Memorial University of Newfoundland

Engineering 4862 MICROPROCESSORS

Assignment 3 Solution

---

0. For each of the CF, PF, ZF, SF, and OF flags, briefly describe the meaning when it is set. Give conditional jump instructions that can be used to test each one.

Flag	Description (when set)	Conditional Jumps
CF	High-order bit carry or borrow	JC, JNC, JAE, JNB, JB, JNAE
PF	Low-order 8 bits or result contain even number of 1-bits	JP, JPE, JNP, JPO
ZF	Result is zero	JE, JZ, JNE, JNZ
SF	Result is positive (if a signed number)	JS, JNS
OF	Signed result cannot be expressed within the number of bits of destination operand	JO, JNO

Notes:

a. The description for CF states a **high-order bit carry** because the bit examined depends on whether the operands are 8-bit or 16-bit. On the addition of two 8-bit numbers, CF is set if there is a carry out of bit 7 (where the bits are numbered from 0 to 7). For two 16-bit numbers, the examined bit is number 15.

b. The microprocessor does not know if you are adding or subtracting two numbers that are signed or unsigned. Thus OF is set or cleared whether or not signed numbers are used. Since the range for an 8-bit signed number is from –128 to 127, if two 8-bit numbers are added to get a value greater than 127, or less than –128, you will have to write code to convert the result to a 16-bit signed number.

1. a. Determine the contents of register BX and the six conditional (status) flags after each of the following instructions executes. If a flag or register contents are unknown, indicate with a '?'.

```
CLC
MOV BL, 4DH
SUB BL, 3EH
XOR BH, BH
MOV SI], BX
```

	BX	CF	PF	AF	ZF	SF	OF
CLC	??	0	?	?	?	?	?
MOV BL, 4DH	??4D	0	?	?	?	?	?
SUB BL, 3EH	??0F	0	1	1	0	0	0
XOR BH, BH	000F	0	1	?	1	0	0
MOV [SI], BX	000F	0	1	?	1	0	0

Notes:

- XOR automatically sets the CF and OF to 0
- It does not matter what values is in BH, as XORing will set the result to 0!

  4DH = 0100 1101
  - 3EH = <u>0011 1110</u>
  0000 1111
- AF is set to 1 because there is a borrow from high-order 4 bits to the low-order 4-bits (between bits 3 and 4).
- CF is set to 0, because there is no borrow into the high-order bit (bit 7)

b. Read how to use debug in appendix A of the textbook. For each instruction in (a), use DEBUG (or some other program) to determine the equivalent machine code.

The machine code is given in bold in the following capture from DEBUG. Note that entering numbers in DEBUG automatically defaults to hexadecimal. This can be seen in the first MOV instruction, as the '4D' is automatically translated into 4D as machine code.

```
12A7:0100 F8            CLC
12A7:0101 B34D          MOV     BL,4D
12A7:0103 80EB3E        SUB     BL,3E
12A7:0106 30FF          XOR     BH,BH
12A7:0108 891C          MOV     [SI],BX
```

The 8086/88 user manual also tells you how many bytes that the instruction will be turned into and what machine code for each instruction.

2. ***Assume that the PUSH instruction does not exist in the 8086/8088 instruction set. Write a sequence of instructions that function equivalently to PUSH DX. You may use any other valid instruction, but restore any registers you change that PUSH DX does not.***

First, read what the push instruction does (page 3-132): it decrements SP by two, and then moves the source to the memory location given my SS:SP. The problem is that there is no addressing mode that allows you to use SP directly. The effective addresses allow different combinations of BP, BX, SI, and DI. We'll use BP, as the microprocessor automatically uses the SS as the segment. However, we must be careful to not lose the existing value in BP!

Attempt 1 (not quite right):

```
SUB SP, 2                   ; decrement stack pointer
XCHG   BP, SP               ; save BP, and use value in SP
MOV    [BP], DX             ; move data to memory at SS:[BP]
XCHG   BP, SP               ; restore BP and SP
```

This is not bad, but unfortunately the SUB instruction modifies a number of flags, and PUSH does not modify any flags. A correct method is to use an instruction we did not look at in class: LEA (load effective address). Read page 3-114 of the Intel User's Manual for details. Essentially, you give the source as a valid memory reference, but the offset (not the value) is placed into the destination. It affects no flags, and uses no push instructions of any kind.

Attempt 2:

```
XCHG   BP, SP               ; save BP, get SP
LEA    BP, [BP] - 2         ; set BP to new memory location
MOV    [BP], DX             ; move data to memory
XCHG   BP, SP               ; restore BP and SP
```

Other solutions are trickier to implement. If you use SUB, then you must figure out how to restore the flags to their original values. Note that PUSHF and POPF are available to your use.

Attempt 3:

```
PUSHF                       ; save flags at (original SP)-2
PUSHF                       ; save flags again at (orig SP)-4
ADD    SP, 2                ; change SP to (original SP) - 2
XCHG   BP, SP               ; swap SP and BP
MOV    [BP], DX             ; save DX
XCHG   BP, SP               ; restore SP and BP
SUB    SP, 2                ; change SP to (original SP)-4
POPF                        ; restore flags, SP = (orig SP)-2
```

This final attempt is the simplest of all (so far): Use PUSHF to update SP, but then overwrite the flags with DX – because none of the flags will actually be changed!

Attempt 4:

```
PUSHF                    ; save flags on stack
XCHG   BP, SP            ; swap SP and BP (SP has been updated)
MOV    [BP], DX          ; save DX to SS:BP
XCHG   BP, SP            ; restore SP and BP
```

3. ***Write a subroutine to replace the multiplication instruction MUL CX. You may use any valid 8086/8088 instructions other than MUL, but take care to properly handle the flags and restore any registers that you use to store temporary values. Start your subroutine with the label mul_cx, and end with the RET instruction.***

MUL CX

Notes on implementation:

- Source is a word (CX), so result will be placed in DX (high word) and AX (low word)
- CF and OF are either set (if DX is non-zero) or cleared (DX is zero)
- AF, PF, SF, and ZF are undefined, but we'll set them to zero
- We'll take care not to affect the other 3 flags (TF, IF, and DF)
- The following algorithm is a simple, repetitive addition

```
         PUSH BX           ; save registers and flags
         PUSH CX
         PUSHF

         MOV DX, 0         ; clear DX & AX, used to store product
         MOV BX, AX        ; use BX to count the number of additions
         MOV AX, 0

         JCXZ Done   ; If multiplier is zero, so is the product
Do_again: ADD AX, CX
         JNC Skip_hi
         INC DX      ; Carry-out from AX, so increment DX
Skip_hi: DEC BX
         JNZ Do_again      ; continue BX times

Done:    POP CX            ; Get flags
               ; Reset all flags except TF, IF, and DF
         AND CX, 0000011100000000b

         CMP DX, 0         ; is DX zero?
         JE Set_flags      ; yes, so leave OF & CF as 0
               ; DX is not zero, so OF = CF = 1
         OR CX, 0000100000000001b

Set_flags: PUSH CX   ; store flags
         POPF        ; restore correct flag values
         POP CX      ; restore registers
         POP BX
         RET         ; Done
```

4. ***Write a MUN-88-compatible program that reads the contents of the DIP switches, and then converts the 8-bit decimal value into two 8-bit ASCII values representing each hex digit. Store the lower digit in AL, and the upper digit in AH. This should be a full program, so include a title, segment definitions, etc., as well as comments.***

Example result: *Suppose that after reading the input port for the DIP switches, AL is 9FH. Your program should place 39H (ASCII for '9') into AH, and 46H (ASCII for 'F') into AL.*

```
DIPS     equ 30h

TITLE    DIP Converter
myseg    SEGMENT
         ASSUME cs:myseg, ds:myseg, es:myseg

Main:    mov ax, cs
         mov ds, ax
         mov es, ax

         in al, DIPS        ; read switch values
         mov ah, al         ; copy values
         and al, 0Fh        ; mask out upper 4 bits
         call Convert       ; subroutine to convert AL to ASCII

         xchg ah, al        ; Swap AH and AL (for Convert)
         mov cl, 4
         shr al, cl         ; Shift AL to right by 4
         call Convert       ; Convert AL to ASCII
         xchg ah, al        ; Swap AH and AL back

         int 6              ; Finished

; ****************************************************************
; Convert - converts AL to ASCII, and stores resulting byte in AL
; ****************************************************************

Convert: cmp al, 9          ; Is AL above 9?
         ja Letter          ; Yes, so AL is a letter
         add al, 30h        ; ASCII 30h to 39h are numeric
         jmp Done
Letter:  add al, 37h        ; ASCII 37h+Ah = 41h
                            ; and ASCII 41h to 46h are 'A' to 'F'
Done:    ret

myseg    ENDS
         END   Main
```

5. Write a program that subtracts two multi-digit ASCII numbers (Data1 – Data2). The result should be saved back to Result in ASCII. The Data Segment is defined as following:

```
DTSEG                   SEGMENT
     Data1       DB     '3546882164'
     Data2       DB     '2345611245'
     Result      DB     10 DUP (?)
DTSEG                   ENDS
```

The approach I used is: first convert the ASCII numbers to packed BCD numbers (also stored in memory), then perform multi-byte packed BCD number subtraction (result also stored in memory), finally convert result to ASCII and save them to the location RESULT as required. All these functions are placed in subroutines.

```
TITLE    Subtracting ASCII Numbers
PAGE     60, 132
STSEG    SEGMENT
         DB 64 DUP(?)
STSEG    ENDS
;------------------
DTSEG    SEGMENT
         DATA1        DB    '3546882164'
         DATA2        DB    '2345611245'
         RESULT       DB    10 DUP (?),"$"
         DATA1_BCD    DB    5 DUP(?)
         DATA2_BCD    DB    5 DUP(?)
         RESULT_BCD   DB    5 DUP(?)
DTSEG    ENDS
;------------------
CDSEG    SEGMENT
MAIN     PROC  FAR
         ASSUME CS:CDSEG, DS:DTSEG, SS:STSEG
         MOV AX, DTSEG
         MOV DS, AX

         MOV BX, OFFSET DATA1
         MOV DI, OFFSET DATA1_BCD
         MOV CX, 10
         CALL CONVERT_BCD

         MOV BX, OFFSET DATA2
         MOV DI, OFFSET DATA2_BCD
         MOV CX, 10
         CALL CONVERT_BCD

         CALL SUBTRACTION

         MOV SI, OFFSET RESULT_BCD
         MOV DI, OFFSET RESULT
         MOV CX, 5
         CALL CONVERT_ASC

         MOV AH, 4CH
         INT 21H
MAIN     ENDP
;--------------------
```

```
; SUBROUTINE CONVERT ASCII NUMBERS TO BCD NUMBERS
CONVERT_BCD    PROC  NEAR
REP0:    MOV AX, [BX]
         XCHG AH, AL
         AND AX, 0F0FH
         PUSH CX
         MOV CL, 4
         SHL AH, CL
         OR AL, AH
         MOV [DI], AL
         ADD BX, 2
         INC DI
         POP CX
         LOOP REP0
         RET
CONVERT_BCD    ENDP
;---------------------
; SUBROUTINE PERFORM PACKED BCD NUMBER SUBTRACTION
SUBTRACTION    PROC  NEAR
         MOV BX, OFFSET DATA1_BCD
         MOV DI, OFFSET DATA2_BCD
         MOV SI, OFFSET RESULT_BCD
         MOV CX, 5
         CLC
REP1:    MOV AL, [BX]+4
         SBB AL, [DI]+4
         DAS
         MOV [SI]+4, AL
         DEC BX
         DEC DI
         DEC SI
         LOOP REP1
         RET
SUBTRACTION    ENDP
;---------------------
; SUBROUTINE CONVERT BCD NUMBERS TO ASCII NUMBERS
CONVERT_ASC    PROC  NEAR
REP3:    MOV AL, [SI]
         MOV AH, AL
         AND AX, 0F00FH
         PUSH CX
         MOV CL, 4
         SHR AH, CL
         OR AX, 3030H
         XCHG AH, AL
         MOV [DI], AX
         INC SI
         ADD DI, 2
         POP CX
         LOOP REP3
         RET
CONVERT_ASC    ENDP
;---------------------
CDSEG    ENDS
         END MAIN
```

6. Write a program that converts an ASCII string saved by Old_String to its uppercase in ASCII and save back to the New_String. Leave the space and period unchanged. The Data Segment is defined as following:

```
DTSEG                     SEGMENT
     Old_String  DB      'This is THE String to be converted.'
     New_String  DB      35 DUP (?)
DTSEG                     ENDS
```

```
TITLE    COVERT LOWER CASE TO UPPER CASE
PAGE     60, 132
STSEG    SEGMENT
         DB 64 DUP(?)
STSEG    ENDS
;------------------
DTSEG    SEGMENT
         Old_String  DB    "This is THE String to be converted."
         New_String  DB    35 DUP (?), "$"
DTSEG    ENDS
;------------------
CDSEG    SEGMENT
MAIN     PROC  FAR
         ASSUME CS:CDSEG, DS:DTSEG, SS:STSEG
         MOV AX, DTSEG
         MOV DS, AX

         MOV SI, OFFSET Old_String
         MOV BX, OFFSET New_String
         MOV CX, 35
REP0:    MOV AL, [SI]
         CMP AL, 61H ;IF LESS THAN 'a', THEN EXIT
         JB OVER
         CMP AL, 7AH ;IF GREATER THAN 'z', THEN EXIT
         JA OVER
         AND AL, 11011111B ; MASK d5 TO CONVERT TO UPPER CASE
OVER:    MOV [BX], AL
         INC SI
         INC BX
         LOOP REP0

         MOV AH, 4CH
         INT 21H
MAIN     ENDP
;---------------------
CDSEG    ENDS
         END MAIN
```