```
                JMP         subroutine1
                 :
                 :

subroutine1:     :
                 :
                 :
                RET
```

Provided in Figure 2.6 is a summary of the MSP430 assembly language instruction set.

2.8 ADDRESSING MODES

Earlier, we mentioned that each assembly instruction has an opcode and an operand. The opcode tells hardware which operation should be executed. The operand specifies how the data necessary to execute the particular operation can be found. Addressing modes are different methods used to identify necessary data for assembly instructions. There are seven different addressing modes in the MSP430 microcontroller. We present them next.

2.8.1 REGISTER ADDRESSING MODE

For this addressing mode, the data needed to execute an instruction are the contents of registers. For example, to execute the following instruction

```
MOV.W   R5,R7
```

contents of register R5 are moved to register R7.
In the following instruction

```
AND.B      R5,R7
```

to execute the AND operation, the MSP430 takes the contents of R5 and R7 and performs bit wise AND operation and stores the result in R7.

2.8.2 INDEXED ADDRESSING MODE

In this addressing mode, the address of the data necessary for an instruction is found by adding an offset value to the contents of a register. For example, suppose you have the following instruction.

```
ADD.W   5(R4), R5
```

The instruction takes the data located at the address specified by 5 plus the contents of R4, adds the value to the value in R5 and stores the result to R5.

Mnemonic		Description		V	N	Z	C
ADC(.B)†	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR†	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)†	dst	Clear destination	0 → dst	–	–	–	–
CLRC†		Clear C	0 → C	–	–	–	0
CLRN†		Clear N	0 → N	–	0	–	–
CLRZ†		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)†	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)†	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)†	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT†		Disable interrupts	0 → GIE	–	–	–	–
EINT†		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)†	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)†	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)†	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–
JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP†		No operation		–	–	–	–
POP(.B)†	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP – 2 → SP, src → @SP	–	–	–	–
RET†		Return from subroutine	@SP → PC, SP + 2 → SP	*	*	*	*
RETI		Return from interrupt		*	*	*	*
RLA(.B)†	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)†	dst	Rotate left through C		0	*	*	*
RRA(.B)	dst	Rotate right arithmetically		*	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)†	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	–	–	–	1
SETC†		Set C	1 → C	–	1	–	–
SETN†		Set N	1 → N	–	–	1	–
SETZ†		Set Z	1 → C	–	–	–	1
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign	dst + 0FFFFh + 1	0	*	*	*
TST(.B)†	dst	Test destination		0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

† Emulated Instruction

Figure 2.6: Texas Instruments MSP430 assembly language instruction set. Used with permission of Texas Instruments.

2.8.3 SYMBOLIC ADDRESSING MODE

To those who are familiar with other microcontrollers' addressing modes, this addressing mode is similar to the program counter (PC) relative addressing mode. The data necessary for an instruction is found by finding the relative offset from the current instruction to a destination location. The contents of the source address (contents of PC + X) are moved to the destination address (contents of PC + Y).

2.8.4 ABSOLUTE ADDRESSING MODE

In this addressing mode, the contents of a memory address are used as the data necessary for an instruction. For example, suppose we have

```
MOV.W    ABC, R10
```

The instruction moves the contents of address ABC to register R10.

2.8.5 INDIRECT REGISTER ADDRESSING MODE

In this addressing mode, the contents of an address are used as the address where data for the instruction is found. For example,

```
MOV.W    @R5, R10
```

moves the contents of address location whose address value was determined by the contents of R5 are moved to register R10.

2.8.6 INDIRECT AUTOINCREMENT ADDRESSING MODE

In this addressing mode, the instruction first performs its task using the register indirect addressing mode and then increments the contents of the register by either one for a byte operation and two for a word operation.

```
MOV.B    @R4+,R5
```

The above instruction takes the contents of R4, uses it as the address location to find the data, moves the data to R5, then increments the contents of R4 by one.

2.8.7 IMMEDIATE ADDRESSING MODE

In this addressing mode, the actual number specified by symbol # is used by an instruction.

```
mov.w  #78F2h, R4
```

This instruction moves 78F2h to R4 where h represents a hexadecimal number.

Examples: Assume that the following instructions are not sequential. Given that the contents of R5 is $(128)_{10}$ and R6 is CAFEh, describe the results of each instruction.

1. ADD #10, R5

   Answer: Add 10 to the contents of register R5.

2. ADD.B #10, R5

   Answer: Add 10 to the low byte of R5.

3. MOV #AA55h, R4

   Answer: Places AA55h into register R4.

4. AND R5, R4 (use R4 from the previous example)

   Answer: Performs the bit-wise AND of R5 and R4 placing the result in R4 0000h.

5. BIC #FC00h, R6

   Answer: Clears the 6 most significant bits of R6 resulting in 02FEh.

6. BIS #FC00h, R6

   Answer: Sets the 6 most significant bits of R6 resulting in FEFEh.

7. CLR R5

   Answer: Set R5 to zero.

8. CLR.B R6

   Answer: Clears the low byte of R6 resulting in a value of CA00h.

2.9 PROGRAMMING CONSTRUCTS

Before closing this section, we present three principle programming constructs: a sequence, a loop, and a branch. A sequence represents a segment of a program, regardless of the programming language used, where instructions are executed in sequence, one after another in the order shown. A loop represents a segment of a program where the same instructions are executed a number of times specified by some condition. Finally, a branch allows programmers to implement IF-THEN-ELSE decisions in their programs. The three principle constructs are shown in Figure 2.7. How these three programming constructs are put together to write programs is the key to becoming a good programmer.

Example: Provided below is the basic assembly language construct for a loop.

```
Loop1:      :
            :
            :
            JMP Loop1
```

Example: Provided below is the basic assembly language construct for the if-then statement.

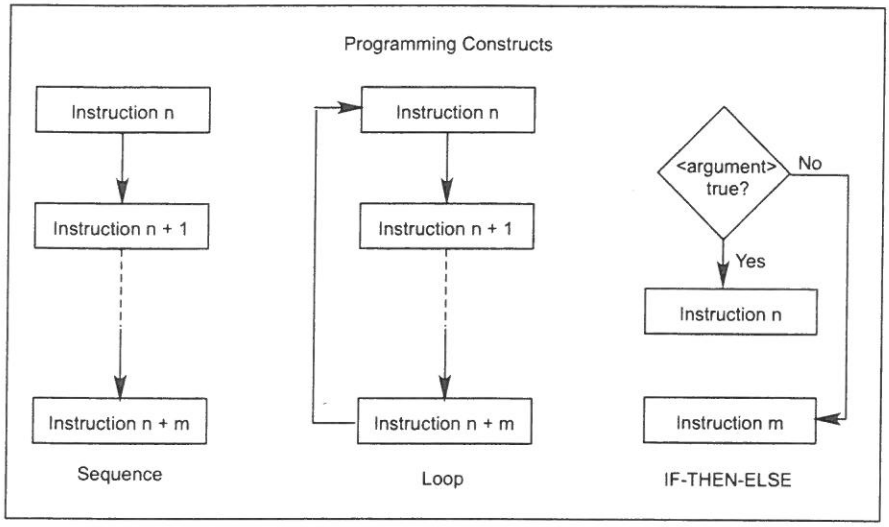

Figure 2.7: Programming constructs

```
                :
                :
               CMP   R6, R7    ;R6 = R7:
               JEQ   EQUAL1
                :

EQUAL1:         :
                :
```

2.10 ORTHOGONAL INSTRUCTION SET

Over the history of the computer industry, there have been two different approaches in designing a Central Processing Unit (CPU): the Complex Instruction Set Computer (CISC) and the Reduced Instruction Set Computer (RISC). The rationale for creating a CISC based computer is to increase computer performance by allowing programmers to write compact programs. This means hardware is designed to accommodate a set of specialized instructions, allowing a programmer to pick and choose appropriate instructions to perform a required task. In contrast, RISC computers are designed to execute only a small number of (thus, reduced) instructions. The related hardware is simple, which allows instructions to be executed in parallel, increasing the performance of a RISC-based computer. Over the past three decades the battle between the two approaches has been intense, but over time,

computer designers have extracted advantages of both approaches to create hybrid computers. Most of today's computers incorporate both CISC and RISC features.

The MSP430 architects wanted to design and develop a RISC-based controller. As mentioned, the controller has 27 unique instructions and 9 emulated instructions. In addition to the RISC-based design, MSP430 designers maximized the number of functional instructions by developing instructions that are orthogonal. The orthogonal instruction set means that each opcode (instruction) can use any of the MSP430's available addressing modes. This provides the programmer with tremendous flexibility in writing the code.

2.11 MEMORY MAP

The size of the MSP430's memory is up to 64K bytes or 2^{16} bytes[7]. As pointed out in Chapter 1, having 16 bit address bus means the controller can access 2^{16} different memory locations. Each memory location is made of eight separate bits, making each location one byte wide. Recall that symbol K is used to refer to memory size of 1024, not 1000. Thus, 64K means we have $64 \times 1024 = 65536$ bytes we can access in the MSP430 microcontroller. The 64K memory space is divided into several different sections all with different purposes. In fact, the electronic technologies used to create the sections are also different. The memory map of a computer shows how different sections of memory are configured, thus called a map of the memory. Figure 2.8 shows the memory map of a MSP430 microcontroller.

It is important for system developers and software programmers to know the memory map of a microcontroller, since it contains information about random access memory (RAM) space, read only memory (ROM) space, and the locations of bootstrap code, special registers, and non-available space. The MSP430 memory system will be discussed in Chapter 5.

2.12 ASSEMBLY VERSUS C

Due to limited onboard resources, including small memory, microcontrollers do not support the sophisticated software architecture of microprocessors. Namely, typical microcontrollers do not have operating systems running onboard. The actual programs that run onboard were written using an assembly programming language rather than high level languages used for laptop/desktop computers. Since the early 1990's, the use of high level languages, mainly C, increased for microcontrollers due to emerging compiler technologies that allowed programmers to be removed from learning the particular hardware and machine-level software architecture of a microcontroller. The compiler technologies, however, did not produce compact assembly code when compared to an assembly program written by a well-trained assembly programmer. Due to the advantages a high level language offers, discussed in the next section, the compiler technologies continued to improve. Today, the difference between a converted assembly program initially written in C and an assembly program written directly is very small.

[7] Some variants have a 20 bit address bus allowing the memory size to be up to 1M bytes or 2^{20} bytes.

Address(h)	Usage
0000 00FF	Peripherals - 4KB
1000 11FF	BSL Segment 0 (512 B)
1200 13FF	BSL Segment 1 (512 B)
1400 15FF	BSL Segment 2 (512 B)
1600 17FF	BSL Segment 3 (512 B)
1800 187F	User Info Segment D (128 B)
1880 18FF	User Info Segment C (128 B)
1900 197F	User Info Segment B (128 B)
1980 19FF	User Info Segment A (128 B)
1A00 1BFF	Factory Data (512 B)
1C00 5BFF	RAM (16 KB)
5C00 FF7F	Program
FF80 FFFF	Interrupt Vectors

Figure 2.8: Memory map of a typical MSP430.

2.12.1 ADVANTAGES/DISADVANTAGES

So why should anyone learn to write assembly language programs? After all, high level languages, in particular, the C language, are programmer friendly, portable, and compact. Furthermore, most programmers are already familiar with high level languages, removing the time required to learn a new assembly language. A C program can be machine independent (portability).[8] Although programs usually become more compact when written in an assembly language, the primary reason for most microcontroller programmers who choose to use a high level language is that a programmer does

[8]This statement is marginally true if specific ports and registers for a platform are not used in a program.

not need to understand the Instruction Set Architecture (ISA) to write programs for a particular platform, while having access to bit-by-bit level instructions, if necessary.

The proponents of assembly language programs point out the advantage of writing programs with instructions that can directly map to designed functions of a hardware platform. That is, each instruction is written as a specific machine level instruction, allowing a programmer to have full control of the execution of those instructions in the use of time (clock cycle by clock cycle) and hardware resources. This leads to writing an efficient program compared to the one written with a high level language and later converted to assembly code. Writing programs at the level of machine language also allows programmers control over where his/her programs will reside in memory, optimizing the use of available memory of a microcontroller. Writing assembly language programs also allow programmers to embed pertinent error messages in their programs at the machine level, while high level language programmers do not have the same option. Finally, programming at the machine specific ISA level allows assembly language programmers to better understand related computer architecture issues, which enables them to take full advantages of the particular hardware and software features of each microcontroller.

2.12.2 OUR APPROACH

Acknowledging the advantages offered by both high level languages and assembly languages, we use both languages in this book. We use more assembly language in earlier chapters as we present hardware architecture and resources. Once we establish the necessary foundation of the hardware and software systems, we use the C programming language to concentrate on functional capabilities of the MSP430 microcontrollers. We assume the reader is already familiar with the C programming language. A brief overview of C programming is provided in the Appendix.

2.13 SOFTWARE PROGRAMMING SKILLS

In this section, we present a system design approach called Top-Down Design and Bottom-Up Implementation. The overall idea of this approach is to break down a given task (your program should be implemented to execute the task) into smaller pieces or subtasks, solve each of the smaller subtasks, and then integrate them together one at a time. Of course if any smaller piece is too big, you should repeat the process until the subtask at the lowest level is defined by a set of simple operations. Recall from Chapter 1 that a structure chart is an appropriate tool to use in this process.

The top-down approach is sometimes called the *divide-and-conquer* method with two immediate benefits. The first one is that you reduce the complexity of the overall task by only concentrating on a simple subtask at a time. The second advantage is the by-product of the first one: test and evaluation of smaller tasks are easy and time saving. In addition to the two advantages, this approach makes it easy to integrate the subtask solutions, making the overall efforts and time spent to perform the original task of writing a program minimal.

We conclude this section with a step-by-step procedure to use the Top-Down Design and Bottom-Up Implementation approach.