UNIT 3

MOTOR CONTROL SIGNAL PROCESSORS

3.1 INTRODUCTION

The Texas Instruments **TMS320LF2407 DSP Controller** (referred to as the LF2407 in this text) is a programmable digital controller with a **C2xx DSP** central processing unit (CPU) as the core processor. The LF2407 contains the **DSP core processor and useful peripherals** integrated onto a single piece of silicon. The LF2407 combines the powerful CPU with on-chip memory and peripherals. With the DSP core and control-oriented peripherals integrated into a single chip, users **can design very compact and cost-effective digital control systems**.

The LF2407 DSP controller offers **40 million instructions per second (MIPS)** performance. This high processing speed of the C2xx CPU allows **users to compute parameters in real time** rather than look up approximations from tables stored in memory. This fast performance is well suited for processing control parameters in applications such as notch filters or sensor less motor control algorithms where a large amount of calculations must be computed quickly.

While the "brain" of the LF2407 DSP is the C2xx core, the LF2407 contains several control-orientated peripherals onboard (see Fig. 3.1). The peripherals on the LF2407 make virtually any digital control requirement possible. Their applications range from analog to digital conversion to pulse width modulation (PWM) generation. Communication peripherals make possible the communication with external peripherals, personal computers, or other DSP processors. Below is a brief listing of the different peripherals onboard the LF2407 followed by a graphical layout depicted in Fig. 3.1.

The LF2407 peripheral set includes:

- Two Event Managers (A and B)
- General Purpose (GP) timers
- PWM generators for digital motor control
- Analog-to-digital converter
- Controller Area Network (CAN) interface

• Serial Peripheral Interface (SPI) – synchronous serial port

• Serial Communications Interface (SCI) – asynchronous serial port

• General-Purpose bi-directional digital I/O (GPIO) pins

• Watchdog Timer (“time-out” DSP reset device for system integrity)

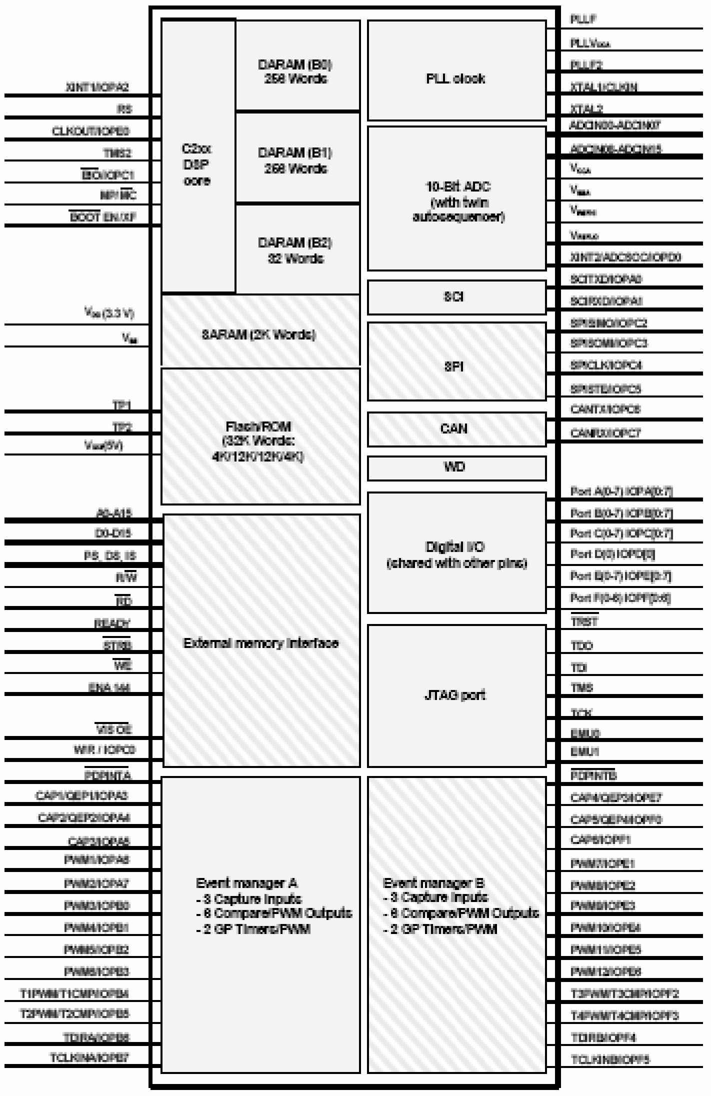


Figure 3.1 Graphical overview of DSP core and peripherals on the LF2407.

3.2 Brief Introduction to Peripherals

The following peripherals are those that are integrated onto the LF2407 chip. Refer to Fig. 1.1 to view the pin-out associated with each peripheral.

Event Managers (EVA, EVB)

There are two Event Managers on the LF2407, the EVA and EVB. The Event Manager is the most important peripheral in digital motor control. **It contains the necessary functions needed to control electromechanical devices.** Each EV is composed of functional "blocks" including **timers, comparators, and capture units for triggering on an event, PWM logic circuits, quadrature-encoder–pulse (QEP) circuits, and interrupt logic**.

The Analog-to-Digital Converter (ADC)

The ADC on the LF2407 is used whenever an **external analog signal needs to be sampled and converted to a digital number**. Examples of ADC applications range from sampling a control signal for use in a digital notch filtering algorithm or using the ADC in a control feedback loop to monitor motor performance. Additionally, the ADC is useful in motor control applications because it allows for current sensing using a shunt resistor instead of an expensive current sensor.

The Control Area Network (CAN) Module

While the CAN module will not be covered in this text, it is a useful peripheral for specific applications of the LF2407. The CAN module is used for **multi-master serial communication between external hardware.** The CAN bus has a **high level of data integrity and is ideal for operation in noisy environments** such as in an automobile, or industrial environments that require reliable communication and data integrity.

Serial Peripheral Interface (SPI) and Serial Communications Interface (SCI)

The SPI is a high-speed synchronous communication port that is mainly used for **communicating between the DSP and external peripherals or another DSP device**. Typical uses of the SPI include communication with external shift registers, display drivers, or ADCs.
The SCI is an **asynchronous communication port that supports asynchronous serial (UART) digital communication between the CPU and other asynchronous peripherals** that use the standard NRZ (non-return-to-zero) format. It is useful in communication between external devices and the DSP. Since these communication peripherals are not directly related to motion control applications, they will not be discussed further in this text.

Watchdog Timer (WD)

The Watchdog timer (WD) peripheral monitors software and hardware operations and asserts a system reset when its internal counter overflows. The WD timer (when enabled) will count for a specific amount of time. **It is necessary for the user's software to reset the WD timer periodically so that an unwanted reset does not occur**. If for some reason there is a CPU disruption, the watchdog will generate a system reset. For example, if the software enters an endless loop or if the CPU becomes temporarily disrupted, the WD timer will overflow and a DSP reset will occur, which will cause the DSP program to branch to its initial starting point. Most error conditions that temporarily disrupt chip operation and inhibit proper CPU function can be cleared by the WD function. In this way, the WD increases the reliability of the CPU, thus ensuring system integrity.

General Purpose Bi-Directional Digital I/O (GPIO) Pins

Since there are only a finite number of pins available on the LF2407 device**, many of the pins are multiplexed to either their primary function or the secondary GPIO function**. In most cases, a pin's second function will be as a general-purpose input/output pin. The GPIO capability of the LF2407 is very useful as a means of controlling the functionality of pins and also provides another method to input or output data to and from the device. **Nine 16-bit control registers control all I/O and shared pins**. There are two types of these registers:

- **I/O MUX Control Registers (MCRx)** – Used to control the multiplexer selection that Chooses between the primary function of a pin or the general-purpose I/O function.
- **Data and Direction Control Registers (PxDATDIR)** – Used to control the data and data Direction of bi-directional I/O pins.

Phase Locked Loop (PLL) Clock Module

The phase locked loop (PLL) module is basically **an input clock multiplier that allows the user to control the input clocking frequency to the DSP core**. External to the LF2407, a clock reference (can oscillator/crystal) is generated. This signal is fed into the **LF2407 and is multiplied or divided by the PLL.** This new (higher or lower frequency) clock signal is then used to clock the DSP core. The LF2407's **PLL allows the user to select a multiplication factor ranging from 0.5X to 4X that of the external clock signal**. The default value of the PLL is **4X**.

Memory Allocation Spaces

The LF2407 DSP Controller has three different allocations of memory it can use: **Data, Program, and I/O memory space**. **Data space** is used for **program calculations**, look-up tables, and any other memory used by an algorithm. Data memory can be in the form of the on-chip random access memory (RAM) or external RAM. Program memory is the location of user's program code. **Program memory on the LF2407 is either mapped to the off-chip RAM** (MP/MC- pin =1) or to the on-chip flash memory (MP/MC- = 0), depending on the logic value of the MP/MC-pin.

I/O space is not really memory but a virtual memory address used to output data to peripherals external to the LF2407. For example, the digital-to-analog converter (DAC) on the Spectrum Digital TM evaluation module is **accessed with I/O memory**. If one desires to output data to the DAC, the data is simply sent to the configured address of I/O space with the "OUT" command. This process is similar to writing to data memory except that the OUT command is used and the data is copied to and outputted on the DAC instead of being stored in memory.

3.3 Types of Physical Memory

Random Access Memory (RAM)

The LF2407 has **544 words of 16 bits each in the on-chip DARAM**. These 544 words are partitioned into **three blocks: B0, B1, and B2**. Blocks B1 and B2 are allocated for use only as data memory. Memory block B0 is different than B1 and B2. This memory block is normally configured as Data Memory, and hence primarily used to hold data, **but in the case of the B0 block, it can also be configured as Program Memory.** B0 memory can be configured as program or data memory depending on the value of the core level "CNF" bit.

- (CNF=0) maps B0 to data memory.
- (CNF=1) maps B0 to program memory.

The LF2407 also has 2K of single-access RAM (SARAM). The addresses associated with the SARAM can be used for both data memory and program memory, and are software configurable to the internal SARAM or external memory.

Non-Volatile Flash Memory

The LF2407 contains 32K of on-chip flash memory that can be mapped to program space if the **MP/MC-pin is made logic 0** (tied to ground). The flash memory provides a permanent location to store code that is unaffected by cutting power to the device. The flash memory can be electronically programmed and erased many times to allow for code development. Usually, the external RAM on the LF2407 Evaluation Module (EVM) board is used instead of the flash for code development due to the fact that a separate "flash programming" routine must be performed to flash code into the flash memory. The on-chip flash is normally used in situations where the DSP program needs to be tested where a JTAG connection is not practical or where the DSP needs to be tested as a "stand-alone" device. For example, if a LF2407 was used to develop a DSP control solution to an automobile braking system, it would be somewhat impractical to have a DSP/JTAG/PC interface in a car that is undergoing performance testing.

3.4 Introduction to the C2xx DSP Core and Code Generation

The heart of the LF2407 DSP Controller is the C2xx DSP core. This core is a 16-bit fixed point processor, meaning that it works with 16-bit binary numbers. One can think of the C2xx as the central processor in a personal computer. The LF2407 DSP consists of the C2xx DSP core plus many peripherals such as Event Managers, ADC, etc., all integrated onto one single chip.

3.5 The Components of the C2xx DSP Core

The DSP core (like all microprocessors) consists of several subcomponents necessary to perform arithmetic operations on 16-bit binary numbers. The following is a list of the multiple subcomponents found in the C2xx core which we will discuss further:

- **A 32-bit central arithmetic logic unit (CALU)**
- **A 32-bit accumulator (used frequently in programs)**
- **Input and output data-scaling shifters for the CALU**
- **A (16-bit by 16-bit) multiplier**
- **A product-scaling shifter**
- **Eight auxiliary registers (AR0 – AR7) and an auxiliary register arithmetic unit (ARAU)**

Each of the above components is either accessed directly by the user code or is indirectly used during the execution of an assembly command.

Central Arithmetic Logic Unit (CALU)

The C2xx performs **2s-complement arithmetic** using the 32-bit CALU. **The CALU uses 16-bit words taken from data memory**, derived from an immediate instruction, or from the 32-bit multiplier result. In addition to arithmetic operations, **the CALU can perform Boolean operations.** The CALU is somewhat transparent to the user. For example, if an arithmetic command is used, the user only needs to write the command and later read the output from the appropriate register. In this sense, the CALU is "transparent" in that it is not accessed directly by the user.

Accumulator

The accumulator stores the output from the CALU and also serves as another input to the CALU (many arithmetic commands perform operations on numbers that are currently stored in the accumulator; versus other memory locations). The accumulator is 32 bits wide and is divided into two sections, each consisting of 16 bits. The high-order bits consist of bits 31 through 16, and the low-order bits are made up of bits 15 through 0. Assembly language instructions are provided for storing the high- and low-order accumulator words to data memory. In most cases, the accumulator is written to and read from directly by the user code via assembly commands. In some instances, the accumulator is also transparent to the user (similar to the CALU operation in that it is accessed "behind the scenes").

Scaling Shifters

The C2xx has three 32-bit shifters that allow for scaling, bit extraction, extended arithmetic, and overflow-prevention operations. The scaling shifters make possible commands that shift data left or right. Like the CALU, the operation of the scaling shifters is "transparent" to the user. For example, the user needs only to use a shift command, and observe the result. Any one of the three shifters could be used by the C2xx depending on the specific instruction entered. The following is a description of the three shifters:

- **Input data-scaling shifter (input shifter):** This shifter left-shifts 16-bit input data by 0 to 16 bits to align the data to the 32-bit input of the CALU. For example, when the user uses a command such as "ADD 300h, 5", the input shifter is responsible for first shifting the data in memory address "300h" to the left by five places before it is added to the contents of the accumulator.

- **Output data-scaling shifter (output shifter):** This shifter left-shifts data from the accumulator by 0 to 7 bits before the output is stored to data memory. The content of the accumulator remains unchanged. For example, when the user uses a command such as “SACL 300h, 4”, the output shifter is responsible for first shifting the contents of the accumulator to the left by four places before it is stored to the memory address “300h”.
- **Product-scaling shifter (product shifter)**: The product register (PREG) receives the output of the multiplier. The product shifter shifts the output of the PREG before that output is sent to the input of the CALU. The product shifter has four product shift modes (no shift, left shift by one bit, left shift by four bits, and right shift by six bits), which are useful for performing multiply/accumulate operations, fractional arithmetic, or justifying fractional products.

Multiplier

The multiplier performs 16-bit, 2s-complement multiplication and creates a 32-bit result. In conjunction with the multiplier, the C2xx uses the 16-bit temporary register (TREG) and the 32-bit product register (PREG).

The operation of the multiplier is not as “transparent” as the CALU or shifters. The TREG **always** needs to be loaded with one of the numbers that are to be multiplied. Other than this prerequisite, the multiplication commands do not require any more actions from the user code. The output of the multiply is stored in the PREG, which can later be read by the user code.

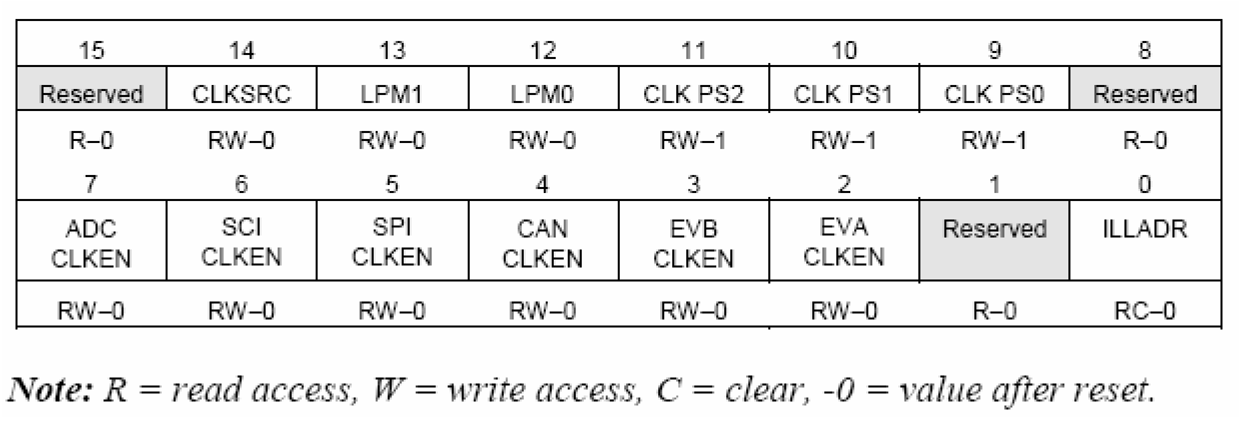
Auxiliary Register Arithmetic Unit (ARAU) and Auxiliary Registers

The ARAU generates data memory addresses when an instruction uses indirect addressing to access data memory (more on indirect addressing will be covered later along with assembly programming). Eight auxiliary registers (AR0 through AR7) support the ARAU, each of which can be loaded with a 16-bit value from data memory or directly from an instruction. Each auxiliary register value can also be stored in data memory. The auxiliary registers are mainly used as “pointers” to data memory locations to more easily facilitate looping or repeating algorithms. They are directly written to by the user code and are automatically incremented or decremented by particular assembly instructions during a looping or repeating operation. The auxiliary register pointer (ARP) embedded in status register ST0 references the auxiliary register. The status registers (ST0, ST1) are core level registers where values such as the Data Page (DP) and ARP located.

3.6 System Configuration Registers

The System Control and Status Registers (SCSR1, SCSR2) are used to configure or display fundamental settings of the LF2407. For example, these fundamental settings include the clock speed (clock pre-scale setting) of the LF2407, which peripherals are enabled, microprocessor/microcontroller mode, etc. Bits are controlled by writing to the corresponding data memory address or the logic level on an external pin as with the microprocessor/microcontroller (MP/MC) select bit. The bit descriptions of these two registers (mapped to data memory) are listed below.

System Control and Status Register 1 (SCSR1) — Address 07018h



15	14	13	12	11	10	9	8
Reserved	CLKSRC	LPM1	LPM0	CLK PS2	CLK PS1	CLK PS0	Reserved
R–0	RW–0	RW–0	RW–0	RW–1	RW–1	RW–1	R–0
7	6	5	4	3	2	1	0
ADC CLKEN	SCI CLKEN	SPI CLKEN	CAN CLKEN	EVB CLKEN	EVA CLKEN	Reserved	ILLADR
RW–0	RW–0	RW–0	RW–0	RW–0	RW–0	R–0	RC–0

Note: R = read access, W = write access, C = clear, -0 = value after reset.

Bit 15 Reserved

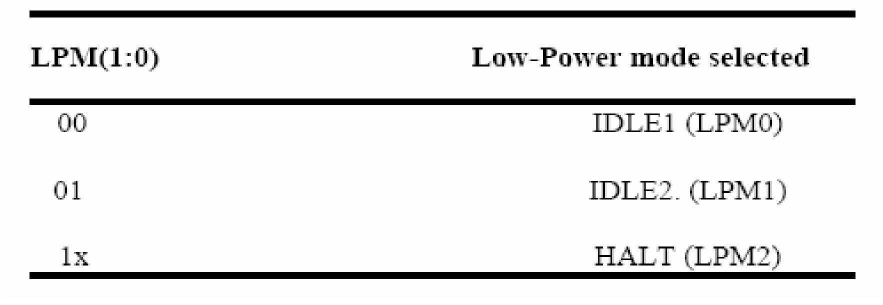
Bit 14 CLKSRC. CLKOUT pin source select

0 CLKOUT pin has CPU Clock (40 MHz on a 40-MHz device) as the output

1 CLKOUT pin has Watchdog clock as the output

Bits 13–12 LPM (1:0). Low-power mode select

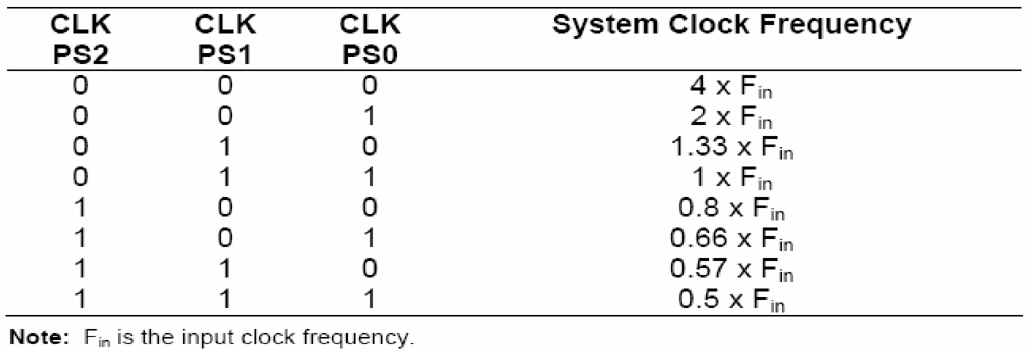
These bits indicate which low-power mode is entered when the CPU executes the IDLE instruction. Description of the low-power modes:



LPM(1:0)	Low-Power mode selected
00	IDLE1 (LPM0)
01	IDLE2. (LPM1)
1x	HALT (LPM2)

Bits 11–9

PLL Clock prescale select. These bits select the PLL multiplication factor for the input clock.



CLK PS2	CLK PS1	CLK PS0	System Clock Frequency
0	0	0	4 x F_{in}
0	0	1	2 x F_{in}
0	1	0	1.33 x F_{in}
0	1	1	1 x F_{in}
1	0	0	0.8 x F_{in}
1	0	1	0.66 x F_{in}
1	1	0	0.57 x F_{in}
1	1	1	0.5 x F_{in}

Note: F_{in} is the input clock frequency.

Bit 8 Reserved

Bit 7 ADC CLKEN. ADC module clock enable control bit.

0	Clock to module is disabled (i.e., shut down to conserve power).
1	Clock to module is enabled and running normally.

Bit 6 SCI CLKEN. SCI module clock enable control bit.

0	Clock to module is disabled (i.e., shut down to conserve power).
1	Clock to module is enabled and running normally.

Bit 5 SPI CLKEN. SPI module clock enable control bit

0	Clock to module is disabled (i.e., shut down to conserve power)
1	Clock to module is enabled and running normally

Bit 4 CAN CLKEN. CAN module clock enable control bit

0	Clock to module is disabled (i.e., shut down to conserve power)
1	Clock to module is enabled and running normally

Bit 3 EVB CLKEN. EVB module clock enable control bit

0	Clock to module is disabled (i.e., shut down to conserve power)
1	Clock to module is enabled and running normally

Bit 2 EVA CLKEN. EVA module clock enable control bit

0	Clock to module is disabled (i.e., shut down to conserve power)
1	Clock to module is enabled and running normally

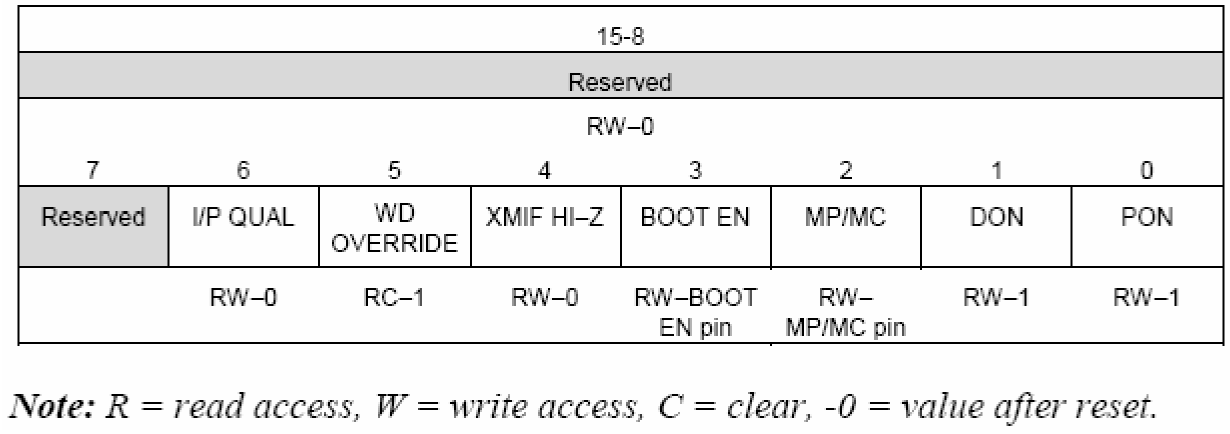
Note: In order to modify/read the register contents of any peripheral, the clock to that peripheral must be enabled by writing a 1 to the appropriate bit.

Bit 1 Reserved

Bit 0 ILLADR. Illegal Address detect bit

If an illegal address has occurred, this bit will be set. It is up to software to clear this bit following an illegal address detects. This bit is cleared by writing a 1 to it and should be cleared as part of the initialization sequence. Note: An illegal address will cause a Non-Mask able Interrupt (NMI).

System Control and Status Register 2 (SCSR2) — Address 07019h



15-8							
Reserved							
RW–0							
7	6	5	4	3	2	1	0
Reserved	I/P QUAL	WD OVERRIDE	XMIF HI–Z	BOOT EN	MP/MC	DON	PON
	RW–0	RC–1	RW–0	RW–BOOT EN pin	RW– MP/MC pin	RW–1	RW–1

Note: R = read access, W = write access, C = clear, -0 = value after reset.

Bits 15–7 Reserved. Writes have no effect; reads are undefined

Bit 6 Input Qualifier Clocks.

An input-qualifier circuitry qualifies the input signal to the CAP1–6, XINT1/2, ADCSOC, and PDPINTA/B pins in the 240xA devices. The I/O functions of these pins do not use the input-qualifier circuitry. The state of the internal input signal will change only after the pin is held high/low for 6 (or 12) clock edges. This ensures that a glitch smaller than (or equal to) 5 (or 11) CLKOUT cycles wide will not change the internal pin input state. The user must hold the pin high/low for 6 (or 12) cycles to ensure that the device will see the level change. This bit determines the width of the glitches (in number of internal clock cycles) that will be blocked. Note that the internal clock is not the same as CLKOUT, although its frequency is the same as CLKOUT.

0	The input-qualifier circuitry blocks glitches up to 5 clock cycles long
1	The input-qualifier circuitry blocks glitches up to 11 clock cycles long

Note: This bit is applicable only for the 240xA devices, not for the 240x devices because they lack an input-qualifier circuitry.

Bit 5 Watchdog Override. (WD protect bit)

After RESET, this bit gives the user the ability to disable the WD function through software (by setting the WDDIS bit = 1 in the WDCR). This bit is a clear-only bit and defaults to a 1 after reset.

Note: This bit is cleared by writing a 1 to it.

0 Protects the WD from being disabled by software. This bit cannot be set to 1 by software. It is a clear-only bit, cleared by writing a 1.

1 This is the default reset value and allows the user to disable the WD through the WDDIS bit in the WDCR. Once cleared, however, this bit can no longer be set to 1 by software, thereby protecting the integrity of the WD timer.

Bit 4 XMIF Hi-Z Control

This bit controls the state of the external memory interface (XMIF) signals.

0 XMIF signals in normal driven mode; i.e., not Hi-Z (high impedance).

1 All XMIF signals are forced to Hi-Z state.

Bit 3 Boot Enable

This bit reflects the state of the BOOT_EN / XF pin at the time of reset. After reset and device has “booted up”, this bit can be changed in software to re-enable Flash memory visibility or return to active Boot ROM.

0 Enable Boot ROM — Address space 0000 — 00FF is now occupied by the on-chip Boot ROM Block. Flash memory is totally disabled in this mode. Note: There is no on-chip boot ROM in ROM devices (i.e., LC240xA)

1 Disable Boot ROM — Program address space 0000 — 7FFF is mapped to on-chip Flash memory in the case of LF2407A and LF2406A. In the case of LF2402A, addresses 0000 – 1FFF are mapped

Bit 2 Microprocessor/Microcontroller Select

This bit reflects the state of the MP/MC pin at time of reset. After reset, this bit can be changed in software to allow dynamic mapping of memory on and off chip.

0 Set to Microcontroller mode — Program Address range 0000 — 7FFF is mapped internally (i.e., Flash)

1 Set to Microprocessor mode — Program Address range 0000 — 7FFF is mapped externally (i.e., customer provides external memory device.)

Bits 1–0 SARAM Program/Data Space Select

DON	PON	SARAM status
0	0	SARAM not mapped (disabled), address space allocated to external memory
0	1	SARAM mapped internally to Program space
1	0	SARAM mapped internally to Data space
1	1	SARAM block mapped internally to both Data and Program spaces.

This is the default or reset value

3.7 Memory Addressing Modes

There are three basic memory addressing modes used by the C2xx instruction set. The three modes are:

- Immediate addressing mode (does not actually access memory)
- Direct addressing mode
- Indirect addressing mode

3.7.1 Immediate Addressing Mode

In the immediate addressing mode, the instruction contains a constant to be manipulated by the instruction. Even though the name "immediate addressing" suggests that a memory location is accessed, immediate addressing is simply dealing with a user-specified constant which is usually included in the assembly command syntax. The "#" sign indicates that the value is an immediate address (just a constant). The two types of immediate addressing modes are:

Short-immediate addressing. The instructions that use short-immediate addressing have an 8-bit, 9-bit, or 13-bit constant as the operand.

For example, the instruction:

```
LACL #44h                ; loads lower bits of accumulator with
                         ; Eight-bit constant (44h in this case)
```

Note: *The LACL command will work only with a short 8-bit constant. If you want to load a long 16-bit constant, then use the LACC command.*

Long-immediate addressing. Instructions that use long-immediate addressing have a 16-bit constant as an operand. This 16-bit value can be used as an absolute constant or as a 2s-complement value.

For example, the instruction:

```
    LACC #4444h              ; loads accumulator with up to a 16-bit
```

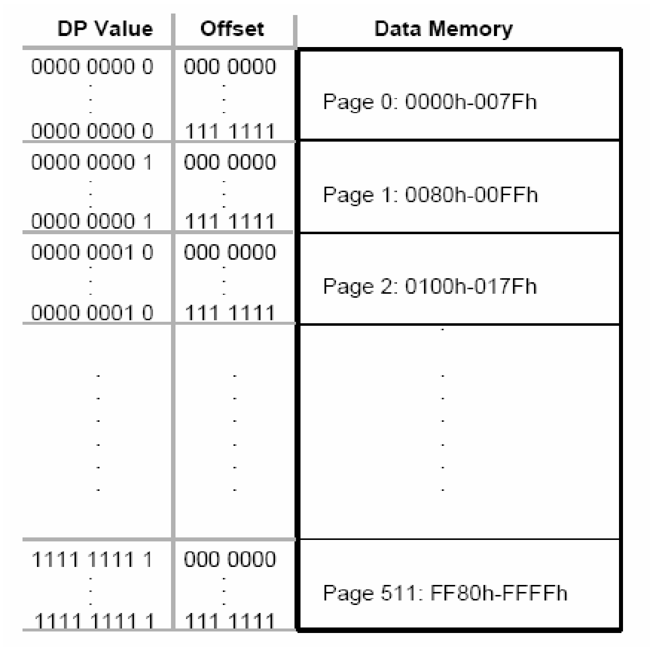
; Constant (4444h in this case)

If you need to use registers or access locations in data memory, you must use either direct or indirect addressing.

3.7.2 Direct Addressing Mode

In direct addressing, data memory is first addressed in blocks of 128 words called data pages. The entire 64K of data memory consists of 512 DPs labeled 0 through 511, as shown in the Fig. 3.2. The current DP is determined by the value in the 9-bit DP pointer in status register ST0. For example, if the DP value is “0 0000 0000”, the current DP is 0. If the DP value is “0 0000 0010”, the current data page is 2. The DP of a particular memory address can be found easily by dividing the address (in hexadecimal) by 80h.

For example: For the data memory address 0300h, 300h/80h = 6h so the DP pointer is 6h. Likewise, the DP pointer for 200h is 4h.



DP Value	Offset	Data Memory
0000 0000 0 : 0000 0000 0	000 0000 : 111 1111	Page 0: 0000h-007Fh
0000 0000 1 : 0000 0000 1	000 0000 : 111 1111	Page 1: 0080h-00FFh
0000 0001 0 : 0000 0001 0	000 0000 : 111 1111	Page 2: 0100h-017Fh
. . . . . .	. . . . . .	. . . . . .
1111 1111 1 : 1111 1111 1	000 0000 : 111 1111	Page 511: FF80h-FFFFh

Figure 3.2 Data pages and corresponding memory ranges.

In addition to the DP, the DSP must know the particular word being referenced on that page. This is determined by a 7-bit offset. The 7-bit offset is simply the 7 least significant bits (LSBs) of the memory address. The DP and the offset make up the 16-bit memory address (see Fig. 3.3).

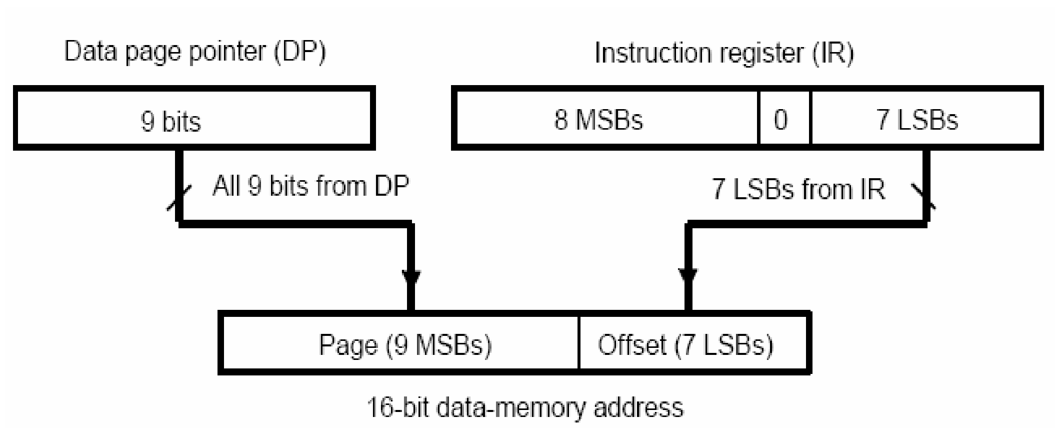


Figure 3.3 Data page and offset make up a 16-bit memory address.

When you use direct addressing, the processor uses the 9 DP bits and the 7 LSBs of the instruction to obtain the true memory address. The following steps should be followed when using direct addressing:

1 Set the DP. Load the appropriate value (from 0 to 511 in decimal or 0-1FF in hex) into the DP. The easiest way to do this is with the LDP instruction. The LDP instruction loads the DP directly to the ST0 register without affecting any other bits of the ST0.

LDP #0E1h *; sets the data page pointer to E1h*

Or

LDP #225 *; sets the data page pointer to 225 decimal*

; Which is E1 in hexadecimal

2 Specify the offset. For example, if you want the ADD instruction to use the value at the second address of the current data page, you would write: ADD 1h

If the data page points to 300h, then the above instruction will add the contents of 301h to the accumulator

Note: You do not have to set the data page prior to every instruction that uses direct addressing. If all the instructions in a block of code access the same data page, you can simply load the DP before the block. However, if various data pages are being accessed throughout the block of code be sure the DP is changed accordingly.

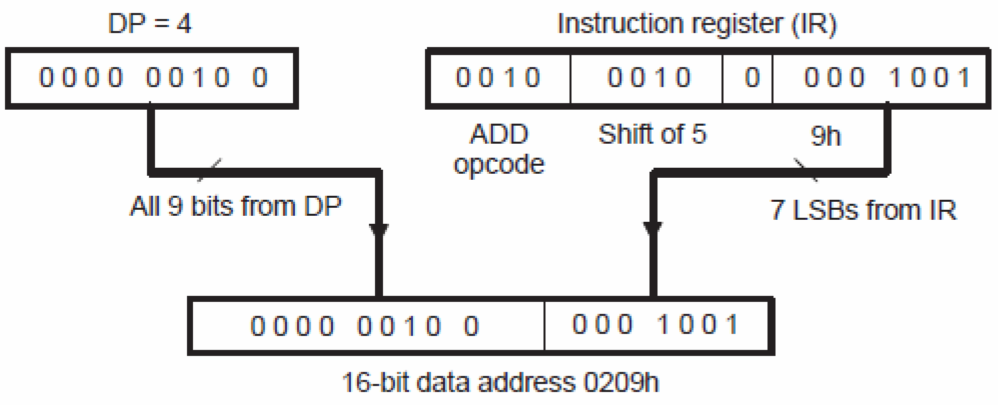
Examples of Direct Addressing

In Example 1, the first instruction loads the DP with 0 0000 0100_2 to set the current data page to 4. The ADD instruction then references a data memory address that is generated as shown following the program code. Before the ADD instruction is executed, the opcode is loaded into the instruction register. Together, the DP and the seven LSBs of the instruction register form the complete 16-bit address, 0000 0010 0000 1001_2 (0209h).

In Example 2, the ADD instruction references a data memory address that is generated as shown following the program code. For any instruction that performs a shift of 16, the shift value is not embedded directly in the instruction word; instead, all eight MSBs contain an opcode that not only indicates the instruction type, but also a shift of 16. The eight MSBs of the instruction word indicate an ADD with a shift of 16.

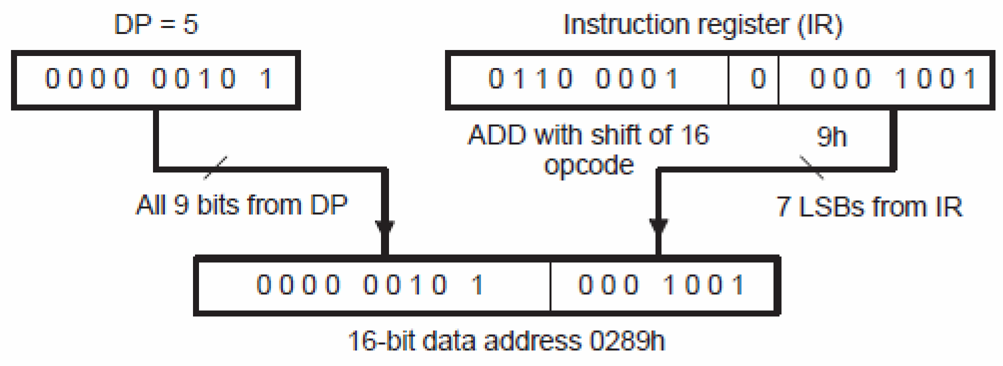
Example 1 **Using Direct Addressing with ADD (Shift of 0 to 15)**

```
LDP #4    ;Set data page to 4 (addresses 0200h-027Fh).
ADD 9h,5  ;The contents of data address 0209h are
          ;left shifted 5 bits and added to the
          ;contents of the accumulator.
```



Example 2 **Using Direct Addressing with ADD (Shift of 16)**

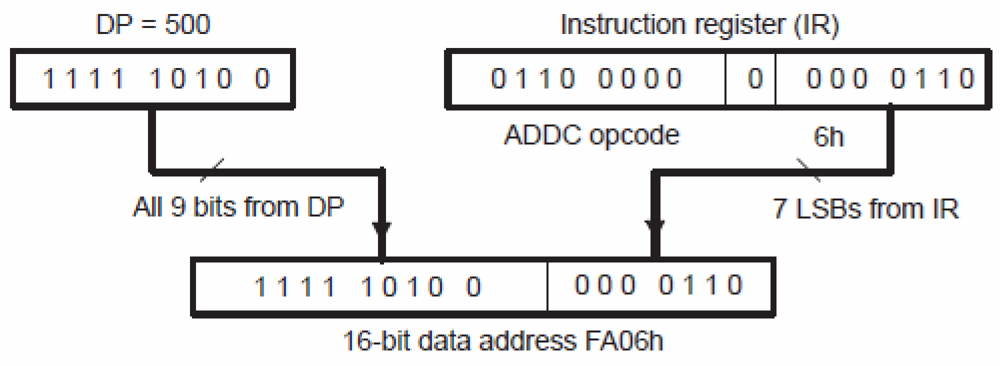
```
LDP #5      ;Set data page to 5 (addresses 0280h-02FFh).
ADD 9h,16   ;The contents of data address 0289h are
            ;left shifted 16 bits and added to the
            ;contents of the accumulator.
```



In Example 3, the ADDC instruction references a data memory address that is generated as shown following the program code. You should note that if an instruction does not perform shifts (such as the ADDC instruction), all eight MSBs of the instruction contain the opcode for the instruction type.

Example 3 Using Direct Addressing with ADDC

```
LDP #500   ;Set data page to 500 (addresses FA00h-FA7Fh).
ADDC 6h    ;The contents of data address FA06h
           ;and the value of the carry bit (C) are
           ;added to the contents of the accumulator.
```



3.7.3 Indirect Addressing Mode

Indirect addressing is a powerful way of addressing data memory. Indirect addressing mode is not dependent on the current data page as is direct addressing. Instead, when using indirect addressing you load the memory space that you would like to access into one of the auxiliary registers (ARx). The current auxiliary register acts as a pointer that points to a specific memory address.

The register pointed to by the ARP is referred to as the current auxiliary register or current AR. To select a specific auxiliary register, load the 3-bit auxiliary register pointer (ARP) with a value from 0 to 7. The ARP can be loaded with the MAR instruction or by the LARP instruction. An ARP value can also be loaded by using the ARx operand after any instruction that supports indirect addressing as seen below.

Example of using MAR:

```
ADD *, AR1          ; Adds using current *, then makes AR1 the
                    ; New current AR for future uses
```

Example of using LARP

```
LARP #2             ; this will make AR2 the current AR
```

The C2xx provides four types of indirect addressing options:

- **No increment or decrement.** The instruction uses the content of the current auxiliary register as the data memory addresses but neither increments nor decrements the content of the current auxiliary register.
- **Increment or decrement by 1.** The instruction uses the content of the current auxiliary register as the data memory address and then increments or decrements the content of the current auxiliary register by one.
- **Increment or decrement by an index amount.** The value in AR0 is the index amount. The instruction uses the content of the current auxiliary register as the data memory address and then increments or decrements the content of the current auxiliary register by the index amount.
- **Increment or decrement by an index amount using reverse carry.** The value in AR0 i the index amount. After the instruction uses the content of the current auxiliary register as the data memory address, that content is incremented or decremented by the index

amount. The addition and subtraction process is accomplished with the carry propagation reversed and is useful in fast Fourier transforms algorithms.

Table 3.1 displays the various operands that are available for use with instructions while using indirect addressing mode.

Operand	Option	Example
*	No increment or decrement	**LT** * loads the temporary register TREG with the content of the data memory address referenced by the current AR.
*+	Increment by 1	**LT** ***+** loads the TREG with the content of the data memory address referenced by the current AR and then adds 1 to the content of the current AR.
*-	Decrement by 1	**LT** *- loads the TREG with the content of the data memory address referenced by the current AR and then subtracts 1 from the content of the current AR.
*0+	Increment by index amount	**LT** ***0+** loads the TREG with the content of the data memory address referenced by the current AR and then adds the content of AR0 to the content of the current AR.
*0-	Decrement by index amount	**LT** ***0-** loads the TREG with the content of the data memory address referenced by the current AR and then subtracts the content of AR0 from the content of the current AR.
*BR0+	Increment by index amount, adding with reverse carry	**LT** ***BR0+** loads the TREG with the content of the data memory address referenced by the current AR and then adds the content of AR0 to the content of the current AR, adding with reverse carry propagation.
*BR0-	Decrement by index amount, subtracting with reverse carry	**LT** ***BR0-** loads the TREG with the content of the data memory address referenced by the current AR and then subtracts the content of AR0 from the content of the current AR, subtracting with bit reverse carry propagation.

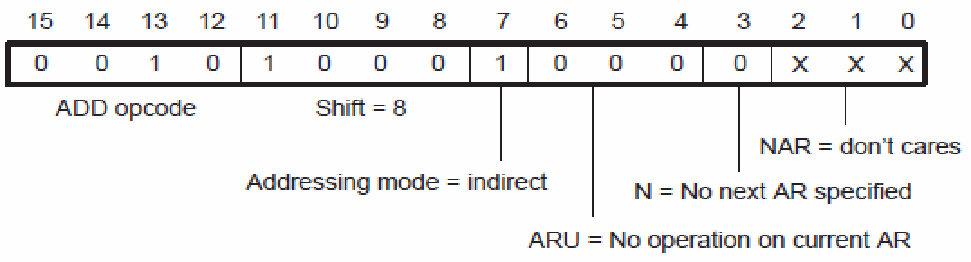
Table 3.1 Indirect addressing operands.

Examples of Indirect Addressing

Example 1 illustrates how the instruction register is loaded with the value shown when the ADD instruction is fetched from program memory.

Example 1. Indirect Addressing—No Increment or Decrement

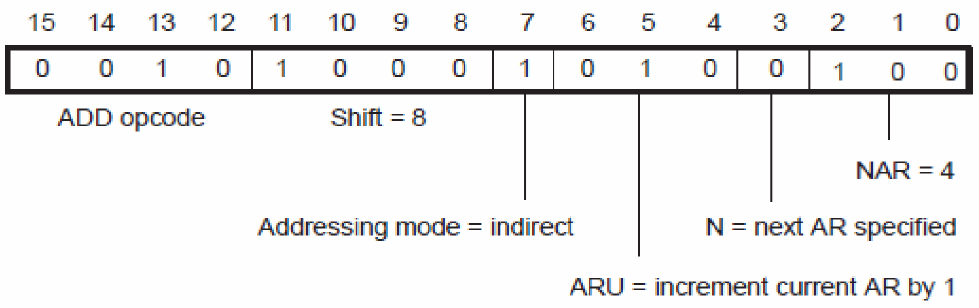
```
ADD *,8      ;Add to the accumulator the content of the
             ;data-memory address referenced by the
             ;current auxiliary register. The data
             ;is left shifted 8 bits before being added.
```



Example 2, illustrates how the instruction register is loaded with the value shown when the ADD instruction is fetched from program memory.

Example 2. **Indirect Addressing—Increment by 1**

```
ADD *+,8,AR4      ;Operates as in Example 1   , but
                  ;in addition, the current auxiliary
                  ;register is incremented by one, and
                  ;AR4 is chosen as the next auxiliary
                  ;register.
```



Example 3. **Indirect Addressing—Decrement by 1**

```
ADD *-,8      ;Operates as in Example 1   , but in
              ;addition, the current auxiliary register
              ;is decremented by one.
```

Example 4. **Indirect Addressing—Increment by Index Amount**

```
ADD *0+,8    ;Operates as in Example 1   , but in
             ;addition, the content of register AR0
             ;is added to the current auxiliary
             ;register.
```

Example 5. **Indirect Addressing—Decrement by Index Amount**

```
ADD *0-,8     ;Operates as in Example 1   , but in
              ;addition, the content of register AR0
              ;is subtracted from the current auxiliary
              ;register.
```

3.8 Assembly Programming Using the C2xx DSP Instruction Set

The complete detailed instruction set for the C2xx DSP core can be found in the Texas Instruments TMS320F/C24x DSP Controllers Reference Guide: CPU and Instruction Set; Literature Number: SPRU160C**.** This reference guide contains a complete descriptive listing on syntax, operands, binary opcode, instruction execution order, status bits affected by the instruction, number of memory words required to store the instruction, and clock-cycles used by the instruction. The Texas Instruments documentation on the assembly instruction set is very well written. Each assembly instruction has a complete explanation of the instruction, all optional operands, and several examples of the instructions used. Since including the instruction set and complete documentation would make this book excessively long, we will assume the reader has access to the documentation referred to above.

We will therefore focus on developing code, not the instruction set itself. Each command starts with the basic assembly instruction. Each command supports specific addressing modes and options. For example, the ADD command will work with direct, indirect, and immediate addressing. In addition to the basic command, many instructions have additional options that may be used with the instruction. For example, the ADD command supports left shifting of the data before it is added to the accumulator.

The following is the instruction syntax for the ADD command:

ADD *dma [, shift]*	*; Direct addressing*
ADD *dma,* ***16***	***;*** *Direct with left shift of 16*
ADD *ind [, shift [,* ***AR****n]]*	*; Indirect addressing*
ADD *ind,* ***16*** *[,* ***AR****n]*	*; Indirect with left shift of 16*
ADD *#k*	*; short immediate addressing*
ADD *#lk [, shift]*	*; Long immediate addressing*

The following is a list of the various notations used in C2xx syntax examples:

Italics Italic symbols in instruction syntax represent variables.

Example:

LACC dma, you can use several ways to address the dma (data memory address).

*LACC **

Or

LACC 200h

Or

LACC v	*; where "v" is any variable assigned to data memory*
	*Where *, 200h, and v are the data memory addresses*

Boldface Characters Boldface characters must be included in the syntax.

Example:

LAR** dma, **16	*; direct addressing with left shift of 16*
LAR AR1, 60h, 16	*; load auxiliary AR1 register with the memory contents of 60h that was left shifted 16 bits*

Example:

***LACC** dma, [shift]*	*; optional left shift from 0, 15; defaults to 0*
LACC main_counter, 8	*; shifts contents of the variable "main_counter" data 8 places to the left before loading accumulator*

[ ] An optional operand may be placed in the placed here.

Example:

***LACC** ind [, shift [, **AR** n] _]*	*Indirect addressing*
*LACC **	*; load Accum. W/contents of the memory* *; Location pointed to by the current AR.*
*LACC *, 5*	*; load Accum. With the contents of the memory* *; Location pointed to by the current AR after* *; The memory contents are left shifted by 5* *; Bits.*
*LACC *, 0, AR3*	*; load Accum. With the contents of the memory* *; Location pointed to by the current AR after* *; The memory contents are left shifted by 5* *; Bits. Now you have the option of choosing* *; A new AR. In this case, AR3 will become the* *; New AR.*

[, x1 [, x2]] Operands x1 and x2 are optional, but you cannot include x2 without also including x1.

It is optional when using indirect addressing to modify the data. Once you supply a left shift value from 0…15 (even a shift of 0), then you have the option of changing to a new current auxiliary register (AR).

# The # sign is prefix that signifies that the number used is a constant as opposed to memory location.

Example:

```
RPT #15         ; this syntax is using short immediate addressing. It will repeat the next
                instruction 15+1 times.
LACC #60h       ; this will load the accumulator with the
                ; Constant 60h
LACC 60h        ; However, this instruction will load the
                ; Accumulator with the contents in the data
                ; Memory location 60h, not the constant #60h
```

We will now provide a few examples of using the instruction set. Example 2.1 performs a few arithmetic functions with the DSP core and illustrates the nature of assembly programming. Programming with the assembly instruction set is somewhat different than languages such as C. In a high-level language, to add two numbers we might just code "c = a + b". In assembly, the user must be sure to code everything that needs to happen in order for a task to be executed. Take the following example:

Example 2.1 - Add the two numbers "2" and "3":

```
LDP #6h             ; loads the proper DP for dma 300h
SPLK #2, 300h       ; store the number "2" in memory address 300h
LACL #3             ; load the accumulator with the number "3"
ADD 300h            ; adds contents of 300h ("2") to the contents
                    ; of the accumulator ("3"); accumulator = 5
Another way:
LDP #6h             ; loads the proper DP for dma 300h
SPLK #2h, 300h      ; store the number "2h" in memory address
                    ; 300h
SPLK #3h, 301h      ; stores the number "3h" into memory address
                    ; 301h
```

```
LACL 300h              ; load the accumulator with the contents in
                       ; Memory location 300h
ADD 301h               ; adds contents of memory address 301h ("3h")
                       ; To the contents of the accumulator ("2h")
                       ;accumulator = 5h
```

Looping algorithms are very common in all programming languages. In high-level languages, the "For" and "While" loops can be used. However, in assembly, we need a slightly different approach to perform a repeating algorithm. The following example is an algorithm that stores the value "1" to memory locations 300h, 301h, 302h, 303h, and 304h.

Example 2.2- Looping Algorithm Using the Auxiliary Register

```
LAR AR0, #4            ; load auxiliary register 0 with #4
LAR AR1, #300h         ; this AR will be used as a memory pointer
LACL #1h               ; loads "1" into the accumulator
LOOPER MAR *, AR1      ; makes AR1 the next current AR
SACL *+, AR0           ; writes contents of accumulator to address
                       ; pointed to by AR1, the "+" increments AR1
                       ; By 1, next current AR is AR0
BANZ LOOPER            ; branch to LOOPER while current AR is not 0;
                       decrements current AR by 1 and branches
                       ; back to LOOPER
```

One might wonder if assembly language is so tedious to use, why not just program in a high-level language all the time. When code written in a high level language is compiled into assembly, the length of the code increases substantially. For example, if an assembly program takes up 50 lines, the same program written in C might take 150 lines after it is compiled. For this reason, code written in assembly almost always executed faster and uses less memory than high-level language code.