**PMAS Arid Agriculture University Rawalpindi**

**University Institute of Information Technology**

**LAB MANUAL – XII,XIII**

**Class/Program: BS (CS)** **Course: COAL (CS530)**



**Objectives:**

**1. Introduction to Logical Instruction**

**2. Introduction to bitwise operations**

**3. AND, OR, NOT, XOR operation**

**4. Shift Instruction**

**5. Types of Shift Instruction**

**LOGICAL INSTRUCTION**

Bitwise Logical instructions are the most primitive operations needed by every computer architecture. At a minimum, an architecture could provide a NAND operations, since all other logical functions can be derived from NAND operations.

These logical operations are semantically different to what is known as in most of high level programming language. The difference lies down at the fact that bitwise logical operations are performed at bit-by-bit basis.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Pentium provides five logical instructions. All logical instructions need two operands except NOT instructions which is a unary.   |  |  | | --- | --- | |  | **AND** destination, source | |  | **OR** destination, source | |  | **XOR** destination, source | |  | **TEST** destination, source | |  | **NOT** destination |  |  |  | | --- | --- | |  | The result of the operation is stored in the Destination, which must be a general **register** or a **memory location**. | |  | The Source may be an immediate value, register, or memory location. | |  | The Destination and Source CANNOT both be memory locations. | |  | The Destination and Source must be of the same size (8-, 16-. 32-bit). | |

All logic instructions, except TEST, modify the Destination operand. The TEST instruction does not modify any of its operands; however it affects the flags similar to the AND instruction.

All logical instructions, except NOT, affect the status flags

**Effects on Status Flag**

Since logical instructions operate on a bit-by-bit basis, no carry or overflow is generated.

|  |  |
| --- | --- |
|  | Except NOT, all logical instructions clear carry flag (CF) and overflow flag (OF). |
|  | AF is undefined |
|  | Remaining three flags record useful information: Zero flag (ZF), Sign flag (SF), Parity flag (PF). |
|  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Carry Flag** | **Overflow Flag** | **Zero Flag** | **Sign Flag** | **Parity Flag** | **Auxiliary Flag** |
| 0 | 0 | Modified | Modified | Modified | Undefined |

**The Truth tables**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Bit1** | **Bit2** | **Bit1 AND Bit2** | **Bit1 TEST Bit2** | **Bit1 OR Bit2** | **Bit1 XOR Bit2** |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

|  |
| --- |
| ***Example: AND BL, 0F0h instruction*** |
| |  |  | | --- | --- | | Mov | BL, 01010111b | | And | BL, 0F0h | |  |  | | Result in BL is: | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | | |

**The Usage**

The main usage of bitwise logical instructions is:

|  |  |
| --- | --- |
|  | to set |
|  | to clear |
|  | to invert |
|  | to isolate |

some selected bits in the Destination operand.

To do this, a Source bit pattern known as a **mask** is constructed. The mask bits are chosen so that the selected bits are modified in the desired manner when an instruction of the form:

LOGIC\_INSTRUCTION Destination , Mask

is executed. The Mask bits are chosen based on the following properties of AND, OR, and XOR :

If X represents a bit (either 0 or 1) then:

|  |  |  |
| --- | --- | --- |
| **AND** | **OR** | **XOR** |
| X AND 0 = 0 | X OR 0 = X | X XOR 0 = X |
| X AND 1 = X | X OR 1 = 1 | X XOR 1 = X' |

Thus,

1. The AND instruction can be used to CLEAR specific Destination bits while preserving the others. A **zero** mask bit **clears** the corresponding Destination bit; a one mask bit preserves the corresponding destination bit.
2. The OR instruction can be used to SET specific Destination bits while preserving the others. A **one** mask bit **sets** the corresponding Destination bit; a zero mask bit preserves the corresponding Destination bit.
3. The XOR instruction can be used to INVERT specific Destination bits while preserving the others. A **one** mask bit **inverts** the corresponding Destination bit; a zero mask bit preserves the corresponding Destination bit.

|  |
| --- |
| ***Example: Clearing bit 2, 4, 6 and 7 of destination using AND operation*** |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Destination | a(7) | a(6) | a(5) | a(4) | a(3) | a(2) | a(1) | a(0) | | Mask | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | | AND | 0 | 0 | a(5) | 0 | a(3) | 0 | a(1) | a(0) | |

|  |
| --- |
| ***Example: Setting bit 7, 6, 5, 3 and 0 of destination using OR operation*** |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Destination | a(7) | a(6) | a(5) | a(4) | a(3) | a(2) | a(1) | a(0) | | Mask | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | | OR | 1 | 1 | 1 | a(4) | 1 | a(2) | a(1) | 1 | |

|  |
| --- |
| ***Example: Toggling bit 7, 2, and 0 of destination using OR operation*** |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Destination | a(7) | a(6) | a(5) | a(4) | a(3) | a(2) | a(1) | a(0) | | Mask | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | | XOR | **~a(7)** | a(6) | a(5) | a(4) | a(3) | **~a(2)** | a(1) | **~a(0)** | |

|  |
| --- |
| ***Example: To clear the 4 low-order bits of BL while leaving the 4 high-order bits unchanged.*** |
| AND BL , 11110000B |

|  |
| --- |
| ***Example: To Set bits 15 and 14 of AX while leaving other bits unchanged.*** |
| AX , 1100000000000000B |

|  |
| --- |
| Example: To Inverts bits 1 and 3 of CL while preserving the others. |
| The instruction XOR CL , 00001010B |

**Changing a letter to its opposite case**

For any alphabetic letter, bit 5 of its ASCII code is **1**; but for the corresponding uppercase letter bit 5 is 0. The remaining bits are similar:

|  |  |  |  |
| --- | --- | --- | --- |
| **Letter** | **ASCII code** | **Letter** | **ASCII code** |
| 'a' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **1** | 0 | 0 | 0 | 0 | 1 | | 'A' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **0** | 0 | 0 | 0 | 0 | 1 | |
| 'b' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **1** | 0 | 0 | 0 | 1 | 0 | | 'B' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **0** | 0 | 0 | 0 | 1 | 0 | |
| 'c' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **1** | 0 | 0 | 0 | 1 | 1 | | 'C' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **0** | 0 | 0 | 0 | 1 | 1 | |
| ... | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **1** | X | X | X | X | X | | ... | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **0** | X | X | X | X | X | |
| 'z' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **1** | 1 | 1 | 0 | 1 | 0 | | 'Z' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | **0** | 1 | 1 | 0 | 1 | 0 | |

Thus a lowercase alphabetic letter can also be converted to uppercase by clearing bit 5 of its ASCII code. This can be done by using an AND instruction with the mask 11011111B or 0DFh.

|  |
| --- |
| ***Example: To convert letter*** |
| |  |  |  |  | | --- | --- | --- | --- | |  | mov |  | DL , 'j' | |  | and |  | DL , 11011111B | |

An uppercase alphabetic letter can also be converted to lowercase by setting bit 5 of its ASCII code. This can be done by using an OR instruction with the mask 00100000B or 20H.

|  |
| --- |
| ***Example: To convert letter*** |
| |  |  |  |  | | --- | --- | --- | --- | |  | mov |  | AL , 'M' | |  | or |  | AL, 20h | |

To convert a lowercase or uppercase letter to its opposite case we need only invert bit 5 of its ASCII code. This can be done by using an XOR instruction with the mask 00100000B.

**Converting an ASCII digit to a Decimal digit and vice versa**

For any ASCII digits, bit 4 and 5 of its ASCII code are **11**; but for the corresponding decimal digit bit 4 and 5 are **00**. The remaining bits are similar:

|  |  |  |  |
| --- | --- | --- | --- |
| **ASCII digit** | **ASCII code** | **Decimal digit** | **Binary code** |
| '0' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **1** | **1** | 0 | 0 | 0 | 0 | | 0 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **0** | **0** | 0 | 0 | 0 | 0 | |
| '1' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **1** | **1** | 0 | 0 | 0 | 1 | | 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **0** | **0** | 0 | 0 | 0 | 1 | |
| '2' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **1** | **1** | 0 | 0 | 1 | 0 | | 2 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **0** | **0** | 0 | 0 | 1 | 0 | |
| x | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **1** | **1** | X | x | x | x | | x | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **0** | **0** | x | x | x | x | |
| '9' | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **1** | **1** | 1 | 0 | 0 | 1 | | 9 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | **0** | **0** | 1 | 0 | 0 | 1 | |

Thus another way of converting an ASCII digit to the corresponding Decimal digit is to use the AND instruction with the mask 00001111B (i.e. 0FH) or with the mask 11001111B (i.e. 0CFH) to clear bits 5 and 6 of the ASCII digit.

|  |
| --- |
| ***Example: To convert ASCII digit 3 into decimal number 3*** |
| |  |  |  |  | | --- | --- | --- | --- | |  | mov |  | BH , '3' | |  | and |  | BH, 0Fh | |

Similarly, another way of converting a Decimal digit to the corresponding ASCII digit is to use the OR instruction with the mask 00110000B (i.e. 30H) to set bits 5 and 6 of the Decimal digit.

|  |
| --- |
| ***Example: To convert decimal number 6 into ASCII digit 6*** |
| |  |  |  |  | | --- | --- | --- | --- | |  | mov |  | BH , 6 | |  | or |  | BH, 30h | |

**Examining selected bits in the Destination Operand**

The Logic Instructions can be used to examine the status of selected bits in the destination operand.

|  |
| --- |
| Example: To check whether bit 2 of AL is set or clear: |
| test AL,00000100b  jz IS\_CLEAR  . . . ; action if bit 2 is set  jmp DONE  IS\_CLEAR: . . . ; action if bit 2 is clear  . . .  DONE: |

|  |
| --- |
| Example: To check whether bits 0, 2, 4 and 5 of DL are all clear: |
| test DL, 00110101B  jz BITS\_CLEAR  . . . ; action if any of bits 0, 2, 4, and 5 is set  jmp DONE  BITS\_CLEAR: ... ; action if each of bits 0, 2, 4, and 5 is clear  DONE: |

|  |
| --- |
| Example: To check if ANY of bits 0, 2, 4 and 5 of DL is clear: |
| push DX  or DL, 11001010B  cmp DL, 11111111B  pop DX  jne ATLEASTONE\_CLEAR  . . .  jmp DONE  ATLEASTONE\_CLEAR: . . .  DONE: |

**Determine whether a general-purpose register is equal to zero**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| The followings are ways to examine whether or not any general-purpose register is equal to zero.   |  |  | | --- | --- | |  | or AL, AL | |  | and EDX, EDX | |  | test ESI, ESI | |

**Clearing a general-purpose register operand or a memory operand to zero**

A register operand can be cleared to zero using any of the instructions: MOV, SUB, AND, and XOR.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| The followings are ways to clear any general-purpose register to zero.   |  |  | | --- | --- | |  | mov BL, 0 | |  | sub AX, AX | |  | and CL, 0 | |  | xor DH, DH | |

A memory operand can be cleared to zero using either the MOV or AND instruction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The followings are ways to clear any memory location to zero.   |  |  | | --- | --- | |  | mov VAR1, 0 | |  | and ARRAY[2], 0 | |

**Bitwise Operations**   
  
There are a whole group of "bitwise" operators that operate on bits.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **C++** | **x86** | **Useful to...** |
| AND | & | and | mask out bits (set other bits to zero) |
| OR | | | or | reassemble bit fields |
| XOR | ^ | xor | invert selected bits |
| NOT | ~ | not | invert all the bits in a number |
| Left shift | << | shl | makes numbers bigger by shifting their bits to higher places |
| Right shift | >> | shr sar | makes numbers smaller by shifting their bits to lower places. sar (arithmetic, signed shift) works for negative numbers. |

If you'd like to see the bits inside a number, you can loop over the bits and use AND to extract each bit:

**int i=9; // 9 == 8 + 1 == 1001**

**for (long bit=31;bit>=0;bit--) { // print each bit**

**long mask=(1L<<bit); // only this bit is set**

**long biti=mask&i; // extract this bit from i**

**if (biti!=0) std::cout<<"1";**

**else std::cout<<"0";**

**if (bit==0) std::cout<<" integer\n";**

**}**

Because binary is almost perfectly unreadable (was that 1000000000000000 or 10000000000000000?), we normally use hexadecimal, base 16.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Decimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **Hex** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F** | **10** |
| Binary | 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 | 10000 |

Remember that every hex digit represents four bits.  So if you shift a hex constant by four bits, it shifts by one entire hex digit:

    0xf0d<<4 == 0xf0d0  
    0xf0d>>4 == 0xf0

If you shift a hex constant by a non-multiple of four bits, you end up interleaving the hex digits of the constant, which is confusing:

   0xf0>>2 == 0x3C (?)

Bitwise operators make perfect sense working with hex digits, because they operate on the underlying bits of those digits:  
    0xff0 & 0x0ff == 0x0f0  
    0xff0 | 0x0ff == 0xfff  
    0xff0 ^ 0x0ff == 0xf0f  
  
You can use these bitwise operators to peel off the hex digits of a number, to print out stuff in hex.

int v=1024+15;  
for (int digit=7;digit>=0;digit--) {  
 char \*digitTable="0123456789abcdef";  
 int d=(v>>(digit\*4))&0xF;  
 std::cout<<digitTable[d];  
}  
std::cout<<std::endl;  
return v;

You could also use printf("%X",v);

**Bitwise Left Shift: <<**

Makes values bigger, by shifting the value's bits into higher places, tacking on zeros in the vacated lower places.

|  |  |
| --- | --- |
| **As Ints** | **As Bits** |
| 3<<0 == 3 | 0011<<0 == 0011 |
| 3<<1 == 6 | 0011<<1 == 0110 |
| 3<<2 == 12 | 0011<<2 == 1100 |

Interesting facts about left shift:

* 1<<n pushes a 1 up into bit number n, creating the bit pattern 1 followed by n zeros.
* The value of (k<<n) is actually k\*2n.  This means bit shifting can be used as a faster multiply by a power of two.
* k<<0 == k, for any k.
* (k<<n) >= k, for any n and k (unless you have "overflow"!).
* On a 32-bit machine, (k<<32) == 0, plus a compiler warning, because all the bits of k have overflowed away.
* Left shift always shifts in fresh new zero bits.
* You can left shift by as many bits as you want.
* You can't left shift by a negative number of bits.

In C++, the << operator is also overloaded for iostream output.  This was a confusing choice, in particular because "cout<<3<<0;" just prints 3, then 0!  To actually print the value of "3<<0", you need parenthesis, like this: "cout<<(3<<0);".  Operator precedence is screwy for bitwise operators, so you really want to use excess parenthesis!  
  
In assembly:

* shl is "shift left".  Use it like "shl eax,4".  Note that the '4' can be a constant, or register cl (low bits of ecx), but not any other register.
* sal is the same instruction (same machine code).
* There's also a "rol" that does a circular left shift: the bits leaving the left side come back in the right side.

**Bitwise Right Shift: >>**

Makes values smaller, by shifting them into lower-valued places.  Note the bits in the lowest places just "fall off the end" and vanish.

|  |  |
| --- | --- |
| **As Ints** | **As Bits** |
| 3>>0 == 3 | 0011>>0 == 0011 |
| 3>>1 == 1 | 0011>>1 == 0001 |
| 3>>2 == 0 | 0011>>2 == 0000 |
| 6>>1 == 3 | 0110>>1 == 0011 |

Interesting facts about right shift:

* The value of (k>>n) is actually k/2n.  This can be used as a faster divide.
* (k<<n)>>n == k, unless overflow has happened.
* On a 32-bit machine, (k>>32) == 0, plus a compiler warning, because all the bits of k have fallen off the end.
* There are two flavors of right shift: signed, and unsigned.  Unsigned shift fills in the new bits with zeros.  Signed shift fills the new bits with copies of the sign bit, so negative numbers stay negative after the shift.

If you're dyslexic, like me, the left shift << and right shift >> can be really tricky to tell apart.  I always remember it like this:

* k<<n  pumps up the value of k (the point of the << is injecting bigness into k)
* k>>n  drains away the value of k (the point of the >> is draining bigness from k)

In assembly:

* shr is the unsigned shift.
* sar is the signed (or "arithmetic") shift.
* Again, there's a circular right shift "ror".

**Bitwise AND: &**

Output bits are 1 only if both corresponding input bits are 1.  This is useful to "mask out" bits you don't want, by ANDing them with zero.

|  |  |
| --- | --- |
| **As Ints** | **As Bits** |
| 3&5 == 1 | 0011&0101 == 0001 |
| 3&6 == 2 | 0011&0110 == 0010 |
| 3&4 == 0 | 0011&0100 == 0000 |

Properties:

* 0=A&0     (AND by 0's creates 0's--used for masking)
* A=A&~0 (AND by 1's has no effect)
* A=A&A    (AND by yourself has no effect)

Bitwise AND is a really really useful tool for extracting bits from a number--you often create a "mask" value with 1's marking the bits you want, and AND by the mask.  For example, this code figures out if bit 2 of an integer is set:  
    int mask=(1<<2); // in binary: 100  
    int value=...;           // in binary: xyz  
    if (0!=(mask&value))  // in binary: x00  
       ...  
  
In C/C++, bitwise AND has the wrong precedence--leaving out the parenthesis in the comparison above gives the wrong answer!  Be sure to use extra parenthesis!  
  
In assembly, it's the "and" instruction.  Very simple!

**Bitwise OR: |**

Output bits are 1 if either input bit is 1.  E.g., 3|5 == 7; or 011 | 101 == 111.

|  |  |
| --- | --- |
| **As Ints** | **As Bits** |
| 3|0 == 3 | 0011|0000 == 0011 |
| 3|3 == 3 | 0011|0011 == 0011 |
| 1|4 == 5 | 0001|0100 == 0101 |

* A=A|0      (OR by 0's has no effect)
* ~0=A|~0 (OR by 1's creates 1's)
* A=A|A      (OR by yourself has no effect)

Bitwise OR is useful for sticking together bit fields you've prepared separately.  Overall, you use AND to pick apart an integer's values, XOR and NOT to manipulate them, and finally OR to assemble them back together.

**Bitwise XOR: ^**

Output bits are 1 if either input bit is 1, but not both. E.g., 3^5 == 6; or 011 ^ 101 == 110.  Note how the low bit is 0, because both input bits are 1.

|  |  |
| --- | --- |
| **As Ints** | **As Bits** |
| 3^5 == 6 | 0011&0101 == 0110 |
| 3^6 == 5 | 0011&0110 == 0101 |
| 3^4 == 7 | 0011&0100 == 0111 |

* A=A^0  (XOR by zeros has no effect)
* ~A = A ^ ~0  (XOR by 1's inverts all the bits)
* 0=A^A  (XOR by yourself creates 0's--used in cryptography)

The second property, that XOR by 1 inverts the value, is useful for flipping a set of bits.  Generally, XOR is used for equality testing (a^b!=0 means a!=b), controlled bitwise inversion, and crypto.

**Bitwise NOT: ~**

Output bits are 1 if the corresponding input bit is zero.  E.g., ~011 == 111....111100.  (The number of leading ones depends on the size of the machine's "int".)

|  |  |
| --- | --- |
| **As Ints** | **As Bits** |
| ~0 == big value | ~...0000 == ...1111 |

I don't use bitwise NOT very often, but it's handy for making an integer whose bits are all 1: ~0 is all-ones.

**Non-bitwise Logical Operators**

Note that the logical operators &&, ||, and ! work exactly the same as the bitwise values, but for exactly one bit.  Internally, these operators map multi-bit values to a single bit by treating zero as a zero bit, and nonzero values as a one bit.  So   
    (2&&4) == 1 (because both 2 and 4 are nonzero)  
     (2&4) == 0 (because 2==0010 and 4 == 0100 don't have any overlapping one bits).

**Shift Instruction**

**Types of Shift Instructions**

There are three types of shift instructions

|  |
| --- |
| ***Logical shift instructions*** |
| |  |  | | --- | --- | |  | SHL (SHift Left) | |  | SHR (SHift Right) |   Another interpretation: Logical shift instructions work on *unsigned binary numbers* |

|  |
| --- |
| ***Arithmetic shift instructions*** |
| |  |  | | --- | --- | |  | SAL (Shift Arithmetic Left) | |  | SAR (Shift Arithmetic Right) |   Another interpretation: Arithmetic shift instructions work on *signed binary numbers* |

|  |
| --- |
| ***Double Precision Shift instructions*** |
| |  |  | | --- | --- | |  | SHLD (Double Precision Shift Left) | |  | SHRD (Double Precision Shift Right) | |

**General syntax format**

There are two different formats for logical and arithmetic shift instructions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | SHL/SHR/SAL/SAR r/m, count | |  | SHL/SHR/SAL/SAR r/m, CL |   **count** is an immediate value between 0 and 31. If a greater value is specified, Pentium takes only the least significant 5 bits as the count value (MODULUS 32). |

Second format specifies count indirectly through CL

|  |  |
| --- | --- |
|  | Only CL register can be used. |
|  | CL contents are not changed. |
|  | Useful if count value is known only at the run time as opposed at assembly time. |

**EFFECT of SHIFT INSTRUCTIONS on FLAGS**

All shift instructions affect some flags like other instructions. However, the way Carry Flag (CF) and Overflow Flag (OF) affected are different.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Carry Flag (CF)** | **Overflow Flag (OF)** | **Zero Flag (ZF)** | **Sign Flag (SF)** | **Parity Flag(PF)** | **Auxiliary Flag (AF)** |
| **Yes** | **Yes** | Yes | Yes | Yes | NO |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | Auxiliary flag (AF): undefined. | |  | Zero flag (ZF), Sign flag (SF) and parity flag (PF)are updated to reflect the result. | |  | Carry flag (CF): Contains the last bit shifted out | |  | Overflow flag (OF)   |  |  | | --- | --- | |  | For multibit shifts : Undefined | |  | For single bit shifts: OF is set if the leftmost bit has *changed* as a result of the shift, otherwise cleared. | | |

**Logical Shift Left Instruction (SHL)**

Semantic:

|  |  |  |  |
| --- | --- | --- | --- |
|  | SHL shifts the leftmost bit into the Carry Flag (CF) and overwrites the current value of the Carry Flag. | | |
|  | Each of the remaining bits is shifted leftwise and 0 is shifted into the rightmost bit. | | |
| Example: Shifting AL=3 leftwise by 1 bit | | |
| mov AL, 3 | |  |
| shl AL, 1 ; shift left AL 1 place | |  |

|  |  |
| --- | --- |
| S O L U T I O N | |
| |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Initially AL = 3 = | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | | |  |
| |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Finally AL = 6 = | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | |  |
| CF = 0 |  |
| SF = 0 |  |
| ZF = 0 because the result is not equal to 0. |  |
| PF = 1 because there are 2 bits 1 in AL. |  |
| OF = 0 Why? |  |

|  |
| --- |
| Shifting left a Destination operand by N bits multiplies it by 2N. (signed and unsigned) |

|  |  |
| --- | --- |
| Example: Shifting BL=3Fh leftwise by 3 bits | |
| mov BL, 3Fh |  |
| shl BL, 3 ;shift left BL by 3 bit |  |

|  |  |
| --- | --- |
| S O L U T I O N | |
| |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | BL = F8h=248 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | | |  |
| CF = 1 |  |
| SF = 1 |  |
| ZF = 0 because the result is not equal to 0. |  |
| PF = 0 because there are odd numbers of bit 1 in BL. |  |
| OF is **undefined** |  |

|  |  |
| --- | --- |
| Example: Shifting leftwise AL by CL bit | |
| mov AX, 8F23h ; now AX=36643 (unsigned) or -28893 (signed) |  |
| mov CX, 1 |  |
| shl AX, CL ; shift left AX 1 place |  |

|  |  |
| --- | --- |
| S O L U T I O N | |
| AX = 1E46h= 7750 |  |
| CF = 1 |  |
| SF = 0 |  |
| ZF = 0 |  |
| PF = 0 because there are 3 bits 1 in **AL**. |  |
| OF = 1 because the leftmost bit of AX has *changed* from 1 to 0. |  |

Unsigned Multiplication by a non-multiple of 2 can be achieved by a combination of SHL, MOV, ADD, or SUB instructions. Such multiplications are usually more efficient than multiplications using the MUL (Multiplication) instruction.

|  |  |
| --- | --- |
| Example: Multiply AX by 19 | |
| ; 19 = 16 + 2 + 1 |  |
| push BX |  |
| mov BX , AX |  |
| shl BX , 1 ; BX := AX\*2 |  |
| add BX , AX ; BX := AX \* 3 |  |
| shl AX , 4 ; AX := AX \* 16 |  |
| add AX , BX ; AX := AX \* 19 |  |
| pop BX |  |

|  |  |
| --- | --- |
| Example: Multiply AX by 15 | |
| ; 15 = 16 - 1 |  |
| push BX |  |
| mov BX , AX |  |
| shl AX , 4 ; AX := AX\*16 |  |
| sub AX , BX ; AX := 16\*AX - AX |  |
| pop BX |  |

Due to the possibility of having discarded bits that may yield incorect result, however, the usage of shift instructions for unsigned multiplication must be examined carefully.

|  |  |
| --- | --- |
| Example: Multiply AX by 15 Accurately | |
| ; The trick is to use 32-bit register |  |
| push EBX |  |
| movzx EAX , AX ; extend AX into EAX |  |
| mov EBX , EAX |  |
| shl EAX , 4 ; EAX := EAX\*16 |  |
| sub EAX , EBX ; EAX := 16\*EAX - EAX |  |
| pop EBX |  |

For multibit left-shifts the Carry Flag (CF) may not accurately reflect whether or not an unsigned out-of-range condition has occurred.

|  |  |
| --- | --- |
| Example | |
| mov BL , 80H ; BL := 10000000b |  |
| shl BL , 2 |  |

The result 00000000B in BL is incorrect although the Carry Flag (CF) is cleared.

**Logical Shift Right Instruction (SHR)**

SHR shifts the rightmost bit of operand into the Carry Flag; the current value of the Carry Flag is lost. Each of the remaining bits is shifted rightwise and 0 is shifted into the leftmost bit of operand.

SHR does not preserve the sign of a negative operand. Thus, SHR cannot be used to perform division on negative numbers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | Shifting right a Destination operand by N bits divides it by 2N (only unsigned). | |  | If the value of the Destination is odd, the division is approximate. | |

|  |  |
| --- | --- |
| Example | |
| mov BL , 00000101B ; BL := 5 |  |
| shr BL , 1 ; BL := 2 |  |

**Shift arithmetic Left (SAL)**

SHL is equivalent to SAL.

**Shift arithmetic Right (SAR)**

SAR shifts the rightmost bit of operand into the Carry Flag; the current value of the Carry Flag is lost. Each of the remaining bits is shifted rightwise and, in addition, the leftmost bit is shifted into itself.

**Double Precision Shift Instructions**

|  |
| --- |
| ***The format of Double Shift instructions*** |
| |  |  | | --- | --- | |  | SHLD/SHRD r/m16, r16, imm8 | |  | SHLD/SHRD r/m32, r32, imm8 | |  | SHLD/SHRD r/m16, r16, CL | |  | SHLD/SHRD r/m32, r32, CL | |

All double precision shift instructions DO NOT MODIFY the second operand. The second operand only feeds their bits to the first operand.