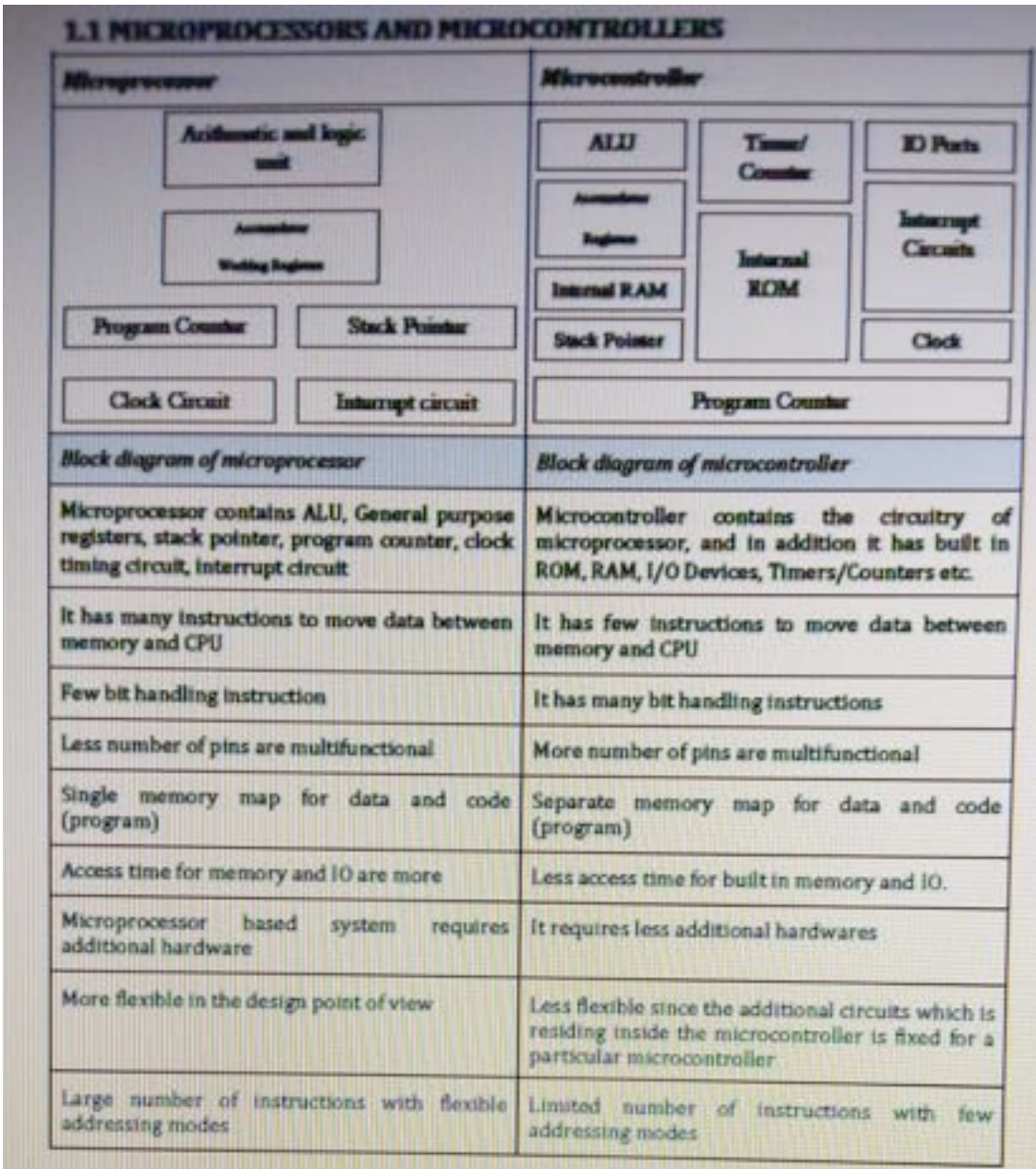


Solutions of Internal Assessment Test – I

Sub:	MSP430 Microcontroller	Code:	15EC555
Date:	08 / 09 / 2018	Duration:	90 mins
		Max Marks:	50
		Sem:	V
		Branch:	ECE
Answer Any FIVE FULL Questions			

	Marks	OBE	
		CO	RBT
1 (a) Differentiate between a Microprocessor and a Microcontroller. Which are the different peripherals that would be available in a Microcontroller?	[05]	CO1	L2
			
(b) Differentiate between Harvard and Von Neumann architecture.	[05]	CO1	L2

Von-Neumann (Princeton architecture)	Harvard architecture
<b>Von-Neumann (Princeton architecture)</b>	<b>Harvard architecture</b>
It uses single memory space for both instructions and data.	It has separate program memory and data memory
It is not possible to fetch instruction code and data	Instruction code and data can be fetched simultaneously
Execution of instruction takes more machine cycle	Execution of instruction takes less machine cycle
Uses CISC architecture	Uses RISC architecture
Instruction pre-fetching is a main feature	Instruction parallelism is a main feature
Also known as control flow or control driven computers	Also known as data flow or data driven computers
Simplifies the chip design because of single memory space	Chip design is complex due to separate memory space
Eg. 8085, 8086, MC6800	Eg. General purpose microcontrollers, special DSP chips etc.

2. Sketch the functional block diagram of MSP430 microcontroller and briefly explain its architecture.

[10]

CO1

L2

## 2.2 The Inside View—Functional Block Diagram

Figure 2.2 shows a block diagram of the F2013. These are its main features:

- On the left is the CPU and its supporting hardware, including the clock generator. The emulation, JTAG interface and Spy-Bi-Wire are used to communicate with a desktop computer when downloading a program and for debugging.

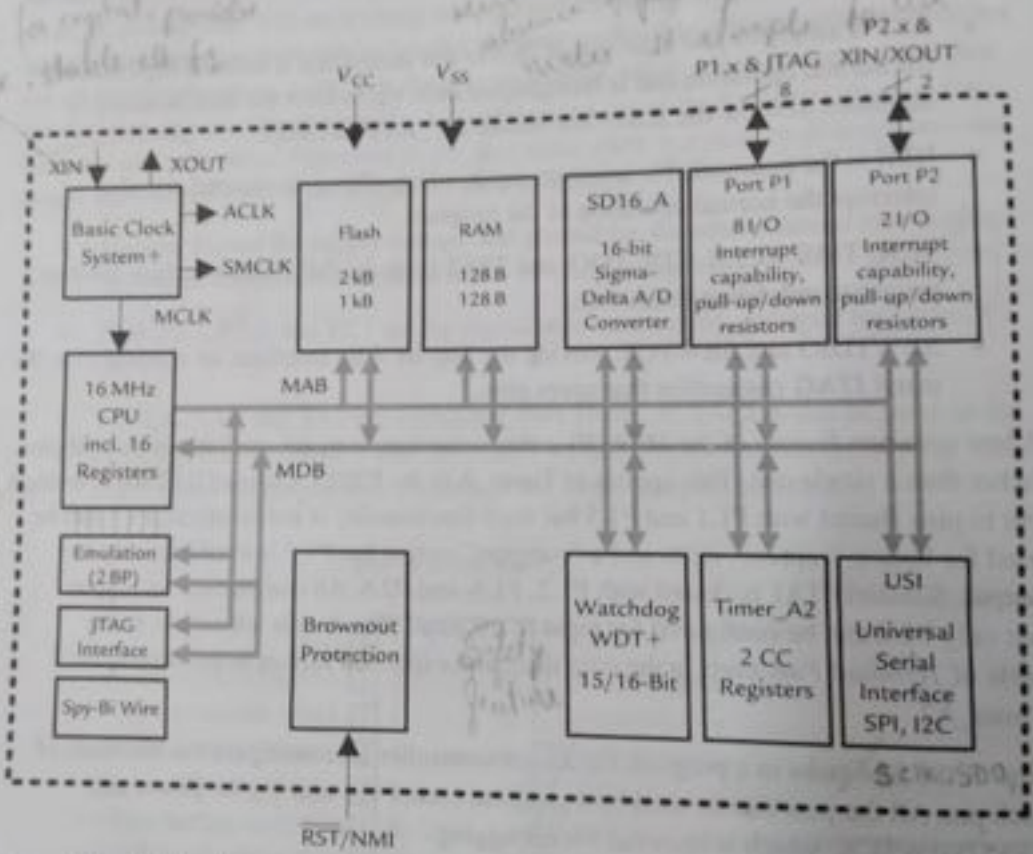


Figure 2.2: Block diagram of the MSP430F2003 and F2013, taken from the data sheet.

- The main blocks are linked by the *memory address bus (MAB)* and *memory data bus (MDB)*.
- These devices have flash memory, 1 KB in the F2003 or 2 KB in the F2013, and 128 bytes of RAM. *→ starts from 0x0200*
- Six blocks are shown for peripheral functions (there are many more in larger devices). All MSP430s include input/output ports, Timer\_A, and a *watchdog timer*, although the details differ. The universal serial interface (USI) and *sigma-delta* analog-to-digital converter (SD16\_A) are particular features of this device.
- The *brownout protection* comes into action if the supply voltage drops to a dangerous level. Most devices include this but not some of the MSP430xxx family.
- There are ground and power supply connections. Ground is labeled  $V_{SS}$  and is taken to define 0 V. The supply connection is  $V_{CC}$ . For many years, the standard for logic was  $V_{CC} = +5\text{ V}$  but most devices now work from lower voltages and a range of 1.8–3.6 V is specified for the F2013. The performance of the device depends on  $V_{CC}$ . For example, it is unable to program the flash memory if  $V_{CC} < 2.2\text{ V}$  and the maximum clock frequency of 16 MHz is available only if  $V_{CC} \geq 3.3\text{ V}$ .

TI uses a quaint notation for the power connections. The *S* stands for the source of a field-effect transistor, while the *C* stands for the collector of a bipolar junction transistor, quite different device. The MSP430, like most modern integrated circuits, is built using complementary metal-oxide-silicon (CMOS) technology and field-effect transistors. I doubt if it contains any bipolar junction transistors except possibly in some of the analog peripherals.

There is only one pair of address and data buses, as expected with a von Neumann architecture. Some addresses must therefore point to RAM and some to flash, so it is a good idea to explore the memory map next.

3 (a) What are the features of MSP430 which makes it suitable for low power & portable applications?

[05]

CO1	L2
-----	----

## MSP430

- Introduced in the late 1990s by Texas Instr
- 16 bit Microcontroller
- Von-neumann architecture.
- Low power applications.
- Reduced Instruction Set Computer (RISC)
- Address & data buses are 16 bits wide
- It can address only  $2^{16} = 64$  KB of memory
- Programmed in C
- 16 registers in its CPU
- Powerful siblings include the TMS470, which is based on the 32/16 bit ARM7.
- C2000 - Digital signal processor.

Features make the MSP430 suitable for low power & portable applications:

- CPU is small & efficient, with large no of registers.
- No need of special instructions to put the device into a low-power mode. The mode is controlled by bits in the static registers. The MSP430 is awakened by an interrupt & returns automatically to its low-power mode after handling

the interrupt

- There are many low-power modes, depending on how much of the device should remain active & how quickly it should return to full-speed operation.
- Wide choice of clocks.
  - 32 kHz clock - Wake the device periodically
  - 16 MHz → CPU by DCO (Digitally Controlled Oscillator)
- Wide range of peripherals is available many of which can run autonomously without the CPU for most of the time.
- MSP430 can drive LCDs (Liquid Crystal Display) directly.
- Low Power Consumption.
  - F2013 draws - In active mode 4.5 mA, 3.5 V at 16 MHz
  - 0.2 mA, 1.8 V at 3 MHz
  - Standby mode - 1 μA
- Some MSP430 devices are classed as application specific standard products (ASSPs) & contain specialized analog hardware for various types of measurement.

(b) Show the Memory map of MSP430F2013 and explain it briefly.

[05]

CO1

L1

Most devices have ranges of addresses that are not used, meaning that there are no registers at such addresses. These are shown in gray in Figure 2.4. In fact the largest regions are unused in devices with small memories, including this one.

Here is a brief description of each region:

**Special function registers:** Mostly concerned with enabling functions of some modules and enabling and signalling interrupts from peripherals.

**Peripheral registers with byte access and peripheral registers with word access:** Provide the main communication between the CPU and peripherals. Some must be accessed as words and others as bytes. They are grouped in this way to avoid wasting

Address	Type of memory	
0xFFFF	interrupt and reset	
0xFFC0	vector table	w/B
0xFFBF	flash code memory	
0xF800	(lower boundary varies)	w/B
0xF7FF		
0x1100		
0x10FF	flash	
0x1000	information memory	w/B
0x0FFF	bootstrap loader	
0x0C00	(not in F20xx)	w/B
0x0BFF		
0x0280		
0x027F	RAM	
0x0200	(upper boundary varies)	w/B
0x01FF	peripheral registers	
0x0100	with word access	w/B
0x00FF	peripheral registers	
0x0100 → x + 10	with byte access	byte
0x000F	special function registers	
0x0000	(byte access)	byte

Figure 2.4: Memory map of the MSP430F2013, based on the data sheet and the MSP430x2xx Family User's Guide. Addresses increase up the page and are not drawn to scale. Gray regions are unused and their size varies considerably between devices. The F2013 does not have a bootstrap loader but I have shown its location because it is present in most variants of the MSP430.

addresses. If the bytes and words were mixed, numerous unused bytes would be needed to ensure that the words were correctly aligned on even addresses.

**Random access memory:** Used for variables. This always starts at address 0x0200 and the upper limit depends on the size of the RAM. The F2013 has 128 B.

**Bootstrap loader:** Contains a program to communicate using a standard serial protocol, often with the COM port of a PC. This can be used to program the chip but improvements in other methods of communication have made it less important than in the past, particularly for development. Details are given in the application note *Features of the MSP430 Bootstrap Loader* (slaa089). All MSP430s had a bootstrap loader until the F20xx, from which it was omitted to improve security.

**Information memory:** A 256 B block of flash memory that is intended for storage of nonvolatile data. This might include serial numbers to identify equipment—an address for a network, for instance—or variables that should be retained even when power is removed. For example, a printer might remember the settings from when it was last used and keep a count of the total number of pages printed. The information memory is laid out with smaller segments of flash than the code memory, which makes it more convenient to erase and rewrite. Segment A contains factory calibration data for the DCO in the MSP430F2xx family and is protected by default.

**Code memory:** Holds the program, including the executable code itself and any constant data. The F2013 has 2 KB but the F2003 only 1 KB.

**Interrupt and reset vectors:** Used to handle “exceptions,” when normal operation of the processor is interrupted or when the device is reset. This table was smaller and started at 0xFFE0 in earlier devices.

The range of addresses has been extended from 64 KB to 1 MB in the MSP430X. This means that addresses require 20 bits rather than 16 and the MAB is therefore 4 bits wider. The bottom 64 KB of memory from 0x00000 to 0x0FFFF is laid out in exactly the same way as in the original MSP430. The additional memory, from 0x10000 to 0xFFFFF, is available for additional ROM. This permits larger programs and tables to be stored.

4 (a) Explain the status register of MSP430 microcontroller.

[05]

CO1	L2
-----	----



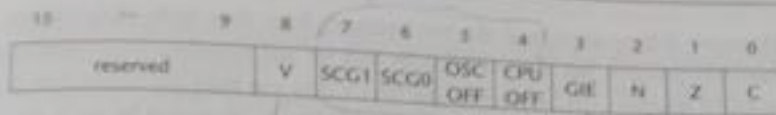


Figure 5.3: Individual bits in the status register.

### 5.1.3 Status Register (SR)

This contains a set of *flags* (single bits) shown in Figure 5.3, whose functions fall into three categories. The reserved bits are not used in the MSP430 but some have been taken up in the MSP430X to extend the length of addresses. The status register is known as the condition code register in some processors.

#### Result of Arithmetic or Logic Operations

The C, Z, N, and V bits are affected by many of the operations performed by the ALU, but not all—see the section “Instruction Set” on page 132 for details:

- The main function of the *carry* bit C is to flag that the result of an arithmetic operation is too large to fit in the space allocated. In other words, an overflow occurred. Here is an example with bytes for simplicity. The hexadecimal sum  $0x75 + 0xC7 = 0x13C$ , where the result is too large to be held in a single byte. The processor would put  $0x3C$  in the destination and set the carry bit to show that the result had overflowed and that a 1 should be carried into the next more significant byte. The carry flag also takes part in rotations and shifts. It is sometimes used as temporary storage to pass a bit from one register to another or to a subroutine.
- The *zero* flag Z is set when the result of an operation is 0. A common application is to check whether two values are equal: They are subtracted and the Z bit is tested to see whether the result is 0, which shows that the values are the same. This is used in Listing 4.6.
- The *negative* flag N is made equal to the msb of the result, which indicates a negative number if the values are signed.
- The *signed overflow* flag V is set when the result of a signed operation has overflowed, even though a carry may not be generated. An example is the sum  $0x75 + 0x67 = 0xDC$ . There are no problems if the variables are all unsigned.

However, if they are signed, the two operands have their msbs clear and are therefore positive, but the result has its msb set and therefore is interpreted as the negative number  $-0x24$ . (Remember that a byte can hold the values 0 to  $0xFF$  if it is unsigned or  $-0x80$  to  $0x7F$  if it is signed.) The V bit would be set in this case to show that an overflow has occurred if the values are signed.

It is rarely necessary to test these flags explicitly because the instruction set contains a set of decisions, such as "jump if greater or equal," which make the appropriate tests. See Maxfield and Brown [37] in "Further Reading" for more background.

**Enable Interrupts**

Setting the *general interrupt enable* or GIE bit enables maskable interrupts, provided that the individual sources of interrupts have themselves been enabled. Clearing the bit disables all maskable interrupts. There are also nonmaskable interrupts, which cannot be disabled with GIE. This is explained in the section "Interrupts" on page 186.

**Control of Low-Power Modes**

The CPUOFF, OSCOFF, SCG0, and SCG1 bits control the mode of operation of the MCU. All systems are fully operational when all bits are clear. Setting combinations of these bits puts the device into one of its low-power modes, which is described in the section "Low-Power Modes of Operation" on page 198.

--	--

- (b) Given  $R5=200H$ ,  $R4=250H$ ,  $[200H]=0ABCDH$ ,  $[202]=5678H$ . Write the output after execution of each instruction.

MOV.W R5,2(R4)

INCD.W R4

MOV.W @R5+, 2(R4)

ADD.W @R5, 2(R4)

[05] CO1 L3

--	--

4b. Given  $R_5 = 200H$ ,  $R_4 = 250H$ ,  $[200H] = 0ABCDH$ ,  
 $[202] = 5678H$ . Write the output after execution of  
each instruction.

$\Rightarrow$  a)  $MOV.W \ R_5, 2(R_4)$

$$R_4 = 250 + 2 = 252$$

Content of  $R_5$  i.e.  $200H$  is copied in  $252$  loc

b)  $INCD.W \ R_4$

Increases the content of  $R_4$  by 2. i.e.

$$R_4 = R_4 + 2 = 250 + 2 = 252H$$

c)  $MOV.W \ @R_5+, 2(R_4)$

Content of  $200H$  ( $R_5$ ) is  $0ABCDH$ . Its moved  
to  $2 + \del{250} = 252$  memory location. then  
 $R_5$  incremented by 2.  $R_5 = 202H$ .

d)  $ADD.W \ @R_5, 2(R_4)$

$$@R_5 = @202H = 5678H$$

$$2(R_4) = 2 + 252 = [252] = 0ABCDH$$

$$\begin{array}{r} \text{ADD.W} \quad 5678 \\ \quad \quad \quad \underline{ABCD} \\ \quad \quad \quad 10245H \end{array}$$

5. Explain all the addressing modes of MSP430 microcontroller with suitable examples.

[10] CO1 L1

## 5.2 Addressing Modes

A key feature of any CPU is its range of addressing modes, the ways in which operands can be specified. The MSP430 has four basic modes for the source but only two for the destination in instructions with two operands. These modes are made more useful by the way in which they interact with the CPU's registers. All 16 of these are treated on an almost equal basis, including the four special-purpose registers R0-R3 or PC, SP, SR/CG1, and CG2. The combination of the basic addressing modes and the registers gives the seven modes listed in the user's guides, although eight could reasonably be claimed.

An instruction itself fits into a single word of 16 bits, although it may be followed by further words to provide addresses or an immediate value. There are three formats of instruction. I shall use TI's standard abbreviations of *src* for source and *dst* for destination. It is important to distinguish between these because the destination has fewer addressing modes.

**Double operand (Format I):** Arithmetic and logical operations with two operands such as `add.w src, dst`, which is the equivalent of `dst += src` in C. Note the different ordering of *src* and *dst* in C and assembly language. Both operands must be specified in the instruction. This contrasts with accumulator-based architectures, where an accumulator or working register is used automatically as the destination and one operand.

**Single operand (Format II):** A mixture of instructions for control or to manipulate a single operand, which is effectively the *source* for the addressing modes. The nomenclature in TI's documents is inconsistent in this respect.

**Jumps:** The jump to the destination rather than its absolute address, in other words the offset that must be added to the program counter.

The "return from interrupt" instruction `reti` is unique in requiring no operands. This would usually be described as *inherent* addressing but TI curiously classifies it as Format II without data. (This follows from its binary opcode.)

We are now in a position to examine the range of addressing modes. The reason for the asymmetry between source and destination is straightforward: Not enough bits are available in an instruction (a 16-bit word). I return to this in the section "Reflections on the CPU and Instruction Set" on page 153.

### 5.2.1 Register Mode

This uses one or two of the registers in the CPU. It is the most straightforward addressing mode and is available for both source and destination. For example,

```
MOV.W R5,R6 ; move (copy) word from R5 to R6
```

The registers are specified in the instruction word; no further data are needed. It is also the fastest mode and this instruction takes only 1 cycle. Any of the 16 registers can be used for either source or destination but there are some special cases:

- The PC is incremented by 2 while the instruction is being fetched, before it is used as a source.
- The constant generator CG2 reads 0 as a source.
- Both PC and SP must be even because they address only words, so the lsb is discarded if they are used as the destination.
- SR can be used as a source and destination in almost the usual way although there are some details about the behavior of individual bits.

For byte instructions,

- Operands are taken from the lower byte; the upper byte is not affected.
- The result is written to the lower byte of the register and the upper byte is cleared.

The upper byte of a register in the CPU cannot be used as a source. If this is needed, the 2 bytes in a word must first be swapped with `swpb`.

### 5.2.2 Indexed Mode

This looks much like an element of an array in C. The address is formed by adding a constant base address to the contents of a CPU register; the value in the register is not

is formatted as  $X(R_n)$

changed. Indexed addressing can be used for both source and destination. For example, suppose that R5 contains the value 4 before this instruction:

```
mov.b 3(R5),R6 ; load byte from address 3+(R5)=7 into R6
```

The address of the source is computed as  $3 + (R5) = 3 + 4 = 7$ . Thus a byte is loaded from address 7 into R6. The value in R5 is unchanged. There is no restriction on the address for a byte but remember that words must lie on even addresses.

Indexed addressing can be used for the source, destination, or both. The base addresses, just the single value 3 here because only one address is indexed, are stored in the words following the instruction. They cannot be produced by the constant generator.

The instruction is not normally used like this with numerical constants. More typically the base is the address of the first element of an array or table and the register holds the index. Thus the indexed address `Message(R5)` is equivalent to the element `Message[i]` of an array of characters in C, assuming that R5 is used for the index and  $R5 = i$ . Look back at Listing 4.15.

However, there is an important difference between C and assembly. The CPU always calculates the indexed address in *bytes*, while C takes account of the size of the object when calculations are performed with pointers. Suppose that `Words[]` is an array of words. In C the two expressions `Word[i]` and `*(Word+i)` are equivalent. The corresponding indexed address would be `Word(R5)` with  $R5 = 2i$  because each word is 2 bytes long.

These examples use general-purpose registers but the full power of indexed addressing is released when the special registers are used as well.

### ***Symbolic Mode (PC Relative)***

In this case the program counter PC is used as the base address, so the constant is the offset to the data from the PC. TI calls this the symbolic mode although it is usually described as PC-relative addressing. It is used by writing the symbol for a memory location without any prefix. For example, suppose that a program uses the variable `LoopCtr`, which occupies a word. The following instruction stores the value of `LoopCtr` in R6 using symbolic mode:

The assembler replaces this by the indexed form

```
mov.w  X(PC),R6      ; load word LoopCtr into R6, symbolic mode
```

where  $X = \text{LoopCtr} - \text{PC}$  is the offset that needs to be added to PC to get the address of LoopCtr. This calculation is performed by the assembler, which also accounts for the automatic incrementing of PC.

This seems a complicated way of specifying the address and is not appropriate for addresses that are fixed in the memory map, such as those of peripheral registers. In a few cases, PC-relative addressing is essential to produce *position-independent code*, which can be moved in memory without affecting its function. Symbolic addressing is useful for such applications but it is highly specialized, such as bootloaders where the code runs from RAM rather than ROM. It is hard to see the attraction of symbolic addressing in the MSP430 apart from this, because absolute addressing can also reach the whole memory map. It is more important with 20-bit addresses in the MSP430X.

**Absolute Mode**      *label is preceded by '&'*

The constant in this form of indexed addressing is the absolute address of the data. This is already the complete address required so it should be added to a register that contains 0. The MSP430 mimics this by using the status register SR. It makes no sense to use the real contents of SR in an address so it behaves as though it contains 0 when it is used as the base for indexed addressing. This is one of its roles as constant generator CG1.

Absolute addressing is shown by the prefix `&` and should be used for special function and peripheral registers, whose addresses are fixed in the memory map. This example copies the port I input register into register R6:

```
mov.b  &PIIN,R6      ; load byte PIIN into R6, absolute mode
```

The assembler replaces this by the indexed form

### SP-Relative Mode

TI does not claim this as a distinct mode of addressing, but many other companies do! The stack pointer SP can be used as the register in indexed mode like any other. Recall from the section "Stack Pointer (SP)" on page 120 that the stack grows down in memory and that SP points to (holds the address of) the most recently added word. Suppose that we wanted to copy the value that had been pushed onto the stack before the most recent one. The following instruction will do this:

```
MOV.W 2(SP),R6 ; copy most recent word but one from stack
```

For example, suppose that the stack were as shown in Figure 5.2(d) with SP = 0x027C. Then the preceding instruction would load 0x1234 into R6. This type of manipulation is important in subroutines and interrupts and is illustrated in Chapter 6.

### 5.2.3 Indirect Register Mode

This is available only for the source and is shown by the symbol @ in front of a register, such as @R5. It means that the contents of R5 are used as the *address* of the operand. In other words, R5 holds a pointer rather than a value. (The contents of R5 would be the operand itself if the @ were omitted.) Suppose that R5 contains the value 4 before this instruction:

```
MOV.W @R5,R6 ; load word from address (R5)=4 into R6
```

The address of the source is 4, the value in R5. Thus a word is loaded from address 4 into R6. The value in R5 is unchanged. This has exactly the same effect as indexed addressing with a base address of 0 but saves a word of program memory, which also makes it faster. This is very loosely the equivalent of  $r6 = *r5$  in C, without worrying about the types of the variables held in the registers.

Indirect addressing cannot be used for the destination so indexed addressing must be used instead. Thus the reverse of the preceding move must be done like this:

```
MOV.W R6,0(R5) ; store word from R6 into address 0+(R5)=4
```

The penalty is that a word of 0 must be stored in the program memory and fetched from it. The constant generator cannot be used.



### 5.2.4 Indirect Autoincrement Register Mode

Again this is available only for the source and is shown by the symbol @ in front of a register with a + sign after it, such as @R5+. It uses the value in R5 as a pointer and automatically increments it afterward by 1 if a byte has been fetched or by 2 for a word. Suppose yet again that R5 contains the value 4 before this instruction:

```
MOV.W @R5+, R6
```

A word is loaded from address 4 into R6 and the value in R5 is incremented to 6 because a word (2 bytes) was fetched. This is useful when stepping through an array or table, where expressions of the form \*c++ are often used in C.

This mode cannot be used for the destination. Instead the main instruction must use indexed mode with an offset of 0, followed by an explicit increment of the register by 1 or 2. The reverse of this move therefore needs two instructions:

```
MOV.W R6, 0(R5)    ; store word from R6 into address 0+(R5)+4
INCD.W R5           ; R5 ++ 2
```

This is undoubtedly a bit clumsy.

Autoincrement is usually called *postincrement* addressing because many processors have a complementary *predecrement* addressing mode, equivalent to \*--c in C, but the MSP430 does not.

An important feature of the addressing modes is that all operations on the first address are fully completed before the second address is evaluated. This needs to be considered when moving blocks of memory. The move itself might be done by a line like this:

```
MOV.W @R5+, 0x0100(R5)
```

Suppose as usual that R5 initially contains the value 4. The contents of address 4 is read and R5 is double-incremented to 6 because a word is involved. Only now is the address for the destination calculated as  $0x0100 + 0x0006 = 0x0106$ . Thus a word is copied from address 0x0004 to 0x0106; the offset is not just the value of 0x0100 used as the base address for the destination. The compiler takes care of these tricky details if you write in C, but you are on your own with assembly language.

**Immediate Mode**

This is a special case of autoincrement addressing that uses the program counter PC. Look at this example:

```
MOV.W  @PC+,R6 ; load immediate word into R6
```

The PC is automatically incremented after the instruction is fetched and therefore points to the following word. The instruction loads this word into R6 and increments PC to point to the next word, which in this case is the next instruction. The overall effect is that the word that followed the original instruction has been loaded into R6. This is how the MSP430 handles immediate or literal values, which are encoded into the stream of instructions. It is the equivalent of `r6 = constant`.

This is available only for the source of an instruction because it is a type of autoincrement addressing but it is hard to see when it would be useful for the destination.

6 (a) Explain how constants are generated using CG1 and CG2 registers in MSP430.

[06]	CO1	L2
[04]	CO1	L3
[04]	CO1	L3

## Constant Generator

- \* Six commonly used constants are generated with constant registers R2 and R3, without requiring an additional 16-bit word of program code.
- \* The constants are selected with the source register addressing modes (As).
- \* Constant generators cannot be used as destination.

Register	As	Constant	Remarks
R2	00	--	Register mode
R2	01	(0)	Absolute mode
R2	10	0004H	+4, bit processing
R2	11	0008H	+8, bit "
R3	00	0000H	0, word "
R3	01	0001H	+1
R3	10	0002H	+2 bit processing
R3	11	0FFFFH	-1, word "

Ex → `mov.w R3, R5 ; clears the R5`

`mov.w @R2, R5 ; returns the value 8 and this will be stored in R5`

b.) Assuming that SP=270H, Show the contents of the stack and registers as each of the following instruction is executed. Given R5=100H, [100H]=1234h, [200H]=5678H

```
PUSH.W #200H
POP.W R5
PUSH.W @R5+
POP.B R6
```

7. a)

What are Emulated instructions? Explain with two examples.

## Emulated Instructions

The RISC instruction set of the MSP430 has only 27 instructions. However, the constant generator allows the MSP430 assembler to support 24 additional emulated instructions.

EX:

- ① "CLR dst" is emulated by  
MOV R3, dst ; R3 used with AS=02
- ② INC dst  
ADD 0(R3), dst

7(b) Explain the following instructions with examples.

i. SBC    ii. ADDC    iii. TST    iv. DECD    v. BIC    vi. BIT

[06] CO1 L2