Embedded System Design: A Unified Hardware/Software Approach

Frank Vahid and Tony Givargis

Department of Computer Science and Engineering
University of California
Riverside, CA 92521
vahid@cs.ucr.edu
http://www.cs.ucr.edu/~vahid

Draft version, Fall 1999

Preface

This book introduces embedded system design using a modern approach. Modern design requires a designer to have a unified view of software and hardware, seeing them not as completely different domains, but rather as two implementation options along a continuum of options varying in their design metrics (cost, performance, power, flexibility, etc.).

Three important trends have made such a unified view possible. First, integrated circuit (IC) capacities have increased to the point that both software processors and custom hardware processors now commonly coexist on a single IC. Second, quality-compiler availability and average program sizes have increased to the point that C compilers (and even C++ or in some cases Java) have become commonplace in embedded systems. Third, synthesis technology has advanced to the point that synthesis tools have become commonplace in the design of digital hardware. Such tools achieve nearly the same for hardware design as compilers achieve in software design: they allow the designer to describe desired processing in a high-level programming language, and they then automatically generate an efficient (in this case custom-hardware) processor implementation. The first trend makes the past separation of software and hardware design nearly impossible. Fortunately, the second and third trends enable their unified design, by turning embedded system design, at its highest level, into the problem of selecting (for software), designing (for hardware), and integrating processors.

ESD focuses on design principles, breaking from the traditional book that focuses on the details a particular microprocessor and its assembly-language programming. While stressing a processor-independent high-level language approach to programming of embedded systems, it still covers enough assembly language programming to enable programming of device drivers. Such processor-independence is possible because of compiler availability, as well as the fact that integrated development environments (IDE's) now commonly support a variety of processor targets, making such independence even more attractive to instructors as well as designers. However, these developments don't entirely eliminate the need for some processor-specific knowledge. Thus, a course with a hands-on lab may supplement this book with a processor-specific databook and/or a compiler manual (both are typically very low cost or even free), or one of many commonly available "extended databook" processor- specific textbooks.

ESD describes not only the programming of microprocessors, but also the design of custom-hardware processors (i.e., digital design). Coverage of this topic is made possible by the above-mentioned elimination of a detailed processor architecture study. While other books often have a review of digital design techniques, ESD uses the new top-down approach to custom-hardware design, describing simple steps for converting high-level program code into digital hardware. These steps build on the trend of digital design books of introducing synthesis into an undergraduate curriculum (e.g., books by Roth, Gajski, and Katz). This book assists designers to become users of synthesis. Using a draft of ESD, we have at UCR successfully taught both programming of embedded microprocessors, design of custom-hardware processors, and integration of the two, in a one-quarter course having a lab, though a semester or even two quarters would be

preferred. However, instructors who do not wish to focus on synthesis will find that the top-down approach covered still provides the important unified view of hardware and software.

ESD includes coverage of some additional important topics. First, while the need for knowledge specific to a microprocessor's internals is decreasing, the need for knowledge of interfacing processors is increasing. Therefore, ESD not only includes a chapter on interfacing, but also includes another chapter describing interfacing protocols common in embedded systems, like CAN, I2C, ISA, PCI, and Firewire. Second, while high-level programming languages greatly improve our ability to describe complex behavior, several widely accepted computation models can improve that ability even further. Thus, ESD includes chapters on advanced computation models, including state machines and their extensions (including Statecharts), and concurrent programming models. Third, an extremely common subset of embedded systems is control systems. ESD includes a chapter that introduces control systems in a manner that enables the reader to recognize open and closed-loop control systems, to use simple PID and fuzzy controllers, and to be aware that a rich theory exists that can be drawn upon for design of such systems. Finally, ESD includes a chapter on design methodology, including discussion of hardware/software codesign, a user's introduction to synthesis (from behavioral down to logic levels), and the major trend towards Intellectual Property (IP) based design.

Additional materials: A web page will be established to be used in conjunction with the book. A set of slides will be available for lecture presentations. Also available for use with the book will be a simulatable and synthesizable VHDL "reference design," consisting of a simple version of a MIPS processor, memory, BIOS, DMA controller, UART, parallel port, and an input device (currently a CCD preprocessor), and optionally a cache, two-level bus architecture, a bus bridge, and an 8051 microcontroller. We have already developed a version of this reference design at UCR. This design can be used in labs that have the ability to simulate and/or synthesize VHDL descriptions. There are numerous possible uses depending on the course focus, ranging from simulation to see first-hand how various components work in a system (e.g., DMA, interrupt processing, arbitration, etc.), to synthesis of working FPGA system prototypes.

Instructors will likely want to have a prototyping environment consisting of a microprocessor development board and/or in-circuit emulator, and perhaps an FPGA development board. These environments vary tremendously among universities. However, we will make the details of our environments and lab projects available on the web page. Again, these have already been developed.

Chapter 1 *Introduction*

1.1 Embedded systems overview

Computing systems are everywhere. It's probably no surprise that millions of computing systems are built every year destined for desktop computers (Personal Computers, or PC's), workstations, mainframes and servers. What may be surprising is that *billions* of computing systems are built every year for a very different purpose: they are embedded within larger electronic devices, repeatedly carrying out a particular function, often going completely unrecognized by the device's user. Creating a precise definition of such embedded computing systems, or simply *embedded systems*, is not an easy task. We might try the following definition: An embedded system is nearly any computing system other than a desktop, laptop, or mainframe computer. That definition isn't perfect, but it may be as close as we'll get. We can better understand such systems by examining common examples and common characteristics. Such examination will reveal major challenges facing designers of such systems.

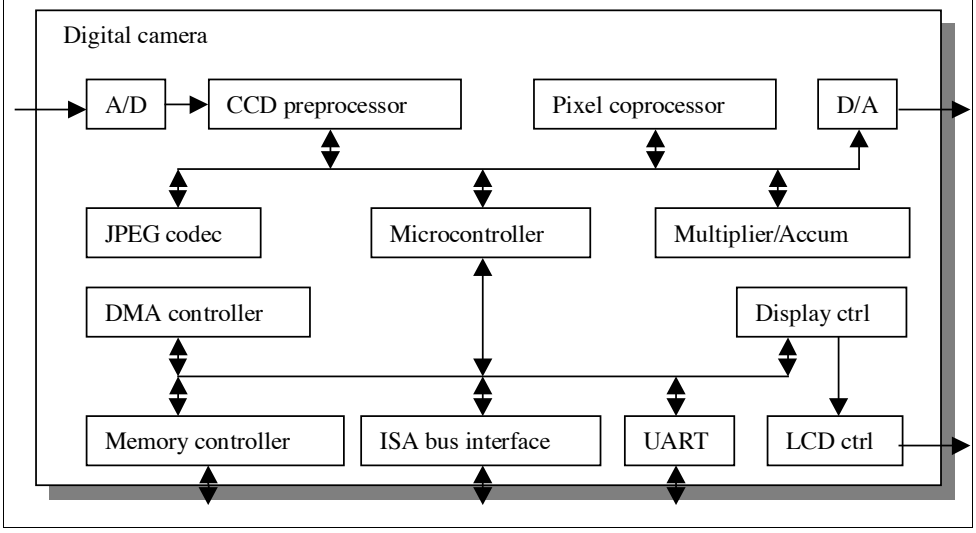
Embedded systems are found in a variety of common electronic devices, such as: (a) consumer electronics -- cell phones, pagers, digital cameras, camcorders, videocassette recorders, portable video games, calculators, and personal digital assistants; (b) home appliances -- microwave ovens, answering machines, thermostat, home security, washing machines, and lighting systems; (c) office automation -- fax machines, copiers, printers, and scanners; (d) business equipment -- cash registers, curbside check-in, alarm systems, card readers, product scanners, and automated teller machines; (e) automobiles -- transmission control, cruise control, fuel injection, anti-lock brakes, and active suspension. One might say that nearly any device that runs on electricity either already has, or will soon have, a computing system embedded within it. While about 40% of American households had a desktop computer in 1994, each household had an average of more than 30 embedded computers, with that number expected to rise into the hundreds by the year 2000. The electronics in an average car cost $1237 in 1995, and may cost $2125 by 2000. Several billion embedded microprocessor units were sold annually in recent years, compared to a few hundred million desktop microprocessor units.

Embedded systems have several common characteristics:

1) *Single-functioned:* An embedded system usually executes only one program, repeatedly. For example, a pager is always a pager. In contrast, a desktop system executes a variety of programs, like spreadsheets, word processors, and video games, with new programs added frequently.[1]
2) *Tightly constrained:* All computing systems have constraints on design metrics, but those on embedded systems can be especially tight. A design metric is a measure of an implementation's features, such as cost, size, performance, and power. Embedded systems often must cost just a few dollars, must be sized to fit on a single chip, must perform fast enough to process data in real-time, and must consume minimum power to extend battery life or prevent the necessity of a cooling fan.

There are some exceptions. One is the case where an embedded system's program is updated with a newer program version. For example, some cell phones can be updated in such a manner. A second is the case where several programs are swapped in and out of a system due to size limitations. For example, some missiles run one program while in cruise mode, then load a second program for locking onto a target.

Figure 1.1: An embedded system example -- a digital camera.



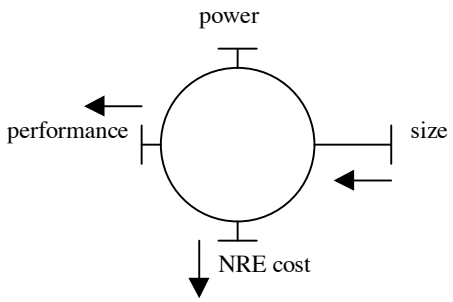
3) *Reactive and real-time:* Many embedded systems must continually react to changes in the system's environment, and must compute certain results in real time without delay. For example, a car's cruise controller continually monitors and reacts to speed and brake sensors. It must compute acceleration or decelerations amounts repeatedly within a limited time; a delayed computation result could result in a failure to maintain control of the car. In contrast, a desktop system typically focuses on computations, with relatively infrequent (from the computer's perspective) reactions to input devices. In addition, a delay in those computations, while perhaps inconvenient to the computer user, typically does not result in a system failure.

For example, consider the digital camera system shown in Figure 1.1. The *A2D* and *D2A* circuits convert analog images to digital and digital to analog, respectively. *The CCD preprocessor* is a charge-coupled device preprocessor. The *JPEG codec* compresses and decompresses an image using the JPEG[2] compression standard, enabling compact storage in the limited memory of the camera. The *Pixel coprocessor* aids in rapidly displaying images. The *Memory controller* controls access to a memory chip also found in the camera, while the *DMA controller* enables direct memory access without requiring the use of the microcontroller. The *UART* enables communication with a PC's serial port for uploading video frames, while the *ISA bus interface* enables a faster connection with a PC's ISA bus. The *LCD ctrl* and *Display ctrl* circuits control the display of images on the camera's liquid-crystal display device. A *Multiplier/Accum* circuit assists with certain digital signal processing. At the heart of the system is a *microcontroller*, which is a processor that controls the activities of all the other circuits. We can think of each device as a processor designed for a particular task, while the microcontroller is a more general processor designed for general tasks.

This example illustrates some of the embedded system characteristics described above. First, it performs a single function repeatedly. The system always acts as a digital camera, wherein it captures, compresses and stores frames, decompresses and displays frames, and uploads frames. Second, it is tightly constrained. The system must be low cost since consumers must be able to afford such a camera. It must be small so that it fits within a standard-sized camera. It must be fast so that it can process numerous images in milliseconds. It must consume little power so that the camera's battery will last a long

[2] JPEG is short for the Joint Photographic Experts Group. The 'joint' refers to its status as a committee working on both ISO and ITU-T standards. Their best known standard is for still image compression.

Figure 1.2: Design metric competition -- decreasing one may increase others.



time. However, this particular system does not posses a high degree of the characteristic of being reactive and real-time, as it only needs to respond to the pressing of buttons by a user, which even for an avid photographer is still quite slow with respect to processor speeds.

1.2 Design challenge – optimizing design metrics

The embedded-system designer must of course construct an implementation that fulfills desired functionality, but a difficult challenge is to construct an implementation that simultaneously optimizes numerous design metrics. For our purposes, an implementation consists of a software processor with an accompanying program, a connection of digital gates, or some combination thereof. A design metric is a measurable feature of a system's implementation. Common relevant metrics include:

- *Unit cost:* the monetary cost of manufacturing each copy of the system, excluding NRE cost.
- *NRE cost (Non-Recurring Engineering cost):* The monetary cost of designing the system. Once the system is designed, any number of units can be manufactured without incurring any additional design cost (hence the term "non-recurring").
- *Size:* the physical space required by the system, often measured in bytes for software, and gates or transistors for hardware.
- *Performance:* the execution time or throughput of the system.
- *Power:* the amount of power consumed by the system, which determines the lifetime of a battery, or the cooling requirements of the IC, since more power means more heat.
- *Flexibility:* the ability to change the functionality of the system without incurring heavy NRE cost. Software is typically considered very flexible.
- *Time-to-market:* The amount of time required to design and manufacture the system to the point the system can be sold to customers.
- *Time-to-prototype:* The amount of time to build a working version of the system, which may be bigger or more expensive than the final system implementation, but can be used to verify the system's usefulness and correctness and to refine the system's functionality.
- *Correctness:* our confidence that we have implemented the system's functionality correctly. We can check the functionality throughout the process of designing the system, and we can insert test circuitry to check that manufacturing was correct.
- *Safety:* the probability that the system will not cause harm.
- Many others.

These metrics typically compete with one another: improving one often leads to a degradation in another. For example, if we reduce an implementation's size, its performance may suffer. Some observers have compared this phenomenon to a wheel with numerous pins, as illustrated in Figure Figure 1.2. If you push one pin (say size) in, the others pop out. To best meet this optimization challenge, the designer must be

Figure 1.3: Market window.

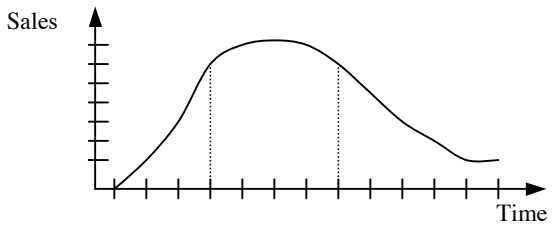
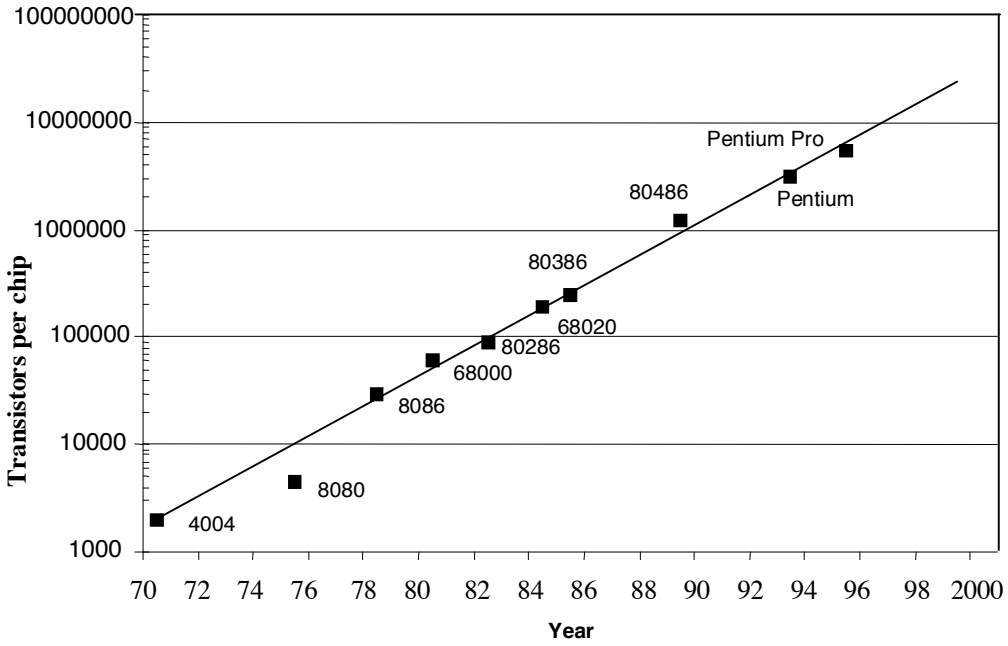


Figure 1.4: IC capacity exponential increase.



comfortable with a variety of hardware and software implementation technologies, and must be able to migrate from one technology to another, in order to find the best implementation for a given application and constraints. Thus, a designer cannot simply be a hardware expert or a software expert, as is commonly the case today; the designer must be an expert in both areas.

Most of these metrics are heavily constrained in an embedded system. The time-to-market constraint has become especially demanding in recent years. Introducing an embedded system to the marketplace early can make a big difference in the system's profitability, since market time-windows for products are becoming quite short, often measured in months. For example, Figure 1.3 shows a sample market window providing during which time the product would have highest sales. Missing this window (meaning the product begins being sold further to the right on the time scale) can mean significant loss in sales. In some cases, each day that a product is delayed from introduction to the market can translate to a one million dollar loss. Adding to the difficulty of meeting the time-to-market constraint is the fact that embedded system complexities are growing due to increasing IC capacities. IC capacity, measured in transistors per chip, has grown

exponentially over the past 25 years[3], as illustrated in Figure 1.4; for reference purposes, we've included the density of several well-known processors in the figure. However, the rate at which designers can produce transistors has not kept up with this increase, resulting in a widening gap, according to the Semiconductor Industry Association. Thus, a designer must be familiar with the state-of-the-art design technologies in both hardware and software design to be able to build today's embedded systems.

We can define *technology* as a manner of accomplishing a task, especially using technical processes, methods, or knowledge. This textbook focuses on providing an overview of three technologies central to embedded system design: processor technologies, IC technologies, and design technologies. We describe all three briefly here, and provide further details in subsequent chapters.

1.3 Embedded processor technology

Processor technology involves the architecture of the computation engine used to implement a system's desired functionality. While the term "processor" is usually associated with programmable software processors, we can think of many other, non-programmable, digital systems as being processors also. Each such processor differs in its specialization towards a particular application (like a digital camera application), thus manifesting different design metrics. We illustrate this concept graphically in Figure 1.5. The application requires a specific embedded functionality, represented as a cross, such as the summing of the items in an array, as shown in Figure 1.5(a). Several types of processors can implement this functionality, each of which we now describe. We often use a collection of such processors to best optimize our system's design metrics, as was the case in our digital camera example.

1.3.1 General-purpose processors -- software

The designer of a *general-purpose* processor builds a device suitable for a variety of applications, to maximize the number of devices sold. One feature of such a processor is a program memory – the designer does not know what program will run on the processor, so cannot build the program into the digital circuit. Another feature is a general datapath – the datapath must be general enough to handle a variety of computations, so typically has a large register file and one or more general-purpose arithmetic-logic units (ALUs). An embedded system designer, however, need not be concerned about the design of a general-purpose processor. An embedded system designer simply uses a general-purpose processor, by programming the processor's memory to carry out the required functionality. Many people refer to this portion of an implementation simply as the "software" portion.

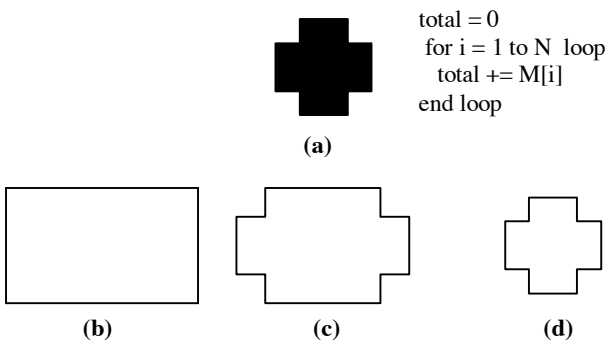
Using a general-purpose processor in an embedded system may result in several design-metric benefits. *Design time* and *NRE cost* are low, because the designer must only write a program, but need not do any digital design. *Flexibility* is high, because changing functionality requires only changing the program. *Unit cost* may be relatively low in small quantities, since the processor manufacturer sells large quantities to other customers and hence distributes the NRE cost over many units. *Performance* may be fast for computation-intensive applications, if using a fast processor, due to advanced architecture features and leading edge IC technology.

However, there are also some design-metric drawbacks. *Unit cost* may be too high for large quantities. *Performance* may be slow for certain applications. *Size* and *power* may be large due to unnecessary processor hardware.

For example, we can use a general-purpose processor to carry out our array-summing functionality from the earlier example. Figure 1.5(b) illustrates that a general-

[3] Gordon Moore, co-founder of Intel, predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18-24 months. His very accurate prediction is known as "Moore's Law." He recently predicted about another decade before such growth slows down.

Figure 1.5: Processors very in their customization for the problem at hand: (a) desired functionality, (b) general-purpose processor, (b) application-specific processor, (c) single-purpose processor.



purpose covers the desired functionality, but not necessarily efficiently. Figure 1.6(a) shows a simple architecture of a general-purpose processor implementing the array-summing functionality. The functionality is stored in a program memory. The controller fetches the current instruction, as indicated by the program counter (PC), into the instruction register (IR). It then configures the datapath for this instruction and executes the instruction. Finally, it determines the appropriate next instruction address, sets the PC to this address, and fetches again.

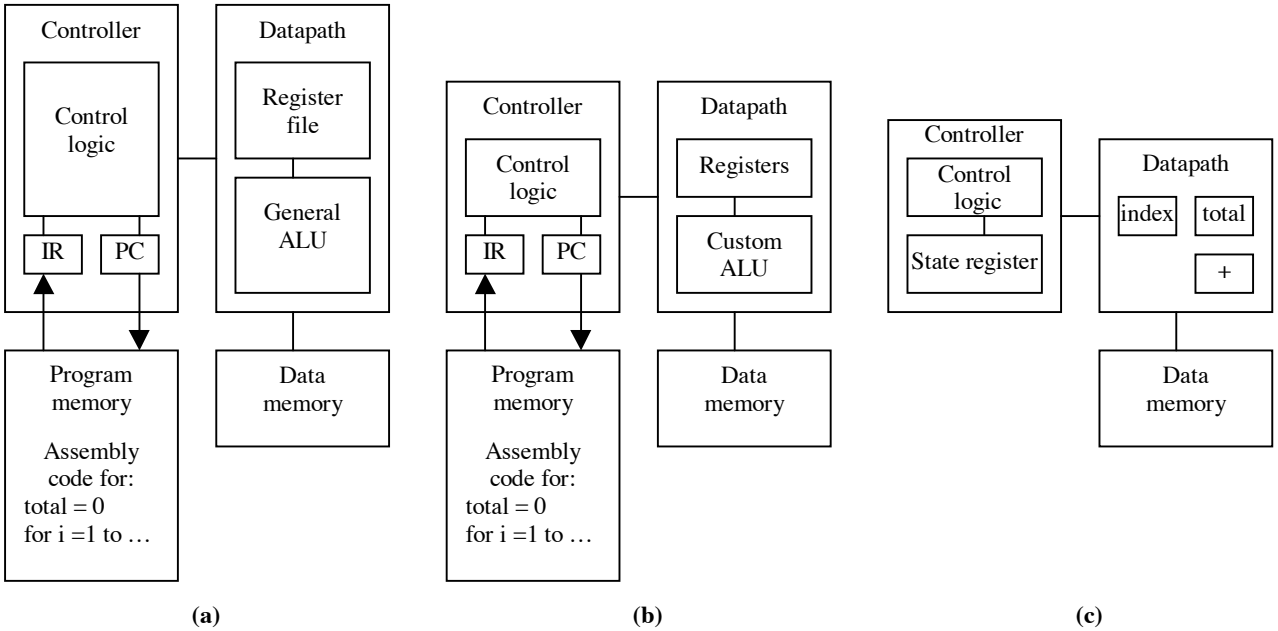
1.3.2 Single-purpose processors -- hardware

A *single-purpose* processor is a digital circuit designed to execute exactly one program. For example, consider the digital camera example of Figure 1.1. All of the components other than the microcontroller are single-purpose processors. The JPEG codec, for example, executes a single program that compresses and decompresses video frames. An embedded system designer creates a single-purpose processor by designing a custom digital circuit, as discussed in later chapters. Many people refer to this portion of the implementation simply as the "hardware" portion (although even software requires a hardware processor on which to run). Other common terms include coprocessor and accelerator.

Using a single-purpose processor in an embedded system results in several design-metric benefits and drawbacks, which are essentially the inverse of those for general-purpose processors. Performance may be fast, size and power may be small, and unit-cost may be low for large quantities, while design time and NRE costs may be high, flexibility is low, unit cost may be high for small quantities, and performance may not match general-purpose processors for some applications.

For example, Figure 1.5(d) illustrates the use of a single-purpose processor in our embedded system example, representing an exact fit of the desired functionality, nothing more, nothing less. Figure 1.6(c) illustrates the architecture of such a single-purpose processor for the example. Since the example counts from one to N, we add an *index* register. The index register will be loaded with N, and will then count down to zero, at which time it will assert a status line read by the controller. Since the example has only one other value, we add only one register labeled *total* to the datapath. Since the example's only arithmetic operation is addition, we add a single adder to the datapath. Since the processor only executes this one program, we hardwire the program directly into the control logic.

Figure 1.6: Implementing desired functionality on different processor types: (a) general-purpose, (b) application-specific, (c) single-purpose.



1.3.3 Application-specific processors

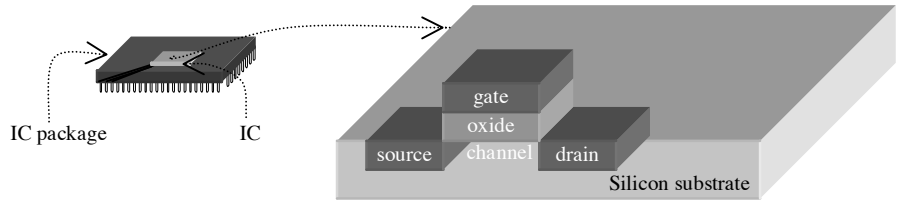
An *application-specific* instruction-set processor (or ASIP) can serve as a compromise between the above processor options. An ASIP is designed for a particular class of applications with common characteristics, such as digital-signal processing, telecommunications, embedded control, etc. The designer of such a processor can optimize the datapath for the application class, perhaps adding special functional units for common operations, and eliminating other infrequently used units.

Using an ASIP in an embedded system can provide the benefit of flexibility while still achieving good performance, power and size. However, such processors can require large NRE cost to build the processor itself, and to build a compiler, if these items don't already exist. Much research currently focuses on automatically generating such processors and associated retargetable compilers. Due to the lack of retargetable compilers that can exploit the unique features of a particular ASIP, designers using ASIPs often write much of the software in assembly language.

Digital-signal processors (DSPs) are a common class of ASIP, so demand special mention. A DSP is a processor designed to perform common operations on digital signals, which are the digital encodings of analog signals like video and audio. These operations carry out common signal processing tasks like signal filtering, transformation, or combination. Such operations are usually math-intensive, including operations like multiply and add or shift and add. To support such operations, a DSP may have special-purpose datapath components such a multiply-accumulate unit, which can perform a computation like T = T + M[i]*k using only one instruction. Because DSP programs often manipulate large arrays of data, a DSP may also include special hardware to fetch sequential data memory locations in parallel with other operations, to further speed execution.

Figure 1.5(c) illustrates the use of an ASIP for our example; while partially customized to the desired functionality, there is some inefficiency since the processor also contains features to support reprogramming. Figure 1.6(b) shows the general architecture of an ASIP for the example. The datapath may be customized for the example. It may have an auto-incrementing register, a path that allows the add of a

Figure 1.7: IC's consist of several layers. Shown is a simplified CMOS transistor; an IC may possess millions of these, connected by layers of metal (not shown).



register plus a memory location in one instruction, fewer registers, and a simpler controller. We do not elaborate further on ASIPs in this book (the interested reader will find references at the end of this chapter).

1.4 IC technology

Every processor must eventually be implemented on an IC. IC technology involves the manner in which we map a digital (gate-level) implementation onto an IC. An IC (Integrated Circuit), often called a "chip," is a semiconductor device consisting of a set of connected transistors and other devices. A number of different processes exist to build semiconductors, the most popular of which is CMOS (Complementary Metal Oxide Semiconductor). The IC technologies differ by how customized the IC is for a particular implementation. For lack of a better term, we call these technologies "IC technologies." IC technology is independent from processor technology; any type of processor can be mapped to any type of IC technology, as illustrated in Figure 1.8.

To understand the differences among IC technologies, we must first recognize that semiconductors consist of numerous layers. The bottom layers form the transistors. The middle layers form logic gates. The top layers connect these gates with wires. One way to create these layers is by depositing photo-sensitive chemicals on the chip surface and then shining light through *masks* to change regions of the chemicals. Thus, the task of building the layers is actually one of designing appropriate masks. A set of masks is often called a *layout*. The narrowest line that we can create on a chip is called the *feature size,* which today is well below one micrometer (sub-micron). For each IC technology, all layers must eventually be built to get a working IC; the question is who builds each layer and when.

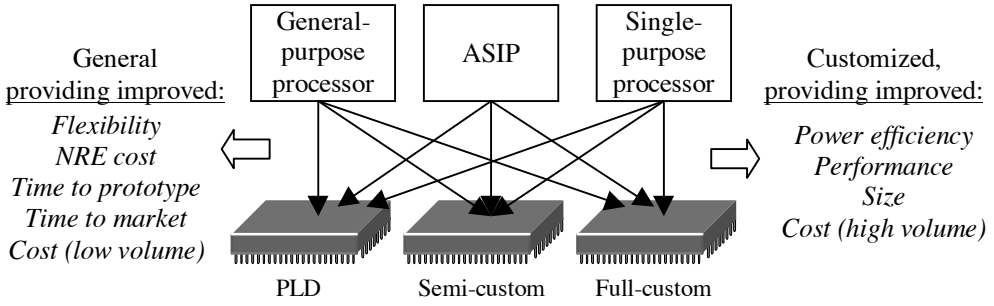
1.4.1 Full-custom/VLSI

In a full-custom IC technology, we optimize all layers for our particular embedded system's digital implementation. Such optimization includes placing the transistors to minimize interconnection lengths, sizing the transistors to optimize signal transmissions and routing wires among the transistors. Once we complete all the masks, we send the mask specifications to a fabrication plant that builds the actual ICs. Full-custom IC design, often referred to as VLSI (Very Large Scale Integration) design, has very high NRE cost and long turnaround times (typically months) before the IC becomes available, but can yield excellent performance with small size and power. It is usually used only in high-volume or extremely performance-critical applications.

1.4.2 Semi-custom ASIC (gate array and standard cell)

In an ASIC (Application-Specific IC) technology, the lower layers are fully or partially built, leaving us to finish the upper layers. In a gate array technology, the masks for the transistor and gate levels are already built (i.e., the IC already consists of arrays of gates). The remaining task is to connect these gates to achieve our particular implementation. In a standard cell technology, logic-level cells (such as an AND gate or an AND-OR-INVERT combination) have their mask portions pre-designed, usually by

Figure 1.8: The independence of processor and IC technologies: any processor technology can be mapped to any IC technology.



hand. Thus, the remaining task is to arrange these portions into complete masks for the gate level, and then to connect the cells. ASICs are by far the most popular IC technology, as they provide for good performance and size, with much less NRE cost than full-custom IC's.

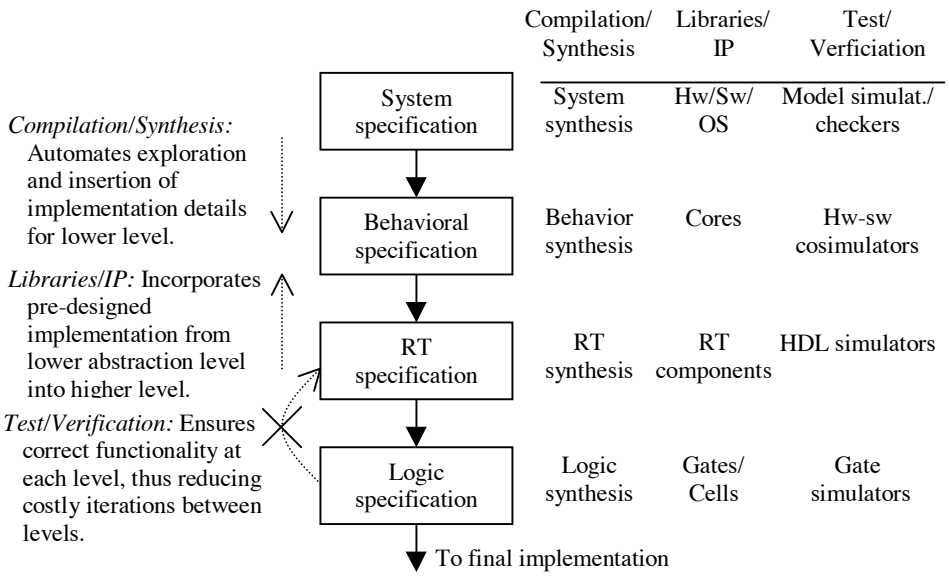
1.4.3 PLD

In a PLD (Programmable Logic Device) technology, all layers already exist, so we can purchase the actual IC. The layers implement a programmable circuit, where programming has a lower-level meaning than a software program. The programming that takes place may consist of creating or destroying connections between wires that connect gates, either by blowing a fuse, or setting a bit in a programmable switch. Small devices, called programmers, connected to a desktop computer can typically perform such programming. We can divide PLD's into two types, simple and complex. One type of simple PLD is a PLA (Programmable Logic Array), which consists of a programmable array of AND gates and a programmable array of OR gates. Another type is a PAL (Programmable Array Logic), which uses just one programmable array to reduce the number of expensive programmable components. One type of complex PLD, growing very rapidly in popularity over the past decade, is the FPGA (Field Programmable Gate Array), which offers more general connectivity among blocks of logic, rather than just arrays of logic as with PLAs and PALs, and are thus able to implement far more complex designs. PLDs offer very low NRE cost and almost instant IC availability. However, they are typically bigger than ASICs, may have higher unit cost, may consume more power, and may be slower (especially FPGAs). They still provide reasonable performance, though, so are especially well suited to rapid prototyping.

As mentioned earlier and illustrated in Figure 1.8, the choice of an IC technology is independent of processor types. For example, a general-purpose processor can be implemented on a PLD, semi-custom, or full-custom IC. In fact, a company marketing a commercial general-purpose processor might first market a semi-custom implementation to reach the market early, and then later introduce a full-custom implementation. They might also first map the processor to an older but more reliable technology, like 0.2 micron, and then later map it to a newer technology, like 0.08 micron. These two evolutions of mappings to a large extent explain why a processor's clock speed improves on the market over time.

Furthermore, we often implement multiple processors of different types on the same IC. Figure 1.1 was an example of just such a situation – the digital camera included a microcontroller (general-purpose processor) plus numerous single-purpose processors on the same IC.

Figure 1.9: Ideal top-down design process, and productivity improvers.



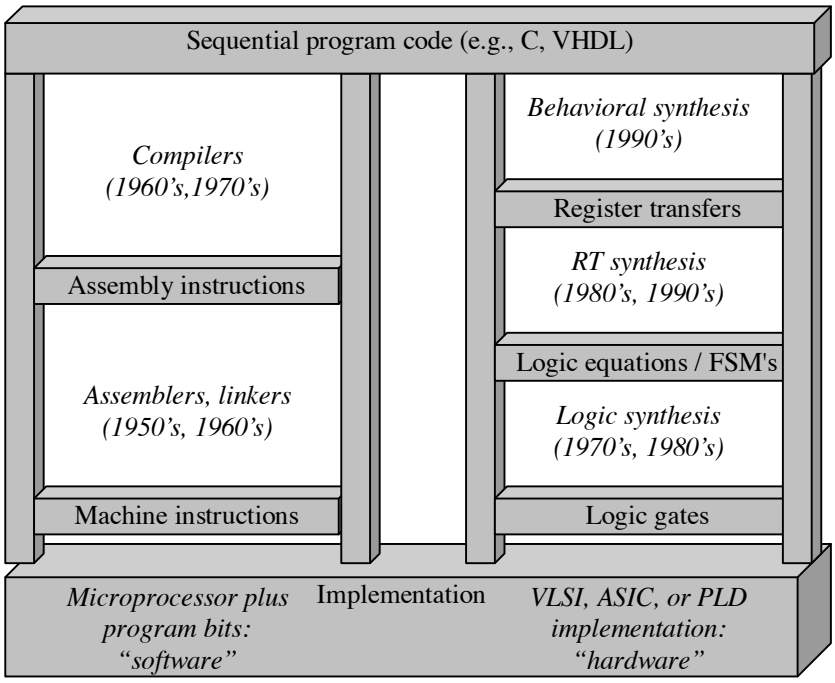
1.5 Design technology

Design technology involves the manner in which we convert our concept of desired system functionality into an implementation. We must not only design the implementation to optimize design metrics, but we must do so quickly. As described earlier, the designer must be able to produce larger numbers of transistors every year, to keep pace with IC technology. Hence, improving design technology to enhance productivity has been a focus of the software and hardware design communities for decades.

To understand how to improve the design process, we must first understand the design process itself. Variations of a *top-down* design process have become popular in the past decade, an ideal form of which is illustrated in Figure 1.9. The designer refines the system through several abstraction levels. At the system level, the designer describes the desired functionality in some language, often a natural language like English, but preferably an executable language like C; we shall call this the *system specification*. The designer refines this specification by distributing portions of it among chosen processors (general or single purpose), yielding *behavioral specifications* for each processor. The designer refines these specifications into *register-transfer (RT) specifications* by converting behavior on general-purpose processors to assembly code, and by converting behavior on single-purpose processors to a connection of register-transfer components and state machines. The designer then refines the register-transfer-level specification of a single-purpose processor into a *logic specification* consisting of Boolean equations. Finally, the designer refines the remaining specifications into an implementation, consisting of machine code for general-purpose processors, and a gate-level netlist for single-purpose processors.

There are three main approaches to improving the design process for increased productivity, which we label as compilation/synthesis, libraries/IP, and test/verification. Several other approaches also exist. We now discuss all of these approaches. Each approach can be applied at any of the four abstraction levels.

Figure 1.10: The co-design ladder: recent maturation of synthesis enables a unified view of hardware and software.



1.5.1 Compilation/Synthesis

Compilation/Synthesis lets a designer specify desired functionality in an abstract manner, and automatically generates lower-level implementation details. Describing a system at high abstraction levels can improve productivity by reducing the amount of details, often by an order of magnitude, that a design must specify.

A logic synthesis tool converts Boolean expressions into a connection of logic gates (called a netlist). A register-transfer (RT) synthesis tool converts finite-state machines and register-transfers into a datapath of RT components and a controller of Boolean equations. A behavioral synthesis tool converts a sequential program into finite-state machines and register transfers. Likewise, a software compiler converts a sequential program to assembly code, which is essentially register-transfer code. Finally, a system synthesis tool converts an abstract system specification into a set of sequential programs on general and single-purpose processors.

The relatively recent maturation of RT and behavioral synthesis tools has enabled a unified view of the design process for single-purpose and general-purpose processors. Design for the former is commonly known as "hardware design," and design for the latter as "software design." In the past, the design processes were radically different – software designers wrote sequential programs, while hardware designers connected components. But today, synthesis tools have converted the hardware design process essentially into one of writing sequential programs (albeit with some knowledge of how the hardware will be synthesized). We can think of abstraction levels as being the rungs of a ladder, and compilation and synthesis as enabling us to step up the ladder and hence enabling designers to focus their design efforts at higher levels of abstraction, as illustrated in Figure 1.10. Thus, the starting point for either hardware or software is sequential programs, enhancing the view that system functionality can be implemented in hardware, software, or some combination thereof. The choice of hardware versus software for a particular function is simply a tradeoff among various design metrics, like performance, power, size, NRE cost, and especially flexibility; there is no fundamental difference

between what the two can implement. *Hardware/software codesign* is the field that emphasizes this unified view, and develops synthesis tools and simulators that enable the co-development of systems using both hardware and software.

1.5.2 Libraries/IP

Libraries involve re-use of pre-existing implementations. Using libraries of existing implementations can improve productivity if the time it takes to find, acquire, integrate and test a library item is less than that of designing the item oneself.

A logic-level library may consist of layouts for gates and cells. An RT-level library may consist of layouts for RT components, like registers, multiplexors, decoders, and functional units. A behavioral-level library may consist of commonly used components, such as compression components, bus interfaces, display controllers, and even general-purpose processors. The advent of system-level integration has caused a great change in this level of library. Rather than these components being IC's, they now must also be available in a form, called *cores*, that we can implement on just one portion of an IC. This change from behavioral-level libraries of IC's to libraries of cores has prompted use of the term *Intellectual Property (IP)*, to emphasize the fact that cores exist in a "soft" form that must be protected from copying. Finally, a system-level library might consist of complete systems solving particular problems, such as an interconnection of processors with accompanying operating systems and programs to implement an interface to the Internet over an Ethernet network.

1.5.3 Test/Verification

Test/Verification involves ensuring that functionality is correct. Such assurance can prevent time-consuming debugging at low abstraction levels and iterating back to high abstraction levels.

Simulation is the most common method of testing for correct functionality, although more formal verification techniques are growing in popularity. At the logic level, gate-level simulators provide output signal timing waveforms given input signal waveforms. Likewise, general-purpose processor simulators execute machine code. At the RT-level, hardware description language (HDL) simulators execute RT-level descriptions and provide output waveforms given input waveforms. At the behavioral level, HDL simulators simulate sequential programs, and co-simulators connect HDL and general-purpose processor simulators to enable hardware/software co-verification. At the system level, a model simulator simulates the initial system specification using an abstract computation model, independent of any processor technology, to verify correctness and completeness of the specification. Model checkers can also verify certain properties of the specification, such as ensuring that certain simultaneous conditions never occur, or that the system does not deadlock.

1.5.4 Other productivity improvers

There are numerous additional approaches to improving designer productivity. *Standards* focus on developing well-defined methods for specification, synthesis and libraries. Such standards can reduce the problems that arise when a designer uses multiple tools, or retrieves or provides design information from or to other designers. Common standards include language standards, synthesis standards and library standards.

Languages focus on capturing desired functionality with minimum designer effort. For example, the sequential programming language of C is giving way to the object-oriented language of C++, which in turn has given some ground to Java. As another example, state-machine languages permit direct capture of functionality as a set of states and transitions, which can then be translated to other languages like C.

Frameworks provide a software environment for the application of numerous tools throughout the design process and management of versions of implementations. For example, a framework might generate the UNIX directories needed for various simulators

and synthesis tools, supporting application of those tools through menu selections in a single graphical user interface.

1.6 Summary and book outline

Embedded systems are large in numbers, and those numbers are growing every year as more electronic devices gain a computational element. Embedded systems possess several common characteristics that differentiate them from desktop systems, and that pose several challenges to designers of such systems. The key challenge is to optimize design metrics, which is particularly difficult since those metrics compete with one another. One particularly difficult design metric to optimize is time-to-market, because embedded systems are growing in complexity at a tremendous rate, and the rate at which productivity improves every year is not keeping up with that growth. This book seeks to help improve productivity by describing design techniques that are standard and others that are very new, and by presenting a unified view of software and hardware design. This goal is worked towards by presenting three key technologies for embedded systems design: processor technology, IC technology, and design technology. Processor technology is divided into general-purpose, application-specific, and single-purpose processors. IC technology is divided into custom, semi-custom, and programmable logic IC's. Design technology is divided into compilation/synthesis, libraries/IP, and test/verification.

This book focuses on processor technology (both hardware and software), with the last couple of chapters covering IC and design technologies. Chapter 2 covers general-purpose processors. We focus on programming of such processors using structured programming languages, touching on assembly language for use in driver routines; we assume the reader already has familiarity with programming in both types of languages. Chapter 3 covers single-purpose processors, describing a number of common peripherals used in embedded systems. Chapter 4 describes digital design techniques for building custom single-purpose processors. Chapter 5 describes memories, components necessary to store data for processors. Chapters 6 and 7 describe buses, components necessary to communicate data among processors and memories, with Chapter 6 introducing concepts, and Chapter 7 describing common buses. Chapters 8 and 9 describe advanced techniques for programming embedded systems, with Chapter 8 focusing on state machines, and Chapter 9 providing an introduction to real-time programming. Chapter 10 introduces a very common form of embedded system, called control systems. Chapter 11 provides an overview of IC technologies, enough for a designer to understand what options are available and what tradeoffs exist. Chapter 12 focuses on design methodology, emphasizing the need for a "new breed" of engineers for embedded systems, proficient with both software and hardware design.

1.7 References and further reading

[1] Semiconductor Industry Association, National Technology Roadmap for Semiconductors, 1997.

1.8 Exercises

1. Consider the following embedded systems: a pager, a computer printer, and an automobile cruise controller. Create a table with each example as a column, and each row one of the following design metrics: unit cost, performance, size, and power. For each table entry, explain whether the constraint on the design metric is very tight. Indicate in the performance entry whether the system is highly reactive or not.
2. List three pairs of design metrics that may compete, providing an intuitive explanation of the reason behind the competition.
3. The design of a particular disk drive has an NRE cost of $100,000 and a unit cost of $20. How much will we have to add to the cost of the product to cover our NRE cost, assuming we sell: (a) 100 units, and (b) 10,000 units.

4. (a) Create a general equation for product cost as a function of unit cost, NRE cost, and number of units, assuming we distribute NRE cost equally among units. (b) Create a graph with the x-axis the number of units and the y-axis the product cost, and then plot the product cost function for an NRE of $50,000 and a unit cost of $5.
5. Redraw Figure 1.4 to show the transistors per IC from 1990 to 2000 on a linear, not logarithmic, scale. Draw a square representing a 1990 IC and another representing a 2000 IC, with correct relative proportions.
6. Create a plot with the three processor technologies on the x-axis, and the three IC technologies on the y-axis. For each axis, put the most programmable form closest to the origin, and the most customized form at the end of the axis. Plot the 9 points, and explain features and possible occasions for using each.
7. Give an example of a recent consumer product whose prime market window was only about one year.

Chapter 2 *General-purpose processors: Software*

2.1 Introduction

A general-purpose processor is a programmable digital system intended to solve computation tasks in a large variety of applications. Copies of the same processor may solve computation problems in applications as diverse as communication, automotive, and industrial embedded systems. An embedded system designer choosing to use a general-purpose processor to implement part of a system's functionality may achieve several benefits.

First, the unit cost of the processor may be very low, often a few dollars or less. One reason for this low cost is that the processor manufacturer can spread its NRE cost for the processor's design over large numbers of units, often numbering in the millions or billions. For example, Motorola sold nearly half a billion 68HC05 microcontrollers *in 1996 alone* (source: Motorola 1996 Annual Report).

Second, because the processor manufacturer can spread NRE cost over large numbers of units, the manufacturer can afford to invest large NRE cost into the processor's design, without significantly increasing the unit cost. The processor manufacturer may thus use experienced computer architects who incorporate advanced architectural features, and may use leading-edge optimization techniques, state-of-the-art IC technology, and handcrafted VLSI layouts for critical components. These factors can improve design metrics like performance, size and power.

Third, the embedded system designer may incur low NRE cost, since the designer need only write software, and then apply a compiler and/or an assembler, both of which are mature and low-cost technologies. Likewise, time-to-prototype will be short, since processor IC's can be purchased and then programmed in the designer's own lab. Flexibility will be high, since the designer can perform software rewrites in a straightforward manner.

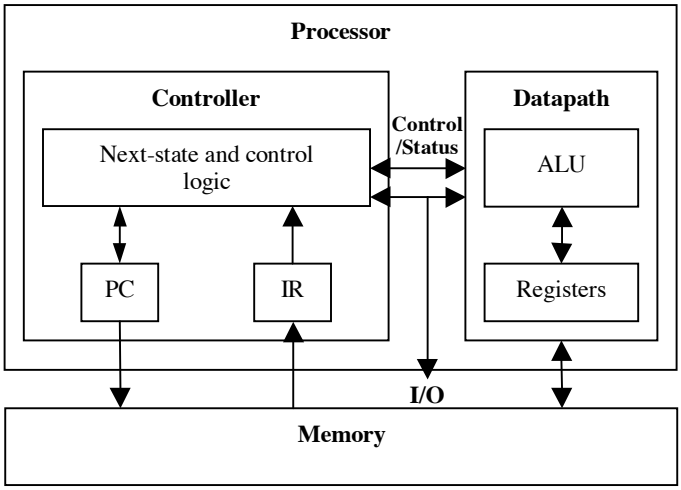
2.2 Basic architecture

A general-purpose processor, sometimes called a CPU (Central Processing Unit) or a microprocessor, consists of a datapath and a controller, tightly linked with a memory. We now discuss these components briefly. Figure 2.1 illustrates the basic architecture.

2.2.1 Datapath

The datapath consists of the circuitry for transforming data and for storing temporary data. The datapath contains an arithmetic-logic unit (ALU) capable of transforming data through operations such as addition, subtraction, logical AND, logical OR, inverting, and shifting. The ALU also generates status signals, often stored in a status register (not shown), indicating particular data conditions. Such conditions include indicating whether data is zero, or whether an addition of two data items generates a carry. The datapath also contains registers capable of storing temporary data. Temporary data may include data brought in from memory but not yet sent through the ALU, data coming from the ALU that will be needed for later ALU operations or will be sent back to memory, and data that must be moved from one memory location to another. The internal data bus is the bus over which data travels within the datapath, while the external data bus is the bus over which data is brought to and from the data memory.

Figure 2.1: General-purpose processor basic architecture.



We typically distinguish processors by their size, and we usually measure size as the bit-width of the datapath components. A *bit*, which stands for binary digit, is the processor's basic data unit, representing either a 0 (low or false) or a 1 (high or true), while we refer to 8 bits as a *byte*. An *N-bit* processor may have N-bit wide registers, an N-bit wide ALU, an N-bit wide internal bus over which data moves among datapath components, and an N-bit wide external bus over which data is brought in and out of the datapath. Common processor sizes include 4-bit, 8-bit, 16-bit, 32-bit and 64-bit processors. However, in some cases, a particular processor may have different sizes among its registers, ALU, internal bus, or external bus, so the processor-size definition is not an exact one. For example, a processor may have a 16-bit internal bus, ALU and registers, but only an 8-bit external bus to reduce pins on the processor's IC.

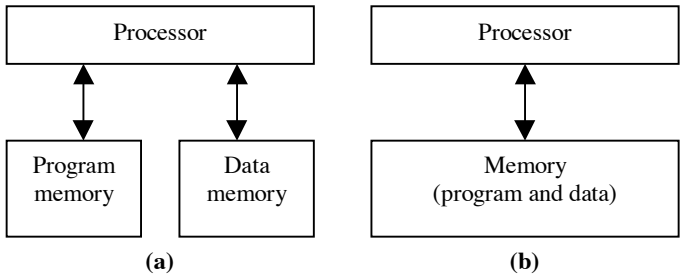
2.2.2 Controller

The controller consists of circuitry for retrieving program instructions, and for moving data to, from, and through the datapath according to those instructions. The controller contains a program counter (PC) that holds the address in memory of the next program instruction to fetch. The controller also contains an instruction register (IR) to hold the fetched instruction. Based on this instruction, the controller's control logic generates the appropriate signals to control the flow of data in the datapath. Such flows may include inputting two particular registers into the ALU, storing ALU results into a particular register, or moving data between memory and a register. Finally, the next-state logic determines the next value of the PC. For a non-branch instruction, this logic increments the PC. For a branch instruction, this logic looks at the datapath status signals and the IR to determine the appropriate next address.

The PC's bit-width represents the processor's address size. The address size is independent of the data word size; the address size is often larger. The address size determines the number of directly accessible memory locations, referred to as the *address space* or *memory space*. If the address size is M, then the address space is 2^M. Thus, a processor with a 16-bit PC can directly address 2^{16} = 65,536 memory locations. We would typically refer to this address space as 64K, although if 1K = 1000, this number would represent 64,000, not the actual 65,536. Thus, in computer-speak, 1K = 1024.

For each instruction, the controller typically sequences through several stages, such as fetching the instruction from memory, decoding it, fetching operands, executing the instruction in the datapath, and storing results. Each stage may consist of one or more clock cycles. A clock cycle is usually the longest time required for data to travel from one

Figure 2.2: Two memory architectures: (a) Harvard, (b) Princeton.



register to another. The path through the datapath or controller that results in this longest time (e.g., from a datapath register through the ALU and back to a datapath register) is called the critical path. The inverse of the clock cycle is the clock frequency, measured in cycles per second, or Hertz (Hz). For example, a clock cycle of 10 nanoseconds corresponds to a frequency of $1/10x10^{-9}$ Hz, or 100 MHz. The shorter the critical path, the higher the clock frequency. We often use clock frequency as one means of comparing processors, especially different versions of the same processor, with higher clock frequency implying faster program execution (though this isn't always true).

2.2.3 Memory

While registers serve a processor's short term storage requirements, memory serves the processor's medium and long-term information-storage requirements. We can classify stored information as either program or data. Program information consists of the sequence of instructions that cause the processor to carry out the desired system functionality. Data information represents the values being input, output and transformed by the program.

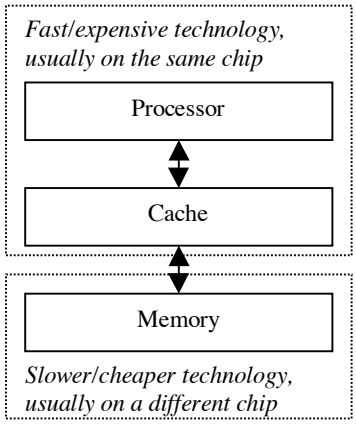
We can store program and data together or separately. In a *Princeton architecture*, data and program words share the same memory space. In a *Harvard architecture*, the program memory space is distinct from the data memory space. Figure 2.2 illustrates these two methods. The Princeton architecture may result in a simpler hardware connection to memory, since only one connection is necessary. A Harvard architecture, while requiring two connections, can perform instruction and data fetches simultaneously, so may result in improved performance. Most machines have a Princeton architecture. The Intel 8051 is a well-known Harvard architecture.

Memory may be read-only memory (ROM) or readable and writable memory (RAM). ROM is usually much more compact than RAM. An embedded system often uses ROM for program memory, since, unlike in desktop systems, an embedded system's program does not change. Constant-data may be stored in ROM, but other data of course requires RAM.

Memory may be on-chip or off-chip. On-chip memory resides on the same IC as the processor, while off-chip memory resides on a separate IC. The processor can usually access on-chip memory must faster than off-chip memory, perhaps in just one cycle, but finite IC capacity of course implies only a limited amount of on-chip memory.

To reduce the time needed to access (read or write) memory, a local copy of a portion of memory may be kept in a small but especially fast memory called *cache*, as illustrated in Figure 2.3. Cache memory often resides on-chip, and often uses fast but expensive static RAM technology rather than slower but cheaper dynamic RAM (see Chapter 5). Cache memory is based on the principle that if at a particular time a processor accesses a particular memory location, then the processor will likely access that location and immediate neighbors of the location in the near future. Thus, when we first access a location in memory, we copy that location and some number of its neighbors (called a block) into cache, and then access the copy of the location in cache. When we access another location, we first check a cache table to see if a copy of the location resides in

Figure 2.3: Cache memory.



cache. If the copy does reside in cache, we have a cache *hit*, and we can read or write that location very quickly. If the copy does not reside in cache, we have a cache *miss*, so we must copy the location's block into cache, which takes a lot of time. Thus, for a cache to be effective in improving performance, the ratio of hits to misses must be very high, requiring intelligent caching schemes. Caches are used for both program memory (often called instruction cache, or I-cache) as well as data memory (often called D-cache).

2.3 Operation

2.3.1 Instruction execution

We can think of a microprocessor's execution of instructions as consisting of several basic stages:

1. *Fetch instruction*: the task of reading the next instruction from memory into the instruction register.
2. *Decode instruction*: the task of determining what operation the instruction in the instruction register represents (e.g., add, move, etc.).
3. *Fetch operands*: the task of moving the instruction's operand data into appropriate registers.
4. *Execute operation*: the task of feeding the appropriate registers through the ALU and back into an appropriate register.
5. *Store results*: the task of writing a register into memory.

If each stage takes one clock cycle, then we can see that a single instruction may take several cycles to complete.

2.3.2 Pipelining

Pipelining is a common way to increase the instruction throughput of a microprocessor. We first make a simple analogy of two people approaching the chore of washing and drying 8 dishes. In one approach, the first person washes all 8 dishes, and then the second person dries all 8 dishes. Assuming 1 minute per dish per person, this approach requires 16 minutes. The approach is clearly inefficient since at any time only one person is working and the other is idle. Obviously, a better approach is for the second person to begin drying the first dish immediately after it has been washed. This approach requires only 9 minutes -- 1 minute for the first dish to be washed, and then 8 more minutes until the last dish is finally dry . We refer to this latter approach as pipelined.

Each dish is like an instruction, and the two tasks of washing and drying are like the five stages listed above. By using a separate unit (each akin a person) for each stage, we can pipeline instruction execution. After the instruction fetch unit fetches the first instruction, the decode unit decodes it while the instruction fetch unit simultaneously

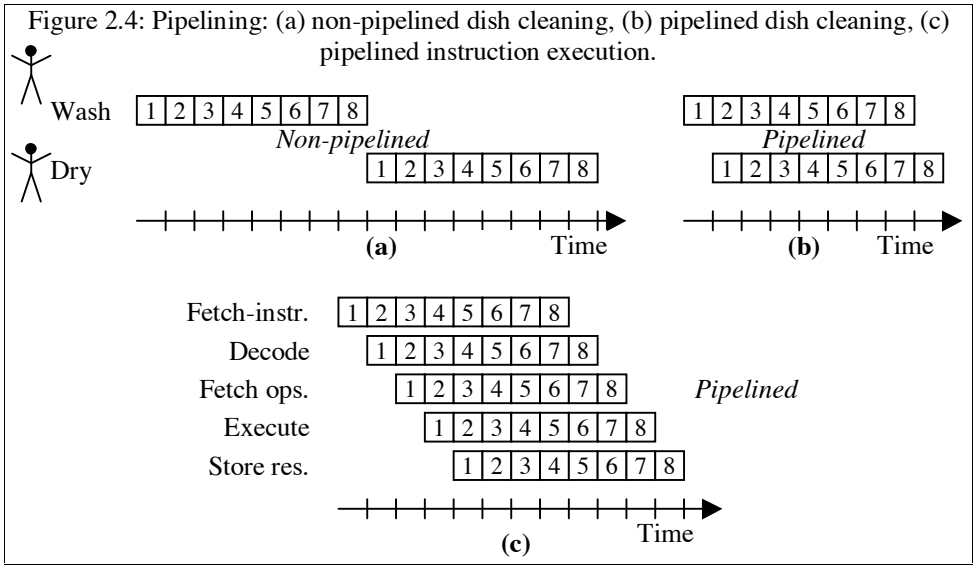


Figure 2.4: Pipelining: (a) non-pipelined dish cleaning, (b) pipelined dish cleaning, (c) pipelined instruction execution.

fetches the next instruction. The idea of pipelining is illustrated in Figure 2.4. Note that for pipelining to work well, instruction execution must be decomposable into roughly equal length stages, and instructions should each require the same number of cycles.

Branches pose a problem for pipelining, since we don't know the next instruction until the current instruction has reached the execute stage. One solution is to stall the pipeline when a branch is in the pipeline, waiting for the execute stage before fetching the next instruction. An alternative is to guess which way the branch will go and fetch the corresponding instruction next; if right, we proceed with no penalty, but if we find out in the execute stage that we were wrong, we must ignore all the instructions fetched since the branch was fetched, thus incurring a penalty. Modern pipelined microprocessors often have very sophisticated branch predictors built in.

2.4 Programmer's view

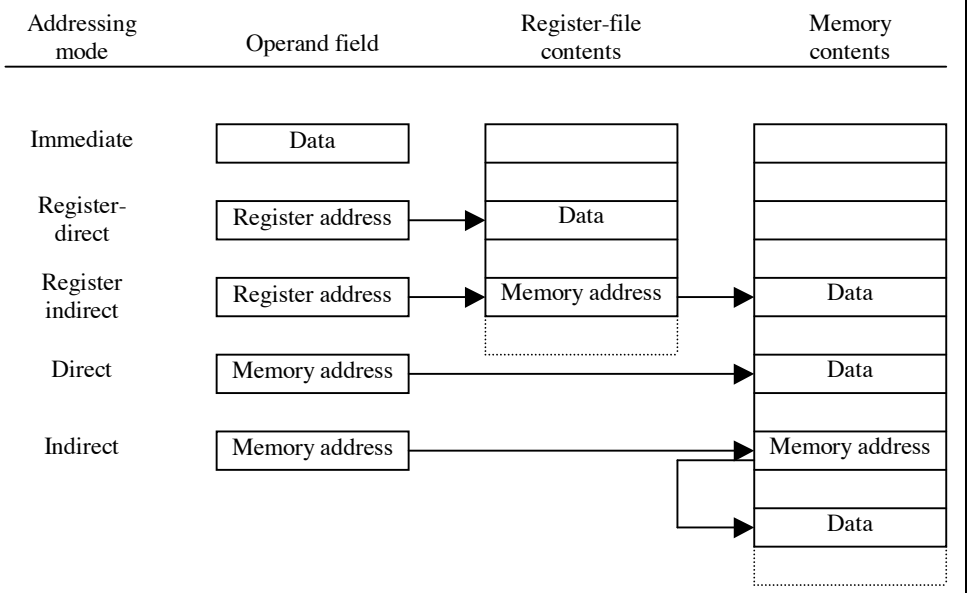
A programmer writes the program instructions that carry out the desired functionality on the general-purpose processor. The programmer may not actually need to know detailed information about the processor's architecture or operation, but instead may deal with an architectural abstraction, which hides much of that detail. The level of abstraction depends on the level of programming. We can distinguish between two levels of programming. The first is assembly-language programming, in which one programs in a language representing processor-specific instructions as mnemonics. The second is structured-language programming, in which one programs in a language using processor-independent instructions. A compiler automatically translates those instructions to processor-specific instructions. Ideally, the structured-language programmer would need no information about the processor architecture, but in embedded systems, the programmer must usually have at least some awareness, as we shall discuss.

Actually, we can define an even lower-level of programming, machine-language programming, in which the programmer writes machine instructions in binary. This level of programming has become extremely rare due to the advent of assemblers. Machine-language programmed computers often had rows of lights representing to the programmer the current binary instructions being executed. Today's computers look more like boxes or refrigerators, but these do not make for interesting movie props, so you may notice that in the movies, computers with rows of blinking lights live on.

2.4.1 Instruction set

The assembly-language programmer must know the processor's instruction set. The instruction set describes the bit-configurations allowed in the IR, indicating the atomic

Figure 2.5: Addressing modes.



processor operations that the programmer may invoke. Each such configuration forms an *assembly instruction*, and a sequence of such instructions forms an *assembly program*.

An instruction typically has two parts, an opcode field and operand fields. An *opcode* specifies the operation to take place during the instruction. We can classify instructions into three categories. *Data-transfer* instructions move data between memory and registers, between input/output channels and registers, and between registers themselves. *Arithmetic/logical* instructions configure the ALU to carry out a particular function, channel data from the registers through the ALU, and channel data from the ALU back to a particular register. *Branch* instructions determine the address of the next program instruction, based possibly on datapath status signals.

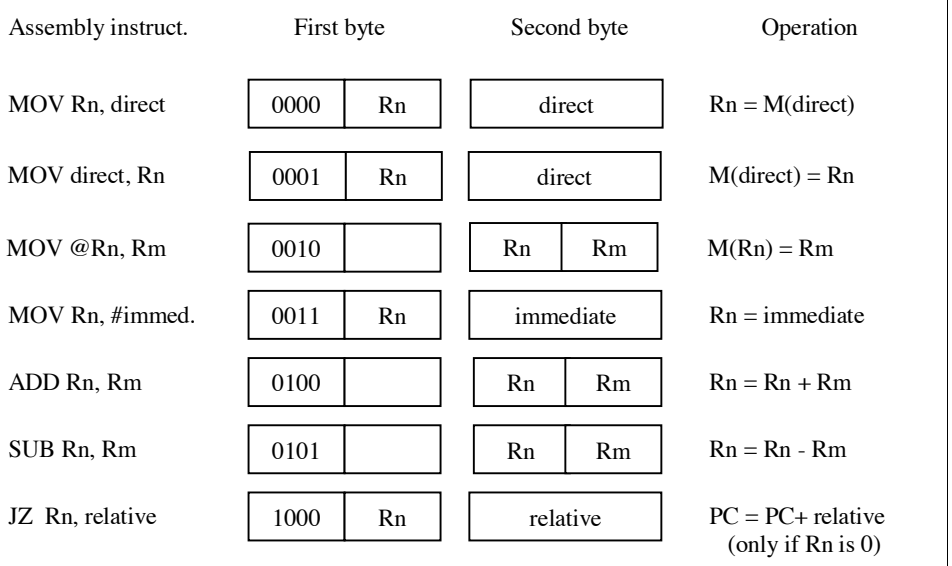
Branches can be further categorized as being *unconditional* jumps, *conditional* jumps or procedure *call* and *return* instructions. Unconditional jumps always determine the address of the next instruction, while conditional jumps do so only if some condition evaluates to true, such as a particular register containing zero. A call instruction, in addition to indicating the address of the next instruction, saves the address of the current instruction[1] so that a subsequent return instruction can jump back to the instruction immediately following the most recent invoked call instruction. This pair of instructions facilitates the implementation of procedure/function call semantics of high-level programming languages.

An *operand* field specifies the location of the actual data that takes part in an operation. Source operands serve as input to the operation, while a destination operand stores the output. The number of operands per instruction varies among processors. Even for a given processor, the number of operands per instruction may vary depending on the instruction type.

The operand field may indicate the data's location through one of several *addressing modes*, illustrated in Figure 2.5. In *immediate* addressing, the operand field contains the data itself. In *register* addressing, the operand field contains the address of a datapath

[1] On most machines, a call instruction increments the stack pointer, then stores the current program-counter at the memory location pointed to by the stack pointer, in effect performing a push operation. Conversely, a return instruction pops the top of the stack and branches back to the saved program location.

Figure 2.6: A simple (trivial) instruction set.



register in which the data resides. In *register-indirect* addressing, the operand field contains the address of a register, which in turn contains the address of a memory location in which the data resides. In *direct* addressing, the operand field contains the address of a memory location in which the data resides. In *indirect* addressing, the operand field contains the address of a memory location, which in turn contains the address of a memory location in which the data resides. Those familiar with structured languages may note that direct addressing implements regular variables, and indirect addressing implements pointers. In *inherent* or *implicit* addressing, the particular register or memory location of the data is implicit in the opcode; for example, the data may reside in a register called the "accumulator." In *indexed* addressing, the direct or indirect operand must be added to a particular implicit register to obtain the actual operand address. Jump instructions may use *relative* addressing to reduce the number of bits needed to indicate the jump address. A relative address indicates how far to jump from the current address, rather than indicating the complete address – such addressing is very common since most jumps are to nearby instructions.

Ideally, the structured-language programmer would not need to know the instruction set of the processor. However, nearly every embedded system requires the programmer to write at least some portion of the program in assembly language. Those portions may deal with low-level input/output operations with devices outside the processor, like a display device. Such a device may require specific timing sequences of signals in order to receive data, and the programmer may find that writing assembly code achieves such timing most conveniently. A *driver* routine is a portion of a program written specifically to communicate with, or drive, another device. Since drivers are often written in assembly language, the structured-language programmer may still require some familiarity with at least a subset of the instruction set.

Figure 2.6 shows a (trivial) instruction set with 4 data transfer instructions, 2 arithmetic instructions, and 1 branch instruction, for a hypothetical processor. Figure 2.7(a) shows a program, written in C, that adds the numbers 1 through 10. Figure 2.7(b) shows that same program written in assembly language using the given instruction set.

2.4.2 Program and data memory space

The embedded systems programmer must be aware of the size of the available memory for program and for data. For example, a particular processor may have a 64K program space, and a 64K data space. The programmer must not exceed these limits. In addition, the programmer will probably want to be aware of on-chip program and data memory capacity, taking care to fit the necessary program and data in on-chip memory if possible.

Figure 2.7: Sample programs: (a) C program, (b) equivalent assembly program.

```
int total = 0;
for (int i=10; i!=0; i--)
   total += i;
// next instructions...
```

(a)

```
        MOV R0, #0;      // total = 0
        MOV R1, #10;     // i = 10
        MOV R2, #1;      // constant 1
        MOV R3, #0;      // constant 0
Loop:   JZ R1, Next;     // Done if i=0
        ADD R0, R1;      // total += i
        SUB R1, R2;      // i--
        JZ R3, Loop;     // Jump always
Next:   // next instructions...
```

(b)

2.4.3 Registers

The assembly-language programmer must know how many registers are available for general-purpose data storage. He/she must also be familiar with other registers that have special functions. For example, a base register may exist, which permits the programmer to use a data-transfer instruction where the processor adds an operand field to the base register to obtain an actual memory address.

Other special-function registers must be known by both the assembly-language and the structured-language programmer. Such registers may be used for configuring built-in timers, counters, and serial communication devices, or for writing and reading external pins.

2.4.4 I/O

The programmer should be aware of the processor's input and output (I/O) facilities, with which the processor communicates with other devices. One common I/O facility is parallel I/O, in which the programmer can read or write a port (a collection of external pins) by reading or writing a special-function register. Another common I/O facility is a system bus, consisting of address and data ports that are automatically activated by certain addresses or types of instructions. I/O methods will be discussed further in a later chapter.

2.4.5 Interrupts

An interrupt causes the processor to suspend execution of the main program, and instead jump to an Interrupt Service Routine (ISR) that fulfills a special, short-term processing need. In particular, the processor stores the current PC, and sets it to the address of the ISR. After the ISR completes, the processor resumes execution of the main program by restoring the PC.The programmer should be aware of the types of interrupts supported by the processor (we describe several types in a subsequent chapter), and must write ISRs when necessary. The assembly-language programmer places each ISR at a specific address in program memory. The structured-language programmer must do so also; some compilers allow a programmer to force a procedure to start at a particular memory location, while recognize pre-defined names for particular ISRs.

For example, we may need to record the occurrence of an event from a peripheral device, such as the pressing of a button. We record the event by setting a variable in memory when that event occurs, although the user's main program may not process that event until later. Rather than requiring the user to insert checks for the event throughout the main program, the programmer merely need write an interrupt service routine and associate it with an input pin connected to the button. The processor will then call the routine automatically when the button is pressed.

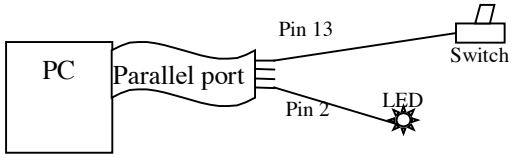
Example: Assembly-language programming of device drivers

This example provides an application of assembly language programming of a low-level driver, showing how the parallel port of an x86 based PC (Personal Computer) can be used to perform digital I/O. Writing and reading three special registers accomplishes parallel communication on the PC. Those three register are actually in an 8255A Peripheral Interface Controller chip. In unidirectional mode, (default power-on-reset mode), this device is capable of driving 12 output and five input lines. In the following table, we provide the parallel port (known as LPT) connector pin numbers and the corresponding register location.

Parallel port signals and associated registers.

LPT Connector Pin	I/O Direction	Register Address
1	Output	0^{th} bit of register #2
2-9	Output	0^{th}-7^{th} bit of register #0
10	Input	6^{th} bit of register #1
11	Input	7^{th} bit of register #1
12	Input	5^{th} bit of register #1
13	Input	4^{th} bit of register #1
14	Output	1^{st} bit of register #2
15	Input	3^{rd} bit of register #1
16	Output	2^{nd} bit of register #2
17	Output	3^{rd} bit of register #2

In our example, we are to build the following system:



A switch is connected to input pin number 13 of the parallel port. An LED (light-emitting diode) is connected to output pin number 2. Our program, running on the PC, should monitor the input switch and turn on/off the LED accordingly.

Figure 2.8 gives the code for such a program, in x86 assembly language. Note that the *in* and *out* assembly instructions read and write the internal registers of the 8255A. Both instructions take two operands, address and data. Address specifies the the register we are trying to read or write. This address is calculated by adding the address of the device, called the *base address*, to the address of the particular register as given in Figure 2.8. In most PCs, the base address of LPT1 is at 3BC hex (though not always). The second operand is the data. For the *in* instruction, the content of this eight-bit operand will be written to the addressed register. For the *out* instruction, the content of the addressed eight-bit register will be read into this operand.

The program makes use of masking, something quite common during low-level I/O. A *mask* is a bit-pattern designed such that ANDing it with a data item D yields a specific part of D. For example, a mask of 00001111 can be used to yield bits 0 through 3, e.g., 00001111 AND 10101010 yields 00001010. A mask of 00010000, or 10h in hexadecimal format, would yield bit 4.

In Figure 2.8, we have broken our program in two source files, assembly and C. The assembly program implements the low-level I/O to the parallel port and the C program implements the high-level application. Our assembly program is a simple form of a *device driver* program that provides a single procedure to the high-level application. While the trend is for embedded systems to be written in structured languages, this example shows that some small assembly program may still need to be written for low-level drivers.

Figure 2.8: Parallel port example.

```
;
; This program consists of a sub-routine that reads
; the state of the input pin, determining the on/off state
; of our switch and asserts the output pin, turning the LED
; on/off accordingly.
;
       .386

CheckPort      proc
       push    ax              ; save the content
       push    dx              ; save the content
       mov     dx, 3BCh + 1    ; base + 1 for register #1
       in      al, dx          ; read register #1
       and     al, 10h         ; mask out all but bit # 4
       cmp     al, 0           ; is it 0?
       jne     SwitchOn        ; if not, we need to turn the LED on

SwitchOff:
       mov     dx, 3BCh + 0    ; base + 0 for register #0
       in      al, dx          ; read the current state of the port
       and     al, f7h         ; clear first bit (masking)
       out     dx, al          ; write it out to the port
       jmp     Done            ; we are done

SwitchOn:
       mov     dx, 3BCh + 0    ; base + 0 for register #0
       in      al, dx          ; read the current state of the port
       or      al, 01h         ; set first bit (masking)
       out     dx, al          ; write it out to the port

Done:  pop     dx              ; restore the content
       pop     ax              ; restore the content
CheckPort      endp
```

```
extern "C" CheckPort(void);       // defined in assembly above

void main(void) {

       while( 1 ) {

              CheckPort();
       }
}
```

2.4.6 Operating system

An operating system is a layer of software that provides low-level services to the *application layer*, a set of one or more programs executing on the CPU consuming and producing input and output data. The task of managing the application layer involves the loading and executing of programs, sharing and allocating system resources to these programs, and protecting these allocated resources from corruption by non-owner programs. One of the most important resource of a system is the central processing unit (CPU), which is typically shared among a number of executing programs. The operating system, thus, is responsible for deciding what program is to run next on the CPU and for how long. This is called process/task scheduling and is determined by the operating system's preemption policy. Another very important resource is memory, including disk storage, which is also shared among the applications running on the CPU.

In addition to implementing an environment for management of high-level application programs, the operating system provides the software required for servicing various hardware-interrupts, and provides device drivers for driving the peripheral devices present in the system. Typically, on startup, an operating system initializes all peripheral devices, such as disk controllers, timers and input/output devices and installs

Figure 2.9: System call invocation.

```
DB file_name "out.txt"     -- store file name

    MOV R0, 1324           -- system call "open" id
    MOV R1, file_name      -- address of file-name
    INT 34                 -- cause a system call
    JZ  R0, L1             -- if zero -> error

    . . . read the file
    JMP L2                 -- bypass error cond.
L1:
    . . . handle the error

L2:
```

hardware interrupt (interrupts generated by the hardware) service routines (ISR) to handle various signals generated by these devices[2]. Then, it installs *software interrupts* (interrupts generated by the software) to process *system calls* (calls made by high-level applications to request operating system services) as described next.

A system call is a mechanism for an application to invoke the operating system. This is analogous to a procedure or function call, as in high-level programming languages. When a program requires some service from the operating system, it generates a predefined software interrupt that is serviced by the operating system. Parameters specific to the requested services are typically passed from (to) the application program to (from) the operating system through CPU registers. Figure 2.9 illustrates how the file "open" system call may be invoked, in assembly, by a program. Languages like C and Pascal provide wrapper functions around the system-calls to provide a high-level mechanism for performing system calls.

In summary, the operating system abstracts away the details of the underlying hardware and provides the application layer an interface to the hardware through the system call mechanism.

2.4.7 Development environment

Several software and hardware tools commonly support the programming of general-purpose processors. First, we must distinguish between two processors we deal with when developing an embedded system. One processor is the *development processor*, on which we write and debug our program. This processor is part of our desktop computer. The other processor is the *target processor*, to which we will send our program and which will form part of our embedded system's implementation. For example, we may develop our system on a Pentium processor, but use a Motorola 68HC11 as our target processor. Of course, sometimes the two processors happen to be the same, but this is mostly a coincidence.

Assemblers translate assembly instructions to binary machine instructions. In addition to just replacing opcode and operand mnemonics by binary equivalents, an assembler may also translate symbolic labels into actual addresses. For example, a programmer may add a symbolic label *END* to an instruction *A*, and may reference *END* in a branch instruction. The assembler determines the actual binary address of *A*, and replaces references to *END* by this address. The mapping of assembly instructions to machine instructions is one-to-one. A *linker* allows a programmer to create a program in

[2] The operating system itself is loaded into memory by a small program that typically resides in a special ROM and is always executed after a power-on reset.

separately-assembled files; it combines the machine instructions of each into a single program, perhaps incorporating instructions from standard library routines.

Compilers translate structured programs into machine (or assembly) programs. Structured programming languages possess high-level constructs that greatly simplify programming, such as loop constructs, so each high-level construct may translate to several or tens of machine instructions. Compiler technology has advanced tremendously over the past decades, applying numerous program optimizations, often yielding very size and performance efficient code. A *cross-compiler* executes on one processor (our development processor), but generates code for a different processor (our target processor). Cross-compilers are extremely common in embedded system development.

Debuggers help programmers evaluate and correct their programs. They run on the development processor and support stepwise program execution, executing one instruction and then stopping, proceeding to the next instruction when instructed by the user. They permit execution up to user-specified breakpoints, which are instructions that when encountered cause the program to stop executing. Whenever the program stops, the user can examine values of various memory and register locations. A *source-level* debugger enables step-by-step execution in the source program language, whether assembly language or a structured language. A good debugging capability is crucial, as today's programs can be quite complex and hard to write correctly.

Device programmers download a binary machine program from the development processor's memory into the target processor's memory.

Emulators support debugging of the program while it executes on the target processor. An emulator typically consists of a debugger coupled with a board connected to the desktop processor via a cable. The board consists of the target processor plus some support circuitry (often another processor). The board may have another cable with a device having the same pin configuration as the target processor, allowing one to plug this device into a real embedded system. Such an *in-circuit emulator* enables one to control and monitor the program's execution in the actual embedded system circuit. In-circuit emulators are available for nearly any processor intended for embedded use, though they can be quite expensive if they are to run at real speeds.

The availability of low-cost or high-quality development environments for a processor often heavily influences the choice of a processor.

2.5 Microcontrollers

Numerous processor IC manufacturers market devices specifically for the embedded systems domain. These devices may include several features. First, they may include several peripheral devices, such as timers, analog to digital converters, and serial communication devices, on the same IC as the processor. Second, they may include some program and data memory on the same IC. Third, they may provide the programmer with direct access to a number of pins of the IC. Fourth, they may provide specialized instructions for common embedded system operations, such as bit-manipulation operations. A *microcontroller* is a device possessing some or all of these features.

Incorporating peripherals and memory onto the same IC reduces the number of required IC's, resulting in compact and low-power implementations. Providing pin access allows programs to easily monitor sensors, set actuators, and transfer data with other devices. Providing specialized instructions improves performance for embedded systems applications; thus, microcontrollers can be considered ASIPs to some degree.

Many manufactures market devices referred to as "embedded processors." The difference between embedded processors and microcontrollers is not clear, although we note that the former term seems to be used more for large (32-bit) processors.

2.6 Selecting a microprocessor

The embedded system designer must select a microprocessor for use in an embedded system. The choice of a processor depends on technical and non-technical

Figure 2.10: General Purpose Processors

Processor	Clock Speed	Peripherals	Bus Width	MIPS	Power	Transistor	Price
Intel SA110	233 MHz	32K cache	32	268	360 mW	2.1 M	$49
VLSI ARM710	25 MHz	8K cache	32	30	120 mW	341 K	$35
IBM 401GF	50 MHz	3K cache	32	52	140 mW	345 K	$11
Mistubishi M32R/D	66 MHz	4K cache	16	52	180 mW	7 M	$80
PIC 12508	8 MHz	512 ROM, 25 RAM, 5 I/O	8	~ .8	NA	~10 K	$6
Intel 8051	12 MHz	4K ROM, 128 RAM, 32 I/O, Timer, UART	8	~ 1	NA	~10 K	$7

Sources: Embedded Systems Programming, Nov. 1998; PIC and Intel datasheets.

aspects. From a technical perspective, one must choose a processor that can achieve the desired speed within certain power, size and cost constraints. Non-technical aspects may include prior expertise with a processor and its development environment, special licensing arrangements, and so on.

Speed is a particularly difficult processor aspect to measure and compare. We could compare processor clock speeds, but the number of instructions per clock cycle may differ greatly among processors. We could instead compare instructions per second, but the complexity of each instruction may also differ greatly among processors -- e.g., one processor may require 100 instructions, and another 300 instructions, to perform the same computation. One attempt to provide a means for a fairer comparison is the *Dhrystone benchmark*. A benchmark is a program intended to be run on different processors to compare their performance. The Dhrystone benchmark was developed in 1984 by Reinhold Weicker specifically as a performance benchmark; it performs no useful work. It focuses on exercising a processor's integer arithmetic and string-handling capabilities. It is written in C and in the public domain. Since most processors can execute it in milliseconds, it is typically executed thousands of times, and thus a processor is said to be able to execute so many Dhrystones per second.

Another commonly-used speed comparison unit, which happens to be based on the Dhrystone, is MIPS. One might think that MIPS simply means Millions of Instructions Per Second, but actually the common use of the term is based on a somewhat more complex notion. Specifically, its origin is based on the speed of Digital's VAX 11/780, thought to be the first computer able to execute one million instructions per second. A VAX 11/780 could execute 1,757 Dhrystones/second. Thus, for a VAX 11/780, 1 MIPS = 1,757 Dhrystones/second. This unit for MIPS is the one used today. So if a machine today is said to run at 750 MIPS, that actually means it can execute 750*1757 = 1,317,750 Dhrystones/second.

The use and validity of benchmark data is a subject of great controversy. There is also a clear need for benchmarks that measure performance of embedded processors.

Numerous general-purpose processors have evolved in the recent years and are in common use today. In Figure 2.10, we summarize some of the features of several popular processors.

2.7 Summary

General-purpose processors are popular in embedded systems due to several features, including low unit cost, good performance, and low NRE cost. A general-purpose processor consists of a controller and datapath, with a memory to store program and data. To use a general-purpose processor, the embedded system designer must write a

program. The designer may write some parts of this program, such as driver routines, using assembly language, while writing other parts in a structured language. Thus, the designer should be aware of several aspects of the processor being used, such as the instruction set, available memory, registers, I/O facilities, and interrupt facilities. Many tools exist to support the designer, including assemblers, compilers, debuggers, device programmers and emulators. The designer often makes use of microcontrollers, which are processors specifically targeted to embedded systems. These processors may include on-chip peripheral devices and memory, additional I/O ports, and instructions supporting common embedded system operations. The designer has a variety of processors from which to choose.

2.8 References and further reading

[1] Philips semiconductors, *80C51-based 8-bit microcontrollers databook*, Philips Electronics North America, 1994. Provides an overview of the 8051 architecture and on-chip peripherals, describes a large number of derivatives each with various features, describes the I2C and CAN bus protocols, and highlights development support tools.

[2] Rafiquzzaman, Mohamed. *Microprocessors and microcomputer-based system design*. Boca Raton: CRC Press, 1995. ISBN 0-8493-4475-1. Provides an overview of general-purpose processor architecture, along with detailed descriptions of various Intel 80xx and Motorola 68000 series processors.

[3] *Embedded Systems Programming*, Miller Freeman Inc., San Francisco, 1999. A monthly publication covering trends in various aspects of general-purpose processors for embedded systems, including programming, compilers, operating systems, emulators, device programmers, microcontrollers, PLDs, and memories. An annual buyer's guide provides tables of vendors for these items, including 8/16/32/64-bit microcontrollers/microprocessors and their features.

[4] *Microprocessor Report,* MicroDesign Resources, California, 1999. A monthly report providing in-depth coverage of trends, announcements, and technical details, for desktop, mobile, and embedded microprocessors.

2.9 Exercises

1. Describe why a general-purpose processor could cost less than a single-purpose processor you design yourself.
2. Detail the stages of executing the MOV instructions of Figure 2.4, assuming an 8-bit processor and a 16-bit IR and program memory following the model of Figure 2.1. Example: the stages for the ADD instruction are -- (1) fetch M[PC] into IR, (2) read Rn and Rm from register file through ALU configured for ADD, storing results back in Rn.
3. Add one instruction to the instruction set of Figure 2.4 that would reduce the size our summing assembly program by 1 instruction. (Hint: add a new branch instruction). Show the reduced program.
4. Create a table listing the address spaces for the following address sizes: (a) 8-bit, (b) 16-bit, (c) 24-bit, (d) 32-bit, (e) 64-bit.
5. Illustrate how program and data memory fetches can be overlapped in a Harvard architecture.
6. Read the entire problem before beginning: (a) Write a C program that clears an array "short int M[256]." In other words, the program sets every location to 0. Hint: your program should only be a couple lines long. (b) Assuming M starts at location 256 (and thus ends at location 511), write the same program in assembly language using the earlier instruction set. (c) Measure the time it takes you to perform parts a and b, and report those times.
7. Acquire a databook for a microcontroller. List the features of the basic version of that microcontroller, including key characteristics of the instruction set (number of instructions of each type, length per instruction, etc.), memory architecture and

available memory, general-purpose registers, special-function registers, I/O facilities, interrupt facilities, and other salient features.

8. For the above microcontroller, create a table listing 5 existing variations of that microcontroller, stressing the features that differ from the basic version.

Chapter 3 *Standard single-purpose processors: Peripherals*

3.1 Introduction

A single-purpose processor is a digital system intended to solve a specific computation task. The processor may be a *standard* one, intended for use in a wide variety of applications in which the same task must be performed. The manufacturer of such an off-the-shelf processor sells the device in large quantities. On the other hand, the processor may be a *custom* one, built by a designer to implement a task specific to a particular application. An embedded system designer choosing to use a standard single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits.

First, performance may be fast, since the processor is customized for the particular task at hand. Not only might the task execute in fewer clock cycles, but also those cycles themselves may be shorter. Fewer clock cycles may result from many datapath components operating in parallel, from datapath components passing data directly to one another without the need for intermediate registers (chaining), or from elimination of program memory fetches. Shorter cycles may result from simpler functional units, less multiplexors, or simpler control logic. For standard single-purpose processors, manufacturers may spread NRE cost over many units. Thus, the processor's clock cycle may be further reduced by the use of custom IC technology, leading-edge IC's, and expert designers, just as is the case with general-purpose processors.

Second, size may be small. A single-purpose processor does not require a program memory. Also, since it does not need to support a large instruction set, it may have a simpler datapath and controller.

Third, a standard single-purpose processor may have low unit cost, due to the manufacturer spreading NRE cost over many units. Likewise, NRE cost may be low, since the embedded system designer need not design a standard single-purpose processor, and may not even need to program it.

There are of course tradeoffs. If we are already using a general-purpose processor, then implementing a task on an additional single-purpose processor rather than in software may add to the system size and power consumption.

In this chapter, we describe the basic functionality of several standard single-purpose processors commonly found in embedded systems. The level of detail of the description is intended to be enough to enable using such processors, but not necessarily to design one.

We often refer to standard single-purpose processors as *peripherals,* because they usually exist on the periphery of the CPU. However, microcontrollers tightly integrate these peripherals with the CPU, often placing them on-chip, and even assigning peripheral registers to the CPU's own register space. The result is the common term "on-chip peripherals," which some may consider somewhat of an oxymoron.

3.2 Timers, counters, and watchdog timers

A *timer* is a device that generates a signal pulse at specified time intervals. A time interval is a "real-time" measure of time, such as 3 milliseconds. These devices are extremely useful in systems in which a particular action, such as sampling an input signal or generating an output signal, must be performed every *X* time units.

Internally, a simple timer may consist of a register, counter, and an extremely simple controller. The register holds a count value representing the number of clock cycles that equals the desired real-time value. This number can be computed using the simple formula:

$$\textit{Number of clock cycles} = \textit{Desired real-time value} / \textit{Clock cycle}$$

For example, to obtain a duration of 3 milliseconds from a clock cycle of 10 nanoseconds (100 MHz), we must count ($3x10^{-6}$ s / $10x10^{-9}$ s/cycle) = 300 cycles. The counter is initially loaded with the count value, and then counts down on every clock cycle until 0 is reached, at which point an output signal is generated, the count value is reloaded, and the process repeats itself.

To use a timer, we must configure it (write to its registers), and respond to its output signal. When we use a timer in conjunction with a general-purpose processor, we typically respond to the timer signal by assigning it to an interrupt, so we include the desired action in an interrupt service routine. Many microcontrollers that include built-in timers will have special interrupts just for its timers, distinct from external interrupts.

Note that we could use a general-purpose processor to implement a timer. Knowing the number of cycles that each instruction requires, we could write a loop that executed the desired number of instructions; when this loop completes, we know that the desired time passed. This implementation of a timer on a dedicated general-purpose processor is obviously quite inefficient in terms of size. One could alternatively incorporate the timer functionality into a main program, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timer functionality to a special-purpose processor becomes evident.

A *counter* is nearly identical to a timer, except that instead of counting clock cycles (pulses on the clock signal), a counter counts pulses on some other input signal.

A *watchdog timer* can be thought of as having the inverse functionality than that of a regular timer. We configure a watchdog timer with a real-time value, just as with a regular timer. However, instead of the timer generating a signal for us every X time units, *we* must generate a signal for the timer every X time units. If we fail to generate this signal in time, then the timer generates a signal indicating that we failed. We often connect this signal to the reset or interrupt signal of a general-purpose processor. Thus, a watchdog timer provides a mechanism of ensuring that our software is working properly; every so often in the software, we include a statement that generates a signal to the watchdog timer (in particular, that resets the timer). If something undesired happens in the software (e.g., we enter an undesired infinite loop, we wait for an input signal that never arrives, a part fails, etc.), the watchdog generates a signal that we can use to restart or test parts of the system. Using an interrupt service routine, we may record information as to the number of failures and the causes of each, so that a service technician may later evaluate this information to determine if a particular part requires replacement. Note that an embedded system often must recover from failures whenever possible, as the user may not have the means to reboot the system in the same manner that he/she might reboot a desktop system.

3.3 UART

A *UART* (Universal Asynchronous Receiver/Transmitter) receives serial data and stores it as parallel data (usually one byte), and takes parallel data and transmits it as serial data. The principles of serial communication appear in a later chapter.

Such serial communication is beneficial when we need to communicate bytes of data between devices separated by long distances, or when we simply have few available I/O pins. Principles of serial communication will be discussed in a later chapter. For our purpose in this section, we must be aware that we must set the transmission and reception rate, called the baud rate, which indicates the frequency that the signal changes. Common rates include 2400, 4800, 9600, and 19.2k. We must also be aware that an extra bit may be added to each data word, called parity, to detect transmission errors -- the parity bit is set to high or low to indicate if the word has an even or odd number of bits.

Internally, a simple UART may possess a baud-rate configuration register, and two independently operating processors, one for receiving and the other for transmitting. The transmitter may possess a register, often called a transmit buffer, that holds data to be sent. This register is a shift register, so the data can be transmitted one bit at a time by shifting at the appropriate rate. Likewise, the receiver receives data into a shift register,

and then this data can be read in parallel. Note that in order to shift at the appropriate rate based on the configuration register, a UART requires a timer.

To use a UART, we must configure its baud rate by writing to the configuration register, and then we must write data to the transmit register and/or read data from the received register. Unfortunately, configuring the baud rate is usually not as simple as writing the desired rate (e.g., 4800) to a register. For example, to configure the UART of an 8051, we must use the following equation:

$$Baudrate = (2^{smod} / 32) * oscfreq / (12 * (256 - TH1)))$$

smod corresponds to 2 bits in a special-function register, *oscfreq* is the frequency of the oscillator, and *TH1* is an 8-bit rate register of a built-in timer.

Note that we could use a general-purpose processor to implement a UART completely in software. If we used a dedicated general-processor, the implementation would be inefficient in terms of size. We could alternatively integrate the transmit and receive functionality with our main program. This would require creating a routine to send data serially over an I/O port, making use of a timer to control the rate. It would also require using an interrupt service routine to capture serial data coming from another I/O port whenever such data begins arriving. However, as with the timer functionality, adding send and receive functionality can detract from time for other computations.

Knowing the number of cycles that each instruction requires, we could write a loop that executed the desired number of instructions; when this loop completes, we know that the desired time passed. This implementation of a timer on a dedicated general-purpose processor is obviously quite inefficient in terms of size. One could alternatively incorporate the timer functionality into a main program, but the timer functionality then occupies much of the program's run time, leaving little time for other computations. Thus, the benefit of assigning timer functionality to a special-purpose processor becomes evident.

3.4 Pulse width modulator

A *pulse-width modulator* (PWM) generates an output signal that repeatedly switches between high and low. We control the duration of the high value and of the low value by indicating the desired period, and the desired *duty cycle*, which is the percentage of time the signal is high compared to the signal's period. A *square wave* has a duty cycle of 50%. The pulse's width corresponds to the pulse's time high.

Again, PWM functionality could be implemented on a dedicated general-purpose processor, or integrated with another program's functionality, but the single-purpose processor approach has the benefits of efficiency and simplicity.

One common use of a PWM is to control the average current or voltage input to a device. For example, a DC motor rotates when power is applied, and this power can be turned on and off by setting an input high or low. To control the speed, we can adjust the input voltage, but this requires a conversion of our high/low digital signals to an analog signal. Fortunately, we can also adjust the speed simply by modifying the duty cycle of the motors on/off input, an approach which adjusts the average voltage. This approach works because a DC motor does not come to an immediate stop when power is turned off, but rather it coasts, much like a bicycle coasts when we stop pedaling. Increasing the duty cycle increases the motor speed, and decreasing the duty cycle decreases the speed. This duty cycle adjustment principle applies to the control other types of electric devices, such as dimmer lights.

Another use of a PWM is to encode control commands in a single signal for use by another device. For example, we may control a radio-controlled car by sending pulses of different widths. Perhaps a 1 ms width corresponds to a turn left command, a 4 ms width to turn right, and 8 ms to forward.

3.5 LCD controller

An *LCD (Liquid crystal display)* is a low-cost, low-power device capable of displaying text and images. LCDs are extremely common in embedded systems, since such systems often do not have video monitors standard for desktop systems. LCDs can be found in numerous common devices like watches, fax and copy machines, and calculators.

The basic principle of one type of LCD (reflective) works as follows. First, incoming light passes through a polarizing plate. Next, that polarized light encounters liquid crystal material. If we excite a region of this material, we cause the material's molecules to align, which in turn causes the polarized light to pass through the material. Otherwise, the light does not pass through. Finally, light that has passed through hits a mirror and reflects back, so the excited region appears to light up. Another type of LCD (absorption) works similarly, but uses a black surface instead of a mirror. The surface below the excited region absorbs light, thus appearing darker than the other regions.

One of the simplest LCDs is 7-segment LCD. Each of the 7 segments can be activated to display any digit character or one of several letters and symbols. Such an LCD may have 7 inputs, each corresponding to a segment, or it may have only 4 inputs to represent the numbers 0 through 9. An LCD driver converts these inputs to the electrical signals necessary to excite the appropriate LCD segments.

A dot-matrix LCD consists of a matrix of dots that can display alphanumeric characters (letters and digits) as well as other symbols. A common dot-matrix LCD has 5 columns and 8 rows of dots for one character. An LCD driver converts input data into the appropriate electrical signals necessary to excite the appropriate LCD bits.

Each type of LCD may be able to display multiple characters. In addition, each character may be displayed in normal or inverted fashion. The LCD may permit a character to be blinking (cycling through normal and inverted display) or may permit display of a cursor (such as a blinking underscore) indicating the "current" character. This functionality would be difficult for us to implement using software. Thus, we use an *LCD controller* to provide us with a simple interface, perhaps 8 data inputs and one enable input. To send a byte to the LCD, we provide a value to the 8 inputs and pulse the enable. This byte may be a control word, which instructs the LCD controller to initialize the LCD, clear the display, select the position of the cursor, brighten the display, and so on. Alternatively, this byte may be a data word, such as an ASCII character, instructing the LCD to display the character at the currently-selected display position.

3.6 Keypad controller

A *keypad* consists of a set of buttons that may be pressed to provide input to an embedded system. Again, keypads are extremely common in embedded systems, since such systems may lack the keyboard that comes standard with desktop systems.

A simple keypad has buttons arranged in an N-column by M-row grid. The device has N outputs, each output corresponding to a column, and another M outputs, each output corresponding to a row. When we press a button, one column output and one row output go high, uniquely identifying the pressed button. To read such a keypad from software, we must scan the column and row outputs.

The scanning may instead be performed by a *keypad controller* (actually, such a device decodes rather than controls, but we'll call it a controller for consistency with the other peripherals discussed). A simple form of such a controller scans the column and row outputs of the keypad. When the controller detects a button press, it stores a code corresponding to that button into a register and sets an output high, indicating that a button has been pressed. Our software may poll this output every 100 milliseconds or so, and read the register when the output is high. Alternatively, this output can generate an interrupt on our general-purpose processor, eliminating the need for polling.

3.7 Stepper motor controller

A *stepper motor* is an electric motor that rotates a fixed number of degrees whenever we apply a "step" signal. In contrast, a regular electric motor rotates continuously whenever power is applied, coasting to a stop when power is removed. We specify a stepper motor either by the number of degrees in a single step, such as 1.8E, or by the number of steps required to move 360E, such as 200 steps. Stepper motors obviously abound in embedded systems with moving parts, such as disk drives, printers, photocopy and fax machines, robots, camcorders, VCRs, etc.

Internally, a stepper motor typically has four coils. To rotate the motor one step, we pass current through one or two of the coils; the particular coils depends on the present orientation of the motor. Thus, rotating the motor 360E requires applying current to the coils in a specified sequence. Applying the sequence in reverse causes reversed rotation.

In some cases, the stepper motor comes with four inputs corresponding to the four coils, and with documentation that includes a table indicating the proper input sequence. To control the motor from software, we must maintain this table in software, and write a step routine that applies high values to the inputs based on the table values that follow the previously-applied values.

In other cases, the stepper motor comes with a built-in controller (i.e., a special-purpose processor) implementing this sequence. Thus, we merely create a pulse on an input signal of the motor, causing the controller to generate the appropriate high signals to the coils that will cause the motor to rotate one step.

3.8 Analog-digital converters

An *analog-to-digital* converter (ADC, A/D or A2D) converts an analog signal to a digital signal, and a *digital-to-analog* converter (DAC, D/A or D2A) does the opposite. Such conversions are necessary because, while embedded systems deal with digital values, an embedded system's surroundings typically involve many analog signals. Analog refers to continuously-valued signal, such as temperature or speed represented by a voltage between 0 and 100, with infinite possible values in between. "Digital" refers to discretely-valued signals, such as integers, and in computing systems, these signals are encoded in binary. By converting between analog and digital signals, we can use digital processors in an analog environment.

For example, consider the analog signal of Figure 3.1(a). The analog input voltage varies over time from 1 to 4 Volts. We sample the signal at successive time units, and encode the current voltage into a 4-bit binary number. Conversely, consider Figure 3.1(b). We want to generate an analog output voltage for the given binary numbers over time. We generate the analog signal shown.

We can compute the digital values from the analog values, and vice-versa, using the following ratio:

$$\frac{e}{V_{\max}} = \frac{d}{2^n - 1}$$

V_{max} is the maximum voltage that the analog signal can assume, n is the number of bits available for the digital encoding, d is the present digital encoding, and e is the present analog voltage. This proportionality of the voltage and digital encoding is shown graphically in Figure 3.1(c). In our example of Figure 3.1, suppose V_{max} is 7.5V. Then for e = 5V, we have the following ratio: 5/7.5 = d/15, resulting in d = 1010 (ten), as shown in Figure 3.1(c). The *resolution* of a DAC or ADC is defined as $V_{max}/(2^n-1)$, representing the number of volts between successive digital encodings. The above discussion assumes a minimum voltage of 0V.

Internally, DACs possess simpler designs than ADCs. A DAC has n inputs for the digital encoding d, a V_{max} analog input, and an analog output e. A fairly straightforward circuit (involving resistors and an op-amp) can be used to convert d to e.

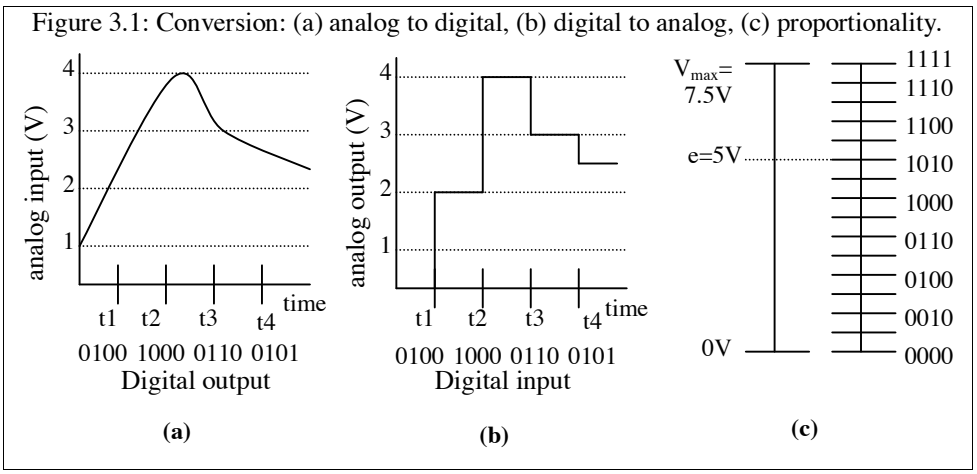


Figure 3.1: Conversion: (a) analog to digital, (b) digital to analog, (c) proportionality.

ADCs, on the other hand, require designs that are more complex, for the following reason. Given a V_{max} analog input and an analog input e, how does the converter know what binary value to assign in order to satisfy the above ratio? Unlike DACs, there is no simple analog circuit to compute d from e. Instead, an ADC may itself contain a DAC also connected to V_{max}. The ADC "guesses" an encoding d, and then evaluates its guess by inputting d into the DAC, and comparing the generated analog output e' with the original analog input e (using an analog comparator). If the two sufficiently match, then the ADC has found a proper encoding. So now the question remains: how do we guess the correct encoding?

This problem is analogous to the common computer-programming problem of finding an item in a list. One approach is sequential search, or "counting-up" in analog-digital terminology. In this approach, we start with an encoding of 0, then 1, then 2, etc., until we find a match. Unfortunately, while simple, this approach in the worst case (for high voltage values) requires 2^n comparisons, so it may be quite slow.

A faster solution uses what programmers call binary search, or "successive approximation" in analog-digital terminology. We start with an encoding corresponding half of the maximum. We then compare the resulting analog value with the original; if the resulting value is greater (less) than the original, we set the new encoding to halfway between this one and the maximum (minimum). We continue this process, dividing the possible encoding range in half at each step, until the compared voltages are equal. This technique requires at most n comparisons. However, it requires a more complex converter.

Because ADCs must guess the correct encoding, they require some time. Thus, in addition to the analog input and digital output, they include an input "start" that starts the conversion, and an ouput "done" to indicate that the conversion is complete.

3.9 Real-time clocks

Much like a digital wristwatch, a *real-time clock* (RTC) keeps the time and date in an embedded system. Read-time clocks are typically composed of a crystal-controlled oscillator, numerous cascaded counters, and a battery backup. The crystal-controlled oscillator generates a very consistent high-frequency digital pulse that feed the cascaded counters. The first counter, typically, counts these pulses up to the oscillator frequency, which corresponds to exactly one second. At this point, it generates a pulse that feeds the next counter. This counter counts up to 59, at which point it generates a pulse feeding the minute counter. The hour, date, month and year counters work in similar fashion. In addition, real-time clocks adjust for leap years. The rechargeable back-up battery is used to keep the real-time clock running while the system is powered off.

From the micro-controller's point of view, the content of these counters can be set to a desired value, (this corresponds to setting the clock), and retrieved. Communication

between the micro-controller and a real-time clock is accomplished through a serial bus, such as I^2C. It should be noted that, given a timer peripheral, it is possible to implement a real-time clock in software running on a processor. In fact, many systems use this approach to maintain the time. However, the drawback of such systems is that when the processor is shut down or reset, the time is lost.

3.10 Summary

Numerous single-purpose processors are manufactured to fulfill a specific function in a variety of embedded systems. These standard single-purpose processors may be fast, small, and have low unit and NRE costs. A timer informs us when a particular interval of time as passed, while a watchdog timer requires us to signal it within a particular interval to indicate that a program is running without error. A counter informs us when a particular number of pulses have occurred on a signal. A UART converts parallel data to serial data, and vice-versa. A PWM generates pulses on an output signal, with specific high and low times. An LCD controller simplifies the writing of characters to an LCD. A keypad controller simplifies capture and decoding of a button press. A stepper-motor controller assists us to rotate a stepper motor a fixed amount forwards or backwards. ADCs and DACs convert analog signals to digital, and vice-versa. A real-time clock keeps track of date and time. Most of these single-purpose processors could be implemented as software on a general-purpose processor, but such implementation can be burdensome. These processors thus simplify embedded system design tremendously. Many microcontrollers integrate these processors on-chip.

3.11 References and further reading

[1] Embedded Systems Programming. Includes information on a variety of single-purpose processors, such as programs for implementing or using timers and UARTs on microcontrollers.

[2] Microcontroller technology: the 68HC11. Peter Spasov. 2nd edition. ISBN 0-13-362724-1. Prentice Hall, Englewood Cliffs, NJ, 1996. Contains descriptions of principles and details for common 68HC11 peripherals.

3.12 Exercises

1. Given a clock frequency of 10 MHz, determine the number of clock cycles corresponding to a real-time interval of 100 ms.
2. A particular motor operates at 10 revolutions per second when its controlling input voltage is 3.7 V. Assume that you are using a microcontroller with a PWM whose output port can be set high (5 V) or low (0 V). (a) Compute the duty cycle necessary to obtain 10 revolutions per second. (b) Provide values for a pulse width and period that achieve this duty cycle.
3. Given a 120-step stepper motor with its own controller, write a C function *Rotate*(int degrees), which, given the desired rotation amount in degrees (between 0 and 360), pulses a microcontroller's output port the correct number of times to achieve the desired rotation.
4. Given an analog output signal whose voltage should range from 0 to 10 V, and a 8-bit digital encoding, provide the encodings for the following desired voltages: (a) 0 V, (b) 1 V, (c) 5.33 V, (d) 10 V. (e) What is the resolution of our conversion?
5. Given an analog input signal whose voltage ranges from 0 to 5 V, and an 8-bit digital encoding, calculate the correct encoding, and then trace the successive-approximation approach (i.e., list all the guessed encodings in the correct order) to finding the correct encoding.
6. Determine the values for *smod* and *TH1* to generate a baud rate of 9600 for the 8051 baud rate equation in the chapter, assuming an 11.981 MHz oscillator. Remember that *smod* is 2 bits and *TH1* is 8 bits.

Chapter 4 *Custom single-purpose processors: Hardware*

4.1 Introduction

As mentioned in the previous chapter, a single-purpose processor is a digital system intended to solve a specific computation task. While a manufacturer builds a standard single-purpose processor for use in a variety of applications, we build a custom single-purpose processor to execute a specific task within our embedded system. An embedded system designer choosing to use a custom single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits, similar to some of those of the previous chapter.

First, performance may be fast, due to fewer clock cycles resulting from a customized datapath, and due to shorter clock cycles resulting from simpler functional units, less multiplexors, or simpler control logic. Second, size may be small, due to a simpler datapath and no program memory. In fact, the processor may be faster and smaller than a standard one implementing the same functionality, since we can optimize the implementation for our particular task.

However, because we probably won't manufacture as many of the custom processor as a standard processor, we may not be able to invest as much NRE, unless the embedded system we are building will be sold in large quantities or does not have tight cost constraints. This fact could actually penalize performance and size.

In this chapter, we describe basic techniques for designing custom processors. We start with a review of combinational and sequential design, and then describe a method for converting programs to custom single-purpose processors.

4.2 Combinational logic design

A transistor is the basic electrical component of digital systems. Combinations of transistors form more abstract components called logic gates, which designers primarily use when building digital systems. Thus, we begin with a short description of transistors before discussing logic design.

A transistor acts as a simple on/off switch. One type of transistor (CMOS -- Complementary Metal Oxide Semiconductor) is shown in Figure 4.1(a). The *gate* (not to be confused with logic gate) controls whether or not current flows from the *source* to the *drain*. When a high voltage (typically +5 Volts, which we'll refer to as logic 1) is applied to the gate, the transistor conducts, so current flows. When low voltage (which we'll refer to as logic 0, typically ground, which is drawn as several horizontal lines of decreasing width) is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustrated in in Figure 4.1(b). When logic 0 is applied to the gate, the transistor conducts, and when logic 1 is applied, the transistor does not conduct. Given these two basic transistors, we can easily build a circuit whose output inverts its gate input, as shown in in Figure 4.1(c). When the input x is logic 0, the top transistor conducts (and the bottom does not), so logic 1 appears at the output F. We can also easily build a circuit whose output is logic 1 when at least one of its inputs is logic 0, as shown in Figure 4.1(d). When at least one of the inputs x and y is logic 0, then at least one of the top transistors conducts (and the bottom transistors do not), so logic 1 appears at F. If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom ones do, so logic 0 appears at F. Likewise, we can easily build a circuit whose output is logic 1 when both of its inputs are logic 0, as illustrated in Figure 4.1(e). The three circuits shown implement three basic logic gates: an inverter, a NAND gate, and a NOR gate.

Figure 4.1: CMOS transistor implementations of some basic logic gates: (a) nMOS transistor, (b) pMOS transistor, (c) inverter, (d) NAND gate, (e) NOR gate.

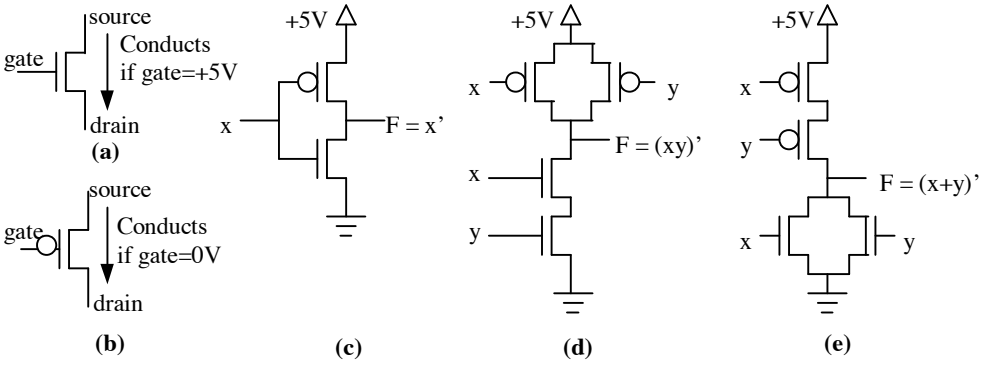
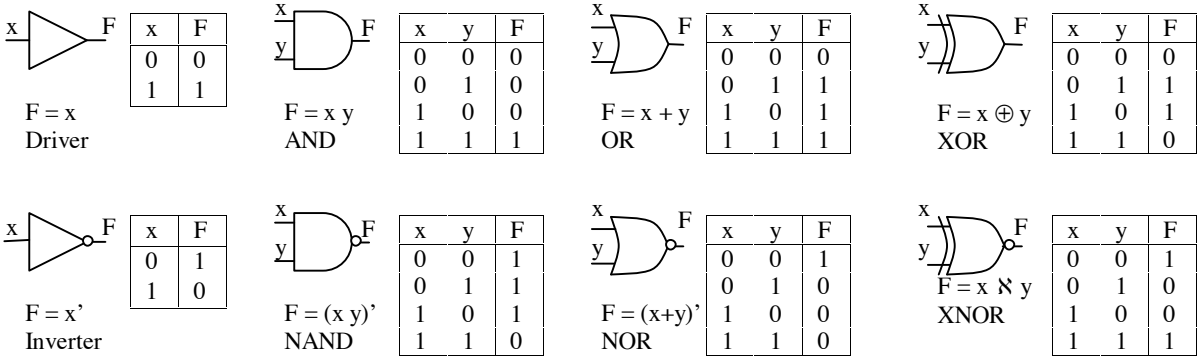


Figure 4.2: Basic logic gates



x	F
0	0
1	1

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

x	F
0	1
1	0

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

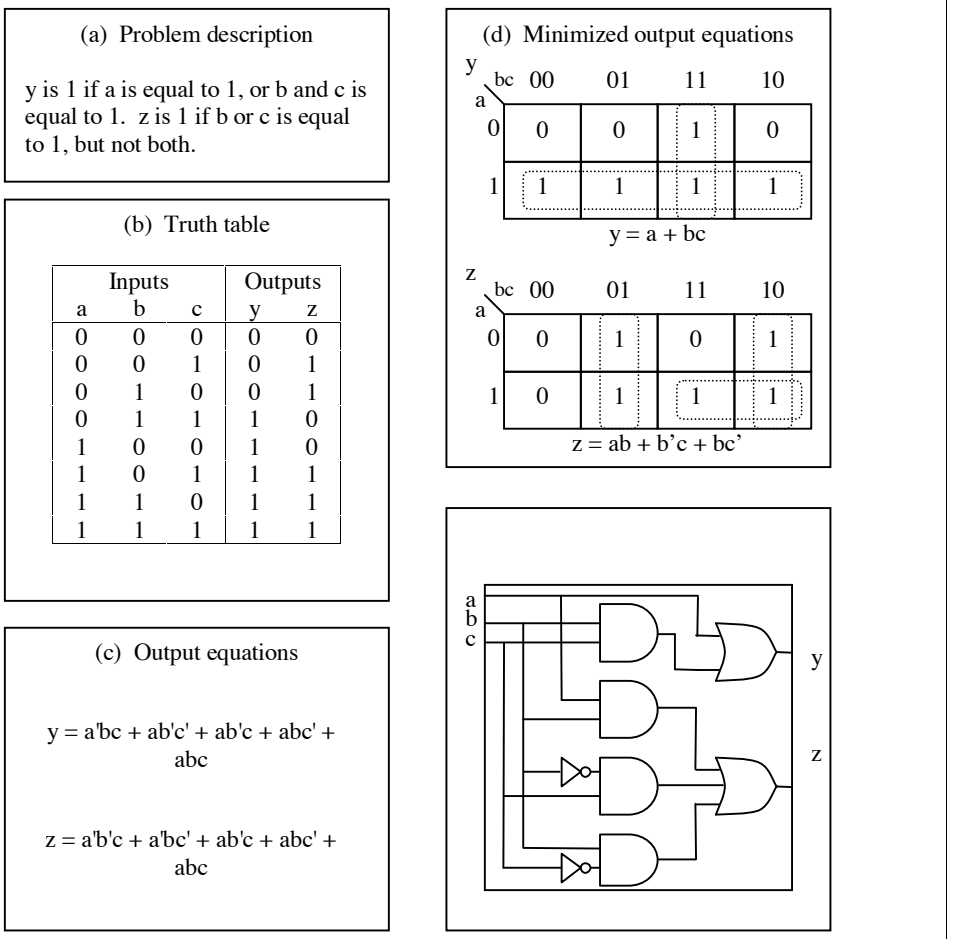
x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

Digital system designers usually work with logic gates, not transistors. Figure 4.2 describes 8 basic logic gates. Each gate is represented symbolically, with a Boolean equation, and with a truth table. The truth table has inputs on the left, and output on the right. The AND gate outputs 1 if and only if both inputs are 1. The OR gate outputs 1 if and only if at least one of the inputs is 1. The XOR (exclusive-OR) gate outputs 1 if and only if exactly one of its two inputs is 1. The NAND, NOR, and XNOR gates output the complement of AND, OR, and XOR, respectively. As you might have noticed from our transistor implementations, the NAND and NOR gates are actually simpler to build than AND and OR gates.

A *combinational* circuit is a digital circuit whose output is purely a function of its current inputs; such a circuit has no memory of past inputs. We can apply a simple technique to design a combinational circuit using our basic logic gates, as illustrated in Figure 4.3. We start with a problem description, which describes the outputs in terms of the inputs. We translate that description to a truth table, with all possible combinations of input values on the left, and desired output values on the right. For each output column, we can derive an output equation, with one term per row. However, we often want to minimize the logic gates in the circuit. We can minimize the output equations by algebraically manipulating the equations. Alternatively, we can use Karnaugh maps, as

Figure 4.3: Combinational logic design.

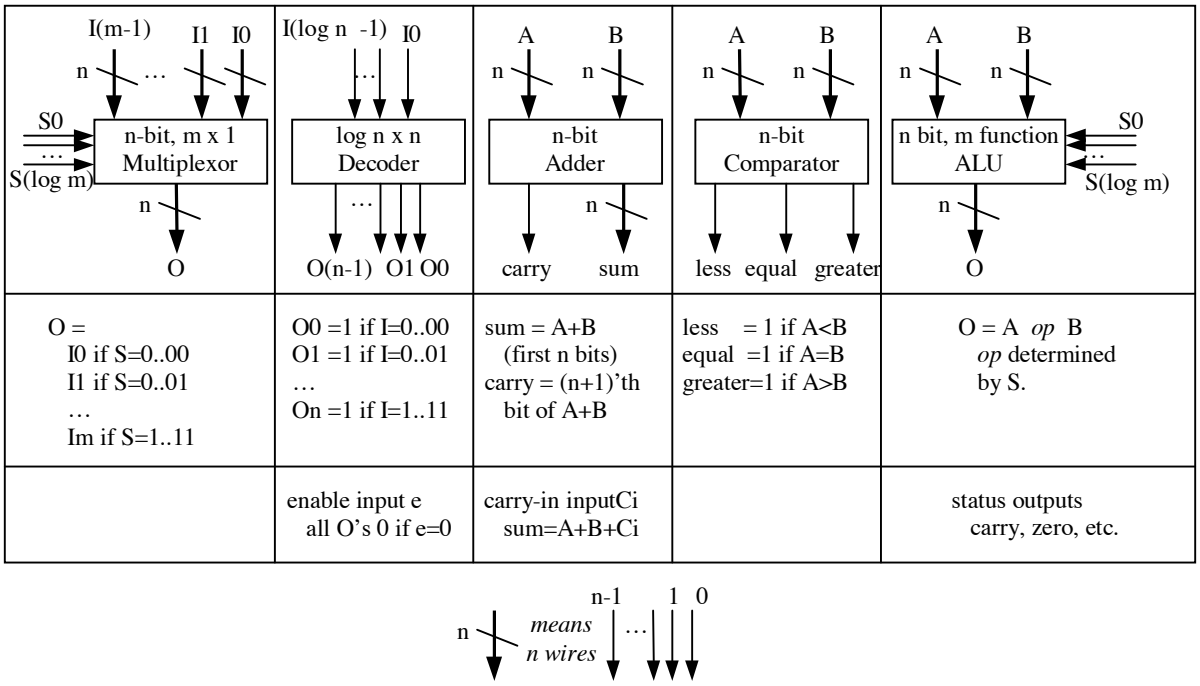


shown in the figure. Once we've obtained the desired output equations (minimized or not), we can draw the circuit diagram.

Although we can design all combinational circuits in the above manner, large circuits would be very complex to design. For example, a circuit with 16 inputs would have 2^{16}, or 64K, rows in its truth table. One way to reduce the complexity is to use components that are more abstract than logic gates. Figure 4.4 shows several such combinational components. We now describe each briefly.

A *multiplexor*, sometimes called a selector, allows only one of its data inputs *Im* to pass through to the output *O*. Thus, a multiplexor acts much like a railroad switch, allowing only one of multiple input tracks to connect to a single output track. If there are *m* data inputs, then there are $log_2(m)$ select lines *S*, and we call this an m-by-1 multiplexor (*m* data inputs, one data output). The binary value of *S* determines which data input passes through; 00...00 means *I0* may pass, 00...01 means *I1* may pass, 00...10 means *I2* may pass, and so on. For example, an 8x1 multiplexor has 8 data inputs and thus 3 select lines. If those three select lines have values of 110, then *I6* will pass through to the output. So if *I6* is 1, then the output would be 1; if *I6* is 0, then the output would be 0. We commonly use a more complex device called an *n*-bit multiplexor, in which each data input, as well as the output, consists of *n* lines. Suppose the previous example used a 4-bit 8x1 multiplexor. Thus, if *I6* is 0110, then the output would be 0110. Note that *n* does not affect the number of select lines.

Figure 4.4: Combinational components.



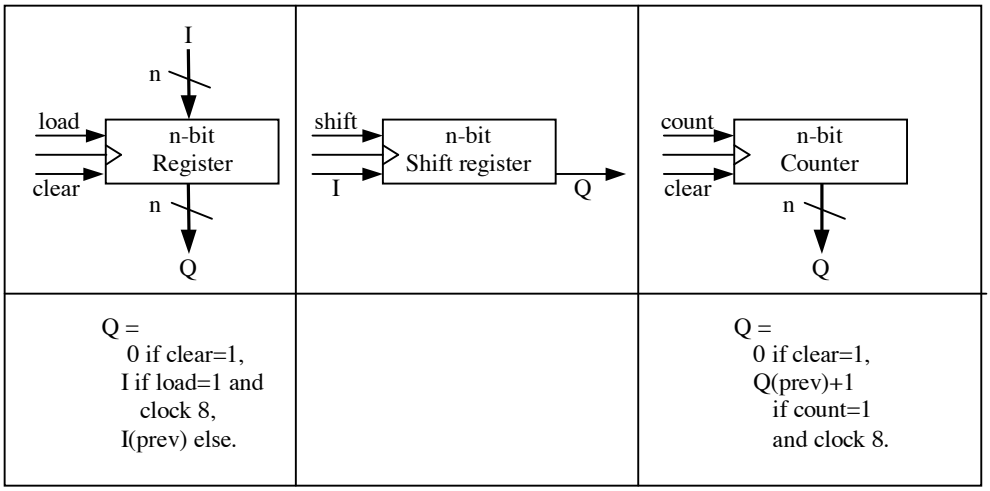
A *decoder* converts its binary input *I* into a one-hot output *O*. "One-hot" means that exactly one of the output lines can be 1 at a given time. Thus, if there are *n* outputs, then there must be $log_2(n)$ inputs. We call this a $log_2(n)xn$ decoder. For example, a 3x8 decoder has 3 inputs and 8 outputs. If the input is 000, then the output *O0* will be 1. If the input is 001, then the output *O1* would be 1, and so on. A common feature on a decoder is an extra input called *enable*. When enable is 0, all outputs are 0. When enable is 1, the decoder functions as before.

An *adder* adds two n-bit binary inputs *A* and *B,* generating an n-bit output *sum* along with an output *carry*. For example, a 4-bit adder would have a 4-bit *A* input, a 4-bit *B* input, a 4-bit *sum* output, and a 1-bit *carry* output. If *A* is 1010 and *B* is 1001, then *sum* would be 0011 and *carry* would be 1.

A *comparator* compares two n-bit binary inputs *A* and *B*, generating outputs that indicate whether *A* is *less* than, *equal* to, or *greater* than *B*. If *A* is 1010 and *B* is 1001, then *less* would be 0, *equal* would be 0, and *greater* would be 1.

An *ALU* (arithmetic-logic unit) can perform a variety of arithmetic and logic functions on its n-bit inputs *A* and *B*. The select lines *S* choose the current function; if there are *m* possible functions, then there must be at least $log_2(m)$ select lines. Common functions include addition, subtraction, AND, and OR.

Figure 4.5: Sequential components.



4.3 Sequential logic design

A *sequential circuit* is a digital circuit whose outputs are a function of the current as well as previous input values. In other words, sequential logic possesses memory. One of the most basic sequential circuits is the *flip-flop*. A flip-flop stores a single bit. The simplest type of flip-flop is the D flip-flop. It has two inputs: *D* and *clock*. When *clock* is 1, the value of *D* is stored in the flip-flop, and that value appears at an output *Q*. When *clock* is 0, the value of D is ignored; the output *Q* maintains its value. Another type of flip-flop is the SR flip-flop, which has three inputs: *S*, *R* and *clock*. When *clock* is 0, the previously stored bit is maintained and appears at output *Q*. When *clock* is 1, the inputs S and R are examined. If *S* is 1, a 1 is stored. If R is 1, a 0 is stored. If both are 0, there's no change. If both are 1, behavior is undefined. Thus, *S* stands for set, and *R* for reset. Another flip-flop type is a JK flip-flop, which is the same as an SR flip-flop except that when both *J* and *K* are 1, the stored bit toggles from 1 to 0 or 0 to 1. To prevent unexpected behavior from signal glitches, flip-flops are typically designed to be *edge-triggered*, meaning they only pay attention to their non-clock inputs when the clock is *rising* from 0 to 1, or alternatively when the clock is *falling* from 1 to 0.

Just as we used more abstract combinational components to implement complex combinational systems, we also use more abstract sequential components for complex sequential systems. Figure 4.5 illustrates several sequential components, which we now describe.

A *register* stores n bits from its n-bit data input *I*, with those stored bits appearing at its output *O*. A register usually has at least two control inputs, *clock* and *load*. For a rising-edge-triggered register, the inputs *I* are only stored when *load* is 1 and *clock* is rising from 0 to 1. The clock input is usually drawn as a small triangle, as shown in the figure. Another common register control input is *clear*, which resets all bits to 0, regardless of the value of *I*. Because all n bits of the register can be stored in parallel, we often refer to this type of register as a parallel-load register, to distinguish it from a shift register, which we now describe.

A *shift register* stores n bits, but these bits cannot be stored in parallel. Instead, they must be shifted into the register serially, meaning one bit per clock edge. A shift register has a one-bit data input *I*, and at least two control inputs *clock* and *shift*. When *clock* is rising and *shift* is 1, the value of *I* is stored in the (n)'th bit, while the (n)'th bit is stored in the (n-1)'th bit, and likewise, until the second bit is stored in the first bit. The first bit is typically shifted out, meaning it appears over an output *Q*.

A *counter* is a register that can also increment (add binary 1) to its stored binary value. In its simplest form, a counter has a *clear* input, which resets all stored bits to 0,

and a *count* input, which enables incrementing on the clock edge. A counter often also has a parallel load data input and associated control signal. A common counter feature is both up and down counting (incrementing and decrementing), requiring an additional control input to indicate the count direction.

The control inputs discussed above can be either synchronous or asynchronous. A *synchronous* input's value only has an effect during a clock edge. An *asynchronous* input's value affects the circuit independent of the clock. Typically, *clear* control lines are asynchronous.

Sequential logic design can be achieved using a straightforward technique, whose steps are illustrated in Figure 4.1. We again start with a problem description. We translate this description to a state diagram. We describe state diagrams further in a later chapter. Briefly, each state represents the current "mode" of the circuit, serving as the circuit's memory of past input values. The desired output values are listed next to each state. The input conditions that cause a transistion from one state to another are shown next to each arc. Each arc condition is implicitly AND'ed with a rising (or falling) clock edge. In other words, all inputs are synchronous. State diagrams can also describe asynchronous systems, but we do not cover such systems in this book, since they are not common.

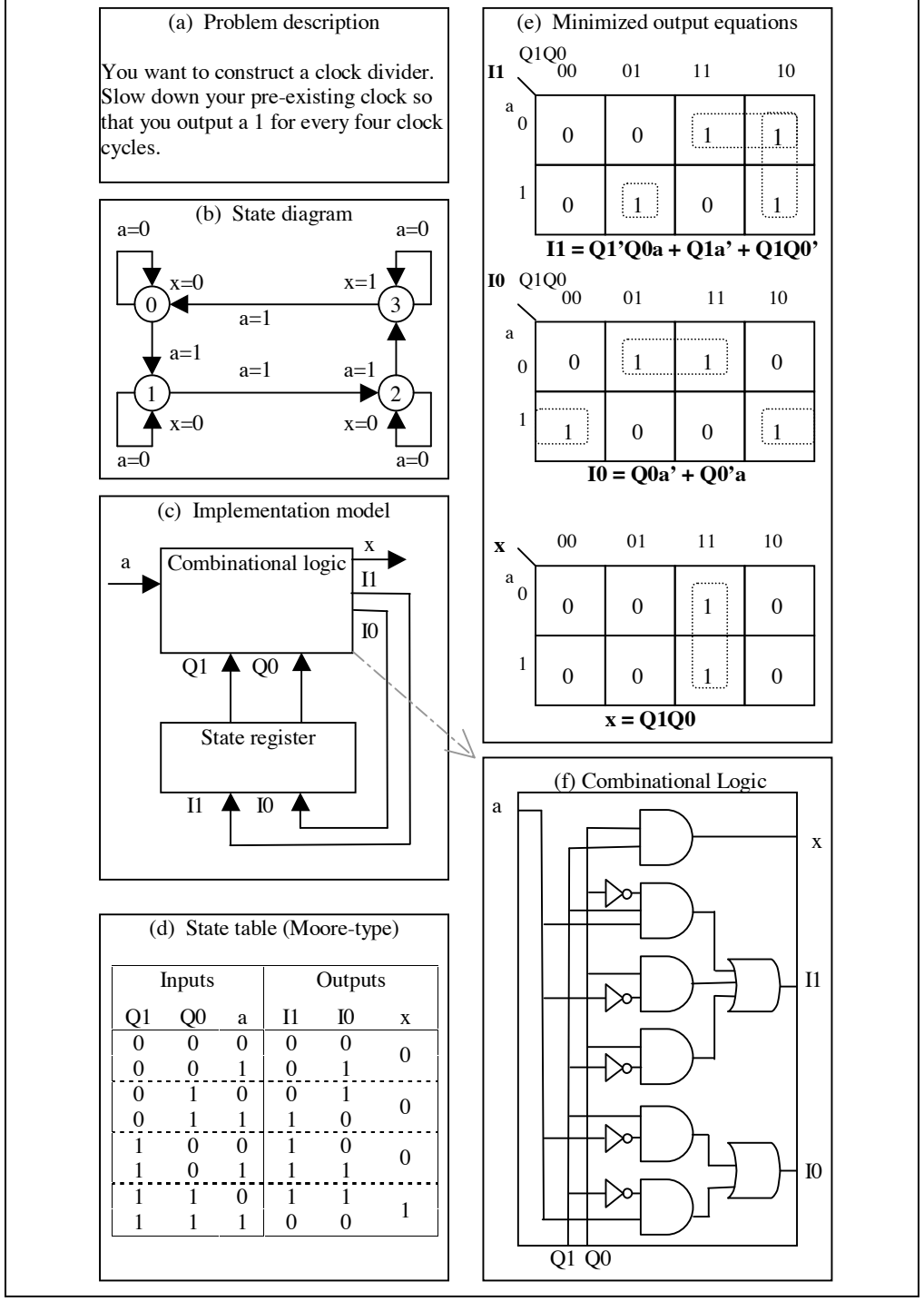
We will implement this state diagram using a register to store the current state, and combinational logic to generate the output values and the next state. We assign each state with a unique binary value, and we then create a truth table for the combinational logic. The inputs for the combinational logic are the state bits coming from the state register, and the external inputs, so we list all combinations of these inputs on the left side of the table. The outputs for the combinational logic are the state bits to be loaded into the register on the next clock edge (the next state), and the external output values, so we list desired values of these outputs for each input combination on the right side of the table. Because we used a state diagram for which outputs were a function of the current state only, and not of the inputs, we list an external output value only for each possible state, ignoring the external input values. Now that we have a truth table, we proceed with combinational logic design as described earlier, by generating minimized output equations, and then drawing the combinational logic circuit.

4.4 Custom single-purpose processor design

We can apply the above combinational and sequential logic design techniques to build datapath components and controllers. Therefore, we have nearly all the knowledge we need to build a custom single-purpose processor for a given program, since a processor consists of a controller and a datapath. We now describe a technique for building such a processor.

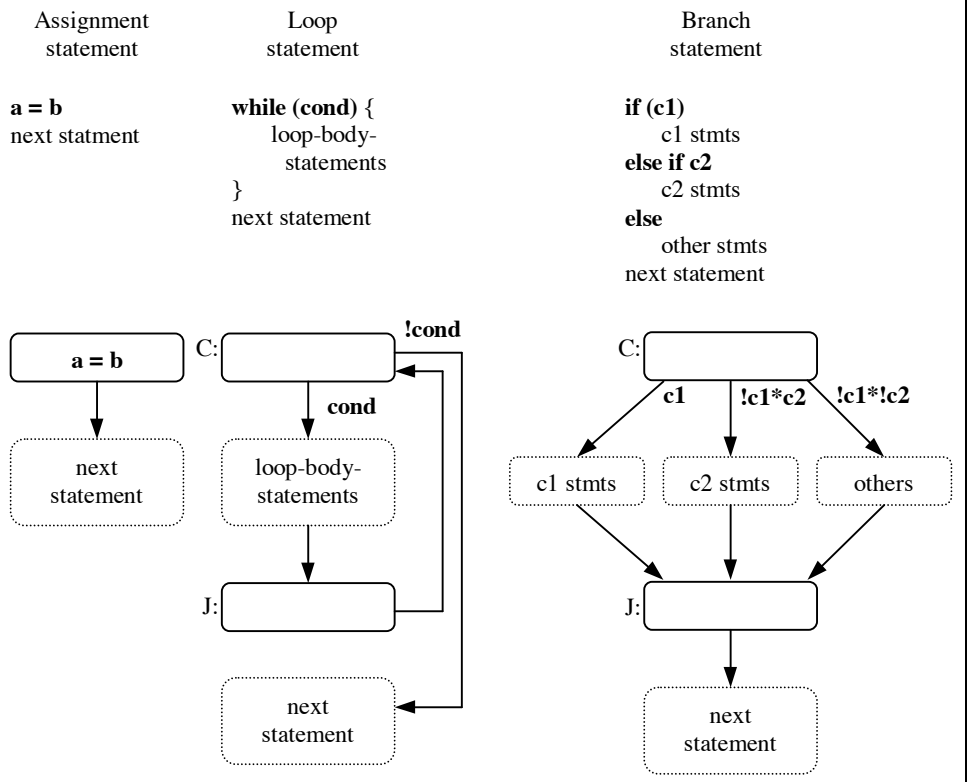
We begin with a sequential program we must implement. Figure 4.3 provides a example based on computing a greatest common divisor (GCD). Figure 4.3(a) shows a black-box diagram of the desired system, having *x_i* and *y_i* data inputs and a data output *d_i*. The system's functionality is straightforward: the output should represent the GCD of the inputs. Thus, if the inputs are 12 and 8, the output should be 4. If the inputs are 13 and 5, the output should be 1. Figure 4.3(b) provides a simple program with this functionality. The reader might trace this program's execution on the above examples to verify that the program does indeed compute the GCD.

Figure 4.1: Sequential logic design.



To begin building our single-purpose processor implementing the GCD program, we first convert our program into a complex state diagram, in which states and arcs may include arithmetics expressions, and these expressions may use external inputs and outputs or variables. In contrast, our earlier state diagrams only included boolean expressions, and these expressions could only use external inputs and outputs, not

Figure 4.2: Templates for creating a state diagram from a program.



variables. Thus, these more complex state diagram looks like a sequential program in which statements have been scheduled into states.

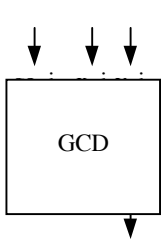
We can use templates to convert a program to a state diagram, as illustrated in Figure 4.2. First, we classify each statement as an assignment statement, loop statement, or branch (if-then-else or case) statement. For an assignment statement, we create a state with that statement as its action. We add an arc from this state to the state for the next statement, whatever type it may be. For a loop statement, we create a condition state *C* and a join state *J*, both with no actions. We add an arc with the loop's condition from the condition state to the first statement in the loop body. We add a second arc with the complement of the loop's condition from the condition state to the next statement after the loop body. We also add an arc from the join state back to the condition state. For a branch statement, we create a condition state *C* and a join state *J*, both with no actions. We add an arc with the first branch's condition from the condition state to the branch's first statement. We add another arc with the complement of the first branch's condition AND'ed with the second branches condition from the condition state to the branches first statement. We repeat this for each branch. Finally, we connect the arc leaving the last statement of each branch to the join state, and we add an arc from this state to the next statement's state.

Using this template approach, we convert our GCD program to the complex state diagram of Figure 4.3(c).

We are now well on our way to designing a custom single-purpose processor that executes the GCD program. Our next step is to divide the functionality into a datapath part and a controller part, as shown in Figure 4.4. The datapath part should consist of an interconnection of combinational and sequential components. The controller part should consist of a basic state diagram, i.e., one containing only boolean actions and conditions.

We construct the datapath through a four-step process:

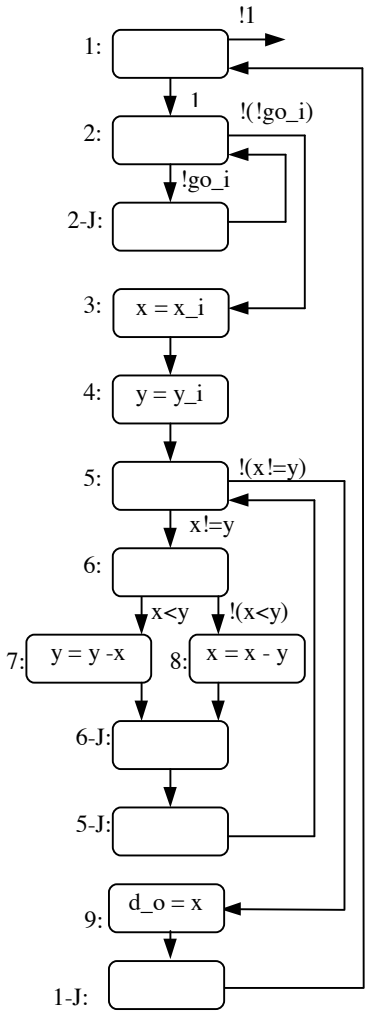
Figure 4.3: Example program -- Greatest Common Divisor (GCD): (a) black-box view, (b) desired functionality, (c) state diagram



(a)

```
0: vectorN  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:      if  (x < y)
7:         y = y - x;
       else
8:         x = x - y;
     }
9:   d_o = x;
   }
```

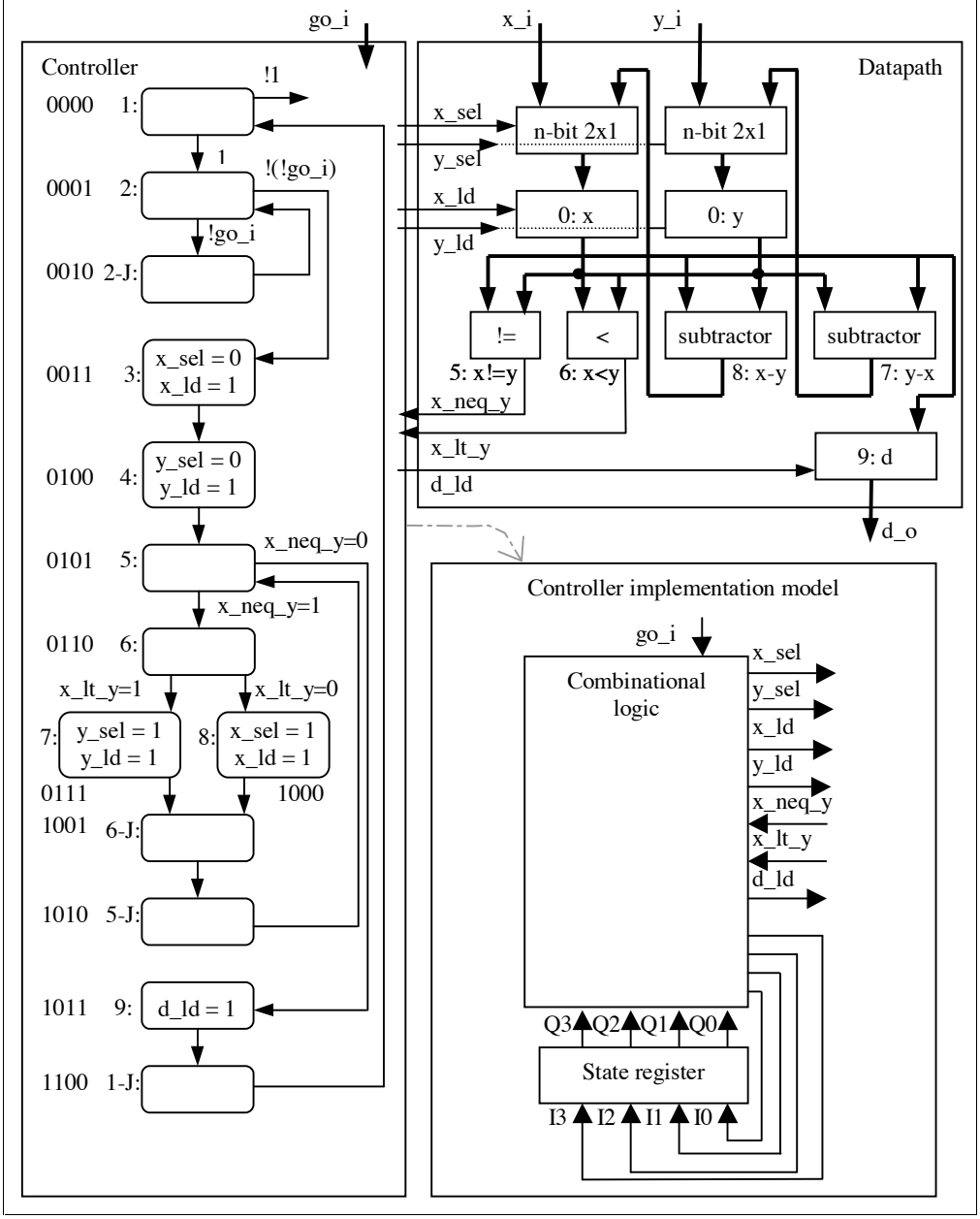
(b)



(c)

1. First, we create a register for any declared variable. In the example, these are *x* and *y*. We treat an output port as having an implicit variable, so we create a register *d* and connect it to the output port. We also draw the input and output ports.
2. Second, we create a functional unit for each arithmetic operation in the state diagram. In the example, there are two subtractions, one comparison for less than, and one comparison for inequality, yielding two subtractors and two comparators, as shown in the figure.
3. Third, we connect the ports, registers and functional units. For each write to a variable in the state diagram, we draw a connection from the write's source (an input port, a functional unit, or another register) to the variable's register. For each arithmetic and logical operation, we connect sources to an input of the operation's corresponding functional unit. When more than one source is connected to a register, we add an appropriately-sized multiplexor.
4. Finally, we create a unique identifier for each control input and output of the datapath components.

Figure 4.4: Example after splitting into a controller and a datapath.



Now that we have a complete datapath, we can build a state diagram for our controller. The state diagram has the same structure as the complex state diagram. However, we replace complex actions and conditions by boolean ones, making use of our datapath. We replace every variable write by actions that set the select signals of the multiplexor in front of the variable's register's such that the write's source passes through, and we assert the load signal of that register. We replace every logical operation in a condition by the corresponding functional unit control output.

We can then complete the controller design by implementing the state diagram using our sequential design technique described earlier. Figure 4.4 shows the controller implementation model, and Figure 4.5 shows a state table.

Note that there are 7 inputs to the controller, resulting in 128 rows for the table. We reduced rows in the state table by using don't cares for some input combinations, but we

Figure 4.5: State table for the GCD example.

State table

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

* - indicates all possible combinations of 0's and 1's
X - indicates don't cares

can still see that optimizing the design using hand techniques could be quite tedious. For this reason, computer-aided design (CAD) tools that automate the combinational as well as sequential logic design can be very helpful; we'll introduce such CAD tools in a later chapter.

Also, note that we could perform significant amounts of optimization to both the datapath and the controller. For example, we could merge functional units in the datapath, resulting in fewer units at the expense of more multiplexors. We could also merge states in the datapath. Such optimizations will be discussed in a later chapter.

Remember that we could alternatively implement the GCD program by programming a microcontroller, thus eliminating the need for this design process, but possibly yielding a slower and bigger design.

4.5 Summary

Designing a custom single-purpose processor for a given program requires an understanding of various aspects of digital design. Design of a circuit to implement boolean functions requires combinational design, which consists of building a truth table with all possible inputs and desired outputs, optimizing, and drawing a circuit. Design of a circuit to implement a state diagram requires sequential design, which consists of drawing an implementation model with a state register and a combinational logic block, assigning a binary encoding to each state, drawing a state table with inputs and outputs, and repeating our combinational design process for this table. Finally, design of a single-purpose processor circuit to implement a program requires us to first schedule the program's statements into a complex state diagram, construct a datapath from the diagram, create a new state diagram that replaces complex actions and conditions by datapath control operations, and then design a controller circuit for the new state diagram

using sequential design. Because processors can be complex, CAD tools would be a great designer's aid.

4.6 References and further reading

Gajski, Daniel D. *Principles of Digital Design.* New Jersey: Prentice-Hall, 1997. ISBN 0-13-301144-5. Describes combinational and sequential logic design, with a focus on optimization techniques, CAD, and higher-levels of design.

Katz, Randy. *Contemporary Logic Design.* Redwood City, California: Benjamin/Cummings, 1994. ISBN 0-8053-2703-7. Describes combinational and sequential logic design, with a focus on logic and sequential optimization and CAD.

4.7 Exercises

1. Build a 3-input NAND gate using a minimum number of CMOS transistors.
2. Design a 2-bit comparator (compares two 2-bit words) with a single output "less-than," using the combinational design technique described in the chapter. Start from a truth table, use K-maps to minimize logic, and draw the final circuit.
3. Design a 3-bit counter that counts the following sequence: 1, 2, 4, 5, 7, 1, 2, ...,. This counter has an output "odd" that is one when the current count value is odd. Use the sequential design technique of the chapter. Start from a state diagram, draw the state table, minimize logic, and draw the final circuit.
4. Compare the GCD custom-processor implementation to a software implementation. (a) Compare the performance. Assume a 100 ns clock for the microcontroller, and a 20 ns clock for the custom processor. Assume the microcontroller uses two operand instructions, and each instruction requires 4 clock cycles. Estimates for the microcontroller are fine. (b) Estimate the number of gates for the custom design, and compare this to 10,000 gates for a simple 8-bit microcontroller. (c) Compare the custom GCD with the GCD running on a 300 MHz processor with 2-operand instructions and 1 clock cycle per instruction (advanced processors use parallelism to meet or exceed 1 cycle per instruction). (d) Compare the estimated gates with 200,000 gates, a typical number of gates for a modern 32-bit processor.
5. Design a custom single-purpose processor implementing the following program, using the technique of the chapter. Start with a complex state diagram, construct a datapath and a simplified state diagram, and draw the truth table for the controller, but do not complete the design for the controller beyond the truth table.

```
input_port U;
int V;
for (int i=0; i<32; i++)
  V = V + U*V;
```

Chapter 5 *Memories*

5.1 Introduction

Any embedded system's functionality consists of three aspects: processing, storage, and communication. Processing is the transformation of data, storage is the retention of data for later use, and communication is the transfer of data. Each of these aspects must be implemented. We use *processors* to implement processing, *memories* to implement storage, and *buses* to implement communication. The earlier chapters described common processor types: general-purpose processors, standard single-purpose processors, and custom single-purpose processors. This chapter describes memories.

A memory stores large numbers of bits. These bits exist as m words of n bits each, for a total of $m*n$ bits. We refer to a memory as an $m \times n$ ("m-by-n") memory. $Log_2(m)$ address input signals are necessary to identify a particular word. Stated another way, if a memory has k address inputs, it can have up to 2^k words. n signals are necessary to output (and possibly input) a selected word. To *read* a memory means to retrieve the word of a particular address, while to *write* a memory means to store a word in a particular address. Some memories can only be read from (ROM), while others can be both read from and written to (RAM). There isn't much demand for a memory that can only be written to (what purpose would such a memory serve?). Most memories have an enable input; when this enable is low, the address is ignored, and no data is written to or read from the memory.

5.2 Read-only memory -- ROM

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

We can use ROM for various purposes. One use is to store a software program for a general-purpose processor. We may write each program instruction to one ROM word. For some processors, we write each instruction to several ROM words. For other processors, we may pack several instructions into a single ROM word. A related use is to store constant data, like large lookup tables of strings or numbers.

Another common use is to implement a combinational circuit. We can implement any combinational function of k variables by using a $2^k \times 1$ ROM, and we can implement n functions of the same k variables using a $2^k \times n$ ROM. We simply program the ROM to implement the truth table for the functions, as shown in Figure 2.

Figure 3 provides a symbolic view of the internal design of an 8x4 ROM. To the right of the 3x8 decoder in the figure is a grid of lines, with word lines running horizontally and data lines vertically; lines that cross without a circle in the figure are *not*

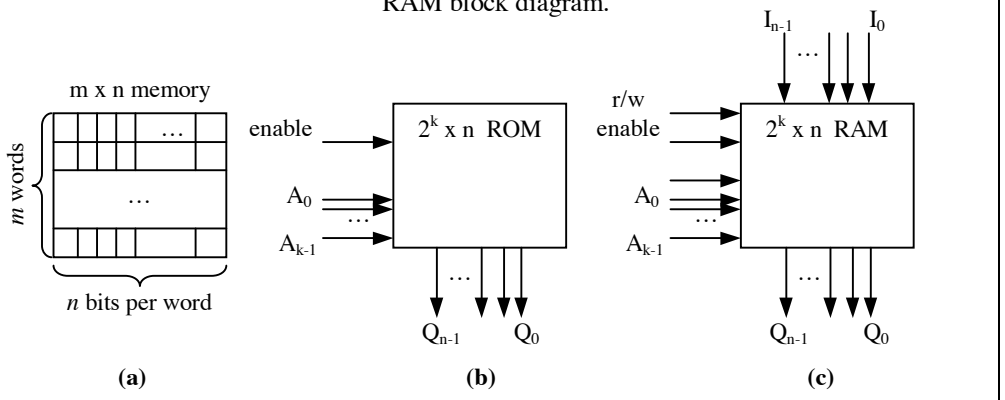


Figure 5.1: Basic memories: (a) words and bits per word, (b) ROM block diagram, (c) RAM block diagram.

connected. Thus, word lines only connect to data lines via the programmable connection lines shown. The figure shows all connection lines in place except for two connections in word 2. To see how this device acts as a read-only memory, consider an input address of "010." The decoder will thus set word 2's line to 1. Because the lines connecting this word line with data lines 2 and 0 do not exist, the ROM output will read "1010." Note that if the ROM enable input is 0, then no word is read. Also note that each data line is shown as a wired-OR, meaning that the wire itself acts to logically OR all the connections to it.

How do we program the programmable connections? The answer depends on the type of ROM being used. In a *mask-programmed* ROM, the connection is made when the chip is being fabricated (by creating an appropriate set of masks). Such ROM types are typically only used in high-volume systems, and only after a final design has been determined.

Most other systems use user-programmable ROM devices, or *PROM*, which can be programmed by the chip's user, well after the chip has been manufactured. These devices are better suited to prototyping and to low-volume applications. To program a PROM device, the user provides a file indicating the desired ROM contents. A piece of equipment called a ROM programmer (note: the programmer is a piece of equipment, not a person who writes software) then configures each programmable connection according to the file. A basic PROM uses a fuse for each programmable connection. The ROM programmer blows fuses by passing a large current wherever a connection should not exist. However, once a fuse is blown, the connection can never be re-established. For this reason, basic PROM is often referred to as one-time-programmable device, or *OTP*.

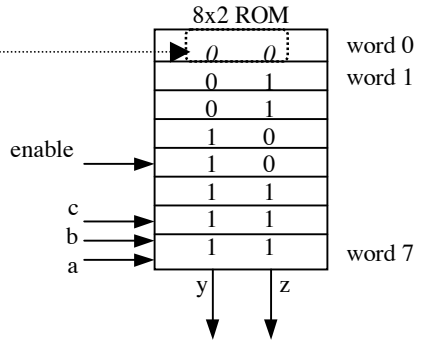
Another type of PROM is an *erasable* PROM, or *EPROM*. This device uses a MOS transistor as its programmable component. The transistor has a "floating gate," meaning its gate is not connected. An EPROM programmer injects electrons into the floating gate, using higher than normal voltage (usually 12V to 25V) that causes electrons to "tunnel" into the gate. When that high voltage is removed, the electrons can not escape, and hence

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

Truth table (ROM contents)

Inputs (address)			Outputs	
a	b	c	y	z
0	0	0	*0*	*0*
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

(a)



(b)

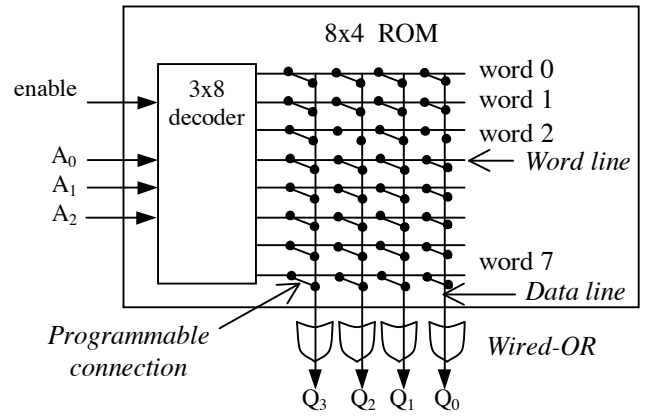
the gate has been charged and programming has occurred. Standard EPROMs are guaranteed to hold their programs for at least 10 years. To erase the program, the electrons must be excited enough to escape from the gate. Ultra-violet (UV) light is used to fulfil this role of erasing. The device must be placed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

Electrically-erasable PROM, or *EEPROM*, is designed to eliminate the time-consuming and sometimes impossible requirement of exposing an EPROM to UV light to erase the ROM. An EEPROM is not only programmed electronically, but is also erased electronically. These devices are typically more expensive the EPROM's, but far more convenient to use. EEPROM's are often called "E-squared's" for short. *Flash* memory is a type of EEPROM in which reprogramming can be done to certain regions of the memory, rather than the entire memory at once.

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

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

Figure 5.3: ROM internals



5.3 Read-write memory -- RAM

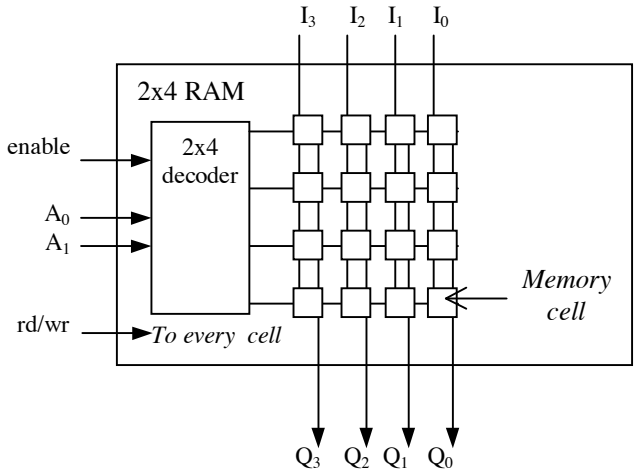
RAM, or random-access memory, is a memory that can be both read and written. In contrast to ROM, a RAM's content is not "programmed" before being inserted into an embedded system. Instead, the RAM contains no data when inserted in the embedded system; the system writes data to and then reads data from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

A RAM's internal structure is somewhat more complex than a ROM's, as shown in Figure 4, which illustrates a 4x4 RAM (note: RAMs typically have thousands of words, not just 4 as in the figure). Each word consists of a number of memory cells, each storing one bit. In the figure, each input data connects to every cell in its column. Likewise, each output data line connects to every cell in its column, with the output of a memory cell being OR'ed with the output data line from above. Each word enable line from the decoder connects to every cell its its row. The read/write input (rd/wr) is assumed to be connected to every cell. The memory cell must possess logic such that it stores the input data bit when rd/wr indicates write and the row is enabled, and such that it outputs this bit when rd/wr indicates read and the row is enabled.

There are two basic types of RAM, static and dynamic. Static RAM is faster but bigger than dynamic RAM. *Static* RAM, or *SRAM*, uses a memory cell consisting of a flip-flop to store a bit. Each bit thus requires about 6 transistors. This RAM type is called static because it will hold its data as long as power is supplied, in contrast to dynamic RAM. Static RAM is typically used for high-performance parts of a system (e.g., cache).

Dynamic RAM, or *DRAM*, uses a memory cell consisting of a MOS transistor and capacitor to store a bit. Each bit thus requires only 1 transistor, resulting in more compact memory than SRAM. However, the charge stored in the capacitor leaks gradually, leading to discharge and eventually to loss of data. To prevent loss of data, each cell must regularly have its charge "refreshed." A typical DRAM cell minimum refresh rate is once

Figure 5.4: RAM internals



every 15.625 microseconds. Because of the way DRAMs are designed, reading a DRAM word refreshes that word's cells. In particular, accessing a DRAM word results in the word's data being stored in a buffer and then being written back to the word's cells. DRAMs tend to be slower to access than SRAMs.

Many RAM variations exist. Pseudo-Static RAMs, or PSRAMs, are DRAMs with a refresh controller built-in. Thus, since the RAM user need not worry about refreshing, the device appears to behave much like an SRAM. However, in contrast to true SRAM, a PSRAM may be busy refreshing itself when accessed, which could slow access time and add some system complexity. Nevertheless, PSRAM is a popular low-cost alternative to SRAM in many embedded systems.

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

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

A common is question is: where does the term "random-access" come from in random-access memory? RAM should really be called read-write memory, to contrast it from read-only memory. However, when RAM was first introduced, it was in stark contrast to the then common sequentially-accessed memory media, like magnetic tapes or drums. These media required that the particular location to be accessed be positioned under an access device (e.g., a head). To access another location not immediately adjacent to the current location on the media, one would have sequence through a number of other locations, e.g., for a tape, one would have to rewind or fast-forward the tape. In contrast, with RAM, any "random" memory location could be accessed in the same amount of time as any other location, regardless of the previously read location. This random-access feature was the key distinguishing feature of this memory type at the time of its introduction, and the name has stuck even today.

5.4 Composing memories

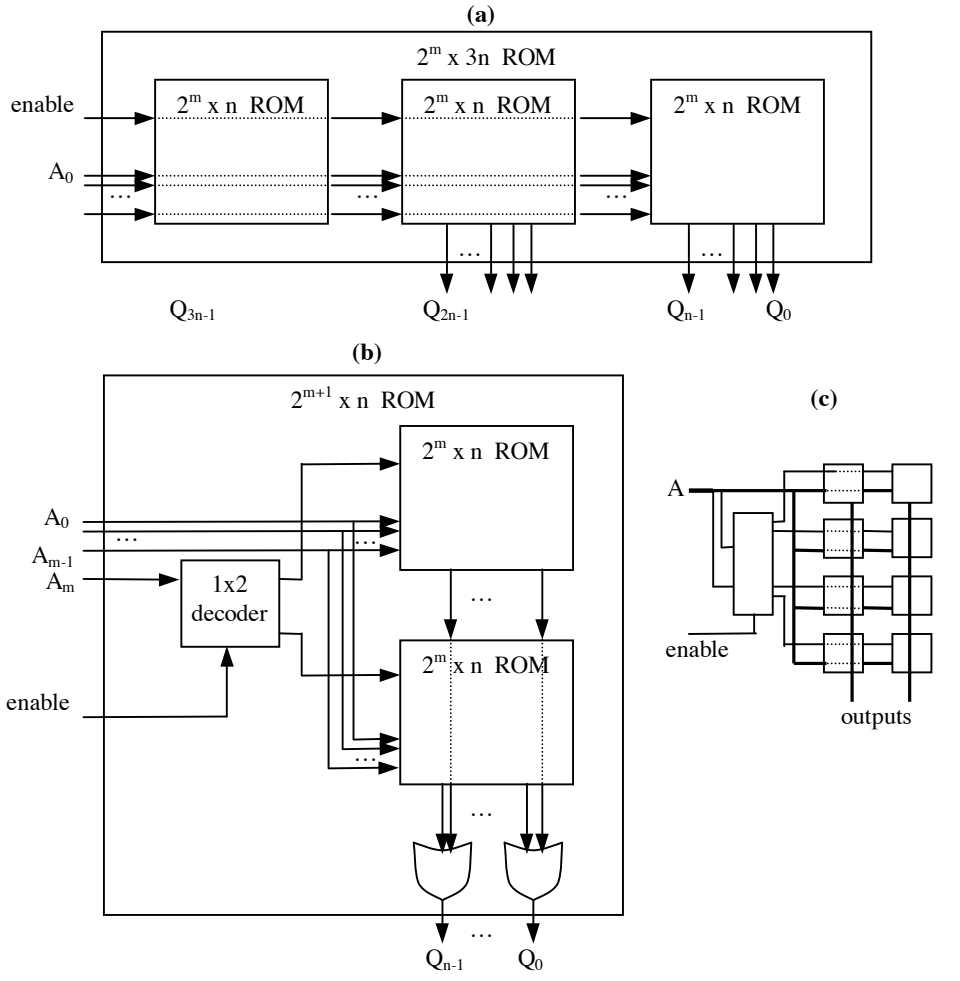
An embedded system designer is often faced with the situation of needing a particular-sized memory (ROM or RAM), but having readily available memories of a different size. For example, the designer may need a 2k x 8 ROM, but may have 4k x 16 ROMs readily available. Alternatively, the designer may need a 4k x 16 ROM, but may have 2k x 8 ROMs available for use.

The case where the available memory is larger than needed is easy to deal with. We simply use the needed lower words in the memory, thus ignoring unneeded higher words and their high-order address bits, and we use the lower data input/output lines, thus ignoring unneeded higher data lines. (Of course, we could use the higher data lines and ignore the lower lines instead).

The case where the available memory is smaller than needed requires more design effort. In this case, we must compose several smaller memories to behave as the larger memory we need. Suppose the available memories have the correct number of words, but each word is not wide enough. In this case, we can simply connect the available memories side-by-side. For example, Figure 5(a) illustrates the situation of needing a ROM three-times wider than that available. We connect three ROMs side-by-side, sharing the same address and enable lines among them, and concatenating the data lines to form the desired word width.

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

Figure 5.5: Composing memories into larger ones.

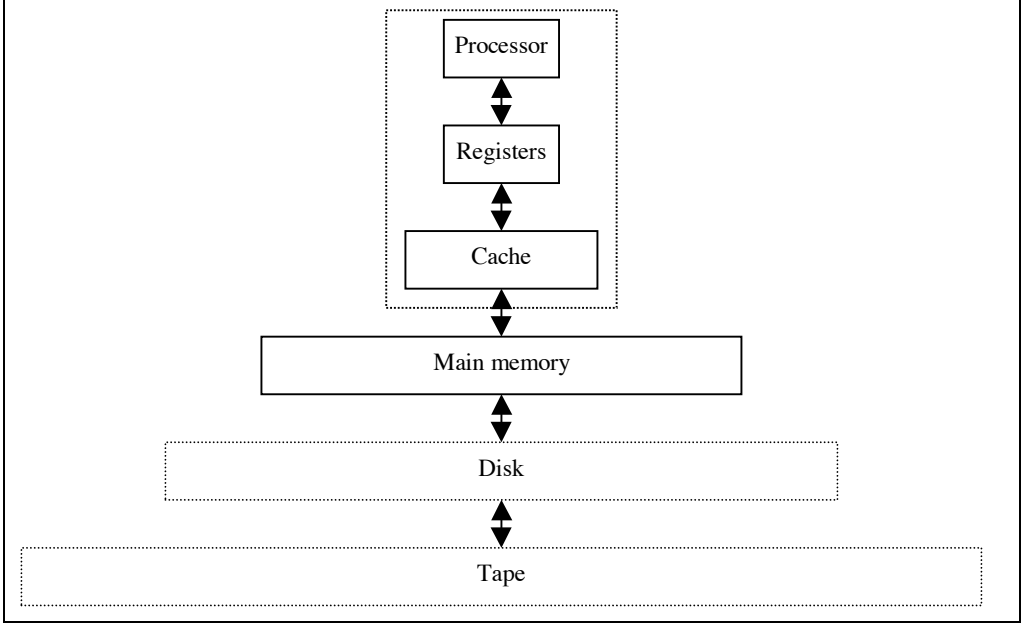


remaining address lines to offset into the selected ROM. Since only one ROM will ever be enabled at a time, the OR'ing of the data lines never actually involves more than one 1.

If we instead needed four times as many words, and hence two extra address lines, we would instead use four ROMs.A 2x4 decoder having the two high-order address lines as input would select which of the four ROMs to access.

Finally, suppose the available memories have a smaller word with as well as fewer words than necessary. We then combine the above two techniques, first creating the number of columns of memories necessary to achieve the needed word width, and then

Figure 5.6: An example memory hierarchy.



creating the number of rows of memories necessary, along with a decoder, to achieve the needed number of words. The approach is illustrated in Figure 5(c).

5.5 Memory hierarchy and cache

When we design a memory to store an embedded system's program and data, we often face the following dilemma: we want an inexpensive and fast memory, but inexpensive memories tend to be slow, whereas fast memories tend to be expensive. The solution to this dilemma is to create a memory hierarchy, as illustrated in Figure 5.6. We use an inexpensive but slow *main memory* to store all of the program and data. We use a small amount of fast but expensive *cache memory* to store copies of likely-accessed parts of main memory. Using cache is analogous to posting on a wall near a telephone a short list of important phone numbers rather than posting the entire phonebook.

Some systems include even larger and less expensive forms of memory, such as disk and tape, for some of their storage needs. However, we do not consider these further as they are not especially common in embedded systems. Also, although the figure shows only one cache, we can include any number of levels of cache, those closer to the processor being smaller and faster than those closer to main memory. A two-level cache scheme is common.

Cache is usually designed using static RAM rather than dynamic RAM, which is one reason that cache is more expensive but faster than main memory. Because cache usually appears on the same chip as a processor, where space is very limited, cache size is

typically only a fraction of the size main memory. Cache access time may be as low as just one clock cycle, whereas main memory access time is typically several cycles.

A cache operates as follows. When we want the processor to access (read or write) a main memory address, we first check for a copy of that location in cache. If the copy is in the cache, called a *cache hit*, then we can access it quickly. If the copy is not there, called a *cache miss*, then we must first read the address (and perhaps some of its neighbors) into the cache. This description of cache operation leads to several cache design choices: cache mapping, cache replacement policy, and cache write techniques. These design choices can have significant impact on system cost, performance, as well as power, and thus should be evaluated carefully for a given application.

5.5.1 Cache mapping techniques

Cache mapping is the method for assigning main memory addresses to the far fewer number of available cache addresses, and for determining whether a particular main memory address' contents are in the cache. Cache mapping can be accomplished using one of three basic techniques:

1. *Direct mapping*: In this technique, the main memory address is divided into two fields, the index and the tag. The index represents the cache address, and thus the number of index bits is determined by the cache size, i.e., index size = log_2(cache size). Note that many different main memory addresses will map to the same cache address. When we store a main memory address' content in the cache, we also store the tag. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and we then compare the tag there with the desired tag.
2. *Fully-associative mapping:* In this technique, each cache address contains not only a main memory address' content, but also the complete main memory address. To determine if a desired main memory address is in the cache, we simultaneously (associatively) compare all the addresses stored in the cache with the desired address.
3. *Set-associative mapping:* This technique is a compromise between direct and fully-associative mapping. As in direct-mapping, an index maps each main memory address to a cache address, but now each cache address contains the content and tags of two or more memory locations, called a set or a line. To determine if a desired main memory address is in the cache, we go to the cache address indicated by the index, and we then simultaneously (associatively) compare all the tags at that location (i.e., of that set) with the desired tag. A cache with a set of size N is called an N-way set-associative cache. 2-way, 4-way and 8-way set associative caches are common.

Direct-mapped caches are easy to implement, but may result in numerous misses if two or more words with the same index are accessed frequently, since each will bump the other out of the cache. Fully-associative caches on the other hand are fast but the comparison logic is expensive to implement. Set-associative caches can reduce misses

compared to direct-mapped caches, without requiring nearly as much comparison logic as fully-associative caches.

Caches are usually designed to treat collections of a small number of adjacent main-memory addresses as one indivisible *block*, typically consisting of about 8 addresses.

5.5.2 Cache replacement policy

The *cache-replacement policy* is the technique for choosing which cache block to replace when a fully-associative cache is full, or when a set-associative cache's line is full. Note that there is no choice in a direct-mapped cache; a main memory address always maps to the same cache address and thus replaces whatever block is already there. There are three common replacement policies. A *random* replacement policy chooses the block to replace randomly. While simple to implement, this policy does nothing to prevent replacing block that's likely to be used again soon. A *least-recently used (LRU)* replacement policy replaces the block that has not been accessed for the longest time, assuming that this means that it is least likely to be accessed in the near future. This policy provides for an excellent hit/miss ratio but requires expensive hardware to keep track of the times blocks are accessed. A *first-in-first-out (FIFO)* replacement policy uses a queue of size N, pushing each block address onto the queue when the address is accessed, and then choosing the block to replace by popping the queue.

5.5.3 Cache write techniques

When we write to a cache, we must at some point update the memory. Such update is only an issue for data cache, since instruction cache is read-only. There are two common update techniques, write-through and write-back.

In the *write-through* technique, whenever we write to the cache, we also write to main memory, requiring the processor to wait until the write to main memory completes. While easy to implement, this technique may result in several unnecessary writes to main memory. For example, suppose a program writes to a block in the cache, then reads it, and then writes it again, with the block staying in the cache during all three accesses. There would have been no need to update the main memory after the first write, since the second write overwrites this first write.

The *write-back* technique reduces the number of writes to main memory by writing a block to main memory only when the block is being replaced, and then only if the block was written to during its stay in the cache. This technique requires that we associate an extra bit, called a dirty bit, with each block. We set this bit whenever we write to the block in the cache, and we then check it when replacing the block to determine if we should copy the block to main memory.

5.6 Summary

Memories store data for use by processors. ROM typically is only read by an embedded system. It can be programmed during fabrication (mask-programmed) or by the user (programmable ROM, or PROM). PROM may be erasable using UV light (EPROM), or electronically-erasable (EEPROM). RAM, on the other hand, is memory that can be read or written by an embedded system. Static RAM uses a flip-flop to store

each bit, while dynamic RAM uses a transistor and capacitor, resulting in fewer transistors but the need to refresh the charge on the capacitor and slower performance. Psuedo-static RAM is a dynamic RAM with a built-in refresh controller. Non-volatile RAM keeps its data even after power is shut off. Designers must not only choose the appropriate type of memories for a given system, but must often compose smaller memories into larger ones. Using a memory hierarchy can improve system performance by keeping copies of frequently-accessed instructions/data in small, fast memories. Cache is a small and fast memory between a processor and main memory. Several cache design features greatly influence the speed and cost of cache, including mapping techniques, replacement policies, and write techniques.

5.7 References and further reading

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

5.8 Exercises

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

Chapter 6 *Interfacing*

6.1 Introduction

As stated in the previous chapter, we use *processors* to implement processing, *memories* to implement storage, and *buses* to implement communication. The earlier chapters described processors and memories. This chapter describes implementing communication with buses, i.e., interfacing.

Buses implement communication among processors or among processors and memories. Communication is the transfer of data among those components. For example, a general-purpose processor reading or writing a memory is a common form of communication. A general-purpose processor reading or writing a peripheral's register is another common form.

A bus consists of wires connecting two or more processors or memories. Figure 6.1(a) shows the wires of a simple bus connecting a processor with a memory. Note that each wire may be uni-directional, as are *rd/wr, enable*, and *addr*, or bi-directional, as is *data*. Also note that a set of wires with the same function is typically drawn as a thick line (or a line with a small angled line drawn through it). *addr* and *data* each represent a set of wires; the *addr* wires transmit an address, while the *data* wires transmit data. The bus connects to "pins" of a processor (or memory). A pin is the actual conducting device (i.e., metal) on the periphery of a processor through which a signal is input to or output from the processor. When a processor is packaged as its own IC, there are actual pins extending from the package, designed to be plugged into a socket on a printed-circuit board. Today, however, a processor commonly co-exists on a single IC with other processors and memories. Such a processor does not have any actual pins on its periphery, but rather "pads" of metal in the IC. In fact, even for a processor packaged in its own IC, alternative packaging-techniques may use something other than pins for connections, such as small metallic balls. For consistency, though, we shall use the term pin in this chapter regardless of the packaging situation.

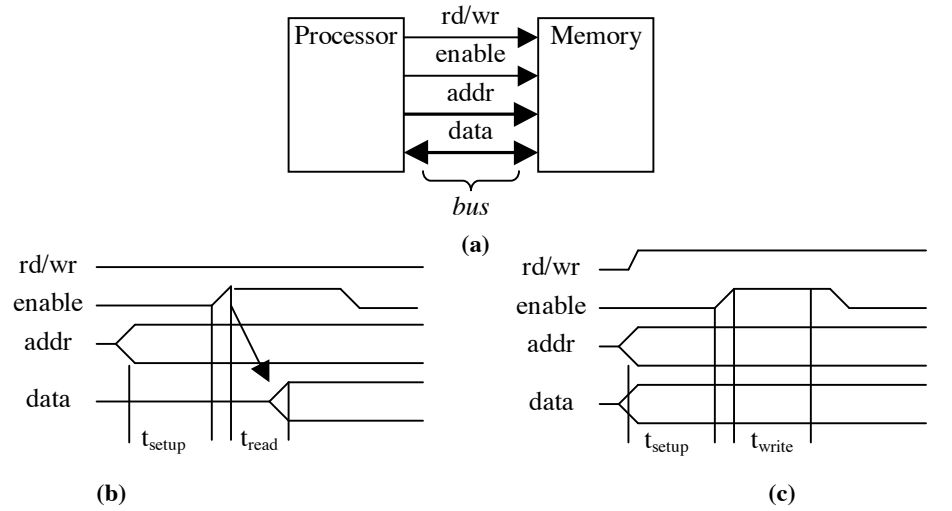
A bus must have an associated *protocol* describing the rules for transferring data over those wires. We deal primarily with low-level hardware protocols in this chapter, while higher-level protocols, like IP (Internet Protocol) can be built on top of these protocols, using a layered approach.

Interfacing with a general-purpose processor is extremely common. We describe three issues relating to such interfacing: addressing, interrupts, and direct memory access.

When multiple processors attempt to access a single bus or memory simultaneously, resource contention exists. This chapter therefore describes several schemes for arbitrating among the contending processors.

6.2 Timing diagrams

Figure 6.1: Timing diagrams: (a) bus structure, (b) read, (c) write.



The most common method for describing a hardware protocol is a timing diagram. Consider the example processor-memory bus of Figure 6.1(a). Figure 6.1(b) uses a timing diagram to describe the protocol for reading the memory over the bus. In the diagram, time proceeds to the right along the x-axis. The diagram shows that the processor must set the *rd/wr* line low for a read to occur. The diagram also shows, using two vertical lines, that the processor must place the address on *addr* for at least t_{setup} time before setting the *enable* line high. The diagram shows that the high *enable* line triggers the memory to put data on the *data* wires after a time t_{read}. Note that a timing diagram represents control lines, like *rd/wr* and *enable*, as either being high or low, while it represents data lines, like *addr* and *data*, as being either invalid (a single horizontal line) or valid (two horizontal lines); the value of data lines is not normally relevant when describing a protocol.

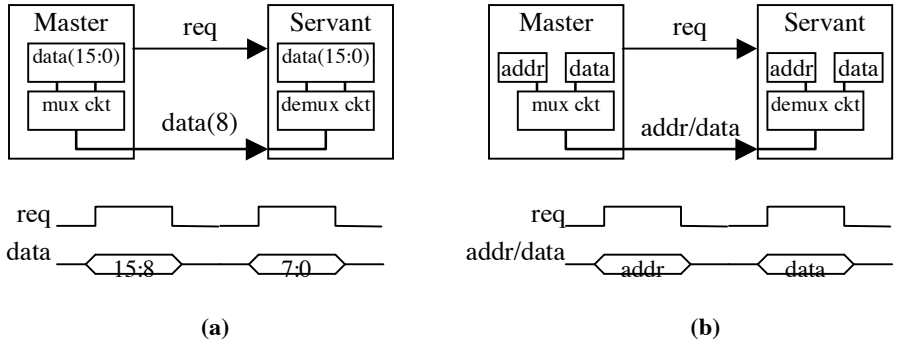
In the above protocol, the control line *enable* is active high, meaning that a 1 on the enable line triggers the data transfer. In many protocols, control lines are instead active low, meaning that a 0 on the line triggers the transfer. Such a control line is typically written with a bar above it, a single quote after it (e.g., *enable'*), or an underscore l after it (e.g., *enable_l*). To be general, we will use the term "assert" to mean setting a control line to its active value (i.e., to 1 for an active high line, to 0 for an active low line), and the term "deassert" to mean setting the control line to its inactive value.

6.3 Hardware protocol basics

6.3.1 Concepts

The protocol described above was a simple one. Hardware protocols can be much more complex. However, we can understand them better by defining some basic protocol

Figure 6.2: Time-multiplexed data transfer: (a) data serializing, (b) address/data muxing.



concepts. These concepts include: actors, data direction, addresses, time-multiplexing, and control methods.

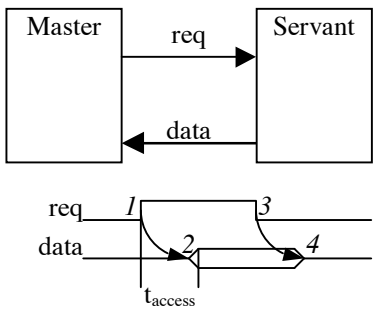
An *actor* is a processor or memory involved in the data transfer. A protocol typically involves two actors: a master and a servant. A *master* initiates the data transfer. A *servant* (usually called a slave) responds to the initiation request. In the example of Figure 6.1, the processor is the master and the memory is the servant, i.e., the memory cannot initiate a data transfer. The servant could also be another processor. Masters are usually general-purpose processors, while servants are usually peripherals and memories.

Data direction denotes the direction that the transferred data moves between the actors. We indicate this direction by denoting each actor as either receiving or sending data. Note that actor types are independent of the direction of the data transfer. In particular, a master may either be the receiver of data, as in Figure 6.1(b), or the sender of data, as shown in Figure 6.1(c).

Addresses represent a special type of data used to indicate where regular data should go to or come from. A protocol often includes both an address and regular data, as did the memory access protocol in Figure 6.1, where the address specified where the data should be read from or written to in the memory. An address is also necessary when a general-purpose processor communicates with multiple peripherals over a single bus; the address not only specifies a particular peripheral, but also may specify a particular register within that peripheral.

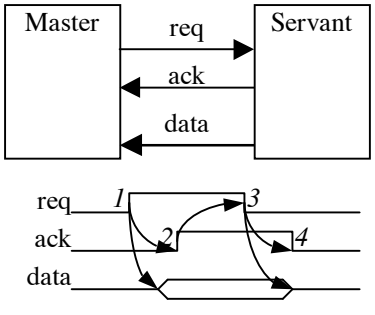
Another protocol concept is *time multiplexing*. To multiplex means to share a single set of wires for multiple pieces of data. In time multiplexing, the multiple pieces of data are sent one at a time over the shared wires. For example, Figure 6.2(a) shows a master sending 16 bits of data over an 8-bit bus using a strobe protocol and time-multiplexed data. The master first sends the high-order byte, then the low-order byte. The servant must receive the bytes and then demultiplex the data. This serializing of data can be done to any extent, even down to a 1-bit bus, in order to reduce the number of wires. As another example, Figure 6.2(b) shows a master sending both an address and data to a servant (probably a memory). In this case, rather than using separate sets of lines for

Figure 6.3: Two protocol control methods: (a) strobe, (b) handshake.



1. Master asserts *req* to receive data
2. Servant puts data on bus **within time t_{access}**
3. Master receives data and deasserts *req*
4. Servant ready for next request

(a)



1. Master asserts *req* to receive data
2. Servant puts data on bus **and asserts *ack***
3. Master receives data and deasserts *req*
4. Servant ready for next request

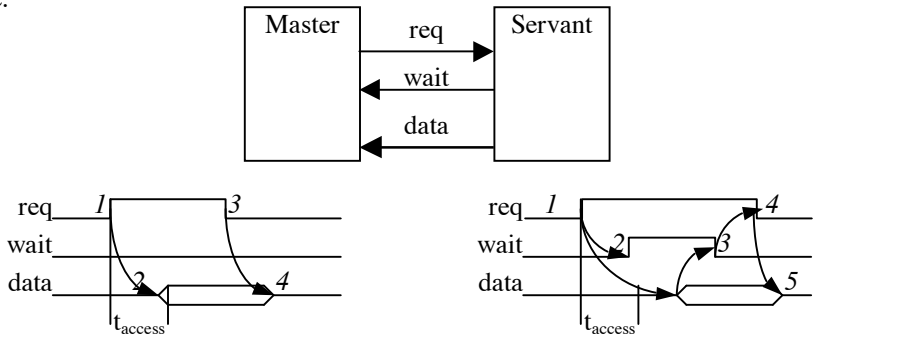
(b)

address and data, as was done in Figure 6.1, we can time multiplex the address and data over a shared set of lines *addr/data*.

Control methods are schemes for initiating and ending the transfer. Two of the most common methods are strobe and handshake. In a *strobe* protocol, the master uses one control line, often called the *request* line, to initiate the data transfer, and the transfer is considered to be complete after some fixed time interval after the initiation. For example, Figure 6.3(a) shows a strobe protocol with a master wanting to receive data from a servant. The master first asserts the request line to initiate a transfer. The servant then has time t_{access} ,to put the data on the data bus. After this time, the master reads the data bus, believing the data to be valid. The master than deasserts the request line, so that the servant can stop putting the data on the data bus, and both actors are then ready for the next transfer. An analogy is a demanding boss who tells an employee "I want that report (the data) on my desk (the data bus) in one hour (t_{access})," and merely expects the report to be on the desk in one hour.

The second common control method is a *handshake* protocol, in which the master uses a request line to initiate the transfer, and the servant uses an *acknowledge* line to inform the master when the data is ready. For example, Figure 6.3(b) shows a handshake protocol with a receiving master. The master first asserts the request line to initiate the transfer. The servant takes however much time is necessary to put the data on the data bus, and then asserts the acknowledge line to inform the master that the data is valid. The master reads the data bus and then deasserts the request line so that the servant can stop putting data on the data bus. The servant deasserts the acknowledge line, and both actors are then ready for the next transfer. In our boss-employee analogy, a handshake protocol corresponds to a more tolerant boss who tells an employee "I want that report on my desk soon; let me know when it's ready." A handshake protocol can adjust to a servant (or

Figure 6.4: A strobe/handshake compromise: (a) fast-response case, (b) slow-response case.



1. Master asserts req to receive data
2. Servant puts data on bus **within time t_{access}**
(wait line is unused)
3. Master receives data and deasserts request
4. Servant ready for next request

(a)

1. Master asserts req to receive data
2. Servant can't put data within **t_{access}, *asserts wait***
3. Servant finally puts data and ***deasserts wait***
4. Master receives data and deasserts request
5. Servant ready for next request

(b)

servants) with varying response times, unlike a strobe protocol. However, when response time is known, a handshake protocol may be slower than a strobe protocol, since it requires the master to detect the acknowledgement before getting the data, possibly requiring an extra clock cycle if the master is synchronizing the bus control signals. A handshake also requires an extra line for acknowledge.

To achieve both the speed of a strobe protocol and the varying response time tolerance of a handshake protocol, a compromise protocol is often used, as illustrated in Figure 6.4. In the case, when the servant can put the data on the bus within time t_{access}, the protocol is identical to a strobe protocol, as shown in Figure 6.4(a). However, if the servant cannot put the data on the bus in time, it instead tells the master to wait longer, by asserting a line we've labeled *wait*. When the servant has finally put the data on the bus, it deasserts the wait line, thus informing the master that the data is ready. The master receives the data and deasserts the request line. Thus, the handshake only occurs if it is necessary. In our boss-employee analogy, the boss tells the employee "I want that report on my desk in an hour; if you can't finish by then, let me know that and then let me know when it's ready."

Example: A simple bus protocol

```
A protocol for a simple bus (ISA) will be described.
```

6.4 Interfacing with a general-purpose processor

Perhaps the most common communication situation in embedded systems is the input and output (I/O) of data to and from a general-purpose processor, as it

communicates with its peripherals and memories. I/O is relative to the processor: input means data comes into the processor, while output means data goes out of the processor. We will describe three processor I/O issues: addressing, interrupts, and direct memory access. We'll use the term microprocessor in this section to refer to a general-purpose processor.

6.4.1 I/O addressing

A microprocessor may have tens or hundreds of pins, many of which are control pins, such as a pin for clock input and another input pin for resetting the microprocessor. Many of the other pins are used to communicate data to and from the microprocessor, which we call processor I/O. There are two common methods for using pins to support I/O: ports, and system buses.

A *port* is a set of pins that can be read and written just like any register in the microprocessor; in fact, the port is usually connected to a dedicated register. For example, consider an 8-bit port named P0. A C-language programmer may write to P0 using an instruction like: P0 = 255, which would set all 8 pins to 1's. In this case, the C compiler manual would have defined P0 as a special variable that would automatically be mapped to the register P0 during compilation. Conversely, the programmer might read the value of a port P1 being written by some other device, by saying something like a=P1. In some microprocessors, each bit of a port can be configured as input or output by writing to a configuration register for the port. For example, P0 might have an associated configuration register called CP0. To set the high-order four bits to input and the low-order four bits to output, we might say: CP0 = 15. This writes 00001111 to the CP0 register, where a 0 means input and a 1 means output. Ports are often bit-addressable, meaning that a programmer can read or write specific bits of the port. For example, one might say: x = P0.2, giving x the value of the number 2 connection of port P0. Port-based I/O is also called *parallel I/O*.

In contrast to a port, a *system bus* is a set of pins consisting of address pins, data pins, and control pins (for strobing or handshaking). The microprocessor uses the bus to access memory as well as peripherals. We normally consider the access to the peripherals as I/O, but don't normally consider the access to memory as I/O, since the memory is considered more as a part of the microprocessor. A microprocessor may use one of two methods for communication over a system bus: standard I/O or memory-mapped I/O.

In *memory-mapped I/O*, peripherals occupy specific addresses in the existing address space. For example, consider a bus with a 16-bit address. The lower 32K addresses may correspond to memory addresses, while the upper 32K may correspond to I/O addresses.

In *standard I/O* (also known as I/O-mapped I/O), the bus includes an additional pin, which we label M/IO, to indicate whether the access is to memory or to a peripheral (i.e., an I/O device). For example, when M/IO is 0, the address on the address bus corresponds to a memory address. When M/IO is 1, the address corresponds to a peripheral.

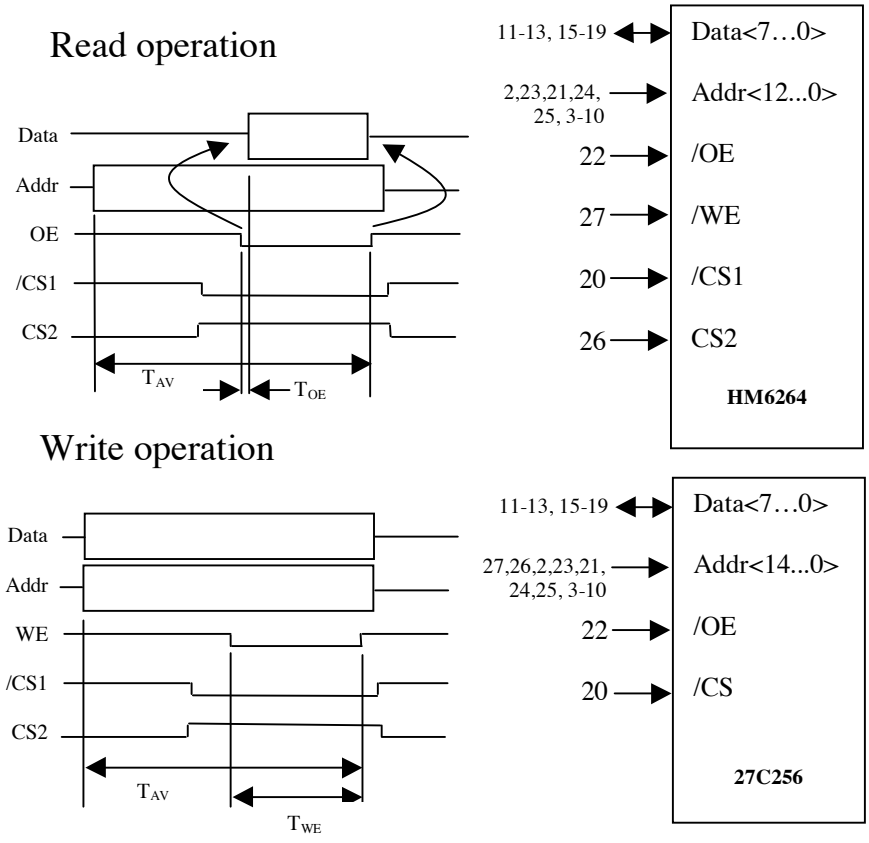
Example: HM6264 and 27C256 RAM/ROM memory devices

In this example, we introduce a pair of low-cost low-capacity memory devices commonly used in 8-bit micro-controller based embedded systems. The first two numeric digits in these devices indicate weather the device is *random-access memory* (RAM), 62, or *read-only memory* (ROM), 27. Subsequent digits give the memory capacity in K bits. Both these devices are available in 4, 8, 16, 32, and 64K bytes, i.e., part numbers 62/27 followed by 32, 64, 128, 256, or 512. The following table summarizes some of the characteristics of these devices.

Device	Access Time (ns)	Standby Pwr. (mw)	Active Pwr. (mw)	Vcc Voltage (V)
HM6264	85-100	.01	15	5
27C256	90	.5	100	5

Source Hitachi HM6264B and Microchip 27C256 datasheets.

Memory access to and from these devices is performed through an 8-bit parallel protocol. Placing a memory address on the address-bus and asserting the read signal *output-enable* (OE) performs a read operation. Placing some data and a memory address on the data/address busses and asserting the write signal *write-enable* (WE) performs a write operation. Read and write timing diagrams are presented here.



In the next example, we demonstrate interfacing of these devices with an Intel 8051 micro-controller.

*T_{WE} =40ns, T_{OE} =75, T_{AV} =40-148

An advantage of memory-mapped I/O is that the microprocessor need not include special instructions for communicating with peripherals. The microprocessor's assembly instructions involving memory, such as MOV or ADD, will also work for peripherals. For example, a microprocessor may have an *ADD A, B* instruction that adds the data at address *B* to the data at address *A* and stores the result in *A*. *A* and *B* may correspond to memory locations, or registers in peripherals. In contrast, if the microprocessor uses standard I/O, the microprocessor requires special instructions for reading and writing peripherals. These instructions are often called *IN* and *OUT*. Thus, to perform the same addition of locations *A* and *B* corresponding to peripherals, the following instructions would be necessary:

```
IN R0, A
IN R1, B
ADD R0, R1
OUT A, R0
```

Advantages of standard I/O include no loss of memory addresses to use as I/O addresses, and potentially simpler address decoding logic in peripherals. Address decoding logic can be simplified with standard I/O if we know that there will only be a small number of peripherals, because the peripherals can then ignore high-order address bits. For example, a bus may have a 16-bit address, but we may know there will never be more than 256 I/O addresses required. The peripherals can thus safely ignore the high-order 8 address bits, resulting in smaller and/or faster address comparators in each peripheral.

Situations often arise in which an embedded system requires more ports than available on a particular microprocessor. For example, one may desire 10 ports, while the microprocessor only has 4 ports. An *extended parallel I/O* peripheral can be used to achieve this goal.

Similarly, a system may require parallel I/O but the microprocessor may only have a system bus. In this case, a *parallel I/O* peripheral may be used. The peripheral is connected to the system bus on one side, and has several ports on the other side. The ports are connected to registers inside the peripheral, and the microprocessor can read and write those registers in order to read and write the ports.

Example: memory mapped and standard I/O

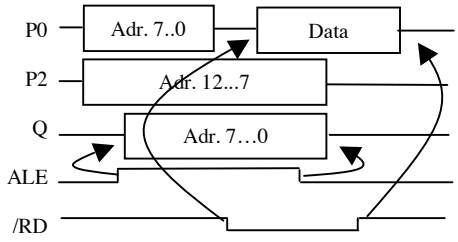
`Two examples of real processor I/O to be added.`

6.4.2 Interrupts

Another microprocessor I/O issue is that of interrupt-driven I/O. In particular, suppose the program running on a microprocessor must, among other tasks, read and process data from a peripheral whenever that peripheral has new data; such processing is called *servicing*. If the peripheral gets new data at unpredictable intervals, then how can the program determine when the peripheral has new data? The most straightforward approach is to interleave the microprocessor's other tasks with a routine that checks for new data in the peripheral, perhaps by checking for a 1 in a particular bit in a register of

Example: A basic memory protocol

In this example, we illustrate how to interface 8K of data and 32 K of program code memory to a micro-controller, specifically the *Intel 8051*. The Intel 8051 uses separate memory address spaces for data and program code. Data or code address space is limited to 64K, hence, addressable with 16 bits through ports P0 (LSBs) and P2 (MSBs). A separate signal, called PSEN (program strobe enable), is used to distinguish between data/code. For the most part, the I8051 generates all of the necessary signals to perform memory I/O, however, since port P0 is used for both LSB address bits and data flow into and out of the RAM an 8-bit latch is required to perform the necessary multiplexing. The following timing diagram illustrates a memory read operation. Memory write operation is performed in a similar fashion with data flow reversed and RD (read) replaced with WR (write).



Memory read operation proceeds as follows. The micro-controller places the source address, i.e., the memory location to be read, on ports P2 and P0. P2, holding the 8-MSB bits of the address, retains its value throughout the read operation. P1, holding the 8-LSB bits of the address is stored inside an 8-bit latch. The ALE signal (address latch enable), is used to trigger the latching of port P0. Now, the micro-controller asserts high impedance on P0 to allow the memory device to drive it with the requested data. The memory device outputs valid data as long as the RD signal is asserted. Meanwhile, the micro-controller reads the data and de-asserts its control and port signals. The following figure gives the interface schematic.

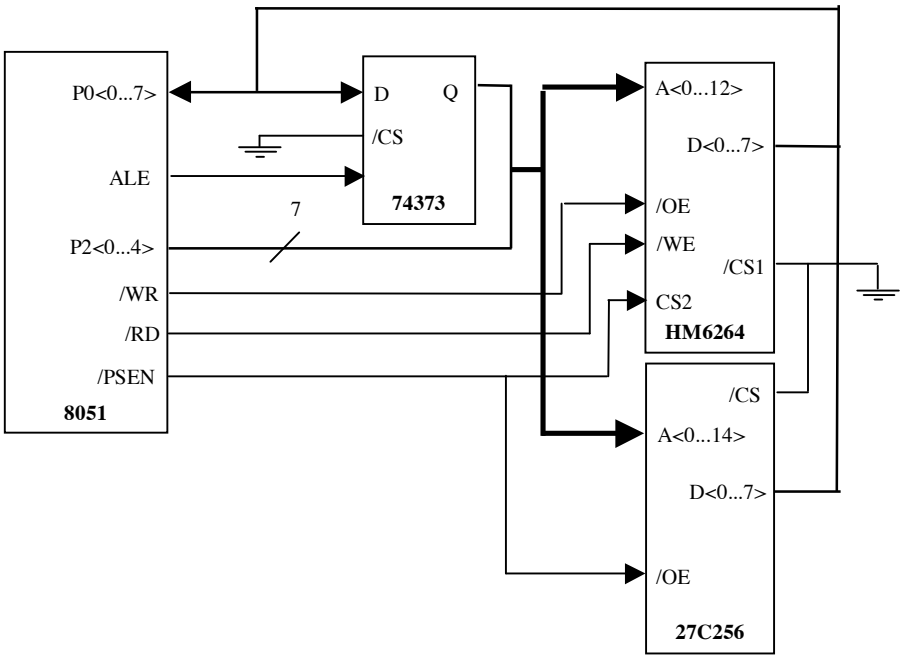
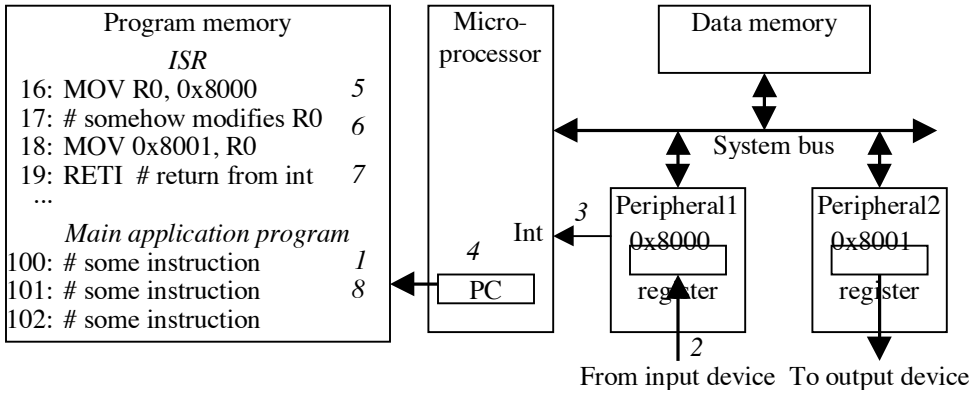


Figure 6.5: Interrupt-driven I/O using fixed ISR location.



1. *Microprocessor is executing its program, say current program counter (PC)=100.*
2. *Peripheral1's input device stores new data in register at 0x8000.*
3. *Peripheral1 asserts Int to request servicing by microprocessor of the new data.*
4. *Microprocessor stops executing its program and stores state, including PC.*
5. *Microprocessor jumps to fixed location of ISR at program memory location 16 (sets PC=16)*
6. *ISR reads data from 0x8000, modifies the data, and writes to 0x8001.*
7. *ISR returns, causing restoration of the microprocessor state, thus setting PC= 100+1 = 101*
8. *Microprocessor resumes executing its program.*

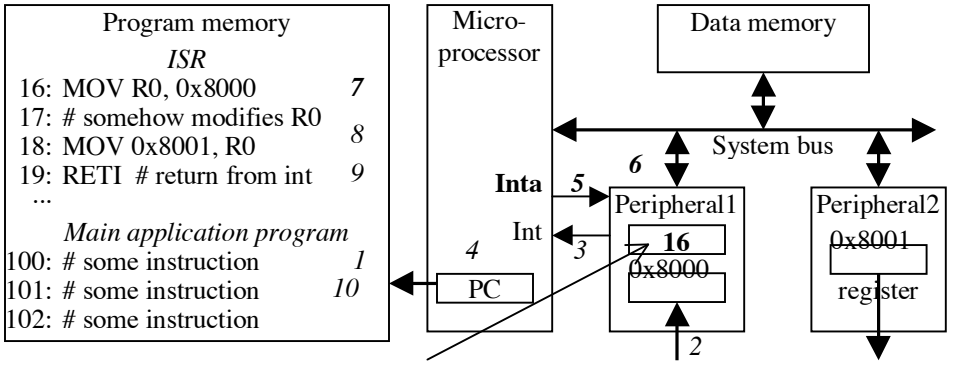
the peripheral. This repeated checking by the microprocessor for data is called *polling*. Polling is simple to implement, but this repeated checking wastes many clock cycles, so may not be acceptable in many cases, especially when there are numerous peripherals to be checked. We could check at less-frequent intervals, but then we may not process the data quickly enough.

To overcome the limitations of polling, most microprocessors come with a feature called *external interrupt*. A microprocessor with this feature has a pin, say *Int*. At the end of executing each machine instruction, the processor's controller checks *Int*. If *Int* is asserted, the microprocessor jumps to a particular address at which a subroutine exists that services the interrupt. This subroutine is called an *Interrupt Service Routine*, or *ISR*. Such I/O is called *interrupt-driven I/O*.

One might wonder if interrupts have really solved the problem with polling, namely of wasting time performing excessive checking, since the interrupt pin is "polled" at the end of every microprocessor instruction. However, in this case, the polling of the pin is built right into the microprocessor's controller hardware, and therefore can be done simultaneously with the execution of an instruction, resulting in no extra clock cycles.

There are two methods by which a microprocessor using interrupts determines the address, known as the *interrupt address vector*, at which the ISR resides. In some processors, the address to which the microprocessor jumps on an interrupt is *fixed*. The assembly programmer either puts the ISR there, or if not enough bytes are available in

Figure 6.6: Interrupt-driven I/O using vectored interrupt.



1. *Microprocessor is executing its program, say current program counter (PC)=100.*
2. *Peripheral1's input device stores new data in register at 0x8000.*
3. *Peripheral1 asserts Int to request servicing by microprocessor of the new data.*
4. *Microprocessor stops executing its program and stores state, including PC.*
5. ***Microprocessor asserts Inta***
6. ***Peripheral1 puts its interrupt address vector 16 on the system bus***
7. ***Microprocessor jumps to the address of ISR read from the data bus, in this case 16.***
8. *ISR reads data from 0x8000, modifies the data, and writes to 0x8001.*
9. *ISR returns, causing restoration of the microprocessor state, thus setting PC= 100+1 = 101*
10. *Microprocessor resumes executing its program.*

that region of memory, merely puts a jump to the real ISR there. For C programmers, the compiler typically reserves a special name for the ISR and then compiles a subroutine having that name into the ISR location. In microprocessors with fixed ISR addresses, there may be several interrupt pins to support interrupts from multiple peripherals. For example, Figure 6.5 provides an example of interrupt-driven I/O using a fixed ISR address. In this example, data received by Peripheral1 must be read, transformed, and then written to Peripheral2. Peripheral1 might represent a sensor and Peripheral2 a display. Meanwhile, the microprocessor is running its main program, located in program memory starting at address 100. When Peripheral1 receives data, it asserts *Int* to request that the microprocessor service the data. After the microprocessor completes execution of its current instruction, it stores its state and jumps to the ISR located at the fixed program memory location of 16. The ISR reads the data from Peripheral1, transforms it, and writes the result to Peripheral2. The last ISR instruction is a return from interrupt, causing the microprocessor to restore its state and resume execution of its main program, in this case executing instruction 101.

Other microprocessors use *vectored interrupt* to determine the address at which the ISR resides. This approach is especially common in systems with a system bus, since there may be numerous peripherals that can request service. In this method, the

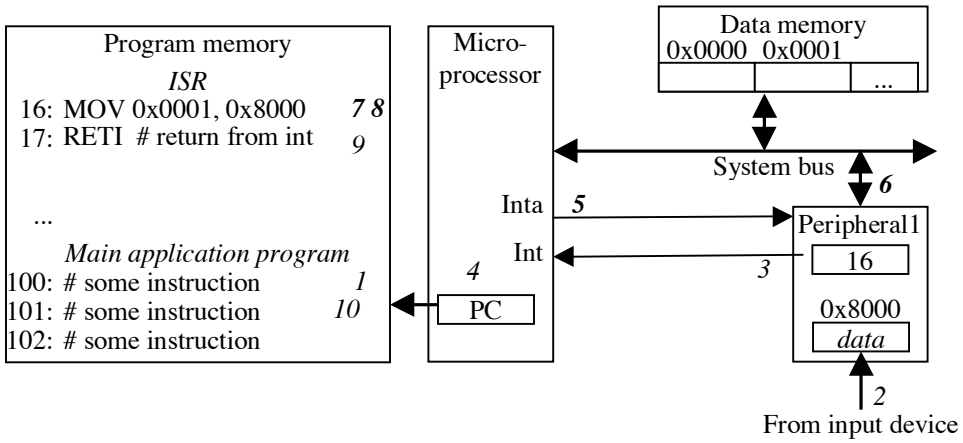
microprocessor has one interrupt pin, say *IntReq*, which any peripheral can assert. After detecting the interrupt, the microprocessor asserts another pin, say *IntAck*, to acknowledge that it has detected the interrupt and to request that the interrupting peripheral provide the address where the relevant ISR resides. The peripheral provides this address on the data bus, and the microprocessor reads the address and jumps to the ISR. We discuss the situation where multiple peripherals simultaneously request servicing in a later section on arbitration. For now, consider an example of one peripheral using vectored interrupt, shown in Figure 6.6, which represents an example very similar to the previous one. In this case, however, the ISR location is not fixed at 16. Thus, Peripheral1 contains an extra register holding the ISR location. After detecting the interrupt and saving its state, the microprocessor asserts *Inta* in order to get Peripheral1 to place 16 on the data bus. The microprocessor reads this 16 into the PC, and then jumps to the ISR, which executes and completes in the same manner as the earlier example.

As a compromise between the fixed and vectored interrupt methods, we can use an *interrupt address table*. In this method, we still have only one interrupt pin on the processor, but we also create in the processor's memory a table that holds ISR addresses. A typical table might have 256 entries. A peripheral, rather than providing the ISR address, instead provides a number corresponding to an entry in the table. The processor reads this entry number from the bus, and then reads the corresponding table entry to obtain the ISR address. Compared to the entire memory, the table is typically very small, so an entry number's bit encoding is small. This small bit encoding is especially important when the data bus is not wide enough to hold a complete ISR address. Furthermore, this approach allows us to assign each peripheral a unique number independent of ISR locations, meaning that we could move the ISR location without having to change anything in the peripheral.

External interrupts may be *maskable* or *non-maskable*. In maskable interrupt, the programmer may force the microprocessor to ignore the interrupt pin, either by executing a specific instruction to disable the interrupt or by setting bits in an interrupt configuration register. A situation where a programmer might want to mask interrupts is when there exists time-critical regions of code, such as a routine that generates a pulse of a certain duration. The programmer may include an instruction that disables interrupts at the beginning of the routine, and another instruction re-enabling interrupts at the end of the routine. Non-maskable interrupt cannot be masked by the programmer. It requires a pin distinct from maskable interrupts. It is typically used for very drastic situations, such as power failure. In this case, if power is failing, a non-maskable interrupt can cause a jump to a subroutine that stores critical data in non-volatile memory, before power is completely gone.

In some microprocessors, the jump to an ISR is handled just like the jump to any other subroutine, meaning that the state of the microprocessor is stored on a stack, including contents of the program counter, datapath status register, and all other registers, and then restored upon completion of the ISR. In other microprocessors, only a few registers are stored, like just the program counter and status registers. The assembly programmer must be aware of what registers have been stored, so as not to overwrite non-stored register data with the ISR. These microprocessors need two types of assembly instructions for subroutine return. A regular return instruction returns from a regular

Figure 6.7: Peripheral to memory transfer *without* DMA.



1. Microprocessor is executing its program, say current program counter (PC)=100.
2. Peripheral1's input device stores new data in register at 0x8000.
3. Peripheral1 asserts *Int* to request servicing by microprocessor of the new data.
4. Microprocessor stops executing its program and stores its state, including PC.
5. **Microprocessor asserts *Inta***
6. **Peripheral1 puts its interrupt address vector 16 on the system bus.**
7. **Microprocessor jumps to the address of ISR read from the data bus, in this case 16.**
8. ISR reads data from 0x8000 and writes that data to 0x0001 in data memory.
9. ISR returns, causing restoration of the microprocessor state, thus setting PC= 100+1 = 101, and terminating handshake with peripheral.
10. Microprocessor resumes executing its program.

subroutine, which was called using a subroutine call instruction. A return from interrupt instruction returns from an ISR, which was jumped to not by a call instruction but by the hardware itself, and which restores only those registers that were stored at the beginning of the interrupt. The C programmer is freed from having to worry about such considerations, as the C compiler handles them.

The reason we used the term "external interrupt" is to distinguish this type of interrupt from internal interrupts, also called traps. An internal interrupt results from an exceptional condition, such as divide-by-0, or execution of an invalid opcode. Internal interrupts, like external ones, result in a jump to an ISR. A third type of interrupt, called software interrupts, can be initiated by executing a special assembly instruction.

Example: Interrupts

`Two examples of real processor interrupt handling to be added, one using fixed interrupt, the other vectored.`

6.4.3 Direct memory access

Commonly, the data being accumulated in a peripheral should be first stored in memory before being processed by a program running on the microprocessor. Such temporary storage to await processing is called buffering. For example, packet-data from an Ethernet card is stored in main memory and is later processed by the different software layers (such as IP stacks). We could write a simple interrupt service routine on the microprocessor, such that the peripheral device would interrupt the microprocessor whenever it had data to be stored in memory. The ISR would simply transfer data from the peripheral to the memory, and then resume running its application. For example, Figure 6.7 shows an example in which Peripheral1 interrupts the microprocessor when receiving new data. The microprocessor jumps to ISR location 16, which moves the data from 0x8000 in the peripheral to 0x0001 in memory. Then the ISR returns. However, recall that jumping to an ISR requires the microprocessor to store its state (i.e., register contents), and then to restore its state when returning from the ISR. This storing and restoring of the state may consume many clock cycles, and is thus somewhat inefficient. Furthermore, the microprocessor cannot execute its regular program while moving the data, resulting in further inefficiency.

The I/O method of direct memory access (DMA) eliminates these inefficiencies. In DMA, we use a separate single-purpose processor, called a DMA controller, whose sole purpose is to transfer data between memories and peripherals. Briefly, the peripheral requests servicing from the DMA controller, which then requests control of the system bus from the microprocessor. The microprocessor merely needs to relinquish control of the bus to the DMA controller. The microprocessor does not need to jump to an ISR, and thus the overhead of storing and restoring the microprocessor state is eliminated. Furthermore, the microprocessor can execute its regular program while the DMA controller has bus control, as long as that regular program doesn't require use of the bus (at which point the microprocessor would then have to wait for the DMA to complete). A system with a separate bus between the microprocessor and cache may be able to execute for some time from the cache while the DMA takes place.

We set up a system for DMA as follows. As shown in Figure 6.8, we connect the peripheral to the DMA controller rather than the microprocessor. Note that the peripheral does not recognize any difference between being connected to a DMA controller device or a microprocessor device; all it knows is that it asserts a request signal on the device, and then that device services the peripheral's request. We connect the DMA controller to two special pins of the microprocessor. One pin, which we'll call Hreq (bus Hold REQuest), is used by the DMA controller to request control of the bus. The other pin, which we'll call Hlda (HoLD Acknowledge), is used by the microprocessor to acknowledge to the DMA controller that bus control has been granted. Thus, unlike the peripheral, the microprocessor must be specially designed with these two pins in order to support DMA. The DMA controller also connects to all the system bus signals, including address, data, and control lines.

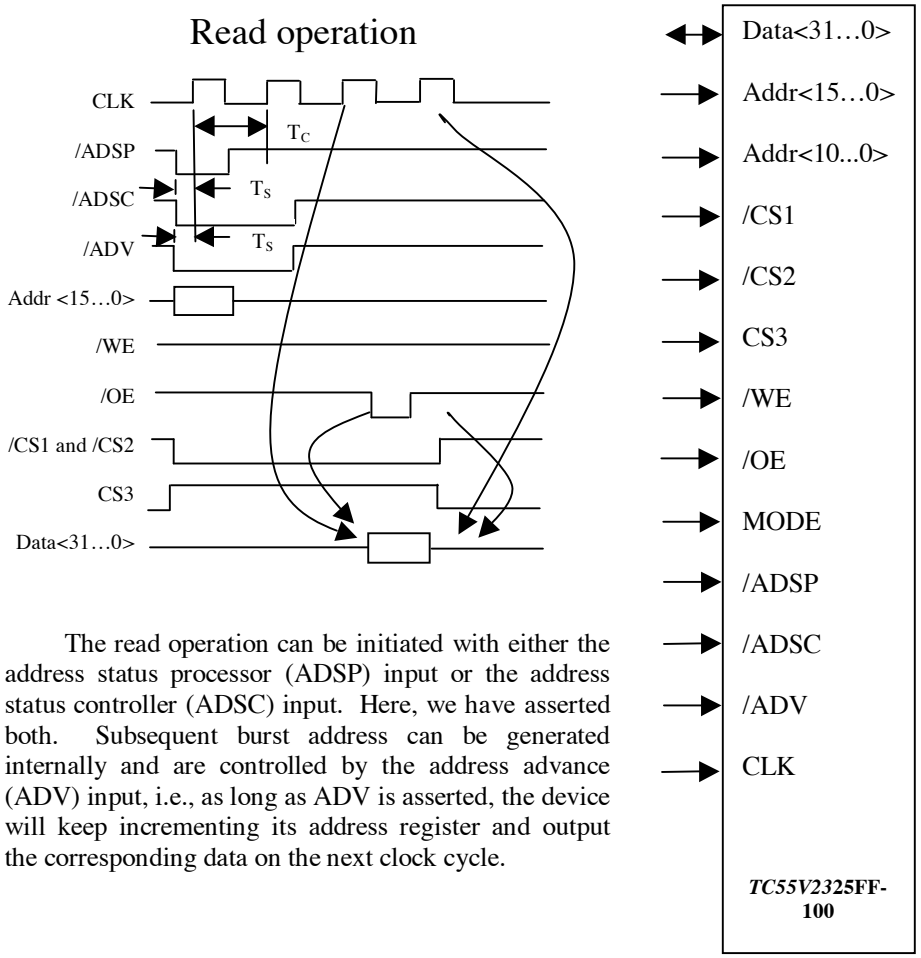
Example: TC55V2325FF-100 memory device

In this example, we introduce a 2M bit synchronous pipelined burst SRAM memory device designed to be interfaced with 32 bit processors. This device, made by Toshiba Inc., is organized as 64K × 32 bits. The following table summarizes some of its characteristics.

Device	Access Time (ns)	Standby Pwr. (mw)	Active Pwr. (mw)	Vcc Voltage (V)
TC55V2325FF-100	10	na	1200	3.3

Source Toshiba TC59S6432CFT datasheets.

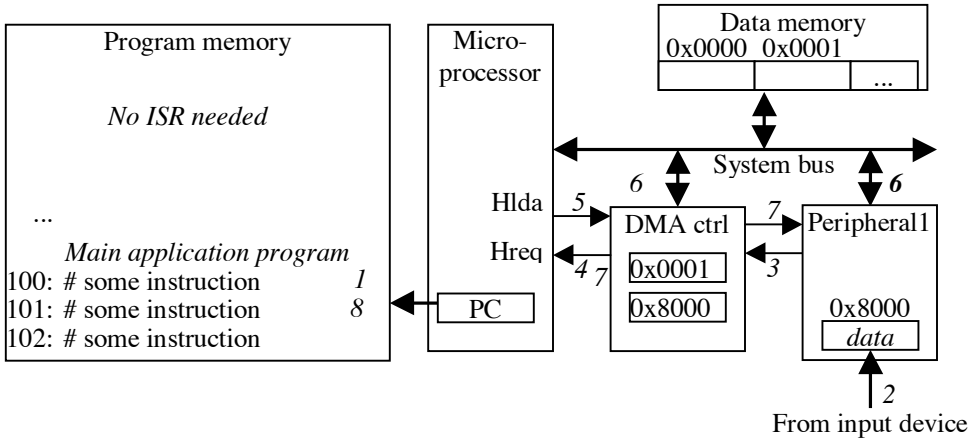
Here, we present the block and timing diagram for a single read operations. Write operation is similar. This device is capable of sequential, fast, reads and writes as well as singles byte I/O. Interested reader should refer to the manufacturer's datasheets for complete pinout and complete timing diagrams.



The read operation can be initiated with either the address status processor (ADSP) input or the address status controller (ADSC) input. Here, we have asserted both. Subsequent burst address can be generated internally and are controlled by the address advance (ADV) input, i.e., as long as ADV is asserted, the device will keep incrementing its address register and output the corresponding data on the next clock cycle.

*T_C =10ns, T_S =3ns

Figure 6.8: Peripheral to memory transfer *with* DMA.

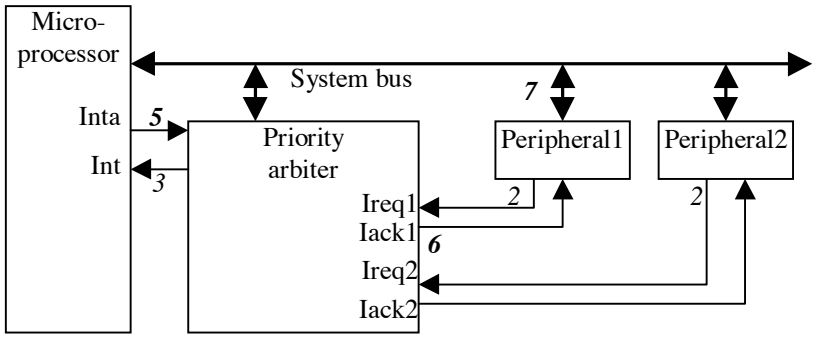


1. Microprocessor is executing its program, say current program counter (PC)=100.
2. Peripheral1's input device stores new data in register at 0x8000.
3. Peripheral1 asserts *Dreq* to request servicing of the new data.
4. DMA controller asserts *Hreq* to request control of the system bus.
5. Microprocessor relinquishes system bus, perhaps stopping after executing statement 100.
6. **DMA controller reads data from 0x8000 and writes that data to 0x0001 in data memory.**
7. DMA controller deasserts Hreq and completes handshake with peripheral
8. Microprocessor resumes executing its program, perhaps starting with statement 101.

To achieve the above, we must have configured the DMA controller to know what addresses to access in the peripheral and the memory. Such setting of addresses may be done by a routine running on the microprocessor during system initialization. In particular, during initialization, the microprocessor writes to configuration registers in the DMA controller just as it would write to any other peripheral's registers. Alternatively, in an embedded system that is guaranteed not to change, we can hardcode the addresses directly into the DMA controller. In the example of Figure 6.8, we see two registers in the DMA controller holding the peripheral register address and the memory address.

During its control of the system bus, the DMA controller might transfer just one piece of data, but more commonly will transfer numerous pieces of data (called a block), one right after other, before relinquishing the bus. This is because many peripherals, such as any peripheral that deals with storage devices (like CD-ROM players or disk controllers) or that deals with network communication, send and receive data in large blocks. For example, a particular disk controller peripheral might read data in blocks of 128 words and store this data in 128 internal registers, after which the peripheral requests servicing, i.e., requests that this data be buffered in memory. The DMA controller gains

Figure 6.9: Arbitration using a priority arbiter.



1. Microprocessor is executing its program.
2. Peripheral1 needs servicing so asserts *Ireq1*. Peripheral2 also needs servicing so asserts *Ireq2*.
3. Priority arbiter sees at least one *Ireq* input asserted, so asserts *Int*.
4. Microprocessor stops executing its program and stores its state.
5. **Microprocessor asserts *Inta*.**
6. **Priority arbiter asserts *Iack1* to acknowledge Peripheral1.**
7. **Peripheral1 puts its interrupt address vector on the system bus**
8. **Microprocessor jumps to the address of ISR read from data bus, ISR executes and returns (and completes handshake with arbiter).**
9. Microprocessor resumes executing its program.

control of the bus, makes 128 peripheral reads and memory writes, and only then relinquishes the bus. We must therefore configure the DMA controller to operate in either single transfer mode or block transfer mode. For block transfer mode, we must configure a base address as well as the number of words in a block.

DMA controllers typically come with numerous channels. Each channel supports one peripheral. Each channel has its own set of configuration registers.

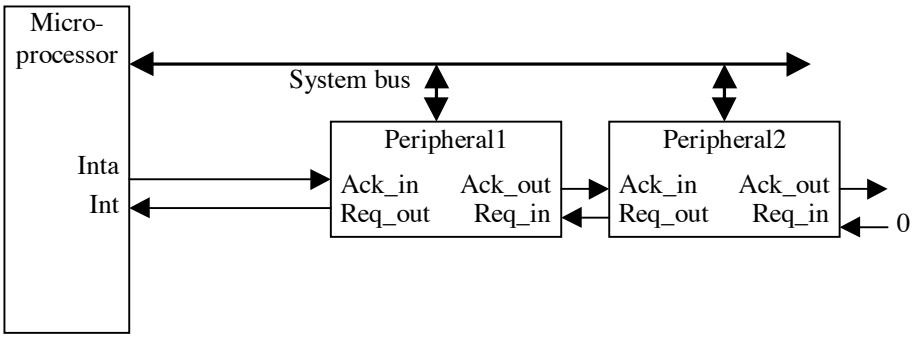
Example: Intel 8237

`Description of the Intel 8237 DMA controller and an example of its use.`

6.5 Arbitration

In our discussions above, several situations existed in which multiple peripherals might request service from a single resource. For example, multiple peripherals might share a single microprocessor that services their interrupt requests. As another example,

Figure 6.10: Arbitration using a daisy-chain configuration.



multiple peripherals might share a single DMA controller that services their DMA requests. In such situations, two or more peripherals may request service simultaneously. We therefore must have some method to arbitrate among these contending requests, i.e., to decide which one of the contending peripherals gets service, and thus which peripherals need to wait. Several methods exist, which we now discuss.

6.5.1 Priority arbiter

One arbitration method uses a single-purpose processor, called a priority arbiter. We illustrate a priority arbiter arbitrating among multiple peripherals using vectored interrupt to request servicing from a microprocessor, as illustrated in Figure 6.9. Each of the peripherals makes its request to the arbiter. The arbiter in turn asserts the microprocessor interrupt, and waits for the interrupt acknowledgment. The arbiter then provides an acknowledgement to exactly one peripheral, which permits that peripheral to put its interrupt vector address on the data bus (which, as you'll recall, causes the microprocessor to jump to a subroutine that services that peripheral).

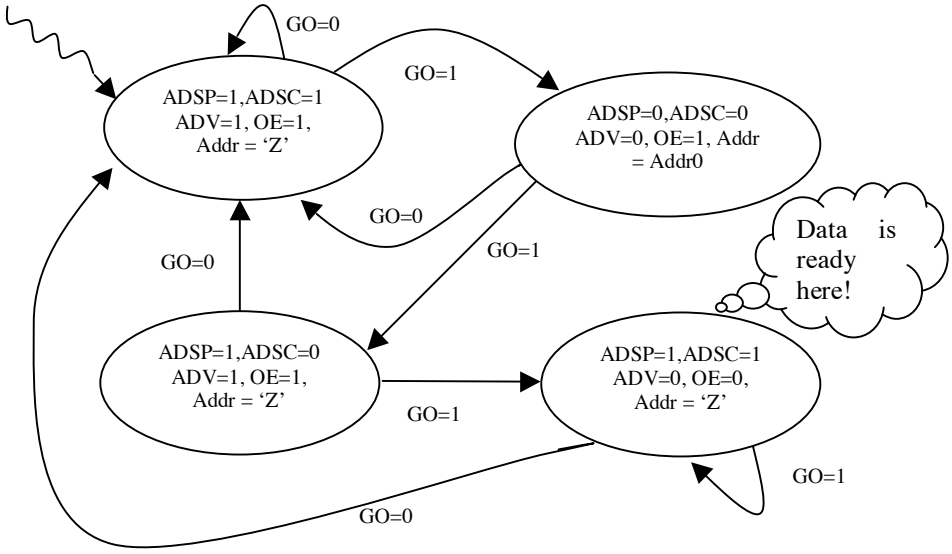
Priority arbiters typically use one of two common schemes to determine priority among peripherals: fixed priority or rotating priority. In *fixed priority* arbitration, each peripheral has a unique rank among all the peripherals. The rank can be represented as a number, so if there are four peripherals, each peripheral is ranked 1, 2, 3 or 4. If two peripherals simultaneously seek servicing, the arbiter chooses the one with the higher rank.

In *rotating priority* arbitration (also called round-robin), the arbiter changes priority of peripherals based on the history of servicing of those peripherals. For example, one rotating priority scheme grants service to the least-recently serviced of the contending peripherals. This scheme obviously requires a more complex arbiter.

We prefer fixed priority when there is a clear difference in priority among peripherals. However, in many cases the peripherals are somewhat equal, so arbitrarily ranking them could cause high-ranked peripherals to get much more servicing than low-ranked ones. Rotating priority ensures a more equitable distribution of servicing in this case.

Example: A complex memory protocol

In this example, we will build a *finite-state machine* FSM controller that will generate all the necessary control signals to drive the TC55V2325FF memory chip in burst read mode, i.e., pipelined read operation, as described in the previous example. Our specification for this FSM is the timing diagram presented in the previous example. The input to our machine is a clock signal (CLK), the starting address (Addr0) and the enable/disable signal (GO). The output of our machine is a set of control signals specific to our memory device. We assume that the chip's enable and WE signals are asserted. Here is the finite-state machine description.



From the above state machine description we can derive the next state and output truth tables. From these truth tables, we can compute output and next state equations. By deriving the next state transition table we can solve and optimize the next state and output equations. These equations, then, can be implemented using logic components. (See chapter 4 for details.)

Any processor that is to be interfaced with one of these memory devices must implement, internally or externally, a state-machine similar to the one presented in this example.

Priority arbiters represent another instance of a standard single-purpose processor. They are also often found built into other single-purpose processors like DMA controllers. A common type of priority arbiter arbitrates interrupt requests; this peripheral is referred to as an *interrupt controller*.

Example

`Description of a real Intel 8259 priority arbiter and an example of its use.`

6.5.2 Daisy-chain arbitration

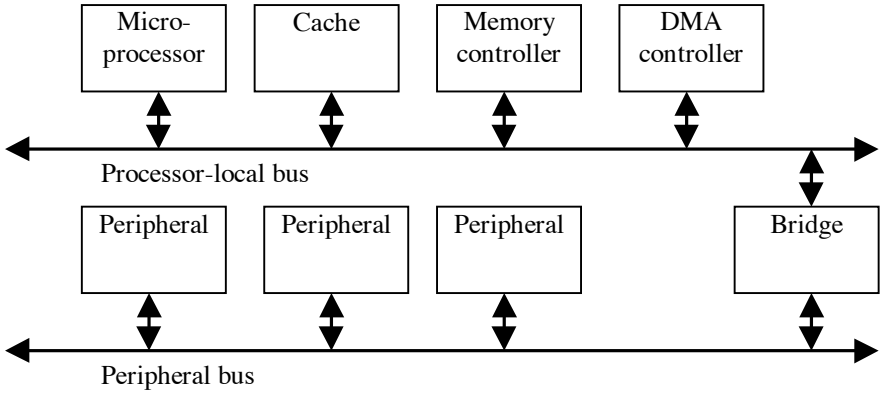
The daisy-chain arbitration method builds arbitration right into the peripherals. A daisy-chain configuration is shown in Figure 6.10, again using vectored interrupt to illustrate the method. Each peripheral has a request output and an acknowledge input, as before. But now each peripheral also has a request *input* and an acknowledge *output*. A peripheral asserts its request output if it requires servicing, OR if its request input is asserted; the latter means that one of the "upstream" devices is requesting servicing. Thus, if any peripheral needs servicing, its request will flow through the downstream peripherals and eventually reach the microprocessor. Even if more than one peripheral requests servicing, the microprocessor will see only one request. The microprocessor acknowledge connects to the first peripheral. If this peripheral is requesting service, it proceeds to put its interrupt vector address on the system bus. But if it doesn't need service, then it instead passes the acknowledgement upstream to the next peripheral, by asserting its acknowledge output. In the same manner, the next peripheral may either begin being serviced or may instead pass the acknowledgement along. Obviously, the peripheral at the front of the chain, i.e., the one to which the microprocessor acknowledge is connected, has highest priority, and the peripheral at the end of the chain has lowest priority.

We prefer a daisy-chain priority configuration over a priority arbiter when we want to be able to add or remove peripherals from an embedded system without redesigning the system. Although conceptually we could add as many peripherals to a daisy-chain as we desired, in reality the servicing response time for peripherals at the end of the chain could become intolerably slow. In contrast to a daisy-chain, a priority arbiter has a fixed number of channels; once they are all used, the system needs to be redesigned in order to accommodate more peripherals. However, a daisy-chain has the drawback of not supporting more advanced priority schemes, like rotating priority. A second drawback is that if a peripheral in the chain stops working, other peripherals may lose their access to the processor.

6.5.3 Network-oriented arbitration methods

The arbitration methods described are typically used to arbitrate among peripherals in an embedded system. However, many embedded systems contain multiple microprocessors communicating via a shared bus; such a bus is sometimes called a network. Arbitration in such cases is typically built right into the bus protocol, since the

Figure 6.11: A two-level bus architecture.



bus serves as the only connection among the microprocessors. A key feature of such a connection is that a processor about to write to the bus has no way of knowing whether another processor is about to simultaneously write to the bus. Because of the relatively long wires and high capacitances of such buses, a processor may write many bits of data before those bits appear at another processor. For example, Ethernet and I2C use a method in which multiple processors may write to the bus simultaneously, resulting in a *collision* and causing any data on the bus to be corrupted. The processors detect this collision, stop transmitting their data, wait for some time, and then try transmitting again. The protocols must ensure that the contending processors don't start sending again at the same time, or must at least use statistical methods that make the chances of them sending again at the same time small.

As another example, the CAN bus uses a clever address encoding scheme such that if two addresses are written simultaneously by different processors using the bus, the higher-priority address will override the lower-priority one. Each processor that is writing the bus also checks the bus, and if the address it is writing does not appear, then that processor realizes that a higher-priority transfer is taking place and so that processor stops writing the bus.

6.6 Multi-level bus architectures

A microprocessor-based embedded system will have numerous types of communications that must take place, varying in their frequencies and speed requirements. The most frequent and high-speed communications will likely be between the microprocessor and its memories. Less frequent communications, requiring less speed, will be between the microprocessor and its peripherals, like a UART. We could try to implement a single high-speed bus for all the communications, but this approach has several disadvantages. First, it requires each peripheral to have a high-speed bus

interface. Since a peripheral may not need such high-speed communication, having such an interface may result in extra gates, power consumption and cost. Second, since a high-speed bus will be very processor-specific, a peripheral with an interface to that bus may not be very portable. Third, having too many peripherals on the bus may result in a slower bus.

Therefore, we often design systems with two levels of buses: a high-speed processor local bus and a lower-speed peripheral bus, as illustrated in Figure 6.11. The processor local bus typically connects the microprocessor, cache, memory controllers, certain high-speed co-processors, and is highly processor specific. It is usually wide, as wide as a memory word.

The peripheral bus connects those processors that do not have fast processor local bus access as a top priority, but rather emphasize portability, low power, or low gate count. The peripheral bus is typically an industry standard bus, such as ISA or PCI, thus supporting portability of the peripherals. It is often narrower and/or slower than a processor local bus, thus requiring fewer gates and less power for interfacing.

A *bridge* connects the two buses. A bridge is a single-purpose processor that converts communication on one bus to communication on another bus. For example, the microprocessor may generate a read on the processor local bus with an address corresponding to a peripheral. The bridge detects that the address corresponds to a peripheral, and thus it then generates a read on the peripheral bus. After receiving the data, the bridge sends that data to the microprocessor. The microprocessor thus need not even know that a bridge exists -- it receives the data, albeit a few cycles later, as if the peripheral were on the processor local bus.

A three-level bus hierarchy is also possible, as proposed by the VSI Alliance. The first level is the processor local bus, the second level a system bus, and the third level a peripheral bus. The system bus would be a high-speed bus, but would offload much of the traffic from the processor local bus. It may be beneficial in complex systems with numerous co-processors.

6.7 Summary

Interfacing processors and memories represents a challenging design task. Timing diagrams provide a basic means for us to describe interface protocols. Thousands of protocols exist, but they can be better understood by understanding basic protocol concepts like actors, data direction, addresses, time-multiplexing, and control methods. Interfacing with a general-purpose processor is the most common interfacing task and involves three key concepts. The first is the processor's approach for addressing external data locations, known as its I/O addressing approach, which may be memory-mapped I/O or standard I/O. The second is the processor's approach for handling requests for servicing by peripherals, known as its interrupt handling approach, which may be fixed or vectored. The third is ability of peripherals to directly access memory, known as direct memory access. Interfacing also leads to the common problem of more than one processor simultaneously seeking access to a shared resource, like a bus, requiring arbitration. Arbitration may be carried out using a priority arbiter or using daisy chain

arbitration. A system often has a hierarchy of buses, such as a high-speed processor local bus, and a lower-speed peripheral bus.

6.8 References and further reading

VSI Alliance, On-Chip Bus Development Working Group, Specification 1 version 1.0, "On-Chip Bus Attributes," August 1998, http://www.vsi.org.

6.9 Excercises

1. Draw the timing diagram for a bus protocol that's handshaked, non-addressed, and transfers 8 bits of data over a 4-bit data bus.

2. (a) Draw a block diagram of a processor, memory, and peripheral connected with a system bus, in which the peripheral gets serviced by using vectored interrupt. Assume servicing moves data from the peripheral to the memory. Show all relavant control and data lines of the bus, and label component inputs/outputs clearly. Use symbolic values for addresses. (b) Provide a timing diagram illustrating what happens over the system bus during the interrupt.

3. (a) Draw a block diagram of a processor, memory, peripheral and DMA controller connected with a system bus, in which the peripheral transfers 100 bytes of data to the memory using DMA. Show all relevant control and data lines of the bus, and label component inputs/outputs clearly. (b) Draw a timing diagram showing what happens during the transfer; skip the 2nd through 99th bytes.

4. (a) Draw a block diagram of a processor, memory, two peripherals and a priority arbiter, in which the peripherals request servicing using vectored interrupt. Show all relavant control and data lines of the bus, and label component inputs/outputs clearly. (b) List the steps that occur during for such an interrupt.

5. Repeat 4(a) and (b) for a daisy chain configuration.

Chapter 8 *Computation models*

8.1 Introduction

We implement a system's processing behavior with processors. But to accomplish this, we must have first described that processing behavior. One method we've discussed for describing processing behavior uses assembly language. Another, more powerful method uses a high-level programming language like C. Both these methods use what is known as a sequential program computation model, in which a set of instructions executes sequentially. A high-level programming language provides more advanced constructs for sequencing among the instructions than does an assembly language, and the instructions are more complex, but nevertheless, the sequential execution model (one statement at a time) is the same.

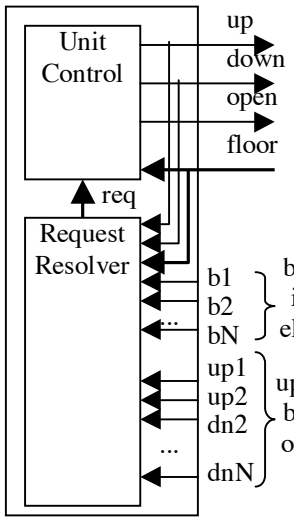
However, embedded system processing behavior is becoming very complex, requiring more advanced computation models to describe that behavior. The increasing complexity results from increasing IC capacity: the more we can put on an IC, the more functionality we want to put into our embedded system. Thus, while embedded systems previously encompassed applications like washing machines and small games requiring perhaps hundreds of lines of code, today they also extend to fairly sophisticated applications like television set-top boxes and digital cameras requiring perhaps hundreds of thousands of lines.

Trying to describe the behavior of such systems can be extremely difficult. The desired behavior is often not even fully understood initially. Therefore, designers must spend much time and effort simply understanding and describing the desired behavior of a system, and some studies have found that most system bugs come from mistakes made describing the desired behavior rather than from mistakes in implementing that behavior. The common method today of using an English (or some other natural language) description of desired behavior provides a reasonable first step, but is not nearly sufficient, because English is not precise. Trying to describe a system precisely in English can be an arduous and often futile endeavor -- just look at any legal document for any example of attempting to be precise in a natural language.

A computation model assists the designer to understand and describe the behavior by providing a means to compose the behavior from simpler objects. A *computation model* provides a set of objects, rules for composing those objects, and execution semantics of the composed objects. For example, the sequential program model provides a set of statements, rules for putting statements one after another, and semantics stating how the statements are executed one at a time. Unfortunately, this model is often not enough. Several other models are therefore also used to describe embedded system behavior. These include the communicating process model, which supports description of multiple sequential programs running concurrently. Another model is the state machine model, used commonly for control-dominated systems. A *control-dominated* system is one whose behavior consists mostly of monitoring control inputs and reacting by setting control outputs. Yet another model is the dataflow model, used for data-dominated systems. A *data-dominated* system's behavior consists mostly of transforming streams of input data into streams of output data, such as a system for filtering noise out of an audio signal as part of a cell phone. An extremely complex system may be best described using an object-oriented model, which provides an elegant means for breaking the complex system into simpler, well-defined objects.

A model is an abstract notion, and therefore we use *languages* to capture the model in a concrete form. For example, the sequential program model can be captured in a variety of languages, such as C, C++, Pascal, Java, Basic, Ada, VHDL, and Verilog. Furthermore, a single language can capture a variety of models. Languages typically are textual, but may also be graphical. For example, graphical languages have been proposed for sequential programming (though they have not been widely adopted).

Figure 8.1: Specifying an elevator controller system: (a) system interface, (b) partial English description, (c) more precise description using a sequential program model.



"Move the elevator either up or down to reach the target floor. Once at the target floor, open the door for at least 10 seconds, and keep it open until the target floor changes. Ensure the door is never open while moving. Don't change directions unless there are no higher requests when moving up or no lower requests when moving down."

(b)

```
Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

void UnitControl()
{
  up = down = 0; open = 1;
  while (1) {
    while (req == floor);
    open = 0;
    if (req > floor) { up = 1;}
    else {down = 1;}
    while (req != floor);
    open = 1;
    delay(10);
    }
  }
}
```

```
void RequestResolver()
{
  while (1)
  ...
    req = ...
  ...

void main()
{
  Call concurrently:
    UnitControl() and
    RequestResolver()
}
```

(c)

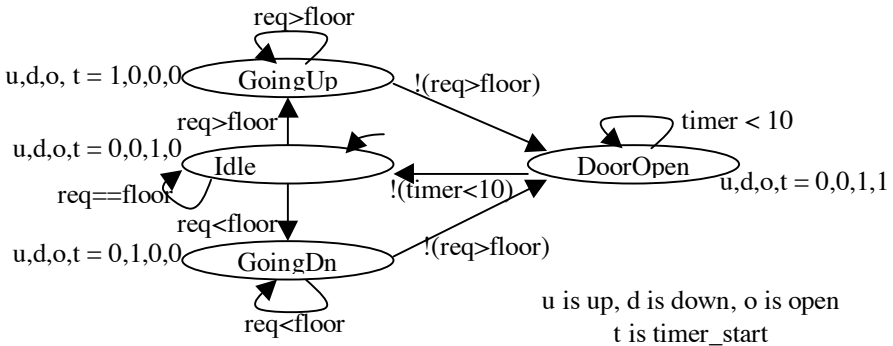
An earlier chapter focused on the sequential program model. This chapter will focus on the state machine and concurrent process models, both of which are commonly used in embedded systems.

8.2 Sequential program model

We described the sequential program model in Chapter 2. Here, we introduce an example system that we'll use in the chapter, and we'll use the sequential program model to describe part of the system. Consider the simple elevator controller system in Figure 8.1(a). It has several control inputs corresponding to the floor buttons inside the elevator and corresponding to the up and down buttons on each of the N floors at which the elevator stops. It also has a data input representing the current floor of the elevator. It has three control outputs that make the elevator move up or down, and open the elevator door. A partial English description of the system's desired behavior is shown in Figure 8.1(b).

We decide that this system is best described as two blocks. `RequestResolver` resolves the various floor requests into a single requested floor. `UnitControl` actually moves the elevator unit to this requested floor, as shown in Figure 8.1. Figure 8.1(c) shows a sequential program description for the `UnitControl` process. Note that this process is more precise than the English description. It firsts opens the elevator door, and then enters an infinite loop. In this loop, it first waits until the requested and current floors differ. It then closes the door and moves the elevator up or down. It then waits until the current floor equals the requested floor, stops moving the elevator, and opens the door for 10 seconds (assuming there's a routine called `delay`). It then goes back to the beginning of the infinite loop. The `RequestResolver` would be written similarly.

Figure 8.2: The elevator's UnitControl process described using a state machine.



8.3 State machine model

In a *state machine* model, we describe system behavior as a set of possible states; the system can only be in one of these states at a given time. We also describe the possible transitions from one state to another depending on input values. Finally, we describe the actions that occur when in a state or when transitioning between states.

For example, Figure 8.2 shows a state machine description of the `UnitControl` part of our elevator example. The initial state, `Idle`, sets `up` and `down` to 0 and `open` to 1. The state machine stays in state `Idle` until the requested floor differs from the current floor. If the requested floor is greater, then the machine transitions to state `GoingUp`, which sets `up` to 1, whereas if the requested floor is less, then the machine transitions to state `GoingDown`, which sets `down` to 1. The machine stays in either state until the current floor equals the requested floor, after which the machine transitions to state `DoorOpen,` which sets `open` to 1. We assume the system includes a timer, so we start the timer while transitioning to `DoorOpen`,. We stay in this state until the timer says 10 seconds have passed, after which we transition back to the `Idle` state.

8.3.1 Finite-state machines: FSM

We have described state machines somewhat informally, but now provide a more formal definition. We start by defining the well-known *finite-state machine* computation model, or *FSM*, and then we'll define extensions to that model to obtain a more useful model for embedded system design. An FSM is a 6-tuple, $<S, I, O, F, H, s_0>$, where:

S is a set of states $\{s_0, s_1, \ldots, s_l\}$,
I is a set of inputs $\{i_0, i_1, \ldots, i_m\}$,
O is a set of outputs $\{o_0, o_1, \ldots, o_n\}$,
F is a next-state function (i.e., transitions), mapping states and inputs to states ($S \times I \rightarrow S$),
H is an output function, mapping current states to outputs ($S \rightarrow O$), and
s_0 is an initial state.

The above is a *Moore*-type FSM above, which associates outputs with states. A second type of FSM is a *Mealy*-type FSM, which associates outputs with transitions, i.e., H maps $S \times I \rightarrow O$. You might remember that Moore outputs are associated with states by noting that the name *Moore* has two *o*'s in it, which look like states in a state diagram. Many tools that support FSM's support combinations of the two types, meaning we can associate outputs with states, transitions, or both.

We can use some shorthand notations to simplify FSM descriptions. First, there may be many system outputs, so rather than explicitly assigning every output in every state, we can say that any outputs not assigned in a state are implicitly assigned 0. Second, we often use an FSM to describe a single-purpose processor (i.e., hardware). Most hardware is synchronous, meaning that register updates are synchronized to clock pulses, e.g., registers are only updated on the rising (or falling) edge of a clock. Such an FSM would have every transition condition AND'ed with the clock edge (e.g., clock'rising and x and y). To avoid having to add this clock edge to every transition condition, we can simply say that the FSM is synchronous, meaning that every transisition condition is implictly AND'ed with the clock edge.

8.3.2 Finite-state machines with datapaths: FSMD

When using an FSM for embedded system design, the inputs and outputs represent boolean data types, and the functions therefore represent boolean functions with boolean operations. This model is sufficient for purely control systems that do not input or output data. However, when we must deal with data, two new features would be helpful: more complex data types (such as integers or floating point numbers), and variables to store data. Gajski refers to an FSM model extended to support more complex data types and variables as an FSM with datapath, or *FSMD*. Most other authors refer to this model as an extended FSM, but there are many kinds of extensions and therefore we prefer the more precise name of FSMD. One possible FSMD model definition is as follows:

<S, I, O, V, F, H, s_0> where:

S is a set of states {s_0, s_1, … , s_l},
I is a set of inputs {i_0, i_1, …, i_m},
O is a set of outputs {o_0, o_1, …, o_n},
V is a set of variables {v_0, v_1, …, v_n},
F is a next-state function, mapping states and inputs and variables to states (S X I X V→ S),
H is an action function, mapping current states to outputs and variables (S → O ⌡ V), and
s_0 is an initial state.

In an FSMD, the inputs, outputs and variables may represent various data types (perhaps as complex as the data types allowed in a typical programming language), and the functions F and H therefore may include arithmetic operations, such as addition, rather than just boolean operations as in an FSM. We now call H an action function rather than an output function, since it describes not just outputs, but also variable updates. Note that the above definition is for a Moore-type FSMD, and it could easily be modified for a Mealy type or a combination of the two types. During execution of the model, the complete system state consists not only of the current state s_i, but also the values of all variables. Our earlier state machine description of `UnitControl` was an FSMD, since it had inputs whose data types were integers, and had arithmetic operations (comparisons) in its transition conditions.

8.3.3 Describing a system as a state machine

Describing a system's behavior as a state machine (in particular, as an FSMD) consists of several steps:

1. List all possible states, giving each a descriptive name.
2. Declare all variables.
3. For each state, list the possible transitions, with associated conditions, to other states.
4. For each state and/or transition, list the associated actions

5. For each state, ensure that exiting transition conditions are exclusive (no two conditions could be true simultaneously) and complete (one of the conditions is true at any time).

If the transitions leaving a state are not exclusive, then we have a *non-deterministic* state machine. When the machine executes and reaches a state with more than one transition that could be taken, then one of those transitions is taken, but we don't know which one that would be. The non-determinism prevents having to over-specify behavior in some cases, and may result in don't-cares that may reduce hardware size, but we won't focus on non-deterministic state machines in this book.

If the transitions leaving a state are not complete, then that usually means that we stay in that state until one of the conditions becomes true. This way of reducing the number of explicit transitions should probably be avoided when first learning to use state machines.

8.3.4 Comparing the state machine and sequential program models

Many would agree that the state machine model excels over the sequential program model for describing a control-based system like the elevator controller. The state machine model is designed such that it encourages a designer to think of all possible states of the system, and to think of all possible transitions among states based on possible input conditions. The sequential program model, in contrast, is designed to transform data through a series of instructions that may be iterated and conditionally executed. Each encourages a different way of thinking of a system's behavior.

A common point of confusion is the distinction between state machine and sequential program models versus the distinction between graphical and textual languages. In particular, a state machine description excels in many cases, not because of its graphical representation, but rather because it provides a more natural means of computing for those cases; it can be captured textually and still provide the same advantage. For example, while in Figure 8.2 we described the elevator's `UnitControl` as a state machine captured in a graphical state-machine language, called a state diagram, we could have instead captured the state machine in a textual state-machine language. One textual language would be a state table, in which we list each state as an entry in a table. Each state's row would list the state's actions. Each row would also list all possible input conditions, and the next state for each such condition. Conversely, while in Figure 8.1 we described the elevator's `UnitControl` as a sequential program captured using a textual sequential programming language (in this case C), we could have instead captured the sequential program using a graphical sequential programming language, such as a flowchart.

Figure 8.3: Capturing the elevator's UnitControl state machine in a sequential programming language (in this case C).

```
#define IDLE            0
#define GOINGUP         1
#define GOINGDN         2
#define DOOROPEN        3
void UnitControl()
{
  int state = IDLE;
  while (1) {
    switch (state) {
      IDLE:       up=0; down=0; open=1; timer_start=0;
        if  (req==floor)    {state = IDLE;}
        if  (req > floor)   {state = GOINGUP;}
        if  (req < floor)   {state = GOINGDN;}
        break;
      GOINGUP: up=1; down=0; open=0; timer_start=0;
        if  (req > floor)   {state = GOINGUP;}
        if  (!(req>floor))  {state = DOOROPEN;}
        break;
      GOINGDN: up=1; down=0; open=0; timer_start=0;
        if  (req > floor)   {state = GOINGDN;}
        if  (!(req>floor))  {state = DOOROPEN;}
        break;
      DOOROPEN: up=0; down=0; open=1; timer_start=1;
        if (timer < 10) {state = DOOROPEN;}
        if (!(timer<10)){state = IDLE;}
        break;
    }
  }
}
```

8.3.5 Capturing a state machine model in a sequential programming language

As elegant as the state machine model is for capturing control-dominated systems, the fact remains that the most popular embedded system development tools use sequential programming languages like C, C++, Java, Ada, VHDL or Verilog. These tools are typically complex and expensive, supporting tasks like compilation, synthesis, simulation, interactive debugging, and/or in-circuit emulation. Unfortunately, sequential programming languages do not directly support the capture of state machines, i.e., they don't possess specific constructs corresponding to states or transitions. Fortunately, we can still describe our system using a state machine model while capturing the model in a sequential program language, by using one of two approaches.

In a *front-end tool* approach, we install an additional tool that supports a state machine language. These tools typically define graphical and perhaps textual state machine languages, and include nice graphic interfaces for drawing and displaying states as circles and transitions as directed arcs. They may support graphical simulation of the state machine, highlighting the current state and active transition. Such tools automatically generate code in a sequential program language (e.g., C code) with the same functionality as the state machine. This sequential program code can then be input

Figure 8.4: General template for capturing a state machine in a sequential programming language.

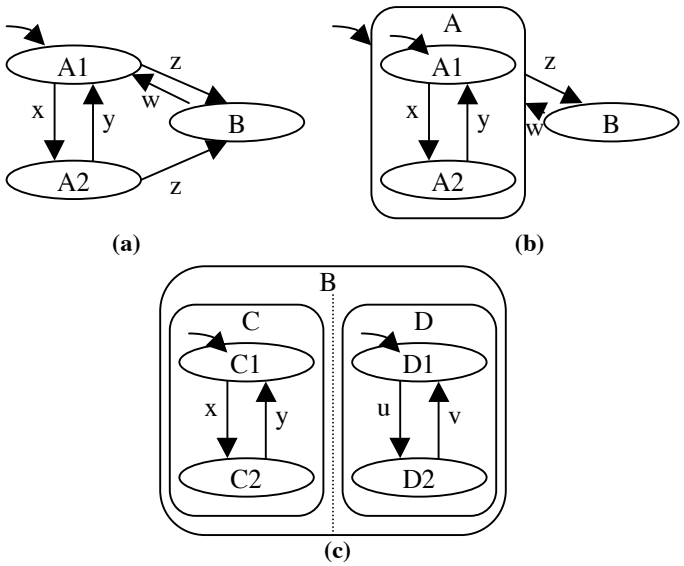
```
#define S0                  0
#define S1                  1
...
#define SN                  N
void StateMachine()
{
  int state = S0; // or whatever is the initial state.
  while (1) {
    switch (state) {
      S0:
        // Insert S0's actions here
        // Insert transitions Ti leaving S0:
        if  (T0's condition is true ) {state = T0's next state; // insert T0's actions here.}
        if  (T1's condition is true ) {state = T1's next state; // insert T1's actions here. }
        ...
        if  (Tm's condition is true ) {state = Tm's next state; // insert Tm's actions here. }
        break;
      S1:
        // Insert S1's actions here
        // Insert transitions Ti leaving S1
        break;
      ...
      SN:
        // Insert SN's actions here
        // Insert transitions Ti leaving SN
        break;
    }
  }
}
```

to our main development tool. In many cases, the front-end tool is designed to interface directly with our main development tool, so that we can control and observe simulations occurring in the development tool directly from the front-end tool. The drawback of this approach is that we must support yet another tool, which includes additional licensing costs, version upgrades, training, integration problems with our development environment, and so on.

In contrast, we can use a *language subset* approach. In this approach, we directly capture our state machine model in a sequential program language, by following a strict set of rules for capturing each state machine construct in an equivalent set of sequential program constructs. This approach is by far the most common approach for capturing state machines, both in software languages like C as well as hardware languages like VHDL and Verilog. We now describe how to capture a state machine model in a sequential program language.

We start by capturing our `UnitControl` state machine in the sequential programming language C, illustrated in Figure 8.3. We enumerate all states, in this case using the `#define` C construct. We capture the state machine as a subroutine, in which we declare a state variable initialized to the initial state. We then create an infinite loop, containing a single switch statement that branches to the case corresponding to the value of the state variable. Each state's case starts with the actions in that state, and then the transitions from that state. Each transition is captured as an if statement that checks if the

Figure 8.5: Adding hierarchy and concurrency to the state machine model: (a) three-state example without hierarchy, (b) same example with hierarchy, (c) concurrency.



transition's condition is true and then sets the next state. Figure 8.4 shows a general template for capturing a state machine in C.

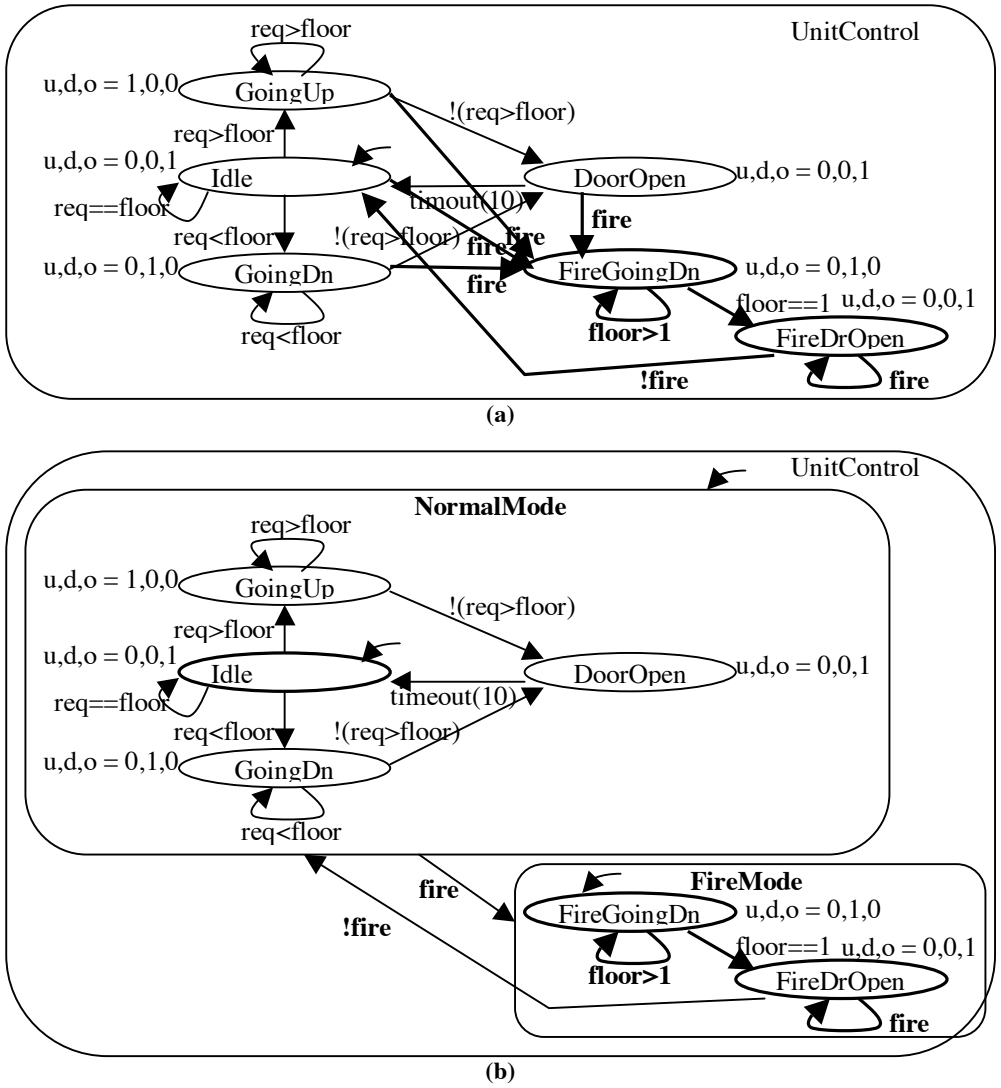
8.3.6 Hiererarchical/Concurrent state machines (HCFSM) and Statecharts

Harel proposed extensions to the state machine model to support hierarchy and concurrency, and developed Statecharts, a graphical state machine language designed to capture that model. We refer to the model as a hierarchical/concurrent FSM, or HCFSM.

The *hierarchy* extension allows us to decompose a state into another state machine, or conversely stated, to group several states into a new hierarchical state. For example, consider the state machine in Figure 8.5(a), having three states A1 (the initial state), A2, and B. Whenever we are in either A1 or A2 and event z occurs, we transition to state B. We can simplify this state machine by grouping A1 and A2 into a hierarchical state A, as shown in Figure 8.5(b). State A is the initial state, which in turn has an initial state A1. We draw the transition to B on event z as originating from state A, not A1 or A2. The meaning is that regardless of whether we are in A1 or A2, event z causes a transition to state B.

As another hierarchy example, consider our earlier elevator example, and suppose that we want to add a control input `fire`, along with new behavior that immediately moves the elevator down to the first floor and opens the door when `fire` is true. As shown in Figure 8.6(a), we can capture this behavior by adding a transition from every state originally in `UnitControl` to a new state called `FireGoingDn`, which moves the elevator to the first floor, followed by a state `FireDrOpen`, which holds the door open on the first floor. When `fire` becomes false, we go to the `Idle` state. While this new state machine captures the desired behavior, it is becoming more complex due to many more transitions, and harder to comprehend due to more states. We can use hierarchy to reduce the number of transitions and enhance understandability. As shown in Figure 8.6(b), we can group the original state machine into a hierarchical state called `NormalMode`, and group the fire-related states into a state called `FireMode`. This grouping reduces the number of transitions, since instead of four transitions from each original state to the fire-related states, we need only one transition, from `NormalMode`

Figure 8.6: The elevator's UnitControl with new behavior for a new input *fire*: (a) without hierarchy, (b) with hierarchy.



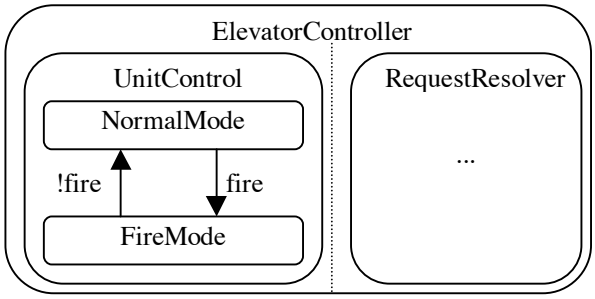
to `FireMode`. This grouping also enhances understandability, since it clearly represents two main operating modes, one normal and one in case of fire.

The *concurrency* extension allows us to use hierarchy to decompose a state into two concurrent states, or conversely stated, to group two concurrent states into a new hierarchical state. For example, Figure 8.5 (c), shows a state B decomposed into two concurrent states C and D. C happens to be decomposed into another state machine, as does D. Figure 8.7 shows the entire `ElevatorController` behavior captured as a HCFSM with two concurrent states.

Therefore, we see that there are two methods for using hierarchy to decompose a state into substates. *OR*-decomposition decomposes a state into sequential states, in which only one state is active at a time (either the first state OR the second state OR the third state, etc.). *AND*-decomposition decomposes a state into concurrent states, all of which are active at a time (the first state AND the second state AND the third state, etc.).

The Statecharts language includes numerous additional constructs to improve state machine capture. A *timeout* is a transition with a time limit as its condition. The transition

Figure 8.7: Using concurrency in an HCFSM to describe both processes of the ElevatorController.



is automatically taken if the transition source state is active for an amount of time equal to the limit. Note that this would have simplified the `UnitControl` state machine; rather than starting and checking an external timer, we could simply have created a transition from `DoorOpen` to `Idle` with the condition `timeout(10)`. *History* is a mechanism for remembering the last substate that an OR-decomposed state A was in before transitioning to another state B. Upon re-entering state A, we can start with the remembered substate rather than A's initial state. Thus, the transition leaving A is treated much like an interrupt and B as an interrupt service routine.

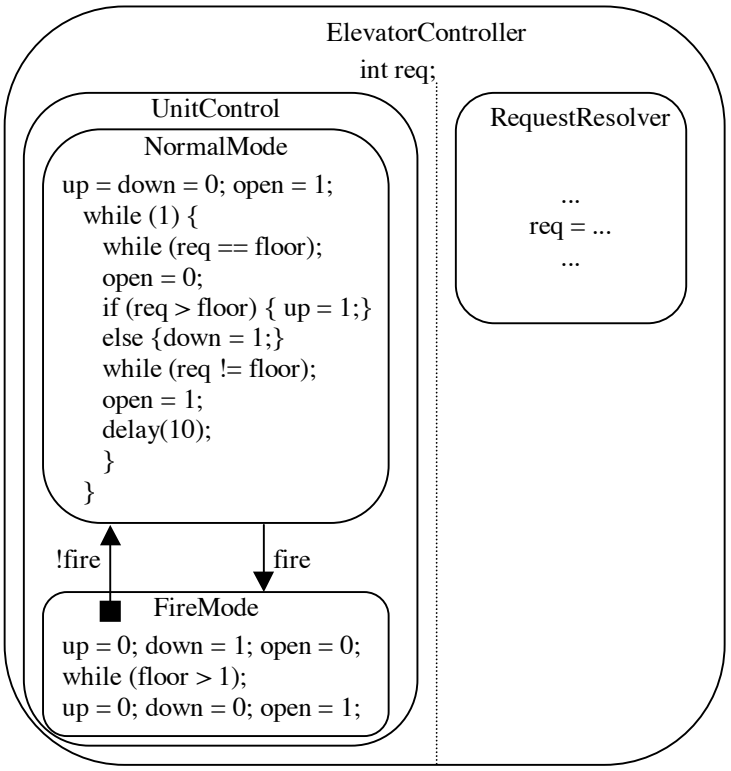
8.3.7 Program-state machines (PSM)

The program-state machine (PSM) model extends state machines to allow use of sequential program code to define a state's actions (including extensions for complex data types and variables), as well as including the hierarchy and concurrency extensions of HCFSM. Thus, PSM is a merger of the HCFSM and sequential program models, subsuming both models. A PSM having only one state (called a program-state in PSM terminology), where that state's actions are defined using a sequential program, is the same as a sequential program. A PSM having many states, whose actions are all just assignment statements, is the same as an HCFSM. Lying between these two extremes are various combinations of the two models.

For example, Figure 8.8 shows a PSM description of the `ElevatorController` behavior, which we AND-decompose into two concurrent program-states `UnitControl` and `RequestResolver`, as in the earlier HCFSM example. Furthermore, we OR-decompose `UnitControl` into two sequential program-states, `NormalMode` and `FireMode`, again as in the HCFSM example. However, unlike the HCFSM example, we describe `NormalMode` as a sequential program (identical to that of Figure 8.1(c)) rather than a state machine. Likewise, we describe `FireMode` as a sequential program. We didn't have to use sequential programs for those program-states, and could have used state machines for one or both -- the point is that PSM allows the designer to choose whichever model is most appropriate.

PSM enforces a stricter hierarchy than the HCFSM model used in Statecharts. In Statecharts, transitions may point to or from a substate within a state, such as the transition in Figure 8.6(b) pointing from the substate of the state to the `NormalMode` state. Having this transition start from `FireDrOpen` rather than `FireMode causes` the elevator to always go all the way down to the first floor when the `fire` input becomes true, even if the input is true just momentarily. PSM, on the other hand, only allows transitions between sibling states, i.e., between states with the same parent state. PSM's model of hierarchy is the same as in sequential program languages that use subroutines for hierarchy; namely, we always enter the subroutine from one point, and when we exit the subroutine we do not specify to where we are exiting.

Figure 8.8: Using PSM to describe the ElevatorController.



As in the sequential programming model, but unlike the HCFSM model, PSM includes the notion of a program-state *completing*. If the program-state is a sequential program, then reaching the end of the code means the program-state is complete. If the program-state is OR-decomposed into substates, then a special *complete* substate may be added. Transitions may occur from a substate to the complete substate (but no transitions may leave the complete substate), which when entered means that the program-state is complete. Consequently, PSM introduces two types of transitions. A t*ransition-immediately* (TI) transition is taken immediately if its condition becomes true, regardless of the status of the source program-state -- this is the same as the transition type in an HCFSM. A second, new type of transition, *transition-on-completion* (TOC), is taken only if the condition is true AND the source program-state is complete. Graphically, a TOC transition is drawn originating from a filled square inside a state, rather than from the state's perimeter. We used a TOC transition in Figure 8.8 to transition from `FireMode` to `NormalMode` only after `FireMode` completed, meaning that the elevator had reached the first floor. By supporting both types of transitions, PSM elegantly merges the reactive nature of HCFSM models (using TI transitions) with the transformational nature of sequential program models (using TOC transitions).

8.4 Concurrent process model

In a *concurrent process* model, we describe system behavior as a set of processes, which communicate with one another. A process refers to a repeating sequential program. While many embedded systems are most easily thought of as one process, other systems are more easily thought of as having multiple processes running concurrently.

For example, consider the following made-up system. The system allows a user to provide two numbers X and Y. We then want to write "Hello World" to a display every X seconds, and "How are you" to the display every Y seconds. A very simple way to

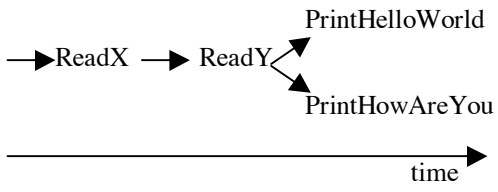
Figure 8.9: A simple concurrent process example: (a) pseudo-code, (b) subroutine execution over time, (c) sample input and output.

```
ConcurrentProcessExample()
  x = ReadX()
  y = ReadY()
  Call concurrently:
    PrintHelloWorld(x) and
    PrintHowAreYou(y)

PrintHelloWorld(x)
  while (1) {
    print "Hello world."
    delay(x);
  }

PrintHowAreYou(x)
  while (1) {
    print "How are you?"
    delay(y);
  }
```

(a)



(b)

```
Enter X: 1
Enter Y: 2
Hello world.      (Time = 1 s)
Hello world.      (Time = 2 s)
How are you?      (Time = 2 s)
Hello world.      (Time = 3 s)
How are you?      (Time = 4 s)
Hello world.      (Time = 4 s)
...
```

(c)

describe this system using concurrent processes is shown in Figure 8.9(a). After reading in X and Y, we call two subroutines concurrently. One subroutine prints "Hello World" every X seconds, the other prints "How are you" every Y seconds. (Note that you can't call two subroutines concurrently in a pure sequential program model, such as the model supported by the basic version of the C language). As shown in Figure 8.9(b), these two subroutines execute simultaneously. Sample output for X=1 and Y=2 is shown in Figure 8.9(c).

To see why concurrent processes were helpful, try describing the same system using a sequential program model (i.e., one process). You'll find yourself exerting effort figuring out how to schedule the two subroutines into one sequential program. Since this example is a trivial one, this extra effort is not a serious problem, but for a complex system, this extra effort can be significant and can detract from the time you have to focus on the desired system behavior.

Recall that we described our elevator controller using two "blocks." Each block is really a process. The controller was simply easier to comprehend if we thought of the two blocks independently.

8.4.1 Processes and threads

In operating system terminology, a distinction is made between regular processes and threads. A regular process is a process that has its own virtual address space (stack, data, code) and system resources (e.g., open files). A *thread*, in contrast, is really a sub-process within a process. It is a lightweight process that typically has only a program counter, stack and register; it shares its address space and system resources with other threads. Since threads are small compared to regular processes, they can be created quickly, and switching between threads by an operating system does not incur very heavy costs. Furthermore, threads can share resources and variables so they can communicate quickly and efficiently.

8.4.2 Communication

Two concurrent processes communicate using one of two techniques: message passing, or shared data. In the *shared data* technique, processes read and write variables

that both processes can access, called global variables. For example, in the elevator example above, the `RequestResolver` process writes to a variable `req`, which is also read by the `UnitControl` process.

In *message passing*, communication occurs using send and receive constructs that are part of the computation model. Specifically, a process P explicitly sends data to another process Q, which must explicitly receive the data. In the elevator example, `RequestResolver` would include a statement: `Send(UnitControl,rr_req)`. Likewise, `UnitControl` would include statements of the form: `Receive(RequestResolver, uc_req)`. `rr_req` and `uc_req` are variables local to each process.

Message passing may be blocking or non-blocking. In *blocking* message passing, a sending process must wait until the receiving process receives the data before executing the statement following the send. Thus, the processes synchronize at their send/receive points. In fact, a designer may use a send/receive with no actual message being passed, in order to achieve the synchronization. In *non-blocking* message passing, the sending process need not wait for the receive to occur before executing more statements. Therefore, a queue is implied in which the sent data must be stored before being received by the receiving process.

8.4.3 Implementing concurrent processes

The most straightforward method for implementing concurrent processes on processors is to implement each process on its own processor. This method is common when each process is to be implemented using a single-purpose processor.

However, we often decide that several processes should be implemented using general-purpose processors. While we could conceptually use one general-purpose processor per process, this would likely be very expensive and in most cases is not necessary. It is not necessary because the processes likely do not require 100% of the processor's processing time; instead, many processes may share a single processor's time and still execute at the necessary rates.

One method for sharing a processor among multiple processes is to *manually rewrite* the processes as a single sequential program. For example, consider our Hello World program from earlier. We could rewrite the concurrent process model as a sequential one by replacing the concurrent running of the `PrintHelloWorld` and `PrintHowAreYou` routines by the following:

```
I = 1;
T = 0;
while (1) {
  Delay(I); T = T + I
  if X modulo T is 0 then call PrintHelloWorld
  if Y modulo T is 0 then call PrintHowAreYou
}
```

We would also modify each routine to have no parameter, no loop and no delay; each would merely print its message. If we wanted to reduce iterations, we could set I to the greatest common divisor of X and Y rather than to 1.

Manually rewriting a model may be practical for simple examples, but extremely difficult for more complex examples. While some automated techniques have evolved to assist with such rewriting of concurrent processes into a sequential program, these techniques are not very commonly used.

Instead, a second, far more common method for sharing a processor among multiple processes is to rely on a multi-tasking *operating system*. An operating system is a low-level program that runs on a processor, responsible for scheduling processes, allocating storage, and interfacing to peripherals, among other things. A real-time operating system (RTOS) is an operating system that allows one to specify constraints on the rate of

processes, and that guarantees that these rate constraints will be met. In such an approach, we would describe our concurrent processes using either a language with processes built-in (such as Ada or Java), or a sequential program language (like C or C++) using a library of routines that extends the language to support concurrent processes. POSIX threads were developed for the latter purpose.

A third method for sharing a processor among multiple processes is to convert the processes to a sequential program that includes a process scheduler right in the code. This method results in less overhead since it does not rely on an operating system, but also yields code that may be harder to maintain.

8.5 Other models

8.5.1 Dataflow

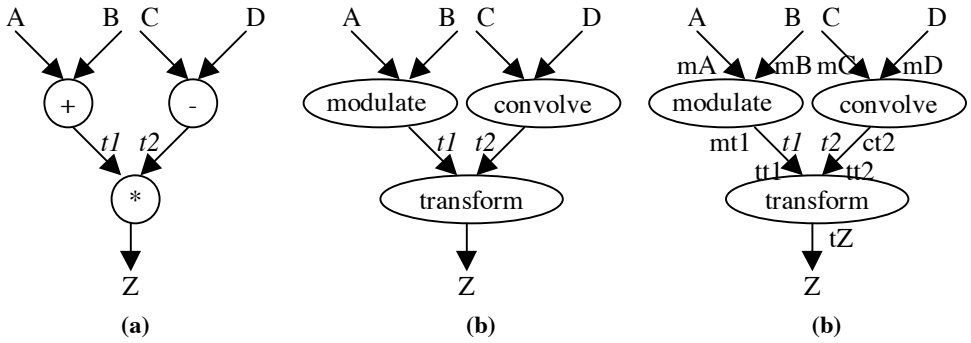
In a *dataflow* model, we describe system behavior as a set of nodes representing transformations, and a set of directed edges representing the flow of data from one node to another. Each node consumes data from its input edges, performs its transformation, and produces data on its output edge. All nodes may execute concurrently.

For example, Figure 8.10(a) shows a dataflow model of the computation Z = (A+B)*(C-D). Figure 8.10(b) shows another dataflow model having more complex node transformations. Each edge may or not have data. Data present on an edge is called a *token*. When all input edges to a node have at least on token, the node may *fire*. When a node fires, it consumes one token from each input edge, executes its data transformation on the consumed token, and generates a token on its output edge. Note that multiple nodes may fire simultaneously, depending only on the presence of tokens.

Several commercial tools support graphical languages for the capture of dataflow models. These tools can automatically translate the model to a concurrent process model for implementation on a microprocessor. We can translate a dataflow model to a concurrent process model by converting each node to a process, and each edge to a channel. This concurrent process model can be implemented either by using a real-time operating system, or by mapping the concurrent processes to a sequential program.

Lee observed that in many digital signal-processing systems, data flows in to and out of the system at a fixed rate, and that a node may consume and produce many tokens per firing. He therefore created a variation of dataflow called *synchronous dataflow*. In this model, we annotate each input and output edge of a node with the number of tokens that node consumes and produces, respectively, during one firing. The advantage of this model is that, rather than translating to a concurrent process model for implementation, we can instead statically schedule the nodes to produce a sequential program model. This model can be captured in a sequential program language like C, thus running without a real-time operating system and hence executing more efficiently. Much effort has gone into developing algorithms for scheduling the nodes into "single-appearance" schedules, in which the C code only has one statement that calls each node's associated procedure (though this call may be in a loop). Such a schedule allows for procedure inlining, which further improves performance by reducing the overhead of procedure calls, without resulting in an explosion of code size that would have occurred had there been many statements that called each node's procedure.

Figure 8.10: Simple dataflow models: (a) nodes representing arithmetic transformations, (b) nodes representing more complex transformations, (c) synchronous dataflow.



8.6 Summary

Embedded system behavior is becoming increasingly complex, and the sequential program model alone is no longer sufficient for describing this behavior. Additional models can make describing this behavior much easier. The state machine model excels at describing control-dominated systems. Extensions to the basic FSM model include: FSMD, which adds complex data types and variables; HCFSM, which adds hierarchy and concurrency; and PSM, which merges the FSMD and HCFSM models. The concurrent process model excels at describing behavior with several concurrent threads of execution. The dataflow model excels at describing data-dominated systems. An extension to dataflow is synchronous dataflow, which assumes a fixed rate of data input and output and specifies the number of tokens consumed and produced per node firing, thus allowing for a static scheduling of the nodes and thus an efficient sequential program implementation.

8.7 References and further reading

8.8 Excercises