

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.

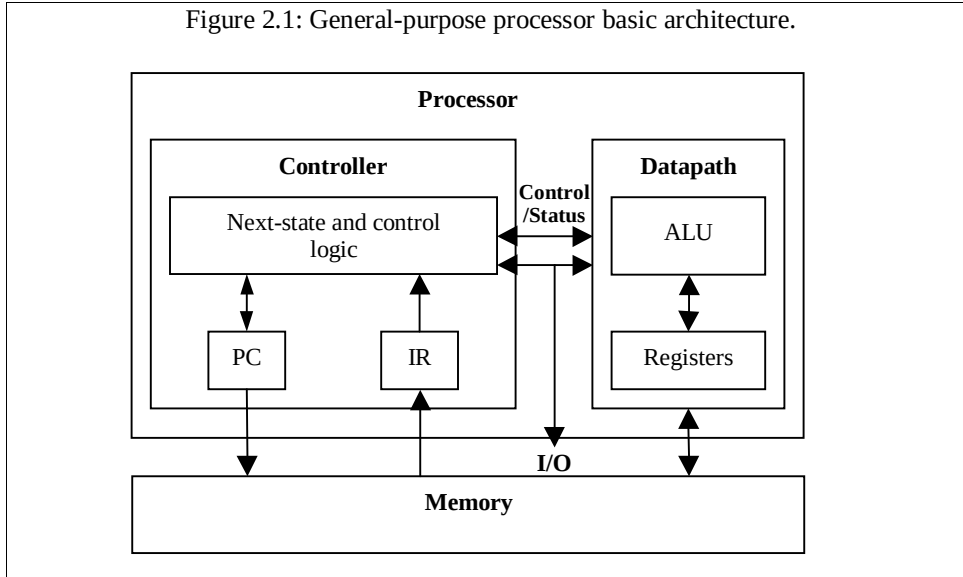
2.2 Basic architecture

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.



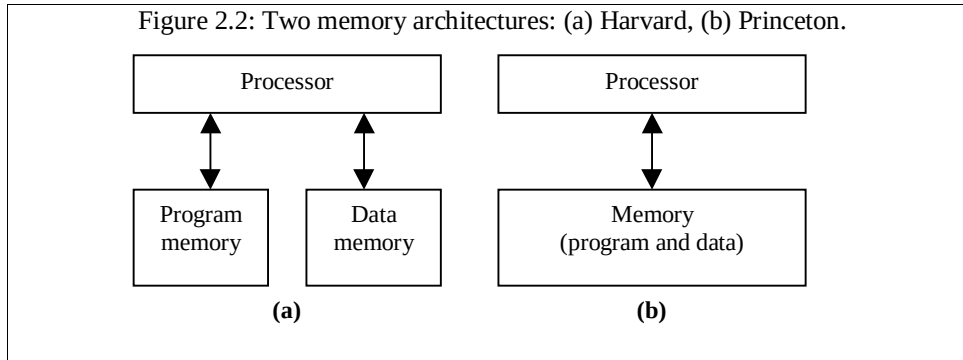
We typically distinguish processors by their size, and we usually measure size as the bit-width of the datapath components. A *bit*, which stands for binary digit, is the processor's basic data unit, representing either a 0 (low or false) or a 1 (high or true), while we refer to 8 bits as a *byte*. An *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



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 $1/10 \times 10^{-9}$ 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

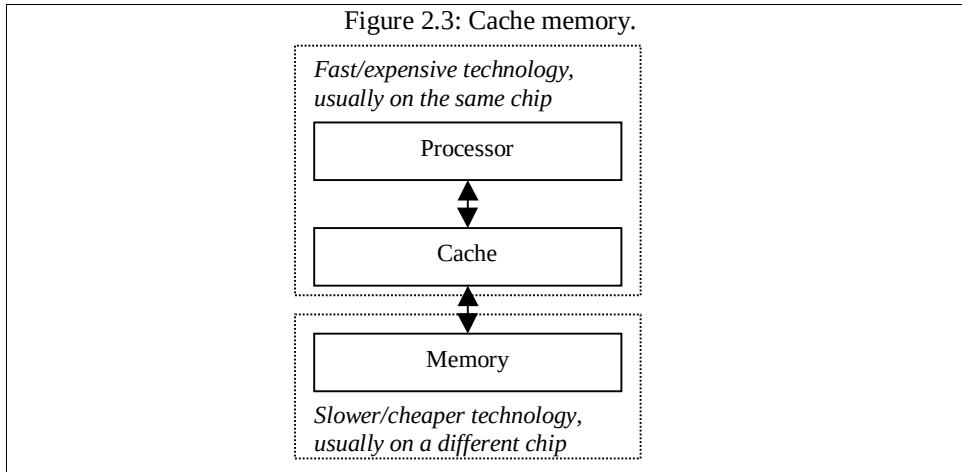
While registers serve a processor's short term storage requirements, memory serves the processor's medium and long-term information-storage requirements. We can classify stored information as either program or data. Program information consists of the sequence of instructions that cause the processor 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 much 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



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:

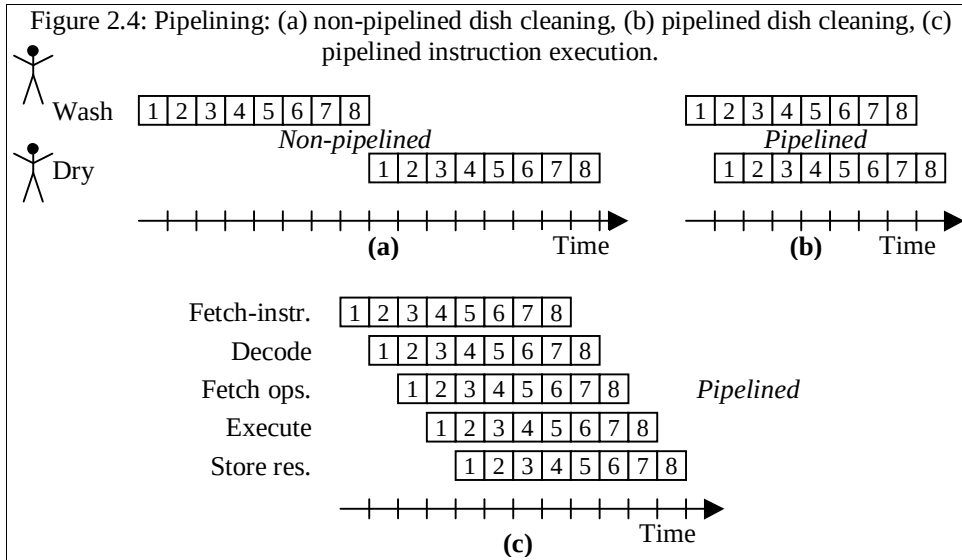
1. *Fetch instruction*: the task of reading the next instruction from memory into the instruction register.
2. *Decode instruction*: the task of determining what operation the instruction in the instruction register represents (e.g., add, move, etc.).
3. *Fetch operands*: the task of moving the instruction's 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



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.

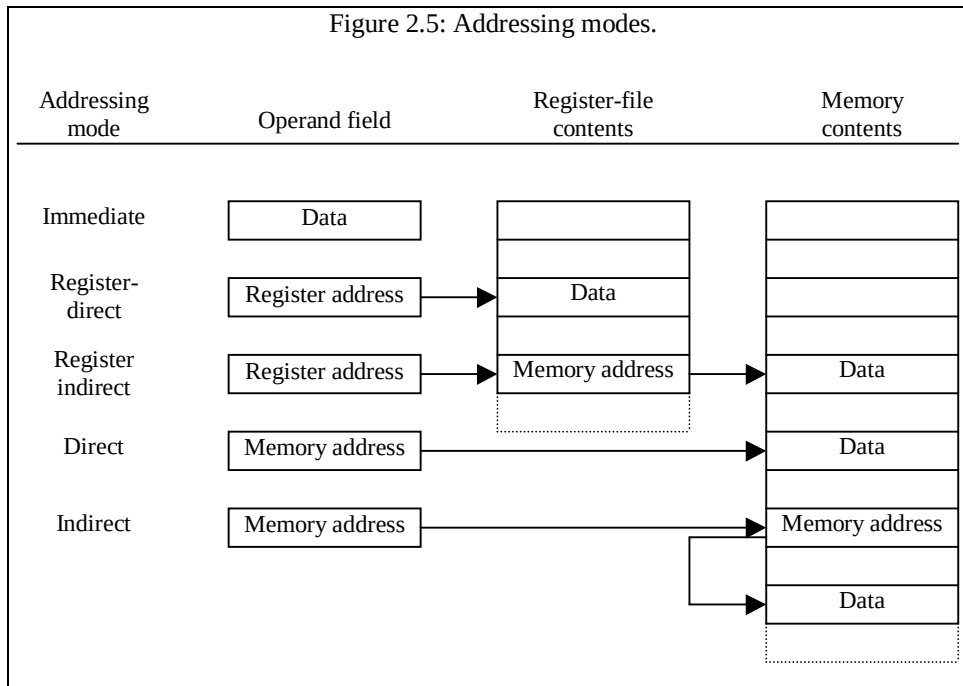
2.4 Programmer's view

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



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 instruction¹ 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.

Assembly instruct.	First byte		Second byte	Operation
MOV Rn, direct	0000	Rn	direct	$Rn = M(\text{direct})$
MOV direct, Rn	0001	Rn	direct	$M(\text{direct}) = Rn$
MOV @Rn, Rm	0010		Rn Rm	$M(Rn) = Rm$
MOV Rn, #immed.	0011	Rn	immediate	$Rn = \text{immediate}$
ADD Rn, Rm	0100		Rn Rm	$Rn = Rn + Rm$
SUB Rn, Rm	0101		Rn Rm	$Rn = Rn - Rm$
JZ Rn, relative	1000	Rn	relative	$PC = PC + \text{relative}$ (only if Rn is 0)

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.

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

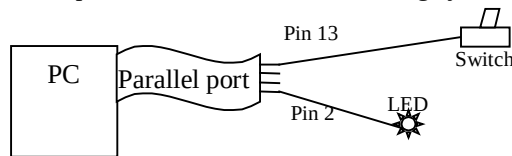
Example: Assembly-language programming of device drivers

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 registers 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 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 devices². 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 manufacturers 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
Mitsubishi 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 $750 \times 1757 = 1,317,750$ 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.