Bit Manipulation (contd). Bitwise Logical Operations

Extended Addition

- ADC, Addition with Carry will help us perform extended addition.
- How ADC works
  - ADC ax, bx ; this is equivalent to ax+bx+CF

```
[org 0x0100]

    stc
    mov al, 3
    adc al, 0 ; al=4

    clc
    mov al, 3
    adc al, 0 ; al=3

    mov ax, 0x4c00
    int 0x21
```

Extended Addition

- For adding numbers with n words
- Add nth words of first and second operand
- For i in n-1 to 1
  - add with carry the ith word of first and ith word of second operand

Extended Addition (Example)

Example: Adding 64 bits

Initially:

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000

Extended Addition (Example)

Step 1

```
mov ax, [num1]
mov bx, [num2]
Add ax, bx
mov [result], ax
```

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000	**0111 1111 1111 1111**

CF=1

Extended Addition (Example)

Step 2

```
mov ax, [num1+2]
mov bx, [num2+2]
ADC ax, bx
mov [result+2], ax
```

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	0000 0000 0000 0000	0000 0000 0000 0000	**1000 0000 0000 0010**	0111 1111 1111 1111

CF=0

Extended Addition (Example)

Step 3

```
mov ax, [num1+4]
mov bx, [num2+4]
ADC ax, bx
mov [result+4], ax
```

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	0000 0000 0000 0000	**0110 0000 1111 1111**	1000 0000 0000 0010	0111 1111 1111 1111

CF=1

Extended Addition (Example)

Step 4

```
mov ax, [num1+6]
mov bx, [num2+6]
ADC ax, bx
mov [result+6], ax
```

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	**0001 0111 0001 0000**	0110 0000 1111 1111	1000 0000 0000 0010	0111 1111 1111 1111

CF=1

Extended Addition (Example)

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	0001 0111 0001 0000	0110 0000 1111 1111	1000 0000 0000 0010	0111 1111 1111 1111

So the result of addition is
CF=1 , 0001 0111 0001 0000 0110 0000 1111 1111 1000 0000 0000 0010 0111 1111 1111 1111
Note that the carry of the most significant word is stored in CF after these 4 steps

Extended Subtraction

- For subtraction the same logic will be used and just like addition with carry, there is an instruction to subtract with borrow called SBB.
- SBB ax, bx; this is equivalent to ax-bx-CF

```
stc
mov al, 3
sbb al, 0 ; al=2

clc
mov al, 3
sbb al, 0 ; al=3
```

Extended Subtraction

- For subtracting numbers with n words
- Sub nth words of first and second operand
- For i in n-1 to 1
    - Subtract with borrow the ith word of first and ith word of second operand

Extended Subtraction (Exercise)

Example: subtracting 64 bits

Initially

num1:	1000 1000 0000 0000	1110 0000 1111 1111	0100 0000 0000 0000	1111 1111 1111 1111
num2:	1000 1111 0000 1111	1000 0000 0000 0000	0100 0000 0000 0001	1000 0000 0000 0000
result:	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000

- Work out the given subtraction as we did addition, show each step

Question

- Given two operands of n words, if you know the value of n, write the code of adding these two operands in loop
- Given two operands of n words, if you know the value of n, write the code of subtracting these two operands in loop

Extended Multiplication

- Previously we saw an example to multiply 4 bit numbers.
- The same algorithm can be now used to multiply numbers of any size.
- The algorithm was as follows

```
Shift the multiplier to the right (extended shift)
If CF=1
    add the multiplicand to the result (extended addition)
Shift the multiplicand to the left (extended shift)
Repeat the algorithm n times (where n is size of multiplier)
```

Example

Example 4.2

```
; 16bit multiplication
[org 0x0100]
              jmp  start

multiplicand: dd   1300              ; 16bit multiplicand 32bit space
multiplier:   dw   500               ; 16bit multiplier
result:       dd   0                 ; 32bit result

start:        mov  cl, 16            ; initialize bit count to 16
              mov  dx, [multiplier]  ; load multiplier in dx

checkbit:     shr  dx, 1             ; move right most bit in carry
              jnc  skip              ; skip addition if bit is zero

              mov  ax, [multiplicand]
              add  [result], ax      ; add less significant word
              mov  ax, [multiplicand+2]
              adc  [result+2], ax    ; add more significant word

skip:         shl  word [multiplicand], 1
              rcl  word [multiplicand+2], 1 ; shift multiplicand left
              dec  cl                ; decrement bit count
              jnz  checkbit          ; repeat if bits left

              mov  ax, 0x4c00        ; terminate program
              int  0x21
```

Example

- Change the code give in previous slide to work for 32 bit multiplication i.e. the result should be 64 bit

Bitwise Logical Operations

Bitwise Logical Operations

- AND operation
  - 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.
- OR operation
  - Examples are “or ax, bx” and “or byte [mem], 5.”
- XOR operation
  - Examples are “xor ax, bx” and “xor byte [mem], 5
- NOT operation
  - Examples are “not ax” and “not byte [mem]”.

Masking Operations (1)

- Selective Bit Clearing
  - Done using AND operations
  - Example - clear the LSB in AL

    ```
    AND AL, 11111110b
    ```
  - This operation is called masking, 11111110 was mask in this example
- Selective Bit Setting
  - Done using OR operations
  - Example - set the LSB in AL

    ```
    OR AL, 00000001b
    ```
  - 00000001b is the mask here

Masking Operations (2)

- Selective Bit Inversion
  - Done using XOR
  - For example - toggle LSB and MSB in AL
    ```
    XOR AL, 1000 0001b
    ```

Masking Operations

- Selective Bit Testing
  - AND operation can be used to test if a certain bit in a number is ON
    - But this will change the the operand
  - TEST instruction is a non destructive alternative for selective bit testing.
  - It doesn’t change the destination and only sets the sign, zero and parity as would have AND operation

```
mov al, 00010001b
test al,00001001b; ZF=0
test al,00010000b; ZF=0
test al,00000010b; ZF=1
```

  - Next slide shows the use of test in multiplication algorithms so that multiplier is retained

Example 4.3

```
; 16bit multiplication using test for bit testing
[org 0x0100]
                jmp  start

multiplicand: dd   1300                  ; 16bit multiplicand 32bit space
multiplier:   dw   500                   ; 16bit multiplier
result:       dd   0                     ; 32bit result

start:        mov  cl, 16                ; initialize bit count to 16
              mov  bx, 1                 ; initialize bit mask

checkbit:     test bx, [multiplier]      ; test right most bit
              jz   skip                  ; skip addition if bit is zero

              mov  ax, [multiplicand]
              add  [result], ax          ; add less significant word
              mov  ax, [multiplicand+2]
              adc  [result+2], ax        ; add more significant word

skip:         shl  word [multiplicand], 1
              rcl  word [multiplicand+2], 1 ; shift multiplicand left
              shl  bx, 1                 ; shift mask towards next bit
              dec  cl                    ; decrement bit count
              jnz  checkbit              ; repeat if bits left

              mov  ax, 0x4c00            ; terminate program
              int  0x21
```

Reading

- Chapter 4 BH