original author: Feisal Mohammed
updated: 24th January 2002– CLR

## Part II

# PIC 16F877 Microcontroller

All microcomputer systems, irrespective of their complexity, are based on similar building blocks. These are shown in Figure 1 and consist of the following:

- CPU - the part that does all logic and arithmetic functions
- RAM - storage for programs and/or program variables
- ROM - read-only parts of programs
- I/O - connection to external devices

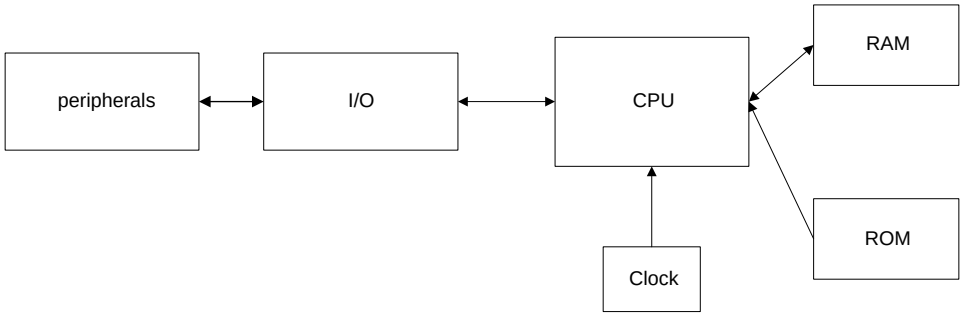


Figure 1: Basic building blocks of a computer

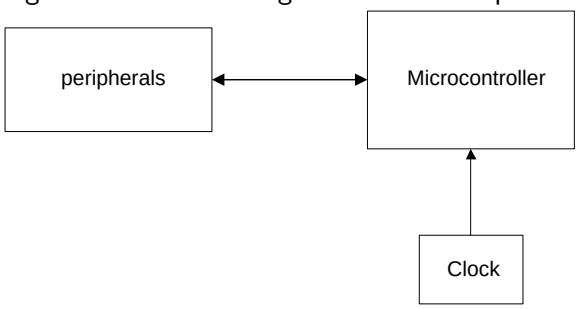


Figure 2: A microcontroller based system

The CPU or microprocessor is the core component of any microcomputer and it requires the external components such as the ROM, RAM, I/O etc. to accomplish its purpose. A microcontroller is a stripped-down version of the very same architecture, with all the important features placed on one chip. The same system as the previous figure using a microcontroller looks like Figure 2. The microcontroller based system requires no additional circuitry except a clock input and it can, in

many cases, directly drive peripheral outputs. The difference between the microprocessor and the microcontroller arises because of their different end-usage.

The microcontroller that will be investigated is the PIC16F877, which is at the upper end of the mid-range series of the microcontrollers developed by MicroChip Inc. It is characterized by a RISC architecture instead of the CISC architecture used, for example, by the Motorola 6809.

# 4 Architecture of the PIC microcontroller

The history of the PIC series of microcontrollers started in in 1965, when General Instruments (GI) formed a Microelectronics Division, and used this division to generate some of the earliest viable EPROM and EEPROM memory architectures. GI also made a 16 bit microprocessor, called the CP1600, in the early 1970s. While this was a reasonable microprocessor, it was not particularly good at handling I/O. Therefore, around 1975, GI designed a Peripheral Interface Controller (or PIC for short) for some very specific applications where good I/O handling was needed. It was designed to be very fast since it was I/O handling for a 16 bit machine, but it did not need a large amount of functionality, so its microcoded instruction set was small. Its architecture was substantially the PIC16C5x architecture of today. The market for the PIC remained small for the next few years. During the early 1980s, GI restructured their business to concentrate more on their core activity which was power semiconductors. As a result of the restructuring, the GI Microelectronics Division became GI Microelectronics Inc (a wholly owned subsidiary) which, in 1985, was finally sold to venture capital investors. The sale included the fabrication plant in Chandler, Arizona. The new owners decided to concentrate on the PICs, the serial and parallel EEPROMs and the parallel EPROMs. A decision was later taken to start a new company, named Arizona MicroChip Technology, with embedded control as its differentiator from the rest of the industry. As part of this strategy, the PIC family was redesigned to use one of the other things that the fledgling company was good at, i.e. EPROM. With the addition of CMOS technology and erasable EPROM program memory the PIC family, as we know it, was born.

The PIC series of microcontrollers are RISC-based processors with an accumulator(also called the working register, W), which use the Harvard[3] architecture; therefore the microcontroller has a program memory data bus and a data memory data bus. Separate buses mean that simultaneous access of program and data can be done, which gives a greater bandwidth over the traditional von Neumann architecture. Separating the program and data memory, allows instructions to be sized differently than the 8-bit wide data word. This separation means that the instruction words can be ideally sized for the specific CPU/application. This is necessary since RISC architectures require that instructions have the source and destination operands be encoded within the instruction. The PIC opcodes for the mid-range processors are 14-bits wide, and the 14-bit wide program bus fetches an instruction in a single cycle.

There are 35 single word instructions. A two-stage pipeline overlaps fetch and execution of instructions. As a result, all instructions execute in a single cycle except for program branches. A typical picture of the pipeline is shown in Figure 3. The pipeline uses *static branch prediction* and assumes that the branch is *never taken*, so that conditional branch instructions can take either one or two

[3]This architecture had been a scientific curiosity since its invention by Harvard University in a Defense Department funded competition that pitted Princeton against Harvard. Princeton won the competition because the MTBF of the simpler single memory architecture was much better, albeit slower, than the Harvard submission. MicroChip has made a number of enhancements to the original architecture, and updated the functional blocks of the original design with modern advancements made possible by the low cost of semiconductors.

cycles. One cycle if the branch is not taken, since it would be in the pipeline and two cycles if the branch was taken. Non-conditional branch instructions such as `call` or `goto` always take two cycles.

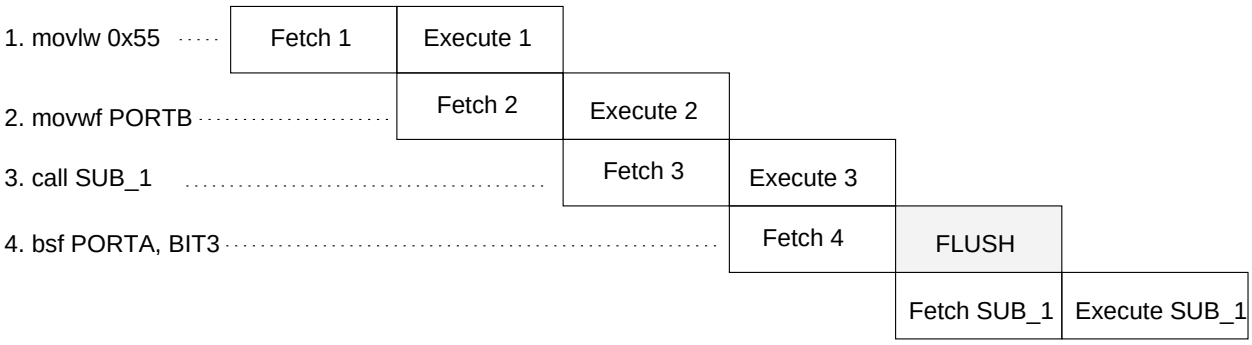


Figure 3: Instruction pipeline flow

All program memory is internal i.e. the program bus is not accessible outside of the chip. The data path is 8 bits wide. Data memory may be accessible from outside of the chip using the Parallel Slave Port; when enabled, the port register is asynchronously readable and writable by the external world. The mid-range PIC processors can directly or indirectly address their data memory[4]. All special function registers[5], including the program counter, are mapped in the data memory. The design of the instruction set is such that it is possible to carry out any operation on any register using either addressing mode. This design simplifies the programming of the device. The term *file register* is, in PIC terminology, used to denote the locations that an instruction can access via an address. The term *register file* is used to collectively refer to the group of registers.

The PIC ALU can perform arithmetic and Boolean functions between data in the working register and any file register. The unit is 8-bits wide and capable of addition, subtraction, shift and logical operations. In two-operand instructions, typically one operand is the Working (W) register and the other operand is a file register or an immediate constant. In single operand instructions, the operand is either the W register or a file register. Depending on the instruction executed, the ALU may affect the values of the Carry (C), Digit Carry (DC) and Zero (Z) bits in the STATUS register. The C and DC bits operate as $\overline{\text{Borrow}}$ and $\overline{\text{Digit Borrow}}$ bits respectively, in subtraction.

## Exercises

1. (a) Which is better: a microcontroller-based system or a microprocessor-based system? What applications is a microcontroller (as opposed to a microprocessor) typically used for?
   (b) Differentiate between the terms "file register" and "register file".
   (c) When pipelining with static branch prediction, is there an advantage in assuming that the branch is always taken?
   (d) Suggest an application where the PIC Parallel Slave Port mode would be useful.
   (e) After the addition of 9 and 8, which of the flags (Carry, Zero and Digit Carry) would be reset(i.e value 0)?

[4]Data memory may be EPROM based, or RAM based. General purpose registers are contained in the RAM
[5]Note: The Working register (W) is an 8-bit register used for all ALU operations; it is not addressable.

# 5 Memory Layout

The PIC16F877 has 8K × 14-bit words of Flash program memory, 368 bytes of data RAM, 256 bytes of data EEPROM and an 8-level x 13 bit wide hardware stack.

The Program Counter (`PC`) is 13 bits wide, thus making it possible to access all 8K x 14 addresses. The low byte is the `PCL` (Program Counter Low byte) register which is a readable and writable register. The high byte of the `PC` (`PC`<12:8>) is not directly readable or writable and comes from the `PCLATH` (Program Counter LATch High) register. The `PCLATH` register is a holding register for the `PC`<12:8>. The contents of the `PCLATH` are transferred to the upper byte of the program counter when the `PC` is loaded with a new value. Although the `PC` is capable of addressing the entire program memory space, conceptually the program memory is represented by four banks of 2K × 14-bit words. Banking is necessary since there are only 11 bits for the address in the instruction word for a `call` or `goto`. The other two bits are obtained from the top two bits of `PCLATH` (i.e. `PCLATH<4:3>`). This means that the user must set those extra bits in `PCLATH` before branching out of the 2K bank that contains the current instruction. Within the program memory space, the reset vector (location to go to on reset) is at 0000h and the interrupt vector (location to go to on interrupt) is at 0004h.

The data memory space effectively has 9 bit addresses, and is also banked. There are 4 banks; each bank holds 128 bytes of addressable memory. The banked arrangement is necessary because there are only 7 bits are available in the instruction word for the addressing of a register, which gives only 128 addresses. The selection of the banks is determined by control bits RP1, RP0 in the `STATUS` register (`STATUS`<6:5>). Together the RP1, RP0 and the specified 7 bits effectively form a 9 bit address. The first 32 locations of Banks 1 and 2, and the first 16 locations of Banks 2 and 3 are reserved for the mapping of the Special Function Registers (SFR's). SFRs (e.g. `PC`, `STATUS`) are used to control the "core" operation of the microcontroller as well as peripherals such as the I/O ports. The SFRs are mapped into the data memory space to facilitate addressing. The 368 bytes of static RAM which are used as general purpose registers (GPR), are arranged in the data memory space so that each is accessed by a unique address, with the exception of the last 16 bytes of each bank, which are shared. Each mapped SFR and GPR is 8-bits wide. The entire data memory can be accessed either directly using the absolute address of each register file, or indirectly through the INDirect File (INDF) register and the File Select Register (FSR). The INDF register is not a physical register. Any instruction using the INDF register actually accesses the register pointed to by the FSR. Reading the INDF register itself, indirectly (i.e. FSR = 0) will read 00h. Writing to the INDF register indirectly results in a no operation (although status bits may be affected). Because the FSR is 8 bits wide, indirect addressing can access two data memory banks simultaneously. The effective 9-bit address is obtained by concatenating the IRP bit (STATUS<7>) and the 8-bit FSR register.

The 256 bytes (8 bit address) of EEPROM memory is indirectly mapped into the data memory using the **EE**PROM ADRess (EEADR), **EE**PROM CONtrol (EECON1) and **EE**PROM DATA (EEDATA) registers[6]. Reading the data EEPROM memory only requires that the desired address to access be written to the EEADR register and the EEPGD bit of the EECON1 register be cleared (to indicate the data memory is to be read). Then, once the RD bit of the EECON1 register is set[7], data will be available in the EEDATA register on the very next instruction cycle. EEDATA

[6] The program memory could be read indirectly in a similar fashion, however this is a multi-cycle operation, and is not normally required.

[7] The RD bit of EECON1 will automatically clear once the read is performed

will hold this value until another read operation is initiated or until it is explicitly written.

The stack is not part of the program or data space and the stack pointer is not readable or writable. There are also no operations to place or remove data to or from the stack. Addresses are placed on the stack by the CPU when a `call` instruction is executed or when an interrupt occurs. The various return instructions then remove the previously stored address from the stack. The stack operates as a circular buffer, meaning that after the stack is full, subsequent 'pushes' start overwriting the previously stored data from the top (the very first push). Similarly when the stack is 'popped' nine times, the ninth value is the same as the first pop. The programmer never has to interact directly with the stack, but should always remember the 8-level limit. The stack does not give any indication if overflow or underflow occurs.

The 8-bit working register (W) is also not part of the program or data memory space. It can however be read/written using particular instructions. Register-Register operations are not possible.

## References

[PIC97] PICmicro Mid-Range MCU Family Reference Manual. Technical Reference 33023a, MicroChip Technology Inc., December 1997.

[PIC01] PIC16F87X Data Sheet: 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. Technical Reference 30292c, MicroChip Technology Inc., 2001.

## Exercises

1. (a) Explain the concept of banking memory space.
   (b) Give reasons why memory banks may be used in a microcontroller.
   (c) How/Where are memory banks used in the PIC 16F877?
   (d) Certain general purpose and special function registers are mapped to multiple addresses in the data memory space of the PIC 16F877. What possible advantage could such a scheme offer?
   (e) For the PIC16F877, what special considerations must be made, when performing/returning from a long subroutine call (i.e. outside the current program bank)?

2. (a) The PIC16F877 has both data EEPROM and static data RAM (general purpose registers) included in the microcontroller. Suggest reasons (using application examples) why the data EEPROM is also included in the microcontroller.
   (b) For the PIC 16F877, "The programmer never has to interact directly with the stack, but should always remember the 8-level limit". For a similar processor with a 2-level stack, illustrate (using diagrams) the problems that may occur if the programmer overruns/underruns the stack.
   (c) The SFRs discussed so far include STATUS, PCL, PCLATH, FSR, INDF, EEADR, EEDATA and EECON1. For each of the above, using the data sheet[PIC01]
      i. locate all the mapped addresses in the data memory space
      ii. describe the meaning of the register/individual bits as appropriate

# 6 Instruction set for the PIC microcontroller

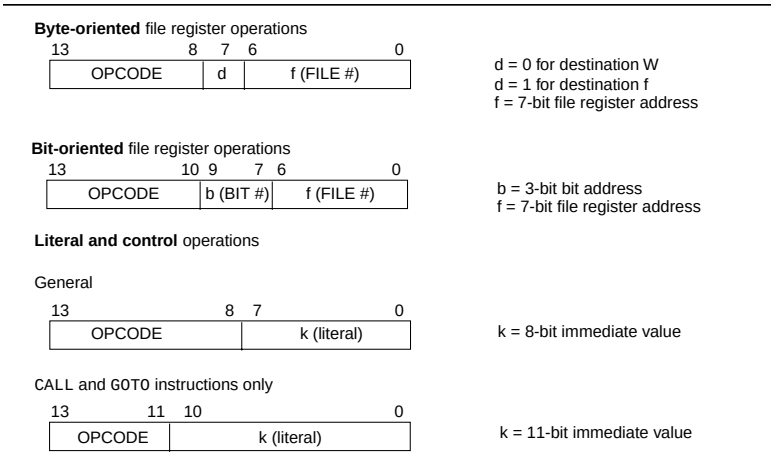


Figure 4: Format of instructions

| Mnemonic, Operands | | Description | Status Affected |
|---|---|---|---|
| **Byte-oriented file register operations** | | | |
| ADDWF | f,d | Add W and f | C,DC,Z |
| ANDWF | f,d | AND W with f | Z |
| CLRF | f | Clear f | Z |
| CLRW | - | Clear W | Z |
| COMF | f,d | Complement f | Z |
| DECF | f,d | Decrement f | Z |
| DECFSZ | f,d | Decrement f, Skip if 0 | |
| INCF | f,d | Increment f | Z |
| INCFSZ | f,d | Increment f, Skip if 0 | |
| IORWF | f,d | Inclusive OR W with f | Z |
| MOVF | f,d | Move f | Z |
| MOVWF | d | Move W to f | |
| NOP | - | No operation | |
| RLF | f,d | Rotate Left f through Carry | C |
| RRF | f,d | Rotate Right f through Carry | C |
| SUBWF | f,d | Subtract W from f | C,DC,Z |
| SWAPF | f,d | Swap nibbles in f | |
| XORWF | f,d | Exclusive OR W with f | Z |
| **Bit-oriented file register operations** | | | |
| BCF | f,b | Bit Clear f | |
| BSF | f,b | Bit Set f | |
| BTFSC | f,b | Bit Test f, Skip if Clear | |
| BTFSS | f,b | Bit Test f, Skip if Set | |
| **Literal and Control operations** | | | |
| ADDLW | k | Add literal and W | C,DC,Z |
| ANDLW | k | AND literal with W | Z |
| CALL | k | Call subroutine | |
| CLRWDT | - | Clear watchdog timer | $\overline{\text{TO}}$, $\overline{\text{PD}}$ |
| GOTO | k | Goto address | |
| IORLW | k | Inclusive OR literal with W | Z |
| MOVLW | k | Move literal to W | |
| RETFIE | - | Return from interrupt | |
| RETLW | k | Return with literal in W | |
| RETURN | - | Return from subroutine | |
| SLEEP | - | Clear watchdog timer | $\overline{\text{TO}}$, $\overline{\text{PD}}$ |
| SUBLW | k | Subtract W from literal | C,DC,Z |
| XORLW | k | Exclusive OR literal with W | Z |

| Field | Description |
|---|---|
| f | Register file address (0x00 to 0x7F) |
| W | Working register (accumulator) |
| b | Bit address within an 8-bit file register |
| k | Literal field, constant data or label |
| d | Destination select:<br>d = 0, Store result in W<br>d = 1, Store result in file register f<br>default is d = 1 |

The format of the instruction words are shown in Figure 4. All instructions can use either direct or indirect addressing to access a register file. Since the addresses range from 00h to 7Fh in each bank, only 7 bits are required to identify the register file address and this is contained within the instruction. The eighth and ninth bits, which identify the bank, then comes from the register bank select bits (RP1,RP0) from the STATUS register. Figure (5) illustrates how the bits are composed to access the appropriate address.

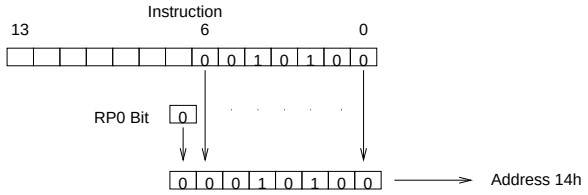


Figure 5: Direct addressing mode

In indirect addressing, the 8-bit register file address is first written to the the FSR, which is a special purpose register that is used as an address pointer to any address in the entire register file. A subsequent direct access to the INDF register will actually access the register file whose address is contained in the FSR i.e. the FSR acts as a pointer. Since the FSR is an 8-bit register, both banks can be addressed without needing to switch between them. The INDF register is not a physical register like the other special function registers. The FSR register is one of several Special Function registers that have been assigned multiple addresses so that they can be accessed in all banks. Figure (6) shows how the address is formed.

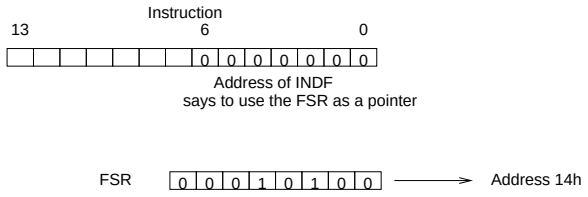


Figure 6: indirect addressing mode

The following bit of code[8] shows how to clear a section of RAM from 20h to 2Fh using indirect addressing.

```
     movlw 0x20    ; put the starting address into W
     movwf FSR     ; ...then into the FSR
Next clrf  INDF    ; clear the address pointed to by the FSR
     incf  FSR,F   ; point to the next address
     btfss FSR,4   ; does the FSR now contain 30h? (is bit 4 set?)
     goto  Next    ; not set, repeat
done  ......       ; execution jumps to here when bit 4 is set
```

[8]Please note that the code presented in this section uses two features of the PIC assembler which are not actually translated into opcodes. The first is the use of labels to represent the addresses of the following instruction (e.g. `Next` and `done`). The second is the use of labels for the Special Function Registers (e.g. `FSR`, `INDF`, `F` ). These are pre-defined in the include file for the microcontroller using the `EQU` preprocessor keyword. See Part IV for details.

The sparseness of the instruction set means that more programming effort is required in the programming of the PIC. A few of the special considerations are listed below.

**Default destination** The instructions from the table of the instruction that have `f,d` as the operands use the `d` as an indicator to tell where to place the result. The default is `f` (file register), but this can be confusing so it is better to always specify the destination directly as the following code fragment demonstrates.

```
W       EQU     H'0000' ; standard definitions
F       EQU     H'0001'; ...to avoid having to write numbers

        incf var,W  ; W = var + 1, var is unchanged
        incf var,F  ; var = var + 1, W is unchanged
```

**Subtraction** The subtract instructions (`subwf` and `sublw`) work differently than most people expect. `subwf` subtracts W *from* the contents of the register, and `sublw` subtracts W *from* the literal. Additionally the Carry flag behaves as $\overline{\text{Borrow}}$ flag when subtraction occurs i.e. if a borrow occurs the Carry flag is *not* set and vice-versa.

If you want to subtract a literal from W, it is easiest to use the `addlw` instruction with the two 's complement of the literal. For example:

```
      addlw   0feh    ; W := W - 2
```

The MicroChip assembler allows this to be written as:

```
      addlw   -2
```

There is no instruction to take the two's complement of W (like the `neg` instruction on Motorola processors), but because of the way the subtract instructions work you can use:

```
      sublw   0
```

**Comparison operations** Comparing two numbers need to be treated with care because of the non-intuitive nature of the Carry flag.

```
        movlw   2           ; W = 2
        subwf   Q,W         ; W = Q - 2
        btfss   STATUS,C
        goto    Less        ; Q < 2
        goto    Gr_eq       ; Q >= 2
```

If `Q` is less than 2, there will be a borrow therefore the Carry flag will *not* be set. The implementation of the subtraction is done by forming the two's complement of the subtrahend and then adding. If `Q` is 1 then the operation looks like the following

```
  0000 0001     ; Q = 1
  1111 1110     ; two's complement of W (= 2)
  ---------
0 1111 1111 ^---------------; Carry not set, i.e. borrow
```

If however `Q` is 3 then

```
  0000 0011     ; Q = 3
  1111 1110     ; two's complement of W (= 2)
  ---------
1 0000 0001 ^---------------; Carry set, i.e. no borrow
```

If now it is required to branch to one spot for `Q` less than or equal to W, then the subtraction will have to be turned around, for example

```
        movf    Q,W         ; W = Q
        sublw   2           ; W = 2 - W (Q)
        btfss   STATUS,C
        goto    Great       ; Q > 2
        goto    Less_eq     ; Q <= 2
```

When `Q` is equal to 3, then the operation looks like this

```
  0000 0010     ; 2
  1111 1101     ; two's complement of W (= 3)
  ---------
0 1111 1111 ^---------------; Carry not set, i.e. borrow
```

and if `Q` is equal to 1

```
  0000 0010     ;
  1111 1111     ; two's complement of W (= 1)
  --------
1 0000 0001 ^---------------; Carry set, i.e. no borrow
```

The above examples illustrates the care must be taken when using the subtraction routines and it is better to work out what actually happens than make assumptions based on other CPUs.

**STATUS register** The STATUS register contains the arithmetic status of the ALU, the RESET status and the bank select bit for the data memory. As with any register, the STATUS register can be the destination for any instruction. If the STATUS is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to the device logic. Furthermore the $\overline{TO}$ and $\overline{PD}$ bits are not writable, therefore the result of an instruction with the STATUS register as destination may be different than intended.

For example: `clrf STATUS` will not set the STATUS register to all zeros as expected but will clear the upper three bits and set the Z bit. This leaves the STATUS register as 000u u1uu (where u = unchanged). Only the `bcf`, `bsf` and `movwf` instructions should be used to alter the STATUS register because these instructions do not affect any status bits.

**Table Lookup** When there is a write to the PCL, for example in a table lookup, the PC is moved to point to a particular place in memory based on some input parameter. In the following section of code, the routine returns different binary numbers based on the value of the `W` register. `W` can range from 1 to 4.

```
table  addwf PCL, F      ; add W to the PCL and store it in the
PCL
       retlw B'00000001' ; return 1 in W if W = 1
       retlw B'00000011' ; return 3 in W if W = 2
       retlw B'00000110' ; return 6 in W if W = 3
       retlw B'00000010' ; return 2 in W if W = 4
       end
```

Since the PCL is only 8-bits wide the table has a restriction in that it cannot cross a 256 word boundary. The PCL and PCLATH are another pair of Special Function registers that are also assigned multiple addresses so that access can occur regardless of the value of the RP0, RP1 bits.

**Swapping data** The case where data must be swapped between W and a File register seems simple enough, just use a few `mov` instructions and a temporary RAM location. However upon examining the instructions, it can be seen that both the `movwf` and the `movf` instructions would require the use of the W register. That is there is no instruction that would move data between two File registers. Therefore, coding the swap routine requires *two* temporary locations and six instructions.

```
movwf   tempA
movf    data, W
movwf   tempB
movf    tempA, W
movwf   data
movf    tempB, W
```

An alternative routine swaps the working register, W, with a file register, `data`, it uses the fact that two exclusive ORs of the same value cancel out.

```
xorwf   data,W
xorwf   data,F
xorwf   data,W
```

Similarly, to swap data between two file registers where the working register contents are not needed, the direct method requires a temporary register and 6 instructions.

```
movf    dataX, W
movwf   temp
movf    dataY, W
movwf   dataX
movf    temp, W
movwf   dataY
```

This can be reduced to 4 instructions, and no temporary location as follows:

```
movf    dataX, W
subwf   dataY, W
addwf   dataX, F
subwf   dataY, F
```

## References

[PIC97] PICmicro Mid-Range MCU Family Reference Manual. Technical Reference Document 33023a, MicroChip Technology Inc., December 1997.

[PIC01] PIC16F87X Data Sheet: 28/40-Pin 8-Bit CMOS FLASH Microcontrollers. Technical Reference Document 30292c, MicroChip Technology Inc., 2001.

## Exercises

1. (a) Explain (and illustrate using pieces of code) the direct and indirect mechanisms of addressing Special Function Registers.
   (b) What happens when the assembly language statement `clrf STATUS` is executed?
   (c) Explain how the INDF register behaves when it is indirectly addressed for read/write.
   (d) Differentiate between the indirect addressing mechanisms for data RAM and data EEPROM.

2. (a) Write a piece of assembler code to branch to program location 1010h, presuming that the location is outside of the current program bank. (Note: the relevant bits must be set in PCLATH before the branch).
   (b) The `movf` instruction copies a byte from a register to the accumulator and affects the Zero flag. Write a piece of assembler code which will copy a byte from a register to accumulator and does not affect any of the status bits. (Hint: use `swapf`).
   (c) The parity of a byte is even '0' if there is an even number of bits set, and odd '1' if there is an odd number of bits set. Write a piece of assembler code that will find the parity of a byte in a register, and leave the result in another register.
   (d) The notes include a code example which clears all memory locations between 20h and 30h. Write a similar piece of code which will set the values of memory locations 40h to 60h with the value that was in the accumulator at the start of the code.

3. 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.

4. 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.

   (d) 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.

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

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

5. Each instruction requires one cycle for its fetch and one cycle for its execution, yet the PIC16F877 executes a new instruction every cycle.

   (a) Explain how this accomplished.

   (b) 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?

6. (a) One of the drawbacks of the Harvard architecture is that instructions cannot directly execute reads of program memory. How does this affect the implementation of lookup tables, and the performance of lookups?

   (b) The PIC16F877 allows an indirect read of the program memory to be performed. Describe the mechanism used, using an assembler code example.