Selection and Iteration

Chapter 7
S. Dandamudi

Outline

• Unconditional jump
• Compare instruction
• Conditional jumps
  * Single flags
  * Unsigned comparisons
  * Signed comparisons
• Loop instructions
• Implementing high-level language decision structures
  * Selection structures
  * Iteration structures
• Illustrative examples
• Indirect jumps
  * Multiway conditional statements
• Logical expression evaluation
  * Full evaluation
  * Partial evaluation
• Performance: Logical expression evaluation
  * Partial vs. full evaluation
Unconditional Jump

• Unconditional jump transfers control to the instruction at the target address
• Format
  \[
  \text{jmp target}
  \]
• Specification of target
  * Direct
    » Target address is specified as a part of the instruction
  * Indirect
    » Target address is specified indirectly either through memory or a general-purpose register

Example

• Two jump instructions
  * Forward jump
    \[
    \text{jmp CX_init_done}
    \]
  * Backward jump
    \[
    \text{jmp repeat1}
    \]
• Programmer specifies target by a label
• Assembler computes the offset using the symbol table
Unconditional Jump (cont’d)

• Address specified in the jump instruction is not the absolute address
  * Uses relative address
    » Specifies relative byte displacement between the target instruction and the instruction following the jump instruction
    » Displacement is w.r.t the instruction following jmp
      – Reason: IP is already pointing to this instruction
  * Execution of jmp involves adding the displacement value to current IP
  * Displacement is a signed 16-bit number
    » Negative value for backward jumps
    » Positive value for forward jumps

Target Location

• Inter-segment jump
  * Target is in another segment
    \[ CS := \text{target-segment} \quad (2 \text{ bytes}) \]
    \[ IP := \text{target-offset} \quad (2 \text{ bytes}) \]
    » Called far jumps (needs five bytes to encode jmp)

• Intra-segment jumps
  * Target is in the same segment
    \[ IP := \text{IP} + \text{relative-displacement} \quad (1 \text{ or } 2 \text{ bytes}) \]
  * Uses 1-byte displacement if target is within \(-128 \text{ to } +127\)
    » Called short jumps (needs two bytes to encode jmp)
  * If target is outside this range, uses 2-byte displacement
    » Called near jumps (needs three bytes to encode jmp)
Target Location (cont’d)

- In most cases, the assembler can figure out the type of jump
- For backward jumps, assembler can decide whether to use the short jump form or not
- For forward jumps, it needs a hint from the programmer
  * Use SHORT prefix to the target label
  * If such a hint is not given
    » Assembler reserves three bytes for `jmp` instruction
    » If short jump can be used, leaves one byte of rogue data
      – See the next example for details

Example

```
8 0005   EB 0C   jmp   SHORT CX_init_done
          0013 - 0007 = 0C
9 0007   B9 000A  mov   CX,10
10 000A  EB 07 90   jmp   CX_init_done
          rogue byte   0013 - 000D = 07
11  init_CX_20:
12 000D  B9 0014  mov   CX,20
13 0010  E9 00D0  jmp   near_jump
          00E3 - 0013 = D0
14  CX_init_done:
15 0013  8B C1   mov   AX,CX
```
Example (cont’d)

16  repeat1:
17 0015  49   dec   CX
18 0016  EB FD   jmp  repeat1

\[0015 - 0018 = -3 = FDH\]

\[0015 - 0018 = -3 = FDH\]

\[0015 - 0018 = -3 = FDH\]

... 

84 00DB  EB 03   jmp  SHORT short_jump

\[00E0 - 00DD = 3\]

85 00DD  B9 FF00   mov   CX, 0FF00H

86  short_jump:

87 00E0  BA 0020   mov   DX, 20H

88  near_jump:

89 00E3  E9 FF27   jmp  init_CX_20

\[000D - 00E6 = -217 = FF27H\]

Compare Instruction

• Compare instruction can be used to test the conditions
• Format
  \[\text{cmp} \quad \text{destination, source}\]
• Updates the arithmetic flags by performing
  \[\text{destination} - \text{source}\]
• The flags can be tested by a subsequent conditional jump instruction
Conditional Jumps

• Three types of conditional jumps
  ∗ Jumps based on the value of a single flag
    » Arithmetic flags such as zero, carry can be tested using these instructions
  ∗ Jumps based on unsigned comparisons
    » The operands of `cmp` instruction are treated as unsigned numbers
  ∗ Jumps based on signed comparisons
    » The operands of `cmp` instruction are treated as signed numbers

Jumps Based on Single Flags

Testing for zero

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jz</code></td>
<td>jump if zero</td>
<td>jumps if ZF = 1</td>
</tr>
<tr>
<td><code>je</code></td>
<td>jump if equal</td>
<td>jumps if ZF = 1</td>
</tr>
<tr>
<td><code>jnz</code></td>
<td>jump if not zero</td>
<td>jumps if ZF = 0</td>
</tr>
<tr>
<td><code>jne</code></td>
<td>jump if not equal</td>
<td>jumps if ZF = 0</td>
</tr>
<tr>
<td><code>jcxz</code></td>
<td>jump if CX = 0</td>
<td>jumps if CX = 0</td>
</tr>
</tbody>
</table>

(Flags are not tested)
Jumps Based on Single Flags (cont’d)

Testing for carry

- **jc**: jump if carry  
  jumps if CF = 1
- **jnc**: jump if no carry  
  jumps if CF = 0

Testing for overflow

- **jo**: jump if overflow  
  jumps if OF = 1
- **jno**: jump if no overflow  
  jumps if OF = 0

Testing for sign

- **js**: jump if negative  
  jumps if SF = 1
- **jns**: jump if not negative  
  jumps if SF = 0

Testing for parity

- **jp**: jump if parity  
  jumps if PF = 1
- **jpe**: jump if parity  
  jumps if PF = 1
  is even
- **jnp**: jump if not parity  
  jumps if PF = 0
- **jpo**: jump if parity  
  jumps if PF = 0
  is odd
### Jumps Based on Unsigned Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>jump if equal</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>jz</td>
<td>jump if zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>jne</td>
<td>jump if not equal</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>jnz</td>
<td>jump if not zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>ja</td>
<td>jump if above</td>
<td>CF = ZF = 0</td>
</tr>
<tr>
<td>jnbe</td>
<td>jump if not below or equal</td>
<td>CF = ZF = 0</td>
</tr>
</tbody>
</table>

### Jumps Based on Unsigned Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<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</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>
## Jumps Based on Signed Comparisons

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>jump if equal</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>jz</td>
<td>jump if zero</td>
<td>ZF = 1</td>
</tr>
<tr>
<td>jne</td>
<td>jump if not equal</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>jnz</td>
<td>jump if not zero</td>
<td>ZF = 0</td>
</tr>
<tr>
<td>jg</td>
<td>jump if greater</td>
<td>ZF=0 &amp; SF=OF</td>
</tr>
<tr>
<td>jnle</td>
<td>jump if not less or equal</td>
<td>ZF=0 &amp; SF=OF</td>
</tr>
</tbody>
</table>

## Jumps Based on Signed Comparisons (cont’d)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Meaning</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>jge</td>
<td>jump if greater or equal</td>
<td>SF = OF</td>
</tr>
<tr>
<td>jnl</td>
<td>jump if not less</td>
<td>SF = OF</td>
</tr>
<tr>
<td>jl</td>
<td>jump if less</td>
<td>SF ≠ OF</td>
</tr>
<tr>
<td>jnge</td>
<td>jump if not greater or equal</td>
<td>SF ≠ OF</td>
</tr>
<tr>
<td>jle</td>
<td>jump if less</td>
<td>ZF=1 or SF ≠ OF</td>
</tr>
<tr>
<td>jng</td>
<td>jump if not greater</td>
<td>ZF=1 or SF ≠ OF</td>
</tr>
</tbody>
</table>
A Note on Conditional Jumps

- All conditional jumps are encoded using 2 bytes
  - Treated as short jumps
- What if the target is outside this range?

<table>
<thead>
<tr>
<th>target:</th>
<th>Use this code to get around</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . .</td>
<td>. . .</td>
</tr>
<tr>
<td>cmp AX, BX</td>
<td>cmp AX, BX</td>
</tr>
<tr>
<td>je target</td>
<td>jne skip1</td>
</tr>
<tr>
<td>mov CX, 10</td>
<td>jmp target</td>
</tr>
<tr>
<td>. . .</td>
<td>skip1:</td>
</tr>
<tr>
<td></td>
<td>mov CX, 10</td>
</tr>
</tbody>
</table>

* target is out of range for a short jump

Loop Instructions

- Loop instructions use CX/ECX to maintain the count value
- target should be within the range of a short jump as in conditional jump instructions
- Three loop instructions

loop target
Action: CX := CX - 1
Jump to target if CX ≠ 0
Loop Instructions (cont’d)

- The following two loop instructions also test the zero flag status

```
loope/loopz  target
Action: CX := CX-1
Jump to target if (CX ≠ 0 and ZF = 1)
```

```
loopne/loopnz  target
Action: CX := CX-1
Jump to target if (CX ≠ 0 and ZF = 0)
```

Instruction Execution Times

- Functionally, `loop` instruction can be replaced by
  
  ```
  dec    CX
  jnz    target
  ```

- `loop` instruction is slower than `dec/jnz` version
- `loop` requires 5/6 clocks whereas `dec/jnz` takes only 2 clocks

- `jcxz` also takes 5/6 clocks
- Equivalent code (shown below) takes only 2 clocks
  
  ```
  cmp    CX, 0
  jz     target
  ```
Implementing HLL Decision Structures

- High-level language decision structures can be implemented in a straightforward way
- See Section 7.5 (page 272) for examples that implement
  - if-then-else
  - if-then-else with a relational operator
  - if-then-else with logical operators AND and OR
  - while loop
  - repeat-until loop
  - for loop

Illustrative Examples

- Two example programs
  - Linear search
    » LIN_SRCH.ASM
    » Searches an array of non-negative numbers for a given input number
  - Selection sort
    » SEL_SORT.ASM
    » Uses selection sort algorithm to sort an integer array in ascending order
Indirect Jumps

- Jump target address is not specified directly as a part of the jump instruction
- With indirect jump, we can specify target via a general-purpose register or memory
  - Example: Assuming CX has the offset value
    \[
    \text{jmp cx}
    \]
  - Note: The offset value in indirect jump is the absolute value (not relative value as in direct jumps)
- Program example
  - \text{IJUMP.ASM}
    - Uses a jump table to direct the jump

Indirect Jumps (cont’d)

- Another example
  - Implementing multiway jumps
    - We use \text{switch} statement of C
  - We can use a table with appropriate target pointers for the indirect jump
  - Segment override is needed
    - \text{jump_table} is in the code segment (not in the data segment)
**Indirect Jumps (cont’d)**

```assembly
_main PROC NEAR
    mov AL, ch
    cbw
    sub AX, 48 ; 48 = '0'
    mov BX, AX
    cmp BX, 3
    ja default
    shl BX, 1 ; BX:= BX*2
    jmp WORD PTR CS:jump_table[BX]

_case_0:
    inc WORD PTR [BP-10]
    jmp SHORT end_switch

case_1:
    inc WORD PTR [BP-8]
    jmp SHORT end_switch

case_2:
    inc WORD PTR [BP-6]
    jmp SHORT end_switch

case_3:
    inc WORD PTR [BP-4]
    jmp SHORT end_switch

default:
    inc WORD PTR [BP-2]

end_switch:
    ...

_main ENDP
```

**jump_table** LABEL WORD

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DW</td>
<td>case_0</td>
</tr>
<tr>
<td>DW</td>
<td>case_1</td>
</tr>
<tr>
<td>DW</td>
<td>case_2</td>
</tr>
<tr>
<td>DW</td>
<td>case_3</td>
</tr>
</tbody>
</table>

---

**Evaluation of Logical Expressions**

- Two basic ways
  - Full evaluation
    - Entire expression is evaluated before assigning a value
    - PASCAL uses full evaluation
  - Partial evaluation
    - Assigns as soon as the final outcome is known without blindly evaluating the entire logical expression
    - Two rules help:
      - `cond1 AND cond2`
      - If `cond1` is false, no need to evaluate `cond2`
      - `cond1 OR cond2`
      - If `cond1` is true, no need to evaluate `cond2`
Evaluation of Logical Expressions (cont’d)

- Partial evaluation
  - Used by C
- Useful in certain cases to avoid run-time errors
- Example
  \[ \text{if } ((X > 0) \text{ AND } (Y/X > 100)) \]
  - If \( x \) is 0, full evaluation results in divide error
  - Partial evaluation will not evaluate \((Y/X > 100)\) if \( x = 0 \)
- Partial evaluation is used to test if a pointer value is NULL before accessing the data it points to

Performance: Full vs. Partial Evaluation

- Chart showing the execution time (seconds) vs. number of calls (in thousands) for full evaluation, partial evaluation, and AL version.