Chapter Overview

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

Lecture 1

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 (bit 7 carry is XORed with bit 6 Carry).
- 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

Conditional jumps will use these flags

Irvine, Kip R. Assembly Language for x86 Processors 6/e, 2010.

AND Instruction

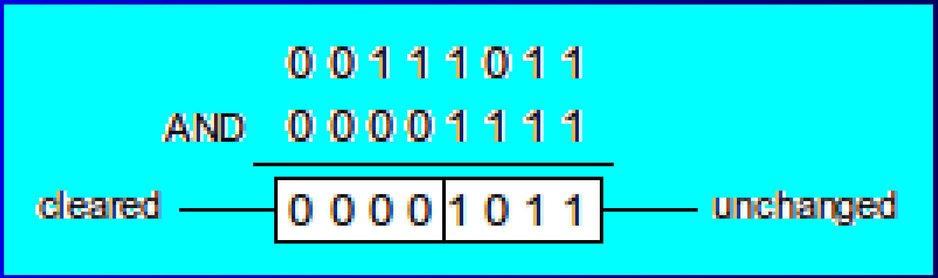
- The following operand combinations are permitted
  - AND reg, reg
  - AND reg, mem
  - AND reg, imm
  - AND mem, reg
  - AND mem, imm
- Operands can be 8, 16, or 32 bits
- Must be the same size
- For each matching bit-pair
  - If both bits equal 1, the result bit is 1
  - Otherwise it is 0
- Lets you clear one or more bits in an operand without affecting other bits (bit masking)

AND Instruction

- Performs a Boolean AND operation between each pair of matching bits in two operands
- Flags
  - Clears Overflow, Cary
  - Modifies Sign, Zero, and Parity
- Syntax:

```
AND destination, source
```

(same operand types as MOV)



AND

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

0 in source clears a bit, 1 leaves it unchanged
and AL, 11110110 ; clears bits 0 and 3, leaves others unchanged

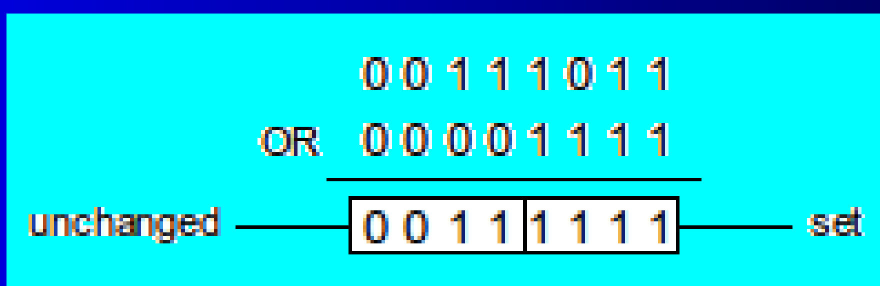
OR Instruction

- The following operand combinations are permitted
  - OR reg, reg
  - OR reg, mem
  - OR reg, imm
  - OR mem, reg
  - OR mem, imm
- Operands can be 8, 16, or 32 bits
- Must be the same size
- For each matching bit-pair
  - The result bit is 1 when at least one input bit is 1
  - Otherwise it is 0
- Useful when you want to set one or more bits without affecting the other bits

OR Instruction

- Performs a Boolean OR operation between each pair of matching bits in two operands
- Flags
  - Clears Overflow, Cary
  - Modifies Sign, Zero, and Parity
- Syntax:

```
OR destination, source
```



OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

1 in source set a bit, 0 leaves it unchanged
or AL, 00000100 ; sets bit 2, leaves others unchanged

Irvine, Kip R. Assembly Language for x86 Processors 6/e, 2010.

XOR Instruction

- Performs a Boolean exclusive-OR operation between each pair of matching bits in two operands
- XOR with 0 retains its value, with 1 reverses value
- Flags
  - Clears Overflow, Cary
  - Modifies Sign, Zero, and Parity
- 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
- The following operand combinations are permitted
  - NOT reg
  - NOT mem
- Flags
  - No flags are affected
- 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

Results called one's complement

Applications

- Task: Convert the character in AL to upper case.
- Solution: Use the AND instruction to clear bit 5.

```
mov al,'a'                  ; AL = 01100001b
and al,11011111b            ; AL = 01000001b
```

APPLICATION

- Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
- Solution: Use the OR instruction to set bits 4 and 5.

APPLICATION

- Task: Jump to a label if an integer is even.
- Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.

Applications

- Task: Jump to a label if the value in AL is not zero.
- Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

TEST Instruction

- Performs a nondestructive 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 either bit 0 or bit 1 in AL is set.

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

- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

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

Example

- The value 00001 001 in this example is called a bit mask.

```
0 0 1 0 0 1 0 1  <- input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 1  <- result:  ZF = 0
0 0 1 0 0 1 0 0   <- input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 0   <- result:  ZF = 1
```

Flags: The TEST instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in the same way as the AND instruction

CMP Instruction (1 of 3)

- 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
```

CMP Instruction (2 of 3)

- Example: destination > source

```
mov al,6
cmp al,5                    ; ZF = 0, CF = 0
```

(both the Zero and Carry flags are clear)

- CMP used to create conditional logic structure.

CMP Instruction (3 of 3)

The comparisons shown here are performed with signed integers.

- Example: destination > source

```
mov al,5
cmp al,-2            ; Sign flag == Overflow flag
```

- Example: destination < source

```
mov al,-1
cmp al,5             ; Sign flag != Overflow flag
```

What's Next

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

Lecture 2

Conditional Jumps

- Jumps Based On . . .
  - Specific flags
  - Equality
  - Unsigned comparisons
  - Signed Comparisons
- Applications
- Encrypting a String

Jcond Instruction

- Two steps to create a logic structure in ASM
  - Execute CMP, AND, or SUB to modify the CPU status flags
  - Execute conditional jump instruction
- A conditional jump instruction branches to a label when specific register or flag conditions are met

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

Irvine, Kip R. Assembly Language for x86 Processors 6/e, 2010.

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)

Irvine, Kip R. Assembly Language for x86 Processors 6/e, 2010.

YOUR TASKS (15 minutes)

1. Task: Jump to a label if unsigned EAX is greater than EBX

- Solution: Use CMP, followed by JA

2. Task: Jump to a label if signed EAX is greater than EBX

- Solution: Use CMP, followed by JG

3. Jump to label L1 if unsigned EAX is less than or equal to Val1

4. Jump to label L1 if signed EAX is less than or equal to Val1

5. Compare unsigned AX to BX, and copy the larger of the two into a variable named Large

6. Compare signed AX to BX, and copy the smaller of the two into a variable named Small

SOLUTION(5 AND 6)

```
    mov Large,bx
    cmp ax,bx
    jna Next
    mov Large,ax
Next:

    mov Small,ax
    cmp bx,ax
    jnl Next
    mov Small,bx
Next:
```

YOUR TASK

- Jump to label L1 if the memory word pointed to by ESI equals Zero

- Jump to label L2 if the doubleword in memory pointed to by EDI is even

YOUR TASK

- Task: Jump to label L1 if bits 0, 1, and 3 in AL are all set.
- Solution: Clear all bits except bits 0, 1,and 3. Then compare the result with 00001011 binary.

YOUR TASK

- Write code that jumps to label L1 if either bit 4, 5, or 6 is set in the BL register.
- Write code that jumps to label L1 if bits 4, 5, and 6 are all set in the BL register.
- Write code that jumps to label L2 if AL has even parity.
- Write code that jumps to label L3 if EAX is negative.
- Write code that jumps to label L4 if the expression (EBX – ECX) is greater than zero.

YOUR TASK(10 MINUTES)

WRITE A PROGRAM IN ASSEMBLY THAT PERFORMS ENCRYPTION BY TRANSFORMING EVERY CHARACTER OF STRING INTO A NEW CHARACTER.

- Tasks:
  - Input a message (string) from the user
  - Encrypt the message
  - Display the encrypted message
  - Decrypt the message
  - Display the decrypted message

What's Next

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

Lecture 3

Conditional Loop Instructions

- LOOPZ and LOOPE
- LOOPNZ and LOOPNE

LOOPZ and LOOPE

- Syntax:
  - LOOPE *destination*
  - LOOPZ *destination*
- Logic:
  - ECX ← 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 ← 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:
```

Your turn . . .

Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0          ; check for zero


     (fill in your code here)


quit:
```

. . . (solution)

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0         ; check for zero
    pushfd                       ; push flags on stack
    add esi,TYPE array
    popfd                        ; pop flags from stack
    loope L1                     ; continue loop
    jz 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

Lecture 4

Conditional Structures

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

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:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    ja  next
    mov eax,5
    mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Your turn . . .

Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```
if( var1 <= var2 )
  var3 = 10;
else
{
  var3 = 6;
  var4 = 7;
}
```

```
    mov eax,var1
    cmp eax,var2
    jle L1
    mov var3,6
    mov var4,7
    jmp L2
L1: mov var3,10
L2:
```

(There are multiple correct solutions to this problem.)

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:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx
    && ecx > edx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    ja  next
    cmp ecx,edx
    jbe next
    mov eax,5
    mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR (1 of 2)

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

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

Compound Expression with OR (2 of 2)

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

We can use "fall-through" logic to keep the code as short as possible:

```
    cmp al,bl               ; is AL > BL?
    ja  L1                  ; yes
    cmp bl,cl               ; no: is BL > CL?
    jbe next                ; no: skip next statement
L1: mov X,1                 ; set X to 1
next:
```

WHILE Loops

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```
while( eax < ebx)
   eax = eax + 1;
```

This is a possible implementation:

```
top:cmp eax,ebx          ; check loop condition
    jae next             ; false? exit loop
    inc eax              ; body of loop
    jmp top              ; repeat the loop
next:
```

Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```
top:cmp ebx,val1            ; check loop condition
    ja  next                ; false? exit loop
    add ebx,5               ; body of loop
    dec val1
    jmp top                 ; repeat the loop
next:
```