CPE 323 MSP430 INSTRUCTION SET ARCHITECTURE

Aleksandar Milenkovic

The MSP430 is a 16-bit, byte-addressable, RISC-like architecture. The main characteristics of the instruction set architecture are as follows.

Registers

The MSP430 has a register file with 16 registers (R0-R15) that are all visible to programmers. Register R0 is reserved for the program counter (PC), register R1 serves as the stack pointer, and register R2 serves as the status register. Register R3 can be used for constant generation, while the remaining registers R4-R15 serve as general-purpose registers.

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.

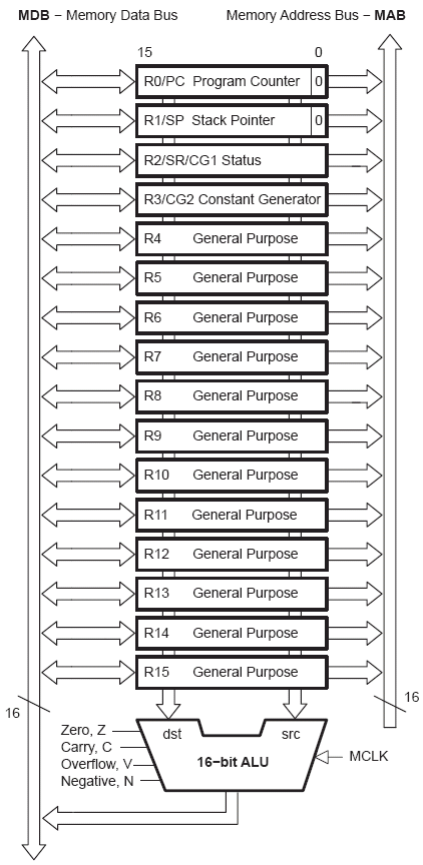


Figure 1. MSP430 CPU Block Diagram.

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/dest) operands. 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.

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

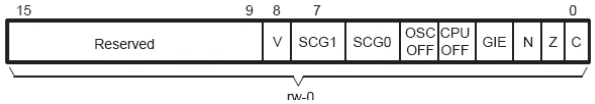
```
MOV #LABEL,PC ; Branch to address LABEL
MOV LABEL,PC ; Branch to address contained in LABEL
MOV @R14,PC ; Branch indirect to address in R14
```

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. 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).

[[Illustrate PUSH and POP operations on 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 and the meaning of the individual bits is show in Figure 2.



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 2. 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). **Error! Reference source not found.** describes values of constant generators.

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 3. Constant generation.

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

MOVE #0, *dst*

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

MOVE R3, dst

performs the same task, but we need 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 a source operand is the register R5, and 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.

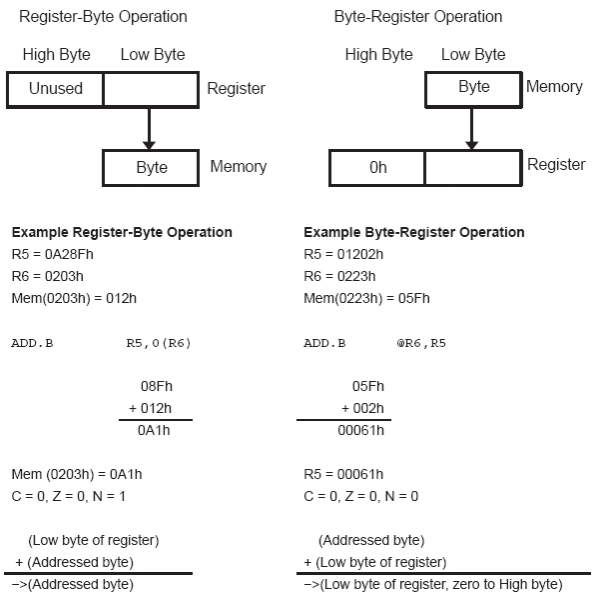


Figure 4. Register-byte (left) and byte-register (right) operations.

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 5), and the first four of these can be used to specify the source/destination operand. Figure 5 also illustrates the syntax and give a short description of the addressing modes. The addressing modes are encoded using As and Ad 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 5. 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).

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.

Symbolic mode. This addressing mode can be considered as a subset of the indexed mode. The only difference is that the address register is the PC, and thus ea=PC+X.

Absolute mode. The instruction specifies the absolute (or direct) address of the operand in memory. The instruction includes a word that specifies this address.

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

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.

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

One should notice a smart encoding of the addressing modes. Only a 2-bit address specifier As is sufficient in encoding 7 addressing modes. How does it work? For example, please note that the absolute addressing mode is encoded in the same way as the indexed and the symbolic modes, As=01. However, the absolute mode specifies the SR register as the address register. It is never used in the indexed mode as an address register, so this combination indicates the absolute addressing. Next, the immediate mode uses the same As=11 as the autoincrement mode. It is distinguished from the autoincrement mode because the specified register is the PC, which is never used in the autoincrement mode. Similarly, we can explain how only a single bit *Ad* suffices in distinguishing 4 addressing modes for the destination operand.

[[Examples]]

Instruction Set

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:

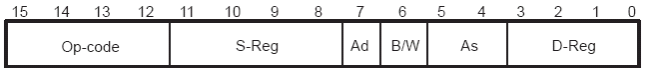
- Double-operand
- Single-operand
- Jump

All single-operand and double-operand instructions can be byte or word instructions 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:

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

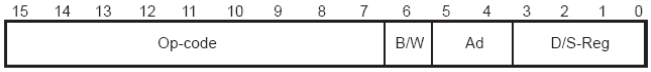


Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src,dst	src → dst	–	–	–	–
ADD(.B)	src,dst	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src,dst	dst – src	*	*	*	*
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 6. Double-operand instruction format (top) and instruction table (bottom).

Figure 7shows the single-operand instruction format and the list of all single-operand core instructions.

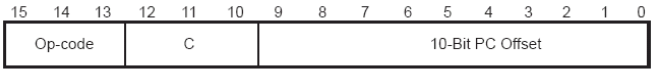


Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
RRC(.B)	dst	C → MSB →.....LSB → C	*	*	*	*
RRA(.B)	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH(.B)	src	SP – 2 → SP, src → @SP	–	–	–	–
SWPB	dst	Swap bytes	–	–	–	–
CALL	dst	SP – 2 → SP, PC+2 → @SP	–	–	–	–
		dst → PC				
RETI		TOS → SR, SP + 2 → SP	*	*	*	*
		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 7. Single-operand instruction format (top) and instruction table (bottom).

Figure 7shows 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 8. Jump instruction format (top) and instruction table (bottom).

Figure 9 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 9. The complete MSP430 Instruction Set (core + emulated instructions).