CHAPTER

CPUs

3

CHAPTER POINTS

- Input and output mechanisms.
- Supervisor mode, exceptions, and traps.
- Memory management and address translation.
- Interrupts, supervisor mode, exceptions, and traps.
- Caches.
- Performance and power consumption of CPUs.
- Design example: Data compressor.

3.1 Introduction

This chapter describes aspects of CPUs that do not directly relate to their instruction sets. We consider mechanisms that are important to interfacing with other system elements, such as interrupts and memory management. We also take a first look at aspects of the CPU other than functionality; performance and power consumption are both vital attributes of programs that are only indirectly related to the instructions they use.

In Section 3.2, we study input and output mechanisms, including both busy/wait and interrupts. Section 3.3 introduces several specialized mechanisms for operations, such as detecting internal errors and protecting CPU resources. Section 3.4 introduces coprocessors that provide optional support for parts of the instruction set. Section 3.5 describes the CPU's view of memory, both memory management and caches. The next sections look at the nonfunctional attributes of execution: Section 3.6 looks at performance, and Section 3.7 considers power consumption. Section 3.8 looks at several issues related to safety and security. Finally, in Section 3.9, we use a data compressor as an example of a simple, yet interesting program.

3.2 Programming input and output

The basic techniques for I/O programming can be understood as relatively independent of the instruction set. In this section, we cover the basics of I/O programming and place them in the contexts of the Arm, C55x, and PIC16F. We begin by discussing the basic characteristics of I/O devices so that we can understand the requirements they place on programs that communicate with them.

3.2.1 I/O devices

Input and output devices usually have some analog or nonelectronic components. However, the digital logic in the device that is most closely connected to the CPU very strongly resembles the logic you would expect in any computer system.

Fig. 3.1 shows the structure of a typical I/O device and its relationship to the CPU. The interface between the CPU and the device is a set of registers. The CPU talks to the device by reading and writing to/from the registers. A device typically has several registers:

- **Data registers** hold values that are treated as data by the device, such as data that are read or written by a disk.
- **Status registers** provide information about the device's operation, such as whether the current transaction has been completed.

Some registers may be read-only, such as a status register that indicates when the device is done, whereas others may be readable or writable.

Application Example 3.1 describes a classic I/O device.

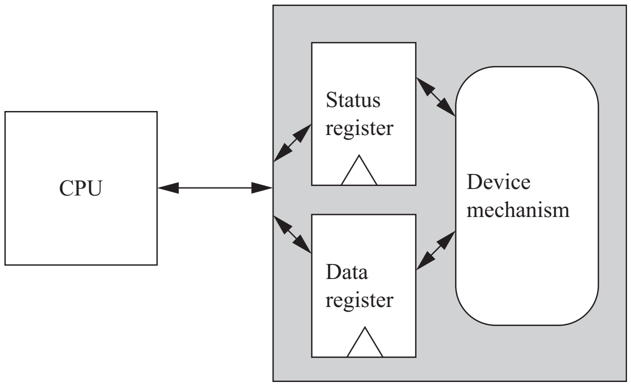


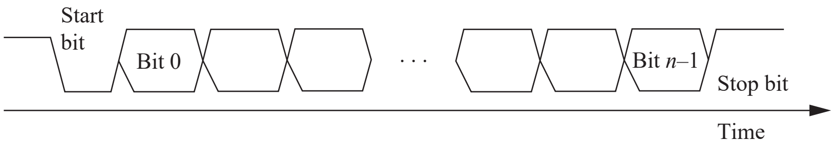
FIGURE 3.1

Structure of a typical I/O device.

Application Example 3.1 The 8251 Universal Asynchronous Receiver/Transmitter

The 8251 Universal Asynchronous Receiver/Transmitter (**UART**) [Int82] is the original device used for serial communications, such as serial port connections on PCs. The 8251 was introduced as a stand-alone integrated circuit for early microprocessors. Today, its functions are typically subsumed by a larger chip, but these more advanced devices still use the basic programming interface defined by the 8251.

The UART is programmable for a variety of transmission and reception parameters. However, the basic format of transmission is simple. Data are transmitted as streams of characters in this form:



Every character starts with a start bit (a 0 value) and a stop bit (a 1 value). The start bit allows the receiver to recognize the start of a new character, and the stop bit ensures that there will be a transition at the start of the stop bit. The data bits are sent as high and low voltages at a uniform rate. That rate is known as the **baud rate**; the period of one bit is the inverse of the baud rate.

Before transmitting or receiving data, the CPU must set the UART's mode register to correspond to the data line's characteristics. The parameters for the serial port are familiar from the parameters for a serial communications program:

- mode[1:0]: mode and baud rate
  - 00: synchronous mode
  - 01: asynchronous mode, no clock prescaler
  - 10: asynchronous mode, 16× prescaler
  - 11: asynchronous mode, 64× prescaler
- mode[3:2]: number of bits per character
  - 00: 5 bits
  - 01: 6 bits
  - 10: 7 bits
  - 11: 8 bits
- mode[5:4]: parity
  - 00, 10: no parity
  - 01: odd parity
  - 11: even parity
- mode[7:6]: stop bit length
  - 00: invalid
  - 01: 1 stop bit
  - 10: 1.5 stop bits
  - 11: 2 stop bits

Setting bits in the command register tells the UART what to do:

- mode[0]: transmit enable
- mode[1]: set nDTR output
- mode[2]: enable receiver
- mode[3]: send break character
- mode[4]: reset error flags

- mode[5]: set nRTS output
- mode[6]: internal reset
- mode[7]: hunt mode

The status register shows the state of the UART and transmission:

- status [0]: transmitter ready
- status [1]: receive ready
- status [2]: transmission complete
- status [3]: parity
- status [4]: overrun
- status [5]: frame error
- status [6]: sync char detected
- status [7]: nDSR value

The UART includes transmit and receive buffer registers. It also includes registers for synchronous mode characters.

The *Transmitter Ready* output indicates that the transmitter is ready to accept a data character; the *Transmitter Empty* signal goes high when the UART has no characters to send. On the receiver side, the *Receiver Ready* pin goes high when the UART has a character ready to be read by the CPU.

3.2.2 Input and output primitives

Microprocessors can provide programming support for input and output in two ways: **I/O instructions** and **memory-mapped I/O.** Some architectures, such as the Intel x86, provide special instructions (`in` and `out` in the case of the Intel x86) for the input and output. These instructions provide a separate address space for I/O devices.

However, the most common way to implement I/O is through memory mapping—even CPUs that provide I/O instructions can also implement memory-mapped I/O. As the name implies, the memory-mapped I/O provides addresses for the registers in each I/O device. Programs use the CPU's normal read and write instructions to communicate with devices.

Example 3.1 illustrates the memory-mapped I/O on Arm.

Example 3.1 Memory-mapped I/O on Arm

We can use the `EQU` pseudo-op to define a symbolic name for the memory location of our I/O device:

```
DEV1    EQU 0x1000
```

Given that name, we can use the following standard code to read and write the device register:

```
LDR r1,#DEV1      ; set up device address
LDR r0,[r1]       ; read DEV1
LDR r0,#8         ; set up value to write
STR r0,[r1]       ; write 8 to device
```

How can we directly write I/O devices in a high-level language like C? When we define and use a variable in C, the compiler hides the variable's address from us. However, we can use pointers to manipulate the addresses of I/O devices. The traditional names for functions that read and write arbitrary memory locations are **peek** and **poke.** The peek function can be written in C as follows:

```
int peek(char *location) {
     return *location; /* de-reference location pointer */
}
```

The argument to `peek` is a pointer that is dereferenced by the C language * operator to read the location. Thus, to read a device register, we can write

```
#define DEV1 0x1000
...
dev_status = peek(DEV1); /* read device register */
```

The `poke` function can be implemented as

```
void poke(char *location, char newval) {
    (* location) = newval; /* write to location */
}
```

To write to the device's register, we can use the following code:

```
poke(DEV1,8); /* write 8 to device register */
```

These functions can, of course, be used to read and write arbitrary memory locations, not just devices.

3.2.3 Busy-wait I/O

The simplest way to communicate with devices in a program is **busy-wait I/O**. Devices are typically slower than the CPU and may require many cycles to complete an operation. If the CPU performs multiple operations on a single device, such as writing several characters to an output device, then it must wait for one operation to complete before starting the next one. If we try to start writing the second character before the device has finished with the first one, for example, the device will probably never print the first character. Asking an I/O device whether it is finished by reading its status register is often called **polling**.

Example 3.2 illustrates busy-wait I/O.

Example 3.2 Busy-wait I/O Programming

In this example, we want to write a sequence of characters to an output device. The device has two registers: one for the character to be written, and one for a status register. The status register's value is 1 when the device is busy writing and 0 when the write transaction has been completed.

We use the peek and poke functions to write the busy-wait routine in C. First, we define symbolic names for the register addresses:

```
#define OUT_CHAR 0x1000 /* output device character register */
#define OUT_STATUS 0x1001 /* output device status register */
```

The sequence of characters is stored in a standard C string, which is terminated by a null (0) character. We use peek and poke to send the characters and wait for each transaction to complete:

```
char *mystring = "Hello, world." /* string to write */
char * current_char; /* pointer to current position in string */
current_char = mystring; /* point to head of string */
while ( *current_char != ' \0') { /* until null character */
     poke(OUT_CHAR, *current_char); /* send character to device */
     while (peek(OUT_STATUS) != 0); /* keep checking status */
     current_char++; /* update character pointer */
}
```

The outer while loop sends the characters one at a time. The inner while loop checks the device status, which implements the busy-wait function by repeatedly checking the device status until the status changes to 0.

Example 3.3 illustrates a combination of input and output.

Example 3.3 Copying Characters from Input to Output Using Busy-wait I/O

We want to read a character repeatedly from the input device and write it to the output device. First, we need to define the addresses for the device registers:

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
```

The input device sets its status register to 1 when a new character has been read; we must set the status register back to 0 after the character has been read so that the device is ready to read another character. When writing, we must set the output status register to 1 to start writing and wait for it to return to 0. We can use peek and poke to repeatedly perform the read/write operation:

```
while (TRUE) { /* perform operation forever */
     /*read a character into achar */
     while (peek(IN_STATUS) == 0); /* wait until ready */
     achar = (char)peek(IN_DATA); /* read the character */
     /* write achar */
     poke(OUT_DATA,achar);
     poke(OUT_STATUS,1); /* turn on device */
     while (peek(OUT_STATUS) != 0); /* wait until done */
}
```

3.2.4 Interrupts

Basics

Busy-wait I/O is extremely inefficient—the CPU does nothing but test the device status while the I/O transaction is in progress. In many cases, the CPU could do useful work in parallel with the I/O transaction:

- computation, as in determining the next output to send to the device or processing the last input received, and
- Control of other I/O devices.

To allow parallelism, we must introduce new mechanisms into the CPU.

The **interrupt** mechanism allows devices to signal the CPU and to force the execution of a particular piece of code. When an interrupt occurs, the program counter's value is changed to point to an **interrupt handler** routine, also commonly known as a **device driver**, which takes care of the device: writing the next data, reading data that have just become ready, and so on. The interrupt mechanism, of course, saves the value of the PC at the interruption, so that the CPU can return to the program that was interrupted. Interrupts therefore allow the flow of control in the CPU to change easily between different **contexts**, such as a foreground computation and multiple I/O devices.

As shown in Fig. 3.2, the interface between the CPU and I/O device includes several signals that control the interrupt process:

- The I/O device asserts the **interrupt request** signal when it wants service from the CPU.
- The CPU asserts the **interrupt acknowledge** signal when it is ready to handle the I/O device's request.

The I/O device's logic decides when to interrupt. For example, it may generate an interrupt when its status register goes into the ready state. The CPU may not be able to

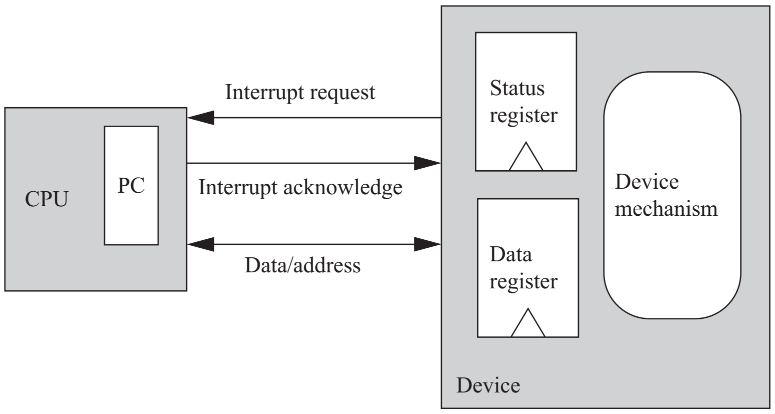


FIGURE 3.2

The interrupt mechanism.

immediately service an interrupt request, because it may be doing something else that must be finished first. For example, a program that talks to both a high-speed disk drive and low-speed keyboard should be designed to finish a disk transaction before handling a keyboard interrupt. Only when the CPU decides to acknowledge the interrupt does the CPU change the program counter to point to the device's handler. The interrupt handler operates much like a subroutine, except that it is not called by the executing program. The program that runs when no interrupt is being handled is often called the **foreground program**; when the interrupt handler finishes running in the background, it returns to the foreground program wherever processing was interrupted.

Before considering the details of how interrupts are implemented, let's look at the interrupt style of processing and compare it to busy-wait I/O. Example 3.4 uses interrupts as a basic replacement for busy-wait I/O.

Example 3.4 Copying Characters from Input to Output with Basic Interrupts

As with Example 3.3, we repeatedly read a character from an input device and write it to an output device. We assume that we can write C functions that act as interrupt handlers. Those handlers will work with the devices in much the same way as in busy-wait I/O by reading and writing status and data registers. The main difference is in handling the output; the interrupt signals that the character is done, so the handler doesn't have to do anything.

We use a global variable, `achar` for the input handler to pass the character to the foreground program. Because the foreground program doesn't know when an interrupt occurs, we also use a global Boolean variable, `gotchar` to signal when a new character has been received. Here is the code for the input and output handlers:

```
void input_handler() { /* get a character and put in global */
     achar = peek(IN_DATA); /* get character */
     gotchar = TRUE; /* signal to main program */
     poke(IN_STATUS,0); /* reset status to initiate next transfer */
}
void output_handler() { /* react to character being sent */
    /* don't have to do anything */
}
```

The main program is reminiscent of the busy-wait program. It looks at `gotchar` to check when a new character has been read, and then, immediately sends it out to the output device.

```
main() {
     while (TRUE) { /* read then write forever */
          if (gotchar) { /* write a character */
               poke(OUT_DATA,achar); /* put character in device */
               poke(OUT_STATUS,1); /* set status to initiate write */
               gotchar = FALSE; /* reset flag */
          }
     }
}
```

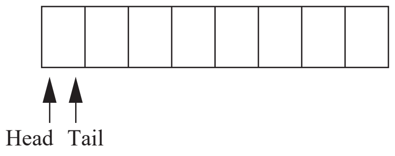
The use of interrupts has made the main program somewhat simpler. However, this program design still does not allow the foreground program to do useful work. Example 3.5 uses a more sophisticated program design to let the foreground program work completely independently of I/O.

Example 3.5 Copying Characters from Input to Output with Interrupts and Buffers

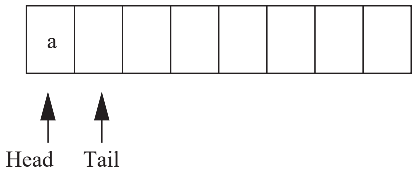
Because we don't need to wait for each character, we can make this I/O program more sophisticated than the one in Example 3.4. Rather than reading a single character and then writing it, the program performs reading and writing independently. We use an elastic buffer to hold the characters. The read and write routines communicate through the global variables used to implement the elastic buffer:

- A character string, `io_buf`, holds a queue of characters that have been read, but not yet written.
- A pair of integers, `buf_start` and `buf_end`, will point to the first and last characters read, respectively.
- An integer, `error`, will be set to zero whenever `io_buf` overflows.

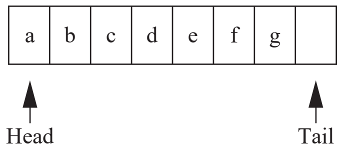
The elastic buffer allows the I/O devices to run at different rates. The queue, `io_buf`, acts as a wraparound buffer; we add characters to the tail when an input is received and take characters from the tail when we are ready for output. The head and tail wrap around the end of the buffer array to make the most efficient use of the array. Here is the situation at the start of the program's execution, where the tail points to the first available character and the head points to the ready character. As seen below, because the head and tail are equal, we know that the queue is empty.



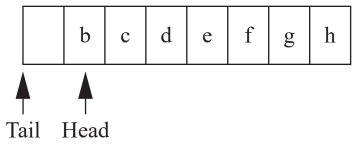
When the first character is read, the tail is incremented after the character is added to the queue, leaving the buffer and pointers to look like this:



When the buffer is full, we leave one character in the buffer unused. As the next figure shows, if we were to add another character and update the tail buffer (wrapping it around to the head of the buffer), we would be unable to distinguish a full buffer from an empty one.



Here is what happens when the output goes past the end of `io_buf`:



This code implements the elastic buffer, including declarations for the above global variables and some service routines for adding and removing characters from the queue. Because interrupt handlers are regular code, we can use subroutines to structure code, as with any program.

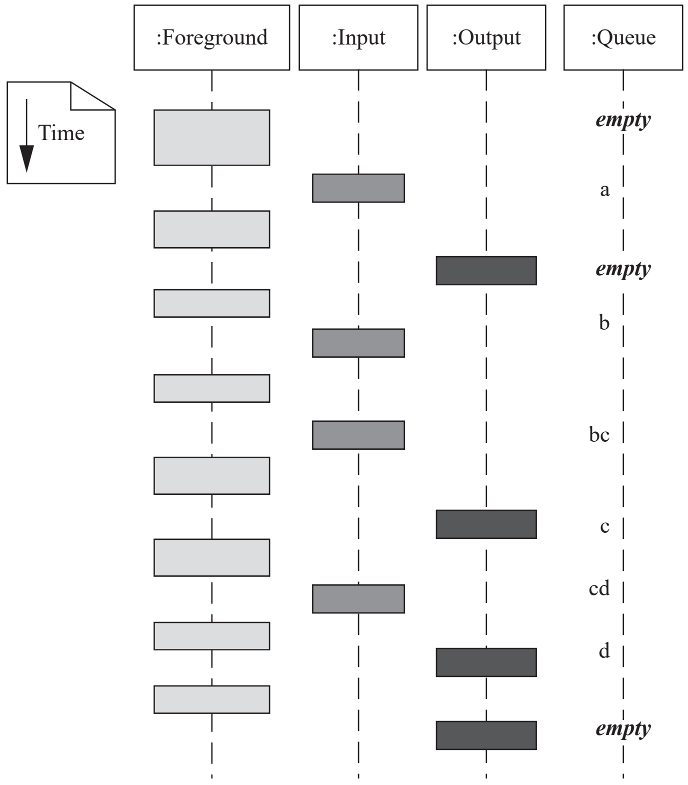
```
#define BUF_SIZE 8
char io_buf[BUF_SIZE]; /* character buffer */
int buf_head = 0, buf_tail = 0; /* current position in buffer */
int error = 0; /* set to 1 if buffer ever overflows */

void empty_buffer() { /* returns TRUE if buffer is empty */
    (buf_head == buf_tail) ? TRUE : FALSE;
}

void full_buffer() { /* returns TRUE if buffer is full */
     ((buf_tail+1) % BUF_SIZE == buf_head) ? TRUE : FALSE;
}

int nchars() { /* returns the number of characters in the buffer */
     if (buf_tail >= buf_head) return buf_tail - buf_head;
     else return BUF_SIZE - buf_head + buf_tail;
}

void add_char(char achar) { /* add a character to the buffer head */
    io_buf[buf_tail++] = achar;
    /* check pointer */
    if (buf_tail == BUF_SIZE)
        buf_tail = 0;
}

char remove_char() { /* take a character from the buffer head */
     char achar;
     achar = io_buf[buf_head++];
     /* check pointer */
     if (buf_head == BUF_SIZE)
         buf_head = 0;
     return achar;
}
```

Assume that we have two interrupt handling routines defined in C: `input_handler` for the input device and `output_handler` for the output device. These routines work with the device in much the same way as the busy-wait routines. The only complication is in starting the output device: If `io_buf` has characters waiting, the output driver can start a new output transaction by itself. However, if there are no characters waiting, an outside agent must start a new output action whenever the new character arrives. Rather than force the foreground program to look at the character buffer, we will have the input handler check to see whether there is only one character in the buffer and start a new transaction.

Here is the code for the input handler:

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
     char achar;
     if (full_buffer()) /* error */
          error = 1;
     else { /* read the character and update pointer */
          achar = peek(IN_DATA); /* read character */
          add_char(achar); /* add to queue */
    }
    poke(IN_STATUS,0); /* set status register back to 0 */
    /* if buffer was empty, start a new output transaction */
   if (nchars() == 1) { /*buffer had been empty until this interrupt */
        poke(OUT_DATA,remove_char()); /* send character */
        poke(OUT_STATUS,1); /* turn device on */
    }
}
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
void output_handler() {
     if (!empty_buffer()) { /* start a new character */
         poke(OUT_DATA,remove_char()); /* send character */
         poke(OUT_STATUS,1); /* turn device on */
     }
}
```

The foreground program does not need to do anything; everything is taken care of by the interrupt handlers. The foreground program is free to do useful work, as it is occasionally interrupted by input and output operations. The following sample execution of the program in the form of a UML sequence diagram shows how input and output are interleaved with the foreground program. We have kept the last input character in the queue until the output is complete to make it clearer when input occurs. The simulation shows that the foreground program is not executing continuously, but it continues to run in its regular state, independent of the number of characters waiting in the queue.



Interrupts allow a great deal of concurrency, which can make proficient use of the CPU. However, when interrupt handlers are buggy, the errors can be complex to find. The fact that an interrupt can occur at any time means that the same bug can manifest itself in different ways when the interrupt handler interrupts different segments of the foreground program.

Example 3.6 illustrates the problems inherent in debugging interrupt handlers.

Example 3.6 Debugging Interrupt Code

Assume that the foreground code performs a matrix multiplication operation, $y = Ax + b$:

```
for (i = 0; i <M; i++) {
    y[i] = b[i];
    for (j = 0; j <N; j++)
        y[i] = y[i] + A[i][j] *x[j];
  }
```

We use the interrupt handlers of Example 3.6 to perform I/O while the matrix computation is performed, but with one small change: `read_handler` has a bug that causes it to change the value of j. Although this may seem far-fetched, remember that when the interrupt handler is written in assembly language, such bugs are easy to introduce. Any CPU register written by the interrupt handler must be saved before it is modified and restored before the handler exits. Any type of bug, such as forgetting to save the register or to properly restore it, can cause that register to mysteriously change value in the foreground program.

What happens to the foreground program when `j` changes the value during an interrupt depends on when the interrupt handler executes. Because the value of `j` is reset at each iteration of the outer loop, the bug will affect only one entry of the result, `y`. Clearly, the entry that changes will depend on when the interrupt occurs. Furthermore, the change observed in `y` depends not only on what new value is assigned to `j`, which may depend on the data handled by the interrupt code, but also when in the inner loop the interrupt occurs. An interrupt at the beginning of the inner loop will give a different result than one that occurs near the end. The number of possible new values for the result vector is much too large to consider manually, and the bug can't be found by enumerating the possible wrong values and correlating them with a given root cause. Even recognizing the error can be difficult. For example, an interrupt that occurs at the very end of the inner loop will not cause any change in the foreground program's results. Finding such bugs generally requires a great deal of tedious experimentation and frustration.

The CPU implements interrupts by checking the interrupt request line at the beginning of the execution of every instruction. If an interrupt request has been asserted, the CPU does not fetch the instructions pointed to by the PC. Instead, the CPU sets the PC to a predefined location, which is the beginning of the interrupt-handling routine. The starting address of the interrupt handler is usually given as a pointer. Rather than defining a fixed location for the handler, the CPU defines a location in the memory that holds the address of the handler, which can then reside anywhere in memory.

Interrupts and subroutines

Because the CPU checks for interrupts at every instruction, it can respond quickly to service requests from devices. However, the interrupt handler must return to the foreground program without disturbing the foreground program's operation. Because subroutines perform a similar function, it is natural to build the CPU's interrupt mechanism to resemble its subroutine function. Most CPUs use the same basic mechanism for remembering the foreground program's PC as is used for subroutines. The subroutine call mechanism in modern microprocessors is typically a stack, so the interrupt mechanism puts the return address on the stack. Some CPUs use the same stack as for subroutines, whereas others define a special stack. The use of a procedure-like interface also makes it easier to provide a high-level language interface for interrupt handlers. The details of the C interface to interrupt handling routines vary with both the CPU and the underlying support software.

Priorities and vectors

Providing a practical interrupt system requires more than a simple interrupt request line. Most systems have more than one I/O device, so there must be some mechanism for allowing multiple devices to interrupt. We also want to have flexibility in the locations of the interrupt-handling routines, the addresses for devices, and so on. There are two ways in which interrupts can be generalized to handle multiple devices and to provide more flexible definitions for the associated hardware and software:

- **Interrupt priorities** allow the CPU to recognize some interrupts as more important than others.
- **Interrupt vectors** allow the interrupting device to specify its handler.

Prioritized interrupts not only allow multiple devices to be connected to the interrupt line, but they also allow the CPU to ignore less important interrupt requests while handling more important requests. As shown in Fig. 3.3, the CPU provides several different interrupt request signals, shown here as *L*1, *L*2,... up to *Ln*. Typically, the lower-numbered interrupt lines are given higher priority, so in this case, if devices 1, 2, and *n* all requested interrupts simultaneously, 1's request would be acknowledged because it is connected to the highest-priority interrupt line. Rather than provide a separate interrupt acknowledge line for each device, most CPUs use a set of signals that provide the priority number of the winning interrupt in binary form, so that interrupt level 7 requires three bits rather than seven. A device knows that its interrupt request was accepted by seeing its own priority number on the interrupt acknowledge lines.

How do we change the priority of a device? Simply connect it to a different interrupt request line. This requires hardware modification. Hence, if priorities need to be changeable, removable cards, programmable switches, or some other mechanism should be provided to make the change easy.

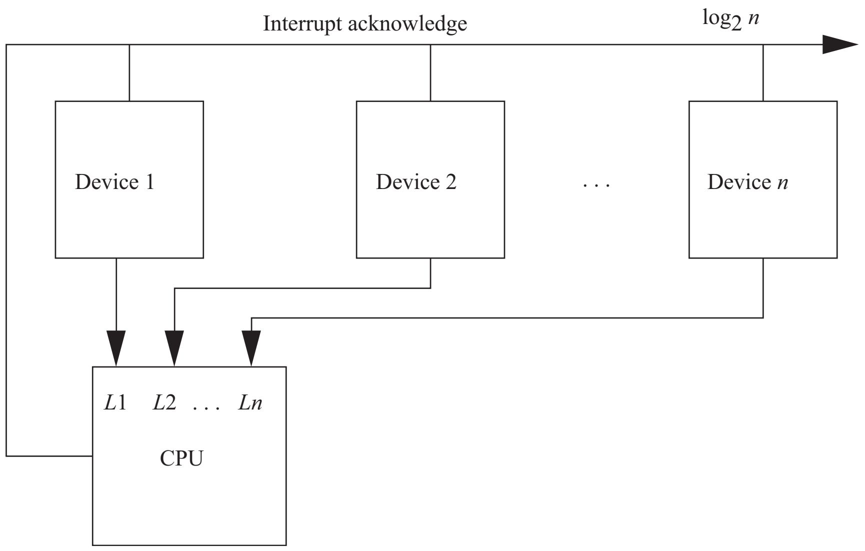


FIGURE 3.3

Prioritized device interrupts.

The priority mechanism must ensure that a lower-priority interrupt does not occur when a higher-priority interrupt is handled. The decision process is known as **masking**. When an interrupt is acknowledged, the CPU stores the priority level of that interrupt in an internal register. When a subsequent interrupt is received, its priority is checked against the priority register; the new request is acknowledged only if it has higher priority than the currently pending interrupt. When the interrupt handler exits, the priority register must be reset. The need to reset the priority register is one reason why most architectures introduce a specialized instruction to return from interrupts, rather than using the standard subroutine return instruction.

Power-down interrupts

The highest-priority interrupt is normally called the **nonmaskable interrupt** (**NMI**). The NMI cannot be turned off and is usually reserved for interruptions caused by power failures. A simple circuit can be used to detect a dangerously low power supply, and the NMI interrupt handler can be used to save critical state in nonvolatile memory, turn off I/O devices to eliminate spurious device operation during power-down, and so on.

Most CPUs provide a relatively small number of interrupt priority levels, such as eight. Although more priority levels can be added with external logic, they may not be necessary in all cases. When several devices naturally assume the same priority, such as when you have several identical keypads attached to a single CPU, you can combine polling with prioritized interrupts to efficiently handle the devices. As shown in Fig. 3.4, you can use a small amount of logic external to the CPU to generate an interrupt whenever any of the devices you want to group together request service.

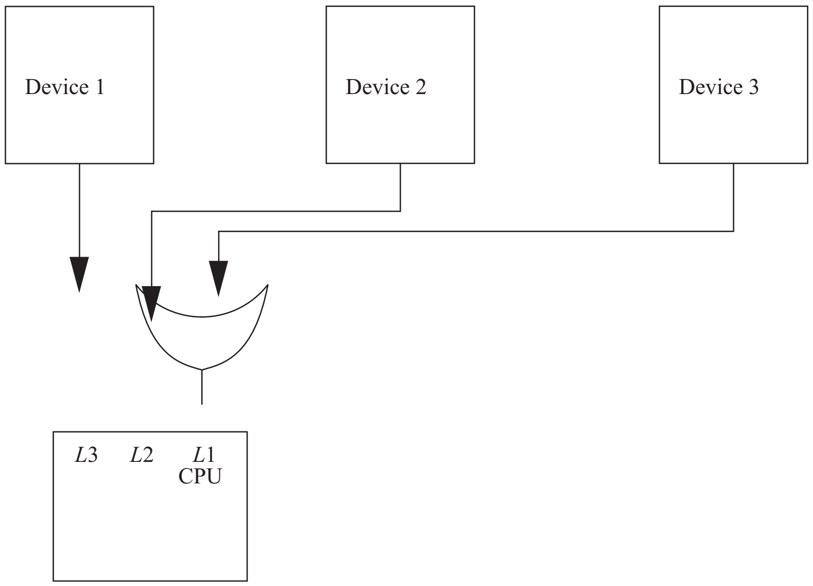


FIGURE 3.4

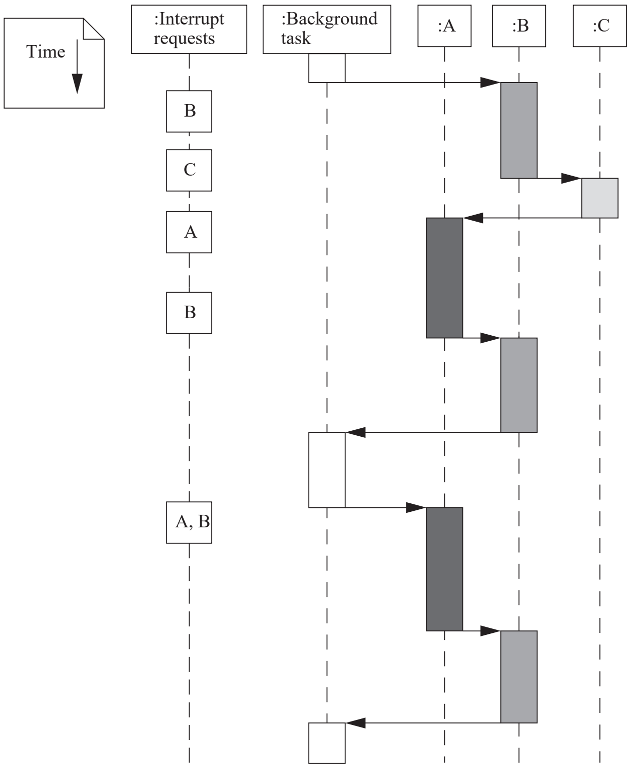
Using polling to share an interrupt over several devices.

The CPU will call the interrupt handler associated with this priority; the handler does not know which of the devices actually requested the interrupt. The handler uses software polling to check the status of each device. In this example, it will read the status registers of 1, 2, and 3 to see which of them is ready and requesting service.

Example 3.7 illustrates how priorities affect the order in which I/O requests are handled.

Example 3.7 I/O with Prioritized Interrupts

Assume that we have devices A, B, and C. A has priority 1 (highest priority), B has priority 2, and C has priority 3. This UML sequence diagram shows which interrupt handler executes as a function of time for a sequence of interrupt requests.



In each case, an interrupt handler keeps running until either it finishes, or a higher-priority interrupt arrives. The C interrupt handler is delayed by the handling required for the first B interrupt. The second B interrupt handler is similarly delayed by service for the A interrupt. When both A and B interrupt simultaneously, A's interrupt gets priority; when A's handler is finished, the priority mechanism automatically answers B's pending interrupt.

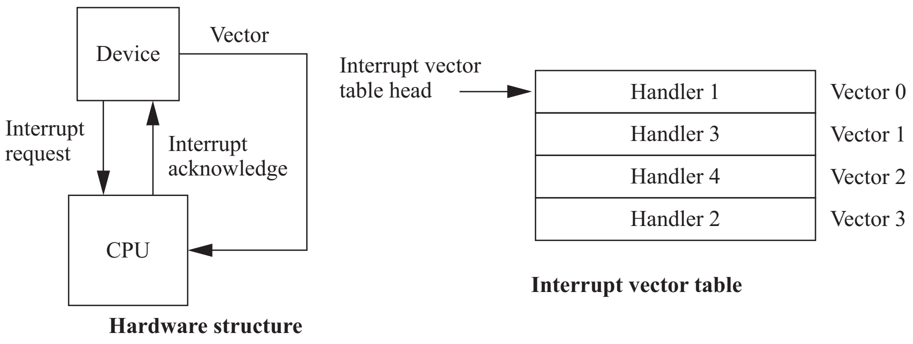


FIGURE 3.5

Interrupt vectors.

Vectors provide flexibility in a different dimension: the ability to define the interrupt handler that should service a request from a device. Fig. 3.5 shows the hardware structure required to support interrupt vectors. In addition to the interrupt request and acknowledge lines, additional interrupt vector lines run from the devices to the CPU. After a device's request is acknowledged, it sends its interrupt vector over those lines to the CPU. The CPU then uses the vector number as an index in a table stored in memory, as shown in Fig. 3.5. The location referenced in the interrupt vector table by the vector number gives the address of the handler.

There are two important things to notice about the interrupt vector mechanism. First, the device, not the CPU, stores its vector number. In this way, a device can be given a new handler simply by changing the vector number it sends without modifying the system software. For example, vector numbers can be changed by programmable switches. The second thing to notice is that there is no fixed relationship between vector numbers and interrupt handlers. The interrupt vector table allows arbitrary relationships between devices and handlers. The vector mechanism provides great flexibility in the coupling of hardware devices and the software routines that service them.

Most modern CPUs implement both prioritized and vectored interrupts. Priorities determine which device is serviced first, and vectors determine what routine is used to service the interrupt. The combination of the two provides a rich interface between hardware and software.

Interrupt overhead

Now that we have a basic understanding of the interrupt mechanism, we can consider the complete interrupt-handling process. When a device requests an interrupt, some steps are performed by the CPU, some by the device, and others by software:

1. *CPU*: The CPU checks for pending interrupts at the beginning of an instruction. It answers the highest-priority interrupt, which has a higher priority than that given in the interrupt priority register.
2. *Device*: The device receives the acknowledgment and sends the CPU its interrupt vector.

3. *CPU*: The CPU looks up the device handler address in the interrupt vector table using the vector as an index. A subroutine-like mechanism is used to save the current value of the PC and possibly other internal CPU states, such as general-purpose registers.
4. *Software*: The device driver may save an additional CPU state. It then performs the required operations on the device. It then restores any saved state and executes the interrupt return instruction.
5. *CPU*: The interrupt return instruction restores the PC and other automatically saved states to return execution to the interrupted code.

Interrupts do not come without a performance penalty. In addition to the execution time required for the code that talks directly to the devices, there is an execution time overhead associated with the interrupt mechanism:

- The interrupt itself has overhead comparable to a subroutine call. Because an interrupt causes a change in the program counter, it incurs a branch penalty. In addition, if the interrupt automatically stores CPU registers that action requires extra cycles, even if the state is not modified by the interrupt handler.
- In addition to the branch delay penalty, the interrupt requires extra cycles to acknowledge the interrupt and obtain the vector from the device.
- In general, the interrupt handler will save and restore CPU registers that were not automatically saved by the interrupt.
- The interrupt return instruction incurs a branch penalty, as well as the time required to restore the automatically saved state.

The time required for the hardware to respond to the interrupt, obtain the vector, and so on cannot be changed by the programmer. CPUs vary quite a bit amounting to internal state automatically saved by an interrupt. The programmer does have control over what state is modified by the interrupt handler, and it must be saved and restored. Careful programming can sometimes result in a small number of registers used by an interrupt handler, thereby saving time in maintaining the CPU state. However, such tricks usually require coding the interrupt handler in assembly language rather than a high-level language.

Interrupts in Arm

Arm7 supports two types of interrupts: fast interrupt requests (FIQs) and interrupt requests (IRQs). An FIQ takes priority over an IRQ. The interrupt table is always kept in the bottom memory addresses, starting at location zero. The entries in the table typically contain subroutine calls to the appropriate handler.

The Arm7 performs the following steps when responding to an interrupt [ARM99B]:

- saves the appropriate value of the PC to be used to return,
- copies the current program status register (CPSR) into a saved program status register (SPSR),

- forces bits in the CPSR to note the interrupt, and
- forces the PC to the appropriate interrupt vector.

When leaving the interrupt handler, the handler should:

- restore the proper PC value,
- restore the CPSR from the SPSR, and
- clear the interrupt-disable flags.

The worst-case latency to respond to an interrupt includes the following components:

- two cycles to synchronize the external request,
- up to 20 cycles to complete the current instruction,
- three cycles for data abort, and
- two cycles to enter the interrupt-handling state.

This adds up to 27 clock cycles. The best-case latency is four clock cycles.

A vectored interrupt controller (VIC) can be used to provide up to 32 vectored interrupts [ARM02]. The VIC is assigned a block of memory for its registers; the VIC base address should be in the upper 4K of memory to avoid increasing the access times of the controller's registers. An array, `VICVECTADDR`, in this block specifies the interrupt service routines' addresses; the array, `VICVECTPRIORITY`, gives the priorities of the interrupt sources.

Interrupts in C55x

Interrupts in the C55x [Tex04] take at least seven clock cycles. In many situations, they take 13 clock cycles.

- A maskable interrupt is processed in several steps when the interrupt request is sent to the CPU.
- The interrupt flag register (IFR) corresponding to the interrupt is set.
- The interrupt enable register (IER) is checked to ensure that the interrupt is enabled.
- The interrupt mask register (INTM) is checked to ensure that the interrupt is not masked.
- The IFR corresponding to the flag is cleared.
- Appropriate registers are saved as context.
- INTM is set to one to disable maskable interrupts.
- The disable bug mask bit (DGBM) is set to one to disable debug events.
- EALLOW is set to zero to disable access to nonCPU emulation registers.
- A branch is performed on the interrupt service routine.

The C55x provides two mechanisms—**fast-return** and **slow-return**—to save and restore registers for interrupts and other context switches. Both processes save the return address and loop context registers. The fast-return mode uses RETA to save the return address and CFCT for the loop context bits. In contrast, the slow-return mode saves the return address and loop context bits on the stack.

Interrupts in PIC16F

The PIC16F recognizes two types of interrupts. Synchronous interrupts generally occur from sources inside the CPU. Asynchronous interrupts are generally triggered from outside the CPU. The INTCON register contains the major control bits for the interrupt system. The Global Interrupt Enable (GIE) bit is used to allow all unmasked interrupts. The Peripheral Interrupt Enable bit (PEIE) enables/disables interrupts from peripherals. The TMR0 Overflow Interrupt Enable bit enables or disables the timer zero overflow interrupt. The INT External Interrupt Enable bit enables/disables the INT external interrupts. Peripheral Interrupt Flag registers PIR1 and PIR2 hold flags for peripheral interrupts.

The RETFIE instruction is used to return from an interrupt routine. This instruction clears the GIE bit, reenabling pending interrupts.

The latency of synchronous interrupts is $3T_{CY}$, where T_{CY} is the length of an instruction. The latency for asynchronous interrupts is 3 to $3.75T_{CY}$. One- and two-cycle instructions have the same interrupt latencies.

3.3 Supervisor mode, exceptions, and traps

In this section, we consider exceptions and traps. These are mechanisms to handle internal conditions, and they are analogous to interrupts in form. We begin with a discussion of supervisor mode, which some processors use to handle exceptional events and protect executing programs from each other.

3.3.1 Supervisor mode

As will become clearer in later chapters, complex systems are often implemented as several programs that communicate with each other. These programs may run under the command of an operating system. Thus, it may be desirable to provide hardware checks to ensure that the programs do not interfere with each other, for example, by erroneously writing into a segment of the memory used by another program. Software debugging is important, but it can leave some problems in a running system; hardware checks ensure an additional level of safety.

In such cases, it is often useful to have a **supervisor mode** provided by the CPU. Normal programs run in **user mode**. The supervisor mode has privileges that the user modes do not. For example, we will study memory management systems in Section 3.5 that allow the physical addresses of memory locations to be changed dynamically. Control of the memory management unit is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers, effectively moving code and data in the middle of program execution.

Arm supervisor mode

Not all CPUs have supervisor modes. Many DSPs, including the C55x, do not provide one. Arm, however, has a supervisor mode. The Arm instruction that puts the CPU in supervisor mode is called SWI:

```
SWI CODE_1
```

It can, of course, be executed conditionally, as with any Arm instruction. SWI causes the CPU to go into supervisor mode and sets the PC to 0x08. The argument to `SWI` is a 24-bit immediate value that is passed on to the supervisor mode code; it allows the program to request various services from the supervisor mode.

In supervisor mode, the bottom five bits of the CPSR are all set to 1 to indicate that the CPU is in supervisor mode. The old value of the CPSR just before the SWI, stored in a register, is the SPSR. There are, in fact, several SPSRs for different modes; the supervisor mode SPSR is referred to as `SPSR_svc`.

To return from supervisor mode, the supervisor restores the PC from register r14 and restores the CPSR from `SPSR_svc`.

3.3.2 Exceptions

An **exception** is an internally detected error. A simple example is a division by zero. One way to handle this problem is to check every divisor before division to be sure it is not zero, but this would substantially increase the size of numerical programs and cost a great deal of CPU time evaluating the divisor's value. The CPU can more efficiently check the divisor's value during execution. Because the time at which a zero divisor will be found is not known in advance, this event is like an interrupt, except that it is generated inside the CPU. The exception mechanism provides a way for the program to react to such unexpected events. Resets, undefined instructions, and illegal memory accesses are other typical examples of exceptions.

Just as interrupts can be seen as an extension of the subroutine mechanism, exceptions are generally implemented as a variation of an interrupt. Because both deal with changes in the flow of control of a program, it makes sense to use similar mechanisms. However, exceptions are generated internally.

Exceptions generally require both prioritization and vectoring. Exceptions must be prioritized because a single operation may generate more than one exception, such as an illegal operand and an illegal memory access. The priority of exceptions is usually fixed by the CPU architecture. Vectoring provides a way for the user to specify the handler for the exception condition. The vector number for an exception is usually predefined by the architecture; it is used to index into a table of exception handlers.

3.3.3 Traps

A **trap**, also known as a **software interrupt**, is an instruction that explicitly generates an exception condition. The most common use of a trap is to enter supervisor mode. The entry into supervisor mode must be controlled to maintain security; if the interface between the user and supervisor mode is improperly designed, a user program may be able to sneak code into the supervisor mode that could be executed to perform harmful operations.

Arm provides the SWI interrupt for software interrupts. This instruction causes the CPU to enter supervisor mode. An opcode is embedded in the instruction that can be read by the handler.

3.4 Coprocessors

CPU architects often want to provide flexibility in terms of what features are implemented in the CPU. One way to provide such flexibility at the instruction set level is to allow **coprocessors**, which are attached to the CPU, and implement some of the instructions. For example, floating-point arithmetic was introduced into the Intel architecture by providing separate chips that implemented the floating-point instructions.

To support coprocessors, certain opcodes must be reserved in the instruction set for coprocessor operations. Because it executes instructions, a coprocessor must be tightly coupled to the CPU. When the CPU receives a coprocessor instruction, the CPU must activate the coprocessor and pass it the relevant instruction. Coprocessor instructions can load and store coprocessor registers, or perform internal operations. The CPU can suspend execution to wait for the coprocessor instruction to finish; it can also take a more superscalar approach and continue executing instructions while waiting for the coprocessor to finish.

A CPU may, of course, receive coprocessor instructions, even when there is no coprocessor attached. Most architectures use illegal instruction traps to handle these situations. The trap handler can detect the coprocessor instruction and, for example, execute it in software on the main CPU. Emulating coprocessor instructions in software is slower but provides compatibility.

Coprocessors in Arm

The Arm architecture provides support for up to 16 coprocessors attached to a CPU. Coprocessors can perform load and store operations on their own registers. They can also move data between the coprocessor registers and the main ARM registers.

An example of an Arm coprocessor is the floating-point unit. The unit occupies two coprocessor units in the Arm architecture, numbered 1 and 2, but it appears as a single unit to the programmer. It provides eight 80-bit floating-point data registers, floating-point status registers, and an optional floating-point status register.

3.5 Memory system mechanisms

Modern microprocessors do more than just read and write monolithic memories. Architectural features improve both the speed and capacity of memory systems. Microprocessor clock rates are increasing at a faster rate than memory speeds, such that memories fall farther behind microprocessors every day. As a result, computer architects resort to **caches** to increase the average performance of memory systems. Although memory capacity is increasing steadily, program sizes are also increasing, and designers may not be willing to pay for all the memory demanded by an application.

Memory management units (MMUs) perform address translations that provide a larger virtual memory space in a small physical memory. **Memory protection units** (**MPUs**) provide memory protection mechanisms. In this section, we review both types of caches: MMUs, and MPUs.

3.5.1 Caches

Caches are widely used to speed up reads and writes in memory systems. Many microprocessor architectures include caches as part of their definitions. The cache speeds up the average memory access time when properly used. This increases the variability of memory access times. Accesses in the cache will be fast, but access to locations not cached will be slow. This variability in performance makes it especially important to understand how caches work so that we can better understand how to predict cache performance and factor these variations into system design.

Cache controllers

A cache is a small, fast memory that holds copies of some of the contents of the main memory. Because the cache is fast, it provides higher-speed access for the CPU, but because it is small, not all requests can be satisfied by the cache, forcing the system to wait for the slower main memory. Caching makes sense when the CPU uses only a relatively small set of memory locations at a time; the set of active locations is often called the **working set**.

Fig. 3.6 shows how the cache supports reads in the memory system. A **cache controller** mediates between the CPU and the memory system, which comprises the cache and main memory. The cache controller sends a memory request to the cache and the main memory. If the requested location is in the cache, the cache controller forwards the location's contents to the CPU and aborts the main memory request. This condition is a **cache hit**. If the location is not in the cache, the controller waits for the value from the main memory and forwards it to the CPU; this situation is a **cache miss**.

We can classify cache misses into several types, depending on the situation that generated them:

- A **compulsory miss** (i.e., **cold miss**) occurs the first time a location is used.

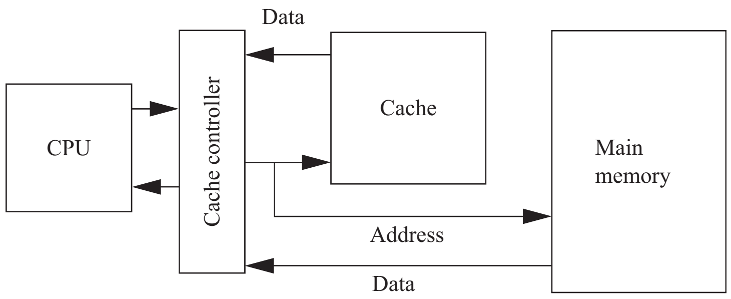


FIGURE 3.6

The cache in the memory system.

- A **capacity miss** is caused by a too-large working set.
- A **conflict miss** happens when two locations map to the same location in the cache.

Memory system performance

Even before we consider ways to implement caches, we can write some basic formulas for memory system performance. Let h be the **hit rate**, the probability that a given memory location is in the cache. It follows that 1-h is the **miss rate** or the probability that the location is not in the cache. Then, we can compute the average memory access time as

$$t_{av} = ht_{cache} + (1-h)t_{main}, \qquad \text{(Eq. 3.1)}$$

where t_{cache} is the access time of the cache, and t_{main} is the main memory access time. The memory access times are basic parameters provided by the memory manufacturer. The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators. The best-case memory access time, ignoring cache controller overhead, is t_{cache}, whereas the worst-case access time is *tmain*. Given that t_{main} is typically 50 to 75 ns, and t_{cache} is at most a few nanoseconds, the spread between worst- and best-case memory delays is substantial.

Cache organization

The simplest way to implement a cache is a **direct-mapped cache**, as shown in Fig. 3.8. The cache consists of cache **blocks**, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is divided into three sections. The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location. If the length of the data field is longer than the minimum addressable unit, the lowest bits of the address will be used as an offset to select the required value from the data field. Given the structure of the cache, only one block must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache.

Writes are slightly more complicated than reads because we must update the main memory as well as the cache. There are several methods by which we can do this. The

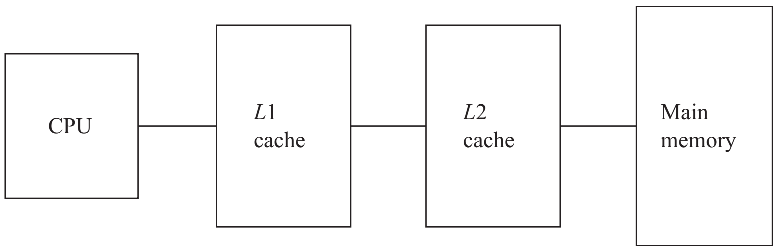


FIGURE 3.7

A two-level cache system.

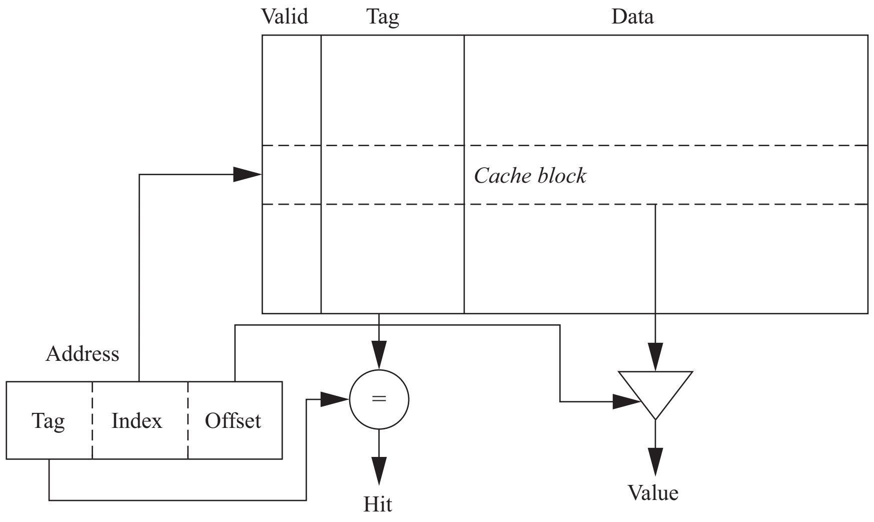


FIGURE 3.8

A direct-mapped cache.

simplest scheme is known as a **write-through**: every write changes both the cache and corresponding main memory location, usually through a write buffer. This scheme ensures that the cache and main memory are consistent but may generate additional main memory traffic. We can reduce the number of times we write to the main memory by using a **write-back** policy. If we write only when we remove a location from the cache, we eliminate the write when a location is written several times before it is removed from the cache.

The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power to its simple scheme for mapping the cache onto the main memory. Consider a direct-mapped cache with four blocks in which locations 0, 1, 2, and 3 all map to different blocks. However, locations 4, 8, 12,... all map to the same block as location 0; locations 1, 5, 9, 13,... all map to a single block; and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. As seen in Section 5.7, this can create program performance problems.

The limitations of the direct-mapped cache can be reduced by using the **set-associative** cache structure that is shown in Fig. 3.9. A set-associative cache is characterized by the number of **banks** or **ways** used, giving an *n*-way set-associative cache. A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit. Although memory locations map onto blocks using the same function, there are *n* separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that

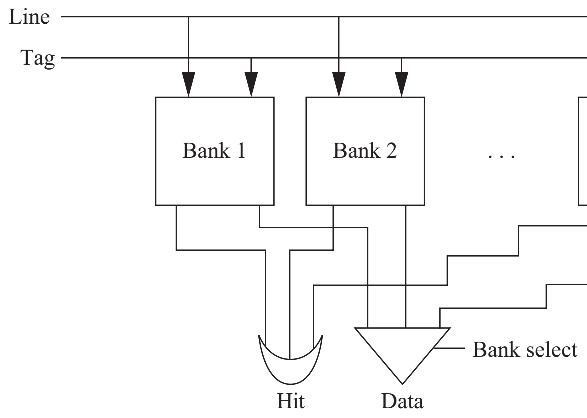


FIGURE 3.9

A set-associative cache.

happen to map onto the same cache block. The set-associative cache structure incurs a little extra overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

The set-associative cache generally provides higher hit rates than the direct-mapped cache because conflicts between a small set of locations can be resolved within the cache. The set-associative cache is somewhat slower, so the CPU designer must be careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs exhibit good behavior in the cache. It is relatively easy to determine when two memory locations conflict in a direct-mapped cache. Conflicts in a set-associative cache are more subtle; thus, the behavior of a set-associative cache is more difficult to analyze for both humans and programs.

Example 3.8 compares the behavior of direct-mapped and set-associative caches.

Example 3.8 Direct-mapped vs. Set-associative Caches

For simplicity, let's consider a very simple caching scheme. We use two bits of the address as the tag. We compare a direct-mapped cache with four blocks and a two-way set-associative cache with four sets, and we use least-recently used (LRU) replacement to make it easy to compare the two caches.

Here are the contents of memory using a 3-bit address for simplicity.

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

We will give each cache the same pattern of addresses in binary to simplify picking out the index: 001, 010, 011, 100, 101, and 111. To understand how the direct-mapped cache works, let's see how its state evolves.

After 001 access:

Block	Tag	Data
00	—	—
01	0	1111
10	—	—
11	—	—

After 010 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	—	

After 011 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	0	0110

After 100 access (notice that the tag bit for this entry is 1):

Block	Tag	Data
00	1	1000
01	0	1111
10	0	0000
11	0	0110

After 101 access (overwrites the 01 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	0	0110

After 111 access (overwrites the 11 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	1	0100

We can use a similar procedure to determine what ends up in the two-way set-associative cache. The only difference is that we have some freedom when we have to replace a block with new data. To make the results easy to understand, we use a least-recently-used replacement policy. For starters, let's make each bank the size of the original direct-mapped cache. The final state of the two-way set-associative cache follows:

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
00	1	1000	—	—
01	0	1111	1	0001
10	0	0000	—	
11	0	0110	1	0100

Of course, this isn't a fair comparison for performance because the two-way set-associative cache has twice as many entries as the direct-mapped cache. Let's use a two-way, set-associative cache with two sets, giving us four blocks, the same number as in the direct-mapped cache. In this case, the index size is reduced to 1 bit, and the tag grows to 2 bits.

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
0	01	0000	10	1000
1	10	0001	11	0100

In this case, the cache contents significantly differ from either the direct-mapped cache or the four-block, two-way set-associative cache.

The CPU knows when it is fetching an instruction (the PC is used to calculate the address, either directly or indirectly) or the data. We can therefore choose whether to cache instructions, data, or both. If cache space is limited, instructions are the highest priority for caching because they usually provide the highest hit rates. A cache that holds both instructions and data is called a **unified cache**.

First- and second-level cache

Modern CPUs may use multiple levels of cache, as shown in Fig. 3.7. The **first-level cache** (commonly known as the **L1 cache**) is closest to the CPU, the **second-level cache (L2 cache)** feeds the first-level cache, and so on. In today's microprocessors, the first-level cache is often on-chip, and the second-level cache is off-chip, although we are starting to see on-chip second-level caches.

The second-level cache is much larger, but is also slower. If h_1 is the first-level hit rate and h_2 is the rate at which access hits the second-level cache, then the average access time for a two-level cache system is

$$t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1 - h_1 - h_2) t_{main} \quad \text{(Eq. 3.2)}$$

As the program's working set changes, we expect locations to be removed from the cache to make way for new locations. When set-associative caches are used, we must think about what happens when we throw out a value from the cache to make room for a new value. We do not have this problem in direct-mapped caches because every location maps onto a unique block. However, in a set-associative cache, we must decide which set will have its block thrown out to make way for the new block. One possible replacement policy is **least recently used** (**LRU**), that is, throw out the block that has been used farthest in the past. We can add relatively small amounts of hardware to the cache to keep track of the time since the last access for each block. Another policy is random replacement, which requires even less hardware to implement.

3.5.2 Memory management units and address translation

An MMU translates addresses between the CPU and physical memory. This translation process is often known as **memory mapping** because addresses are mapped from a logical space into a physical space. MMUs may not be used in processors for hard real-time applications because the memory mapping process introduces too much variation in execution time. However, many embedded systems with less stringent timing requirements include MMUs to support operating systems like Linux and rich sets of applications. It is helpful to understand the basics of MMUs for embedded systems that are complex enough to require them.

Many DSPs, including the C55x, don't use MMUs. Because DSPs are used for compute-intensive tasks, they often don't require hardware assist for logical address spaces.

Memory mapping philosophy

Early computers used MMUs to compensate for limited address space in their instruction sets. When memory became cheap enough that physical memory could be larger than the address space defined by the instructions, MMUs allowed software to manage multiple programs in a single physical memory, each with its own address space.

Virtual addressing

MMUs are used to provide **virtual addressing.** As shown in Fig. 3.10, the MMU accepts logical addresses from the CPU. Logical addresses refer to the program's abstract address space, but do not correspond to actual RAM locations. The MMU translates them from tables to physical addresses that correspond to the RAM. By changing the MMU's tables, you can change the physical location at which the program resides without modifying the program's code or data. We must, of course, move the program in the main memory to correspond to the memory mapping change.

Furthermore, if we add a secondary storage unit, such as a solid state disk, we can eliminate parts of the program from the main memory. In a virtual memory system, the MMU keeps track of which logical addresses are resident in the main memory; those

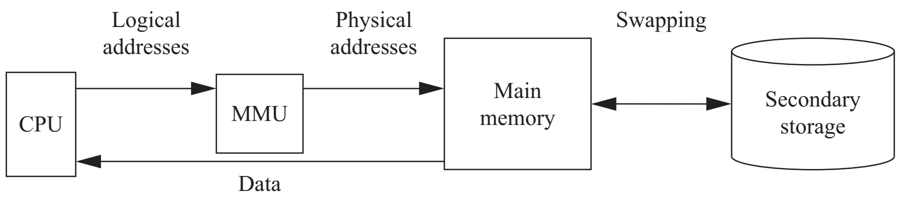


FIGURE 3.10

A virtually addressed memory system.

that do not reside in the main memory are kept on the secondary storage device. When the CPU requests an address that is not in the main memory, the MMU generates an exception called a **page fault.** The handler for this exception executes code that reads the requested location from the secondary storage device into the main memory. The program that generated the page fault is restarted by the handler only after:

- the required memory has been read back into the main memory, and
- the MMU's tables have been updated to reflect the changes.

Of course, loading a location into the main memory will usually require throwing something out of the main memory. The displaced memory is copied into secondary storage before the requested location is read in. As with caches, LRU is a good replacement policy.

There are two styles of address translation: **segmented** and **paged**. Each has advantages, and the two can be combined to form a segmented, paged addressing scheme. As illustrated in Fig. 3.11, segmenting is designed to support a large, arbitrarily sized region of memory, whereas pages describe small, equally sized regions.

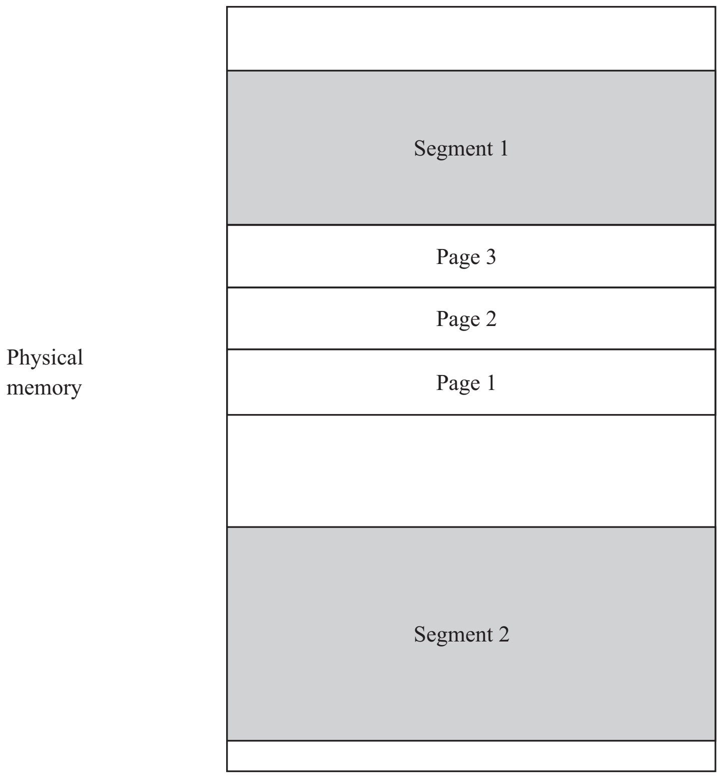


FIGURE 3.11

Segments and pages.

A segment is usually described by its start address and size, allowing different segments to be of different sizes. Pages are of uniform size, which simplifies the hardware required for address translation. A segmented, paged scheme is created by dividing each segment into pages and using two steps for address translation. Paging introduces the possibility of **fragmentation**, as program pages are scattered around physical memory.

In a simple segmenting scheme, as shown in Fig. 3.12, the MMU maintains a segment register that describes the currently active segment. This register would point to the base of the current segment. The address extracted from an instruction or from any other source for addresses (e.g., a register) would be used as the offset for the address. The physical address is formed by adding the segment base to the offset. Most segmentation schemes also check the physical address against the upper limit of the segment by extending the segment register to include the segment size and comparing the offset to the allowed size.

The translation of paged addresses requires more MMU states, but a simpler calculation than is the case for segmented addressing. As shown in Fig. 3.13, the logical address is divided into two sections: a page number and an offset. The page number is used as an index in a page table, which stores the physical address for the start of each page. However, because all pages have the same size, and it is easy to ensure that page boundaries fall on the proper boundaries, the MMU simply needs to concatenate the top bits of the page starting address with the bottom bits from the page offset to form the physical address. Pages are small, typically between 512 bytes and 4 KB. As a result, an architecture with a large address space requires a large page table. The page table is normally kept in the main memory, which means that an address translation requires memory access.

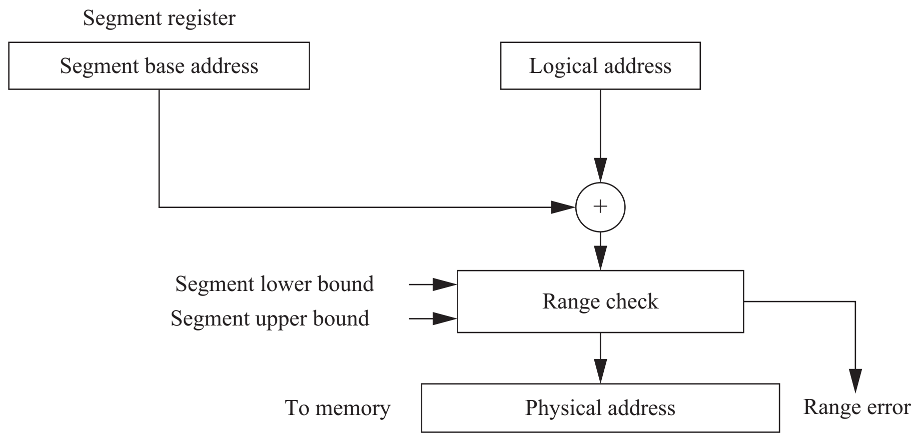


FIGURE 3.12

Address translation for a segment.

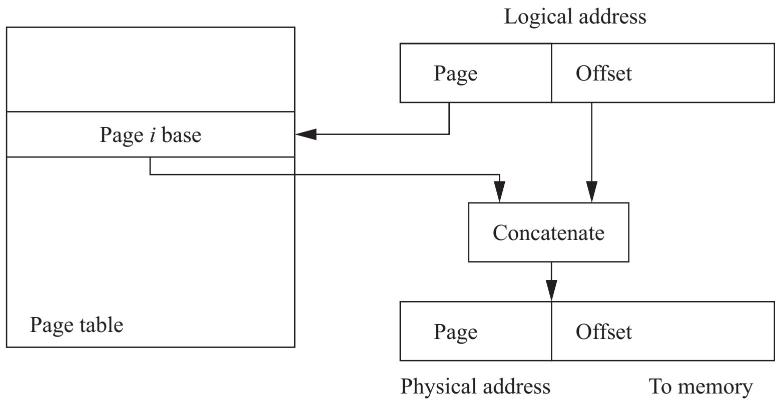


FIGURE 3.13

Address translation for a page.

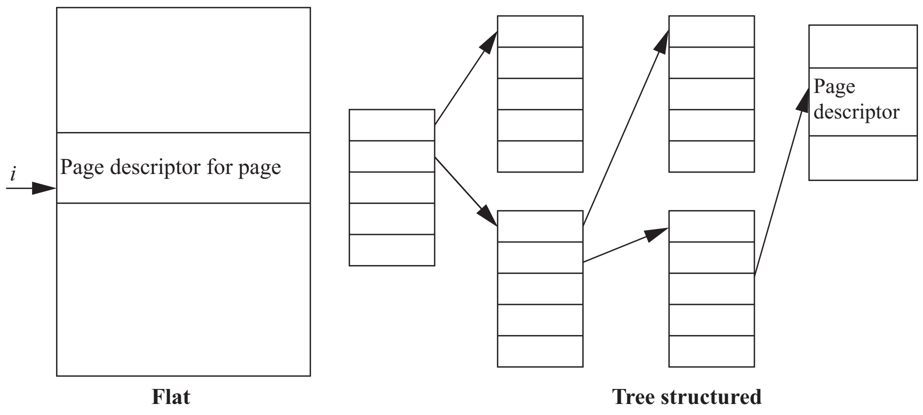


FIGURE 3.14

Alternative schemes for organizing page tables.

The page table may be organized in several ways, as shown in Fig. 3.14. The simplest scheme is a flat table. The table is indexed by the page number, and each entry holds the page descriptor. A more sophisticated method is a tree. The root entry of the tree holds pointers to pointer tables at the next level of the tree; each pointer table is indexed by a part of the page number. We eventually, after three levels in this case, arrive at a descriptor table, which includes the page descriptor in which we are interested. A tree-structured page table incurs some overhead for the pointers, but it also allows us to build a partially populated tree. If some part of the address space is not used, we do not need to build the part of the tree that covers it.

The efficiency of paged address translation may be increased by caching page translation information. The cache for address translation is known as a **translation lookaside buffer (TLB)**. The MMU reads the TLB to check whether a page number is currently in the TLB cache, and if so, uses that value rather than reading from memory.

Virtual memory is typically implemented in a paging or segmented paged scheme so that only page-sized regions of memory need to be transferred on a page fault. Some extensions to both segmenting and paging are useful for virtual memory.

- At a minimum, a present bit is necessary to show whether the logical segment or page is currently in physical memory.
- A dirty bit shows whether the page/segment has been written. This bit is maintained by the MMU because it knows about every write performed by the CPU.
- Permission bits are often used. Some pages/segments may be readable, but not writable. If the CPU supports modes, pages/segments may be accessible by the supervisor, but not in user mode.

A data or instruction cache may operate either on logical or physical addresses, depending on where it is positioned relative to the MMU.

An MMU is an optional part of the Arm architecture. The Arm MMU supports both virtual address translation and memory protection; the architecture requires that the MMU be implemented when cache or write buffers are implemented. The Arm MMU supports the following types of memory regions for address translation:

- A **section** is a 1-MB block of memory.
- A **large page** is 64 KB.
- A **small page** is 4 KB.

An address is marked as a section mapped or page mapped. A two-level scheme is used to translate addresses. The first-level table, which is pointed to by the Translation Table Base register, holds descriptors for section translation and pointers to the second-level tables. The second-level tables describe the translations of both large and small pages. The basic two-level process for a large or small page is illustrated in Fig. 3.15. The details differ between large and small pages, such as the size of the second-level table index. The first- and second-level pages also contain access control bits for virtual memory and protection.

3.5.3 Memory protection units

MMUs can provide access protection for memory, but they incur a significant run-time penalty as well as power consumption. An MPU provides access control with a lower overhead. The next example looks at the Arm MPU.

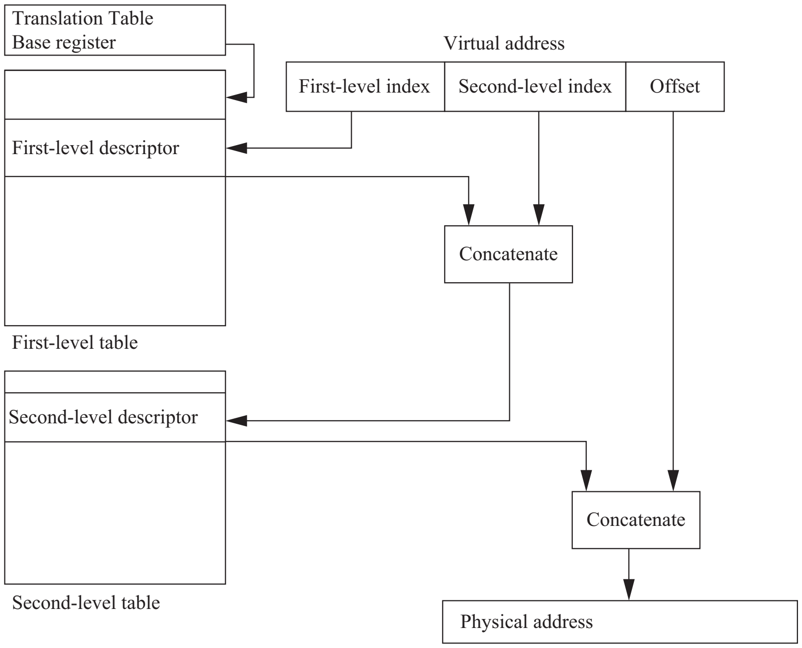


FIGURE 3.15

ARM two-stage address translation.

Example 3.9 Arm Memory Protection Unit

The Arm MPU allows privileged software, such as the operating system, to define up to 16 protected regions of memory. A protected region must be at least 32 bytes in size and no more than 4 GB; a region's size must be a multiple of 32 bytes and begin at a 32-byte aligned location. Normal memory can be assigned several attributes, including cacheability, shareability, and execute/never-execute. Memory associated with I/O devices may be given many attributes, including whether to gather multiple accesses into a single bus transaction, reordering/nonreordering, early/nonearly write acknowledgement.

3.6 CPU performance

Execution time (how fast the CPU executes instructions) is a particularly important topic in embedded computing. In this section, we consider two factors that can substantially influence program performance: pipelining and caching.

3.6.1 Pipelining

Modern CPUs are designed as **pipelined** machines in which several instructions are executed in parallel. Pipelining greatly increases the CPU's efficiency. However, like any pipeline, a CPU pipeline works best when its contents flow smoothly. Some sequences of instructions can disrupt the flow of information in the pipeline and, perhaps temporarily, slow down the operation of the CPU.

Arm7 pipeline

The Arm7 has a three-stage pipeline:

1. *Fetch:* The instruction is fetched from memory.
2. *Decode:* The instruction's opcode and operands are decoded to determine what function to perform.
3. *Execute:* The decoded instruction is executed.

Each of these operations requires one clock cycle for typical instructions. Thus, a normal instruction requires three clock cycles to execute completely. This is known as the **latency** of instruction execution. However, because the pipeline has three stages, an instruction is completed in every clock cycle. Thus, the pipeline has a **throughput** of one instruction per cycle. Fig. 3.16 illustrates the position of the instructions in the pipeline during execution using the notation introduced by Hennessy and Patterson [Hen06]. A vertical slice through the timeline shows all instructions in the pipeline at that time. By following an instruction horizontally, we can see the progress of its execution.

C55x pipeline

The C55x includes a seven-stage pipeline [Tex00B]:

1. *Fetch*,
2. *Decode*,
3. *Address:* computes data and branch addresses,
4. *Access 1:* reads data,
5. *Access 2:* finishes data read,
6. *Read stage:* puts operands onto internal busses, and
7. *Execute:* performs operations.

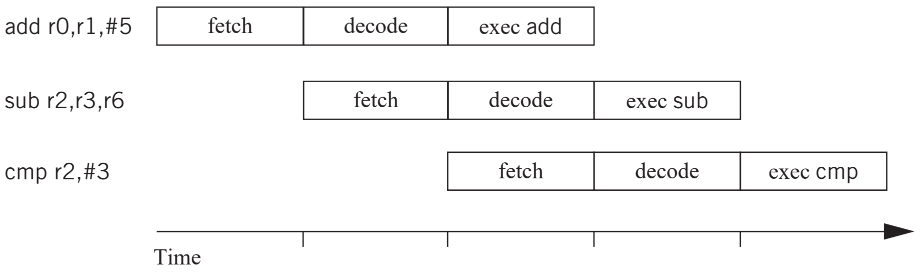


FIGURE 3.16

Pipelined execution of ARM instructions.

RISC machines are designed to keep the pipeline busy. Complex instruction-set computers may display a wide variation in instruction timing. Pipelined RISC machines typically have more regular timing characteristics, and most instructions that do not have pipeline hazards display the same latency.

Pipeline stalls

The one-cycle-per-instruction completion rate does not hold in every case. The simplest case for extended execution is when an instruction is too complex to complete the execution phase in a single cycle. A multiple-load instruction is an example of an instruction that requires several cycles in the execution phase. Fig. 3.17 illustrates a **data stall** in the execution of a sequence of instructions, starting with a load-multiple (`LDMIA`) instruction. Because there are two registers to load, the instruction must stay in the execution phase for two cycles. In a multiphase execution, the decode stage is also occupied because it must continue to remember the decoded instruction. As a result, the `SUB` instruction is fetched at the normal time, but is not decoded until the `LDMIA` is finished. This delays the fetching of the third instruction, the `CMP`.

Branches also introduce **control stall** delays into the pipeline, commonly referred to as a **branch penalty**, as shown in Fig. 3.18. The decision whether to take the conditional branch, `BNE`, is not made until the third clock cycle of that instruction's execution, which computes the branch target address. If the branch is taken, the succeeding

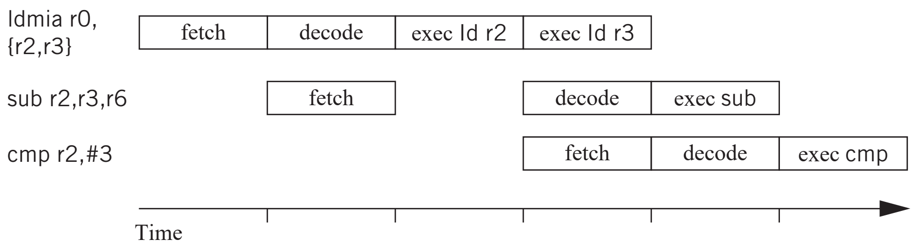


FIGURE 3.17

Pipelined execution of multicycle ARM instructions.

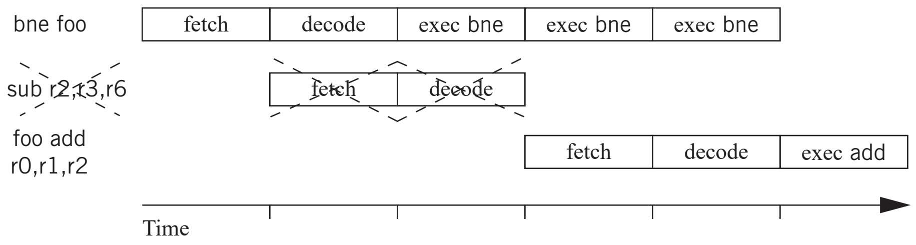


FIGURE 3.18

Pipelined execution of a branch in ARM.

instruction at PC+4 has been fetched and has started to be decoded. When the branch is taken, the branch target address is used to fetch the branch target instruction. Because we must wait for the execution cycle to complete before knowing the target, we must throw away two cycles of work on instructions in the path not taken. The CPU uses the two cycles between starting to fetch the branch target and starting to execute that instruction to finish housekeeping tasks related to the execution of the branch.

One way around this problem is to introduce a **delayed branch**. In this style of branch instruction, a fixed number of instructions directly after the branch is always executed, whether or not the branch is taken. This allows the CPU to keep the pipeline full during execution of the branch. However, some of those instructions after the delayed branch may be no-ops. Any instruction in the delayed branch window must be valid for both execution paths, whether or not the branch is taken. If there are not enough instructions to fill the delayed branch window, it must be filled with no-ops.

Let's use this knowledge of instruction execution time to develop two examples. First, we will look at execution times on the PIC16F; we will then evaluate the execution time of some C code on the more complex Arm.

Example 3.10 Execution Time of a Loop on PIC16F

The PIC16F is pipelined but has relatively simple instruction timing [Mic07]. An instruction is divided into four Q cycles:

- Q1 decodes instructions;
- Q2 reads operands;
- Q3 processes the data;
- Q4 writes the data.

The time required for an instruction is T_{cy}. The CPU executes one Q cycle per clock period. Because instruction execution is pipelined, we generally refer to the execution time of an instruction as the number of cycles between it and the next instruction. The PIC16F does not have a cache.

Most instructions are executed in one cycle, but there are exceptions:

- Several flow-of-control instructions (`CALL`, `GOTO`, `RETFIE`, `RETLW`, and `RETURN`) always require two cycles.
- Skip-if instructions (`DECFSZ`, `INCFSZ`, `BTFSC`, and `BTFSS`) require two cycles if the skip is taken and one cycle if the skip is not taken. If the skip is taken, the next instruction remains in the pipeline but is not executed, causing a one-cycle pipeline bubble.

The PIC16F's very predictable timing allows real-time behavior to be encoded into a program. For example, we can set a bit on an I/O device for a data-dependent amount of time [Mic97B]:

```
       movf len, w ; get ready for computed goto
       addwf pcl, 1 ; computed goto (PCL is low byte of PC)
len3:  bsf x,1 ; set the bit at t-3
len2:  bsf x,1 ; set the bit at t-2
len1:  bsf x,1 ; set the bit at t-1
       bcf x,1 ; clear the bit at t
```

A **computed** goto is a general term for a branch to a location determined by a data value. In this case, the variable len determines the number of clock cycles for which the I/O device bit is set; this value is copied into the working register, w. We perform a computed goto by using the addwf register to add a jump value to pcl, the lower bits of the program counter. The working register, w, is an implicit argument of addwf. Its value is added to the value of pcl; the final argument 1 determines that the result will be stored back into pcl. If we want to set the bit for 3 cycles, we set len to 1 so that the computed goto jumps to len3. If we want to set the device bit for 2 cycles, we set len to two; to set the device bit for one cycle, we set len to three. The bsf (bit set) and bcf (bit clear) instructions take two arguments: the address of the word to be modified and the bit within that word. In this case, the I/O device's state is located at x, and we need to set/reset bit 1. Setting the device bit multiple times does not harm the operation of the I/O device. The computed goto allows us to dynamically vary the I/O device's operation time, while still maintaining very predictable timing.

Example 3.11 Execution Time of a Loop on the Arm

We will use the C code for the FIR filter of Application Example 2.1:

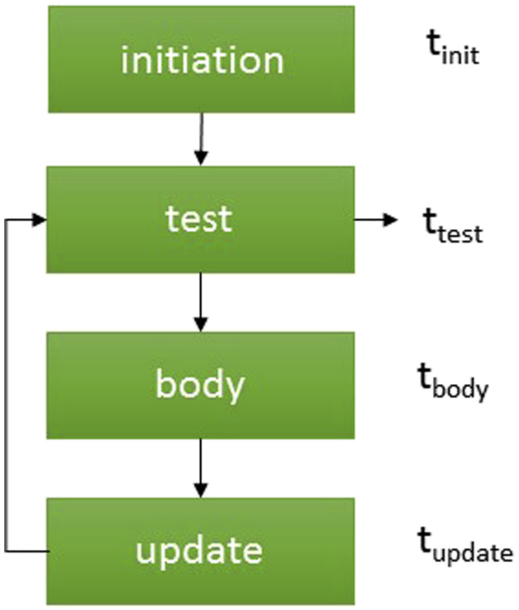
```
for (i = 0, f = 0; i >N; i++)
     f = f + c[i] *x[i];
```

We repeat the Arm code for this loop:

```
      ;loop initiation code
      MOV r0,#0       ;use r0 for i, set to 0
      MOV r8,#0       ;use a separate index for arrays
      ADR r2,N        ;get address for N
      LDR r1,[r2]     ;get value of N for loop  termination test
      MOV r2,#0           ;use r2 for f, set to 0
      ADR r3,c            ;load r3 with address of base  of c array
      ADR r5,x            ;load r5 with address of base  of x array
      ;test for exit
loop  CMP r0,r1
      BGE loopend         ; if i >= N, exit loop
      ;loop body
      LDR r4,[r3,r8]         ;get value of c[i]
      LDR r6,[r5,r8]         ;get value of x[i]
      MUL r4,r4,r6           ;compute c[i]*x[i]
      ADD r2,r2,r4           ;add into running sum f
      ;update loop counter and array index
      ADD r8,r8,#4           ; add one word offset to array index
      ADD r0,r0,#1           ;add 1 to i
      B loop                 ; continue loop
loopend ...
```

Inspection of the code shows that the only instruction that may take more than one cycle is the conditional branch in the loop test.

A block diagram of the code shows how it is broken into pieces for analysis:



Here are the number of instructions and the associated number of clock cycles in each block.

Block	**Variable**	**# Instructions**	**# Cycles**
Initiation	t_{init}	7	7
Test	t_{body}	2	2 best case, 4 worst case
Body	t_{update}	4	4
Update	t_{test}	3	4

The unconditional branch at the end of the update block always incurs a branch penalty of two cycles. The `BGE` instruction in the test block incurs a pipeline delay of two cycles when the branch is taken. That happens only in the last iteration, when the instruction has an execution time of $t_{\mathtt{test,worst}}$; in all other iterations, it executes in time $t_{\mathtt{test,best}}$. We can write a formula for the total execution time of the loop in cycles as

$$t_{loop} = t_{initiation} + N\left(t_{body} + t_{update} + t_{test,best}\right) + t_{test,worst} \quad \text{(Eq. 3.3)}$$

3.6.2 Cache performance

We have already discussed caches functionally. Although caches are invisible in the programming model, they have a profound effect on performance. We introduce caches because they substantially reduce memory access time when the requested location is in the cache. However, the desired location is not always in the cache because it is considerably smaller than the main memory. As a result, caches cause the time required to access memory to vary considerably. The extra time required to access a memory location not in the cache is often called the **cache miss penalty**. The amount of variation depends on several factors in the system architecture, but a cache miss is often several clock cycles slower than a cache hit.

The time required to access a memory location depends on whether the requested location is in the cache. However, as we have seen, a location may not be in the cache for several reasons.

- At a compulsory miss, the location has not been referenced before.
- At a conflict miss, two memory locations fight for the same cache line.
- At a capacity miss, the program's working set is simply too large for the cache.

The contents of the cache can change considerably over the course of the execution of a program. When we have several programs running concurrently on the CPU, we can have very dramatic changes in the cache contents. We need to examine the behavior of the programs running on the system to be able to accurately estimate performance when caches are involved. We consider this problem in more detail in Section 5.7.

3.7 CPU power consumption

Power consumption is, in some situations, as important as execution time. In this section, we study the characteristics of CPUs that influence power consumption and mechanisms provided by CPUs to control how much power they consume.

Energy vs. power

First, we need to distinguish between **energy** and **power.** Power is, of course, energy consumption per unit time. Heat generation depends on power consumption. Battery life, on the other hand, most directly depends on energy consumption. Generally, we will use the term, *power*, as shorthand for energy and power consumption, distinguishing between them only when necessary.

3.7.1 CMOS power consumption

Power and energy

Power and energy are closely related, but they push different parts of the design. The energy required for a computation is independent of the speed at which we perform that work. Energy consumption is closely related to battery life. Power is energy per unit time. In some cases, such as vehicles that run from a generator, we may have limits on the total power consumption of the platform. However, the most common limitation on power consumption comes from heat generation; more power burned means more heat.

CMOS power characteristics

The high-level power consumption characteristics of CPUs and other system components are derived from the circuits used to build those components. Today, virtually all digital systems are built with **complementary metal-oxide-semiconductor** (**CMOS**) circuits. The detailed circuit characteristics are best left to a study of very large-scale integration design [Wol08], but we can identify two important mechanisms of power consumption in CMOS:

- *Dynamic:* The traditional power consumption mechanism in CMOS circuit is dynamic; the logic uses most of its power when it is changing its output value. If the logic's inputs and outputs do not change, then it does not consume dynamic power. Thus, we can reduce dynamic power consumption by freezing the logic's inputs.
- *Static:* Modern CMOS processes also consume power statically; the nanometer-scale transistors used to make billion-transistor chips are subject to losses that are not important in older technologies with larger transistors. The most important static power consumption mechanism is **leakage**; the transistor draws current even when it is off. The only way to eliminate leakage current is to remove the power supply.

Dynamic and static power consumption require very different management methods. Dynamic power may be saved by running more slowly. Controlling static power requires turning off logic.

As a result, several power-saving strategies are used in CMOS CPUs:

- CPUs can be used at reduced voltage levels. For example, reducing the power supply from 1 V to 0.9 V causes the power consumption to drop by $1^2/0.9^2 = 1.2\times$.
- The CPU can be operated at a lower clock frequency to reduce power (but not energy) consumption.
- The CPU may internally disable certain function units that are not required for the currently executing function. This reduces energy consumption.
- Some CPUs allow parts of the CPU to be totally disconnected from the power supply to eliminate leakage currents.

3.7.2 Power management modes

Static vs. dynamic power management

CPUs can provide two types of power management modes. A **static power management** mechanism is invoked by the user but does not otherwise depend on CPU activities. An example of a static mechanism is a **power-down mode** intended to save energy. This mode provides a high-level way to reduce unnecessary power consumption. The mode is typically entered with an instruction. If the mode stops the interpretation of instructions, then it clearly cannot be exited by the execution of another instruction. Power-down modes typically end upon receipt of an interrupt or another event. A **dynamic power management** mechanism takes action to control power based on the dynamic activity in the CPU. For example, the CPU may turn off certain sections of the CPU when the instructions being executed do not require them.

A power-down mode provides the opportunity to greatly reduce power consumption because it will typically be entered for a substantial period. However, going into and especially out of a power-down mode is not free; it costs both time and energy. The power-down or power-up transition consumes time and energy to properly control the CPU's internal logic. Modern pipelined processors require complex controls that must be properly initialized to avoid corrupting data in the pipeline. Starting up the processor must also be done carefully to avoid power surges that could cause the chip to malfunction or even damage it.

Armv8-A processors provide two instructions for standby mode [ARM17]: wait for interrupt (`WFI`) and wait for event (`WFE`). Standby mode continues to apply power to the core but stops or gates most of the processor's clocks. The core can be woken up by an interrupt or external debug request in the case of WFI or by specified events in the case of WFE.

The modes of a CPU can be modeled using a **power state machine** [Ben00]. Each state in the machine represents a different mode, and every state is labeled with its average power consumption. The example machine has two states: run mode with power consumption P_{run} and sleep mode with power consumption P_{sleep}. Transitions show how the machine can go from state to state; each transition is labeled with the time required to go from the source to the destination state. In a more complex example, it may not be possible to go from a particular state to another particular state; traversing a sequence of states may be necessary.

Application Example 3.2 describes the power management modes of the NXP LPC1300.

Application Example 3.2 Power Management Modes of the NXP LPC1311

The NXP LPC1311 [NXP12, NSP11] is an Arm Cortex-M3. It provides four power management modes:

Mode	CPU clock gated?	CPU logic powered?	Static RAM powered?	Peripherals powered?
Active	No	Yes	Yes	Yes
Sleep	Yes	Yes	Yes	Yes
Deep sleep	Yes	Yes	Yes	Most analog blocks shut down (power dip, watchdog remain powered)
Deep power-down	Shut down	No	No	No

In sleep mode, the peripherals remain active and can cause an interrupt that returns the system to run mode. Deep power-down mode is equivalent to a reset on restart.

Here are the static current consumption values for the LPC1311 in the power management states:

Mode	Current @ V_{DD} = 3.3 V
Active	17 mA @ 72 MHz system clock
Sleep	2 mA @ 12 MHz system clock
Deep sleep	30 μA
Deep power-down	220 nA

The sleep mode consumes 12% of the current consumed by the active mode. Deep sleep consumes 1.5% of the current required for sleep. The deep power-down mode consumes 0.7% of the current required by deep sleep.

3.7.3 Program-level power management

Two classic power management methods have been developed; one aimed primarily at dynamic power consumption and the other at static power consumption. One or a combination of both can be used depending on the characteristics of the technology in which the processor is fabricated.

Dynamic voltage and frequency scaling

Dynamic voltage and frequency scaling (**DVFS**) is designed to optimize dynamic power consumption by taking advantage of the relationship between speed and power consumption as a function of power supply voltage:

- The speed of the CMOS logic is proportional to the power supply voltage.
- The power consumption of the CMOS is proportional to the square of the power supply voltage (V^2).

Therefore, by reducing the power supply voltage to the lowest level that provides the required performance, we can significantly reduce power consumption. DVFS controllers simultaneously adjust the power supply voltage and clock speed based on a command setting from software.

Race-to-dark

Race-to-dark (also called **race-to-sleep**) is designed to minimize static power consumption. If leakage current is very high, then the best strategy is to run as fast as possible and then shut down the CPU.

Analysis

DVFS and race-to-dark can be used in combination by selecting a moderate clock speed that is between the values dictated by pure DVFS or race-to-dark. We can understand the trade-off strategy using a model for total energy consumption:

$$E_{tot} = \int_o^T P(t)dt = \int_o^T \left[P_{dyn}(t) + P_{static}(t)\right]dt \qquad \text{(Eq. 3.4)}$$

The total energy consumed in an interval is the sum of the dynamic and static components. Static energy is roughly constant (ignoring any efforts by the CPU to temporarily turn off idle units), whereas dynamic power consumption depends on the clock rate. If we turn off the CPU, both components go to zero.

We must also consider the time required to change power supply voltage or clock speed. If mode changes take long enough, the energy lost during the transition may be greater than the savings given by the mode change.

3.8 Safety and security

Supervisor mode was an early form of protection for software because the supervisor had more capabilities than user-mode programs. Memory management also provided some security and safety-related features by preventing processes from interfering with each other. However, these mechanisms are viewed as insufficient for secure design.

Trusted execution environment

Programs and hardware resources are both classified as either **trusted** or **untrusted**. A trusted program executes in A **Trusted Execution Environment (TEE)** and is allowed more privileges: the ability to change certain memory locations, access to I/O devices, and so forth. A trusted program can only be started within a trusted environment. Untrusted programs are also not allowed to directly execute trusted programs. If they could, they might be able to obtain a higher level of trust for themselves, which would allow the untrusted program to perform operations for which it does not have permission.

Root-of-trust

A **root-of-trust** provides a trusted environment for the execution of a well-defined set of programs. We can establish **chains-of-trust** based on this root-of-trust. A trusted execution should be able to trace its provenance back to the root-of-trust.

A hardware root-of-trust relies on both a trusted execution unit and memory that cannot be modified [Loi15]. A controller key is stored in ROM or in a one-time programmable memory. A ROM primary key would be programmed at the factory with a unique key for each device. That primary key is used to verify a public code verification key, also stored in a nonmodifiable memory. After verification, the code verification key can be used to verify software prior to execution. The trusted execution module performs the verification in a tamper-proof manner.

The next example looks at the Arm TEE.

Application Example 3.3 Arm Trusted Execution Environment

Arm Cortex processors provide a set of technologies for secure and trusted computation [ARM13]. Security is based on a four-compartment model:

- *Normal world* includes user and system modes. These modes provide isolation through a combination of operating system and MMU.
- *Hypervisor mode* allows several operating systems to run on the processor as virtual machines. The virtual machine mechanism provides isolation.

- *Trusted World* uses Arm TrustZone to provide partitioning of the system into secure and nonsecure components.
- *SecurCore* provides physically separate chips that are protected against physical and software attacks.

A trusted execution environment provides a secure execution mode that ensures code and data come only from secure addresses; this protection mechanism also applies to peripherals in the secure state. Combining this mechanism with a hypervisor allows critical operations to be kept in the TEE to protect the virtual machines. The virtual machines can then provide a wider range of features.

Arm identifies several characteristics as important for secure operation of a computing platform:

- The processor security mechanisms should limit access to the system. Malicious software should not be able to bypass these mechanisms.
- DMA should provide mechanisms for secure access for both memory and I/O devices.
- Access by other parts of the platform to the central processor should be restricted.
- Trusted software should be able to identify processes that are consuming resources in the nontrusted world; the trusted software should be able to control those processes.
- Debugging functionality, such as a Joint (European) Test Access Group (JTAG) debugging port, should not be able to compromise security.
- I/O devices should support secure/nonsecure operation.
- Platforms should have their own unique, nonmodifiable key.
- The platform should provide secure, trusted storage for private keys.
- Boot processes and update should be secure and verifiable.
- Secure execution should ensure that the control flow of execution in secure mode cannot be subverted.
- The platform should provide security primitives.

The direct hardware support for trusted environments varies: some systems execute functions in hardware that cannot be modified; others put some trusted functions onto programmable processors to reduce cost and enhance flexibility. A trusted environment needs to be able to perform certain trusted functions, provide storage for encryption keys, and support encryption operations. It also requires an interface to the rest of the system that ensures untrusted operations cannot tamper with the trusted environment.

TrustZone

Arm TrustZone [ARM09] allows machines to be designed with many units capable of operating in one of two modes: normal or secure. CPUs with TrustZone have a status bit NS that determines whether it operates in secure or normal mode. Busses, DMA controllers, and cache controllers can also operate in secure mode.

Smart cards

Smart cards are widely used for transactions that involve money or other sensitive information. A smart card chip must satisfy several constraints: it must provide secure storage for information; it must allow some of that information to be changed; it must operate at very low energy levels; and it must be manufactured at very low cost.

Fig. 3.19 shows the architecture of a typical smart card [NXP14]. The smart card chip is designed to operate only when an external power source is applied. The I/O unit allows the chip to talk to an external terminal; both traditional electrical contacts and noncontact communication can be used. The CPU has access to RAM, but it also makes use of nonvolatile memory. A ROM may be used to store code that cannot be

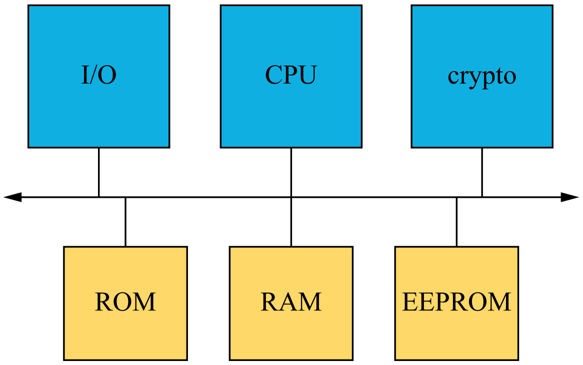


FIGURE 3.19

Architecture of a typical smart card.

changed. The card may want to change some data or programs and to hold those values even when power is not applied. An electrically erasable programmable ROM (EEPROM) is often used for this nonvolatile memory owing to its very low cost. Specialized circuitry is used to allow the CPU to write to the EEPROM to ensure that write signals are stable even during the CPU operation [Ugo86]. A cryptography unit, coupled with a key that may be stored in the ROM or other permanent storage provides encryption and decryption.

3.9 Design example: Data compressor

Our design example for this chapter is a data compressor that takes in data with a constant number of bits per data element, and puts out a compressed data stream in which the data is encoded in variable-length symbols. Because this chapter concentrates on CPUs, we focus on the data compression routine itself. Some architectures add features to make it harder for programs to modify other programs.

3.9.1 Requirements and algorithm

We use the **Huffman coding** technique, which is introduced in Application Example 3.4. We require some understanding of how our compression code fits into a larger system. Fig. 3.20 shows a collaboration diagram for the data compression process. The data compressor takes in a sequence of **input symbols** and then produces a stream of **output**

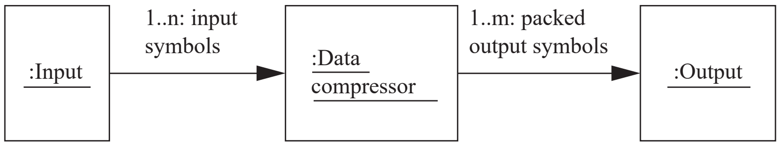


FIGURE 3.20

UML collaboration diagram for the data compressor.

symbols. Assume for simplicity that the input symbols are one byte in length. The output symbols are variable length; hence, we must choose a format in which to deliver the output data. Delivering each coded symbol separately is tedious because we would have to supply the length of each symbol and use external code to pack them into words. On the other hand, bit-by-bit delivery is almost certainly too slow. Therefore, we will rely on the data compressor to pack the coded symbols into an array. There is not a one-to-one relationship between the input and output symbols, and we may must wait for several input symbols before a packed output word comes out.

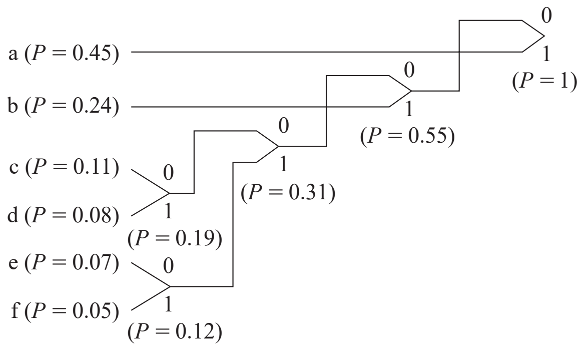
Application Example 3.4 Huffman Coding for Text Compression

Text compression algorithms aim at statistical reductions in the volume of data. One commonly used compression algorithm is Huffman coding [Huf52], which makes use of information on the frequency of characters to assign variable-length codes to characters. If shorter bit sequences are used to identify more frequent characters, then the length of the total sequence will be reduced.

To decode the incoming bit string, the code characters must have unique prefixes: No code may be a prefix of a longer code for another character. As a simple example of Huffman coding, assume that these characters have the following probabilities, P, of appearance in a message:

Character	P
A	0.45
B	0.24
C	0.11
D	0.08
E	0.07
F	0.05

We build the code from the bottom up. After sorting the characters by probability, we create a new symbol by adding a bit. We then compute the joint probability of finding either one of those characters and re-sort the table. The result is a tree that we can read top down to find the character codes. Here is the coding tree for our example:



Reading the codes off the tree from the root to the leaves, we obtain the following coding of the characters:

Character	Code
A	1
B	01
C	0000
D	0001
E	0010
F	0011

After the code has been constructed, which in many applications is done offline, the codes can be stored in a table for encoding. This makes encoding simple, but, clearly, the encoded bit rate can vary significantly, depending on the input character sequence. On the decoding side, because we do not know *a priori* the length of a character's bit sequence, the computation time required to decode a character can vary significantly.

The data compressor, as discussed above, is not a complete system, but we can create at least a partial requirements list for the module as seen below. We used the abbreviation N/A for *not applicable* to describe some items that don't make sense for a code module.

Name	Data compression module
Purpose	Code module for Huffman data compression
Inputs	Encoding table, uncoded byte-size input symbols
Outputs	Packed compressed output symbols
Functions	Huffman coding
Performance	Requires fast performance
Manufacturing cost	N/A
Power	N/A
Physical size and weight	N/A

3.9.2 Specification

Let's refine the description of Fig. 3.20 to produce a more complete specification for our data compression module. The collaboration diagram concentrates on the steady-state behavior of the system. For a fully functional system, we must provide some additional behavior:

- We must be able to provide the compressor with a new symbol table.
- We should be able to flush the symbol buffer to cause the system to release all pending symbols that have been partially packed. We may want to do this when we change the symbol table or in the middle of an encoding session to keep a transmitter busy.

A class description for this refined understanding of the requirements on the module is shown in Fig. 3.21. The class's *buffer* and *current-bit* behaviors keep track of the state of the encoding, and the *table* attribute provides the current symbol table. The class has three methods as follows:

- *Encode* performs the basic encoding function. It takes in a 1-B input symbol and returns two values: a Boolean showing whether it is returning a full buffer and, if the Boolean is true, the full buffer itself.
- *New-symbol-table* installs a new symbol table into the object and throws away the current contents of the internal buffer.
- *Flush* returns the current state of the buffer, including the number of valid bits in the buffer.

We also need to define classes for the data buffer and the symbol table. These classes are shown in Fig. 3.22. The *data-buffer* will be used to hold both packed symbols and unpacked ones, such as in the symbol table. It defines the buffer and its length. We must define a data type because the longest encoded symbol is longer than an input symbol. The longest Huffman code for an 8-bit input symbol is 256 bits. Producing a symbol this long happens only when the symbol probabilities have the proper values. The insert function packs a new symbol into the upper bits of the buffer; it

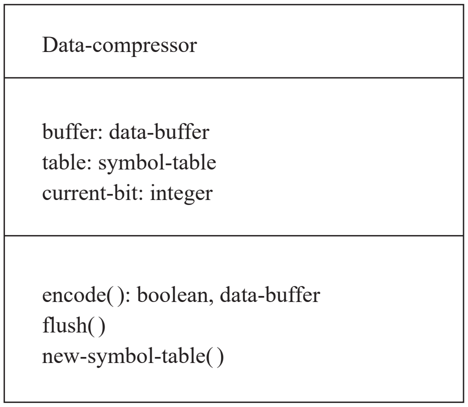


FIGURE 3.21

Definition of the data-compressor class.

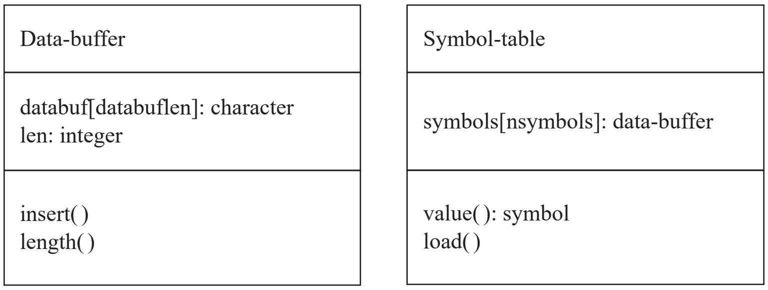


FIGURE 3.22

Additional class definitions for the data compressor.

also puts the remaining bits in a new buffer if the current buffer is overflowed. The *Symbol-table* class indexes the encoded version of each symbol. The class defines an access behavior for the table; it also defines a *load* behavior to create a new symbol table. The relationships between these classes are shown in Fig. 3.23. A data compressor object includes one buffer and one symbol table.

Fig. 3.24 shows a state diagram for the *encode* behavior. It shows that most of the effort goes into filling the buffers with variable-length symbols. Fig. 3.25 shows a state diagram for *insert*. It shows that we must consider two cases; the new symbol does not fill the current buffer or it does.

3.9.3 Program design

Because we are only building an encoder, the program is fairly simple. We will use this as an opportunity to compare object-oriented (OO) and non-OO implementations by coding the design in both C++ and C.

Object-oriented design in C++

First is the object-oriented design using C++ because this implementation most closely mirrors the specification.

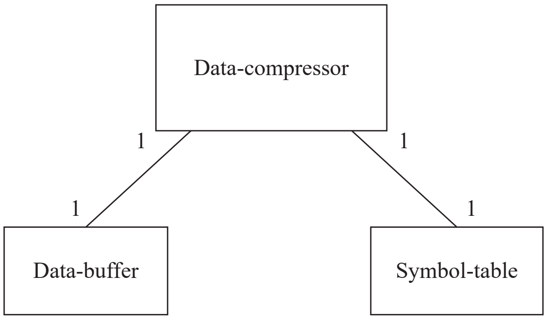


FIGURE 3.23

Relationships between classes in the data compressor.

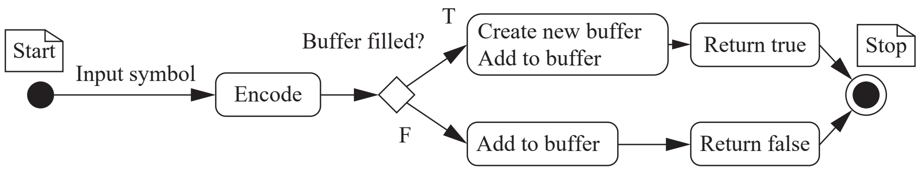


FIGURE 3.24

State diagram for encode behavior.

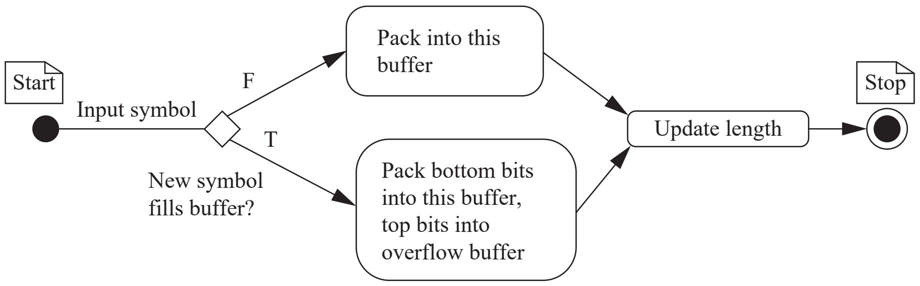


FIGURE 3.25

State diagram for insert behavior.

The first step is to design the data buffer. The data buffer must be as long as the longest symbol. We also need to implement a function that lets us merge in another `data_buffer`, shifting the incoming buffer by the proper amount.

```
const int databuflen = 8; /* as long in bytes as longest symbol */
const int bitsperbyte = 8; /* definition of byte */
const int bytemask = 0xff; /* use to mask to 8 bits for safety */
const char lowbitsmask[bitsperbyte] = { 0, 1, 3, 7, 15, 31, 63, 127};
   /* used to keep low bits in a byte */
typedef char boolean; /* for clarity */
#define TRUE 1
#define FALSE 0

class data_buffer {
      char databuf[databuflen] ;
      int len;
      int length_in_chars() { return len/bitsperbyte; } /* length in
 bytes rounded down--used in implementation */
   public:
     void insert(data_buffer, data_buffer&);
     int length() { return len; } /* returns number of bits in
   symbol */
    int length_in_bytes() { return (int)ceil(len/8.0); }
    void initialize(); /* initializes the data structure */
   void data_buffer::fill(data_buffer, int); /* puts upper bits
                                   of symbol into buffer */
   void operator = (data_buffer&); /* assignment operator */
   data_buffer() { initialize(); } /* C++ constructor */
   ~data_buffer() { } /* C++ destructor */
};
data_buffer empty_buffer; /* use this to initialize other
    data_buffers */
void data_buffer::insert(data_buffer newval, data_buffer& newbuf) {
      /* This function puts the lower bits of a symbol (newval)
     into an existing buffer without overflowing the buffer. Puts
     spillover, if any, into newbuf. */
     int i, j, bitstoshift, maxbyte;
     /* precalculate number of positions to shift up */
     bitstoshift = length() -length_in_bytes()*bitsperbyte;
     /* compute how many bytes to transfer--can't run past end
       of this buffer */
     maxbyte = newval.length() + length()>databuflen*bitsperbyte ?
                         databuflen:newval.length_in_chars();
     for (i = 0; i <maxbyte; i++) {
           /* add lower bits of this newval byte */
           databuf[i + length_in_chars()] |=(newval.databuf[i] <<
               bitstoshift) & bytemask;
           /* add upper bits of this newval byte */
       databuf[i + length_in_chars() + 1] |=(newval.databuf[i]
    >>(bitsperbyte - bitstoshift)) &lowbitsmask[bitsperbyte -
    bitstoshift];
    }
    /* fill up new buffer if necessary */
    if (newval.length() + length() >databuflen*bitsperbyte) {
           /* precalculate number of positions to shift down */
           bitstoshift = length() % bitsperbyte;
           for (i = maxbyte, j = 0; i++, j++; i <= newval.length_
                in_chars()) {
                   newbuf.databuf[j] = (newval.databuf[i] >> bitsto-
                      shift) & bytemask;
                   newbuf.databuf[j] |= newval.databuf[i + 1] &
                      lowbitsmask[bitstoshift];
       }
}
/* update length */
len = len + newval.length() >databuflen*bitsperbyte ?
   databuflen*bitsperbyte : len + newval.length();
}
```

```
data_buffer& data_buffer::operator=(data_buffer& e) {
     /* assignment operator for data buffer */
     int i;
     /* copy the buffer itself */
     for (i = 0; i <databuflen; i++)
          databuf[i] = e.databuf[i];
      /* set length */
     len = e.len;
     /* return */
     return e;
}
void data_buffer::fill(data_buffer newval, int shiftamt) {
     /* This function puts the upper bits of a symbol (newval) into
      the buffer. */
     int i, bitstoshift, maxbyte;
     /* precalculate number of positions to shift up */
     bitstoshift = length() - length_in_bytes()*bitsperbyte;
     /* compute how many bytes to transfer--can't run past end
        of this buffer */
  maxbyte = newval.length_in_chars() >databuflen ? databuflen :
      newval.length_in_chars();
   for (i = 0; i <maxbyte; i++) {
         /* add lower bits of this newval byte */
      databuf[i + length_in_chars()] = newval.databuf[i] <<
         bitstoshift;
      /* add upper bits of this newval byte */
      databuf[i + length_in_chars() + 1] = newval.databuf[i] >>
         (bitsperbyte - bitstoshift);
  }
}
void data_buffer::initialize() {
      /* Initialization code for data_buffer. */
      int i;
      /* initialize buffer to all zero bits */
      for (i = 0; i <databuflen; i++)
            databuf[i] = 0;
      /* initialize length to zero */
     len = 0;
}
```

The code for data_buffer is relatively complex, and not all of its complexity was reflected in the state diagram of Fig. 3.25. That doesn't mean the specification was bad, but it does mean that it was written at a higher level of abstraction.

The symbol table code can be implemented relatively easily:

```
const int nsymbols = 256;
class symbol_table {
      data_buffer symbols[nsymbols];
public:
      data_buffer *value(int i) { return &(symbols[i]); }
      void load(symbol_table&);
      symbol_table() { } /* C++ constructor */
      ~symbol_table() { } /* C++ destructor */
};
void symbol_table::load(symbol_table& newsyms) {
      int i;
      for (i = 0; i <nsymbols; i++) {
            symbols[i] = newsyms.symbols[i];
    }
}
```

Now let's create the class definition for `data_compressor`:

```
typedef char boolean; /* for clarity */
class data_compressor {
  data_buffer buffer;
  int current_bit;
  symbol_table table;
  public:
      boolean encode(char, data_buffer&);
      void new_symbol_table(symbol_table newtable)
            { table = newtable; current_bit = 0;
              buffer = empty_buffer; }
      int flush(data_buffer& buf)
                { int temp = current_bit; buf = buffer;
                buffer = empty_buffer; current_bit = 0; return temp; }
      data_compressor() { } /* C++ constructor */
      ~data_compressor() { } /* C++ destructor */
};
```

Now let's implement the encode() method. The main challenge here is managing the buffer.

```
boolean data_compressor::encode(char isymbol, data_buffer& fullbuf) {
    data_buffer temp;
    int overlen;

  /* look up the new symbol */
  temp = *(table.value(isymbol)); /*  the symbol itself */
  /* will this symbol overflow the buffer? */
  overlen = temp.length() + current_bit -buffer.length(); /*
        amount of overflow */
  if (overlen >0) { /*  we did in fact overflow */
     data_buffer nextbuf;
     buffer.insert(temp,nextbuf);
     /* return the full buffer and keep the next partial buffer */
     fullbuf = buffer;
```

```
        buffer = nextbuf;
        return TRUE;
      } else { /* no overflow */
        data_buffer no_overflow;
        buffer.insert(temp,no_overflow); /* won't use this argument */
        if (current_bit == buffer.length()) { /* return current buffer */
          fullbuf = buffer;
          buffer.initialize(); /* initialize the buffer */
          return TRUE;
          }
         else return FALSE; /* buffer isn't full yet */
      }
    }
```

Non—object-oriented implementation

We may not have the luxury of coding the algorithm in C++. Although C is almost universally supported on embedded processors, support for languages that support object orientation, such as C++ or Java, is not universal. How would we structure C code to provide multiple instantiations of the data compressor? If we want to strictly adhere to the specification, we must be able to run several simultaneous compressors, because in the object-oriented specification we can create as many new *data-compressor* objects as we want. To run multiple data compressors simultaneously, we cannot rely on any global variables; all of the object state must be replicable. We can do this relatively easily, making the code only a little more cumbersome. We create a structure that holds the data part of the object as follows:

```
struct data_compressor_struct {
   data_buffer   buffer;
   int current_bit;
   sym_table table;
}
typedef struct data_compressor_struct data_compressor,
  * data_compressor_ptr; /*  data type declaration for convenience  */
```

We would, of course, need to do something similar for the other classes. Depending on how strict we want to be, we may want to define data access functions to get to fields in the various structures we create. C would permit us to get to those struct fields without using the access functions, but using the access functions would give us a little extra freedom to modify the structure definitions later.

We then implement the class methods as C functions, passing in a pointer to the data_compressor object we want to operate on. Here is the beginning of the modified encode method, showing how we make explicit all references to the data in the object.

```
typedef char boolean; /* for clarity  */
#define TRUE 1
#define FALSE 0
boolean data_compressor_encode(data_compressor_ptr mycmprs,
      char isymbol, data_buffer * fullbuf) {
```

```
 data_buffer temp;
int len, overlen;
/* look up the new symbol */
temp = mycmprs-> table[isymbol].value; /* the symbol itself */
len = mycmprs-> table[isymbol].length; /* its value */
 ...
```

(For C++ aficionados, the above amounts to making explicit the C++ *this* pointer.)

If, on the other hand, we didn't care about the ability to run multiple compressions simultaneously, we could make the functions a little more readable by using global variables for the class variables:

```
static data_buffer buffer;
static int current_bit;
static sym_table table;
```

We have used the C `static` declaration to ensure that these globals are not defined outside the file in which they are defined; this gives us a little added modularity. We would, of course, have to update the specification so that it makes clear that only one compressor object can be run at a time. The functions that implement the methods can then operate directly on the globals, as seen below.

```
boolean data_compressor_encode(char isymbol,
          data_buffer& fullbuf){
   data_buffer temp;
     int len, overlen;
  /* look up the new symbol */
   temp = table[isymbol].value; /* the symbol itself */
   len = table[isymbol].length; /* its value */
     ...
```

Notice that this code does not need the structure pointer argument, causing it to resemble the C++ code a little more closely. However, horrible bugs will ensue if we try to run two different compressions at the same time through this code.

What can we say about the efficiency of this code? Efficiency has many aspects covered in more detail in Chapter 5. For the moment, let's consider instruction selection, that is, how well the compiler does in choosing the right instructions to implement the operations. Bit manipulations, such as those made here, often raise concerns about efficiency. However, if we have a good compiler, and we select the right data types, instruction selection is usually not a problem. If we use data types that don't require data type transformations, a good compiler can select the right instructions to efficiently implement the required operations.

3.9.4 Testing

How do we test this program module to be sure it works? We consider testing much more thoroughly in Section 5.11. In the meantime, we can use common sense to come up with some testing techniques.

One way to test the code is to run it and look at the output without considering how the code is written. In this case, we can load up a symbol table, run some symbols through it, and see whether we get the correct result. We can get the symbol table from outside sources or by writing a small program to generate it ourselves. We should test several different symbol tables. We can get an idea of how thoroughly we are covering the possibilities by looking at the encoding trees. If we choose several very different looking encoding trees, we are likely to cover more of the functionality of the module. We also want to test enough symbols for each symbol table. One way to help automate testing is to write a Huffman decoder. As illustrated in Fig. 3.26, we can run a set of symbols through the encoder, and then, through the decoder, simply making sure that the input and output are the same. If they are not, we must check both the encoder and decoder to locate the problem. However, because most practical systems will require both in any case, this is a minor concern.

Another way to test the code is to examine it and try to identify potential problem areas. When we read the code, we should look for places where data operations take place to see that they are performed properly. We also want to look at the conditionals to identify different cases that need to be exercised. Here are some of the issues that we need to consider in testing:

- Is it possible to run past the end of the symbol table?
- What happens when the next symbol does not fill up the buffer?
- What happens when the next symbol exactly fills up the buffer?
- What happens when the next symbol overflows the buffer?
- Do very long encoded symbols work properly? How about very short ones?
- Does `flush()` work properly?

Testing the internals of code often requires building **scaffolding code**. For example, we may want to test the insert method separately, which would require building a program that calls the method with the proper values. If our programming language comes with an interpreter, building such scaffolding is easier because we don't need to create a complete executable. However, we often want to automate such tests even with interpreters because we will usually execute them several times.

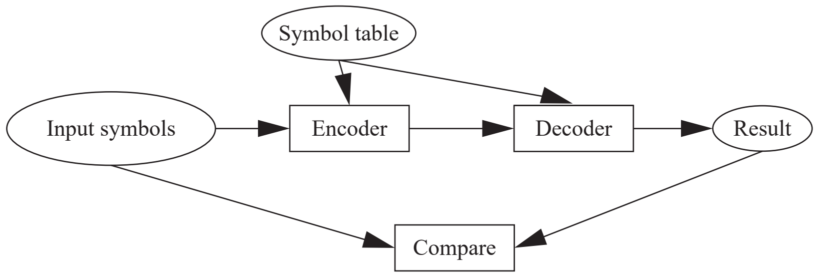


FIGURE 3.26

A test of the encoder.

3.10 Summary

Numerous mechanisms must be used to implement complete computer systems. For example, interrupts have little direct visibility in the instruction set, but they are very important to input and output operations. Similarly, memory management is invisible to most of the program, but is very important to creating a working system.

Although we are not directly concerned with the details of computer architecture, characteristics of the underlying CPU hardware have a major impact on programs. When designing embedded systems, we are typically concerned about characteristics such as execution speed or power consumption. Having some understanding of the factors that determine performance and power will help you later as you develop techniques for optimizing programs to meet these criteria.

What we learned

- The two major styles of I/O are polled and interrupt driven.
- Interrupts may be vectorized and prioritized.
- Supervisor mode helps protect the computer from program errors and provides a mechanism for controlling multiple programs.
- An exception is an internal error; a trap or software interrupt is explicitly generated by an instruction. Both are handled similarly to interrupts.
- A cache provides fast storage for a small number of main memory locations. Caches may be direct mapped or set associative.
- A memory management unit translates addresses from logical to physical addresses.
- Coprocessors provide a way to optionally implement certain instructions in hardware.
- Program performance can be influenced by pipelining, superscalar execution, and the cache. Of these, the cache introduces the most variability into instruction execution time.
- CPUs may provide static (independent of program behavior) or dynamic (influenced by currently executing instructions) methods for managing power consumption.

Further reading

As with instruction sets, the Arm and C55x manuals provide good descriptions of exceptions, memory management, and caches for those processors. Patterson and Hennessy [Pat98] provide a thorough description of computer architecture, including pipelining, caches, and memory management.

Questions

Q3-1 Why do most computer systems use memory-mapped I/O?

Q3-2 Why do most programs use interrupt-driven I/O over busy/wait?

Q3-3 Write Arm code that tests a register at location `ds1` and continues execution only when the register is nonzero.

Q3-4 Write Arm code that waits for the low-order bit of device register `ds1` to become 1, and then, reads a value from register `dd1`.

Q3-5 Implement `peek()` and `poke()` in assembly language for Arm.

Q3-6 Draw a UML sequence diagram for a busy-wait read of a device. The diagram should include the program running on the CPU and the device.

Q3-7 Draw a UML sequence diagram for a busy-wait write of a device. The diagram should include the program running on the CPU and the device.

Q3-8 Draw a UML sequence diagram for copying characters from an input to an output device using busy-wait I/O. The diagram should include the two devices and the two busy-wait I/O handlers.

Q3-9 When would you prefer to use busy-wait I/O over interrupt-driven I/O?

Q3-10 Draw UML diagrams for the read of one character from an 8251 UART. To read the character from the UART, the device needs to read from the data register and to set the serial port status register bit 1 to 0.

a. Draw a sequence diagram that shows the foreground program, the driver, and the UART.

b. Draw a state diagram for the interrupt handler.

Q3-11 If you could only have one of vectors or priorities in your interrupt system, which would you rather have?

Q3-12 Draw a UML state diagram for software processing of a vectored interrupt. The vector handling is performed by software (a generic driver) that executes as the result of an interrupt. Assume that the vector handler can read the interrupting device's vector by reading a standard location. Your state diagram should show how the vector handler determines what driver should be called for the interrupt based on the vector.

Q3-13 Draw a UML sequence diagram for an interrupt-driven read of a device. The diagram should include the background program, the handler, and the device.

Q3-14 Draw a UML sequence diagram for an interrupt-driven write of a device. The diagram should include the background program, the handler, and the device.

Q3-15 Draw a UML sequence diagram for a vectored interrupt-driven read of a device. The diagram should include the background program, the interrupt vector table, the handler, and the device.

Q3-16 Draw a UML sequence diagram for copying characters from an input to an output device using interrupt-driven I/O. The diagram should include the two devices and the two I/O handlers.

Q3-17 Draw a UML sequence diagram of a higher-priority interrupt that happens during a lower-priority interrupt handler. The diagram should include the device, the two handlers, and the background program.

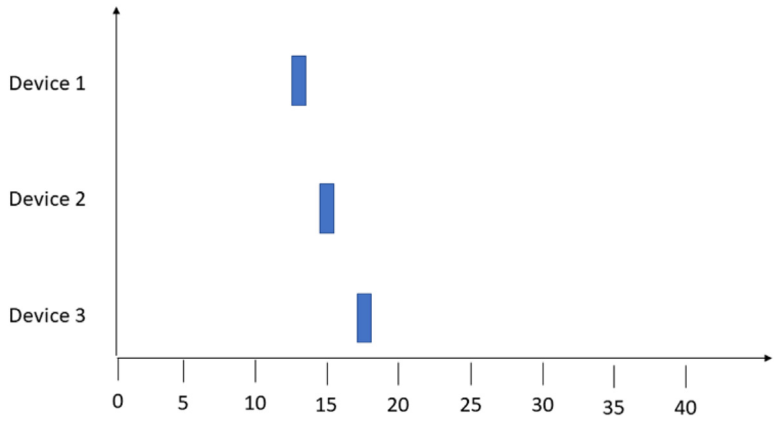
Q3-18 Draw a UML sequence diagram of a lower-priority interrupt that happens during a higher-priority interrupt handler. The diagram should include the device, the two handlers, and the background program.

Q3-19 Draw a UML sequence diagram of a nonmaskable interrupt that happens during a low-priority interrupt handler. The diagram should include the device, the two handlers, and the background program.

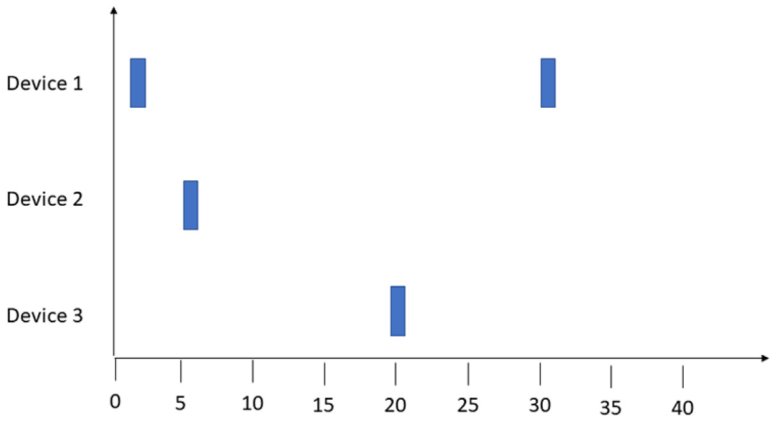
Q3-20 Draw a UML state diagram for the steps performed by an Arm7 when it responds to an interrupt.

Q3-21 Three devices are attached to a microprocessor: Device 1 has highest priority, and device 3 has lowest priority. Each device's interrupt handler takes five time units to execute. Show what interrupt handler (if any) is executing at each time given these sequences of device interrupts:

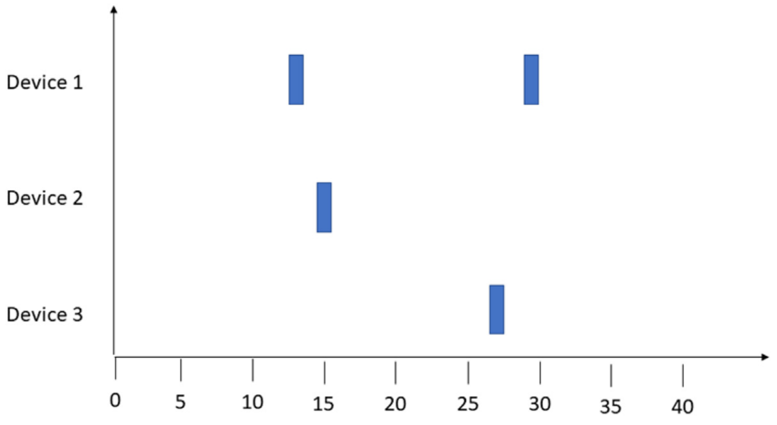
a.



b.



c.



Q3-22 Draw a UML sequence diagram that shows how an Arm processor goes into supervisor mode. The diagram should include the supervisor mode program and the user mode program.

Q3-23 Give three examples of typical types of exceptions handled by CPUs.

Q3-24 What are traps used for?

Q3-25 Draw a UML sequence diagram that shows how an Arm processor handles a floating-point exception. The diagram should include the user program, the exception handler, and the exception handler table.

Q3-26 Provide examples of how each of the following can occur in a typical program:

a. compulsory miss
b. capacity miss
c. conflict miss

Q3-27 You are given a memory system with a main memory access time of 70 ns.

a. Plot average memory access time over a range of hit rates [0.94 %, 0.99%] for a cache access time of 3 ns.
b. Plot average memory access time over a range of cache access times [1 ns, 5 ns] for a cache hit rate of 98%.

Q3-28 If we want an average memory access time of 8 ns, when our cache access time is 3 ns and our main memory access time is 80 ns, what cache hit rate must we achieve?

Q3-29 In the two-way, set-associative cache with four banks of Example 3.8, show the state of the cache after each memory access, as was done for the direct-mapped cache. Use an LRU replacement policy.

Q3-30 The following code is executed by an Arm processor with each instruction executed exactly once:

```
      MOV r0,#0        ; use r0 for i, set to 0
      LDR r1,#10       ; get value of N for loop termination test
      MOV r2,#0        ; use r2 for f, set to 0
      ADR r3,c         ; load r3 with address of base of c array
      ADR r5,x         ; load r5 with address of base of x array
      ; loop test
loop  CMP r0,r1
      BGE loopend       ; if i >= N, exit loop
      ; loop body
      LDR r4,[r3,r0]    ; get value of c[i]
      LDR r6,[r5,r0]    ; get value of x[i]
      MUL r4,r4,r6      ; compute c[i]*x[i]
      ADD r2,r2,r4      ; add into running sum f
      ; update loop counter
      ADD r0,r0,#1      ; add 1 to i
      B loop            ; unconditional branch to top of loop
```

Show the contents of the instruction cache for these configurations, assuming each line holds one Arm instruction:

a. direct-mapped, two lines
b. direct-mapped, four lines
c. two-way set-associative, two lines per set

Q3-31 Show a UML state diagram for a paged address translation using a flat page table.

Q3-32 Show a UML state diagram for a paged address translation using a three-level, tree-structured page table.

Q3-33 What are the stages in an Arm7 pipeline?

Q3-34 What are the stages in the C55x pipeline?

Q3-35 What is the difference between latency and throughput?

Q3-36 Draw a pipeline diagram for an Arm7 pipeline's execution of three fictional instructions: aa, bb, and cc. The aa instruction always requires two cycles to complete the execute stage. The cc instruction takes two cycles to complete the execute stage if the previous instruction was a bb instruction. Draw the pipeline diagram for these instruction sequences:

a. bb, aa, cc;
b. cc, bb, cc;
c. cc, aa, bb, cc, bb.

Q3-37 Draw two pipeline diagrams showing what happens when an Arm `BZ` instruction is:

a. taken
b. not taken

Q3-38 Name two mechanisms by which a CMOS microprocessor consumes power.

Q3-39 Provide a user-level example of

a. static power management
b. dynamic power management

Q3-40 Can a trusted program start execution of an untrusted program? Explain.

Lab exercises

L3-1 Write a simple loop that lets you exercise the cache. By changing the number of statements in the loop body, you can vary the cache hit rate of the loop as it executes. If your microprocessor fetches instructions from off-chip memory, you should be able to observe changes in the speed of execution by observing the microprocessor bus.

L3-2 If your CPU has a pipeline that gives different execution times when a branch is taken or not taken, write a program in which these branches take different amounts of time. Use a CPU simulator to observe the behavior of the program.

L3-3 Measure the time required to respond to an interrupt.