CPE 323
MODULE 03
MSP430 INSTRUCTION SET ARCHITECTURE (ISA)

Aleksandar Milenković

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module introduces the MSP430 Instruction Set Architecture (ISA). MSP430 is a family of microcontrollers designed by Texas Instruments. It is a system-on-a-chip encompassing a processor core, flash memory, RAM memory, and a number of IO peripheral devices including parallel ports, timers, serial communication interfaces, analog-to-digital and digital-to-analog converters, LCD display controllers, and others. To write programs in assembly or C/C++ programming languages, engineers need to have a good understanding of processor's ISA - an interface between hardware and software. An ISA encompasses the following aspects of processor: registers, memory, data types, addressing modes, instruction set, instruction encoding, and exception mechanism.

Objectives

Upon completion of this module learners will be able to:

- *Specify ISA and its aspects (registers, memory, data types, addressing modes, instruction set, instruction encoding, and exception mechanism)*
- *Distinguish between logical and physical organization of memory in general and MSP430 in particular*
- *Demonstrate comprehension of all aspects of MSP430 ISA*

Contents

1 Introduction

Computers cannot directly execute high-level language (HLL) constructs found in C or C++. Rather, they execute a relatively small set of machine instructions, such as, addition, subtraction, Boolean operations (bitwise or, xor, and), and data transfers. The statements from a HLL are translated into sequences of machine code instructions by a compiler. The compiler is a program that takes source code written in a HLL and produces an equivalent sequence of machine code instructions. A machine code instruction is represented by a binary string. Reading these binary strings is hard for humans. A human-readable form of the machine code is assembly language. In addition to machine instructions, assembly language may also include some constructs that help programmers write a better and more efficient code, faster.

Instruction set architecture, or ISA for short, refers to a portion of a computer that is visible to low-level programmers, such as assembly language programmers or compiler writers. It is an abstract view of the computer describing what it does rather than how it does it. Note: Computer Organization describes how the computer achieves the specified functionality, but it is out of scope in this course. The CPE 221 and CPE 431 courses cover computer organization topics of interest.

The ISA aspects include:

(a) class of ISA,

(b) memory model,

(c) registers,

(d) types and sizes of operands,

(e) instruction set - data processing and control flow operations supported by machine instructions,

(f) instruction encoding,

(g) addressing modes, and

(h) exception processing (this topic is further discussed in Module 06).

In this module we will discuss the ISA aspects for the MSP430 family of devices.

Before we start discussing the MSP430 ISA, let us first introduce MSP430. MSP430 stands for Mixed-Signal Processor and it is a microcontroller family developed by Texas Instruments. An MSP430 microcontroller is a system-on-a-chip – a monolithic piece of silicon that combines a 16-bit low-power processor core, Flash and RAM memories, and a rich set of analog and digital input/output (I/O) peripherals. The common peripherals include digital input/output ports, serial communication interfaces supporting various serial communication protocols, timers, liquid crystal display controllers, analog-to-digital converters (ADCs), digital-to-analog converters (DACs), comparators, and even operational amplifiers. The MSP430 family targets low-cost and low-power battery-powered embedded applications. Microcontrollers from the MSP430 family are used in a range of consumer and industrial applications, such as utility metering, medical instruments, wearables, home appliances, and others. The MSP430 family

includes over 230 parts that range from those that cost as low as 10 cents to those that cost over $10 in high volumes. The individual parts vary in the type and capacity of memory and the number and type of I/O peripheral devices.

Things to remember 1-1. ISA Definition.

Instruction Set Architecture specifies an interface between the hardware and software worlds and includes the following aspects:

(a) class of ISA,

(b) memory model,

(c) registers,

(d) types and sizes of operands,

(e) instruction set - data processing and control flow operations supported by machine instructions,

(f) instruction encoding,

(g) addressing modes, and

(h) exception processing (this topic is further discussed in Module 06).

When you are learning a new ISA, start by figuring out aspects of ISA described above.

2 Class of ISA

Virtually all recent instruction set architectures have a set of general-purpose registers visible to programmers. These architectures are known as *general-purpose register architectures*. Machine instructions in these architectures specify all operands in memory or general-purpose registers explicitly. In older architectures, machine instructions specified one or more operands implicitly on the stack – so-called *stack architectures*, or in the accumulator – so-called *accumulator architectures*. The stack architectures use PUSH and POP operations to put operands to and remove them from the stack, respectively, whereas all other operations find their operands on the stack. Thus, arithmetic-logic instructions do not specify operands explicitly as they are implicitly always on the top of the stack. In accumulator architectures, one operand is always defined implicitly in a special register called accumulator. The other operand is specified explicitly by the instruction.

There are many reasons why general-purpose register architectures dominate in today's computers. Keeping frequently used variables, pointers, and intermediate results of calculations in *registers* reduces memory traffic; improves processor performance since registers are much faster than memory; and reduces code size since naming registers requires fewer bits than naming memory locations directly. A general trend in recent architectures is to increase the number of general-purpose registers.

The following example illustrates how a simple statement in a HLL that computes Z = X + Y is implemented in the three architectures: the stack, the accumulator, and the general-purpose register.

Example 2-1. Stack, accumulator, and general-purpose register architectures.

Consider a problem where we have to compute Z = X + Y. All three variables are in memory. Below are code snippets in assembly for the stack, accumulator, and general-purpose register architectures.

Stack machine (instructions find their operands on the top of the stack, specified implicitly):

```
PUSH X  ; read X from memory and put it to the top of the stack (TOS)
PUSH Y  ; read Y from memory and put it to the TOS
ADD     ; pops two operands from the TOS, sums them up,
        ; and writes the result back to the TOS
POP Z   ; pops the result from the TOS and writes it to memory location Z
```

Accumulator machine (one operand is in the accumulator, specified implicitly):

```
LOAD X  ; Accumulator ← M[X]
ADD Y   ; Accumulator ← Accumulator + M[Y]
STORE Z ; M[Z] ← Accumulator
```

General-purpose register machine (2-address instructions):

```
MOV X, R5 ; R5 ← M[X]
ADD Y, R5 ; R5 ← M[Y] + R5
MOV R5, Z ; M[Z] ← R5
```

General-purpose register architectures can be classified into *register-memory* and *load-store architectures*, depending on the location of operands used in typical arithmetic and logic instructions. In *register-memory* architectures arithmetic and logic machine instructions can have one or more operands in memory. In *load-store* architectures only load and store instructions can access memory, and common arithmetic and logic instructions are performed on operands in registers. Depending on the number of operands that can be specified by an instruction, ISAs can be classified into *2-operand* or *3-operand* architectures. With *2-operand* architectures, typical arithmetic and logic instructions specify one operand that is both a source and the destination for the operation result, and another operand is a source. For example, the arithmetic instruction *ADD R1, R2* adds the operands from the registers *R1* and *R2* and writes the result back to the register *R2*. With *3-operand* architectures, instructions can specify two source operands and the destination operand. For example, the arithmetic instruction *ADD R1, R2, R3* adds the operands from the registers *R1* and *R2* and writes the result to the register *R3*.

The MSP430 belongs to *register-memory* type of architectures, which means that machine instructions find their operands in either general-purpose registers or memory locations. The number of operands that can be specified by machine instructions is *maximum two*. We

distinguish between double-operand (two-operand), single-operand, and jump instructions. In double-operand instructions, the first operand is usually referred to as *source* (*src*) and the second operand is referred to as *destination* (*dst*) or *source/destination* (*src/dst*), depending on instruction type. In single-operand instructions, the only operand is usually referred to as *source/destination* (*src/dst*) operand.

```
Things to remember 2-1. Types of Architectures.
```

Types of architectures:

(a) stack

(b) accumulator

(c) general-purpose register.

Based on the number of operands explicitly specified by an instruction we can have:

(a) null-operand architectures (stack)

(b) single-operand architectures (accumulator)

(c) two-operand architectures (one source, one source/destination)

(d) three-operand architecture (two sources, one destination).

```
Things to remember 2-2. Type of MSP430 Architecture.
```

MSP430 is a register-memory architecture with 2-operand instructions.

3 Memory Architecture

The MSP430 family uses *the Von-Neumann architecture* – the program and data share a single address space. An alternative to the Von-Neumann architecture is *the Harvard architecture* where the program and data occupy separate address spaces. The operations can be performed on byte- or word-sized operands and the smallest addressable unit is a byte. All addresses are 16-bit long, and the address space is thus 65,536 (2^{16}) bytes. The address space is divided into several sections: for special-purpose system registers (typically occupy byte addresses 0x0000 – 0x000F), 8-bit peripheral device registers (0x0010 – 0x00FF), 16-bit peripheral device registers (0x0100 – 0x01FF), RAM memory (volatile, read/write), and Flash (non-volatile, read-only) memory. The processor communicates with memory and I/O peripherals through a 16-bit address bus, a 16-bit data bus, and a control bus.

Memory is organized in such a way that in one bus operation an entire 16-bit word can be read from or written to. All words are aligned – i.e., a 16-bit word must be aligned to an even address in the address space – no word size operand can be placed at an odd address in the address space. A byte operand can be placed at any address (odd or even) in address space.

An important question is how to store multi-byte objects in memory. For example, let us assume that you have a 16-bit constant 0x4567 and you want to store it in memory at the address 0x0600. The question is how do you do it? You can place the lower byte of the operand (0x67) to the byte location at the address 0x0600 and the upper byte 0x45 to the byte location at the address 0x0601 (see Figure 1). This placement policy is known as *little-endian* - the lower byte is placed at a lower address in address space. Alternatively, you can place the upper byte of the operand 0x45 to the location at 0x600 and the lower byte 0x67 to the location at 0x601. This placement policy is known as *big-endian*. Whereas Figure 1 shows a byte view of address space, we will often see the address space as a collection of words or two bytes.

Little-endian placement	
Address	Content M[7 . . . 0]
0x0600	0x67
0x0601	0x45

Big-endian placement	
Address	Content M[7 . . . 0]
0x0600	0x45
0x0601	0x67

Figure 1. Little-endian vs. big-endian.

In MSP430 ISA multi-byte objects are stored in memory using *little-endian policy* (the lower byte of the object is stored at a memory location with lower address).

Let us consider address mapping of an MSP430 with the following characteristics:

- 16 KiB of flash memory placed at the top of address space (the uppermost quarter of address space),
- 4KiB of RAM memory that is placed at the based address 0x3100.

The address space of 64 KiB can be viewed as a collection of 16-bit words, placed at word addresses 0x0000, 0x0002, 0x0004, and so on, up to 0xFFFE. Figure 2 shows how actual memory modules map into the 64 KiB address space. Blue blocks represent actual physical locations, whereas green blocks represent portions of address space that are not occupied, i.e., there is no physical storage residing in these areas. As discussed above, the first 256 bytes of address space is reserved for special-purpose registers and registers of 8-bit I/O peripherals. This block occupies the address range from 0x0000 – 0x00FE (we use here word addresses, rather than byte addresses). The next 256 bytes is reserved for registers of 16-bit I/O peripherals; it occupies a block starting at the address 0x0100 (one word above the previous block) and ending at the address 0x01FE. The RAM memory starts at the address 0x3100. This means that we do not have any physical memory in the address range from 0x0200 to 0x30FE. Consequently, your program should never try to read from or write to these locations. The size of the RAM memory is 4 KiB or 2^{12} bytes. If we were to place this block at the address 0x0000, it would have the range from 0x0000 to 0x0FFE ($2^{12} - 2$ in hex is 0x0FFE). However, as the RAM block starts at 0x3100, the RAM memory address range is from 0x3100 to 0x40FE (0x3100+0x0FFE). The 16 KiB flash memory is placed at the uppermost quarter of the 64 KiB

address space. Thus, the flash memory occupies the address range from 0xC000 to 0xFFFE. An easy way to reason about the flash memory address range is to recall address decoding. The 16-bit memory address with individual address bits A_{15} to A_0 can be divided into 4 sub-sections, 16 KiB each (64 KiB/4 = 16 KiB). Common for all addresses in the first quarter is that A_{15}=0 and A_{14}=0, common for all addresses in the second quarter is that A_{15}=0 and A_{14}=1, common for all addresses in the third quarter is that A_{15}=1 and A_{14}=0, and common for all addresses in the fourth quarter is that A_{15}=1 and A_{14}=1. Thus the first address in the fourth quarter is 1100_0000_0000_0000 in binary or 0xC000 in hex and the last address is 1111_1111_1111_1110 or 0xFFFE.

Please note that this is just an example, and different MSP430 parts will have different address ranges depending on the size of RAM and Flash memory modules.

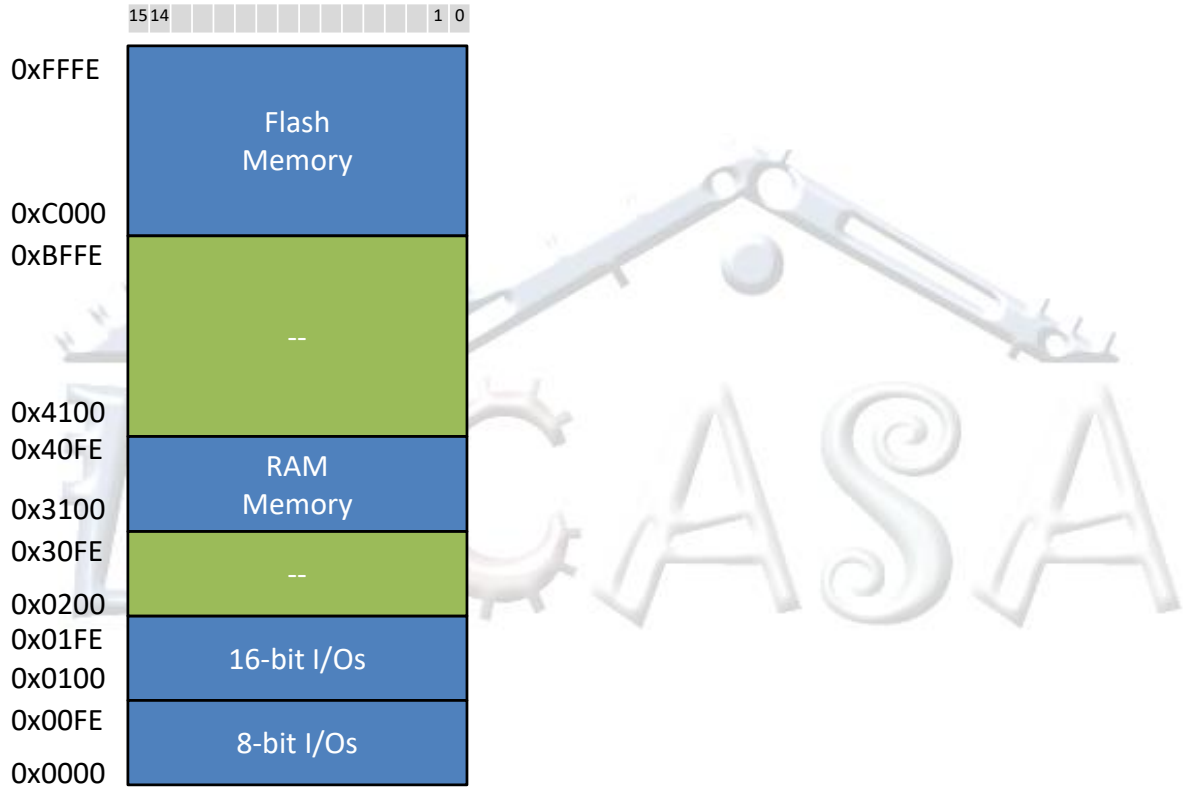


Figure 2. Address mapping of an example MSP430.

Warning!

Very often students provide an incorrect answer when determining address ranges in the address space. The address range asks you to determine the address of the first byte (word) and the last byte (word) within the module. Let us assume that we have a 256 byte memory module that is placed at the the address 0x0200 in the 64 KiB address space. 256 bytes = 2^8 bytes, which is equivalent to 1_0000_0000 b or 0x0100. The address range for this memory module is thus 0x0200 – 0x02FF (using byte addresses) or 0x0200-0x02FE (using word addresses). Often students will offer an incorrect answer, as follows: 0x0200 + 0x0100 = 0x0300, thus the last address within the module is 0x0300. This is wrong! Make sure you

understand why. Hint: if I give you a module of just two bytes and you place it at address 0x0000, is the address range 0x0000-0x0001 or 0x0000-0x0002?

Things to remember 3-1. MSP430 Address Space.

MSP430 address space is byte-addressable (the smallest unit that can be addressed in address space), word-aligned (16-bit words are aligned to even addresses), and little-endian (multi-byte objects are placed using little-endian placement policy).

Things to remember 3-2. MSP430 Address Space Sections.

MSP430 address space is partitioned into sections reserved for non-volatile read-only flash memory holding code and constant data, volatile read/write RAM memory holding data, and registers for input/output peripheral devices.

4 Registers

Figure 3 shows a block diagram of the MSP430 processor's datapath with registers and ALU. The MSP430 has sixteen 16-bit registers named R0 – R15 that are all visible to programmers. Some of these registers are reserved for special functions as follows: the register R0 serves as the program counter (PC), the register R1 serves as the stack pointer (SP), and the register R2 serves as the status register (SR). The R3 register is used for constant generation, while the remaining registers R4-R15 serve as general-purpose registers (i.e., they can be used by programmers freely for storing local variables and addresses).

A relatively large number of general-purpose registers, compared to other microcontrollers, allows the majority of program computation to take place on operands in general-purpose registers, rather than operands in main memory. This helps improve performance and reduce code size.

Register-register operations (both operands are in registers) are performed in a single clock cycle. For example, ADD R4, R5 instruction will read the contents of registers R4 and R5; the register R4 goes to the *src* input of the arithmetic-logic unit (ALU) and register R5 goes to the *dst* input of the ALU. The result shows up on the ALU output and the memory data bus (MDB), and it is stored back in the register R5.

Program counter (PC/R0). PC always points to the next instruction to be executed. MS430 instructions can be encoded with 2 bytes, 4 bytes, or 6 bytes depending on addressing modes used for source (*src*) and source/destination (*src/dst*) operands. The MSP430 has byte-addressable architecture (the smallest unit in memory that can be addressed directly is a byte). Hence, the instructions always have an even number of bytes (they are word-aligned), so the least significant bit of the PC is always zero as shown in Figure 4.

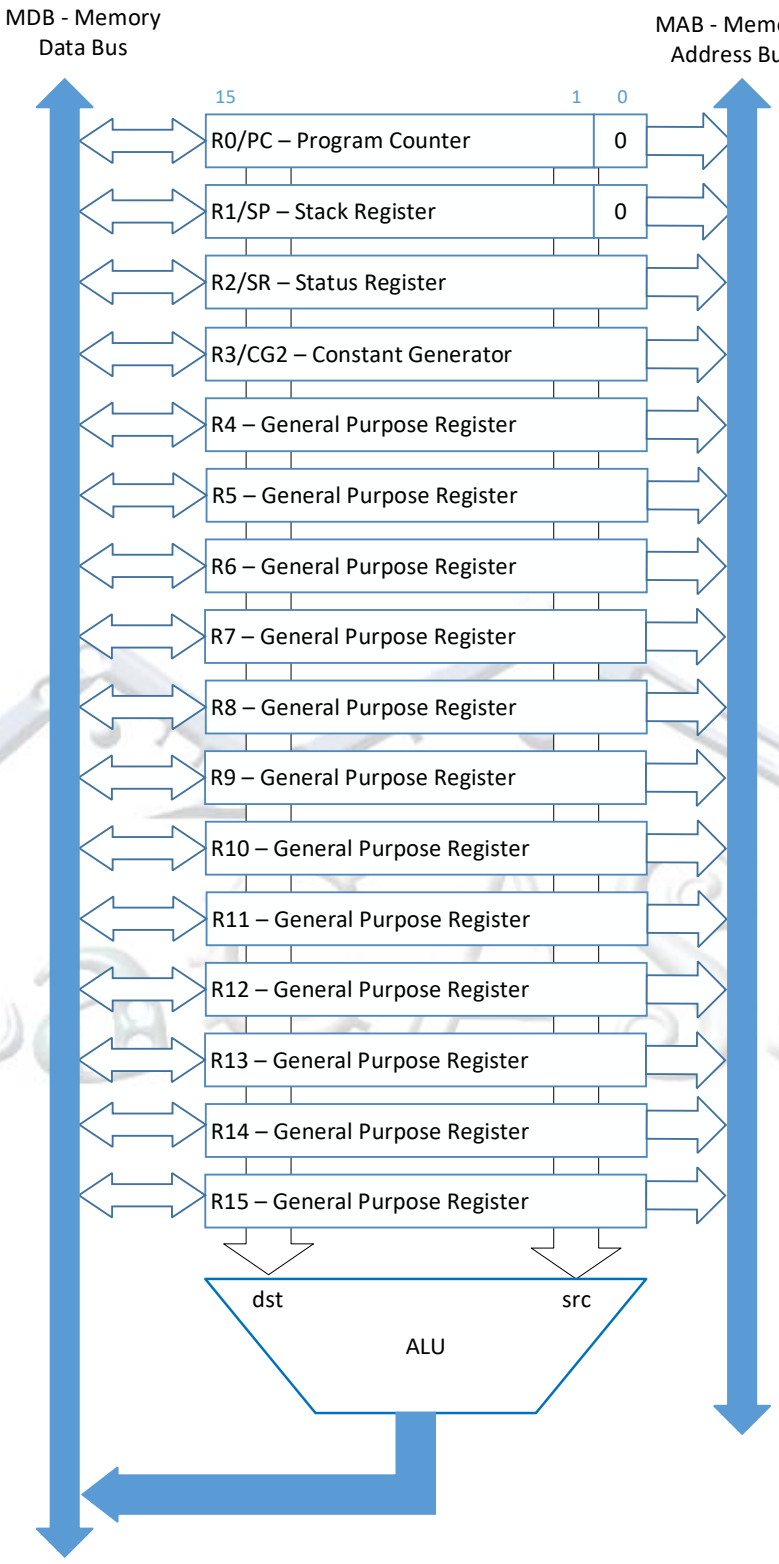


Figure 3. MSP430 CPU Datapath: Registers and ALU.

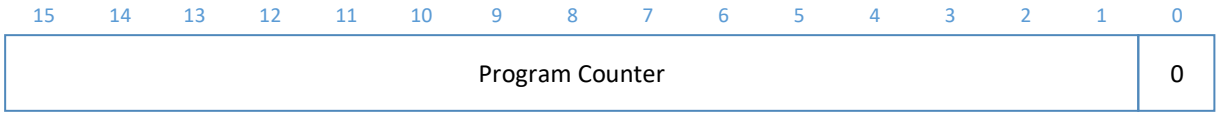


Figure 4. Program Counter.

Stack pointer (SP/R1). The program stack is a dynamic LIFO (Last-In-First-Out) structure allocated in RAM memory. The program stack is used to store the return addresses of subroutine calls and interrupts. In addition, the stack is used as the storage for local data allocated in subroutines and for passing input parameters to subroutines and returning results from subroutines.

The MSP430 architecture assumes the following stack convention: the SP points to the last full location on the top of the stack and the stack grows towards lower addresses in memory. The stack is also word-aligned, so the LSB bit of the SP is always 0.

Two main stack operations are PUSH and POP as shown in Table 1. The PUSH operation pushes content to the top of the stack and the POP operation retrieves the content from the top of the stack. Please note that even when you are pushing a single byte to the stack, it will occupy an entire word in memory.

Table 1. Example instructions dealing with the program stack.

Instruction	RTL	Description
`PUSH R7`	`SP ← SP - 2` `M[SP] ← R7`	Decrement SP to allocate a new word on the stack; copy the content of the register R7 to the location on the top of the stack
`POP R6`	`R6 ← M[SP]` `SP ← SP + 2`	Retrieve the content from the top of the stack and copy it to R6; increment SP by 2 to move SP to point to the new top of the stack.

Let us consider the instruction PUSH R7. This instruction pushes the content of the register R7 to the stack. Since the SP points to the last full (occupied) location on the stack, the first step is to decrement SP to point to the next free location on the stack. Since the stack is growing towards lower addresses in RAM memory, the SP is decremented as follows: SP ← SP – 2. The next step is to copy the content of the register R7 into the memory location that SP is pointing to. We describe this step as follows: M[SP] ← R7, or the memory location with the address contained in SP/R1 is loaded with the content of register R7.

The instruction POP R6 retrieves the content from the current top of the stack and copies it into the register R6. We describe that step as follows: R6 ← M[SP]. To free the location on the top of the stack, the SP register is updated as follows: SP ← SP + 2. The location above the original top of the stack becomes a new top of the stack.

Status register (SR/R2). The status register keeps the content of arithmetic flags (C, V, N, Z), as well as some control bits, such as, SCG1, SCG0, OSCOFF, CPUOFF, and GIE. The exact format of the status register is shown in Figure 5. Table 2 describes the status register flags and their meaning.

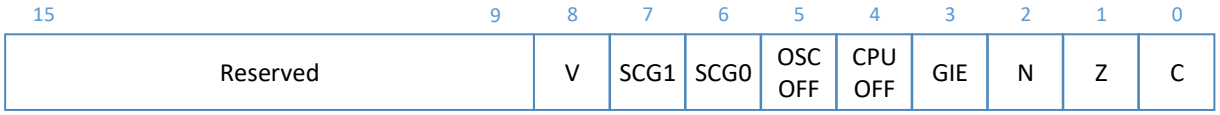


Figure 5. Status register format (top) and bits description (bottom).

Table 2. Description of the status register flags.

Flag	Bits	Description
Reserved	15 – 9	Reserved.
V	8	Overflow. This bit is set when the result of arithmetic operations on signed integers overflows.
SCG1	7	System clock generator 1. Can be used to enable or disable functions in the clock subsystem depending on the device.
SCG0	6	System clock generator 0. Can be used to enable or disable functions in the clock subsystem depending on the device.
OSCOFF	5	Oscillator off. When set, it turns off LFXT1CLK when it is not used for MCLK or SMCLK clocks.
CPUOFF	4	CPU off. When set, it turns off the CPU.
GIE	3	Global Interrupt Enable . When set, it enable maskable interrupts.
N	2	Negative. The bit is set when the result of an operation is negative.
Z	1	Zero. The bit is set when the result of an operation is equal to 0.
C	0	Carry. This bit is set when the result of an operation produced a carry.

Constant generator (R2-R3). Registers R2 and R3 with certain combination of addressing modes can be used to create common constants in hardware thus reducing the size of instructions. More information about this is given at the end of this module.

General-purpose registers (R4-R15). These registers can be used to store temporary data values, addresses of memory locations or index values, and can be accessed with BYTE or WORD instructions.

`Things to remember 4-1. MSP430 Registers.`

MSP430 has 16 16-bit register R0-R15. Registers R0-R3 are special-purpose and R4-R15 are general-purpose registers. R0 is program counter, R1 is stack pointer, R2 is status register, and register R3 is constant generator.

Things to remember 4-2. MSP430 Stack.

MSP430 stack is a Last-In-First-Out abstraction placed at the top of RAM memory that uses the following stack policy: SP points to the last full location on the top of the stack, and the stack is growing from higher to lower addresses.

5 Operands and Data Types

Data transfer, arithmetic and logic instructions of the MSP430 instruction set can operate on either BYTE operands or WORD (2 byte) operands. Instructions operating on word operands use suffix .W (.w) and instructions operating on byte operands use suffix .B (.b). The instructions without any suffix by default refer to word operands. Byte or word operands represent unsigned and signed 8-bit and 16-bit integers. There is only one instruction that operates on data encoded as unsigned BCD integers. Thus, the only data types that machine instructions can operate on are 8-bit and 16-bit signed and unsigned integers in binary and 8-bit and 16-bit unsigned integers in the BCD format.

The MSP430 ISA follows specific rules when dealing with byte and word operands in registers and memory as discussed below. The byte operands in registers deal with a lower byte (bits 7 to 0) of the 16-bit register.

Let us consider a byte register-memory operation (src is in a register, src/dst is in memory, byte size) using the following instruction (assume the following initial conditions: R5=0xA28F, R6=0x0202, and M[0x0203]=0x12):

```
ADD.B R5, 1(R6)
```

Figure 6, top illustrates the content of relevant registers and memory locations before and after the instruction execution. The suffix .B indicates that the operation should be performed on byte-size operands. This instruction specifies that the *src* operand is the register R5 (lower 8 bits of R5), and the *src/dest* operand is in memory at the address (R6+1). Thus, a lower byte from the register R5, 0x8F, is added to the byte read from the memory location M[0x0203]=0x12, and the result is written back to memory, so the new value of the memory location at the address 0x0203 is M[0x0203]=0xA1. The content of the register R5 remains intact. You will notice that a side effect of this instruction is a new value in register R2 (SR) - the result is a negative, so the N bit is set.

Let us now consider an instruction that operates on words:

```
ADD.W R5, 0(R6)
```

Figure 6, bottom illustrates the content of relevant registers and memory locations before and after the instruction execution. The source operand is the content of register R5, which is 0xA28F. The source/destination operand is the word from memory location R6+0=0x0202,

which is 0x1245. The result of addition is 0xB4D4 and this result should be written back to memory.

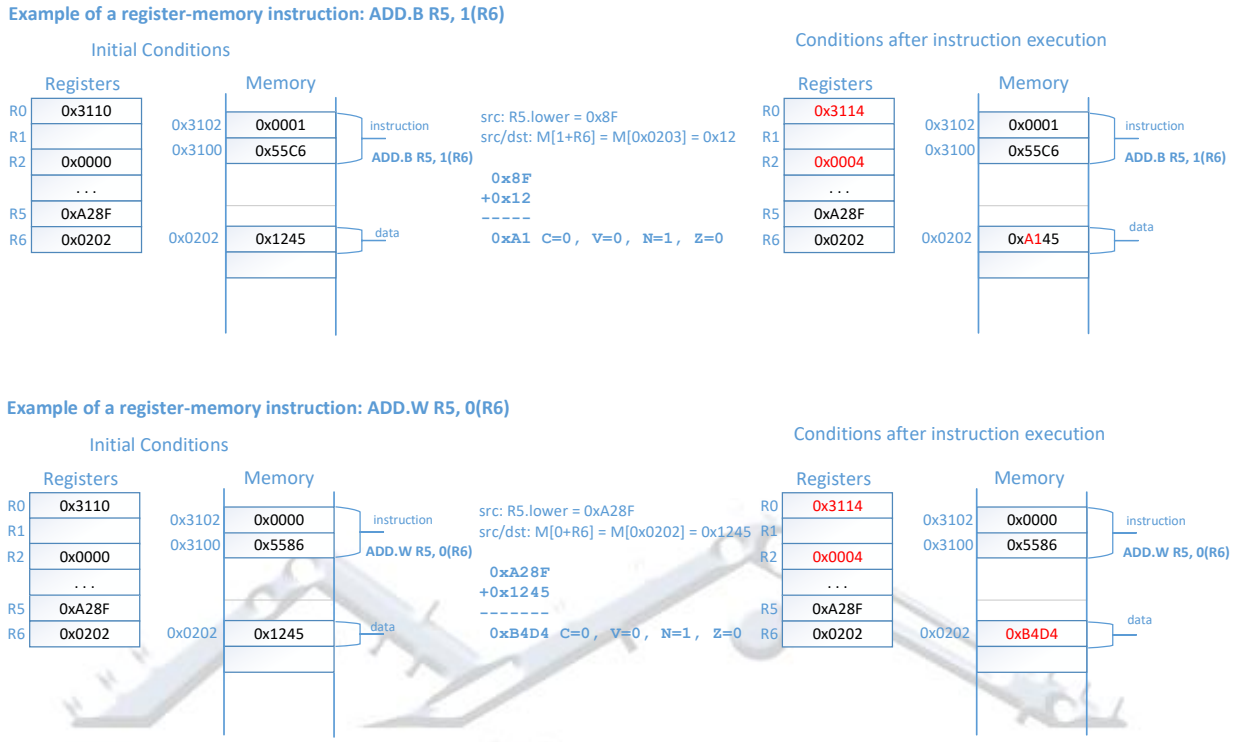


Figure 6. Examples of a byte and a word register-memory instruction.

Let us now consider a byte memory-register operation (src is in memory, src/dst is in a register) using the following instruction (assume the following initial conditions: R5=0xA28F, R6=0x0202, and M[0x0202]=0x1245):

```
ADD.B 1(R6), R5
```

Figure 7, top illustrates the content of relevant registers and memory locations before and after the instruction execution. This instruction specifies a source operand in memory at the address contained in R6+1, and the source/destination operand is in the register R5. A suffix .B indicates that the instruction uses byte-sized operands. As shown below, a byte value M[0x0203]=0x12 is added to the lower byte of R5, 0x8F. The result 0x12+8F=0xA1 is zero extended to a 16-bit word, and the result is written back to register R5. So, the upper byte is always cleared in case of byte-sized memory-register operations. This brings as to the MSP430 specific rule that you should remember: instructions operating on bytes with destination in memory store the result into the specified memory location of size one byte; instructions operating on bytes with the destination in a register store the result in lower 8 bits and clear upper 8 bits of the destination register.

Let us now consider an instruction that operates on words:

```
ADD.W @R6, R5
```

Figure 7, bottom illustrates the content of relevant registers and memory locations before and after the instruction execution. The source operand is in memory, M[R6]=0x1245 and the source/destination operand is in register R5, 0xA28F. The result of addition 0xB4D4 is written back to the register R5.

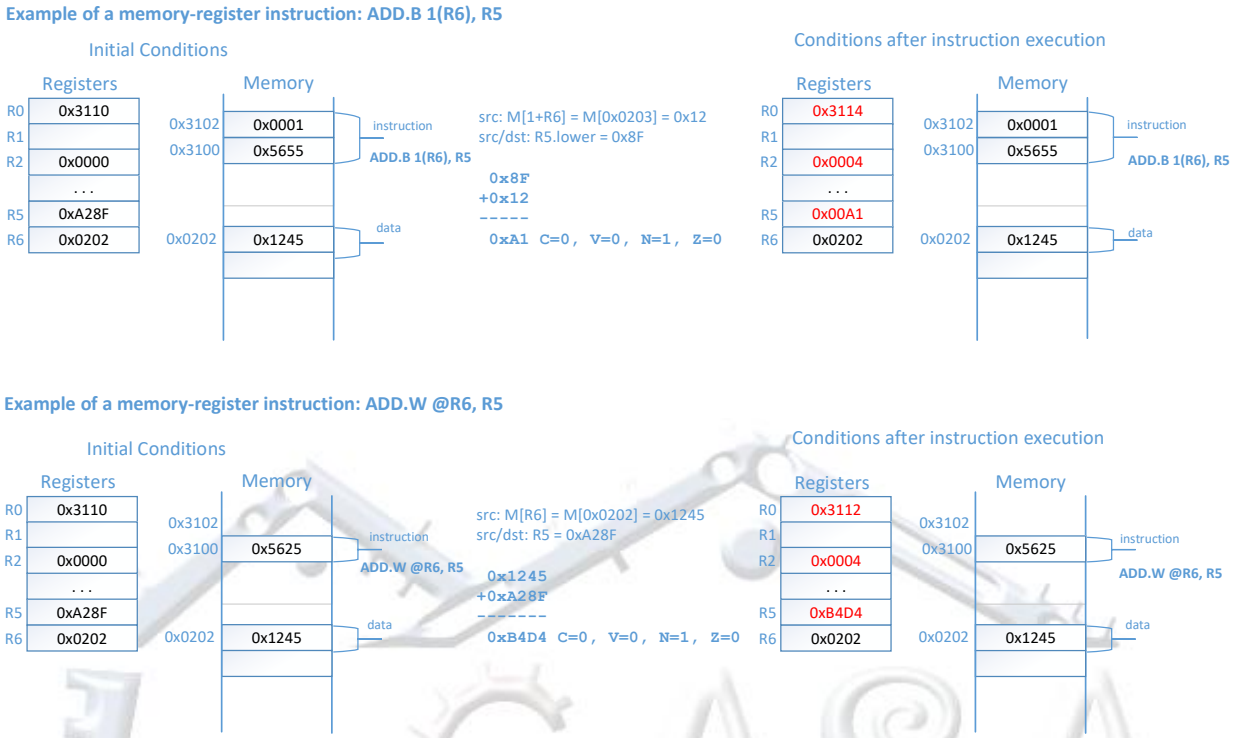


Figure 7. Examples of a byte and a word memory-register instruction.

> `Things to remember 5-1. Byte and word operands.`
>
> MSP430 instructions can operate on byte- and word-sized operands representing 8-bit signed and unsigned integers given in binary and unsigned integers given in BCD. Byte instructions end with a suffix .b and word instruction end with a suffix .w. instruction. Byte instructions operate on lower 8 bits of general-purpose registers. The result of a byte instruction is written back to lower 8 bit of the specified register and the upper 8 bits is always cleared.

6 Basic Instruction Encoding

Depending on addressing modes, double-operand instructions can be 1, 2, or 3 words long. The first-word instruction format is shown in Figure 8.

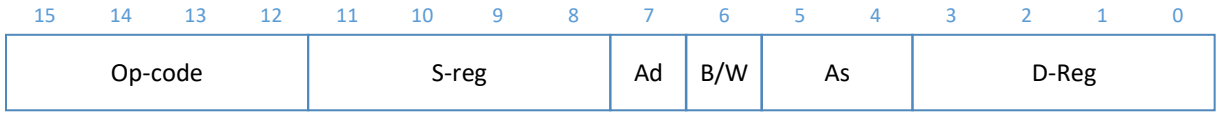


Figure 8. First-word instruction format for double-word instruction.

The meaning of individual fields is as follows.

- *Op-code* – Encodes the instruction (type of operation like MOV, ADD, ..)
- *src* – The source operand is specified by *As* and *S-reg*
  - *As* – Specifies the addressing mode used for the source (src)
  - S-reg – Specifies a general-purpose register used for the source (src)
- *dst* – The destination operand is specified by *Ad* and *D-reg*
  - *Ad* – Specifies the addressing mode used for the destination (dst)
  - *D-reg* – Specifies a general-purpose register used for the destination (dst)
- *B/W* – Byte or word operation:
  - 0: word operation
  - 1: byte operation

7 Addressing Modes

The MSP430 architecture supports a relatively rich set of addressing modes. Seven addressing modes can be used to specify a source operand in a register or any location in memory (Table 3), and the first four of these can be used to specify the source/destination operand. Table 3 also illustrates the syntax and gives a short description of the addressing modes. The addressing modes are encoded using *As* (2-bit long) and *Ad* (1-bit long) address specifiers in the instruction word, and the first column shows how they are encoded.

Table 3. Addressing Modes.

Addressing Mode	Address Specifier As/Ad	Syntax	Description
Register	00/0	Rn	Operand is in register Rn. Instruction specifies register index n.
Indexed	01/1	X(Rn)	Operand is in memory at the address EA=Rn+X. Instruction specifies register index n and offset X (the next word of instruction)
Symbolic	01/1	ADDR	Operand is in memory at the address ADDR=EA=PC+X. This is a special case of the indexed mode (Rn=R0, As=01).
Absolute	01/1	&ADDR	Operand is in memory at the address X, which is specified in the next instruction word. This

			is also treated as a special case of the indexed mode (Rn=R2, As=01).
Indirect register	**10/-**	@Rn	Operand is in memory at the address contained in register Rn. The instruction specifies Rn. Applies only to src operand.
Indirect autoincrement	**11/-**	@Rn+	Operand is in memory at the address contained in register Rn. After getting the operand, the register is incremented for the size of the operand (1 for byte, 2 for word)
Immediate	**11/-**	#N	The operand is a constant encoded in the next instruction word. To distinguish from autoincrement, the S-reg is set to PC.

Register mode. The fastest and shortest mode is used to specify operands in registers. The address field specifies the register number (4 bits – 0000b for R0, 1111b for R15). Address specifiers are As=00 for source operand and Ad=0 for destination operand.

`Example 7-1. MOV.B R5, R7;  R7 ← R5`

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: register, register id: 5 => As=00, S-reg=0101
- Destination addressing mode: register, register id: 7 => Ad=0, D-reg=0111
- Operation: the content of register R5 is copied into R7 (R7 ← R5; read as "R7 gets R5")
- Machine code: 0100_0101_0100_0111 or 0x4547

Indexed mode. The operand is located in memory and its address is calculated as a sum of the specified address register and the displacement X, which is specified in the next instruction word. The effective address of the operand is *EA*, *EA*=Rn+X.

`Example 7-2. MOV.B 10(R5), 12(R7);  M[R7+12] ← M[R5+10]`

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: indexed, register id: 5, offset=10 =>
  As=01, S-reg=0101, 2nd instruction word=10;
  The effective address of the src operand EA.src = R5 + 10
- Destination addressing mode: indexed, register id: 7, offset=12 =>
  Ad=1, D-reg=0111; 3rd instruction word=12;
  The effective address of the dst/src operand is EA.src/dst = R7+12

- Operation: the source operand (a single byte) from a memory location at the address R5+10 is read and copied into the destination operand in memory at the address R7+12;
- Machine code (3-word instruction):
  1st word: 0100_0101_1101_0111;
  2nd word: 0000_0000_0000_1010;
  3rd word: 0000_0000_0000_1100; or
  0x4547; 0x000A; 0x000C

Symbolic mode. This addressing mode can be considered as a subset of the indexed mode. The only difference is that the address register is PC, and thus *ea*=PC+Offset, that is, the address of the operand is specified relatively to the current PC.

`Example 7-3. MOV.B TONI, EDE;` M[EDE] ← M[TONI]

Assume TONI and EDE are symbolic names for addresses 0x0200 and 0x0300, respectively.

Next, assume that this instruction starts at the address 0xE000 (address of the first instruction word).

This instruction requires 3 words, where the second and third ones carry the offsets to the source and destination operand, relatively to the current PC. The second word is at the address 0xE002 and the third word is at 0xE004.

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: symbolic, register id: 0, offset=Offset.src =>
  As=01, S-reg=0000, 2nd instruction word=Offset.src;
  The effective address of the src operand EA.src = 0x0200 = PC + 2 + Offset.src
  => EA.src=0x0200 = 0xE002+Offset.src=> Offset.src=0x0200-0xE002=0x21FE
- Destination addressing mode: symbolic, register id: 0, offset=Offset.src/dst =>
  Ad=1, D-reg=0000; 3rd word=Offset.src/dst;
  The effective address of the dst/src operand is EA.src/dst = 0x0300 = PC + 4 +
  Offset.src/dst =>
  EA.dst=0x0300 = 0xE004+Offset.src/dst => Offset.src/dst=0x0300-0xE004=0x22FC
- Operation: the source operand (a single byte) from a memory location at the address 0x0200 (TONI) is read and copied into the destination operand in memory at the address 0x0300 (EDE);
- Machine code (3-word instruction):
  1st word: 0100_0000_1101_0000;
  2nd word: 0010_0001_1111_1110;

3rd word: 0010_0010_1111_1100; or
0x4040; 0x21FE; 0x22FC

Absolute mode. The instruction specifies the absolute (or direct) address of the operand in memory. The instruction includes a word that specifies this address. This mode can also be considered as a special-case of indexed mode. As the instruction specifies the address of the operand directly, the first instruction word uses register R2 (status register) as the address register.

`Example 7-4. MOV.B &TONI, &EDE;` M[EDE] ← M[TONI]

Assume TONI and EDE are symbolic names for addresses 0x0200 and 0x0300, respectively.

This instruction requires 3 words, where the second and third ones carry the absolute addresses of the source and destination operands, respectively.

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: absolute, register id: 2, As=01, S-reg=0010,
  2nd instruction word=EA.src = 0x0200;
- Destination addressing mode: absolute, register id: 2 =>
  Ad=1, D-reg=0010; 3rd word=EA.src/dst = 0x0300;
- Operation: the source operand (a single byte) from a memory location at the address 0x0200 (TONI) is read and copied into the destination operand in memory at the address 0x0300 (EDE);
- Machine code (3-word instruction):
  1st word: 0100_0010_1101_0010;
  2nd word: 0000_0010_0000_0000;
  3rd word: 0000_0011_0000_0000; or
  0x4242; 0x0200; 0x0300

Please note that indexed, symbolic, and absolute mode all share the same address specifiers As=01 and Ad=1. How do we distinguish between them? We have shown that the symbolic addressing mode is just a special case of the indexed mode where the PC (or R0) is used as the base register in calculating the operand address. However, the absolute addressing mode does not compute the address of the operand, rather it is given directly in the instruction. Note that register R2 is the status register and one would never ever use this register to calculate an operand address – it would be meaningless. This fact can be used so that when register R2 is specified as the address register, the hardware actually interprets this as the additional address field to select absolute addressing mode. So, the instruction decoder would check not only the *As* field to determine which source addressing mode is used, but also the source register field (*src*).This way, a unique combination of S-reg and As or D-reg and Ad fields helps us distinguish the absolute addressing mode from the indexed addressing mode. What is the rationale behind

this complex encoding? You will notice that we have 7 addressing modes that would normally require at least 3 bits for encoding them fully. To make instruction fit in single word, the MSP430 architects decided to squeeze as much information as possible in one instruction word.

Indirect register mode. It can be only used for source operands, and the instruction specifies the address register Rn, and the *ea*=Rn.

`Example 7-5. MOV.B @R5, &EDE;  M[EDE] ← M[R5]`

Assume EDE is a symbolic name for addresses 0x0300; Assume R5=0x0200.

This instruction requires 2 words, where the second one carries the absolute address of the destination operand.

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: register indirect, register id: 5, As=10, S-reg=0101 => EA.src = R5 = 0x0200
- Destination addressing mode: absolute, register id: 2 => Ad=1, D-reg=0010; 2nd word=EA.src/dst = 0x0300;
- Operation: the source operand (a single byte) from a memory location at the address contained in register R5 is read and copied into the destination operand in memory at the address 0x0300 (EDE);
- Machine code (2-word instruction): 1st word: 0100_0101_1110_0010; 2nd word: 0000_0011_0000_0000; or 0x45E2; 0x0300

Indirect autoincrement. The effective address of the operand in memory is the content of the specified address register Rn, but the content of the register is incremented afterwards by +1 for byte operations and by +2 for word operations.

`Example 7-6: MOV.B @R5+, &EDE;  M[EDE] ← M[R5]; R5 ← R5 + 1;`

Assume EDE is a symbolic name for addresses 0x0300; Assume R5=0x0200.

This instruction requires 2 words, where the second one carries the absolute address of the destination operand.

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: register indirect, register id: 5, As=11, S-reg=0101 => EA.src = R5 = 0x0200; R5 ← R5 + 1;
- Destination addressing mode: absolute, register id: 2 => Ad=1, D-reg=0010; 2nd word=EA.src/dst = 0x0300;

- Operation: the source operand (a single byte) from a memory location at the address contained in register R5 is read and copied into the destination operand in memory at the address 0x0300 (EDE); Register R5 is incremented for the size of the operand (+1 for byte operation, +2 for word operation)
- Machine code (2-word instruction):
  1st word: 0100_0101_1111_0010;
  2nd word: 0000_0011_0000_0000; or
  0x45F2; 0x0300

Immediate mode. The instruction specifies the immediate constant that is operand, and is encoded directly in the instruction.

`Example 7-7: MOV.B #45, &EDE; M[EDE] ← #45`

Assume EDE is a symbolic name for addresses 0x0300;

This instruction requires 3 words, where the second one carries the immediate operand 45.

- Instruction: MOV.B => opcode = 0100; B/W# = 1
- Source addressing mode: register indirect with autoincrement, register id: 0, As=11, S-reg=0000 => Operand is given as the second instruction word;
- Destination addressing mode: absolute, register id: 2 =>
  Ad=1, D-reg=0010; 3rd word=EA.src/dst = 0x0300;
- Operation: the source operand encoded in the second instruction word is copied into the destination operand in memory at the address 0x0300 (EDE);
- Machine code (2-word instruction):
  1st word: 0100_0000_1111_0010;
  2nd word: 0000_0000_0010_1101
  2nd word: 0000_0011_0000_0000; or
  0x40F2; 0x003D, 0x0300

You probably noticed that the immediate addressing mode shares the same address specifier As=11 with the autoincrement mode. It is distinguished from the autoincrement mode because the specified register is the R0 (PC), which is never used in the autoincrement mode (R0 is dedicated register serving as the program counter and incrementing it during operand fetch state would be illogical). Thus, this is another example of clever encoding where As and S-reg fields are used together to uniquely encode an addressing mode.

`Things to remember 7-1. MSP430 Addressing Modes.`

MSP430 supports 7 addressing modes applicable to src operands (top 4 are applicable to src/dst operands) as follows:

(a) register – operand is in a register

(b) indexed – operand is in memory, its address is computed as Rn+X

(c) symbolic – operand is in memory, its address is computed as PC+X

(d) absolute – operand is in memory, its address is specified directly by the instruction

(e) register indirect – operand is in memory, its address is specified by Rn

(f) register indirect with autoincrement – operand is in memory, its address is specified by Rn, Rn is automatically updated to point to the next operand

(g) immediate – operand is encoded in the instruction.

8 Instruction Set and Instruction Encoding

The MSP430 instruction set consists of 27 core instructions and 24 emulated instructions. The core instructions are instructions that have unique op-codes decoded by the CPU. The emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves; instead, they are replaced automatically by the assembler with an equivalent core instruction. There is no code or performance penalty for using emulated instruction.

There are three core-instruction formats:

- Double-operand
- Single-operand
- Jump

All single-operand and double-operand instructions can be byte or word instructions (operate on byte-size or word-size operands) by using .B or .W extensions. Byte instructions are used to access byte data or byte peripherals. Word instructions are used to access word data or word peripherals. If no extension is used, the instruction is a word instruction.

8.1 Double-Operand Instructions

The fields of double-operand instructions are as follows:

- *Op-code* – Encodes the instruction (type of operation like MOV, ADD, ..)
- *src* – The source operand is specified by *As* and *S-reg*
    - *As* – Specifies the addressing mode used for the source (src)
    - S-reg – Specifies a general-purpose register used for the source (src)
- *dst* – The destination operand is specified by *Ad* and *D-reg*
    - *Ad* – Specifies the addressing mode used for the destination (dst)
    - *D-reg* – Specifies a general-purpose register used for the destination (dst)

- *B/W* – Byte or word operation:
  - 0: word operation
  - 1: byte operation

Figure 9 shows the double-operand instruction format and Table 4 shows the list of all double-operand core instructions. The table gives instruction mnemonics, operands, operation performed, and how VNZC flags are updated.

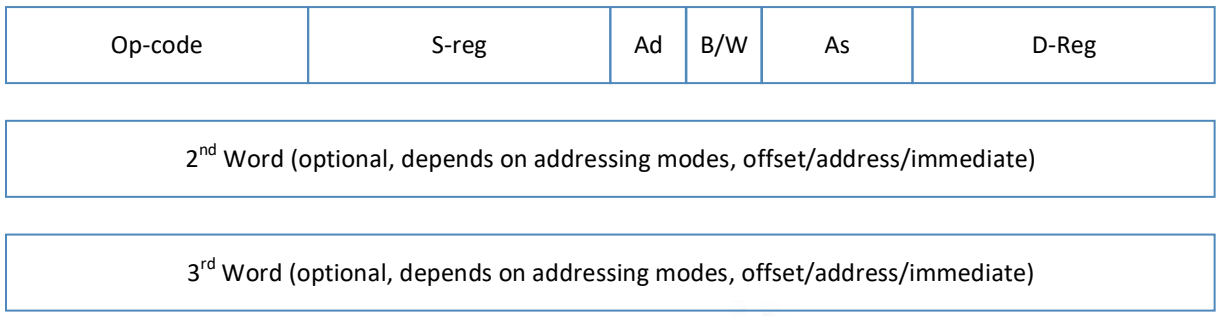


Figure 9. Double-operand first-word instruction format (top) and optional 2nd and 3rd word.

Table 4. Double operand instructions. Flags are affected (*), not affected (-), cleared (0) or set (1).

Mnemonic	Operands	Op-code	Operation	Status Bits			
		(binary)		V	N	Z	C
MOV(.B)	src,dst	0100	src → dst	-	-	-	-
ADD(.B)	src,dst	0101	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	0110	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	1000	dst + .not(src) + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	0111	dst + .not(src) + C → dst	*	*	*	*
CMP(.B)	src,dst	1001	dst – src	*	*	*	*
DADD(.B)	src,dst	1010	src + dst + C (decimal) → dst	*	*	*	*
BIT(.B)	src,dst	1011	src .and. dst	0	*	*	*
BIC(.B)	src,dst	1100	.not(src) .and. dst → dst	-	-	-	-
BIS(.B)	src,dst	1101	src .or. dst → dst	-	-	-	-
XOR(.B)	src,dst	1110	src .xor. dst → dst	*	*	*	*
AND(.B)	src,dst	1111	src .and. dst → dst	*	*	*	*

Below is a detailed description of the double operand instructions with examples. For examples below we will assume that operands are in registers R5 and R7 and the initial conditions are as follows: R5 = 0x42CE and R7=0x200F, and R2=0x0001 (C=1, N=0, Z=0, V=0). These assumptions will apply to all examples, unless it is explicitly stated otherwise.

```
MOV(.B) src, dst  ; dst ← src
```

This instruction copies the source operand to the destination operand. The flags are not affected by this instruction, i.e., they retain their original value they had before execution of the MOV instruction.

```
Example 8-1: MOV.B R5, R7;    R7 ← R5 (lower byte)
```

- Instruction word: 0x4547 (opcode=4, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: R7=0x00CE, R5=0x42CE (no change), R2=0x0001`

```
Example 8-2: MOV.W R5, R7;    R7 ← R5 (copy R5 to R7, word)
```

- Instruction word: 0x4507 (opcode=4, S-reg=5, D-reg=7, AdB/W#As=0000)
- Result: R7=0x42CE, R5=0x42CE (no change), R2=0x0001

`ADD(.B) src, dst  ; dst ← src + dst`

This instruction adds the source operand to the destination operand and stores the result back to the destination operand. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if there is a carry out from the MSB bit of the result, reset otherwise;
- V bit is set if the addition of positive input operands produces a negative result or if the addition of negative operands produces a positive result; it is reset otherwise.

```
Example 8-3: ADD.B R5, R7;    R7 ← R5 + R7 (byte operation)
```

- Instruction word: 0x5547 (opcode=5, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: 0xCE + 0x0F = DD (C=0, N=1, V=0, Z=0) or R7=0x00DD, R5=0x42CE (no change), R2=0x0004`

```
Example 8-4: ADD.W R5, R7;    R7 ← R5 + R7
```

- Instruction word: 0x5507 (opcode=5, S-reg=5, D-reg=7, AdB/W#As=0000)
- `Result: 0x42CE + 0x200F = 0x62DD (N=0, V=0, Z=0, C=0), R7=0x62DD, R5=0x42CE (no change), R2=0x0000`

`ADDC.(B) src, dst ; dst ← src + dst + C`

This instruction adds the source operand and carry to the destination operand and stores the result back to the destination operand. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;

- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if there is a carry out from the MSB bit of the result, reset otherwise;
- V bit is set if the result is negative and input operands are both positive or if the result is positive and input operands are both negative, reset otherwise.

```
Example 8-5: ADDC.B R5, R7;   R7 ← R5 + R7 + C (byte operation)
```

- Instruction word: 0x6547 (opcode=6, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: 0xCE + 0x0F + 1= DE (C=0, N=1, V=0, Z=0) or R7=0x00DE, R5=0x42CE (no change), R2=0x0004`

```
Example 8-6: ADDC.W R5, R7;   R7 ← R5 + R7 + C
```

- Instruction word: 0x6507 (opcode=6, S-reg=5, D-reg=7, AdB/W#As=0000)
- `Result: 0x42CE + 0x200F + 1 = 0x62DE (N=0, V=0, Z=0, C=0), R7=0x62DE, R5=0x42CE (no change), R2=0x0000`

```
SUB(.B) src, dst  ; dst ← dst + (.not.src) + 1 (or dst ← dst - src)
```

This instruction subtracts the source operand from the destination operand and stores the result back to the destination operand. This is done by adding the first complement of source to 1 and to the destination. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if there is a carry out from the MSB bit of the result, reset otherwise;
- V bit is set if the subtraction of a negative source operand from a positive destination operand produces a negative result, or if the subtraction of a positive source from a negative destination operand produces a positive result; reset otherwise.

```
Example 8-7. SUB.B R5, R7;    R7 ← R7 + (.not.R5) + 1 (byte operation)
```

- Instruction word: 0x8547 (opcode=8, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: 0x0F + (.not.CE) + 1 = 0x0F+1+0x31=0x41 (C=0, N=0, V=0, Z=0) or R7=0x0041, R5=0x42CE (no change), R2=0x0000`

```
Example 8-8. SUB.W R5, R7;    R7 ← R7 + (.not.R5) + 1
```

- Instruction word: 0x8507 (opcode=8, S-reg=5, D-reg=7, AdB/W#As=0000)
- `Result: 0x200F + (.not.0x42CE) + 1 = 0x200F + 1 + 0xBD31 = 0xDD41 (N=1, V=0, Z=0, C=0), R7=0xDD41, R5=0x42CE (no change), R2=0x0004`

SUBC(.B) src, dst ; dst ← dst + (.not.src) + C or dst ← dst – (src – 1) + C

This instruction subtracts the source operand with carry from the destination operand and stores the result back to the destination operand. This is done by adding the first complement of source to the carry flag and to the destination. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if there is a carry out from the MSB bit of the result, reset otherwise;
- V bit is set if the subtraction of a negative source operand from a positive destination operand produces a negative result, or if the subtraction of a positive source from a negative destination operand produces a positive result; reset otherwise.

Example 8-9: SUBC.B R5, R7; R7 ← R7 + (.not.R5) + C (byte operation)

- Instruction word: 0x7547 (opcode=7, S-reg=5, D-reg=7, AdB/W#As=0100)
- Result: 0x0F + (.not.CE) + 1 = 0x0F+1+0x31=0x41 (C=0, N=0, V=0, Z=0) or R7=0x0041, R5=0x42CE (no change), R2=0x0000

Example 8-10: SUBC.W R5, R7; R7 ← R7 + (.not.R5) + C

- Instruction word: 0x7507 (opcode=7, S-reg=5, D-reg=7, AdB/W#As=0000)
- Result: 0x200F + (.not.0x42CE) + 1 = 0x200F + 1 + 0xBD31 = 0xDD41 (N=1, V=0, Z=0, C=0), R7=0xDD41, R5=0x42CE (no change), R2=0x0004

(Note: the results match the ones for the SUB instruction in this example because we assume that the Carry bit is initially set to 1.)

CMP(.B) src, dst ; dst + (.not.src) + 1 or dst – src

This instruction subtracts the source operand from the destination operand and set flags accordingly. It does not write the result back into the destination. This is done by adding the first complement of source to 1 and to the destination. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if there is a carry out from the MSB bit of the result, reset otherwise;
- V bit is set if the subtraction of a negative source operand from a positive destination operand produces a negative result, or if the subtraction of a positive source from a negative destination operand produces a positive result; reset otherwise.

Example 8-11: CMP.B R5, R7; R7 + (.not.R5) + 1 (byte operation)

- Instruction word: 0x9547 (opcode=9, S-reg=5, D-reg=7, AdB/W#As=0100)
- Result: 0x0F + (.not.CE) + 1 = 0x0F+1+0x31=0x41 (C=0, N=0, V=0, Z=0) or R7=0x200F (no change), R5=0x42CE (no change), R2=0x0000

Example 8-12: CMP.W R5, R7; R7 + (.not.R5) + 1

- Instruction word: 0x9507 (opcode=9, S-reg=5, D-reg=7, AdB/W#As=0000)
- Result: 0x200F + (.not.0x42CE) + 1 = 0x200F + 1 + 0xBD31 = 0xDD41 (N=1, V=0, Z=0, C=0), R7=0x200F (no change), R5=0x42CE (no change), R2=0x0004

DADD(.B) src, dst ; dst ← src + dst + C (decimally)

This instruction adds the source operand with carry to the destination operand decimally and stores the result back to the destination operand. This instruction assumes that source and destination operands contain valid BCD (Binary-Coded-Decimal) positive numbers. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise (meaning set to one if the result is > 0x79 for byte operands or > 0x7999 for word operands);
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if there is a carry out from the MSB bit of the result (>0x99 or >0x9999), reset otherwise;
- V bit is undefined (could be anything, thus it is not meaningful).

Example 8-13: DADD.B R5, R7; R7 ← R5 + R7 (byte operation), decimally

- Assume C=1, R5=0x4492, R7=0x9025;
- Instruction word: 0xA547 (opcode=0xA, S-reg=5, D-reg=7, AdB/W#As=0100)
- Result: 0x92 + 0x25 + 1 = 28 (C=1, N=0, V=?, Z=0) or R7=0x0018, R5=0x4492 (no change), R2=0x0001 (please note that the Carry bit here indicates that the result is actually 118)

Example 8-14: DADD.W R5, R7; R7 ← R5 + R7 + C (decimally)

- Assume C=1, R5=0x4492, R7=0x9025;
- Instruction word: 0xA507 (opcode=0xA, S-reg=5, D-reg=7, AdB/W#As=0000)
- Result: 0x4492 + 0x9025 = 0x3528 (N=0, V=?, Z=0, C=1), R7=0x3528, R5=0x4492 (no change), R2=0x0001 (the Carry bit indicates that the results is 0x13528).

```
BIT(.B) src, dst  ; src .AND. dst
```

This instruction tests bits that are set in the source operand in the destination operand. This operation is carried out by determining the result of the bitwise logical AND between the source and destination operands. The instruction set the status bits (flags) in the status register, but the result is not stored. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if the results in not 0, reset otherwise;
- V bit is reset.

```
Example 8-15: BIT.B R5, R7;   R5 .AND. R7 (byte operation)
```

- Instruction word: 0xB547 (opcode=0xB, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: 0xCE .and. 0x0F = 0x0E (C=1, N=0, V=0, Z=0) or R7=0x200F (no change), R5=0x42CE (no change), R2=0x0001`

```
Example 8-16: BIT.W R5, R7;   R5 .AND. R7
```

- Instruction word: 0xB507 (opcode=0xB, S-reg=5, D-reg=7, AdB/W#As=0000)
- `Result: 0x42CE .and. 0x200F = 0x000E (N=0, V=0, Z=0, C=1), R7=0x200F (no change), R5=0x42CE (no change), R2=0x0001.`

The BIT instruction is typically used when you want to test value of an individual bit of an operand. Let's say you want to test the sign bit (MSB) in a byte operand in the register R7. You will use the following instruction to test the sign bit: BIT.B #128, R7. The constant 128 corresponds to the bit vector 1000_0000 that actually provides a correct mask for bit 7. If the MSB bit is set, we will have a non-zero result (C=1 and Z=0); if the MSB bit is reset, the result of bitwise AND operation is equal to 0 (C=0, Z=1). By inspecting either the C status bit or the Z status bit you can act accordingly. What constant would you use to test bit 5?

```
BIC(.B) src, dst  ;  dst ← (.not. src) .AND. dst
```

This instruction clears bits that are set in the source operand in the destination operand. This operation is carried out by calculating the result of the bitwise logical AND between the first complement of the source operand and the destination operand. The result is stored into the destination operand. The instruction does not affect the status bits (flags) in the status register.

```
Example 8-17: BIC.B R5, R7;   R7 ← (.not.R5) .AND. R7 (byte operation)
```

- Instruction word: 0xC547 (opcode=0xC, S-reg=5, D-reg=7, AdB/W#As=0100)

- Result: (.not.0xCE) .and. 0x0F = 0x31 .and. 0x0F = 0x01,
  or R7=0x0001, R5=0x42CE (no change), R2=0x0001 (no change).

Example 8-18: BIC.W R5, R7; R7 ← (.not.R5) .AND. R7

- Instruction word: 0xC507 (opcode=0xC, S-reg=5, D-reg=7, AdB/W#As=0000)
- Result: (.not.0x42CE) .and. 0x200F = 0xBD31 .and. 0x200F = 0x2001,
  R7=0x2001, R5=0x42CE (no change), R2=0x0001 (no change).

The BIC instruction is typically used when you want to clear a certain bit in an operand. Let's say you want to clear the N status bit in R2. To do so, you will use the following instruction: BIC.W #4, R2. The source operand is the constant 4 that creates the bit vector 0x0004 that will keep all other bits of R2 unchanged, except the bit 2 that will be forced to 0. Recall that bit 2 of R2 is the N flag.

BIS(.B) src, dst ; dst ← src .OR. dst

This instruction sets bits that are set in the source operand in the destination operand. This operation is carried out by calculating the result of the bitwise logical OR between the source operand and the destination operand. The result is stored into the destination operand. The instruction does not affect the status bits (flags) in the status register.

Example 8-19: BIS.B R5, R7; R7 ← R5 .OR. R7 (byte operation)

- Instruction word: 0xD547 (opcode=0xD, S-reg=5, D-reg=7, AdB/W#As=0100)
- Result: 0xCE .or. 0x0F = 0xCF,
  or R7=0x00CF, R5=0x42CE (no change), R2=0x0001 (no change).

Example 8-20: BIS.W R5, R7; R7 ← R5 .OR. R7

- Instruction word: 0xD507 (opcode=0xD, S-reg=5, D-reg=7, AdB/W#As=0000)
- Result: 0x42CE .and. 0x200F = 0x62CF, R7=0x62CF, R5=0x42CE (no
  change), R2=0x0001 (no change).

The BIS instruction is typically used when you want to force a certain bit in an operand to a logic 1. Let's say you want to set bit 4 of a byte operand in the register R7. To do so, you will use the following instruction: BIS.B #16, R7. The source operand is the constant 16 that creates the bit vector 0x0010 that will keep all other bits of R7 unchanged, except the bit 4 that will be forced to 1.

XOR(.B) src, dst ; dst ← src .XOR. dst

This instruction carries out bitwise logical XOR operation between the source and the destination operand and stores the result into the destination operand. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if the results in not 0, reset otherwise;
- V bit is set if both operands are negative before execution, reset otherwise.

```
Example 8-21: XOR.B R5, R7;   R7 ← R5 .XOR. R7 (byte operation)
```

- Instruction word: 0xE547 (opcode=0xE, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: 0xCE .xor. 0x0F = 0xC1 (C=1, N=1, V=0, Z=0) or R7=0x00C1, R5=0x42CE (no change), R2=0x0005`

```
Example 8-22: XOR.W R5, R7;   R7 ← R5 .XOR. R7
```

- Instruction word: 0xE507 (opcode=0xE, S-reg=5, D-reg=7, AdB/W#As=0000)
- `Result: 0x42CE .xor. 0x200F = 0x62C1 (N=0, V=0, Z=0, C=1), R7=0x62C1, R5=0x42CE (no change), R2=0x0001.`

```
AND(.B) src, dst  ; dst ← src .AND. dst
```

This instruction carries out bitwise logical AND operation between the source and the destination operand and stores the result into the destination operand. This instruction is the same as BIT, except that it stores the result in addition to setting the flags. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is set if the results in not 0, reset otherwise;
- V bit is reset.

```
Example 8-23. AND.B R5, R7;   R7 ← R5 .AND. R7 (byte operation)
```

- Instruction word: 0xF547 (opcode=0xF, S-reg=5, D-reg=7, AdB/W#As=0100)
- `Result: 0xCE .and. 0x0F = 0x0E (C=1, N=0, V=0, Z=0) or R7=0x000E, R5=0x42CE (no change), R2=0x0001`

```
Example 8-24. AND.W R5, R7;   R7 ← R5 .AND. R7
```

- Instruction word: 0xF507 (opcode=0xF, S-reg=5, D-reg=7, AdB/W#As=0000)

- Result: 0x42CE .and. 0x200F = 0x000E (N=0, V=0, Z=0, C=1), R7=0x000E, R5=0x42CE (no change), R2=0x0001.

Things to remember 8-1. MSP430 double operand instructions.

MSP430 supports 12 double-operand core instructions. Their size ranges from 1 to 3 words, depending on the addressing modes.

8.2 Single-Operand Instructions

Figure 10 shows the single-operand instruction format and Table 5 lists all single-operand core instructions. Please note that these instruction use a different encoding format. The single-operand is referred to as source/destination, and there is one address specifier Ad which is 2-bit long. The encoding for Ad is the same as for As in Table 3.

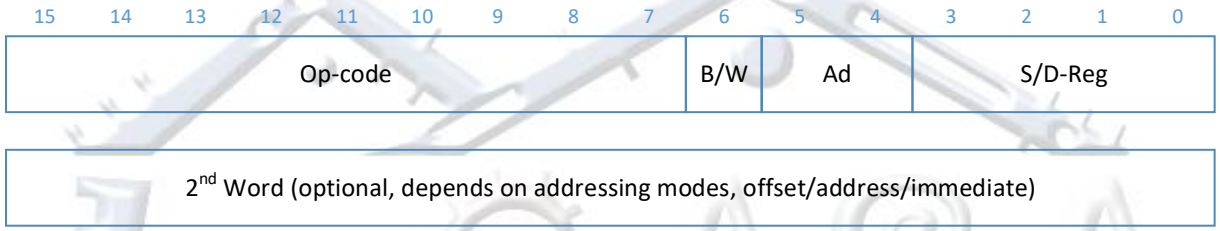


Figure 10. Single-operand first-word instruction format.

Table 5. Single-operand instructions. Flags are affected (*), not affected (-), cleared (0) or set (1).

Mnemonic	Operands	Op-code, B/W	Operation	Status Bits			
				V	N	Z	C
RRC(.B)	dst	0001_0000_0, B/W	C → MSB → … → LSB → C	0	*	*	*
RRA(.B)	dst	0001_0001_0, B/W	MSB → MSB → … → LSB → C	0	*	*	*
PUSH(.B)	src	0001_0010_0, B/W	SP - 2 → SP; src → M[SP]	-	-	-	-
SWPB	dst	0001_0000_1, 0	Swap byte of the word operand	-	-	-	-
CALL	dst	0001_0010_1, 0	SP – 2 → SP; PC → M[SP]; push ret. adr. dst → PC; target address to PC	-	-	-	-
RETI	-	0001_0011_0, 0	M[SP] → SR; SP + 2 → SP; M[SP] → PC; SP + 2 → SP;	*	*	*	*
SXT	dst	0001_0001_1, 0	Bit7 → Bit8… Bit15	0	*	*	*

Below is a detailed description of the single operand instructions with examples.

`RRC(.B) dst ; shift to the right by one bit position through Carry`

This instruction carries out rotate right through carry bit on the operand. The operand is shifted to the right by one bit position. The MSB bit gets the current value of the Carry bit, and the LSB bit is shifted out into the Carry bit. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is loaded from the LSB;
- V bit is reset.

```
Example 8-25. RRC.B R7;
```

- Initial conditions: R7=0x200F, C=1
- Instruction word: 0x1047 (opcode=0001_0000_0b, B/W#=1, Ad=00, D-reg=7)
- `Result:  0x0F .. 1 => 0x87 (C=1, N=1, V=0, Z=0) or R7=0x0087, R2=0x0005`

```
Example 8-26. RRC.W R7;
```

- Initial conditions: R7=0x200F, C=1
- Instruction word: 0x1007 (opcode=0001_0000_0b, B/W#=0, Ad=00, D-reg=7)
- `Result:  0x200F .. 1 => 0x9007 (C=1, N=1, V=0, Z=0) or R7=0x9007, R2=0x0005`

`RRA(.B) dst ; rotate right arithmetic`

This instruction carries out rotate right arithmetic on the destination operand. The destination operand is shifted to the right by one bit position. The MSB bit is shifted back into the MSB bit position as well as shifted to the right. The LSB bit is shifted out into the Carry bit. The flags are affected as follows:

- N bit is set (1) if the result is negative (in 2's complement) and reset (0) otherwise;
- Z bit is set if the result is equal to 0, reset otherwise;
- C bit is loaded from the LSB;
- V bit is reset.

```
Example 8-27. RRA.B R7;
```

- Initial conditions: R7=0xC00F, C=1
- Instruction word: 0x1147 (opcode=0001_0001_0b, B/W#=1, Ad=00, D-reg=7)
- `Result:  0x0F  => 0x07 (C=1, N=0, V=0, Z=0) or R7=0x0007, R2=0x0001`

Example 8-28. RRA.W R7;

- Initial conditions: R7=0xC00F, C=1
- Instruction word: 0x1107 (opcode=0001_0001_0b, B/W#=0, Ad=00, D-reg=7)
- Result: 0xC00F => 0xE007 (C=1, N=1, V=0, Z=0) or R7=0xE007, R2=0x0005

The RRA preserves the sign bit of the destination operand. You can divide your signed operands by 2 using this instruction.

PUSH(.B) dst ; push dst to the stack SP ← SP – 2; M[SP] ← dst

This instruction pushes the operand to the stack. As described above, the first step is to decrement SP by 2 and then copy the content of the operand to the stack. This instruction does not affect the flags. Please note that even if you push a single byte, an entire word is reserved on the stack.

Example 8-29. PUSH.B R7; SP ← SP – 2; M[SP] ← R7 (byte operation)

- Initial conditions: R7=0xC00F, R1=0x0800; M[0x07FE] = 0x3320
- Instruction word: 0x1247 (opcode=0001_0010_0b, B/W#=1, Ad=00, D-reg=7)
- Result: SP=R1=0x07FE; M[0x07FE] = 0x330F; R7=0xC00F; Please note that upper byte of the memory word at the address 0x07FE is going to retain its original value;

Example 8-30. PUSH.W R7; SP ← SP – 2; M[SP] ← R7

- Initial conditions: R7=0xC00F, R1=0x0800; M[0x07FE] = 0x3320
- Instruction word: 0x1207 (opcode=0001_0010_0b, B/W#=0, Ad=00, D-reg=7)
- Result: SP=R1=0x07FE; M[0x07FE] = 0xC00F; R7=0xC00F (no change);

SWPB dst ; swap bytes

This instruction swaps the lower and the upper bytes in the destination operand. It does not affect flags. The operand is a word (this instruction is meaningless for byte-sized operands).

Example 8-31. SWPB R7;

- Initial conditions: R7=0xC00F;
- Instruction word: 0x1087 (opcode=0001_0000_1b, B/W#=0, Ad=00, D-reg=7)
- Result: R7=0x0FC0;

```
CALL dst    ; call subroutine, temp← dst; SP ← SP – 2; M[SP] ← PC; PC ← temp
```

This instruction calls a subroutine specified by the dst. A subroutine call is made from an address in the lower 64 KiB to a subroutine address in the lower 64 KiB. All seven source addressing modes can be used. The call instruction is a word instruction. The current PC is pushed to the stack, the subroutine target address is determined based on the addressing mode, and the PC is then loaded with the starting address of the subroutine. This instruction does not affect the flags. The return is made with the RET instruction. The RET is an emulated instruction which is equivalent to MOV.W @SP+, PC.

Example 8-32. CALL R7;

- Initial conditions: R7=0xC000; PC=0xE000; SP=R1=0x0800
- Instruction word: 0x1287 (opcode=0001_0010_1b, B/W#=0, Ad=00, D-reg=7)
- Result: SP=0x07FE; M[0x07FE]=0xE000; PC=0xC000

```
RETI  ; return from interrupt, SR← M[SP]; SP ← SP + 2; PC ← M[SP]; SP ← SP + 2
```

This instruction is the last instruction inside interrupt service routines. During exception processing when interrupt is accepted, both the PC (R0) and SR (R2) registers are pushed to the stack. Consequently, the RETI expects to find these two registers on the top of the stack. The first value popped from the stack goes into the SR and the next one goes into PC.

Example 8-33. RETI;

- Initial conditions: SP=0x07FC; M[0x07FC]=0x0005; M[0x07FE]=0xC000; PC=0xEF08; SR=0x0000
- Instruction word: 0x1300 (opcode=0001_0011_0b, B/W#=0, Ad=00, D-reg=0)
- Result: SP=0x0800; PC=0xC000; SR=0x0005;

Things to remember 8-2. MSP430 single-operand instructions.

MSP430 supports 7 single-operand core instructions. Their size ranges from 1 to 2 words, depending on the addressing modes.

8.3 Control-Flow Instructions

Figure 11 shows the jump instruction format and Table 6 shows the list of all jump core instructions. Conditional jumps support program branching relative to the PC and they do not affect the status bits. The possible jump range is from −511 to +512 words relative to the PC

value at the jump instruction. The 10-bit program-counter offset is treated as a signed 10-bit value that is doubled and added to the program counter:

PCnew = PCold + 2 + PCoffset × 2

JNE/JNZ jumps to label if the Z flag is reset (condition=000b), JEQ/JZ jumps to label if the Z flag is set (condition=001b), JNC jumps to label if the C bit is reset (condition=010b), JC (jump if higher of same, condition=011b), and so on. Please note that based on the type of operands you are using in your program you should use appropriate control-flow conditions. Thus, inspecting N and V bits is meaningful only if your operands are signed integers, whereas inspecting C bit is meaningful if your operands are unsigned integers.

Figure 11. Jump instruction format (top) and instruction table (bottom).

Table 6. Control-flow instructions (Op-code=001).

Mnemonic	Operands	Condition (binary)	Operation
JNE/JNZ	Label	000	Jump to label if Z bit is reset
JEQ/JZ	Label	001	Jump to label if Z bit is set
JNC	Label	010	Jump to label if C bit is reset
JC	Label	011	Jump to label if C bit is set
JN	Label	100	Jump to label if N bit is set
JGE	Label	101	Jump to label if (N .xor. V)=0
JL	Label	110	Jump to label if (N .xor. V)=1
JMP	Label	111	Jump to label unconditionally

In addition to these control-flow instruction, we can always use MOV instruction to change the flow of the program unconditionally if our destination is PC. Let us consider several examples shown in Table 7.

Table 7. Examples of unconditional branches using MOV instruction.

Instruction	RTL	Description
MOV #LABEL, PC	PC ← #LABEL	The value of the symbol LABEL is copied into PC; equivalent to unconditional jump to the address LABEL

MOV LABEL, PC	PC ← M[LABEL]	The value contained in a memory location at the address LABEL is moved into PC; equivalent to branch to the address contained in the memory location at the address LABEL
MOV @R14, PC	PC ← M[R14]	The value contained in a memory location at the address contained in register R14 is moved into PC; equivalent to branch to the address contained in memory location with the address contained in register R14

Things to remember 8-3. MSP430 control-flow instructions.

MSP430 supports 8 control-flow core instructions. Conditional instructions evaluate a subset of arithmetic flags (C, V, N, and Z) and jump to the target address if the condition is met. The target is computed as PC + 2 + SignExtend(10-bit offset). All control-flow instructions are 1 word long.

In addition, a MOV instruction with PC as the destination register can be used to carry out unconditional jump.

8.4 Emulated Instructions

The previous subsections covered all core instructions. Though the list of core instructions is quite small compared to other architectures, you will not find yourself constrained when developing assembly language programs. To make your development easier, you can use a number of emulated instructions. E.g., if you want to clear the Carry bit you can used BIC instruction by specifying bit 0 in the register R2. So, rather than saying BIC.W #1, R2 you can simply say CLC (Clear Carry). Assembler will figure out how to encode this instruction. Consequently, you should make yourself familiar not only with the core instructions, but also with the emulated instructions as they will make your life easier. Table 8 lists emulated instructions and describes their implementation.

Table 8. Emulated instructions and their implementation.

Mnemonic	Operation	Emulation	Description
Arithmetic instructions			
ADC(.B) dst	dst + C → dst	ADDC(.B) #0,dst	Add carry to destination
DADC(.B) dst	dst + C → dst (decimally)	DADD(.B) #0,dst	Decimal add carry to destination
DEC(.B) dst	dst - 1 → dst	SUB(.B) #1,dst	Decrement destination

DECD(.B) dst	dst – 2 → dst	SUB(.B) #2,dst	Decrement destination twice
INC(.B) dst	dst + 1 → dst	ADD(.B) #1,dst	Increment destination
INCD(.B) dst	dst + 2 → dst	ADD(.B) #2,dst	Increment destination twice
SBC(.B) dst	dst + 0xFFFF + C → dst dst + 0xFF → dst	SUBC(.B) #0,dst	Subtract source and borrow /.NOT. carry from dest.
Logical and register control instructions			
INV(.B) dst	.not. dst → dst	XOR(.B) #0(FF)FFh,dst	Invert bits in destination
RLA(.B) dst	C←MSB←MSB-1 . . . LSB+1←LSB←0	ADD(.B) dst,dst	Rotate left arithmetically
RLC(.B) dst	C←MSB←MSB-1…LSB+1←LSB←C	ADDC(.B) dst,dst	Rotate left through carry
Data instructions			
CLR(.B) dst	0 → dst	MOV(.B) #0,dst	Clear destination
CLRC	0 → C	BIC #1,SR	Clear carry flag
CLRN	0 → N	BIC #4,SR	Clear negative flag
CLRZ	0 → Z	BIC #2,SR	Clear zero flag
POP(.B) dst	M[SP]→dst SP+2→SP	MOV(.B) @SP+,dst	Pop byte/word from stack to destination
SETC	1 → C	BIS #1,SR	Set carry flag
SETN	1 → N	BIS #4,SR	Set negative flag
SETZ	1 → Z	BIS #2,SR	Set zero flag
TST(.B) dst	dst + 0FFFFh + 1dst + 0FFh + 1	CMP(.B) #0,dst	Test destination
Program flow control			
BR dst	dst → PC	MOV dst,PC	Branch to destination
DINT	0 → GIE	BIC #8,SR	Disable (general) interrupts
EINT	1 → GIE	BIS #8,SR	Enable (general) interrupts
NOP	None	MOV #0,R3	No operation
RET	M[SP] → PC SP+2 → SP	MOV @SP+,PC	Return from subroutine

Things to remember 8-4. MSP430 emulated instructions.

MSP430 assembly allows us to use emulated instructions. These instructions do not have their unique opcode, but they are rather derived from the existing core instructions. You should use them to make your programs more readable and to make your task easier.

Figure 12 shows a complete list of the MSP430 core and emulated instructions.

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		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)†	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
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		0	*	*	*
TST(.B)†	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

† Emulated Instruction

Figure 12. The complete MSP430 Instruction Set (core + emulated instructions).

9 Additional Notes On Constant Generator

Constant generator (R2-R3). Profiling common programs for constants shows that just a few constants, such as 0, +1, +2, +4, +8, -1, account for the majority of constants used in programs. However, to encode such constants we will need 16 bits in an instruction. In order to reduce the number of bits needed to encode frequently used constants, a trick called constant generation is used. By specifying dedicated registers R2 and R3 in combination with certain addressing modes, we tell hardware to generate some of the most frequently used constants. This results in a shorter instruction (we need fewer bits to encode such an instruction). Table 9 lists constants created using the constant generator (4, 8, 0, 1, 2, -1). This is achieved by clever encoding as R2/SR is never used as an address register (As=10 and As=11) for indexed or immediate/indirect addressing modes. Register R3 is reserved for generating various constants with different source addressing modes.

Let's say you want to clear a word in memory at the address *dst*. To do this, a MOV instruction could be used:

```
MOV.W #0, dst
```

This instruction would have required 3 words to encode: the first contains the *opcode* and addressing mode specifiers for *src* and *src/dst* operands. The second word would have contained the constant zero, and the third word would have contained the address of the memory location. Alternatively, the instruction

```
MOV.W R3, dst
```

performs the same task, but requires only 2 words. The combination of (As=00) and register R3 as the source register creates a constant 0 within the CPU.

Minimizing the size of a program is referred often as improving code density. We prefer our programs to occupy as little space in memory as possible. It means that we can use chips with smaller memory that are typically cheaper, reducing the cost of our systems and maximizing our profits margins.

Table 9. Constant generation using registers R2 and R3 and address specifier for the source operand As.

Register	As	Constant	Remarks
R2	00	-	Register mode (use SR)
R2	01	(0)	Designates absolute addressing mode.
R2	10	0x0004	+4, used in bit processing
R2	11	0x0008	+8, used in bit processing
R3	00	0x0000	0, word processing
R3	01	0x0001	+1
R3	10	0x0002	+2, bit processing
R3	11	0xFF, 0xFFFF	-1, word processing

10 To Learn More

1. MSP430x5xx and MSP430x6xx Family User's Guide, https://www.ti.com/lit/slau208
2. MSP430x4xx Family User's Guide, https://www.ti.com/lit/pdf/slau056

11 Exercises

Problem #1.

Consider the following instructions given in the table below. For each instruction determine its length (in words), the instruction words (in hexadecimal), source operand addressing mode, and the content of register R7 after execution of each instruction. Fill in the empty cells in the table. The initial content of memory is given in the table below. Initial value of registers R5, R6, and R7 is as follows: R5=0xF002, R6=0xF00A, R7=0xFF88. Assume the starting conditions are the same for each question (i.e., always start from initial conditions in memory) and given register values.

Label	Address [hex]	Memory[15:0] [hex]
	0xF000	0x0504
	0xF002	0xFFEE
TONI	0xF004	0xCC06
	0xF006	0x3304
	0xF008	0xF014
	0xF00A	0x2244
EDE	0xF00C	0xABBA
	0xF00E	0xEFDD

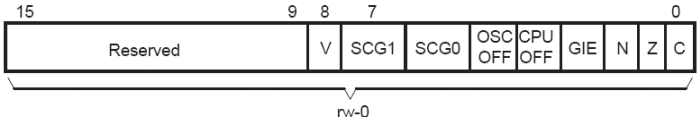
	Instr. Address	**Instruction**	**Instr. Length [words]**	**Instruction Word(s) [hex]**	**Source Operand Addressing Mode**	**R7=? [HEX]**
(i)	**0x1116**	**MOV R5, R7**	**1**	**0x4507**	**Register**	**0xF002**
(ii)	**0x1116**	**MOV.B R5, R7**	**1**	**0x4447**	**Register**	**0x0002**
(a)	0x1116	MOV 4(R5), R7				
(b)	0x1116	MOV.B 3(R5), R7				
(c)	0x1116	MOV.B -3(R6), R7				
(d)	0x1116	MOV TONI, R7				
(e)	0x1116	MOV.B EDE, R7				
(f)	0x1116	MOV &EDE, R7				
(g)	0x1116	MOV.B @R5, R7				
(h)	0x1116	MOV @R5+, R7				
(i)	0x1116	MOV #45, R7				
(j)	0x1116	MOV.B #45, R7				

Problem #2.

Consider the following instructions given in the table below. For each instruction determine addressing modes of the source and destination operands, and the result of the operation. Fill

in the empty cells in the table. The initial content of memory is given in the table. Initial value of registers R2, R5, R6, and R7 is as follows: SR=R2=0x0003 (V=0, N=0, Z=1, C=1), R5=0xC001, R6=0xC008. Assume the starting conditions are the same for each question (i.e., always start from initial conditions in memory and given register values).

Note: Format of the status register (R2) is as follows.



Label	Address [hex]	Memory[15:0] [hex]
	0xC000	0x0504
	0xC002	0xFEEE
TONI	0xC004	0xA821
	0xC006	0x33F4
	0xC008	0xF014
	0xC00A	0x2244
EDE	0xC00C	0xCDDA
	0xC00E	0xEFDD

	Instruction	**Source Addressing Mode**	**Destination Operand Addressing Mode**	**Source Address**	**Dest. Address**	**Result (content of memory location or register)**
(a)	`MOV.B &TONI, R5`					
(b)	`SUBC.B @R6, 5(R5)`					
(c)	`RRC TONI`					
(d)	`AND #0x0AC2, -2(R6)`					

Notes of setting flags: Instructions that set flags, set N and Z flags as usual. Specific details for C and V are as follows: RRC clears V bit.