A linguagem Assembly a ser utilizada depende essencialmente do processador ao qual se destina, ou seja, é dependente da arquitetura da CPU. A seguir estão algumas das instruções mais utilizadas para processadores da família Intel. Para que possam ser entendidas perfeitamente, é preciso ter uma boa noção da arquitetura do microprocessador. Caso tenha dúvidas, leia antes "[Arquitetura Intel](http://www.numaboa.com.br/informatica/tutos/assembly/1123-arquitetura-intel)".

**Instrução AND**

|  |  |
| --- | --- |
| **AND** | |
| **Sintaxe** | and destino,fonte |
| **Lógica** | destino <- destino **AND** fonte |
| **Descrição** | AND realiza uma operação lógica AND bit a bit nos seus operandos e põe o resultado no destino. |
| **Flags** | CF <- 0 OF <- 0 CF OF PF SF ZF (AF indefinido) |

Há 5 modos diferentes de se **AND**ar dois números:

1. AND dois registradores
2. AND um registrador com uma variável
3. AND uma variável com um registrador
4. AND um registrador com uma constante
5. AND uma constante com um registrador

Ou seja:

variável1 db ? variável2 dw ? and cl, dh and al, variável1 and variável2, si and dl, 0C2h and variável1, 01001011b

Observe que as constantes estão em notação hexadecimal e binária, as únicas aceitas por que são o único meio de expressar números bit a bit. É claro que a notação hexadecimal precisa ser convertida em 4 dígitos binários. AND retorna **1 quando ambos os operandos forem 1**, senão retorna zero, conforme a tabela abaixo:

|  |  |  |
| --- | --- | --- |
| **Operando 1** | **Operando 2** | **Resultado** |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Pode-se verificar se um registrador está zerado utilizando a instrução AND, como em **and ecx,ecx**. Caso algum bit em ecx estiver setado (valor 1), este mesmo bit estará setado no resultado e a flag zero (ZF) estará zerada (valor falso). Se não houver bits setados, o resultado também não terá bits setados e a flag zero (ZF) recebe o valor 1. Nenhum bit será alterado e ecx mantém o seu valor original. Este é o modo padrão de se checar valores zerados. Uma alternativa para este teste é usando a instrução TEST.

Também é possível testar uma variável com uma constante (fazer AND em duas variáveis dá erro!). No caso de **and variável1, 11111111b** testa-se todos os bits da variável1 com bits setados. Se algum bit da variável1 estiver setado, aparece setado no resultado e ZF = 0; se todos estiverem zerados, continuam zerados no resultado e ZF = 1. O valor da variável1, em ambos os casos, não é alterado.

AND também é utilizado em máscaras. Caso se queira testar o bit na posição 0 do registrador ecx, podemos utilizar a instrução **and ecx, 00000001b**. O estado deste bit (setado ou zerado) será transportado para o resultado enquanto todos os outros serão zerados.

**Instrução CMP**

|  |  |
| --- | --- |
| **CMP** | |
| **Sintaxe** | CMP destino, fonte |
| **Flags** | AF CF OF PF SF ZF |
| **Descrição** | Subtrai a fonte do destino, atualiza as flags porém não armazena o resultado. |

A fonte pode ser um registrador, um endereço de memória ou um valor. O destino pode ser um registrador ou um endereço de memória. Exemplos:

cmp eax, variável1 cmp variável2, TRUE

**Instrução DEC**

|  |
| --- |
| **DEC** |
| Decrementa o valor de um registrador ou de uma variável em 1. |

Exemplos:

dec eax dec variável1

**Instrução INC**

|  |
| --- |
| **INC** |
| Incrementa o valor de um registrador ou de uma variável em 1. |

Exemplos:

inc eax inc variável1

**Instruções de salto**

Apenas os principais tipos de salto (jump) estão na tabela abaixo:

|  |  |  |
| --- | --- | --- |
| **Asm** | **Hexa** | **Descrição** |
| ja | 77 ou 0F87 | salte se acima (jump if above) |
| jae | 73 ou 0F83 | salte se acima ou igual (jump if above or equal) |
| jb | 72 ou 0F82 | salte se abaixo (jump if below) |
| jbe | 76 ou 0F86 | salte se abaixo ou igual (jump if below or equal) |
| je | 74 ou 0F84 | salte se igual (jump if equal) |
| jg | 7F ou 0F8F | salte se maior (jump if greater) |
| jge | 7D ou 0F8D | salte se maior ou igual (jump if greater or equal) |
| jl | 7C ou 0F8C | salte se menor (jump if less) |
| jle | 7E ou 0F8E | salte se menor ou igual (jump if less or equal) |
| jmp | EB ou E9 | salto incondicional |
| jna | 76 ou 0F86 | salte se não acima (jump if not above) |
| jnae | 72 ou 0F82 | salte se não acima ou igual (jump if not above or equal) |
| jnb | 73 ou 0F83 | salte se não abaixo (jump if not below) |
| jnbe | 77 ou 0F87 | salte se não abaixo ou igual (jump if not below or equal) |
| jne | 75 ou 0F85 | salte se não igual (jump if not equal) |
| jng | 7E ou 0F8E | salte se não maior (jump if not greater) |
| jnge | 7C ou 0F8C | salte se não maior ou igual (jump if not greater or equal) |
| jnl | 7D ou 0F8D | salte se não menor (jump if not less) |
| jnle | 7F ou 0F8F | salte se não menor ou igual (jump if not less or equal) |
| jnz | 75 ou 0F85 | salte se não zero (jump if not sero) |
| jz | 74 ou 0F84 | salte se zero (jump if zero) |

**Instrução MOV**

|  |  |
| --- | --- |
| **MOV** | |
| **Sintaxe** | MOV destino, fonte |
| **Flags** | nenhuma |
| **Descrição** | Copia um byte ou word do operando fonte para o operando destino. |

A instrução MOV transfere (MOVe) o conteúdo da fonte para o destino. Ao se executar a transferência, o conteúdo da fonte fica preservado e o conteúdo do destino é substituído pelo conteúdo da fonte.

**Instruções NEG e NOT**

NOT é uma operação lógica e NEG é uma operação aritmética. Ambas são descritas em conjunto para que as diferenças fiquem claras. NOT alterna o valor de cada bit individual:

1 -> 0 0 -> 1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **NOT** | | | **Sintaxe** | NOT destino | | **Lógica** | destino <- NOT (destino) | | **Descrição** | Inverte cada bit do operando destino formando seu complemento de um. O operando pode ser um byte ou um word. | | **Flags** | nenhuma |  |  |  | | --- | --- | | **NEG** | | | **Sintaxe** | NEG destino | | **Lógica** | destino <- NEG (destino) | | **Descrição** | Subtrai o destino de 0 (zero) e salva o complemento de 2 no próprio destino. | | **Flags** | AF CF OF PF SF ZF | |

NEG subtrai o operando destino de 0 e retorna o resultado a este mesmo destino. O efeito é um complemento de dois do operando. O operando também pode ser um byte ou um word. NEG nega o valor do registrador ou da variável numa operação COM sinal.

NEG executa (0 - número), ou seja: neg eax neg variável1

é o mesmo que (0 - EAX) e (0 - variável1) respectivamente. NEG atualiza as flags da mesma maneira que (0 - número). Se o operando for 0 (zero), a flag de carry (CF) é zerada. Em todos os outros casos, a CF é setada para 1.

**Instrução OR**

|  |  |
| --- | --- |
| **OR** | |
| **Sintaxe** | or destino,fonte |
| **Lógica** | destino <- destino **OR** fonte |
| **Descrição** | OR realiza uma operação lógica OR INCLUSIVE bit a bit nos seus operandos e põe o resultado no destino. Todos os bits ativos em qualquer dos operandos estará ativo no resultado. |
| **Flags** | CF OF PF SF ZF (AF indefinido) |

OR retorna **0 quando ambos os operandos forem 0**, senão retorna 1, conforme a tabela abaixo:

|  |  |  |
| --- | --- | --- |
| **Operando 1** | **Operando 2** | **Resultado** |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

OR é usado para ativar bits específicos. No exemplo a seguir, apenas o bit da posição 7 é ativado e os restantes não sofrem alteração:

or dl, 10000000b ; ativa o bit da posição 7 (as posições dos 8 bits são de 0 a 7, em ordem inversa)

OR também pode ser utilizado para checar se um registrador está zerado ou não porque o resultado atualiza o estado da flag de zero (ZF). Por exemplo:

or ebx, ebx ; ebx é igual a zero ? jz ...

**Instrução POP**

|  |  |
| --- | --- |
| **POP** | |
| **Sintaxe** | POP destino |
| **Descrição** | OR Transfere o word do topo da pilha (SS:SP) para o destino e incrementa SP em dois para apontar para o novo topo da pilha. CS não é um destino válido. |
| **Flags** | nenhuma |

O destino pode ser um registrador ou um endereço de memória. A pilha é uma área de memória que armazena dados temporariamente. O registrador SP (stack pointer) sempre contém o endereço da localização que corresponde ao topo da pilha. O princípio de funcionamento da pilha é "último a entrar - primeiro a sair". A pilha é utilizada principalmente pelas instruções push, pop, call e return.

**Instrução PUSH**

|  |  |
| --- | --- |
| **PUSH** | |
| **Sintaxe** | PUSH fonte PUSH valor (apenas para 80188+) |
| **Descrição** | Decrementa SP pelo tamanho do operando (dois ou quatro, valores byte são estendidos por sinal) e transfere um word da fonte para o topo da pilha (SS:SP). |
| **Flags** | nenhuma |

A fonte pode ser um registrador, um endereço de memória ou um valor literal. A pilha é uma área de memória que armazena dados temporariamente. O registrador SP (stack pointer) sempre contém o endereço da localização que corresponde ao topo da pilha. O princípio de funcionamento da pilha é "último a entrar - primeiro a sair". A pilha é utilizada principalmente pelas instruções push, pop, call e return.

**Instruções REP, REPE e REPNE**

|  |  |  |
| --- | --- | --- |
| **REP / REPE / REPNE** | | |
| REP (repeat / repetir), REPE (repeat if equal / repetir se igual) e REPNE (repeat if not equal / repetir se não for igual) são prefixos para instruções string que forçarão a repetição das instruções de acordo com as seguintes condições: | | |
| **Prefixo** | **ECX** | **Efeito** |
| rep | decrementa ecx | repetir se ecx não for zero |
| repe | decrementa ecx | repetir se ecx não 0 e ZF = 1 |
| repz | decrementa ecx | repetir se ecx não 0 e ZF = 1 |
| repne | decrementa ecx | repetir se ecx não 0 e ZF = 0 |
| repnz | decrementa ecx | repetir se ecx não 0 e ZF = 0 |

REPE e REPZ (repeat if zero / repetir se zero) têm o mesmo efeito. O mesmo acontece com REPNE e REPNZ (repeat if not zero / repetir se diferente de zero).

**Instrução SCAS**

|  |
| --- |
| **SCAS** |
| SCAS compara AL (ou AX) com o byte (ou word) apontado por ES:[DI] e incrementa (ou decrementa) DI dependendo do valor de DF, a flag de direção. O incremento ou decremento é feito de 1 em 1 para bytes e de 2 em 2 para words. OVERRIDES NÃO SÃO PERMITIDOS. |

As formas permitidas são:

scasb scasw scas BYTE PTR ES:[DI] scas WORD PTR ES:[DI]

**Instrução SHL**

|  |  |
| --- | --- |
| **SHL** | |
| **Uso** | SHL destino, vezes |
| **Descrição** | Desloca os bits do destino para a esquerda as "vezes" indicadas com zeros colocados à direita. A flag de carry conterá o valor do último bit deslocado. |
| **Flags** | CF OF |

O destino pode ser um registro ou um endereço de memória. As "vezes" podem ser valores ou CL.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **...** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** | **Descrição** | **Carry** |
| ... | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |  |  |
| ... | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | SHL eax,1 | 1 |
| ... | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | SHR eax,2 | 1 |
| ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SHR eax,2 | 0 |

**Instrução SHR**

|  |  |
| --- | --- |
| **SHR** | |
| **Uso** | SHR destino, vezes |
| **Descrição** | Desloca os bits do destino para a direita as "vezes" indicadas com zeros colocados à esquerda. A flag de carry conterá o valor do último bit deslocado. |
| **Flags** | CF OF |

O destino pode ser um registro ou um endereço de memória. As "vezes" podem ser valores ou CL.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **...** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** | **Descrição** | **Carry** |
| ... | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |  |  |
| ... | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | SHR eax,1 | 0 |
| ... | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | SHR eax,2 | 0 |
| ... | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | SHR eax,2 | 1 |

**Instrução SUB**

|  |  |
| --- | --- |
| **SUB** | |
| **Sintaxe** | SUB destino, fonte |
| **Lógica** | destino <- destino - fonte |
| **Descrição** | Subtrai a fonte do destino e o resultado é armazenado no destino. |
| **Flags** | AF CF OF PF SF ZF |

Ambos os operandos podem ser bytes ou words e ambos também podem ser números binários com ou sem sinal.

**Instrução TEST**

|  |  |
| --- | --- |
| **TEST** | |
| **Sintaxe** | TEST destino, fonte |
| **Lógica** | (destino **AND** fonte) |
| **Descrição** | TEST realiza uma operação lógica AND bit a bit nos seus operandos sem alterá-los. **Apenas modifica as flags.** |
| **Flags** | CF <- 0 OF <- 0 CF OF PF SF ZF (AF indefinido) |

Esta instrução é uma variação da instrução AND. TEST faz exatamente o mesmo que AND, apenas descarta os resultados obtidos. Não modifica o destino. Isto significa que pode checar coisas específicas sem alterar os dados. Em outras palavras, TEST faz um AND lógico em seus dois operandos e atualiza as flags **sem modificar o destino e a fonte**.

test ebx,ebx ; EBX é zero ? jz ... ; se sim, então salta

Para otimizar a velocidade, quando comparar um valor num registrador com 0, use o comando TEST. Use TEST quando for comparar o resultado de um comando lógico AND com uma constante imediata se o registrador utilizado for EAX. Também pode ser usado para testar se determinado valor é zero (exemplo: test ebx,ebx seta a flag zero (ZF) se EBX for zero).

TEST é muito útil para examinar o status de bits individuais. Por exemplo, o snippet abaixo passará o controle para UM\_CINCO\_OFF se **ambos** os bits 1 e 5 do registrador AL estiverem zerados (lembre-se de que os bits são numerados de 0 a 7 em ordem inversa). O status de todos os outros bits será ignorado.

test al,00100010b ; filtre os bits 1 e 5 jz UM\_CINCO\_OFF ; se o bit 1 ou o bit 5 estiverem setados, o resultado ; será diferente de zero ... AMBOS\_NAO\_OFF: ... UM\_CINCO\_OFF: ...

TEST oferece as mesmas possibilidades que AND:

variável1 db ? variável2 dw ? test cl, dh test al, variável1 test variável2, si test dl, 0C2h test variável1, 01001011b

Um bom exemplo é para a placa de vídeo. Em modo texto, a tela tem 80 x 25 pixels, perfazendo 2000 células. Cada célula possui um byte de caracter e um byte de atributos. O byte do caracter é o valor ASCII do mesmo. O byte de atributos indica a cor do caracter, a cor de fundo, se o caracter está em alta ou baixa intensidade ou se está piscando. Um byte de atributos tem a seguinte aparência:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| X | R | G | B | I | R | G | B |

Os bits 0, 1 e 2 (RGB) indicam a cor do caracter. 2 é vermelho (Red), 1 é verde (Green) e 0 é azul (Blue). Os bits 4, 5 e 6 (RGB) contém a cor de fundo do caracter, onde 6 é vermelho (Red), 5 é verde (Green) e 4 é azul (Blue). O bit 3 indica alta intensidade e o bit 7 é piscante. Se o bit estiver setado (valor 1), o componente correspondente está ativado. Se o bit estiver zerado, o componente correspondente está desativado.

A primeira coisa que chama a atenção é o quanto de memória é economizado pelo fato das informações estarem todas juntas. Claro que seria possível usar um byte para cada uma das características, mas a memória requerida seria de 8 x 2 000 bytes = 16 000 bytes. Se adicionarmos os 2 000 bytes referentes aos caracteres, o total já seria 18 000 bytes. Da forma explicada acima, obtém-se o mesmo resultado com apenas 4 000 bytes, ou seja, uma economia de 75%. Como há quatro telas (páginas) diferentes numa placa com cores, os totais seriam 72 000 (18 000 x 4) contra 16 000 (4 000 x 4).

Imagine agora que um dos bytes de atributos esteja no registrador DL - pode-se achar quais os bits que estão setados, bastando para isto fazer um TEST DL com um padrão de bits específico. Se a flag zero (ZF) for setada, significa que o resultado é zero e que o bit estava zerado.

test dl, 10000000b ; está piscando ? test dl, 00010000b ; tem azul no fundo ? test dl, 00000100b ; a cor do caracter é vermelho ?

A flag zero (ZF) indica se o componente está ativo ou desativo. Esta flag não vai mostrar se a cor de fundo é azul, porque o vermelho e o verde do fundo também podem estar setados. Apenas um dos componentes pode ser testado em cada test. E lembre-se: TEST não altera os valores da fonte ou do destino, apenas atualiza as flags.

**Máscaras**

Usaremos o byte de atributos do monitor para exemplificar o uso de máscaras.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| X | R | G | B | I | R | G | B |

Os bits 0, 1 e 2 (RGB) indicam a cor do caracter. 2 é vermelho (Red), 1 é verde (Green) e 0 é azul (Blue). Os bits 4, 5 e 6 (RGB) contém a cor de fundo do caracter, onde 6 é vermelho (Red), 5 é verde (Green) e 4 é azul (Blue). O bit 3 indica alta intensidade e o bit 7 é piscante. Se o bit estiver setado (valor 1), o componente correspondente está ativado. Se o bit estiver zerado, o componente correspondente está desativado.

Caso se queira ativar ou desativar determinados bits, sem alterar o valor dos outros, podemos lançar mão de uma máscara AND: **and byte\_do\_video, 10001111b**. Lembrando que a instrução AND retorna 1 apenas quando ambos os bits estiverem setados (tiverem valor 1), sabemos que neste caso os bits 4, 5 e 6 serão zerados enquanto que os outros permanecem inalterados. Como os bits zerados correspondem à cor de fundo, esta operação tornou a cor de fundo preta.

Se quisermos definir a cor de fundo, precisamos de duas operações. A primeira, uma operação de máscara AND como descrito acima, para fazer a cor de fundo preta zerando os bits desejados sem modificar os restantes. A segunda, uma operação de máscara OR, para ativar os bits desejados sem alterar os restantes (cor de fundo azul):

and byte\_de\_video, 10001111b

or byte\_de\_video, 00010000b

As constantes binárias utilizadas para fazer o AND e o OR são chamadas de máscaras. Estas constantes podem estar no formato binário ou hexadecimal. Por exemplo, and byte\_de\_video, 10001111b é o mesmo que and byte\_de\_video, 8Fh e or byte\_de\_video, 00010000b é o mesmo que or byte\_de\_vídeo, 10h.

Reveja as instruções AND e OR caso ainda tenha alguma dúvida.

**Instrução XOR**

|  |  |
| --- | --- |
| **XOR** | |
| **Sintaxe** | XOR destino, fonte |
| **Lógica** | destino <- destino **XOR** fonte |
| **Descrição** | XOR realiza uma operação lógica OR EXCLUSIVE bit a bit nos seus operandos e põe o resultado no destino. |
| **Flags** | CF OF PF SF ZF (AF indefinido) |

XOR retorna **1 quando os operandos forem diferentes**, senão retorna 0, conforme a tabela abaixo:

|  |  |  |
| --- | --- | --- |
| **Operando 1** | **Operando 2** | **Resultado** |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

|  |  |  |
| --- | --- | --- |
|  | **Decimal** | **Binário** |
|  | 77 | 0**1**0**0**1**1**01 |
| **XOR** | 25 | 0**0**0**1**1**0**01 |
| **Resultado** | 84 | 0**1**0**1**0**1**00 |

O OU-Exclusivo Lógico (XOR) significa que o resultado SÓ É VERDADEIRO SE AS CONDIÇÕES FOREM DIFERENTES. Falso e verdadeiro também podem indicar o estado de bits, portanto, podemos efetuar uma operação de OU-Exclusivo Lógico entre dois bits (como na tabela acima) ou numa sequência de bits. Tomemos como exemplo a operação 77 XOR 25. Como sabemos que a operação lógica XOR também é feita bit a bit, precisamos dos valores binários desses dois números para efetuar a operação:

Observe que apenas nas posições onde os bits são diferentes o resultado possui bits com valor 1, portanto, 77 XOR 25 = 84. Um aspecto interessante da operação XOR é que ela é reversível: se fizermos um XOR do resultado com o primeiro operando, obtemos o valor do segundo operando. Da mesma forma, se fizermos um XOR do resultado com o segundo operando, o resultado é o primeiro operando. Outra característica é que, fazendo o XOR de um número com ele mesmo, o resultado sempre será zero. Faça os testes e verifique wink

Este é o cache do Google de <http://www.woodmann.com/crackz/Tutorials/Drme2.htm>. Ele é um instantâneo da página com a aparência que ela tinha em 30 jul. 2015 10:21:52 GMT.

A [página atual](http://www.woodmann.com/crackz/Tutorials/Drme2.htm) pode ter sido alterada nesse meio tempo. [Saiba mais](http://support.google.com/websearch/bin/answer.py?hl=pt-BR&p=cached&answer=1687222)

Versão completa[Versão somente texto](http://webcache.googleusercontent.com/search?q=cache:u9UmiNdaOL8J:www.woodmann.com/crackz/Tutorials/Drme2.htm&hl=pt-BR&gl=br&strip=1&vwsrc=0)[Ver código-fonte](http://webcache.googleusercontent.com/search?q=cache:u9UmiNdaOL8J:www.woodmann.com/crackz/Tutorials/Drme2.htm&hl=pt-BR&gl=br&strip=0&vwsrc=1)Dica: para localizar rapidamente o termo de pesquisa nesta página, pressione **Ctrl+F** ou **⌘-F** (Mac) e use a barra de localização.

# Assembly Language Reference

### Compiled by Dr. ME!

### LDS

LDS Load Pointer using DS

LDS des-reg, source

Logic: DS <- (source + 2)

dest-reg <- (source)

LDS loads into two registers the 32-bit pointer variable found in memory at source.

LDS stores the segment value (the higher order word of source) in DS and the offset

value (the lower-order word of source) in the destination register. The destination

register may be any 16-bit general register (that is, all registers except segment

registers). [LES](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#LES), Load Pointer Using ES, is a comparable instruction that loads the

ES register rather than the DS register.

Example:

var1 dd 25,00,40,20

..

..

Before LDS

DX = 0000

DS = 11F5

LDS DX,var1

After LDS

DX = 0025

DS = 2040

### LES

LES Load Pointer using ES

LES des-reg, source

Logic: ES <- (source)

dest-reg <- (source + 2)

LES loads into two registers the 32-bit pointer variable found in memory at source.

LES stores the segment value (the higher order word of source) in ES and the offset

value (the lower-order word of source) in the destination register. The destination

register may be any 16-bit general register (that is, all registers except segment

registers). [LDS](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#LDS), Load Pointer Using DS, is a comparable instruction that loads the

DS register rather than the ES register.

### LODS

LODS source\_string

Logic: Accumulator <- (ds:si)

if df = 0 si <- si+n ; n = 1 for byte

else si <- si-n ; n = 2 for word

LODS (load from string) moves a byte or word from DS:[si] to AL or AX, and

increments (or decrements) SI depending on the setting of DF, the direction flag

(by 1 for bytes and by 2 for words).

You may use CS:[si], SS:[si] or ES:[si]. This performs the same action (except for

changing SI) as:

mov ax, DS:[SI] ; or AL for bytes

The allowable forms are:

lodsb

lodsw

lods BYTE PTR SS:[si] ; or CS:[si], DS:[si], ES:[si]

lods WORD PTR SS:[si] ; or CS:[si], DS:[si], ES:[si]

Note this instruction is always translated by the compiler into LODSB,

Load String Byte, or LODSW, Load String Word, depending on whether source\_string

refers to a string of bytes or words. In either case, however, you must explicitly

load the SI register with the offset of the string.

### LODSB

Load String Byte

LODSB

Logic: al <- (ds:si)

if df = 0 si <- si+1

else si <- si-1

LODSB transfers the byte pointed to by DS:SI into AL register and increments or

decrements SI (depending on the state of the Direction Flag) to point to the next

byte of the string.

### LODSW

Load String Word

LODSW

Logic: ax <- (ds:si)

if df = 0 si <- si+2

else si <- si-2

LODSW transfers the word pointed to by DS:SI into AX register and increments or

decrements SI (depending on the state of the Direction Flag) to point to the next

word of the string.

Example:

NAME DW 'ALA'

CLD

LEA SI,NAME

LODSW

The first word of NAME will be transferred to rigister AX.

These instructions as well as [LODS](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#LODS) can use [REP/REPE/REPNE/REPZ/REPNZ](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#5_7) to move several

bytes or words

### STOS

STOS (store to string) moves a byte (or a word) from AL (or AX) to ES:[di], and

increments (or decrements) DI depending on the setting of DF, the direction flag

(by 1 for bytes and by 2 for words). NO OVERRIDES ARE ALLOWED. This performs the

same action (except for changing DI) as:

mov ES:[DI], ax ; or AL for bytes

The allowable forms are:

stosb

stosw

stos BYTE PTR ES:[di] ; no override allowed

stos WORD PTR ES:[di] ; no override allowed

### SCAS

SCAS compares AL (or AX) to the byte (or word) pointed to by ES:[di], and

increments (or decrements) DI depending on the setting of DF, the direction flag

(by 1 for bytes and by 2 for words). NO OVERRIDES ARE ALLOWED. This sets the flags

the same way as:

cmp ax, ES:[DI] ; or AL for bytes

The allowable forms are:

scasb

scasw

scas BYTE PTR ES:[di] ; no override allowed

scas WORD PTR ES:[di] ; no override allowed

### SET

SET destination

Logic: If condition, then destination <- 1

else destination <- 0

The SET instructions set the destination byte to 1 if the specified condition is true;

0 otherwise. Here are the SET instructions and the condition they use:

**SET Instruction Flags Explanation**

SETB/SETNAE CF = 1 Set if Below/Not Above or Equal

SETAE/SETNB CF = 0 Set if Above or Equal/Not Below

SETBE/SETNA CF = 1 or Set if Below or Equal/Not Above

ZF = 1

SETA/SETNBE CF = 0 and Set if Above/Not Below or Equal

ZF = 0

SETE/SETZ ZF = 1 Set if Equal/Zero

SETNE/SETNZ ZF = 0 Set if Not Equal/Not Zero

SETL/SETNGE SF <> OF Set if Less/Not Greater or Equal

SETGE/SETNL SF = OF Set if Greater or Equal/Not Less

SETLE/SETNG ZF = 1 or Set if Less or Equal/Not Greater

SF <> OF

SETG/SETNLE ZF = 0 or

SF = OF Set if Greater/Not Less or Equal

SETS SF = 1 Set if Sign

SETNS SF = 0 Set if No Sign

SETC CF = 1 Set if Carry

SETNC CF = 0 Set if No Carry

SETO OF = 1 Set if Overflow

SETNO OF = 0 Set if No Overflow

SETP/SETPE PF = 1 Set if Parity/Parity Even

SETNP/SETPO PF = 0 Set if No Parity/Parity Odd

destination can be either a byte-long register or memory location.

### MOVS

MOVS moves a byte (or a word) from DS:[si] to ES:[di], and increments

(or decrements) SI and DI, depending on the setting of DF, the direction flag

(by 1 for bytes and by 2 for words). You may use CS:[si], SS:[si] or ES:[si], but

you MAY NOT OVERRIDE ES:[di]. Though the following is not a legal instruction, it

signifies the equivalent action to MOVS (not including changing DI and SI):

mov WORD PTR ES:[DI], DS:[SI] ; or BYTE PTR for bytes

The allowable forms are:

movsb

movsw

movs BYTE PTR ES:[di], SS:[si] ;or CS, DS, ES:[si]

movs WORD PTR ES:[di], SS:[si] ;or CS, DS, ES:[si]

### CMPS

CMPS Compare String (Byte or Word)

CMPS destination-string, source-string

Logic: CMP (DS:SI),(ES:DI) ; sets flags only

if DF=0

SI <- SI + n ; n = 1 for byte, 2 for word.

DI <- DI + n

else

SI <- SI - n

DI <- DI - n

This instruction compares two values by subtracting the byte or word pointed to by

ES:DI, from the byte or word pointed to by DS:SI, and sets the flags according to

the result of comparison. The operands themselves are not altered. After the

comparison, SI and DI are incremented (if the Direction Flag is cleared) or

decremented (if the Direction Flag is set), in preparation for comparing the next

element of the string.

This instruction is always translated by the assembler into CMPSB, Compare String

Byte, or CMPSW, Compare String Word, depending on whether source refers to a string

of bytes or words. In either case, you must explicitly load the SI and DI registers

with the offset of the source and destination strings.

You may use CS:[si], SS:[si] or ES:[si], but you MAY NOT OVERRIDE ES:[di]. Although

the following is not a legal action, it signifies the equivalent action to CMPS (not

including changing DI and SI):

cmp WORD PTR DS:[SI], ES:[DI] ; or BYTE PTR for bytes

The allowable forms are:

cmpsb

cmpsw

cmps BYTE PTR SS:[si], ES:[di] ;or CS, DS, ES:[si]

cmps WORD PTR SS:[si], ES:[di] ;or CS, DS, ES:[si]

### CMP

CMP Compare

CMP destination, source

Logic: Flags set according to result of (destination - source)

CMP compares two **numbers** by subtracting the source from the destination and updates

the flags. CMP does not change the source or destination. The operands may be bytes

or words.

**Compare in Key Generating Routines**

Registers are divided into higher and lower registers. for example: eax is divided

into eah eal ah al (h=high, l=low) which looks like:

76 54 32 10 : Byte No. Each of the four (eah,eal,ah,al) represents one byte.

(total:4 bytes = 32 bit)

| | | |

eah | ah |

eal al

So if there´s a compare ah,byteptr[exc] the ByteNo 3&2 are compared with the first

two bytes of ecx (0&1)

Let´s look at the numbers to understand the whole thing a bit better. I take a

fictional input like 123456 and the real serial 987654.

eax: 3938 3736 (9876)

ecx: 3132 3334 (1234)

cmp al,byte ptr [ecx] ;compares 36 with 34

cmp ah,byte ptr [ecx+01] ;compares 37 with 33

[shr](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#SHR) eax,10 ;this prepares the next two numbers in ah,al

;shr 39383736,10 ------> 0000 3938

cmp al, byte prt[ecx+02] ;compares now (after the shift right) 38 with 32

cmp ah, byte ptr[ecx+03] ;compares now (after the shift right) 39 with 31

..

..

add ecx, 00000004 ;get next 4 numbers from input

add edx, 00000004 ;get next 4 numbers from real serial

;"4" is added to both registers. This is obvious because after compering 4

;characters we have to get the next ones by "shifting" the compared 4 away. why do

;we add 4 and not 10? With the help of one register we are able to compare 4

;charaters because one char needs 1 byte and one register has 4 Bytes.

### REP/REPE/REPNE

The string instructions may be prefixed by REP/REPE/REPNE which will repeat the

instructions according to the following conditions:

rep decrement cx ; repeat if cx is not zero

repe decrement cx ; repeat if cx not zero AND zf = 1

repz decrement cx ; repeat if cx not zero AND zf = 1

repne decrement cx ; repeat if cx not zero AND zf = 0

repnz decrement cx ; repeat if cx not zero AND zf = 0

Here, 'e' stands for equal, 'z' is zero and 'n' is not. These repeat instructions

should NEVER be used with a segment override, since the 8086 will forget the

override if a hardware interrupt occurs in the middle of the REP loop.

### FLAGS

SF shows '+' for a positive number. PF shows 'O,' for odd parity. Every time you

perform an arithmetic or logical operation, the 8086 checks parity. Parity is

whether the number contains an even or odd number of 1 bits. If a number contains 3

'1' bits, the parity is odd. Possible settings are 'E' for even and 'O' for odd. [SAL](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#SHL)

checks for parity.

For (1110 0000) SF is now '-'. OF, the overflow flag is set because you changed the

number from positive to negative (from +112 to -32). OF is set if the high bit

changes. What is the unsigned number now? 224. CF is set if a '1' bit moves off the

end of the register to the other side. CF is cleared. PF is '0'. Change the number

to (1100 0000). OF is cleared because you didn't change signs. (Remember, the

leftmost bit is the sign bit for a signed number). PF is now 'E' because you have

two '1' bits, and two is even. CF is set because you shifted a '1' bit off the left

end. CF always signals when a '1' bit has been shifted off the end. If you shift

(0111 0000), the OF flag will be set because the sign changed. The overflow flag,

OF, will never change; if the left bit stays the same.

**'HARD' FLAGS**

IEF, TF and DF are 'hard' flags. Once they are set they remain in the same setting.

If you use DF, the direction flag, in a subroutine, you must save the flags upon

entry and restore the flags on exiting to make sure that DF has not been altered.

### MOVSX

MOVSX destination, source

Logic: destination <- sign extend(source)

This instruction copies a source operand to a destination operand and extends its

sign. This is particularly useful to preserve sign when copying from 8-bit register

to 16-bit one, or from 16-bit register to 32-bit one.

### MOVZX

MOVZX destination, source

Logic: destination <- zero extend(source)

This instruction copies a source operand to a destination operand and zero-extends

it. This is particularly useful to preserve signs when copying from 8-bit register

to 16-bit one, or from 16-bit register to 32-bit one.

The MOVZX takes four cycles to execute due to due zero-extension wobblies. A better

way to load a byte into a register is by:

xor eax,eax

mov al,memory

As the xor just clears the top parts of EAX, the xor may be placed on the OUTSIDE of

a loop that uses just byte values. The 586 shows greater response to such actions.

It is recommended that 16 bit data be accessed with the [MOVSX](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#MOVSX) and MOVZX if you

cannot place the XOR on the outside of the loop.

N.B. Do the "replacement" only for movsx/zx inside loops.

### SBB

SBB Subtract with Borrow

SBB destination, source

Logic: destination <- destination - source - CF

SBB subtracts the source from the destination; subtracts 1 from that result if the

Carry Flag is set, and stores the result in destination. The operands may be bytes

or words; or both may be signed or unsigned binary numbers.

SBB is useful for subtracting numbers that are larger than 16 bits, since it

subtracts a borrow (in the Carry Flag) from a previous operation.

You may subtract a byte-length immediate value from a destination that is a word;

in this case, the byte is sign-extended to 16 bits before the subtraction.

**sbb eax, eax**

Consider the following code snippet:

:0040D437 E8740A0000 call 0040DEB0 ;compares serials. sets eax=1 if

bad; 0 if good

:0040D43C F7D8 [neg](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#NEG) eax

:0040D43E 59 pop ecx

:0040D43F 1BC0 sbb eax, eax ;sets eax = -1 if bad serial else

;(eax = 0)

:0040D441 59 pop ecx

:0040D442 40 [inc](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#INC) eax ;sets eax = 0 if bad serial

;(-1+ 1 = 0)

As a second example, consider the following code snippet:

:004271DA sbb eax, eax ;eax=-1 (if not previously 0)

:004271DC sbb eax, FFFFFFFF ;FFFFFFFF = -1

:004271DF [test](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#TEST) eax, eax <-- is eax=0?

:004271E1 jnz 00427228 <-- jump if eax is not 0

For the third example, study the following code snippet:

:0040DEF4 1BC0 sbb eax, eax

:0040DEF6 D1E0 [shl](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#SHL) eax, 1

:0040DEF8 40 [inc](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#INC) eax

:0040DEF9 C3 ret

Also see how eax, as a Reg Flag, is set equal to 1 in the following code snippet:

1000243E mov al,byte ptr[esi]

10002441 pop edi

10002442 [sub](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#SUB) al,37 ; if al is 37 (7 decimal), the result = 0

10002444 pop esi

10002445 pop ebx

10002446 cmp al,01 ; if at this point al is less than 1, the Carry Flag is set

; To end up with Reg Flag (eax = 1), al must be less than 1

10002448 **sbb** eax,eax

1000244A neg eax

1000244C ret

Note that al at address :1000243E must be = 37 (7 decimal) to make eax = 1 at

:1000244A.

But what is the meaning of the following three code pieces?

1):

Segment: \_TEXT DWORD USE32 00000018 bytes

0000 8b 44 24 04 example1 mov eax,+4H[esp]

0004 23 c0 [and](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#and) eax,eax

0006 0f 94 c1 [sete](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#set) cl

0009 0f be c9 [movsx](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#movsx) ecx,cl

000c 0f 95 c0 [setne](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#set) al

000f 0f be c0 movsx eax,al

0012 03 c1 add eax,ecx

0014 c3 ret

0015 90 nop

0016 90 nop

0017 90 nop

2):

Segment: \_TEXT DWORD USE32 0000001c bytes

0000 55 \_example2 push ebp

0001 8b ec mov ebp,esp

0003 53 push ebx

0004 8b 55 08 mov edx,+8H[ebp]

0007 f7 da neg edx

0009 19 d2 sbb edx,edx

000b 42 inc edx

000c 8b 5d 08 mov ebx,+8H[ebp]

000f f7 db neg ebx

0011 19 db sbb ebx,ebx

0013 f7 db neg ebx

0015 89 d0 mov eax,edx

0017 03 c3 add eax,ebx

0019 5b pop ebx

001a 5d pop ebp

001b c3 ret

3)

Segment: \_TEXT DWORD USE32 00000016 bytes

0000 8b 44 24 04 \_example3 mov eax,+4H[esp]

0004 f7 d8 neg eax

0006 19 c0 sbb eax,eax

0008 40 inc eax

0009 8b 4c 24 04 mov ecx,+4H[esp]

000d f7 d9 neg ecx

000f 19 c9 sbb ecx,ecx

0011 f7 d9 neg ecx

0013 03 c1 add eax,ecx

0015 c3 ret

Well, they mean the SAME - the following simple function:

int example( int g ) {

int x,y;

x = !g;

y = !!g;

return x+y;

}

First code is made by HighC. It IS OPTIMIZED as you see. Second piece is by

Zortech C. Not so well optimized, but shows interesting NON-obvious

calculations:

NEG reg; SBB reg,reg; INC reg; means: if (reg==0) reg=1; else

reg=0; NEG reg; SBB reg,reg; NEG reg; means: if (reg==0) reg=0; else reg=1;

And it is WITHOUT any JUMPS or special instructions (like SETE/SETNE from 1st

example)! Only pure logics and arithmetics! Now one could figure out many

similar uses of the flags, sign-bit-place-in-a-register,

flag-dependent/influencing instructions etc...

(as you see, HighC names functions exactly as they are stated by the

programmer; Zortech adds an underscore at start; Watcom adds underscore

afterwards; etc..)

The third example is again by Zortech C, but for the (same-optimized-by-hand)

function:

int example( int g ) { return !g + !!g; }

I put it here to show the difference between compilers - HighC just does not

care if you optimize the source yourself or not - it always produces the same

most optimized code (it is because the optimization is pure logical; but it will

NOT figure out that the function will always return 1, for example ;)... well,

sometimes it does!); while Zortech cannot understand that x,y,z are not needed,

and makes a new stack frame, etc... Of course, it could even be optimized more

(but by hand in assembly!): e.g. MOV ECX,EAX (2bytes) after taking EAX from

stack, instead of taking ECX from stack again (4bytes)... but hell, you're

better off to replace it with the constant value 1!

Other similar "strange" arithmetics result from the compiler's way of

optimizing calculations. Multiplications by numbers near to powers of 2 are

substituted with combinations of logical shifts and arithmetics. For example:

reg\*3 could be (2\*reg+reg): MOV eax,reg; [SHL](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#shl) eax,1; add eax,reg; (instead of

MUL reg,3); but it can be even done in ONE instruction (see above about LEA

instruction): [LEA](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#lea) eax,[2\*reg+reg]

reg\*7 could be (8\*reg-reg): MOV eax,reg; SHL eax,3; sub eax,reg

### SUB

SUB Subtract

SUB destination,source

Logic: destination <- destination - source

SUB subtracts the source operand from the destination operand and stores the

results in destination. Both operands may be bytes or words; and both may be

signed or unsigned binary numbers.

You may wish to use [SBB](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#SBB) if you need to subtract numbers that are larger than

16 bits, since SBB subtracts a borrow from a previous operation.

You may subtract a byte-length immediate value from a destination that is a word;

in this case, the byte is sign-extended to 16 bits before the subtraction.

### CBW

Convert Byte to Word

Logic: if (AL < 80h then

AH <- 0

else

AH <- FFh

CBW extends the sign bit of the byte in the AL register into the AH register. In

other words, this instruction extends a signed byte value into the equivalent word

value. This means that the instruction gives value to AH according to the sign bit

of AL. If the sign bit of AL is 1, then all bits in AH will become 1 too (negative

number). If the sign bit of AL is 0, then all bits of AH will also become 0.

Note: This instruction will set AH to 0FFh if the sign bit (bit 7) of AL is

set; if bit 7 of AL is not set, AH will be set to 0. The instruction is useful for

generating a word from a byte prior to performing byte [multiplication](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IMUL) or [division](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IDIV).

### CWD

Convert Word to Doubleword

Logic: if (AX < 8000h) then

DX <- 0

else

DX <- FFFFh

If the sign bit in AX is 1, then this instruction will set all bits in DX, making

them all 1 (negative number); and if the sign bit in AX is 0, it will clear all bits

in DX, making them all 0.

In other words, CWD extends the sign bit of the AX register into the DX register.

This instruction generates the double-word equivalent of the signed number in the AX

register.

Note: This instruction will set DX to 0FFFFh if the sign bit (bit 15) of AX is set;

if bit 15 of AX is not set, DX will be set to 0.

### CDQ

Convert Double to Quad

Logic: EDX:EAX <- Sign extend(EAX)

This instruction converts a signed double word in EAX to a quad word, also signed,

in EDX:EAX. It extends the sign bit.

### IMUL, MUL

MUL Integer Multiply, Unsigned

Multiplies two unsigned integers (always positive)

IMUL Integer Multiply, Signed

Multiplies two signed integers (either positive or negitive)

Syntax:

MUL source ; (register or variable)

IMUL source ; (register or variable)

Logic:

AX <- AL \* source ;if source is a byte

DX:AX <- AX \* source ;if source is a word

This multiplies the register given by the number in AL or AX depending on the

size of the operand. The answer is given in AX. If the answer is bigger than

16 bits then the answer is in DX:AX (the high 16 bits in DX and the low 16

bits in AX).

On a 386, 486 or Pentium the EAX register can be used and the answer is stored

in EDX:EAX. (See also [Multiplication](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#3_7).)

64-bit multiplications are handled in the same way, using EDX:EAX instead.

IMUL has two additional uses that allow for 16-bit results:

1) IMUL register16, immediate16

In this form, register16 is multiplied by immediate16, and the result is placed

in register16.

2) IMUL register16, memory16, immediate16

Here, memory16 is multiplied by immediate16 and the result is placed in register16.

In both of these forms, the carry and over flow flags will be set if the result16

is too large to fit into 16 bits.

**INTEGER MULTIPLY**

The integer multiply by an immediate can usually be replaced with a faster

and simpler series of shifts, subs, adds and lea's.

As a rule of thumb when 6 or fewer bits are set in the binary representation

of the constant, it is better to look at other ways of multiplying and not use

INTEGER MULTIPLY. (the thumb value is 8 on a 586)

A simple way to do it is to shift and add for each bit set, or use LEA.

Here the [LEA](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#LEA) instruction comes in as major cpu booster, for example:

LEA ECX,[EDX\*2] ; multiply EDX by 2 and store result into ECX

LEA ECX,[EDX+EDX\*2] ; multiply EDX by 3 and store result into ECX

LEA ECX,[EDX\*4] ; multiply EDX by 4 and store result into ECX

LEA ECX,[EDX+EDX\*4] ; multiply EDX by 5 and store result into ECX

LEA ECX,[EDX\*8] ; multiply EDX by 8 and store result into ECX

LEA ECX,[EDX+EDX\*9] ; multiply EDX by 9 and store result into ECX

And you can combine leas too!!!!

lea ecx,[edx+edx\*2] ;

lea ecx,[ecx+ecx\*8] ; ecx <-- edx\*27

(of course, if you can, put three instructions between the two LEA so even on

Pentiums, no AGIs will be produced).

For examples of multiplication, consider the following code snippets:

Byte1 DB 80h

Byte2 DB 40h

WORD1 DW 8000h

WORD2 DW 2000h

MAIN PROC NEAR

CALL C10MUL

CALL D10IMUL

RET

MAIN ENDP

C10MUL PROC ; Multiplication of **unsigned** numbers

MOV AL, BYTE1

[MUL](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IMUL) BYTE2 ; two bytes; result in AX

MOV AX,WORD1 ; two words; result in DX:AX

MUL WORD2

MOV AL, BYTE1 ; one byte and one word; result in DX:AX

SUB AH, AH

MUL WORD1

RET

C10MUL ENDP

D10IMUL PROC ; Multiplication of **signed** numbers

MOV AL, BYTE1 ; one byte by another byte; result in AX

[IMUL](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IMUL) BYTE2

MOVE AX, WORD1 ; one word by another word; result in DX:AX

IMUL WORD2

MOVE AL, BYTE1 ; one byte by one word; result in DX:AX

[CBW](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#CBW)

IMUL WORD1

RET

D10IMUL ENDP

### IDIV, DIV

DIV Divides two unsigned integers(always positive)

IDIV Divides two signed integers (either positive or negitive)

Syntax:

DIV source ;(register or variable)

IDIV source ;(register or variable)

Logic:

AL <- AX/source ; Byte source

AH <- remainder

or

AX <- DX:AX/source ; Word source

DX <- remainder

This works in the same way as [IMUL](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IMUL) and MUL by dividing the number in AX by the

register or variable given. The answer is stored in two places. AL stores the

answer and the remainder is in AH. If the operand is a 16 bit register then

the number in DX:AX is divided by the operand and the answer is stored in AX

and remainder in DX. (See also [Division](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#3_8).)

**INTEGER DIVIDE**

In most cases, an Integer Divide is preceded by a [CDQ](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#CDQ) instruction.

This is a divide instruction using EDX:EAX as the dividend and CDQ sets up EDX.

It is better to copy EAX into EDX, then arithmetic-right-shift EDX 31 places to sign

extend.

The copy/shift instructions take the same number of clocks as CDQ, however, on 586's

allows two other instructions to execute at the same time. If you know the value is

a positive, use XOR EDX,EDX.

For examples of Division, consider the following code snippets:

BYTE1 DB 80h

BYTE2 DB 16h

WORD1 DW 2000h

WORD2 DW 0010h

WORD3 DW 1000h

MAIN PROC NEAR

CALL D10DIV

CALL E10IDIV

RET

MAIN ENDP

..

..

D10DIV PROC ;Division of **unsigned** numbers

MOV AX,WORD1 ;division of one word by one byte

[DIV](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IDIV) BYTE1 ;quotiont in AL, and the remainder in AH

MOV AL, BYTE1 ;division of one byte by one byte

SUB AH,AH ;quotiont in AL, and remainder in AH

DIV BYTE2

MOV DX, WORD2 ;division of a doubleword by one word

MOV AX, WORD3

DIV WORD1

MOV AX, WORD1 ;division of one word by another word

SUB DX, DX

DIV WORD3

RET

D10DIV ENDP

..

..

E10IDIV PROC ;Division of **signed** numbers

MOV AX, WORD1 ;division of one word by a byte

[IDIV](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#IDIV) BYTE1

MOV AL, BYTE1 ;division of one byte by another byte

[CBW](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#CBW)

IDIV BYPTE2

MOV DX, WORD2 ;division of a doubleword by another word

MOV AX, WORD3

IDIV WORD1

MOV AX, WORD1 ;division of one word by another word

[CWD](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#CWD)

IDIV WORD3

RET

E10IDIV ENDP

### LEA

Intel's i80x86 has an instruction called LEA (Load Effective Addressing). It calculates the

address through the usual processor's addressing module, and afterwards does not use it for

memory-access, but stores it into a target register. So, if you write LEA AX,[SI]+7, you will

have AX=SI+7 afterwards. In i386, you could have LEA EDI, [EAX\*4][EBX]+37. In one instruction!

But, if the multiplier is not 1,2,or 4 (i.e. sub-parts of the processor's Word) - you can not

use it - it is not an addressing mode.

LEA means Load Effective Address.

Syntax:

LEA destination,source

Desination can be any 16 bit register and the source must be a memory operand

(bit of data in memory). It puts the offset address of the source in the

destination.

The way we usually enter the address of a message we want to print out is a bit

cumbersome. It takes three lines and it isn’t the easiest thing to remember

mov dx,OFFSET MyMessage

mov ax,SEG MyMessage

mov ds,ax

We can replace all this with just one line. This makes the code easier to read

and it easier to remember. This only works if the data is only in in one segment i.e. small memory model.

lea dx,MyMessage

or mov dx,**OFFSET** MyMessage

Using lea is slightly slower and results in code which is larger. Note that with

LEA, we use only the name of the variable, while with:

mov si, offset variable4

we need to use the word '**offset**'.

LEA's generally increase the chance of AGI's (ADDRESS GENERATION STALLS). However,

LEA's can be advantageous because:

\* In many cases an LEA instruction may be used to replace constant

multiply instructions. (a sequence of LEA, add and shift for example)

(See also [INTEGER MULTIPLY](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#INTEGER MULTIPLY).)

\* LEA may be used as a three/four operand addition instruction.

LEA ECX, [EAX+EBX\*4+ARRAY\_NAME]

\* Can be advantageous to avoid copying a register when both operands to

an ADD are being used after the ADD as LEA need not overwrite its

operands.

The general rule is that the "generic"

LEA A,[B+C\*INDEX+DISPLACEMENT]

where A can be a register or a memory location and B,C are registers

and INDEX=1,2,4,8

and DISPLACEMENT = 0 ... 4\*1024\*1024\*1024

or (if performing signed int operations)

-2\*1024\*1024\*1024 ... + (2\*1024\*1024\*1024 -1 )

replaces the "generic" worst-case sequence

MOV X,C ; X is a "dummy" register

MOV A,B

MUL X,INDEX ;actually SHL X, (log2(INDEX))

ADD A,DISPLACEMENT

ADD A,X

So using LEA you can actually "pack" up to FIVE instructions into one

Even counting a "worst case" of TWO OR THREE AGIs caused by the LEA

this is very fast compared to "normal" code.

What's more, cpu registers are precious, and using LEA

you don't need a dummy "X" register to preserve the value of B and C.

### LOGIC

There are a number of operations which work on individual bits of

a byte or word. Before we start working on them, it is necessary

for you to learn the Intel method of numbering bits. Intel starts

with the low order bit, which is #0, and numbers to the left. If

you look at a byte:

7 6 5 4 3 2 1 0

that will be the ordering. If you look at a word:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

that is the ordering. The overwhelming advantage of this is that

if you extend a number, the numbering system stays the same. That

means that if you take the number 45 :

7 6 5 4 3 2 1 0

0 0 1 0 1 1 0 1 (45d)

and sign extend it:

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1

each of the bits keeps its previous numbering. The same is true

for negative numbers. Here's -73:

7 6 5 4 3 2 1 0

1 0 1 1 0 1 1 1 (-73d)

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 (-73d)

In addition, the bit-position number denotes the power of 2 that

it represents. Bit 7 = 2 \*\* 7 = 128, bit 5 = 2 \*\* 5 = 32,

bit 0 = 2 \*\* 0 = 1. {1}.

Whenever a bit is mentioned by number, e.g. bit 5, this is what

is being talked about.

**AND**

AND destination, source

Logic: destination <- destination AND source

AND performs bit-by-bit logical AND operation on its operands and

**stores the result in destination**.

There are five different ways you can AND two numbers:

1. AND two register

2. AND a register with a variable

3 AND a variable with a register

4. AND a register with a constant

5. AND a variable with a constant

That is:

variable1 db ?

variable2 dw ?

and cl, dh

and al, variable1

and variable2, si

and dl, 0C2h

and variable1, 01001011b

You will notice that this time the constants are expressed in hex

and binary. These are the only two reasonable alternatives. These

instructions work bit by bit, and hex and binary are the only two

ways of displaying a number bitwise (bit by bit). Of course, with

hex you must still convert a hex digit into four binary digits.

The table of bitwise actions for AND is:

1 1 -> 1

1 0 -> 0

0 1 -> 0

0 0 -> 0

That is, a bit in the result will be set if and only if that bit

is set in both the source and the destination. What is this used

for? Several things. **First, if you AND a register with itself,**

**you can check for zero.**

**and cx, cx**

(This can also be used to set the flags correctly before starting.)

If any bit is set, then there will be a bit set in the result and

the zero flag will be cleared. If no bit is set, there will be no

bit set in the result, and the zero flag will be set. No bit will

be altered, and CX will be unchanged. This is the standard way of

checking for zero. You can't AND a variable that way:

and variable1, variable1

is an illegal instruction. But you can AND it with a constant

with all the bits set:

**and variable1, 11111111b**

If the bit is set in variable1, then it will be set in the

result. If it is not set in variable1, then it won't be set in

the result. This also sets the zero flag without changing the

variable.

**AND ecx, 00000001**

00000000 ecx, our Target Indicator.

00000001 is simply the value "1", our Source Indicator with which ecx

is ANDed.

--------

00000000

Our result is "0" because no bit PAIRS are set. The result of AND would

only be "1" if the first bit of ecx would be set to "1".

AND is also used in [masks](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#MASK).

**TEST**

Test destination, source

Logic: (destination **and** source)

CF <- 0

OF <- 0

**It sets the flags only.**

There is a variant of [AND](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#AND) called TEST. TEST does exactly

the same thing as AND but throws away the results when it is

done. It does not change the destination. This means that it can

check for specific things without altering the data. In other words,

Test performs a logical and on its two operands and updates the flags.

**Neither destination nor source is changed.**

**test ebx, ebx ; Is ebx zero?**

**jz ---- ; If yes, then jump**

For speed optimization, when comparing a value in a register with 0,

use the TEST command.

TEST operates by ANDing the operands together without spending any

internal time worrying about a destination register.

Use test when comparing the result of a boolean AND command with an

immediate constant for equality or inequality if the register is EAX.

You can also use it for zero testing.

(i.e. test ebx,ebx sets the zero flag if ebx is zero)

TEST is useful for examining the status of individual bits. For

example, the following code snippet will transfer control to

ONE\_FIVE\_ARE\_OFF if **both** bits 1 and 5 of register AL are

cleared. The status of all other bits will be ignored.

test al,00100010b ; mask out all bits except for 1 and 5

jz ONE\_FIVE\_ARE\_OFF ; if either bit was set, the result will

**not** be zero

NOT\_BOTH\_ARE\_OFF:

..

..

ONE\_FIVE\_ARE\_OFF:

..

..

TEST has the same possibilities as AND:

variable1 db ?

variable2 dw ?

test cl, dh

test al, variable1

test variable2, si

test dl, 0C2h

test variable1, 01001011b

will set the flags exactly the same as the similar AND

instructions but will not change the destination. We need another

concrete example, and for that we'll turn to your video card. In

text mode, your screen is 80 X 25. That is 2000 cells. Each cell

has a character byte and an attribute byte. The character byte

has the actual ascii number of the character. The attribute byte

says what color the character is, what color the background is,

whether the character is high or low intensity and whether it

blinks. An attribute byte looks like this:

7 6 5 4 3 2 1 0

X R G B I R G B

Bits 0,1 and 2 are the foreground (character) color. 0 is blue, 1

is green, and 2 is red. Bits 4, 5, and 6 are the background

color. 4 is blue, 5 is green, and 6 is red. Bit 3 is high

intensity, and bit 7 is blinking. If the bit is set (1) that

particular component is activated, if the bit is cleared (0),

that component is deactivated.

The first thing to notice is how much memory we have saved by

putting all this information together. It would have been

possible to use a byte for each one of these characteristics, but

that would have required 8 X 2000 bytes = 16000 bytes. If you add

the 2000 bytes for the characters themselves, that would be 18000

bytes. As it is, we get away with 4000 bytes, a savings of over

75%. Since there are four different screens (pages) on a color

card, that is 18000 X 4 = 72000 bytes compared to 4000 X 4 =

16000. That is a huge savings.

We don't have the tools to access these bytes yet, but let's

pretend that we have moved an attribute byte into dl. We can find

out if any particular bit is set. TEST dl with a specific bit

pattern. If the zero flag is cleared, the result is not zero so

the bit was on. If the zero flag is set, the result is zero so

that bit was off

test dl, 10000000b ; is it blinking?

test dl, 00010000b ; is there blue in the background?

test dl, 00000100b ; is there red in the foreground?

If we look at the zero flag, this will tell us if that component

is on. It won't tell us if the background is blue, because maybe

the green or the red is on too. Remember, test alters neither the

source nor the destination. Its purpose is to set the flags, and

the results go into the Great Bit Bucket in the Sky.

**OR**

The table for OR is:

1 1 -> 1

1 0 -> 1

0 1 -> 1

0 0 -> 0

If either the source or the destination bit is set, then the

result bit is set. If both are zero then the result is zero.

OR is used to turn on a specific bit.

or dl, 10000000b ; turn on blinking

or dl, 00000001b ; turn on blue foreground

After this operation, those bits will be on whether or not they

were on before. It changes none of the bits where there is a 0.

They stay the same as before.

**or ebx, ebx ; Is ebx zero?**

**jz ---- ; If yes, then jump**

To have 1 in ecx:

**or ecx, 00000001**

**XOR**

The table for XOR is:

1 1 -> 0

1 0 -> 1

0 1 -> 1

0 0 -> 0

That is, if both are on or if both are off, then the result is

zero. If only one bit is on, then the result is 1. This is used

to toggle a bit off and on.

xor dl, 10000000b ; toggle blinking

xor dl, 00000001b ; toggle blue foreground

Where there is a 1, it will reverse the setting. Where there is a

0, the setting will stay the same. This leads to one of the

favorite pieces of code for programmers.

**xor ax, ax**

zeros the ax register. There are three ways to zero the ax

register:

mov ax, 0

sub ax, ax

xor ax, ax

The first one is very clear, but slightly slower. For the second

one, if you subtract a number from itself, you always get zero.

This is slightly faster and fairly clear.{2} For the third one,

any bit that is 1 will become 0, and and bit that is 0 will stay

0. It zeros the register as a side effect of the XOR instruction.

You'll never guess which one many programmers prefer. That's

right, XOR. Many programmers prefer the third because it helps

make the code more obsure and unreadable. That gives a certain

aura of technical complexity to the code.

Exchanging A and B without temporary variables could be done by

xor A,B; xor B,A; xor A,B (i.e. A=A^B; B=A^B; A=A^B) sequence and

it WILL work on ANY processor/language supporting XOR operation.

**NEG and NOT**

NOT is a logical operation and NEG is an arithmetical operation.

We'll do both here so you can see the difference. NOT toggles the

value of each individual bit:

1 -> 0

0 -> 1

NOT destination

Logic: destination <- NOT(destination) ; One's complement

NOT inverts each bit of its operand (that is, forms the one's

complement). The operand can be a byte or a word.

NEG destination

Logic: destination <- -destination ; [Two's complement](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#TWO'S COMPLEMENT)

NEG subtracts the destination operand from 0, and returns the result

in the destination. This effectively produces the two's complement

of the operand. The operand may be a byte or a word.

NEG negates the value of the register or variable (a signed

operation). NEG performs (0 - number) so:

neg ax

neg variable1

are equivalent to (0 - AX) and (0 - variable1) respectively. NEG

sets the flags in the same way as (0 - number).

Note: If the operand is zero, the Carry Flag is cleared; in all

other cases, the Carry Flag is set.

**MASKS**

To explain masks, we'll need some data, and we'll use the

attribute byte for the monitor. Here it is again:

7 6 5 4 3 2 1 0

X R G B I R G B

Bits 0,1 and 2 are the foreground (character) color. 0 is blue, 1

is green, and 2 is red. Bits 4, 5, and 6 are the background

color. 4 is blue, 5 is green, and 6 is red. Bit 3 is high

intensity, and bit 7 is blinking.

What we want to do is turn certain bits on and off without

affecting other bits. What if we want to make the background

black without changing anything else? We use and AND mask.

and video\_byte, 10001111b

Bits 0, 1, 2, 3 and 7 will remain unchanged, while bits 4, 5 and

6 will be zeroed. This will make the background black. What if we

wanted to make the background blue? This is a two step process.

First we make the background black, then set the blue background

bit. This involves first the AND mask, then an OR mask.

and video\_byte, 10001111b

or video\_byte, 00010000b

The first instruction shuts off certain bits without changing

others. The second turns on certain bits without effecting

others. The binary constant that we are using is called a mask.

You may write this constant as a binary or a hex number. You

should never write it as a signed or unsigned number (unless you

are one of those people who just adores making code unreadable).

If you want to turn off certain bits in a piece of data, use an

AND mask. The bits that you want left alone should be set to 1,

the bits that you want zeroed should be set to 0. Then AND the

mask with the data.

If you want to turn on certain bits in a piece of data, use an OR

mask. The bits that you want left alone should be set to 0. The

bits that you want turned on should be set to 1. Then OR the mask

with the data.

Go back to [AND](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#AND) and [OR](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#OR) to make sure you believe that this is what

will happen.

### JUMPS

Hex: Asm: Description:

75 or 0F85 jne jump if not equal

74 or 0F84 je jump if equal

77 or 0F87 ja jump if above

0F86 jna jump if not above

0F83 jae jump if above or equal

0F82 jnae jump if not above or equal

0F82 jb jump if below

0F83 jnb jump if not below

0F86 jbe jump if below or equal

0F87 jnbe jump if not below or equal

0F8F jg jump if greater

0F8E jng jump if not greater

0F8D jge jump if greater or equal

0F8C jnge jump if not greater or equal

0F8C jl jump if less

0F8D jnl jump if not less

0F8E jle jump if less or equal

0F8F jnle jump if not less or equal

EB jmp or jmps jump directly to

84 test test

90 nop no operation

### NUMBERS AND ARITHMETIC

You don't habitually use the base two system to balance your

checkbook, so it would be counterproductive to teach you machine

arithmetic on a base two system. What number systems have you had

a lot of experience with? The base 10 system springs to mind. I'm

going to show you what happens on a base 10 system so you will

understand the structure of what happens with computer

arithmetic.

BASE 10 MACHINE

Each place inside the microprocessor that can hold a number is

called a REGISTER. Normally there are a dozen or so of these. Our

base 10 machine has 4 digit registers. They can represent any

number from 0000 to 9999. They are exactly like an industrial

counters or the counters on your tape machines.{1} If you add 27

to a register, the microprocessor counts forward 27; if you

subtract 153 from a register, the microprocessor counts backwards

153. Every time you add 1 to a register, it increments by 1 -

that is 0245, 0246, 0247, 0248. Every time you subtract 1 from a

register, it decrements by 1 - that is 3480, 3479, 3478, 3477.

Let's do some more incrementing. 9997, 9998, 9999, 0000, 0001,

0002. Whoops! That's a problem. When the register reaches 9999

and we add 1, it changes to 0000, not 10,000. How can we tell the

difference between 0000 and 10,000? We can't without a little

help from the CPU.{2} Immediately after an arithmetical

operation, the CPU knows whether you have gone through 10,000

(9999->0000). The CPU has something called a carry flag. It is

internal to the CPU and can have the value 0 or 1. After each

arithmetical operation, the CPU sets the CARRY FLAG to 1 if you

went through the 9999/0000 boundary, and sets the carry flag to 0

if you didn't.{3}

Here are some examples, showing addition, the result, and the

carry flag. The carry flag is normally abbreviated by CF.

number 1 number 2 result CF

0289 4782 5071 0

4398 2964 7382 0

8177 5826 4003 1

6744 4208 0952 1

Note that you must check the carry flag immediately after the

arithmetical operation. If you wait, the CPU will reset it after

the next arithmetical operation.

Now let's do some decrementing. 0003, 0002, 0001, 0000, 9999,

9998. Golly gosh! Another problem. When we got to 0000, rather

than getting -1, -2, we got 9999, 9998. Apparently 9999 stands

for -1, 9998 stands for -2. Yes, that's the system on this, on

the 8086, and on all computers. (Back to that in a moment.) How

do we tell that the number went through 0 ; i.e. 0000->9999? The

carry flag comes to the rescue again. If the number goes through

the 9999/0000 boundary in either direction, the CPU sets the CF

to 1; if it doesn't, the CPU sets the CF to 0. Here's some

subtraction, with the result and the carry flag.

number 1 number 2 result CF

8473 2752 5721 0

2836 4583 1747 1

0654 9281 8627 1

9281 0654 8627 0

Look at examples 3 and 4. The numbers are reversed. The results

are the same but they have different signs. But that is as it

should be. When you reverse the order in a subtraction, you get

the same absolute value, only a different sign (15 - 7 = 8 but

7 - 15 = -8). Remember, the CF is reliable only immediately after

the operation.

**NEGATIVE NUMBERS**

The negative numbers go 9999=-1, 9998=-2, 9997=-3, 9996=-4,

9995=-5 etc. A more negative number is denoted by a smaller

number in the register; -5 = 10,000 -5 = 9995; -498 = 10,000 -498

= 9502, and in general, -x = 10,000 -x. Here are some negative

numbers and their representations on our machine.

number machine no number machine no

-27 9973 -4652 5348

-8916 1084 -6155 3845

As you will notice, these numbers look exactly the same as the

unsigned numbers. They ARE exactly the same as the unsigned

numbers. The machine has no way of knowing whether a number in a

register is signed or unsigned. Unlike BASIC or PASCAL which will

complain whenever you try to use a number in an incorrect way,

the machine will let you do it. This is the power and the curse

of machine language. You are in complete control. It is your

responsibility to keep track of whether a number is signed or

unsigned.

Which signed numbers should be positive and which negative? This

has already been decided for you by the computer, but let's think

out what a reasonable solution might be. We could have from 0000

to 8000 positive and from 9999 to 8001 negative, but that would

give us 8001 positive numbers and 1999 negative numbers. That

seems unbalanced. More importantly, if we take -(3279) the

machine will give us 6721, which is a POSITIVE number. We don't

want that. For reasons of symmetry, the positive numbers are

0000-4999 and the negative numbers are 9999-5000.{4} Our most

negative number is -5000 = 10,000 -5000 = 5000.

**10'S COMPLEMENT**

It's time for a digression. If we are going to be using negative

numbers like -(473), changing from an external number to an

internal number is going to be a bother: i.e. -473 -> 9527. Going

the other way is going to be a pain too: i.e. 9527 -> -473. Well,

it would be a problem except that we have some help.

0000 = 10,000 = 9999 +1

- 473

result 9526 +1 = 9527

Let's work this through carefully. On our machine, 0000 and

10000 (9999+1) are the same thing, so 0 - 473 is the same as

9999+1-473 which is the same as 9999-473+1. But when we have all

9s, this is a cinch. We never have to borrow - all we have to do

is subtract each digit from 9 and then add 1 to the total. We may

have to carry at the end, but that is a lot better than all those

borrows. We'll do a few examples:

(-4276)

0000 = 10,000 = 9999 +1

-4276

result 5723 +1 = 5724

(-3982)

0000 = 10,000 = 9999 +1

-3982

result 6017 +1 = 6018

4. That way, if we tell the machine that we are working with

signed numbers, all it has to do is look at the left digit. If

the digit is 5-9, we have a negative number, if it is 0-4, we

have a positive number. Note that 0000 is considered to be

positive. This is true on all computers.

-1989

result 8010 +1 = 8011

This is called 10s complement. Subtract each digit from 9, then

add 1 to the total. One thing we should check is whether we get

the same number back if we negate the negative result; i.e. does

-(-1989)) = 1989? From the last example, we see that -1989 =

8011, so:

(-8011)

0000 = 10,000 = 9999 +1

-8011

result 1988 +1 = 1989

It seems to work. In fact, it always works. See the footnote for

the proof.{5} You are going to use this from time to time, so you

might as well practice some. Here are 10 numbers to put into 10s

complement form. The answers are in the footnote. (1) -628, (2)

-4194, (3) -9983, (4) -1288, (5) -4058, (6) -6952, (7) -162, (8)

-9, (9) -2744, (10) -5000.{6}

The computer keeps track of whether a number is positive or

negative. After an arithmetical operation, it sets a flag to tell

whether the result is positive or negative. This flag has no

meaning if you are using unsigned numbers. The computer is

saying, "If the last arithmetical operation was with signed

numbers, then this is the sign of the result." The flag is called

the sign flag (SF). It is 0 if the number is positive and 1 if

the number is negative. Let's decrement again and look at both

the sign flag and carry flag.

NUMBER SIGN CARRY

3 0 0

2 0 0

1 0 0

0 0 0

9999 1 1

=================================================================

5. Let x be any number. Then:

-x = ( 10,000 - x) = ( 9999 + 1 - x ) ;

-(-x) = ( 10,000 - (-x) ) = ( 9999 + 1 - (-x) )

= ( 9999 + 1 - ( 9999 + 1 - x ) )

= ( 9999 + 1 - 9999 - 1 + x )

= x

6. (1) -628 = 9372 , (2) -4194 = 5806 , (3) -9983 = 0017,

(4) -1288 = 8712 , (5) -4058 = 5942 , (6) -6952 = 3048

(7) -162 = 9838 , (8) -9 = 9991 , (9) -2744 = 7256,

(10) -5000 = 5000.

This last one is a little strange. It changes 5000 into itself.

In our system, 5000 is a negative number and it winds up as a

negative number. This happens on all computers. If you take the

maximum negative number and take its negative, you get the same

number back.

=================================================================

9998 1 0

9997 1 0

9996 1 0

That worked pretty well. The sign flag changed from 0 to 1 when

we went from 0 to 9999 and the carry flag was set to 1 for that

one operation so we could see that we had gone through the

9999/0000 boundary.

Let's do some more decrementing.

NUMBER SIGN CARRY

5003 1 0

5002 1 0

5001 1 0

5000 1 0

4999 0 0

4998 0 0

4997 0 0

4996 0 0

This one didn't work too well. 5000 is our most negative number

(-5000) and 4999 is our most positive number; when we crossed the

4999/5000 boundary, the sign changed but there was nothing to

tell us that the sign had changed. We need to make another flag.

This one is called the overflow flag. We check the carry flag

(CF) for the 0000/9999 boundary and we check the overflow flag

for the 5000/4999 boundary. The last decrementing example with

the overflow flag:

NUMBER SIGN CARRY OVERFLOW

5003 1 0 0

5002 1 0 0

5001 1 0 0

5000 1 0 0

4999 0 0 1

4998 0 0 0

4997 0 0 0

4996 0 0 0

This time we can find out that we have gone through the boundary.

We'll come back to how the computer sets the overflow flag later,

but let's do some addition and subtraction now.

**UNSIGNED ADDITION AND SUBTRACTION**

Unsigned addition is done the same way as normally. The computer

adds the two numbers. If the result is over 9999, it sets the

carry flag and drops the left digit (i.e. 14625 -> 4625, CF = 1,

19137 -> 9137 CF = 1, 10000 -> 0000 CF = 1). The largest possible

addition is 9999 + 9999 = 19998. This still has a 1 in the left

digit. If the carry flag is set after an addition, the result

must be between 10000 and 19998.

Since this is unsigned addition, we won't worry about the sign

flag or the overflow flag for the moment. Here are some examples

of unsigned addition.

NUMBER 1 NUMBER 2 RESULT CF

5147 2834 7981 0

6421 8888 5309 1

2910 6544 9454 0

6200 6321 2521 1

Directly after the addition, the computer has complete

information about the number. If the carry flag is set, that

means that there is an extra 10,000, so the result of the second

example is 15309 and the result of the fourth example is 12521.

There is no way to store all that information in 4 digits in

memory so that extra information will be lost if it is not used

immediately.

Subtraction is similar. The machine subtracts, and if the answer

is below 0000, it sets the carry flag, borrows 10000 and adds it

to the result. -3158 -> -3135 + 10000 -> 6842 CF = 1 ; -8197 ->

-8197 + 10000 -> 1803 CF = 1. After a subtraction, if the carry

flag is set, you know the number is 10000 too big. Once again,

the carry flag information must be used immediately or it will be

lost. Here are some examples:

NUMBER 1 NUMBER 2 RESULT CF

3872 2655 1217 0

9826 5967 3859 0

4561 7143 7418 1

2341 4907 7434 1

If the carry flag is set, the computer borrowed 10000, so example

3 is 7418 - 10000 = -2582 and example 4 is 7434 - 10000 = -2566.

**MODULAR ARITHMETIC**

What the computer is doing is modular arithmetic. Modular

arithmetic is like a clock. If it is 11 o'clock and you go

forward 1 hour it's now 12 o'clock; if it's 11 and you go

backwards 1 hour it's now 10. If it's 11 and you go forward 4

hours it's not 15, it's 3. If it's 11 and you go backward 15

hours it's not -4, it's 8.

The clock is doing mod 12 arithmetic.{7}

(A+B) mod 12

(A-B) mod 12

From the clock's viewpoint, 11 o'clock today, 11 o'clock

yesterday and 11 o'clock, June 8, 1754 are all the same thing. If

you go forward 200 hours (that's 12X16 + 8) you will have the

same result as going forward 8 hours. If you go backwards 200

hours (that's -(12X16 + 8) = -(12X16) -8) you get the same result

as going backwards 8 hours. If you go forward 4 hours from 11

(11+4) mod 12 = 3 you get the same result as going backwards 8

hours (11-8) mod 12 = 3. In fact, these come in pairs. If A + B =

12, then going forward A hours gives the same result as going

backwards B hours. Forwards 9 = backwards 3; forwards 7 =

backwards 5; forwards 11 = backwards 1.

In the mod 12 system, the following things are equivalent:

(+72 + 4) (+72 - 8)

(+60 + 4) (+60 - 8)

(+48 + 4) (+48 - 8)

(+36 + 4) (+36 - 8)

(+24 + 4) (+24 - 8)

(+12 + 4) (+12 - 8)

( 0 + 4) ( 0 - 8)

(-12 + 4) (-12 - 8)

(-24 + 4) (-24 - 8)

(-36 + 4) (-36 - 8)

(-48 + 4) (-48 - 8)

(-60 + 4) (-60 - 8)

They form what is known as an equivalence class mod 12. If you

use any one of them for addition or subtraction, you will get the

same result (mod 12) as with any other one. Here's some

addition:{8}

(+48 + 4) + 7 = (48 + 11) mod 12 = 11

(-48 - 8) + 7 = (48 - 1 ) mod 12 = 11

( 0 - 8) + 7 = ( 0 - 1 ) mod 12 = 11

(-60 + 4) + 7 = (-60 +11) mod 12 = 11

And some subtraction:

(+48 + 4) - 2 = (48 + 2 ) mod 12 = 2

(-48 - 8) - 2 = (48 - 10) mod 12 = 2

( 0 - 8) - 2 = ( 0 - 10) mod 12 = 2

(-60 + 4) - 2 = (-60 + 2) mod 12 = 2

Our pretend computer doesn't cycle every 12 numbers, it cycles

every 10,000 numbers - it is a mod 10,000 machine. On our

machine, the number 6453 has the following equivalence class:

(+30000 + 6453) (+30000 - 3547)

(+20000 + 6453) (+20000 - 3547)

(+10000 + 6453) (+10000 - 3547)

( 0 + 6453) ( 0 - 3547)

(-10000 + 6453) (-10000 - 3547)

(-20000 + 6453) (-20000 - 3547)

(-30000 + 6453) (-30000 - 3547)

=================================================================

8. (-10) mod 12 = 2 ; (-11) mod 12 = 1

=================================================================

Any one of these will act the same as any other one. Notice that

10000 - 3547 is the subtraction that we did to get the

representation of -3547 on the machine.

-3547 = 9999 + 1

3547

6452 + 1 = 6453

6453 and -3547 act EXACTLY the same on this machine. What this

means is that there is no difference in adding signed or unsigned

numbers on the machine. The result will be correct if interpreted

as an unsigned number; it will also be correct if interpreted as

a signed number.

6821 + 3179 = 10000 so -3179 = 6821 and 3179 = -6821

5429 + 4571 = 10000 so -4571 = 5429 and 4571 = -5429

Since -3179 and 6821 act the same on our machine and since -4571

and 5429 act the same, let's do some addition. Take your time so

you understand why the signed and unsigned numbers are giving the

same results mod 10000:

=================================================================

6821 + 497 = 7318

-3179 + 497 = (10000 - 3179) + 497 = 10000 -2682 = -2682

7318 + 2682 = 10000 so -2682 = 7318

==================================================================

5429 + 876 = 6305

-4571 + 876 = (10000 - 4571) + 876 = 10000 - 3695 = -3695

6305 + 3695 = 10000 so -3695 = 6305

==================================================================

Here's some subtraction:

6821 - 507 = 6314

-3179 - 507 = (10000 - 3179) - 507 = 10000 - 3686 = -3686

6314 + 3686 = 10000 so -3686 = 6314

5429 - 178 = 5251

-4571 - 178 = (10000 - 4571) - 178 = 10000 - 4749 = -4749

5251 + 4749 = 10000 so -4749 = 5251

It is the same addition or subtraction. Interpreted one way it is

signed addition or subtraction; interpreted another way it is

unsigned addition or subtraction.

The machine could have one operation for signed addition and

another operation for unsigned addition, but this would be a

waste of computer resources. These operations are exactly the

same. This machine, like all computers, has only one integer

addition operation and one integer subtraction operation. For

each operation, it sets the flags of importance for both signed

and unsigned arithmetic.

For unsigned addition and subtraction, CF, the carry flag tells

whether the 0000/9999 boundary has been crossed.

For signed addition and subtraction, SF, the sign flag tells the

sign of the result and OF, the overflow flag tells whether the

result was too negative or too positive.

**SIGN EXTENSION**

Although our base 10 machine is set up for 4 digit numbers, it is

possible to use it for numbers of any size by writing the

appropriate software. We'll use 12 digit numbers as an example,

though they could be of any length. The first problem is

converting 4 digit numbers into 12 digit numbers. If the number

is an unsigned number, this is no problem (we'll write the number

in groups of 4 digits to keep it readable):

4816 -> 0000 0000 4816

9842 -> 0000 0000 9842

127 -> 0000 0000 0127

what if it is a signed number? The first thing we need to know

about signed numbers is, what is positive and what is negative?

Once again, for reasons of symmetry, we choose positive to be

0000 0000 0000 to 4999 9999 9999 and negative to be 5000 0000

0000 to 9999 9999 9999.{9} This longer number system cycles from

9999 9999 9999 to 0000 0000 0000. Therefore, for longer numbers,

0000 0000 0000 = 1 0000 0000 0000. They are equivalent.

0000 0000 0000 = 9999 9999 9999 + 1.

If it is a positive signed number, it is still no problem (recall

that in our 4 digit system, a positive number is between 0000 and

4999, a negative signed number is between 5000 and 9999). Here

are some positive signed numbers and their conversions:

1974 -> 0000 0000 1974

1 -> 0000 0000 0001

3909 -> 0000 0000 3909

=================================================================

9. Once again, the sign will be decided by the left hand

digit. If it is 0-4 it is a positive number; if it is 5-9 it is a

negative number.

==================================================================

If it is a negative number, where did its representation come

from in our 4 digit system? -x -> 9999 + 1 -x = 9999 - x + 1.

This time it won't be 9999 + 1 but 9999 9999 9999 + 1. Let's have

some examples.

4 DIGIT SYSTEM 12 DIGIT SYSTEM

-1964

9999 + 1 9999 9999 9999 + 1

-1964 -1964

8035 -> 8036 9999 9999 8035 + 1 -> 9999 9999 8036

-2867

9999 + 1 9999 9999 9999 + 1

-2867 -2867

7132 -> 7133 9999 9999 7132 + 1 -> 9999 9999 7133

-182

9999 + 1 9999 9999 9999 + 1

-182 -182

9817 -> 9818 9999 9999 9817 + 1 -> 9999 9999 9818

As you can see, all you need to do to sign extend a negative

number is to put 9s to the left.

Can't those 9s on the left become 0s when we add that 1 at the

end? No. In order for that to happen, the right four digits must

be 9999. But that can only happen if the number to be negated is

0000:

9999 9999 9999 + 1

-0000

9999 9999 9999 + 1 -> 0000 0000 0000

In all other cases, adding 1 does not carry anything out of the

right four digits.

It is impossible to truncate one of these 12 digit numbers to a 4

digit number without making the results unreliable. Here are two

examples:

(number) 0000 0168 7451 -> 7451 (now a negative number)

(actual value) +168 7451 -2549

(number) 9999 9643 2170 -> 2170 (now a positive number)

(actual value) -356 7830 +2170

We now have 12 digit numbers. Is it possible to add them and

subtract them? Yes but only 4 digits at a time. When you add with

pencil and paper you carry left from each digit. The computer can

carry left from each group of 4 digits. We'll do the following

addition:

0138 6715 6037

+ 2514 2759 7784

Do this with pencil and paper and write down all the carries. The

computer is going to do this in 3 parts:

1) 6037 + 7784

2) 6715 + 2759 + carry (if any)

3) 0138 + 2514 + carry (if any)

The first addition is our regular addition. It will set the carry

flag if the 0000/9999 boundary was crossed (i.e. the result was

larger than 9999). In our case CF = 1 since the result is 13821.

The register holds 3821. We store 3821. Next, we need to add

three things: 6715 + 2759 + CF (=1). There is an instruction like

this on all computers. It adds two numbers plus the value of the

carry flag. Our first addition was ADD (add two numbers). This

time the machine instruction is ADC (add two numbers and the

carry). The result of our second addition is 9475. The register

holds 9475 and CF = 0. We store 9475. Finally, we need to add

three more things: 0138 + 2514 + CF (=0). Once again we use ADC.

The result is 2652, CF = 0. We store the 2652. That is the whole

result:

2652 9475 3821

If CF = 1 at this point, the number has crossed the

9999,9999,9999/0000,0000,0000 boundary. This will work for signed

numbers also. The only difference is that at the very end we

don't check CF, we check OF to see if the

4999,9999,9999/5000,0000,0000 boundary has been crossed.

Just to give you one more example we'll do a subtraction using

the same numbers:

0138 6715 6037

2514 2759 7784

Notice that in order for you to do this with pencil and paper

you'll have to put the larger number on top before you subtract.

With the machine this is unnecessary. Go ahead and do the

subtraction with pencil and paper.

The machine can do this 4 digits at a time, so this is a three

step process:

1) 6037 - 7784

2) 6715 - 2759 - borrow (if any)

3) 0138 - 2514 - borrow (if any)

The first one is a regular subtraction and since the bottom

number is larger, the result is 8253, CF = 1. (Perhaps you are

puzzled because that's not the result that you got. Don't worry,

it all comes out in the wash). Step two subtracts but also

subtracts any borrow (We had a borrow because CF = 1). There is a

special instruction called SBB (subtract with borrow) that does

just that. 6715 - 2759 - 1 = 3955, CF = 0. We store the 3955 and

go on to the third part. This also is SBB, but since we had no

borrow, we have 0138 - 2514 - 0 = 7624, CF = 1. We store 7624.

This is the end result, and since CF = 1, we have crossed the

9999,9999,9999/0000,0000,0000 boundary. This is going to be the

representation of a negative number mod 1,0000,0000,0000. With

pencil and paper, your result was:

-2375 6044 1747

The machine result was:

7624 3955 8253

But CF was 1 at the end, so this represents a negative number.

What number does it represent? Let's take its negative to get a

positive number with the same absolute value:

9999 9999 9999 + 1

7624 3955 8253

2375 6044 1746 + 1 = 2375 6044 1747

This is the same thing you got with pencil and paper. The reason

it looked wierd is that a negative number is always stored as its

modular equivalent. If you want to read a negative number, you

need to take its negative to get a positive number with the same

absolute value.

If we had been working with signed numbers, we wouldn't have

checked CF at the very end, we would have checked OF to see if

the 4999,9999,9999/5000,0000,0000 boundary had been crossed. If

OF = 1 at the end, then the result was either too negative or too

positive.

**OVERFLOW**

How does the machine decide that overflow has occured? First,

what exactly is overflow and when is it possible for overflow to

occur?

Overflow is when the result of a signed addition or subtraction

is either larger than the largest positive number or more

negative than the most negative number. In the case of the 4

digit machine, larger than +4999 or more negative than -5000.

If one number is negative and the other is positive, it is not

possible for overflow to occur. Take +32 and -4791 as examples.

If we start with the positive number (+32) and add the negative

number (-4791), the result can't possibly be too positive.

Similarly, if we start with the negative number (-4791) and add

the positive number (+32), the result can't be too negative.

Therefore, the result can be neither too positive nor too

negative. Make sure you understand this before going on.

What if both are positive? Then overflow is possible. Here are

some examples:

(+3500) + (+4500) = 8000 = -2000

(+2872) + (+2872) = 5744 = -4256

(+1799) + (+4157) = 5956 = -4044

In each case, two positive numbers give a negative result. How

about two negative numbers?

(7154) + (6000) = 3154 = +3154

(actual value) -2946 -4000

(5387) + (5826) = 1213 = +1213

(actual value) -4613 -4174

(8053) + (6191) = 4244 = +4244

(actual value) -1947 -3809

The numbers underneath are the negative numbers that the numbers

above them represent. In these cases, adding two negative numbers

gives a positive result.

This is what the machine checks for. Before the addition, it

checks the signs of the numbers. If the signs are the same, then

the result must also be the same sign or overflow has

occurred.{10} Thus + and + must have a + result; - and - must

have a - result. If not, OF (the overflow flag) is set (OF = 1).

Otherwise OF is cleared (OF = 0).

**MULTIPLICATION**

Unsigned multiplication is easy. The machine simply multiplies

the two numbers. Since the result can be up to 8 digits (the

maximum result is 9999 X 9999 = 9998 0001) the machine uses two

registers to hold the result. We'll call them R1 and R2.

5436 X 174 R1 0094

R2 5864

2641 X 2003 R1 0528

R2 9923

You need to know which register holds which half of the result,

but besides that, everything is straightforward. On this machine

R1 holds the left four digits and R2 holds the right four digits.

Notice that our machine has changed the modular base from N to

N\*N (from 1 0000 to 1 0000 0000). What this means is that two

things which are modularly equivalent under addition and

subtraction are not necessarily equivalent under multiplication

and division. 6281 and -3719 will not work the same.

The machine can't do signed multiplication. What it actually does

is convert the numbers to positive numbers (if necessary),

perform unsigned multiplication, and then do sign adjustment of

the results (if necessary). It uses 2 registers for the result.

SIGNED MULTIPLICATION REGS RESULT

(number) (5372) X (3195) R1 8521 = -1478 6460

(actual value) -4628 X +3195 R2 3540

(number) (9164) X (8746) R1 0104 = +104 8344

(actual value) -836 X -1254 R2 8344

(number) (9927) X (0013) R1 9999 = -949

(actual value) -73 X +13 R2 9051

Looking at the last example, if we performed unsigned

multiplication on those two numbers, we would have

9927 X 0013 = 0012 9051, a completely different answer from the

one we got. Therefore, whenever you do multiplication, you have

to tell the machine whether you want unsigned or signed

multiplication.

**DIVISION**

Unsigned division is easy too. The machine divides one number by

the other, puts the quotient in one register and the remainder in

another. Once again, the only problem is remembering which

register has the quotient and which register has the remainder.

For us, the quotient is R1 and the remainder is R2.

6190 / 372 R1 0016 16 remainder 238

R2 0238

9845 / 11 R1 0895 895 remainder 0

R2 0000

As with multiplication, signed division is handled by the machine

changing all numbers to positive numbers, performing unsigned

division, then putting back the appropriate signs.

SIGNED DIVISION REGS RESULT

(number) (7192) / (9164) R1 0003 +3 rem. -300

(actual value)-2808 / -836 R2 9700

(number) (3753) / (9115) R1 9996 -4 rem. +213

(actual value)+3753 / -885 R2 0213

Looking at the last example, 3753 / 9115, if that were unsigned

multiplication the answer would be 0 remainder 3753, a completely

different answer from the signed division. Every time you do a

division, you have to state whether you want unsigned or signed

division.

BASES 2 AND 16

I'm making the assumption that if you are along for the ride you

already know something about binary and hex numbers. This is a

review only.

**BASE 2 AND BASE 16**

Base 2 (binary) allows only 0s and 1s. Base 16 (hexadecimal)

allows 0 - 9, and then makes up the next six numbers by using the

letters A - F. A = 10, B=11, C=12, D=13, E=14 and F=15. You can

directly translate a hex number to a binary number and a binary

number to a hex number. A group of four digits in binary is the

same as a single digit in hex. We'll get to that in a moment.

The binary digits (BITS) are the powers of 2. The values of the

digits (in increasing order) are 1, 2, 4, 8, 16, 32, 64, 128, 256

and so on. 1 + 2 + 4 + 8 = 15, so the first four digits can

represent a hex number. This repeats itself every four binary

digits. Here are some numbers in binary, hex, and decimal

BINARY HEX DECIMAL

0100 4 4

1111 F 15

1010 A 10

0011 3 3

Let's go from binary to hex. Here's a binary number.

0110011010101101

To go from binary to hex, first divide the binary number up into

groups of four starting from the right.

0110 0110 1010 1101

Now simply change each group into a hex number.

0110 -> 4 + 2 -> 6

0110 -> 4 + 2 -> 6

1010 -> 8 + 2 -> A

1101 -> 8 + 4 + 1 -> D

and we have 66AD as the result. Similarly, to go from hex to

binary:

D39F

change each hex digit into a set of four binary digits:

D = 13 -> 8 + 4 + 1 -> 1101

3 -> 2 + 1 -> 0011

9 -> 8 + 1 -> 1001

F = 15 -> 8+4+2+1 -> 1111

and then put them all together:

1101001110011111

Of course, having 16 digits strung out like that makes it totally

unreadable, so in this book, if we are talking about a binary

number, it will always be separated every 4 digits for

clarity.{1}

All computers operate on binary data, so why do we use hex

numbers? Take a test. Copy these two binary numbers:

1011 1000 0110 1010 1001 0101 0111 1010

0111 1100 0100 1100 0101 0110 1111 0011

Now copy these two hex numbers:

B86A957A

7C4C56F3

As you can see, you recognize hex numbers faster and you make

fewer mistakes in transcription with hex numbers.

**ADDITION AND SUBTRACTION**

The rules for binary addition are easy:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 (carry 1 to the next digit left)

similarly for binary subtraction:

0 - 0 = 0

0 - 1 = 1 (borrow 1 from the next digit left)

1 - 0 = 1

1 - 1 = 0

On the 8086, you can have a 16 bit (binary digit) number

represent a number from 0 - 65535. 65535 + 1 = 0 (65536). For

binary numbers, the boundary is 65535/0. You count up or down

through that boundary. The 8086 is a mod 65536 machine. That

means the things that are equivalent to 35631 mod 65536 are:{2}

================================================================

1. This will not be true of the actual assembler code, since

the assembler demands an unseparated number.

2. 35631 + 29905 = 65536. -29905 = 35631 (mod 65536)

================================================================

(3\*65536 + 35631) (3\*65536 - 29905)

(2\*65536 + 35631) (2\*65536 - 29905)

(1\*65536 + 35631) (1\*65536 - 29905)

( 0 + 35631) ( 0 - 29905)

(-1\*65536 + 35631) (-1\*65536 - 29905)

(-2\*65536 + 35631) (-2\*65536 - 29905)

(-3\*65536 + 35631) (-3\*65536 - 29905)

The unsigned number 35631 and the signed number -29905 look the

same. They ARE the same. In all addition, they will operate in

the same fashion. The unsigned number will use CF (the carry

flag) and the signed number will use OF (the overflow flag).

On all 16 bit computers, 0-32767 is positive and 32768 - 65535 is

negative. Here's 32767 and 32768.

32767 0111 1111 1111 1111

32768 1000 0000 0000 0000

32768 and all numbers above it have the left bit 1. 32767 and all

numbers below it have the left bit 0. This is how to tell the

sign of a signed number. If the left bit is 0 it's positive and

if the left bit is 1 it's negative.

**TWO'S COMPLEMENT**

In base 10 we had [10's complement](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#10'S COMPLEMENT) to help us with negative

numbers. In base 2, we have 2s complememt.

0 = 65536 = 65535 + 1

so we have:

1 0000 0000 0000 0000 = 1111 1111 1111 1111 + 1

To get the negative of a number, we subtract:

-49 = 0 - 49 = 65536 - 49 = 65535 - 49 + 1

(65536) 1111 1111 1111 1111 + 1

(49) 0000 0000 0011 0001

result 1111 1111 1100 1110 + 1 -> 1111 1111 1100 1111 (-49)

; - - - - -

-21874

(65536) 1111 1111 1111 1111 + 1

(21874) 0101 0101 0101 0111

result 1010 1010 1010 1000 + 1 -> 1010 1010 1010 1001 (-21847)

; - - - - -

-11628

(65536) 1111 1111 1111 1111 + 1

(11628) 0010 1101 0110 1100

result 1101 0010 1001 0011 + 1 -> 1101 0010 1001 0100 (-11628)

; - - - - -

-1764

(65536) 1111 1111 1111 1111 + 1

(1764) 0000 0110 1110 0100

result 1111 1001 0001 1011 + 1 -> 1111 1001 0001 1100 (-1764)

; - - - - -

Notice that since:

1 - 0 = 1

1 - 1 = 0

when you subtract from 1, you are simply switching the value of

the subtrahend (that's the number that you subtract).

1 -> 0

0 -> 1

1 becomes 0 and 0 becomes 1. You don't even have to think about

it. Just switch the 1s to 0s and switch the 0s to 1s, and then

add 1 at the end. Well do one more:

-348

(65536) 1111 1111 1111 1111 + 1

(348) 0000 0001 0101 1100

result 1111 1110 1010 0011 + 1 -> 1111 1110 1010 0100 (-348)

Now two more, this time without the crutch of having the top

number visible. Remember, even though you are subtracting, all

you really need to do is switch 1s to 0s and switch 0s to 1s, and

then add 1 at the end.

-658

(658) 0000 0010 1001 0010

result 1111 1101 0110 1101 + 1 -> 1111 1101 0110 1110 (-658)

; - - - - -

-31403

(34103) 0111 1010 0100 0111

result 1000 0101 1011 1000 + 1 -> 1000 0101 1011 1001 (-31403)

**SIGN EXTENSION**

If you want to use larger numbers, it is possible to use multiple

words to represent them.{3} The arithmetic will be done 16 bits

at a time, but by using the method described in Chapter 0.1, it

is possible to add and subtract numbers of any length. One normal

length is 32 bits. How do you convert a 16 bit to a 32 bit

number? If it is unsigned, simply put 0s to the left:

0100 1100 1010 0111 -> 0000 0000 0000 0000 0100 1100 1010 0111

What if it is a signed number? The first thing we need to know

about signed numbers is what is positive and what is negative.

Once again, for reasons of symmetry, we choose positive to be

from 0000 0000 0000 0000 0000 0000 0000 0000

to 0111 1111 1111 1111 1111 1111 1111 1111

(hex 00000000 to 7FFFFFFF)

and we choose negative to be

from 1000 0000 0000 0000 0000 0000 0000 0000

to 1111 1111 1111 1111 1111 1111 1111 1111

(hex 10000000 to FFFFFFFF).{4}

This longer number system cycles

from 1111 1111 1111 1111 1111 1111 1111 1111

to 0000 0000 0000 0000 0000 0000 0000 0000

(hex FFFFFFFF to 00000000).

Notice that by using binary numbers we are innundating ourselves

with 1s and 0s.

If it is a positive signed number, it is still no problem (recall

that in our 16 bit system, a positive number is between 0000 0000

0000 0000 and 0111 1111 1111 1111, a negative signed number is

between 1000 0000 0000 0000 and 1111 1111 1111 1111). Just put 0s

to the left. Here are some positive signed numbers and their

conversions:

(1974)

0000 0111 1011 0110 -> 0000 0000 0000 0000 0000 0111 1011 0110

(1)

0000 0000 0000 0001 -> 0000 0000 0000 0000 0000 0000 0000 0001

(3909)

0000 1111 0100 0101 -> 0000 0000 0000 0000 0000 1111 0100 0101

If it is a negative number, where did its representation come

from in our 16 bit system? -x -> 1111 1111 1111 1111 + 1 -x =

1111 1111 1111 1111 - x + 1. This time it won't be FFFFh + 1 but

FFFFFFFFh + 1. Let's have some examples. (Here we have 8 bits to

the group because there is not enough space on the line to

accomodate 4 bits to the group).

16 BIT SYSTEM 32 BIT SYSTEM

-1964

11111111 11111111 + 1 11111111 11111111 11111111 11111111 + 1

00000111 10101100 00000000 00000000 00000111 10101100

11111000 01010011 + 1 11111111 11111111 11111000 01010011 + 1

11111000 01010100 11111111 11111111 11111000 01010100

=================================================================

4. Once again, the sign will be decided by the left hand

digit. If it is 0 it is a positive number; if it is 1 it is a

negative number.

=================================================================

-2867

11111111 11111111 + 1 11111111 11111111 11111111 11111111 + 1

00001011 00110011 00000000 00000000 00001011 00110011

11110100 11001100 + 1 11111111 11111111 11110100 11001100 + 1

11110100 11001101 11111111 11111111 11110100 11001101

-182

11111111 11111111 + 1 11111111 11111111 11111111 11111111 + 1

00000000 10110110 00000000 00000000 00000000 10110110

11111111 01001001 + 1 11111111 11111111 11111111 01001001 + 1

11111111 01001010 11111111 11111111 11111111 01001010

As you can see, all you need to do to sign extend a negative

number is to put 1s to the left.

Can't those 1s on the left become 0s when we add that 1 at the

end? No. In order for that to happen, the right 16 bits must be

1111 1111 1111 1111. But that can only happen if the number to be

negated is 0:

1111 1111 1111 1111 1111 1111 1111 1111 + 1

-0000 0000 0000 0000

1111 1111 1111 1111 1111 1111 1111 1111 + 1 ->

0000 0000 0000 0000 0000 0000 0000 0000

In all other cases, adding 1 does not carry anything out of the

right 16 bits.

It is impossible to truncate one of these 32 bit numbers to a 16

bit number without making the results unreliable. Here are two

examples:

+1,687,451

00000000 00011001 10111111 10011011 -> 10111111 10011011 (-16485)

-3,524,830

11111111 11001010 00110111 00100010 -> 00110111 00100010 (+14114)

Truncating has changed both the sign and the absolute value of

the number.

### ADDRESSING MODES AND POINTERS

In this section we are going to cover all possible ways of

getting data to and from memory with the different addressing

modes. Read this carefully, since it is likely this is the only

time you will ever see ALL addressing possibilities covered.

The easiest way to move data is if the data has a name and the

data is one or two bytes long. Take the following data:

; -----

variable1 dw 2000

variable2 db -26

variable3 dw -589

; -----

We can write:

mov variable1, ax

mov cl, variable2

mov si, variable3

and the assembler will write the appropriate machine code for

moving the data. **What can we do if the data is more than two**

**bytes long?** Here is some more data:

; -----

variable4 db "This is a string of ascii data."

variable5 dd -291578

variable6 dw 600 dup (-11000)

; -----

Variable4 is the address of the first byte of a string of ascii

data. Variable5 is a single piece of data, but it won't fit into

an 8086 register since it is 4 bytes long. Variable6 is a 600

element long array, with each element having the value -11000. In

order to deal with these, we need pointers.

Some of you will be flummoxed at this point, while those who are

used to the C language will feel right at home. **A pointer is**

**simply the address of a variable.** We use one of the 8086

registers to hold the address of a variable, and then tell the

8086 that the register contains the address of the variable, not

the variable itself. It "points" to a place in memory to send the

data to or retrieve the data from. If this seems a little

confusing, don't worry; you'll get the hang of it quickly.

As I have said before, the 8086 does not have general purpose

registers. Many instructions (such as LOOP, MUL, IDIV, ROL) work

only with specific registers. The same is true of pointers. You

may use only BX, SI, DI, and BP as pointers. The assembler will

give you an error if you try using a different register as a

pointer.

There are two ways to put an address in a pointer. For variable4,

we could write either:

**lea si, variable4**

or:

**mov si, offset variable4**

Both instructions will put the offset address of variable4 in

SI.{1} SI now 'points' to the first byte (the letter 'T') of

variable4. If we wanted to move the third byte of that array

(the letter 'i') to CL, how would we do it? First, we need to

have SI point to the third byte, not the first. That's easy:

add si, 2

But if we now write:

mov cl, si

we will generate an assembler error because the assembler will

think that we want to move the data in SI (a two byte number) to

CL (one byte). How do we tell the assembler that we are using SI

as a pointer? By enclosing SI in square brackets:

**mov cl, [si]**

since CL is one byte, the assembler assumes you want to move one

byte. If you write:

mov cx, [si]

then the assembler assumes that you want to move a word (two

bytes). The whole thing now is:

lea si, variable4

add si, 2

mov cl, [si]

This puts the third byte of the string in CL. Remember, if a

register is in square brackets, then it is holding the ADDRESS of

a variable, and the 8086 will use the register to calculate where

the data is in memory.

What if we want to put 0s in all the elements of variable6?

=================================================================

1 [LEA](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#LEA) stands for load effective address. Note that with LEA,

we use only the name of the variable, while with:

mov si, offset variable4

we need to use the word '**offset**'. The exact difference between

the two will be explained later.

===============================================================

Here's the code:

mov bx, offset variable6

mov ax, 0

mov cx, 600

zero\_loop:

mov [bx], ax

add bx, 2

loop zero\_loop

We add 2 to BX each time since each element of variable6 is a

word (two bytes) long. There is another way of writing this:

mov bx, offset variable6

mov cx, 600

zero\_loop:

mov [bx], 0

add bx, 2

loop zero\_loop

Unfortunately, this will generate an assembler error. Why? If the

assembler sees:

mov [bx], ax

it knows that you want to move what is in AX to the address in

BX, and AX is one word (two bytes) long so it generates the

machine code for a word move. If the assembler sees:

mov [bx], al

it knows that you want to move what is in AL to the address in

BX, and AL is one byte long, so it generates the machine code for

a byte move. If the assembler sees:

mov [bx], 0

it doesn't know whether you want a byte move or a word move. The

8086 assembler has implicit sizing. It is the assembler's job to

look at each instruction and decide whether you want to operate

on a byte or a word. Other microprocessors do things differently.

Back to the 8086. If the 8086 assembler looks at an instruction

and it can't tell whether you want to move a byte or a word, it

generates an error. When you use pointers with constants, you

should explicitly state whether you want a byte or a word. The

proper way to do this is to use the reserved words **BYTE PTR** or

**WORD PTR**.

mov [bx], BYTE PTR 213

mov [bx], WORD PTR 213

These stand for byte pointer and word pointer respectively. I

find this terminology exceptionally clumsy, but that's life.

Whenever you are moving a constant with a pointer, you should

specify either BYTE PTR or WORD PTR.

The Microsoft assembler makes some assumptions about the size of

a constant. If the number is 256 or below (either positive or

negative), you MUST explicitly state whether it is a byte or a

word operation. If the number is 257 or above (either positive or

negative), the assembler assumes that you want a word operation.

Here's the previous code rewritten correctly:

mov bx, offset variable6

mov cx, 600

zero\_loop:

mov [bx], WORD PTR 0

add bx, 2

loop zero\_loop

Let's add 435 to every element in the variable6 array:

mov bx, offset variable6

mov cx, 600

add\_loop:

add [bx], WORD PTR 435

add bx, 2

loop add\_loop

How about multiplying every element in the array by 12?

mov di, offset variable6

mov cx, 600

mov si, 12

mult\_loop:

mov ax, [di]

imul si

mov [di], ax

add di, 2

loop mult\_loop

None of these examples did any error checking, so if the result

was too large, the overflow was ignored. This time we used DI for

a change of pace. Remember, we may use BX, SI, DI or BP, but no

others. You will notice that in all these examples, we started at

the beginning of the array and went step by step through the

array. That's fine, and that's what we normally would do, but

what if we wanted to look at individual elements? Here's a sample

program:

; START DATA BELOW THIS LINE

;

poem\_array db "She walks in Beauty, like the night"

db "Of cloudless climes and starry skies;"

db "And all that's best of dark and bright"

db "Meet in the aspect ratio of 1 to 3.14159"

character\_count db 149

; END DATA ABOVE THIS LINE

; START CODE BELOW THIS LINE

mov bx, offset poem\_array

mov dl, character\_count

character\_loop:

sub ax, ax ; clear ax

call get\_unsigned\_byte

dec al ; character #1 = array[0]

cmp al, dl ; out of range?

ja character\_loop ; then try again

mov si, ax ; move char # to pointer register

mov al, [bx+si] ; character to al

call print\_ascii\_byte

jmp character\_loop

; + + + + + END CODE ABOVE THIS LINE

You enter a number and the program prints the corresponding

character. Before starting, we put the array address in BX and

the maximum character count in DL. After getting the number from

get\_unsigned\_byte, we decrement AL since the first character is

actually poem\_array[0]. The character count has been reduced by 1

to reflect this fact. It also makes 0 an illegal entry. Notice

that the program checks to make sure you don't go past the end of

the poem. This time we use BX to mark the beginning of the array

and SI to count the number of the character.

Once again, there are only specific combinations of pointers that

can be used. They are:

BX with either SI or DI (but not both)

BP with either SI or DI (but not both)

My version of the Microsoft assembler (v5.1) recognizes the forms

[bx+si], [si+bx], [bx][si], [si][bx], [si]+[bx] and [bx]+[si] as

the same thing and produces the same machine code for all six.

We can get even more complicated, but to show that, we need

**structures**. In databases they are called records. In C they are

called structures; in any case they are the same thing - a group

of different types of data in some standard order. After the

group is defined, we usually make an array with the identical

structure for each element of the array.{4} Let's make a

structure for an address book.

last\_name db 15 dup (?)

first\_name db 15 dup (?)

age db ?

tel\_no db 10 dup (?)

In this case, all the data is bytes, but that is not necessary.

It can be anything. Each separate piece of data is called a

FIELD. We have the last\_name field, the first\_name field, the age

field, and the tel\_no field. Four fields in all. The structure is

41 bytes long. What if we want to have a list of 100 names in our

telephone book? We can allocate memory space with the following

definition:

address\_book db 100 dup ( 41 dup (' ')) {5}

Well, that allocates room in memory, but how do we get to

anything? First, we need the array itself:

mov bx, offset address\_book

Then we need one specific entry. Let's take entry 29 (which is

address\_book[28]). Each entry is 41 bytes long, so:

mov ax, 28 ; entry (less 1)

mov cx, 41 ; entry length

mul cx

mov di, ax ; move to pointer

That gives us the entry, but if we want to get the age, that's

not the first byte of the structure, it's the 31st byte (actually

address\_book[28] + 30 **since the first byte is at +0**). We get it

by writing:

mov dl, [bx+di+30]

This is the most complex thing we have - two pointers plus a

constant. The total code is then:

mov bx, offset address\_book

mov ax, 28 ; entry (less 1)

mov cx, 41 ; entry length

mul cx ; entry offset from array[0]

mov di, ax ; move entry offset to pointer

mov dl, [bx+di+30] ; total address

Though the machine code has only one constant in the code, the

assembler will allow you to put a number of constants in the

assembler instruction. It will add them together for you and

resolve them into one number.

Once again, there are a limited number of registers - they are

the same registers as before:

BX with either SI or DI (but not both) plus constant

BP with either SI or DI (but not both) plus constant

We can work with structures on the machine level, but it looks

like it's going to be hard to keep track of where each field is.

Actually, it isn't so bad because of:

OUR FRIEND, THE EQU STATEMENT

The assembler allows you to do substitution. If you write:

somestuff EQU 37 \* 44

then every place that the assembler finds the word "somestuff",

it will substitute what is on the right side of the EQU. Is that

a number or text? Sometimes it's a number, sometimes it's text.

Here are four statements which are defined totally in terms of

numbers. This is from the assembler listing. (The assembler lists

how it has evaluated the EQU statement on the left after the

equal sign.)

= 0023 statement1 EQU 5 \* 7

= 000F statement3 EQU statement2 - 22

and the assembler thinks of these as numbers (these numbers are

in hex). Now in the next set, with only a minor change:

= [bp + 3] statement1 EQU [bp + 3]

= [bp + 3] + 6 - 4 - 22 statement3 EQU statement2 - 22

the assembler thinks of it as text. Obviously, the fact that it

can be either may cause you some problems along the way. Consult

the assembler manual for ways to avoid the problem.

Now we have a tool to deal with structures. Let's look at that

structure again.

last\_name db 15 dup (?)

first\_name db 15 dup (?)

age db ?

tel\_no db 10 dup (?)

We don't actually need a data definition to make the structure,

we need equates:

LAST\_NAME EQU 0

FIRST\_NAME EQU 15

AGE EQU 30

TEL\_NO EQU 31

this gives us the offset from the beginning of each record. If we

again define:

address\_book db 100 dup ( 41 dup (' '))

then to get the age field of entry 87, we write:

mov bx, offset address\_book

mov ax, 86 ; entry (less 1)

mov cx, 41 ; entry length

mul cx ; entry offset from array[0]

mov di, ax ; move entry offset to pointer

mov dl, [bx+di+AGE] ; total address

This is a lot of work for the 8086, but that is normal with

complex structures. The only thing that takes a lot of time is

the multiplication, but if you need it, you need it.

How about a two dimensional array of integers, 60 X 40

int\_array dw 40 dup ( 60 dup ( 0 ))

These are initialized to 0. For our purposes, we'll assume that

the first number is the row number and the second number is the

column number; i.e. array [6,13] is row 6, column 13. We will

have 40 rows of 60 columns. For ease of calculation, the first

array element is int\_array [0,0]. (If it is your array, you can

set it up any way you want {8}). Each row is 60 words (120 bytes)

long. To get to int\_array [23, 45] we have:

mov ax, 120 ; length of one row in bytes

mov cx, 23 ; row number

mul cx

mov bx, ax ; row offset to bx

mov si, 45 ; column offset

sal si, 1 ; multiply column offset by 2 (for word size)

mov dx, [bx+si] ; integer to dx

Using SAL instead of MUL is about 50 times faster. Since most

arrays you will be working with are either byte, word, or double

word (4 bytes) arrays, you can save a lot of time. Let

ELEMENT\_NUMBER be the array number (starting at 0) of the desired

element in a one-dimensional array. For byte arrays, no

multiplication is needed. For a word:

mov di, ELEMENT\_NUMBER

sal di,1 ; multiply by 2

and for a double word (4 bytes):

mov di, ELEMENT\_NUMBER

sal di, 1

sal di, 1 ; multiply by 4

This means that a one-dimensional array can be accessed very

quickly as long as the element length is a power of 2 - either 2,

4 or 8. Since the standard 8086 data types are all 1, 2, 4, or 8

bytes long, one dimensional arrays are fast. Others are not so

fast.

As a quick review before going on, these are the legal ways to

address a variable on the 8086:

(1) by name.

mov dx, variable1

It is also possible to have name + constant.

mov dx, variable1 + 27

The assembler will resolve this into a single offset number

and will give the appropriate information to the linker.

(2) with the single pointers BX, SI, DI and BP (which are

enclosed in square brackets).

mov cx, [si]

xor al, [bx]

add [di], cx

sub [bp], dh

(3) with the single pointers BX, SI, DI and BP (which are

enclosed in square brackets) plus a constant.

mov cx, [si+421]

xor al, 18+[bx]

add 93+[di]-7, cx

sub (54/7)+81-3+[bp]-19, dh

(4) with the double pointers [bx+si], [bx+di], [bp+si],

[bp+di] (which are enclosed in square brackets).

mov cx, [bx][si]

xor al, [di][bx]

add [bp]+[di], cx

sub [di+bp], dh

(5) with the double pointers [bx+si], [bx+di], [bp+si],

[bp+di] (which are enclosed in square brackets) plus a

constant.

mov cx, [bx][si+57]

xor al, 45+[di+23][bx+15]-94

add [bp]+[di]-444, cx

sub [6+di+bp]-5, dh

These are ALL the addressing modes allowed on the 8086. As for

the constants, it is the ASSEMBLER'S job to resolve all numbers

in the expression into a single constant. If your expression

won't resolve into a constant, it is between you and the

assembler. It has nothing to do with the 8086 chip.

We can consolidate all this information into the following list:

All the following addressing modes can be used with or

without a constant:

variable\_name (+constant)

[bx] (+constant)

[si] (+constant)

[di] (+constant)

[bp] (+constant)

[bx+si] (+constant)

[bx+di] (+constant)

[bp+si] (+constant)

[bp+di] (+constant)

This is a complete list.

Thus, you can access a variable by name or with one of the eight

pointer combinations. There are no other possibilities.

One thing that may confuse you about an addressing statement is

all the plusses and minuses. As an example:

mov cx, -45+27[bx+22]+[-195+di]+23-44

the total address is:

-45+27[bx+22]+[-195+di]+23-44

When the 8086 performs this instruction, it will ADD (1) BX (2)

DI and (3) a single constant. That single constant can be a

positive or a negative number; the 8086 will ADD all three

elements. The '+' in front of 'di' is for convenience of the

assembler only; [-195-di] is illegal and the assembler will

generate an error. If you actually want the negative of what is

in one of the registers, you must negate it before calling the

addressing instruction:

neg di

mov cx, -45+27[bx+22]+[-195+di]+23-44

once again, the only allowable forms are +[di], [di] or [+di].

Either -[di] or [-di] will generate an assembler error.

If you ever see a technical description of the addressing modes,

you will find a list of 24 different machine codes. The reason

for this is that:

[bx]

[bx] + byte constant

[bx] + word constant

are three different machine codes. Here is a listing of the same

machine instruction with the three different styles:

MACHINE CODE ASSEMBLER INSTRUCTION

03 04 add ax, [si]

03 44 1B add ax, [si+27]

03 44 E5 add ax, [si-27]

03 84 5BA7 add ax, [si+23463]

03 84 A459 add ax, [si-23463]

(27d = 1Bh , 23463d = 5BA7h). The first byte of code (03) is the

add (word) instruction. The second byte is the addressing code,

and the third and fourth bytes (if any) are the constant (in

hex). Addressing code 04 is: (ax, [si]). Addressing code 44 is:

(ax, [si] + byte constant). Addressing code 84 is: (ax, [si] +

word constant). The fact that there are three different machine

codes is of concern to the assembler, not to you. It is the

assembler's job to make the machine code as efficient as

possible. It is your job to write quality, robust code.

**SEGMENT OVERRIDES**

So far, we haven't talked about segment registers. You will

remember from the last chapter that the 8086 assumes that a named

variable is in the DS segment:

mov ax, variable1

If it isn't, the Microsoft assembler puts the correct segment

override in the machine code. The segment overrides are:

SEGMENT OVERRIDE MACHINE CODE (hex)

CS 2E

DS 3E

ES 26

SS 36

As an example:

MACHINE CODE ASSEMBLER INSTRUCTIONS

2E: 03 06 0000 R add ax, variable3

26: 2B 1E 0000 R sub bx, variable2

31 36 0000 R xor variable1, si ; no override

36: 21 3E 00C8 R and variable4, di

when the different variables were in segments with different

ASSUME statements. If you don't remember this, you should reread

the section on overrides in the last chapter. Remember, the colon

is in the listing only to tell you that we have a segment

override. The colon is not in the machine code.

What about pointers? The natural segment for anything with [bp]

is SS, the stack segment.{1} Everything else has DS as its

natural segment. The natural segments are:

(1) DS

variable + (constant)

[bx] + (constant)

[si] + (constant)

[di] + (constant)

[bx+si] + (constant)

[bx+di] + (constant)

(2) SS

[bp] + (constant)

[bp+si] + (constant)

[bp+di] + (constant)

where the constant is always optional. Can you use segment

overrides? Yes, in all cases.{2} Here is some assembler code

along with the machine code which was generated.

MACHINE CODE ASSEMBLER INSTRUCTIONS

26: 03 07 add ax, es:[bx]

2E: 01 05 add cs:[di], ax

36: 2B 44 11 sub ax, ss:[si+17]

2E: 29 46 00 sub cs:[bp], ax

3E: 33 03 xor ax, ds:[bp+di]

26: 31 02 xor es:[bp+si], ax

26: 89 43 16 mov es:[bp+di+22], ax

03 04 add ax, [si]

03 44 1B add ax, [si+27]

03 84 A459 add ax, [si-23463]

26: 03 04 add ax, es:[si]

26: 03 44 1B add ax, es:[si+27]

26: 03 84 A459 add ax, es:[si-23463]

(17d = 11h, 22d = 16h, 27d = 1Bh, -23463d = 0A459h). The first

number (which is followed by a colon) is the segment override

that the assembler has inserted in the machine code. Remember,

the colon is in the listing to inform you that an override is

involved; it is not in the machine code itself.

Unfortunately, when you use pointers you must put the override

into the assembler instructions yourself. The assembler has no

way of knowing that you want an override. This can cause some

truly gigantic errors (if you reference a pointer seven times and

forget the override once, the 8086 will access the wrong segment

that one time), and those errors are extremely difficult to

detect.

As you can see from above, you put the override in the

instructions by writing the appropriate segment (CS, DS, ES or

SS) followed by a colon. As always, it is your responsibility to

make sure that the segment register holds the address of the

appropriate segment before using an override.

We have talked about two different types of constants in the

chapter, a constant which is part of the address:

mov ax, [bx+17]

add [si+2190], dx

and [di-8179], cx

and a constant which is a number to used for an arithmetical or

logical operation:

add ax, 17

sub dl, 45

add dx, 22187

They are both part of the machine instruction, and are

unchangeable (true constants). This machine code is going to be

difficult to read, so just look for (1) the constant DATA and (2)

the constant in the ADDRESS. All constants in the assembler

instructions are in hex so that they look the same as in the

listing of the machine code. Here's a listing of different

combinations.

1. Pointer + constant as an address:

MACHINE CODE ASSEMBLER INSTRUCTIONS

01 44 1B add [si+1Bh], ax

29 85 0A04 sub [di+0A04h], ax

30 5C 1F xor [si+1Fh], bl

20 9E 1FAB and [bp+1FABh], bl

2. Arithmetic instruction with a constant:

MACHINE CODE ASSEMBLER INSTRUCTIONS

05 1065 add ax, 1065h

2D 6771 sub ax, 6771h

80 F3 37 xor bl, 37h

80 E3 82 and bl, 82h

3. Pointer + constant as an address; arithmetic with a constant

MACHINE CODE ASSEMBLER INSTRUCTIONS

81 44 1B 1065 add [si+1Bh], 1065h

81 AD 0A04 6771 sub [di+0A04h], 6771h

80 74 1F 37 xor [si+1Fh], BYTE PTR 37h

80 A6 1FAB 82 and [bp+1FABh], BYTE PTR 82h

You will notice that the ADD instruction (as well as the other

instructions) changes machine code depending on the complete

format of the instruction (byte or word? to a register or from a

register? what addressing mode? is AX one of the registers?).

That's part of the 8086 machine language encoding, and it makes

the 8086 machine code extremely difficult to decipher without a

table listing all the options.

**OFFSET AND SEG**

There are two special instructions that the assembler has -

offset and seg. For any variable or label, offset gives the

offset from the beginning of the segment, and seg gives the

segment address. If you write:

mov ax, offset variable1

the assembler will calculate the offset of variable1 and put it

in the machine code. It also signals the linker and loader; if

the linker should change the offset during linking, it will also

adjust this number. If you write:

mov dx, seg variable1

The assembler will signal to the linker and the loader that you

want the address of the segment that variable1 is in. The linker

and loader will put it in the machine code at that spot. You

don't need to know the name of the segment. The linker takes care

of that. We will use the seg operator later.

Addressing Modes

SUMMARY

These are the natural (default) segments of all addressing modes:

(1) DS

variable + (constant)

[bx] + (constant)

[si] + (constant)

[di] + (constant)

[bx+si] + (constant)

[bx+di] + (constant)

(2) SS

[bp] + (constant)

[bp+si] + (constant)

[bp+di] + (constant)

Where the constant is optional. Segment overrides may be used.

The segment overrides are:

SEGMENT OVERRIDE MACHINE CODE (hex)

CS: 2E

DS: 3E

ES: 26

SS: 36

OFFSET

The reserved word 'offset' tells the assembler to calculate the

offset of the variable from the beginning of the segment.

mov ax, offset variable2

SEG

The reserved word 'seg' tells the assembler, linker and loader to

get the segment address of the segment that the variable is in.

mov ax, seg variable2

LEA

LEA calculates an address using any of the 8086 addressing modes,

then puts the address in a register.

lea cx, [bp+di+27]

**SHIFT AND ROTATE**

There are seven instructions that move the individual bits of a

byte or word either left or right. Each instruction works

slightly differently. We'll make a standard program and then

substitute each instruction into that program.

**SHL - SAL**

SHL destination,count

CF <-- destination <-- 0

SHL is the same instruction as SAL, Shift Arithmatic Left.

SHL shifts the word or byte at the destination to the left by

the number of bit positions specified in the second operand,COUNT.

As bits are transferred out the left (high-order) end of the

destination, zeros are shifted in the right (low-order) end.

The Carry flag is updated to match the last bit shifted out of

the left end. It is used for multiplying an unsigned number by

powers of 2.

There are two (and only two) forms of this instruction. All other

shift and rotate instructions have these two (and only these two)

forms as well. The first form is:

shl al, 1

Which shifts each bit to the left one bit. The number MUST be 1.

No other number is possible. The other form is:

shl al, cl

shifts the bits in AL to the left by the number in CL. If CL = 3,

it shifts left by 3. If CL = 7, it shifts left by 7. The count

register MUST be CL (not CX). The bits on the left are shifted

out of the register into the bit bucket, and zeros are inserted

on the right.

For a register, it is faster to use a series of 1 shifts than to

load cl. For a variable in memory, anything over 1 shift is

faster if you load cl. CF always signals when a 1 bit has been

shifted off the end.

**Summary**

SHL (shift logical left) and SAL (shift arithmetic left) are

exactly the same instruction. They move bits left. 0s are

placed in the low bit. Bits are shoved off the register (or

memory data) on the left side, and CF indicates whether the

last bit shoved was a 1 or a 0. It is used for multiplying

an unsigned number by powers of 2.

All shift and rotate instructions operate on either a register or

on memory. They can be either 1 bit shifts:

sal cx, 1

ror variable1, 1

shr bl, 1

or shifts indexed by CL **(it must be CL)**:

rcl variable2, cl

sar si, cl

rol ah, cl

**SHR and SAR**

SHR destination,count

0 -> destination -> CF

Shifts the bits in destination to the right by the number of positions

specified in the count operand, (or in cl, if no count operand is

included). 0's are shifted in on the left. If the sign bit retains

its original value the Overflow flag is cleared; it is set if the sign

changes. The Carry flag is updated to reflect the last bit shifted.

Unlike the left shift instruction, there are two completely

different right shift instructions. SHR (shift logical right)

shifts the bits to the right, setting CF if a 1 bit is pushed off

the right end. It puts 0s in the leftmost bit. It is dividing

by two and is once again MUCH faster than division. For a single

shift, the remainder is in CF. For a shift of more than one bit,

you lose the remainder, but there is a way around this which we

will discuss in a moment.

If you want to divide by 16, you will shift right four times, so

you'll lose those 4 bits. But those bits are exactly the value of

the remainder. All we need to do is:

mov dx, ax ; copy of number to dx

and dx, 0000000000001111b ; remainder in dx

mov cl, 4 ; shift right 4 bits

shr ax, cl ; quotient in ax

Using a [mask](http://www.woodmann.com/crackz/Tutorials/Drme2.htm#MASK), we keep only the right four bits, which is the

remainder.

**SAR**

SAR destination,count

SF -> destination -> CF

SAR (shift arithmetic right) is different. It shifts right like

SHR, but the leftmost bit always stays the same. The overflow flag

will never change since the left bit will always stay the same.

SAR shifts the word or byte in destination to the right by the number

of bit positions specified in the second operand, COUNT. As bits are

transferred out the right (low-order) end of the destination, bits

equal to the original sign bit are shifted into the left (high-order)

end, thereby preserving the sign bit. The Carry flag is set equal to

the last bit shifted out of the right end.

**SAR is an instruction for doing signed division by 2 (sort of).**

**It is, however, an incomplete instruction. The rule for SAR is:**

**SAR gives the correct answer if the number is positive. It gives**

**the correct answer if the number is negative and the remainder is**

**zero. If the number is negative but there is a remainder, then**

**the answer is one too negative.**

You will never or almost never use SAR for signed division,

while you will find lots of opportunity to use SHR and SHL

for unsigned multiplication and division.

**Summary**

SHR (shift logical right) does the same thing as SHL but in

the opposite direction. Bits are shifted right. 0s are

placed in the high bit. Bits are shoved off the register (or

memory data) on the right side and CF indicates whether the

last bit shoved off was a 0 or a 1. It is used for dividing

an unsigned number by powers of 2.

SAR (shift arithmetic right) shifts bits right. The high

(sign) bit stays the same throughout the operation. Bits are

shoved off the register (or memory data) on the right side.

CF indicates whether the last bit shoved off was a 1 or a 0.

It is used (with difficulty) for dividing a signed number by

powers of 2.

**ROR and ROL**

ROR destination,count

ROR shifts the word or byte at the destination to the right by

the number of bit positions specified in the second operand, COUNT.

--------<------

| |

-> destination ---> CF

As bits are transferred out the right (low-order) end of the

destination, they re-enter on the left (high-order) end. The Carry

flag is updated to match the last bit shifted out of the right end.

ROL destination,count

CF <--- destination <--

| |

------->----------

As bits are transferred at the left (high-order) end of the

destination, they re-enter on the right (low-order) end. The Carry

flag is updated to match the last bit shifted out of the left end.

ROR (rotate right) and ROL (rotate left) rotate the bits around

the register. The only flags that are defined are OF and CF. OF

is set if the high bit changes, and CF is set if a 1 bit moves

off the end of the register to the other side.

**Summary**

ROR and ROL

ROR (rotate right) and ROL (rotate left) rotate the bits of

a register (or memory data) right and left respectively. The

bit which is shoved off one end is moved to the other end.

CF indicates whether the last bit moved from one end to the

other was a 1 or a 0.

**RCR and RCL**

RCR destination,count

--------<----------

| |

-> destination -> CF

RCR shifts the word or byte at the destination to the right by

the number of bit positions specified in the second operand,COUNT.

A bit shifted out of the right (low-order) end of the destination

enters the Carry flag, and the displaced Carry flag rotates around

to enter the vacated left-most bit position of the destination. This

"bit rotation" continues the number of times specified in COUNT.

Another way of looking at this is to consider the Carry flag as the

lowest order bit of the word being rotated.

RCL destination,count

---------->----------

| |

CF <- destination <-

Another way of looking at this instruction is to consider the Carry

flag as the highest order bit of the word being rotated.

RCR (rotate through carry right) and RCL (rotate through carry

left) rotate the same as the above instructions except that the

carry flag is involved. Rotating right, the low bit moves to CF,

the carry flag and CF moves to the high bit. Rotating left, the

high bit moves to CF and CF moves to the low bit. There are 9

bits (or 17 bits for a word) involved in the rotation. There are only

two flags defined, OF and CF. Obviously, CF is set if there is

something in it. OF is wierd. In RCL (the opposite instruction to

the one we are using), OF operates normally, signalling a change

in the top (sign) bit. In RCR, OF signals a change in CF. Why? I

don't have the slightest idea. You really have no need for the OF

flag anyways, so this is unimportant.

**Summary**

RCR and RCL

RCR (rotate through carry right) and RCL (rotate through

carry left) rotate the bits of a register (or of memory

data) right and left respectively. The bit which is shoved

off the register (or data) is placed in CF and the old CF is

placed on the other side of the register (or data).

Well, those are the seven instructions, but what can you do with

them besides multiply and divide?

First, you can work with multiple bit data. The 8087 has a word

length register called the status register. Looking at the upper

byte:

15 14 13 12 11 10 9 8

X X X

bits 11, 12 and 13 contain a number from 0 to 7. The data in this

register is not directly accessable. You need to move the

register into memory, then into an 8086 register. If you want to

find what this number is, what do you do?

mov bx, status\_register\_data

mov cl, 3

ror bx, cl

and bh, 00000111b

we rotate right 3 and then mask off everything else. The number

is now in BH. We could have used SHR if we wanted. Another 8087

register is the control register. In the upper byte it has:

15 14 13 12 11 10 9 8

X X

a number from 0 to 3 in bits 10 and 11. If we want the

information, we do the same thing:

mov bx, control\_register\_data

mov cl, 2

ror bx, cl

and bh, 00000011b

and the number is in BH.

One thing to know is that just inside a loop we must push CX.

That is because we use CL for the ROL instruction. It is then

POPped just before the loop instruction. This is typical. CX is

the only register that can be used for counting in indexed

instructions. It is common for indexing instructions to be

nested, so you temporarily store the old value of CX while you

are using CX for something different.

push cx ; typical code for a shift

mov cl, 7

shr si, cl

pop cx

**INC**

INC increments a register or a variable by 1.

inc ax

inc variable1

**DEC**

DEC decrements a register or a variable by 1.

dec ax

dec variable1