Feisal Mohammed
12th January 2001

Assembly language

1 The MPLAB assembler

The MPLAB software by MicroChip Inc is available in the resource directory for this course it is large (10.5MB) and can run under Windows 95/98 and Windows NT. There is an online tutorial from the help section that shows how files are managed and other important functions. The following sections describe how to assemble programs using MPLAB.

1.1 Using the MPLAB assembler

Programming the PIC, in this case the PIC16F84, is quite different from other CISC processors that you may be familiar with and the following series of exercises serve to illustrate the differences that must be taken into account. When using the MPLAB assembler, there is a directive to specify the type of processor and an *include* file that has the standard definitions. These are

```
    list      p=16F84A        ; list directive to define processor
    #include <p16F84A.inc>    ; processor specific  definitions
```

The “include” file has the following definitions and will be assumed to be always included in any of the examples that will be shown. The use of the include file makes it easier to refer to the SFRs and associated bits/flags by name. The convention used will be all uppercase for any definitions and macro assembler directives and lowercase for the instructions and variables. This coding convention must be used for all of your own course-work submissions.

```
; Contents of the p16F8A.inc file
        LIST
;===========================
;       Register Definitions
;===========================

W       EQU     H’0000’
F       EQU     H’0001’

;----- Register Files--------
; addresses of the Special Function Registers

INDF    EQU     H’0000’
TMR0    EQU     H’0001’
```

```
PCL     EQU     H'0002'
STATUS  EQU     H'0003'
FSR     EQU     H'0004'
PORTA   EQU     H'0005'
PORTB   EQU     H'0006'
EEDATA  EQU     H'0008'
EEADR   EQU     H'0009'
PCLATH  EQU     H'000A'
INTCON  EQU     H'000B'

OPTION_REG  EQU     H'0081'
TRISA       EQU     H'0085'
TRISB       EQU     H'0086'
EECON1      EQU     H'0088'
EECON2      EQU     H'0089'

;----- STATUS Bits ----------
; Bit positions for the STATUS register

IRP     EQU     H'0007'
RP1     EQU     H'0006'
RP0     EQU     H'0005'
NOT_TO  EQU     H'0004'
NOT_PD  EQU     H'0003'
Z       EQU     H'0002'
DC      EQU     H'0001'
C       EQU     H'0000'

;----- INTCON Bits ----------
; Bit positions for the INTCON register

GIE     EQU     H'0007'
EEIE    EQU     H'0006'
T0IE    EQU     H'0005'
INTE    EQU     H'0004'
RBIE    EQU     H'0003'
T0IF    EQU     H'0002'
INTF    EQU     H'0001'
RBIF    EQU     H'0000'

;----- OPTION_REG Bits ------
; Bit positions for the OPTION register

NOT_RBPU    EQU     H'0007'
INTEDG      EQU     H'0006'
```

```
T0CS        EQU     H'0005'
T0SE        EQU     H'0004'
PSA         EQU     H'0003'
PS2         EQU     H'0002'
PS1         EQU     H'0001'
PS0         EQU     H'0000'

;----- EECON1 Bits ----------
; Bit positions for the EECON1 register

EEIF    EQU     H'0004'
WRERR   EQU     H'0003'
WREN    EQU     H'0002'
WR      EQU     H'0001'
RD      EQU     H'0000'

        LIST
```

1.2 Standard template

The MPLAB assembler also has a standard template file which serves as a model of what a typical assembly language program will look like. This file is as follows:

```
        list      p=16F84A     ; list directive to define processor
        #include <p16F84A.inc> ; processor specific definitions

        __CONFIG   _CP_OFF & _WDT_ON & _PWRTE_ON & _RC_OSC

; The __CONFIG directive is used to embed configuration
; data within the .asm file.  The labels following the
; directive are located in the respective .inc file.
; See respective data sheet for additional information
; on configuration word.


;***** VARIABLE DEFINITIONS
w_temp        EQU     0x0C   ; variable used for context saving
status_temp   EQU     0x0D   ; variable used for context saving

;**************************
        ORG     0x000         ; processor reset vector
        goto    Main          ; go to beginning of main program
```

```
        ORG     0x004        ; interrupt vector location
goto ISR


ISR     movwf   w_temp       ; save W and STATUS
        movf    STATUS,W
        movwf   status_temp

; isr code goes here
; isr code goes here

        movf    status_temp,W ; restore pre-isr STATUS register
        movwf   STATUS        ;
        swapf   w_temp,F      ; restore pre-isr W register contents
        swapf   w_temp,W      ; ... without affecting the zero flag
        retfie                ; return from interrupt

Main
; main code goes here

        END                   ; directive 'end of program'
```

Notice the use of the `swapf` instead of a `movf` instruction when the W register is being restored. The `swapf` does not affect the STATUS register in any way unlike the the `movf` instruction.

1.3 Directives

There are special commands, called directives, that are specific to the MPLAB environment that help in the programming of the PIC chips. You have already seen the "ORG", "EQU" and "END" directives from the code given previously. Some are the important ones are presented here and a more complete listing can be obtained from the help file of the assembler.

BANKSEL Syntax: `BANKSEL <label>` This directive is an instruction to the linker to generate bank selecting code to set the bank to the bank containing bank containing the designated label. The linker will generate the appropriate bank selecting code. In the case of the PIC16F8x, the RP0 bit in the STATUS register will be set/cleared. The example shows sample usage.

```
        BANKSEL     TRISA       ; switch to bank 1 to read TRISA
        movwf       TRISA,W
```

DA Syntax: `DA <expr> [, <expr2>, ...,<exprn>]` This directive generates a packed 14-bit number representing two 7-bit ASCII characters. This is useful for storing strings in program memory.

```
Lett    DA   "abcdef"  ; stores the string "abcdef" into memory
```

```
                              ; ... which can be accessed at label Lett
```

DATA Syntax: `DATA <expr> [, <expr2>, ...,<exprn>]` or `DATA "<string>" [, "<string>", ...]` Initialize one ore more words of *program memory* with data. The data may be in the form of constants, relocatable or external labels or expressions of any of the previous. The data may also consist of ASCII character strings, enclosed in single quotes for one character or double quotes for strings.

```
Num     DATA  num+10          ; constants
Lett    DATA  "test string"   ; text string
Char    DATA  'N'             ; single character
Startp  DATA  start_of_prog   ; relocatable label
```

DB Syntax: `DB <expr> [, <expr2>, ...,<exprn>]` Reserve program memory words with packed 8-bit values. Multiple expressions continue to fill bytes consecutively until the end of expressions.

```
Stuff   DB   't', 0x0f, 'e', '\n'
```

DW Syntax: `DW <expr> [, <expr2>, ...,<exprn>]` Reserve program memory words for data initializing that space to specific values. Expressions can be literal strings and are stored as described by the `data` directive.

```
Init    DW   39, "diagnostic", (d_list*2+d_offset)
```

2 Simple operations

It is important to work through these exercises in the MPLAB simulator in order to get a good idea of the operation of the PIC. It is very difficult to learn the programming without actual hands-on experience.

2.1 Decrementing a 16-bit variable

Given a 16-bit variable with its low byte in `countl` and high byte in `counth`, write a routine to decrement it.

```
Decr   movf     countl,F   ; set the Zero flag if countl is zero
       btfsc    STATUS,Z   ; countl zero? decrement counth (next instr)
       decf     counth,F
       decf     countl,F   ; always decrement countl
```

2.2 Test a 16-bit variable for zero

Using the same 16-bit variable as before, test it for zero.

```
Iszero movf     countl,F   ; test for zero in low byte
       btfsc    STATUS,Z   ; if not, don't test high byte
       movf     counth,F   ; test high byte
       btfsc    STATUS,Z
       goto     Bothzero   ; Zero? branch to some appropriate place
       ...
```

3 Exercises

1. Write code to decrement a 16-bit variable and test the result for zero, branching to `Bothzero` if the result is zero. Can this be done using fewer lines of code and/or faster execution than simply concatenating the two routines shown previously?

2. It is necessary to implement a stack in software using indirect addressing. It must be 16 bytes long and can be located in any part of the GPR space. A variable `stkptr` is allocated as a pointer to the stack.

   (a) When the microcontroller is reset, does `stkptr` need to be initialized? If so, what should its initial value be?

   (b) Assuming that the stack is never popped when empty, is it necessary that it be cleared when initialized?

   (c) Assuming that more than 16 items are never pushed onto the stack, write an algorithm for the push routine and implement it in PIC assembly language.

   (d) Do the same for the above for the pop routine.

3. Another type of data structure is the queue i.e. a first-in, first-out data structure. The size of the queue in RAM is to be 16 bytes and an additional three bytes (variables) are allocated for tracking the data in the queue. The three variables are

   `inptr` Points to the next available input location.

   `outptr` Points to the next available output location.

   `qcount` Number of bytes of data in the queue.

   (a) When the CPU is reset, how should each of the 19 bytes be initialized, if at all?

   (b) Assume , simplistically, that more than 16 bytes will never be stored in the queue. Write an algorithm for the `put` routine that puts a byte of data on the queue and implement it in assembly language.

   (c) Write and implement an algorithm for the `get` routine that retrieves a byte of data from the queue. It may be assumed that a `get` will never be executed unless `qcount` is non-zero.

4 Arithmetic routines

Use the simulator, code, pencil and paper to work through each of the routines presented to ensure thorough understanding.

4.1 16-bit Subtraction

Subtraction is very similar to addition, with the inclusion of the borrow. The only difference to look out for is that the PIC16xxx treats the carry bit as a $\overline{\text{Borrow}}$. Assume that the operation required is R = Q_1 - Q_2, where Q_1 and Q_2 are two 16-bit numbers. This can be accomplished by the following code:

```
Subr    movf    Q2_lo,W
        subwf   Q1_lo,W    ; W = Q1_lo - Q2_lo
        movwf   R_lo
        btfss   STATUS,C   ; need to borrow?
        decf    Q1_hi,F    ; get it from hi byte
        movf    Q2_hi,W
        subwf   Q1_hi,W    ; W = Q1_hi - Q2_hi
        movwf   R_hi
        return
```

Note that the carry flag at the end of the routine will not be set correctly if Q2_hi is 0. An improved routine is as follows

```
Subrgd  movf    Q2_lo,W
        subwf   Q1_lo,W    ; W = Q1_lo - Q2_lo
        movwf   R_lo
        movf    Q2_hi,W
        btfss   STATUS,C   ; need to borrow?
        goto    Borrow

        subwf   Q1_hi,W    ; W = Q1_hi - Q2_hi
        movwf   R_hi
        goto    Done

Borrow  subwf   Q1_hi,W    ; W = Q1_hi - Q2_hi
        movwf   R_hi
        decf    R_hi,F     ; do the decrement after
Done    return
```

4.2 8-bit by 8-bit Unsigned Multiplication

There are two ways to implement the multiplication of two 8-bit numbers, one is by repeated addition and the other is via partial sums. The first example multiplies M1 by M2 leaving the result in R_hi and R_lo. It loops through the addition of M2 to itself M1 times.

```
; 8x8 unsigned multiply routine.
; No checks made for M1 or M2 equal to zero

        clrf    R_hi      ; clear result location
        clrf    R_lo
        clrw

M8x8    addwf   M2,W      ; add M2 to itself
        btfsc   STATUS,C  ; if carry set
        inc     R_hi,F    ; ... increment high byte
        decfsz  M1,F
        goto    M8x8
        movwf   R_lo
        return
```

This second routine, does the same thing but loops only 8 times instead of M1 times as the previous case. M1 is added to the result which is then shifted. It makes use of the fact that multiplying a number by one results in the same number so all that is required is either addition if the bit, in M2, was one or no addition if it was zero.

```
; 8x8 unsigned multiply routine.
; No checks made for M1 or M2 equal to zero

        clrf    R_hi      ; Clear result location
        clrf    R_lo

        movlw   0x08      ; setup loop count
        movwf   cntr
        movf    M1,W      ; initialize W with M1

Umul    rrf     M2,F      ; rotate into carry to check bits
        btfsc   STATUS,C  ; if carry set i.e. bit was 1
        addwf   R_hi,F    ; ... we add
        rrf     R_hi,F    ; shift over for the next addition
        rrf     R_lo,F
        decf    cntr,F    ; check the loop count
        btfss   STATUS,Z
        goto    Umul
        return
```

4.3 Division

Division can be implemented in a variety of ways, one of the simplest is integer division with no provision for a remainder. Essentially division is the reverse of addition and the divisor is repeatedly subtracted from the dividend and the quotient is incremented each time.

4.3.1 16-bit by 8-bit Unsigned Division

```
; 16-bit unsigned division
; Quot = (Div_hi, Div_lo) / Divsor

        clrf    Quot

Divide  movf    Divsor,W    ; setup for subtraction
        subwf   Div_lo,F    ; Div_lo = Div_lo - Divisor
        btfss   STATUS,C
        goto    Borrow
        goto    Div_2

Borrow  movlw   0x01
        subwf   Div_hi,F    ; use subtract instead of decf
        btfss   STATUS,C    ; ... because it sets the carry
        goto    Done        ; generated a borrow so finish

Div_2   incf    Quot,F      ; add one and loop again
        goto    Divide

Done    return
```

5 Sample questions

1. Discuss the differences between the Harvard architecture, which has separate program and data memory, and the von Neumann architecture. One of the drawbacks of the Harvard architecture is that instructions cannot execute reads of program memory. How does this affect table lookups?

2. Each instruction requires one cycle for its fetch and one cycle for its execution, yet the CPU executes a new instruction every cycle. Explain how this accomplished? Why does a branch instruction, which also requires the same two cycles for fetching and execution, introduce an extra cycle in the CPU's execution of instructions?

3. Design an algorithm and implement it in PIC assembly language to perform unsigned division of a 24-bit number by an 8-bit number and produce a 16-bit quotient and 8-bit remainder.

4. Design an algorithm that works more efficiently than repeated subtraction for unsigned 8-bit by 8-bit division. Implement it in PIC assembly language.

5. In the queue put routine, add additional functionality that checks if the end of the queue has been reached and "wraps" the input and output pointers back to the beginning of the queue.

6. In the queue get routine, add a check to prevent retrieval of data if the queue is empty.

7. In the queue put routine, add a check to prevent insertion of data if the queue is full.

6 Solutions to exercises

6.1 Question 2c

For the “push” routine, it is assumed that the value to be pushed is in the working register and that there is a variable, `temp`, that is used for temporary storage.

```
temp    EQU     0x0D      ; temporary storage

; stkptr has been assigned elsewhere
Push    movwf  temp       ; save the parameter
        incf   stkptr,F   ; point stkptr to new location
        movf   stkptr,W   ; get the stkptr into FSR
        movwf  FSR        ; ... to do indirect addressing
        movf   temp,W     ; now get the parameter
        movwf  INDF       ; store it in location stkptr points to
        return
```

6.2 Question 2d

Here there is no need for a temporary variable, and the popped parameter is returned in the working register.

```
Pop    movf   stkptr,W   ; get the stkptr into FSR
       movwf  FSR        ; ... to do indirect addressing
       movf   INDF,W     ; retrieve the value
       decf   stkptr,F   ; point to the new location
       return
```

6.3 Question 3b

Assuming that the queue variables have all been initialized and that the queue ranges from 0xB0 to 0xBF then the `put` routine can be as follows with the working register having the parameter to be stored.

```
temp    equ     0x0D
starq   equ     0xB0        ; for information only
endq    equ     0xBF        ; ditto

; assume qcount has been assigned elsewhere
Put     movwf   temp        ; Save the parameter
        incf    qcount,F    ; update number in queue
        movf    inptr,W     ; get the inptr ready for indirect
        movwf   FSR         ; ... addressing
        movf    temp,W
        movwf   INDF        ; store parameter
        incf    inptr,F     ; point to new storage locn
        return
```

6.4 Question 3c

Assume that the same definitions exist as for the queue `Put` routine. The value from the queue is returned in the working register.

```
Get     decf    qcount,F    ; reduce number in queue
        movf    outptr,W    ; setup for indirect addressing
        movwf   FSR         ; ...using outptr
        decf    outptr,F    ; point to new output locn
        movf    INDF,W      ; get value from queue
        return
```