University of Central Punjab

(Incorporated by Ordinance No. XXIV of 2002 promulgated by Government of the Punjab)

FACULTY OF INFORMATION TECHNOLOGY

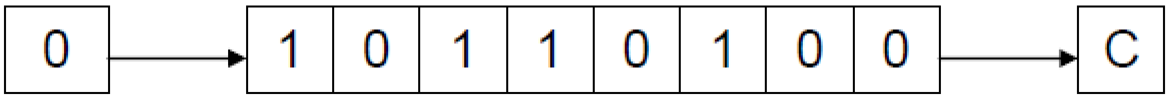
Computer Organization and Assembly Language

Lab 06	
Topic	▪ Arithmetic & Logical instructions ▪ Selective bit setting/clearing/complimenting ▪ Shifting and Rotations variations ▪ Extended addition and subtraction

Part 1

Shift Logical Right (SHR)

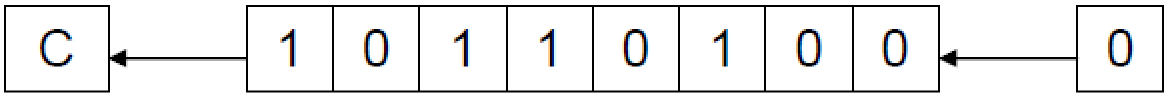
The shift logical right operation inserts a zero from the left and moves every bit one position to the right and copies the rightmost bit in the carry flag. Imagine that there is a pipe filled to capacity with eight balls. The pipe is open from both ends and there is a basket at the right end to hold anything dropping from there. The operation of shift logical right is to force a white ball from the left end. The operation is depicted in the following illustration.



White balls represent zero bits while black balls represent one bits. Sixteen bit shifting is done the same way with a pipe of double capacity.

Shift Logical Left (SHL) / Shift Arithmetic Left (SAL)

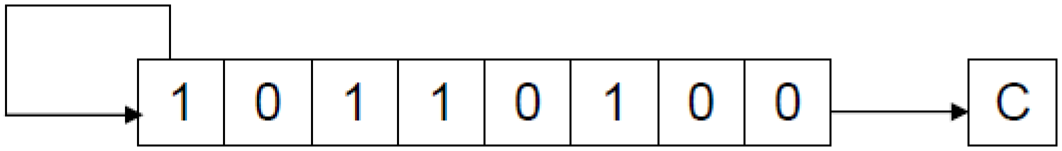
The shift logical left operation is the exact opposite of shift logical right. In this operation the zero bit is inserted from the right and every bit moves one position to its left with the most significant bit dropping into the carry flag. Shift arithmetic left is just another name for shift logical left. The operation is again exemplified with the following illustration of ball and pipes.



Shift Arithmetic Right (SAR)

A signed number holds the sign in its most significant bit. If this bit was one a logical right shifting will change the sign of this number because of insertion of a zero from the left. The sign of a signed number should not change because of shifting.

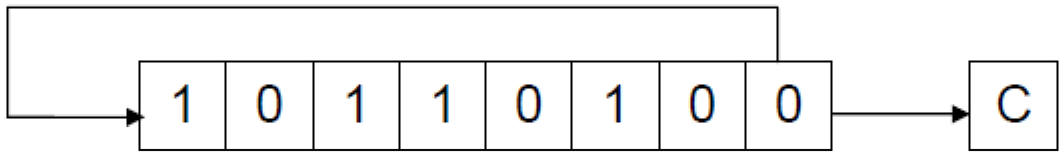
The operation of shift arithmetic right is therefore to shift every bit one place to the right with a copy of the most significant bit left at the most significant place. The bit dropped from the right is caught in the carry basket. The sign bit is retained in this operation. The operation is further illustrated below.



The left shifting operation is basically multiplication by 2 while the right shifting operation is division by two. However for signed numbers division by two can be accomplished by using shift arithmetic right and not shift logical right. The left shift operation is equivalent to multiplication except when an important bit is dropped from the left. The overflow flag will signal this condition if it occurs and can be checked with JO. For division by 2 of a signed number logical right shifting will give a wrong answer for a negative number as the zero inserted from the left will change its sign. To retain the sign flag and still effectively divide by two the shift arithmetic right instruction must be used on signed numbers.

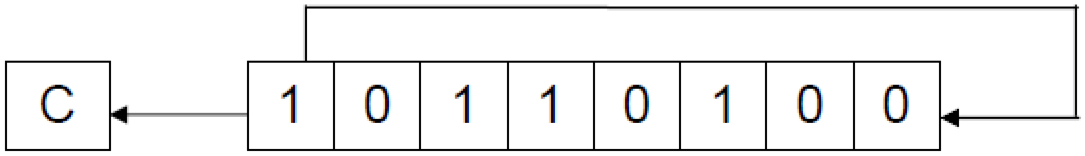
Rotate Right (ROR)

In the rotate right operation every bit moves one position to the right and the bit dropped from the right is inserted at the left. This bit is also copied into the carry flag. The operation can be understood by imagining that the pipe used for shifting has been molded such that both ends coincide. Now when the first ball is forced to move forward, every ball moves one step forward with the last ball entering the pipe from its other end occupying the first ball's old position. The carry basket takes a snapshot of this ball leaving one end of the pipe and entering from the other.



Rotate Left (ROL)

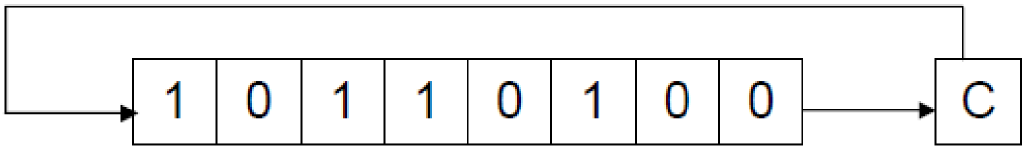
In the operation of rotate left instruction, the most significant bit is copied to the carry flag and is inserted from the right, causing every bit to move one position to the left. It is the reverse of the rotate right instruction. Rotation can be of eight or sixteen bits. The following illustration will make the concept clear using the same pipe and balls example.



Rotate Through Carry Right (RCR)

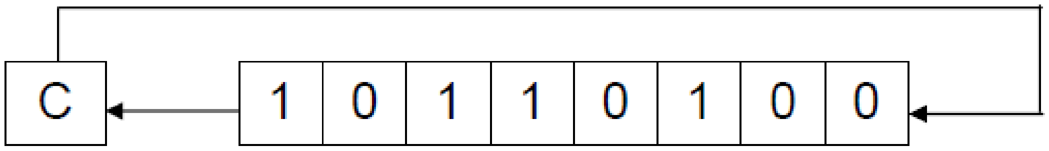
In the rotate through carry right instruction, the carry flag is inserted from the left, every bit moves one position to the right, and the right most bit is dropped in the carry flag. Effectively this is a nine bit or a seventeen bit rotation instead of the eight or sixteen bit rotation as in the case of simple rotations.

Imagine the circular molded pipe as used in the simple rotations but this time the carry position is part of the circle between the two ends of the pipe. Pushing the carry ball from the left causes every ball to move one step to its right and the right most bit occupying the carry place. The idea is further illustrated below.



Rotate Through Carry Left (RCL)

The exact opposite of rotate through carry right instruction is the rotate through carry left instruction. In its operation the carry flag is inserted from the right causing every bit to move one location to its left and the most significant bit occupying the carry flag. The concept is illustrated below in the same manner as in the last example.



AND operation

AND performs the logical bitwise *and* of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the bit is cleared as shown in the truth table. Examples are “and ax, bx” and “and byte [mem], 5.” All possibilities that are legal for addition are also legal for the AND operation. The different thing is the bitwise behavior of this operation.

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

OR operation

OR performs the logical bitwise "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set otherwise the result bit is cleared as shown in the truth table. Examples are "or ax, bx" and "or byte [mem], 5."

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR operation

XOR (Exclusive Or) performs the logical bitwise "exclusive or" of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared) otherwise the result bit is cleared as shown in the truth table. XOR is a very important operation due to the property that it is a reversible operation. It is used in many cryptography algorithms, image processing, and in drawing operations. Examples are "xor ax, bx" and "xor byte [mem], 5."

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT operation

NOT inverts the bits (forms the one's complement) of the byte or word operand. Unlike the other logical operations, this is a single operand instruction, and is not purely a logical operation in the sense the others are, but it is still traditionally counted in the same set. Examples are "not ax" and "not byte [mem]".

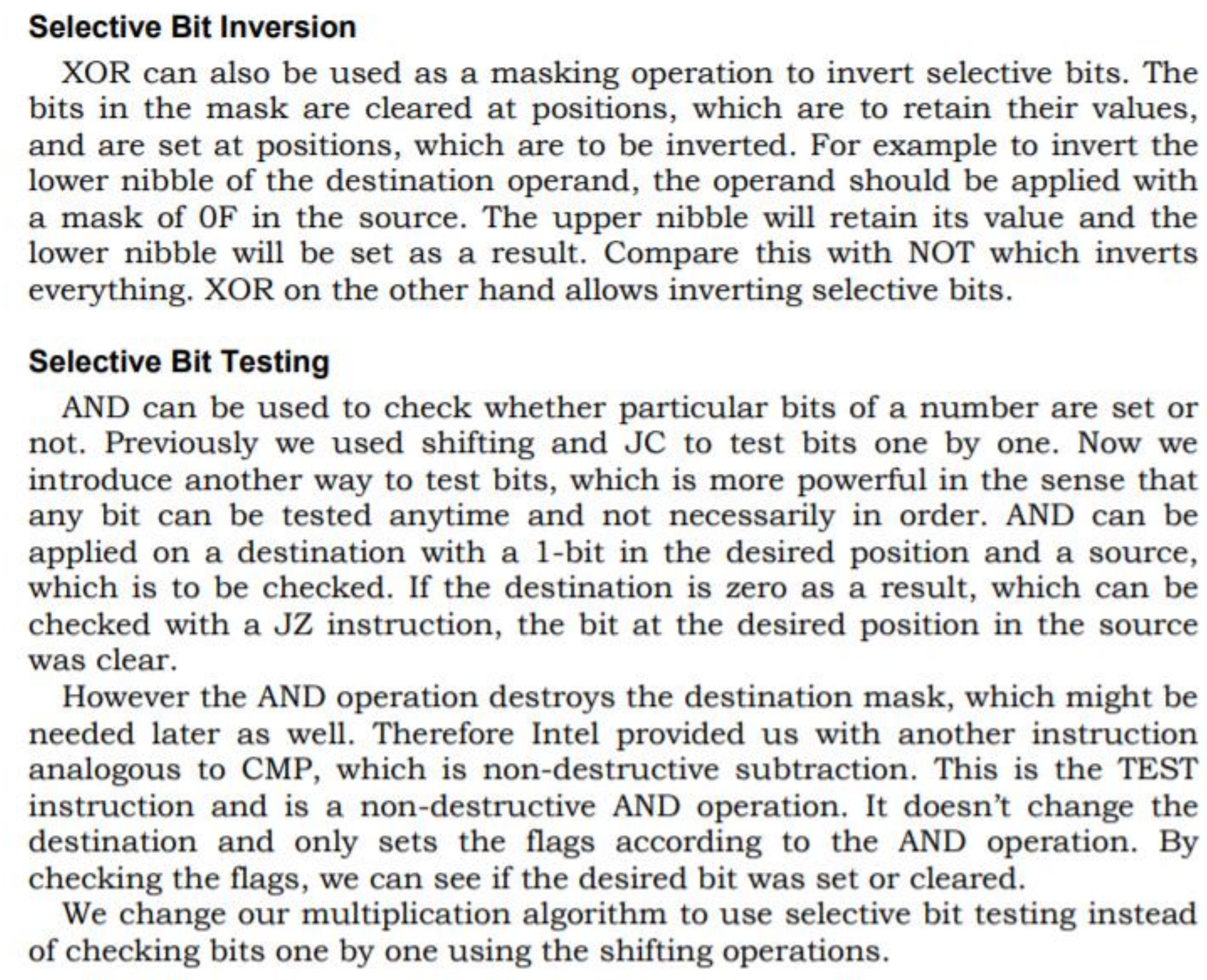
4.6. MASKING OPERATIONS

Selective Bit Clearing

Another use of AND is to make selective bits zero in its destination operand. The source operand is loaded with a mask containing one at positions which are retain their old value and zero at positions which are to be zeroed. The effect of applying this operation on the destination with mask in the source is to clear the desired bits. This operation is called masking. For example if the lower nibble is to be cleared then the operation can be applied with F0 in the source. The upper nibble will retain its old value and the lower nibble will be cleared.

Selective Bit Setting

The OR operation can be used as a masking operation to set selective bits. The bits in the mask are cleared at positions which are to retain their values, and are set at positions which are to be set. For example to set the lower nibble of the destination operand, the operation should be applied with a mask of 0F in the source. The upper nibble will retain its value and the lower nibble will be set as a result.



Selective Bit Inversion

XOR can also be used as a masking operation to invert selective bits. The bits in the mask are cleared at positions, which are to retain their values, and are set at positions, which are to be inverted. For example to invert the lower nibble of the destination operand, the operand should be applied with a mask of 0F in the source. The upper nibble will retain its value and the lower nibble will be set as a result. Compare this with NOT which inverts everything. XOR on the other hand allows inverting selective bits.

Selective Bit Testing

AND can be used to check whether particular bits of a number are set or not. Previously we used shifting and JC to test bits one by one. Now we introduce another way to test bits, which is more powerful in the sense that any bit can be tested anytime and not necessarily in order. AND can be applied on a destination with a 1-bit in the desired position and a source, which is to be checked. If the destination is zero as a result, which can be checked with a JZ instruction, the bit at the desired position in the source was clear.

However the AND operation destroys the destination mask, which might be needed later as well. Therefore Intel provided us with another instruction analogous to CMP, which is non-destructive subtraction. This is the TEST instruction and is a non-destructive AND operation. It doesn't change the destination and only sets the flags according to the AND operation. By checking the flags, we can see if the desired bit was set or cleared.

We change our multiplication algorithm to use selective bit testing instead of checking bits one by one using the shifting operations.

Example 1:

Let the binary of a number (0XABCD) is 10101011 11001101.

a) **Set the fourth bit.**

```
mov ax,0xABCD
or ax,0000000000010000b
mov ax,0x4c00
int 21h
```

b) Clear the L.S.B.

```
mov ax,0xABCD
and ax,1111111111111110b
mov ax,0x4c00
int 21h
```

c) Invert the M.S.B.

```
mov ax,0xABCD
xor ax,1000000000000000b
mov ax,0x4c00
int 21h
```

d) Set the 4th ,7th and 13th bit.

```
mov ax,0xABCD
or ax,0010000010010000b
mov ax,0x4c00
int 21h
```

Note: logical operations are bitwise operations.

Example 2:

Multiply the number by 8 using shift operator.
Let the number is 7.

```
mov al,7
shl al,3
```

Example 3:

Rotate right 4 times the value in register bx.
Let BX=0xEFCD

```
mov bx,0xEFCD
ror bx,4
```

Example 4(Extended addition)

```
MOV AX, [Num1] ;loads two bytes into AX register, AX=FFFF
MOV BX, [Num1+2] ;loads Next two bytes into AX register, AX=0001
ADD AX, [Num2] ; adds into AX; AX=AX+0002;
ADC BX, [Num2+2]; Add with carry instruction
MOV [SUM],AX ; Move the lower bits into Sum variable
MOV [SUM+2],BX  ; Move the higher bits into Sum variable higher bits
mov ax,0x4c00
int 21h
 Num1: dd 0x0001FFFF
 Num2: dd 0x00010002
 SUM: dd 0
```

Example 5(Extended subtraction)

```
MOV AX,[Num2] ;loads two bytes into AX register, AX=0002
MOV BX, [Num2+2] ;loads Next two bytes into AX register, AX=0001
SUB AX, [Num1] ; sub into AX; AX=AX-FFFF;
SBB BX, [Num1+2]; Subtraction with borrow.
MOV [SUM],AX ; Move the lower bits into Sum variable
MOV [SUM+2],BX  ; Move the higher bits into Sum variable higher bits
mov ax,0x4c00
int 21h
Num1: dd 0x0001FFFF
 Num2: dd 0x00010002
 SUM: dd 0
```

```
a = 10, b = 20, c = 5 , sum=0;

        if (a < =b)
        {
         // L1
                if (a < =c)
                {
                   // L2
                        if (b>c)
                        {
                           // L3
                                sum = a + b + c;
                        }
                        else
                        {
                           // L4
                                sum = a - b - c;
                        }
                }
                else
                {
                   // L5
                        sum = a + b - c;
                }
        }
        else
        {
          // L6
                sum = a - b + c;
        }
```

```
[org 0x100]
mov al,[a]
mov bl,[b]
mov cl,[c]

cmp al,bl
jng l1
jg  l6
l1:
   cmp al,cl
   jng l2
   jg  l5
l2:
   cmp bl,cl
   jg  l3
   jng l4
l3:
   mov [sum],al
   add [sum],bl
   add [sum],cl
   jmp exit
l4:
   mov [sum],al
   sub [sum],bl
   sub [sum],cl
   jmp exit
l5:
   mov [sum],al
   add [sum],bl
   sub [sum],cl
   jmp exit
l6:
   mov [sum],al
   sub [sum],bl
   add [sum],cl
exit:
mov ax,0x4c00
int 21h

a: db 10
b: db 20
c: db 5
sum: db 0
```

1. Write a program to swap every pair of bits in the AX register.

2. Give the value of the AX register and the carry flag after each of the following instructions.

```
xor ah, al
mov cl, 4
shr al, cl
rcr ah, cl
```

3. Write a program to swap the nibbles in each byte of the AX register.

4. Calculate the number of one bits in BX and complement an equal number of least significant bits in AX.

HINT: Use the XOR instruction

5. Write a program to multiply two 32bit numbers and store the answer in a 64bit location

6. Write a program that will first invert the 9th, 10th and 4th bit of Ax register.

Now Set the 8th, 12th and 3rd bit of Ax.

7. Write a program that will multiply the AX with 8.

Now after multiply you need to Shift right the Ax by the value of num1 variable.

Explanation: for shifting you need to do this one by one in a loop. If the value of num1 is 5 you need to shift it by 5 time but in a loop one by one.

Hint: SHR AX,1

Note: Suppose Ax have 0x150F value on start of your program. You also need to use loop so that you can make the code dynamic depending upon the num1 variable.

8. Write code for the following task.

```
a = 5, b = 10, c = 20 , sum=0;

            if (a < b)
            {
              // L1
                        if (a < c)
                        {
                            // L2
                                    if (b>=c)
                                    {
                                       // L3
                                                sum = a - b + a;
                                    }
                                    else
                                    {
                                       // L4
                                                sum = a + b - b;
                                    }
                        }
                        else
                        {
                            // L5
                                    sum = a - b + c;
                        }
            }
            else
            {
              // L6
                        sum = a + b - c;
            }
```