CPE 323
MSP430 INSTRUCTION SET ARCHITECTURE (ISA)

Aleksandar Milenković
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

Objective

Introduce MSP430 Instruction Set Architecture
(Class of ISA, Memory, Registers, Instructions, Instruction Encoding, Addressing Modes. and Exceptions)

Contents

1. Introduction

Computers cannot execute high-level language constructs like ones found in C or C++. Rather they execute a relatively small set of machine instructions, such as addition, subtraction, Boolean operations, and data transfers. The statements from high-level language are translated into sequences of machine code instructions by a compiler. A machine code instruction is represented by a binary string which is hard to be read by humans. A human-readable form of the machine code is assembly language. However, assembly language may also include some constructs that serve only to help programmers write a better and more efficient code, faster. These constructs may translate into a sequence of machine code instructions.

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. 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 topics of interest for computer organization.

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 (g) exception processing (which will not be discussed in this section). In this document we will discuss the ISA aspects for the MSP430 family of devices.

Before we start discussing the MSP430 ISA, let us introduce MSP430. The MSP430 is a Mixed-Signal Processor (thus MSP) microcontroller family from Texas Instruments. It combines a 16-bit low-power processor with on-chip Flash and RAM memories, and a rich set of analog and digital input/output (I/O) peripherals, including, digital input/output ports, serial communication interfaces (supporting various serial communication protocols), timers, liquid crystal display controllers, analog-to-digital (ADC) converters, digital-to-analog converters (DAC), comparators, and even operational amplifiers. The MSP430 targets low-cost and low-power battery-powered embedded applications. It is used in a range of consumer and industrial applications, such as metering, medical equipment, home appliances, and others. The MSP430 family includes over 230 parts available that range from those that cost as low as 10 cents to those that cost over $10 in high volumes.

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*. There are many reasons why general-purpose register architectures dominate in today's computers. Allocating 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.

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 logical instructions. In register-memory architectures arithmetic and logical 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 logical 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 logical 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 result 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.

3. Memory Architecture

The MSP430 family follows the Von-Neumann architecture – the program and data share a single address space. 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 registers (typically addresses 0x0000 – 0x000F), 8-bit peripheral device registers (0x0010 – 0x00FF), 16-bit peripheral device registers (0x0100 – 0x01FF), RAM memory, and Flash (non-volatile) memory. The processor communicates with memory and I/O peripheral modules 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 word from an even address can be read from or written to. All words are aligned in memory – 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 address space. A byte operand in memory 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 as assume you have a constant 0x4567 and you want to store in memory at address 0x0600. The question is how do you do it? You can place the lower byte of the operand (0x67) to the byte at 0x600 in memory and the upper byte 0x45 to the byte at 0x601 in memory (known as *little-endian* placement policy). 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*).

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

4. Registers

The MSP430 has a fairly large set of registers (a.k.a. register file) for a microcontroller with 16 registers named R0 – R15 that are all visible to programmers. Some of these registers are reserved for special functions. For example, the register R0 is reserved for 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 can be 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. A block diagram of the processor core is shown in Figure 1.

Register-register operations 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.

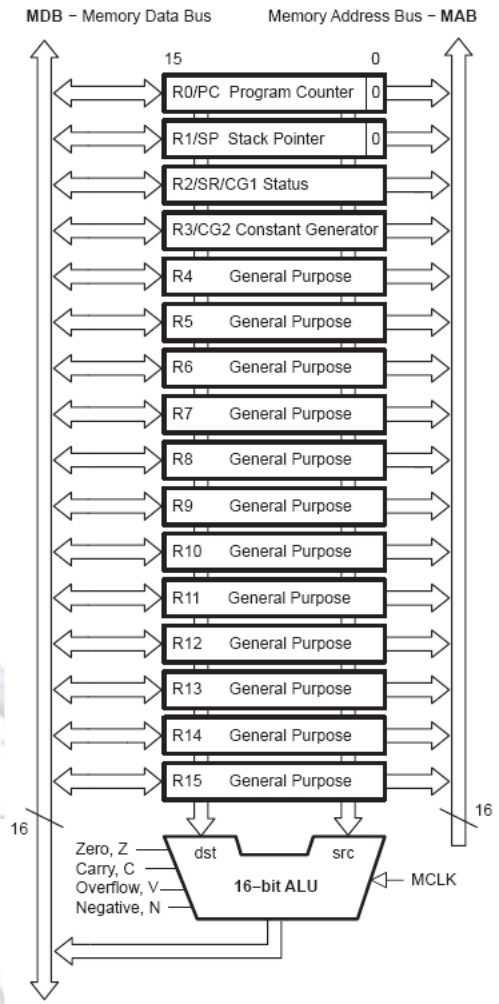


Figure 1. MSP430 CPU Block Diagram.

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 have always an even number of bytes (they are word-aligned), so the least significant bit of the PC is always zero.

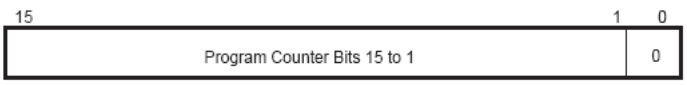


Figure 2. Program Counter.

The PC can be addressed by all instructions. Let us consider several examples:

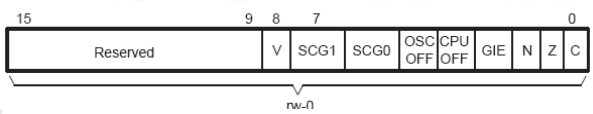
```
MOV #LABEL,PC ; Branch to address LABEL
MOV LABEL,PC ; Branch to address contained in LABEL
MOV @R14,PC ; Branch indirect to address contained in R14
```

Stack pointer (SP/R1). The program stack is a dynamic LIFO (Last-In-First-Out) structure allocated in RAM memory. The stack is used to store the return addresses of subroutine calls and interrupts, as well as the storage for local data and passing parameters for 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 toward 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 (the SP is first decremented by 2, and then the operand is stored in memory at the location addressed by the SP), and POP (the content from the top of the stack is retrieved; the SP is incremented by 2).

Let us consider the instruction PUSH R7. This instruction pushes the content of the register R7 onto 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 toward lower addresses in RAM memory, decrementing stack is equivalent to the following operation: SP ← SP – 2. The next step is move the content of the register R7 onto that location. We describe this step as follows: M[SP] ← SP (or the memory location with the address contained in SP/R1 is loaded with the content of register R7).

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 and the meaning of the individual bits is show in Figure 3.



Bit	Description	
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range.	
	`ADD(.B),ADDC(.B)`	Set when: Positive + Positive = Negative Negative + Negative = Positive, otherwise reset
	`SUB(.B),SUBC(.B),CMP(.B)`	Set when: Positive – Negative = Negative Negative – Positive = Positive, otherwise reset
SCG1	System clock generator 1. This bit, when set, turns off the SMCLK.	
SCG0	System clock generator 0. This bit, when set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
OSCOFF	Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK	
CPUOFF	CPU off. This bit, when set, turns off the CPU.	
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative.	
	Word operation:	N is set to the value of bit 15 of the result
	Byte operation:	N is set to the value of bit 7 of the result
Z	Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

Figure 3. Status register format (top) and bits description (bottom).

Constant generator (R2-R3). Profiling common programs for constants shows that just a few constants, such as 0, +1, +2, +4, +8, -1, are responsible for majority of program constants. However, to encode such a constant we will need 16 bits in our instruction. In order to reduce the number of bits spent for encoding 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 certain constants. This results in shorter instructions (we need less bits to encode such an instruction). Figure 4 describes values of constant generators. 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.

Register	As	Constant	Remarks
R2	00	-----	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	−1, word processing

Figure 4. Constant generation.

An example: 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 #0, *dst*

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

MOV R3, dst

performs the same task, but requires only 2 words to encode it.

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.

Let us consider a register-byte operation using the following instruction:

ADD.B R5, 0(R6).

This instruction specifies that the *src* operand is the register R5 (lower 8 bite or R5), and the *src/dest* operand is in memory at the address (R6+0). The suffix .B indicates that the operation should be performed on byte-size operands. Thus, a lower byte from the register R5, 0x8F, is

added to the byte from the memory location Mem(0x0203)=0x12, and the result is written back, so the new value of Mem(0x0203)=0xA1. The content of the register R5 is intact.

Let us now consider a byte-register operation using the following instruction:

ADD.B @R6, R5.

This instruction specifies a source operand in memory at the address contained in R6, and the destination operand is in the register R5. A suffix .B is used to indicate that the operation uses byte-sized operands. A suffix .W indicates that operations are performed on word-sized operands and is default (i.e., by omitting .W we imply word-sized operands). As shown below, a byte value Mem(0x0223)=0x5F is added to the lower byte of R5, 0x02. The result of 0x61 is zero extended to whole word, and the result is written back to register R6. So, the upper byte is always cleared in case of byte-register operations.

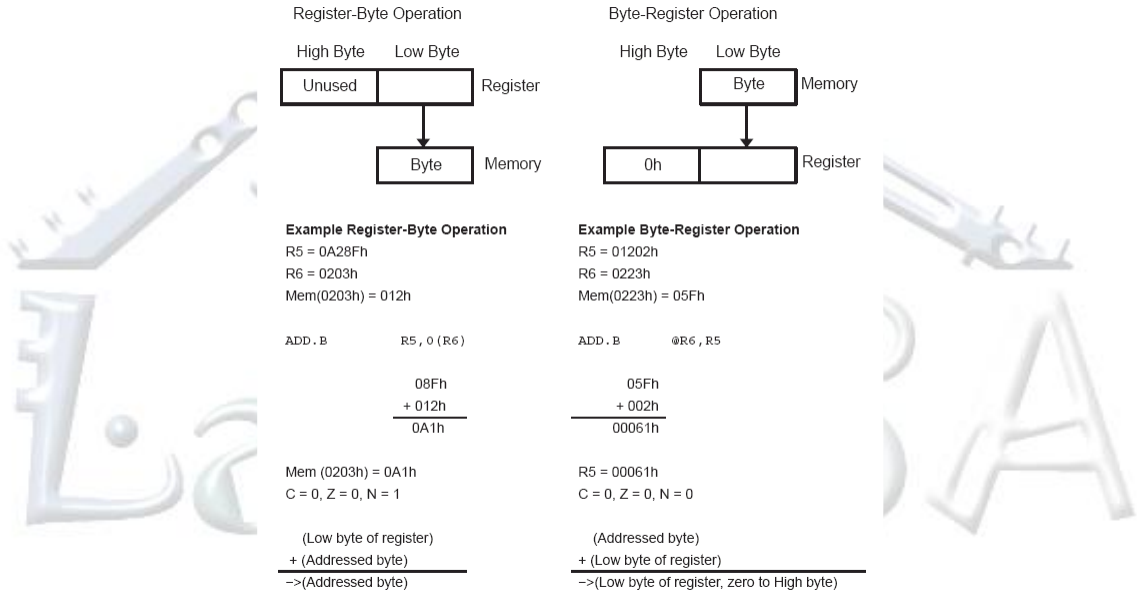


Figure 5. Register-byte (left) and byte-register (right) operations.

5. 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 6.

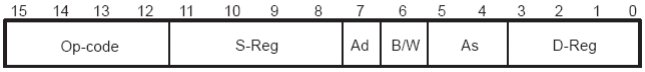


Figure 6. First-word instruction format for double-word instruction.

The meaning of individual fields is as follows.

- *Op-code* – Encodes the instruction (type of operation like MOV, ADD, ..)
- *src* – The source operand defined by *As* and *S-reg*

    - *As* – The addressing bits responsible for the addressing mode used for the source (src)
    - S-reg – The working register used for the source (*src*)
- *dst* – The destination operand defined by *Ad* and *D-reg*
    - *Ad* – The addressing bits responsible for the addressing mode used for the destination (dst)
    - *D-reg* – The working register used for the destination (*dst*)
- *B/W* – Byte or word operation:
    - 0: word operation
    - 1: byte operation

6. Addressing Modes

The MSP430 architecture supports a relatively rich set of addressing modes. Seven of addressing modes can be used to specify a source operand in any location in memory (Figure 7), and the first four of these can be used to specify the source/destination operand. Figure 7 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.

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/–	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/–	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/–	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Figure 7. Addressing Modes.

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

Example: `MOV.B R5, R7;  R7 ← R5`
Source addressing mode: register, register id: 5 => As=00, S-reg=0101
Destination addressing mode: register, register id: 7 => Ad=0, D-reg=0111

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: **`MOV.B 10(R5), 12(R7); M[R7+12] ← M[R5+10]`**
Source addressing mode: indexed, register id: 5, offset=10 => As=01, S-reg=0101, 2nd word=10
Destination addressing mode: indexed, register id: 7, offset=12 => Ad=1, D-reg=0111; 3rd word=12
Machine code:
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+X.

Example: **`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 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.
Source addressing mode: symbolic, register id: 0 => As=01, S-reg=0000,
EA.src=0x0200=PC+Offset.src =0xE002+Offset.src=> Offset.src=0x0200-0xE002=0x21FE
Destination addressing mode: symbolic, register id: 0 => Ad=1, D-reg=0000
EA.dst=0x0300=PC+Offset.dst=0xE004+Offset.dst => Offset.dst=0x0300-0xE004=0x22FC

Machine code:
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.

Example: **`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.

Source addressing mode: absolute, register id: 2 => As=01, S-reg=0010, EA.src=0x0200 => 2nd word of instruction is 0x0200.
Destination addressing mode: absolute, register id: 2 => Ad=1, D-reg=0010
EA.dst=0x0300 => 3rd word is 0x0300.

Machine code:
1st word: 0100_0010_1101_0010;
2nd word: 0000_0010_0000_0000;
3rd word: 0000_0011_0000_0000
or 0x4242; 0x0200; 0x0300

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

Example: **`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.
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
EA.dst=0x0300 => 2nd word is 0x0300.

Machine code:
1st word: 0100_0101_1110_0010;
2nd word: 0000_0011_0000_0000;
or 0x45E2; 0x0300

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

Example: **`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.
Source addressing mode: register indirect with autoincrement, register id: 5 => As=11, S-reg=0101, EA.src=R5=0x0200
Destination addressing mode: absolute, register id: 2 => Ad=1, D-reg=0010
EA.dst=0x0300 => 2nd word is 0x0300

Machine code:

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: **`MOV.B #45, &EDE;M[EDE] ← #45`**
Assume EDE is a symbolic name for addresses 0x0300;
This instruction requires 2 words, where the second one carries the immediate operand 45.
Source addressing mode: immediate, register id: 0 => As=11, S-reg=0000
Destination addressing mode: absolute, register id: 2 => Ad=1, D-reg=0010
EA.dst=0x0300 => 2nd word is 0x0300

Machine code:
1st word: 0100_0000_1111_0010;
2nd word: 0000_0000_0010_1101;
or 0x40F2; 0x002D

One should notice a smart encoding of the addressing modes. Only a 2-bit address specifier *As* is sufficient to encode 7 addressing modes. How does it work? Please note that the absolute addressing mode is encoded in the same way as the indexed and the symbolic modes, *As*=01. With indexed mode we specify a general-purpose register that is used as the base address and an address displacement that is encoded in a following instruction word. However, to use the status register in calculating the effective memory address would be meaningless and it is never used for that. This fact can be used so that when register R2 is specified as the address register, we actually interpret 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*). Similarly, when using register (R0) as the address register we actually specify the symbolic addressing mode (please note that the register R0 is actually PC). Strictly speaking, the symbolic mode is a special case of the indexed addressing mode. This way, we have one address specifier (*As*=01) that covers 3 different addressing modes.

Next, the immediate mode uses the same address specifier as the autoincrement mode, As=*11*. 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). Similarly, we can explain how only a single bit *Ad* suffices in distinguishing 4 addressing modes for the destination operand (Ad=1 covers the same 3 addressing modes, symbolic, absolute, indexed using an additional *src/dst* register field).

7. 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:

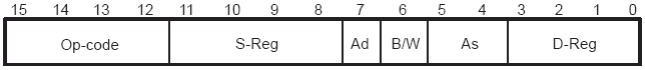
- Double-operand
- Single-operand
- Jump

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.

The source and destination of an instruction are defined by the following fields:

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

Figure 8 shows the double-operand instruction format and the list of all double-operand core instructions.

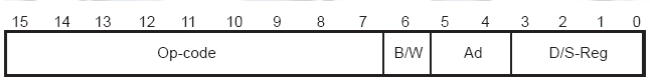


Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src,dst	src → dst	–	–	–	–
ADD(.B)	src,dst	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src,dst	dst – src	*	*	*	*
DADD(.B)	src,dst	src + dst + C → dst (decimally)	*	*	*	*
BIT(.B)	src,dst	src .and. dst	0	*	*	*
BIC(.B)	src,dst	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	src .or. dst → dst	–	–	–	–
XOR(.B)	src,dst	src .xor. dst → dst	*	*	*	*
AND(.B)	src,dst	src .and. dst → dst	0	*	*	*

* The status bit is affected
– The status bit is not affected
0 The status bit is cleared
1 The status bit is set

Figure 8. Double-operand first-word instruction format (top) and instruction table (bottom).

Figure 9 shows the single-operand instruction format and the list of all single-operand core instructions.



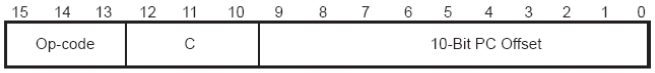
Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
RRC(.B)	dst	C → MSB →.....LSB → C	*	*	*	*
RRA(.B)	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH(.B)	src	SP – 2 → SP, src → @SP	–	–	–	–
SWPB	dst	Swap bytes	–	–	–	–
CALL	dst	SP – 2 → SP, PC+2 → @SP	–	–	–	–
		dst → PC				
RETI		TOS → SR, SP + 2 → SP	*	*	*	*
		TOS → PC,SP + 2 → SP				
SXT	dst	Bit 7 → Bit 8.....Bit 15	0	*	*	*

* The status bit is affected
– The status bit is not affected
0 The status bit is cleared
1 The status bit is set

Figure 9. Single-operand first-word instruction format (top) and instruction table (bottom).

Figure 9 shows the jump instruction format and the list of all jump core instructions. Conditional jumps support program branching relative to the PC and 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



Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if (N .XOR. V) = 0
JL	Label	Jump to label if (N .XOR. V) = 1
JMP	Label	Jump to label unconditionally

Figure 10. Jump instruction format (top) and instruction table (bottom).

Figure 11 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 11. The complete MSP430 Instruction Set (core + emulated instructions).

8. To Learn More

1. MSP430 User Manual, http://www.ece.uah.edu/~milenka/npage/data/cpe323/Documents/slau056j-4xx-UG.pdf
2. Textbook, Chapter 5