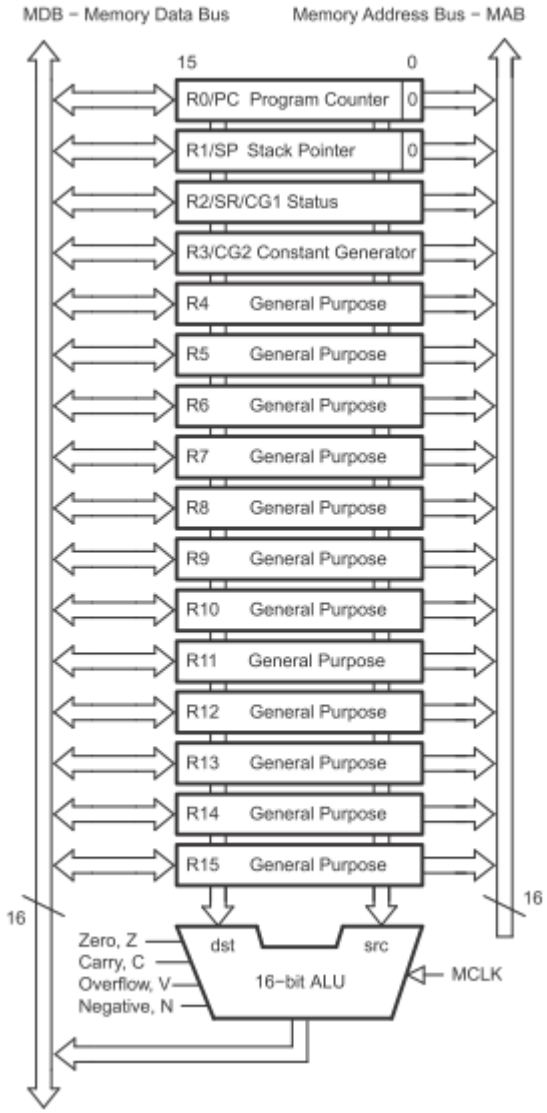
MSP 430 Architecture

Properties of MSP430G2553:

- 16-bit RISC CPU,
- 16 16-bit registers,
- Instructions processing on either bits, bytes or words,
- 51 instructions (27 core, 24 emulated),
- 7 addressing modes,
- Extensive vectored-interrupt capability.

CPU and Registers



The MSP430 CPU has a 16-bit RISC architecture. All operations, other than program-flow instructions, are performed as register operations in conjunction with seven addressing modes for source operand and four addressing modes for destination operand.

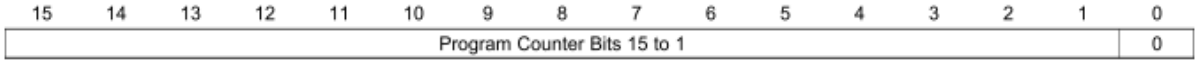
The CPU is integrated with 16 registers that provide reduced instruction execution time. The register-to-register operation execution time is one cycle of the CPU clock.

Four of the registers, **R0 to R3**, are dedicated as **program counter**, **stack pointer**, **status register**, and **constant generator**, respectively. **The remaining registers are general-purpose registers**.

The instruction set consists of the original 51 instructions with three formats and seven address modes and additional instructions for the expanded address range. **Each instruction can operate on word and byte data.**

Program Counter (PC)

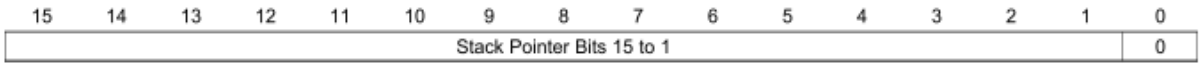
The 16-bit program counter (PC/R0) **points to the next instruction to be executed**. **Each instruction uses an even number of bytes (two, four, or six), and the PC is incremented accordingly**. **PC is aligned to even addresses**.




```
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
```

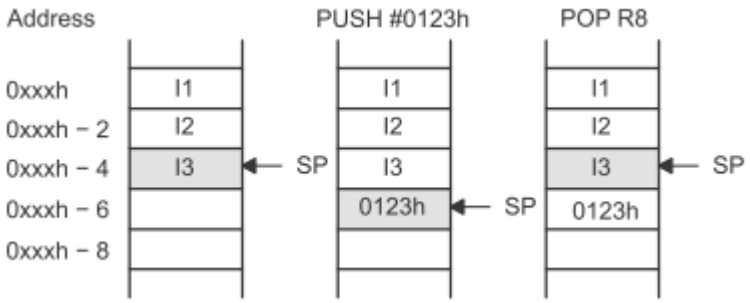
Stack Pointer (SP)

The stack pointer (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. It uses a predecrement, postincrement scheme. In addition, the SP can be used by software with all instructions and addressing modes. Figure below shows the SP. The SP is initialized into RAM by the user, and is aligned to even addresses.




```
MOV 2(SP),R6 ; Item I2 -> R6
MOV R7,0(SP) ; Overwrite top of the stack with R7
PUSH #0123h ; Put 0123h onto top of the stack
POP R8 ; R8 = 0123h
```

Figure below shows stack usage.



Status Register (SR)

The status register (SR/R2), used as a source or destination register, can be used in the register mode only addressed with word instructions. The remaining combinations of addressing modes are used to support the constant generator. Figure below shows the SR bits.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C
rw-0							rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

Description of the bits in SR are shown below:

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. When set, turns off the SMCLK.	
SCG0	System clock generator 0. When set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
OSCOFF	Oscillator Off. When set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK.	
CPUOFF	CPU off. When set, turns off the CPU.	
GIE	General interrupt enable. When set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. 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. Set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. Set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

General Purpose Registers R4 to R15

The twelve registers, R4-R15, are general-purpose registers. All of these registers can be used as data registers, address pointers, or index values and can be accessed with byte or word instructions.

Register byte operations are shown on the next page.

Example Register-Byte Operation

R5 = 0A28Fh
R6 = 0203h
Mem(0203h) = 012h

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

```
  08Fh
+ 012h
 0A1h
```

Mem (0203h) = 0A1h
C = 0, Z = 0, N = 1

(Low byte of register)
+ (Addressed byte)
->(Addressed byte)

Example Byte-Register Operation

R5 = 01202h
R6 = 0223h
Mem(0223h) = 05Fh

```
ADD.B     @R6,R5
```

```
  05Fh
+ 002h
 00061h
```

R5 = 00061h
C = 0, Z = 0, N = 0

(Addressed byte)
+ (Low byte of register)
->(Low byte of register, zero to High byte)

Addressing Modes

Seven addressing modes for the source operand and four addressing modes for the destination operand can address the complete address space with no exceptions. The bit numbers in figure below describe the contents of the As and S (source) and Ad and D (destination) mode bits.

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.

ADDRESS MODE	S	D	SYNTAX	EXAMPLE	OPERATION
Register	✓	✓	MOV Rs,Rd	MOV R10,R11	R10 -- --> R11
Indexed	✓	✓	MOV X(Rn),Y(Rm)	MOV 2(R5),6(R6)	M(2+R5) -- --> M(6+R6)
Symbolic (PC relative)	✓	✓	MOV EDE,TONI		M(EDE) -- --> M(TONI)
Absolute	✓	✓	MOV &MEM,&TCDAT		M(MEM) -- --> M(TCDAT)
Indirect	✓		MOV @Rn,Y(Rm)	MOV @R10,Tab(R6)	M(R10) -- --> M(Tab+R6)
Indirect autoincrement	✓		MOV @Rn+,Rm	MOV @R10+,R11	M(R10) -- --> R11 R10 + 2-- --> R10
Immediate	✓		MOV #X,TONI	MOV #45,TONI	#45 -- --> M(TONI)

Instruction Set

The complete 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:

• Dual-operand

• Single-operand

• Jump

Three types of instruction formats are shown below:

INSTRUCTION FORMAT	EXAMPLE	OPERATION
Dual operands, source-destination	ADD R4,R5	R4 + R5 ---> R5
Single operands, destination only	CALL R8	PC -->(TOS), R8--> PC
Relative jump, un/conditional	JNE	Jump-on-equal bit = 0

All single-operand and dual-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
dst	The destination operand defined by Ad and D-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)
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

Addressing modes are described above where S stands for Source and D stands for Destination.

Destination addresses are valid anywhere in the memory map. However, when using an instruction that modifies the contents of the destination, **the user must ensure the destination address is writable**.

Double Operand 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

Single Operand 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

All addressing modes are possible for the CALL instruction. If the symbolic mode (ADDRESS), the immediate mode (#N), the absolute mode (&EDE) or the indexed mode x(RN) is used, the word that follows contains the address information.

Jumps

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

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:

$$PC_{new} = PC_{old} + 2 + PC_{offset} \times 2$$

Memory Organization

		MSP430G2153	MSP430G2253 MSP430G2213	MSP430G2353 MSP430G2313	MSP430G2453 MSP430G2413	MSP430G2553 MSP430G2513
Memory	Size	1kB	2kB	4kB	8kB	16kB
Main: interrupt vector	Flash	0xFFFF to 0xFFC0	0xFFFF to 0xFFC0	0xFFFF to 0xFFC0	0xFFFF to 0xFFC0	0xFFFF to 0xFFC0
Main: code memory	Flash	0xFFFF to 0xFC00	0xFFFF to 0xF800	0xFFFF to 0xF000	0xFFFF to 0xE000	0xFFFF to 0xC000
Information memory	Size	256 Byte	256 Byte	256 Byte	256 Byte	256 Byte
	Flash	010FFh to 01000h	010FFh to 01000h	010FFh to 01000h	010FFh to 01000h	010FFh to 01000h
RAM	Size	256 Byte	256 Byte	256 Byte	512 Byte	512 Byte
		0x02FF to 0x0200	0x02FF to 0x0200	0x02FF to 0x0200	0x03FF to 0x0200	0x03FF to 0x0200
Peripherals	16-bit	01FFh to 0100h	01FFh to 0100h	01FFh to 0100h	01FFh to 0100h	01FFh to 0100h
	8-bit	0FFh to 010h	0FFh to 010h	0FFh to 010h	0FFh to 010h	0FFh to 010h
	8-bit SFR	0Fh to 00h	0Fh to 00h	0Fh to 00h	0Fh to 00h	0Fh to 00h

As shown in the figure above, MSP430G2553 has 16KB of memory, where:

- Interrupt vectors are located between 0xFFFF to 0xFFC0,
- Code memory is located between 0xFFFF to 0xC000,
- Information memory between 0x10FF to 0x1000,
- RAM is between 0x03FF to 0x0200,
- Rest is for peripherals.

General Purpose Input/Output (I/O)

There are two digital input/output ports, namely P1 and P2. The Input/Output ports can be configured as interruptible or non-interruptible. Each interrupt on these I/O lines can be individually configured to provide an interrupt on a rising edge or falling edge of an input signal. All I/O lines with interrupt capacity use a single interrupt vector. Additionally, the port pins can be individually configured for general-purpose use, or as special function I/Os, such as USARTs, comparator signals and ADCs.

Independent of the I/O port type**, the configuration of the port operation is defined in software using the following registers**:

- Direction Registers (**PxDIR**),
- Input Registers (**PxIN**),
- Output Registers (**PxOUT**),
- Pull-up/Pull-down Resistor Enable Registers (**PxREN**),
- Function Select Registers: (**PxSEL**) and (**PxSEL2**).

Direction Registers (PxDIR)

Read/write 8-bit registers. Selects the direction of the corresponding I/O pin, regardless of the selected function of the pin (general purpose I/O or as a special function I/O).

PxDIR configuration:

- Bit = 1: The port pin is set up as an output;
- Bit = 0: the port pin is set up as an input.

Input Registers (PxIN)

Each bit of these **read-only** registers reflects the input signal at the corresponding I/O pin (pin configured as general purpose I/O);

PxIN configuration:

- Bit = 1: The input is high;
- Bit = 0: The input is low;

Avoid writing to these read-only registers because it will result in increased current consumption.

Output Registers (PxOUT)

The output registers are **read-write**. Each bit of these registers reflects the value written to the corresponding output pin.

PxOUT configuration:

- Bit = 1: The output is high;
- Bit = 0: The output is low.

Pull-up/Pull-down Resistor Enable Registers (PxREN)

Each bit of this register enables or disables the pullup/pulldown resistor of the corresponding I/O pin.

PxREN configuration:

- Bit = 1: Pullup/pulldown resistor enabled;
- Bit = 0: Pullup/pulldown resistor disabled.

Function Select Registers: (PxSEL) and (PxSEL2)

Some port pins are multiplexed with other peripheral module functions (see the device-specific datasheet);

The bits: (PxSEL) and (PxSEL2 – 2xx family), are used to select the pin function: I/O general-purpose port or peripheral module function.

PxSEL	PxSEL2	Pin Function
0	0	Selects general-purpose I/O function
0	1	Selects the primary peripheral module function
1	0	Reserved (See device-specific data sheet)
1	1	Selects the secondary peripheral module function

Interruptible ports (P1 and P2)

Each pin of ports P1 and P2 is able to generate an interrupt request (pin is interruptible) and is configured using the PxIFG, PxIE, and PxIES registers. The port makes use of all the same configuration registers as non-interruptible ports (as described above), but with three additional registers:

Interrupt Enable (PxIE)

Read-write register to enable interrupts on individual pins. PxIE configuration:

- Bit = 1: The interrupt is enabled;
- Bit = 0: The interrupt is disabled.

Each PxIE bit enables the interrupt request associated with the corresponding PxIFG interrupt flag. Writing to PxOUT and/or PxDIR can result in setting PxIFG.

Interrupt Edge Select Registers (PxIES)

This read-write register selects the transition on which an interrupt occurs for the corresponding I/O pin (if PxIE and GIE are set). PxIES configuration:

- Bit = 1: Interrupt flag is set on a high-to-low transition;
- Bit = 0: Interrupt flag is set on a low-to-high transition.

Interrupt Flag Registers (PxIFG)

The bit of this read-write register is set automatically when the programmed signal transition (edge) occurs on the corresponding I/O pin, provided that the corresponding PxIE bit and the GIE bit are set;

Each PxIFG flag can be set by software, enabling an interrupt generated by software; Each PxIFG flag must be reset with software;

PxIFG configuration:

- Bit = 0: No interrupt is pending;
- Bit = 1: An interrupt is pending.

Assembly Language Programming Characteristics

Hexadecimal Notation

"x" is often used in hexadecimal number notations. It shows that a number is in hexadecimal notation. Below a conversion table is shown with corresponding decimal, binary and hexadecimal numbers.

Decimal	Binary	Hexadecimal
00	0000	0x00
01	0001	0x01
02	0010	0x02
03	0011	0x03
04	0100	0x04
05	0101	0x05
06	0110	0x06
07	0111	0x07
08	1000	0x08
09	1001	0x09
10	1010	0x0A
11	1011	0x0B
12	1100	0x0C
13	1101	0x0D
14	1110	0x0E
15	1111	0x0F
16	1 0000	0x10

Figure 1 Hexadecimal Notation (ref. TI MSP430 documents)

Variable Declaration Rules

Variables must be at most 31 characters long.

Only letters, numbers or the character "_" is allowed.

Masks

Masks are used to set (make 1) certain bits in a variable, or to clear certain bits (make 0).

Examples:

```
P5OUT = 0x80;           // P5OUT= 1000 0000
P5OUT |= 0x04           // P5OUT= XXXX X1XX
P5OUT &= ~0x08          // P5OUT= XXXX 0XXX
```

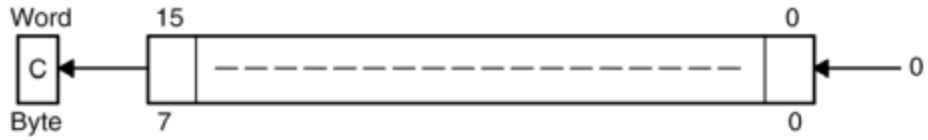
Bit shifts

The bit shifts operation consists of the movement of digits, or shift, to the left or right, to modify the binary representation of an integer value. Registers in the processor have a fixed number of bits available for storing numbers, so some bits will be "shifted out" of the register at one end, while the same number of bits will be "shifted in" from the other end; the differences between bit shift operations are how the values of the bits shifted in are calculated.

Left-shift (with carry)

In the case of an arithmetic shift, the bits that are shifted out of either end are discarded (not in assembler - there it would placed in Carry bit). In an arithmetic shift left, zeros are shifted in on the right.

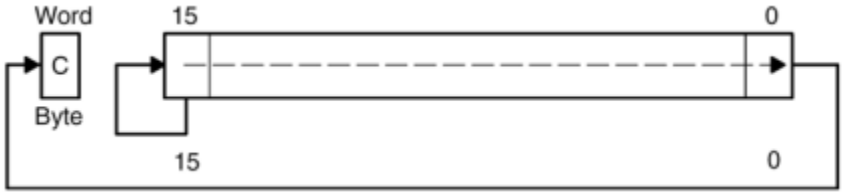
Example : Arithmetic Left shift value 0011 1001 in an 8-bit register.



A shift of one to the right is a quick way to multiply by two.

Right-shift (with carry)

In the case of an arithmetic shift right, the sign bit is shifted in on the left, thus preserving the sign of the operand.



This is a quick way to perform a divide by two.

Bit modification

The state of one or more bits of a value can be changed by the bit clear (BIC) and bit set (BIS) instructions, as described below.

The BIC instruction clears one or more bits of the destination operand. This is carried out by inverting the source value then performing a logical & (AND) operation with the destination. Therefore, if any bit of the source is one, then the corresponding bit of the destination will be cleared to zero.

BIC source,destination or BIC.W source,destination

BIC.B source,destination

Consider the instruction:

```
BIC   #0x000C,R5
```

This clears bits 2 and 3 of register R5 to zero, leaving the remaining bits unchanged.

There is also the bit set (BIS) instruction:

BIS source,destination or BIS.W source,destination

BIS.B source,destination

This sets one or more bits of the destination using a similar procedure to the previous instruction. The instruction performs a logical | (OR) between the contents of the source and the destination.

For example, the instruction:

```
BIS   #0x000C,R5
```

Sets bits 2 and 3 of register R5, leaving the remaining bits unchanged.

CPU status bits modification

The CPU contains a set of flags in the Status Register (SR) that reflect the status of the CPU operation, for example that the previous instruction has produced a carry (C) or an overflow (V). It is also possible to change the status of these flags directly through the execution of emulated instructions, which use the BIC and BIS instructions described above.

```
CLRC; clears carry flag (C). Emulated by BIC   #1,SR
CLRN; clears negative flag (N). Emulated by BIC   #4,SR
CLRZ; clears the zero flag (Z). Emulated by BIC   #2,SR
SETC; set the carry flag (C). Emulated by BIS   #1,SR
SETN; set the negative flag (N). Emulated by BIS   #4,SR
SETZ; set the zero flag (Z). Emulated by BIS   #2,SR
```

Enable/disable interrupts

Two other instructions allow the flag that enables or disables the interrupts to be changed. The global interrupt enable GIE flag of the register SR may be set to disable interrupts:

```
DINT; Disable interrupts. (emulated by BIC   #8,SR)
```

or it may be cleared to enable interrupts:

```
EINT; Enable interrupts. (emulated by BIS   #8,SR)
```

Sign extension

The CPU supports 8-bit and 16-bit operations. Therefore, the CPU needs to be able to extend the sign of a 8-bit value to a 16-bit format. The extension of the sign bit is produced by the instruction:

SXT destination

For example, the sign extension of the value contained in R5 is:

```
MOV.B #0xFC,R5  ; place the value 0xFC in R5
SXT   R5    ; sign extend word. R5 = 0xFFFC
```

Program flow control

Bit testing

Bit testing can be used to control program flow. Hence, the CPU provides an instruction to test the state of individual or multiple bits. The operation performs a logical & (AND) logical operation between the source and destination operands. The result is ignored and none of the operands are changed. This task is performed by the following instruction:

BIT source,destination or BIT.W source,destination

BIT.B source,destination

As a result of the operation, the CPU state flags are updated:

V: reset;

N: set if the MSB of the result is 1, otherwise reset;

Z: set if the result is zero, otherwise reset;

C: set if the result is not zero, otherwise reset.

For example, to test if bit R5.7 is set:

```
MOV   #0x00CC,R5  ; load the value #0x00CC to R5
BIT   #0x0080,R5  ; test R5.7
```

The result of the logical AND operation with 0x0080, the bit R5.7 is tested. Reasoning this case, the result modifies the flags (V = 0, N = 0, Z = 0, C = 1).

Comparison with zero

Another operation typically used in a program is the comparison of a value with zero, to determine if the value has reached zero. This operation is performed using the following emulated instruction:

TST source,destination or TST.W source,destination

TST.B source,destination

As a result of the operation, the CPU state flags are updated:

V: reset;

N: set if the MSB of the result is 1, otherwise reset;

Z: set if the result is zero, otherwise reset;

C: set.

For example, to test if register R5 is zero:

```
MOV   #0x00CC,R5  ; move the value #0x00CC to R5
TST   R5    ; test R5
```

The comparison of the register R5 with #0x0000 modifies the flags (V = 0, N = 0, Z = 0, C = 1).

Value comparison

Two operands can be compared using the instruction:

CMP source,destination or CMP.W source,destination

CMP.B source,destination

The comparison result modifies the CPU status flags:

V: set if an overflow occurs;

N: set if the result is negative, otherwise reset (source >= destination);

Z: set if the result is zero, otherwise reset (source = destination);

C: set if there is a carry, otherwise reset (source <= destination).

In the following example, the contents of register R5 are compared with the contents of register R4:

```
MOV   #0x0012,R5  ; move the value 0x0012 to R5
MOV   #0x0014,R4  ; move the value 0x0014 to R4
CMP   R4,R5
```

The register comparison modifies the CPU status flags (V = 0, N = 1, Z = 0, C = 0).

Program flow branch

A branch in program flow without any constraint is performed by the instruction:

BR destination

This instruction execution is only able to reach addresses in the address space below 64 kB.

Each of the addressing modes can be used. For example:

```
BR  #EXEC   ;Branch to label EXEC or direct branch (e.g. #0A4h),
BR   EXEC   ; Branch to the address contained in EXEC ,
BR  &EXEC    ; Branch to the address contained in  absolute address EXEC,
BR  R5       ; Branch to the address contained in R5
BR   @R5    ; Branch to the address contained in the word pointed to by R5.
BR   @R5+   ; Branch to the address contained in  the word pointed to by R5
and  increment pointer in R5 afterwards.
BR   X(R5)  ; Branch to the address contained in  the address pointed to by
R5   + X  (e.g. table with address starting at  X). X can be an address or
a label
```

In addition to the previous instructions it is possible to jump to a destination in the range +512 to -511 words using the instruction:

JMP destination

Conditional jump

Action can be taken depending on the values of the CPU status flags. Using the result of a previous operation, it is possible to produce jumps in the program flow execution. The new memory address must be in the range +512 to -511 words.

The following instructions are available for conditional jumps:

Jump if equal (Z = 1):

JEQ destination or *JZ destination*

```
label1:
     MOV   0x0100,R5
     MOV   0x0100,R4
     CMP   R4,R5
     JEQ   label1
```

The example on the left compares the register R4 and R5 contents. As they are equal, the flag Z is set, and therefore, the jump to position label1 is executed.

Jump if different (Z = 0):

JNE destination or *JNZ destination*

```
label2:
     MOV   #0x0100,R5
     MOV   #0x0100,R4
     CMP   R4,R5
     JNZ   label2
```

The example on the left compares the contents of registers R4 and R5. As they are equal, the flag Z is set, and therefore, the jump to position label2 is not executed.

Jump if higher or equal (C = 1) – without sign:

JC destination or *JHS destination*

```
label3:
     MOV   #0x0100,R5

     BIT   #0x0100,R5
     JC    label3
```

The example on the left tests the state of bit R5.8. As this bit is set, the flag C is set, and therefore, the jump to position label3 is executed.

Jump if lower (C = 0) – without sign:

JNC destination or *JLO destination*

```
label4:
     MOV   #0x0100,R5
     BIT   #0x0100,R5
     JNC   label4
```

The example on the left tests the state of bit R5.8. As it is set, the flag C is set, and therefore, the jump to position label4 is not executed.

Jump if higher or equal (N = 0 and V = 0) or (N = 1 and V = 1) – with sign:

JGE destination

```
label5:
     MOV   #0x0100,R5

     CMP   #0x0100,R5
     JGE   label5
```

The example on the left compares the contents of register R5 with the constant #0x0100. As they are equal, both flags N and V are reset, and therefore, the jump to the address label5 is executed.

Jump if lower (N = 1 and V = 0) or (N = 0 and V = 1) – with sign:

JL destination

```
label6:
     MOV   #0x0100,R5

     CMP   #0x0100,R5
     JL    label6
```

The example on the left compares the contents of register R5 with the constant #0x0100. As they are equal, both flags N and V are reset, and therefore, the jump to the address label6 is not xecuted.

Routines

During the development of an application, repeated tasks can be identified and separated out into routines. This piece of code can then be executed whenever necessary. It can substantially reduce the code size.

Furthermore, the use of routines allows structuring the application. It also helps code debugging and facilitates understanding of the operation.

Invoking a routine

A routine is identified by a label in assembly language. A routine call is made at the point in the program where execution must be changed to perform a task. When the task is complete, it is necessary to return to the point just after where the routine call was called.

Two different instructions are available to perform the routine call, namely CALL and CALLA.

The following instruction can be used if the routine is located in the address below 64 kB:

CALL destination

This instruction decrements the register SP by two, to store the return address. The register PC is then loaded with the routine address and the routine executed. The *CALL* instruction can be used with any of the addressing modes. The return is performed by the instruction *RET*.

Routine return

The routine execution return depends on the call type that was used. If the routine is called using a instruction CALL, the following instruction must be used to return:

RET

This instruction extracts the value pointed to by the register SP and places it in the PC.

Passing parameters to the routine

There are two different ways to move data to a routine.

The first makes use of a register. The data values needed for the routine execution are placed in pre-defined registers before the routine call. The return from the routine execution can use a similar method.

The second method makes use of the stack. The parameters necessary to execute the routine are placed on the stack using the PUSH instructions.

The routine can use any of the methods already discussed to access the information.

To return from the routine, the stack can again be used to pass the parameters, using a POP instruction.

Care is needed in the use of this method to avoid stack overflow problems. Generally, the stack pointer must set back to the value just before pushing parameters after execution of the routine.

Example for the first method:

```
;--------------------
; Routine
;--------------------
adder:
 ADD   R4,R5
 RET               ; return from routine
;--------------------
; Main
;--------------------
 MOV   &var1,R4    ; parameter var1 in R4
 MOV   &var2,R5    ; parameter var2 in R5

 CALL  #adder      ; call routine adder
 MOV   R5,&var3    ; result R5 in var3
```

Key assembly directives

The MPS430 source code programs are a sequence of statements that have:

- Assembly directives;
- Assembly instructions;
- Macros, and;
- Comments.

A line can have four fields (label, mnemonic, operand list, and comment). The general syntax is:

[label[:]] mnemonic [operand list] [;comment]

Some line examples are:

```
            .sect       ".sysmem"         ; Data Space
var1        .word   2                     ; variable var1 declaration
            .text                         ; Program Space
Label1:     MOV.W R4,R5                   ; move R4 contents to R5
```

The general guidelines for writing the code are:

- All statements must begin with a label, a blank, an asterisk, or a semicolon;
- Labels are optional and if used, they must begin in column 1;
- One or more blanks must separate each field. Tab and space characters are blanks. You must separate the operand list from the preceding field with a blank;
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column must begin with a semicolon;
- A mnemonic cannot begin in column 1, as it will be interpreted as a label.

Assembler directives supply data to the program and control the assembly process. Assembler directives enable you to do the following:

- Assemble code and data into specified sections;
- Reserve space in memory for uninitialized variables;
- Control the appearance of listings;
- Initialize memory;
- Assemble conditional blocks;
- Define global variables;
- Specify libraries from which the assembler can obtain macros;
- Examine symbolic debugging information.

Directives which initialize sections are:

Mnemonic and Syntax	Description
.bss *symbol, size in bytes*[, *alignment*]	Reserves *size* bytes in the .bss (uninitialized data) section
.data	Assembles into the .data (initialized data) section
.sect " *section name*"	Assembles into a named (initialized) section
.text	Assembles into the .text (executable code) section
symbol **.usect "***section name*", *size in bytes* [, *alignment*]	Reserves *size* bytes in a named (uninitialized) section

Directives which initialize constants (data and memory) are:

Mnemonic and Syntax	Description
.byte $value_1$[, ..., $value_n$]	Initializes one or more successive bytes in the current section
.char $value_1$[, ..., $value_n$]	Initializes one or more successive bytes in the current section
.double $value_1$[, ..., $value_n$]	Initializes one or more 32-bit, IEEE double-precision, floating-point constants
.field *value*[, *size*]	Initializes a field of *size* bits (1-32) with *value*
.float $value_1$[, ..., $value_n$]	Initializes one or more 32-bit, IEEE single-precision, floating-point constants
.half $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers (halfword)
.int $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers
.long $value_1$[, ... , $value_n$]	Initializes one or more 32-bit integers
.short $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers (halfword)
.string {$expr_1$|"$string_1$"}[,... , {$expr_n$|"$string_n$"}]	Initializes one or more text strings
.word $value_1$[, ... , $value_n$]	Initializes one or more 16-bit integers

Directives that perform alignment and reserve space.are:

Mnemonic and Syntax	Description
.align [*size in bytes*]	Aligns the SPC on a boundary specified by *size in bytes*, which must be a power of 2; defaults to word (2 byte) boundary
.bes *size*	Reserves *size* bytes in the current section; a label points to the end of the reserved space
.space *size*	Reserves *size* bytes in the current section; a label points to the beginning of the reserved space