

Chapter 5 Memories

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, *memories* to implement storage, and *buses* to implement communication. The earlier chapters described common processor types: general-purpose processors, standard single-purpose processors, and custom single-purpose processors. This chapter describes memories.

A memory stores large numbers of bits. These bits exist as m words of n bits each, for a total of $m*n$ bits. We refer to a memory as an $m \times n$ ("m-by-n") memory. $\log_2(m)$ address input signals are necessary to identify a particular word. Stated another way, if a memory has k address inputs, it can have up to 2^k words. n signals are necessary to output (and possibly input) a selected word. 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. Some memories can only be read from (ROM), while others can be both read from and written to (RAM). There isn't much demand for a memory that can only be written to (what purpose would such a memory serve?). Most memories have an enable input; when this enable is low, the address is ignored, and no data is written to or read from the memory.

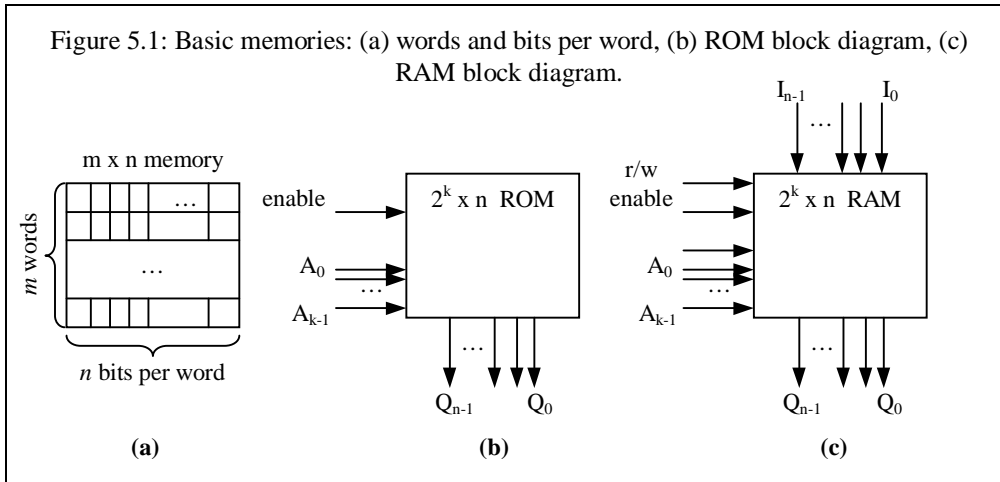
5.2 Read-only memory -- ROM

ROM, or read-only memory, is a memory that can be read from, but not typically written to, during execution of an embedded system. Of course, there must be a mechanism for setting the bits in the memory (otherwise, of what use would the read data serve?), but we call this "programming," not writing. Such programming is usually done off-line, i.e., when the memory is not actively serving as a memory in an embedded system. We usually program a ROM before inserting it into the embedded system. Figure 1(b) provides a block diagram of a 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.

Another 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 2.

Figure 3 provides a symbolic view of the internal design of an 8x4 ROM. To the right of the 3x8 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*



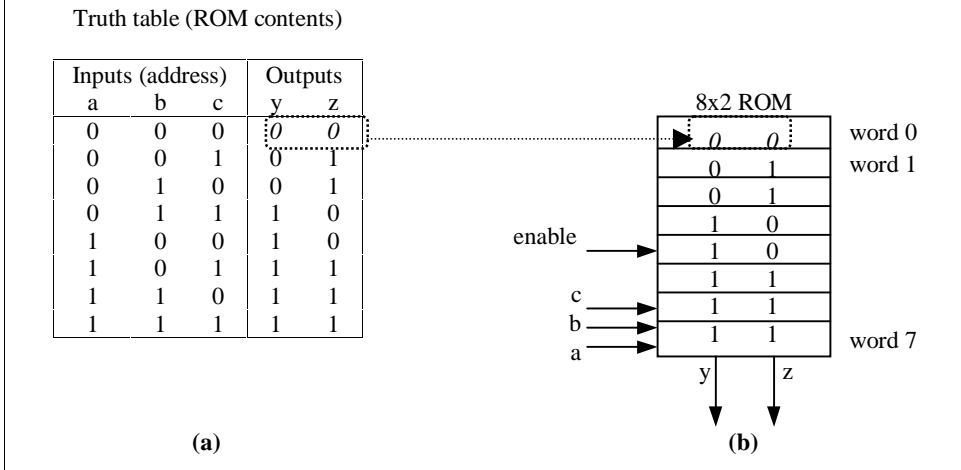
connected. Thus, word lines only connect to data lines via the programmable connection lines shown. The figure shows all connection lines in place except for two connections in word 2. To see how this device acts as a read-only memory, consider an input address of "010." The decoder will thus set word 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 is 0, then no word is read. 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. In a *mask-programmed* ROM, the connection is made when the chip is being fabricated (by creating an appropriate set of masks). Such ROM types are typically only used in high-volume systems, and only after a final design has been determined.

Most other systems use user-programmable ROM devices, or *PROM*, which can be programmed by the chip's user, well after the chip has been manufactured. These devices are better suited to prototyping and to low-volume applications. To program a PROM device, the user provides a file indicating the desired ROM contents. A piece of equipment called a ROM programmer (note: the programmer is a piece of equipment, not a person who writes software) then configures each programmable connection according to the file. A basic PROM uses a fuse for each programmable connection. 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 re-established. For this reason, basic PROM is often referred to as one-time-programmable device, or *OTP*.

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," meaning its gate is not connected. An EPROM programmer injects electrons into the floating gate, using higher than normal voltage (usually 12V to 25V) that causes electrons to "tunnel" into the gate. When that high voltage is removed, the electrons can not escape, and hence

Figure 5.2: Implementing combinational functions with a ROM: (a) truth table, (b) ROM contents.

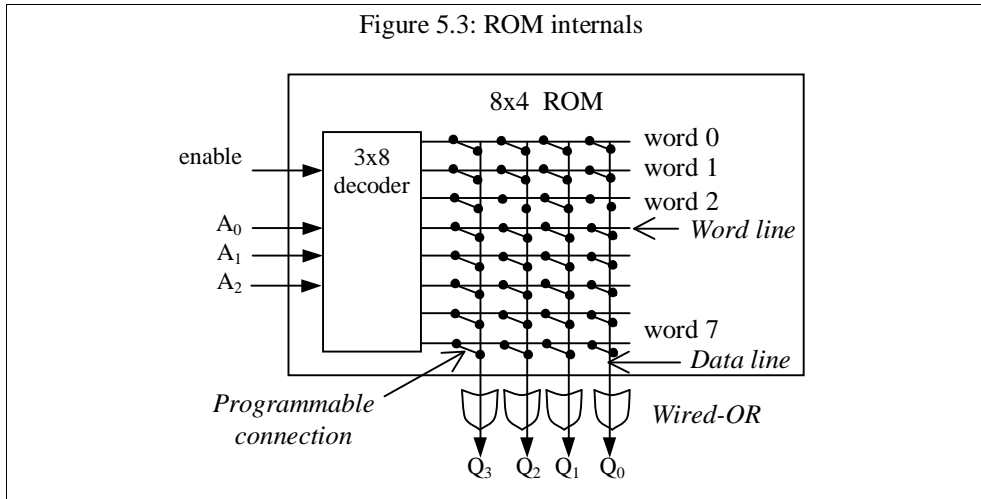


the gate has been charged and programming has occurred. Standard EPROMs are guaranteed to hold their programs for at least 10 years. To erase the program, the electrons must be excited enough to escape from the gate. Ultra-violet (UV) light is used to fulfil this role of erasing. The device must be placed under a UV eraser for a period of time, typically ranging from 5 to 30 minutes, after which the device can be programmed again. In order for the UV light to reach the chip, EPROM's come with a small quartz window in the package through which the chip can be seen. For this reason, EPROM is often referred to as a *windowed* ROM device.

Electrically-erasable PROM, or *EEPROM*, is 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 is also erased electronically. These devices are typically more expensive than EPROM's, but far more convenient to use. EEPROM's are often called "E-squared's" for short. *Flash* memory is a type of EEPROM in which reprogramming can be done to certain regions of the memory, rather than the entire memory at once.

Which device should be used during development? The answer depends on cost and convenience. For example, OTP's are typically quite inexpensive, so they are quite practical unless frequent reprogramming is expected. In that case, windowed devices are typically cheaper than E-squared's. However, if one can not (or does not want to) deal with the time required for UV erasing, or if one can not move the device to a UV eraser (e.g., if it's being used in a microcontroller emulator), then E-squared's may be used.

For final implementation in a product, masked-ROM may be best for high-volume production, since its high up-front cost can be amortized over the large number of products. OTP has the advantage of low cost as well as resistance to undesired program changes caused by noise. Windowed parts if used in production should have their windows covered by a sticker to prevent undesired changes of the memory.



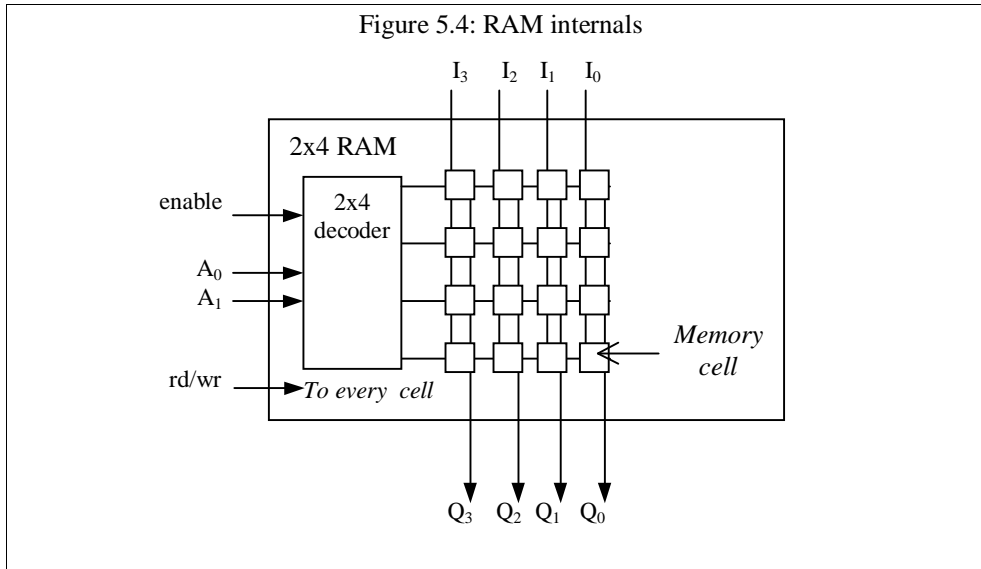
5.3 Read-write memory -- RAM

RAM, or random-access memory, is a memory that can be both read and written. In contrast to ROM, a RAM's content is not "programmed" before being inserted into an embedded system. Instead, the RAM contains no data when inserted in the embedded system; the system writes data to and then reads data from the RAM during its execution. Figure 1(c) provides a block diagram of a RAM.

A RAM's internal structure is somewhat more complex than a ROM's, as shown in Figure 4, which illustrates a 4x4 RAM (note: RAMs typically have thousands of words, not just 4 as in the figure). Each word consists of a number of memory cells, each storing one bit. In the figure, each input data 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 OR'ed 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 bigger than dynamic RAM. *Static* RAM, or *SRAM*, uses a memory cell consisting of a flip-flop to store a bit. Each bit thus requires about 6 transistors. This RAM type is called static because it will hold its data as long as power is supplied, in contrast to dynamic RAM. Static RAM is typically used for high-performance parts of a system (e.g., cache).

Dynamic RAM, or *DRAM*, uses a memory cell consisting of a MOS transistor and capacitor to store a bit. Each bit thus requires only 1 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 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.

Many RAM variations exist. Pseudo-Static RAMs, or PSRAMs, are DRAMs with a 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 alternative to SRAM in many embedded systems.

Non-volatile RAM, or *NVRAM*, is another RAM variation. Non-volatile storage is storage that can hold its data even after power is no longer being supplied. Note that all forms of ROM are non-volatile, while normal forms of RAM (static or dynamic) are volatile. One type of NVRAM contains a static RAM along with its own permanently connected battery. A second type contains a static RAM and its own (perhaps flash) EEPROM. This type stores RAM data into the EEPROM just before power is turned off (or whenever instructed to store the data), and reloads that data from EEPROM into RAM after power is turned back on. NVRAM is very popular in embedded systems. For example, a digital camera must digitize, store and compress an image in a fraction of a second when the camera's button is pressed, requiring writes to a fast RAM (as opposed to programming of a slower EEPROM). But it also must store that image so that the image is saved even when the camera's power is shut off, requiring EEPROM. Using NVRAM accomplishes both these goals, since the data is originally and quickly stored in RAM, and then later copied to EEPROM, which may even take a few seconds.

Note that the distinction we made between ROM and RAM, namely that ROM is programmed before insertion into an embedded system while RAM is written by the embedded system, does not hold in every case. As in the digital camera example above, EEPROM may be programmed by the embedded system during execution, though such programming is typically infrequent due to its time-consuming nature.

A common question is: where does the term "random-access" come from in 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 the media, one would have sequence through a number of other locations, e.g., for a tape, one would have to rewind or fast-forward the tape. 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 read 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.

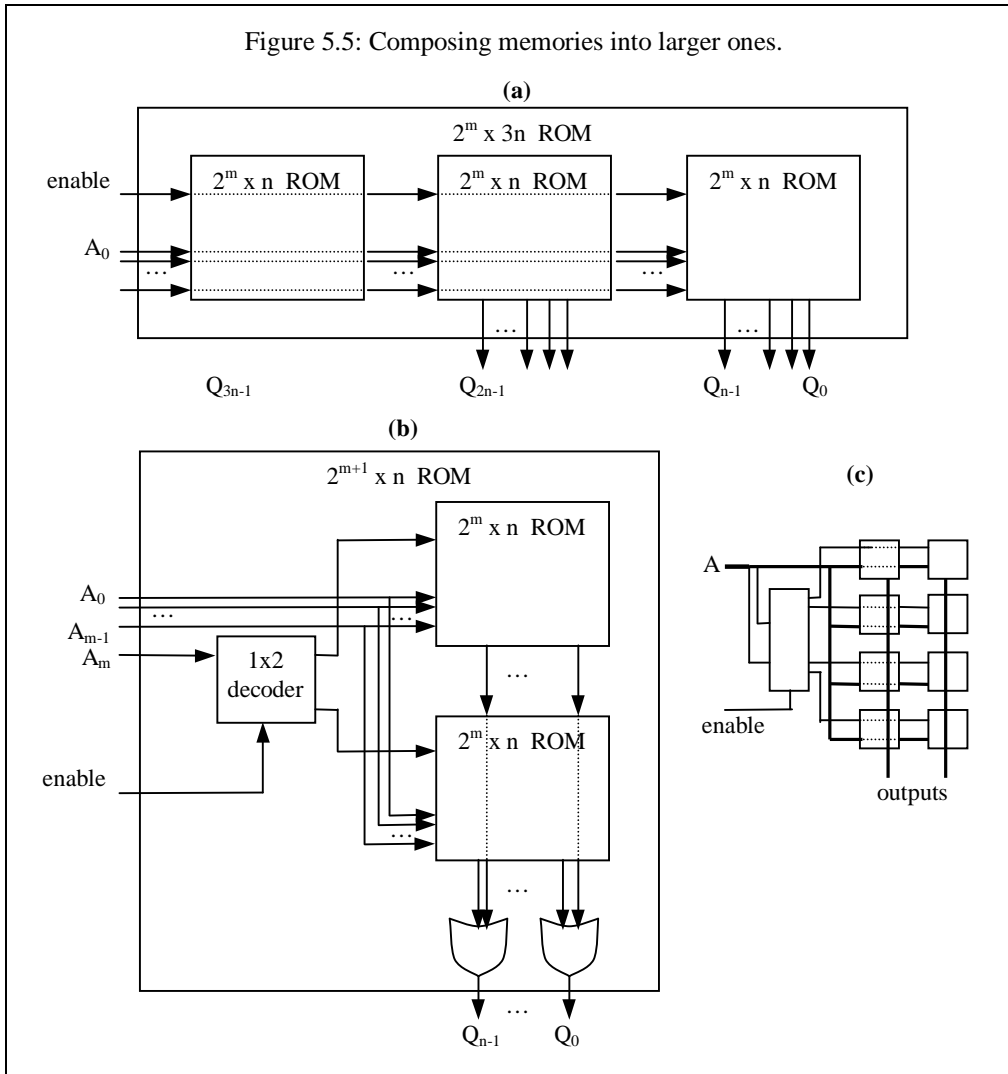
5.4 Composing memories

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 2k x 8 ROM, but may have 4k x 16 ROMs readily available. Alternatively, the designer may need a 4k x 16 ROM, but may have 2k x 8 ROMs available for use.

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(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.

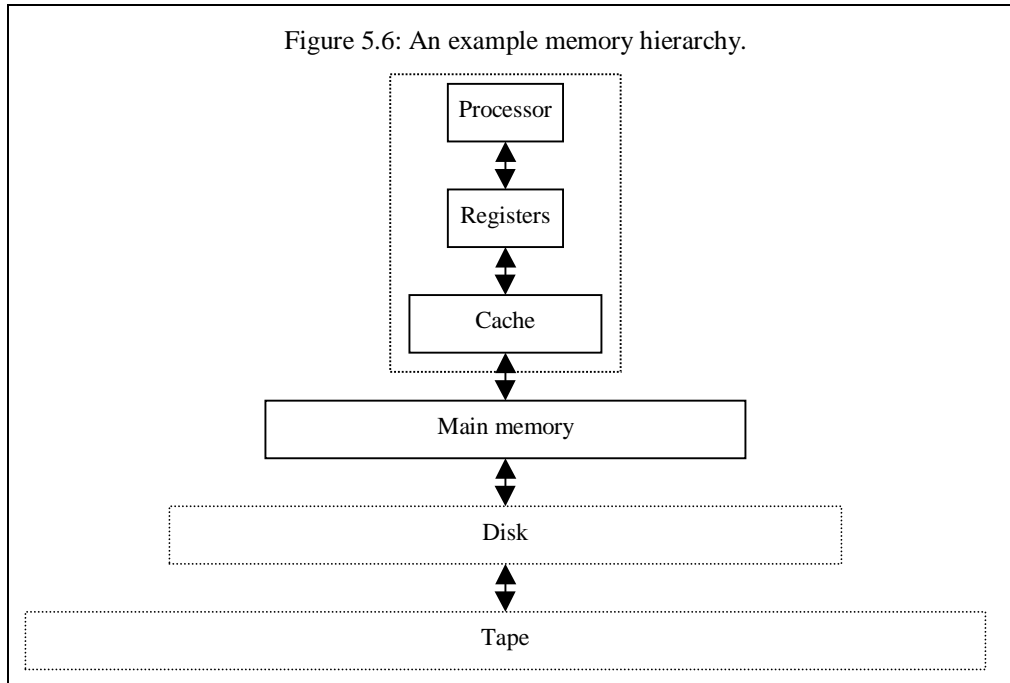
Suppose instead that the available memories have the correct word width, but not enough words. In this case, we can connect the available memories top-to-bottom. For example, Figure 5(b) illustrates the situation of needing a ROM with twice as many words, and hence needing one extra address line, than that available. We connect the ROMs top-to-bottom, OR'ing the corresponding data lines of each. We use the extra high-order address line to select the higher or lower ROM (using a 1x2 decoder), and the



remaining address lines to offset into the selected ROM. Since only one ROM will ever be enabled at a time, the OR'ing of the data lines never actually involves more than one 1.

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 lines as input would select which of the four ROMs to access.

Finally, 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(c).

5.5 Memory hierarchy and cache

When we design a memory to store an embedded system's program and data, we often face the following dilemma: we want an inexpensive and fast memory, but inexpensive memories tend to be slow, whereas fast memories tend to be expensive. The solution to this dilemma is to create a memory hierarchy, as illustrated in Figure 5.6. 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 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.

5.5.1 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' contents are in the cache. Cache mapping can be accomplished using one of three basic techniques:

1. *Direct mapping*: In this technique, 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., $\text{index size} = \log_2(\text{cache size})$. Note that many different main memory addresses will map to the same cache address. When we store a main memory address' content 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 we then compare the tag there with the desired tag.
2. *Fully-associative mapping*: In this technique, each cache address contains not only a main memory address' content, 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. *Set-associative mapping*: This technique is a compromise 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, called a set or a line. 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.

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*, typically consisting of about 8 addresses.

5.5.2 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 block that's 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 replace by popping the queue.

5.5.3 Cache write techniques

When we write to a cache, we must at some point update the 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. While 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.

5.6 Summary

Memories store data for use by processors. 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). 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. Pseudo-static RAM is a dynamic RAM with a built-in refresh controller. Non-volatile 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 memories into larger ones. 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.

5.7 References and further reading

The Free Online Dictionary of Computing (<http://www.instantweb.com/~foldoc/contents.html>) includes definitions of a variety of computer-related terms. These include definitions of various ROM and RAM variations beyond those discussed in the chapter, such as Extended Data Output (EDO) RAM, Video RAM (VRAM), Synchronous DRAM (SDRAM), and Cached DRAM (CDRAM).

5.8 Exercises

1. Briefly define each of the following: mask-programmed ROM, PROM, EPROM, EEPROM, flash EEPROM, RAM, SRAM, DRAM, PSRAM, NVRAM.
2. Sketch the internal design of a 4x3 ROM.
3. Sketch the internal design of a 4x3 RAM.
4. Compose 1kx8 ROM's into a 1kx32 ROM (note: 1k actually means 1028 words).
5. Compose 1kx8 ROM's into an 8kx8 ROM.
6. Compose 1kx8 ROM's into a 2kx16 ROM.
7. Show how to use a 1kx8 ROM to implement a 512x6 ROM.
8. Design your own 8kx32 PSRAM using an 8kx32 DRAM, by designing a refresh controller. The refresh controller should guarantee refresh of each word every 15.625 microseconds. Because the PSRAM may be busy refreshing itself when a read or write access request occurs (i.e., the enable input is set), it should have an output signal *ack* indicating that an access request has been completed.