

Architecture and Instruction Set

8.1 Introduction

The instruction set of the *ultra low power*-microcomputer MSP430 family differs strongly from the instruction sets used by other 8-bit and 16-bit microcomputers. The reasons why this instruction set is appreciated though, are explained in the following pages in detail. It is the return to clarity and especially the return to orthogonality, an attribute of microcomputer architectures that has disappeared more and more during the last 20 years. A customer commented that it is an instruction set to fall in love with.

The MSP430 Family was developed to fulfill the ever increasing requirements of Texas Instruments Ultra Low Power microcomputers. It was not possible to increase the computing power and the real-time processing capability of the MSP430 predecessor (TSS400) as far as was needed. Therefore, a complete new 16-bit architecture was developed to stay competitive and be viable for several years.

The benchmark numbers shown in relation to competition's products (bytes used, number of program lines) are taken from an unbiased comparison executed by a British software consultant.

8.2 Reasons for the Popularity of the MSP430

The following sections are intended to explain the different reasons why the MSP430 instruction set, which closely mirrors the architecture, has become so popular.

8.2.1 Orthogonality

This notation of computer science means that a single operand instruction can use any addressing mode or that any double operand instruction can use any combination of source and destination addressing modes. Figure 8–1 shows this graphically: the existing combinations fill the complete possible space.

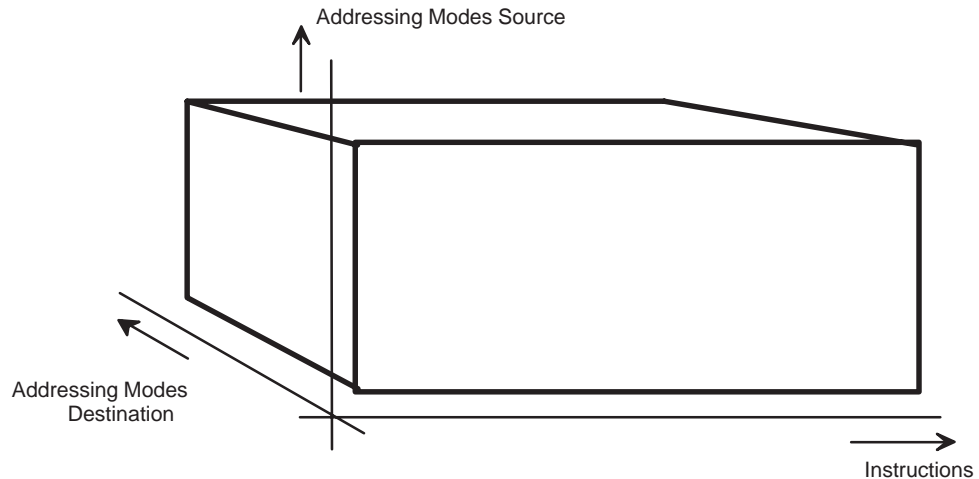


Figure 8–1. Orthogonal Architecture (Double Operand Instructions)

The opposite of orthogonal, a non-orthogonal architecture is shown in Figure 8–2. Any instruction can use only a part of the existing addressing modes. The possible combinations are arranged like small blocks in the available space.

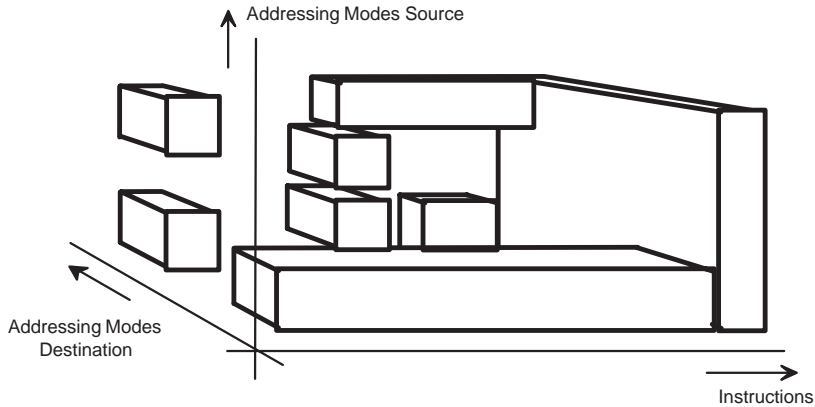




Figure 8–2. Non-Orthogonal Architecture (Dual Operand Instructions)

8.2.2 Addressing Modes


The MSP430 architecture has seven possibilities to address its operands. Four of them are implemented in the CPU, two of them result from the use of the program counter (PC) as a register, and a further one is claimed by indexing a register that always contains a zero (status register).

The single operand instructions can use all of the seven addressing modes, the double operand instructions can use all of them for the source operand, and four of them for the destination operand. Figure 8–3 shows this context:

Double Operand Instructions

Mnemonic	Source, Destination
	<div style="display: flex; justify-content: space-around;"> <div style="text-align: center;">  </div> <div style="text-align: center;">  </div> </div>
Register	Register
Indexed	Indexed
Absolute	Absolute
Symbolic	Symbolic
Immediate	
Register indirect	
Register indirect autoincrement	

Single Operand Instructions

Mnemonic	Destination
	<div style="text-align: center;">  </div>
Register	Register
Indexed	Indexed
Absolute	Absolute
Symbolic	Symbolic
Immediate	Immediate
Register indirect	Register indirect
Register indirect autoincrement	Register indirect autoincrement

12 Instructions 28 Combinations

7 Instructions 7 Addressing Modes

Figure 8–3. Addressing Modes

8.2.2.1 Register Addressing

The operand is contained in one of the registers R0 to R15. This is the fastest addressing mode and the one that needs the least memory. Example:

```
; Add the contents of R7 to the contents of R8
;
      ADD    R7,R8          ; (R7) + (R8) → (R8)
```

8.2.2.2 Indirect Register Addressing

The register used contains the address of the operand. The operand can be located anywhere in the entire memory space (64K). Example:

```
; Add the byte addressed by R8 to the contents of R9
;
      ADD.B  @R8,R9        ; ((R8)) + (R9) → (R9)
```

8.2.2.3 Indirect Register Addressing With Autoincrement

The register used contains the address of the operand. This operand can be located anywhere in the entire memory space (64K). After the access to the operand the used register is incremented by two (word instruction) respective one (byte instruction). The increment occurs immediately after the reading of the source operand. Example:

```
; Copy the byte operand addressed by R8 to R9
; and increment the pointer register R8 by one afterwards
;
      MOV.B  @R8+,R9       ; ((R8)) → (R9), (R8) + 1 → (R8)
```

8.2.2.4 Indexed Addressing

The address of the operand is the sum of the index and the contents of the register used. The index is contained in an additional word located after the instruction word. Example:

```
; Compare the 2nd byte of a table addressed by R5 with the
; low Byte of R15. Result to the Status Register SR
;
      CMP.B  1(R5),R15     ; (R15) - (1 + (R5))
;
```

If the register in use is the program counter then two additional, important addressing modes result:

8.2.2.5 Immediate Addressing

Any 16-bit or 8-bit constant can be used with an instruction. The PC points to the following word after reading the instruction word. By the use of the *register indirect autoincrement* addressing mode, this word (the immediate value) can be read and the PC is incremented by two afterwards. The word after the instruction word is treated this way as an 8-bit or a 16-bit immediate value. Example:

```
; Test bit 8 in the 3rd word of a table R10 points to.
; Start address of the table is 0(R10), 3rd word is 4(R10)
;
        BIT        #0100h,4(R10)    ; Bit 8 = 1?
;
; The assembler inserts for the instruction above:
;
        BIT        @PC+,4(R10)      ; Executed instruction
        .WORD      0100h            ; Source constant 0100h
        .WORD      0004h            ; Index 4 of the destination
```

8.2.2.6 Symbolic Addressing

This is the normal addressing mode for the random access to the entire 64K memory space. The word located after the instruction word contains the difference in bytes to the destination address relative to the PC. This difference can be seen as an index to the PC. Any address in the 64K memory map is addressable this way, both as a source and as a destination.

Example: \$ = address the PC points to

```
; Subtract the contents of the ROM word EDE from the contents
; of the RAM word TONI
;
        SUB        EDE,TONI          ; (TONI) - (EDE) → (TONI)
;
; The assembler inserts for the instruction above:
;
        SUB        X(PC),Y(PC)      ; Executed instruction
        .WORD      X                ; Index X = EDE-$
        .WORD      Y                ; Index Y = TONI-$
```

8.2.2.7 Absolute Addressing

Addresses that are fixed (e.g., the hardware addresses of the peripherals like ADC, UART) can be addressed absolutely. The absolute addressing mode is a special case of the indexed addressing mode. The register used (SR) always contains a zero in this case (without losing its former information!). Example:

```
; Set the Power Down Bit in the ADC Control Register ACTL
;
        BIS    #PD,&ACTL        ; Power Down Bit ADC <- 1
;
; The assembler inserts for the instruction above:
;
        BIS    @PC+,X(SR)        ; Executed instruction
        .WORD  01000h            ; PD Bit Hardware Address
        .WORD  00114h            ; X: Hardware Address of ACTL
```

8.2.3 RISC Architecture Without RISC Disadvantages

Classic RISC architectures provide several addressing modes only for the LOAD and STORE instructions; all other instructions can only access the (numerous) registers. The MSP430 can be programmed this way too. An example of this programming style is the floating point package FPP4. The registers are loaded during the initialization, the calculations are made exclusively in the registers, and the result is placed onto the stack.

In real time applications, this kind of programming is less usable, here it is important to access operands at random addresses without any delays. An example of this is the incrementing of a counter during an interrupt service routine:

```
; Pure RISC program sequence for the incrementing of a counter
;
INT_HND  PUSH   R5                ; Save register
         LOAD   R5,COUNTER        ; Load COUNTER to register
         INC    R5                ; Increment this register
         STORE  R5,COUNTER        ; Store back the result
         POP    R5                ; Restore used register
         RETI                     ; Return from interrupt
;
; The MSP430 program sequence for the incrementing of a counter
;
INT_HND  INC    COUNTER           ; Increment COUNTER
         RETI                     ; Return from interrupt
```

As shown in the previous example, the pure RISC architecture is not optimal in cases with few calculations, but necessary for fast access to the memory.

Here, the MSP430 architecture is advantageous due to the random access to the entire memory (64K) with any instruction, seven source addressing modes and four destination addressing modes.

8.2.4 Constant Generator

One of the reasons for the high code efficiency of the MSP430 architecture is the constant generator. The constants, appearing most often in assembler software, are small numbers. Out of these, six were chosen for the constant generator:

Table 8–1. Constants implemented in the Constant Generator

CONSTANT	HEX REPRESENTATION	USE
–1	0FFFFh	Constant, all bits are one
0	0000h	Constant, all bits are zero
+1	00001h	Constant, increment for byte addresses
+2	00002h	Constant, increment for word addresses
+4	00004h	Constant, value for bit tests
+8	00008h	Constant, value for bit tests

These six constants can also be used for byte processing. Only the lower byte is in use then.

The use of numbers out of the constant generator has two advantages:

Memory Space: The constant does not need an additional 16 bit word as it is the case with the normal immediate mode. Two useless addressing modes of the status registers SR and all four addressing modes of the otherwise un-serviceable register R3 are used.

Speed: The constant generator is implemented inside the CPU which results in an access time similar to a general purpose register (shortest access time).

Most of the emulated instructions use the constant generator. See Chapter *The MSP430 Instruction Set* for examples.

8.2.5 Status Bits

The influence of the instructions to the status bits contained in SR is not as uniform as the instructions appear. Dependent on the main use of the instruction, the status bits are influenced in one of the following three ways shown:

- 1) Not at all, the status bits are not affected. This is, for example, the case with the instructions bit clear, bit set and move.

- 2) Normal: the status bits reflect the result of the last instruction. This is used with all arithmetical and logical instructions (except bit set and bit clear)
- 3) Normal, but the carry bit contains the inverted zero bit. The logical instructions XOR (exclusive or), BIT (bit test) and AND use the carry bit for the non-zero information. This feature can save ROM space and time. No preparations or conditional jumps are necessary. The tested information, which is contained in the carry bit, is simply shifted into a register or a RAM word respective byte.

8.2.6 Stack Processing

The stack processing capability of the MSP430 allows any nesting of interrupts, subroutines, and user data. It is only necessary to have enough RAM space. Due to the function of the SP as a general purpose register, it is possible to use all seven of the addressing modes for the SP. This allows any needed manipulation of data on the stack. Any word or byte on the stack can be addressed and may therefore be read and written. (The addressing modes implemented for the MSP430 were chosen primarily for the addressing of the stack; but they proved to be very effective also for the other registers).

8.2.7 Usability of the Jumps

Remarkable is the uncommonly wide reach of the jumps which is ± 512 words. This value is eight times the reach of other architectures that use normally ± 128 bytes. Inside program parts it is, therefore, necessary only very rarely to use the branch instruction with its normal two memory words and longer execution time. The implemented eight jumps are classified in three categories:

- 1) Signed jumps: Numbers range from -32768 to $+32767$ (word instructions) respective -128 to $+127$ (byte instructions)
- 2) Unsigned jumps: Numbers range from 0 to 65535 respective 0 to 255
- 3) Unconditional jump: (replaces the branch instruction normally)

8.2.8 Byte and Word Processor

Any MSP430 instruction is implemented for byte and word processing. Exceptions are only the instructions where a byte instruction would not make sense (subroutine call CALL, return from interrupt RETI) or instructions that are used as an interface between words and bytes (swap bytes SWPB, sign extension SXT). The addressable memory of the MSP430 is divided into bytes and words as shown in Figure 8–4.

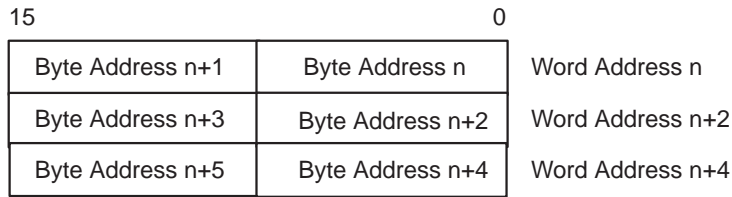


Figure 8–4. Word and Byte Addresses of the MSP430

This way, the entire 64K address space is organized. The planned memory extension will be addressed in the same clear manner. Due to this memory organization, any table can be allocated in the most favorable manner. Dependent on the maximum value of the operands, the table can be implemented as a byte table or a word table. Any general purpose register from R4 to R15 can be used as a pointer to the tables. The implemented addressing modes indexed, indirect, and indirect with autoincrement are intended for table processing.

8.2.9 High Register Count

In addition to the PC and the SP, which are usable for several purposes, twelve identical general purpose registers (R4 to R15) are available. Anyone of these registers can be used as a data register, as an address pointer, as an auto-incrementing address pointer, and as an index register. The bottleneck of the accumulator architectures, which have to pass any operation through the accumulator (with corresponding LOAD and STORE instructions), does not exist for the MSP430.

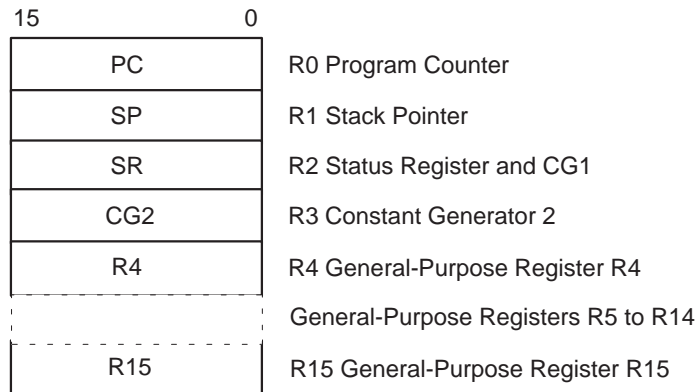


Figure 8–5. Register Set of the MSP430

8.2.10 Emulation of Non-Implemented Instructions

The 27 implemented instructions allow the emulation of additional 24 instructions. This is normally reached with the help of the constant generator, but other ways are used also. As the constants used are taken from the constant generator, no additional memory space is needed.

The assembler automatically inserts the correct instructions if emulated instructions are used. The emulation of the 24 instructions led to a remarkable smaller central processing unit. The MSP430 CPU is even smaller than some 4-bit CPUs. The emulated instructions are completely listed in Section 8.4.2, *Emulated Instructions*.

8.2.11 No Paging

The 16-bit architecture of the MSP430 allows the direct addressing of the entire 64K memory bytes without paging of the memory. This feature greatly simplifies the development of software.

8.3 Effects and Advantages of the MSP430 Architecture

The reasons for the popularity of the MSP430 instruction set (and by its architecture) shown in Section 8.2, have effects and advantages that also result in money saved. These effects and advantages are shown and explained in the following.

8.3.1 Less Program Space

The direct access to any address, without loading of the previous address into a register, together with the constant generator results in program lengths for a given task that are between 55% and 90% compared to other microcomputers. This means that with the MSP430, a 4K version may be sufficient where with other microcomputers a 6K or 8K version is needed.

8.3.2 Shorter Programs

Any necessary code line is a source of errors. The less code lines that are necessary for a given task, the simpler a program is to read, understand, and service. The MSP430 needs between 33% and 55% of the code lines compared to its competition's products. The reason for this is the same as described previously. Any address can be accessed directly and that both for the source operand and for the destination operand. It is not necessary to create troublesome 16-bit addresses, handle the operands byte-wise, and store the final result afterwards indirectly via a composed destination address. All this happens with only one MSP430 instruction.

8.3.3 Reduced Development Time

The clearly smaller program length and the less troublesome access to ROM, RAM, and other peripherals reduce the necessary development time. In addition to that advantage, the considerations omit completely how the actual problem can be solved at all with the given architecture. A part that can take up to one third of the development time with other architectures. (Whoever has developed with 4-bit microcomputers knows what is meant).

8.3.4 Effective Code Without Compressing

The clear assembler language of the MSP430 allows, from the start, the writing of dense and legible code. If the developed program is well prepared and coded clearly, it is nearly impossible to reduce the program length seriously afterwards by compressing. This is no disadvantage, it simply means that optimized code was developed from the start.

8.3.5 Optimum C Code

The C compiler of a microcomputer can use only the instructions that have a regular structure. Typical CISC (complex instruction set computer) instructions, which normally show strong addressing mode restrictions, are not used by the compilers. Instead, the compilers emulate the complex instructions with several of the simple instructions, resulting in a use of only 30% (!) of the implemented instructions.

This is completely different with the MSP430. As the instructions (apart from the executed operation) are completely uniform, 100% of them are used by the compiler and not just 30%. As logical and arithmetical operations are executed directly and not by composed instructions, the execution time of the compiled code is shorter and less memory space is needed. Therefore, the same advantages that are valid for assembler programming are valid also for high-level language programming.

8.4 The MSP430 Instruction Set

In the following are all implemented and emulated instructions.

Description of the used abbreviations:

@	Logical NOT-Function
*	Status Bit is affected by the instruction
–	Status Bit is not affected by the instruction
0	Status Bit is cleared by the instruction
1	Status Bit is set by the instruction
V	Overflow Bit in Status Register SR
N	Negative Bit in Status Register SR
Z	Zero Bit in Status Register SR
C	Carry Bit in Status Register SR
src	Source Operand with 7 Addressing Modes
dst	Destination Operand with 4 Addressing Modes
xx.B	Byte Operation of the Instruction xx
Label	Label of the source or destination

8.4.1 Implemented Instructions

The instructions that are implemented in the CPU follow.

8.4.1.1 Two Operand Instructions

				Status Bits			
				V	N	Z	C
ADD	ADD.B	src,dst	Add src to dst	*	*	*	*
ADDC	ADDC.B	src,dst	Add src + Carry to dst	*	*	*	*
AND	AND.B	src,dst	src .and. dst → dst	0	*	*	@Z
BIC	BIC.B	src,dst	@src .and. dst → dst	–	–	–	–
BIS	BIS.B	src,dst	src .or. dst → dst	–	–	–	–
BIT	BIT.B	src,dst	src .and. dst → SR	0	*	*	@Z
CMP	CMP.B	src,dst	Compare src and dst (dst – src)	*	*	*	*
DADD	DADD.B	src,dst	Add src + Carry to dst (dec.)	*	*	*	*
MOV	MOV.B	src,dst	Copy src to dst	–	–	–	–
SUB	SUB.B	src,dst	Subtract src from dst	*	*	*	*
SUBC	SUBC.B	src,dst	Subtract src with Carry from dst	*	*	*	*
XOR	XOR.B	src,dst	src .xor. dst → dst	*	*	*	@Z

8.4.1.2 Single Operand Instructions

The operand (src or dst) can be addressed with all seven addressing modes.

				Status Bits			
				V	N	Z	C
CALL		dst	Subroutine call	–	–	–	–
PUSH	PUSH.B	src	Copy operand onto stack	–	–	–	–
RETI			Interrupt return	*	*	*	*
RRA	RRA.B	dst	Rotate dst right arithmetically	0	*	*	*
RRC	RRC.B	dst	Rotate dst right through Carry	*	*	*	*
SWPB		dst	Swap bytes	–	–	–	–
SXT		dst	Sign extension into high byte	0	*	*	@Z

8.4.1.3 Conditional Jumps

The status bits are not affected by the jumps. With the signed jumps (JGE, JLT), the overflow bit is evaluated also, so that the jumps are executed correctly even in the case of overflow. Some jumps are the same (JC/JHS, JZ/JEQ, JNC/JLO, JNE/JNZ) but two mnemonics are used to get a better understanding of the program code. In case of a comparison JHS gives a better understanding of the code than JC.

JC	Label	Jump if Carry = 1
JHS	Label	Jump if dst is higher or same than src (C = 1)
JEQ	Label	Jump if dst equals src (Z = 1)
JZ	Label	Jump if Zero Bit = 1
JGE	Label	Jump if dst is greater than or equal to src (N .xor. V = 0)
JLT	Label	Jump if dst is less than src (N .xor. V = 1)
JMP	Label	Jump unconditionally
JN	Label	Jump if Negative Bit = 1
JNC	Label	Jump if Carry = 0
JLO	Label	Jump if dst is lower than src (C = 0)
JNE	Label	Jump if dst is not equal to src (Z = 0)
JNZ	Label	Jump if Zero Bit = 0

8.4.2 Emulated Instructions

The emulated instructions use implemented instructions together with constants coming out of the constant generator.

				Status Bits			
				V	N	Z	C
ADC	ADC.B	dst	Add Carry to dst	*	*	*	*
BR		dst	Branch indirect dst	—	—	—	—
CLR	CLR.B	dst	Clear dst	—	—	—	—
CLRC			Clear Carry Bit	—	—	—	0
CLRN			Clear Negative Bit	—	0	—	—
CLRZ			Clear Zero Bit	—	—	0	—
DADC	DADC.B	dst	Add Carry to dst (decimally)	*	*	*	*
DEC	DEC.B	dst	Decrement dst by 1	*	*	*	*
DECD	DECD.B	dst	dst - 2 → dst	*	*	*	*
DINT			Disable interrupts	—	—	—	—
EINT			Enable interrupts	—	—	—	—
INC	INC.B	dst	Increment dst by 1	*	*	*	*
INCD	INCD.B	dst	dst + 2 → dst	*	*	*	*
INV	INV.B	dst	Invert dst	*	*	*	@Z
NOP			No operation	—	—	—	—
POP	POP.B	dst	Pop top of stack to dst	—	—	—	—
RET			Subroutine return	—	—	—	—
RLA	RLA.B	dst	Rotate left dst arithmetically	*	*	*	*
RLC	RLC.B	dst	Rotate left dst through Carry	*	*	*	*
SBC	SBC.B	dst	Subtract Carry from dst	*	*	*	*
SETC			Set Carry Bit	—	—	—	1
SETN			Set Negative Bit	—	1	—	—
SETZ			Set Zero Bit	—	—	1	—
TST	TST.B	dst	Test dst	0	*	*	1

8.5 Benefits

The specification for the architecture of the MSP430 CPU contains the following requirements in order of importance:

- 1) High processing speed
- 2) Small CPU area on-chip
- 3) High ROM efficiency
- 4) Easy software development
- 5) Usable into the future
- 6) High flexibility
- 7) Usable for modern programming techniques

The following shows the finding of the optimum architecture out of the previous list of priorities. Several of the listed solutions affect more than one item of the list of priorities; these are shown at the item where they have the biggest impact.

8.5.1 High Processing Speed

To increase the processing speed to a multiple of the speed of 4-bit or 8-bit microcomputers, software and hardware related attributes were chosen.

Hardware related attributes

- Using 16-bit words, the analog-to-digital converter result can be processed immediately. Two operands (source and destination) are possible in one instruction.
- No microcoding: every instruction is decoded separately and allows one-cycle instructions. This is the case for register-to-register addressing, the normal addressing mode used for time critical software parts.
- Interrupt capability for anyone of the 8 I/O-Ports: The periodical polling of the inputs is not necessary.
- Vectored interrupts: This allows the fastest reaction to interrupts.

Software related attributes

- Implementation of the constant generator: The six most often used constants (-1, 0, 1, 2, 4, 8) are retrieved from the CPU with the highest possible speed.
- High register count: Twelve commonly usable registers allow the storage of all time critical values to achieve the fastest possible access.

8.5.2 Small CPU Area

To get low overall cost for the MSP430, the smallest CPU without limiting its processing capability was achieved:

- Use of a RISC structure: With few but strong instructions, any algorithm can be processed. Together with the constant generator, all commonly used instructions, not contained in the implemented instructions, are executable.
- Use of 100% orthogonality: Every instruction inside one of the three instruction formats is completely similar to the other ones. This results in a strongly simplified CPU.
- Only three instruction formats: Restriction to dual operand instructions, single operand instructions, and conditional jumps.

8.5.3 High ROM Efficiency

To solve a given task with a small ROM, the following steps were taken:

- Implementation of seven addressing modes: The possibility to select out of seven addressing modes for the source and out of four addressing modes for the destination allows direct access to all operands without any intermediate operations necessary.
- Placing of PC, SP, and SR inside of the register set: The possibility to address these as registers saves ROM space.
- Wide reach of the conditional jumps: Due to the eightfold jump distance of the MSP430 compared to other microcomputers, in most cases a branch instruction, that normally needs two words, is not necessary.
- Use of a byte/word structure: ROM and RAM are addressable both as bytes and as words. This allows the selection of the most favorable format.

8.5.4 Easy Software Development

Nearly all of the previously mentioned attributes of the architecture ease the development of software for the MSP430.

8.5.5 Usability on Into the Future

The chosen *von-Neumann*-architecture allows a simple system expansion far beyond the currently addressable 64K bytes. If necessary, memory expansion up to 16M bytes is possible.

8.5.6 Flexibility of the Architecture

To ensure that all intended peripheral modules, including currently unknown ones, can be connected easily to the MSP430 system. The following definitions were made:

- ❑ Placing of the peripheral control registers into the memory space (memory mapped I/O). The use of the normal instructions for the peripheral modules makes special peripheral instructions superfluous.
- ❑ All of the control registers and data registers of the peripheral modules can be read and written to.

8.5.7 Usable for Modern Programming Techniques

Programming techniques like *position independent coding* (PIC), *reentrant coding*, *recursive coding*, or the use of high-level languages like C force the implementation of a stack pointer. The system SP is therefore implemented as a CPU register.

8.6 Conclusion

This section demonstrates that the instruction set and the architecture of the MSP430 are easy to understand and that it is easy too to write software for the MSP430. Everyone who has written large program parts with the MSP430 assembler language has an antipathy to adapt again to the more or less unstructured architectures of the other 4-bit, 8-bit, and 16-bit microcomputers.