CHAPTER 6: CONDITIONAL PROCESSING

Chapter Overview

- **Boolean and Comparison Instructions**
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Boolean and Comparison Instructions

- CPU Status Flags
- AND Instruction
- OR Instruction
- XOR Instruction
- NOT Instruction
- Applications
- TEST Instruction
- CMP Instruction

Status Flags - Review

- The Zero flag is set when the result of an operation equals zero.
- The Carry flag is set when an instruction generates a result that is too large (or too small) for the destination operand.
- The Sign flag is set if the destination operand is negative, and it is clear if the destination operand is positive.
- The Overflow flag is set when an instruction generates an invalid signed result.
- The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
- The Auxiliary Carry flag is set when an operation produces a carry out from bit 3 to bit 4

AND Instruction

- Performs a Boolean AND operation between each pair of matching bits in two operands
- Syntax:

AND *destination, source*

(same operand types as MOV)

```
         0 0 1 1 1 0 1 1
AND      0 0 0 0 1 1 1 1
         ---------------
cleared  0 0 0 0 1 0 1 1  unchanged
```

AND

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

OR Instruction

- Performs a Boolean OR operation between each pair of matching bits in two operands
- Syntax:

  OR *destination, source*

```
               0 0 1 1 1 0 1 1
          OR   0 0 0 0 1 1 1 1
               ---------------
unchanged ---  0 0 1 1 1 1 1 1  --- set
```

OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Instruction

- Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands
- Syntax:

XOR *destination, source*

```
                 0 0 1 1 1 0 1 1
           XOR   0 0 0 0 1 1 1 1
                 ---------------
unchanged ----  |0 0 1 1|0 1 0 0| ---- inverted
```

XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

XOR is a useful way to toggle (invert) the bits in an operand.

NOT Instruction

- Performs a Boolean NOT operation on a single destination operand
- Syntax:

  NOT *destination*

NOT 0 0 1 1 1 0 1 1

1 1 0 0 0 1 0 0 inverted

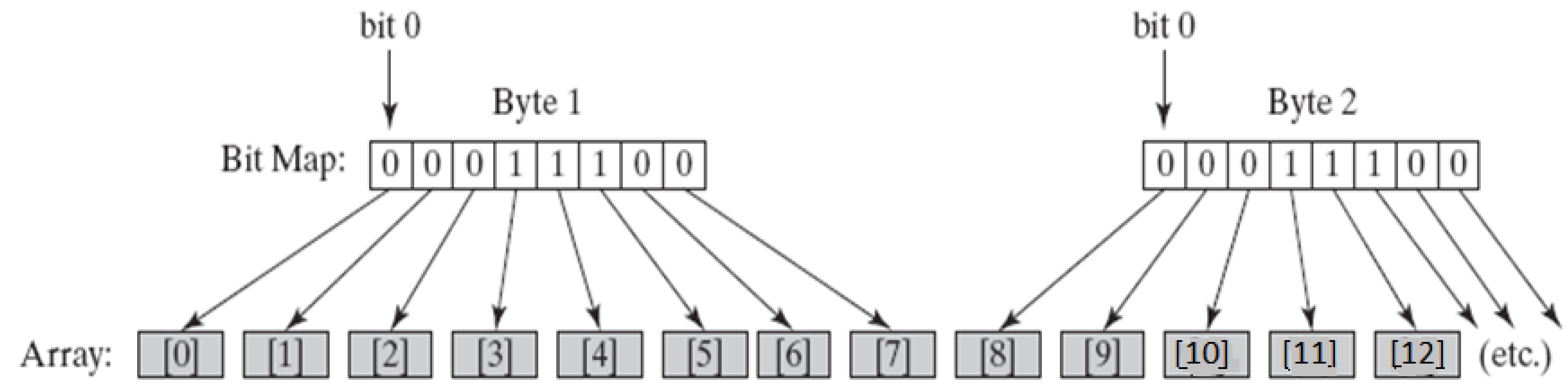
NOT

X	¬X
F	T
T	F

Bit-Mapped Sets

- Binary bits indicate set membership
- Efficient use of storage
- Also known as *bit vectors*

Figure 6–1 Mapping Binary Bits to an Array.



Bit-Mapped Set Operations

- Set Complement
  - mov eax,SetX
  - not eax
- Set Intersection
  - mov eax,setX
  - and eax,setY
- Set Union
  - mov eax,setX
  - or eax,setY

TEST Instruction

- Performs a nondestructive (非破坏性的) AND operation between each pair of matching bits in two operands
- No operands are modified, but the Zero flag is affected.
- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
test al,00000011b
jz   ValueNotFound
```

- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test al,00000011b
jnz  ValueFound
```

CMP Instruction

- Compares the destination operand to the source operand
  - Nondestructive subtraction of source from destination (destination operand is not changed)
- Syntax: CMP *destination, source*
- Example: destination == source

```
mov al,5
cmp al,5                    ; Zero flag set
```

- Example: destination < source

```
mov al,4
cmp al,5                    ; Carry flag set
```

What's Next

- Boolean and Comparison Instructions
- **Conditional Jumps**
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Conditional Jumps

- Jumps Based On . . .
  - Specific flags
  - Equality
  - Unsigned comparisons
  - Signed Comparisons
- Applications
- Encrypting a String
- Bit Test (BT) Instruction

J*cond* Instruction

- A conditional jump instruction branches to a label when specific register or flag conditions are met
- Examples:
  - JB, JC jump to a label if the Carry flag is set
  - JE, JZ jump to a label if the Zero flag is set
  - JS jumps to a label if the Sign flag is set
  - JNE, JNZ jump to a label if the Zero flag is clear
  - JECXZ jumps to a label if ECX equals 0

J*cond* Ranges

- Prior to the 386:
  - jump must be within –128 to +127 bytes from current location counter
- x86 processors:
  - 32-bit offset permits jump anywhere in memory

Offset	Encoding		ASM Source
0040101A	**B0 80**	**mov al,80h**	L1: mov al, -128
0040101C	**3C 0A**	**cmp al,0Ah**	cmp al, 10
0040101E	**74 FA**	**je L1 (40101Ah)**	jz L1
00401020	**8A D8**	**mov bl,al**	mov bl, al

FA: -6

```
  00401020
+ FFFFFFFA
------------
  0040101A
```

Jumps Based on Specific Flags

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Jumps Based on Equality

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Jumps Based on Unsigned Comparisons

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

Jumps Based on Signed Comparisons

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

Applications

- Task: Jump to a label if unsigned EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
cmp eax,ebx
ja  Larger
```

- Task: Jump to a label if signed EAX is greater than EBX
- Solution: Use CMP, followed by JG

```
cmp eax,ebx
jg  Greater
```

BT (Bit Test) Instruction

- Copies bit *n* from an operand into the Carry flag
- Syntax: BT *bitBase, n*
  - bitBase may be *r/m16* or *r/m32*
  - n may be *r16, r32*, or *imm8*
- Example: jump to label L1 if bit 9 is set in the AX register:

```
bt AX,9                              ; CF = bit 9
jc L1                                ; jump if Carry
```

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- **Conditional Loop Instructions**
- Conditional Structures
- Application: Finite-State Machines
- Conditional Control Flow Directives

Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE

LOOPZ and LOOPE

- Syntax:

  LOOPE *destination*

  LOOPZ *destination*

- Logic:
  - ECX $\leftarrow$ ECX – 1
  - if ECX > 0 and ZF=1, jump to *destination*
- Useful when scanning an array for the first element that does not match a given value.

In 32-bit mode, ECX is the loop counter register. In 16-bit real-address mode, CX is the counter, and in 64-bit mode, RCX is the counter.

LOOPNZ and LOOPNE

- LOOPNZ (LOOPNE) is a conditional loop instruction
- Syntax:

  LOOPNZ *destination*

  LOOPNE *destination*
- Logic:
  - ECX $\leftarrow$ ECX – 1;
  - if ECX > 0 and ZF=0, jump to *destination*
- Useful when scanning an array for the first element that matches a given value.

LOOPNZ Example

The following code finds the first positive value in an array:

```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
   mov esi,OFFSET array
   mov ecx,LENGTHOF array
next:
   test WORD PTR [esi],8000h  ; test sign bit
   pushfd                     ; push flags on stack
   add esi,TYPE array
   popfd                      ; pop flags from stack
   loopnz next                ; continue loop
   jnz quit                   ; none found
   sub esi,TYPE array         ; ESI points to value
quit:
```

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- **Conditional Structures**
- Application: Finite-State Machines
- Conditional Control Flow Directives

Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops

Block-Structured IF Statements

Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
```

```
    mov eax,op1
    cmp eax,op2
    jne L1
    mov X,1
    jmp L2
L1: mov X,2
L2:
```

Compound Expression with AND (1 of 3)

- When implementing the logical AND operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is false, the second expression is skipped:

```
if (al > bl) AND (bl > cl)
  X = 1;
```

Compound Expression with AND (2 of 3)

```
if (al > bl) AND (bl > cl)
  X = 1;
```

This is one possible implementation . . .

```
    cmp al,bl                      ; first expression...
    ja  L1
    jmp next
L1:
    cmp bl,cl                      ; second expression...
    ja  L2
    jmp next
L2:                                ; both are true
    mov X,1                        ; set X to 1
next:
```

Compound Expression with AND (3 of 3)

```
if (al > bl) AND (bl > cl)
  X = 1;
```

But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
    cmp al,bl                  ; first expression...
    jbe next                   ; quit if false
    cmp bl,cl                  ; second expression...
    jbe next                   ; quit if false
    mov X,1                    ; both are true
next:
```

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- **Application: Finite-State Machines**
  - **Reading material**
- Conditional Control Flow Directives

What's Next

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Loop Instructions
- Conditional Structures
- Application: Finite-State Machines
- **Conditional Control Flow Directives**

Creating IF Statements

- Runtime Expressions
- Relational and Logical Operators
- MASM-Generated Code
- .REPEAT Directive
- .WHILE Directive

Runtime Expressions

- .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime expressions and create block-structured IF statements.
- Examples:

```
.IF eax > ebx
    mov edx,1
.ELSE
    mov edx,2
.ENDIF
```

```
.IF eax > ebx && eax > ecx
    mov edx,1
.ELSE
    mov edx,2
.ENDIF
```

- MASM generates "hidden" code for you, consisting of code labels, CMP and conditional jump instructions.

Relational and Logical Operators

Operator	Description
expr1 == *expr2*	Returns true when *expression1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 >= *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 <= *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

MASM-Generated Code

```
.data
val1   DWORD 5
result DWORD ?
.code
mov eax,6
.IF eax > val1
  mov result,1
.ENDIF
```

Generated code:

```
    mov eax,6
    cmp eax,val1
    jbe @C0001
    mov result,1
@C0001:
```

MASM automatically generates an unsigned jump (JBE) because val1 is unsigned.

```
.IF 2 > eax
```

MASM-Generated Code

```
.data
val1   SDWORD 5
result SDWORD ?
.code
mov eax,6
.IF eax > val1
  mov result,1
.ENDIF
```

Generated code:

```
    mov eax,6
    cmp eax,val1
    jle @C0001
    mov result,1
@C0001:
```

MASM automatically generates a signed jump (JLE) because val1 is signed.

MASM-Generated Code

```
.data
result DWORD ?
.code
mov ebx,5
mov eax,6
.IF eax > ebx
  mov result,1
.ENDIF
```

Generated code:

```
    mov ebx,5
    mov eax,6
    cmp eax,ebx
    jbe @C0001
    mov result,1
@C0001:
```

MASM automatically generates an unsigned jump (JBE) when both operands are registers . . .

MASM-Generated Code

```
.data
result SDWORD ?
.code
mov ebx,5
mov eax,6
.IF SDWORD PTR eax > ebx
  mov result,1
.ENDIF
```

Generated code:

```
    mov ebx,5
    mov eax,6
    cmp eax,ebx
    jle @C0001
    mov result,1
@C0001:
```

. . . unless you prefix one of the register operands with the SDWORD PTR operator. Then a signed jump is generated.

.REPEAT Directive

Executes the loop body before testing the loop condition associated with the .UNTIL directive.

Example:

```
; Display integers 1 – 10:

mov eax,0
.REPEAT
    inc eax
    call WriteDec
    call Crlf
.UNTIL eax == 10
```

.WHILE Directive

Tests the loop condition before executing the loop body
The .ENDW directive marks the end of the loop.

Example:

```
; Display integers 1 – 10:

mov eax,0
.WHILE eax < 10
    inc eax
    call WriteDec
    call Crlf
.ENDW
```

CHAPTER 7:
INTEGER ARITHMETIC

Chapter Overview

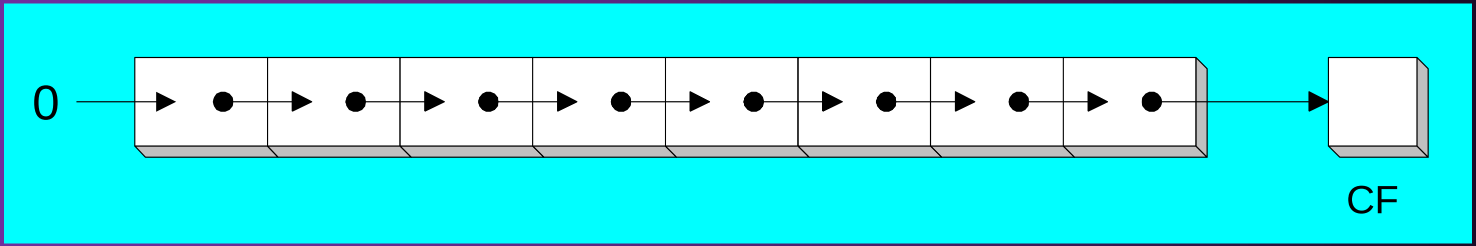
- **Shift and Rotate Instructions**
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

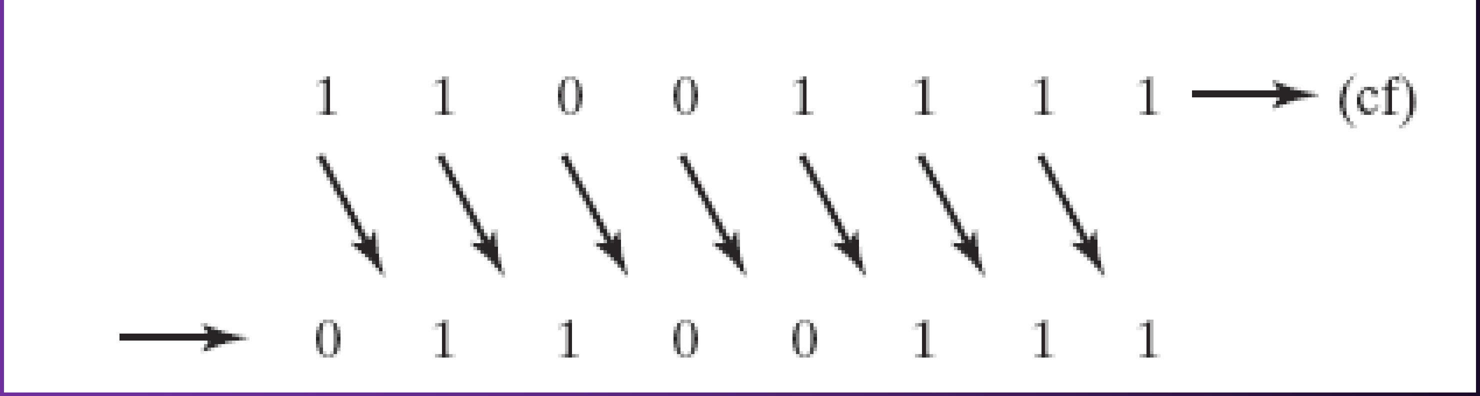
7.1 Shift and Rotate Instructions

- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- SAL and SAR Instructions
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

Logical Shift

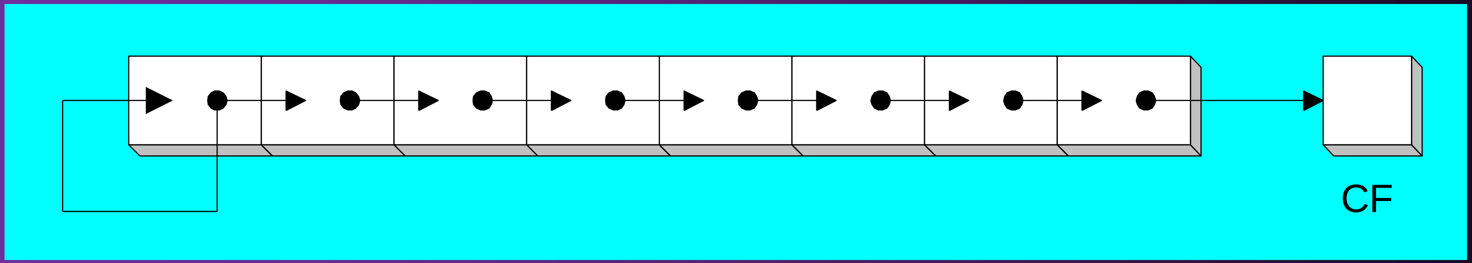
- A logical shift fills the newly created bit position with zero:

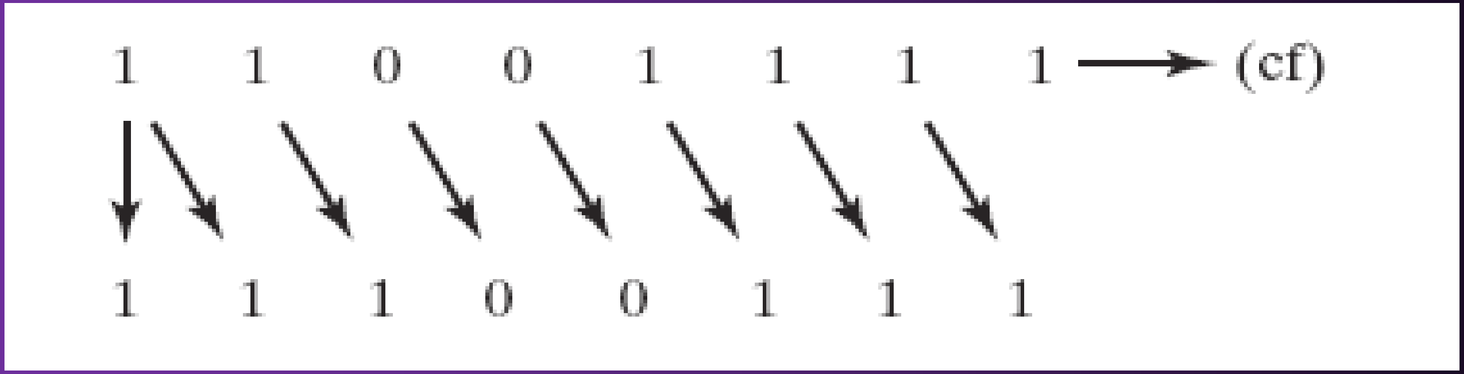




Arithmetic Shift

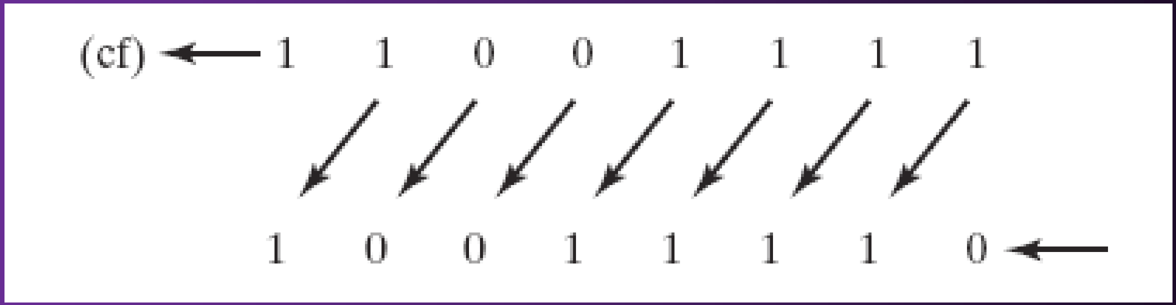
- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:





SHL Instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.



- Operand types for SHL:

```
SHL reg,imm8
SHL mem,imm8
SHL reg,CL
SHL mem,CL
```

(Same for all shift and rotate instructions)

Fast Multiplication

Shifting left 1 bit multiplies a number by 2

```
mov dl,5
shl dl,1
```

Before:	0 0 0 0 0 1 0 1	= 5
After:	0 0 0 0 1 0 1 0	= 10

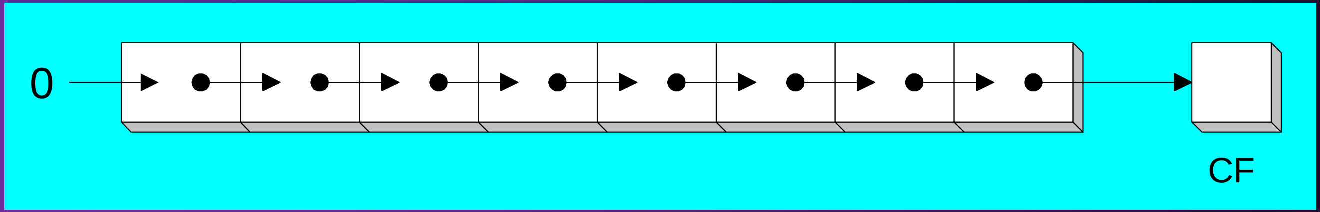
Shifting left *n* bits multiplies the operand by 2^n

For example, $5 * 2^2 = 20$

```
mov dl,5
shl dl,2                          ; DL = 20
```

SHR Instruction

- The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.

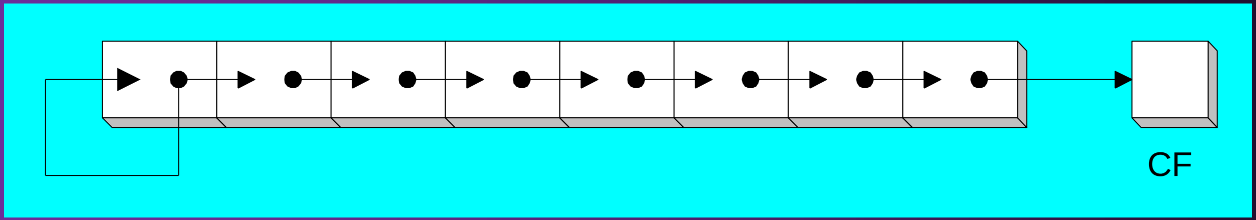


Shifting right *n* bits divides the operand by 2^n

```
mov dl,80
shr dl,1                        ; DL = 40
shr dl,2                        ; DL = 10
```

SAL and SAR Instructions

- SAL (shift arithmetic left) is identical to SHL.
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.

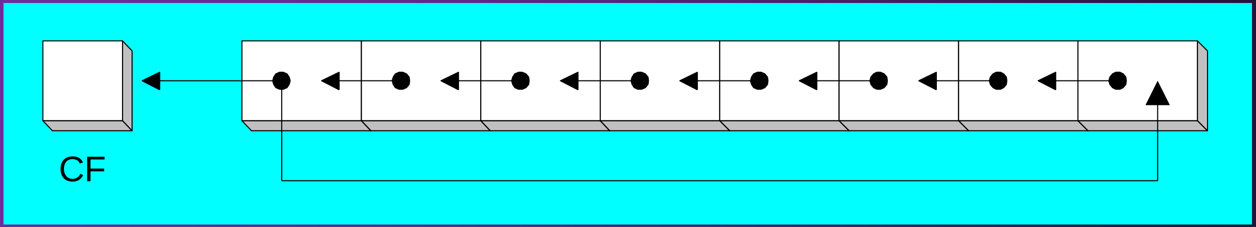


An arithmetic shift preserves the number's sign.

```
mov dl,-80
sar dl,1                            ; DL = -40
sar dl,2                            ; DL = -10
```

ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost

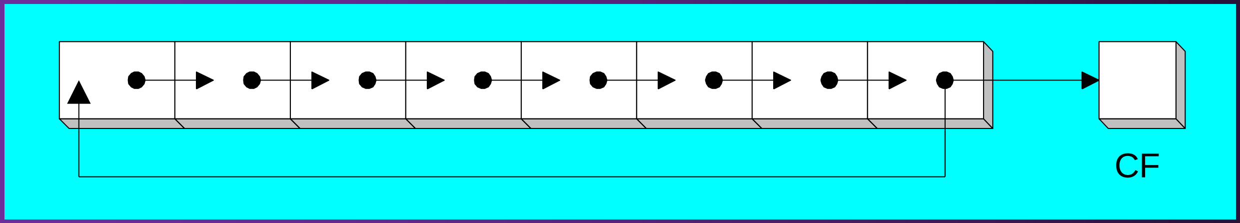



```
mov al,11110000b
rol al,1                          ; AL = 11100001b

mov dl,3Fh
rol dl,4                          ; DL = F3h
```

ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost

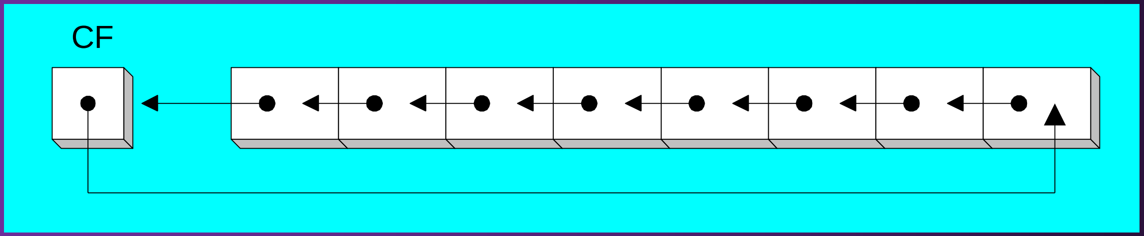



```
mov al,11110000b
ror al,1                           ; AL = 01111000b

mov dl,3Fh
ror dl,4                           ; DL = F3h
```

RCL Instruction

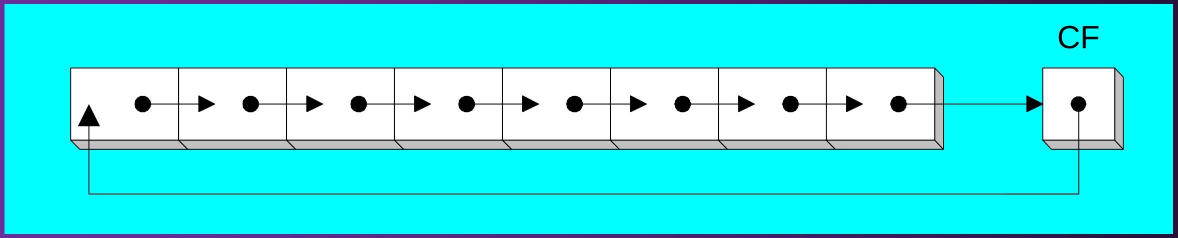
- RCL (rotate carry left) shifts each bit to the left
- Copies the most significant bit to the Carry flag
- Copies the Carry flag to the least significant bit




```
clc                 ; CF = 0
mov bl,88h          ; CF,BL = 0 10001000b
rcl bl,1            ; CF,BL = 1 00010000b
rcl bl,1            ; CF,BL = 0 00100001b
```

RCR Instruction

- RCR (rotate carry right) shifts each bit to the right
- Copies the least significant bit to the Carry flag
- Copies the Carry flag to the most significant bit




```
stc                      ; CF = 1
mov ah,10h               ; CF,AH = 1 00010000b
rcr ah,1                 ; CF,AH = 0 10001000b
```

SHLD Instruction (Shift Left Double)

- Shifts a destination operand a given number of bits to the left
- The bit positions opened up by the shift are filled by the most significant bits of the source operand
- The source operand is not affected
- Syntax:

  SHLD *destination, source, count*

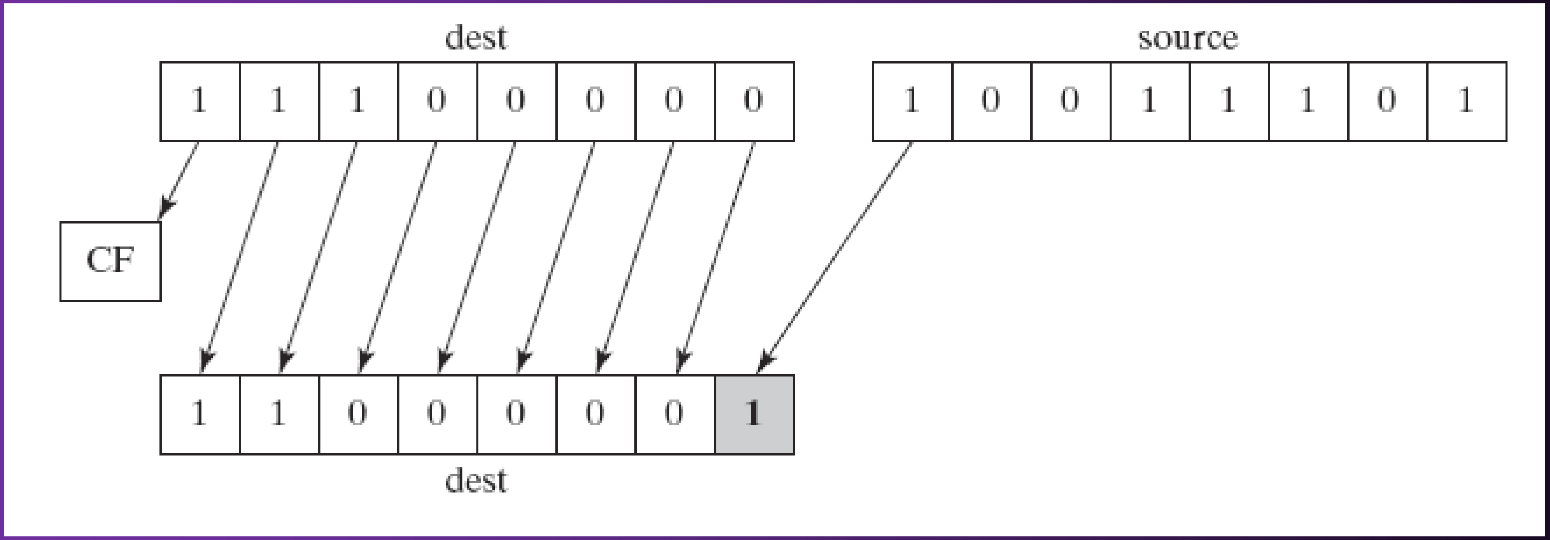
- Operand types:

```
SHLD reg16/32, reg16/32, imm8/CL
SHLD mem16/32, reg16/32, imm8/CL
```

SHLD Example

Shift count of 1:

```
mov al,11100000b
mov bl,10011101b
shld al,bl,1
```



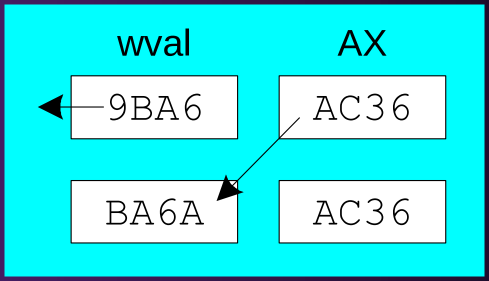
Another SHLD Example

Shift wval 4 bits to the left and replace its lowest 4 bits with the high 4 bits of AX:

```
.data
wval WORD 9BA6h
.code
mov  ax,0AC36h
shld wval,ax,4
```

Before:

After:



SHRD Instruction

- Shifts a destination operand a given number of bits to the right
- The bit positions opened up by the shift are filled by the least significant bits of the source operand
- The source operand is not affected
- Syntax:

  SHRD *destination, source, count*

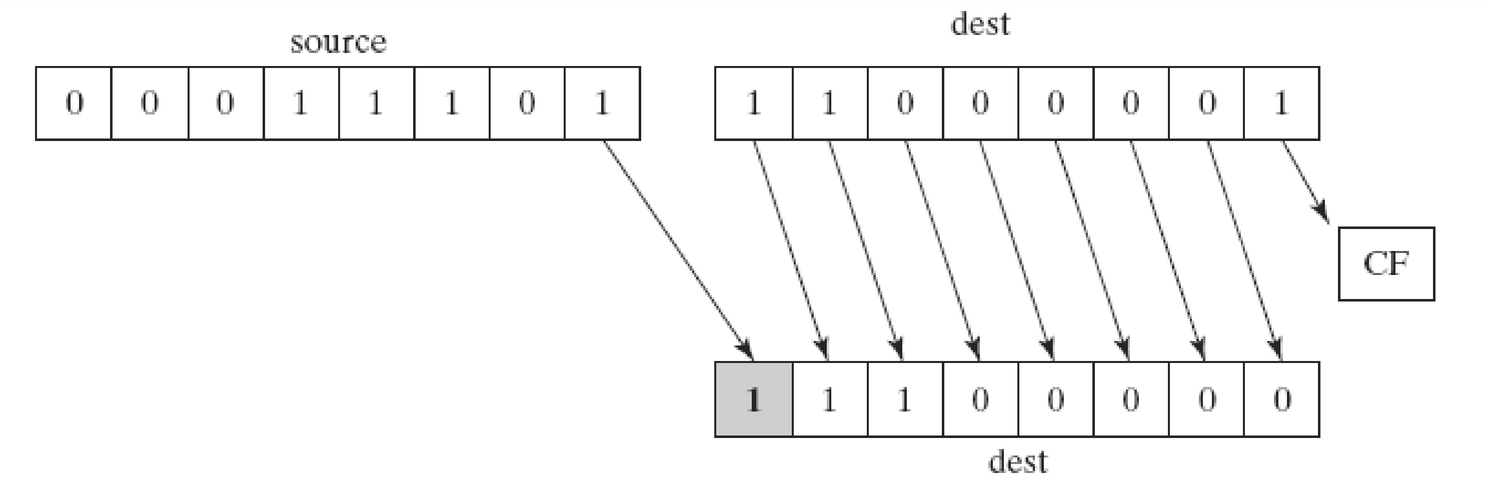
- Operand types:

```
SHRD reg16/32, reg16/32, imm8/CL
SHRD mem16/32, reg16/32, imm8/CL
```

SHRD Example

Shift count of 1:

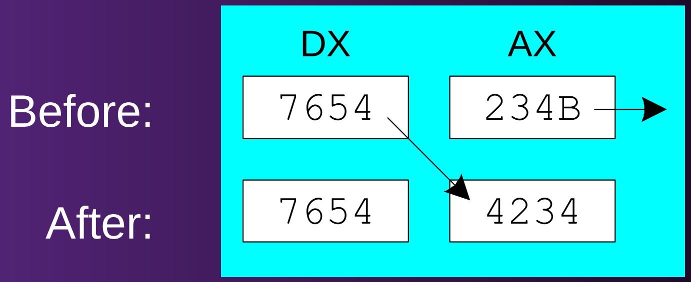
```
mov al,11000001b
mov bl,00011101b
shrd al,bl,1
```



Another SHRD Example

Shift AX 4 bits to the right and replace its highest 4 bits with the low 4 bits of DX:

```
mov  ax,234Bh
mov  dx,7654h
shrd ax,dx,4
```



What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- **Multiplication and Division Instructions**
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

7.2 Multiplication and Division Instructions

- MUL Instruction
- IMUL Instruction
- DIV Instruction
- Signed Integer Division
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

MUL Instruction

- In 32-bit mode, MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.

- The instruction formats are:

```
MUL r/m8
MUL r/m16
MUL r/m32
```

Implied operands:

Multiplicand	Multiplier	Product
AL	*r/m8*	AX
AX	*r/m16*	DX:AX
EAX	*r/m32*	EDX:EAX

64-Bit MUL Instruction

- In 64-bit mode, MUL (unsigned multiply) instruction multiplies a 64-bit operand by RAX, producing a 128-bit product.

- The instruction formats are:

```
MUL r/m64
```

Example:

```
mov rax,0FFFF0000FFFF0000h
mov rbx,2
mul rbx       ; RDX:RAX = 0000000000000001FFFE0001FFFE0000
```

MUL Examples

100h * 2000h, using 16-bit operands:

```
.data
val1 WORD 2000h
val2 WORD 100h
.code
mov ax,val1
mul val2      ; DX:AX = 00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

12345h * 1000h, using 32-bit operands:

```
mov eax,12345h
mov ebx,1000h
mul ebx       ; EDX:EAX = 0000000012345000h, CF=0
```

IMUL Instruction

- IMUL (signed integer multiply) multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX
- Preserves the sign of the product by sign-extending it into the upper half of the destination register

Example: multiply 48 * 4, using 8-bit operands:

```
mov  al,48
mov  bl,4
imul bl             ; AX = 00C0h, OF=1
```

OF=1 because AH is not a sign extension of AL.

DIV Instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
- A single operand is supplied (register or memory operand), which is assumed to be the divisor
- Instruction formats:

```
DIV r/m8
DIV r/m16
DIV r/m32
```

Default Operands:

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

DIV Examples

Divide 8003h by 100h, using 16-bit operands:

```
mov dx,0                ; clear dividend, high
mov ax,8003h            ; dividend, low
mov cx,100h             ; divisor
div cx                  ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands:

```
mov edx,0               ; clear dividend, high
mov eax,8003h           ; dividend, low
mov ecx,100h            ; divisor
div ecx                 ; EAX = 00000080h, DX = 3
```

64-Bit DIV Example

Divide 000001080000000033300020h by 00010000h:

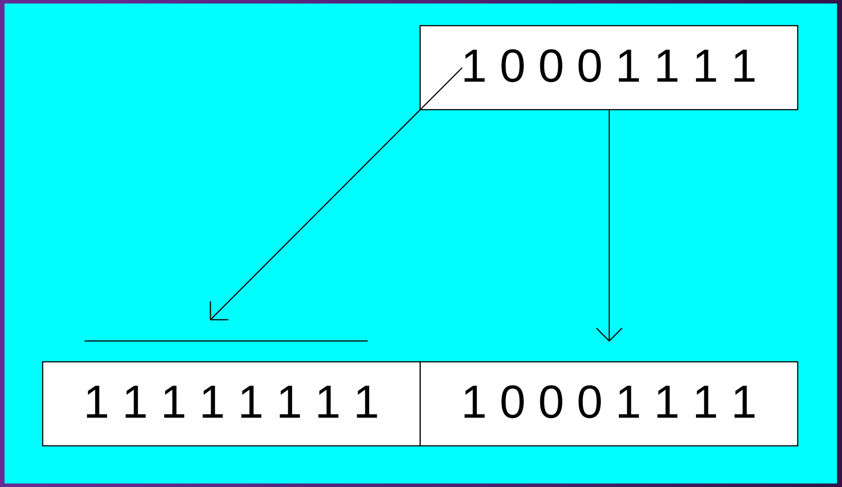
```
.data
dividend_hi QWORD 00000108h
dividend_lo QWORD 33300020h
divisor QWORD 00010000h

.code
mov rdx, dividend_hi
mov rax, dividend_lo
div divisor                          ; RAX = quotient
                                     ; RDX = remainder
```

RAX (quotient): 0108000000003330h
RDX (remainder): 0000000000000020h

Signed Integer Division (IDIV)

- Signed integers must be sign-extended before division takes place
  - fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit
- For example, the high byte contains a copy of the sign bit from the low byte:



CBW, CWD, CDQ Instructions

- The CBW, CWD, and CDQ instructions provide important sign-extension operations:
  - CBW (convert byte to word) extends AL into AH
  - CWD (convert word to doubleword) extends AX into DX
  - CDQ (convert doubleword to quadword) extends EAX into EDX
- Example:

```
mov eax,0FFFFFF9Bh      ; (-101)
cdq            ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

IDIV Instruction

- IDIV (signed divide) performs signed integer division
- Same syntax and operands as DIV instruction

Example: 8-bit division of –48 by 5

```
mov  al,-48
cbw                    ; extend AL into AH
mov  bl,5
idiv bl                ; AL = -9,  AH = -3
```

What's Next

- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- **Extended Addition and Subtraction**
- ASCII and UnPacked Decimal Arithmetic
- Packed Decimal Arithmetic

7.3 Extended Addition and Subtraction

- ADC Instruction
- Extended Precision Addition
- SBB Instruction
- Extended Precision Subtraction

The instructions in this section do not apply to 64-bit mode programming.

CHAPTER 8: ADVANCED PROCEDURES

Chapter Overview

- **Stack Frames**
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive

Stack Frame (堆栈框架, 栈帧)

- Also known as an *activation record* (活动记录)
- Area of the stack set aside for a procedure's passed parameters, return address, saved registers, and local variables
- Created by the following steps:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP to ESP.
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

Stack Parameters

- More convenient than register parameters
- Two possible ways of calling DumpMem. Which is easier?

```
pushad
mov esi,OFFSET array
mov ecx,LENGTHOF array
mov ebx,TYPE array
call DumpMem
popad
```

```
push TYPE array
push LENGTHOF array
push OFFSET array
call DumpMem
```

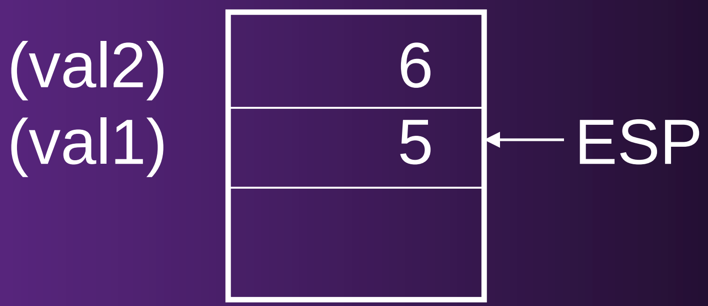
Passing Arguments by Value

- Push argument values on stack
  - (Use only 32-bit values in protected mode to keep the stack aligned)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called-procedure did not remove them

Example

```
.data
val1  DWORD 5
val2  DWORD 6

.code
push val2
push val1
```



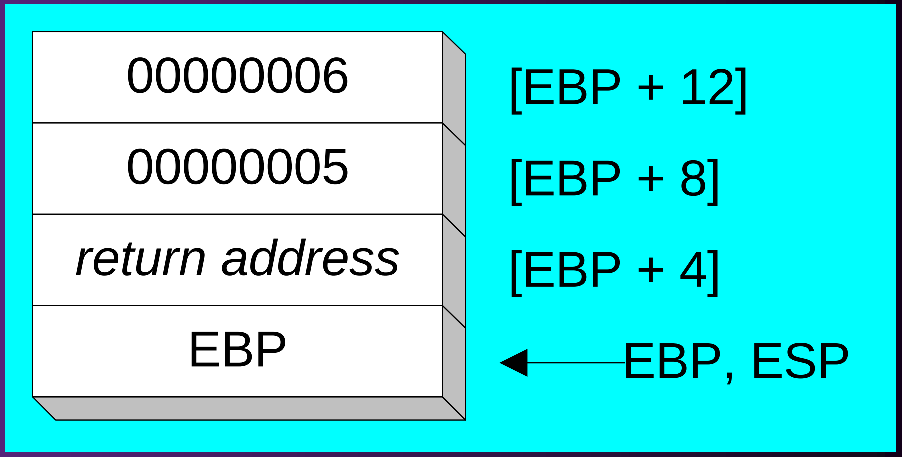
Stack prior to CALL

Passing Arguments by Value: AddTwo

int n = AddTwo(5, 6);

```
.data
sum DWORD ?
.code
    push 6                      ; second argument
    push 5                      ; first argument
    call AddTwo                 ; EAX = sum
    mov  sum,eax                ; save the sum
```

```
AddTwo PROC
    push ebp
    mov  ebp,esp
    .
    .
```



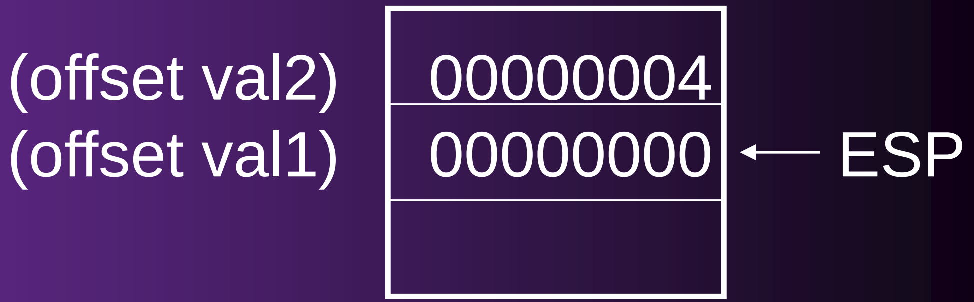
Passing by Reference

- Push the **`offsets of arguments`** on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called procedure did not remove them

Example

```
.data
val1  DWORD 5
val2  DWORD 6

.code
push OFFSET val2
push OFFSET val1
```



Stack prior to CALL

Stack after the CALL

value or addr of val2	[EBP+12]
value or addr of val1	[EBP+8]
return address	[EBP+4]
EBP	← ESP, EBP

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP[1].
  - Example: [ebp + 8]
- EBP is called the base pointer or frame pointer because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

[1] BP in Real-address mode

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive

RET Instruction

- *Return from subroutine*
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
  - **RET**
  - **RET *n***
- Optional operand *n* causes *n* bytes to be added to the **stack pointer** after EIP (or IP) is assigned a value.

Who removes parameters from the stack?

Caller (C) ...... or ...... Called-procedure (STDCALL):

```
push val2
push val1
call AddTwo
add   esp,8
```

```
AddTwo PROC
    push  ebp
    mov   ebp,esp
    mov   eax,[ebp+12]
    add   eax,[ebp+8]

    pop   ebp
    ret   8
```

(Covered later: The MODEL directive specifies calling conventions)

C Call : Caller releases stack

RET does not clean up the stack.

```
AddTwo_C PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 12]     ; second parameter
    add  eax,[ebp + 8]      ; first parameter
    pop  ebp
    ret                     ; caller cleans up the stack
AddTwo_C ENDP

_Example1 PROC
       push 6
       push 5
       call AddTwo_C
       add  esp,8           ; clean up the stack
       call DumpRegs        ; sum is in EAX
       ret
_Example1 ENDP
```

STDCall : Procedure releases stack

The RET n instruction cleans up the stack.

```
AddTwo PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 12]     ; second parameter
    add  eax,[ebp + 8]      ; first parameter
    pop  ebp
    ret  8                  ; clean up the stack
AddTwo ENDP

_Example2 PROC
       push 6
       push 5
       call AddTwo
       call DumpRegs        ; sum is in EAX
       ret
_Example2 ENDP
```

Summary (Chap 6)

- Bitwise instructions (AND, OR, XOR, NOT, TEST)
  - manipulate individual bits in operands
- CMP – compares operands using implied subtraction
  - sets condition flags
- Conditional Jumps & Loops
  - equality: JE, JNE
  - flag values: JC, JZ, JNC, JP, ...
  - signed: JG, JL, JNG, ...
  - unsigned: JA, JB, JNA, ...
  - LOOPZ, LOOPNZ, LOOPE, LOOPNE

Summary (Chap 7)

- Shift and rotate instructions are some of the best tools of assembly language
  - finer control than in high-level languages
  - SHL, SHR, SAR, ROL, ROR, RCL, RCR
- MUL and DIV – integer operations
  - close relatives of SHL and SHR
  - CBW, CDQ, CWD: preparation for division
- Extended precision arithmetic: ADC, SBB

Summary (Chap 8)

- Stack parameters
  - more convenient than register parameters
  - passed by value or reference

Homework

- Reading Chap 6 -- 8
- Exercises

Thanks!

- Happy National Day!