Chapter 3
Microcomputer Organization

The minimal set of components required to establish a computing system is denominated a *Microcomputer*. The basic structural description of an embedded system introduced in Chap. 1 showed us the integration between hardware and software components. This chapter discusses the basic hardware and software elements required to establish a computer system and how they interact to provide the operation of a stored program computer.

3.1 Base Microcomputer Structure

The minimal hardware configuration of a microcomputer system is composed of three fundamental components: a Central Processing Unit (CPU), the system memory, and some form of Input/Output (I/O) Interface. These components are interconnected by multiple sets of lines grouped according to their functions, and globally denominated the *system buses*. An additional set of components provide the necessary power and timing synchronization for system operation. Figure 3.1 illustrates the integration of such a basic structure.

The components of a microcomputer can be implemented in diverse ways. They could be deployed with multiple chips on a board-level microcomputer or integrated into a single chip, in a structure named a microcomputer-on-a-chip, or simply a *Microcontroller*. Nowadays most embedded systems are developed around microcontrollers.

Regardless of the implementation style, each component of a microcomputer has the same specific function, as described below.

Central Processing Unit (CPU): The CPU forms the heart of the microcontroller system. It retrieves instructions from program memory, decodes them, and accordingly operates on data and/or on peripherals devices in the Input-Output subsystem to give functionality to the system.

M. Jiménez et al., *Introduction to Embedded Systems*,
DOI: 10.1007/978-1-4614-3143-5_3,

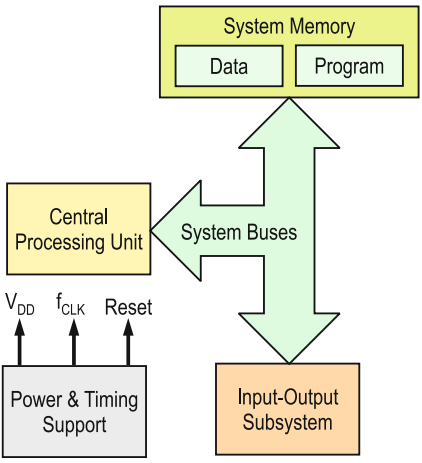

Fig. 3.1 General architecture of a microcomputer system

System Memory: The place where programs and data are stored to be accessed by the CPU is the system memory. Two types of memory elements are identified within the system: Program Memory and Data Memory. Program memory stores programs in the form of a sequence of instructions. Programs dictate the system's operation. Data Memory stores data to be operated on by programs.

Input/Output subsystem The I/O subsystem, also called *Peripheral Subsystem* includes all the components or peripherals that allow the CPU to exchange information with other devices, systems, or the external world. As was introduced in Chap. 1, the I/O Subsystem includes all the components that complement the CPU and memory to form a computer system.

System Buses The set of lines interconnecting CPU, Memory, and I/O Subsystem are denominated the system buses. Groups of lines in the system buses perform different functions. Based on their function the system bus lines are sub-divided into address bus, data bus, and control bus.

Looking back at the hardware structure of an embedded system as was introduced in Sect. 1.2.1, we can notice that all the hardware components of Fig. 1.8, other than CPU and Memory, were condensed, in Fig. 3.1 into the block representing the Input/Output subsystem. This allows us to see that both diagrams are equivalent, confirming our asseveration that embedded systems are indeed microcomputers.

3.2 Microcontrollers Versus Microprocessors

Before we delve any deeper into the structure of the different components of a microcomputer system, let's first establish the fundamental difference between microprocessors and microcontrollers.

3.2.1 Microprocessor Units

A Microprocessor Unit, commonly abbreviated MPU, fundamentally contains a general purpose CPU in its die. To develop a basic system using an MPU, all components depicted in Fig. 1.8 other than the CPU, i.e., the buses, memory, and I/O interfaces, are implemented externally.[1] Other characteristics of MPUs include an optimized architecture to move code and data from external memory into the chip such as queues and caches, and the inclusion of architectural elements to accelerate processing such as multiple functional units, ability to issue multiple instructions at once, and other features such as branch prediction units and numeric co-processors. The general discussion of these features fall beyond the scope of this book. Yet, whenever appropriate, we may introduce some related concepts.

The most common examples of systems designed around an MPUs are personal computers and mainframes. But these are not the only ones. There are many other systems developed around traditional MPUs. Manufacturers of MPU's include INTEL, Freescale, Zilog, Fujitsu, Siemens, and many others. Microprocessor design has advanced from the initial models in the early 1970s to present day technology. Intel's initial 4004 in 1971 was built using 10 μm technology, ran at 400 kHz and contained 2,250 transistors. Intel's Xeon E7 MPU, released in 2011, was built using 32 nm technology, runs at 2 GHz and contains 2.6×10^9 transistors. MPUs indeed, represent the most powerful type of processing components available to implement a microcomputer.

Most small embedded systems, however, do not need the large computational and processing power that characterize microprocessors, and hence the orientation to microcontrollers for these tasks.

3.2.2 Microcontroller Units

A microcontroller unit, abbreviated MCU, is developed using a microprocessor core or central processing unit (CPU), usually less complex than that of an MPU. This basic CPU is then surrounded with memory of both types (program and data) and several types of peripherals, all of them embedded into a single integrated circuit, or chip. This blending of CPU, memory, and I/O within a single chip is what we call a microcontroller.

The assortment of components embedded into a microcontroller allows for implementing complete applications requiring only a minimal number of external components or in many cases solely using the MCU chip. Peripheral timers, input/output (I/O) ports, interrupt handlers, and data converters are among those commonly found in most microcontrollers. The provision of such an assortment of resources inside the same chip is what has gained them the denomination of *computers-on-a-chip*. Figure 3.2 shows a typical microcontroller configuration.

[1] Hence, the name peripheral.

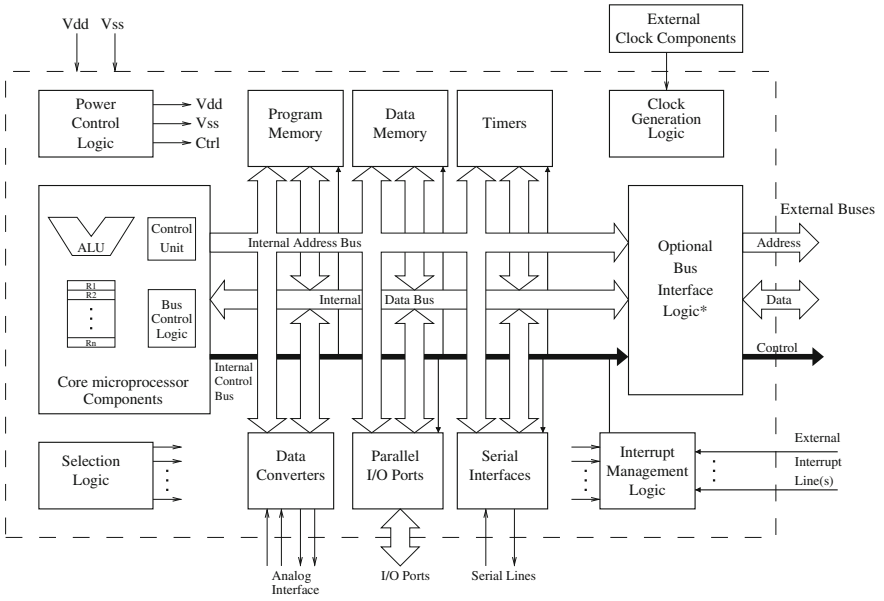

Fig. 3.2 Structure of a typical microcontroller

Microcontrollers share a number of characteristics with general purpose microprocessors. Yet, the core architectural components in a typical MCU are less complex and more application oriented than those in a general purpose microprocessor.

Microcontrollers are usually marketed as family members. Each family is developed around a basic architecture which defines the common characteristics of all members. These include, among others, the data and program path widths, architectural style, register structure, base instruction set, and addressing modes. Features differentiating family members include the amount of on-chip data and program memory and the assortment of on-chip peripherals.

There are literally hundreds, perhaps thousands, of microprocessors and microcontrollers on the market. Table 3.1 shows a very small sample of microcontroller family models of different sizes from six companies. We will concentrate on the MSP430 family for practical hands on introduction.

3.2.3 RISC Versus CISC Architectures

As we have already emphasized, microcomputer systems run with software which is supported by its hardware architecture. These systems are designed according to which of the two components, hardware or software, should be optimized. Under this point of view, we speak of CISC and RISC architectures.

Table 3.1 A sample of MCU families/series

Company	4-bits	8-bits	16-bits	32-bits
EM Microelectronic	EM6807	EM6819		
Samsung	S3P7xx	S3F9xxx	S3FCxx	S3FN23BXZZ
Freescale semiconductor		68HC11	68HC12	
Toshiba		TLCS-870	TLCS-900/L1	TLCS-900/H1
Texas instruments			MSP430	TMS320C28X
			TMS320C24X	Stellaris line
Microchip		PIC1X	PIC2x	PIC32

CISC (*Complex Instruction Set Computing*) machines are characterized by variable length instruction words, i.e., with different number of bits, small code sizes, and multiple clocked-complex instructions at machine level. CISC architecture focuses in accomplishing as much as possible with each instruction, in order to generate simple programs. This focus helps the programmer's task while augmenting hardware complexity.

RISC (*Reduced Instruction Set Computing*) machines, on the other hand, are designed with focus on simple instructions, even if that results in longer programs. This orientation simplifies the hardware structure. The design expects that any single instruction execution is reduced—at most a single data memory cycle—when compared to the "complex instructions" of a CISC system.

It is usually accepted that RISC microcontrollers are faster, although this may not be necessarily true for all instructions.[2]

3.2.4 Programmer and Hardware Model

Most readers might already have programming experience with some high level language such as Java, C, or some other language. Most probably, the experience did not require knowledge of the hardware system supporting the execution of the program.

Embedded systems programmers need to go one step forward and consider both the hardware and software issues. Hence, they need to look at the system both from a hardware point of view, the *hardware model*, as well as a software point of view, the *programmer's model*.

In the hardware model, the user focuses on the hardware characteristics and subsystems that support the instructions and the interactions with the outer world. This knowledge is indispensable from the beginning especially because of the intimate relationship with the programming possibilities. In this model we speak of hardware

[2] Since speed has become an important feature to consider in applications, the term "RISC" has become almost a buzzword in the microcontroller market. The designer should check the truth of such labeling when selecting a microcontroller.

subsystems, characteristics of peripherals, interfacing with memory, peripherals and outer world, timing, and so on. The hardware supports the programmer's model.

In the programmer's model, we focus on the instruction set and syntaxis, addressing modes, the memory map, transfers and execution time, and so on. Very often, when a microcontroller is designed from scratch, the process starts with the desired instruction set.

This chapter focuses first on a general system view of the hardware architecture, and then the software characteristics.

3.3 Central Processing Unit

The Central Processing Unit (CPU) in a microcomputer system is typically a microprocessor unit (MPU) or core. The CPU is where instructions become signals and hardware actions that command the microcomputer operation. The minimal list of components that define the architecture of a CPU include the following:

- Hardware Components:
  - An Arithmetic Logic Unit (ALU)
  - A Control Unit (CU)
  - A Set of Registers
  - Bus Interface Logic (BIL)
- Software Components:
  - Instruction Set
  - Addressing Modes

The instructions and addressing modes will be defined by the specifics of the hardware ALU and CU units. In this section we concentrate on the hardware components. Instructions and addressing modes are considered in Sect. 3.7.

Figure 3.3 illustrates a simplified model view of the CPU with its internal hardware components. These components allow the CPU to access programs and data stored somewhere in memory or input/output subsystem, and to operate as a stored program computer. The sequence of instructions that make a program are chosen from the processor's instruction set. A memory stored program dictates the sequence of operations to be performed by the system. In the processing of data, each CPU component plays a necessary role that complements those of the others.

The collection of hardware components within the CPU performing data operations is called the processor's *datapath*. The CPU datapath includes the ALU, the internal data bus, and other functional components such as floating-point units, hardware multipliers, and so on. The hardware components performing system control operations are designated *Control Path*. The control unit is at the heart of the CPU control path. The bus control unit and all timing and synchronization hardware components are also considered part of the control path.

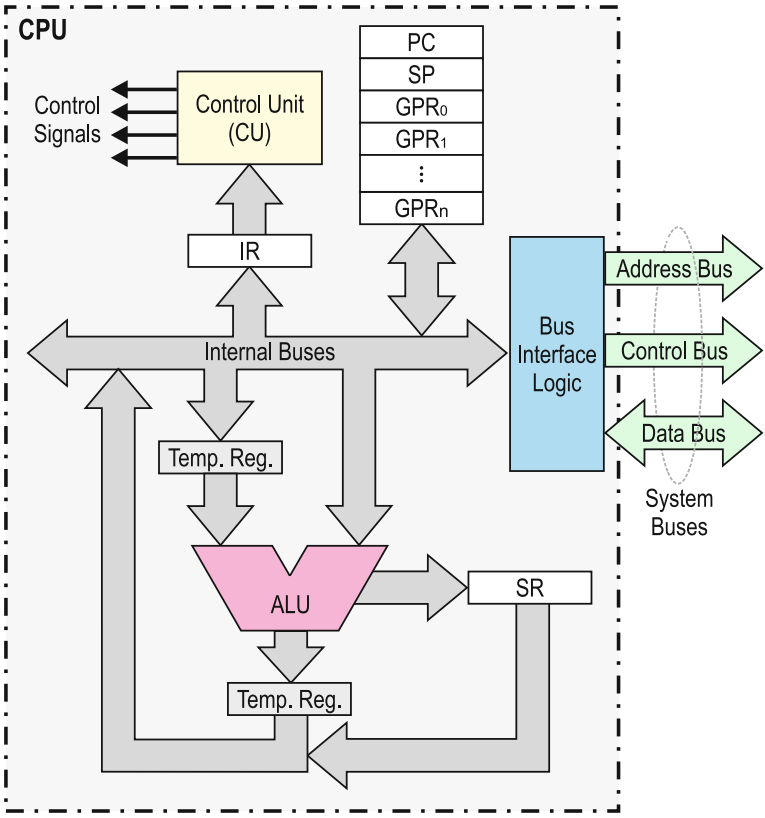

Fig. 3.3 Minimal architectural components in a simple CPU

3.3.1 Control Unit

The control unit (CU) governs the CPU operation working like a finite state machine that cycles forever through three states: fetch, decode, and execute, as illustrated in Fig. 3.4. This fetch-decode-execute cycle is also known as *instruction cycle* or *CPU cycle*. The complete cycle will generally take several clock cycles, depending on the instruction and operands. It is usually assumed as a rule of the thumb that it takes at least four clock cycles.[3] Since the instruction may contain several words, or may require several intermediate steps, the actual termination of the execution process may require more than one instruction cycle.

[3] Often, CPU have internal clocks that run faster than system clocks, at least four times. Thus, in literature we may see that an instruction takes only one clock cycle. This is true with respect to the system cycle, but the process was driven by the internal CPU clock which actually took several cycles.

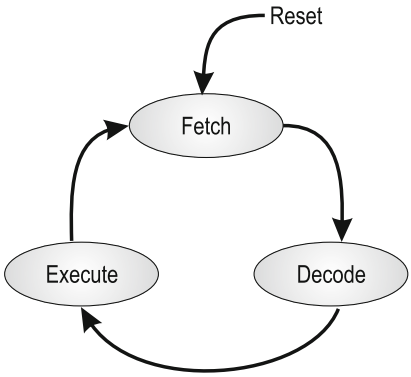

Fig. 3.4 States in control unit operation: fetch, decode, and execute

Several CPU blocks participate in the fetch-decode-execute process, among which we find special purpose registers PC and IR (see Fig. 3.3). The cycle can be described as follows:

1. **Fetch State**: During the fetch state a new instruction is brought from memory into the CPU through the bus interface logic (BIL). The *program counter* (PC) provides the address of the instruction to be fetched from memory. The newly fetched instruction is read along the data bus and then stored in the *instruction register* (IR).
2. **Decoding State**: After fetching the instruction, the CU goes into a decoding state, where the instruction meaning is deciphered. The decoded information is used to send signals to the appropriate CPU components to execute the actions specified by the instruction.
3. **Execution State**: In the execution state, the CU commands the corresponding CPU functional units to perform the actions specified by the instruction. At the end of the execution phase, the PC has been incremented to point to the address of the next instruction in memory.

After the execution phase, the CU commands the BIL to use the information in the program counter to fetch the next instruction from memory, initiating the cycle again.

The cycle may require intermediate cycles similar to this one whenever the decoding phase requires reading (fetching) other values from memory. This is dictated by the addressing mode use in the instruction, as it will be explained later.

Being the CU a finite state machine, it needs a *Reset* signal to initiate the cycle for the first time. The program counter is hardwired to, upon reset, load the memory address of the first instruction to be fetched. That is how the first cycle begins operation. The address of this first instruction is called *reset vector*.

3.3.2 Arithmetic Logic Unit

The arithmetic logic unit (ALU) is the CPU component where all logic and arithmetic operations supported by the system are performed. Basic arithmetic operations

such as addition, subtraction, and complement, are supported by most ALUs. Some may also include hardware for more complex operations such as multiplication and division, although in many cases these operations are supported via software or by other peripherals, such as a hardware multiplier.

Logic operations performed in the ALU may include bitwise logic operations AND, OR, NOT, and XOR, as well as register operations like SHIFT and ROTATE. Bitwise operations are an important tool for manipulating individual bits of a register without affecting the other ones. This is possible by exploiting the properties of logic Boolean operations.

The CU governs the ALU by specifying which particular operation is to be performed, the source operands, and the destination of the result. The width of the operands accepted by the ALU of a particular CPU (datapath width) is typically used as an indicator of the CPU computational capacity. When for example, a microprocessor is referred to as a 16-bit unit, its ALU has the capability of operating on 16-bit data. The ALU data width shapes the CPU datapath architecture establishing the width of data bus and data registers.

3.3.3 Bus Interface Logic

The Bus Interface Logic (BIL) refers to the CPU structures that coordinate the interaction between the internal buses and the system buses. The BIL defines how the external address, data, and control buses operate. In small embedded systems the BIL is totally contained within the CPU and transparent to the designer. In distributed and high performance systems the BIL may include dedicated peripherals devoted to establish the CPU interface to the system bus. Examples of such extensions are bus control peripherals, bridges, and bus arbitration hardware included in the chip set of contemporary microprocessor systems.

3.3.4 Registers

CPU registers provide temporary storage for data, memory addresses, and control information in a way that can be quickly accessed. They are the fastest form of information storage in a computer system, while at the same time they are the smallest in capacity. Register contents is *volatile*, meaning that it is lost when the CPU is de-energized. CPU registers can be broadly classified as general purpose and specialized.

General purpose registers (GPR) are those not tied to specific processor functions and may be used to hold data, variables, or address pointers as needed. Based on this usage, some authors classify them also as *data* or *address registers*. Depending on the processor architecture, a CPU can contain from as few as two to several dozen GPRs.

Special purpose registers perform specific functions that give functionality to the CPU. The most basic CPU structure includes the following four specialized registers:

- Instruction Register (IR)
- Program Counter (PC), also called Instruction Pointer (IP)
- Stack Pointer (SP)
- Status Register (SR)

Instruction Register (IR)

This register holds the instruction that is being currently decoded and executed in the CPU. The action of transferring an instruction from memory into the IR is called *instruction fetch*. In many small embedded systems, the IR holds one instruction at a time. CPUs used in distributed and high-performance systems usually have multiple instruction registers arranged in a queue, allowing for concurrently *issuing*[4] instructions to multiple functional units. In these architectures the IR is commonly called an *instruction queue*.

Program Counter (PC)

This register holds the address of the instruction to be fetched from memory by the CPU. It is sometimes also called the *instruction pointer (IP)*. Every time an instruction is fetched and decoded, the control unit increments the value of the PC to point to the next instruction in memory. This behavior may be altered, among others, by *jump instructions* which, when executed, replace the contents of the PC with a new address. Being the PC an address register, its width may also determine the size of the largest program memory space directly addressable by the CPU.

The PC is usually not meant to be directly manipulated by programs. This rule is enforced in many traditional architectures by making the PC not accessible as an operand to general instructions. Newer RISC architectures have relaxed this rule in an attempt to make programming more flexible. Nevertheless, this flexibility has to be used with caution to maintain a correct program flow.

Stack Pointer (SP)

The *stack* is a specialized memory segment used for temporarily storing data items in a particular sequence. The operations of storing and retrieving the items according to this sequence is managed by the CPU with the stack pointer register (SP). Few MCU's models have the stack hardwired defined. Most models, however, allow the

[4] Instruction issue is a term frequently used in high-performance computer architectures to denote the transfer of decoded instruction information to a functional unit like an ALU, for execution.

user to define the stack within the RAM section, or else it is automatically defined during the compiling process.

The SP contents is referred to as the Top of Stack (TOS). This one tells the CPU where a new data is stored (*push operation*) or read (*pull operation* or *pop operation*). A detailed explanation of how SP works with these operations is given in Sect. 3.7.4.

Status Register (SR)

The status register, also called the *Processor Status Word (PSW)*, or *Flag Register* contains a set of indicator bits called *flags*, as well as other bits pertaining to or controlling the CPU status. A flag is a single bit that indicates the occurrence of a particular condition.

The number of flags and conditions indicated by a status register depends on the MCU model. Most flags in the SR reflect the situation just after an operation is executed by the ALU, although in general they can also be manipulated by software. This dependence is emphasized in Fig. 3.3 by the position of the SR.

The status of the flags depends also on the size of the ALU operands. Generally, both operands will have the same size, n bits, while the ALU operation may produce $n + 1$ bits. By "ALU result" we mean then the n least significant bits while the most significant bit is the Carry Flag. This remark is clarified Example 3.1 below, but let us first identify the most common flags to be found in almost all MCUs. These are

- *Zero Flag (ZF)*: Also called the zero bit. It is set when the result of an ALU operation is zero, and cleared otherwise. It may also be tied to other instructions.
- *Carry Flag (CF)*: This flag is set when an arithmetic ALU operation produces a carry. Some non arithmetic operations may affect the carry without having a direct relation to it.
- *Negative* or *sign flag* (*NF*): This flag is set if the result of an ALU operation is negative and cleared otherwise. This flag in fact reflects the most significant bit of the result.
- *Overflow Flag (VF)*: This flag signals overflow in addition or subtraction with signed numbers (see Sect. 2.8.1). The flag is set if the operation produces an overflow, and cleared otherwise.[5]
- *Interrupt Flag (IF)*: This flag, also called *General Interrupt Enable* (GIE), is not associated to the ALU. It indicates whether a program can be interrupted by an external event (interrupt) or not. Interrupts blocked by the IF are called *maskable*. Section 3.9 discusses the subject of interrupts in microprocessor based systems.

MCUs may have more flags than the one mentioned. The user must consult the specifications of the microcontroller or microprocessor being used to check available flags and other bits included in the SR. The following example shows how the ALU affects flags after an addition.

[5] There are other types of overflow, like when the result of an arithmetic operation exceeds the number of bits allocated for its result. This condition however is not signaled by this flag, although particular CPUs may have another flag for this purpose.

Example 3.1 *The following operations are additions performed by the ALU using 8-bit data. For each one, determine the Carry, Zero, Negative, and Overflow flags.*

01001010 +	10110100 +	10011010 +	11001010 +
01111001 =	01001100 =	10111001 =	00011011 =
0 11000011	1 00000000	1 01010011	0 11100101
↑ ↑	↑ ↑	↑ ↑	↑ ↑
C N	C N	C N	C N

Solution: *The operands have eight bits, so this length is our reference for the flags when we look at the result. The most significant bit in this group is flag* N. *The bit to the left is* C. *In hex form, these additions are, respectively,* 4Ah + 79h = C3h*;* B4h + 4Ch = 100h*;* 9Ah + B9h = 153h*; and* CAh + 1Bh = E5h. *The zero flag is set if the result is 0, discarding the carry, and the overflow flag is set if the addition of numbers of the same sign (that is, with equal most significant bit) yield a result of different sign (signaled by N). With this information we have then:*

Operation 4Ah + 79h = C3h : C = 0, N = 1, Z = 0 *and* V = 1.
Operation B4h + 4Ch = 100h : C = 1, N = 0, Z = 1 *and* V = 0.
Operation 9Ah + B9h = 153h : C = 1, N = 0, Z = 0 *and* V = 1.
Operation CAh + 1Bh = E5h : C = 0, N = 1, Z = 0 *and* V = 0.

Notice that there is no overflow whenever both operands in an addition are of different sign[6]; that is, when their most significant bits are different. When both operands are negative and yield a positive result, as in the third case of the above example, we say that an *underflow* has happened. The term overflow is generic, though, and can be used in any situation.

Although the interpretation of what a flag means depends ultimately on the programmer, the meaning of the N, Z, and V flags is, we think, clear. The N and V flags are mainly related to the use of signed numbers in arithmetic operations of subtraction and addition. The Z flag tells us if the result is 0 or not, no matter the operation, arithmetic or logic or any of other type. The C flag needs however further remarks.

The Carry Flag

The Carry flag may be of interest in several operations besides addition and subtraction. Let us first discuss these two operations.

When an addition is performed, the C flag may or may not be considered part of the result. In the first case, it also provides a sign extension when working with signed numbers, as it can be appreciated in the previous example. Or it can also be used to work with numbers larger than the ALU capacity. When it is not part of the addition result, then its meaning will depend on the programmer's intention.

---

[6] One can visualize this with the full interval in a numerical line. Starting at the midpoint, two consecutive walks in opposite directions, neither greater than half the interval length, will keep you inside the interval.

Table 3.2 Flags and number comparison with $A - B$

Comparison	Unsigned Numbers	Signed numbers
$A = B$	Z = 1	Z = 1
$A \neq B$	Z = 0	Z = 0
$A \geq B$	C = 1[1]	N = V
$A > B$	C = 1 and Z = 0	N = V and Z = 0
$A < B$	C = 0	N ≠ V
$A \leq B$	C = 0 or Z = 1	N ≠V or Z = 1

Remember: C = 1 no borrow needed, C = 0 borrow is needed

On the other hand, in subtraction we are more interested in knowing if there has been a borrow or not. Some CPU's may have another flag to work with borrow. But it is also normal to use the Carry flag in a dual role to signal a carry in addition operations and a borrow in subtraction operations. Since subtraction is usually hardware realized with addition of two's complement, two standpoints have been adopted:

(1) The C is set (C = 1) after a subtraction if no borrow is generated. This mechanism is adopted in some families like PIC, Atmel, and INTEL microcontrollers.
(2) C is reset (C = 0) if the subtraction needs a borrow. This is usual in RISC microcontrollers, including the MSP430, since it needs less hardware realization.

We adopt hereafter the second option, following our choice of MCU for practical introduction, the MSP430.

The C flag is associated to other instructions, depending on the CPU. A common application is to use it as a mid step for bits in shifting and rotation operations, connecting different registers. Other instructions depend on the microcontroller selected by the user and should be consulted in the proper documentation. For the MSP430, these instructions will be introduced as we study them.

SR Flags and Number Comparison

A very useful characteristic of the flags is the information they provide when comparing numbers. In particular, this feature is used by the CPU to decide actions to take in conditional jumps, thereby making it possible to write non sequential programs.

When comparing two numbers by subtraction, A − B, the status of the flags allows us—and the CPU—to decide the relationship between the two numbers, as illustrated in Table 3.2. This table assumes that subtraction is carried out by two's complement addition, which is the case in the majority of CPU's.

Let us explain briefly why the above interpretations. Remember that all subtractions are made with binary words, whose interpretation as signed or unsigned numbers pertains to the user, not to the machine.

The first two lines in the table are obvious, since A–B = 0 if and only if A = B. For the inequalities, let us consider the unsigned and signed cases separately.

For unsigned cases, the sign and overflow flags are meaningless. However, we know that the presence of a carry in a subtraction done with two's complement addition means that the result is not negative (See remark on Sect. 2.5.3). That is, $A-B \geq 0$ if and only if there is a carry, (C = 1). Strict inequality requires the result to be non-zero, i.e., Z = 0. This proves the third and fourth lines, respectively. The other two lines are just the opposite of the previous cases—$A < B$ means not($A \geq B$) and $A \leq B$ means not($A > B$) –.

Now, consider the case for signed numbers. The sign flag N, which reflects the most significant bit of the result, has sense only when subtraction is correctly performed, meaning by this that V = 0. In this case: (a) N = 0 means that the difference A–B is not negative, so $A \geq B$ iff N = V = 0, and (b) N = 1 means that the difference is negative, so $A < B$ iff N = 1 $\neq$ V = 0, where iff means *if and only if*.

On the other hand, overflow occurs in the subtraction when A and B have different sign, but A–B has the same sign as B, triggering V = 1. In this case consider the following: (a) If $B < 0$, we can say that $A > 0 > B$, and V = 1 = N = 1. (b) if $B > 0$ then $A < B$ and N = 0 $\neq$ V = 1.

The two paragraphs above prove then the third and fifth lines for signed numbers. Strict > inequality requires Z = 0 because the subtraction cannot yield 0. The last case is the opposite of this strict inequality.

Example 3.2 *Consider the following bytes:* X = 3Ch, Y = 74h, W = A2h, *and* Z = 89h.

(a) Write down the correct relationships (excluding $\neq$) substituting the question mark sign in the following expressions when the bytes represent unsigned numbers, and when they represent signed numbers. Do it intuitively.

(b) Verify that the flags provide the same information as in (a) when comparing the numbers by subtraction using two's complement addition.

$$X?Y \quad Y?W \quad W?Z \quad Z?X$$

Solution: *(a) For easier analysis, let us translate the given bytes into their decimal equivalents.*

Unsigned case: $X = 60$; $Y = 116$; $W = 162$; $Z = 137$. *Hence,*

$$X < Y\,(also\ X \leq Y); \quad Y < W\,(also\ Y \leq W);$$
$$W > Z\,(also\ W \geq Z); \quad Z > X\,(also\ Z \geq X)$$

Signed case: $X = 60$; $Y = 116$; $W = -94$; $Z = -119$. *Therefore,*

$$X < Y\,(also\ X \leq Y); \quad Y > W\,(also\ Y \geq W);$$
$$W > Z\,(also\ W \geq Z); \quad Z < X\,(also\ Z \leq X)$$

(b) To subtract, we use the two's complements of the given bytes: $X' = C4h$; $Y' = 8Ch$; $W' = 5Eh$; $Z' = 97h$ *in the following table:*

Subtraction	Operation	Unsigned numbers		Signed numbers	
		Flags	Relation	Flags	Relation
X-Y	3Ch + 8Ch = C8h	C = 0	X < Y	V = 0, N = 1	X < Y
		(Z = 0)	X ≤ Y	(Z = 0)	X ≤ Y
$Y - W$	74h + 5Eh = D2h	C = 0	Y < W	V = 1, N = 1	Y ≥ W
		(Z = 0)	Y ≤ W	& Z = 0	Y > W
$W - Z$	A2h + 77h = 119h	C = 1	W ≥ Z	V = 0, N = 0	W ≥ z
		& Z = 0	W > Z	& & Z = 0	W > Z
$Z - X$	89h + C4h = 14Dh	C = 1	Z ≥ X	V = 1, N = 0	Z < X
		& Z = 0	Z > X	& (Z = 0)	Z ≤ X

In the case of **or** *decisions, the second flag to consider was put in parenthesis. Thus* ($Z = 0$) *should be considered part of the process of verifying* $C = 0$ *or* $Z = 1$*, which in the shown cases was valid because of the* $C = 0$ *compliance.*

3.3.5 MSP430 CPU Basic Hardware Structure

The MSP430 family is based on a 16-bit CPU which was introduced in the early models of the series 3xx. Later on the architecture was extended to 20-bits, the CPUX, keeping full compatibility with the original 16-bit CPU. Thus, the instruction set of the CPU is fully supported by the CPUX, which works like a 16-bit CPU for these operations. There are special instructions proper to the CPUX targeting 20-bit data.

In this book, hands on programming is focused on the MSP430 16-bit CPU. Accordingly, this section provides a quick overview on its characteristics. An explanation for the MSP430X control processing unit CPUX is given in Appendix D.

MSP430 registers: There are sixteen 16-bit registers in the MSP430 CPU named R0, R1 …, R15. Registers R4 to R15 are of the general purpose type. The specialized purpose registers are:

- Program Counter register, named R0 or PC.
- Stack Pointer Register, named R1 or SP.
- Status Register, with a dual function also as Constant Generator. It is named R2, SR or CG1.
- Constant Generator, named CG2.

Register R3 is exclusively used as a constant generator supplying instruction constants, and is not used for data storage. Its function is normally transparent to the user. This is explained in Chap. 4.

The PC and SP registers always point to an even address and have the least significant bit hardwired to 0. A particular feature in the MSP430 family is that these two registers can be used as operands in instructions, allowing programmers to develop applications with simpler software algorithms.

Status Register The SR register has the common flags of Carry (C), Zero (Z), Sign or Negative (N), overflow (V) and general interrupt enable (GIE). It contains in addition a set of bits, CPUOFF, OSCOFF, SCG1 and SCG0, used to configure the CPU, oscillator and power mode operations. These bits are explained in later sections and chapters. The bit distribution in the SR is shown in Fig. 3.5.

Arithmetic-Logic Unit: The MSP430 CPU ALU has a 16-bit operand capacity; the CPUX has 16- or 20-bit operand capacity. It handles the arithmetic operations of addition with and without carry, decimal addition with carry (BCD addition—see Sect. 2.7.3), subtraction with and without carry. These operations affect the overflow, zero, negative, and carry flags. The logical operations AND and XOR affect the flags, but other ones do not, like the Bit Set (OR) or Bit Clear.

Like many microcontrollers, the MSP430 ALU does not handle multiplications or divisions, which may be programmed. Several models contain a *hardware multiplier* peripheral for faster operation.

3.4 System Buses

Memory and I/O devices are accessed by the CPU through the system buses. A bus is simply a group of lines that perform a similar function. Each line carries a bit of information and the group of bits may be interpreted as a whole. The system buses are grouped in three clases: *address*, *data*, and *control* buses. These are described next.

3.4.1 Data Bus

The set of lines carrying data and instructions to or from the CPU is called the *data bus*. A *read* operation occurs when information is being transferred *into* the CPU. A data bus transfer *out from* the CPU into memory or into a peripheral device, is called a *write* operation. Note that the designation of a transfer on the data bus as read or write is always made with respect to the CPU. This convention holds for every system component. Data bus lines are generally bi-directional because the same set of lines allows us to carry information to or from the CPU. One transfer of information is referred to as *data bus transaction*.

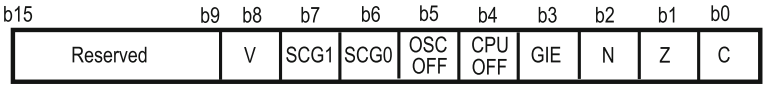

Fig. 3.5 MSP430 status register

The number of lines in the data bus determines the maximum *data width* the CPU can handle in a single transaction; wider data transfers are possible, but require multiple data bus transactions. For example, an 8-bit data bus can transfer at most one byte (or two nibbles) in a single transaction, and a 16-bit transaction would require two data bus transactions. Similarly, a 16-bit data bus would be able to transfer at most, two bytes per transaction; transferring more than 16 bits would require multiple transactions.

3.4.2 Address Bus

The CPU interacts with only one memory register or peripheral device at a time. Each register, either in memory or a peripheral device, is uniquely identified with an identifier called *address*. The set of lines transporting this address information form the *address bus*. These lines are usually unidirectional and coming out from the CPU. Addresses are usually named in hexadecimal notation.

The width of the address bus determines the size of the largest memory space that the CPU can address. An address bus of m bits will be able to address at most 2^m different memory locations, which are referred to by hex notation. For example, with a 16-bit address bus, the CPU can access up to $2^{16} = 64$ K locations named 0x0000, 0x0001, . . ., 0xFFFF. Notice that the bits of the address bus lines work as a group, called *address word*, and are not considered meaningful individually.

Example 3.3 *Determine how many different memory locations can be accessed and the address range (i.e., initial and final addresses) in hex notation with an address bus of (a)* 12 *bits, and (b)* 22 *bits. Justify your answer.*

Solution: *(a) For* 12 *bits, there are* $2^{12} = 2^2 \times 2^{10} = 4$ *K different locations that can be addressed. In binary terms, we think of them as the integer representations of* 0000 0000 0000B *to* 1111 1111 1111B*, which in hex notation become* 0x000 *to* 0xFFF. *(b) Working similarly, for* 22 *bits there are* $2^{22} = 2^2 \times 2^{20} = 4M$ *locations. Yet, since there are 22 bits and hex integers represent only four bits, the largest number with the two most significant bits is 3, so the address range is* 0x000000 *to* 0x3FFFFF.

Example 3.4 *A certain system has a memory size of 32 K memory words. What is the minimum number of lines required for the address bus?*

In powers of 2, $32\ K = 2^5 \times 2^{10} = 2^{15}$*. Therefore, the address bus must contain at least 15 lines, one per bit. (The answer could have been also expressed as* $n = \log_2(32 * 1024) = 5 + 10 = 15$).

3.4.3 Control Bus

The *control bus* groups all the lines carrying the signals that regulate the system activity. Unlike the address and data buses lines which are usually interpreted as a group (address or data), the control bus signals usually work and are interpreted separately. Control signals include those used to indicate whether the CPU is performing a read or write access, those that synchronize transfers saying when the transaction begins and ends, those requesting services to the CPU, and other tasks. Most control lines are unidirectional and enter or leave the CPU, depending on their function. The number and function of the lines in a control bus will vary depending on the CPU architecture and capabilities.

3.5 Memory Organization

The memory subsystem stores instructions and data. Memory consists of a large number of hardware components which can store one bit each. These bits are organized in n-bit words, working as a register, usually referred to as cell or location. The contents of cells is a basic unit of information called *memory word*. In addition, each memory location is identified by a unique identifier, its *memory address*, which is used by the CPU to either read or write over the memory word stored at the location. In general, a memory unit consisting of m cells of n bits each is referred to as an $m \times n$ memory; for $n = 1$ and $n = 8$ it is customary to indicate a number followed by b or B, respectively. Thus, we speak of 1Mb (one Mega Bits) and 1 MB (one Mega Bytes) memories to refer to $1M \times 1$ and $1M \times 8$ cases.

Usually, addresses are sequentially numbered as illustrated in Fig. 3.6. However, in a specific MCU model some addresses may not be present. The example in the figure shows a memory module of 64 K cells, each storing an 8-bit word (byte), forming a 64 kilo-byte (64 KB) memory. In the illustration, for example, the cell at address

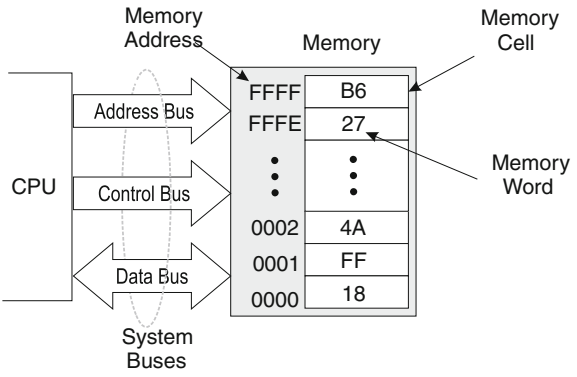

Fig. 3.6 Memory structure

0FFFEh contains the value 27 h. The CPU uses the address bus to select only the cell with which it will interact. The interaction with the contents is realized through the data bus. In a write operation, the CPU modifies the information contained in the cell, while in a read operation it retrieves this word without changing the contents. The CPU uses control bus signals to determine the type of operation to be realized, as well the direction in which the data bus will operate—remember that the data bus lines are bidirectional.

Remark on notation: Hereafter, to simplify writing, memory addresses and contents will be written with notations such as [*address*] = *contents*, or *address*: *contents*. In figures, the contents of memory is shown always in hexadecimal notation without suffix or prefix. Taking as reference Fig. 3.6 for an example, [0002h] = 0x4A.

3.5.1 Memory Types

Hardware memory is classified according to two main criteria: *storage permanence* and *write ability*. The first criterion refers to the ability of memory to hold its bits after these have been written. Write ability, on the other hand, refers on how easily the contents of memory can be written by the embedded system itself. All memory is readable, since otherwise it would be useless.

From the storage permanence point of view, the two basic subcategories are the *nonvolatile* and *volatile* groups. The first group encompasses those memories that can hold the data after power is no longer supplied. Volatile memory, on the other hand, looses its contents when power is removed.

The nonvolatile category includes the different *read only memory* (ROM) structures as well as *Ferro electric RAM* (FRAM or FeRAM). Another special one is the nonvolatile RAM (NVRAM) which is in fact a volatile memory with a battery backup; for this reason, NVRAMs are not used in microcontrollers. The volatile group includes the *static RAM*(SRAM) and the *dynamic RAM*(DRAM).

From the write ability point of view, memory is classified into *write/read* or *in-system programmable* and *read only* memories. The first group refers to those memories that can be written to by the processor in the embedded system using the memory. Most volatile memories, SRAM and DRAM, can be written to during program execution, and therefore these memory sections are useful for temporary data or data that will be generated or modified by the program. The FRAM memory is non-volatile, but the writing speed is faster than the DRAM. Hence, microcontrollers with FRAM memory are very convenient in this aspect.

Most in-system programmable nonvolatile memories are written only when loading the program, but not during execution. One reason is the writing speed, too slow for the program. Another aspect of Flash memory and EEROM is the fact that writing requires higher voltages than the ones used during the program execution, thereby consuming power and requiring special power electronics hardware to increase

Storage	Memory	In-system Writable	Comments
Nonvolatile	Masked ROM	No	Non programmable
	OTPROM	No	One time programmable with programming device
	EPROM	No	Erasable and programmable with external device
	EEPROM	Yes	Slow to erase/write. Not advisable to write during program execution. Requires higher voltage.
	Flash	Yes	Similar to EEPROM
	FRAM	Yes	Fast to write at low voltage
Volatile	Static RAM	Yes	Fastest to write/read
	DRAM	Yes	Fast to write/read

Fig. 3.7 Memory types

voltage levels. The FRAM, on the other side, is writable at the program execution voltage levels.

The FRAM, is both nonvolatile and writable with speeds comparable to DRAMS and at operational voltage levels. Moreover, unlike the Flash or DRAM, consumes power only during writing and reading operations, making it a very low power device. One disadvantage at this moment is that the temperature ranges of operation are limited, so their application must be in environments where temperature is not so variable. Further research is being done to solve this limitation. Table in Fig. 3.7 summarizes this discussion.

As a marginal note, for historical reasons, in the embedded community volatile read/write memories are referred to as RAM (random access memories)[7] while the ROM term is used for non volatile memory. This convention is hereafter adopted unless specific details are needed.

3.5.2 Data Address: Little and Big Endian Conventions

Most hardware memory words nowadays are one byte length. As explained above, each memory word has an address attached to it, referred to as its *physical address*, which is encoded by the group of the MCU address bus bits. Memory blocks one

[7] RAM, an acronym for "Random Access Memory", is a term coined in the 1950s to refer to solid state memories in which data could be accessed randomly, as opposed to other memory devices such as magnetic tapes and discs. Under this definition, semiconductor ROM devices are random access ones also, but the DRAM is not. But the terms are already stuck and understood by the embedded community.

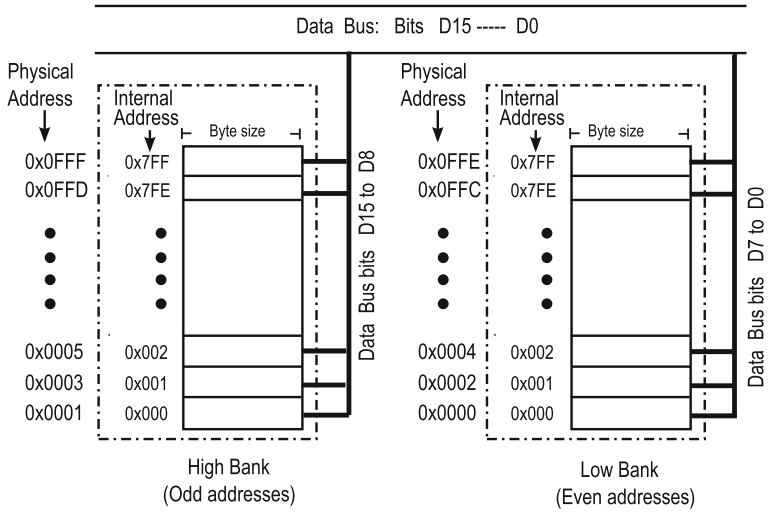

Fig. 3.8 Example of bank connection

byte length are called *banks*. For data buses wider than a byte, two or more banks are needed to connect all the data bus lines, as illustrated in Fig. 3.8 for a 16-bit data bus. In this figure, physical address refers to the address seen by the CPU with the address bus, while the internal addresses are the addresses proper to the bank. In the case of 8-bit data bus MCU's, these two sets may be identical for an example like this one.

Here, two 2 KB hardware memory banks are connected to the 16-bit data bus forming a 4 KB memory segment starting at 0x0000. A *memory segment* is a set of memory words with continuous addresses. One of the banks, called *low bank* has only even addresses attached to its memory words while the other one, the *high bank* has odd addresses. For that reason, they are also known as *even address bank* and *odd address bank*, respectively. The data bus bits D15, D14, …, D1, D0 are divided in two byte groups, which are connected to the banks.

On the other hand, instructions and data may be longer than a byte, which means that they cannot be stored in one memory cell, but need to be broken into one byte pieces, which are stored in consecutive memory locations. In this case, use the terms *instruction address* and *data address* using the lowest of the set of physical addresses containing the bytes, with the understanding that this number encompasses all the physical addresses of the set. For example, if the data is 23AFh with address F804h, than the physical addresses covered in this case are F804h and F805h, one per byte. Similarly, if the instruction of three words "40Bf 003A 0120" has the address F8AAh, this means that the physical addresses covered are F8AAh to F8AFh.

In particular, if there is a reference to the size of the data, the terms become *word address* for 16-bit long data, *double word address* for 32-bit long data, and so on. Thus, a word address includes two physical addresses and a double word address covers four physical addresses. The physical addresses of the individual

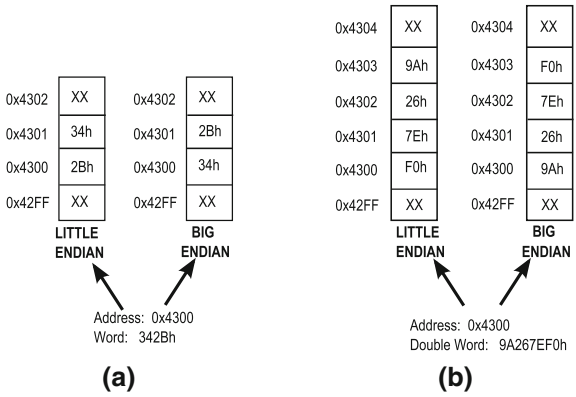

Fig. 3.9 Big and Little Endian conventions for (**a**) a word ([4300 h = 3428 h]) and (**b**) a double-word ([4300 h] = [9A267EF0h])

bytes comprising the word or double word depend on the convention used to store them: *big endian* or *little endian*.

Big Endian: In the big endian convention, data is stored with the most significant byte in the lowest address and the least significant byte in the highest address.

Little Endian: In the little endian convention, data is stored with the least significant byte in the lowest address and the most significant byte in the highest address.

Second Remark on Notation: Hereafter, all data addresses and contents will also be written using the same notation as for memory contents introduced on (3.5), that is, [*address*] = *contents*, or *address*: *contents*. For example, 3840:23A2 is for the address 3840h of the word 23A2h. The same information is written as [3840h] = 23A2h. The addresses for individual bytes will follow depending on the endian convention used by the specific microcontroller.

Figure 3.9 illustrates the endian conventions for word (16-bit word) and double word (32-bit word) cases. The XX in this figure are "don't cares", irrelevant for the example.

It is convenient to have word-sized and double word-sized data at even addresses when the data bus is 16-bits wide. Similarly, if the data bus has 32 bits, then we should have double word-sized data at addresses multiples of 4. The reason may be understood taking a closer look at Fig. 3.8. The physical addresses of both banks differ only by bit A0 of the address bus: it is 0 for all bytes in the low bank, and 1 for those in the high bank. Since these are 2K cells, we can use A11 A10 A9 …A2 A1 to access the internal cells in the banks. Thus, two consecutive physical addresses with the smaller one being even will catch in both banks cells with identical internal addresses. For example, the physical addresses pair 0x0301 − 0x0300 (0000 0011 0000 0001 and 0000 0011 0000 0000 in binary) will access the same internal address in both banks: 001 1000 0000, or 180h. Hence, when the word address is even, a single data bus transaction is needed for any transfer, since the bytes in two contiguous physical addresses corresponding to the two data bytes are accessed simultaneously. This is not possible if the word address is odd.

Associated with the data address, the term *word boundary*, refers to an even address, and *double word boundaries* which refers to addresses which are multiple of 4; in hex notation, the latter means any number ending in 0, 4, 8, C.

By the way, the connection of banks illustrated in Fig. 3.8 corresponds to a little endian convention; for a big endian case, the high bank has the even addresses.

Example 3.5 *The debugger of a certain microcontroller presents memory information in chunks of words in the list form shown below, where the first column is the address of the word in the second column. Following the debuggers' conventions, all numbers are in hex system. If more than one word is on the line, the address is for the first word only. Assuming that all data is effectively 16-bit wide, break the information into bytes with the respective address, (a) assuming little endian convention, and (b) assuming big endian convention.*

F81E: E0F2 0041 0021
F824: 403F 5000
F828: 831F

Solution: *According to instructions, F81E is the address of word E0F2, F820 that of the second word 0041 and so on. With this in mind, we can break the memory information in byte chunks as follows:*

Little endian		Big endian	
F81E:F2	F824: 3F	F81E: E0	F824: 40
F81F: E0	F825: 40	F81F: F2	F825: 3F
F820: 41	F826: 00	F820: 00	F826: 50
F821: 00	F827: 50	F821: 41	F827: 00
F822: 21	F828: 1F	F822: 00	F828: 83
F823: 00	F829: 83	F823: 21	F829: 1F

3.5.3 Program and Data Memory

The previous section focused on address aspects of memory. With respect to the contents, the structure of a microcomputer includes two differentiable types of memory depending on the kind of information they store: *Program Memory* and *Data Memory*.

Program memory, as inferred by its name, refers to the portion of memory that stores the system programs in a form directly accessible by the CPU. A *program* is a logical sequence of instructions that describe the functionality of a computer system.

In embedded systems, programs are fixed and must always be accessible to the CPU to allow the system to operate correctly. Thus, when power is removed from the system and later restored, programs must still be there to allow the system to function properly. This implies that program is usually stored in nonvolatile memory. Program memory capacity in typical embedded systems is in the order of kilo-words. Some microcontroller models may run a program from the RAM portion while others enforce using only ROM sections.

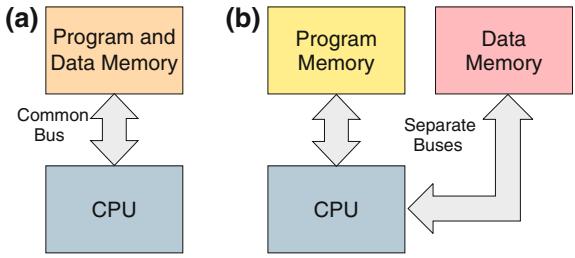

Fig. 3.10 Topological difference between (**a**) von Neumann and (**b**) Harvard architectures

Data memory is used for storing variables and data expected to change during program execution. Therefore, this type of memory should allow for easily modifying its contents. Most embedded systems implement data memory in RAM (Random-Access Memory). Since data memory is meant to hold temporary information, like operation results or measurements for making decisions, the volatility of RAM is not an inconvenience for system functionality. The average amount of RAM needed in most embedded applications is relatively small. For this reason it is quite common to find embedded systems with data memory capacity measured in only a few hundreds words.

Particularly important data which cannot be lost when the system is de-energized, must be stored in the ROM section. Some microcontroller families may access these data directly from there, although they cannot change data during program execution. Other models always copy the data into the RAM section and enforce data manipulation only from these volatile segments.

3.5.4 Von Neumann and Harvard Architectures

Program and data memories may share the same system buses or not, depending on the MCU architecture. Systems with a single set of buses for accessing both programs and data are said to have a *Von Neumann* architecture or *Princeton* architecture.[8]

An alternate organization is offered by the *Harvard Architecture*. This topology has physically separate address spaces for programs and data, and therefore uses separate buses for accessing each. Data and address buses may be of different width for both subsystems.

Figure 3.10 graphically depicts the topological differences between a Von Neumann and a Harvard architecture.

Numerous arguments can be brought in favor of either of these architectural styles. Both are present in embedded systems. Texas Instruments MSP430 series uses a Von Neumann architecture while Microchip PIC and Intel's 8051 utilize Harvard archic-

[8] In the early days of computing, much of the work that defined the von Neumann architecture was developed by Princeton professor John von Neumann, after whom its named. Due to this affiliation, the von Neumann architecture is also called Princeton architecture.

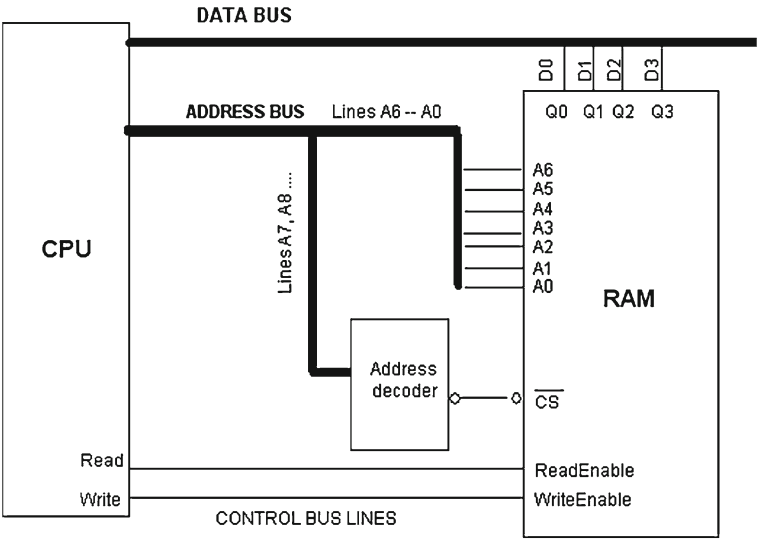

Fig. 3.11 CPU to memory connection: a conceptual scheme

tectures. Without any preference in this respect, for most aspects of our discussion we will assume a Von Neumann architecture.

The IR and PC register widths will depend on the architecture. In Von Neumman models, they have the same widths as the other CPU registers that may hold addresses, while in Harvard architecture models they are independent of the other registers, since the buses do not have to be of the same size.

3.5.5 Memory and CPU Data Exchange: An Example

To illustrate the process by which the CPU and memory interact, let us utilize a simplified example of a static RAM memory interface. Figure 3.11 shows a conceptual connection scheme between the CPU and this RAM.

The RAM terminals can be divided in three groups: (a) the data Input/Output terminals Q0, Q1, …; (b) The (internal) address terminals A0, A1, …used to select a specific word cell inside the block; and (c) the select (CS) and control terminals (ReadEnable and WriteEnable) used to operate with the memory. The CS terminal is used to activate the block, i.e. make it accessible, while the other two determine the type of operation to be performed.

The Data Bus lines are connected to the data Input/Output terminals. Internal to the block, these terminals are connected via three state buffers. These buffers allow to set the direction of data flow (read or write) and also set high impedance to disconnect from the Data Bus when the RAM is not activated. The read and write transactions

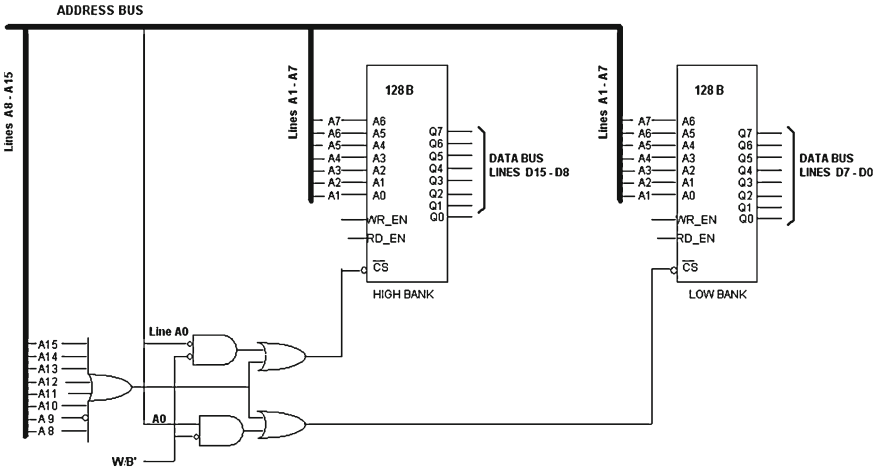

Fig. 3.12 256B memory bank example

are controlled by the CPU with signals that go through the control bus. The figure is by no means exhaustive and other control lines may intervene.

The *n* address bus lines leaving the CPU are separated in two groups: a set of lines that are directly connected to the memory block internal address lines, and another set to an decoder used to activate the memory block. In the example figure, lines A1 to A7 connect directly to the decoder selectors in the RAM while the rest of the address bus lines go to a decoder that will activate the RAM block. The address of a memory location is then the word formed with the address bus bits that activate the RAM and those selecting the word line for the internal location.

Example 3.6 *Figure 3.12 illustrates an example of the connection illustrated by Fig. 3.8. It shows two 128B memory modules connected to a 16-bit data bus and driven by a 16-bit address bus and a control signal* ***W/B'****. Since each block has only eight I/O terminals, two of them are required to cover the data bus lines. The control signal* ***W/B'*** *indicates if the CPU is reading or writing a word (W/B' = 1) or a byte (W/B' = 0). Determine the range of addresses for this memory bank, and what addresses are for each block.*

Solution: *The blocks are activated when a low signal appears at the* $\overline{CS}$ *terminal, which are connected to OR outputs of the encoder subsystem. If the output of the eight input OR is high, the system is disconnected from the data bus. Hence, we need* $A15 = A14 = A13 = A12 = A11 = A10 = 0$, $A9 = 1$, *and* $A8 = 0$ *to ensure that the blocks can be activated. The address lines A7 to A1 activate the internal address lines of the activated RAM block.*

Now, if signal W/B' = 1 then both blocks are enabled, irrespectively of A0's value. If W/B' = 0 then A0 = 0 activates the low bank RAM and A0 = 1 the high bank RAM. Therefore, the range of addresses covered by the memory bank is as follows:

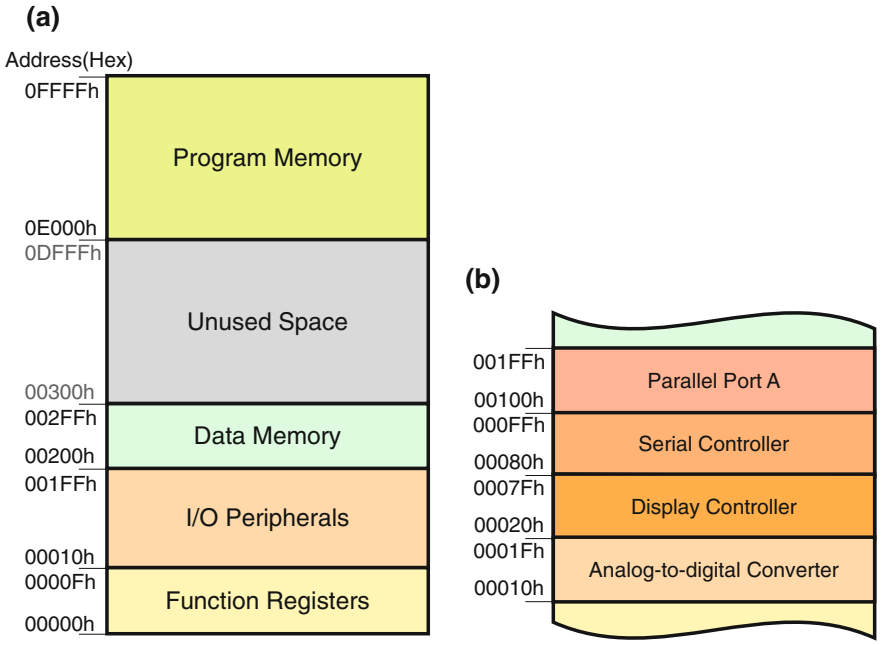

Fig. 3.13 Example memory map for a microcomputer with a 16-bit address bus. **a** Global memory gap, **b** Partial memory map

Range: From 0000 0010 0000 0000B to 0000 0010 1111 1111B, i.e., 0200h to 02FFh.

Low bank: Activated when A0 = 0, meaning even addresses in the range, that is: 0200h, 0202h, 0204h …, 02FEh.

High bank: Activated when A0 = 1, meaning odd addresses in range: 0201h, 0203, …, 02FFh.

3.5.6 Memory Map

A *memory map* is a model representation of the usage given to the addressable space of a microprocessor based system. It is an important tool for program planning and for selecting the convenient microcontroller for our application. As implied by its name, the memory map of a microcomputer provides the location in memory of important system addresses. Figure 3.13a illustrates a pictorial representation of the memory map of an example computer system with a 16-bit address bus, with an addressable memory space of 64 K words for a Von Neumann model. In the map, memory is organized as a single flat array.

In this particular case, I/O device addresses and function registers are also mapped as part of the same memory array (see Sect. 3.6). This example is just one of the possible ways of arranging the address space of a microcomputer. In this particular example, the first 16 memory words (addresses 0h through 0Fh) are allocated for function registers. The next 496 locations are assigned to input-output peripheral devices, and only 256 words are reserved for data memory (addresses from 0200h through 02FFh). Program memory is located at the end of the addressable space with 8 K words in addresses from 0E000h through 0FFFFh. Addresses from 00300h through 0DFFFh correspond to unused memory space, that could be used for system expansion.

Memory maps can be global or partial. A *global memory map* depicts the entire addressable space, as illustrated in Fig. 3.13a. A *partial map* provides detail of only a portion of the addressable space, allowing for more insight in that portion of the global map. Figure 3.13b illustrates a partial map of a possible distribution of the I/O peripheral addresses.

3.5.7 MSP430 Memory Organization

The MSP430 has a Von-Neumann architecture with a little endian data address organization. Since it also works with an I/O mapped architecture, a concept discussed later in Sect. 3.6, the memory address space is shared with special function registers (SFRs), peripherals and ports. The address bus width depends on the microcontroller model. All models are based on the original 16-bit address bus with an address space of 64 K bytes, called simply the MSP430 architecture. The extended MSP430X architecture has a 20-bit address bus with an address space of 1 M byte. However, the first 64 K addresses in this case have the same map to provide complete compatibility. Figure 3.14 shows the basic global memory map for both models. Models with a relatively large RAM capacity, like those of the family 5xx may be somewhat different. See the device-specific data sheets for specific global and partial memory maps.

The amount of RAM and Flash or ROM depends on the model. RAM memory, which may start with only 128 bytes of capacity, usually starts at address 0200h, and ends depending on the model. Similarly, in the 16-bit model, the Flash/ROM memory ends at address 0FFFFh but the start depends on the capacity. Later models may differ because of capacity needs, but are nevertheless compatible. That is, although RAM may not start at 0200h, this address is still contained in the RAM section so programs written for other models may run.

The interrupt vector table, which is explained later, is mapped into the upper 16 words of Flash/ROM address space, with the highest priority interrupt vector at the highest Flash/ROM word address (0FFFEh). Some models have the possibility of other interrupts besides the basic sixteen original ones, and provide larger tables. See specific data sheets for more information. In addition, some models contain specialized modules such as Direct Memory Access (DMA), flash controllers, or RAM controllers, to optimize power and access time.

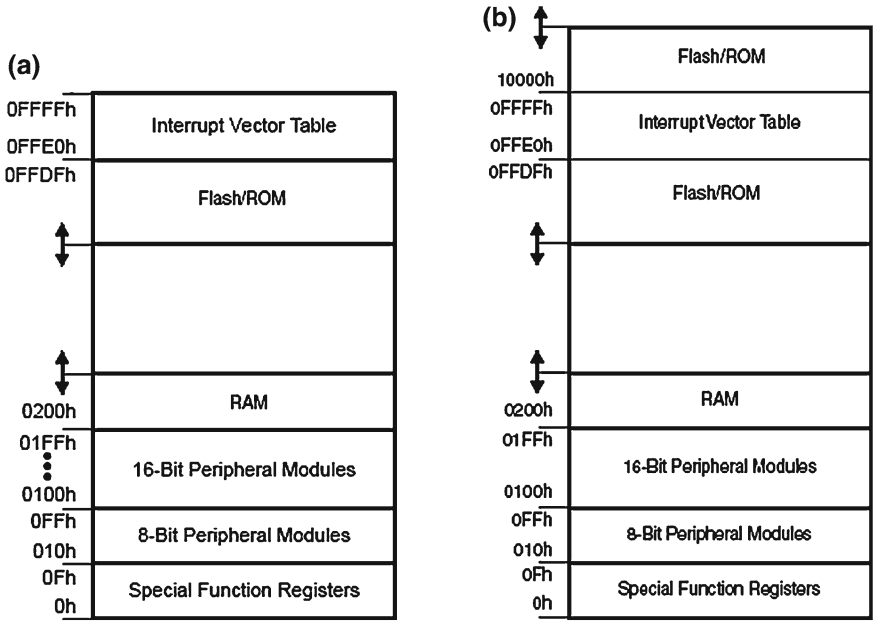

Fig. 3.14 MSP430 global memory maps (*Courtesy of Texas Instruments, Inc*). **a** 16-bit address bus memory map, **b** 20-bit address bus memory map

Data and Program Memory Basically, we should expect to have the program in the flash/ROM section and data in the RAM section. This is the way in which assemblers organize memory unless otherwise instructed. The MSP430 has however an important feature. Since only one 16-bit memory data bus is used for any memory access, programs can be run from the RAM section and data accessed from the Flash/ROM section. This unique feature gives the MSP430 an advantage over some microcontrollers, because the data tables do not have to be copied to RAM for usage. Instruction fetches from program memory are always 16-bit accesses, whereas data memory can be accessed using word (16-bit) or byte (8-bit) instructions.

Important Remark: Attempts to access non existent addresses (vacant space) result in errors that reset the system.

3.6 I/O Subsystem Organization

The I/O subsystem is composed by all the devices (peripherals) connected to the system buses, other than memory and CPU. The I/O designation is collectively given to devices that serve as either input, output, or both in a microprocessor-based system, and also includes special registers or devices used to manage the system operation

without external signals. Like in the read/write convention, the CPU is used as the reference to designate a device as either input or output. An *input transaction* moves information into the CPU from a peripheral device making it an input peripheral. *Output transactions* send information out from the CPU to external devices, making these output devices. Examples of input devices include switches and keyboards, bar-code readers, position encoders, and analog-to-digital converters. Output devices include LEDs, displays, buzzers, motor interfaces, and digital-to-analog converters. Some devices can perform both input and output transactions. Representative examples include communication interfaces such as serial interfaces, bi-directional parallel port adapters, and mass storage devices such as flash memory cards and hard disk interfaces.

Without giving an exhaustive list, common peripherals to be found in most embedded microcomputer systems include those listed below. Detailed discussion of these and other peripherals is found in Chaps. 6 and 7.

Timers These peripherals can be programmed for any use prescribing time intervals. For example, to measure time intervals between two events, generate events at specified time intervals, or generate signals at a specified frequency, as it is the case of *pulse width modulators*, and many others.

Watchdog Timer (WDT) This is a special type of timer, used as a safety device. It resets if it does not receive a signal generated by the program every X time units, a feature useful in several applications. It may also be configured to generate interrupt signals by itself at regular intervals of time.

Communication interfaces Used to exchange information with another IC or system. They use different protocols such as serial peripheral interface (SPI), universal serial bus (USB), and many others.

Analog-to-Digital Converter (ADC) Very common since many input variables from the real world vary continuously, that is, they are analog.

Digital-to-Analog Converter (DAC) It performs an opposite function to the ADC, delivering analog output signals.

Development peripherals These are used during development to download the program into the MCU and for debugging. They include the monitor, background debugger, and an embedded emulator.

The I/O subsystem organization resembles the organization of memory. Each I/O device requires an I/O interface to communicate with the CPU, as illustrated in Fig. 3.15. The interface serves as a bridge between the I/O device and the system buses. Each I/O interface contains one or several registers that allow for exchanging data, control, and status information between the CPU and the device. As a consequence, as far as it concerns the CPU, the I/O device "looks" very much alike hardware memory, with I/O interface registers connected to data bus and selected with the address bus. In the case illustrated in Fig. 3.15, the interface contains eight registers, accessible at addresses 0100h through 0107h.

An I/O interface, like in the case of memory system, may also contain address decoders to decode the addresses assigned to each particular device, buffers, latches, and drivers, depending on the application.

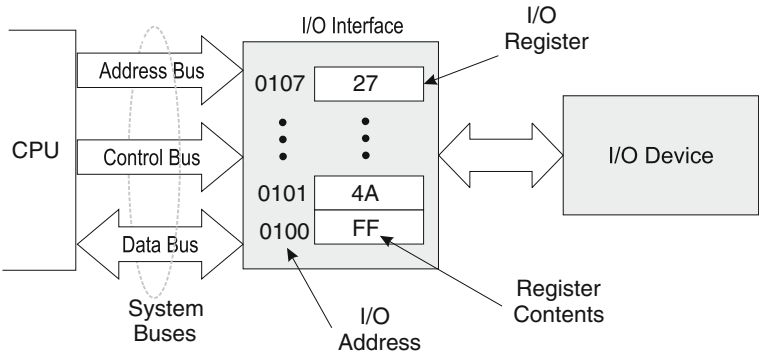

Fig. 3.15 Structure of an input/output (I/O) interface

Microprocessors like the Intel 80x86 family and Zilog Z80 have separate address spaces for memory and I/O devices, requiring specific input and output instructions to access peripherals in such a space. This strategy, called *Standard I/O* or *I/O-mapped I/O*, requires special instructions and control signals to indicate when an address is intended to the memory or to the I/O system.

An alternative is to include the I/O devices inside the memory space; this scheme is named *memory-mapped I/O*. This is normal since I/O registers are seen just as other memory locations by the CPU. This has become the dominant scheme in embedded systems, making I/O and memory locations equally accessible from a programmer's point of view. The memory map illustrated in Fig. 3.13 shows an example of memory assignments to the I/O peripherals (00000h to 001FFh) with more details in the partial map (b).

Other Registers and Peripherals

Besides the I/O peripherals, the system may include other registers and devices which are not intended to connect to the external world or to be used as memory. This set includes special function registers, timers, and other devices whose function are related to the functionality of the system, power management, and so on. The CPU communicates with these registers and devices just as it does with the rest of the system, using the system buses. Figure 3.13 illustrates inclusion of special function registers in the memory map.

3.6.1 Anatomy of an I/O Interface

The most general view of an I/O interface includes lines to connect to the address, data, and control buses of the system, I/O device connection lines, and a set of internal

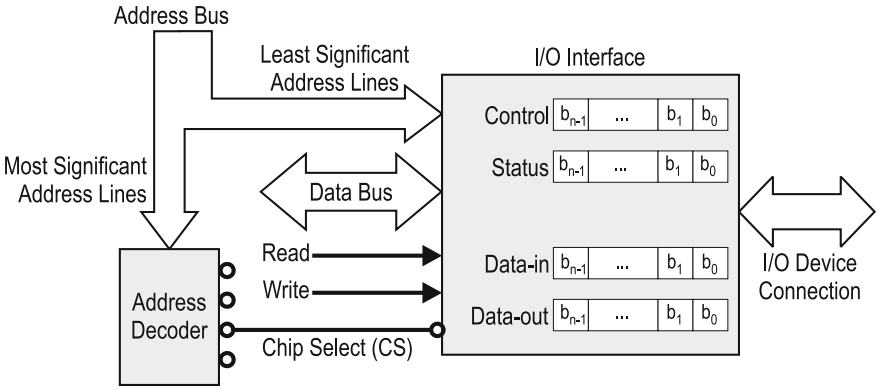

Fig. 3.16 Anatomy of an input/output (I/O) interface

registers. Figure 3.16 illustrates such an organization. A few least significant address lines are used to select the internal interface registers. The upper address lines are usually connected to an off-interface address decoder to generate select signals for multiple devices. Data lines in the designated data bus width (8- or 16-bit in most cases) are used to carry data to and from the internal registers. At least two control lines are used to indicate read (input) or write (output) operations and synchronize transfers. The device side includes dedicated lines to connect with the actual I/O device. The number and functions of such lines change with the device itself.

Internal registers in the I/O interface may be read-only, write-only, or write-and-read type from the CPU viewpoint, depending on the register type, the interface model, and the MCU model. Please refer to the appropriate user guide or data sheet. Three types of internal registers can be found in any interface:

Control Registers: Allow for configuring the operation of the device and the interface itself. One or several control registers can be provided depending on the complexity of the interface. Sometimes this type of register is called Mode or Configuration Register.

Status Register: Allow for inquiring about the device and interface status. Flags inside these registers indicate specific conditions such as device ready, error, or other condition.

Data Registers: Allow for exchanging data with the device itself. Unidirectional devices might have only one data register (Data-in for input devices or Data-out for output devices). Bi-directional I/O interfaces include both types.

Further and more detailed discussion on IO interfaces is found in Chaps. 6 and 7. Let us for the moment use the following examples to illustrate the above concepts.

Example 3.7 *Consider a hypothetical printer interface as an example of an output device. This interface contains three 8-bit internal registers: control, status, and Data-out registers. The control register allows the CPU to control how the printer operates while the status register contains information about the status of the printer.*

Table 3.3 Internal addresses for sample printer interface

A1	A0	Register
0	0	Control
0	1	Status
1	0	Data-out
1	1	Not used

The data-out register is an 8-bit write-only location, accepting the characters to be printed.

Since there are three registers, at least two address lines, A1 and A0, are needed to internally select each of them. Assume the internal addresses are assigned as shown in Table 3.3.

The address bus interface in this case will connect the two least significant address lines: A1 and A0 to the respective address signals in the printer interface. An external address decoder uses lines A15 to A2 to set the interface base address at 0100h. Thus, the addresses of the control, status, and Data-out registers will then be 0100h, 0101h, and 0102h, respectively. Address 0103h will be unused since that address is not assigned to any register and cannot be redirected because the select signal is controlled by address lines A2, A3, ... A15. Figure 3.17 shows a diagram of how the interface would connect to the CPU buses and to the printer.

Let's assume the status and control register bits have the meaning shown in Fig. 3.18.

Using address 0100h, the CPU may then write on the control register a word that tells the printer how the character will be printed (normal, bold, etc.) or if the paper should be ejected, and so on. If a character is to be printed, the CPU then writes on the Data-out register, with address 0102h, the code of the desired character. Reading the status register, on memory address 0101h, the CPU can check if the printer is ready to receive a character, or if it has problems and what type of problem, and so on.

Thus, by issuing the corresponding transfer instructions, the operation of this printer interface can be easily managed by the CPU and have a document printed by sending one character at a time.

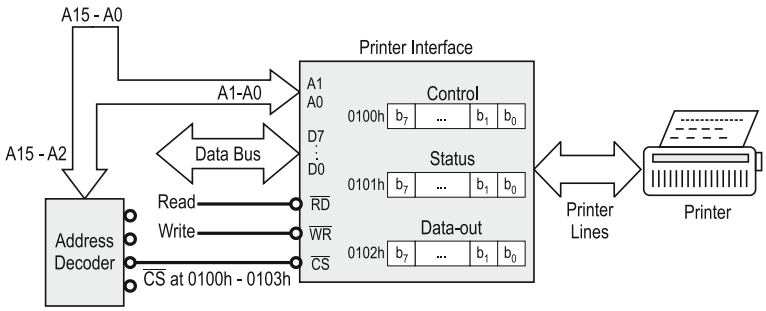

Fig. 3.17 Connection of printer interface to CPU buses and printer itself

Let us now consider the I/O port registers for the MSP430 microcontrollers as an actual interface example of interest for us. In general, MCU I/O ports work as blocks, half blocks, or single lines. For example, if a port has eight bits, all eight bits could work as input or output terminals. Sometimes they can be configured part as output and part as input. MSP430 I/O ports have all pins independently configurable. Hence, even if all the set is associated with one register, the individual bits in the register may be working each one independent of the others. The following example gives an introduction to the digital I/O ports of the MSP30 architecture. A detailed discussion of I/O ports is provided in Chap. 6.

Ports in the MSP430 are named as Px, where x may go from 1 to 10. The pins in a port (Px) are identified as P$x.n$, where n goes from 0 to 7. Not all ten ports are present in all MSP430 microcontrollers.

Example 3.8 *The MSP430 I/O port pins work independently. They can function either as input or output pins. That flexibility makes necessary at least three registers associated to the port: one for configuring the flow of information, one for input data, and one for output data. In addition, to optimize resources, many MCU pins are shared by two or more internal devices. This makes sense because it is very rare to use all of them in one application. Hence, one more register is necessary to select between the I/O or the other modules. The registers are listed next and their application as indicated in Fig. 3.19a.*

Direction Register (PxDIR): *Selects in or out direction function for pin, with 1 for output direction and 0 for input direction.*

Input Register (PxIN): *This is a read-only register. The value changes automatically when the input itself changes.*

Output Register (PxOUT): *to write signal to output. This is a read-and-write register.*

Function Select Register (PxSEL): *Used to select between I/O port or peripheral module function. With PxSEL.n = 0, pin Px.n operates as an I/O pin port; with PxSEL.n = 1, as a module pin.*

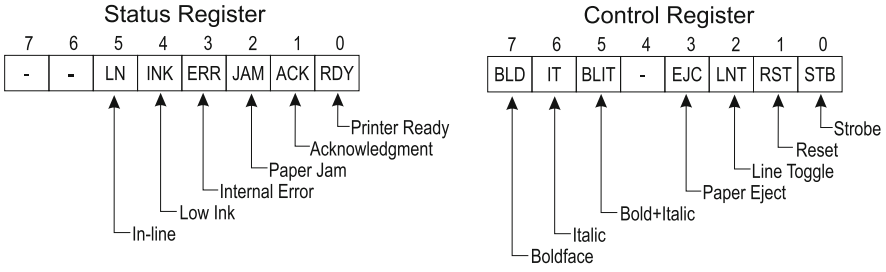

Fig. 3.18 Status and control registers in printer interface example

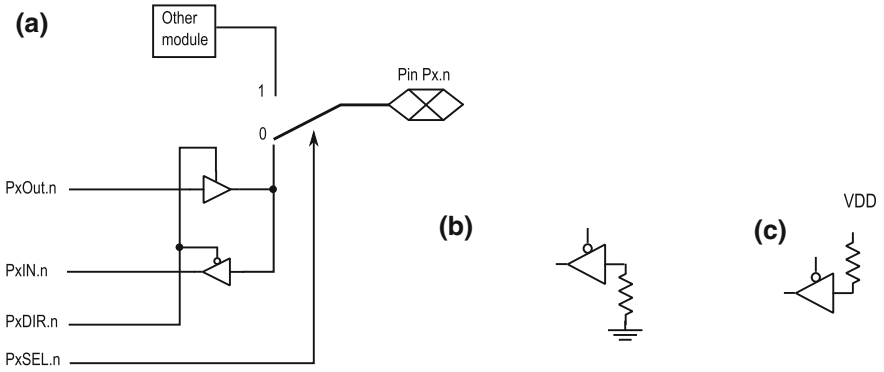

Fig. 3.19 Basic IO Pin hardware configuration: **a** Basic I/O register's functions; **b** pull-down resistor for inputs; **c** pull-up resistors

Let's say for example, pins 7 to 3 of port 6 are to be used as output pins, and pin 2 for operating the module, then we should use appropriate CPU instructions to write 0xF8 *in the P6DIR register and* 04h *in the P6SEL register.*

Since the input port goes through a three-state buffer, it is not advisable to leave it floating when the pin operates in input mode. It is necessary to connect a pull-up or a pull-down resistor, as shown in Figs. 3.19b, c.

Additional configuration registers might be included as part of an I/O port. Examples include registers to configure interrupt capabilities in the port, or to use internal pull-up/pull-down resistors, and other functions. Section 8.1 *in* Chap. 6 *provides a detailed discussion of I/O port capabilities in MSP430 microcontrollers.*

3.6.2 Parallel Versus Serial I/O Interfaces

In microcomputer-on-a-chip systems, most peripherals are connected to the data bus via a parallel interface, i.e., all bits composing a single word are communicated simultaneously, requiring one wire per bit. But I/O ports interacting with devices external to the system, may connect via parallel or serial interfaces. The ports are then referred to as *parallel I/O ports* and *serial I/O ports*. Serial interfaces require only one wire to transfer the information, sending one bit at a time.

Serial interfaces were the first to be in use for inter-system communications, especially the RS-232 used for devices requiring long distance connections. A reason for preferring serial lines was the economy resulting of using only one wire (plus ground). However, in early computer systems serial ports were slow, and for short length, fast connections, parallel interfaces were preferred. Thus in early computer systems, most printers, mass storage devices, and I/O sybsystems made broad use of

parallel ports. Protocols like the ISA/EISA/PCI Buses, GPIB, SCSI, and Centronics[9] are just a few examples of the myriad of parallel communications standards that flourished in that era.

On the other side, parallel connections have their own problems as speeds get higher and higher. In modern technology for example, wires can electromagnetically interfere with each other; also, the timing of signals must be the same and this becomes difficult with faster connections so now the pendulum is swinging back toward highly-optimized serial channels even for short length interconnections. Improvements to the hardware and software process of dividing, labeling, and reassembling packets have led to much faster serial connections. Nowadays we encounter printers, scanners, hard disks, GPS receivers, display devices, and many other peripherals that reside relatively close to the CPU using serial channels such as USB, FireWire, and SATA. All forms of wireless communication are serial. Even for board-level interconnections, standards like SPI, I^2C, and at a higher note, PCIe, to mention just a few examples, are representative of the trend serializing most types of inter- and intra-system communications.

3.6.3 I/O and CPU Interaction

Two major operations are involved when operating with the I/O subsystem: one is data transfer, i.e., sending or receiving data numbers, and the other is timing or synchronizing. In the first category, input and output port registers are the major players.

Input ports always transfers data toward the CPU. They may be of the *buffered input* or the *latched input* type. In the first case, they do not hold input data and the CPU can only read input present at that instant. Latched inputs do hold the data until this is read by the CPU, after which they are usually cleared and ready for another input.

Output port registers may be write-only or read-and-write registers, depending on the hardware design. The output state is in most cases latched, allowing to hold the output data at the pins until the next output operation is executed.

Timing or synchronizing is necessary when the nature of the peripheral or external device is such that interaction with the MCU must wait until the device is ready. For example, hard printing is a slower operation than electronic transfer. Hence, printers usually receive the information in a buffer, from which the data is processed. A printer cannot receive more data from the MCU once its buffer is full. Another example are data converters, which must complete conversion before delivering the result to the MCU.

Devices use a *flag* to indicate their readiness to receive or deliver data. A flag is a flip flop output that is set or cleared by the device when it is ready to communicate

[9] Centronics, although widely used in printers, never actually reached an official standard certification.

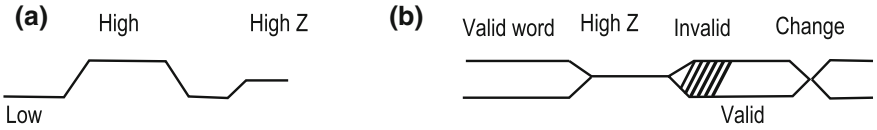

Fig. 3.20 Definitions of timing concepts. **a** Single signal timing convention, **b** Bus timing convention

with the MCU. Using this flag, two methods of synchronizing I/O devices are used: *polling* and *interrupt*.

Polling tests or polls the flag repeatedly to determine when the device is ready. The interrupt method continues with the normal program run and reacts to the flag when the device is ready. When a system is programmed so that it will only react to interrupts, it is said to be *interrupt driven*. Polling and interrupts are explained in more detail in later sections.

3.6.4 Timing Diagrams

Data transfer or communication between the CPU and other hardware components uses signals sent through the system buses. As explained earlier, data bus lines are used to transfer data, the address bus to select which memory cell or other piece of hardware the CPU works with, and signals from the control bus dictate what and how the transfer takes place. A bus requires a protocol to work, that is, a set of rules describing how to transfer data over them. This protocol includes information about the times required for each step in the process, to allow settling of states, delays and so on, as well as the order in which the signals are activated. The most common way to define the protocols is with *timing diagrams*, whose basic definitions are shown in Fig. 3.20.

In a timing diagram time flows from left to right. For single bits, two states are defined: high or low. Tri-state lines also include a third state called *high impedance* or "High Z" for short. State changes occur within a finite time greater than zero, as denoted by the oblique lines in Fig. 3.20. For buses, where several bits are to be considered, valid words are represented by parallel lines at high and low levels. High impedance states for the bus are represented by a line that is neither high or low.

Figure 3.21 shows a simplified example of time diagram protocols for write and read transfers. In this particular example, only two bus control signals are assumed to intervene: $\overline{enable}$, an active low signal which activates the memory block and marks the transfer time, and $read/\overline{write}$ which indicates the transfer direction. When $read/\overline{write}$ is high, it enables reading, that is, transfer from memory to the Data Bus to be retrieved by the CPU; and when low, enables writing from Data Bus into memory. Actual timing diagrams provided in manuals are more elaborated and

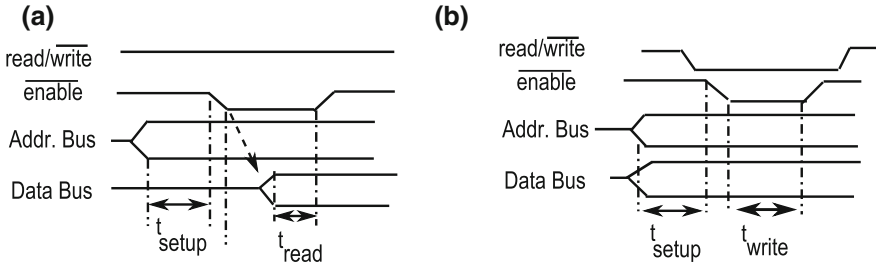

Fig. 3.21 Example of simplified write and read timings. **a** Read timing diagram, **b** Write timing diagram

may include several signals from the control bus. Chapter 6 deals with more detail on this topic.

3.6.5 MSP430 I/O Subsystem and Peripherals

The I/O and peripheral subsystem of the MSP430 family of microcontrollers is memory mapped, occupying addresses 0x0000 to 0x01FF. The operation of the different MSP430 members is controlled mainly by the information stored in special function registers (SFRs) located in the lower region of the address space, 0x0000 to 0x000F. The SFRs enable interrupts, provide information about the status of interrupt flags, and define operating modes of the peripherals. The ability of disabling peripherals not needed during an operation is a low power feature since the total current consumption can be reduced.

There is a wide selection of MSP430 microcontrollers models, even within one generation, with diverse offerings in peripherals. A partial list of peripherals available in different models include internal and external oscillator options, 16-bit timers, hardware for pulse width modulation (PWM), watchdog timer, USART, SPI, I2C, 10/12-bit ADCs, and brownout reset circuitry. Some family members include less common features such as analog comparators, operational amplifiers, programmable on-chip op-amps for signal conditioning, 12-bit DACs, LCD drivers, hardware multipliers, 16-bit sigma-delta ADCs, and multiple DMA channels. The user must see the specific device data sheet for available peripherals and SFRs. Peripherals are described later in this book.

Peripheral Map

Peripheral modules are mapped into the address space. SFRs occupy the space from 0h to 0Fh, followed by the 8-bit peripheral modules from 010h to 0FFh. These modules should be accessed with byte instructions. Read access of byte modules

using word instructions results in unpredictable data in the high byte. If word data is written to a byte module only the low byte is written into the peripheral register, ignoring the high byte.

The address space from 0100 to 01FFh is reserved for 16-bit peripheral modules.

I/O Ports and Pinout

MSP430 microcontroller models, as most microcontrollers, usually have more peripherals than necessary for a given application. Therefore, to minimize resources, several modules may share pins. Figure 3.22 shows package pinout descriptions for two models.

Inset (a) shows the MSP430G2331 pinout. We see here that, with the exception of pin 5, which has only one port connection to bit 3 of port 1, P1.3, all other port connections have more than one functions. It becomes therefore necessary to configure the hardware so as to include the selection of the desired function. This is yet another control register to be included in the port interface.

In the MSP430 families '1xx to '4xx, I/O ports are 8-bit wide, and are numbered P0 to P8; not all ports are available in a chip. Later families '5xx and '6xx have 8-bit ports P1 to P11, and may combine pairs to create 16-bit I/O ports named PA to PE. For example, P1 and P2 together can formed one 16-bit port PA. The reader should consult the specific device data sheet for further information.

3.7 CPU Instruction Set

The architecture and operation of a CPU are intimately related to how its instructions are organized in the *Instruction Set* and how instructions access data operands, the *Addressing Modes*. Instructions for the CPU are stored in memory just as any other word. What makes it an instruction is the fact that it is going to be decoded by the CPU during the instruction cycle, to find out what to do.

This section focuses on the software component of the CPU, that is, the instructions and the addressing modes. As such, the focus is now turned to the programmer's model of the CPU while describing this software side of the architecture. Before going into these details, let us first introduce useful notation to describe instructions, operations, and their programmer's model.

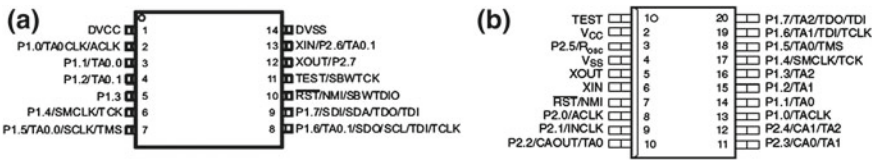

Fig. 3.22 Package pinout of two MSP430 microcontroller models: **a** MSPG2231 Pin description, **b** MSP430x1xx Pin description (*Courtesy of Texas Instruments, Inc.*)

3.7.1 Register Transfer Notation

For the programmer, the detailed description of hardware is not the main interest. The concepts more important to know are what registers the CPU provide for data transactions, what are the memory and I/O maps of the device, the instruction set, and so on. Hardware details become important when fine tuning the program, optimizing, debugging, and so on.

It is important for the programmer to have a notation available for operations in MCU environments, independent of the specific MCU architecture but taking into consideration the features of the systems. One such notation is the *register transfer notation* (RTN).

After executing an instruction, the contents of a register or cells in memory may be written upon with a new datum. In this case, the register or cell being modified is called *destination*. The *source* causing the change at destination may be a datum being transferred (copied) or the result of an operation. This process is denoted in abstract form as

$$destination \leftarrow source \quad \text{or} \quad source \rightarrow destination \tag{3.1}$$

In this book we adhere to the left hand notation. The notation in programmer's model for the different operands that may be used in RTN are as follows:

1. Constants: These are expressed by their value or by a predefined constant name, for example 24, 0xF230, MyConstant. Constants cannot be used in destination.
2. Registers: These are referred to by their name. If it is in abstract form without reference to a particular CPU, it is customary to use Rn, where n is a number or letters.
3. Memory and I/O: These are referred to by the address in parenthesis, as in (0x345E), which means "The data in memory at address 0x345E." Notice that it is data address, not physical address. If the address information is contained in register Rn we write (Rn), meaning by that "The data in memory at address given by Rn". We also say that the register *points* to the data.

Operations are allowed in address expressions, as in (Rn + 24h). Addressing modes can also be applied to RTN expressions, if there is no conflict in using them. Alternate notation for memory addresses is using the letter M before the parenthesis, as in M(0x345E), M(Rn + Offset).

When the size of an operand is not clear from notation, it must be explicitly stated at least once, to avoid ambiguity. (0240h), the datum at address 0240h, may be a byte, a 16-bit word, or a double word. If the size is not clear by the context, we write for example byte(0240h) or word(0240h).

Observe that Rn and (Rn) are distinct operands. The first one refers to the contents of register Rn, and also defines the data size. The second refers to the data in memory or I/O register whose address is the contents of Rn and does not specify the data size.

If the destination operand appears in the source too, the datum to be used in the source is that before the operation, while in the destination goes the result of the operation.

Example 3.9 *The following transactions illustrate RTN for memory operands. For these examples, let us assume word-size data at addresses before each transaction as* [0246h] = 32AFh *and* [028C] = 1B82h. *Moreover, let us assume little endian storage.*

RTN	Meaning	Result
(0246h) ← 028Ch	Store 028Ch at data address 0246h	[0246h] = 028Ch
Word(0246h) ← (028Ch)	Copy at memory location 0246h the word at memory address 028Ch	[0246h] = IB82h
Byte(0246h) ← (028Ch)	Copy at memory location 0246h the byte at memory address 028Ch	[0246h] = 3282h
(0246h) ← (0246h) + 3847h	Add 3847h to the current contents of memory location 0246h	[0246h] = 6AF6h
Byte(028Dh) ← (028Dh) − (0247h)	Subtract the byte memory location 0247h from the byte at memory address 028Dh	[028Ch] = E982h

Observe that in the second, third, and fifth transactions the data size had to be explicitly given, at least once, because otherwise the transaction become ambiguous.

The next example combines registers and memory locations.

Example 3.10 *Assume two 16-bit registers,* R6 *and* R7*, with contents* R6 = 4AB2h *and* R7 = 354Fh*, respectively. Moreover, assume words at addresses* [4AB2h] = 02ACh, [4C26h] = 94DFh *and* [4AB8h] = 3F2Ch. *Assume little endian convention when necessary. The following examples illustrate more RTN expressions:*

RTN	Meaning	Result
R6 ← 3FA0h	Load R6 with 3FA0h	R6 = 3FA0h
R7 ← R6	Copy in R7 the current contents of R6	R7 = 4AB2h
R7 ← (R6)	Copy in R7 the word whose memory address is stored in R6. Also: Copy in R7 the word in memory pointed at by R6.	R7 = 02ACh
byte(4C26h)← byte(4C26h) − (R6)	Subtract from the byte at address 4C26h the byte whose address is given by R6	[4C26h] = 9433h
(R6) ← (R6) + R7	Add the contents of R7 to the word pointed at by R6.	[4AB2h] = 37FBh
R7 ← (R6 + 6h)	Copy in R7 the memory word whose address is given by R6 + 6h	R7 = 3F2Ch
byte(R6) ← (R6) − 0x92	subtract 0x92 from the current memory byte at address given by contents of R6.	[4AB2h] = 021Ah

Sometimes we may simplify notation by simply stating in words the objective. For example, when comparing register contents arithmetically or testing register bits, the destination is the status register flags, but we may simply state the comparison or testing being done, and leave the SR implicit.

3.7.2 Machine Language and Assembly Instructions

As explained in Sect. 3.3.1, an instruction cycle starts by fetching a word from memory using the address stored at the Program Counter. This word is the *instruction word*. The complete instruction may consist of more than one word, but only the leading one is the instruction word. Depending on the number of words comprising an instruction, as well as its type, the CU will require one or more instruction cycles to complete the operation.

Since instructions are found in memory and loaded in registers, it is obvious that they consist of 0's and 1's. We say that they are encoded in *machine language*.

Machine Language Instructions

The bits of an instruction word are grouped by fields. Usually, the most significant one is the *operating code* (OpCode) which defines the operation to be performed. Other fields contain information about the *operands* and the *addressing mode*. The addressing mode is the way in which an operand defines the datum. For example, in an RTN expression we may have Rn or (Rn). In both cases the operand is Rn, but the datum is found in different places.

Following the OpCode, there might be information for zero, one or two operands, depending on the instruction. Some MCU models include up to three operands. This case is not considered in this book. The operands, together with addressing mode and other fields, provide information about data needed to execute the operation. Figure 3.23 shows four CPU instructions in the MSP430 machine language together with the operations done shown in RTN.

Notice that the suffix "h" is not shown for the words in hex notation. This is not an accident. *By default, it is customary to consider the base for machine language and addresses as hexadecimal.*

Assembly Language Instructions

To avoid using binary or hexadecimal representation for instruction words, scientists and engineers devised *high level* and *assembly* languages. Java, C, and Basic belong to the first group. Instructions in assembly language, the first to appear, on the other side, are the same as machine language. That is, each assembly instruction is associated with one, and only one, machine language instruction. But assembly language

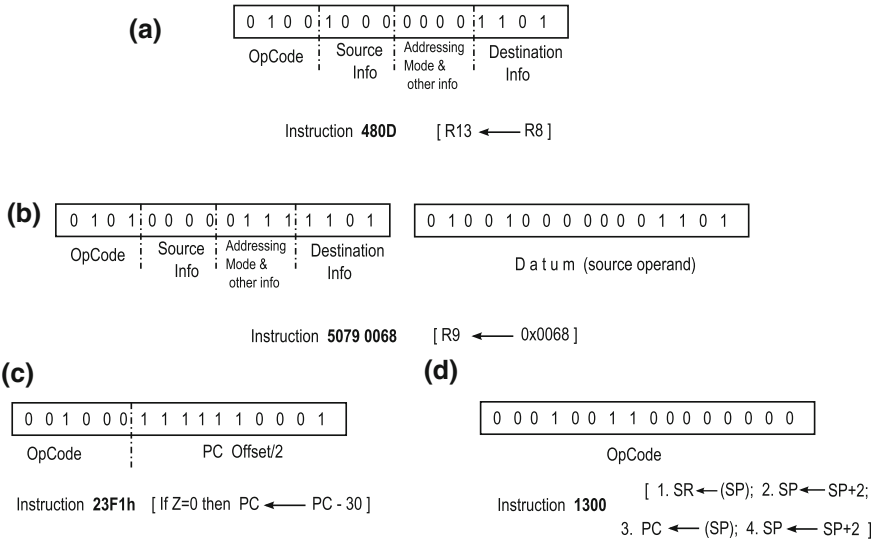

Fig. 3.23 Four machine language instructions for the MSP430

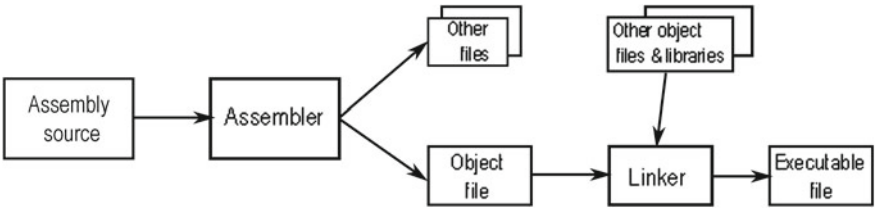

Fig. 3.24 Basic assembly process

targets a more human-like notation. The left table below shows the MSP430 assembly translations for the machine language instructions of Fig. 3.23. The table on the right shows examples of machine and assembly encoding for Freescale HC11.

Machine language	Assembly language
480D	`mov R8,R13`
5079 006B	`add.b #0x6B,R9`
23F1	`jnz 0x3E2`
1300	`reti`

Basic assembly process. The basic principles of the assembly process are illustrated in Fig. 3.24.

Machine language	Assembly language
C8 34	`EORB #$34`
C0 07	`LDD #7`
7E 10 00	`JMP $1000`
5C	`INCB`

Programmers write the program using a text processor to generate a *source file*. This source contains the assembly instructions as well as *directives*, instructions, *comments*, and *macro directives*. Directives are used to control the assembly process and are not executed by the CPU. Macro directives allow to group a set of assembly instructions into only one line; they serve to simplify the source from the human point of view.

The source file is then fed to an *assembler* which expands the macros and translates assembly instructions into machine language generating an object file and other files useful for debugging. *Interpreters* are a special type of assemblers that interpret assembly instructions in a program one line at a time.[10]

The object file is sent to the *linker*, which combines if necessary the file with previously assembled object files and libraries to generate the final *executable file*. This is the one to be actually downloaded onto the microcontroller memory.

Directives allow to organize the memory allocation, to define constants, labels, and introduce other features which facilitate the programmers work. Two pass assemblers run more than once over the source text to properly identify labels and constants.

Assembly instructions syntax depends on the microcontroller family, but the underlying principles are similar: a name for the OpCode, called **Mnemonics**, and **Operands**, specified using special syntax for addressing modes. Similarly, each assembler has its own directives, but again the underlying principles are very similar.

Chapter 4 is dedicated to assembly language programming for the MSP430 family. We focus in one family to have a real world reference, but the principles are similar for other MCU families, with the corresponding changes in syntax and adaptation to the underlying hardware.

3.7.3 Types of Instructions

At the most basic level, the instruction set of any CPU is composed of three types of instructions: data transfer, arithmetic-logic, and program control. The number and format of specific machine code instructions in each of these groups can change from one CPU architecture to another. However, in any case we expect a basic set of operations that serve as the basis for developing assemblers and high-level language compilers.

[10] An interesting online interpreter for the MSP430 instructions is found at http://mspgcc.sourceforge.net/assemble.html.

Data Transfer Instructions

Data transfer instructions, as implied by their name, move data information from one source to a destination. In fact, they only copy the source in the destination without modifying the source. The exception to this behavior is the instruction `swap`.

The execution of data transfer instructions involves the BIL and registers. Flags in the status register SR are usually not affected by these instructions. Examples of common data transfer operations/instructions include:

- `move` or `load`: Copies data from a source to a destination operand. In RTN, $dest \leftarrow src$.
- `swap`: Exchanges the contents of two operands. In RTN, $dest \leftrightarrow src$.
- Stack operations `push`, and `pop` or `pull`: These are explained in Sect. 3.7.4
- Port operations `in` and `out`: to read and write data from I/O ports in I/O mapped architectures.

In early processors based on a load-store architecture, data transfer instructions always involve a special data register called *accumulator*. Nowadays some families continue with this tradition.

Port operations are required in I/O mapped architectures. I/O memory mapped devices use the same data transfer instructions for memory and I/O port operands.

The stack operations always involve the stack pointer. The importance of these operations makes it worth a special section, Sect. 3.7.4, and will be explained there.

Arithmetic-logic Instructions

Arithmetic-logic instructions are those devoted to perform arithmetic and/or logic operations with data. They also include other operations with the contents of a register or memory word. Their execution uses the arithmetic logic unit, registers, and the BIL. Instructions of this type usually affect the flags in the SR.

Most MCUs, especially RISC type, involve at most two operands in the instructions. Therefore, this type of instructions are of the format

$$destination \leftarrow (DestinationOperand \oslash SourceOperand) \tag{3.2}$$

where $\oslash$ is an operation. Typical expected operations are addition, subtraction, and logic bitwise operations such as AND, OR, and XOR. Multiplication and division are not supported by all ALU's.

The actual set supported by an MCU depends on the architecture design of the CPU and ALU. Those operations not supported by it, usually have to be realized with software when necessary.

Compare and Test Operations: These two types of instructions realize an operation, usually subtraction or AND, without modifying any operand. The objective is to affect flags in the status register, usually to make decisions.

Rotate and Shift Operations: These type of instructions are not of the format shown in (3.2). Instead, they displace the bits internal to the destination operand.

Load-store architectures restrict all arithmetic-logic instructions to use the accumulator as one of the source operands and always as destination for storing the result. This limitation is overcome in RISC architectures and orthogonal architectural designs where virtually any GPR or memory location can be used as source or destination.

Bitwise Operations When the operation in (3.2) is of the logic type, very often it is bitwise. That is, the execution is realized bit-by-bit without one affecting the other, in the form

$$dest\_bit(0) \leftarrow (dest\_bit(0) \oslash src\_bit(0), dest\_bit(1) \leftarrow (dest\_bit(1) \oslash src\_bit(1), \ldots$$

Bitwise logic instructions allow us to set, clear or modify specific bits in an operand without affecting others, by exploiting the logic properties of the operation in the way shown in Table 3.4 and illustrated in Fig. 3.25. Also, we may test if a particular bit is 1 or 0.

Program Control Instructions

Program control instructions allow us to modify the default flow of execution in a program. By default, the address of the instruction to be fetched next follows right after the one being currently executed. This address is automatically loaded into the PC after the decode phase. When a program control instruction is executed, the contents of the PC may be changed so the following instruction to be fetched is not the default one. There are three types of program control instructions:

Unconditional Jump An unconditional jump always changes the program control flow to an address as indicated by the instruction. The action is $PC \leftarrow NewAddress$.

Conditional Jump The PC contents will be changed only if some condition of the status flags is met. That is : "If flag X = n, then $PC \leftarrow NewAddress$". Table 3.5 illustrates the basic conditional jumps using mnemonics for the notation.

Subroutine Calls and Returns These instructions allow the programmer to transfer program flow to and from special sections of code called subroutines which are outside the memory segment of the main code. The modus operandi of these instructions is explained in Sect.3.7.4.

Jumps are also called Branches in literature. Table 3.5 does not show all conditional jumps, only the basic ones depending on the common flags. Other conditional jumps use combination or operations with flags. We can think of conditional jumps as the instructions that will be associated to flowchart decision symbol, as illustrated in Fig. 3.26.

Common uses of the `jnz` instruction are in wait or delay loops and in iteration loops where a process has to be repeated N times. This is illustrated in Fig. 3.27.

Table 3.4 Logic properties and applications

`0.AND.X = 0;` `1.AND.X = X`	To **clear** specific bits in destination, the binary expression of the source has 0 at the bit positions to clear and 1 elsewhere (Fig. 3.25a)
`0.OR.X = X;` `1.OR.X = 1`	To **set** specific bits in destination, the binary expression of the source has 1 at the bit positions to set and 0 elsewhere (Fig. 3.25b)
`0.XOR.X = X;` `1.XOR.X =` $\overline{X}$	To **toggle or invert** specific bits in destination, the binary expression of the source has 1 at the bit positions to invert and 0 elsewhere (Fig. 3.25c)

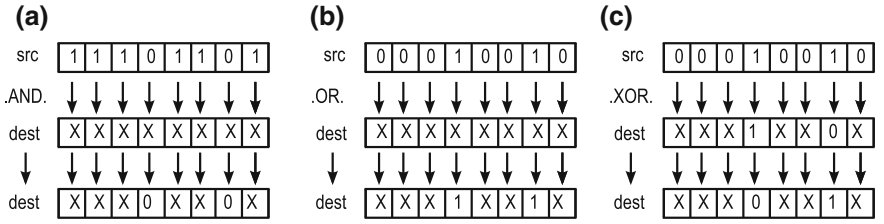

Fig. 3.25 Using logic properties to work with bits 1 and 5 only

Table 3.5 Conditional jumps

Mnemonics	Meaning	Mnemonics	Meaning
`jz`	Jump if zero (Z = 1)	`jn`	Jump if negative (N = 1)
`jnz`	Jump if not zero (Z = 0)	`jp`	Jump if positive (N = 0)
`jc`	Jump if carry (C = 1)	`jv`	Jump if overflow (V = 1)
`jnc`	Jump if no carry (C = 0)	`jnv`	Jump if not overflow (V = 0)

Notice in the pseudo code presented the use of the *label*. A label is attached to an instruction and it can be used in a jump instruction to indicate the address of which instruction should result in the PC if the branch is executed. We will go over labels later again.

3.7.4 The Stack and the Stack Pointer

The stack, a specialized volatile memory segment used for temporarily storing data, is managed with the *Stack Pointer* (SP), both by programming or by the CPU itself.

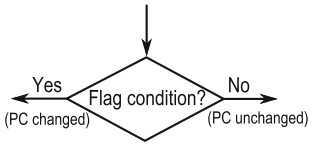

Fig. 3.26 Decision symbol associated to conditional jumps

To store data in this segment using a stack transaction is *to push*. Retrieving a datum from the stack is to *pop*, or to *pull*. Any stack transaction makes implicit use of the SP, which points to the address where the push or pop is to be realized.

The term "stack" was coined in analogy with tray stacks,which operate in a natural Last-In-First-Out (LIFO) fashion: the item on the top is the last one put there, and the first to be pulled out from the stack. For this reason, the contents of the stack pointer is usually called *Top-of-stack* (TOS). After each operation, the SP is updated with the new TOS.

In most MCU architectures, the stack is user or compiler defined, and the stack usually grows *downward*, not upwards.[11] This means that the SP contents diminishes each time a new item is pushed onto the stack and increases when an item is pulled. The SP usually starts pushing at the top address of RAM, opposite to usual memory operations which start storing at the bottom. Considering this, it makes sense for the stack to grow downward.

On the other hand, the analogy with the term "top of stack" address should be handled with care, since the memory address will depend on whether we are pushing or popping a datum. Let us illustrate the stack process with an analogy to a rack for DVDs, using Fig. 3.28. Each shelf has a number attached to it: 10, 9, …, 0. Let these be the shelves' addresses.

In (a), five DVD's have been stored, starting at shelf 10. The last one in is occupying shelf 6. If another DVD is pushed, the TOS address to consider is shelf 5, where this DVD will go. If another item is pushed afterwards, it will go to TOS = 4. Now, if we pull or pop the last item in, TOS = 6 as illustrated in (b). After this pop, the new "last-in" item is in address 7, which becomes the TOS for popping.

Hereafter, we will simply use the term "top of stack" (TOS) for any case, unless it becomes necessary to differentiate. Since SP is precisely related to the TOS address, we refer to its contents as the TOS. The user, or the compiler, may define the stack segment by initializing the SP register.

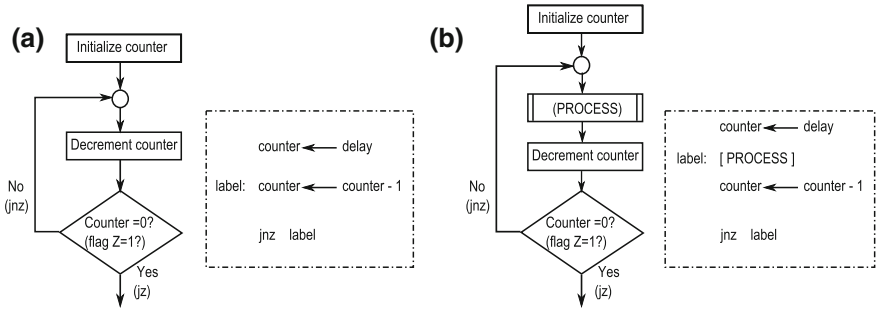

Fig. 3.27 Flow diagram and instruction skeleton associated to (**a**) delay loops and (**b**) iteration loops

[11] Just like the stacks of most graphic calculators and computer monitors, where new items appear at the bottom and not at the top.

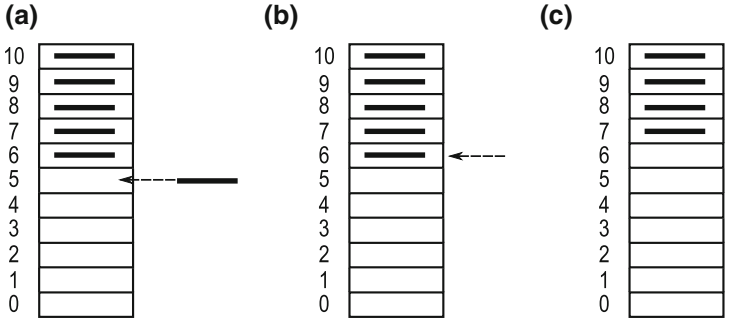

Fig. 3.28 Illustrating the stack operation. **a** TOS = 5 for pushing, **b** TOS = 6 for pulling, **c** After pulling, new TOS = 7

Depending on the MCU, the SP will contain the TOS for pushing or the TOS for pulling. The case will determine how the push and pop operations are physically performed, and also how the programmer may read or change contents of previously pushed items without going into a stack transaction.

CASE A: Pushing when SP points at TOS:
The operations are performed as follows:
1. Pushing a source:

$$(SP) \leftarrow src$$

$$SP \leftarrow SP - N$$

That is, it first stores the source and then updates SP.

2. Pulling into a destination:

$$SP \leftarrow SP + N$$

$$dst \leftarrow (SP)$$

That is, it first updates SP, and then retrieves the data.

CASE B: Popping with SP pointing at TOS.
The operations are performed as follows:
1. When pushing a source:

$$SP \leftarrow SP - N$$

$$(SP) \leftarrow src$$

That is, it first updates SP and then stores the data.

2. When pulling into destination:

$$dst \leftarrow (SP)$$

$$SP \leftarrow SP + N$$

That is, it first retrieves the data and then updates the source.

The value of N is 1, 2 or 4, depending on the MCU model. For 4 and 8-bit MCU's, N = 1. For 16-bit MCU's, N = 2. For 32-bit MCU's, N may be fixed to 4, or else N = 2 or 4, depending on the length of the manipulated data. Let us illustrate the stack operation with two examples. The user should consult the MCU's documentation to know which case applies to the system being used. The MSP430, in particular, uses the TOS for pop.

Example 3.11 *This example illustrates the stack process in the 16-bit MSP430 family using Fig. 3.29. In this family, register SP points to the last item pushed onto the stack (next to be pulled). SP updating is done by adding/subtracting 2 to its contents.*

For simplicity, memory contents are expressed without the "h" suffix and refer to word data, occupying two physical byte space locations. Word addresses are even.

Figure 3.29a illustrates the push operation, with initial contents for registers SP and R9 being 027Eh *and* 165Ah, *respectively (Fig. 3.29a.1). Notice that SP is pointing to the address* 027Eh, *which is the TOS. The push operation is executed in two steps:*

Step 1: Update TOS: SP ← (SP − 2), i.e., SP becomes 027Eh-2 = 027Ch

Step2: Copy the contents of R9 at the TOS, (SP) ← R9.

The result is shown in Fig. 3.29a.2.

Now consider the pop operation illustrated by (b), with similar initial conditions, as shown in Fig. 3.29b.1. The pop operation is executed in two steps:

Step 1: Copy the contents of the TOS onto R9, R9 ← (SP)

Step2: Update TOS: SP ← (SP + 2), i.e., SP = 027Eh+2 = 0280h

The result is shown in Fig. 3.29b.2.

Example 3.12 *Now let us illustrate the push and pop operations for the 8 bit HC11 microcontroller, where SP points to the address for the new item. Now N = 1 for SP updating; the register being used in the example is named Accumulator A (ACCA). Fig. 3.30 illustrate the operations.*

Figure 3.30a illustrates the push operation. Assume that before the operation the contents for registers SP and Accumulator A are 027Eh *and* 5Ah, *respectively (Fig. 3.30a.1). Notice that SP is pointing to the address* 027Eh, *which is the TOS. The* **push operation** *is executed in two steps:*

Step 1: Copy the contents of Accumulator A at the TOS, (SP) ← ACCA.

Step2: Update TOS: SP ← (SP − 1), i.e., SP becomes = 027Eh − 1 = 027Dh

The result is shown in Fig. 3.30a.2.

The pop operation illustrated by (b), with similar initial conditions, as shown in Fig. 3.30b.1. The pop operation is executed in two steps:

Step2: Update TOS: SP ← (SP + 1), i.e., SP = 027Eh + 1 = 027Fh

Step 1: Copy the contents of the TOS onto Accumulator A, ACCA ← (SP)

The result is shown in Fig. 3.30b.2.

Observe in both examples that when a push is done, the original contents at the memory location is lost but this does not happen when a pop occurs. However, the

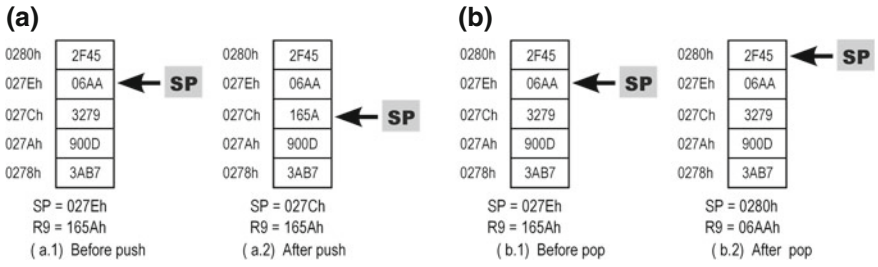

Fig. 3.29 The stack, SP register. **a** Push operation, **b** Pop operation

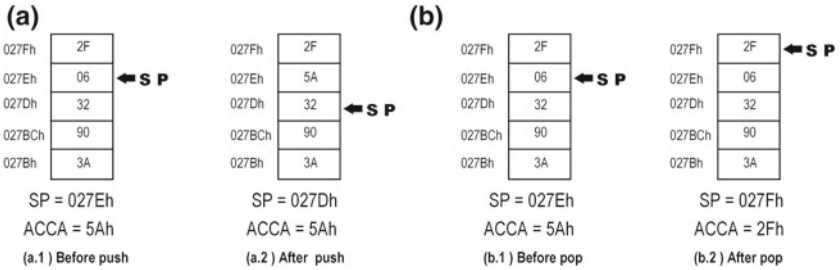

Fig. 3.30 Another example: SP register. **a** Push operation, **b** Pop operation

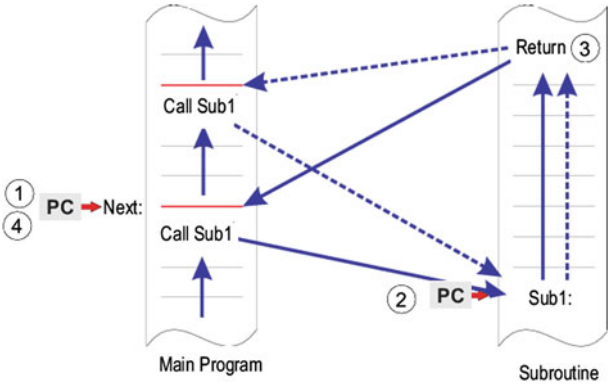

Fig. 3.31 Invoking a subroutine

same data cannot be retrieved twice with a stack operation, because of the dependence on the pointer SP, which no longer refers to that location. Usually, other operations are available for this end.

Hereafter, irrespectively of how SP operates, push and pop operations will be denoted in RTN as "(TOS)← source" and "dest ← (TOS)", respectively.

Stack and Subroutines Even if the user does not utilize push and pop operations explicitly, these may occur transparent to the programmer when dealing with subroutines and interrupts, which involves a special kind of subroutine called Interrupt Service Routine (ISR). Subroutines are program modules logically separated and independent of the main program, which are invoked to do a specific task and then continue the main program with the instruction following the call or interrupt. The process is illustrated in Fig. 3.31 where a subroutine `Sub1` is called from `Main Program`.

When a program invokes (calls) a subroutine, the current contents of the PC is pushed onto the stack. Observe that just before executing the call instruction, the PC contains the address of the next instruction to be fetched, the one right after the call. Therefore, by pushing the PC, the CPU is effectively storing in memory the address of the instruction following the call. Returning from a subroutine causes the

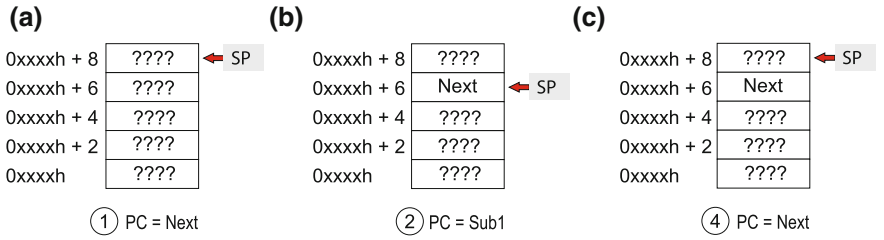

Fig. 3.32 Stack pointer use in the invocation and return of subroutines. **a** Stack before call, **b** Stack after call, **c** Stack after return

PC to be popped from the stack. This way program execution can always resume at the instruction next to where the subroutine was invoked, no matter where in the program the invocation occurs. Figure 3.32 illustrates the stack operations for the previous figure. Just before the call, the SP is pointing at location 0xxxxh+8 as illustrated by inset (a). The process is then as follows:

Step 1: After the CALL instruction is fetched and decoded, but still not executed, the PC will point to the next instruction, at address `Next`.

Step 2: The execution of the invocation results in the current PC value being pushed onto the stack and loaded with the start address of `Sub1`, as depicted in Fig. 3.32b. Now the instruction to be fetched is the first one in the subroutine.

Step 3: The subroutine instructions are executed in sequence up to the Return Instruction.

Step 4: When the `Return` instruction is executed, the PC is popped, restoring the value `Next`, allowing the program execution to continue in the main program at the instruction following the call. At this point, SP will be back at address 0xxxxh+8 as shown in Fig. 3.32c.

Manipulating the SP in a program that uses subroutines requires caution to avoid loosing the return addresses. Chapter 4 provides a more detailed explanation of the use of the stack and push an pop operations.

3.7.5 Addressing Modes

Addressing modes can be defined as the way in which an operand is specified within an instruction so as to indicate where to find the data with which the operation is executed. The addressing mode is denoted using a specific syntax format, proper of the microcontroller family. Instructions with implicit operands are said to use *Implicit Addressing Modes*.

The machine language instruction contains information about the addressing mode which, when decoded, directs the sequence of operations in the CU to achieve the desired results. However, in some cases a strict disassembly of the machine language

instruction, out of the context of a program, can yield a different result of what we could expect. There are several reasons for this. For example, there are modes used in the instruction fetching process which should not be of concern to the programmer, but are necessary to consider in the hardware design. For our purposes, we focus on the addressing modes used in assembly language.

Now, there are two different type of data we should consider. One associated with program flow instructions, like jumps, in which the data information refers to how the PC contents is to be modified. The second type is the one used in transfers and general type of instructions manipulating data. We discuss first this second group.

In general, the data to be used or stored in a transfer or in an arithmetic or logic instruction can be located in only one of the following possible places:

1. It may be explicitly given,
2. It may be stored in a CPU register,
3. It may be stored at a memory location, or
4. It may be stored in an I/O port or peripheral register.

In memory I/O mapped systems, the fourth and third options are equivalent. The syntax used in the addressing mode in an assembly language instruction tells the assembler which of these cases apply. Unfortunately, the syntax is not standard for the different microcontroller families, so the user must learn which one applies to the machine being used. Also, the names given to similar addressing modes is not completely standard, although this one tends to be more uniform. In what follows, the syntax used by the MSP430 family, which is shared by many other families, will be used. But the reader must bear in mind that it is absolutely necessary to look at the particular list of the family being used.

For the four cases listed above for the data location, the first two are almost standard for the different processors: The most basic types of addressing modes supported by almost all processors are listed next.

Immediate Mode—syntax: **#Number**. In this mode, the value of the operand Number is the datum.[12] Immediate mode is reserved only for source operands, since a number cannot be changed by an operation.

Register—Syntax: **Rn**. The operand is the CPU register Rn and the datum is contained in Rn.

When the datum is found in memory, we need to indicate the address. The information or mode is *absolute* if the address is given, and *relative* if it is expressed as an offset with respect to a given address. The basic modes are the following

Absolute or Direct Mode—syntax: **Number**. "Number" is the address where the datum is located. This mode is sometimes referred to as *symbolic mode*.

Indirect—syntax: **@Rn**. The datum is found in the memory location whose address is given by the contents of register Rn. We say that the register points at the datum. This mode is also called *Indirect Register Mode*.

[12] It receives the name "immediate" because at the machine level, the word containing the number goes immediately after the instruction word.

Table 3.6 Addressing modes examples

Assembly	RTN	Comment
`mov src,dest`	dest ← src	Copy or load source to destination
`add src,dest`	dest ← dest + src	Add source to destination
`sub src,dest`	dest ← dest − src	Subtract source from destination
`and src,dest`	dest ← dest .AND. src	Bitwise AND source to destination
`xor src,dest`	dest ← dest .XOR. src	Bitwise XOR source to destination
`cmp src,dest`	dest − src	Compare does not affect dest., only flags

Indexed—syntax: **X(Rn)**. This mode specifies the datum address as the sum of number X address plus the contents of register Rn. X is sometimes called *base* and Rn is said to be used as an *index*; for this reason this mode is also named *base indexed mode*.

Although the basic concepts may the same, remember that neither the nomenclature nor the syntax are standard. Let us look now at the jumps.

Jump instructions can only have an address as an operand. Normally, but again not always, the machine language form uses what is called *Relative addressing mode*, which indicates an offset with respect to the current PC contents. The new PC contents is the sum of the current PC contents and a two's complement offset, (PC← PC + offset). Hence, the maximum jump is limited by the number of bits used in the offset. In some processors, unconditional jumps use an absolute mode also in the machine language form.

To make programmers' life easier, the assembly form uses the absolute or direct mode only and the assembler calculates the needed offset. When using two-pass assemblers, labels can be used. A label is a symbolic name given by the programmer to an address. The RTN codes of Fig. 3.27 use labels.[13]

Two-pass assemblers allow programmers to use custom defined constants, including labels, thereby making the programmer's task easier. They are called "two-pass" because they need to go over the text more than once to identify and translate the constants that have been defined.

Other addressing modes can also be supported by a particular processor, which can enhance the flexibility of the instruction set. Also, remember that the syntax mentioned above is not standard. A discussion of the addressing modes supported by the MSP430 is offered in Chap. 4. The following examples illustrate the above addressing modes. For these examples we use some MSP430 assembly language instructions, shown in Table 3.6. The examples show the syntax for the source and destination to identify where the data is.

Example 3.13 *This example illustrates the addressing modes mentioned before with the help of Fig. 3.33. This figure shows the contents of the 16-bit registers R6, R8*

[13] Some interpreters, which work line by line, require the offset to be directly specified, while others accept the address and calculate the offset.

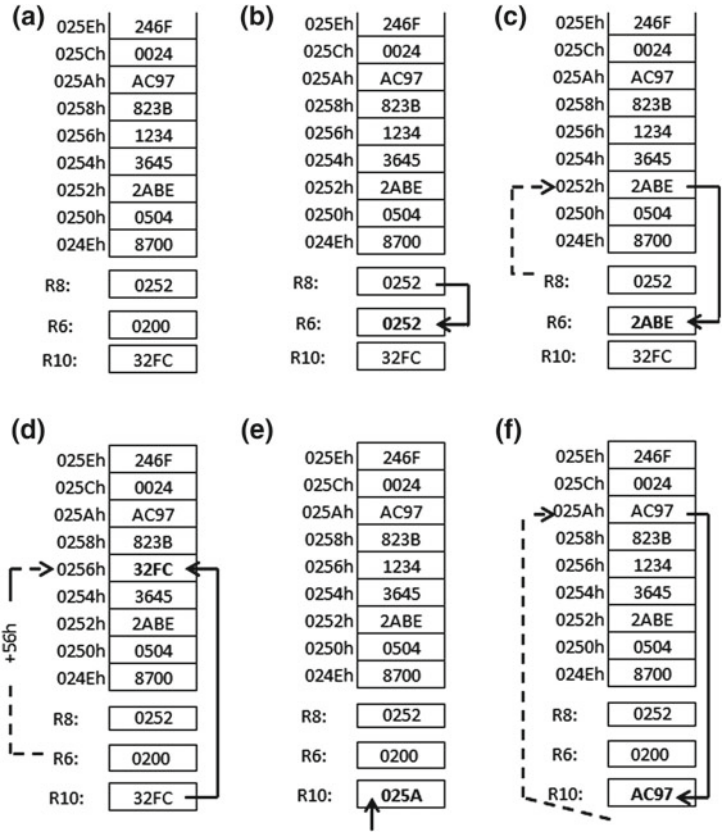

Fig. 3.33 Illustrating addressing modes. **a** Intial condition, **b** mov R8, R6, **c** mov @R8, R6, **d** mov R10, 56h(R6), **e** mov #0x025A, R10, **f** mov 0x025A, R10

and R10, and a memory segment of a 64 KB address space. Memory is shown in word format, that is, only even addresses are presented and contents are shown for a complete 16-bit word, four hex digits. Contents of registers and memory are in hex notation without suffix.

Part (a) shows the contents before any instruction. Parts (b) to (f) illustrate the addressing mode using data transfer instruction in format `mov source, destination`. *The solid arrows show the transfer and the dashed lines the address pointers; the result is highlighted in each case. Now let us look into details.*

(b) `mov R8, R6`, R6 ← R8 *in RTN, means "copy the contents of register R8 onto Register R6".* R6 = 0252h *after this instruction.*

(c) `mov @R8, R6`, R6 ←(R8) *in RTN, means "copy the memory data word whose address is given by register R8, onto Register R6". Since* R8 = 0252h, *we search for address 0252h in the memory space where we see* (0252h) = 2ABEh. *Therefore,* R6 = 2ABEh *after this instruction.*

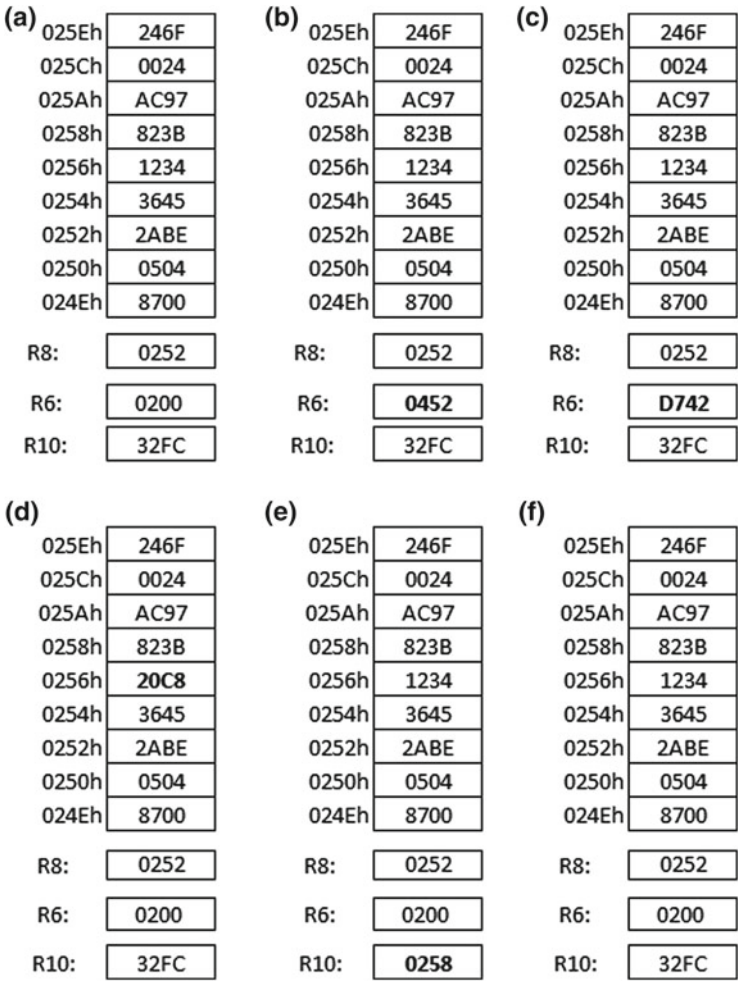

Fig. 3.34 Illustrating addressing modes with arithmetic-logic instructions

(d) `mov R10,56h(R6)`, *or* (R6 + 56h) ←R10 *in RTN, means "copy the contents of register R10 onto the memory location whose address is found by adding* 56h *to the contents of Register R6". Since* R10 = 32FCh, *then* (R6 + 56h) = (0256h) = 32FCh *after this instruction.*

(e) `mov #0x025A, R10`, *or* R10 ← 0x025A *in RTN, means "Register R10 is loaded with number 025Ah" Then* R10 = 025Ah *after this instruction.*

(f) `mov 0x025A, R10`, *or* R10 ← (025Ah) *in RTN, means "Copy at Register R10 the data in memory whose address is 025Ah" Looking at the memory space, we find* (025Ah) = AC97h. *Then* R10 = AC97h *after this instruction.*

Let us now work with other instructions, combined with different addressing modes. This is done in the following example.

Example 3.14 *Using the same set of data shown in Fig. 3.33a, let us now work another set of operations, as shown in Fig. 3.34. As before, contents of registers and memory are in hex notation and inset (a) shows the contents before any of the other instructions.*

(b) `add R8,R6`, R6 ← R6 + R8 *in RTN, means "add the contents of registers R8 and R6, and store the result in Register R6". Since* 0200h + 0252h = 0452h, *then* R6 = 0452h *after this instruction.*

(c) `sub @R8,R6`, R6 ← R6 − (R8) *in RTN, means "subtract from the contents of R6 the data in memory located at the address provided by register R8, and store result in Register R6". Since* R8 = 0252h, *we search for address 0252h in the memory space where we see* [0252h] = 2ABEh. *Now* 0200h − 2ABEh = D742h. *Therefore,* R6 = D742h *after this instruction.*

(d) `xor R10,56h(R6)`, (R6 + 56h) ← R10.xor.(R6 + 56h) *in RTN, means "perform a bitwise XOR operation with the contents of register R10 and the content of the memory location whose address is found by adding 56h to the contents of register R6, and store the result in the same memory location". Since* R10 = 32FCh *and* (R6 + 56h) = (0256h) = 1234h *then the performed operation is as shown below. Notice the toggling effect according to the source.*

$$\begin{array}{l} 0001001000110100 \text{ .xor.} \\ \underline{0011001011111100} = \\ 0010000011001000 \end{array}$$

so $(R6 + 56h) = 20C8h$ after this instruction.

(e) `and #0x025A, R10`, R10 ← 0x025A .and. R10 *in RTN, means "the contents of Register R10 is bitwise AND-ded with number 025Ah, the result being stored in R10" Since* R10 = 32FCh, *the operation is*

$$\begin{array}{l} 0000001001011010 \text{.and.} \\ \underline{0011001011111100} = \\ 0000001001011000 \end{array}$$

so R10 = 0258h *after this instruction.*

(f) `cmp 0x025A, R10`, *will take the difference between* R10 *and the data word at memory address* 0x025A, *that is* R10 − (0x025A), *without affecting any of the operands, just the flags. Since* (025Ah) = AC97h *and* R10 = 32FC, *and subtraction is done by two's complement addition, then* R10 − (0x025A) ⇒ 32FC + 5369 = 8665, *with the flags being then* C = 0, Z = 0, N = 1, V = 1.

3.7.6 Orthogonal CPU's

A CPU is said to be *orthogonal* if all its registers and addressing modes can be used as operands, except for the immediate mode as a destination. Very few microcontrollers accept the immediate mode as destination operand, which make sense only for compare and testing purposes but not for cases where the destination should store a result.

Orthogonality has several advantages, among which the ability to write compact codes. Many designers prevent however the use of special registers as operands, either source or destination, to prevent unexpected results. Orthogonality requires the programmer to use some operands like the Program Counter register with great care.

3.8 MSP430 Instructions and Addressing Modes

The MSP430 CPU supports 27 machine language *core instructions*, that is, instructions that have unique opcodes decoded by the CPU. The length of the machine language instructions are one to three words. There are 24 additional *emulated instructions* supported by assemblers, introducing particular and more popular mnemonics for easier program writing and reading. For example instruction "`inv R5`" is emulated by the core instruction "`xor #0xFFFF,R5`". It is easier for the programmer to write the former when intending to invert the contents of R5. In a certain sense, emulated instructions may be considered as standard macros.

The MSP430 supports seven addressing modes, including the ones mentioned before, albeit with a slightly different notation. The instructions and addressing modes for the MSP430 are discussed in Chap. 4, which is devoted to assembly programming.

Although advertised as orthogonal, MSP430 microcontrollers are not fully orthogonal since two of its non immediate addressing modes are invalid for destination. This is a small price paid by the MSP430 designers to maintain all instruction words of equal length.

3.9 Introduction to Interrupts

The topic of interrupts and resets involve both hardware and software subjects, but it is also closely related to how a CPU operates. For this reason it becomes important to provide an introductory discussion on the subject at this point. An in-depth discussion of the hardware and software components for supporting a reset are offered in Chap. 6, Sect. 6.5. Moreover, Chap. 7 offers a comprehensive discussion of hardware and software components required for handling interrupts in general.

3.9.1 Device Services: Polling Versus Interrupts

The CPU in a microcontroller or microprocessor is a sequential machine. The control unit sequentially fetches, decodes, and executes instructions from the program memory according to the stored program. This implies that a single CPU can only be executing one instruction at any given time. When it comes to a CPU servicing its peripherals, the program execution needs to be taken to the specific set of instructions that perform the task associated to servicing each device. Take for example the case of serving a keypad.

A keypad can be visualized at its most basic level as an array of switches, one per key, where software gives meaning to each key. The switches are organized in a way that each key depressed yields a different code. For every keystroke, the CPU needs to retrieve the associated key code. The action of retrieving the code of each depressed key and passing them to a program for its interpretation, is what we call *servicing the keypad*. Like the keypad in this example, the CPU might serve many other peripherals, like a display by passing it the characters or data to be displayed, or a communication channel by receiving or sending characters that make up a message, etc.

When it comes to the CPU serving a device, one of two different approaches can be followed: *service by polling* or *service by interrupts*. Let's look at each approach with some detail.

Service by Polling

In service by polling, the CPU continuously interrogates or polls the status of the device to determine if service is needed. When the service conditions are identified, the device is served. This action can be exemplified with a hypothetical case of real life where you will act like a polling CPU:

Assume you are given the task of answering a phone to take messages. You don't know when calls will arrive, and you are given a phone that has no ringer, no vibrator, or any other means of knowing that a call has arrived. Your only choice for not missing a single call is by periodically picking-up the phone, placing it to your ear and asking "Hello! Anybody there?" hoping someone will be in the other side of the line. This would be quite an annoying job, particularly if you had other things to do. However, if you don't want to miss a single call you'll have to put everything else aside and devote yourself to continuously perform the polling sequence: pick-up the phone, bring it to your ear, and hope someone is on the line. Since the line must be available for calls to enter (sorry, no call-waiting service), you have to hang-up and repeat the sequence over and over to catch every incoming call and taking the messages. What a waste of time! Well, that is polling.

Service by Interrupts

When a peripheral is served by interrupts, it sends a signal to the CPU notifying of its need. This signal is called an *interrupt request* or IRQ. The CPU might be busy performing other tasks, or even in sleep mode if there were no other tasks to perform, and when the interrupt request arrives, if enabled, the CPU suspends the task it might be performing to execute the specific set of instructions needed to serve the device. This event is what we call an *interrupt*. The set of instructions executed to serve the device upon an IRQ form what is called the *interrupt service routine* or ISR.

We can bring this interrupting capability to our phone example above.

Let's assume that in this case your phone has a ringer. While expecting to receive incoming calls, now you can rely on the ringer to let you know that a call has arrived. In the mean time, while you wait for calls to arrive you are free to perform other tasks. You could even get a nap if there were nothing else to do. When a call arrives, the ringer sounds and you suspend whatever task you are doing to pick-up the phone, now with the certainty that a caller is in the other end of the line to take his or her message. The ring sound acts like an interrupt request to you. A much more efficient way to take the messages.

Interrupts can be used to serve different tasks. The following are just a few simple examples that illustrate the concept:

- A system that toggles an LED when a push-button is depressed. The push-button interface can be configured to trigger an interrupt request to the CPU when the push-button is depressed, having an associated ISR that executes the code that turns the LED on or off.
- A message arriving at a communication port can have the port interface configured to trigger an interrupt request to the CPU so that the ISR executes the program that receives, stores, and/or interprets the message.
- A voltage monitor in a battery operated system might be interfaced to trigger an interrupt when a low-voltage condition is detected. The ISR might be programmed to save the system state and shut it down or to warn the user about the situation.

Servicing Efficiency: Polling or Interrupts?

There innumerable events that can be configured to be served by interrupts. Note that any or all of them could be served by polling, but that would require having the CPU tied to interrogating (polling) the corresponding interface to determine when service is needed. This would result in a very inefficient use of the CPU.

To give an idea of how inefficient polling results, let's analyze one of the examples above. Let's assume the CPU serving the push-button and LED example is running at a clock frequency of 1 MHz and executes one instruction every four clock cycles. This is a very conservative scenario as today's processors con run at frequencies well over 1 GHz. This implies the CPU will roughly execute 250,000 instructions every second, or one instruction every four microseconds. To know if the push-button has

been depressed the CPU only needs to execute two instructions: one to read the I/O pin associated to the push-button and a conditional jump instruction to make the decision: eight microseconds in total.

Consider a user repeatedly depressing the push button. Let's say the user is really fast, and able to push the key ten times per second. This is once every 100 ms. If we have the CPU polling this key interface to determine when the button was depressed, in 100 ms the processor would have checked it 12,500 times to find it was depressed only once. The efficiency in the CPU usage under these circumstances would be 1/12500 x 100 = 0.008 %. This implies that 99.992 % of the CPU cycles were wasted checking for a push-button that was not depressed. Servicing the same event with an interrupt would require only one action: toggling the LED when the push-button was depressed, which can be performed with a single instruction. This yields an efficiency of nearly 100 % and tons of CPU cycles saved. The CPU can use the rest of the time for attending other requests, executing tasks in the main program, or just saving power in a low-power sleep mode.

Despite this illustrative example, and all said about polling and interrupts, it deserves to mention that not in every situation using interrupts will be better than polling. There are situations where interrupts will not serve the purpose and polling will be the best alternative. Interrupts suffer from latency and are susceptible to false triggering. So in applications where the interrupt response lag might be too long or where noise might render interrupts unusable, polling becomes the first alternative to serving peripherals. Chapter 7 discusses these aspects in more detail.

3.9.2 Interrupt Servicing Events

In general, a CPU might support two different types of interrupt requests: maskable and non-maskable.

Maskable Interrupt Requests: A type of request that can be masked meaning that they can be disabled by clearing the CPU's global interrupt enable (GIE) flag in the processor status word (PSW). When the GIE is clear, the CPU will not honor any maskable interrupt request. This gives the programmer an ability of deciding when the CPU is to accept or not interrupt requests. Maskable interrupts are the most common type of interrupts managed in embedded systems. In most cases they are referred to as simply *interrupts*.

Non-maskable Interrupt Requests (NMI): Cannot be masked by clearing the GIE in the CPU and are therefore always served when they arrive at the CPU. This type of interrupt requests are reserved for system critical events than cannot be made to wait. We'll refer to them as *NMIs*.

Although different CPU architectures might have different ways of serving interrupts, there are several steps that are common to any processor. Once an interrupt request is accepted, and assuming the processor was in the middle of executing an

arbitrary instruction in a program, the fundamental steps taking place in the CPU to serve an interrupt include the following:

Step 1 Finish the instruction being executed. The CPU never truncates instructions. Interrupt requests are always served between instructions.
Step 2 Save the current program counter (PC) value and, in most CPUs, the status register (SR), onto the stack. Some processors might save other registers as well.
Step 3 Clear the global interrupt enable flag. This action, performed by most CPUs, disables interrupting an ISR under execution. This condition could be overridden, but is generally not recommended.
Step 4 Load the program counter (PC) with the address of the ISR to be executed. The way this step is accomplished changes with the interrupt handling style, but the net result is always the same: getting into the PC the address of the first instruction of the ISR to be executed.
Step 5 Execute the corresponding ISR. The ISR ends when the special instruction *interrupt return* (IRET or RETI) is executed.
Step 6 Restore the program counter and any other register that was automatically saved onto the stack in Step 2. This action is the result of executing the interrupt return instruction. Note that by restoring the status register, the interrupts become unmasked. Restoring the PC causes resuming the interrupted program at the next instruction where it was left.

Supporting interrupts in an application fundamentally needs four basic requirements:

1. A means for saving the program counter and status register, as indicated above in Step 2. This capability is provided by having a property allocated stack area in your program.
2. Having an interrupt service routine designed to render the service needed by the interrupt requesting device. This is the code executed in Step 5 above.
3. A means for the CPU locating the ISR corresponding to a particular request. This takes special meaning when the CPU can be interrupted by multiple devices, each having a different ISR. Usually this provision also deals with resolving any conflict that might develop if multiple devices place simultaneous requests to the CPU. This allows accomplishing Step 4 in the list above.
4. Enabling the interrupts in the system, otherwise they would not be served. By default, when the CPU is powered-up, interrupts are masked, so explicit software action is needed to unmask them. Enabling interrupts requires two levels of action: one at the CPU with the GIE flag, and another enabling the device(s) that will issue interrupt requests. This requirement will make possible to start the interrupt cycle at Step 1 in the sequence above.

Satisfying these four basic requirements will need additional details about how the CPU is designed to support interrupt processing. Chapter 7 provides a detailed discussion of the subject of interrupts.

3.9.3 What is Reset?

A *reset* is an asynchronous signal that when fed to an embedded system causes the CPU and most of the sequential peripherals to start from a predefined, known state. This state is specified in the device's user's guide or data sheet. In a CPU, the reset state allows for fetching the first instruction to be executed by the CPU after a power-up.

A reset occurs when power is applied for the first time to the system, or when some event that might compromise the integrity of the system occurs. These events include a power on reset (POR), Brown-out resets (BOR), and others.

A reset may also be generated by a hardware event, like for example the expiration of a watchdog timer, or through the assertion of the CPU reset pin, caused either by a user or an external hardware component.

A reset can be viewed as a special kind of interrupt. Upon a reset, the program counter (PC) is loaded with the address of the first instruction to be executed by the CPU, starting the fetch-decode-execute cycle. Section 6.5 in Chap. 6 provides a detailed discussion of how a reset is configured and programmed.

3.10 Notes on Program Design

Before leaving the general aspects in this chapter, let us discuss some directives on program design, especially considering some aspects pertaining embedded systems.

A program is a logical sequence of instructions, associated data values and compiler directives written down to carry out an algorithm to perform a specific task using a computer or embedded system hardware.

Programming should not be left to chance. The steps in creating a program include design and planning of the algorithm, documenting the different phases, encoding into a specific programming language using appropriate syntax rules, testing and debugging. Despite our efforts, many programs contain bugs and require testing and maintenance.

Bugs are mistakes that the programmer has introduced in the program, either of syntax or logical natures. *Testing* is done to prove the existence of bugs, while *debugging* consists in fixing the bugs. Poor programming practice not only has higher probability of producing more bugs, but also makes it almost impossible to debug without incurring in enormous costs.

Discipline in programming and documenting, from the first steps in design and planning to the last steps in devising tests and debugging procedures is absolutely necessary for embedded system designers. Consider that once the system is in work, it will have limited human interaction.

As an embedded system programmer, try to follow a minimum set of goals when programming:

Write the shortest possible program. A short program reduces use of resources, it is easier to maintain and modify, as well as to understand. If the planning yields a long program, try to break it in short identifiable subtasks (see top–down design below).

Evaluate shortcuts before implementing them. To write a short program never, never, try to introduce shortcuts or tricks which may seriously affect the other qualities of the program. It is better to sacrifice length than understanding.

Write programs easy to understand. Spend some time assigning resources, giving appropriate names to constants and variables, documenting the program, and so on. Make it a habit to follow similar structures for similar tasks whenever possible. The time spent in making the program understandable pays itself soon.

Write programs easy to modify. You will soon discover that different tasks can be achieved by simple changes in code when the design is clear. This easiness reflects in lower costs and faster development of future projects.

Document your program as you work on it. The importance of documentation can never be overemphasized. Without it, programs are difficult to read, modify, reuse and debug. Moreover, documenting after finishing the program usually yields a lower quality documentation and higher cost.

Create your own catalog of codes for common tasks. Hundreds of different projects require similar steps to be followed, subtasks to be accomplished and so on. For example, configuring the I/O ports and the peripherals, setting modes of operations are common to most projects. Having a catalog of subroutines, macros and particular codes will accelerate developments. Do not reinvent simple tested codes.

Program for lowest power consumption. Always try to consume the lowest as possible choosing the mode suitable for your application. Prefer interrupts to polling, consider putting to sleep the system or the unused components when possible, study power profiles if available, etc.

Know your hardware system. Don't forget that embedded systems require a good match between software and hardware. You have to program for the hardware and connections you have available. Hardware and program design very often goes in parallel.

Avoid using unimplemented bits of memory or registers. Future hardware revisions or enhancements can make your program useless if you do not keep this goal in mind. Also, moving modules to new memory locations have less or no effect on the correctness of the program.

If possible, use a single resource for a single purpose. For example, if you have decided to use register R15 as a counter, don't use it for other purpose unless you're forced to it by lack of resources. Plan for any multiple use in advance.

Of course one might look for more goals that a program must pursue. When solving one problem try to identify any new goal. This will certainly help in the future.

3.11 The TI MSP430 Microcontroller Family

Previous sections have introduced characteristics of the MPS430 family. The rest of the chapter is devoted to this particular set of microcontrollers.

The MSP430 family of microcontrollers developed by Texas Instruments targets the market of battery-powered, portable embedded applications. Since its beginning, it has been designed with an architecture optimized for low-power consumption.

The first members of the family were introduced in early 1992, with the series MSP430x3xx, targeted specifically at low-power metering equipment. Since then, new series have been added, with every new addition offering improved characteristics over its predecessor: the 1xx in 1996, the 4xx in 2002, the 2xx in 2005 and the 5xx/6xx in 2008 and 2009. The most recent family introduced in 2010 is the ferroelectric memory family, which incorporates the ferroelectric memory technology, making the full memory space both writable and non volatile. There are over one hundred different configurations for MSP430 MCUs. New members maintain downward software compatibility, keeping up with their original philosophy of being an architecture tailored for ultra-low-power embedded applications.

Aside from some early EPROM versions in the 3xx series and high volume mask ROM devices in the 4xx series, all family members are in-system programmable via a JTAG port or a built in bootstrap loader (BSL) using RS-232 or USB ports.

The 3xx and 1xx generations were limited to a 16 bit address space. Later generations include what is called '430X' instructions that allow a 20 bit address space. Most applications however work fine with the 64 K memory space, and we will work with this space. With the exception of a glance at the CPUX and the characteristics specific to the extended 20 bit address space, which are explained in Appendix D.

3.11.1 General Features

All MSP430 family members are developed around a 16-bit RISC CPU with a Von-Neumann architecture. The assortment of peripherals and features in each MSP430 device varies from one series to another, and within a series from one family member to another. Common features to devices include:

- Standard 16-bit Architecture: All devices share the same core 16-bit architecture and instruction set. The 20-bit CPUX registers are also based on this architecture.
- Different Ultra Low-power Operation modes: The devices can operate with nominal supply voltages from 1.8 to 5.5 V. The nominal operating current at 1 MHz ranges from 0.1 to 400 μA, depending on the supply voltage. The wake-up time from standby mode is 6 μS.
- Flexible and powerful processing capabilities: The programmer's model provides seven source-address modes and four destination-address modes with 27 core instructions. Extensive interrupt capability with prioritized, nested interrupts with

unlimited depth level. A large register file with RAM execution capability, table processing, and hex-to-decimal conversion modes.
- Extensive, memory-mapped peripheral set including: a 14-bit SAR A/D converter, multiple timers and PWM capability, slope A/D conversion (all devices); integrated USART, LCD driver, Watchdog Timer, multiple I/O lines with interrupt capability, programmable oscillator, 32-kHz crystal oscillator (all devices).
- Versatile device options include:
  - Masked ROM
  - OTP and EEPROM models with wide temperature range of applications
  - Models with ferroelectric memory
  - Up to 64 K addressing space for the MSP430 and 1M for the MSP430X
  - Memory mixes to support all types of applications.
- JTAG/debugger element, used for debugging embedded systems directly on the chip.

Low-power Philosophy

The MSP430 was designed since the beginning targeting low power consumption. This means not only using low power low voltage IC designs for the hardware, but also configuring the operation to different power modes, depending on application needs, and the possibility of turning off everything, including peripherals, while not in use. This feature is very useful for event driven designs, in particular for portable equipment.

The MSP430 can function in six different power modes, named Active, and the low power modes LPM0 to LPM4, where the CPU is off and can only be awaken by an interrupt or a reset. The modes are configurable via signals (bits) SCG1, SCG0, OSCOFF, and CPUOFF of the status register, which control the clock operations, as explained later in Chap. 7.

Oscillators and Clocks

The clock generator in MSP430 devices is designed to support low power consumption. It includes three or four clock sources, depending on the family and the specific member, that generate three internal clock signals for the best balance of performance and low power consumption. The clock sources include:

LFXT1CLK: A low-frequency/high-frequency oscillator that can be used with low-frequency watch crystals, namely external clock sources of 32,768-Hz, standard crystals, resonators, or with external clock sources up to 8 MHz.

XT2CLK: Optional high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 400-kHz to 16-MHz range.

DCOCLK: Internal digitally controlled oscillator (DCO).

VLOCLK: Internal very low power, low frequency oscillator with a 12 kHz typical frequency.

The three clock signals available from the basic clock are the auxiliary clock (ACLK), sub-main clock (SMCLK), and the master clock (MCLK) used for the feeding CPU the clock. They are software selectable to be fed from the available clock sources in the chip via software controlled configuration registers. Chapter 6 discusses in detail the MSP430 clock system.

Mixed Signal Product Philosophy

This philosophy is reflected in the very a acronym MSP. A vast number of applications fall in the category of signal processing, and nowadays most are of the mixed-signal type, requiring both analog and digital hardware. The analog components used in the analog signal-chain design include analog-to-digital converters (ADC), analog comparators, operational amplifiers, and so on.

Most MSP430 models include some of these analog signal-chain components allowing the designer to reduce power consumption, component count, and space. This mixed-signal design of the MSP430 series is another attractive feature for the user. Analog signal-chain components in the MSP430 are discussed in Chap. 7.

3.11.2 Naming Conventions

The MSP430 microcontrollers follow a more or less ordered sequence of fields in naming the parts. "MSP" stands for *Mixed Signal Processor*, emphasizing the fact that the analog and digital aspects of signal processing are included in the design and applications. The "430" refers to the MCU plattform.[14] The rest of the name is composed of several fields, some of which are optional. The name format is

MSP430{Device Memory}{Generation}{Family}{Series and Device Number}
... [A][Temperature Range][Packaging][Distribution Format][Additional Feature]

where fields between square brackets ([]) are optional. The definition of the fields are as follows:

Device Memory Which indicates the type of memory and specialized application:

Memory Type

- C = ROM
- F = Flash

[14] Some people say that the number stems from the date April 30, associated somehow to the development of the first prototype. We mention this reference without actually knowing its truthness.

- FR = FRAM
- G = Flash (value line)
- L = Non-volatile memory not included

Specialized Application

- FG = Flash Medical
- CG = ROM Medical
- FE = Flash Energy Metering
- FW = Flash Electronic Flow Meters
- AFE = Analog Front End
- BT = Pre-programmed with Bluetooth
- BQ = Contactless Power

Generation Refers to the series generation number, which does not necessarily refers to the order in which they were released.

- 1 = up to 8 MHz
- 2 = up to 16 MHz
- 3 = Legacy series, OTP
- 4 = up to 16 MHz with LCD
- 5 = up to 25 MHz
- 6 = up to 25 MHz with LCD
- 0 = Low Voltage Series

Family it refers to a family within the generation series type.
Series and Device Number within the given family
On the other hand, the optional fields are:
A, which means a revision model
Temperature Range, which comes with the following option:

- S = 0 °C to 50 °C
- I = −40 °C to 85 °C
- T = −40 °C to 105 °C

Packaging A group of three letters describing the type of packaging for the device, as defined in http://www.ti.com/sc/docs/psheets/type/type.html
Distribution Format

- T = Small 7-in reel
- R = Large 11-in reel
- No marking = Tube or tray

Additional Features:

- −QI = Automotive qualified
- −EP = Enhanced product (−40 °C to 125 °C)
- −HT = Extreme high temperature (−55°C to 150°C)

3.12 Development Tools

Several tools are available for working with the MSP430. Below we mention some of them as well as internet sites where they are available or where more information can be found.

3.12.1 Internet Sites

Some important internet URLs where we can find tools are

CCS http://processors.wiki.ti.com/index.php/Download_CCS
IAR http://supp.iar.com/Download/SW/?item=EW430-KS4
naken430 http://www.mikekohn.net/micro/naken430asm_msp430_assembler.php
pds-430 http://www.phyton.com/htdocs/tools_430/tools_mca430.shtml
cdk4msp http://cdk4msp.sourceforge.net/ver
mspgcc http://mspgcc.sourceforge.net/
mcc-430 http://www.mikekohn.net/micro/naken430asm_msp430_assembler.php

Some of these links include several of these features in the same package. Table 3.7 summarizes the availability of several tools described below.

3.12.2 Programming and Debugging Tools

Among the programming and debugging tools we have MSP430 simulators, C compilers and assemblers, linkers, and real time operating systems (RTOS). Simulators are of great help whenever a real system is not available or when it is not feasible to put to work dozens, hundreds, or even more systems at the same time. They also provide a tool for testing designs before actual loading it into the hardware system.

Several Real-Time operating systems have been developed around MSP430 systems. The RTOS' are good for systems with several tasks which are required to be

Table 3.7 Programming and debugging tools for MSP430

Site	Simulator	C compiler	Assembler	Linker
CCS	X	X	X	X
IAR	X	X	X	X
mspgcc		X	X	X
naken	X		X	X
pds-430	X	X	X	
cdk4msp	X			
mcc-430		X		

completed within a predetermined time line. The following is a non-exclusive list of sites with RTOS' for MSP430 systems:

TinyOS	http://www.tinyos.net/
FreeRTOS	http://www.freertos.org/index.html
	http://www.freertos.org/portmspgcc.html
PowerPac	http://www.iar.com/website1/1.0.1.0/964/1/
scmRTOS	http://scmrtos.sourceforge.net/ScmRTOS
ecos	http://www.ecoscentric.com/index.shtml

Additional information about programming and debugging tools can be found at http://processors.wiki.ti.com/index.php/.

3.12.3 Development Tools

TI application engineers as well as third parties have designed several development tools which are readily available for the beginner, as well as for the intermediate and experienced users. These tools consist of emulators, evaluation kits, and device programmers. There are also a number of available tools specific for a very handy MSP430 development kit known as the eZ430.

Emulators

An emulator is a machine acting as if it were another "emulated" machine, by running the code of the latter. Though the result is probably a slower "version" of the original machine, this allows the user to examine the system behavior in a more complete fashion as opposed to simulation. The CCS and IAR sites mentioned before provide emulators. Others can be found at

FET http://focus.ti.com/docs/toolsw/folders/print/msp-fet430uif.html
Project430 http://www.testech-elect.com/phyton/pice-msp430.htm

Evaluation Kits

Evaluation kits are relative small subsystems in a board which include a full featured specimen of an MCU (or other ICs). These kits typically expose several pins of the MCU that will allow external connections or the study of particular features.

The evaluation kits presented below typically reflects the complexity of the system and/or the availability or lack of additional components such as LCDs, speakers, battery connections, microphones, etc.

MSP430-EXP430F5438 Includes: Dot-matrix LCD, 3-axis accelerometer, microphone, audio output, USB communication, joystick, 2 buttons, 2 LEDs

MSP430-EXP430FG4618 Includes: LCD, capacitive sensitive input, audio output, buzzer, RS-232 communication, 2 buttons, 3 LED

MSP430-EXP430G2 [15] Includes: Integrated flash emulation tool, 14/20-pin DIP target socket, 2 buttons, 2 LEDs, and PCB connectors. It is compatible with several 14-pin and 20-pin models.

Device Programmers

In case you need to program several MSP430 MCUs at the same time there are at least two device programmers available, namely the MSPGANG430 and the GangPro430:

MSPGANG430 Connection type: Serial Programs 8 devices at a time. It works with PC or as standalone.

GangPro430 Connection type: USB Program 6 devices at a time via JTAG, Spy Bi-Wire, and BSL. Fast programming time.

Tools for eZ430

Probably the most handy of all the MSP430 tools available for development are the eZ430 tools. These tools can go for as low as $20.00 (eZ430-F2013) and could be very handy. Available tools include the EX430-RF2560, eZ430-Chronos, eZ430-F2013, and others.

3.12.4 Other Resources

One of the best source of information for this family of MCU's can be found at the official TI's page for the MSP430: www.msp430.com. Here the reader can find several resources such as application notes, code examples, open source libraries, white papers, guides and books, and MCU selection wizards.

3.13 Summary

- A microcomputer system includes a Central Processing Unit (CPU), Data and Program Memory blocks, and Peripheral or Input/Output (I/O) subsystem, interconnected by system buses. Another set of components provide the necessary power and timing synchronization for system operation.

[15] Price: $4.30 (Really!).

- The Data Bus is used to transfer instructions to, and data to and from the CPU. Through the Address Bus, the CPU selects the location with which interaction takes place. The Control Bus includes all signals that regulate the system activity.
- The CPU fetches instructions from the program memory, decodes them and executes them accordingly. Hence, the CPU performs the basic operations of data transfer as well as the arithmetical and logical operations with data.
- Memory is organized as a set of cells or locations, characterized by the contents, i.e. the memory word, and the address. The program memory is stored in the ROM section, and the data memory in the RAM section. Von Neumann architecture systems include the program and data memories in the same space; the Harvard architecture systems use separate memory spaces and buses for program and data.
- Data wider than memory words utilize two or more of these for storage. The address of a datum is that of the lower byte. In little endian convention the least significant byte is stored there, while in big endian convention the most significant byte goes to that byte.
- A memory map is a representation of the usage given to the addressable space of a microprocessor based system.
- The minimal hardware components of the CPU are: Arithmetic Logic Unit (ALU), Control Unit (CU), a set of Registers, and a Bus Interface Logic (BIL). They are connected by an internal system of buses.
- The CPU registers provide temporary storage for operands, addresses, and control information. Some specialized registers for specific use are the Instruction Register (IR), the Program Counter (PC), the Stack Pointer (SP), and the Status Register (SR).

3.14 Problems

3.1 Answer the following questions.

a. The main components of a microcomputer system are _____ .
b. The acronym CPU means _____ .
c. Which are the three system buses?
d. What is a bus?
e. What is a data bus transaction?
f. What is a read operation? What is a write operation?
g. How many data transactions are needed if the data consists of four bytes and the Data Bus width is 16 bits?
h. If the address bus has N lines, what is the maximum number of addresses that the CPU may reach?
i. What is the difference between physical address and data address?
j. What is the difference between Harvard and a Von Neummann architectures?
k. What is the difference between a microcontroller and a microprocessor?

l. If the address of the double word 2AF30456h is 0402h, what are the physical addresses of the individual bytes in a little endian memory system?
m. Repeat the previous question for a big endian system.

3.2 Figure 3.6 shows an example of a memory structure for a memory space of 64 KB, assuming that the address bus has 16-bits, so each address word will point to the respective cell in this space. Assume now a 20-bit address bus and an 8-bit data bus, together with several 64 KB memory modules similar to the one in the figure, with each module being independently activated or deactivated. The modules can all share 16 lines of the address bus, say A15 to A0, but only the activated module would be accessed by the data bus. The four additional lines A19 to A16 of the address bus could then be used to activate the appropriate module.

a. What is the maximum number of modules that can be used?
b. If all the possible memory modules were used, what memory size would be available?
c. Assume that only four memory modules are used. What is the effective size of the address space being used and what happens when the address words bits go from 0000xxx...xxB to 1111xxx...xxB?
d. It is said that each memory cell has a unique address. Considering the above item, is this an absolute truth? That is, is it possible for a memory cell to have more than one physical address? Justify your answer.

3.3 Figure 3.12 shows two memory banks being used for connection with a 16-bit data bus, one bank set for the even numbered addresses and the other for the odd numbered addresses.

a. What would happen if the least significant address line from the microprocessor is connected to the least significant address line in both banks?
b. How would you solve the problem presented in (a)?

3.4 The following data was obtained from the program memory of a particular microprocessor:

4031 0280 40B2 5A80 0120 D3D2 C022

E3D2 0021 403F 5000 831F 23FE 3FF9

Assuming the first address is 0xF800, could you order the data by physical addresses using the little endian convention? Repeat for the big endian convention.

3.5 Design the following decoders a. 2 to 4 b. 3 to 8 c. 4 to 16

3.6 If a particular microprocessor has only one data bus but has an area designated for program memory and another area designated for data memory, is it a Von Neumann or a Harvard architecture?

3.7 Show how you would connect eight 1 KB memory modules using a 3 to 8 decoder with a chip enable terminal. Assume each memory module has ten

input address lines, eight data lines, and one chip enable terminal. Also assume the address bus coming from the microprocessor is a 16-bit address bus.

3.8 Consider a 1024B memory bank composed of two 512B memory modules: one for the low byte and one for the high byte. A signal from the microprocessor, R/W' is used to distinguish between a memory read and a memory write. The address bus from the microprocessor is 16-bit wide.

a. Determine the address lines that will be connected to each of the memory modules.
b. Determine the address lines that should be used to enable the chip select, an active low input signal for each of the memory modules.
c. Design the decoder needed to enable the memory modules if the memory space for these modules begin at address 512.

3.9 Answer the following questions.

a. What is the characteristic of an I/O memory mapped system?
b. What other scheme exists beside the I/O memory mapped system for dealing with the I/O subsystem?
c. Name four common peripherals to be found in most embedded systems.
d. What is the main function of the watchdog timer?
e. What is the difference for the CPU between an interface register with an I/O device and a memory location?

3.10 The following expressions are given in register transfer notation. All registers are 16-bit wide and memory addresses point to 16-bit word data. Before each expression, it is assumed that register and memory contents are as follows: R8 = 4286h; R9 = 32F4h; [4286h] = 3AC5h; [027Ch] = 90EEh. Complete the following table filling up the column of results. Write your results using hex notation.

Transaction	Result
R9 ← R8	
R9 ← (R8)	
(R8) ← R9	
R9 ← R9 + R8	
(027Ch) ← R8	
(R8) ← (R8) + 013Fh	
R8 ← 2468h	
(037Ch) ← (027Ch)–(4286h)	