8 Embedded Processors

In **general-purpose computing**, the variety of instruction set architectures today is limited, with the Intel x86 architecture overwhelmingly dominating all. There is no such dominance in embedded computing. On the contrary, the variety of processors can be daunting to a system designer. Our goal in this chapter is to give the reader the tools and

vocabulary to understand the options and to critically evaluate the properties of processors. We particularly focus on the mechanisms that provide concurrency and control over timing, because these issues loom large in the design of cyber-physical systems.

When deployed in a product, embedded processors typically have a dedicated function. They control an automotive engine or measure ice thickness in the Arctic. They are not asked to perform arbitrary functions with user-defined software. Consequently, the processors can be more specialized. Making them more specialized can bring enormous benefits. For example, they may consume far less energy, and consequently be usable with small batteries for long periods of time. Or they may include specialized hardware to perform operations that would be costly to perform on general-purpose hardware, such as image analysis.

When evaluating processors, it is important to understand the difference between an **instruction set architecture** (**ISA**) and a **processor realization** or a **chip**. The latter is a piece of silicon sold by a semiconductor vendor. The former is a definition of the instructions that the processor can execute and certain structural constraints (such as word size) that realizations must share. x86 is an ISA. There are many realizations. An ISA is an abstraction shared by many realizations. A single ISA may appear in many different chips, often made by different manufacturers, and often having widely varying performance profiles.

The advantage of sharing an ISA in a family of processors is that software tools, which are costly to develop, may be shared, and (sometimes) the same programs may run correctly on multiple realizations. This latter property, however, is rather treacherous, since an ISA does not normally include any constraints on timing. Hence, although a program may execute logically the same way on multiple chips, the system behavior may be radically different when the processor is embedded in a cyber-physical system.

8.1 Types of Processors

As a consequence of the huge variety of embedded applications, there is a huge variety of processors that are used. They range from very small, slow, inexpensive, low-power devices, to high-performance, special-purpose devices. This section gives an overview of some of the available types of processors.

8.1.1 Microcontrollers

A **microcontroller** (μC) is a small computer on a single integrated circuit consisting of a relatively simple **central processing unit** (**CPU**) combined with peripheral devices such as memories, I/O devices, and timers. By some accounts, more than half of all CPUs sold worldwide are microcontrollers, although such a claim is hard to substantiate because the difference between microcontrollers and general-purpose processors is indistinct. The simplest microcontrollers operate on 8-bit words and are suitable for applications that require small amounts of memory and simple logical functions (vs. performance-intensive arithmetic functions). They may consume extremely small amounts of energy, and often include a **sleep mode** that reduces the power consumption to nanowatts. Embedded components such as sensor network nodes and surveillance devices have been demonstrated that can operate on a small battery for several years.

Microcontrollers can get quite elaborate. Distinguishing them from general-purpose processors can get difficult. The Intel Atom, for example, is a family of x86 CPUs used mainly in netbooks and other small mobile computers. Because these processors are designed to use relatively little energy without losing too much performance relative to processors used in higher-end computers, they are suitable for some embedded applications and in servers where cooling is problematic. AMD's Geode is another example of a processor near the blurry boundary between general-purpose processors and microcontrollers.

8.1.2 DSP Processors

Many embedded applications do quite a bit of signal processing. A signal is a collection of sampled measurements of the physical world, typically taken at a regular rate called the sample rate. A motion control application, for example, may read position or location information from sensors at sample rates ranging from a few Hertz (Hz, or samples per second) to a few hundred Hertz. Audio signals are sampled at rates ranging from 8,000 Hz (or 8 kHz, the sample rate used in telephony for voice signals) to 44.1 kHz (the sample rate of CDs). Ultrasonic applications (such as medical imaging) and high-performance music applications may sample sound signals at much higher rates. Video typically uses sample rates of 25 or 30 Hz for consumer devices to much higher rates for specialty measurement applications. Each sample, of course, contains an entire image (called a frame), which itself has many samples (called pixels) distributed in space rather than time. Software-defined radio applications have sample rates that can range from hundreds of kHz (for baseband processing) to several GHz (billions of Hertz). Other embedded applications

Microcontrollers

Most semiconductor vendors include one or more families of microcontrollers in their product line. Some of the architectures are quite old. The **Motorola 6800** and **Intel 8080** are 8-bit microcontrollers that appeared on the market in 1974. Descendants of these architectures survive today, for example in the form of the **Freescale 6811**. The **Zilog Z80** is a fully-compatible descendant of the 8080 that became one of the most widely manufactured and widely used microcontrollers of all time. A derivative of the Z80 is the Rabbit 2000 designed by Rabbit Semiconductor.

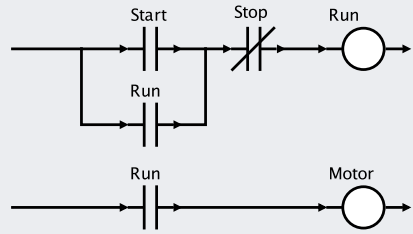
Another very popular and durable architecture is the **Intel 8051**, an 8-bit microcontroller developed by Intel in 1980. The 8051 ISA is today supported by many vendors, including Atmel, Infineon Technologies, Dallas Semiconductor, NXP, ST Microelectronics, Texas Instruments, and Cypress Semiconductor. The **Atmel AVR** 8-bit microcontroller, developed by Atmel in 1996, was one of the first microcontrollers to use on-chip flash memory for program storage. Although Atmel says AVR is not an acronym, it is believed that the architecture was conceived by two students at the Norwegian Institute of Technology, Alf-Egil Bogen and Vegard Wollan, so it may have originated as Alf and Vegard's RISC.

Many 32-bit microcontrollers implement some variant of an **ARM** instruction set, developed by ARM Limited. ARM originally stood for Advanced RISC Machine, and before that Acorn RISC Machine, but today it is simply ARM. Processors that implement the ARM ISA are widely used in mobile phones to realize the user interface functions, as well as in many other embedded systems. Semiconductor vendors license the instruction set from ARM Limited and produce their own chips. ARM processors are currently made by Alcatel, Atmel, Broadcom, Cirrus Logic, Freescale, LG, Marvell Technology Group, NEC, NVIDIA, NXP, Samsung, Sharp, ST Microelectronics, Texas Instruments, VLSI Technology, Yamaha, and others.

Other notable embedded microcontroller architectures include the **Motorola ColdFire** (later the Freescale ColdFire), the **Hitachi H8** and SuperH, the **MIPS** (originally developed by a team led by John Hennessy at Stanford University), the **PIC** (originally Programmable Interface Controller, from Microchip Technology), and the **PowerPC** (created in 1991 by an alliance of Apple, IBM, and Motorola).

Programmable Logic Controllers

A **programmable logic controller** (**PLC**) is a specialized form of a microcontroller for industrial automation. PLCs originated as replacements for control circuits using electrical relays to control machinery. They are typically designed for continuous operation in hostile environments (high temperature, humidity, dust, etc.).



Ladder Logic Diagram

PLCs are often programmed using **ladder logic**, a notation originally used to specify logic constructed with relays and switches. A **relay** is a switch where the contact is controlled by coil. When a voltage is applied to the coil, the contact closes, enabling current to flow through the relay. By interconnecting contacts and coils, relays can be used to build digital controllers that follow specified patterns.

In common notation, a contact is represented by two vertical bars, and a coil by a circle, as shown in the diagram above. The above diagram has two **rungs**. The Motor coil on the lower rung turns a motor on or off. The Start and Stop contacts represent pushbutton switches. When an operator pushes the Start button, the contact is closed, and current can flow from the left (the power rail) to the right (ground). Start is a **normally open** contact. The Stop contact is **normally closed**, indicated by the slash, meaning that it becomes open when the operator pushes the switch. The logic in the upper rung is interesting. When the operator pushes Start, current flows to the Run coil, causing both Run contacts to close. The motor will run, even after the Start button is released. When the operator pushes Stop, current is interrupted, and both Run contacts become open, causing the motor to stop. Contacts wired in parallel perform a logical OR function, and contacts wired in series perform a logical AND. The upper rung has feedback; the meaning of the rung is a fixed point solution to the logic equation implied by the diagram.

Today, PLCs are just microcontrollers in rugged packages with I/O interfaces suitable for industrial control, and ladder logic is a graphical programming notation for programs. These diagrams can get quite elaborate, with thousands of rungs. For details, we recommend Kamen (1999).

that make heavy use of signal processing include interactive games; radar, sonar, and LIDAR (light detection and ranging) imaging systems; video analytics (the extraction of information from video, for example for surveillance); driver-assist systems for cars; medical electronics; and scientific instrumentation.

Signal processing applications all share certain characteristics. First, they deal with large amounts of data. The data may represent samples in time of a physical processor (such as samples of a wireless radio signal), samples in space (such as images), or both (such as video and radar). Second, they typically perform sophisticated mathematical operations on the data, including filtering, system identification, frequency analysis, machine learning, and feature extraction. These operations are mathematically intensive.

Processors designed specifically to support numerically intensive signal processing applications are called **DSP processors**, or **DSP**s (**digital signal processor**s), for short. To get some insight into the structure of such processors and the implications for the embedded software designer, it is worth understanding the structure of typical signal processing algorithms.

A canonical signal processing algorithm, used in some form in all of the above applications, is **finite impulse response** (**FIR**) filtering. The simplest form of this algorithm is straightforward, but has profound implications for hardware. In this simplest form, an input signal x consists of a very long sequence of numerical values, so long that for design

The x86 Architecture

The dominant ISA for desktop and portable computers is known as the **x86**. This term originates with the Intel 8086, a 16-bit microprocessor chip designed by Intel in 1978. A variant of the 8086, designated the 8088, was used in the original IBM PC, and the processor family has dominated the PC market ever since. Subsequent processors in this family were given names ending in "86," and generally maintained backward compatibility. The Intel 80386 was the first 32-bit version of this instruction set, introduced in 1985. Today, the term "x86" usually refers to the 32-bit version, with 64-bit versions designated "x86-64." The **Intel Atom**, introduced in 2008, is an x86 processor with significantly reduced energy consumption. Although it is aimed primarily at netbooks and other small mobile computers, it is also an attractive option for some embedded applications. The x86 architecture has also been implemented in processors from AMD, Cyrix, and several other manufacturers.

purposes it should be considered infinite. Such an input can be modeled as a function $x: \mathbb{N} \to D$, where D is a set of values in some data type.[1] For example, D could be the set of all 16-bit integers, in which case, $x(0)$ is the first input value (a 16-bit integer), $x(1)$ is the second input value, etc. For mathematical convenience, we can augment this to $x: \mathbb{Z} \to D$ by defining $x(n) = 0$ for all $n < 0$. For each input value $x(n)$, an FIR filter must compute an output value $y(n)$ according to the formula,

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i) \,, \tag{8.1}$$

[1] For a review of this notation, see Appendix A on page 493.

DSP Processors

Specialized computer architectures for signal processing have been around for quite some time (Allen, 1975). Single-chip DSP microprocessors first appeared in the early 1980s, beginning with the Western Electric DSP1 from Bell Labs, the S28211 from AMI, the TMS32010 from Texas Instruments, the uPD7720 from NEC, and a few others. Early applications of these devices included voiceband data modems, speech synthesis, consumer audio, graphics, and disk drive controllers. A comprehensive overview of DSP processor generations through the mid-1990s can be found in Lapsley et al. (1997).

Central characteristics of DSPs include a hardware multiply-accumulate unit; several variants of the Harvard architecture (to support multiple simultaneous data and program fetches); and addressing modes supporting auto increment, circular buffers, and bit-reversed addressing (the latter to support FFT calculation). Most support fixed-point data precisions of 16-24 bits, typically with much wider accumulators (40-56 bits) so that a large number of successive multiply-accumulate instructions can be executed without overflow. A few DSPs have appeared with floating point hardware, but these have not dominated the marketplace.

DSPs are difficult to program compared to RISC architectures, primarily because of complex specialized instructions, a pipeline that is exposed to the programmer, and asymmetric memory architectures. Until the late 1990s, these devices were almost always programmed in assembly language. Even today, C programs make extensive use of libraries that are hand-coded in assembly language to take advantage of the most esoteric features of the architectures.

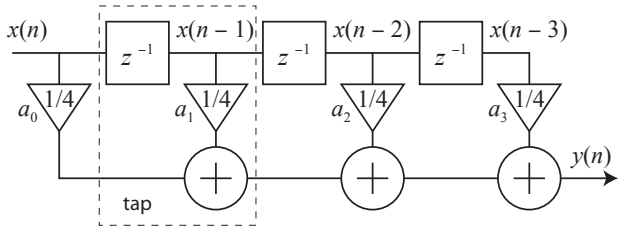


Figure 8.1: Structure of a tapped delay line implementation of the FIR filter of example 8.1. This diagram can be read as a dataflow diagram. For each $n \in \mathbb{N}$, each component in the diagram consumes one input value from each input path and produces one output value on each output path. The boxes labeled z^{-1} are unit delays. Their task is to produce on the output path the previous value of the input (or an initial value if there was no previous input). The triangles multiply their input by a constant, and the circles add their inputs.

where N is the length of the FIR filter, and the coefficients a_i are called its **tap values**. You can see from this formula why it is useful to augment the domain of the function x, since the computation of $y(0)$, for example, involves values $x(-1)$, $x(-2)$, etc.

Example 8.1: Suppose $N = 4$ and $a_0 = a_1 = a_2 = a_3 = 1/4$. Then for all $n \in \mathbb{N}$,

$$y(n) = (x(n) + x(n-1) + x(n-2) + x(n-3))/4 \,.$$

Each output sample is the average of the most recent four input samples. The structure of this computation is shown in Figure 8.1. In that figure, input values come in from the left and propagate down the **delay line**, which is tapped after each delay element. This structure is called a **tapped delay line**.

The rate at which the input values $x(n)$ are provided and must be processed is called the **sample rate**. If you know the sample rate and N, you can determine the number of arithmetic operations that must be computed per second.

Example 8.2: Suppose that an FIR filter is provided with samples at a rate of 1 MHz (one million samples per second), and that $N = 32$. Then outputs must be computed at a rate of 1 MHz, and each output requires 32 multiplications and 31 additions. A processor must be capable of sustaining a computation rate of 63 million arithmetic operations per second to implement this application. Of course, to sustain the computation rate, it is necessary not only that the arithmetic hardware be fast enough, but also that the mechanisms for getting data in and out of memory and on and off chip be fast enough.

An image can be similarly modeled as a function $x\colon H \times V \to D$, where $H \subset \mathbb{N}$ represents the horizontal index, $V \subset \mathbb{N}$ represents the vertical index, and D is the set of all possible pixel values. A **pixel** (or picture element) is a sample representing the color and intensity of a point in an image. There are many ways to do this, but all use one or more numerical values for each pixel. The sets H and V depend on the **resolution** of the image.

Example 8.3: Analog television is steadily being replaced by digital formats such as **ATSC**, a set of standards developed by the Advanced Television Systems Committee. In the US, the vast majority of over-the-air **NTSC** transmissions (National Television System Committee) were replaced with ATSC on June 12, 2009. ATSC supports a number of frame rates ranging from just below 24 Hz to 60 Hz and a number of resolutions. High-definition video under the ATSC standard supports, for example, a resolution of 1080 by 1920 pixels at a frame rate of 30 Hz. Hence, $H = \{0, \cdots, 1919\}$ and $V = \{0, \cdots, 1079\}$. This resolution is called 1080p in the industry. Professional video equipment today goes up to four times this resolution (4320 by 7680). Frame rates can also be much higher than 30 Hz. Very high frame rates are useful for capturing extremely fast phenomena in slow motion.

For a grayscale image, a typical filtering operation will construct a new image y from an original image x according to the following formula,

$$\forall\, i \in H, j \in V, \quad y(i,j) = \sum_{n=-N}^{N} \sum_{m=-M}^{M} a_{n,m} x(i-n, j-m)\ , \tag{8.2}$$

where $a_{n,m}$ are the filter coefficients. This is a two-dimensional FIR filter. Such a calculation requires defining x outside the region $H \times V$. There is quite an art to this (to avoid edge effects), but for our purposes here, it suffices to get a sense of the structure of the computation without being concerned for this detail.

A color image will have multiple **color channels**. These may represent luminance (how bright the pixel is) and chrominance (what the color of the pixel is), or they may represent colors that can be composed to get an arbitrary color. In the latter case, a common choice is an **RGBA** format, which has four channels representing red, green, blue, and the alpha channel, which represents transparency. For example, a value of zero for R, G, and B represents the color black. A value of zero for A represents fully transparent (invisible). Each channel also has a maximum value, say 1.0. If all four channels are at the maximum, the resulting color is a fully opaque white.

The computational load of the filtering operation in (8.2) depends on the number of channels, the number of filter coefficients (the values of N and M), the resolution (the sizes of the sets H and V), and the frame rate.

Example 8.4: Suppose that a filtering operation like (8.2) with $N = 1$ and $M = 1$ (minimal values for useful filters) is to be performed on a high-definition video signal as in Example 8.3. Then each pixel of the output image y requires performing 9 multiplications and 8 additions. Suppose we have a color image with three channels (say, RGB, without transparency), then this will need to performed 3 times for each pixel. Thus, each frame of the resulting image will require $1080 \times 1920 \times 3 \times 9 = 55,987,200$ multiplications, and a similar number of additions. At 30 frames per second, this translates into $1,679,616,000$ multiplications per second, and a similar number of additions. Since this is about the simplest operation one may perform on a high-definition video signal, we can see that processor architectures handling such video signals must be quite fast indeed.

In addition to the large number of arithmetic operations, the processor has to handle the movement of data down the delay line, as shown in Figure 8.1 (see box on page 221). By providing support for delay lines and multiply-accumulate instructions, as shown in Example 8.6, DSP processors can realize one tap of an FIR filter in one cycle. In that cycle, they multiply two numbers, add the result to an accumulator, and increment or decrement two pointers using modulo arithmetic.

8.1.3 Graphics Processors

A **graphics processing unit** (**GPU**) is a specialized processor designed especially to perform the calculations required in graphics rendering. Such processors date back to the 1970s, when they were used to render text and graphics, to combine multiple graphic patterns, and to draw rectangles, triangles, circles, and arcs. Modern GPUs support 3D graphics, shading, and digital video. Dominant providers of GPUs today are Intel, NVIDIA and AMD.

Some embedded applications, particularly games, are a good match for GPUs. Moreover, GPUs have evolved towards more general programming models, and hence have started to appear in other compute-intensive applications, such as instrumentation. GPUs are typically quite power hungry, and therefore today are not a good match for energy constrained embedded applications.

8.2 Parallelism

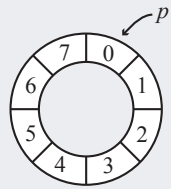
Most processors today provide various forms of parallelism. These mechanisms strongly affect the timing of the execution of a program, so embedded system designers have to understand them. This section provides an overview of the several forms and their consequences for system designers.

8.2.1 Parallelism vs. Concurrency

Concurrency is central to embedded systems. A computer program is said to be concurrent if different parts of the program *conceptually* execute simultaneously. A program is said to be **parallel** if different parts of the program *physically* execute simultaneously

Circular Buffers

An FIR filter requires a delay-line like that shown in Figure 8.1. A naive implementation would allocate an array in memory, and each time an input sample arrives, move each element in the array to the next higher location to make room for the new element in the first location. This would be enormously wasteful of memory bandwidth. A better approach is to use a **circular buffer**, where an array in memory is interpreted as having a ring-like structure, as shown below for a length-8 delay line:



Here, 8 successive memory locations, labeled 0 to 7, store the values in the delay line. A pointer p, initialized to location 0, provides access.

An FIR filter can use this circular buffer to implement the summation of (8.1). One implementation first accepts a new input value $x(n)$, and then calculates the summation backwards, beginning with the $i = N-1$ term, where in our example, $N = 8$. Suppose that when the n^{th} input arrives, the value of p is some number $p_i \in \{0, \cdots, 7\}$ (for the first input $x(0)$, $p_i = 0$). The program writes the new input $x(n)$ into the location given by p and then increments p, setting $p = p_i + 1$. All arithmetic on p is done modulo 8, so for example, if $p_i = 7$, then $p_i + 1 = 0$. The FIR filter calculation then reads $x(n-7)$ from location $p = p_i + 1$ and multiplies it by a_7. The result is stored in an **accumulator** register. It again increments p by one, setting it to $p = p_i + 2$. It next reads $x(n-6)$ from location $p = p_i + 2$, multiplies it by a_6, and adds the result to the accumulator (this explains the name "accumulator" for the register, since it accumulates the products in the tapped delay line). It continues until it reads $x(n)$ from location $p = p_i + 8$, which because of the modulo operation is the same location that the latest input $x(n)$ was written to, and multiplies that value by a_0. It again increments p, getting $p = p_i + 9 = p_i + 1$. Hence, at the conclusion of this operation, the value of p is $p_i + 1$, which gives the location into which the next input $x(n+1)$ should be written.

on distinct hardware (such as on multicore machines, on servers in a server farm, or on distinct microprocessors).

Non-concurrent programs specify a *sequence* of instructions to execute. A programming language that expresses a computation as a sequence of operations is called an **imperative** language. C is an imperative language. When using C to write concurrent programs, we must step outside the language itself, typically using a **thread library**. A thread library uses facilities provided not by C, but rather provided by the operating system and/or the hardware. Java is a mostly imperative language extended with constructs that directly support threads. Thus, one can write concurrent programs in Java without stepping outside the language.

Every (correct) execution of a program in an imperative language must behave as if the instructions were executed exactly in the specified sequence. It is often possible, however, to execute instructions in parallel or in an order different from that specified by the program and still get behavior that matches what would have happened had they been executed in sequence.

Example 8.5: Consider the following C statements:

```
double pi, piSquared, piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = pi * pi * pi;
```

The last two assignment statements are independent, and hence can be executed in parallel or in reverse order without changing the behavior of the program. Had we written them as follows, however, they would no longer be independent:

```
double pi, piSquared, piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = piSquared * pi;
```

In this case, the last statement depends on the third statement in the sense that the third statement must complete execution before the last statement starts.

A compiler may analyze the dependencies between operations in a program and produce parallel code, if the target machine supports it. This analysis is called **dataflow analysis**. Many microprocessors today support parallel execution, using multi-issue instruction streams or VLIW (very large instruction word) architectures. Processors with multi-issue instruction streams can execute independent instructions simultaneously. The hardware analyzes instructions on-the-fly for dependencies, and when there is no dependency, executes more than one instruction at a time. In the latter, VLIW machines have assembly-level instructions that specify multiple operations to be performed together. In this case, the compiler is usually required to produce the appropriate parallel instructions. In these cases, the dependency analysis is done at the level of assembly language or at the level of individual operations, not at the level of lines of C. A line of C may specify multiple operations, or even complex operations like procedure calls. In both cases (multi-issue and VLIW), an imperative program is analyzed for concurrency in order to enable parallel execution. The overall objective is to speed up execution of the program. The goal is improved **performance**, where the presumption is that finishing a task earlier is always better than finishing it later.

In the context of embedded systems, however, concurrency plays a part that is much more central than merely improving performance. Embedded programs interact with physical processes, and in the physical world, many activities progress at the same time. An embedded program often needs to monitor and react to multiple concurrent sources of stimulus, and simultaneously control multiple output devices that affect the physical world. Embedded programs are almost always concurrent programs, and concurrency is an intrinsic part of the logic of the programs. It is not just a way to get improved performance. Indeed, finishing a task earlier is not necessarily better than finishing it later. *Timeliness* matters, of course; actions performed in the physical world often need to be done at the *right time* (neither early nor late). Picture for example an engine controller for a gasoline engine. Firing the spark plugs earlier is most certainly not better than firing them later. They must be fired at the *right* time.

Just as imperative programs can be executed sequentially or in parallel, concurrent programs can be executed sequentially or in parallel. Sequential execution of a concurrent program is done typically today by a **multitasking operating system**, which interleaves the execution of multiple tasks in a single sequential stream of instructions. Of course, the hardware may parallelize that execution if the processor has a multi-issue or VLIW architecture. Hence, a concurrent program may be converted to a sequential stream by an operating system and back to concurrent program by the hardware, where the latter translation is done to improve performance. These multiple translations greatly complicate

the problem of ensuring that things occur at the *right* time. This problem is addressed in Chapter 12.

Parallelism in the hardware, the main subject of this chapter, exists to improve performance for computation-intensive applications. From the programmer's perspective, concurrency arises as a consequence of the hardware designed to improve performance, not as a consequence of the application problem being solved. In other words, the application does not (necessarily) demand that multiple activities proceed simultaneously, it just demands that things be done very quickly. Of course, many interesting applications will combine both forms of concurrency, arising from parallelism and from application requirements.

The sorts of algorithms found in compute-intensive embedded programs has a profound affect on the design of the hardware. In this section, we focus on hardware approaches that deliver parallelism, namely pipelining, instruction-level parallelism, and multicore architectures. All have a strong influence on the programming models for embedded software. In Chapter 9, we give an overview of memory systems, which strongly influence how parallelism is handled.

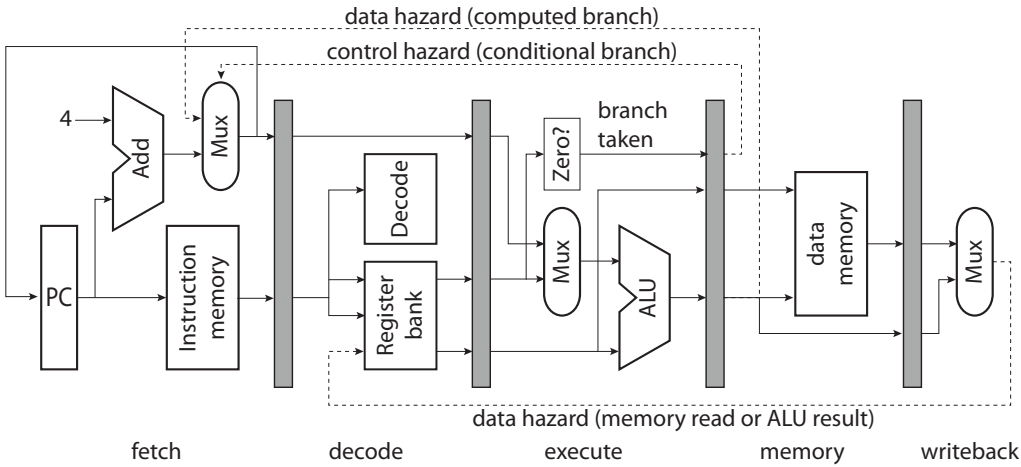


Figure 8.2: Simple pipeline (after Patterson and Hennessy (1996)).

8.2.2 Pipelining

Most modern processors are **pipelined**. A simple five-stage pipeline for a 32-bit machine is shown in Figure 8.2. In the figure, the shaded rectangles are latches, which are clocked at processor clock rate. On each edge of the clock, the value at the input is stored in the latch register. The output is then held constant until the next edge of the clock, allowing the circuits between the latches to settle. This diagram can be viewed as a synchronous-reactive model of the behavior of the processor.

In the fetch (leftmost) stage of the pipeline, a **program counter** (**PC**) provides an address to the instruction memory. The instruction memory provides encoded instructions, which in the figure are assumed to be 32 bits wide. In the fetch stage, the PC is incremented by 4 (bytes), to become the address of the next instruction, unless a conditional branch instruction is providing an entirely new address for the PC. The decode pipeline stage extracts register addresses from the 32-bit instruction and fetches the data in the specified registers from the register bank. The execute pipeline stage operates on the data fetched from the registers or on the PC (for a computed branch) using an **arithmetic logic unit** (**ALU**), which performs arithmetic and logical operations. The memory pipeline stage reads or writes to a memory location given by a register. The writeback pipeline stage stores results in the register file.

DSP processors normally add an extra stage or two that performs a multiplication, provide separate ALUs for address calculation, and provide a dual data memory for simultaneous access to two operands (this latter design is known as a Harvard architecture). But the simple version without the separate ALUs suffices to illustrate the issues that an embedded system designer faces.

The portions of the pipeline between the latches operate in parallel. Hence, we can see immediately that there are simultaneously five instructions being executed, each at a different stage of execution. This is easily visualized with a **reservation table** like that in Figure 8.3. The table shows hardware resources that may be simultaneously used on the left. In this case, the register bank appears three times because the pipeline of Figure 8.2 assumes that two reads and write of the register file can occur in each cycle.

The reservation table in Figure 8.3 shows a sequence A, B, C, D, E of instructions in a program. In cycle 5, E is being fetched while D is reading from the register bank, while C is using the ALU, while B is reading from or writing to data memory, while A is writing results to the register bank. The write by A occurs in cycle 5, but the read by B occurs in cycle 3. Thus, the value that B reads will not be the value that A writes.

This phenomenon is known as a **data hazard**, one form of **pipeline hazard**. Pipeline hazards are caused by the dashed lines in Figure 8.2. Programmers normally expect that if instruction A is before instruction B, then any results computed by A will be available to B, so this behavior may not be acceptable.

Computer architects have tackled the problem of pipeline hazards in a variety of ways. The simplest technique is known as an **explicit pipeline**. In this technique, the pipeline hazard is simply documented, and the programmer (or compiler) must deal with it. For the example where B reads a register written by A, the compiler may insert three **no-op** instructions (which do nothing) between A and B to ensure that the write occurs before the read. These no-op instructions form a **pipeline bubble** that propagates down the pipeline.

A more elaborate technique is to provide **interlock**s. In this technique, the instruction decode hardware, upon encountering instruction B that reads a register written by A, will detect the hazard and delay the execution of B until A has completed the writeback stage. For this pipeline, B should be delayed by three clock cycles to permit A to complete, as shown in Figure 8.4. This can be reduced to two cycles if slightly more complex **forwarding** logic is provided, which detects that A is writing the same location that B is reading, and directly provides the data rather than requiring the write to occur before the read. Interlocks therefore provide hardware that automatically inserts pipeline bubbles.

A still more elaborate technique is **out-of-order execution**, where hardware is provided that detects a hazard, but instead of simply delaying execution of B, proceeds to fetch C, and if C does not read registers written by either A or B, and does not write registers read

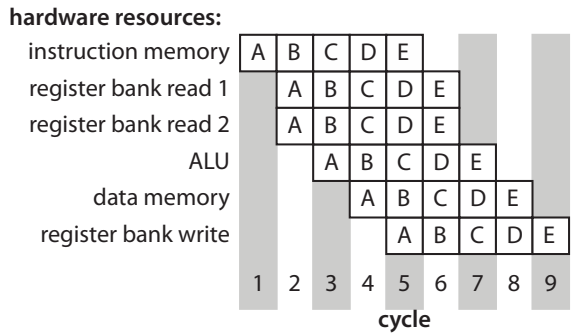


Figure 8.3: Reservation table for the pipeline shown in Figure 8.2.

by B, then proceeds to execute C before B. This further reduces the number of pipeline bubbles.

Another form of pipeline hazard illustrated in Figure 8.2 is a **control hazard**. In the figure, a conditional branch instruction changes the value of the PC if a specified register has value zero. The new value of the PC is provided (optionally) by the result of an ALU operation. In this case, if A is a conditional branch instruction, then A has to have reached the memory stage before the PC can be updated. The instructions that follow A in memory will have been fetched and will be at the decode and execute stages already by the time it is determined that those instructions should not in fact be executed.

Like data hazards, there are multiple techniques for dealing with control hazards. A **delayed branch** simply documents the fact that the branch will be taken some number of cycles after it is encountered, and leaves it up to the programmer (or compiler) to ensure that the instructions that follow the conditional branch instruction are either harmless (like no-ops) or do useful work that does not depend on whether the branch is taken. An interlock provides hardware to insert pipeline bubbles as needed, just as with data hazards. In the most elaborate technique, **speculative execution**, hardware estimates whether the branch is likely to be taken, and begins executing the instructions it expects to execute. If its expectation is not met, then it undoes any side effects (such as register writes) that the speculatively executed instructions caused.

Except for explicit pipelines and delayed branches, all of these techniques introduce variability in the timing of execution of an instruction sequence. Analysis of the timing of

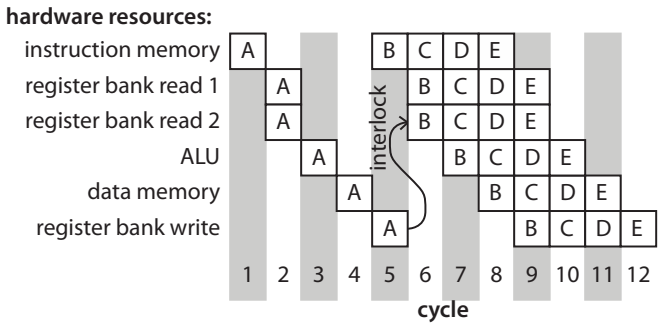


Figure 8.4: Reservation table for the pipeline shown in Figure 8.2 with interlocks, assuming that instruction B reads a register that is written by instruction A.

a program can become extremely difficult when there is a deep pipeline with elaborate forwarding and speculation. Explicit pipelines are relatively common in DSP processors, which are often applied in contexts where precise timing is essential. Out-of-order and speculative execution are common in general-purpose processors, where timing matters only in an aggregate sense. An embedded system designer needs to understand the requirements of the application and avoid processors where the requisite level of timing precision is unachievable.

8.2.3 Instruction-Level Parallelism

Achieving high performance demands parallelism in the hardware. Such parallelism can take two broad forms, multicore architectures, described later in Section 8.2.4, or **instruction-level parallelism** (**ILP**), which is the subject of this section. A processor supporting ILP is able to perform multiple independent operations in each instruction cycle. We discuss four major forms of ILP: CISC instructions, subword parallelism, superscalar, and VLIW.

CISC Instructions

A processor with complex (and typically, rather specialized) instructions is called a **CISC** machine (**complex instruction set computer**). The philosophy behind such processors is distinctly different from that of **RISC** machines (**reduced instruction set computers**) (Patterson and Ditzel, 1980). DSPs are typically CISC machines, and include instructions specifically supporting FIR filtering (and often other algorithms such as FFTs (fast Fourier transforms) and Viterbi decoding). In fact, to qualify as a DSP, a processor must be able to perform FIR filtering in one instruction cycle per tap.

Example 8.6: The Texas Instruments TMS320c54x family of DSP processors is intended to be used in power-constrained embedded applications that demand high signal processing performance, such as wireless communication systems and personal digital assistants (**PDA**s). The inner loop of an FIR computation (8.1) is

```
1 RPT numberOfTaps - 1
2 MAC *AR2+, *AR3+, A
```

The first instruction illustrates the **zero-overhead loop**s commonly found in DSPs. The instruction that comes after it will execute a number of times equal to one plus the argument of the `RPT` instruction. The `MAC` instruction is a **multiply-accumulate instruction**, also prevalent in DSP architectures. It has three arguments specifying the following calculation,

$$a := a + x * y \ ,$$

where a is the contents of an accumulator register named `A`, and x and y are values found in memory. The addresses of these values are contained by auxiliary registers `AR2` and `AR3`. These registers are incremented automatically after the access. Moreover, these registers can be set up to implement circular buffers, as described in the box on page 221. The c54x processor includes a section of on-chip memory that supports two accesses in a single cycle, and as long as the addresses refer to this section of the memory, the `MAC` instruction will execute in a single cycle. Thus, each cycle, the processor performs two memory fetches, one multiplication, one ordinary addition, and two (possibly modulo) address increments. All DSPs have similar capabilities.

CISC instructions can get quite esoteric.

Example 8.7: The coefficients of the FIR filter in (8.1) are often symmetric, meaning that N is even and

$$a_i = a_{N-i-1} \ .$$

The reason for this is that such filters have linear phase (intuitively, this means that symmetric input signals result in symmetric output signals, or that all frequency components are delayed by the same amount). In this case, we can reduce the number of multiplications by rewriting (8.1) as

$$y(n) = \sum_{i=0}^{(N/2)-1} a_i(x(n-i) + x(n-N+i+1)) \ .$$

The Texas Instruments TMS320c54x instruction set includes a `FIRS` instruction that functions similarly to the `MAC` in Example 8.6, but using this calculation

rather than that of (8.1). This takes advantage of the fact that the c54x has two ALUs, and hence can do twice as many additions as multiplications. The time to execute an FIR filter now reduces to 1/2 cycle per tap.

CISC instruction sets have their disadvantages. For one, it is extremely challenging (perhaps impossible) for a compiler to make optimal use of such an instruction set. As a consequence, DSP processors are commonly used with code libraries written and optimized in assembly language.

In addition, CISC instruction sets can have subtle timing issues that can interfere with achieving hard real-time scheduling. In the above examples, the layout of data in memory strongly affects execution times. Even more subtle, the use of zero-overhead loops (the `RPT` instruction above) can introduce some subtle problems. On the TI c54x, interrupts are disabled during repeated execution of the instruction following the `RPT`. This can result in unexpectedly long latencies in responding to interrupts.

Subword Parallelism

Many embedded applications operate on data types that are considerably smaller than the word size of the processor.

Example 8.8: In Examples 8.3 and 8.4, the data types are typically 8-bit integers, each representing a color intensity. The color of a pixel may be represented by three bytes in the RGB format. Each of the RGB bytes has a value ranging from 0 to 255 representing the intensity of the corresponding color. It would be wasteful of resources to use, say, a 64-bit ALU to process a single 8-bit number.

To support such data types, some processors support **subword parallelism**, where a wide ALU is divided into narrower slices enabling simultaneous arithmetic or logical operations on smaller words.

Example 8.9: Intel introduced subword parallelism into the widely used general purpose Pentium processor and called the technology MMX (Eden and Kagan, 1997). MMX instructions divide the 64-bit datapath into slices as small as 8 bits, supporting simultaneous identical operations on multiple bytes of image pixel data. The technology has been used to enhance the performance of image manipulation applications as well as applications supporting video streaming. Similar techniques were introduced by Sun Microsystems for Sparc™ processors (Tremblay et al., 1996) and by Hewlett Packard for the PA RISC processor (Lee, 1996). Many processor architectures designed for embedded applications, including many DSP processors, also support subword parallelism.

A **vector processor** is one where the instruction set includes operations on multiple data elements simultaneously. Subword parallelism is a particular form of vector processing.

Superscalar

Superscalar processors use fairly conventional sequential instruction sets, but the hardware can simultaneously dispatch multiple instructions to distinct hardware units when it detects that such simultaneous dispatch will not change the behavior of the program. That is, the execution of the program is identical to what it would have been if it had been executed in sequence. Such processors even support out-of-order execution, where instructions later in the stream are executed before earlier instructions. Superscalar processors have a significant disadvantage for embedded systems, which is that execution times may be extremely difficult to predict, and in the context of multitasking (interrupts and threads), may not even be repeatable. The execution times may be very sensitive to the exact timing of interrupts, in that small variations in such timing may have big effects on the execution times of programs.

VLIW

Processors intended for embedded applications often use VLIW architectures instead of superscalar in order to get more repeatable and predictable timing. **VLIW (very large in-**

struction word) processors include multiple function units, like superscalar processors, but instead of dynamically determining which instructions can be executed simultaneously, each instruction specifies what each function unit should do in a particular cycle. That is, a VLIW instruction set combines multiple independent operations into a single instruction. Like superscalar architectures, these multiple operations are executed simultaneously on distinct hardware. Unlike superscalar, however, the order and simultaneity of the execution is fixed in the program rather than being decided on-the-fly. It is up to the programmer (working at assembly language level) or the compiler to ensure that the simultaneous operations are indeed independent. In exchange for this additional complexity in programming, execution times become repeatable and (often) predictable.

Example 8.10: In Example 8.7, we saw the specialized instruction `FIRS` of the c54x architecture that specifies operations for two ALUs and one multiplier. This can be thought of as a primitive form of VLIW, but subsequent generations of processors are much more explicit about their VLIW nature. The Texas Instruments TMS320c55x, the next generation beyond the c54x, includes two multiply-accumulate units, and can support instructions that look like this:

```
1 MAC      *AR2+, *CDP+, AC0
2 :: MAC   *AR3+, *CDP+, AC1
```

Here, `AC0` and `AC1` are two accumulator registers and `CDP` is a specialized register for pointing to filter coefficients. The notation `::` means that these two instructions should be issued and executed in the same cycle. It is up to the programmer or compiler to determine whether these instructions can in fact be executed simultaneously. Assuming the memory addresses are such that the fetches can occur simultaneously, these two `MAC` instructions execute in a single cycle, effectively dividing in half the time required to execute an FIR filter.

For applications demanding higher performance still, VLIW architectures can get quite elaborate.

Example 8.11: The Texas Instruments c6000 family of processors have a VLIW instruction set. Included in this family are three subfamilies of processors, the

c62x and c64x fixed-point processors and the c67x floating-point processors. These processors are designed for use in wireless infrastructure (such as cellular base stations and adaptive antennas), telecommunications infrastructure (such as voice over IP and video conferencing), and imaging applications (such as medical imaging, surveillance, machine vision or inspection, and radar).

Example 8.12: The **TriMedia** processor family, from NXP, is aimed at digital television, and can perform operations like that in (8.2) very efficiently. NXP Semiconductors used to be part of Philips, a diversified consumer electronics company that, among many other products, makes flat-screen TVs. The strategy in the TriMedia architecture is to make it easier for a compiler to generate efficient code, reducing the need for assembly-level programming (though it includes specialized CISC instructions that are difficult for a compiler to exploit). It makes things easier for the compiler by having a larger register set than is typical (128 registers), a RISC-like instruction set, where several instructions can be issued simultaneously, and hardware supporting IEEE 754 floating point operations.

8.2.4 Multicore Architectures

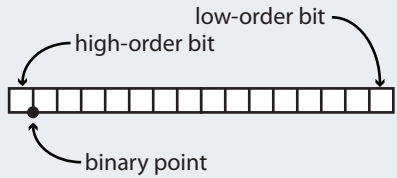
A **multicore** machine is a combination of several processors on a single chip. Although multicore machines have existed since the early 1990s, they have only recently penetrated into general-purpose computing. This penetration accounts for much of the interest in them today. **Heterogeneous multicore** machines combine a variety of processor types on a single chip, vs. multiple instances of the same processor type.

Example 8.13: Texas Instruments **OMAP** (open multimedia application platform) architectures are widely used in cell phones, which normally combine one or more DSP processors with one or more processors that are closer in style to

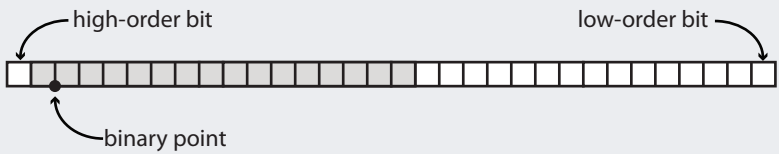
general-purpose processors. The DSP processors handle the radio, speech, and media processing (audio, images, and video). The other processors handle the user interface, database functions, networking, and downloadable applications. Specifically, the OMAP4440 includes a 1 GHz dual-core ARM Cortex processor, a c64x DSP, a GPU, and an image signal processor.

Fixed-Point Numbers

Many embedded processors provide hardware for integer arithmetic only. Integer arithmetic, however, can be used for non-whole numbers, with some care. Given, say, a 16-bit integer, a programmer can *imagine* a **binary point**, which is like a decimal point, except that it separates bits rather than digits of the number. For example, a 16-bit integer can be used to represent numbers in the range -1.0 to 1.0 (roughly) by placing a (conceptual) binary point just below the high-order bit of the number, as shown below:



Without the binary point, a number represented by the 16 bits is a whole number $x \in \{-2^{15}, \cdots, 2^{15}-1\}$ (assuming the twos-complement binary representation, which has become nearly universal for signed integers). With the binary point, we *interpret* the 16 bits to represent a number $y = x/2^{15}$. Hence, y ranges from -1 to $1-2^{-15}$. This is known as a **fixed-point number**. The format of this fixed-point number can be written 1.15, indicating that there is one bit to the left of the binary point and 15 to the right. When two such numbers are multiplied at full precision, the result is a 32-bit number. The binary point is located as follows:

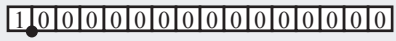


... **Continued on page 235.**

Fixed-Point Numbers (continued)

The location of the binary point follows from the **law of conservation of bits**. When multiplying two numbers with formats $n.m$ and $p.q$, the result has format $(n+p).(m+q)$. Processors often support such full-precision multiplications, where the result goes into an accumulator register that has at least twice as many bits as the ordinary data registers. To write the result back to a data register, however, we have to extract 16 bits from the 32 bit result. If we extract the shaded bits on page 235, then we preserve the position of the binary point, and the result still represents a number roughly in the range -1 to 1.

There is a loss of information, however, when we extract 16 bits from a 32-bit result. First, there is a possibility of **overflow**, because we are discarding the high-order bit. Suppose the two numbers being multiplied are both -1, which has binary representation in twos complement as follows:



When these two number are multiplied, the result has the following bit pattern:

which in twos complement, represents 1, the correct result. However, when we extract the shaded 16 bits, the result is now -1! Indeed, 1 is not representable in the fixed-point format 1.15, so overflow has occurred. Programmers must guard against this, for example by ensuring that all numbers are strictly less than 1 in magnitude, prohibiting -1.

A second problem is that when we extract the shaded 16 bits from a 32-bit result, we discard 15 low-order bits. There is a loss of information here. If we simply discard the low-order 15 bits, the strategy is known as **truncation**. If instead we first add the following bit pattern the 32-bit result, then the result is known as **rounding**:

Rounding chooses the result that is closest to the full-precision result, while truncation chooses the closest result that is smaller in magnitude.

DSP processors typically perform the above extraction with either rounding or truncation in hardware when data is moved from an accumulator to a general-purpose register or to memory.

For embedded applications, multicore architectures have a significant potential advantage over single-core architectures because real-time and safety-critical tasks can have a dedicated processor. This is the reason for the heterogeneous architectures used for cell phones, since the radio and speech processing functions are hard real-time functions with considerable computational load. In such architectures, user applications cannot interfere with real-time functions.

This lack of interference is more problematic in general-purpose multicore architectures. It is common, for example, to use multi-level caches, where the second or higher level cache is shared among the cores. Unfortunately, such sharing makes it very difficult to isolate the real-time behavior of the programs on separate cores, since each program can trigger cache misses in another core. Such multi-level caches are not suitable for real-time applications.

A very different type of multicore architecture that is sometimes used in embedded applications uses one or more **soft core**s together with custom hardware on a **field-programmable gate array** (**FPGA**). FPGAs are chips whose hardware function is programmable using hardware design tools. Soft cores are processors implemented on FPGAs. The advantage of soft cores is that they can be tightly coupled to custom hardware more easily than off-the-shelf processors.

8.3 Summary

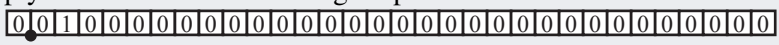
The choice of processor architecture for an embedded system has important consequences for the programmer. Programmers may need to use assembly language to take advantage of esoteric architectural features. For applications that require precise timing, it may be difficult to control the timing of a program because of techniques in the hardware for dealing with pipeline hazards and parallel resources.

Fixed-Point Arithmetic in C

Most C programmers will use `float` or `double` data types when performing arithmetic on non-whole numbers. However, many embedded processors lack hardware for floating-point arithmetic. Thus, C programs that use the `float` or `double` data types often result in unacceptably slow execution, since floating point must be emulated in software. Programmers are forced to use integer arithmetic to implement operations on numbers that are not whole numbers. How can they do that?

First, a programmer can *interpret* a 32-bit `int` differently from the standard representation, using the notion of a binary point, explained in the boxes on pages 234 and 235. However, when a C program specifies that two `int`s be multiplied, the result is an `int`, not the full precision 64-bit result that we need. In fact, the strategy outlined on page 234, of putting one bit to the left of the binary point and extracting the shaded bits from the result, will not work, because most of the shaded bits will be missing from the result. For example, suppose we want to multiply 0.5 by 0.5. This number can be represented in 32-bit `int`s as follows:

Without the binary point (which is invisible to C and to the hardware, residing only in the programmer's mind), this bit pattern represents the integer 2^{30}, a large number indeed. When multiplying these two numbers, the result is 2^{60}, which is not representable in an `int`. Typical processors will set an overflow bit in the processor status register (which the programmer must check) and deliver as a result the number 0, which is the low-order 32 bits of the product. To guard against this, a programmer can shift each 32 bit integer to the right by 16 bits before multiplying. In that case, the result of the multiply 0.5×0.5 is the following bit pattern:



With the binary point as shown, this result is interpreted as 0.25, the correct answer. Of course, shifting data to the right by 16 bits discards the 16 low-order bits in the `int`. There is a loss of precision that amounts to truncation. The programmer may wish to round instead, adding the `int` 2^{15} to the numbers before shifting to the right 16 times. Floating-point data types make things easier. The hardware (or software) keeps track of the amount of shifting required and preserves precision when possible. However, not all embedded processors with floating-point hardware conform with the **IEEE 754** standard. This can complicate the design process for the programmer, because numerical results will not match those produced by a desktop computer.

Exercises

1. Consider the reservation table in Figure 8.4. Suppose that the processor includes forwarding logic that is able to tell that instruction A is writing to the same register that instruction B is reading from, and that therefore the result written by A can be forwarded directly to the ALU before the write is done. Assume the forwarding logic itself takes no time. Give the revised reservation table. How many cycles are lost to the pipeline bubble?

2. Consider the following instruction, discussed in Example 8.6:

```
1 MAC *AR2+, *AR3+, A
```

   Suppose the processor has three ALUs, one for each arithmetic operation on the addresses contained in registers `AR2` and `AR3` and one to perform the addition in the `MAC` multiply-accumulate instruction. Assume these ALUs each require one clock cycle to execute. Assume that a multiplier also requires one clock cycle to execute. Assume further that the register bank supports two reads and two writes per cycle, and that the accumulator register `A` can be written separately and takes no time to write. Give a reservation table showing the execution of a sequence of such instructions.

3. Assuming fixed-point numbers with format 1.15 as described in the boxes on pages 234 and 235, show that the *only* two numbers that cause overflow when multiplied are -1 and -1. That is, if either number is anything other than -1 in the 1.15 format, then extracting the 16 shaded bits in the boxes does not result in overflow.