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

```
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*

Unconditional Jump (cont'd)

Example

- Two jump instructions
  - ∗ Forward jump

    `jmp    CX_init_done`
  - ∗ Backward jump

    `jmp    repeat1`
- Programmer specifies target by a label
- Assembler computes the offset using the symbol table

```
          . . .
      mov    CX,10
      jmp    CX_init_done
init_CX_20:
      mov    CX,20
CX_init_done:
      mov    AX,CX
repeat1:
      dec    CX
            . . .
      jmp    repeat1
            . . .
```

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 := target-segment` (2 bytes)

    `IP := target-offset` (2 bytes)
    - » Called *far jumps* (needs five bytes to encode **`jmp`**)
- Intra-segment jumps
  - ∗ Target is in the same segment

    `IP := IP + relative-displacement` (1 or 2 bytes)
  - ∗ Uses 1-byte displacement if target is within –128 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

```
                  . . .
0005   EB 0C          jmp    SHORT CX_init_done
                                0013 - 0007 = 0C
0007   B9 000A        mov    CX,10
000A   EB 07 90       jmp    CX_init_done
              rogue byte        0013 - 000D = 07
                  init_CX_20:
000D   B9 0014        mov    CX,20
0010   E9 00D0        jmp    near_jump
                                00E3 - 0013 = D0
                  CX_init_done:
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
.  .  .
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

  `cmp    destination, source`
- Updates the arithmetic flags by performing

  `destination - 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

`jz`	jump if zero	jumps if ZF = 1
`je`	jump if equal	jumps if ZF = 1
`jnz`	jump if not zero	jumps if ZF = 0
`jne`	jump if not equal	jumps if ZF = 0
`jcxz`	jump if CX = 0	jumps if CX = 0 (Flags are not tested)

To be used with S. Dandamudi, "Introduction to Assembly Language Programming," Springer-Verlag, 1998.

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

Jumps Based on Single Flags (cont'd)

Testing for parity

`jp`	jump if parity	jumps if PF = 1
`jpe`	jump if parity is even	jumps if PF = 1
`jnp`	jump if not parity	jumps if PF = 0
`jpo`	jump if parity is odd	jumps if PF = 0

Jumps Based on Unsigned Comparisons

Mnemonic	Meaning	Condition
je	jump if equal	ZF = 1
jz	jump if zero	ZF = 1
jne	jump if not equal	ZF = 0
jnz	jump if not zero	ZF = 0
ja	jump if above	CF = ZF = 0
jnbe	jump if not below or equal	CF = ZF = 0

Jumps Based on Unsigned Comparisons

Mnemonic	Meaning	Condition
jae	jump if above or equal	CF = 0
jnb	jump if not below	CF = 0
jb	jump if below	CF = 1
jnae	jump if not above or equal	CF = 1
jbe	jump if below or equal	CF=1 or ZF=1
jna	jump if not above	CF=1 or ZF=1

Jumps Based on Signed Comparisons

Mnemonic	Meaning	Condition
je	jump if equal	ZF = 1
jz	jump if zero	ZF = 1
jne	jump if not equal	ZF = 0
jnz	jump if not zero	ZF = 0
jg	jump if greater	ZF=0 & SF=OF
jnle	jump if not less or equal	ZF=0 & SF=OF

Jumps Based on Signed Comparisons (cont'd)

Mnemonic	Meaning	Condition
jge	jump if greater or equal	SF = OF
jnl	jump if not less	SF = OF
jl	jump if less	SF ≠ OF
jnge	jump if not greater or equal	SF ≠ OF
jle	jump if less or equal	ZF=1 or SF ≠ OF
jng	jump if not greater	ZF=1 or SF ≠ OF

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?

```
target:
        . . .
   cmp     AX,BX
   je     target
   mov     CX,10
        . . .
```

traget is out of range for a short jump

- Use this code to get around

```
target:
        . . .
   cmp     AX,BX
   jne     skip1
   jmp     target
skip1:
   mov     CX,10
        . . .
```


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

```
jmp    CX
```

  * Note: The offset value in indirect jump is the absolute value (not relative value as in direct jumps)
- Program example
  * IJUMP.ASM
    » Uses a jump table to direct the jump

Indirect Jumps (cont’d)

- Another example
  * Implementing multiway jumps
    » We use **switch** statement of C
  * We can use a table with appropriate target pointers for the indirect jump
  * Segment override is needed
    » **jump_table** is in the code segment (not in the data segment)

```
switch (ch)
{
    case '0':
              count[0]++;
              break;
    case '1':
              count[1]++;
              break;
    case '2':
              count[2]++;
              break;
    case '3':
              count[3]++;
              break;
    default:
              count[4]++;
}
```

Indirect Jumps (cont'd)

```
  _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
      DW     case_0
      DW     case_1
      DW     case_2
      DW     case_3
           .  .  .
```


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

```
if ((X > 0) 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