CHAPTER

Embedded Multiprocessors

10

CHAPTER POINTS

- Why we need networks and multiprocessors in embedded computing systems.
- Embedded multiprocessor architectures.
- System design for parallel and distributed computing.
- Design example: video accelerator.

10.1 Introduction

Many embedded computing systems require more than one CPU. To build such systems, we must use networks to connect processors, memory, and devices. We must then program the system to take advantage of the parallelism inherent in multiprocessing and to account for the communication delays incurred by networks. This chapter introduces some basic concepts in parallel and distributed embedded computing systems. Section 10.2 outlines the case for using multiprocessors in embedded systems. Section 10.3 examines the categories of multiprocessors. Section 10.4 considers shared memory multiprocessors and multiprocessor systems-on-chip (MPSoCs). Section 10.5 walks through the design of a video accelerator as an example of a specialized processing element (PE).

10.2 Why multiprocessors?

Definitions

Programming a single CPU is hard enough. Why make life more difficult by adding more processors? A **multiprocessor** is, in general, any computer system with two or more processors coupled together. Multiprocessors used for scientific or business applications tend to have regular architectures that include several identical processors that can access a uniform memory space. We use the term processing element (**PE**) to mean any unit responsible for computation, whether or not it is

Computers as Components. https://doi.org/10.1016/B978-0-323-85128-2.00010-4

programmable. We use the term **network** (or **interconnection network**) to describe the interconnections between the processing elements.

Why so many?

Embedded system designers must take a more general view of the nature of multiprocessors. As we will see, embedded computing systems are built atop a complete spectrum of multiprocessor architectures. Why is there no single multiprocessor architecture for all types of embedded computing applications? And why do we need embedded processors at all? The reasons for multiprocessors are the same reasons that drive embedded system design: real-time performance, power consumption, and cost.

Cost/performance

The first reason for using an embedded multiprocessor is that it can offer significantly better cost/performance—that is, functionality per dollar spent on the system—than would be obtained by spending the same amount of money on a uniprocessor system. The reason for this is that the processing element purchase price is a *nonlinear* function of performance [Wol08]. The cost of a microprocessor increases greatly as clock speed increases. We would expect this trend as a normal consequence of very large-scale integration (VLSI) fabrication and market economics. Clock speeds are normally distributed by normal variations in VLSI processes; because the fastest chips are rare, they command a high price in the marketplace.

Because the fastest processors are very costly, splitting the application so that it can be performed on several smaller processors is much cheaper. Even with the added costs of assembling components, the total system is less expensive. Of course, splitting the application across multiple processors entails higher engineering costs and lead times, which must be factored into the project.

Real-time performance

In addition to reducing costs, using multiple processors can also help with real-time performance. We can often meet deadlines and be responsive to interactions much more easily when we put those time-critical processes on separate processors. Given that scheduling multiple processes on a single CPU incurs overhead in most realistic scheduling models, as discussed in Chapter 6, putting the time-critical processes on processing elements that have little or no time sharing reduces scheduling overhead. Because we pay for that overhead at a nonlinear rate for the processor, as illustrated in Fig. 10.1, the savings by segregating time-critical processes can be large; it may take an extremely large and powerful CPU to provide the same responsiveness that can be had from a distributed system.

Cyber-physical considerations

We may also need to use multiple processors to put some of the processing power near the physical systems being controlled. Cars, for example, put control elements near the engine, brakes, and other major components. Analog and mechanical needs often dictate that critical control functions be performed very close to the sensors and actuators.

Power

Many of the technology trends that encourage us to use multiprocessors for performance also lead us to multiprocessing for low-power embedded computing. Several processors running at slower clock rates consume less power than a single large processor; performance scales linearly with power supply voltage, but power scales with V^2.

Austin et al. [Aus04] showed that general-purpose computing platforms aren't keeping up with the strict energy budgets of battery-powered embedded computing.

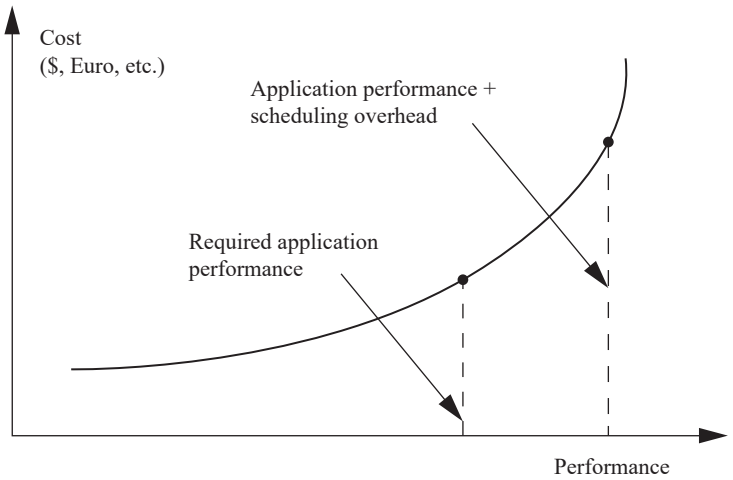


FIGURE 10.1

Scheduling overhead is paid for at a nonlinear rate.

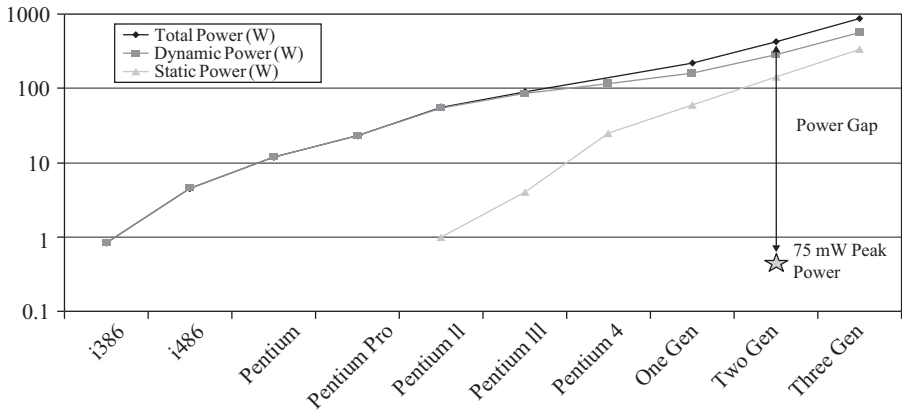


FIGURE 10.2

Power consumption trends for desktop processors [Aus04] 2004 IEEE Computer Society.

Fig. 10.2 compares the performance of the power requirements of desktop processors with available battery power. Batteries can provide only about 75 mW of power. Desktop processors require close to 1000 times the amount of power to run. This huge gap cannot be solved by tweaking processor architectures or software. Multiprocessors provide a way to break through this power barrier and build substantially more efficient embedded computing platforms.

10.3 Categories of multiprocessors

Multiprocessors in general-purpose computing have a long and rich history. Embedded multiprocessors have been widely deployed for several decades. The range of embedded multiprocessor implementations is also impressively broad. Multiprocessing has been used both for relatively low-performance systems and to achieve very high levels of real-time performance at very low energy levels.

Shared memory vs. message passing

There are two major types of multiprocessor architectures, as illustrated in Fig. 10.3:

- **Shared memory** systems have a pool of processors (P1, P2, *etc*.) that can read and write a collection of memories (M1, M2, *etc*.).
- **Message passing** systems have a pool of processors that can send messages to each other. Each processor has its own local memory.

Both shared memory and message-passing machines use an interconnection network; the details of these may vary considerably. These two types are functionally equivalent. We can turn a program written for one style of machine into an equivalent program for the other style. We may choose to build one or the other based on a variety of considerations, including performance, cost, and so on.

System-on-chip vs. distributed

The shared memory *vs*. message-passing distinction doesn't tell us everything we would like to know about a multiprocessor. The physical organization of the processing elements and memory play a large role in determining the characteristics of the system. We have already seen in Chapter 4 single-chip microcontrollers that include the processor, memory, and I/O devices. A multiprocessor system-on-chip (MPSoC) [Wol08B] is a system-on-chip with multiple processing elements. In contrast, we use the term **distributed system** for a multiprocessor in which the processing elements are physically separated. In general, the networks used for MPSoCs will be fast

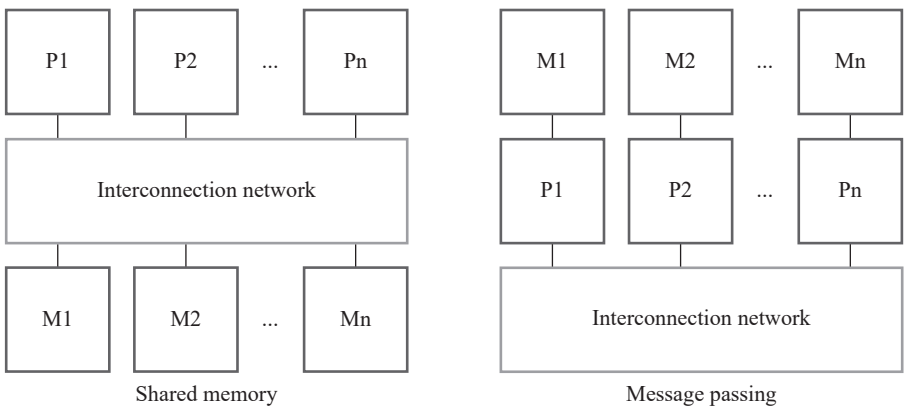


FIGURE 10.3

The two major multiprocessor architectures.

and will provide lower-latency communication between the PEs. The networks for distributed systems give higher latencies than are possible on a single chip, but many embedded systems require us to use multiple chips that may be physically very far apart. The differences in latencies between MPSoCs and distributed systems influence the programming techniques used for each.

MPSoCs

Shared memory systems are very common in single-chip embedded multiprocessors. Shared memory multiprocessors show up in low-cost systems Section 10.6. They also appear in higher-cost, high-performance systems. Shared memory systems offer relatively fast access to shared memory.

The next example describes a heterogeneous multicore embedded processor for smartphones, The Apple A15.

Example 10.1: Apple A15

Apple A15 [Fru21] is an SoC for smartphones. The CPU cluster includes two high-performance cores and four high-efficiency cores. Two variants of the chip offer somewhat different GPUs: four cores in one variant and five cores in the other. Accelerators include video encoding and decoding, an image signal processor, a display engine, and a neural engine.

10.4 MPSoCs and shared memory multiprocessors

Shared memory processors are well suited to applications that require a large amount of data to be processed. Signal processing systems stream data and can be well suited for shared memory processing. Most MPSoCs are shared memory systems.

Shared memory allows processors to communicate with varying patterns. If the pattern of communication is very fixed and if the processing of different steps is performed in different units, then a networked multiprocessor may be most appropriate. If the communication patterns between steps can vary, then shared memory provides flexibility. If one processing element is used for several different steps, then shared memory also allows for the required flexibility in communication.

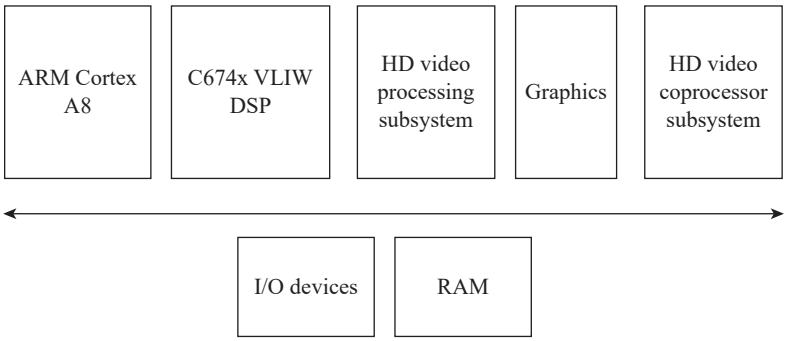
10.4.1 Heterogeneous shared memory multiprocessors

Many high-performance embedded platforms are heterogeneous multiprocessors. Different processing elements perform different functions. PEs may be programmable processors with different instruction sets or specialized accelerators that provide little or no programmability. In both cases, the motivation for using different types of PEs is efficiency. Processors with different instruction sets can perform different tasks faster and use less energy. Accelerators provide even faster and lower-power operations for a narrow range of functions.

The next example studies the TI TMS320DM816x DaVinci digital media processor.

Example 10.2: TI TMS320DM816x DaVinci

DaVinci 816x [Tex11, Tex11B] was designed for high-performance video applications. It includes a CPU, a digital signal processor (DSP), and several specialized units:



The 816x has two main programmable processors. The Arm Cortex A8 includes Neon multimedia instructions. It is an in-order dual-issue machine. The C674x is a very long instruction word DSP. It has six arithmetic logic units and 64 general-purpose registers.

The HD video coprocessor subsystem (HDVICP2) provides image and video acceleration. It natively supports several standards, such as H.264 (used in BluRay), MPEG-4, MPEG-2, and JPEG. It includes specialized hardware for major image and video operations, including transform and quantization, motion estimation, and entropy coding. It also has its own DMA engine. It can operate at resolutions up to 1080P/I at 60 fps. The HD video processing subsystem provides additional video processing capabilities. It can process up to three high-definition and one standard-definition video stream simultaneously. It can perform operations, such as scan rate conversion, chromakey, and video security. The graphics unit is designed for 3D graphics operations that can process up to 30 M triangles/s.

10.4.2 Accelerators

One important category of PEs for embedded multiprocessors is the **accelerator.** Accelerators can provide large performance increases for applications with **computational kernels** that spend a great deal of time on a small section of code. Accelerators can also provide critical speedups for low-latency I/O functions.

The design of accelerated systems is one example of **hardware/software co-design**—the simultaneous design of hardware and software to meet system objectives. Thus far, we have taken the computing platform as a given; by adding accelerators, we can customize the embedded platform to better meet our application's demands.

As illustrated in Fig. 10.4, a CPU accelerator is attached to the CPU bus. The CPU is often called the **host**. The CPU talks to the accelerator through the data and control registers in the accelerator. These registers allow the CPU to monitor the accelerator's operation and give the accelerator commands.

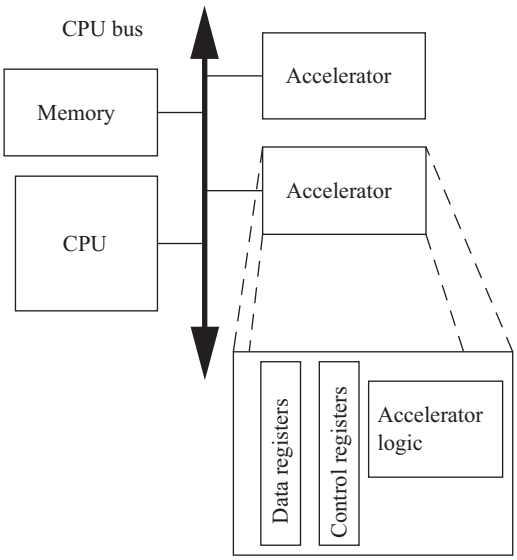


FIGURE 10.4

CPU accelerators in a system.

The CPU and accelerator may also communicate via shared memory. If the accelerator needs to operate on a large volume of data, it is usually more efficient to leave the data in memory and have the accelerator read and write memory directly, rather than to have the CPU shuttle data from memory to accelerator registers and back. The CPU and accelerator use synchronization mechanisms to ensure that they do not destroy each other's data.

An accelerator is not a co-processor. A co-processor is connected to the internals of the CPU and processes instructions. An accelerator interacts with the CPU through the programming model interface; it does not execute instructions. Its interface is functionally equivalent to an I/O device, although it usually does not perform input or output.

The first task in designing an accelerator is to determine that our system actually needs one. We must make sure that the function we want to accelerate will run more quickly on our accelerator than it will execute as software on a CPU. If our system CPU is a small microcontroller, the race may easily be won, but competing against a high-performance CPU is a challenge. We must also make sure that the accelerated function will speed up the system. If some other operation is in fact the bottleneck, or if moving data into and out of the accelerator is too slow, then adding the accelerator may not be a net gain.

Once we have analyzed the system, we need to design the accelerator itself. To identify our need for an accelerator, we must have a good understanding of the algorithm to be accelerated, which is often in the form of a high-level language program.

We must translate the algorithm description into a hardware design—a considerable task. We must also design the interface between the accelerator core and the CPU bus. The interface includes more than bus handshaking logic. For example, we must determine how the application software on the CPU will communicate with the accelerator and provide the required registers; we may have to implement shared memory synchronization operations; and we may have to add address generation logic to read and write large amounts of data from the system memory.

Finally, we will have to design the CPU-side interface for the accelerator. The application software will have to talk to the accelerator, providing it data, and telling it what to do. We have to somehow synchronize the operation of the accelerator with the rest of the application so that the accelerator knows when it has the required data, and the CPU knows when it has received the desired results.

Field-programmable gate arrays (FPGAs) provide a useful platform for custom accelerators. An FPGA has a **fabric** with both programmable logic gates and programmable interconnects that can be configured to implement a specific function. Most FPGAs also provide on-board memory that can be configured with different ports for custom memory systems. Some FPGAs provide on-board CPUs to run software that can talk to the FPGA fabric. Small CPUs can also be implemented directly in the FPGA fabric; the instruction sets of these processors can be customized for the required function.

The next example describes an MPSoC with both an on-board multiprocessor and an FPGA fabric.

Example 10.3: Xilinx Zynq UltraScale+ MPSoCs

The Xilinx Zynq UltraScale+ family (http://www.xilinx.com) combines a multiprocessor, FPGA fabric, memory, and other system components. The chips include both a quad-core Arm Cortex-A53 and a dual-core Arm Cortex-R5, as well as a Mali graphics unit. A variety of dynamic and static memory interfaces is provided. I/O devices include PCIe, SATA, USB, CAN, SPI, and GPIO. Several security units are provided. The chips also include an array of combinational logic blocks and block RAM.

10.4.3 Accelerator performance analysis

In this section, we are most interested in **speedup:** How much faster is the system with the accelerator than the system without it? We may, of course, be concerned with other metrics, such as power consumption and manufacturing cost. However, if the accelerator doesn't provide an attractive speedup, questions of cost and power will be moot.

Performance analysis of an accelerated system is a more complex task than what we have done thus far. In Chapter 6, we found that performance analysis of a CPU with multiple processes was more complex than the analysis of a single program.

When we have multiple processing elements, performance analysis becomes even more difficult.

The speedup factor depends in part on whether the system is **single-threaded** or **multithreaded**, that is, whether the CPU sits idle while the accelerator runs in the single-threaded case, or the CPU can do useful work in parallel with the accelerator in the multithreaded case. Another equivalent description is **blocking** vs. **nonblocking**. Does the CPU's scheduler block other operations and wait for the accelerator call to complete, or does the CPU allow some other process to run in parallel with the accelerator? The possibilities are shown in Fig. 10.5. Data dependencies allow *P2* and *P3* to run independently on the CPU, but *P2* relies on the results of the *A1* process implemented by the accelerator. However, in the single-threaded case, the CPU blocks to wait for the accelerator to return the results of its computation. As a result, it doesn't matter whether *P2* or *P3* runs next on the CPU. In the multithreaded case, the CPU continues to do useful work while the accelerator runs, so the CPU can start *P3* just after starting the accelerator and finishing the task earlier.

The first task is to analyze the performance of the accelerator. As illustrated in Fig. 10.6, the execution time for the accelerator depends on more than just the time required to execute the accelerator's function. It also depends on the time required to get the data into the accelerator and back out of it.

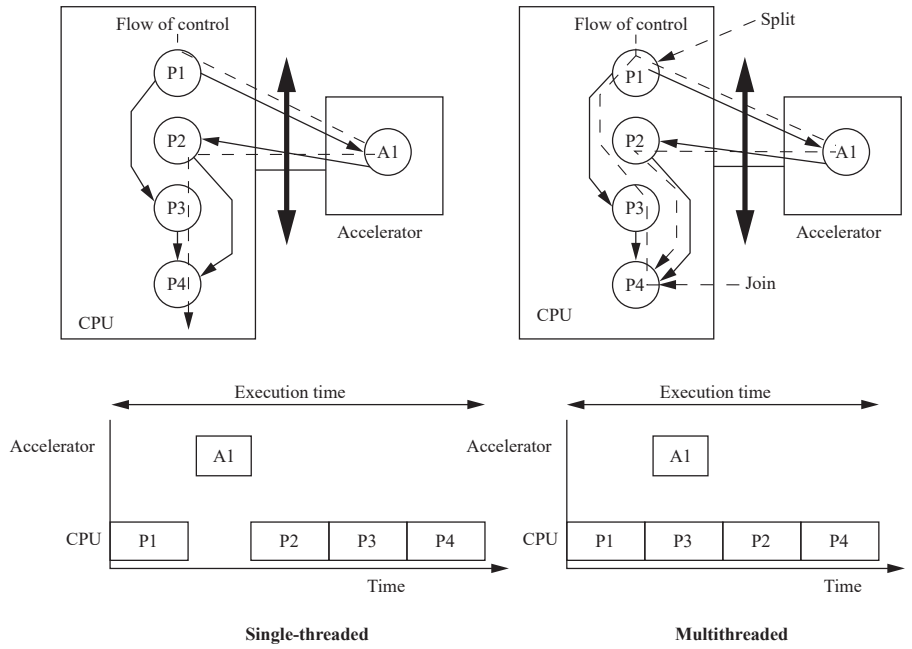


FIGURE 10.5

Single-threaded vs. multithreaded control of an accelerator.

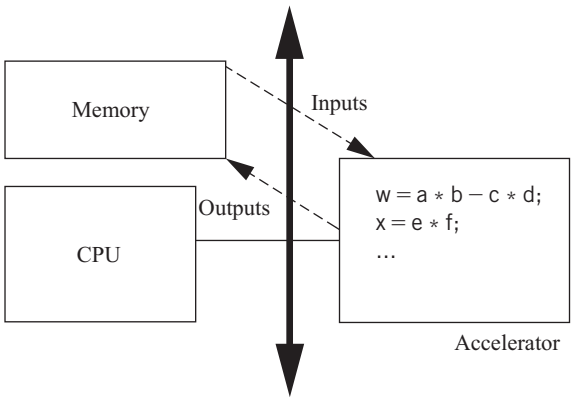


FIGURE 10.6

Components of execution time for an accelerator.

Because the CPU's registers are probably not addressable by the accelerator, the data probably reside in the main memory.

Accelerator execution time

A simple accelerator reads all its input data, performs the required computation, and then, writes all its results. In this case, the total execution time may be written as

$$t_{accel} = t_{in} + t_x + t_{out} \quad \text{(Eq. 10.1)}$$

where t_x is the execution time of the accelerator assuming all data are available, and t_{in} and t_{out} are the times required for reading and writing the required variables, respectively. The values for t_{in} and t_{out} must reflect the time required for bus transactions, including two factors:

- the time required to flush any register or cache values to the main memory, if those values are needed in the main memory to communicate with the accelerator; and
- the time required for the transfer of control between the CPU and accelerator.

Transferring data into and out of the accelerator may require the accelerator to become a bus controller. Because the CPU may delay bus controllership requests, some worst-case values for bus controllership acquisition must be determined based on the CPU characteristics.

A more sophisticated accelerator could try to overlap input and output with computation. For example, it could read a few variables and start computing on those values, while reading other values in parallel. In this case, the t_{in} and t_{out} terms would represent the nonoverlapped read/write times rather than the complete input and output times. One important example of overlapped I/O and computation is streaming data applications, such as digital filtering. As illustrated in Fig. 10.7, an accelerator may take in one or more streams of data and output a stream. A typical stream is sufficiently large that it cannot be fetched at once; a series of fetches and stores is required. Latency requirements generally require that outputs be produced on the

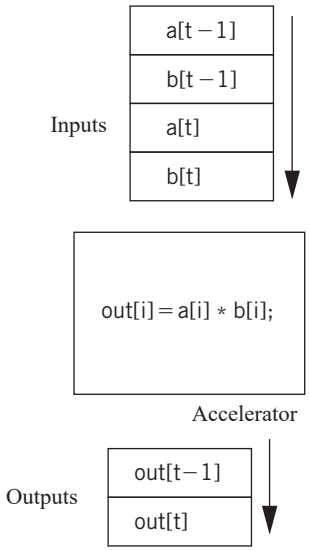


FIGURE 10.7

Streaming data into and out of an accelerator.

fly rather than storing all the data and then computing; furthermore, it may be impractical to store long streams at all. In this case, the t_{in} and t_{out} terms are determined by the amount of data read before starting computation and the length of time between the last computation and the last data output.

We are most interested in the speedup obtained by replacing the software implementation with the accelerator. The total speedup S for a kernel can be written as [Hen94]:

$$S = n(t_{CPU} - t_{accel}) = n\,[t_{CPU} - (t_{in} + t_x + t_{out})] \qquad \text{(Eq. 10.2)}$$

where t_{CPU} is the execution time of the equivalent function in the software on the CPU, and n is the number of times the function will be executed. We can use the techniques described in Chapter 5 to determine the value of t_{CPU}. Clearly, the more times the function is evaluated, the more valuable the speedup provided by the accelerator becomes.

System speedup

Ultimately, we don't care as much about the accelerator's speedup as the speedup for the complete system—how much faster is the entire application's execution? In a single-threaded system, the evaluation of the accelerator's speedup to the total system speedup is simple: the system execution time is reduced by S. The reason is illustrated in Fig. 10.8; the single thread of control gives us a single path whose length we can measure to determine the new execution speed.

Evaluating system speedup in a multithreaded environment requires more subtlety. As shown in Fig. 10.9, there is now more than one execution path. The total system

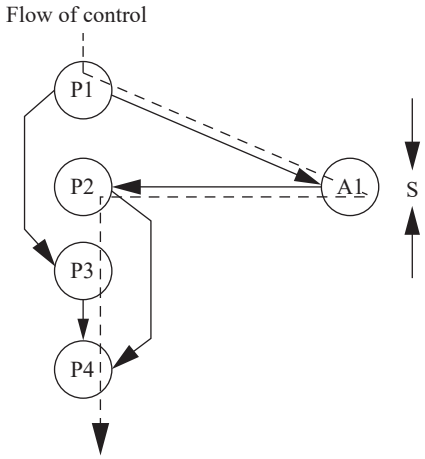


FIGURE 10.8

Evaluating system speedup in a single-threaded implementation.

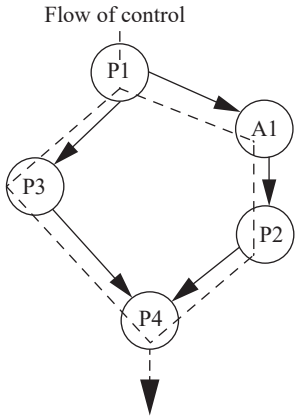


FIGURE 10.9

Evaluating system speedup in a multithreaded implementation.

execution time depends on the **longest path**, from the beginning of execution to the end of execution. In this case, the system execution time depends on the relative speeds of *P3* and *P2* plus *A1*. If *P2* and *A1* together take the most time, *P3* will not play a role in determining the system execution time. If *P3* takes longer, then *P2* and *A1* will not be factors. To determine the system execution time, we must label each node in the graph with its execution time. In simple cases, we can enumerate the paths, measure the length of each, and select the longest one as the system execution time. Efficient graph algorithms can also be used to compute the longest path.

This analysis shows the importance of selecting the proper functions to be moved to the accelerator. Clearly, if the function selected for speedup isn't a big portion of system execution time, taking the number of times it is executed into account, you won't see much system speedup. We also learned from Equation 10.2 that, if too much overhead is incurred in getting data into and out of the accelerator, we won't see much speedup.

10.4.4 Scheduling and allocation

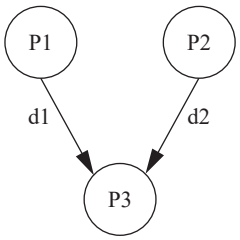
When designing a distributed embedded system, we must deal with the **scheduling** and **allocation** design problems described below.

- We must *schedule* operations in time, including communication on the network and computations on the processing elements. The scheduling of operations on the PEs and the communications between the PEs are linked. If one PE finishes its computations too late, it may interfere with another communication on the network, as it tries to send its result to the PE that needs it. This is bad for both the PE that needs the results and the other PEs whose communication is interfered with.
- We must *allocate* computations to the processing elements. The allocation of computations to the PEs determines what communications are required; if a value computed on one PE is needed on another PE, it must be transmitted over the network.

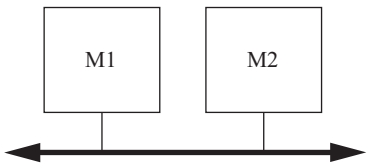
Example 10.4 illustrates scheduling and allocation in accelerated embedded systems.

Example 10.4: Scheduling and Allocating Processes on a Distributed Embedded System

We can specify the system as a task graph. However, different processes may result in different PEs. Here is a task graph:



We have labeled the data transmissions on each arc so that we can refer to them later. We want to execute the task on this platform:



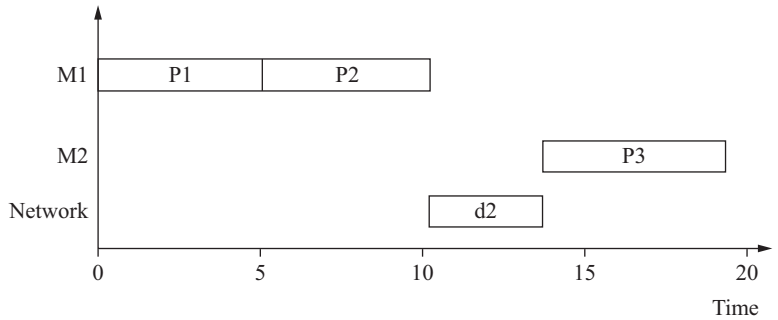
The platform has two PEs and a single bus connecting both PEs. To make decisions about where to allocate and when to schedule processes, we need to know how fast each process runs on each PE. Here are the process speeds:

	M1	M2
P1	5	5
P2	5	6
P3	–	5

The dash (–) entry signifies that the process cannot run on that type of processing element. In practice, a process may be excluded from some PEs for several reasons. If we use an ASIC to implement a special function, it will be able to implement only one process. A small CPU, such as a microcontroller, may not have enough memory for the process's code or data; it may also simply run too slowly to be useful. A process may run at different speeds on different CPUs for many reasons. Even when the CPUs run at the same clock rate, differences in the instruction sets can cause a process to be better suited to a particular CPU.

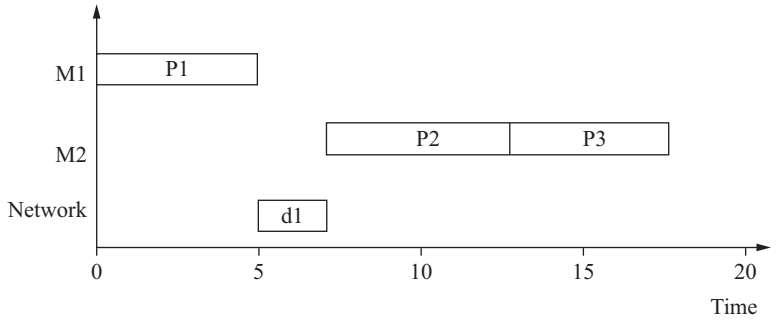
If two processes are allocated to the same PE, they can communicate using the processing element's internal memory and incur no network communication time. Each edge in the task graph corresponds to a data communication that must be carried over the network. Because all PEs communicate at the same rate, the data communication rate is the same for all transmissions between PEs. We need to know how long each communication takes. In this case, *d1* is a short message requiring two time units, and *d2* is a longer communication requiring four time units.

As an initial design, let us allocate *P1* and *P2* to *M1* and *P3* to *M2*. This allocation would, on the surface, appear to be a good one, because *P1* and *P2* are both placed on the processor that runs them the fastest. This schedule shows what happens to all the PEs and the network:



The schedule has a length of 19. The *d1* message is sent between the processes internal to *P1* and does not appear on the bus.

Let's try a different allocation: *P1* on *M1* and *P2* and *P3* on *M2*. This makes *P2* run more slowly. Here is the new schedule:



The length of this schedule is 18 or one time units less than the other schedule. The increased computation time of P2 is more than made up for by being able to transmit a shorter message on the bus. If we had not taken communication into account when analyzing the total execution time, we could have made the wrong choice of which processes to put on the same PE.

10.4.5 System integration

The design of an accelerated system often requires combining several different types of components. Serial busses are often used for module-to-module communication, particularly for tasks such as initialization and configuration.

The **I^2C bus** [Phi92] is a well-known bus commonly used to link microcontrollers and other modules into systems. It has even been used for the command interface in an MPEG-2 video chip [van97]; while a separate bus was used for high-speed video data, setup information was transmitted to the on-chip controller through an I^2C bus interface.

Physical layer

I^2C is designed to be low cost, easy to implement, and of moderate speed (up to 100 kbits/s for the standard bus and up to 400 kbits/s for the extended bus). As a result, it uses only two lines: the **serial data line (SDL)** for data and the **serial clock line (SCL)**, which indicates when valid data are on the data line. Fig. 10.10 shows the structure of a typical I^2C bus system. Every node in the network is connected to both SCL and SDL. Some nodes may be able to act as bus controllers, and the bus may have more than one controller. Other nodes may act as responders that respond only to requests from controllers.

Electrical interface

The basic electrical interface to the bus is shown in Fig. 10.11. The bus does not define particular voltages to be used for high or low, so that either bipolar or MOS circuits can be connected to the bus. Both bus signals use open collector/open drain circuits.[1] A pull-up resistor keeps the default state of the signal high, and transistors

[1] An open collector uses a bipolar transistor, while an open drain circuit uses an MOS transistor

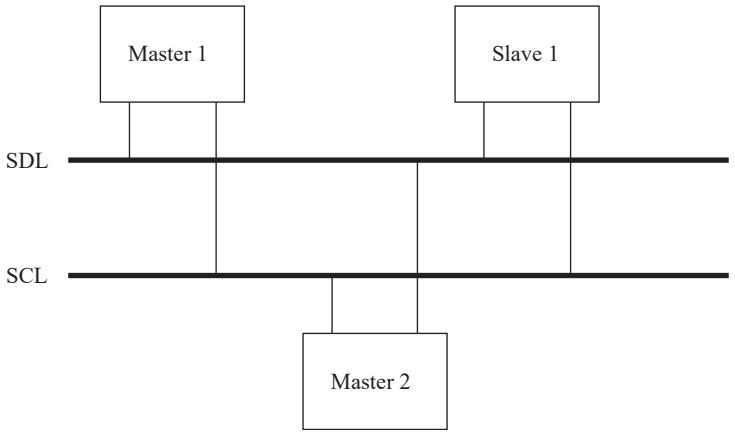


FIGURE 10.10

Structure of an I^2C bus system.

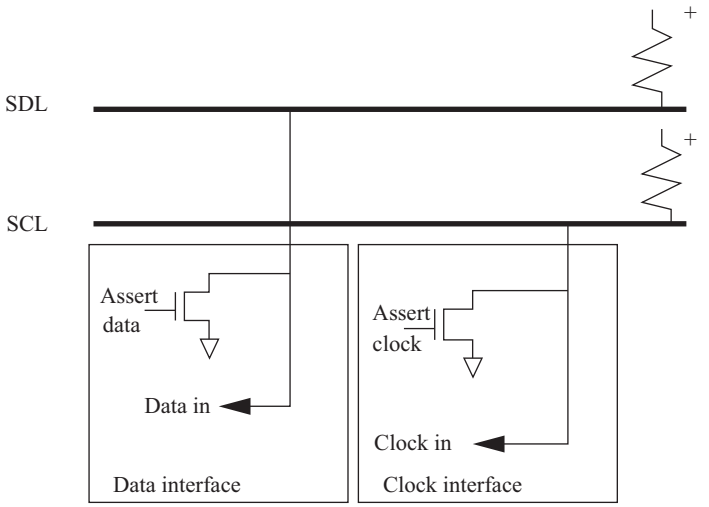


FIGURE 10.11

Electrical interface to the I^2C bus.

are used in each bus device to pull down the signal when a zero is to be transmitted. Open collector/open drain signaling allows several devices to simultaneously write the bus without causing electrical damage.

The open collector/open drain circuitry allows a responder device to stretch a clock signal during a read from a responder. The controller handles generating the SCL clock, but the responder can stretch the low period of the clock (but not the high period) if necessary.

The I^2C bus is designed as a multicontroller bus; any one of several different devices may act as the controller at various times. As a result, there is no global controller to generate the clock signal on the SCL. Instead, a controller drives both the SCL and SDL when it is sending data. When the bus is idle, both the SCL and SDL remain high. When two devices try to drive either SCL or SDL to different values, the open collector/open drain circuitry prevents errors, but each controller device must listen to the bus while transmitting to be sure that it is not interfering with another message. If the device receives a different value than it is trying to transmit, then it knows that it is interfering with another message.

Data link layer

Every I^2C device has an address. The system designer determines the addresses of the devices, usually as part of the program for the I^2C driver. The addresses must, of course, be chosen so that no two devices in the system have the same address. A device address is 7 bits in the standard I^2C definition (the extended I^2C allows 10-bit addresses). Address 0000000 is used to signal a **general call** or bus broadcast, which can be used to signal all devices simultaneously. Address 11110XX is reserved for the extended 10-bit addressing scheme; there are several other reserved addresses as well.

A **bus transaction** comprises a series of one-byte **transmissions** and an address followed by one or more data bytes. I^2C encourages a data-push programming style. When a controller wants to write a responder, it transmits the responder's address, followed by the data. Because a responder cannot initiate a transfer, the controller must send a read request with the responder's address and let the responder transmit the data. Therefore, an address transmission includes the 7-bit address and one bit for data direction: 0 for writing from the controller to the responder and 1 for reading from the responder to the controller. (This explains the 7-bit addresses on the bus.) The format of the address transmission is shown in Fig. 10.12.

A bus transaction is initiated by a start signal and completed with an end signal:

- A start is signaled by leaving the SCL high and sending a one-to-zero transition on SDL.
- A stop is signaled by setting the SCL high and sending a zero-to-one transition on SDL.

However, starts and stops must be paired. A controller can write and then read (or read and then write) by sending a start after the data transmission, followed by another address transmission and then more data. The basic state transition graph for the controller's actions in a bus transaction is shown in Fig. 10.13.

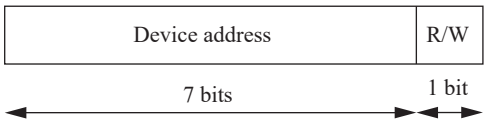


FIGURE 10.12

Format of an I^2C address transmission.

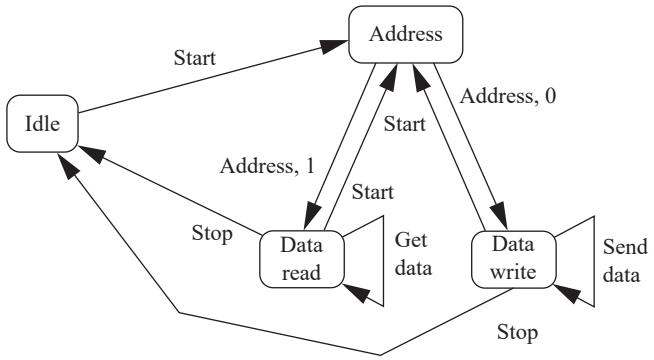


FIGURE 10.13

State transition graph for an I^2C bus controller.

The formats of some typical complete bus transactions are shown in Fig. 10.14. In the first example, the controller writes two bytes to the addressed responder. In the second, the controller requests a read from a responder. In the third, the controller writes one byte to the responder, and then, sends another start to initiate a read from the responder.

Byte format

Fig. 10.15 shows how a data byte is transmitted on the bus, including start and stop events. The transmission starts when SDL is pulled low while SCL remains high. After this start condition, the clock line is pulled low to initiate the data transfer. At each bit, the clock line goes high, while the data line assumes its proper value of 0 or 1. An acknowledgment is sent at the end of every 8-bit transmission, whether it is an address or data. For acknowledgment, the transmitter does not pull down the SDL, allowing the receiver to set the SDL to zero if it properly receives the byte. After

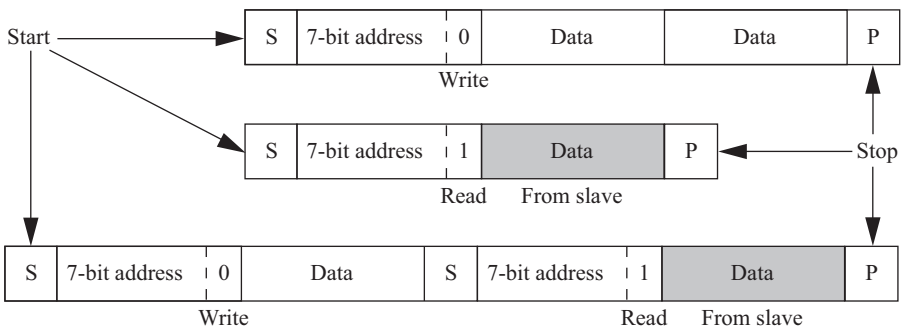


FIGURE 10.14

Typical bus transactions on the I^2C bus.

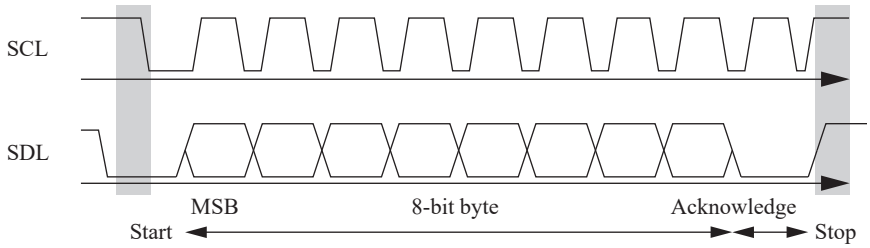


FIGURE 10.15

Transmitting a byte on the I^2C bus.

the acknowledgment, the SDL goes from low to high, while the SCL is high, signaling the stop condition.

Bus arbitration

The bus uses this feature to arbitrate each message. When sending, devices listen to the bus as well. If a device is trying to send logic 1 but hears a logic 0, it immediately stops transmitting and gives the other sender priority. The devices should be designed so that they can stop transmitting in time to allow a valid bit to be sent. In many cases, arbitration will be completed during the address portion of a transmission, but arbitration may continue into the data portion. If two devices are trying to send identical data to the same address, then of course they never interfere, and both succeed in sending their message. This form of arbitration is like the CAN bus arbitration seen in Section 10.3.1.

Application interface

The I^2C interface on a microcontroller can be implemented with varying percentages of functionality in software and hardware [Phi89]. As illustrated in Fig. 10.16, a typical system has a one-bit hardware interface with routines for byte-level functions. The I^2C device takes care to generate the clock and data. The application code calls

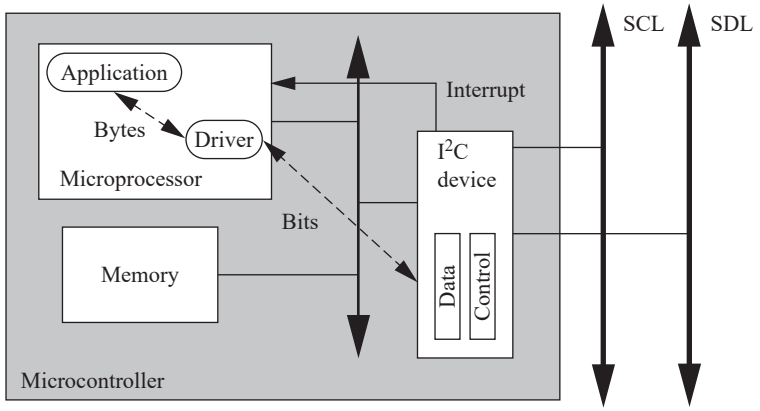


FIGURE 10.16

An I^2C interface in a microcontroller.

for routines to send an address, send a data byte, and so on, which then generates the SCL and SDL, acknowledges, and so forth. One of the microcontroller's timers is typically used to control the length of bits on the bus. Interrupts may be used to recognize bits. However, when used in controller mode, polled I/O may be acceptable if no other pending tasks can be performed because controllers initiate their own transfers.

10.4.6 Debugging

It is generally good policy to separately debug the basic interface between the accelerator and the rest of the system before integrating the full accelerator into the platform.

Hardware/software co-simulation can be effective in accelerator design. Because the co-simulator allows you to run software relatively efficiently alongside a hardware simulation, it allows you to exercise the accelerator in a realistic but simulated environment. It is especially difficult to exercise the interface between the accelerator core and the host CPU without running the CPU's accelerator driver. It is much better to do so in a simulator before fabricating the accelerator rather than to have to modify the hardware prototype of the accelerator.

10.5 Design example: video accelerator

In this section, we consider the design of a video accelerator, specifically a motion estimation accelerator. Digital video is a computationally intensive task, so it is well suited to acceleration. Motion estimation engines are used in real-time search engines; we may want to have one attached to our PC to experiment with video processing techniques.

10.5.1 Video compression

Before examining the video accelerator itself, let's look at video compression algorithms to understand the role played by a motion estimation engine.

Fig. 10.17 shows a block diagram for MPEG-2 video compression [Has97]. MPEG-2 forms the basis for U.S. HDTV broadcasting. This compression uses several component algorithms together in a feedback loop. The discrete cosine transform (DCT) used in JPEG also plays a key role in MPEG-2. As in still image compression, the DCT of a block of pixels is quantized for lossy compression, and then, subjected to lossless variable-length coding to further reduce the number of bits required to represent the block.

Motion-based coding

However, JPEG-style compression alone does not sufficiently reduce video bandwidth for many applications. MPEG uses motion to encode one frame in terms of another. Rather than sending each frame separately, as in motion JPEG, some frames are sent as modified forms of other frames using a technique known as **block motion estimation**. During encoding, the frame is divided into **macroblocks**. Macroblocks

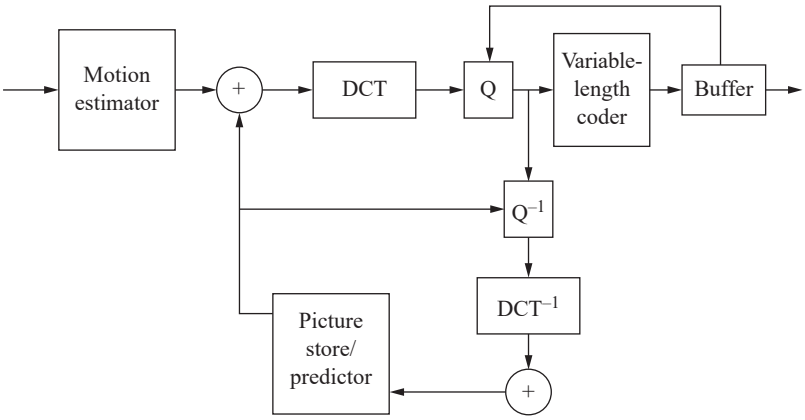


FIGURE 10.17

Block diagram of the MPEG-2 compression algorithm.

from one frame are identified in other frames using correlation. The frame can then be encoded using a vector that describes the motion of the macroblock from one frame to another without explicitly transmitting all the pixels. As shown in Fig. 10.17, the MPEG-2 encoder also uses a feedback loop to further improve image quality. This form of coding is lossy, and several different conditions can cause prediction to be imperfect: objects within a macroblock may move from one frame to the next or a macroblock may not be found by the search algorithm, *etc*. The encoder uses the encoding information to recreate the lossily-encoded picture, compares it to the original frame, and generates an error signal that can be used by the receiver to fix smaller errors. The decoder must keep some recently decoded frames in the memory so that it can retrieve the pixel values of the macroblocks. This internal memory saves a great deal of transmission and storage bandwidth.

The concept of block motion estimation is illustrated in Fig. 10.18. The goal is to perform a two-dimensional correlation to find the best match between the regions in the two frames. We divide the current frame into 16 × 16 macroblocks. For every macroblock in the frame, we want to find the region in the previous frame that most closely matches the macroblock. Searching over the entire previous frame would be too expensive, so we usually limit the search to a given area, centered around the macroblock and larger than the macroblock. We try the macroblock at various offsets in the search area. We measure similarity using the following sum-of-differences measure:

$$\sum_{1 \le i,j \le n} \left| M\,(i,j) - S\left(i - o_x, j - o_y\right) \right| \qquad \text{(Eq. 10.3)}$$

where $M(i,j)$ is the intensity of the macroblock at pixel i,j, $S(i,j)$ is the intensity of the search region, n is the size of the macroblock in one dimension, and $< o_x,\ o_y >$ is the

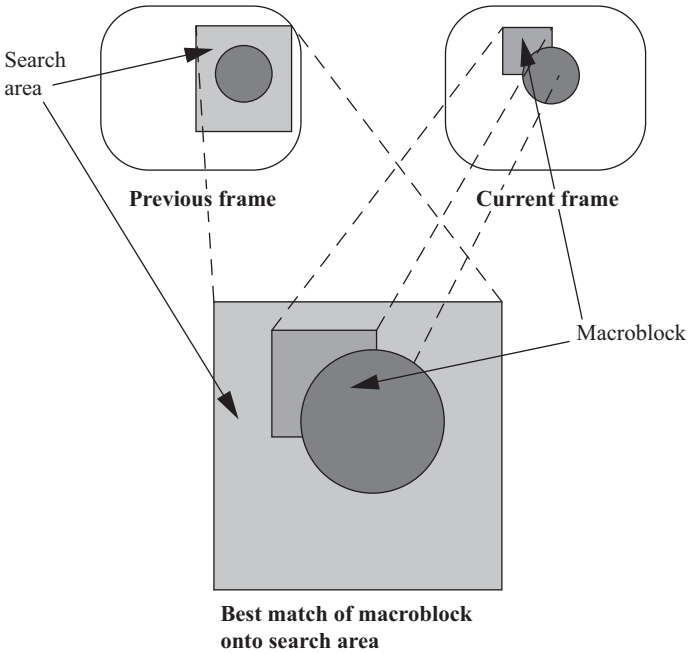


FIGURE 10.18

Block motion estimation.

offset between the macroblock and the search region. Intensity is measured as an 8-bit luminance that represents a monochrome pixel; color information is not used in motion estimation. We choose the macroblock position relative to the search area that gives us the smallest value for this metric. The offset at this chosen position describes a vector from the search area center to the macroblock's center, which is called the **motion vector**.

10.5.2 Algorithm and requirements

For simplicity, we will build an engine for a full search that compares the macroblock and search area at every possible point. Because this is an expensive operation, several methods have been proposed for conducting a sparser search of the search area. More advanced algorithms are used in practice. We choose a full-motion search here to concentrate on some basic issues in the relationship between the accelerator and the rest of the system.

A good way to describe the algorithm is in C. Some basic parameters of the algorithm are illustrated in Fig. 10.19. The image being searched includes some features such as a circle; in this case, parts of the circle lie outside the search area. Here is the C

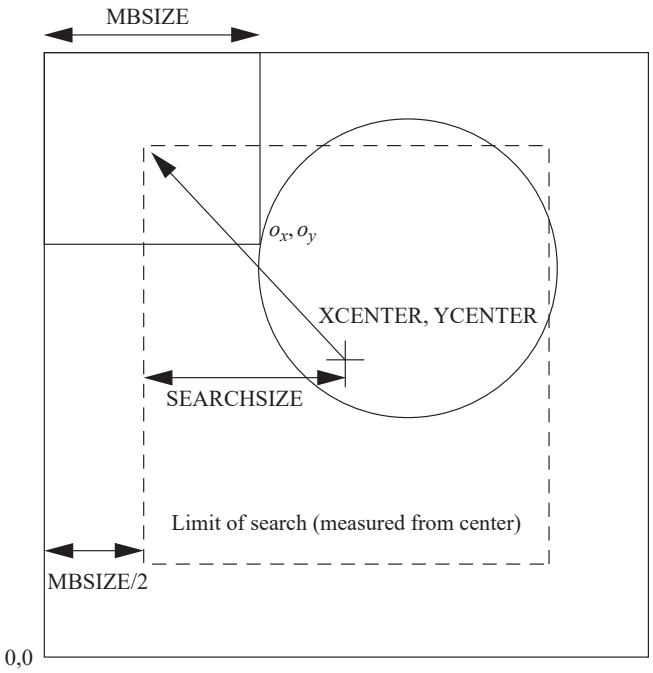


FIGURE 10.19

Block motion search parameters.

code for a single search, which assumes that the search region does not extend past the boundary of the frame.

```
bestx = 0; besty = 0; /*initialize best location--none yet */
bestsad = MAXSAD; /*best sum-of--difference thus far */
for (ox = –SEARCHSIZE; ox < SEARCHSIZE; ox++) {
    /*x search ordinate */
    for (oy = –SEARCHSIZE; oy < SEARCHSIZE; oy++) {
        /*y search ordinate */
        int result = 0;
        for (i = 0; i <MBSIZE; i++) {
            for (j = 0; j <MBSIZE; j++) {
                 result = result + iabs(mb[i][j] –
                      search[i –ox + XCENTER][j – oy + YCENTER]);
            }
        }
        if (result <= bestsad) { /* found better match */
             bestsad = result;
             bestx = ox; besty = oy;
    }
}
```

The arithmetic for each pixel is simple, but we must process a lot of pixels. If `MBSIZE` is 16 and `SEARCHSIZE` is eight, and remembering that the search distance in each dimension is $8 + 1 + 8$, then we must perform

$$n_{ops} = (16 \times 16) \times (17 \times 17) = 73,984 \quad \text{(Eq. 10.4)}$$

difference operations to find the motion vector for a single macroblock, which requires looking at twice as many pixels; one from the search area and one from the macroblock. We can now see an interest in algorithms that do not require a full search. To process the video, we will have to perform this computation on every macroblock of every frame. Adjacent blocks have overlapping search areas, so we will want to avoid reloading pixels we already have.

One relatively low-resolution standard video format, common intermediate format (CIF), has a frame size of 352×288, which gives an array of 22×18 macroblocks. If we want to encode video, we will have to perform motion estimation on every macroblock of most frames (some frames are sent without using motion compensation).

We will build the system using an FPGA connected to the PCIe bus of a personal computer. We clearly need a high-bandwidth connection, such as the PCIe between the accelerator and the CPU. We can use the accelerator to experiment with video processing, among other things. Here are the requirements for the system:

Name	Block motion estimator
Purpose	Perform block motion estimation within a PC system
Inputs	Macroblocks and search areas
Outputs	Motion vectors
Functions	Compute motion vectors using full search algorithms
Performance	As fast as we can get
Manufacturing cost	$100
Power	Powered by PC power supply
Physical size and weight	Packaged as PCIe card for PC

10.5.3 Specification

The specifications for the system are relatively straightforward because the algorithm is simple. Fig. 10.20 defines some classes that describe basic data types in the system: the motion vector, the macroblock, and the search area. These definitions are straightforward. Because the behavior is simple, we need to define only two classes to describe it: the accelerator itself and the PC. These classes are shown in Fig. 10.21. The PC makes its memory accessible to the accelerator. The accelerator provides a behavior `compute-mv()` that performs the block motion estimation algorithm. Fig. 10.22 shows a sequence diagram that describes the operation of `compute-mv()`.

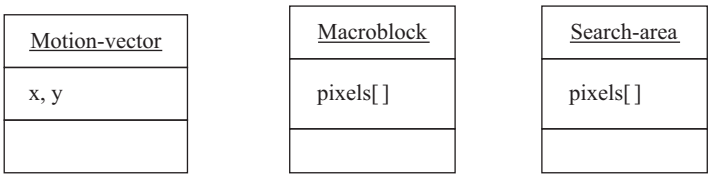


FIGURE 10.20

Classes describing basic data types in the video accelerator.

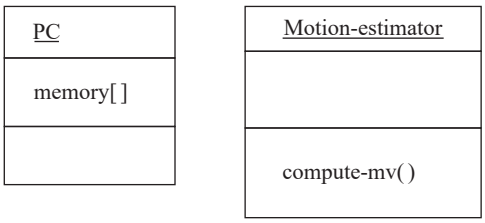


FIGURE 10.21

Basic classes for the video accelerator.

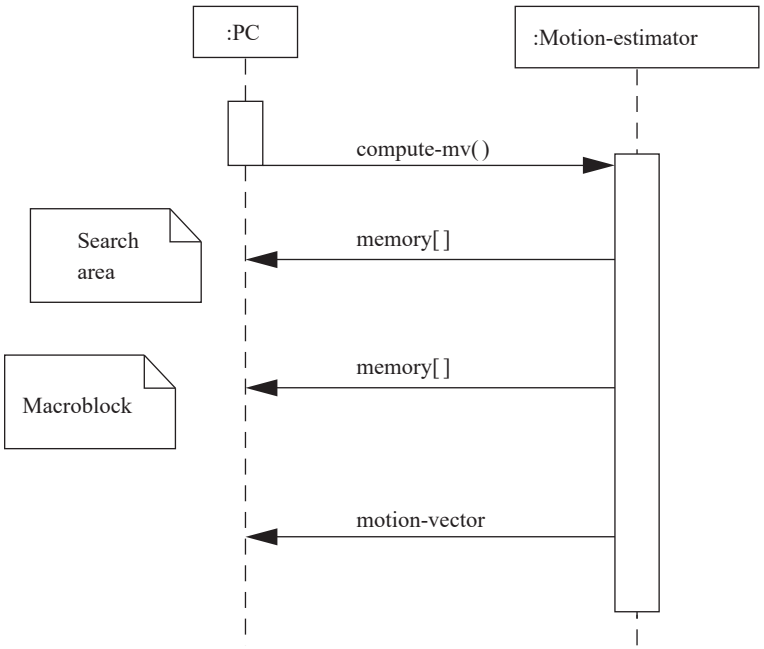


FIGURE 10.22

Sequence diagram for the video accelerator.

After initiating the behavior, the accelerator reads the search area and macroblock from the PC; after computing the motion vector, it returns it to the PC.

10.5.4 Architecture

The accelerator will be implemented in an FPGA on a card connected to a computer's PCIe slot. Such accelerators can be purchased or can be designed from scratch. If you design such a card from scratch, you must decide early on whether the card will be used only for this video accelerator or whether it should be made general enough to support other applications as well.

The architecture for the accelerator requires some thought because of the large amount of data required by the algorithm. The macroblock has $16 \times 16 = 256$ pixels; the search area has $(8 + 8 + 1 + 8 + 8)2^2 = 1089$ pixels. The FPGA may not have enough memory to hold 1089 8-bit values. We have to use a memory external to the FPGA, but on the accelerator board to hold the pixels.

There are many possible architectures for motion estimators. One is shown in Fig. 10.23. The machine has two memories: one for the macroblock and another for the search memories. It has 16 PEs that perform the difference calculation on a pair of pixels; the comparator sums them up and selects the best value to find the motion vector. This architecture can be used to implement algorithms other than a full search by changing address generation and control. Depending on the number of

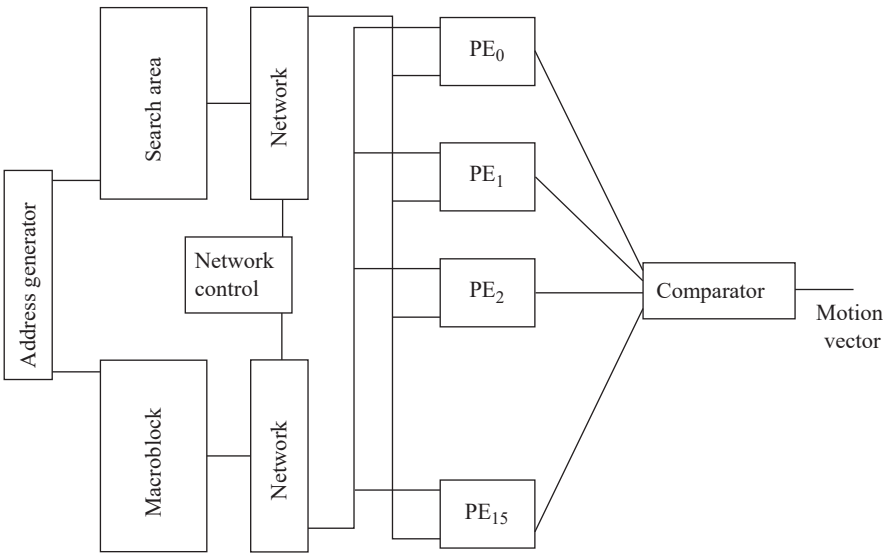


FIGURE 10.23

An architecture for the motion estimation accelerator [Dut96].

t	M	S	S9	PE_0	PE_1	PE_2
0	M(0,0)	S(0,0)		|M(0,0) – S(0,0)|		
1	M(0,1)	S(0,1)		|M(0,1) – S(0,1)|	|M(0,0) – S(0,1)|	
2	M(0,2)	S(0,2)		|M(0,2) – S(0,2)|	|M(0,1) – S(0,2)|	|M(0,0) – S(0,2)|
3	M(0,3)	S(0,3)		|M(0,3) – S(0,3)|	|M(0,2) – S(0,3)|	|M(0,1) – S(0,3)|
4	M(0,4)	S(0,4)		|M(0,4) – S(0,4)|	|M(0,3) – S(0,4)|	|M(0,2) – S(0,4)|
5	M(0,5)	S(0,5)		|M(0,5) – S(0,5)|	|M(0,4) – S(0,5)|	|M(0,3) – S(0,5)|
6	M(0,6)	S(0,6)		|M(0,6) – S(0,6)|	|M(0,5) – S(0,6)|	|M(0,4) – S(0,6)|
7	M(0,7)	S(0,7)		|M(0,7) – S(0,7)|	|M(0,6) – S(0,7)|	|M(0,5) – S(0,7)|
8	M(0,8)	S(0,8)		|M(0,8) – S(0,8)|	|M(0,7) – S(0,8)|	|M(0,6) – S(0,8)|
9	M(0,9)	S(0,9)		|M(0,9) – S(0,9)|	|M(0,8) – S(0,9)|	|M(0,7) – S(0,9)|
10	M(0,10)	S(0,10)		|M(0,10) – S(0,10)|	|M(0,9) – S(0,10)|	|M(0,8) – S(0,10)|
11	M(0,11)	S(0,11)		|M(0,11) – S(0,11)|	|M(0,10) – S(0,11)|	|M(0,9) – S(0,11)|
12	M(0,12)	S(0,12)		|M(0,12) – S(0,12)|	|M(0,11) – S(0,12)|	|M(0,10) – S(0,12)|
13	M(0,13)	S(0,13)		|M(0,13) – S(0,13)|	|M(0,12) – S(0,13)|	|M(0,11) – S(0,13)|
14	M(0,14)	S(0,14)		|M(0,14) – S(0,14)|	|M(0,13) – S(0,14)|	|M(0,12) – S(0,14)|
15	M(0,15)	S(0,15)		|M(0,15) – S(0,15)|	|M(0,14) – S(0,15)|	|M(0,13) – S(0,15)|
16	M(1,0)	S(1,0)	S(0,16)	|M(1,0) – S(1,0)|	|M(0,15) – S(0,16)|	|M(0,14) – S(0,16)|
17	M(1,1)	S(1,1)	S(0,17)	|M(1,1) – S(1,1)|	|M(1,0) – S(1,1)|	|M(0,15) – S(0,17)|

FIGURE 10.24

A schedule of pixel fetches for a full search [Yan89].

motion estimation algorithms that you want to execute on the machine, the networks connecting the memories to the PEs may also be simplified.

Fig. 10.24 shows how we can schedule the transfer of pixels from the memories to the processing elements to efficiently compute a full search of this architecture. The schedule fetches one pixel from the macroblock memory and (in steady state) two pixels from the search area memory per clock cycle. The pixels are distributed to the PEs in a regular pattern, as shown by the schedule. This schedule computes 16 correlations between the macroblock and the search area simultaneously. The computations for each correlation are distributed among the processing elements; the comparator handles collecting the results, finding the best match value, and remembering the corresponding motion vector.

Based on our understanding of efficient architectures for accelerating motion estimation, we can derive a more detailed definition of the architecture in a unified modeling language, which is shown in Fig. 10.25. The system includes the two memories for pixels, one a single-port memory and the other a dual-port memory. A bus

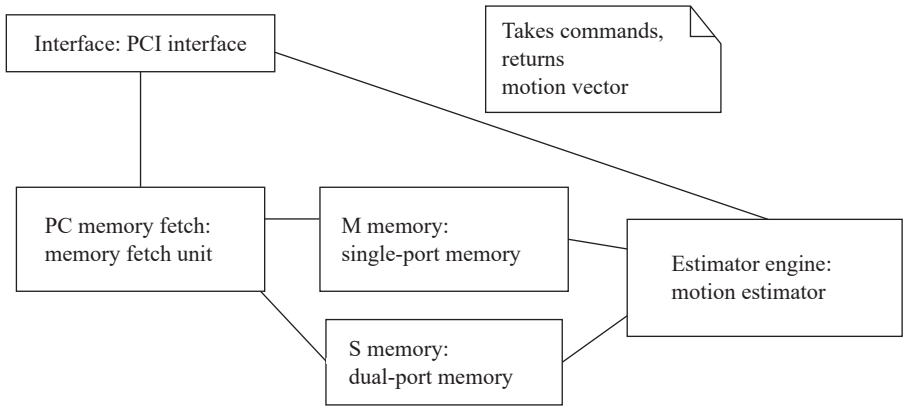


FIGURE 10.25

Object diagram for the video accelerator.

interface module handles communicating with the PCIe bus and the rest of the system. The estimation engine reads pixels from the *M* and *S* memories, takes commands from the bus interface, and returns the motion vector to the bus interface.

10.5.5 Component design

If we want to use a standard FPGA accelerator board to implement the accelerator, we must first make sure that it provides the proper memory required for *M* and *S*. Once we have verified that the accelerator board has the required structure, we can concentrate on designing FPGA logic. Designing an FPGA is, for the most part, a straightforward exercise in logic design. Because the logic for the accelerator is very regular, we can improve the FPGA's clock rate by properly placing the logic in the FPGA to reduce wire lengths.

If we are designing our own accelerator board, we must design both the video accelerator design proper and the interface to the PCIe bus. We can create and exercise the video accelerator architecture in a hardware description language like VHDL or Verilog and simulate its operation. Designing the PCIe interface requires somewhat different techniques because we may not have a simulation model for a PCIe bus. We may want to verify the operation of the basic PCIe interface before we finish implementing the video accelerator logic.

The host PC will probably deal with the accelerator as an I/O device. The accelerator board will have its own driver that is responsible for talking to the board. Because most of the data transfers are performed directly by the board using DMA, the driver can be relatively simple.

10.5.6 System testing

Testing video algorithms requires a large amount of data. Luckily, the data represent images and video, which are plentiful. Because we are designing only a motion estimation accelerator and not a complete video compressor, it is probably easiest to use images, not video, for test data. You can use standard video tools to extract a few frames from a digitized video and store them in JPEG format. An open source for JPEG encoders and decoders is available. These programs can be modified to read JPEG images and put out pixels in the format required by your accelerator. With a little more cleverness, the resulting motion vector can be written back onto the image for a visual confirmation of the result. If you want to be adventurous and try motion estimation on video, open-source MPEG encoders and decoders are also available.

10.6 Summary

Multiprocessors provide both absolute performance and efficiency. They do, however, introduce new levels of system complexity. Programming multiprocessors requires both new programming models and development methodologies. Multiprocessors are often heterogeneous, so that different parts of an application can be mapped to specialized PEs. A programmable PE may be specialized by, for example, adding new instructions. Accelerators are PEs designed to perform specific tasks. When adding accelerators to the system, we must be sure that the system can send data to and receive data from the rest of the system at the required rates.

What we learned

- Multiprocessors can help improve real-time performance and energy consumption.
- Shared memory and message passing systems are different organizations of multiprocessors.
- MPSoCs are single-chip multiprocessors with low-latency communication while distributed systems are physically larger and generally have longer-latency communication systems.
- Shared memory multiprocessors are often used in single-chip signal processing and control systems.
- Performance analysis of an accelerated system is challenging. We must consider the performance of several implementations of an algorithm (CPU, accelerator) as well as communication costs for various configurations.
- We must partition the behavior, schedule operations in time, and allocate operations to PEs to design the system.

Further reading

Kopetz [Kop97] provides a thorough introduction to the design of distributed embedded systems. Staunstrup and Wolf's edited volume [Sta97] surveys hardware/software co-design, including techniques for accelerated systems, such as those described in this chapter. Gupta and De Micheli [Gup93] and Ernst et al. [Ern93] describe early techniques for cosynthesis of accelerated systems. Callahan et al. [Cal00] describe an on-chip reconfigurable coprocessor connected to a CPU. The book *DVD Demystified* [Tay06] provides a thorough introduction to the DVD.

Questions

Q10-1 Describe an I^2C bus at the following OSI-compliant levels of detail:

- **a.** physical
- **b.** data link
- **c.** network
- **d.** transport

Q10-2 You are designing an embedded system using an Intel Atom as a host. Does it make sense to add an accelerator to implement the function $z = ax + by + c$? Explain.

Q10-3 You are designing an embedded system using an embedded processor with no floating-point support as host. Does it make sense to add an accelerator to implement the floating-point function $S = A\sin(2\pi f + \phi)$? Explain.

Q10-4 You are designing an embedded system using a high-performance embedded processor with floating point as host. Does it make sense to add an accelerator to implement the floating-point function $S = A\sin(2\pi f + \phi)$? Explain.

Q10-5 You are designing an accelerated system that performs the following function as its main task:

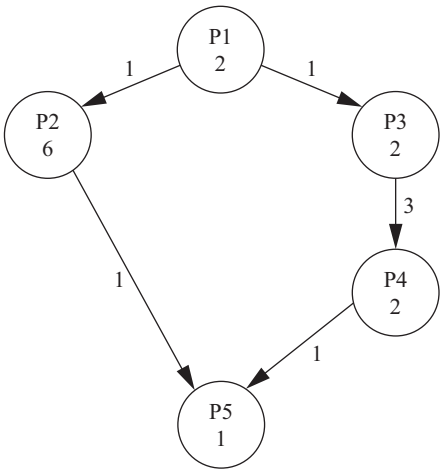
```
for (i = 0; i <M; i++)
     for (j = 0; j <N; j++)
          f[i][j] = (pix[i][j –1] + pix[i – 1][j] + pix[i][j] +
          pix[i + 1][j] +
          pix[i][j + 1])/(5*MAXVAL);
```

Assume that the accelerator has the entire `pix` and `f` arrays in its internal memory during the entire computation—`pix` is read into the accelerator before the operations begin and `f` is written out after all computations have been completed.

a. Show a system schedule for the host, accelerator, and bus, assuming that the accelerator is inactive during all data transfers. All data are sent to the accelerator before it starts, and data are read from the accelerator after the computations are finished.

b. Show a system schedule for the host, accelerator, and bus, assuming that the accelerator has enough memory for two `pix` and `f` arrays and that the host can transfer data for one set of computations while another set is being performed.

Q10-6 Find the longest path through the graph below, using the computation times on the nodes and the communication times on the edges.



Q10-7 Write pseudocode for an algorithm to determine the longest path through a system execution graph. The longest path is to be measured from one designated entry point to one exit point. Each node in the graph is labeled with a number, giving the execution time of the process represented by that node.

Lab exercises

L10-1 Determine how much logic in an FPGA must be devoted to a PCIe bus interface and how much would be left for an accelerator core.

L10-2 Develop a debugging scheme for an accelerator. Determine how you would easily enter data into the accelerator and easily observe its behavior. You will need to verify the system thoroughly, starting with basic communication and going through algorithmic verification.

L10-3 Develop a generic streaming interface for an accelerator. The interface should allow streaming data to be read by the accelerator from the host's memory. It should also allow streaming data to be written from the accelerator back to the memory. The interface should include a host-side mechanism for filling and draining streaming data buffers.