Computer Organization and Assembly Language (COAL)

Lecture 6

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Conditional Processing

- **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

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

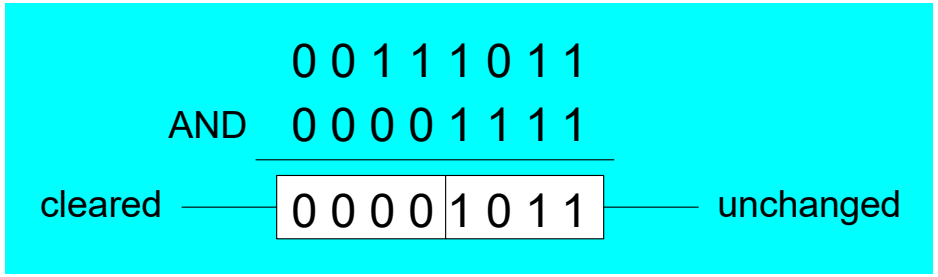
What are Boolean Operations?

Operation	Description
AND	Boolean AND operation between a source operand and a destination operand.
OR	Boolean OR operation between a source operand and a destination operand.
XOR	Boolean exclusive-OR operation between a source operand and a destination operand.
NOT	Boolean NOT operation on a destination operand.
TEST	Implied boolean AND operation between a source and destination operand, setting the CPU flags appropriately.

AND Instruction (1/2)

- Performs a **Boolean AND operation** between each pair of matching bits in two operands
- Operands should be of same size
- Syntax:

```
AND destination, source
```



AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

(same operand types as MOV)

```
AND reg,reg
AND reg,mem
AND reg,imm
AND mem,reg
AND mem,imm
```

AND Instruction (2/2)

- The AND instruction lets **you clear 1 or more bits** in an operand without affecting other bits

- The technique is called bit masking

```
AND AL, 11110110b  ; clear bits 0 and 3
```

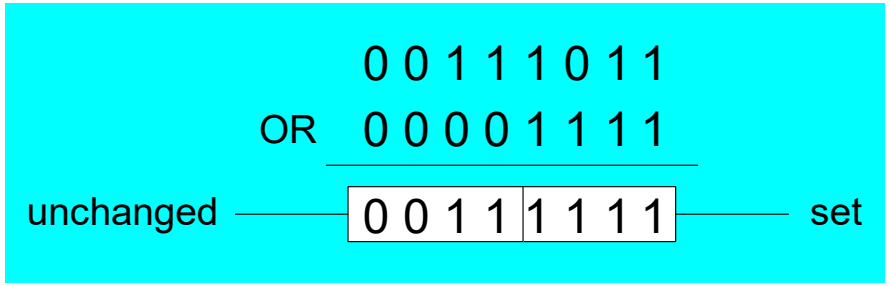
- The AND instruction always clears the **Overflow** and **Carry flags**

- It modifies the **Sign**, **Zero**, and **Parity flags** in a way that is consistent with the value assigned to the destination operand

OR Instruction (1/2)

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

```
OR destination, source
```



OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

```
OR reg,reg
OR reg,mem
OR reg,imm
OR mem,reg
OR mem,imm
```

OR Instruction (2/2)

- The OR instruction is particularly useful when you need **to set 1 or more bits** in an operand without affecting any other bits

```
OR AL, 00000100b  ; set bit 2
```

- The OR instruction always clears the **Carry** and **Overflow flags**

- It modifies the **Sign**, **Zero**, and **Parity flags** in a way that is consistent with the value assigned to the destination operand

- you can OR a number with itself (or zero) to obtain certain information about its value:

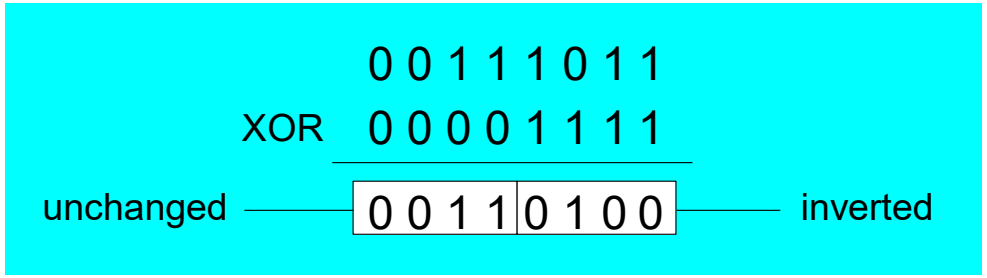
```
or   al,al
```

Zero Flag	Sign Flag	Value in AL Is . . .
Clear	Clear	Greater than zero
Set	Clear	Equal to zero
Clear	Set	Less than zero

XOR Instruction (1/2)

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

```
XOR destination, source
```



XOR

x	y	x ⊕ 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.
  - A bit exclusive-ORed with 0 retains its value
  - A bit exclusive-ORed with 1 is toggled (complemented)
- XOR reverses itself when applied twice to the same operand

XOR Instruction (2/2)

- The XOR instruction always clears the **Overflow** and **Carry flags**
- XOR modifies the **Sign**, **Zero**, and **Parity flags** in a way that is consistent with the value assigned to the destination 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

No flags are affected by the NOT instruction

Applications (1 of 5)

- Task: Convert the character in AL to upper case.

Hint:
To convert an uppercase letter to lowercase, we note that ASCII codes for the uppercase letters '**A**' to '**Z**' form a sequence from **65** to **90.**

The corresponding lowercase letters '**a**' to '**z**' have codes in sequence from **97** to **122**.

- Solution: Use the AND instruction to clear bit 5.

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

Applications (2 of 5)

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

```
mov al,6                    ; AL = 00000110b
or  al,00110000b            ; AL = 00110110b
```

The ASCII digit '6' = 00110110b

HINT:

048	060	30	0011 0000	0
049	061	31	0011 0001	1
050	062	32	0011 0010	2
051	063	33	0011 0011	3
052	064	34	0011 0100	4
053	065	35	0011 0101	5
054	066	36	0011 0110	6
055	067	37	0011 0111	7
056	070	38	0011 1000	8
057	071	39	0011 1001	9

Applications (3 of 5)

- Task: Turn on the keyboard CapsLock key
- Solution: Use the OR instruction to set bit 6 in the keyboard flag byte at 0040:0017h in the BIOS data area.

```
mov ax,40h                      ; BIOS segment
mov ds,ax
mov bx,17h                      ; keyboard flag byte
or BYTE PTR [bx],01000000b      ; CapsLock on
```

This code only runs in Real-address mode, and it does not work under Windows NT, 2000, or XP.

Applications (4 of 5)

- 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.

```
mov ax,wordVal
and ax,1                ; low bit set?
jz  EvenValue           ; jump if Zero flag set
```

JZ will be covered in next Section

Applications (5 of 5)

- 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.

```
or  al,al
jnz IsNotZero            ; jump if not zero
```

ORing any number with itself does not change its value.

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
```

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)

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

- Specific jumps:

  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 - jump to a label if the Sign flag is set

  JNE, JNZ - jump to a label if the Zero flag is clear

  JECXZ - jump to a label if ECX = 0

Jcond 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

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 (1 of 5)

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

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

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)

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

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

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 (2 of 5)

- Jump to label **L1** if unsigned **EAX** is less than or equal to **Val1**

```
cmp eax,Val1
jbe L1              ; below or equal
```

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)

- Jump to label **L1** if signed **EAX** is less than or equal to **Val1**

```
cmp eax,Val1
jle L1
```

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 (3 of 5)

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

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

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)

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

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

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 (4 of 5)

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

```
cmp WORD PTR [esi],0
je  L1
```

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

```
test DWORD PTR [edi],1
jz   L2
```

Applications (5 of 5)

- 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.

```
and al,00001011b        ; clear unwanted bits
cmp al,00001011b        ; check remaining bits
je  L1                  ; all set? jump to L1
```

Home 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.

Encrypting a String

The following loop uses the XOR instruction to transform every character in a string into a new value.

```
KEY = 239                    ; can be any byte value
BUFMAX = 128
.data
buffer  BYTE BUFMAX+1 DUP(0)
bufSize DWORD BUFMAX

.code
    mov ecx,bufSize          ; loop counter
    mov esi,0                ; index 0 in buffer
L1:
    xor buffer[esi],KEY      ; translate a byte
    inc esi                  ; point to next byte
    loop L1
```

String Encryption Program

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

Sample output:

```
Enter the plain text: Attack at dawn.
Cipher text: «¢¢Äîä-Ä¢-ïÄÿü-Gs
Decrypted: Attack at dawn.
```

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 ← 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.

```
.data
array SWORD 50 DUP(?)
.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(?)
.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

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:
```