Chapter 4
Assembly Language Programming

Programming provides us with the flexibility to use the same hardware core for different applications. A device whose performance can be modified through some sort of instructions is called a *programmable* or *configurable device*. Most I/O ports and peripherals are examples of these blocks; their configuration should be part of our program design.

Assembly programming for the MSP430 microcontrollers is introduced in this chapter. We limit ourselves to the CPU case; CPUX instructions are briefly discussed in Appendix D. The assembly language instructions are particular to the MSP430 family, but the techniques and principles apply to all microcontroller families with the appropriate adaptations. The material should enable you to proceed to more advanced topics on your own by consulting manuals, examples available elsewhere[1] and, very important, by practicing programming.

The two most popular assemblers for the MSP430 are the IAR assembler by Softbaugh and Code Composer Studio (CCS) from Texas Instruments. They both have free limited versions to download from the Internet, and also come included in evaluation boards. In this text we use the IAR assembler since its directives and structure are more along line of most assemblers. It also has no limit in the memory size for assembly language code. Appendix C briefly discusses the CCS assembler.

4.1 An Overview of Programming Levels

Programs are written using a *programming language* with specific rules of syntax.These languages are found in three main levels:

a) *Machine language*,
b) *Assembly language*, and
c) *Level language*

[1] Texas Instruments has many code examples available for MSP430 at http://msp430.com.

M. Jiménez et al., *Introduction to Embedded Systems*,
DOI: 10.1007/978-1-4614-3143-5_4,

To illustrate the differences among them, let us consider the code of Laboratory 1[2] in the three levels. The program toggles on and off an LED connected to pin 0 of port 1 (P1.0) of the microcontroller, as illustrated in Fig. 4.1. Toggling is achieved by changing the voltage level at the pin.

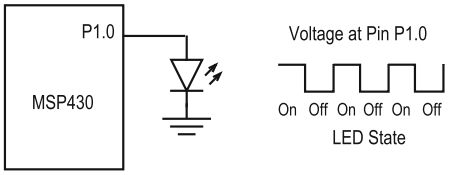


Fig. 4.1 Simplified hardware connection diagram for LED in board

The algorithm to achieve this goal is described by the flowcharts of Fig. 4.2, where (a) illustrates the process in its general terms, and (b) expands the different steps in more detailed actions.

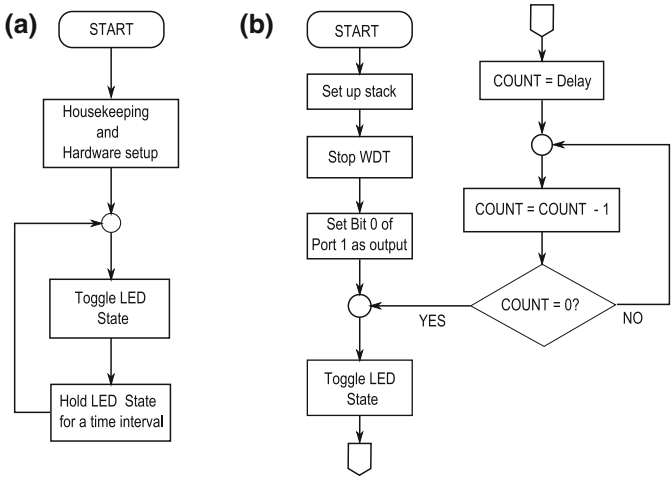


Fig. 4.2 Flow diagram for the blinking LED: **a** General Concept; **b** Expanded flow diagram

4.1.1 High-Level Code

Figure 4.3 shows a code in C language which executes the desired task. Readers with some proficiency in C will understand this *source code* without problem and associate it with the flow graph. The compiler takes care of the "set up stack" step.

[2] This hands on exercise is adapted from TI with permission

Listing 4.1: C Language Listing

```
#include  <msp430x12x.h>
void main(void)
{ WDTCTL = WDTPW + WDTHOLD; /* Stop watchdog timer */
  P1DIR |= 0x01;      /* Set P1.0 to output direction */
  for (;;)
  { unsigned int i;
    P1OUT ^= 0x01;    /* Toggle P1.0 using ex-or */
    i = 50000;        /* Delay */
    do (i--);
    while (i != 0);
  }
}
```

Fig. 4.3 C language code for LED toggling

Some differences with respect to C program sources for general purpose computers are noticeable. For example, hardware characteristics now require some attention. Despite this, high-level programming is more independent of the hardware structure, closer to our way of thinking, and easier to grasp. These are some of many reasons why this level is preferred for most applications. Chapter 5 introduces C programming for the MSP430.

A drawback is that this language is not understood by the machine. We need tools, called *compilers* and *interpreters*, to make the appropriate translation. The final machine product depends on the compiler used. Engineers also developed software tools called *debuggers* to help correct programming errors, usually called *bugs*; correcting them is called *debugging*.

4.1.2 Machine Language

Since digital systems only "understand" zeros (0) and ones (1), this is how the *executable code*, i.e., the set of instructions executed by the CPU, is presented to the system. Although possible, it is difficult to associate the machine code to an original high level source.

A program written in zeros and ones is a *binary machine language* program. The example in Listing 1.2 of Fig. 4.4 is for the flow graph of Fig. 4.2. Here, each line represents an instruction. The hex notation version in Listing 1.3, called *hex machine language*, is the form in which debuggers present machine language to users and also the syntax with which embedded programmers deal with this level. Obviously, machine language is not user friendly and hence the need of other programming levels.

Listing 4.2: Binary Machine Language

```
1  0100000000110001 0000001100000000
2  0100000010110010 0101101010000000 0000000100100000
3  1101001111010010 0000000000100010
4  1110001111010010 0000000000100001
5  0100000000111111 1100001101010000
6  1000001100011111
7  0010001111111110
8  0011111111111001
```

Listing 4.3: Hex

```
4031 0300
40B2 5A80 0120
D3D2 0022
E3D2 0021
403F C350
831F
23FE
3FF9
```

Fig. 4.4 Executable machine language code for the example of Fig. 4.1

Listing 4.4: Assembly Version

```
1        mov.w   #0x300,SP
2        mov.w   #0x5A80,&0x0120
3        bis.b   #001,&0x0022
4        xor.b   #001,&0x0021
5        mov.w   #0xC350,R15
6        dec.w   R15
7        jnz     0x3FC
8        jmp     0x3F2
```

Hex Machine Lang.

```
4031 0300
40B2 5A80 0120
D3D2 0022
E3D2 0021
403F C350
831F
23FE
3FF9
```

Fig. 4.5 Assembly version for Fig. 4.4—Hex machine version shown for comparison

Instructions for the MSP430 CPU consist of one, two or three 16-bit words. The leading one is the *instruction word*, with information relative to the instruction and operands. Machine language for the MSP430CPU is considered in Appendix B.

4.1.3 Assembly Language Code

The first step toward a friendlier syntax was the *assembly language*, compiled with *assemblers*. Each instruction is now a machine instruction encoded in a more "human like" form. Translating a machine instruction into its assembly form is to *disassembly* and the software tool for this task is a *disassembler*. Figure 4.5 shows the assembly and hex versions of the executable code. Lines in both listings correspond one to one. Thus, "`mov.w #0x300,SP`" is the assembly for hex "4031 0300".

Basic MSP430 Assembly Instruction Format

As illustrated, the very basic format of MSP430 instructions contains two components: the *mnemonic* and the *operands*. The mnemonic is a name given to the machine language opcode, and it is by extension sometimes also called opcode.[3] Mnemonics

[3] Strictly speaking, the opcode is part of the machine language code only and the mnemonic is the assembly language name for the opcode.

have a suffix ".b" or ".w" to differentiate byte and word size operands, as explained later in this chapter. No suffix is by default equivalent to ".w".

Operands are written in an appropriate MSP430 addressing mode, as introduced in Chap. 3. Operands are usually called *source* (src) and *destination* (dest). The instruction may have two, one or no operands using one of the format of expressions (4.1)–(4.3). The only core instruction without operand is `reti`, "Return from Interrupt"; all the other zero operand instructions are emulated. The operand in (4.2) may be a source or a destination, depending on the instruction.

Mnemonic *src*, *dest* (4.1)

Mnemonic *operand* (4.2)

Mnemonic (4.3)

4.2 Assembly Programming: First Pass

Although simpler to read than machine form, the code of Fig. 4.5, which is a pure assembly listing, still has unfriendly notes. For example, the user needs the memory map to identify addresses 0x0022 and 0x0021, and knowledge of machine instruction lengths to know how many bytes the PC should jump. A friendlier version is found in Fig. 4.6, written with IAR assembler syntax. This is part of a complete *source file*. Each line in the source file is a *source statement*.

Listing 4.5: Assembly Code

```
1  ;Constants Declarations
2  #include  "msp430g2231.h"
3  LED       EQU    0x0001    ; LED at P1.0
4  DELAY     EQU    50000     ;
5  #define   COUNTER  R15
6  ;- - - - - - - - - - - - - - - - - - - - - -
7                ORG 0F800h       ;Start Code
8  ;- - - - - - - - - - - - - - - - - - - - - -
9  RESET     mov.w   #300h, SP       ; Set stack
10 StopWDT   mov.w   #WDTPW+WDTHOLD,&WDTCTL
11                                   ; Stop WDT
12 SetupP1   bis.b   #001h,&P1DIR  ;P1.0 output
13    ;
14 Mainloop  xor.b   #LED,&P1OUT  ; Toggle P1.0
15 Wait      mov.w   #DELAY,COUNTER
16                        ;Load Delay to Counter
17 L1        dec.w   COUNTER          ; wait
18           jnz     L1          ; Delay over?
19           jmp     Mainloop         ; Again
```

Hex Code

```
4031 0300
40B2 5A80 0120

D3D2 0022

E3D2 0021
403F C350

831F
23FE
3FF9
```

Fig. 4.6 Assembly language code for Fig. 4.4

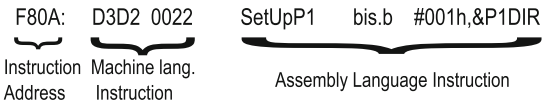


Fig. 4.7 Example of a list file line

Now assembly instructions have the format illustrated by (4.4); only the original instruction fields, mnemonics and operands, are compulsory. Mnemonics cannot start on the first column.

$$\underbrace{\text{SetupP1}}_{\text{Label}} \quad \underbrace{\text{bis.b}}_{\text{Mnemonics}} \quad \underbrace{\text{\#001h, \&P1DIR}}_{\text{Operands}} \quad \underbrace{\text{; P1.0 Output}}_{\text{Comment}} \tag{4.4}$$

In fact, every source statement has the same format, all fields being in general optional. The mnemonics may be a machine instruction mnemonics or a directive. Operands must always go after a mnemonic.

Observe that the machine language and instruction mnemonics have not changed. But now there are new features in the listing that make it easier to read. Namely, we find *comments*, *labels*, *symbolic names*, and *directives*.

On the first column we can only have a comment (starting with the semicolon), a label, a C-type preprocessor directive, or a blank.

Assembling is done with a *two pass assembler*, which runs over the source code more than once to resolve constants and labels. The assembler receives the source file and outputs an *object file*, as well as other optional files. In particular, the list file is very helpful. For each source file statement which generates code or memory contents, the list file shows the address and contents. In the case of an instruction, this contents is the machine language instruction in hex notation, word size data. An example of a line of a list file is illustrated by Fig. 4.7.

4.2.1 Directives

Directives are for the assembler only. They do not translate into machine code or data to be loaded into the microcontroller memory. They serve to organize the program, create labels and symbolic names, and so on. Directives depend on the assembler, but instructions belong to the CPU architecture and are independent of the assembler. We will work with the IAR assembler.[4]

Comments: Anything on a line after and including a semicolon (;) is a comment. It is ignored by the assembler and serves as a documentation tool and to explain the purpose of an instruction. Remark how comments relate the code with the flow graph of Fig. 4.2b.

[4] For a more in depth treatment than the introductory level provided in this book, the reader may consult the document "MSP430 IAR: Reference Guide for Texas Instruments' MSP430 Microcontroller Family", published by IAR Systems Inc. and downloadable from the Internet.

Symbolic Names: These are names given to numbers, expressions, hardware devices, and so on. Labels fall in this category. The values and equivalences for these names are assigned during assembly and are thus called *assembly time constants*. The names can be used by the programmer as if they were already known before assembling.
Labels: These always start on the first column. They may contain alphanumeric characters, the underscore "_", and the dollar sign "$"; they may end with a colon (:) but never start with a number. The nature of this constant value will depend on the directive used with the label or if it associated to an instruction. In the latter case, the label takes the value of the memory address of the instruction word.
Directives EQU, #define, #include and ORG: The code example of Fig. 4.6 introduces some directives. IAR directives are usually written with all-caps letters. Those like "#define", "#include", and others are C-type pre-processor directives, so called because they come from C language and maintain its syntax. The directives included in the example are the following:

- EQU and #define which are used to define assembly time constants and some symbolic names, valid in the module, for numeric values, hardware pieces, and so on. EQU cannot be used with registers or hardware devices.
- #include, which serves to append external files. In this case, and almost always done, a header file where more assembly time constants are defined.
- ORG will tell the assembler the address in memory where the items following the directive will be stored.

The formats for EQU and #define are

```
LABEL  EQU <<Value or expression>>
```

and

```
#define <<Symbolic Name>> Value or expression or register
```

EQU is an example of a value assignment directive. Another one is SET. For #include, the format is

```
#include "filename"
```

MSP430 header files contain definitions for constants to be used. Since all MSP430 microcontrollers have the same addresses for similar peripherals and ports, related constants are available for MSP430 programmers in the *header files*. Therefore, this is a common line in IAR source files, with filename usually of the type *msp430xxxx.h*, where the xxxx field corresponds to either a model or a family. In the example of Fig. 4.6 we see `#include "msp430g3221.h"`. More recent IAR versions allow *msp430.h* only, and the selection is made by the assembler among the different files available in a library. All the symbolic names that appear in the listing of Fig. 4.6 which were not defined explicitly with EQU or #define, are defined in the header file.

The format for the ORG directive is

```
ORG expr
```

where `expr` is the physical address where the program location counter will point to store the next items going to memory. The ORG directive requires the user to know the memory map, or at least the address of interest, of the microcontroller being used.

Example 4.1 *Even if the source file is not yet complete, let us look at what happens when the assembly code of Fig. 4.6 is processed.*

Labels DELAY *and* LED*, as well as the symbolic name* COUNTER *are identified as* 50000 (=C350h)*,* 1 (=01h)*, and* R15*, respectively. All other symbolic names found, such as* P1DIR *and* P1OUT*, are defined in the header file. For other labels, the value will be generated, and the* `ORG` *directive allows us to understand the process.*

`ORG 0F800h` *is used just before the beginning of the source code. Therefore the first machine language instruction* 4031 0300 *following the directive goes to that address,* 0F800h*. From there on, every instruction will be stored continuously one after the other. The result can be appreciated with the list file, as partially illustrated below:*

```
                        ORG     0F800h                  ;
F800 4031 0300          RESET   mov.w   #300h,SP         ; Set stack
F804 40B2 5A80 0120     StopWDT mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
F80A D3D2 0022                  bis.b   #LED,&P1DIR     ; P1.0 output
F80E------
      --                  - - - - -
                            - -
```

The assembly listing has the label RESET *for this first instruction, so* RESET *becomes* 0F800h*. Similarly, the label* StopWDT *takes the value* F804h*, which is the address where the instruction is stored. Every label attached to an instruction will be then assigned the value of the corresponding instruction address.*

Using directive `ORG` requires from the user knowledge of the MCU memory map. In particular, the programmer needs to be aware where the executable code usually goes, whose starting address is machine dependent. RAM memory, used for variables and other uses, most often starts at 0200h. Other places need also to be known. Again, some names are defined in the header file to help the programmer in this task.

Standard Symbolic Constants and Header Files

Even though the assembly instructions work only with integer constants, the definition of symbolic constants is for all practical matters “a must” in programming, and very important in assembly programming. To simplify readability and comprehension, as well as portability, symbolic names for the addresses of the peripheral registers, and for bits or groups of bits within registers are standard. Texas Instruments has generated *header files* files with these constant names. These are written in C language code, but shared by C compilers and assemblers. In the IAR assembler, the files are included in the source code with the directive `#include`.

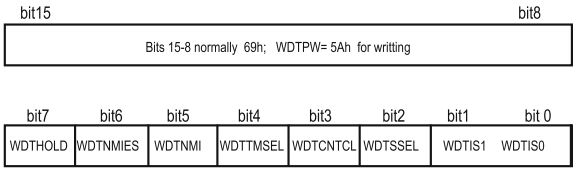


Fig. 4.8 Information for watchdog timer control register (WDTCTL) (*Courtesy of Texas Instruments, Inc.*)

Fortunately for us, the user guides and data sheets for the different series and models use these symbolic names when explaining the peripheral registers, so the programmer does not need to know the details of the header files. But when necessary, these files are usually available in the IDE.[5]

Let us illustrate the naming conventions using the Watchdog Timer control register, presented in the user guides as shown in Fig. 4.8. The address is also given in the user guide. The meanings of the bit names are summarized in Table 4.1.

Names associated to single bits are usually declared indicating the action asserted when the bit is set (as in WTDHOLD for Watchdog Timer Hold) when this is possible, or to the type of control that it is exerted, as in WDTNMIES for Watchdog Timer Non Maskable Interrupt Edge Selection (0 for falling edge, 1 for rising edge).

With these definitions, "`mov #WDTPW+WDTHOLD,&WDTCTL`" is equivalent to "`mov #0x5A80,&0x0120`", and is used to stop (hold) the WDT operation. Remember that the first byte needs to be 5Ah in order to make changes to this register. Therefore, in the particular case of this register, every change must include WDTPWD in the immediate source operand.

For a group of bits, referred to as a set with a symbolic name ending as x, there are two general conventions. It is up to the reader to verify if both are used in the header file or only one of them. One is when x is substituted with the format "_N", then N represents the decimal value in the combination of the respective bits.

Table 4.1 Symbolic constants associated to watchdog timer control register (WDTCTL)

Name	Number	Comment
WTDCTL	0x0120	Register address
WTDPW	0x5A00	Required to make changes
WTDHOLD	0x0080	When set, stops WDT
WDTNMIES	0x0040	selects the interrupt edge for the NMI interrupt when WDTNMI = 1
WDTNMI	0x0020	Selects pin $\overline{RST}$/NMI function: 0 for Reset, 1 for NMI
WDTTMSEL	0x0010	WDT working mode: 0 for WDT, 1 for interval timer
WDTCNTCL	0x0008	Resets or clears the counter when set
WDTSSEL	0x0004	WDT clock source select: 0 for SMCLK, 1 for ACLK
WDTISx	(bits 1, 0)	WDT interval select. (Explanation below)

[5] Unfortunately, now and then TI edits the files and changes names. If one of the errors shown during assembly indicates unknown definition, first check spelling. If this is not the source of error, check for an updated header file.

Table 4.2 Symbolic constants associated to watchdog timer control register (WDTCTL)

Name	b1-b0 value	Using bits	Time interval
WTDIS_0	0x00	None	Watchdog clock source/32768
WTDIS_1	0x01	WTDIS0	Watchdog clock source/8192
WTDIS_2	0x02	WTDIS1	Watchdog clock source/512
WTDIS_3	0x03	WTDIS1+WTDIS0	Watchdog clock source/64

In the other convention, x may be substituted by the bit number generating as many names as bits are in the group. Both conventions are illustrated for the group WDTISx in Table 4.2 in the first format. Remember that by default, the bits are 0.

The numbers 32768, 8192, 512, and 64 are associated to a frequency of 32.768 kHz for a crystal resonator. With this clock frequency, the time intervals are of 1000, 250, 16 and 1.9 ms. For other frequencies, these intervals will be different.

With these definitions,

```
mov #WDTPW+WDTTMSEL+WDTCNTCL+WDTSSEL+WDTIS0,&WDTCTL
```

is equivalent to `mov #561Dh,&0120h` and means: "Use the WDT as a normal timer, with the ACLK as its source and reseting every f_{ACLK}/8192 s". In particular, if we are using a watch crystal for the frequency source, the instruction is for using the WDT as a 250 ms interval timer. Every 250 ms, the interrupt flag is set. To avoid long operands, the complete sum is defined as another constant, "WDT_ADLY_250".

The naming principle illustrated is similarly applied to all registers, addresses for interrupt vectors, or even individual bits and other important combinations. The reader should look at different source examples and user guides to grasp the general idea.

4.2.2 Labels in Instructions

The above example gives more insight into labels. Although labels are optional, a good assembly programming practice follows some unwritten rules. Always use a label for

- Entry statement of the main code and of an Interrupt Service Routine (ISR). The label takes the value of the reset vector or interrupt vector.
- Entry statement of a subroutine. The label can be used to call the subroutine using it in immediate addressing mode in the **call** instruction, for example `call #Label`.
- Instruction to which reference is made, for example for a jump.

Labels are also useful as highlights for instructions, even if no reference is needed. Take a look at the listing in Fig. 4.6.

Example 4.2 *Continuing with the same listing used in example 4.1, the reader can verify that the label* `Mainloop` *will be equal to* 0xF80E. *The label is used for jump instructions and can be used as any other integer constant with addressing modes as illustrated next.*
`mov.w #Mainloop,R6` *(immediate mode for Mainloop) yields* R6 = F80Eh.
`mov.w Mainloop,R6` *(direct or symbolic mode for Mainloop) yields* R6 = 403F.
`call #Mainloop` *(immediate mode for Mainloop) calls a subroutine with entry line at address* 0xF80E. *Hence, the CPU pushes the PC onto the top of the stack (TOS) and then loads it with* 0xF80E.
`call Mainloop` *(direct or symbolic mode for Mainloop) will push the PC onto the TOS and load it with* 0x403F. *In this particular example, there will be an error since the values at PC must be even. The CPU will reset.*

On the other hand, operands in jump instructions work differently depending on whether they are in a source being compiled or in an interpreter or single line assembler. Labels only apply to compilation with two-pass assemblers. Now, only the address of the target instruction is valid, where the address is given explicitly either as a label or a number, as illustrated in (a) and (b) below. Other modes are illegal, as illustrated in (c). Notice that the syntax is the same as a direct one but does not have the same meaning.

(a) Instruction `jmp Mainloop` *will load the PC with* F80E, *making* `mov.w #50000,R15` *the next instruction to fetch.*
(b) Instruction `jmp 0xF80E` *has the same effect but requires the user to know the address of the target instruction.*
(c) The instructions `jmp #Mainloop` *or* `jmp & Mainloop` *are illegal.*

4.2.3 Source Executable Code

The executable code is the set of CPU executable source statements, namely, the instructions. It is the most important part of the source file. It consists of the main code and, optionally, subroutines and Interrupt Service Routines. Important characteristics to consider are

1. *Label for Entry line*: Names like “RESET”, “INIT”, “START” are common, but any one will do. This label serves to load the reset vector.
2. *Housekeeping and peripherals configuration*: This task includes
   - stack initialization,
   - configuration of the watchdog timer,
   - configuration of peripherals and I/O ports as needed
3. *Main routine or algorithms*: The instructions for the intended program.

Unlike programs written to be run on computers, microcontrollers programs usually do not have a stop or ending instruction. Here, they are terminated with either an

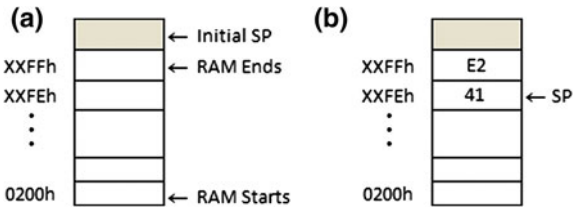


Fig. 4.9 Initializing SP. **a** At address just after RAM, **b** First pushed word occupies top of RAM

infinite loop or with the CPU off, waiting for an interrupt. Instruction `jmp $`, repeats itself indefinitely. This is useful for debugging and for interrupt driven designs.

Housekeeping and configuration: In the stack definition, the SP register is usually initialized with the *next* even value after the RAM, so the first pushed item occupies the highest word address in RAM, as illustrated in Fig. 4.9. RAM always begins at location 0200h in MSP430 microcontrollers, and its last address depends on the model. Thus, in the absolute code, for a 128 byte RAM the last address will be 0x027F so SP is initialized with 0x0280.

Strictly speaking, if the program does not include subroutines or interrupts, and neither includes push or pop instruction, then the stack pointer needs not to be initialized. Our example belongs to this kind of programs. However, it is a good practice to always include this initialization.

If the main code ends with an infinite loop, the Watch Dog Timer (WDT) should be stopped or else it will restart the CPU after some time. Hence, it is common to stop the WDT. This is usually done with the statement

```
mov.w #WDTPW+WDTHOLD,&WDTCTL                    (4.5)
```

as illustrated in our examples. It is recommended to stop the WDT whenever we have relative long loops in the code. Other WDT configurations are possible.

Finally, an embedded designer must be aware of how hardware connections interact with the CPU. This interaction takes place with internal peripherals and external hardware through I/O ports. Hence the need to configure ports and peripherals. The LED example given so far configures pin P1.0 of port 1 as an output, by setting bit 0 of the byte sized register named P1DIR.

Although a more detailed discussion on the peripherals and ports is provided in Chap. 7, there is an important point to consider with respect to ports. By default, port pins are selected as input pins at power up. Hence, if the pin is being used as input, there is no need to explicitly configure it at the beginning. Yet, if it is not being used, be sure the pin is not a floating input, or several problems can arise due to false signals and noise. Either connect a pull-up or pull-down resistor as illustrated in Fig. 4.10; some models have internal software configurable resistors. The resistor values are usually in the 30–56 kΩ. This practice should always be used when using input devices such as switches or pushbuttons.

If not using resistors, configure the pin as an output. In short, it is a good practice to *configure unused port pins as outputs to avoid hardware hazards*. In an example

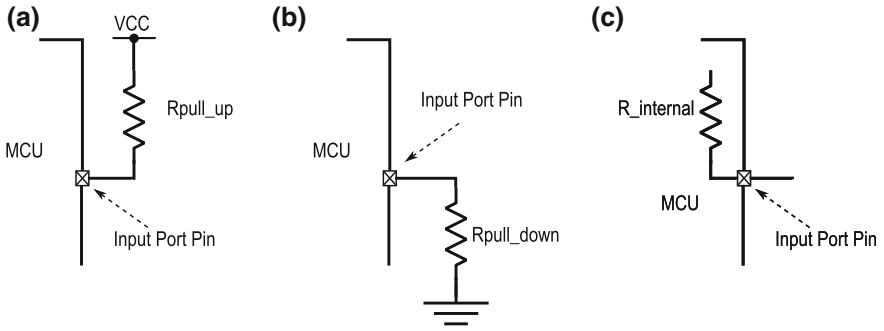


Fig. 4.10 Input pins with **a** pull-up resistor, **b** pull-down resistor and **c** internal software configurable resistor

like the one used here, it would have been better to use `mov #0xFF, P1DIR`, since no pin is used as input.

Main Algorithm: After housekeeping and hardware configuration, we may start writing the instructions for the intended task. If interrupts are being enabled, the `eint` should be introduced after the hardware configuration, to avoid unwanted interruptions.

Since most, if not all, embedded systems work continuously and independently of a user intervention, many systems do not include a particular instruction to "stop" the program. MSP430 microcontrollers fall in this category. Instead, two common methods to "terminate" the main code are an infinite loop using a unconditional jump or setting the system in a proper low power mode, also turning the CPU off.

4.2.4 Source File: First Pass

Let us end this discussion about the assembly process with a first look at the layout of the full source file that includes the code considered up to now. Remember that the source file is the one that the programmer writes, with all necessary comments, instructions, directives, etc. to feed to the assembler. Figure 4.11 shows an *absolute source file* for our example. It is called absolute because all the memory allocations are given explicitly by the programmer with the directive `ORG`.

We recognize in this figure the part we have been working with. Namely, constant declarations and executable code. In addition we have a *documentation*, and the *reset vector allocation*. This last group must be in any assembly source file.

This example introduces two new directives: `END` and `DW`, equivalent to `DC16`.

`END` directive is compulsory to finish the source file, since the assembler does not read anything after this directive.

`DW` stands for "define word", and it is equivalent to `DC16`, which stands for "define 16-bit word". This directive tells the assembler to store in memory the 16-bit words,

```
1  ;*******************************************************;
2     MSP430G2xx1 Demo - Software Toggle P1.0
3  ;
4  ;Description: Toggle P1.0 by xor'ing P1.0 inside
5  ;   of a software loop.
6  ;     ACLK = n/a, MCLK = SMCLK = default DCO
7  ;
8  ;                MSP430G2xx1
9  ;             -----------------
10 ;         /|\|               XIN|-
11 ;          | |                  |
12 ;          --|RST          XOUT|-
13 ;            |                  |
14 ;            |             P1.0|-->LED
15 ;
16 ;   Based on code written by D. Dang
17 ;   Texas Instruments Inc.
18 ;   October 2010
19 ;   Built with IAR Embedded Workbench Version: 5.10
20 ;*******************************************************
21 #include  "msp430g2231.h"                ; standard constants
22 LED         EQU    01h                  ;LED at pin P1.0
23 DELAY       EQU    50000
24 #define     COUNTER R15                 ; R15 as counter
25 ;- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
26            ORG     0F800h               ; Program Reset
27 ;- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
28 RESET       mov.w   #0300h,SP          ; Initialize stack
29 StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT
30 SetupP1     bis.b   #001h,&P1DIR         ; P1.0 output
31                                          ;
32 Mainloop    xor.b   #LED,&P1OUT          ; Toggle LED
33 Wait        mov.w   #DELAY,COUNTER       ; Delay to counter
34 L1          dec.w   COUNTER              ; Decrement counter
35            jnz     L1                   ; Delay over?
36            jmp     Mainloop             ; Again
37                                         ;
38 ;- - - - - - - - - - - - - - - - - - - - - -;
39            Interrupt Vectors
40 ;- - - - - - - - - - - - - - - - - - - - - - -;
41            ORG     0FFFEh               ; Address for
42            DW      RESET                ; RESET Vector
43            END
```

Documentation

Constants Declaration

Absolute directive

Executable Code

Reset vector allocation

Fig. 4.11 An absolute IAR listing for blinking LED (*Courtesy Texas Instruments Inc.*)

separated by a comma, that follow the directive. The storage starts at the memory address at which the assembler's *program location counter* (PLC) is pointing at that moment. It is convenient to have it in this case pointing at an even address.

Reset Vector Allocation: It is absolutely necessary to allocate the reset vector at address 0xFFFE. Remember that the reset vector is the address of the first instruction to be fetched. When the MCU is powered up or reset, the control unit makes the PC register be loaded with the word found at the mentioned address. We tell the assembler to allocate the reset vector with the lines.

```
ORG  0xFFFE
DW <<ResetVectorLabel>>
```

In our example, and very often, the label used is "RESET", but any valid name will do. Header files usually contain the definition of `RESET_VECTOR` as 0xFFFE, so we can use `ORG RESET_VECTOR` if this is the case. We will be back at this topic later.

4.2.5 Why Assembly?

High level languages increase productivity and are easier for working with complex algorithms. Why then should we bother studying assembly language?[6]

Assembly instructions are in an isomorphic relation with machine language instructions and thus intimately related to the embedded system. This provides many advantages among which we have: lower power consumption, the least memory usage and fastest operation, as well as lower costs and easier maintenance, desirable features in embedded applications, in particular portable ones. Hence, assembly language is a good choice for short to medium programs, where complexity is still manageable and we can keep good control on the machine performance. Typical uses include device drivers, and low-level embedded and real-time systems.

Another advantage appears when debugging, since programs are always stored in machine language form, no matter the original source program. Access to the system memory through debuggers provides invaluable room for maintenance and improvement. For these reasons, assembly language is a good choice for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues.

High level code cross compilers work by first translating the code into an assembly language one, giving us the opportunity to optimize the ultimate executable code applying our assembly language skills. For compilers not going through the assembly translation, we always have the possibility of disassembling the machine language to obtain the assembly version.

On the down side, vocabulary in assembly language is very limited and thus the program generally needs more lines, yielding slower code writing and higher costs in development. Some assemblers include high level features to mitigate this disadvantage, in particular for standard assembly code formats. Anyway, we should be clear that for complex algorithms and large codes, high–level languages are definitely preferred.

To close this discussion, let us point out that in embedded systems it is very often highly desirable to mix high-level and assembly instructions. This feature is very useful when we need to use the microcontroller with precise timing and special instruction sequences.

4.3 Assembly Instruction Characteristics

Using the source in Fig. 4.11 as an initial template, we are now in a position to look at the MSP430 instructions using the assembler. Of course, any other assembler or interpreter can be used to test our understanding of the instructions and partial listings.

---

[6] An excellent and more in depth discussion about this topic can be found at http://en.wikipedia.org/wiki/Assembly_language. We encourage the reader to go to this site and also look several of the references listed therein.

Table 4.3 MSP430 addressing modes

Addressing mode	Syntax	Comment
Immediate mode	#X	Data is X
Register Mode	Rn	Data is Register Rn Contents
Data in Memory:		
Indexed mode	X(Rn)	Address is X + Rn contents
Indirect mode	@Rn	Address is Register Rn contents
Indirect autoincrement	@Rn+	Address is Register Rn contents as before. Now, Rn is incremented after execution by 2 for word instructions and by 1 for byte ones.
Direct mode	X	Address is X
Absolute mode	&X	Address is X

Addressing Modes:

Remember that operands in an instructions must be written with an appropriate addressing mode syntax. For instructions other than jumps, the seven modes used by the MSP430 are recalled and summarized in Table 4.3. The immediate, the indirect and autoincrement modes are not valid as destination operands.

Core and Emulated Instructions

The MSP430 architecture has twenty-seven *hardwired* core instructions, i.e., each one with a specific OpCode in machine language syntax. In addition, assemblers support twenty-four emulated instructions, with mnemonics easier to remember. For example, to "**inv**ert" all bits in R5, "`inv R5`" is easier to recognize than the equivalent core instruction "`xor #0xFFFF,R5`." There is no penalty for the use of emulated instructions.

Tables 4.4 and 4.5 list the complete sets of MSP430 core and emulated instructions, respectively.

4.3.1 Instruction Operands

Excepting for the jump instructions, an operand in an instruction is either

- a CPU register name, or
- an integer constant, or
- a character enclosed in single quotes (‘’), which is equivalent to its ascii value, or
- a valid user defined constant

Registers PC, SP and SR may also be referred to as R0, R1 and R2, respectively. Integer constants may be in

- decimal notation, without any suffix or prefix; only decimal numbers can have the minus sign ("–") attached;

Table 4.4 Core MSP430 instructions

Type	Instruction	Description	V	N	Z	C
Data	`mov src,dest`	Loads destination with source	-	-	-	-[1]
Transfer	`push src`	Pushes source onto top of stack	-	-	-	-
	`swpb dest`	Swap bytes in destination word	-	-	-	-
	`add src,dest`	Adds source to destination	*	*	*	*
	`addc src,dest`	Adds source and carry to destination	*	*	*	*
	`sub src,dest`	Adds $\overline{\text{source}} + 1$ to destination	*	*	*	*
Arithmetic		(subtract source from destination)				
	`subc src,dest`	Adds $\overline{\text{source}} + CF$ to destination	*	*	*	*
		(subtract with borrow)				
	`dadd src,dest`	Adds source and carry to destination	*	*	*	*
		in Decimal (BCD) form[2]				
	`cmp src,dest`	$dest - source$, but only affects flags[3]	*	*	*	*
	`sxt dest`	Sign extend LSB to 16-bit word	0	*	*	*
	`and src,dest`	"AND"s source to destination bitwise	0	*	*	*
	`xor src,dest`	"XOR"s source to destination bitwise	*	*	*	*
	`bit src,dest`	Like `and`, but only affects flags[4]	0	*	*	*
Logic	`bic src,dest`	Resets bits in destination	-	-	-	-
and bit	`bis src,dest`	Sets bits in destination.	-	-	-	-
management	`rra dest`	Roll bits to right arithmetically, i.e.,	0	*	*	*
		$B_n \rightarrow B_(n-1) \ldots B_1 \rightarrow B_0 \rightarrow C$				
	`rrc dest`	Roll destinations to right through Carry,	*	*	*	*
		$C \rightarrow B_n \rightarrow B_(n-1) \ldots B_1 \rightarrow B_0 \rightarrow C$				
	`jz/jeq label`	Jump if zero/equal (Z = 1)	-	-	-	-
	`jnz/jne label`	Jump not zero/equal (Z = 0)	-	-	-	-
	`jc/jhe label`	Jump if carry (C = 1) – if higher or equal–-	-	-	-	–
		($\geq$, for unsigned numbers)				
	`jnc/jlo label`	Jump if not carry (C = 0)– if lower,– -	-	-	-	–
		($<$, for unsigned numbers)				
Program	`jn label`	Jump if negative (N = 1)	-	-	-	-
Flow	`jge label`	Jump if $V = N$	-	-	-	-
		($\geq$, for signed numbers)				
	`jl label`	Jump if V$\neq$N	-	-	-	-
		(if $<$, signed numbers)				
	`jmp label`	Jump to label unconditionally	-	-	-	-
	`call dest`	Call subroutine at destination	-	-	-	-
	`reti`	Return from interrupt	-	-	-	-

[1]: For Flags: – means there is no effect; * there is an effect; "0", flag is reset.
[2]: Result is irrelevant if operands are not in format BCD
[3]: Used to compare numbers, usually followed by a conditional jump
[4]: Used to test if bits are set, usually followed by a conditional jump

- binary notation with suffix b, as in 00101101b, or in the form b'00101101';
- hex notation with suffix h, as in 23FEh or prefix 0x as in 0x23FE, or prefix h as in h'23FE'. Suffixed numbers must never start with a letter;
- octal notation with suffix q, as in 372q, or prefix q as in q'372'.

Table 4.5 Emulated instructions in the MSP430

Type	Instruction	Description	Core Inst.
Data Transfer	`pop dest`	Loads destination from TOS	`mov @SP+,dest`
	`adc dest`	Add carry to destination	`addc #0,dest`
	`dadc src,dest`	Decimal add Carry to destination	`addc #0,dest`
	`dec dest`	Decrement destination	`sub #1,dest`
Arithmetic	`decd dest`	Decrement destination twice	`sub #2,dest`
	`inc dest`	Increment destination	`add #1,dest`
	`incd dest`	Increment destination twice	`add #2,dest`
	`sbc dest`	Subtract Carry from destination	`subc #0,dest`
	`tst dest`	Test destination	`cmp #0,dest`
	`inv dest`	Invert bits in destination	`xor #0FFFFh,dest`
	`rla dest`	Roll (shift) bits to left	`add dest,dest`
Logic	`rlc dest`	Roll bits left through carry	`addc dest,dest`
and bit	`clr dest`	Clear destination	`mov #0,dest`
Management	`clrc`	Clear carry flag	`bic #1,SR`
	`clrz`	Clear zero flag	`bic #2,SR`
	`clrn`	Clear negative flag	`bic #4,SR`
	`setc`	Clear carry flag	`bis #1,SR`
	`setz`	Clear zero flag	`bis #2,SR`
	`setn`	Clear negative flag	`bis #4,SR`
	`br dest`	Branch to destination	`mov dest,PC`
Program	`dint`	Disable interrupts	`bic #8,SR`
Flow	`eint`	Enable interrupts	`bis #8,SR`
	`nop`	no operation	`mov R3,R3`
	`ret`	Return from subroutine	`mov @SP+,PC`

Operands for Jump Instructions

In MSP430, in a source program to be compiled by an assembler, the operand is direct, giving the address for the jump. This value is usually given as a labe, as in `jmp mainloop`, since it is very improbable that the programmer knows the address before compiling.

However, for line assemblers it must be an even signed number within the limits ±512. This is half the offset to be added to the PC register, and hence represents the number of bytes in program memory to be jumped to, forward or backward. Lines 7 and 8 in Fig. 4.5, which appear as `jn 0x3FC` and `jmp 0x3F2` could have also be written as `jn-4` and `jmp-14`, respectively.

4.3.2 Word and Byte Instructions

With some exceptions, instruction operands may be either byte-size or word-size. Word and byte instructions are differentiated with a suffix **.w** or **.b**, respectively. The default when no suffix is included is a word instruction.

In byte instructions, when the datum is in memory, the source or destination refers specifically to the byte in the cell. Addresses can be even or odd, with no particular problem. In addition, in the indirect autoincrement addressing mode @Rn+, register Rn is automatically incremented by 1 when working with byte instructions.

Now, when data is in a register, since the register itself is 16-bit wide, the situation is a little different. A source operand in register mode in byte operations points only to the least significant byte. When a destination, the result goes to the least significant byte and the most significant byte is cleared. These remarks are illustrated in Fig. 4.12.

Observe that the most significant byte of a register is not available with a byte instruction. Therefore, to access or change only this part of the register, `swpb Rn` may be utilized before/after the byte instructions.

The following example illustrates the operation of byte and word instructions.

Example 4.3 *The following table shows examples of individual and sequence of instructions for byte and word operations. All numbers in the table are in hex notation without suffix or prefix. MSB(Rn) amd LSB(Rn) mean, respectively, most and least significant byte of Rn. Before each instruction or sequence, contents are*

R5 = 03DAh, R6 = 0226h, R15 = BAF4h, [03DAh] = 2B40h, [03DCh] = 4580h and [0226] = F35Ah.

Since the MSP430 uses little endian method, [03DAh] = 40h, [03DBh] = 2Bh *and so on. In byte operations with memory, only the byte is taken. Hence* @R5, @R5+, 0(R6) and 1(R6) *in the instructions point only to the byte, not the word. This was indicated only once in the RTN of the second column to eliminate any ambiguity.*

4.3.3 Constant Generators

Operands in register mode yield faster execution and require less program memory. Any non-register operand data, like a number in immediate mode or an address,

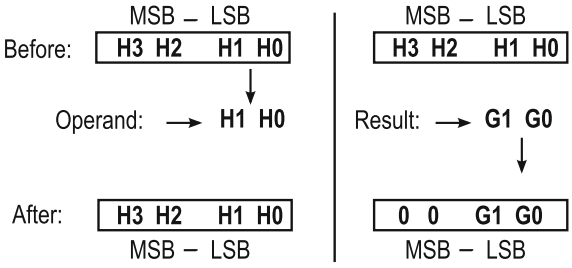


Fig. 4.12 Byte operations with data in register: **a** Register as source only, **b** Register as destination (Hx and Gx stand for generic hex digits.)

generates one word in machine language, as it can be verified in listing of Fig. 4.5. The immediate mode always generates the word immediately after the instruction word[7] whenever two words are appended. Since some immediate values are very often used, CPU designers usually include one or two registers hardwired to them in such a way that when the programmer write the corresponding immediate operand, the actual machine code uses a register mode expression.

Instruction(s)	Register notation	After
`mov.w R5,R6`	R6 ← R5	R5 = 03DA, R6 = 03DA
`mov.b R5,R6`	LSB(R6) ← LSB(R5) MSB(R6) ← 00h	R5 = 03DA, R6 = 00DA
`mov @R5,R6`	R6 ← (R5)	R5 = 03DA, R6 = 2B40, [03DA] = 2B40
`mov.b @R5,R6`	MSB(R6)←0, LSB(R6)←(R5)	R5 = 03DA, R6 = 0040 [03DA] = 2B40
`mov @R5,0(R6)`	(R6) ← (R5)	R5 = 03DA, R6 = 0226 [03DA] = 2B40, [0226] = 2B40
`mov.b @R5,1(R6)`	Byte(R6 + 1) ← Byte(R5)	R5 = 03DA, R6 = 0226 [03DA] = 2B40, [0226] = 405A
Sequence:		
`mov @R5+,R6`	R6 ← (R5); R5 ← R5 + 2	R5 = 03DC, R6 = 2*B*40, [03DA] = 2B40, [03DC] = 4580
`mov @R5+,R15`	R15 ← (R5) R5 ← R5 + 2	R5 = 03DE, R15 = 4580, [03DA] = 2B40, [03DC] = 4580
Sequence:		
`mov.b @R5+,R6`	MSB(R6)←0, LSB(R6)←(R5); R5 ← R5 + 1	R5 = 03DB, R6 = 0040 [03DA] = 2B40
`mov.b @R5+,R15`	MSB(R15)←0, LSB(R15)←(R5); R5 ← R5 + 1	R5 = 03DC, R15 = 002B [03DA] = 2B40
`mov.b R6,&0227`	(0227)←LSB(R6)	R6 = 0226, [0226] = 265A

MSP430 designers adopted a similar philosophy, with interesting variations. Both R2 and R3 are used as *constant generators* for several constants, as indicated below:

- R3 for immediate values 0, 1, 2 and −1 (0xFFFF)
- R2 for immediate values 4 and 8, and for absolute value 0.

Some bits in the instruction word identify the individual cases, as explained in Appendix B.2. Notice that R3 is not a general purpose register but may be used as an operand. It does not store any result as a destination, and as an explicit source is equivalent to #0. On the other hand, when R2 is explicitly mentioned as an operand, it refers to the SR register.

[7] Hence the name immediate mode.

Example 4.4 *To illustrate the constant generation, consider lines 1–6 in listings of Fig. 4.5. They all have immediate mode sources, since instruction* `dec.w R15` *emulates* `sub.w #1,R15`*. Let us now look at the following table:*

Non-constant generation		Constant generation	
Assembly Instruction	Machine language	Assembly Instruction	Machine language
`mov.w #0x300,SP`	4031 **0300**	`bis.b #001,& 0x0022`	D**3**D2 0022
`mov.w #0x5A8,& 0x0120`	40B2 **5A80** 0120	`xor.b #001,& 0x0021`	E**3**D2 0021
`mov.w #0xC350,R15`	403F **C350**	`sub.w #001,R15`	8**3**1F

Immediate values of instructions on the left hand column do not belong to the constant generation set. Therefore, they are explicitly included in the machine instruction after the instruction word (highlighted with bold fonts). On the right column, the source "# 1" *is generated by register* R3*. The highlighted nibble* 3 *in the machine language instruction word refers to register R3, and there is no extra word for the immediate value.*

Therefore, the constant generation property of R3 has saved one word (two memory locations) in memory, as well as power while achieving a faster execution.

Arithmetic and Logic Operations

As illustrated in the previous examples, assemblers interpret logic and arithmetic expressions used in operands. Table 4.6 shows some valid operators for these expressions. Operations follow rules of precedence, with a smaller group number having more preference. In a same group, precedence is from left to right, except for unary operators which apply exclusively to the operand following them.

4.4 Details on MSP430 Instruction Set

MSP430 instructions may be classified into four groups:

1. Data transfer instructions;
2. Arithmetic instructions;
3. Logic instructions;
4. Program flow instructions.

Tables 4.4 and 4.5 indicate which instructions fall in each group. Before going into the assembly process, let us look quickly into the different groups.

Table 4.6 Valid Operators in expressions listed by precedence order

Group	Operator	Meaning
1	+	Unary plus symbol
	−	Unary minus symbol
	~	1s complement
		Logical NOT
2	*	Multiplication
	/	Division
	%	Modulo
3	+	Addition
	−	Subtraction
4	<<	Shift left
	>>	Shift right
5	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
6	=	Equal to
	!=	Not equal to
7	&	AND
	^	XOR
	||	OR

4.4.1 Data Transfer Instructions

The three core and one emulated MSP430 data transfer instructions are: **mov**, **push**, **swpb** and **pop**. They operate as indicated in the tables and do not affect any flag. The **mov** instruction is the most often used in assembly programs. In RTN notation:

`mov src,dest`	dest ← src	`swpb dest`	LSB(dest) ↔ MSB(dest)
`push src`	(SP) ← src	`pop dest`	dest ← (SP)
	SP ← SP-2		SP ← SP+2

Push and Pop: The instructions `push` and `pop`, being common to all CPU's, have been explained in Chap. 3. In the MSP430, all SP updates are always by steps of two, and SP always points to an even address. These instructions require careful management to avoid bugs because of mishandling of these operations. Between the push operation of the datum and the pop operation to retrieve the same datum, either we have an equal number of push and pop operations, or else we should manipulate the SP register properly. Also, retrieving must be done in the reverse order of pushing. The following example illustrates this point.

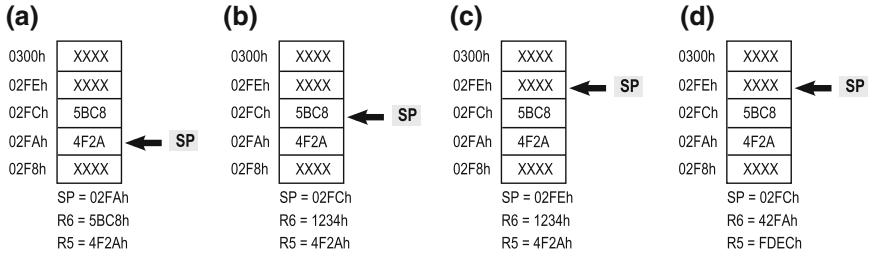


Fig. 4.13 Illustration of **push** and **pop** in Example 4.5. **a** After push operations, **b** Just before pop R6 in correct sequence, **c** After pop R6 in correct sequence, **d** After pop R6 in incorrect sequence

Example 4.5 *Assume* R6 = 5BC8h, R5 = 4F2AhSP = 02FE *before the following two sequences:*

Correct Sequence:

```
FIRST:  push R6          ;save R6
        push R5
        mov #1234h,R6
        mov #0xFEDC,R5
        pop R5           ;recover R5
LAST:   pop  R6          ;recover R6
```

Incorrect Sequence:

```
FIRST:  push R6          ;save R6
        push R5          ;save R5
        mov #1234h,R6
        mov #0xFEDC,R5
LAST:   pop  R6          ;recover R6
```

Figure 4.13 illustrates the differences. Case (a) *shows the* SP *value just after the* `push R5` *operation, valid for both sequences. Case* (b) *corresponds to the first case. Notice that the original* R5 *was retrieved and R6 is about to be recovered too after the pop operation, as seen in* (c). *This is the situation for an equal number of push and pop operations, in a correct sequence, after* `push R6`*; notice that* SP *ends up with the same original value. An unequal number of push and pops case, without proper manipulation of* SP *is illustrated by* (d). *Notice that neither* R5 *nor* R6 *were recovered, and* SP *ends at a different address. Also, if we add* `pop R5` *after* `pop R6`*, the registers will not be loaded with their original values.*

Swap Bytes: The swap-bytes instruction (**swpb**) exchanges the most and least significant bytes of the destination word, usually a register. For example, if R5 = $34FAh$,

then `swpb R5` yields R5 = $FA34h$. This is handy because the programmer has direct access only to the least significant byte of registers in byte operations.

4.4.2 Arithmetic Instructions

For easiness, the arithmetic instructions are repeated in Table 4.7 with their RTN equivalent. These operations are performed by the ALU and have effect on flags C, Z, N and V. The "normal" effect mentioned in the table is defined as follows:

Carry flag SR(0): C = 1 carry occurs, C = 0 if no carry occurs;
Zero flag SR(1): Z = 1 if result is cleared; Z = 0 otherwise;
Negative or Sign flag SR(2): N reflects the MSB of the result (Bit 7 for byte operations, Bit 15 for word operations);
Overflow flag SR(8): V = 1 if one of the following occurs:

1. addition of two equally signed number produces a number of the opposite sign
2. if the remainder in subtraction when subtrahend and minuend are of different sign, has the sign of the subtrahend.

Flags N and V make sense for signed numbers only. For unsigned numbers, overflow is flagged by the Carry flag. Yet, remember that interpretation of flags is a task that belongs to the programmer, not to the controller. Notation SR(X) refers to the position of the bit in the status register. Emulated instructions affect flags according to the core equivalent.

The subtraction operations, with and without borrow, are actually realized as two's complement addition. Thus, "`sub src, dest`" executes "dest + NOT src + 1". In MSP430, carry C = 0 denotes the presence of a borrow while C = 1 indicates that no borrow was needed. See p. XXX for a reminder. Notice that subtraction with borrow has two mnemonics, so the programmer may use whichever makes more sense for the program.

The MSP430 family ALU supports BCD addition, also called decimal addition (see Sect. 2.7 .3) with instruction `dadd`, although involving the C flag. This instruction assumes that the operands are unsigned integers encoded in BCD. Therefore, the result of `dadd src,dest` is meaningless if any of the operands is not in this code. After the `dadd`, flag V is undefined and C = 1 if result is greater than 9999 for word operations or 99 for byte operations. Since the decimal addition also includes the carry, the programmer must secure that the carry flag is 0 before addition whenever necessary.

The sign extension instruction `sxt` affects the most significant byte of the word destination. If bit 7 is 0, then the MSB becomes 0x00; if the bit is 1, the MSB becomes 0xFF. In other words, it extends an 8-bit signed number, the LSB of the destination, into a 16-bit one. This is a useful instruction when processing data from 8-bit ports delivering signed data, as illustrated by the following sequence:

Table 4.7 Arithmetic instructions for the MSP430

Core Instructions			
Mnemonics & Operands	Description	Flags	Comments
`add src,dest`	dest ← src + dest	Normal	Add src to dest
`addc src,dest`	dest ← src + dest + C	Normal	Add with Carry
`dadd src,dest`	BCD algorithm used in	Special[a]	Decimal version of `addc`.
	dest ← src + dest + C		Data in BCD format
`sub src,dest`	dest ← dest + `.NOT.` src + 1	Normal	Subtract src from dest. ($dest \leftarrow dest - src$)[b]
`subc src,dest` `sbb src,dest`	dest ← dest + `.NOT.` src + C	Normal	Subtract with borrow ($dest \leftarrow dest - src + C$)[b]
`cmp src,dest`	dest + `.NOT.` src + 1	Normal	Only affect flags.
`sxt dest`	MSB ← FFh×(Bit 7)	Special[c]	Word operand only. Signed LSB extended to 16 bit.

Emulated			
Mnemonics & Operands	Description	Emulated Instruction	Comments
`adc dest`	dest ← dest + carry	`addc #0,dest`	Add carry to dest.
`dadc dest`	BCD version for `adc`	`dadd #0,dest`	
`inc dest`	dest ← dest + 1	`add #1,dest`	
`incd dest`	dest ← dest + 2	`add #2,dest`	
`dec dest`	dest ← dest − 1	`sub #1,dest`	
`decd dest`	dest ← dest − 2	`sub #2,dest`	
`tst dest`	dest ← dest − 0	`cmp #0,dest`	To test for sign or zero

[a] C = 1 if result > 99 for bytes or result > 9999 for words; V is undefined
[b] Borrow needed if C = 0; Borrow = $\overline{Carry}$
[c] C = `NOT`Z, V = 0

```
mov.b &P1IN,R14   ;Read input data
sxt   R14         ;Sign extend for processing
```

Notice that multiplication and division are not supported by the MSP430 ALU. Some models include a hardware multiplier. We shall deal with this peripheral in later sections.

Let us illustrate some operations with core instructions, since emulated ones function similarly.

Example 4.6 *Each case in the tables below is independent of others. For all cases, contents of registers and memory before the instruction are as follows, unless otherwise indicated. All contents and addresses are in hex notation without suffix or prefix:*

$$R5 = 35DA,\ R6 = EF26,\ R7 = 5469,\ R8 = 0268,$$
$$[0268] = 364A,\ [026A] = 2FD1,\ [03BC] = 1087$$

dummy

Instruction	Operation	Results	Flage C	Z	N	V
Addition:						
`add R5,R6` or						
`add.w R5,R6`	35DAh+EF26h=12500h	R6=2500	1	0	0	0
`add.b #0x26,R5`	26h + 0DAh = 100h	R5=0000	1	1	0	0
Decimal addition:						
A. Assuming carry C=0.						
`dadd.b #0x96,R6`	0 + 96 + 26 = 122	R6=0022	1	0	0	X
B. Assuming carry C=1.						
`dadd R7,&0x03BC` or						
`dadd.w R7,&0x03BC`	1+5469+1087 = 6557	[03BC]=6557	0	0	0	X
Instruction	Operation	Results	C	Z	N	V
Subtraction, which actually uses two's complement addition						
`sub 2(R8),R6` or						
`sub.w 2(R8),R6`	EF26h+D02Eh+1h = 19F55h	R6=9F55	1	0	1	0
`sub.b #67,R5`	0DAh + 0BCh+1h = 197h	R5=0097	1	0	1	0
Sign extention						
`sxt R5`	Bit7=1: MSB(R5)←FFh	R5=FFDA	1	0	1	0
`sxt R6`	Bit7=0: MSB(R6)←FFh	R6=0026	0	0	0	0
Compare						
`cmp R6,R7` or						
`cmp.w R6,R7`	5469h+10D9h+1 = 6543h	No change	0	0	0	0

Instructions `addc`, `sbb`, and `dadd`, involve the carry/borrow in addition and subtraction. There are situations in which this property becomes very useful. One such example is for addition or subtraction of data larger than the operand limitations. This is analogous to the hardware use of two or more adders, as illustrated by Fig. 4.14. The following example illustrates a software code for the same objective.

Example 4.7 *To add* 36,297,659+2,382,878*, 16-bit data cannot be used. However, we may combine registers to represent them and utilize more than one instruction to operate with them. The operation in hex equivalent numbers is*

$$36297659 + 2382878 = 38680537 \Rightarrow 0x0229DBBB + 0x00245C1E = 0x024E37D9$$

Since more than two bytes are required for each number, let us use registers by pairs, with the combination R10-R11 *for the first operand and for the result, and* R14-R15 *for the second operand. The following sequence of instructions realize the operation:*

```
mov  #0xDBBB,R11  ;LSW of first term
```

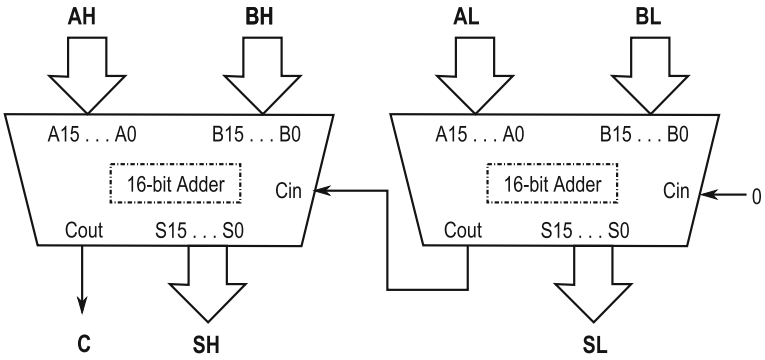


Fig. 4.14 Adding 32-bit words A and B, where A = AHAL and B = BHBL.

```
mov  #0x0229,R10  ;MSW of first term
mov  #0x5C1E,R15  ;LSW of second term
mov  #0x0024,R14  ;MSW of second term
add  R15,R11      ;add LSW's, R11 = 3729h, C = 1
addc R14,R10      ;add MSW's and C, R10 = 024Eh, C = 0
```

We could also work with the decimal addition, encoding with BCD:

```
mov  #0x7659,R11   ;LSW of first term, BCD
mov  #0x3629,R10   ;MSW of first term, BCD
mov  #0x2878,R15   ;LSW of second term, BCD
mov  #0x0238,R11   ;MSW of second term, BCD
clrc               ;Clear Carry flag
dadd  R15,R11      ;add LSW's, R11 = 0537h in BCD, C = 1
dadd  R14,R10      ;add MSW's and C; R10 = 3868h in BCD, C = 0
```

4.4.3 Logic and Register Control Instructions

The logic and register control instructions are shown in Table 4.8, together with the effect on flags. Emulated instructions affect flags according to the respective equivalent core instruction. Note that some of these are in fact arithmetic instructions. The "normal" effect on flags for logic operations is defined as follows:

Carry flag SR(0): C is the opposite of Z (C = NOTZ);
Zero flag SR(1): Z = 1 if result is cleared; Z = 0 otherwise;
Negative Flag SR(2): N reflects the MSB of result ;
Overflow Flag SR(8): V = 0.

Bitwise Logic Operations and Manipulation

Logic instructions are bitwise operations, which means that the operation is done bit-by-bit without any reference to other bits in the word. We can therefore target

Table 4.8 Logic and register control core instructions for the MSP430

Core Instructions			
Mnemonics & Operands	Description	Flags	Comments
`and src,dest`	dest ← src.`AND`.dest	Normal	Bitwise AND
`xor src,dest`	dest ← src.`XOR`.dest	See Note *	Bitwise XOR
`bic src,dest`	dest ← (`.NOT`.src).`AND`.dest	Not affected	Clear bits in dest with mask src.
`bis src,dest`	dest ← src.`OR`.dest	Not affected	Set bits in dest with mask src.
`bit src,dest`	src.`AND`.dest	Normal	Test bits in dest with mask src. Only affects flags
`rra dest`	$b_n \rightarrow b_{n-1} \rightarrow \cdots$ $\cdots \rightarrow b_0 \rightarrow C$	C←LSB	Roll dest right arithmetically.
`rrc dest`	$C_{old} \rightarrow b_n \rightarrow \cdots$ $\cdots \rightarrow b_0 \rightarrow C_{new}$	C←LSB	Rotate dest right logically through C.
Emulated Instructions			
Mnemonics & Operands	Description	Emulated Instruction	Comments
`inv dest`	bit(h)←.`NOT`.bit(h)	`xor #0xFFFF,dest`	Inverts bits in dest.
`rla dest`	$C \leftarrow b_n \leftarrow \cdots$ $\cdots \leftarrow b_0 \leftarrow 0$	`add dest,dest`	Roll dest left.
`rlc dest`	$C_{new} \leftarrow b_n \leftarrow \cdots$ $\cdots \leftarrow b_0 \leftarrow C_{old}$	`addc dest,dest`	Rotate dest left through Carry.
`clr dest`	dest ← 0	`mov #0,dest`	Clears destination.
`clrc`	C ← 0	`bic #1,SR`	Clears Carry flag.
`clrn`	N ← 0	`bic #4,SR`	Clears Sign flag.
`clrz`	Z ← 0	`bic #2,SR`	Clears Zero flag.
`setc`	C ← 1	`bis #1,SR`	Sets Carry flag.
`setn`	N ← 1	`bis #4,SR`	Sets Sign flag.
`setz`	Z ← 1	`bis #2,SR`	Sets Zero flag.

C = NOT(Z), N reflects MSB, V = 1 if both operands are negative

specific bits in data without affecting the others. For this purpose, we use *masks*. A mask X is a source whose binary expression has 1's in the target bit positions and 0's elsewhere. The properties of the logic operations which we use to achieve this manipulation are the following:

$$X + A = \begin{cases} A & X = 0 \text{ Leaves A unchanged} \\ 1 & X = 1 \text{ Forces a set} \end{cases} \tag{4.6}$$

$$X \oplus A = \begin{cases} A & X = 0 \text{ Leaves A unchanged} \\ \bar{A} & X = 1 \text{ Toggles or inverts A} \end{cases} \tag{4.7}$$

$$X \cdot A = \begin{cases} 0 & X = 0 \text{ Forces a clear or reset} \\ A & X = 1 \text{ Leaves A unchanged} \end{cases} \tag{4.8}$$

$$\bar{X} \cdot A = \begin{cases} 0 & X = 1 \text{ Forces a clear or reset} \\ A & X = 0 \text{ Leaves A unchanged} \end{cases} \tag{4.9}$$

Using these operations, we can therefore target individual bits for manipulation:

Clear bits Instruction (**bic**) uses (4.9) to *clear* the bits selected by the mask, i.e. make the bits equal to 0.
Set bits Instruction (**bis**) uses (4.6) to *set* the bits according to the mask, i.e., force them to be 1.
Toggling Instruction **xor** uses (4.7) to *toggle* bit values, i.e. inverts the current values of the target bits.
Bit testing Instruction (**bit**) uses (4.8) to *test* if at least one of the target bits is set, i.e., equal to 1.

From (4.8), instruction **and** can also be used to *clear* bits. However, now the source should have 0's at the bit positions to be cleared. We illustrate these operations with an example.

Example 4.8 *Assume contents of the registers and memory* **before** *any instruction as*

R12 = 25A3h = 0010010110100011, R15 = 8B94h = 1000101110010100,

[25A5h] = 6Ch = 01101100

(a) **AND** *and* **BIT TEST**

Instructions: `and R15,R12` *or* `and.w R15,R12` *and* `bit R15,R12` *or* `bit.w R15,R12`

Operation:	*Flags*: C = 1Z = 0N = 0V = 0
0010 0101 1010 0011 (R12) AND	
1000 1011 1001 0100 (R15) =	`and R15,R12` *yields*
0000 0001 1000 0000	R12 = 0180 *but*
	`bit R15,R12` *leaves* R12 *unchanged*

Instructions: `and.b 2(R12),R15` *and* `bit.b 2(R12),R15`

Operation:	*Flags*: C = 1Z = 0N = 0V = 0
0110 1100 (Memory) AND	
1001 0100 (LowByteR15) =	`and.b 2(R12),R15` *yields*
0000 0100 (new Low Byte R15)	R15 = 0004, *but*
	`bit.b 2(R12),R15` *leaves* R15 *unchanged*.

(b) **BIT CLEAR (BIC)**
Instruction: `bis R15,R12` *or* `bis.w R15,R12`

Operation:
0010 0101 1010 0011 (R12) AND
0111 0100 0110 1011 ($\overline{R15}$) =
0010 0100 0010 0011 (new R12)

New Contents: R12 = 2423
Flags: not affected

Instruction: `bic.b 2(R12),R15`

Operation:
1001 0011 ($\overline{Memory}$) AND
1001 0100 (LowByteR15) =
1001 0000 (new Low Byte R15)

New Contents: R15 = 0090
Flags: not affected

(c) **BIT SET (BIS)**
Instruction: `bis R15,R12` *or* `bis.w R15,R12`

Operation:
1000 1011 1001 0100 (R15) OR
0010 0101 1010 0011 (R12) =
1010 1111 1011 0111

New Contents: R12 = AFB7
Flags: not affected.

(d) **XOR**
Instruction: `xor.b #0x75,R15`

Operation:
0111 0101 (0x75) XOR
1001 0100 (LowByteR15) =
1110 0001

New Contents: R15 = 00E1
Flags: C = 1Z = 0N = 1V = 0

(e) **Invert** *(emulated instruction)*
Instruction: `inv.b &0x25A5` *equivalent to* `xor.b 0xFF,&0x25A5` *yields* [25A5h] = 10010011 = 93, *with flags:* C = 1Z = 0N = 1V = 0
Instruction: `inv R15` *equivalent to* `xor 0FFFFh,R15` *yields R15 = 0111 0100 0110 1011 = 746B with flags:* C = 1Z = 0N = 0V = 0

Remark on BIT instruction: When using instruction `bit`, the result is Z = 0 and C = 1 iff at least one of the target bits in datum is 1. An interesting and often applied case is when the set consists of only one bit. If this bit is 0, then Z = 1, C = 0; if the bit is 1, then Z = 0, C = 1. Hence, this instruction "transfers" or "copies" the tested bit onto the carry flag and the inverted bit onto the zero flag. This characteristic is useful to retrieve a bit or its complement from a word or byte.

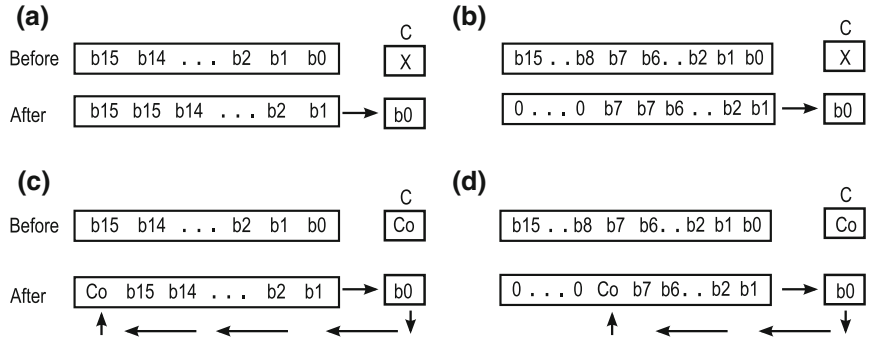


Fig. 4.15 Right arithmetic rolling (shifting): **a** `rra.w dest` and **b** `rra.b dest`; Right rotation through carry: **c** `rrc.w dest` and **b** `rrc.b dest`

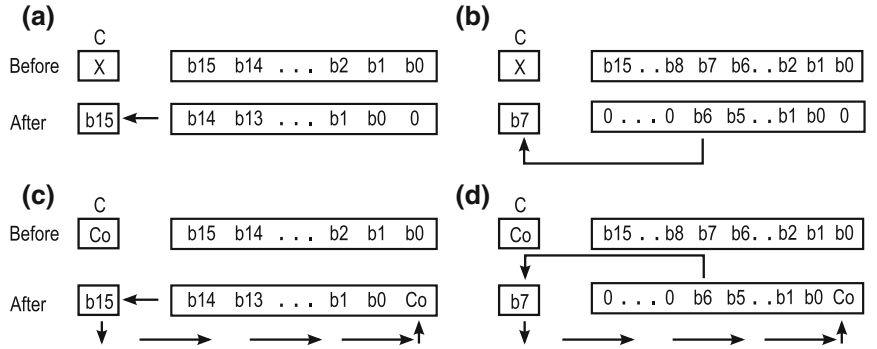


Fig. 4.16 Left "arithmetic" rolling: **a** `rla.w dest` and **b** `rla.b dest`; Left rotation through carry: **c** `rlc.w dest` and **b** `rlc.b dest`

Register Control Instructions

Although the logical operations may be considered as register control instructions, their bitwise operating characteristic makes them more appropriate for targeting specific bits inside the register. Other register control instructions are of the shift and rotating type.These instructions can be visualized on Figs. 4.15 and 4.16. Recall that in the MSP430, left rolling and rotations are actually emulated instructions. The following example illustrates these operations.

Example 4.9 *Each case below is independent of others. For all cases, contents of register* R5 *before instruction is:* R5 = 8EF5 = 1000 1110 1111 0101

(a) **Right shift/rotations**

Instruction: `rra R5` *or* `rra.w R5`

Instruction: `rra.b R5`

Instructions: `clc` *followed by* `rrc R5` *or* `rrc.w R5`

(b) **Left Shifts/Rotations**

Operation: 1000 1110 1111 0101 $\overset{rra}{\rightarrow}$ 1100 0111 0111 1010 (LSB 1⇒ C)
New Contents: R5 = C77A *Flags*: C = 1Z = 0N = 1V = 0

Operation:1000 1110 1111 0101 $\overset{rra}{\rightarrow}$ $\overbrace{00000000}^{HigherByte}\overbrace{11111010}^{rrahere}$ (LSB 1⇒ C)
New Contents: R5 = 00FA *Flags*: C = 1Z = 0N = 1V = 0

Operation: C = 0*and then* 1000 1110 1111 0101 $\overset{rra}{\rightarrow}$ 0100 0111 0111 1010 1⇒ CF
New Contents: R5 = 477A *Flags*: C = 1 Z = 0 N = 0 V = 0

Instruction: `rla R5` *or* `rla.w R5`, *equivalent to* `add R5,R5`.

Operation:1000 1110 1111 0101 *yields* C←1 0001 1101 1110 1010
New Contents: R5 = 1DEA *Flags*: C = 1 Z = 0 N = 0 V = 1

Instruction: `rla.b R5`, *equivalent to* `add.b R5,R5`

Operation: 1000 1110 1111 0101 ⇒ 0000 0000 1110 1010, C←1
New Contents: R5 = 00EA *Flags*: C = 1Z = 0N = 1V = 0

Instructions: `setc` *followed by* `rlc R5` *or* `rlc.w R5`, *equivalent to* `addc R5,R5`.

Operation: C = 1 *and then* 1000 1110 1111 0101 ⇒ 0001 1101 1110 1011 with 1⇒ C
New Contents: R5 = 1DEB *Flags*:C = 1Z = 0N = 0V = 1

Notice that the flags for left rolls result from the core instructions, which are additions.

Division and Mulitiplication by 2 Right rotation arithmetically, `rra`, may be interpreted as the division of a signed number by two in the sense

$$dividend = divisor \times quotient + residue$$

with the residue always nonnegative and less than the absolute value of the divisor (2). The carry holds the residue.

Right rotation through carry, on the other hand, may also be interpreted as a division by 2 for unsigned numbers, provided the carry is initially 0 (use `clc` before rotating).

Since left shifts are emulated by addition of destination with itself, they may be interpreted as unsigned multiplications by 2. Specifically, `rla A` may be interpreted as 2×destination, and `rlc A` as 2 × A + carry. For decimal, BCD, `dadd A,A` may be interpreted as 2 × A + carry in decimal system.

Let us use the above remarks in the following example.

Example 4.10 *Give an arithmetic interpretation to the instructions of the previous example.*

(a) Right rotations

For `rra R5`*: The signed equivalents for the numbers involved are* 8EF5h $\rightarrow$ $-28{,}939$*; signed* C77Ah $\rightarrow$ $-14{,}470$*. The carry flag has the residue 1, so* $-28,939 = -14,470 \times 2 + 1$*, corresponding to* $-28939 \div 2$*.*

For `rra.b R5`*: The signed equivalents for the numbers involved are* F5h $\rightarrow$ -11*; signed* FAh $\rightarrow$ -6*. The carry flag has the residue 1, so* $-11 = -6 \times 2 + 1$*, corresponding to* $-11 \div 2$*.*

For `clc` *followed by* `rrc R5`*, the division is for unsigned numbers. Since* 8EF5h $\rightarrow$ $36,597$ *and* $477Ah \rightarrow 18,298$*. The carry flag has the residue, and* $36,597 = 18,298 \times 2 + 1$*, which means a division by 2.*

The unsigned division occurs if a 0 is forced as a most significant bit. Therefore, preceding `rrc R5` *by* `setc` *does not result in a division.*

(b) Left shifts: Since they are emulated by addition and addition with carry, `rla` *and* `rlc` *may be interpreted, respectively, as 2x and 2x+C, including the Carry as the most significant bit for interpretations. The result is valid for both unsigned and two's complement signed encodings.*

Up to this point, we have illustrated the instructions as isolated steps. Let us look at another example, using left shift for binary to BCD conversion.

Example 4.11 Binary to BCD conversion: *Let us convert an 8-bit binary encoding into its equivalent BCD encoding. Since the largest decimal equivalent of an 8-bit number is 255, one 16-bit register suffices for the result.*

Recall that

$$\underbrace{b_7 b_6 \cdots b_1 b_0}_{\text{binary}} = \underbrace{b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0}_{\text{Decimal}}$$

A little algebraic exercise shows that the right hand decimal expression can be transformed into the so called nested multiplication form

$$(((((((2\times 0+b_7)\times 2+b_6)\times 2+b_5)\times 2+b_4)\times 2+b_3)\times 2+b_2)\times 2+b_1)\times 2+b_0$$

Therefore, the power expansion in decimal form can be calculated by eight iterations of operations of the form 2X+C*, as expressed with the following sequence:*

$$\begin{aligned} A_1 &= 2 \times 0 + b_7 \\ A_2 &= 2 \times A_1 + b_6 \\ A_3 &= 2 \times A_2 + b_5 \\ A_4 &= 2 \times A_3 + b_4 \\ &\vdots \end{aligned}$$

(a)

	R7 (in hex)				C	LSB of R6 (in binary)							
Initial State:	0	0	0	0	x	1	1	0	1	0	1	0	1
rla.b R6	0	0	0	0	1	1	0	1	0	1	0	1	0
dadd R7,R7	0	0	0	1	0	1	0	1	0	1	0	1	0
rla.b R6	0	0	0	1	1	0	1	0	1	0	1	0	0
dadd R7,R7	0	0	0	3	0	0	1	0	1	0	1	0	0
rla.b R6	0	0	0	3	0	1	0	1	0	1	0	0	0
dadd R7,R7	0	0	0	6	0	1	0	1	0	1	0	0	0
rla.b R6	0	0	0	6	1	0	1	0	1	0	0	0	0
dadd R7,R7	0	0	1	3	0	0	1	0	1	0	0	0	0
rla.b R6	0	0	1	3	0	1	0	1	0	0	0	0	0
dadd R7,R7	0	0	2	6	0	1	0	1	0	0	0	0	0
rla.b R6	0	0	2	6	1	0	1	0	0	0	0	0	0
dadd R7,R7	0	0	5	3	0	0	1	0	0	0	0	0	0
rla.b R6	0	0	5	3	0	1	0	0	0	0	0	0	0
dadd R7,R7	0	1	0	6	0	1	0	0	0	0	0	0	0
rla.b R6	0	1	0	6	1	0	0	0	0	0	0	0	0
dadd R7,R7	0	2	1	3	0	0	0	0	0	0	0	0	0

(b)

```
NUMB:  mov.b  #0xD5,R6
       mov    #0,R7
       mov    #8,R15
LOOP:  rla.b  R6
       dadd   R7,R7
       dec    R15
       jnz    LOOP
```

Fig. 4.17 Illustrating the binary to BCD conversion (11010101b → 213). **a** Sequence of steps in loop excluding counter; **b** assembly code for the loop

$$A_8 = Result = 2 \times A_7 + b_0$$

The decimal 2X+C *can be realized with* `dadd`*, provided that the carry flag is the appropriate bit. The bits can be extracted into the carry by left rollings using* `rla.b`*. Assuming that the result is written in register* R7 *and the number of interest is stored as the least significant byte of* R6*, a pseudo code for the above sequence is:*

Initialize R7 = 0
Iterate 8 times:
  Roll left LSB of R6 (`rla.b R6`)
  2×R7+C in decimal (`dadd R7`)

Figure 4.17 illustrates as an example the conversion of 11010101 *and the code for executing the algorithm.*

4.4.4 Program Flow Instructions

These instructions are listed in Table 4.9. The set can be divided in three groups: miscellaneous, jump, and subroutine handling instructions. The third group is discussed

Table 4.9 Program flow instructions for the MSP430

Core Instructions		
Mnemonics & Operands	Description	Comments
`call dest`	Push PC and PC ← dest	Subroutine Call
`jmp label`	PC ← label	Unconditional jump (goto)
`jc label`	If C = 1, then PC ← label	"Jump if carry"
(or `jhs label`)		"Jump if higher than or same as"
`jnc label`	If C = 0, then PC ← label	"Jump if no carry"
(or `jlo label`)		"Jump if lower than"
`jge label`	If N = V, then PC ← label	"Jump if greater than or equal to"
`jl label`	If N ≠ V, then PC ← label	"Jump if less than"
`jn label`	If N = 1, then PC ← #label	"Jump if negative"
`jnz label`	If Z = 0, then PC ← label	"Jump if not zero"
(or `jne label`)		"Jump if not equal"
`jz label`	If Z = 1, then PC ← label	"Jump if zero"
(or `jeq label`)		"Jump if equal"
`reti`	Pops SR and then Pops PC	Return from interrupt
Emulated Instructions		
Mnemonics & Operands	Description	Emulated Instruction
`br dest`	Branch (go) to label	`mov dest,PC`
`dint`	Disable Interrupts (GIE = 0 in SR)	`bic #8, SR`
`eint`	Enable Interrupts (GIE = 1 in SR)	`bis #8, SR`
`nop`	No operation	`mov R3, R3`
`ret`	Return from subroutine	`mov @SP+, PC`

in Sect. 4.8. The two latter groups control the flow of the program by changing the contents of the PC register. Jumps are limited to a range between −1024 and +1022 (even number) memory addresses from current PC contents.

Miscellaneous Program Flow Instructions

These instructions are emulated. Two are for interrupt handling: `eint`, to enable maskable interrupts, and `dint`, to disable them.

The third instruction is the "no operation", `nop`, emulated by `mov R3,R3`. It achieves nothing, but takes one instruction cycle to execute. It is used to introduce one cycle delay. Useful to synchronize operations with peripherals like the hardware multiplier. Enabling and disabling interrupts require one extra cycle for the hardware to actually configure. Therefore, these operations usually go together with instruction `nop` too, as illustrated next.

```
eint        ;enable interrupts
nop         ; wait for enabling

dint        ;disable interrupts
nop         ; wait for disabling
```

Similar strategies are suggested when enabling and disabling interrupt requests from peripherals.

Unconditional Jump and Branch Instructions

Unconditional jumps are realized with the jump instruction `jmp label`, and the the branch instruction `br dest`, which is emulated by `mov dest,PC`. The emulated branch instruction may go to any even address in the full memory range. Here, the operand follows the normal rules of addressing modes.

In embedded systems it is normal to close the main executable code with an unconditional jump to continuously repeat the task for which the system is being used. Now, when debugging or testing codes, it is customary to use

jmp $

to finish the code or introduce breaks. This instruction tells the CPU to repeat itself.[8]

Conditional Jumps and Control Structures

Figure 3.27 introduced the delay and iteration loops, using the `jnz` instructions. Let us expand the discussion.

Structured programming utilizes control structures of the IF-THEN and IF-THEN-ELSE type, and loop structures like FOR loops, WHILE loops, etc. We briefly review these structures and their coding in the assembly language using conditional jumps.

A conditional jump tests if a flag or a relation among flags is set or not set. The operation is illustrated by Fig. 4.18 using the flowchart decision symbols, for a single flag/relation, and for compound AND/OR.

During program design, the programmer will use statements which give origin to the flag testing, like "A≥B?", "Hardware ON", "Conversion Finished", and so on. Alternative mnemonics reinforces this process. In particular, after comparing A and B with `comp B,A`, we prefer the mnemonics as shown in Table 4.10.

Remember that "$A > B$" is equivalent to "$A \geq B$ AND $A \neq B$", or "NOT($A < B$) OR $A = B$". Similarly, "$A \leq B$" is equivalent to "$A < B$ OR $A = B$". For these compound statements we may apply the structures (b) or (c) of Fig. 4.18.

[8] It is meaningless to end a real world application with this command, just waiting until battery dries up!

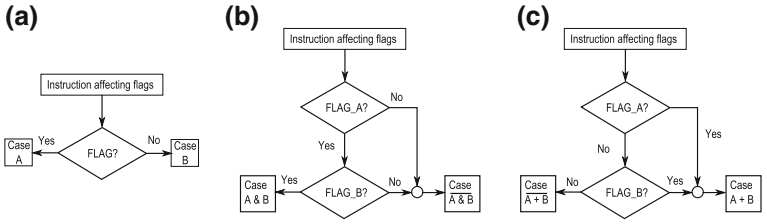


Fig. 4.18 Illustration of conditional jump operation: **a** Single Flag/condition; **b** AND-compound statement; **c** OR-compound statement

Table 4.10 Comparing A and B by A-B

Case	Unsigned numbers	Signed numbers
$A = B$	jeq	jeq
$A \neq B$	jne	jne
$A \geq B$	jhs	jge
$A < B$	jlo	jl

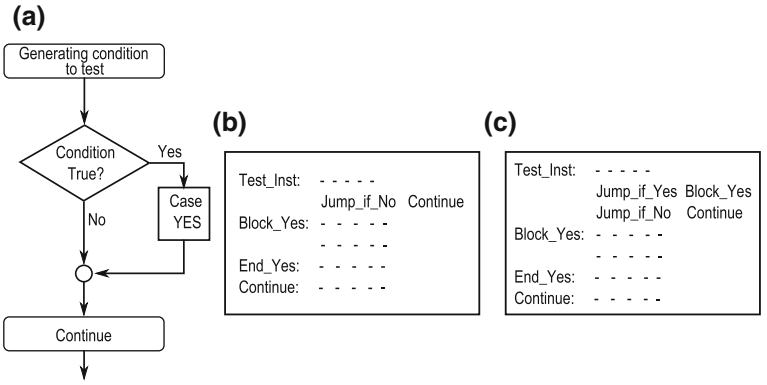


Fig. 4.19 IF-Structure **a** Flowchart, **b** and **c** Assembly code examples

IF-THEN and IF-THEN-ELSE structures The flowchart for the simple IF-THEN construct is shown in Fig. 4.19a. Insets (b) and (c) show two assembly code formats to implement this flowchart. Other formats are of course possible. The flowchart for the simple IF-THEN construct is shown in Fig. 4.20a, together with two code examples in insets (b) and (c).

Example 4.12 *Ten consecutive unsigned 16-bit numbers are stored in memory, with the first one at address NUMBERS. The objective is to add the first two numbers multiples of 4 and if the addition is not overflowed, store the result at address RESULT. Recall that a binary number is a multiple of 4 if the two least significant bits are 0.*

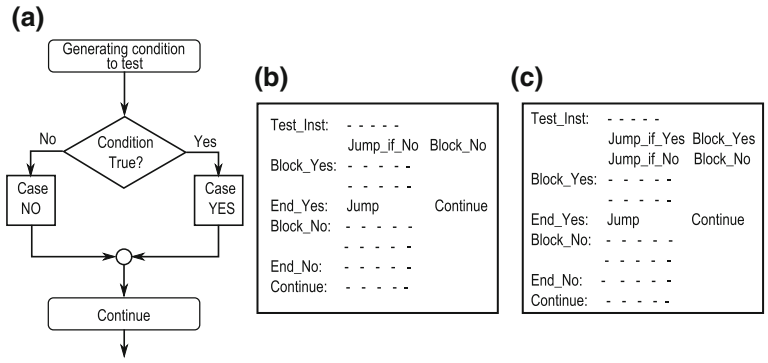


Fig. 4.20 IF-ELSE Structure **a** Flow chart, **b** and **c** Assembly code examples

The pseudo code in the left column may be programmed by the instructions of the right column:

Step 1. Initialize
pointer = NUMBERS,
SUM = 0,
COUNTER = 2.
	`Step1: mov #NUMBERS,R8`
	`mov #0,R9`
	`mov #2,R10`

Step 2. Read number & increment pointer.	`Read: mov @R8+,R11`
Step 3. If number is not multiple of 4, go to step 2.	`Step3: tst #3,R11` `jnz Read`
Step 4. Add to SUM	`Step4: add R11,R9`
Step 5. Decrement COUNTER.	`Step5: dec R10`
Step 6. If COUNTER $\neq 0$ go to step 2.	`Step6: jnz Read`
Step 7. Store Result.	`Step7: mov R9,&RESULT`

Example 4.13 *Working with unsigned numbers, assume we have a number A in register* R5 *and B* $>$ 5 *in register* R6 *Our objective is to achieve the following:*

1. IfA $<$ 5, thenB $\leftarrow$ B $-$ A

   else if A $>$ 5, *then B* $\leftarrow$ B $+$ A
   else B $\leftarrow$ 3.

2. Multiply B by 2.

An assembly listing for this pseudo code is the following:

```
test_inst:    cmp   #5,A          ; Test A-5
              jhs   NotLess       ; jump if A >= 5
Case_Less:    sub   A,B           ; if A < 5, B - A
              jmp   Dupl          ; endif
NotLess:      jz    Case_equal    ; else
```

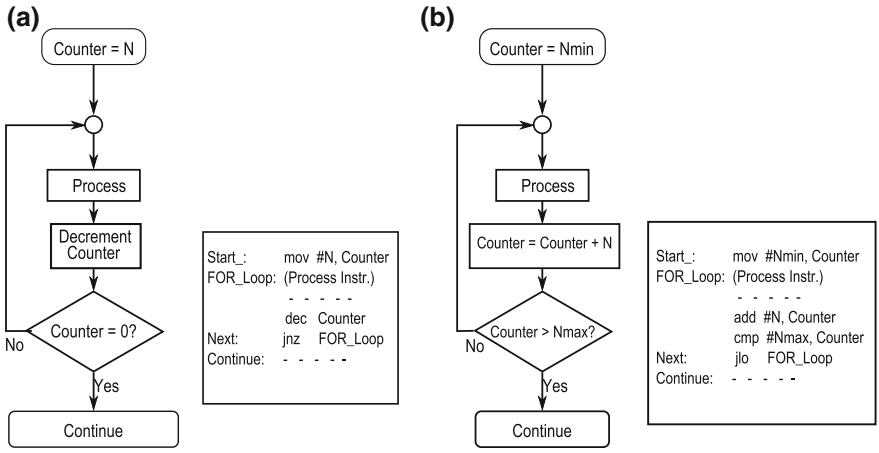


Fig. 4.21 FOR-loop structure **a** To repeat N times a process, **b** General case

```
Case_Higher: add   A,B          ; if A > 5, B + A
             jmp   Dupl         ; endif
Case_equal:  mov   #3,B         ; else if A = 5, B = 3
Dupl:        rla   B            ; Endif, 2B
```

FOR-loops Two flowcharts for FOR-Loops are shown in Fig. 4.21, together with code examples. That in (a) is very common and particularly useful when a process is to be repeated N times, as in several examples presented so far, including delay loops. In (b) we find the more traditional loop.

WHILE and REPEAT loops The principles for the WHILE-loop are illustrated in Fig. 4.22, and those for the REPEAT-UNTIL-Loop in Fig. 4.23. The loop process should include an instruction that affects the condition to be tested.

In a repeat loop, also known as *DO-UNTIL loop*, the process is executed at least once, irrespectively of the truth value of the condition to be stop, because this one is tested at the end of the loop. In a while loop, the condition is tested before going into the loop, so the loop process may not be executed at all.

Example 4.14 A polling example

Let us illustrate polling with an example: a red LED and a green LED are driven by pins P1.0 and P1.6 of port 1, respectively. An LED is on if the output at the pin is high. A pushbutton is connected at pin P1.3, provoking a low voltage when down and a high voltage when up. The objective of the code is to turn on the red LED with the green LED off while the button is kept down, and conversely when it is up. Figure 4.24 illustrates the flow chart and code for the infinite loop of the main code. This is only part of the complete source program.

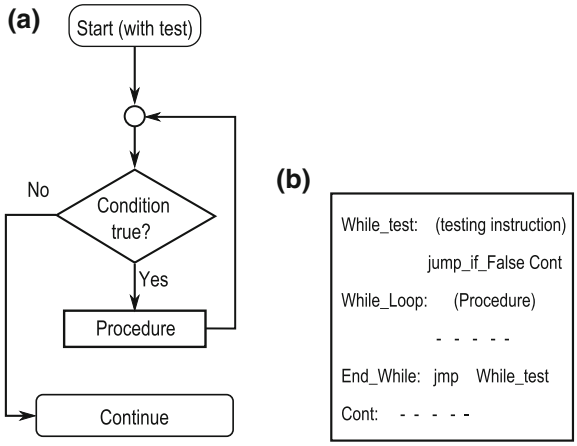


Fig. 4.22 WHILE-Loop structure: **a** Generic pseudo code; **b** Flowchart; **c** Assembly structure

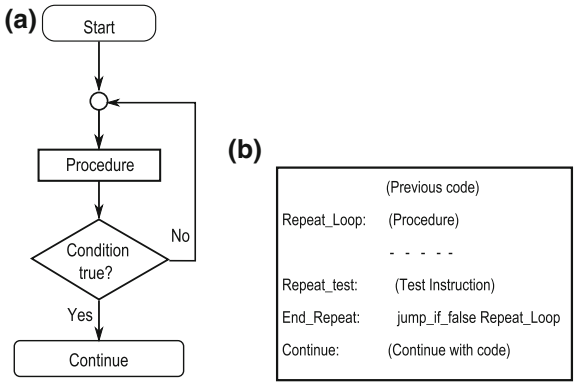
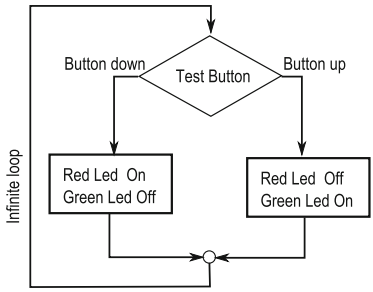


Fig. 4.23 REPEAT-UNTIL-Loop structure: **a** Generic pseudo code; **b** Flowchart; **c** Assembly structure

4.5 Multiplication and Division

Two instructions absent from the MSP430 instruction set are those for multiplication and division. Traditionally, these two operations have not been supported by the CPU hardware. As mentioned before, successive right and left shifts/rotations can be used for multiplication or division by powers of 2. This property can be applied for multiplication by small factors, based on the binary power expansion of an unsigned number. For example, to multiply $A \times 10 = 2^3A + 2A$, three left shifts plus an addition, and some extra instructions for temporary storage, will do it. There exist in the literature several general algorithms for multiplication and division. (See problems 8–10 for examples).




```
Test_Button:   bit.b  #BIT3, P1IN
               jnz    GREEN_ON      ; if button up
RED_ON:        mov.b  #BIT0, P1OUT  ; if button down
               jmp    Test_Button   ; Test again
GREEN_ON:      mov.b  #BIT6,P1OUT
               jmp    Test_Button   ; Test again
```

Fig. 4.24 Polling a button down with LEDs: flowchart and code

4.5.1 16 Bit Hardware Multiplier

Although not all MSP430 devices have a hardware multiplier for all devices, it exists in all families.[9] It is a memory mapped peripheral. In families '1xxx to '4xxx, the input registers to the multiplier are 16-bit wide but accept 8-bit data. Also, all registers have the same memory address. Families '4xxx and '5xxx/'6xxx have a 32-bit multiplier with more functions, but are software compatible with previous models. However, the source program must be written in symbolic name terms because addresses are not preserved in the new families. The 16-bit hardware multiplier is described next.

The hardware multiplier allows the following operations:

- Multiplication of unsigned 8-bit and 16-bit operands (MPY)
- Multiplication of signed 8-bit and 16-bit operands (MPYS)
- Multiply-and-accumulate function (MAC) using unsigned 8-bit and 16-bit operands
- Multiply-and-accumulate function (MAC) using signed 8-bit and 16-bit operands

Operand sizes may be mixed. The names in parentheses are the standard names (addresses) for the registers to be used. MAC multiplication actually carries an operation of the type

$$A \leftarrow A + B \times C$$

by adding the product to an accumulator. The other two types only work in the form $A \leftarrow B \times C$.

The multiplier registers are listed in Table 4.11. With the exception of SUMEXT, which is a read-only register, all others are read/write type.

The steps for multiplying are as follows:

1. Load first operand in the MPY, MPYS, or MAC register, depending on the operation. The register defines the operation.

[9] Most information in this section follows the application report slaa042, available from Texas Instruments website. Yet, this report has the old names for the registers: SUMLO and SUMHI for current RESLO and RESHI, respectively.

Table 4.11 16-bit hardware multiplier registers

Register short form name	Address	Comment
MPY	0130h	First operand, unsigned multiplication
MPYS	0132h	First operand, signed multiplication
MAC	0134h	First operand, unsigned MAC
MACS	0136h	First operand, signed MAC
OP2	0138h	Second operand, any case
RESLO	013Ah	Result low word
RESHI	013Ch	Result lhi word
SUMEXT	013Eh	Sign extension register for result

2. Load the second operand in the OP2 register. As soon as this is done, multiplication is carried out.
3. For MAC operations, add a `nop` instruction to allow time for accumulation to be achieved. This step is optional in other multiplications but it is recommended as a rule.
4. The result is available in registers RESLO, RESHIGH and SUMEXT.

The delay mentioned in step 3 above for accessing the results is necessary for MAC operations and for special cases in other multiplications. It is therefore recommended to include it as a rule, easier to remember than to remember the cases.

Example 4.15 Signed and Unsigned multiplications:

The largest unsigned multiplication with words is FFFFh × FFFFh = FFFE0001 *with no carry. Therefore, for all cases we have register* SUMEXT = 0000h. *Let us now look at some operations:*

Multiplying unsigned bytes 0x34 *and* 0xB2*, will yield a 16-bit result to be kept in register* R5*:*

```
mov.b  #0x34,&MPY  ;load first factor
mov.b  #0xB2,&OP2  ; second operand
nop                ; delay to get results
mov    &RESLO,R5   ; result to R5
```

Multiplying unsigned words stored in memory at addresses FIRSTFACTOR *and* SECONDFACTOR*, will yield a 32-bit result to be kept in memory at address* RESULT*:*

```
mov    &FIRSTFACTOR,&MPY  ; load first factor
mov    &SECONFACTOR,&OP2  ; second operand
nop                       ; delay to get results
mov    &RESLO,RESULT      ; Lower 16 bits of result
mov    &RESHI,RESULT+2    ; Higher 16 bits of result
```

Let us now consider signed multiplication: the largest negative result is obtained with 8000h$\times$ $7FFFh$ $=$ $C0008000h$ *with* SUMEXT = FFFFh *since this register offers a 16-bit sign extension for the result. The largest positive results arises from* 8000h$\times$ $8000h$ $=$ $40000000h$ *with* SUMEXT = 0000h. *The availability of* SUMEXT *allows us to obtain signed results with 48, 64 or more bits. Let us now look at the previous examples, but this time for signed multiplication:*

```
mov.b  #0x34,&MPYS ;load first factor for signed
                    multiplication
sxt    MPYS        ; sign extend MPYS register
mov.b  #0xB2,&OP2  ; second operand
sxt    OP2         ; sign extend OP2 register
nop                ; delay to get results
mov    &RESLO,R5   ; low 16 bits of result to R5
mov    &RESHI,R6   ; high 16 bits result to R6
```

The sign extension for the multiplier registers become necessary because the byte operation leaves the MSB cleared. The higher 16 bit storage is optional, depending on the needs. The following code shows the use of a 48 bit result.

```
mov    &FIRSTFACTOR,&MPYS  ; load first factor
mov    &SECONFACTOR,&OP2  ; second operand
nop                       ; delay to get results
mov    &RESLO,RESULT      ; bits 0-15 of result
mov    &RESHI,RESULT+2    ; bits 16-31 of result
mov    &SUMEXT,RESULT+4   ; bits 32-47 of result
```

When using the unsigned multiply-and-accumulate function, the register pair RESHI-RESLO constitutes the accumulator to which the product will be added, and SUMEXT will be 0000h if the addition does not generate a carry, and 0001h if it does. In the signed case, it will contain the sign extension.

Example 4.16 *Let us multiply two 2-dimensional vectors with word size unsigned numbers,*

$$[a1, a2] \times [b1, b2]^T = a1b1 + a2b2$$

with the result to be placed in memory address RESULT. *The vectors are stored in addresses* VECTOR1 *and* VECTOR2, *respectively.*

To accomplish this task, we initialize the first product and then we multiply and accumulate the second one. We store the result including SUMEXT *for the carry.*

```
mov    #VECTOR1,R4         ;initialize pointers
mov    #VECTOR2, R5
mov    @R4+,&MPY           ; initialize sum with
mov    @R5+,&OP2           ; first product
nop                        ; delay to get results
mov    @R4,&MAC            ; accumulate second product
```

```
mov    @R5,&OP2            ; to previous result
nop                        ; delay to get results
mov    &RESLO,RESULT       ; bits 0-15 of result
mov    &RESHI,RESULT+2     ; bits 16-31 of result
mov    &SUMEXT,RESULT+4    ; store carry
```

4.5.2 Remarks on Hardware Multiplier

The sum in a MAC operation does not generate an overflow flag. Hence, if it is necessary, this should be handled by software.

On the other hand, given that the multiplier, although a combinatorial system, takes time to accomplish multiplication, it is recommended to disable maskable interrupts during multiplication with a sequence such as

```
dint               ;disable interrupts
nop                ;delay to settle
mov  #A,&MPY       ;multiplication
mov  #B,&OP2       ;AB
nop                ;delay to settle
eint               ;restore interrupts
```

4.6 Macros

If your source program is rather long, chances are that it contains blocks of code that are repeated several times, either verbatim or with minor changes in operands and labels. To simplify this source encoding, assemblers allow us to create *macros*.

A macro is a set of instructions grouped under one user-defined mnemonic, used wherever the set of instructions should go. When compiling, the assembler substitutes this mnemonic by all the corresponding instructions and labels. We say that it *expands* the macro.

Notice that use of macros simplifies and shortens the user source program only, not the object files. Hence, if memory resources are limited and you must expand the macro several times, consider using subroutines instead.

On the other hand, in time intensive applications, the overhead in calling and returning from subroutines could become a problem, and macros offer good alternatives for a modular solution. At the end, a trade-off between using macros or subroutines should be evaluated.

A macro must be defined before it can be used. It can be defined either in the source file, in another file which can be copied or included, or in a macro library. It is possible to pass parameters to the macro, and to define local variables, especially labels, which the assembler will differentiate in the different expansion instances.

In the IAR assembler the macro definition starts with the statement

```
macroname MACRO [arg] [,arg]
```

and ends with the directive `ENDM`.

Here, "macroname" is the user-defined mnemonic for the macro. Optional arguments separated by commas are parameters to be passed and substituted by the assembler during expansion. Local variables are defined with the directive `LOCAL` within the macro definition.

Example 4.17 *Assume that a red LED is connected to pin P1.0 of port 1, and a green LED to pin P1.6. (Let us define therefore for this connection* `RLED EQU BIT0` *and* `GLED EQU BIT6`*). Now, the following macro is defined to turn-on an LED the number of times defined by* Times*, with a speed determined by the number defined with the parameter* DelayTime*.*

```
ledflash  MACRO LED, Times, Delaytime
          LOCAL Counting, Delay
          mov   #2*Times, R12          ;On-Off count
Counting: xor.b #LED,P1OUT             ;toggle LED
          mov   #DelayTime,R15         ;Load delay to counter
Delay:    dec   R15                    ; Delay Loop
          jnz   Delay                  ;until counter=0
          dec   R12                    ;repeat toggling
          jnz   Counting               ;
          ENDM                         ;end definition
```

After this definition, `ledflash RLED, 20, 60000` *will cause the CPU to turn on and off the Red Led 20 times with a medium speed,* `ledflash GLED, 10, 30000` *will work with the green led 10 times, twice the speed. Also,* `ledflash GLED+RLED, 10, 50000` *will work with both leds.*

When using the hardware multiplier, it may be convenient to create macros to simplify and make the code easier to read and write with templates like

```
Mult_16   MACRO   FACT1,FACT2  ; unsigned word
                                 multiplication
          mov     FACT1,MPY
          mov     FACT2,OP2
          nop
          ENDM
```

Hence, `Mult_16 R4,R5` becomes a convenient mnemonics for multiplication of registers R4 and R5.

The IAR assembler offers other techniques to simplify the source writing. The reader is encouraged to consult the IAR manual.

4.7 Assembly Programming: Second Pass

The source file layout was introduced in Sect. 4.2.4. The basic layout for a source file consists of (1) documentation, (2) Assembly time constants definitions, (3) Data and variables memory allocation, (4) Executable code, (5) reset and interrupt vector allocation, and (6) directive `END`.

The order is not necessary as mentioned, except for the ending directive. Also, only item(4), (5) and (6) are compulsory. When uploaded in the microcontroller memory, the executable code normally goes in the program or code memory section in the flash-ROM space, and data and variables to the RAM space, unless otherwise directed.[10] Every IAR source file ends with the directive `END`; anything after this directive is ignored by the assembler.

Let us now look at more directives and considerations about the source file.

4.7.1 Relocatable Source

Figure 4.11 was an example of an absolute source file, where the *program location counter* (PLC) used by the linker to load each instruction and data at the appropriate address, is controlled with the **ORG** directive. An alternative to this method is a *relocatable* source file, illustrated by Fig. 4.25 for the same task as before.

In relocatable codes we use *segment directives* to define or work memory segments, whose starting address is specified by the linker.

Absolute codes have the disadvantage that a program written for one model may not be suitable for another one because of a different memory map. Relocatable codes try to solve this problem and leave the storage work to the linker. The programmer must rely on the linker for doing a good work. Unfortunately, not all linkers are that reliable. Some assemblers, like the CCS from TI, only work relocatable codes.

Segment directives In relocatable codes the object file is organized by *relocatable sections*, managed with the use of *segment directives* . A section is a block of code or data that occupies contiguous space in the memory map. Each one has an associated *section program counter* (SPC), also called *section location counter* (SLC), which is initially set to 0.

The SLC functions within the relocatable section only, assigning an offset address with respect to the beginning of the section to the stored information. The actual physical address allocation is done by the linker during the compiling process. All sections are independently relocatable, which allows the user a more efficient use of memory.

IAR starts a relocatable section with directive **RSEG**, followed by the type or name of the section, as illustrated by Fig. 4.25. In the MSP430 models, some addresses or sections have names attached. An important one is RESET, for the reset vector

[10] The MSP430 CPU can run programs from the RAM space. This is not a feature of every microcontroller.

```
1  ;*******************************************************
2  ;   MSP430G2xx1 Demo - Software Toggle P1.0
3  ;   Description: Toggle P1.0 by xor'ing P1.0 inside
4  ;   of a software loop.
5       ACLK = n/a, MCLK = SMCLK = default DCO
6  ;
7  ;                MSP430G2xx1
8  ;             - - - - - - - - -
9  ;         /|\|              XIN|-
10 ;          | |                 |
11 ;          - -|RST           XOUT|-
12 ;             |                 |
13 ;             |             P1.0|- ->LED
14 ;
15 ;   Based on code written by D. Dang
16 ;   Texas Instruments Inc.
17 ;   October 2010
18 ;   Built with IAR Embedded Workbench Version: 5.10
19 ;*******************************************************
20 #include  "msp430g2231.h"                    ; standard constants
21 LED          EQU    01h                     ;LED at pin P1.0
22 DELAY        EQU    50000
23 #define      COUNTER R15                     ;using R15 as counter
24 ;- - - - - - - - - - - - - - - - - - - - - - - - - - - -
25              RSEG    CSTACK                  ; creates a stack in RAM
26              RSEG    CODE                    ; Program goes to code
27 ;- - - - - - - - - - - - - - - - - - - - - - - - - - - -
28 RESET        mov.w   #SFE(CSTACK),SP     ; Initialize stack
29 StopWDT      mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
30 SetupP1      bis.b   #001h,&P1DIR            ; P1.0 output
31                                              ;
32 Mainloop     xor.b   #LED,&P1OUT             ; Toggle P1.0
33 Wait         mov.w   #DELAY,COUNTER          ; Delay to R15
34 L1           dec.w   COUNTER                 ; Decrement R15
35              jnz     L1                      ; Delay over?
36              jmp     Mainloop                ; Again
37                                              ;
38 ;- - - - - - - - - - - - - - - - - - - - - - - - - - - -
39 ;            Interrupt Vectors
40 ;- - - - - - - - - - - - - - - - - - - - - - - - - - - -
41              RSEG    RESET                   ; MSP430 segment for
42              DW      RESET                   ; RESET Vector
43              END
```

Documentation

Constants Declaration

Segment directives

Executable Code

Reset vector allocation

Fig. 4.25 Relocatable IAR listing for blinking LED (*Courtesy Texas Instruments Inc.*)

allocation. Types of sections include CSTACK, CODE, DATA. Notice that `RSEG CSTACK` creates a stack at the top of the RAM; in this case, initialization of SP uses standard defined constant **SFE(CSTACK)**, whose value depends on the model.

Example 4.18 *Let us now see how the source code is assembled in relocatable mode. The difference with the previous example can be seen next:*

```
                              RSEG   CSTACK                  ; creates stack
0000                          RSEG   CODE                ; starts code section
0000  4031 0300       RESET   mov.w  #SFE(STACK),SP        ; Set stack
0004  40B2 5A80 0120  StopWDT mov.w  #WDTPW+WDTHOLD,&WDTCTL ; Stop WDT
000A  D3D2 0022               bis.b  #001h,&P1DIR           ; P1.0 output
000E  ------                  ------
      --                      --
                              RSEG   RESET                 ; Allocates reset
0000  ----                    DW     RESET                 ; vector address
                              END
```

We can see here the action of the section program counter. It only keeps track of the offset addresses within the section itself. Every time you change section, the respective SPC goes into effect from the last value where it was left. Labels do not

Table 4.12 Initialized Data definition or allocation IAR directives

Directive	Use
`DC8` or `DB`	Generates 8-bit constants, including strings.
`DC16` or `DW`	Generates 16-bit constants.
`DC32` or `DL`	Generates 32-bit constants.
`DC64`	Generates 64-bit constants.
`DF32` or `DF`	Generates 32-bit floating point constants.
`DF64`	Generates 64-bit floating point constants.
`.double`	Generates 48-bit floating point constants in TI Format.

Texas Instrument's 32-bit

have yet any value assigned because the physical memory addresses are not known at this point.

4.7.2 Data and Variable Memory Assignment and Allocation

In addition to the assembly-time constants, we also have variables and data. We can define *memory initialized constants* using a label as an address pointer with the format

```
LABEL <<Data-allocation-Directive>> data,[data,]
```

One or more data can be written on the same line, separated by commas. The Label in this case is the address value of the location for the first datum only. The IAR directives used in MSP430 assembly programs for this purpose are shown in Table 4.12.

If the constant is to be maintained fixed, and above all kept when the system is de-energized, then it is preferably stored in the flash ROM section. It can be defined in the code segment before the reset vector or after the executable code. On the other hand, if it is a value that in fact is going to be changed later during the program execution, we store it in the RAM section. We usually call it then *initialized variable*.

The following are examples for data allocation:

Example 4.19 *The memory contents is all in hex notation without suffix h or prefix* 0x. *In all cases, label stands for the address value of the first byte. Directive* `EVEN` *aligns address to an even address, which is important for data larger than a byte. Data in memory is presented horizontally in byte-size information, so little endian convention should be observed. The IEEE floating point single precision equivalent for* −357.892 *is* 0xC3B2F22D *and the double precision equivalent is* 0xC0765E45A1CAC083.

It must be stressed that the label name points only to the address of the first byte, in all cases. In assembly language, the nature of the data is not defined by types, as

it is the case in high level language. Thus, even though the user can define a floating number with the directive DF as shown in the example, the memory contents can be used as bytes, words or double words, without any reference to the origin. Thus,

```
Single DB 45,-14,0xB2,195
```

yields the same memory contents as before. This is an important point to remember, and of course a disadvantage of assembly language for structured programming.

For these types of definitions, the values are accessed with the labels in absolute or direct addressing mode. In the following example we illustrate the difference between the assembly-time constant and the memory initialized ones using the directive DW.

Example 4.20 *The DELAY constant defined in the program examples is used to illustrate the differences between the constant definitions. In the left column below, the DELAY is defined as 50,000 using the* EQU *directive, while in the right column the value is stored in memory using the* DW *directive. In both columns, register* R15 *will be loaded with this value* 50000 = 0xC350. *However, in the first case we use immediate mode, while in the second case we can use either absolute or direct mode, because here DELAY has the value of the address where the datum is stored.*

```
DELAY EQU 50000 | DELAY DW 50000
...             | ...
mov #DELAY,R15  | mov DELAY,R15
                | or
                | mov &DELAY,R15
```

Memory locations can be assigned to constants using the appropriate *data definition or data allocation* directives using this format.

Variables are names given to those labels attached to memory assignment directives in the RAM section, pointing to addresses of memory data cells whose contents will be changed during the program execution. If the variable is initialized, we use one of the directives in Table 4.12. If we only separate memory segments, then we use the directives in Table 4.13 with the format

```
LABEL    DIRECTIVE    number-of-spaces-allocated
```

For example, "table DS8 0xA" reserves space for ten bytes, and "table DS16 0xA" for ten words.

Example 4.21 *A pulse is sent to pin 2.5 to start a device, which will deliver ten signed byte size data, once at a time. The device sets a flag at pin 2.6 to indicate a datum is ready. The data is retrieved from Port 1 and stored in RAM as 16-bit sized data. The ten data are also added. The partial listing below achieves this objective. I/O ports configuration is not included.*

Table 4.13 Initialized Data definition or allocation IAR directives

Directive	Use
DS8 or DS	Allocates space for 8-bit data.
DS16 or DS 2	Allocates space for 16-bit data.
DS32 or DS 4	Allocates space for 32-bit data.
DS64 or DS 8	Allocates space for 64-bit data.
.float	Allocates space for 48-bit floating point constants in TI Format.

```
          ORG     0X0200
TABLE     DS16    0x0A          ; reserves 10 word spaces in RAM
SUM       DW      0             ; initialize addition

- - - - - - - -
;       The following goes somewhere in the code component

          mov     #0,R12        ; pointer to table start
          bis.b   #BIT5,&P2OUT  ; one cyle
          nop                   ; pulse
          bic.b   #BIT5,&P2OUT  ; to pin P2.5
          mov     #10,R15       ; data

POLL_lp:  bit.b   #BIT6,&P2IN   ; test until
          jz      POLL_lp       ; ready
          mov.b   &P1IN,R4      ; retrieve datum
          sxt     R4            ; extend to 16-bits
          mov     R4,TABLE(R12) ; store datum
          incd    R12           ; point to next item
          add     R4,SUM        ; add datum
          dec     R15           ; retrieve next datum
          jnz     POLL_lp       ; until finish.
```

4.7.3 Interrupt and Reset Vector Allocation

The compulsory allocation of the the reset vector at address 0xFFFE was briefly discussed in Sect. 4.2. We also need to allocate interrupt vectors, so the CPU can go to the corresponding ISR when requested.

Each peripheral and I/O port with interrupt capability has an assigned address for its interrupt vector in the interrupt vector table. Table 4.14 shows the interrupt vector table for devices in the MSP430x20x3 family.

The format to allocate these vectors in an assembly program is the same as that for the reset vector, using the appropriate address from the interrupt vector table:

```
ORG   0xFFFE                ;reset vector address
DW    <<ResetVectorLabel>>   ;Label used by user
ORG   <<Assigned address>> ;address of ISR
DW    <<IntVectorLabel>>   ;Label used by user for ISR
```

Table 4.14 Interrupt vector table for MSP430x20x3 devices.

Vector address	Standard name	Source of interrupt	Interrupt flags	System interrupt
0xFFE0	– –	Unused		
0xFFE2	– –	Unused		
0xFFE4	PORT1_VECTOR	I/O Port 1[a]	P1IFG.0 to P1IFG.7	Maskable
0xFFE6	PORT2_VECTOR	I/O Port 2[a]	P1IFG.6 to P1IFG.7	Maskable
0xFFE8	USI_VECTOR	USI[a]	USIIFG, USISTTIFG	Maskable
0xFFEA	SD16_VECTOR	SD16_A[a]	SD16CCTL0 SD16IFG and SD16OVIFG	Maskable
0xFFEC	– –	Unused		
0xFFEE	– –	Unused		
0xFFF0	TIMERA1_VECTOR	Timer_A2[a]	TACCR1 CCIFG and TAIFG	Maskable
0xFFF2	TIMERA0_VECTOR	Timer_A2	TACCR0 CCIFG	Maskable
0xFFF4	WDT_VECTOR	Watchdog Timer+	WDTIFG	Maskable
0xFFF6	– –	Unused		
0xFFF8	– –	Unused		
0xFFFA	– –	Unused		
0xFFFC	NMI_VECTOR	NMI	NMIIFG	Non maskable
		Oscillator Fault	OFIFG	
		Flash memory access violation[a]	ACCVIFG	
0xFFFE	RESET_VECTOR[a]	Power-Up	PORIFG	Reset
		External Reset	RSTIFG	
		Watchdog Timer+	WDTIFG	
		Flash Key Violation	KEYV	
		PC Out of range[b]		

[a]Multiple source flags
[b]PC out of range if it tries to fetch instructions from addresses 0h to 0x1FF, or unused addresses

Notice that each address has a standard name, which is defined in the header file. Thus, instead of `ORG 0xFFFE` we can write `ORG RESET_VECTOR`. The following example works the LED toggling with an interrupt driven program. The LED toggles each time the pushbutton is pressed. Chapter 7 discusses in detail the subject of interrupts and the mechanisms used by the CPU to identify an interrupt request source.

Example 4.22 *The listing in Fig. 4.26 shows an interrupt driven example for toggling the LED with a pushbutton; an internal resistor is used. The CPU is turned off and is awaken by the interrupt request that happens when the pushbutton at pin P1.3 is pressed. Observe the allocation of the reset and interrupt vectors at the end of the listing.*

```
1  #include  "msp430g2231.h"
2  ;-------------------------------------------------------
3              ORG     0F800h                  ; Program Reset
4  ;-------------------------------------------------------
5  RESET       mov.w   #0280h,SP               ; Initialize stac
6  StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ;Stop WDT
7  SetupP1     bis.b   #0xF7,&P1DIR            ;P1.0 output
8                                              ;unused P1 pins as output
9                                              ;Pin P1.3 input by default
10             bis.b   #0x4,&P1REN             ;P1.3 Resistor enabled
11             bis.b   #0x4,&P1OUT             ; as pullup resistor
12             bis.b   #0x4,&P1IE              ;enable interrupt at P1.3
13
14 SetupP2     bis.b   #0FFh,&P1DIR            ;Unused P2 pins as output
15
16 Mainloop
17             bis.w   #CPUOFF+GIE,SR          ;Turn off CPU, and
18                                             ; enable interrupts
19             nop                             ; Breakpoint for assembler
20             jmp     Mainloop                ; Again
21 ;-------------------------------------------------------
22 PORT1_ISR  ; Begin ISR
23 ;-------------------------------------------------------
24             bic.b   #BIT3,&P1IF             ; Reset Interrupt Flag
25             xor.b   #0x04,&P1OUT            ; toggle LED
26             reti                            ; return from interrupt
27 ;-----------------------------------------------------------------------
28 ;           Interrupt Vectors
29 ;-----------------------------------------------------------------------
30             ORG     RESET_VECTOR            ; MSP430 RESET Vector
31             DW      RESET                   ;
32             ORG     PORT1_VECTOR            ; Port 1 Int. Vector
33             DW      PORT1_ISR
34             END
```

Fig. 4.26 Listing with an interrupt vector allocation

4.7.4 Source Executable Code

Let us add some comments on the executable code. Remember that this is the set of CPU instructions. We usually place this code in the flash-ROM section of the memory space. Let us focus now on some characteristics of the main algorithm.

Main Algorithm: After housekeeping and hardware configuration, we may start writing the instructions for the intended task. If interrupts are being enabled, the `eint` should be introduced after the hardware configuration, to avoid unwanted interruptions.

Since most, if not all, embedded systems work continuously and independently of a user's intervention, many systems do not include a particular instruction to "stop" the program. MSP430 microcontrollers fall in this category. Instead, two common methods to "terminate" the main code are with an infinite loop through an unconditional jump or sending the system into a low power mode, which turns the CPU off.

Examples of how to make an infinite jump have been provided in this chapter, like for example the main code loop in Fig. 4.6. Terminating the main program with a low-power mode requires a few additional considerations, like determining the events that will bring the CPU back into activity and configuring and activating their interrupts.

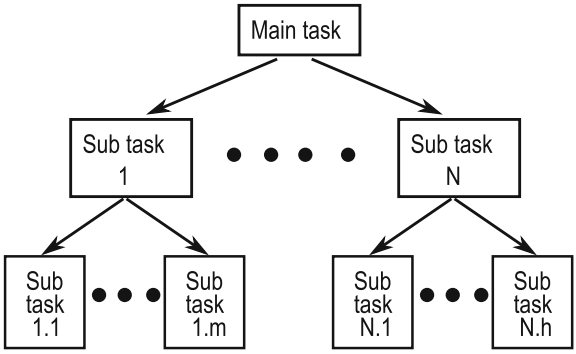


Fig. 4.27 Modular programming concept

Low-power modes are extremely useful to reduce the power consumption in embedded designs. They can be achieved by configuring the SR bits CPUOFF, SCG1, SCG0 and OSCOFF, to produce one of four different CPU sleep modes with different power consumption rates, or the default active mode where the CPU and all enabled clocks and peripherals are active. When the CPU is powered-up, it begins operation in active mode.. Low-power modes need to be explicitly activated to take effect. Chapter 7 provides a detailed discussion of how to use low-power modes in MSP430 microcontrollers.

4.8 Modular and Interrupt Driven Programming

To simplify the design and debugging of large programs, it is a good practice to decompose it in short modules, called *subroutines* or *functions*. This is the so called *modular approach* illustrated by Fig. 4.27. We may include in this practice interrupt driven programming in which the particular Interrupt Service Routines (ISR) are intended to respond mainly to hardware interrupt requests, but may also obey to software requests; in this case they are called *software interrupts*. An interrupt driven program is one which exclusively responds to interrupt requests.

4.8.1 Subroutines

If a piece of code is used several times in a program, it may be worth writing a subroutine for it. It makes the main code easier to write and understand, and makes better use of resources. In addition, if it is the type of code that can be used in many different applications, including the subroutine in a library is very advantageous. We call this a *reusable function or subroutine*.

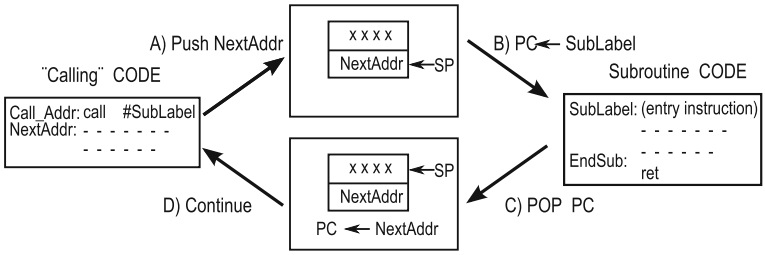


Fig. 4.28 The program flow after Call and ret instructions

In short, an assembly subroutine or function is a collection of instructions, a process by itself, separated from the main code and ending with the instruction `ret`, to return to the place in memory from which the subroutine was called. The executable subroutine code appears only once in memory and is invoked each time it is needed with the instruction `call dest`. The operand is the address of the entry line of the subroutine, and it will be loaded into the PC. Any valid addressing mode can be used for `dest`.

Subroutines should be as short as possible, and focus on doing one and only one task. It is common to think of them as reusable, since different programs may need the task programmed in this function. They can be included in the source file or in libraries.

The subroutine execution process is illustrated with Fig. 4.28 using the immediate mode for the destination operand. The call process takes five cycles and the return process seven cycles. Most of the time, this time expense is a very good investment when compared to the advantages. The steps shown are:

(a) The instruction **call** pushes the current contents of the PC register, which has the address of the next instruction (NextAddr), and then loads the program counter with the address SubLabel of the entry line of the subroutine.

(b) The fetched instruction in the following cycle is then the first one of the subroutine. The CPU continues with the sequence of instructions in the function, up to the last instruction **ret**, which means "return from subroutine".

(c) This emulated instruction pops back NextAddr into PC

(d) Hence, the next instruction to be fetched is the one after the call instruction in the main program.

This sequence is followed every time that the subroutine is invoked. Because of the "hidden" push and pop operations in this process, it is of extreme importance to verify that the register SP is pointing at the right address when the returning instruction is executed. This requires a good use of the `push` and `pull` instructions within the function, or additional manipulation of the SP before exiting the subroutine.

Data Passing and Local Variables and Constants

Remark that the CPU and memory resources are shared between the main code and subroutines. Therefore, any register and memory location keeps the same contents before and after calling subroutines, except for the program counter. This is something to consider both for *passing data* and for local variables and constant values.

The process of passing data consists of providing a subroutine data information needed to realize the intended algorithm, and returning results from the subroutine to the calling module. Local variables and constants are those used exclusively in the subroutine, but not outside of it. When a subroutine is planned, it is necessary to define how both items will be managed, especially for reusable functions. We discuss these points next.

Local Variables and Constants

Some recommendations and measures presented below may be skipped for subroutines proper to the program and not designed to be in a library, assuming that good care is taken to avoid difficulties.

- The fastest and simplest way to introduce local values or results is to use registers. Several designers suggest to use registers R4–R11, but you may proceed as necessary. In the subroutine code, start by pushing onto the stack all registers used locally and pop them in reverse order before returning.
- Assign a RAM segment for local variables. These may be overwritten outside the subroutine.
- Local constants not supposed to change, may be defined in ROM space. If they are only initial values, and may change, use a RAM address.
- The stack is a valid segment for local values if accompanied with appropriate manipulation of the SP register. For many designers, it is the best option after register use.

If you make a habit to use a specific set of registers for local variables and another set for passing data, these may or may not be pushed or popped, depending on the strategy needed.

Example 4.23 *A subroutine reverses the bit order in register* R11*—data passed to function in* R11*. The routine returns as output the original* R11 *value and the reversed word in* R12*—results returned in* R11 *and* R12*. To iterate, we use* R15 *as a local counter or variable initialized with* 16*, the number of bits to be reversed. The following code does the work.*

```
INIT:   push R15      ; save R15 for local use
        push R11      ; To recover R11 at output
        mov #16,R15   ; load local counter
LOOP:   rla R11       ; msb to carry
```

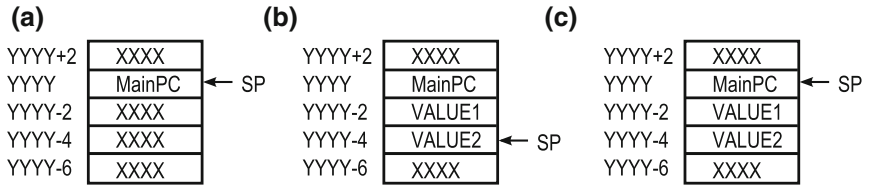


Fig. 4.29 Illustration of stack for local constants

```
        rrc R12       ; push it at right on R7
        dec R15       ; if not finished
        jnz LOOP      ; go for next bit
        pop R11       ; recover original value
        pop R15       ; retrieve register used for local
                        value
_ENDS:  ret           ; return from subroutine
```

When using the stack for local variables, each time a word is pushed, it will be referred to by the address X(SP), X = 0, 2, 4, …, in a reverse order to that being pushed. Before returning, the instruction `add #2*n,SP`, with *n* being the number of local variables pushed, will return SP to the correct address to point to. This is illustrated by Fig. 4.29 for two local values VALUE1 and VALUE2.

Figure 4.29a shows the stack just after the call, with SP pointing at address YYYY to the returning address. Inset (b) shows the situation after pushing both local constants. Notice that VALUE2 is at address 0(SP) while VALUE1 at 2(SP). Finally, (c) shows the situation after the SP has been adjusted for return with `add #4,SP`. Let us change the subroutine of the previous example using the stack for the local variable. The listing would go then like this:

```
INIT:   push R11      ; To recover R11 at output
        push #16      ; load local counter
LOOP:   rla R11       ; msb to carry
        rrc R12       ; push it at right on R7
        dec 0(SP)     ; if not finished
        jnz LOOP      ; go for next bit
        add #2,SP     ; adjust SP
        pop R11       ; retrieve register used for local
                        value
_ENDS:  ret           ; return from subroutine
```

Passing Data

Data passed to the subroutine or function, also called *parameters*, is most commonly transferred to and from a function

- By register
- By memory
- By stack

The preferred method is by register, since it is faster and does not use memory. The MSP430 has many general purpose registers, making this method quite feasible in most cases. The other two methods consume memory space and are slower, but are also an option. Using the stack may be dangerous if no precaution is taken, as explained below.

Example 4.24 *Assume a data* VALUE *is to be passed to a subroutine starting at address* SUBR. *Assume that the first instruction in the function is to transfer the data to* R7*, followed by a byte exchange. Let us look at the three methods to pass the data.*

(a) If data is passed using register R6*, then the corresponding instructions in the main code and in the subroutine are as follows:*

Subroutine:	*Maincode*:
`SUBR: mov R6,R7`	`mov #VALUE,R6`
`swpb R7`	`call #SUBR`

(b) If a memory address, say SUBDATA *is used to transfer data to the subroutine, then the instructions are as follows:*

Subroutine:	*Maincode*
`SUBR: mov &SUBDATA,R7`	`mov #VALUE,&SUBDATA`
`swpb R7`	`call #SUBR`

(c) The stack for passing data is not the best option unless carefully used, especially when the subroutine is used frequently. And don't forget that embedded microcontrollers usually run in infinite loops, or are not supposed to be turned off during lifetime, always reacting to an event. A frequent format for passing data with the stack is as shown next:

Subroutine:	*Maincode*:
`SUBR: mov 2(SP),R7`	`push #VALUE`
`swpb R7`	`call #SUBR`

The problem with this approach can be appreciated with Fig. 4.30. We see in inset (b) that after return one memory stack space (two bytes) is exhausted. Even if this is the only subroutine in the program the stack, and hence the RAM space, will be depleted after a certain moment. One way to avoid this phenomenon is to add the instruction `incd SP` *to move the stack pointer up just after the call instruction. It is a small price to pay for not having the RAM depleted. This is illustrated in inset (c).*

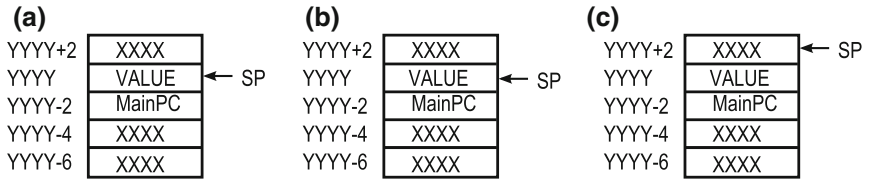


Fig. 4.30 Passing data with the stack. **a** Push value and call subroutine; **b** After return; **c** Saving memory after return

Passing data back to main code from subroutine may be done using similar procedures. Notice that the stack needs again special considerations, being that one reason why is not so popular for this task.

4.8.2 Placing Subroutines in Source File

Subroutines can be placed anywhere in the code section, as long as they end with `ret`. Although it is technically possible to include a subroutine within the main code, this is not at all recommended. The following example using polling illustrates placing of subroutines.

Example 4.25 *Each pin of port 1 of the MSP430 has an interrupt flag associated in the P1IFG register. The operation of this flag can be enabled with the Interrupt Enable Register (P1IE) and its operation configured with the Interupt Edge Select Register (P1IES). When bit* x *of P1IE is set, then bit* x *of the P1IFG register will be set every time that there is an appropriate voltage transition at pin* P1.x *of Port 1. The bit* x *of P1IES determines the type of transition: if* 1 *the transition is high to low, if* 0 *low to high. By default, all these registers are cleared. In this example we use P1IFG to do polling (Figs. 4.31, 4.32).*

Our objective is as follows: A LED is connected to pin P1.0, while a push button is connected to P1.3. We want to toggle the LED by pressing and releasing the push button. Since this action on the push button will produce a pulse, the flag P1IFG.3 will be then set. We poll then this flag and whenever it is set we will call a subroutine that changes the LED state.

Two listing examples for this algorithm are shown in Figs. 4.3 and 4.32, with the subroutine placed at different positions with respect to the main code.

Subroutines in Libraries

It is advisable to include reusable functions in a library with other functions of similar nature. In this case, the linker incorporates the object code of the function together with other codes, and gets the final version to be loaded onto the memory.

```
1  #include "msp430.h"
2  ;-------------------------------------------------------------
3                RSEG    CSTACK                  ; Stack segment
4                RSEG    CODE                    ; Start program code
5  ;-------------------------------------------------------------
6  RESET         mov.w   #SFE(CSTACK),SP         ; Initialize stackpointer
7  StopWDT       mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
8  SetupP1       mov.b   #0xF7,&P1DIR            ; P1.3 as input, P1.0 and
9                                                ; unused P1. pins as output
10               bis.b   #8,&P1REN               ; Resistor at P1.3
11               bis.b   #8,&P1OUT               ; as pullup resistor
12 IFEnab        bis.b   #8,&P1IE                ; enable P1.3 int flag
13               bic.b   #8,&P1SEL               ; flag set on low-to-high
14               bis.b   #1,&P1OUT               ; turn LED on.
15 Pollng_Lp     bit.b   #BIT0,&P1IFG            ; test for button
16               jz      Pollng_Lp               ; until pressed
17               call    #Toggle                 ; toggle LED
18               jmp     Pollng_Lp               ; poll again
19
20 ;-------------------------------------------------------------
21 ;                Subroutines
22 ;-------------------------------------------------------------
23 Toggle        bic.b   #BIT0,&P1IF             ; reset flag
24               xor.b   #BIT0,&P1OUT            ;
25                     ret                               ; return from sub
26 ;
27 ;-------------------------------------------------------------
28 ;            Interrupt Vectors
29 ;-------------------------------------------------------------
30               RSEG    RESET                   ; MSP430 RESET Vector
31               DW      RESET                    ;
32               END
```

Fig. 4.31 This listing has the subroutine after the main code, and uses immediate addressing mode when calling

4.8.3 Resets, Interrupts, and Interrupt Service Routines

Resets and interrupts were introduced in Chap. 3, Sect. 3.9 and considered briefly in this chapter in previous sections, including the manipulation of the GIE flag and the allocation of interrupt vectors. We deal now with some additional aspects considerations in the assembly source. We start by looking at general issues and then go to some specifics.

MSP430 Interrupt Process

When the request for an interrupt is received, with flag GIE active for maskable interrupts to be enabled, the MSP430 processor responds with the following sequence, which is illustrated in Fig. 4.33:

1. It completes the instruction being currently executed, if any.
2. The PC and the SR registers are pushed, in that order, onto the stack. In this way, the state of the CPU is preserved, including the operating power mode before the interrupt.
3. The SR is cleared, except for the OSCOFF flag, which is left in the current state. This means that the system is taken into the active mode.

```
1  #include "msp430.h"
2  ;-------------------------------------------------------------
3                RSEG    CSTACK                  ; Stack segment
4                RSEG    CODE                    ; Start program code
5  ;-------------------------------------------------------------
6  ;                          Subroutines
7  ;-------------------------------------------------------------
8  Toggle       bic.b   #BIT0,&P1IF             ; reset flag
9               xor.b   #BIT0,&P1OUT            ; toggle LED state
10              ret                             ; return from sub
11 ;-------------------------------------------------------------
12 ;                  Main code
13 ;-------------------------------------------------------------
14 RESET        mov.w   #SFE(CSTACK),SP         ; Initialize stackpointer
15 StopWDT      mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
16 SetupP1      mov.b   #0xF7,&P1DIR            ; P1.3 as input, P1.0 and
17                                              ; unused P1 pins as output
18              bis.b   #8,&P1REN               ; Resistor at P1.3
19              bis.b   #8,&P1OUT               ; as pullup resistor
20 IFEnab       bis.b   #00001000b,&P1IE        ; enable P1.3 int flag
21              bic.b   #4,&P1SEL               ; flag set on low-to-high
22              bis.b   #1,&P1OUT               ; turn LED on.
23 Pollng_Lp    bit.b   #BIT0,&P1IFG            ; test for button
24                      jz      Pollng_Lp       ; until pressed
25                      call    #Toggle         ; toggle LED
26                      jmp     Pollng_Lp       ; poll again
27 ;-------------------------------------------------------------
28 ;            Interrupt Vectors
29 ;-------------------------------------------------------------
30              RSEG    RESET                   ; MSP430 RESET Vector
31              DW      RESET                   ;
32              END
```

Fig. 4.32 This listing has the subroutine after the main code, and uses symbolic addressing mode when calling

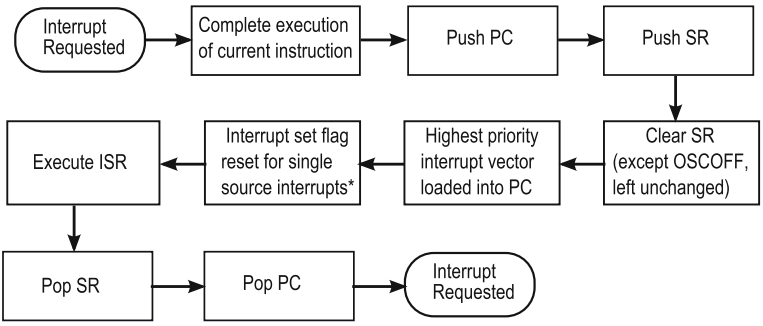


Fig. 4.33 Sequence of events during interrupt processing

4. The highest priority interrupt vector is loaded into the PC. For single source interrupts, the interrupt flag is reset. Multiple sources interrupt flags should be reset by software, and priority established within the ISR.
5. The Interrupt Service Routine (ISR) is executed. When the reti instruction is executed, it will pop the SR and PC, returning to the place where the interrupt occurred, restoring also the previous controller state.

All the steps in the sequence above, just before the ISR execution, are automatic, transparent to the programmer in the sense that no code is require for them to take

place. For this sequence to take place, we already identified in Sect. 3.9.2 four fundamental provisions that need to be made at the software level, for interrupts to work: Providing a stack, having an ISR, configuring the vector table, and enabling interrupts at the global and local levels. These issues are discussed in detail in Chap. 7. Here we remark them to put them in perspective within the context of designing assembly language programs.

Remarks on Interrupt Service Routines

Service routines are planned and written by the programmer. They contain the instructions that service the device that requested the interrupt. ISRs must end with the instruction `reti`, for "return from interrupt", which is responsible for pulling back the SR and PC registers from the stack to finish the interrupt sequence. When writing ISRs, the code shall be kept as short as possible. If a long processing is necessary, send most of it to the main code. Also, unless it is needed, avoid nested interrupts. The CPU, by default takes care of that. And lastly, do not forget the problem of data sharing. Define carefully what data is passed and any local variables or values.

Additional recommendations are discussed in Chap. 7. In particular Sect. 7.3 provides general considerations for ISRs and interrupt driven software from a broader design perspective.

4.9 Summary

- There are three levels for programming: High level, assembly level, and machine language programming. Compilers and assemblers are used to translate the first two into machine instructions, native to the CPU.
- MSP430 CPU machine language instructions consist of one to three 16-bit words. The leading word is the instruction word, in which a group of bits is the opcode and the rest of the bits determine the operands and addressing modes. The CPU has 27 core instructions.
- Assembly instructions are in a one to one correspondence with machine instructions and have as components mnemonics and operands. The mnemonics is the assembly name given to the opcode. The operands are stated in an addressing mode syntax.
- Two pass assemblers have directives that allow us to use labels, comments, and symbolic names to help us in writing and reading the source program. The format of an instruction is
  "`label mnemonics source,destination ;comment`,
  where the label and comment are optional. The operands depend on the instruction.
- The label in an instruction is a constant that takes, after compilation, the address of the instruction as its value.

- The MSP430 assemblers accept 24 emulated instructions with mnemonics easier to remember than their core counterpart because they are easier to associate to the operation. There is no penalty in using them.
- There are seven addressing modes for the MSP430: (1) register mode – Rn –, (2) immediate mode –#X –, (3) indexed mode – X(Rn) –, (4) indirect mode – @Rn –, (5) Indirect auto increment – @Rn+ –, (6) direct mode – X –, and (7) absolute mode – &X –. The immediate, indirect, and indirect autoincrement modes are not valid in the destination. Any register is valid as an operand.
- Most instructions may work with word size or byte size operands. This is indicated by adding a suffix to the mnemonics: ".w" for words, ".b" for bytes. Word size is the default.
- In byte operations with register mode, only the LSB is considered. When the register is the destination, the MSB is cleared.
- The source program must include allocation of reset vector and interrupt vectors for each ISR present in the code.
- The executable code in the source must include hardware configuration and stack pointer initialization. Interrupt enabling should be done only after the system is ready to work.
- The source program may be designed in absolute or relocatable form. In the first case, using the `ORG` directive, the user must know the memory map of the microcontroller being used.
- When placing the CPU in a low power mode, only a reset or an interrupt request will awaken the CPU. The ISR may do the service or else may return the PC to the main code with an appropriate strategy.
- Macros and subroutines help write shorter and more efficient codes.

4.10 Problems

4.1 Answer the following questions

a. What are the programming levels used?
b. What relationship exists between assembly and machine language?
c. Describe the basic format of an assembly instruction with two and one operands.
d. What is an assembler?
e. What is a debugger and what is it used for?
f. Why is R3 used as a constant generator and how does it function?
g. What is a source program or source file?
h. What is the difference between directives and instructions?
i. What is the general format of a source statement?
j. What is meant by "source executable code"?

4.2 In the MSP430 machine language, the instruction word with two operands is decoded in four nibbles, where the most significant nibble is the opcode.

For register modes in the operand, the least significant nibble is the number for the destination register, and the nibble next to the opcode is the source register. Consider now assembly and hex machine versions for the executable code in Fig. 4.5 and answer the following questions:

a. How many two operand instructions are there? (You have to identify emulated ones!)
b. For each two operand instruction, associate the mnemonic with the corresponding opcode.
c. In the format Rn, what value of n corresponds to register SP?
d. Does `dec.w R15` have a source register? If affirmative, tell which one is it and why is it there. If negative, justify your answer.
e. What is the machine instruction word for `mov.w #0xC350, R10` and `dec R6`?

4.3 Example 4.1 illustrates what value was assigned to label RESET as well as how the addresses for instructions are assigned once given the reset vector. Using the machine code of Fig. 4.6 as a reference, find the values that are assigned to each label of the source code.

4.4 Verify that the **jge** instruction works correctly by testing $N \oplus V = 0$ when there is an overflow.

4.5 Eleven byte sized signed numbers are stored in RAM, starting at address 0x200. The numbers are to be added and the result stored in the next even address available in RAM. Write a code to do this and include it in a program. Test your source with the set $-120, +38, -57, -110, +18, -97, +60, +85, -125, 78$, and -1.

4.6 Example 4.11 illustrates an algorithm to convert from binary to BCD, using the decimal addition. The example converts an 8-bit binary number. Modify the algorithm to convert 16-bit binary numbers, and store the result in memory.

4.7 Multiplication by 10 in binary can be done thinking in $A \times 10 = 2^3A + 2A$ or $A \times 10 = (2^2A + A) \times 2$. Write a subroutine to multiply a nibble by 10 and use it in nested multiplication of the power expansion form of a decimal number to convert a BCD number to its binary equivalent. Limit the numbers to 9999, so a 16-bit register can hold the result.

4.8 Neither multiplication nor division are supported by the MSP430 CPU ALU. For multiplication some microcontroller models include hardware multiplier peripherals which can be used; otherwise, multiplication must be realized by software.

Both multiplication and division of unsigned numbers can be realized by direct brute force methods. That is

Multiplication: To multiply AxB, add A times B.

Division: To divide A/B, subtract B from A as many times as possible until the difference is smaller than B. The difference is the residue R and the number of subtractions is the quotient Q.

a. Test your understanding by multiplying 11x3 and dividing 11/3 by hand with the algorithms.
b. If each factor in a multiplication is n bits wide, what is the number of bits that we must reserve for the product?
c. Write pseudo codes or draw a flowchart for each operation, and encode them in assembly language. For the multiplication, assume both factors are bytes. For the division A/B, assume A is a 16-bit word and B a byte.
d. Test your code in the CPU (you may use the simulator).

4.9 The sum algorithm for multiplication is not efficient. There are several other multiplication algorithms available which are much better. One of them is an old one, used already in ancient times, and known as the *Russian Peasant Algorithm*. Two multiply AxB, the algorithm can be stated as follows:

Step 1 **IF** A is even, **THEN** initialize P = 0. **ELSE**, P = B.
Step 2 While A≥1 do
2.1 A←A/2 and B←2×B
2.2 **If** A is even **THEN** P←P+B

In this algorithm, A/2 is the integer quotient, discarding the residue or fractional part.

a. Test your understanding of the algorithm by multiplying 34 × 27 by hand in decimal terms.
b. Multiplying two n-bit unsigned numbers, A and B, the result is a number P with at most $2n$ bits. With this in mind, write a pseudocode or draw a flowchart to implement the algorithm and implement it in assembly language for byte operands.
c. Repeat the above procedure for 16-bit operands.
d. Test your codes in the microcontroller (You may use the simulator).

4.10 For integer division M/D, D≠0 and M being an n-bit word, yielding a quotient Q and remainder R, one algorithm that mimics long division is the following:

Step 1: Initialize Q = 0, R = 0
Step 2: **FOR** $j = n - 1$ to 0 **DO**
2.1 R← 2R + M(j)
2.2 **IF** R≥D **THEN**
a. R ← R − D
b. Q(j) = M(j)
ENDIF

In this algorithm, M(j) means the bit of N in the j-th position.

a. Write an assembly code for the above algorithm and test it. Assume M is a 16-bit word.
b. Use your code as a function in a program to convert a 16-bit unsigned binary word into a BCD number.