9 Memory Architectures

Many processor architects argue that memory systems have more impact on overall system performance than data pipelines. This depends, of course, on the application, but for many applications it is true. There are three main sources of complexity in memory. First, it is commonly necessary to mix a variety of memory technologies in the same embedded system. Many memory technologies are **volatile**, meaning that the contents of the

memory is lost if power is lost. Most embedded systems need at least some non-volatile memory and some volatile memory. Moreover, within these categories, there are several choices, and the choices have significant consequences for the system designer. Second, memory hierarchy is often needed because memories with larger capacity and/or lower power consumption are slower. To achieve reasonable performance at reasonable cost, faster memories must be mixed with slower memories. Third, the address space of a processor architecture is divided up to provide access to the various kinds of memory, to provide support for common programming models, and to designate addresses for interaction with devices other than memories, such as I/O devices. In this chapter, we discuss these three issues in order.

9.1 Memory Technologies

In embedded systems, memory issues loom large. The choices of memory technologies have important consequences for the system designer. For example, a programmer may need to worry about whether data will persist when the power is turned off or a power-saving standby mode is entered. A memory whose contents are lost when the power is cut off is called a **volatile memory**. In this section, we discuss some of the available technologies and their tradeoffs.

9.1.1 RAM

In addition to the register file, a microcomputer typically includes some amount of **RAM** (random access memory), which is a memory where individual items (bytes or words) can be written and read one at a time relatively quickly. **SRAM** (static RAM) is faster than **DRAM** (dynamic RAM), but it is also larger (each bit takes up more silicon area). DRAM holds data for only a short time, so each memory location must be periodically refreshed. SRAM holds data for as long as power is maintained. Both types of memories lose their contents if power is lost, so both are volatile memory, although arguably DRAM is more volatile than SRAM because it loses its contents even if power is maintained.

Most embedded computer systems include an SRAM memory. Many also include DRAM because it can be impractical to provide enough memory with SRAM technology alone. A programmer that is concerned about the time it takes a program to execute must be aware of whether memory addresses being accessed are mapped to SRAM or DRAM. More-

over, the refresh cycle of DRAM can introduce variability to the access times because the DRAM may be busy with a refresh at the time that access is requested. In addition, the access history can affect access times. The time it takes to access one memory address may depend on what memory address was last accessed.

A manufacturer of a DRAM memory chip will specify that each memory location must be refreshed, say, every 64 ms, and that a number of locations (a "row") are refreshed together. The mere act of reading the memory will refresh the locations that are read (and locations on the same row), but since applications may not access all rows within the specified time interval, DRAM has to be used with a controller that ensures that all locations are refreshed sufficiently often to retain the data. The memory controller will stall accesses if the memory is busy with a refresh when the access is initiated. This introduces variability in the timing of the program.

9.1.2 Non-Volatile Memory

Embedded systems invariably need to store data even when the power is turned off. There are several options for this. One, of course, is to provide battery backup so that power is never lost. Batteries, however, wear out, and there are better options available, known collectively as **non-volatile memories**. An early form of non-volatile memory was **magnetic core memory** or just **core**, where a ferromagnetic ring was magnetized to store data. The term "core" persists in computing to refer to computer memories, although this may change as multicore machines become ubiquitous.

The most basic non-volatile memory today is **ROM** (read-only memory) or **mask ROM**, the contents of which is fixed at the chip factory. This can be useful for mass produced products that only need to have a program and constant data stored, and these data never change. Such programs are known as **firmware**, suggesting that they are not as "soft" as software. There are several variants of ROM that can be programmed in the field, and the technology has gotten good enough that these are almost always used today over mask ROM. **EEPROM**, electrically-erasable programmable ROM, comes in several forms, but it is possible to write to all of these. The write time is typically much longer than the read time, and the number of writes is limited during the lifetime of the device. A particularly useful form of EEPROM is flash memory. Flash is commonly used to store firmware and user data that needs to persist when the power is turned off.

Flash memory, invented by Dr. Fujio Masuoka at Toshiba around 1980, is a particularly convenient form of non-volatile memory, but it presents some interesting challenges for

embedded systems designers. Typically, flash memories have reasonably fast read times, but not as fast as SRAM and DRAM, so frequently accessed data will typically have to be moved from the flash to RAM before being used by a program. The write times are much longer than the read times, and the total number of writes are limited, so these memories are not a substitute for working memory.

There are two types of flash memories, known as NOR and NAND flash. NOR flash has longer erase and write times, but it can be accessed like a RAM. NAND flash is less expensive and has faster erase and write times, but data must be read a block at a time, where a block is hundreds to thousands of bits. This means that from a system perspective it behaves more like a secondary storage device like a hard disk or optical media like CD or DVD. Both types of flash can only be erased and rewritten a bounded number of times, typically under 1,000,000 for NOR flash and under 10,000,000 for NAND flash, as of this writing.

The longer access times, limited number of writes, and block-wise accesses (for NAND flash), all complicate the problem for embedded system designers. These properties must be taken into account not only while designing hardware, but also software.

Disk memories are also non-volatile. They can store very large amounts of data, but access times can become quite large. In particular, the mechanics of a spinning disk and a read-write head require that the controller wait until the head is positioned over the requested location before the data at that location can be read. The time this takes is highly variable. Disks are also more vulnerable to vibration than the solid-state memories discussed above, and hence are more difficult to use in many embedded applications.

9.2 Memory Hierarchy

Many applications require substantial amounts of memory, more than what is available on-chip in a microcomputer. Many processors use a **memory hierarchy**, which combines different memory technologies to increase the overall memory capacity while optimizing cost, latency, and energy consumption. Typically, a relatively small amount of on-chip SRAM will be used with a larger amount of off-chip DRAM. These can be further combined with a third level, such as disk drives, which have very large capacity, but lack random access and hence can be quite slow to read and write.

The application programmer may not be aware that memory is fragmented across these technologies. A commonly used scheme called **virtual memory** makes the diverse tech-

nologies look to the programmer like a contiguous **address space**. The operating system and/or the hardware provides **address translation**, which converts logical addresses in the address space to physical locations in one of the available memory technologies. This translation is often assisted by a specialized piece of hardware called a **translation lookaside buffer** (**TLB**), which can speed up some address translations. For an embedded system designer, these techniques can create serious problems because they make it very difficult to predict or understand how long memory accesses will take. Thus, embedded system designers typically need to understand the memory system more deeply than general-purpose programmers.

9.2.1 Memory Maps

A **memory map** for a processor defines how addresses get mapped to hardware. The total size of the address space is constrained by the address width of the processor. A 32-bit processor, for example, can address 2^{32} locations, or 4 gigabytes (GB), assuming each address refers to one byte. The address width typically matches the word width, except for 8-bit processors, where the address width is typically higher (often 16 bits). An ARM Cortex™ - M3 architecture, for example, has the memory map shown in Figure 9.1. Other architectures will have other layouts, but the pattern is similar.

Notice that this architecture separates addresses used for program memory (labeled A in the figure) from those used for data memory (B and D). This (typical) pattern allows these memories to be accessed via separate buses, permitting instructions and data to be fetched simultaneously. This effectively doubles the memory bandwidth. Such a separation of program memory from data memory is known as a **Harvard architecture**. It contrasts with the classical **von Neumann architecture**, which stores program and data in the same memory.

Any particular realization in silicon of this architecture is constrained by this memory map. For example, the Luminary Micro[1] LM3S8962 controller, which includes an ARM Cortex™ - M3 core, has 256 KB of on-chip flash memory, nowhere near the total of 0.5 GB that the architecture allows. This memory is mapped to addresses `0x00000000` through `0x0003FFFF`. The remaining addresses that the architecture allows for program memory, which are `0x00040000` through `0x1FFFFFFF`, are "reserved addresses," meaning that they should not be used by a compiler targeting this particular device.

[1] Luminary Micro was acquired by Texas Instruments in 2009.

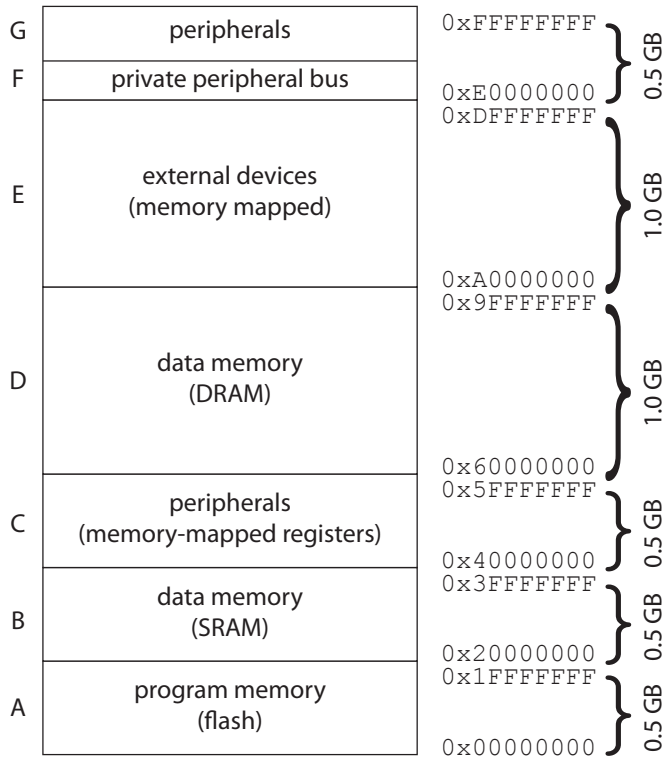


Figure 9.1: Memory map of an ARM Cortex™ - M3 architecture.

The LM3S8962 has 64 KB of SRAM, mapped to addresses `0x20000000` through `0x2000FFFF`, a small portion of area B in the figure. It also includes a number of on-chip **peripherals**, which are devices that are accessed by the processor using some of the memory addresses in the range from `0x40000000` to `0x5FFFFFFF` (area C in the figure). These include timers, ADCs, GPIO, UARTs, and other I/O devices. Each of these devices occupies a few of the memory addresses by providing **memory-mapped registers**. The processor may write to some of these registers to configure and/or control the peripheral, or to provide data to be produced on an output. Some of the registers may be read to retrieve input data obtained by the peripheral. A few of the addresses in the private peripheral bus region are used to access the interrupt controller.

The LM3S8962 is mounted on a printed circuit board that will provide additional devices such as DRAM data memory and additional external devices. As shown in Figure 9.1, these will be mapped to memory addresses in the range from `0xA0000000` to `0xDFFFFFFF` (area E). For example, the Stellaris® LM3S8962 evaluation board from Luminary Micro includes no additional external memory, but does add a few external devices such as an LCD display, a MicroSD slot for additional flash memory, and a USB interface.

This leaves many memory addresses unused. ARM has introduced a clever way to take advantage of these unused addresses called **bit banding**, where some of the unused addresses can be used to access individual bits rather than entire bytes or words in the memory and peripherals. This makes certain operations more efficient, since extra instructions to mask the desired bits become unnecessary.

Harvard Architecture

The term "Harvard architecture" comes from the Mark I computer, which used distinct memories for program and data. The Mark I was made with electro-mechanical relays by IBM and shipped to Harvard in 1944. The machine stored instructions on punched tape and data in electro-mechanical counters. It was called the Automatic Sequence Controlled Calculator (ASCC) by IBM, and was devised by Howard H. Aiken to numerically solve differential equations. Rear Admiral Grace Murray Hopper of the United States Navy and funding from IBM were instrumental in making the machine a reality.

9.2.2 Register Files

The most tightly integrated memory in a processor is the **register file**. Each register in the file stores a **word**. The size of a word is a key property of a processor architecture. It is one byte on an 8-bit architecture, four bytes on a 32-bit architecture, and eight bytes on a 64-bit architecture. The register file may be implemented directly using flip flops in the processor circuitry, or the registers may be collected into a single memory bank, typically using the same SRAM technology discussed above.

The number of registers in a processor is usually small. The reason for this is not so much the cost of the register file hardware, but rather the cost of bits in an instruction word. An instruction set architecture (ISA) typically provides instructions that can access one, two, or three registers. To efficiently store programs in memory, these instructions cannot require too many bits to encode them, and hence they cannot devote too many bits to identifying the registers. If the register file has 16 registers, then each reference to a register requires 4 bits. If an instruction can refer to 3 registers, that requires a total of 12 bits. If an instruction word is 16 bits, say, then this leaves only 4 bits for other information in the instruction, such as the identity of the instruction itself, which also must be encoded in the instruction. This identifies, for example, whether the instruction specifies that two registers should be added or subtracted, with the result stored in the third register.

9.2.3 Scratchpads and Caches

Many embedded applications mix memory technologies. Some memories are accessed before others; we say that the former are "closer" to the processor than the latter. For example, a close memory (SRAM) is typically used to store working data temporarily while the program operates on it. If the close memory has a distinct set of addresses and the program is responsible for moving data into it or out of it to the distant memory, then it is called a **scratchpad**. If the close memory duplicates data in the distant memory with the hardware automatically handling the copying to and from, then it is called a **cache**. For embedded applications with tight real-time constraints, cache memories present some formidable obstacles because their timing behavior can vary substantially in ways that are difficult to predict. On the other hand, manually managing the data in a scratchpad memory can be quite tedious for a programmer, and automatic compiler-driven methods for doing so are in their infancy.

As explained in Section 9.2.1, an architecture will typically support a much larger address space than what can actually be stored in the physical memory of the processor, with a virtual memory system used to present the programmer with the view of a contiguous address space. If the processor is equipped with a **memory management unit** (**MMU**), then programs reference **logical addresses** and the MMU translates these to **physical addresses**. For example, using the memory map in Figure 9.1, a process might be allowed to use logical addresses `0x60000000` to `0x9FFFFFFF` (area D in the figure), for a total of 1 GB of addressable data memory. The MMU may implement a cache that uses however much physical memory is present in area B. When the program provides a memory address, the MMU determines whether that location is cached in area B, and if it is, translates the address and completes the fetch. If it is not, then we have a **cache miss**, and the MMU handles fetching data from the secondary memory (in area D) into the cache (area B). If the location is also not present in area D, then the MMU triggers a **page fault**, which can result in software handling movement of data from disk into the memory. Thus, the program is given the illusion of a vast amount of memory, with the cost that memory access times become quite difficult to predict. It is not uncommon for memory access times to vary by a factor of 1000 or more, depending on how the logical addresses happen to be disbursed across the physical memories.

Given this sensitivity of execution time to the memory architecture, it is important to understand the organization and operation of caches. That is the focus of this section.

Basic Cache Organization

Suppose that each address in a memory system comprises m bits, for a maximum of $M = 2^m$ unique addresses. A cache memory is organized as an array of $S = 2^s$ **cache set**s. Each cache set in turn comprises E **cache line**s. A cache line stores a single **block** of $B = 2^b$ bytes of data, along with **valid** and **tag** bits. The valid bit indicates whether the cache line stores meaningful information, while the tag (comprising $t = m - s - b$ bits) uniquely identifies the block that is stored in the cache line. Figure 9.2 depicts the basic cache organization and address format.

Thus, a cache can be characterized by the tuple (m, S, E, B). These parameters are summarized in Table 9.1. The overall cache size C is given as $C = S \times E \times B$ bytes.

Suppose a program reads the value stored at address a. Let us assume for the rest of this section that this value is a single data word w. The CPU first sends address a to the cache to determine if it is present there. The address a can be viewed as divided into three

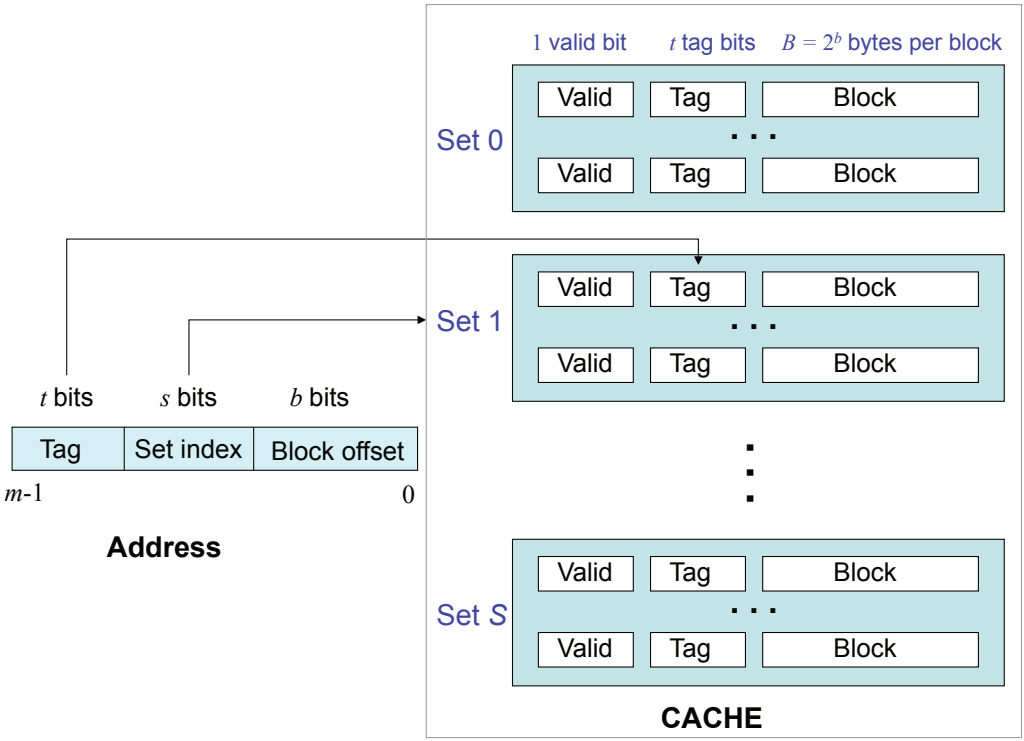


Figure 9.2: Cache Organization and Address Format. A cache can be viewed as an array of sets, where each set comprises of one or more cache lines. Each cache line includes a valid bit, tag bits, and a cache block.

Parameter	Description
m	Number of physical address bits
$S = 2^s$	Number of (cache) sets
E	Number of lines per set
$B = 2^b$	Block size in bytes
$t = m - s - b$	Number of tag bits
C	Overall cache size in bytes

Table 9.1: Summary of cache parameters.

segments of bits: the top t bits encode the tag, the next s bits encode the set index, and the last b bits encode the position of the word within a block. If w is present in the cache, the memory access is a **cache hit**; otherwise, it is a **cache miss**.

Caches are categorized into classes based on the value of E. We next review these categories of cache memories, and describe briefly how they operate.

Direct-Mapped Caches

A cache with exactly one line per set ($E = 1$) is called a **direct-mapped cache**. For such a cache, given a word w requested from memory, where w is stored at address a, there are three steps in determining whether w is a cache hit or a miss:

1. *Set Selection:* The s bits encoding the set are extracted from address a and used as an index to select the corresponding cache set.

2. *Line Matching:* The next step is to check whether a copy of w is present in the unique cache line for this set. This is done by checking the valid and tag bits for that cache line. If the valid bit is set and the tag bits of the line match those of the address a, then the word is present in the line and we have a cache hit. If not, we have a cache miss.

3. *Word Selection:* Once the word is known to be present in the cache block, we use the b bits of the address a encoding the word's position within the block to read that data word.

On a cache miss, the word w must be requested from the next level in the memory hierarchy. Once this block has been fetched, it will replace the block that currently occupies the cache line for w.

While a direct-mapped cache is simple to understand and to implement, it can suffer from **conflict misses**. A conflict miss occurs when words in two or more blocks that map to the same cache line are repeatedly accessed so that accesses to one block evict the other, resulting in a string of cache misses. Set-associative caches can help to resolve this problem.

Set-Associative Caches

A **set-associative cache** can store more than one cache line per set. If each set in a cache can store E lines, where $1 < E < C/B$, then the cache is called an *E-way* set-associative cache. The word "associative" comes from **associative memory**, which is a memory that is addressed by its contents. That is, each word in the memory is stored along with a unique key and is retrieved using the key rather than the physical address indicating where it is stored. An associative memory is also called a **content-addressable memory**.

For a set-associative cache, accessing a word w at address a consists of the following steps:

1. *Set Selection:* This step is identical to a direct-mapped cache.
2. *Line Matching:* This step is more complicated than for a direct-mapped cache because there could be multiple lines that w might lie in; i.e., the tag bits of a could match the tag bits of any of the lines in its cache set. Operationally, each set in a set-associative cache can be viewed as an associative memory, where the keys are the concatenation of the tag and valid bits, and the data values are the contents of the corresponding block.
3. *Word Selection:* Once the cache line is matched, the word selection is performed just as for a direct-mapped cache.

In the case of a miss, cache line replacement can be more involved than it is for a direct-mapped cache. For the latter, there is no choice in replacement since the new block will displace the block currently present in the cache line. However, in the case of a set-associative cache, we have an option to select the cache line from which to evict a block.

A common policy is **least-recently used** (**LRU**), in which the cache line whose most recent access occurred the furthest in the past is evicted. Another common policy is **first-in, first-out** (**FIFO**), where the cache line that is evicted is the one that has been in the cache for the longest, regardless of when it was last accessed. Good cache replacement policies are essential for good cache performance. Note also that implementing these cache replacement policies requires additional memory to remember the access order, with the amount of additional memory differing from policy to policy and implementation to implementation.

A **fully-associative cache** is one where $E = C/B$, i.e., there is only one set. For such a cache, line matching can be quite expensive for a large cache size because an associative memory is expensive. Hence, fully-associative caches are typically only used for small caches, such as the translation lookaside buffers (TLBs) mentioned earlier.

9.3 Memory Models

A **memory model** defines how memory is used by programs. The hardware, the operating system (if any), and the programming language and its compiler all contribute to the memory model. This section discusses a few of the common issues that arise with memory models.

9.3.1 Memory Addresses

At a minimum, a memory model defines a range of **memory addresses** accessible to the program. In C, these addresses are stored in **pointer**s. In a **32-bit architecture**, memory addresses are 32-bit unsigned integers, capable of representing addresses 0 to $2^{32} - 1$, which is about four billion addresses. Each address refers to a byte (eight bits) in memory. The C `char` data type references a byte. The C `int` data type references a sequence of at least two bytes. In a 32-bit architecture, it will typically reference four bytes, able to represent integers from -2^{31} to $2^{31} - 1$. The `double` data type in C refers to a sequence of eight bytes encoded according to the IEEE floating point standard (IEEE 754).

Since a memory address refers to a byte, when writing a program that directly manipulates memory addresses, there are two critical compatibility concerns. The first is the **alignment** of the data. An `int` will typically occupy four consecutive bytes starting at an

address that is a multiple of four. In hexadecimal notation these addresses always end in 0, 4, 8, or c.

The second concern is the byte order. The first byte (at an address ending in 0, 4, 8, or c), may represent the eight low order bits of the int (a representation called **little endian**), or it may represent the eight high order bits of the int (a representation called **big endian**). Unfortunately, although many data representation questions have become universal standards (such as the bit order in a byte), the byte order is not one those questions. Intel's x86 architectures and ARM processors, by default, use a little-endian representation, whereas IBM's PowerPC uses big endian. Some processors support both. Byte order also matters in network protocols, which generally use big endian.

The terminology comes from Gulliver's Travels, by Jonathan Swift, where a royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, while in the rival kingdom of Blefuscu, inhabitants crack theirs at the big end.

9.3.2 Stacks

A **stack** is a region of memory that is dynamically allocated to the program in a last-in, first-out (**LIFO**) pattern. A **stack pointer** (typically a register) contains the memory address of the top of the stack. When an item is pushed onto the stack, the stack pointer is incremented and the item is stored at the new location referenced by the stack pointer. When an item is popped off the stack, the memory location referenced by the stack pointer is (typically) copied somewhere else (e.g., into a register) and the stack pointer is decremented.

Stacks are typically used to implement procedure calls. Given a procedure call in C, for example, the compiler produces code that pushes onto the stack the location of the instruction to execute upon returning from the procedure, the current value of some or all of the machine registers, and the arguments to the procedure, and then sets the program counter equal to the location of the procedure code. The data for a procedure that is pushed onto the stack is known as the **stack frame** of that procedure. When a procedure returns, the compiler pops its stack frame, retrieving finally the program location at which to resume execution.

For embedded software, it can be disastrous if the stack pointer is incremented beyond the memory allocated for the stack. Such a **stack overflow** can result in overwriting memory that is being used for other purposes, leading to unpredictable results. Bounding the stack

usage, therefore, is an important goal. This becomes particularly difficult with **recursive programs**, where a procedure calls itself. Embedded software designers often avoid using recursion to circumvent this difficulty.

More subtle errors can arise as a result of misuse or misunderstanding of the stack. Consider the following C program:

```
1   int* foo(int a) {
2     int b;
3     b = a * 10;
4     return &b;
5   }
6   int main(void) {
7     int* c;
8     c = foo(10);
9     ...
10  }
```

The variable `b` is a local variable, with its memory on the stack. When the procedure returns, the variable `c` will contain a pointer to a memory location *above the stack pointer*. The contents of that memory location will be overwritten when items are next pushed onto the stack. It is therefore incorrect for the procedure `foo` to return a pointer to `b`. By the time that pointer is de-referenced (i.e., if a line in `main` refers to `*c` after line 8), the memory location may contain something entirely different from what was assigned in `foo`. Unfortunately, C provides no protection against such errors.

9.3.3 Memory Protection Units

A key issue in systems that support multiple simultaneous tasks is preventing one task from disrupting the execution of another. This is particularly important in embedded applications that permit downloads of third party software, but it can also provide an important defense against software bugs in safety-critical applications.

Many processors provide **memory protection** in hardware. Tasks are assigned their own address space, and if a task attempts to access memory outside its own address space, a **segmentation fault** or other exception results. This will typically result in termination of the offending application.

9.3.4 Dynamic Memory Allocation

General-purpose software applications often have indeterminate requirements for memory, depending on parameters and/or user input. To support such applications, computer scientists have developed dynamic memory allocation schemes, where a program can at any time request that the operating system allocate additional memory. The memory is allocated from a data structure known as a **heap**, which facilitates keeping track of which portions of memory are in use by which application. Memory allocation occurs via an operating system call (such as `malloc` in C). When the program no longer needs access to memory that has been so allocated, it deallocates the memory (by calling `free` in C).

Support for memory allocation often (but not always) includes garbage collection. For example, garbage collection is intrinsic in the Java programming language. A **garbage collector** is a task that runs either periodically or when memory gets tight that analyzes the data structures that a program has allocated and automatically frees any portions of memory that are no longer referenced within the program. When using a garbage collector, in principle, a programmer does not need to worry about explicitly freeing memory.

With or without garbage collection, it is possible for a program to inadvertently accumulate memory that is never freed. This is known as a memory leak, and for embedded applications, which typically must continue to execute for a long time, it can be disastrous. The program will eventually fail when physical memory is exhausted.

Another problem that arises with memory allocation schemes is memory fragmentation. This occurs when a program chaotically allocates and deallocates memory in varying sizes. A fragmented memory has allocated and free memory chunks interspersed, and often the free memory chunks become too small to use. In this case, defragmentation is required.

Defragmentation and garbage collection are both very problematic for real-time systems. Straightforward implementations of these tasks require all other executing tasks to be stopped while the defragmentation or garbage collection is performed. Implementations using such "stop the world" techniques can have substantial pause times, running sometimes for many milliseconds. Other tasks cannot execute during this time because references to data within data structures (pointers) are inconsistent during the task. A technique that can reduce pause times is incremental garbage collection, which isolates sections of memory and garbage collects them separately. As of this writing, such techniques are experimental and not widely deployed.

9.3.5 Memory Model of C

C programs store data on the stack, on the heap, and in memory locations fixed by by the compiler. Consider the following C program:

```
1  int a = 2;
2  void foo(int b, int* c) {
3    ...
4  }
5  int main(void) {
6    int d;
7    int* e;
8    d = ...;                    // Assign some value to d.
9    e = malloc(sizeInBytes);    // Allocate memory for e.
10   *e = ...;                   // Assign some value to e.
11   foo(d, e);
12   ...
13 }
```

In this program, the variable `a` is a **global variable** because it is declared outside any procedure definition. The compiler will assign it a fixed memory location. The variables `b` and `c` are **parameter**s, which are allocated locations on the stack when the procedure `foo` is called (a compiler could also put them in registers rather than on the stack). The variables `d` and `e` are **automatic variable**s or **local variable**s. They are declared within the body of a procedure (in this case, `main`). The compiler will allocate space on the stack for them.

When the procedure `foo` is called on line 11, the stack location for `b` will acquire a *copy* of the value of variable `d` assigned on line 8. This is an example of **pass by value**, where a parameter's value is copied onto the stack for use by the called procedure. The data referred to by the pointer `e`, on the other hand, is stored in memory allocated on the heap, and then it is **passed by reference** (the pointer to it, `e`, is passed by value). The *address* is stored in the stack location for `c`. If `foo` includes an assignment to `*c`, then then after `foo` returns, that value can be read by dereferencing `e`.

The global variable `a` is assigned an initial value on line 1. There is a subtle pitfall here, however. The memory location storing `a` will be initialized with value 2 *when the program is loaded*. This means that if the program is run a second time without reloading, then the initial value of `a` will not necessarily be 2! Its value will be whatever it was when the first invocation of the program ended. In most desktop operating systems, the program is reloaded on each run, so this problem does not show up. But in many embedded systems,

the program is not necessarily reloaded for each run. The program may be run from the beginning, for example, each time the system is reset. To guard against this problem, it is safer to initialize global variables in the body of `main`, rather than on the declaration line, as done above.

9.4 Summary

An embedded system designer needs to understand the memory architecture of the target computer and the memory model of the programming language. Incorrect uses of memory can lead to extremely subtle errors, some of which will not show up in testing. Errors that only show up in a fielded product can be disastrous, for both the user of the system and the technology provider.

Specifically, a designer needs to understand which portions of the address space refer to volatile and non-volatile memory. For time-sensitive applications (which is most embedded systems), the designer also needs to be aware of the memory technology and cache architecture (if any) in order to understand execution times of the program. In addition, the programmer needs to understand the memory model of the programming language in order to avoid reading data that may be invalid. In addition, the programmer needs to be very careful with dynamic memory allocation, particularly for embedded systems that are expected to run for a very long time. Exhausting the available memory can cause system crashes or other undesired behavior.

Exercises

1. Consider the function `compute_variance` listed below, which computes the variance of integer numbers stored in the array `data`.

```
1  int data[N];
2
3  int compute_variance() {
4    int sum1 = 0, sum2 = 0, result;
5    int i;
6
7    for(i=0; i < N; i++) {
8      sum1 += data[i];
9    }
10   sum1 /= N;
11
12   for(i=0; i < N; i++) {
13     sum2 += data[i] * data[i];
14   }
15   sum2 /= N;
16
17   result = (sum2 - sum1*sum1);
18
19   return result;
20 }
```

   Suppose this program is executing on a 32-bit processor with a direct-mapped cache with parameters $(m, S, E, B) = (32, 8, 1, 8)$. We make the following additional assumptions:

   - An `int` is 4 bytes wide.
   - `sum1`, `sum2`, `result`, and `i` are all stored in registers.
   - `data` is stored in memory starting at address `0x0`.

   Answer the following questions:

   (a) Consider the case where `N` is 16. How many cache misses will there be?

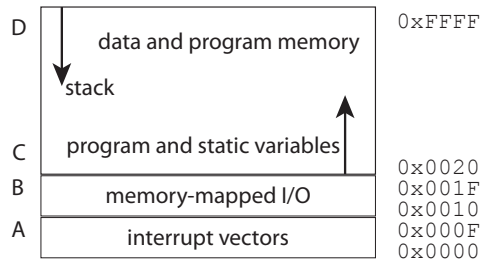
   (b) Now suppose that `N` is 32. Recompute the number of cache misses.

   (c) Now consider executing for `N` = 16 on a 2-way set-associative cache with parameters $(m, S, E, B) = (32, 8, 2, 4)$. In other words, the block size is halved, while there are two cache lines per set. How many cache misses would the code suffer?

2. Recall from Section 9.2.3 that caches use the middle range of address bits as the set index and the high order bits as the tag. Why is this done? How might cache performance be affected if the middle bits were used as the tag and the high order bits were used as the set index?

3. Consider the C program and simplified memory map for a 16-bit microcontroller shown below. Assume that the stack grows from the top (area D) and that the program and static variables are stored in the bottom (area C) of the data and program memory region. Also, assume that the entire address space has physical memory associated with it.

```
1  #include <stdio.h>
2  #define FOO 0x0010
3  int n;
4  int* m;
5  void foo(int a) {
6     if (a > 0) {
7        n = n + 1;
8        foo(n);
9     }
10 }
11 int main() {
12    n = 0;
13    m = (int*)FOO;
14    foo(*m);
15    printf("n = %d\n", n);
16 }
```



You may assume that in this system, an `int` is a 16-bit number, that there is no operating system and no memory protection, and that the program has been compiled and loaded into area C of the memory.

   (a) For each of the variables `n`, `m`, and `a`, indicate where in memory (region A, B, C, or D) the variable will be stored.

   (b) Determine what the program will do if the contents at address 0x0010 is 0 upon entry.

   (c) Determine what the program will do if the contents of memory location 0x0010 is 1 upon entry.

4. Consider the following program:

```
1   int a = 2;
2   void foo(int b) {
```

```
3       printf("%d", b);
4     }
5     int main(void) {
6       foo(a);
7       a = 1;
8     }
```

Is it true or false that the value of `a` passed to `foo` will always be 2? Explain. Assume that this is the entire program, that this program is stored in persistent memory, and that the program is executed on a bare-iron microcontroller each time a reset button is pushed.