CONTROL FLOW INSTRUCTIONS

• The basic instructions for branching are *jumps* and *loops*

1. Loops are instructions to repeat a block of code a certain number of times

2. Jumps are instructions that branch to a distant labelled instruction when a flag condition is met

   Status flags are modified by arithmetic instructions so these are normally used just before a jump. Example:

   \[
   \text{JNZ Destination-label} \rightarrow \text{(Jump if Not-Zero)}
   \]

   jumps to the destination label if \(ZF = 0\)
Example with JNZ

.386
.model Flat
include Cs266.inc
.code
main:
    MOV EAX 61h ;letter 'a' in EAX
again:
    MOV EBX, 7Bh ;the next character after 'z'
                 ;in EBX
    PUTCHEAX
    INC EAX
    SUB EBX, EAX
    JNZ again
    Ret
end

• This program prints all the lower case letters in increasing order

• We jump to again whenever ZF = 0
  (when EBX != EAX)

• We do not jump (and execute Ret) when ZF = 1
  (when EBX = EAX)

• The code would be simpler if the SUB instruction
  would not change the value of the destination operand

  We do have such an instruction. It is the CMP (Compare) instruction
The CMP Instruction

.386
.model Flat
include Cs266.inc
.code
main:
    MOV EAX 61h ; letter 'a' in EAX
again:
    PUTCH EAX
    INC EAX
    CMP EAX, 7Bh
    JNZ again
    Ret
end

- CMP Destination, Source → performs the operation Destination - Source but does not modify Destination

- But the flags are affected just like SUB

- Same restrictions on operands as for SUB

- Very often used just before a jump

- The previous program is now simpler
Single-Flags Jumps

- This are jumps where the flag condition consists of a single flag. The following are often used

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Descriptions</th>
<th>Flag Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>JZ</td>
<td>Jump if Zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JE</td>
<td>Jump if Equal</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>JNZ</td>
<td>Jump if Not-Zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JNE</td>
<td>Jump if Not-Equal</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>JS</td>
<td>Jump if Negative</td>
<td>SF = 1</td>
</tr>
<tr>
<td>JNS</td>
<td>Jump if Non-Negative</td>
<td>SF = 0</td>
</tr>
<tr>
<td>JO</td>
<td>Jump if Overflow</td>
<td>OF = 1</td>
</tr>
<tr>
<td>JNO</td>
<td>Jump if No-Overflow</td>
<td>OF = 0</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if Carry</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if No-Carry</td>
<td>CF = 0</td>
</tr>
</tbody>
</table>

- Note: Sometimes the same instruction has two different mnemonics (like JZ and JE)
• There exist two jumps where the condition for jumping is given by a register

\[
\text{JCXZ } \text{Destination-label} \rightarrow \text{jumps if } CX \text{ contains 0}
\]

\[
\text{JECXZ } \text{Destination-label} \rightarrow \text{jumps if } ECX \text{ contains 0}
\]

• Often, we need to branch when some value is larger (smaller) than another

\[
\text{CMP EAX, EBX}
\]

\[
\ldots \ldots \; \text{;now jump somewhere when } EAX > EBX
\]

• However, integer order depends on the chosen interpretation. Ex: if \( AL = 05h \) and \( BL = A0h \) then

\[
\text{AL} > \text{BL} \text{ for a signed interpretation}
\]

\[
\text{AL} < \text{BL} \text{ for an unsigned interpretation}
\]

• Hence, we have these types of comparison jumps

1. Unsigned-comparison jumps: for an unsigned \ldots

2. Signed-comparison jumps: for a signed \ldots

\ldots interpretation of a comparison used just before the jump (ex: with \text{CMP})
Unsigned-Comparison Jumps

- We normally use them just after a CMP Op1, Op2 instruction and the jumping condition is given by an unsigned interpretation of the comparison

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Descriptions</th>
<th>Jump Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if Above (Op1 &gt; Op2)</td>
<td>CF = 0 and ZF = 0</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if Not Below or Equal</td>
<td>CF = 0 and ZF = 0</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if Above or Equal</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if Not Below</td>
<td>CF = 0</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if Below (Op1 &lt; Op2)</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JNAE</td>
<td>Jump if Not Above or Equal</td>
<td>CF = 1</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if Below or Equal</td>
<td>CF = 1 or ZF = 1</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if Not Above</td>
<td>CF = 1 or ZF = 1</td>
</tr>
</tbody>
</table>

- Note: Each of these instructions has two different mnemonics

Unsigned-Comparison Jumps

- We normally use them just after a CMP Op1, Op2 instruction and the jumping condition is given by an signed interpretation of the comparison

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Descriptions</th>
<th>Jump Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>JG</td>
<td>Jump if Greater (Op1 &gt; Op2)</td>
<td>ZF = 0 and SF = 0F</td>
</tr>
<tr>
<td>JNLE</td>
<td>Jump if Not Less or Equal</td>
<td>ZF = 0 and SF = 0F</td>
</tr>
<tr>
<td>JGE</td>
<td>Jump if Greater or Equal</td>
<td>SF = 0F</td>
</tr>
<tr>
<td>JNL</td>
<td>Jump if Not Less</td>
<td>SF = 0F</td>
</tr>
<tr>
<td>JL</td>
<td>Jump if Less (Op1 &lt; Op2)</td>
<td>SF ≠ 0F</td>
</tr>
<tr>
<td>JNGE</td>
<td>Jump if Not Greater or Equal</td>
<td>SF ≠ 0F</td>
</tr>
<tr>
<td>JLE</td>
<td>Jump if Less or Equal</td>
<td>ZF = 1 or SF ≠ 0F</td>
</tr>
<tr>
<td>JNG</td>
<td>Jump if Not Greater</td>
<td>ZF = 1 or SF ≠ 0F</td>
</tr>
</tbody>
</table>

- Note: Each of these instructions has two different mnemonics
Using Comparison Jumps

• CMP is normally used before a comparison jump. Ex: if AX = 1 and BX = FFFFh, to branch to exit when

1. AX > BX under a signed interpretation
   
   CMP AX, BX
   JG exit

2. AX > BX under an unsigned interpretation
   
   CMP AX, BX
   JA exit

Note: No jumping when AX = 1 and BX = FFFFh

Unconditional Jump

• JMP Destination-label jumps without any condition

   .386
   .model Flat
include Cs266.inc
.code
main:
   JMP over
   MOV EAX, 'c'
over:
   PUTCHEAX ;anything may be written
   Ret
end
Application: an Echo Program

```assembly
.386
.model Flat
.include Cs266.inc
.code
main:
    GETCH
    CMP EAX, -1 ;<ctrl_z> ?
    JE exit ;yes! then exit
    PUTCH EAX ;no! then print character
    JMP main
exit:
    Ret
end
```

- Input buffer is initially empty. So `GETCH` returns only when the user hits `<CR>`

- `PUTCH` prints the first character entered

- When `GETCH` is executed again, the next character in the buffer is printed

- Hence this program echoes on the screen the user string entered on the keyboard

- But if the user presses `<ctrl_z>`, then `GETCH` returns -1 in EAX and the program exits
File Redirection of I/O

- I/O operations done with macros in Cs266.inc can be redirected to/from files. If the previous (executable) program is called Echo.exe, then with

  \[ \text{Echo1} < \text{InFile} > \text{OutFile} \]

  \( \text{Echo1} \) reads its input from \( \text{InFile} \) and writes its output into \( \text{OutFile} \)

- If \( \text{InFile} \) is a text file, then \( \text{OutFile} \) will be an identical copy

  Because \( \text{GETCH} \) returns -1 when the EOF is reached

- If \( \text{InFile} \) is a binary file, then \( \text{OutFile} \) will generally not be an identical copy, because

  1. \( \text{GETCH} \) returns -1 if it reads 1Ah (ASCII code of \(<\text{ctrl}_z>\)) from \( \text{InFile} \). So \( \text{Echo1} \) will exit on the first occurrence of 1Ah in \( \text{InFile} \)

  2. When \( \text{GETCH} \) reads 0Ah, PUTCH will output 0Dh, 0Ah

  3. Each 0Dh on input will be ignored by \( \text{GETCH} \)

- This does not happen in Unix, and an equivalent C program that uses \text{getchar()} and \text{putchar()} can copy any file
Exercise #1

• Without modifying the content of AX, write a sequence of 2 instructions that will transfer the execution to the instruction labelled by L1 when

1. The signed value of AX is greater than -128

2. The unsigned value of AX is lower or equal to 255

3. AL contains an upper case letter (assuming AL always contains a letter)

4. AL contains a lower case letter (assuming AL always contains a letter)
High-Level Control Flow Structures

• High-level languages use high-level structures such as if-then-else, case, while, ... to control the flow of execution

  Algorithms are normally expressed in terms of these high-level structures

• Processors only provide conditional and unconditional loops

  Thus we need to decompose the high-level control flow structures in low-level ones

  **If-Then-Else Structure**

• If Condition \{ CodeBlock1 \} Else \{ CodeBlock2 \}

  Program jumps to CodeBlock2 if Condition is False

• Assembler observation: JXXX CodeBlock2

  Program jumps to CodeBlock2 if XXX is true

• Therefore HLL’s not Condition $\leftrightarrow$ Assembler’s JXXX
• Example:

\[
\text{If } (\text{Op1} < \text{Op2}) \text{ Then} \\
\quad \text{Then}\_\text{Block} \\
\text{Else} \\
\quad \text{Else}\_\text{Block} \\
\text{End}\_\text{If}
\]

• Analysis

1. JXXX to Else\_Block when \( \text{Op1} \geq \text{Op2} \)

2. JMP from end of Then\_Block to End\_If

• ASM solution for signed-comparison

\[
\text{CMP Op1, Op2} \\
\text{JGE else_} \\
\quad ;\text{put then}\_\text{block’s code here} \\
\ldots \ldots \\
\text{JMP end_if} \\
\text{else_:} \\
\quad \ldots \ldots \\
\quad ;\text{put else}\_\text{block’s code here} \\
\text{end_if:} \\
\ldots \ldots \\
\quad ;\text{statements after the If-Then-Else structure}
\]

Note that else is an ASM reserved word. We use else_ instead.
While Structure

• Example:

While (Op1 < Op2)

    While_Block

End_While

• Analysis

1. JXXX to End_While when Op1 ≥ Op2

2. JMP from end of While_Block to While

• ASM solution for unsigned-comparison

while:
    CMP Op1, Op2
    JAE end_while
    ... ...
    : ;put while_block’s code here
    ... ...
    JMP while
end_while:
    ... ...
    : ;statements after the While structure
    ... ...

Case Structure

- Example:

Case Value is
   'A': Case_Block_1
   'B': Case_Block_2
   'C': Case_Block_3
End_Case

- Analysis

1. CMP and JXXX for each case
2. JMP from end of each Case_Block to End_Case

- ASM solution for signed-comparison

    CMP Value, 'A'
    JNE case_2
    : ;put case_block_1’s code here
    JMP end_case

    case_2:
    CMP Value, 'B'
    JNE case_3
    : ;put case_block_2’s code here
    JMP end_case

    case_3:
    CMP Value, 'C'
    JNE end_case
    : ;put case_block_3’s code here

end_case:
    ;statements after the Case structure
The **LOOP** Instruction

- **LOOP Destination-label** → provides the easiest way to repeat a block of statements a specific number of times

- **Destination-label** must precede **LOOP** by less than 128 bytes of code

  This restriction does not exist for jumps

- The execution of **LOOP** produces the following sequence of events

  1. **ECX** is decremented by 1

  2. If **ECX** = 0 Then
      
      Jump to instruction following **LOOP**
      
      Else

      Jump to **Destination-label**
- Example: the following code fragment will print all the ASCII codes by starting with 7Fh

```assembly
MOV ECX 7Fh
next:
    PUTCH ECX
    LOOP next
```

- If (instead) ECX would be initialized to zero, then

1. After executing the block for the first time, ECX would be decremented by 1 and thus contain 0FFFFFFFFh

2. The loop would thus be repeated again 0FFFFFFFFh times!!

- Hence, if ECX contains an unspecified value, it is better to write

```assembly
;Executes the loop ECX times
JECXZ over
next:
    PUTCH ECX
    LOOP next
over:
    ;statements after the Loop structure
```
Conditional Loops

- LOOPZ and LOOPE continue a loop while ZF = 1 and ECX ≠ 0

  ECX is first decremented, then the condition is tested (a trivial extension of LOOP)

- LOOPNZ and LOOPNE continue a loop while ZF = 0 and ECX ≠ 0

- Syntax (same as LOOP) → LOOPXX Destination-label

- Destination-label must precede LOOPXX by less than 128 bytes of codes

```
.386
.model Flat
include Cs266.inc
.code
main:
  MOV ECX, 7Fh
next:
  PUTCH ECX
  CMP ECX, 20h
  LOOPNZ next
  Ret
end
```
Exercise #2

• Write a small piece of code that will display the character in \textit{AL} iff it is an uppercase letter

• Write a small piece of code that will count the number of characters in a user input line