-way, 4-way, and 8-way set associative caches are common.

C addr



(index size = Cache (size))

Figure 5.12: Cache mapping techniques: (a) direct-mapped, (b) fully associative, (c) two-way set associative.

Direct-mapped caches are easy to implement, but may result in numerous misses if two or more words with the same index are accessed frequently, since each will bump the other out of the cache. Fully associative caches on the other hand are fast, but the comparison logic is expensive to implement. Set-associative caches can reduce misses compared to direct-mapped caches, without requiring nearly as much comparison logic as fully associative caches.

Caches are usually designed to treat collections of a small number of adjacent main-memory addresses as one indivisible *block*, also known as a *line*, typically consisting of about eight addresses.

Cache-Replacement Policy

The cache replacement policy is the technique for choosing which cache block to replace when a fully associative cache is full, or when a set-associative cache's line is full. Note that there is no choice in a direct-mapped cache; a main-memory address always maps to the same cache address and thus replaces whatever block is already there. There are three common replacement policies. A *random* replacement policy chooses the block to replace randomly. While simple to implement, this policy does nothing to prevent replacing a block that is likely to be used again soon. A *least-recently used* (LRU) replacement policy replaces the block that has not been accessed for the longest time, assuming that this means that it is least likely to be accessed in the near future. This policy provides for an excellent hit/miss ratio but requires expensive hardware to keep track of the times blocks are accessed. A *first-in-first-out* (FIFO) replacement policy uses a queue of size-*N*, pushing each block address onto the queue when the address is accessed, and then choosing the block to be replaced by popping the queue.

Cache Write Techniques

When we write to a cache, we must at some point update the main memory. Such update is only an issue for data cache, since instruction cache is read-only. There are two common update techniques, write-through and write-back.

In the *write-through* technique, whenever we write to the cache, we also write to main memory, requiring the processor to wait until the write to main memory completes. Although easy to implement, this technique may result in several unnecessary writes to main memory. For example, suppose a program writes to a block in the cache, then reads it, and then writes it again, with the block staying in the cache during all three accesses. There would have been no need to update the main memory after the first write, since the second write overwrites this first write.

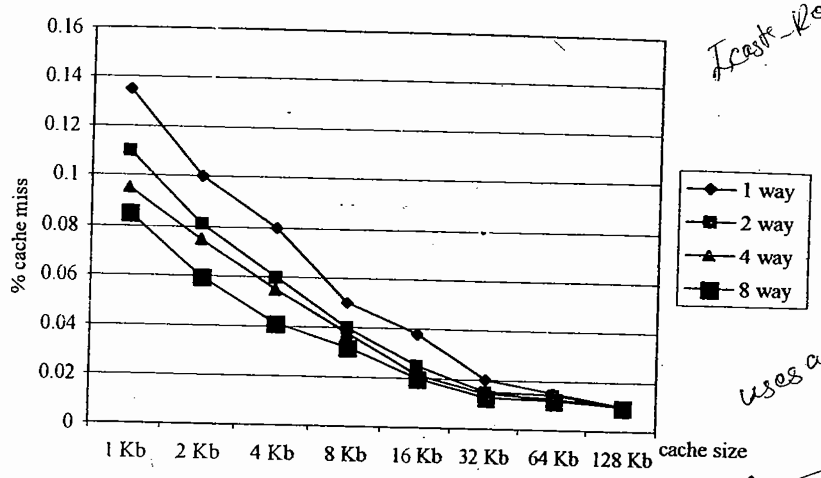
The *write-back* technique reduces the number of writes to main memory by writing a block to main memory only when the block is being replaced, and then only if the block was written to during its stay in the cache. This technique requires that we associate an extra bit, called a dirty bit, with each block. We set this bit whenever we write to the block in the cache, and we then check it when replacing the block to determine if we should copy the block to main memory.

Cache Impact on System Performance

The design and configuration of caches can have a large impact on performance and power consumption of a system. So far, we looked at cache mapping, associativity, write back, and replacement policies. From a performance point of view, the most important parameters in

hit / miss

Least_Recently.



uses a queue of N size.

Figure 5.13: Sample cache performance trade-offs.

cache design are the total size of the cache, its degree of associativity, and the data block size, a.k.a., line size, that is read or written during each cache access by the microprocessor.

The total size of the cache is measured as the total number of data bytes that the cache can hold. Notice that a cache stores other information, such as the tags and valid bits, which do not contribute to the size of the cache. So, a 32 Kbyte cache has room for 32,678 bytes of data, plus additional storage for tag and house keeping bits. By making a cache larger, one achieves lower miss rates, which is one of the design goals. However, accessing words within a large cache will be slower than accessing words in a smaller cache. To clarify this, we will give an example. First, let us assume that we are designing a small 2 Kbyte cache for our processor. With this cache, we have measured the miss rate to be 15%, meaning 15 out of every 100 accesses to the cache result in a miss on the average. The cost of going to main memory (i.e., the cost of memory access when there is a miss) is 20 cycles. The cost of going only to the cache (i.e., the cost of memory access when there is a hit) is two cycles. Hence, on the average, the cost of a memory access is (0.85 * 2) + (0.15 * 20) = 4.7 cycles. Now let us double the size of the cache, and assume this improves our hit rate to 93.5%, at the expense of slowing the cache down by an extra clock cycle. Now, the average cost of a memory access becomes (0.935 * 3) + (0.065 * 20) = 4.105 cycles. This second cache will perform better than our first one. Now, we double the size of our cache one more time, resulting in an additional clock cycle per hit but achieving 94.435% improvement in terms of hit rate. The average cost of memory access, thus, becomes (0.94435 * 4) + (0.05565 * 20) = 4.8904 cycles. This larger cache will perform worse than our first two designs.

Note that the problem of making a cache larger is additional access time penalty, which quickly offsets the benefits of improved hit rates. Designers often use other methods to improve cache hit rate without increasing the cache size. For example, they make a cache set associative or increase the line size. These methods too incur additional logic and add to the access time latency. Increasing the line size can, additionally, improve main memory access time, at the expense of more complex multiplexing of data and thus increased access latency. Figure 5.13 summarizes the effects of cache size and associativity in terms of average miss rate for a number of commonly used programs under the Unix environment, such as gcc.

The behavior of caches is very dependent on the type of applications that run on the processor. Fortunately, for an embedded system, the set of applications are well defined and known at design time, so the designer has the ability to measure the performance of some candidate cache designs and choose one that best meets the performance, cost, and power constraints. One way to perform such analysis is as follows. We instrument the executable with additional code such that, when executed, it outputs a trace of memory references. Then, we feed these traces through a cache simulator, which outputs cache statistics at the end of its execution. We can perform all this analysis on our development computer.

5.6 Advanced RAM

Earlier we described a DRAM as a type of storage device that uses a single transistor/capacitor pair to store a bit. Because of such architecture and the resulting high capacity and low cost, DRAMs are commonly used as the main memory in processor based embedded systems. In order for DRAMs to keep pace with processor speeds, many variations on the basic DRAM interface has been proposed. In this section, we describe the structure of a basic DRAM as well as some of the more recent and advanced DRAM designs.

The Basic DRAM

The basic DRAM architecture is depicted in Figure 5.14. The addressing mechanism for a memory read works as follow. The address bus is multiplexed between row and column components. Using the row address select (*ras*) signal, the row component of the address is latched into the row address buffer. Likewise, using the column address select (*cas*) signal, the column component of the address is latched into the column address buffer. (Note that in earlier days, the number of I/O pins were limited, hence manufacturers of DRAMs adopted this multiplexed scheme to reduce the overall I/O requirements. In fact, some DRAM devices used the same I/O pins for multiplexed data as well as multiplexed address signals.) As soon as the row address component is latched into the row address buffer, the row decoder activates an entire row of bits. The length of this bit-row depends on the word size and column address space. Once the column address buffer is latched, the column decoder enables the particular word (referred by the address) in order for it to propagate to the sense amplifier. The sense amplifier's task is to detect the voltage level of the bits (transistor/capacitor pairs) corresponding to the referenced word and amplify them to a high enough level for latching into the output buffers. Once the data is in the output buffers, it can be read by asserting the

"3 questions" "3 questions"

(8d)

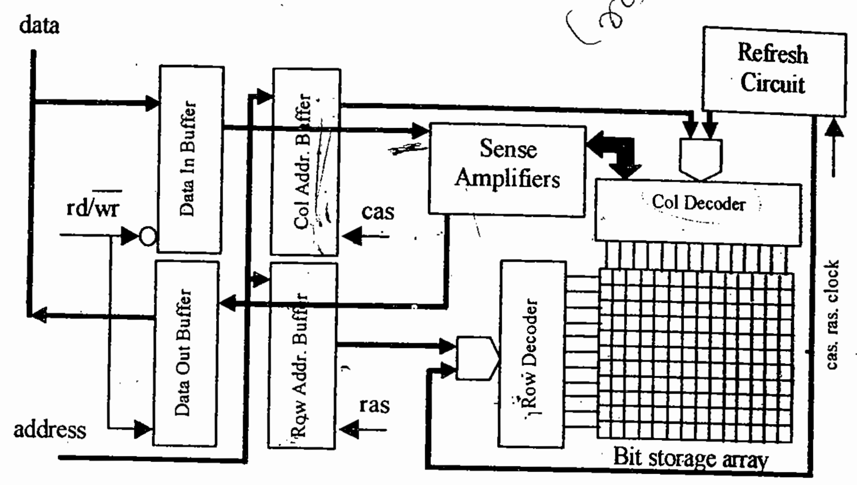


Figure 5.14: Basic DRAM architecture.

data bus read (*rd*) signal. Memory write works in the reverse order. First the word is written to the input buffers using the data bus write (*wr*) signal. Then, the row and column addresses are latched in, sequentially, by strobing the *ras* and *cas* signals, respectively.

Figure 5.14 also depicts the refresh circuitry. As noted earlier in this chapter, due to discharge of the capacitors, the content of a DRAM must be periodically read and written back. This is accomplished by the refresh circuitry (which may be internal or external to the particular DRAM device). An external clock drives the refresh circuitry. Periodically, the refresh circuitry strobes consecutive memory addresses, causing the memory content to be refreshed. (Recall that selecting a row of bits through the row decoder will automatically charge the capacitors in that row.) The refresh circuitry becomes disabled during a memory read or write operation (i.e., when any of the *ras* or *cas* signals are asserted).

m.add.

Fast Page Mode DRAM (FPM DRAM)

The fast page mode DRAM design is an improvement on the basic DRAM architecture. In this design, each row of the memory bit-array is viewed as a page. A page contains multiple words. Each word is addressed by a different column address. The sense amplifier in FPM DRAM amplifies the entire page once its address is strobed into the row address latch. Thereafter, each word of that page is read (or written) by strobing the corresponding column address. The timing diagram for FPM DRAM is depicted in Figure 5.15. Here, after selecting a particular page (row), three data words within that page are read consecutively. The page

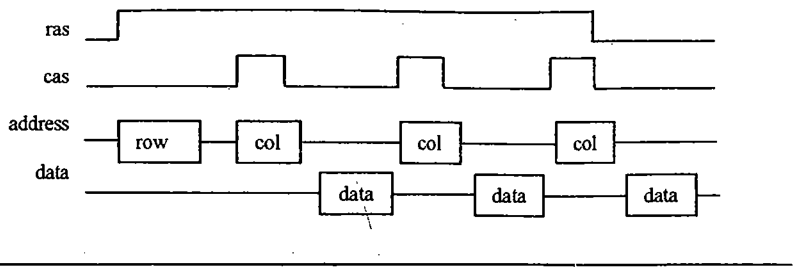


Figure 5.15: FPM DRAM timing.

mode design eliminates an extra cycle on each read/write of words from the same page when compared to the basic DRAM design.

Extended Data Out DRAM (EDO DRAM)

The extended data out DRAM design is an improvement on the FPM DRAM architecture. In this design, an extra output latch is added between the sense-amplifier and the output buffers. This allows overlapping of the column select and data read operations. In other words, while a previously selected word is being read from the output buffer, a new column address can be strobed by asserting the *cas* signal. The timing diagram for EDO DRAM is depicted in Figure 5.16. The extra overlap reduces the read/write latency by an additional cycle when compared to FPM DRAM.

Synchronous (S) and Enhanced Synchronous (ES) DRAM

The FPM and EDO RAM architectures described so far are controlled asynchronously by the processor or the memory controller. This means that a transaction takes place when the *ras/cas* and *rd/wr* signals are asserted in appropriate order. An alternative is to make the interface to the DRAM synchronous such that the DRAM latches information to and from the controller on the active edge of the clock signal. A synchronous interface will eliminate a small amount of time (thus latency) that is needed by the DRAM to detect the *ras/cas* and *rd/wr* signals. This architecture is referred to as *synchronous* DRAM, or SDRAM.

In addition to a lower latency I/O, after a proper page and column setup, an SDRAM may store the starting address internally and output new data on each active edge of the clock signal, as long as the requested data are consecutive memory locations. This is accomplished by adding a column address counter to the base DRAM architecture. This counter is seeded with a starting column address strobed in by the processor (or memory controller) and is thereafter incremented internally by the DRAM on each clock cycle. The timing diagram of a SDRAM device is depicted in Figure 5.17. Note that in this timing diagram, we have added the clock signal. The clock signal was not present in Figure 5.15 and Figure 5.16 because those DRAMs were asynchronous.

a faster clocking and lower

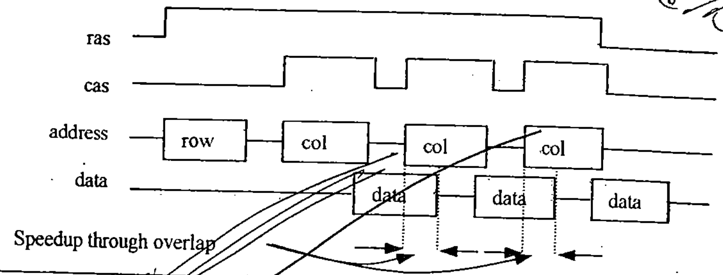


Figure 5.16: EDO DRAM timing.

LL7/0.

The enhanced synchronous DRAM, or ESDRAM, is an improvement to the SDRAM design. The improvement is analogous to that made to the FPM DRAM by the EDRAM. In short, caches (buffers) have been added to the sense amplifiers to enable overlapping of the column addressing. This enables faster clocking and lower latency in reading and writing data.

(column addressing)

Synchronous & Enhanced Synchronous DRAM

Rambus DRAM (RDRAM)

Rambus is really more of a bus interface architecture than DRAM architecture. Rambus uses multiplexed address/data lines to connect the memory controller (or processor) to the RDRAM device. The specification for this interface states that the clock runs at 300 MHz. In addition, data is latched on both rising and falling edge of the clock. Using such a bus, theoretically, a transfer rate of 600 million cycles is possible. In addition, each 64-Mbit RDRAM is broken into four banks (parts) each with its own row decoders. So, at any given time, four pages remain open. The RDRAM protocol is packet driven, where address packets are followed by data packets. The smallest transaction requires a minimum of four cycles. Because of its multiple open page schemes and fast bus I/O, RDRAM, when utilized properly, is capable of very high throughput.

DRAM Integration Problem

LL7/0.

So far, we have discussed static and dynamic types of RAMs and brought up the benefits and disadvantages of each type. In this section, we describe the problem of integrating memory and conventional logic (gates) on the same IC. While most static types of RAMs can easily be integrated with other logic on a single chip (e.g., ICs containing a cache and a microprocessor), it is very difficult to integrate DRAMs and conventional logic. The difficulty arises from the different chip making process that is involved when making DRAMs as opposed to conventional logic. When designing conventional logic ICs, the goal of the designers is to minimize the parasitic capacitance in order to reduce signal propagation delays and power consumption. In contrast, when designing DRAMs, the goal of the designers is to

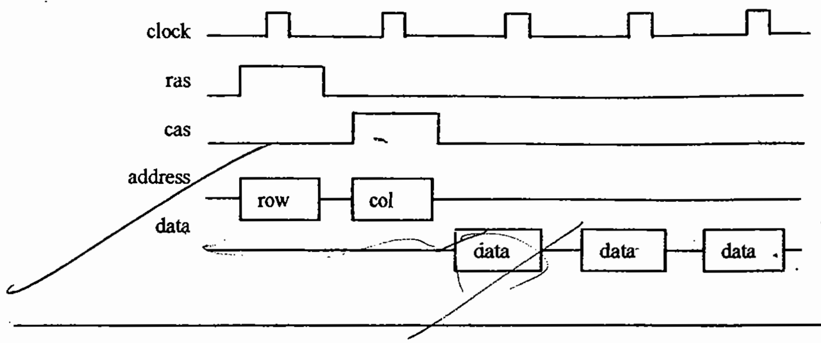


Figure 5.17: SDRAM timing.

create capacitor cells in order to retain stored information. This difference in design goal leads to a design process that is considerably different between DRAM and conventional logic. However, integrated processes are beginning to appear.

Memory Management Unit (MMU)

We conclude this section by briefly discussing the duties of a MMU. A system that contains DRAM requires some processor that handles tasks such as refresh, DRAM bus interface and arbitration and sharing of a memory among multiple processors. In addition, the MMU translates logical memory addresses, issued by attached processors, to physical memory addresses that make sense to the particular DRAM architecture in use. Modern CPUs often come with a MMU built as part of the processor's core. Otherwise, single-purpose processors can be designed or purchased to handle such memory management tasks.

5.7 Summary

Memory stores data for use by processors. We have categorized memory using two characteristics, namely, write ability and storage permanence. ROM typically is only read by an embedded system. It can be programmed during fabrication (mask-programmed) or by the user (programmable ROM, or PROM). PROM may be erasable using UV light (EPROM), or electronically erasable (EEPROM) word by word, or in large blocks (Flash). RAM, on the other hand, is memory that can be read or written by an embedded system. Static RAM uses a flip-flop to store each bit, while dynamic RAM uses a transistor and capacitor, resulting in fewer transistors but the need to refresh the charge on the capacitor and slower performance. Psuedo-static RAM is a dynamic RAM with a built-in refresh controller. Nonvolatile RAM keeps its data even after power is shut off. Designers must not only choose the appropriate type of memories for a given system, but must often compose smaller memory parts into larger memory. Using a memory hierarchy can improve system performance by keeping

copies of frequently accessed instructions/data in small, fast memories. Cache is a small and fast memory between a processor and main memory. Several cache design features greatly influence the speed and cost of cache, including mapping techniques, replacement policies, and write techniques. Several advanced DRAMs provide high-speed memory access, like FPM RAM, EDO RAM, and SDRAM. Integrating DRAM with on-chip processors can be difficult due to different IC processes. Thus, choice of memory types and design of a memory architecture is an important part of embedded system design, and can greatly impact performance, power, size, and cost.

5.8 References and Further Reading

- http://www.instantweb.com/~foldoc/contents.html, *The Free Online Dictionary of Computing*. Provides definitions of a variety of computer-related terms, including numerous ROM and RAM variations.
- David Patterson and John Hennessy, *Computer Organization and Design*. San Francisco, CA: Morgan Kaufmann Publishers, Inc. Includes discussion of memory hierarchy and cache.

5.9 Exercises

5.1 Briefly define each of the following: mask-programmed ROM, PROM, EPROM, EEPROM, flash EEPROM, RAM. SRAM, DRAM, PSRAM, and NVRAM.

5.2 Define the two main characteristics of memories as discussed in this chapter. From the types of memory mentioned in Excercise 5.1, list the worst choice for each characteristic. Explain.

5.3 Sketch the internal design of a 4 × 3 ROM.

5.4 Sketch the internal design of a 4 × 3 RAM.

5.5 Compose 1K x 8 ROMs into a 1K × 32 ROM (Note: 1K actually means 1,024 words).

5.6 Compose 1K x 8 ROMs into an 8K × 8 ROM.

5.7 Compose 1K x 8 ROMs into a 2K × 16 ROM.

5.8 Show how to use a 1K × 8 ROM to implement a 512 × 6 ROM.

5.9 Given the following three cache designs, find the one with the best performance by calculating the average cost of access. Show all calculations. (a) 4 Kbyte, 8-way set-associative cache with a 6% miss rate; cache hit costs one cycle, cache miss costs 12 cycles. (b) 8 Kbyte, 4-way set-associative cache with a 4% miss rate; cache hit costs two cycles, cache miss costs 12 cycles. (c) 16 Kbyte, 2-way set-associative cache with a 2% miss rate; cache hit costs three cycles, cache miss costs 12 cycles.

5.10 Given a 2-level cache design where the hit rates are 88% for the smaller cache and 97% for the larger cache, the access costs for a miss are 12 cycles and 20 cycles, respectively, and the access cost for a hit is one cycle, calculate the average cost of access.