## 4.1 Introduction

The high performance of the PICmicro™ devices can be attributed to a number of architectural features commonly found in RISC microprocessors. These include:

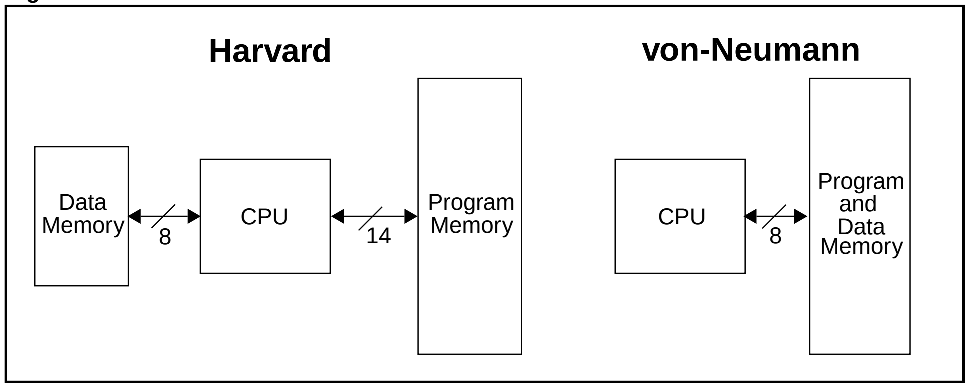
- Harvard architecture
- Long Word Instructions
- Single Word Instructions
- Single Cycle Instructions
- Instruction Pipelining
- Reduced Instruction Set
- Register File Architecture
- Orthogonal (Symmetric) Instructions

Figure 4-2 shows a simple core memory bus arrangement for Mid-Range MCU devices.

### Harvard Architecture:

Harvard architecture has the program memory and data memory as separate memories and are accessed from separate buses. This improves bandwidth over traditional von Neumann architecture in which program and data are fetched from the same memory using the same bus. To execute an instruction, a von Neumann machine must make one or more (generally more) accesses across the 8-bit bus to fetch the instruction. Then data may need to be fetched, operated on, and possibly written. As can be seen from this description, that bus can be extremely conjested. While with a Harvard architecture, the instruction is fetched in a single instruction cycle (all 14-bits). While the program memory is being accessed, the data memory is on an independent bus and can be read and written. These separated buses allow one instruction to execute while the next instruction is fetched. A comparison of Harvard vs. von-Neumann architectures is shown in Figure 4-1.

**Figure 4-1: Harvard vs. von Neumann Block Architectures**



### Long Word Instructions:

Long word instructions have a wider (more bits) instruction bus than the 8-bit Data Memory Bus. This is possible because the two buses are separate. This further allows instructions to be sized differently than the 8-bit wide data word which allows a more efficient use of the program memory, since the program memory width is optimized to the architectural requirements.

### Single Word Instructions:

Single Word instruction opcodes are 14-bits wide making it possible to have all single word instructions. A 14-bit wide program memory access bus fetches a 14-bit instruction in a single cycle. With single word instructions, the number of words of program memory locations equals the number of instructions for the device. This means that all locations are valid instructions.

Typically in the von Neumann architecture, most instructions are multi-byte. In general, a device with 4-KBytes of program memory would allow approximately 2K of instructions. This 2:1 ratio is generalized and dependent on the application code. Since each instruction may take multiple bytes, there is no assurance that each location is a valid instruction.

### Instruction Pipeline:

The instruction pipeline is a two-stage pipeline which overlaps the fetch and execution of instructions. The fetch of the instruction takes one $T_{CY}$, while the execution takes another $T_{CY}$. However, due to the overlap of the fetch of current instruction and execution of previous instruction, an instruction is fetched and another instruction is executed every single $T_{CY}$.

### Single Cycle Instructions:

With the Program Memory bus being 14-bits wide, the entire instruction is fetched in a single machine cycle ($T_{CY}$). The instruction contains all the information required and is executed in a single cycle. There may be a one cycle delay in execution if the result of the instruction modified the contents of the Program Counter. This requires the pipeline to be flushed and a new instruction to be fetched.

### Reduced Instruction Set:

When an instruction set is well designed and highly orthogonal (symmetric), fewer instructions are required to perform all needed tasks. With fewer instructions, the whole set can be more rapidly learned.

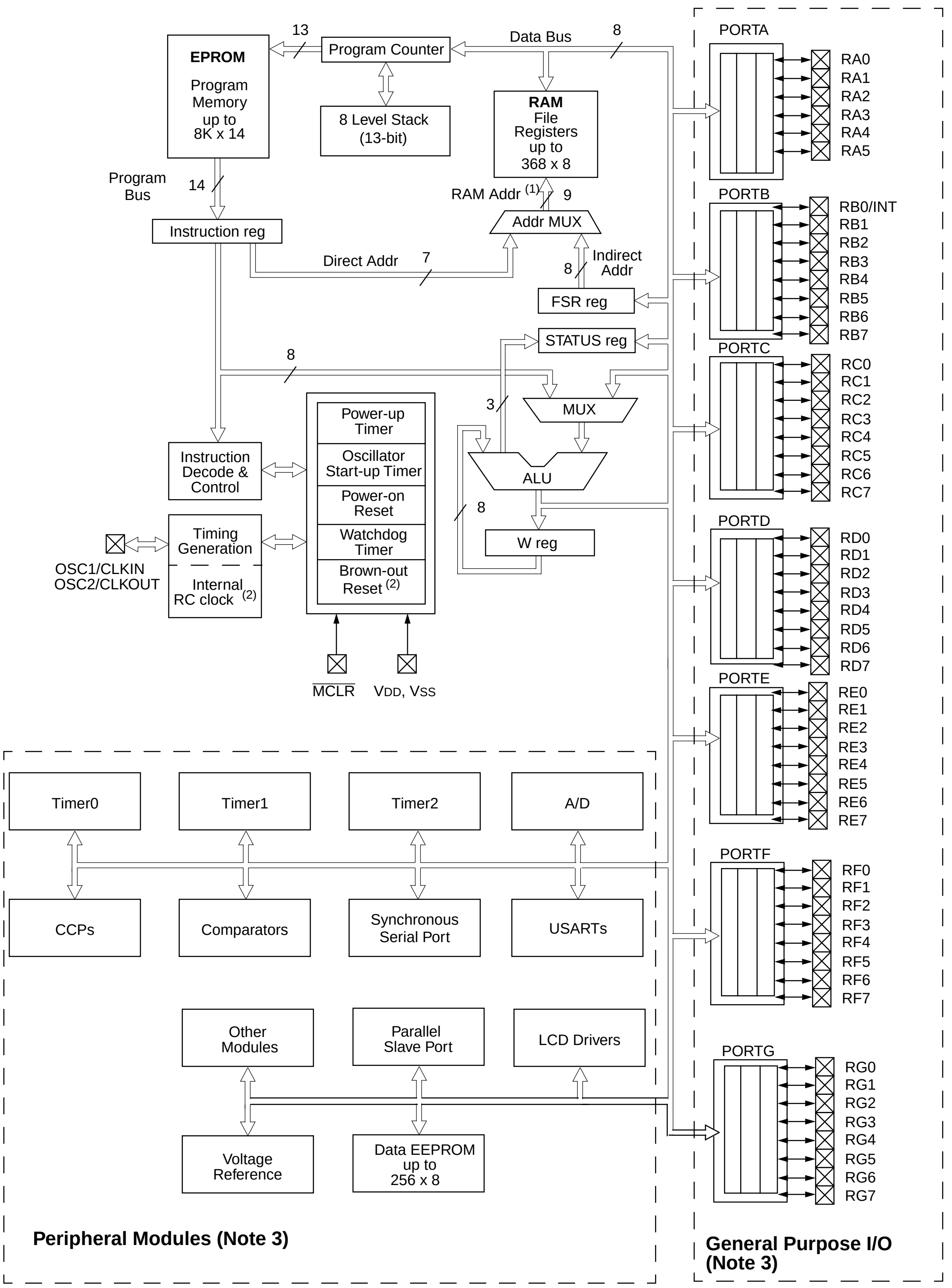
### Register File Architecture:

The register files/data memory can be directly or indirectly addressed. All special function registers, including the program counter, are mapped in the data memory.

### Orthogonal (Symmetric) Instructions:

Orthogonal instructions make it possible to carry out any operation on any register using any addressing mode. This symmetrical nature and lack of “special instructions” make programming simple yet efficient. In addition, the learning curve is reduced significantly. The mid-range instruction set uses only two non-register oriented instructions, which are used for two of the cores features. One is the `SLEEP` instruction which places the device into the lowest power use mode. The other is the `CLRWDT` instruction which verifies the chip is operating properly by preventing the on-chip Watchdog Timer (WDT) from overflowing and resetting the device.

**Figure 4-2: General Mid-range PICmicro Block Diagram**

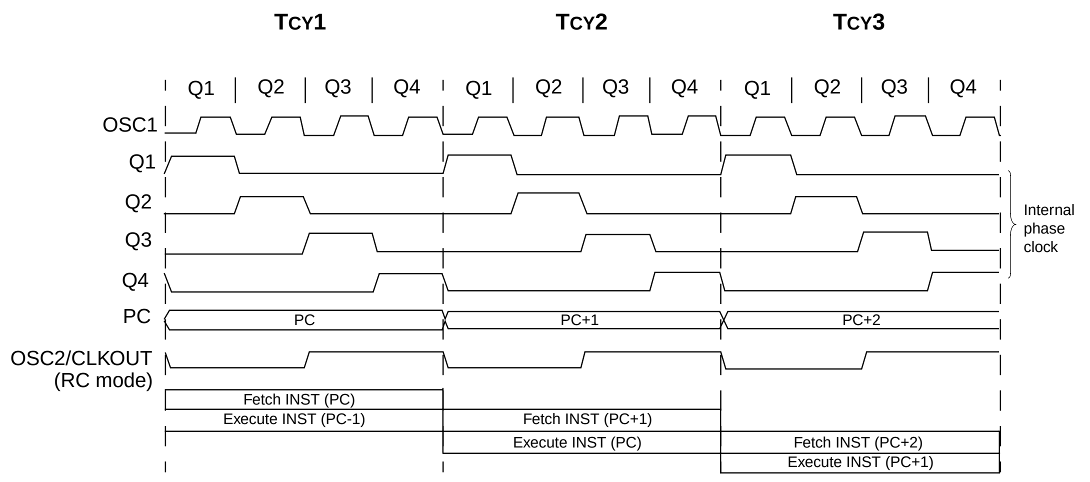


Note 1: The high order bits of the Direct Address for the RAM are from the STATUS register.
2: Not all devices have this feature, please refer to device data sheet.
3: Many of the general purpose I/O pins are multiplexed with one or more peripheral module functions. The multiplexing combinations are device dependent.

## 4.2 Clocking Scheme/Instruction Cycle

The clock input (from OSC1) is internally divided by four to generate four non-overlapping quadrature clocks, namely Q1, Q2, Q3, and Q4. Internally, the program counter (PC) is incremented every Q1, and the instruction is fetched from the program memory and latched into the instruction register in Q4. The instruction is decoded and executed during the following Q1 through Q4. The clocks and instruction execution flow are illustrated in Figure 4-3, and Example 4-1.

**Figure 4-3: Clock/Instruction Cycle**



## 4.3 Instruction Flow/Pipelining

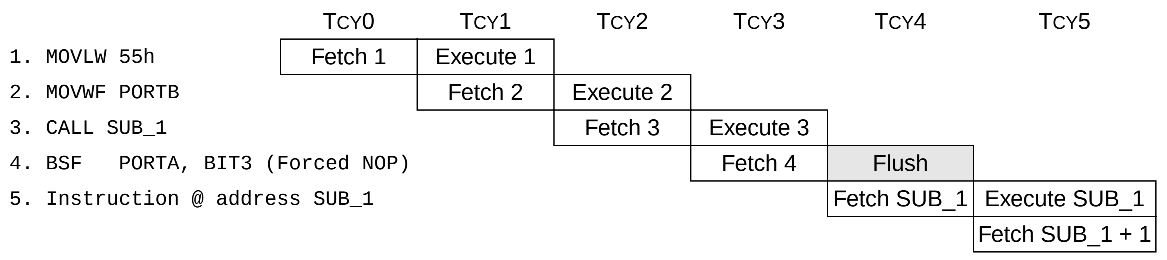
An “Instruction Cycle” consists of four Q cycles (Q1, Q2, Q3, and Q4). Fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to Pipelining, each instruction effectively executes in one cycle. If an instruction causes the program counter to change (e.g. GOTO) then an extra cycle is required to complete the instruction (Example 4-1).

The instruction **fetch** begins with the program counter incrementing in Q1.

In the **execution** cycle, the fetched instruction is latched into the “Instruction Register (IR)” in cycle Q1. This instruction is then decoded and executed during the Q2, Q3, and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write).

Example 4-1 shows the operation of the two stage pipeline for the instruction sequence shown. At time TCY0, the first instruction is fetched from program memory. During TCY1, the first instruction executes while the second instruction is fetched. During TCY2, the second instruction executes while the third instruction is fetched. During TCY3, the fourth instruction is fetched while the third instruction (CALL SUB_1) is executed. When the third instruction completes execution, the CPU forces the address of instruction four onto the Stack and then changes the Program Counter (PC) to the address of SUB_1. This means that the instruction that was fetched during TCY3 needs to be “flushed” from the pipeline. During TCY4, instruction four is flushed (executed as a NOP) and the instruction at address SUB_1 is fetched. Finally during TCY5, instruction five is executed and the instruction at address SUB_1 + 1 is fetched.

**Example 4-1: Instruction Pipeline Flow**



All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is “flushed” from the pipeline while the new instruction is being fetched and then executed.

**Table 5-1: Mid-Range MCU Instruction Set**

| Mnemonic, Operands | | Description | Cycles | 14-Bit Instruction Word | | | | Status Bits Affected | Notes |
|---|---|---|---|---|---|---|---|---|---|
| | | | | MSb | | | LSb | | |
| **BYTE-ORIENTED FILE REGISTER OPERATIONS** | | | | | | | | | |
| ADDWF | f, d | Add W and f | 1 | `00` | `0111` | `dfff` | `ffff` | C,DC,Z | 1,2 |
| ANDWF | f, d | AND W with f | 1 | `00` | `0101` | `dfff` | `ffff` | Z | 1,2 |
| CLRF | f | Clear f | 1 | `00` | `0001` | `lfff` | `ffff` | Z | 2 |
| CLRW | - | Clear W | 1 | `00` | `0001` | `0xxx` | `xxxx` | Z | |
| COMF | f, d | Complement f | 1 | `00` | `1001` | `dfff` | `ffff` | Z | 1,2 |
| DECF | f, d | Decrement f | 1 | `00` | `0011` | `dfff` | `ffff` | Z | 1,2 |
| DECFSZ | f, d | Decrement f, Skip if 0 | 1(2) | `00` | `1011` | `dfff` | `ffff` | | 1,2,3 |
| INCF | f, d | Increment f | 1 | `00` | `1010` | `dfff` | `ffff` | Z | 1,2 |
| INCFSZ | f, d | Increment f, Skip if 0 | 1(2) | `00` | `1111` | `dfff` | `ffff` | | 1,2,3 |
| IORWF | f, d | Inclusive OR W with f | 1 | `00` | `0100` | `dfff` | `ffff` | Z | 1,2 |
| MOVF | f, d | Move f | 1 | `00` | `1000` | `dfff` | `ffff` | Z | 1,2 |
| MOVWF | f | Move W to f | 1 | `00` | `0000` | `lfff` | `ffff` | | |
| NOP | - | No Operation | 1 | `00` | `0000` | `0xx0` | `0000` | | |
| RLF | f, d | Rotate Left f through Carry | 1 | `00` | `1101` | `dfff` | `ffff` | C | 1,2 |
| RRF | f, d | Rotate Right f through Carry | 1 | `00` | `1100` | `dfff` | `ffff` | C | 1,2 |
| SUBWF | f, d | Subtract W from f | 1 | `00` | `0010` | `dfff` | `ffff` | C,DC,Z | 1,2 |
| SWAPF | f, d | Swap nibbles in f | 1 | `00` | `1110` | `dfff` | `ffff` | | 1,2 |
| XORWF | f, d | Exclusive OR W with f | 1 | `00` | `0110` | `dfff` | `ffff` | Z | 1,2 |
| **BIT-ORIENTED FILE REGISTER OPERATIONS** | | | | | | | | | |
| BCF | f, b | Bit Clear f | 1 | `01` | `00bb` | `bfff` | `ffff` | | 1,2 |
| BSF | f, b | Bit Set f | 1 | `01` | `01bb` | `bfff` | `ffff` | | 1,2 |
| BTFSC | f, b | Bit Test f, Skip if Clear | 1 (2) | `01` | `10bb` | `bfff-` | `ffff` | | 3 |
| BTFSS | f, b | Bit Test f, Skip if Set | 1 (2) | `01` | `11bb` | `bfff` | `ffff` | | 3 |
| **LITERAL AND CONTROL OPERATIONS** | | | | | | | | | |
| ADDLW | k | Add literal and W | 1 | `11` | `111x` | `kkkk` | `kkkk` | C,DC,Z | |
| ANDLW | k | AND literal with W | 1 | `11` | `1001` | `kkkk` | `kkkk` | Z | |
| CALL | k | Call subroutine | 2 | `10` | `0kkk` | `kkkk` | `kkkk` | | |
| CLRWDT | - | Clear Watchdog Timer | 1 | `00` | `0000` | `0110` | `0100` | $\overline{TO}$,$\overline{PD}$ | |
| GOTO | k | Go to address | 2 | `10` | `1kkk` | `kkkk` | `kkkk` | | |
| IORLW | k | Inclusive OR literal with W | 1 | `11` | `1000` | `kkkk` | `kkkk` | Z | |
| MOVLW | k | Move literal to W | 1 | `11` | `00xx` | `kkkk` | `kkkk` | | |
| RETFIE | - | Return from interrupt | 2 | `00` | `0000` | `0000` | `1001` | | |
| RETLW | k | Return with literal in W | 2 | `11` | `01xx` | `kkkk` | `kkkk` | | |
| RETURN | - | Return from Subroutine | 2 | `00` | `0000` | `0000` | `1000` | | |
| SLEEP | - | Go into standby mode | 1 | `00` | `0000` | `0110` | `0011` | $\overline{TO}$,$\overline{PD}$ | |
| SUBLW | k | Subtract W from literal | 1 | `11` | `110x` | `kkkk` | `kkkk` | C,DC,Z | |
| XORLW | k | Exclusive OR literal with W | 1 | `11` | `1010` | `kkkk` | `kkkk` | Z | |

Note 1: When an I/O register is modified as a function of itself ( e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

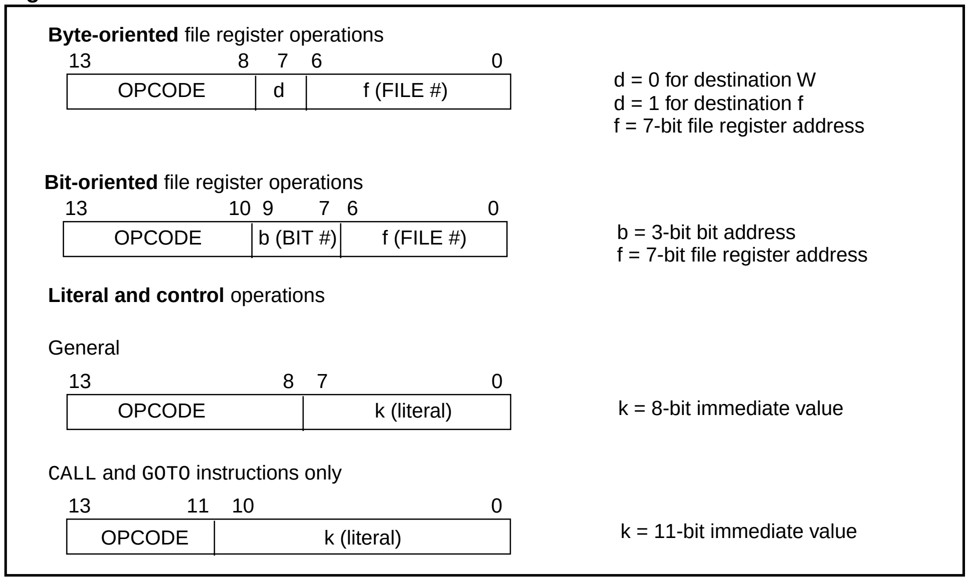
2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

## 5.2 General Instruction Format

The Mid-Range MCU instructions can be broken down into four general formats as shown in Figure 5-1. As can be seen the opcode for the instruction varies from 3-bits to 6-bits. This variable opcode size is what allows 35 instructions to be implemented.

**Figure 5-1: General Format for Instructions**


**Byte-oriented** file register operations
13 8 7 6 0
OPCODE
d
f (FILE #)
d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address
**Bit-oriented** file register operations
13 10 9 7 6 0
OPCODE
b (BIT #)
f (FILE #)
b = 3-bit bit address
f = 7-bit file register address
**Literal and control** operations
General
13 8 7 0
OPCODE
k (literal)
k = 8-bit immediate value
CALL and GOTO instructions only
13 11 10 0
OPCODE
k (literal)
k = 11-bit immediate value

## 5.3 Central Processing Unit (CPU)

The CPU can be thought of as the “brains” of the device. It is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction.

The CPU sometimes works in conjunction with the ALU to complete the execution of the instruction (in arithmetic and logical operations).

The CPU controls the program memory address bus, the data memory address bus, and accesses to the stack.

## 5.4 Instruction Clock

Each instruction cycle (TCY) is comprised of four Q cycles (Q1-Q4). The Q cycle time is the same as the device oscillator cycle time (TOSC). The Q cycles provide the timing/designation for the Decode, Read, Process Data, Write, etc., of each instruction cycle. The following diagram shows the relationship of the Q cycles to the instruction cycle.

The four Q cycles that make up an instruction cycle (TCY) can be generalized as:

Q1: Instruction Decode Cycle or forced No operation

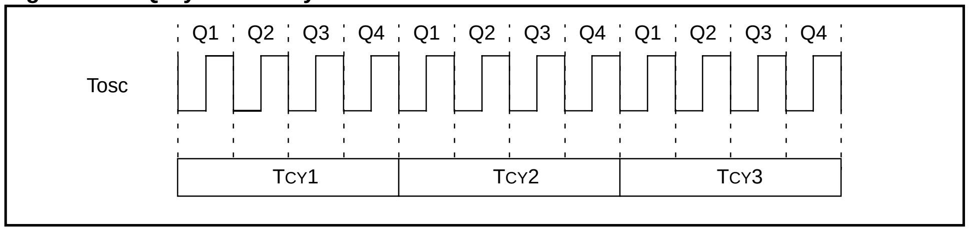
Q2: Instruction Read Data Cycle or No operation

Q3: Process the Data

Q4: Instruction Write Data Cycle or No operation

Each instruction will show a detailed Q cycle operation for the instruction.

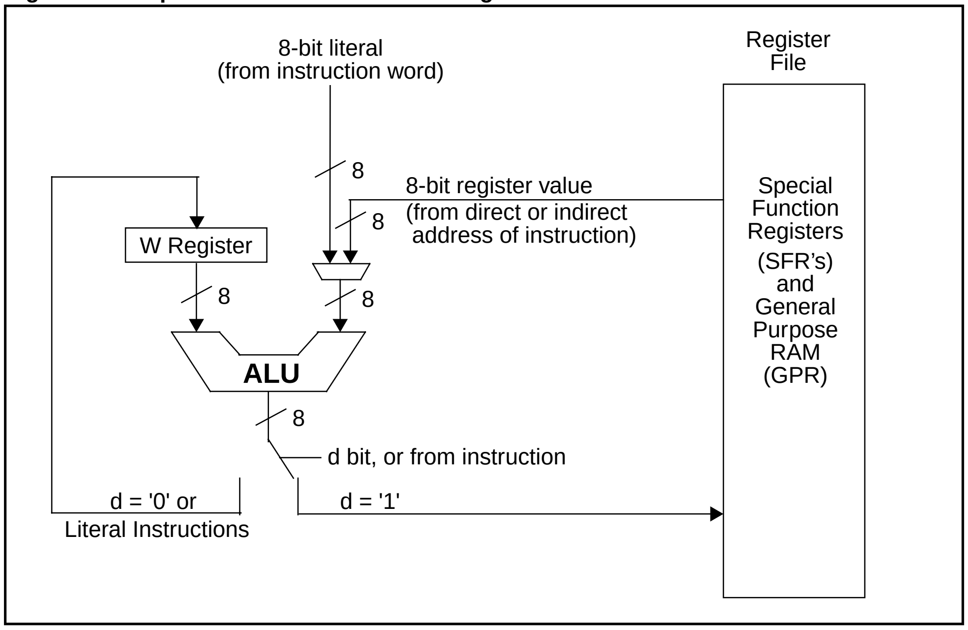
**Figure 5-2: Q Cycle Activity**



## 5.5 Arithmetic Logical Unit (ALU)

PICmicro MCUs contain an 8-bit ALU and an 8-bit working register. The ALU is a general purpose arithmetic and logical unit. It performs arithmetic and Boolean functions between the data in the working register and any register file.

**Figure 5-3: Operation of the ALU and W Register**


8-bit literal
(from instruction word)
Register File
8
8-bit register value
(from direct or indirect address of instruction)
8
W Register
8
8
ALU
8
d bit, or from instruction
d = '0' or Literal Instructions
d = '1'
Special Function Registers (SFR's) and General Purpose RAM (GPR)

The ALU is 8-bits wide and is capable of addition, subtraction, shift and logical operations. Unless otherwise mentioned, arithmetic operations are two's complement in nature. In two-operand instructions, typically one operand is the working register (W register). 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.

The W register is an 8-bit working register used for ALU operations. It is not an addressable 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 a $\overline{\text{borrow}}$ bit and a $\overline{\text{digit borrow}}$ out bit, respectively, in subtraction. See the `SUBLW` and `SUBWF` instructions for examples.

## 5.6 STATUS Register

The STATUS register, shown in Figure 5-1, contains the arithmetic status of the ALU, the RESET status and the bank select bits for data memory. Since the selection of the Data Memory banks is controlled by this register, it is required to be present in every bank. Also, this register is in the same relative position (offset) in each bank (see **Figure 6-5: "Register File Map"** in the **"Memory Organization"** section).

The STATUS register can be the destination for any instruction, as with any other register. If the STATUS register 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 $\overline{\text{TO}}$ and $\overline{\text{PD}}$ 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 clear the upper-three bits and set the Z bit. This leaves the STATUS register as `000u u1uu` (where u = unchanged).

It is recommended, therefore, that only `BCF, BSF, SWAPF` and `MOVWF` instructions are used to alter the STATUS register because these instructions do not affect the Z, C or DC bits from the STATUS register. For other instructions, not affecting any status bits, see Table 5-1.

**Note 1:** Some devices do not require the IRP and RP1 (STATUS<7:6>) bits. These bits are not used by the Section 5. CPU and ALU and should be maintained clear. Use of these bits as general purpose R/W bits is NOT recommended, since this may affect upward code compatibility with future products.

**Note 2:** The C and DC bits operate as a $\overline{\text{borrow}}$ and $\overline{\text{digit borrow}}$ bit, respectively, in subtraction.

**Register 5-1: STATUS Register**

| R/W-0 | R/W-0 | R/W-0 | R-1 | R-1 | R/W-x | R/W-x | R/W-x |
|---|---|---|---|---|---|---|---|
| IRP | RP1 | RP0 | $\overline{\text{TO}}$ | $\overline{\text{PD}}$ | Z | DC | C |
| bit 7 | | | | | | | bit 0 |

bit 7 **IRP:** Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h - 1FFh)
0 = Bank 0, 1 (00h - FFh)

For devices with only Bank0 and Bank1 the IRP bit is reserved, always maintain this bit clear.

bit 6:5 **RP1:RP0**: Register Bank Select bits (used for direct addressing)
`11` = Bank 3 (180h - 1FFh)
`10` = Bank 2 (100h - 17Fh)
`01` = Bank 1 (80h - FFh)
`00` = Bank 0 (00h - 7Fh)

Each bank is 128 bytes. For devices with only Bank0 and Bank1 the IRP bit is reserved, always maintain this bit clear.

bit 4 **$\overline{\text{TO}}$**: Time-out bit
1 = After power-up, `CLRWDT` instruction, or `SLEEP` instruction
0 = A WDT time-out occurred

bit 3 **$\overline{\text{PD}}$**: Power-down bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit carry/$\overline{\text{borrow}}$ bit (`ADDWF`, `ADDLW`, `SUBLW`, `SUBWF` instructions) (for $\overline{\text{borrow}}$ the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{\text{borrow}}$ bit (`ADDWF`, `ADDLW`,`SUBLW`,`SUBWF` instructions)
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result occurred

**Note:** For $\overline{\text{borrow}}$ the polarity is reversed. A subtraction is executed by adding the two's complement of the second operand. For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high or low order bit of the source register.

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as '0' | - n = Value at POR reset |

## 5.7 OPTION_REG Register

The OPTION_REG register is a readable and writable register which contains various control bits to configure the TMR0/WDT prescaler, the external INT Interrupt, TMR0, and the weak pull-ups on PORTB.

**Register 5-2: OPTION_REG Register**

| R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 |
|---|---|---|---|---|---|---|---|
| $\overline{\text{RBPU}}$ | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 |
| bit 7 | | | | | | | bit 0 |

bit 7 **$\overline{\text{RBPU}}$:** PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of INT pin
0 = Interrupt on falling edge of INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

| Bit Value | TMR0 Rate | WDT Rate |
|---|---|---|
| 000 | 1 : 2 | 1 : 1 |
| 001 | 1 : 4 | 1 : 2 |
| 010 | 1 : 8 | 1 : 4 |
| 011 | 1 : 16 | 1 : 8 |
| 100 | 1 : 32 | 1 : 16 |
| 101 | 1 : 64 | 1 : 32 |
| 110 | 1 : 128 | 1 : 64 |
| 111 | 1 : 256 | 1 : 128 |

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as ‘0’ | - n = Value at POR reset |

**Note:** To achieve a 1:1 prescaler assignment for the TMR0 register, assign the prescaler to the Watchdog Timer.

## 5.8 PCON Register

The Power Control (PCON) register contains flag bit(s), that together with the TO and PD bits, allows the user to differentiate between the device resets.

**Note 1:** $\overline{\text{BOR}}$ is unknown on Power-on Reset. It must then be set by the user and checked on subsequent resets to see if $\overline{\text{BOR}}$ is clear, indicating a brown-out has occurred. The $\overline{\text{BOR}}$ status bit is a don't care and is not necessarily predictable if the brown-out circuit is disabled (by clearing the BODEN bit in the Configuration word).

**Note 2:** It is recommended that the $\overline{\text{POR}}$ bit be cleared after a power-on reset has been detected, so that subsequent power-on resets may be detected.

**Register 5-3: PCON Register**

| R-u | U-0 | U-0 | U-0 | U-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| MPEEN | — | — | — | — | $\overline{\text{PER}}$ | $\overline{\text{POR}}$ | $\overline{\text{BOR}}$ |
| bit 7 | | | | | | | bit 0 |

bit 7 **MPEEN**: Memory Parity Error Circuitry Status bit
This bit reflects the value of the MPEEN configuration bit.

bit 6:3 **Unimplemented:** Read as '0'

bit 2 **$\overline{\text{PER}}$**: Memory Parity Error Reset Status bit
1 = No error occurred
0 = A program memory fetch parity error occurred
(must be set in software after a Power-on Reset occurs)

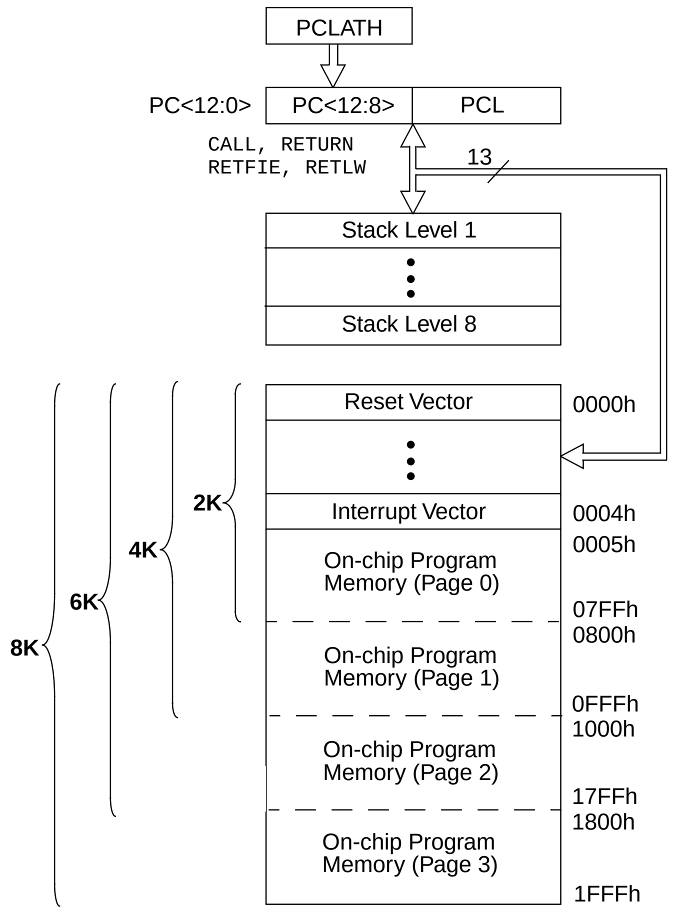
bit 1 **$\overline{\text{POR}}$**: Power-on Reset Status bit
1 = No Power-on Reset occurred
0 = A Power-on Reset occurred (must be set in software after a Power-on Reset occurs)

bit 0 **$\overline{\text{BOR}}$**: Brown-out Reset Status bit
1 = No Brown-out Reset occurred
0 = A Brown-out Reset occurred (must be set in software after a Brown-out Reset occurs)

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as ‘0’ | - n = Value at POR reset |

# Section 6. Memory Organization

**Figure 6-1: Architectural Program Memory Map and Stack**



Note 1: Not all devices implement the entire program memory space
2: Calibration Data may be programmed into program memory locations.

### 6.2.1 Reset Vector

On any device, a reset forces the Program Counter (PC) to address 0h. We call this address the “Reset Vector Address” since this is the address that program execution will branch to when a device reset occurs.

Any reset will also clear the contents of the PCLATH register. This means that any branch at the Reset Vector Address (0h) will jump to that location in PAGE0 of the program memory.

### 6.2.2 Interrupt Vector

When an interrupt is acknowledged the PC is forced to address 0004h. We call this the “Interrupt Vector Address”. When the PC is forced to the interrupt vector, the PCLATH register is not modified. Once in the service interrupt routine (ISR), this means that before any write to the PC, the PCLATH register should be written with the value that will specify the desired location in program memory. Before the PCLATH register is modified by the Interrupt Service Routine (ISR) the contents of the PCLATH may need to be saved, so it can be restored before returning from the ISR.

### 6.2.3 Calibration Information

Some devices have calibration information stored in their program memory. This information is programmed by Microchip when the device is under final test. The use of these values allows the application to achieve better results. The calibration information is typically at the end of program memory, and is implemented as a `RETLW` instruction with the literal value being the specified calibration information.

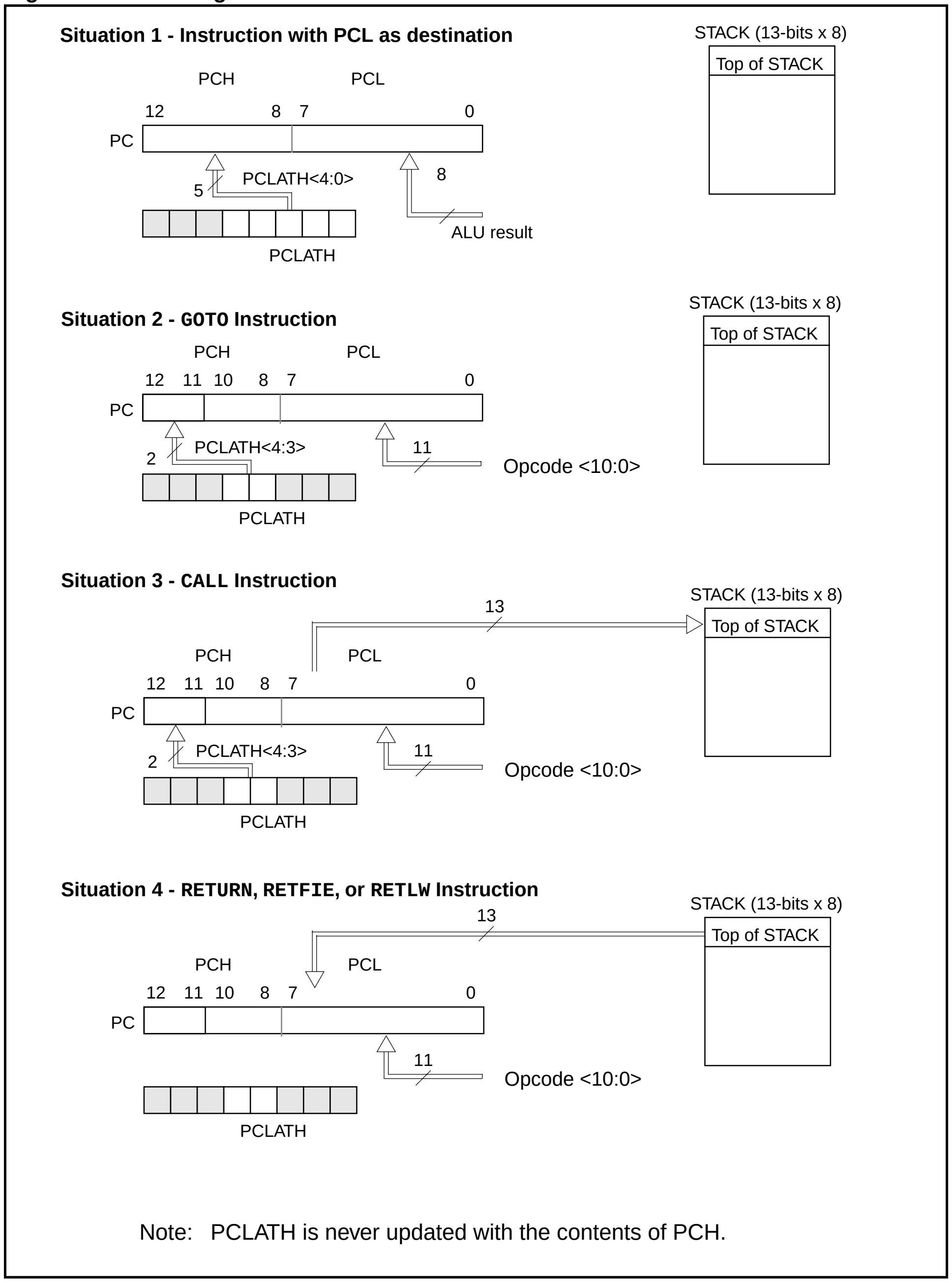
**Note:** For windowed devices, write down all calibration values **BEFORE** erasing. This allows the device’s calibration values to be restored when the device is re-programmed. When possible writing the values on the package is recommended.

### 6.2.4 Program Counter (PC)

The program counter (PC) specifies the address of the instruction to fetch for execution. The PC is 13-bits wide. The low byte is called the PCL register. This register is readable and writable. The high byte is called the PCH register. This register contains the PC<12:8> bits and is not directly readable or writable. All updates to the PCH register go through the PCLATH register.

Figure 6-2 shows the four situations for the loading of the PC. Situation 1 shows how the PC is loaded on a write to PCL (PCLATH<4:0> → PCH). Situation 2 shows how the PC is loaded during a `GOTO` instruction (PCLATH<4:3> → PCH). Situation 3 shows how the PC is loaded during a `CALL` instruction (PCLATH<4:3> → PCH), with the PC loaded (PUSHed) onto the Top of Stack. Situation 4 shows how the PC is loaded during one of the return instructions where the PC loaded (POPed) from the Top of Stack.

**Figure 6-2: Loading of PC In Different Situations**



#### 6.2.4.1 Computed GOTO

A computed GOTO is accomplished by adding an offset to the program counter (`ADDWF PCL`). When doing a table read using a computed GOTO method, care should be exercised if the table location crosses a PCL memory boundary (each 256 byte block).

**Note:** Any write to the Program Counter (PCL), will cause the lower five bits of the PCLATH to be loaded into PCH.

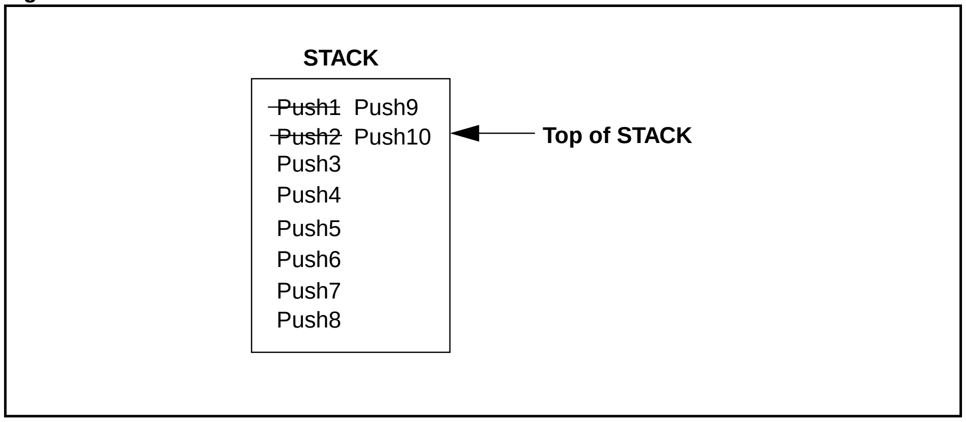
### 6.2.5 Stack

The stack allows a combination of up to 8 program calls and interrupts to occur. The stack contains the return address from this branch in program execution.

Mid-Range MCU devices have an 8-level deep x 13-bit wide hardware stack. The stack space is not part of either program or data space and the stack pointer is not readable or writable. The PC is PUSHed onto the stack when a `CALL` instruction is executed or an interrupt causes a branch. The stack is POPed in the event of a `RETURN, RETLW` or a `RETFIE` instruction execution. PCLATH is not modified when the stack is PUSHed or POPed.

After the stack has been PUSHed eight times, the ninth push overwrites the value that was stored from the first push. The tenth push overwrites the second push (and so on). An example of the overwriting of the stack is shown in Figure 6-3.

**Figure 6-3: Stack Modification**



**Note 1:** There are no status bits to indicate stack overflow or stack underflow conditions.

**Note 2:** There are no instructions/mnemonics called PUSH or POP. These are actions that occur from the execution of the `CALL, RETURN, RETLW,` and `RETFIE` instructions, or the vectoring to an interrupt address.

### 6.2.6 Program Memory Paging

Some devices have program memory sizes greater then 2K words, but the CALL and GOTO instructions only have a 11-bit address range. This 11-bit address range allows a branch within a 2K program memory page size. To allow CALL and GOTO instructions to address the entire 1K program memory address range, there must be another two bits to specify the program memory page. These paging bits come from the PCLATH<4:3> bits (Figure 6-2). When doing a CALL or GOTO instruction, the user must ensure that page bits (PCLATH<4:3>) are programmed so that the desired program memory page is addressed (Figure 6-2). When one of the return instructions is executed, the entire 13-bit PC is POPed from the stack. Therefore, manipulation of the PCLATH<4:3> is not required for the return instructions.

**Note:** Devices with program memory sizes 2K words and less, ignore both paging bits (PCLATH<4:3>), which are used to access program memory when more than one page is available. The use of PCLATH<4:3> as general purpose read/write bits (for these devices) is not recommended since this may affect upward compatibility with future products.

Devices with program memory sizes between 2K words and 4K words, ignore the paging bit (PCLATH<4>), which is used to access program memory pages 2 and 3 (1000h - 1FFFh). The use of PCLATH<4> as a general purpose read/write bit (for these devices) is not recommended since this may affect upward compatibility with future products.

Example 6-1 shows the calling of a subroutine in page 1 of the program memory. This example assumes that PCLATH is saved and restored by the interrupt service routine (if interrupts are used).

**Example 6-1: Call of a Subroutine in Page1 from Page0**

```
     ORG 0x500
     BSF    PCLATH,3  ; Select Page1 (800h-FFFh)
     CALL   SUB1_P1   ; Call subroutine in Page1 (800h-FFFh)
       :              ;
       :              ;
     ORG 0x900        ;
SUB1_P1:              ; called subroutine Page1 (800h-FFFh)
       :              ;
     RETURN           ; return to Call subroutine in Page0 (000h-7FFh)
                      ;
```

## 6.3 Data Memory Organization

Data memory is made up of the Special Function Registers (SFR) area, and the General Purpose Registers (GPR) area. The SFRs control the operation of the device, while GPRs are the general area for data storage and scratch pad operations.

The data memory is banked for both the GPR and SFR areas. The GPR area is banked to allow greater than 96 bytes of general purpose RAM to be addressed. SFRs are for the registers that control the peripheral and core functions. Banking requires the use of control bits for bank selection. These control bits are located in the STATUS Register (STATUS<7:5>). Figure 6-5 shows one of the data memory map organizations, this organization is device dependent.

To move values from one register to another register, the value must pass through the W register. This means that for all register-to-register moves, two instruction cycles are required.

The entire data memory can be accessed either directly or indirectly. Direct addressing may require the use of the RP1:RP0 bits. Indirect addressing requires the use of the File Select Register (FSR). Indirect addressing uses the Indirect Register Pointer (IRP) bit of the STATUS register for accesses into the Bank0 / Bank1 or the Bank2 / Bank3 areas of data memory.

### 6.3.1 General Purpose Registers (GPR)

Some Mid-Range MCU devices have banked memory in the GPR area. GPRs are not initialized by a Power-on Reset and are unchanged on all other resets.

The register file can be accessed either directly, or using the File Select Register FSR, indirectly. Some devices have areas that are shared across the data memory banks, so a read / write to that area will appear as the same location (value) regardless of the current bank. We refer to this area as the Common RAM.

### 6.3.2 Special Function Registers (SFR)

The SFRs are used by the CPU and Peripheral Modules for controlling the desired operation of the device. These registers are implemented as static RAM.

The SFRs can be classified into two sets, those associated with the “core” function and those related to the peripheral functions. Those registers related to the “core” are described in this section, while those related to the operation of the peripheral features are described in the section of that peripheral feature.

All Mid-Range MCU devices have banked memory in the SFR area. Switching between these banks requires the RP0 and RP1 bits in the STATUS register to be configured for the desired bank. Some SFRs are initialized by a Power-on Reset and other resets, while other SFRs are unaffected.

**Note:** The Special Function Register (SFR) Area may have General Purpose Registers (GPRs) mapped in these locations.

The register file can be accessed either directly, or using the File Select Register FSR, indirectly.

### 6.3.3 Banking

The data memory is partitioned into four banks. Each bank contains General Purpose Registers and Special Function Registers. Switching between these banks requires the RP0 and RP1 bits in the STATUS register to be configured for the desired bank when using direct addressing. The IRP bit in the STATUS register is used for indirect addressing.

**Table 6-1: Direct and Indirect Addressing of Banks**

| Accessed Bank | Direct (RP1:RP0) | Indirect (IRP) |
|---|---|---|
| 0 | 0 0 | 0 |
| 1 | 0 1 | |
| 2 | 1 0 | 1 |
| 3 | 1 1 | |

Each Bank extends up to 7Fh (128 bytes). The lower locations of each bank are reserved for the Special Function Registers. Above the Special Function Registers are General Purpose Registers. All data memory is implemented as static RAM. All Banks may contain special function registers. Some "high use" special function registers from Bank0 are mirrored in the other banks for code reduction and quicker access.

Through the evolution of the products, there are a few variations in the layout of the Data Memory. The data memory organization that will be the standard for all new devices is shown in Figure 6-5. This Memory map has the last 16-bytes mapped across all memory banks. This is to reduce the software overhead for context switching. The registers in **bold** will be in every device. The other registers are peripheral dependent. Not every peripheral's registers are shown, because some file addresses have a different registers from those shown. As with all the figures, tables, and specifications presented in this reference guide, verify the details with the device specific data sheet.

**Figure 6-4: Direct Addressing**

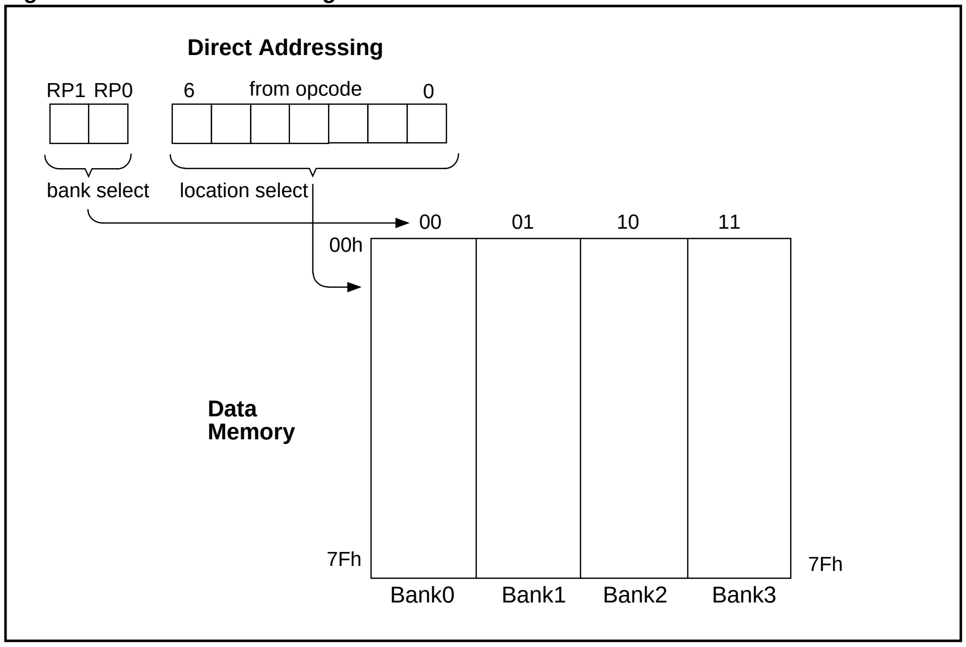


**Figure 6-5: Register File Map**

| | File Address | | File Address | | File Address | | File Address |
|---|---|---|---|---|---|---|---|
| **INDF** | 00h | **INDF** | 80h | **INDF** | 100h | **INDF** | 180h |
| **TMR0** | 01h | **OPTION_REG** | 81h | **TMR0** | 101h | **OPTION_REG** | 181h |
| **PCL** | 02h | **PCL** | 82h | **PCL** | 102h | **PCL** | 182h |
| **STATUS** | 03h | **STATUS** | 83h | **STATUS** | 103h | **STATUS** | 183h |
| **FSR** | 04h | **FSR** | 84h | **FSR** | 104h | **FSR** | 184h |
| PORTA | 05h | TRISA | 85h | ▨ | 105h | ▨ | 185h |
| PORTB | 06h | TRISB | 86h | PORTB | 106h | TRISB | 186h |
| PORTC | 07h | TRISC | 87h | PORTF | 107h | TRISF | 187h |
| PORTD | 08h | TRISD | 88h | PORTG | 108h | TRISG | 188h |
| PORTE | 09h | TRISE | 89h | ▨ | 109h | ▨ | 189h |
| **PCLATH** | 0Ah | **PCLATH** | 8Ah | **PCLATH** | 10Ah | **PCLATH** | 18Ah |
| **INTCON** | 0Bh | **INTCON** | 8Bh | **INTCON** | 10Bh | **INTCON** | 18Bh |
| **PIR1** | 0Ch | **PIE1** | 8Ch | ▨ | 10Ch | ▨ | 18Ch |
| PIR2 | 0Dh | PIE2 | 8Dh | ▨ | 10Dh | ▨ | 18Dh |
| TMR1L | 0Eh | **PCON** | 8Eh | ▨ | 10Eh | ▨ | 18Eh |
| TMR1H | 0Fh | OSCCAL | 8Fh | ▨ | 10Fh | ▨ | 18Fh |
| T1CON | 10h | ▨ | 90h | ▨ | 110h | ▨ | 190h |
| TMR2 | 11h | ▨ | 91h | ▨ | 111h | ▨ | 191h |
| T2CON | 12h | PR2 | 92h | ▨ | 112h | ▨ | 192h |
| SSPBUF | 13h | SSPADD | 93h | ▨ | 113h | ▨ | 193h |
| SSPCON | 14h | SSPATAT | 94h | ▨ | 114h | ▨ | 194h |
| CCPR1L | 15h | ▨ | 95h | ▨ | 115h | ▨ | 195h |
| CCPR1H | 16h | ▨ | 96h | ▨ | 116h | ▨ | 196h |
| CCP1CON | 17h | ▨ | 97h | ▨ | 117h | ▨ | 197h |
| RCSTA | 18h | TXSTA | 98h | ▨ | 118h | ▨ | 198h |
| TXREG | 19h | SPBRG | 99h | ▨ | 119h | ▨ | 199h |
| RCREG | 1Ah | ▨ | 9Ah | ▨ | 11Ah | ▨ | 19Ah |
| CCPR2L | 1Bh | ▨ | 9Bh | ▨ | 11Bh | ▨ | 19Bh |
| CCPR2H | 1Ch | ▨ | 9Ch | ▨ | 11Ch | ▨ | 19Ch |
| CCP2CON | 1Dh | ▨ | 9Dh | ▨ | 11Dh | ▨ | 19Dh |
| ADRES | 1Eh | ▨ | 9Eh | ▨ | 11Eh | ▨ | 19Eh |
| ADCON0 | 1Fh | ADCON1 | 9Fh | ▨ | 11Fh | ▨ | 19Fh |
| General Purpose Registers [(2)] | 20h | General Purpose Registers [(3)] | A0h | General Purpose Registers [(3)] | 120h | General Purpose Registers [(3)] | 1A0h |
| | | | EFh | | 16Fh | | 1EFh |
| | | Mapped in Bank0 70h - 7Fh [(4)] | F0h | Mapped in Bank0 70h - 7Fh [(4)] | 170h | Mapped in Bank0 70h - 7Fh [(4)] | 1F0h |
| | 7Fh | | FFh | | 17Fh | | 1FFh |
| Bank0 | | Bank1 | | Bank2 [(5)] | | Bank3 [(5)] | |

Note 1: Registers in **BOLD** will be present in every device.
2: Not all locations may be implemented. Unimplemented locations will read as '0'.
3: These locations may not be implemented. Depending on the device, accesses to the unimplemented locations operate differently. Please refer to the specific device data sheet for details.
4: Some device do not map these registers into Bank0. In devices where these registers are mapped into Bank0, these registers are referred to as common RAM
5: Some devices may not implement these banks. Locations in unimplemented banks will read as '0'.
6: General Purpose Registers (GPRs) may be located in the Special Function Register (SFR) area.

The map in Figure 6-6 shows the register file memory map of some 18-pin devices. Unimplemented registers will read as '0'.

**Figure 6-6: Register File Map**

| Bank0 | File Address | Bank1 | File Address |
|---|---|---|---|
| **INDF** | 00h | **INDF** | 80h |
| **TMR0** | 01h | **OPTION_REG** | 81h |
| **PCL** | 02h | **PCL** | 82h |
| **STATUS** | 03h | **STATUS** | 83h |
| **FSR** | 04h | **FSR** | 84h |
| **PORTA** | 05h | **TRISA** | 85h |
| **PORTB** | 06h | **TRISB** | 86h |
| ▨ | 07h | **PCON** | 87h |
| ADCON0 / EEDATA (2) | 08h | ADCON1 / EECON1 (2) | 88h |
| ADRES / EEADR (2) | 09h | ADRES / EECON2 (2) | 89h |
| **PCLATH** | 0Ah | **PCLATH** | 8Ah |
| **INTCON** | 0Bh | **INTCON** | 8Bh |
| General Purpose Registers (3) | 0Ch – 7Fh | General Purpose Registers (4) | 8Ch – FFh |

Note 1: Registers in **BOLD** will be present in every device.
2: These registers may not be implemented, or are implemented as other registers in some devices.
3: Not all locations may be implemented. Unimplemented locations will read as ’0’.
4: These locations are unimplemented in Bank1. Access to these unimplemented locations will access the corresponding Bank0 register.

### 6.3.4 Indirect Addressing, INDF, and FSR Registers

Indirect addressing is a mode of addressing data memory where the data memory address in the instruction is not fixed. An SFR register is used as a pointer to the data memory location that is to be read or written. Since this pointer is in RAM, the contents can be modified by the program. This can be useful for data tables in the data memory. Figure 6-7 shows the operation of indirect addressing. This shows the moving of the value to the data memory address specified by the value of the FSR register.

Indirect addressing is possible by using the INDF register. Any instruction using the INDF register actually accesses the register pointed to by the File Select Register, FSR. Reading the INDF register itself indirectly (FSR = '0') will read 00h. Writing to the INDF register indirectly results in a no-operation (although status bits may be affected). An effective 9-bit address is generated by the concatenation of the IRP bit (STATUS<7>) with the 8-bit FSR register, as shown in Figure 6-8.

**Figure 6-7: Indirect Addressing**

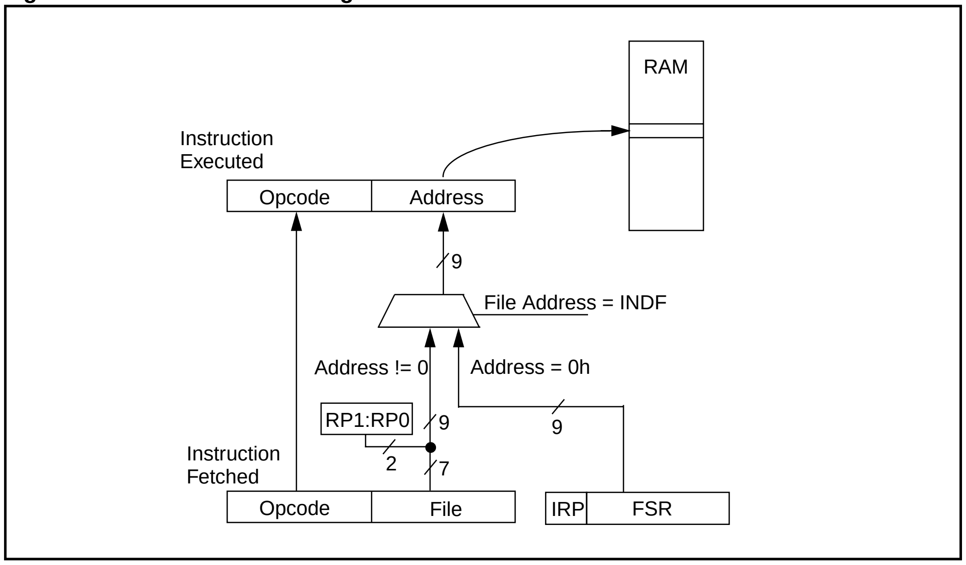
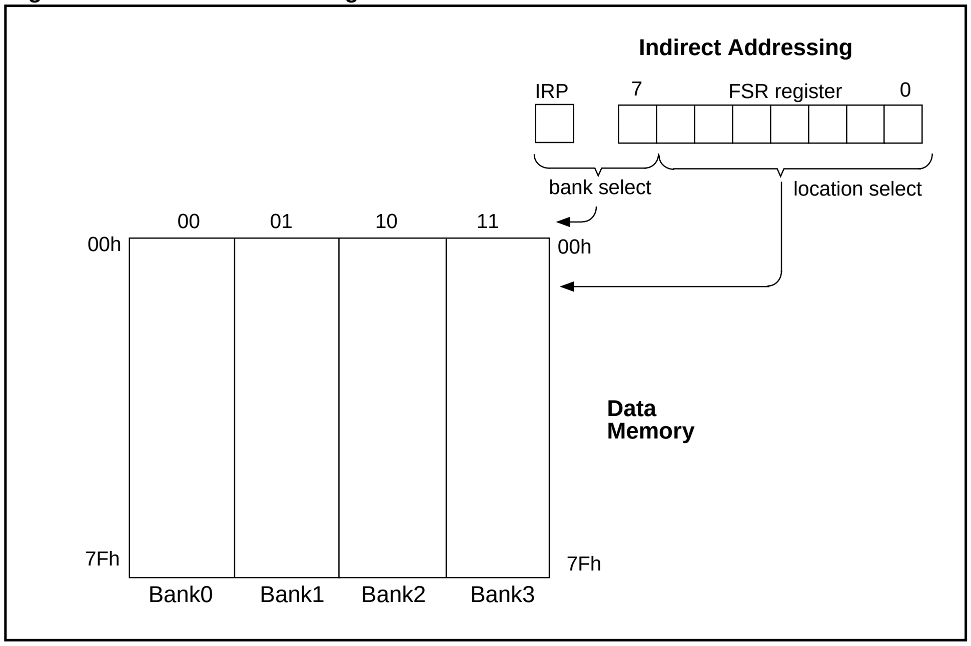


Figure 6-8: Indirect Addressing



Example 6-2 shows a simple use of indirect addressing to clear RAM (locations 20h-2Fh) in a minimum number of instructions. A similar concept could be used to move a defined number of bytes (block) of data to the USART transmit register (TXREG). The starting address of the block of data to be transmitted could easily be modified by the program.

**Example 6-2: Indirect Addressing**

```
         BCF      STATUS, IRP    ; Indirect addressing Bank0/1
         MOVLW    0x20           ; Initialize pointer to RAM
         MOVWF    FSR            ;
NEXT     CLRF     INDF           ; Clear INDF register
         INCF     FSR,F          ; Inc pointer
         BTFSS    FSR,4          ; All done?
         GOTO     NEXT           ; NO, clear next
CONTINUE                         ;
         :                       ; YES, continue
```

## 6.4 Initialization

Example 6-3 shows how the bank switching occurs for Direct addressing, while Example 6-4 shows some code to do initialization (clearing) of General Purpose RAM.

**Example 6-3: Bank Switching**

```
CLRF   STATUS        ; Clear STATUS register (Bank0)
:                    ;
BSF    STATUS, RP0   ; Bank1
:                    ;
BCF    STATUS, RP0   ; Bank0
:                    ;
MOVLW  0x60          ; Set RP0 and RP1 in STATUS register, other
XORWF  STATUS, F     ;   bits unchanged (Bank3)
:                    ;
BCF    STATUS, RP0   ; Bank2
:                    ;
BCF    STATUS, RP1   ; Bank0
```

## 6.5 Design Tips

**Question 1:** ***Program execution seems to get lost.***

**Answer 1:**

When a device with more then 2K words of program memory is used, the calling of subroutines may require that the PCLATH register be loaded prior to the CALL (or GOTO) instruction to specify the correct program memory page that the routine is located on. The following instructions will correctly load PCLATH register, regardless of the program memory location of the label SUB_1.

```
          MOVLW    HIGH (SUB_1)    ; Select Program Memory Page of
          MOVWF    PCLATH          ;     Routine.
          CALL     SUB_1           ; Call the desired routine
          :
          :
SUB_1     :                        ; Start of routine
          :
          RETURN                   ; Return from routine
```

**Question 2:** ***I need to initialize RAM to '0's. What is an easy way to do that?***

**Answer 2:**

Example 6-4 shows this. If the device you are using does not use all 4 data memory banks, some of the code may be removed.

**Table 29-1: Midrange Instruction Set**

| Mnemonic, Operands | | Description | Cycles | 14-Bit Instruction Word | | | | Status Affected | Notes |
|---|---|---|---|---|---|---|---|---|---|
| | | | | MSb | | | LSb | | |
| **BYTE-ORIENTED FILE REGISTER OPERATIONS** | | | | | | | | | |
| ADDWF | f, d | Add W and f | 1 | 00 | 0111 | dfff | ffff | C,DC,Z | 1,2 |
| ANDWF | f, d | AND W with f | 1 | 00 | 0101 | dfff | ffff | Z | 1,2 |
| CLRF | f | Clear f | 1 | 00 | 0001 | lfff | ffff | Z | 2 |
| CLRW | - | Clear W | 1 | 00 | 0001 | 0xxx | xxxx | Z | |
| COMF | f, d | Complement f | 1 | 00 | 1001 | dfff | ffff | Z | 1,2 |
| DECF | f, d | Decrement f | 1 | 00 | 0011 | dfff | ffff | Z | 1,2 |
| DECFSZ | f, d | Decrement f, Skip if 0 | 1(2) | 00 | 1011 | dfff | ffff | | 1,2,3 |
| INCF | f, d | Increment f | 1 | 00 | 1010 | dfff | ffff | Z | 1,2 |
| INCFSZ | f, d | Increment f, Skip if 0 | 1(2) | 00 | 1111 | dfff | ffff | | 1,2,3 |
| IORWF | f, d | Inclusive OR W with f | 1 | 00 | 0100 | dfff | ffff | Z | 1,2 |
| MOVF | f, d | Move f | 1 | 00 | 1000 | dfff | ffff | Z | 1,2 |
| MOVWF | f | Move W to f | 1 | 00 | 0000 | lfff | ffff | | |
| NOP | - | No Operation | 1 | 00 | 0000 | 0xx0 | 0000 | | |
| RLF | f, d | Rotate Left f through Carry | 1 | 00 | 1101 | dfff | ffff | C | 1,2 |
| RRF | f, d | Rotate Right f through Carry | 1 | 00 | 1100 | dfff | ffff | C | 1,2 |
| SUBWF | f, d | Subtract W from f | 1 | 00 | 0010 | dfff | ffff | C,DC,Z | 1,2 |
| SWAPF | f, d | Swap nibbles in f | 1 | 00 | 1110 | dfff | ffff | | 1,2 |
| XORWF | f, d | Exclusive OR W with f | 1 | 00 | 0110 | dfff | ffff | Z | 1,2 |
| **BIT-ORIENTED FILE REGISTER OPERATIONS** | | | | | | | | | |
| BCF | f, b | Bit Clear f | 1 | 01 | 00bb | bfff | ffff | | 1,2 |
| BSF | f, b | Bit Set f | 1 | 01 | 01bb | bfff | ffff | | 1,2 |
| BTFSC | f, b | Bit Test f, Skip if Clear | 1 (2) | 01 | 10bb | bfff | ffff | | 3 |
| BTFSS | f, b | Bit Test f, Skip if Set | 1 (2) | 01 | 11bb | bfff | ffff | | 3 |
| **LITERAL AND CONTROL OPERATIONS** | | | | | | | | | |
| ADDLW | k | Add literal and W | 1 | 11 | 111x | kkkk | kkkk | C,DC,Z | |
| ANDLW | k | AND literal with W | 1 | 11 | 1001 | kkkk | kkkk | Z | |
| CALL | k | Call subroutine | 2 | 10 | 0kkk | kkkk | kkkk | | |
| CLRWDT | - | Clear Watchdog Timer | 1 | 00 | 0000 | 0110 | 0100 | $\overline{TO}$,$\overline{PD}$ | |
| GOTO | k | Go to address | 2 | 10 | 1kkk | kkkk | kkkk | | |
| IORLW | k | Inclusive OR literal with W | 1 | 11 | 1000 | kkkk | kkkk | Z | |
| MOVLW | k | Move literal to W | 1 | 11 | 00xx | kkkk | kkkk | | |
| RETFIE | - | Return from interrupt | 2 | 00 | 0000 | 0000 | 1001 | | |
| RETLW | k | Return with literal in W | 2 | 11 | 01xx | kkkk | kkkk | | |
| RETURN | - | Return from Subroutine | 2 | 00 | 0000 | 0000 | 1000 | | |
| SLEEP | - | Go into standby mode | 1 | 00 | 0000 | 0110 | 0011 | $\overline{TO}$,$\overline{PD}$ | |
| SUBLW | k | Subtract W from literal | 1 | 11 | 110x | kkkk | kkkk | C,DC,Z | |
| XORLW | k | Exclusive OR literal with W | 1 | 11 | 1010 | kkkk | kkkk | Z | |

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

## 29.2 Instruction Formats

Figure 29-1 shows the three general formats that the instructions can have. As can be seen from the general format of the instructions, the opcode portion of the instruction word varies from 3-bits to 6-bits of information. This is what allows the midrange instruction set to have 35 instructions.

**Note 1:** Any unused opcode is Reserved. Use of any reserved opcode may cause unexpected operation.

**Note 2:** To maintain upward compatibility with future midrange products, do not use the `OPTION` and `TRIS` instructions.

All instruction examples use the following format to represent a hexadecimal number:

0xhh

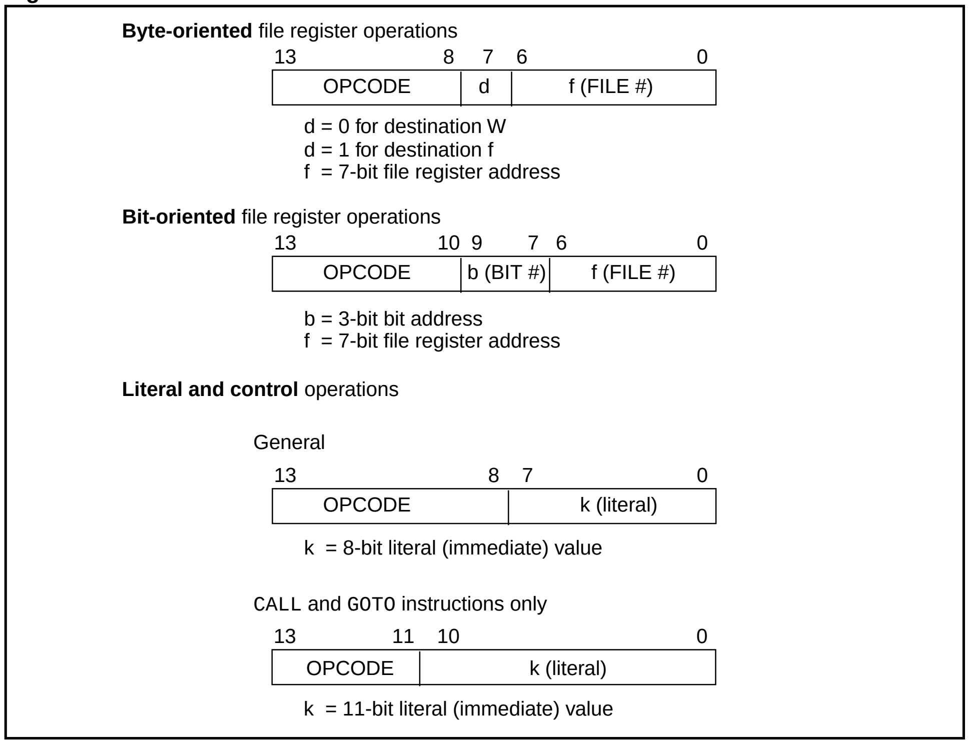
where h signifies a hexadecimal digit.

To represent a binary number:

`00000100b`

where b is a binary string identifier.

**Figure 29-1: General Format for Instructions**



**Byte-oriented** file register operations
13
8
7
6
0
OPCODE
d
f (FILE #)
d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address
**Bit-oriented** file register operations
13
10
9
7
6
0
OPCODE
b (BIT #)
f (FILE #)
b = 3-bit bit address
f = 7-bit file register address
**Literal and control** operations
General
13
8
7
0
OPCODE
k (literal)
k = 8-bit literal (immediate) value
CALL and GOTO instructions only
13
11
10
0
OPCODE
k (literal)
k = 11-bit literal (immediate) value

**Table 29-2: Instruction Description Conventions**

| Field | Description |
|---|---|
| f | Register file address (0x00 to 0x7F) |
| W | Working register (accumulator) |
| b | Bit address within an 8-bit file register (0 to 7) |
| k | Literal field, constant data or label (may be either an 8-bit or an 11-bit value) |
| x | Don't care (0 or 1)<br>The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools. |
| d | Destination select;<br>d = 0: store result in W,<br>d = 1: store result in file register f. |
| dest | Destination either the W register or the specified register file location |
| label | Label name |
| TOS | Top of Stack |
| PC | Program Counter |
| PCLATH | Program Counter High Latch |
| GIE | Global Interrupt Enable bit |
| WDT | Watchdog Timer |
| $\overline{\text{TO}}$ | Time-out bit |
| $\overline{\text{PD}}$ | Power-down bit |
| [ ] | Optional |
| ( ) | Contents |
| $\rightarrow$ | Assigned to |
| < > | Register bit field |
| $\in$ | In the set of |
| *italics* | User defined term (font is courier) |

## 29.3 Special Function Registers as Source/Destination

The Section 29. Instruction Set's orthogonal instruction set allows read and write of all file registers, including special function registers. Some special situations the user should be aware of are explained in the following subsections:

### 29.3.1 STATUS Register as Destination

If an instruction writes to the STATUS register, the Z, C, DC and OV bits may be set or cleared as a result of the instruction and overwrite the original data bits written. For example, executing `CLRF STATUS` will clear register STATUS, and then set the Z bit leaving `0000 0100b` in the register.

### 29.3.2 PCL as Source or Destination

Read, write or read-modify-write on PCL may have the following results:

| | |
|---|---|
| Read PC: | PCL $\rightarrow$ dest; PCLATH does not change; |
| Write PCL: | PCLATH $\rightarrow$ PCH;<br>8-bit destination value $\rightarrow$ PCL |
| Read-Modify-Write: | PCL$\rightarrow$ ALU operand<br>PCLATH $\rightarrow$ PCH;<br>8-bit result $\rightarrow$ PCL |

Where PCH = program counter high byte (not an addressable register), PCLATH = Program counter high holding latch, dest = destination, W register or register file f.

### 29.3.3 Bit Manipulation

All bit manipulation instructions will first read the entire register, operate on the selected bit and then write the result back (read-modify-write (R-M-W)) the specified register. The user should keep this in mind when operating on some special function registers, such as ports.

**Note:** Status bits that are manipulated by the device (including the interrupt flag bits) are set or cleared in the Q1 cycle. So there is no issue with executing R-M-W instructions on registers which contain these bits.

## 29.4 Q Cycle Activity

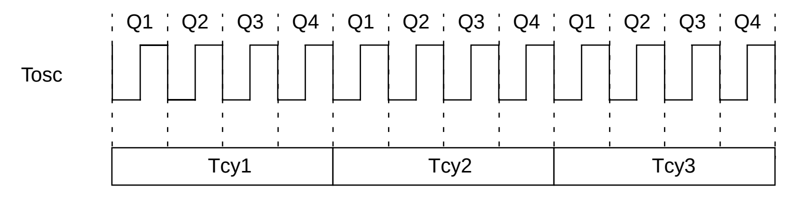
Each instruction cycle (Tcy) is comprised of four Q cycles (Q1-Q4). The Q cycle is the same as the device oscillator cycle (TOSC). The Q cycles provide the timing/designation for the Decode, Read, Process Data, Write etc., of each instruction cycle. The following diagram shows the relationship of the Q cycles to the instruction cycle.

The four Q cycles that make up an instruction cycle (Tcy) can be generalized as:

Q1: Instruction Decode Cycle or forced No Operation
Q2: Instruction Read Cycle or No Operation
Q3: Process the Data
Q4: Instruction Write Cycle or No Operation

Each instruction will show the detailed Q cycle operation for the instruction.

**Figure 29-2: Q Cycle Activity**



## 29.5 Instruction Descriptions

### ADDLW Add Literal and W

| | |
|---|---|
| Syntax: | [ *label* ] ADDLW k |
| Operands: | $0 \le k \le 255$ |
| Operation: | $(W) + k \rightarrow W$ |
| Status Affected: | C, DC, Z |
| Encoding: | `11` \| `111x` \| `kkkk` \| `kkkk` |
| Description: | The contents of the W register are added to the eight bit literal 'k' and the result is placed in the W register. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

Example1 `ADDLW 0x15`

Before Instruction
W = 0x10
After Instruction
W = 0x25

Example 2 `ADDLW MYREG`

Before Instruction
W = 0x10
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x47

Example 3 `ADDLW HIGH (LU_TABLE)`

Before Instruction
W = 0x10
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0xA3

Example 4 `ADDLW MYREG`

Before Instruction
W = 0x10
Address of `PCL` † = 0x02
† `PCL` is the symbol for the Program Counter low byte location
After Instruction
W = 0x12

## ADDWF    Add W and f

| | |
|---|---|
| Syntax: | [ *label* ] ADDWF f,d |
| Operands: | 0 ≤ f ≤ 127<br>d ∈ [0,1] |
| Operation: | (W) + (f) → destination |
| Status Affected: | C, DC, Z |
| Encoding: | `00` \| `0111` \| `dfff` \| `ffff` |
| Description: | Add the contents of the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1    `ADDWF FSR, 0`

Before Instruction
```
W   = 0x17
FSR = 0xC2
```
After Instruction
```
W   = 0xD9
FSR = 0xC2
```

Example 2    `ADDWF INDF, 1`

Before Instruction
```
W   = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x20
```
After Instruction
```
W   = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x37
```

Example 3    `ADDWF PCL`

Case 1: Before Instruction
```
W   = 0x10
PCL = 0x37
C   = x
```
After Instruction
```
PCL = 0x47
C   = 0
```

Case 2: Before Instruction
```
W   = 0x10
PCL = 0xF7
PCH = 0x08
C   = x
```
After Instruction
```
PCL = 0x07
PCH = 0x08
C   = 1
```

## ANDLW And Literal with W

| | |
|---|---|
| Syntax: | [ *label* ] ANDLW k |
| Operands: | 0 ≤ k ≤ 255 |
| Operation: | (W).AND. (k) → W |
| Status Affected: | Z |
| Encoding: | `11` `1001` `kkkk` `kkkk` |
| Description: | The contents of W register are AND'ed with the eight bit literal 'k'. The result is placed in the W register. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

Example 1

```
ANDLW   0x5F
```

Before Instruction
W = 0xA3
After Instruction
W = 0x03

```
; 0101 1111  (0x5F)
; 1010 0011  (0xA3)
;----------  ------
; 0000 0011  (0x03)
```

Example 2

```
ANDLW   MYREG
```

Before Instruction
W = 0xA3
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x23

Example 3

```
ANDLW   HIGH (LU_TABLE)
```

Before Instruction
W = 0xA3
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0x83

## ANDWF    AND W with f

| | |
|---|---|
| Syntax: | [ *label* ] ANDWF    f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | (W).AND. (f) $\rightarrow$ destination |
| Status Affected: | Z |
| Encoding: | `00` \| `0101` \| `dfff` \| `ffff` |
| Description: | AND the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1    `ANDWF   FSR, 1`

```
Before Instruction                    ; 0001 0111  (0x17)
        W   =  0x17                   ; 1100 0010  (0xC2)
        FSR =  0xC2                   ;----------  ------
After Instruction                     ; 0000 0010  (0x02)
        W   =  0x17
        FSR =  0x02
```

Example 2    `ANDWF   FSR, 0`

```
Before Instruction                    ; 0001 0111  (0x17)
        W   =  0x17                   ; 1100 0010  (0xC2)
        FSR =  0xC2                   ;----------  ------
After Instruction                     ; 0000 0010  (0x02)
        W   =  0x02
        FSR =  0xC2
```

Example 3    `ANDWF   INDF, 1`

```
Before Instruction
        W   =  0x17
        FSR =  0xC2
        Contents of Address (FSR) = 0x5A
After Instruction
        W   =  0x17
        FSR =  0xC2
        Contents of Address (FSR) = 0x15
```

## BCF — Bit Clear f

| | |
|---|---|
| Syntax: | [ *label* ] BCF f,b |
| Operands: | $0 \le f \le 127$<br>$0 \le b \le 7$ |
| Operation: | $0 \rightarrow$ f<b> |
| Status Affected: | None |
| Encoding: | `01` \| `00bb` \| `bfff` \| `ffff` |
| Description: | Bit 'b' in register 'f' is cleared. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write register 'f' |

Example 1 `BCF FLAG_REG, 7`

Before Instruction

FLAG_REG = 0xC7 `; **1**100 0111`

After Instruction

FLAG_REG = 0x47 `; **0**100 0111`

Example 2 `BCF INDF, 3`

Before Instruction

W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x2F

After Instruction

W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x27

## BSF **Bit Set f**

| | |
|---|---|
| Syntax: | [ *label* ] BSF f,b |
| Operands: | 0 ≤ f ≤ 127<br>0 ≤ b ≤ 7 |
| Operation: | 1 → f<b> |
| Status Affected: | None |
| Encoding: | `01` `01bb` `bfff` `ffff` |
| Description: | Bit 'b' in register 'f' is set. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write register 'f' |

Example 1 `BSF FLAG_REG, 7`

Before Instruction
FLAG_REG =0x0A ; **0**000 1010

After Instruction
FLAG_REG =0x8A ; **1**000 1010

Example 2 `BSF INDF, 3`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x20

After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x28

## BTFSC **Bit Test, Skip if Clear**

| | |
|---|---|
| Syntax: | [ *label* ] BTFSC f,b |
| Operands: | $0 \le f \le 127$<br>$0 \le b \le 7$ |
| Operation: | skip if (f<b>) = 0 |
| Status Affected: | None |
| Encoding: | `01` `10bb` `bfff` `ffff` |
| Description: | If bit 'b' in register 'f' is '0' then the next instruction is skipped.<br>If bit 'b' is '0' then the next instruction (fetched during the current instruction execution) is discarded, and a NOP is executed instead, making this a 2 cycle instruction. |
| Words: | 1 |
| Cycles: | 1(2) |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | No operation |

If skip (2nd cycle):

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example 1

```
HERE    BTFSC  FLAG, 4
FALSE   GOTO   PROCESS_CODE
TRUE    •
        •
        •
```

Case 1: Before Instruction

PC = address`HERE`
FLAG= `xxx0 xxxx`

After Instruction

Since FLAG<4>= 0,
PC = address`TRUE`

Case 2: Before Instruction

PC = address`HERE`
FLAG= `xxx1 xxxx`

After Instruction

Since FLAG<4>=1,
PC = address`FALSE`

## BTFSS **Bit Test f, Skip if Set**

| | |
|---|---|
| Syntax: | [ *label* ] BTFSS f,b |
| Operands: | $0 \le f \le 127$<br>$0 \le b < 7$ |
| Operation: | skip if (f<b>) = 1 |
| Status Affected: | None |
| Encoding: | `01` \| `11bb` \| `bfff` \| `ffff` |
| Description: | If bit 'b' in register 'f' is '1' then the next instruction is skipped.<br>If bit 'b' is '1', then the next instruction (fetched during the current instruction execution) is discarded and a NOP is executed instead, making this a 2 cycle instruction. |
| Words: | 1 |
| Cycles: | 1(2) |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | No operation |

If skip (2nd cycle):

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example 1

```
HERE    BTFSS  FLAG, 4
FALSE   GOTO   PROCESS_CODE
TRUE    •
        •
        •
```

Case 1: Before Instruction
PC = address`HERE`
FLAG= `xxx0 xxxx`
After Instruction
Since FLAG<4>= 0,
PC = address`FALSE`

Case 2: Before Instruction
PC = address`HERE`
FLAG= `xxx1 xxxx`
After Instruction
Since FLAG<4>=1,
PC = address`TRUE`

## CALL Call Subroutine

| | |
|---|---|
| Syntax: | [ *label* ] CALL k |
| Operands: | $0 \le k \le 2047$ |
| Operation: | (PC)+ 1$\rightarrow$ TOS,<br>k $\rightarrow$ PC<10:0>,<br>(PCLATH<4:3>) $\rightarrow$ PC<12:11> |
| Status Affected: | None |
| Encoding: | `10` `0kkk` `kkkk` `kkkk` |
| Description: | Call Subroutine. First, the 13-bit return address (PC+1) is pushed onto the stack. The eleven bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH<4:3>. CALL is a two cycle instruction. |
| Words: | 1 |
| Cycles: | 2 |

Q Cycle Activity:

1st cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | No operation |

2nd cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example 1

```
HERE    CALL   THERE
```

Before Instruction

PC = AddressHERE

After Instruction

TOS = Address HERE+1

PC = Address THERE

## CLRF — Clear f

| | |
|---|---|
| Syntax: | [ *label* ] CLRF f |
| Operands: | $0 \le f \le 127$ |
| Operation: | $00h \rightarrow f$<br>$1 \rightarrow Z$ |
| Status Affected: | Z |
| Encoding: | `00` \| `0001` \| `1fff` \| `ffff` |
| Description: | The contents of register 'f' are cleared and the Z bit is set. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write register 'f' |

Example 1 `CLRF    FLAG_REG`

Before Instruction
FLAG_REG=0x5A
After Instruction
FLAG_REG=0x00
Z = 1

Example 2 `CLRF    INDF`

Before Instruction
FSR = 0xC2
Contents of Address (FSR)=0xAA
After Instruction
FSR = 0xC2
Contents of Address (FSR)=0x00
Z = 1

## CLRW Clear W

| | |
|---|---|
| Syntax: | [ *label* ] CLRW |
| Operands: | None |
| Operation: | 00h $\rightarrow$ W<br>1 $\rightarrow$ Z |
| Status Affected: | Z |
| Encoding: | `00` \| `0001` \| `0xxx` \| `xxxx` |
| Description: | W register is cleared. Zero bit (Z) is set. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write register 'W' |

Example 1 `CLRW`

Before Instruction
```
W = 0x5A
```
After Instruction
```
W = 0x00
Z = 1
```

## CLRWDT Clear Watchdog Timer

| | |
|---|---|
| Syntax: | [ *label* ] CLRWDT |
| Operands: | None |
| Operation: | 00h → WDT<br>0 → WDT prescaler count,<br>1 → $\overline{\text{TO}}$<br>1 → $\overline{\text{PD}}$ |
| Status Affected: | $\overline{\text{TO}}$, $\overline{\text{PD}}$ |
| Encoding: | `00` `0000` `0110` `0100` |
| Description: | `CLRWDT` instruction clears the Watchdog Timer. It also clears the prescaler count of the WDT. Status bits $\overline{\text{TO}}$ and $\overline{\text{PD}}$ are set. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | No operation | Process data | Clear WDT Counter |

Example 1 `CLRWDT`

Before Instruction
WDT counter= x
WDT prescaler =1:128

After Instruction
WDT counter=0x00
WDT prescaler count=0
$\overline{\text{TO}}$ = 1
$\overline{\text{PD}}$ = 1
WDT prescaler =1:128

**Note:** The `CLRWDT` instruction does not affect the assignment of the WDT prescaler.

## COMF — Complement f

| | |
|---|---|
| Syntax: | [ *label* ] COMF f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | $(\bar{f}) \rightarrow$ destination |
| Status Affected: | Z |
| Encoding: | `00` `1001` `dfff` `ffff` |
| Description: | The contents of register 'f' are 1's complemented. If 'd' is 0 the result is stored in W. If 'd' is 1 the result is stored back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1 `COMF REG1, 0`

Before Instruction
REG1= 0x13
After Instruction
REG1= 0x13
W = 0xEC

Example 2 `COMF INDF, 1`

Before Instruction
FSR = 0xC2
Contents of Address (FSR)=0xAA
After Instruction
FSR = 0xC2
Contents of Address (FSR)=0x55

Example 3 `COMF REG1, 1`

Before Instruction
REG1= 0xFF
After Instruction
REG1= 0x00
Z = 1

## DECF — Decrement f

| | |
|---|---|
| Syntax: | [ *label* ] DECF f,d |
| Operands: | 0 ≤ f ≤ 127<br>d ∈ [0,1] |
| Operation: | (f) - 1 → destination |
| Status Affected: | Z |
| Encoding: | `00` \| `0011` \| `dfff` \| `ffff` |
| Description: | Decrement register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1

```
DECF    CNT, 1
```

Before Instruction
```
CNT = 0x01
Z   = 0
```
After Instruction
```
CNT = 0x00
Z   = 1
```

Example 2

```
DECF    INDF, 1
```

Before Instruction
```
FSR = 0xC2
Contents of Address (FSR) = 0x01
Z   = 0
```
After Instruction
```
FSR = 0xC2
Contents of Address (FSR) = 0x00
Z   = 1
```

Example 3

```
DECF    CNT, 0
```

Before Instruction
```
CNT = 0x10
W   = x
Z   = 0
```
After Instruction
```
CNT = 0x10
W   = 0x0F
Z   = 0
```

## DECFSZ **Decrement f, Skip if 0**

| | |
|---|---|
| Syntax: | [ *label* ] DECFSZ f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | (f) - 1 $\rightarrow$ destination; skip if result = 0 |
| Status Affected: | None |
| Encoding: | `00` \| `1011` \| `dfff` \| `ffff` |
| Description: | The contents of register 'f' are decremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.<br>If the result is 0, then the next instruction (fetched during the current instruction execution) is discarded and a NOP is executed instead, making this a 2 cycle instruction. |
| Words: | 1 |
| Cycles: | 1(2) |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

If skip (2nd cycle):

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example

```
HERE      DECFSZ   CNT, 1
          GOTO     LOOP
CONTINUE  •
          •
          •
```

Case 1: Before Instruction

PC = address HERE
CNT = 0x01

After Instruction

CNT = 0x00
PC = address CONTINUE

Case 2: Before Instruction

PC = address HERE
CNT = 0x02

After Instruction

CNT = 0x01
PC = address HERE + 1

## GOTO — Unconditional Branch

| | |
|---|---|
| Syntax: | [ *label* ] GOTO k |
| Operands: | 0 ≤ k ≤ 2047 |
| Operation: | k → PC<10:0><br>PCLATH<4:3> → PC<12:11> |
| Status Affected: | None |
| Encoding: | `10` \| `1kkk` \| `kkkk` \| `kkkk` |
| Description: | `GOTO` is an unconditional branch. The eleven bit immediate value is loaded into PC bits <10:0>. The upper bits of PC are loaded from PCLATH<4:3>. `GOTO` is a two cycle instruction. |
| Words: | 1 |
| Cycles: | 2 |

Q Cycle Activity:

1st cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k'<7:0> | Process data | No operation |

2nd cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example

```
GOTO THERE
```

After Instruction

PC =Address`THERE`

## INCF Increment f

| | |
|---|---|
| Syntax: | [ *label* ] INCF f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | $(f) + 1 \rightarrow$ destination |
| Status Affected: | Z |
| Encoding: | `00` `1010` `dfff` `ffff` |
| Description: | The contents of register 'f' are incremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1 `INCF CNT, 1`

Before Instruction
```
CNT = 0xFF
Z   = 0
```
After Instruction
```
CNT = 0x00
Z   = 1
```

Example 2 `INCF INDF, 1`

Before Instruction
```
FSR = 0xC2
Contents of Address (FSR) = 0xFF
Z   = 0
```
After Instruction
```
FSR = 0xC2
Contents of Address (FSR) = 0x00
Z   = 1
```

Example 3 `INCF CNT, 0`

Before Instruction
```
CNT = 0x10
W   = x
Z   = 0
```
After Instruction
```
CNT = 0x10
W   = 0x11
Z   = 0
```

## INCFSZ **Increment f, Skip if 0**

| | |
|---|---|
| Syntax: | [ *label* ] INCFSZ f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | (f) + 1 $\rightarrow$ destination, skip if result = 0 |
| Status Affected: | None |
| Encoding: | `00` `1111` `dfff` `ffff` |
| Description: | The contents of register 'f' are incremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.<br>If the result is 0, then the next instruction (fetched during the current instruction execution) is discarded and a NOP is executed instead, making this a 2 cycle instruction. |
| Words: | 1 |
| Cycles: | 1(2) |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

If skip (2nd cycle):

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example

```
HERE      INCFSZ   CNT, 1
          GOTO     LOOP
CONTINUE  •
          •
          •
```

Case 1: Before Instruction

| | | |
|---|---|---|
| PC | = | address HERE |
| CNT | = | 0xFF |

After Instruction

| | | |
|---|---|---|
| CNT | = | 0x00 |
| PC | = | address CONTINUE |

Case 2: Before Instruction

| | | |
|---|---|---|
| PC | = | address HERE |
| CNT | = | 0x00 |

After Instruction

| | | |
|---|---|---|
| CNT | = | 0x01 |
| PC | = | address HERE + 1 |

## IORLW — Inclusive OR Literal with W

| | |
|---|---|
| Syntax: | [ *label* ] IORLW k |
| Operands: | $0 \leq k \leq 255$ |
| Operation: | (W).OR. k $\rightarrow$ W |
| Status Affected: | Z |
| Encoding: | `11` \| `1000` \| `kkkk` \| `kkkk` |
| Description: | The contents of the W register is OR'ed with the eight bit literal 'k'. The result is placed in the W register. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

Example 1 `IORLW 0x35`

Before Instruction
W = 0x9A
After Instruction
W = 0xBF
Z = 0

Example 2 `IORLW MYREG`

Before Instruction
W = 0x9A
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x9F
Z = 0

Example 3 `IORLW HIGH (LU_TABLE)`

Before Instruction
W = 0x9A
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0x9B
Z = 0

Example 4 `IORLW 0x00`

Before Instruction
W = 0x00
After Instruction
W = 0x00
Z = 1

## IORWF Inclusive OR W with f

| | |
|---|---|
| Syntax: | [ *label* ] IORWF f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | (W).OR. (f) $\rightarrow$ destination |
| Status Affected: | $\overline{Z}$ |
| Encoding: | `00` `0100` `dfff` `ffff` |
| Description: | Inclusive OR the W register with register 'f'. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1 `IORWF RESULT, 0`

Before Instruction
```
RESULT=0x13
W    =    0x91
```
After Instruction
```
RESULT=0x13
W    =    0x93
Z    =    0
```

Example 2 `IORWF INDF, 1`

Before Instruction
```
W    =    0x17
FSR =     0xC2
Contents of Address (FSR) = 0x30
```
After Instruction
```
W    =    0x17
FSR =     0xC2
Contents of Address (FSR) = 0x37
Z    =    0
```

Example 3 `IORWF RESULT, 1`

Case 1: Before Instruction
```
RESULT=0x13
W    =    0x91
```
After Instruction
```
RESULT=0x93
W    =    0x91
Z    =    0
```

Case 2: Before Instruction
```
RESULT=0x00
W    =    0x00
```
After Instruction
```
RESULT=0x00
W    =    0x00
Z    =    1
```

## MOVLW Move Literal to W

| | |
|---|---|
| Syntax: | [ *label* ] MOVLW k |
| Operands: | $0 \le k \le 255$ |
| Operation: | $k \rightarrow W$ |
| Status Affected: | None |

Encoding:

| | | | |
|---|---|---|---|
| `11` | `00xx` | `kkkk` | `kkkk` |

| | |
|---|---|
| Description: | The eight bit literal 'k' is loaded into W register. The don't cares will assemble as 0's. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

Example 1 `MOVLW 0x5A`

After Instruction
W = 0x5A

Example 2 `MOVLW MYREG`

Before Instruction
W = 0x10
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x37

Example 3 `MOVLW HIGH (LU_TABLE)`

Before Instruction
W = 0x10
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0x93

## MOVF Move f

| | |
|---|---|
| Syntax: | [ *label* ] MOVF f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | (f) $\rightarrow$ destination |
| Status Affected: | Z |
| Encoding: | `00` \| `1000` \| `dfff` \| `ffff` |
| Description: | The contents of register 'f' is moved to a destination dependent upon the status of 'd'. If 'd' = 0, destination is W register. If 'd' = 1, the destination is file register 'f' itself. 'd' = 1 is useful to test a file register since status flag Z is affected. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1 `MOVF FSR, 0`

Before Instruction
W = 0x00
FSR = 0xC2
After Instruction
W = 0xC2
Z = 0

Example 2 `MOVF INDF, 0`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x00
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x00
Z = 1

Example 3 `MOVF FSR, 1`

Case 1: Before Instruction
FSR = 0x43
After Instruction
FSR = 0x43
Z = 0

Case 2: Before Instruction
FSR = 0x00
After Instruction
FSR = 0x00
Z = 1

## MOVWF Move W to f

| | |
|---|---|
| Syntax: | [ *label* ] MOVWF f |
| Operands: | 0 ≤ f ≤ 127 |
| Operation: | (W) → f |
| Status Affected: | None |
| Encoding: | `00` `0000` `1fff` `ffff` |
| Description: | Move data from W register to register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write register 'f' |

Example 1 `MOVWF OPTION_REG`

Before Instruction
OPTION_REG=0xFF
W = 0x4F
After Instruction
OPTION_REG=0x4F
W = 0x4F

Example 2 `MOVWF INDF`

Before Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x00
After Instruction
W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x17

## NOP — No Operation

| | |
|---|---|
| Syntax: | [ *label* ] NOP |
| Operands: | None |
| Operation: | No operation |
| Status Affected: | None |
| Encoding: | `00` `0000` `0xx0` `0000` |
| Description: | No operation. |
| Words: | 1 |
| Cycles: | 1 |
| Q Cycle Activity: | |

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | No operation | No operation | No operation |

Example: `HERE NOP`

Before Instruction
PC = address `HERE`
After Instruction
PC = address `HERE + 1`

## OPTION — Load Option Register

| | |
|---|---|
| Syntax: | [ *label* ] OPTION |
| Operands: | None |
| Operation: | (W) $\rightarrow$ OPTION |
| Status Affected: | None |
| Encoding: | `00` `0000` `0110` `0010` |
| Description: | The contents of the W register are loaded in the OPTION register. This instruction is supported for code compatibility with PIC16C5X products. Since OPTION is a readable/writable register, the user can directly address it. |
| Words: | 1 |
| Cycles: | 1 |

**To maintain upward compatibility with future PIC16CXX products, do not use this instruction.**

## RETFIE — Return from Interrupt

|  |  |
|---|---|
| Syntax: | [ *label* ] RETFIE |
| Operands: | None |
| Operation: | TOS $\rightarrow$ PC,<br>1 $\rightarrow$ GIE |
| Status Affected: | None |
| Encoding: | `00` `0000` `0000` `1001` |
| Description: | Return from Interrupt. The 13-bit address at the Top of Stack (TOS) is loaded in the PC. The Global Interrupt Enable bit, GIE (INTCON<7>), is automatically set, enabling Interrupts. This is a two cycle instruction. |
| Words: | 1 |
| Cycles: | 2 |

Q Cycle Activity:

1st cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | No operation | Process data | No operation |

2nd cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example `RETFIE`

After Instruction

```
PC  =  TOS
GIE =  1
```

## RETLW **Return with Literal in W**

| | |
|---|---|
| Syntax: | [ *label* ] RETLW k |
| Operands: | $0 \le k \le 255$ |
| Operation: | $k \rightarrow W$; TOS $\rightarrow$ PC |
| Status Affected: | None |
| Encoding: | `11` \| `01xx` \| `kkkk` \| `kkkk` |
| Description: | The W register is loaded with the eight bit literal 'k'. The program counter is loaded 13-bit address at the Top of Stack (the return address). This is a two cycle instruction. |
| Words: | 1 |
| Cycles: | 2 |

Q Cycle Activity:

1st cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

2nd cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

Example

```
HERE    CALL TABLE   ; W contains table
                     ; offset value
        •            ; W now has table value
        •
        •
TABLE   ADDWF PC     ;W = offset
        RETLW k1     ;Begin table
        RETLW k2     ;
        •
        •
        •
        RETLW kn     ; End of table
```

Before Instruction

W = 0x07

After Instruction

W = value of k8

PC = TOS = Address `Here + 1`

## RETURN **Return from Subroutine**

| | |
|---|---|
| Syntax: | [ *label* ] RETURN |
| Operands: | None |
| Operation: | TOS → PC |
| Status Affected: | None |
| Encoding: | `00` `0000` `0000` `1000` |
| Description: | Return from subroutine. The stack is POPed and the top of the stack (TOS) is loaded into the program counter. This is a two cycle instruction. |
| Words: | 1 |
| Cycles: | 2 |

Q Cycle Activity:

1st cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | No operation | Process data | No operation |

2nd cycle:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| No operation | No operation | No operation | No operation |

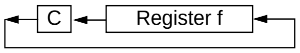
Example
```
HERE     RETURN
```

After Instruction

PC = TOS

## RLF — Rotate Left f through Carry

| | |
|---|---|
| Syntax: | [ *label* ] RLF f,d |
| Operands: | 0 ≤ f ≤ 127<br>d ∈ [0,1] |
| Operation: | See description below |
| Status Affected: | C |
| Encoding: | `00` \| `1101` \| `dfff` \| `ffff` |
| Description: | The contents of register 'f' are rotated one bit to the left through the Carry Flag. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is stored back in register 'f'. |



| | |
|---|---|
| Words: | 1 |
| Cycles: | 1 |
| Q Cycle Activity: | |

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1 `RLF REG1,0`

Before Instruction
```
REG1=  1110 0110
C   =   0
```
After Instruction
```
REG1=1110 0110
W   =1100 1100
C   =1
```

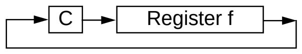
Example 2 `RLF INDF, 1`

Case 1: Before Instruction
```
W   =  xxxx xxxx
FSR =  0xC2
Contents of Address (FSR) = 0011 1010
C   =  1
```
After Instruction
```
W   =  0x17
FSR =  0xC2
Contents of Address (FSR) = 0111 0101
C   =  0
```

Case 2: Before Instruction
```
W   =  xxxx xxxx
FSR =  0xC2
Contents of Address (FSR) = 1011 1001
C   =  0
```
After Instruction
```
W   =  0x17
FSR =  0xC2
Contents of Address (FSR) = 0111 0010
C   =  1
```

## RRF — Rotate Right f through Carry

| | |
|---|---|
| Syntax: | [ *label* ] RRF f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | See description below |
| Status Affected: | C |
| Encoding: | `00` `1100` `dfff` `ffff` |
| Description: | The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'. |



| | |
|---|---|
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1 `RRF REG1,0`

Before Instruction
```
REG1= 1110 0110
W   = xxxx xxxx
C   = 0
```
After Instruction
```
REG1= 1110 0110
W   = 0111 0011
C   = 0
```

Example 2 `RRF INDF, 1`

Case 1: Before Instruction
```
W   = xxxx xxxx
FSR = 0xC2
Contents of Address (FSR) = 0011 1010
C   = 1
```
After Instruction
```
W   = 0x17
FSR = 0xC2
Contents of Address (FSR) = 1001 1101
C   = 0
```

Case 2: Before Instruction
```
W   = xxxx xxxx
FSR = 0xC2
Contents of Address (FSR) = 0011 1001
C   = 0
```
After Instruction
```
W   = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0001 1100
C   = 1
```

## SLEEP

| | |
|---|---|
| Syntax: | [ *label* ] SLEEP |
| Operands: | None |
| Operation: | 00h $\rightarrow$ WDT,<br>0 $\rightarrow$ WDT prescaler count,<br>1 $\rightarrow$ $\overline{\text{TO}}$,<br>0 $\rightarrow$ $\overline{\text{PD}}$ |
| Status Affected: | $\overline{\text{TO}}$, $\overline{\text{PD}}$ |
| Encoding: | `00` `0000` `0110` `0011` |
| Description: | The power-down status bit, $\overline{\text{PD}}$ is cleared. Time-out status bit, $\overline{\text{TO}}$ is set. Watchdog Timer and its prescaler count are cleared.<br>The processor is put into SLEEP mode with the oscillator stopped. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | No operation | No operation | Go to sleep |

Example: SLEEP

**Note:** The SLEEP instruction does not affect the assignment of the WDT prescaler

## SUBLW Subtract W from Literal

Syntax: [ *label* ] SUBLW k

Operands: 0 ≤ k ≤ 255

Operation: k - (W) → W

Status Affected: C, DC, Z

Encoding:

| | | | |
|---|---|---|---|
| 11 | 110x | kkkk | kkkk |

Description: The W register is subtracted (2’s complement method) from the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

Example 1: `SUBLW 0x02`

Case 1: Before Instruction

```
W = 0x01
C = x
Z = x
```

After Instruction

```
W = 0x01
C = 1          ; result is positive
Z = 0
```

Case 2: Before Instruction

```
W = 0x02
C = x
Z = x
```

After Instruction

```
W = 0x00
C = 1          ; result is zero
Z = 1
```

Case 3: Before Instruction

```
W = 0x03
C = x
Z = x
```

After Instruction

```
W = 0xFF
C = 0          ; result is negative
Z = 0
```

Example 2 `SUBLW MYREG`

Before Instruction

W = 0x10

Address of `MYREG` † = 0x37

† `MYREG` is a symbol for a data memory location

After Instruction

```
W = 0x27
C = 1     ; result is positive
```

## SUBWF — Subtract W from f

| | |
|---|---|
| Syntax: | [ *label* ] SUBWF f,d |
| Operands: | 0 ≤ f ≤ 127<br>d ∈ [0,1] |
| Operation: | (f) - (W) → destination |
| Status Affected: | C, DC, Z |
| Encoding: | `00` `0010` `dfff` `ffff` |
| Description: | Subtract (2’s complement method) W register from register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1: `SUBWF REG1,1`

Case 1: Before Instruction

```
REG1= 3
W   = 2
C   = x
Z   = x
```

After Instruction

```
REG1= 1
W   = 2
C   = 1            ; result is positive
Z   = 0
```

Case 2: Before Instruction

```
REG1= 2
W   = 2
C   = x
Z   = x
```

After Instruction

```
REG1= 0
W   = 2
C   = 1            ; result is zero
Z   = 1
```

Case 3: Before Instruction

```
REG1= 1
W   = 2
C   = x
Z   = x
```

After Instruction

```
REG1= 0xFF
W   = 2
C   = 0            ; result is negative
Z   = 0
```

## SWAPF Swap Nibbles in f

| | |
|---|---|
| Syntax: | [ *label* ] SWAPF f,d |
| Operands: | $0 \le f \le 127$<br>$d \in [0,1]$ |
| Operation: | (f<3:0>) $\rightarrow$ destination<7:4>,<br>(f<7:4>) $\rightarrow$ destination<3:0> |
| Status Affected: | None |
| Encoding: | `00` `1110` `dfff` `ffff` |
| Description: | The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0 the result is placed in W register. If 'd' is 1 the result is placed in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1

```
SWAPF   REG, 0
```

Before Instruction

REG1= 0xA5

After Instruction

REG1= 0xA5
W = 0x5A

Example 2

```
SWAPF   INDF, 1
```

Before Instruction

W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x20

After Instruction

W = 0x17
FSR = 0xC2
Contents of Address (FSR) = 0x02

Example 3

```
SWAPF   REG, 1
```

Before Instruction

REG1= 0xA5

After Instruction

REG1= 0x5A

## TRIS — Load TRIS Register

| | |
|---|---|
| Syntax: | [ *label* ] TRIS f |
| Operands: | $5 \le f \le 7$ |
| Operation: | (W) $\rightarrow$ TRIS register f; |
| Status Affected: | None |
| Encoding: | `00` `0000` `0110` `0fff` |
| Description: | The instruction is supported for code compatibility with the PIC16C5X products. Since TRIS registers are readable and writable, the user can directly address them. |
| Words: | 1 |
| Cycles: | 1 |

Example

**To maintain upward compatibility with future PIC16CXX products, do not use this instruction.**

## XORLW — Exclusive OR Literal with W

| | |
|---|---|
| Syntax: | [ *label*] XORLW k |
| Operands: | $0 \le k \le 255$ |
| Operation: | (W).XOR. k $\rightarrow$ W |
| Status Affected: | Z |
| Encoding: | `11` \| `1010` \| `kkkk` \| `kkkk` |
| Description: | The contents of the W register are XOR'ed with the eight bit literal 'k'. The result is placed in the W register. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read literal 'k' | Process data | Write to W register |

Example 1

```
XORLW   0xAF              ; 1010 1111   (0xAF)
Before Instruction        ; 1011 0101   (0xB5)
        W  =  0xB5        ; ---------   ------
After Instruction         ; 0001 1010   (0x1A)
        W  =  0x1A
        Z  =  0
```

Example 2

```
XORLW   MYREG
```

Before Instruction
W = 0xAF
Address of `MYREG` † = 0x37
† `MYREG` is a symbol for a data memory location
After Instruction
W = 0x18
Z = 0

Example 3

```
XORLW   HIGH (LU_TABLE)
```

Before Instruction
W = 0xAF
Address of `LU_TABLE` † = 0x9375
† `LU_TABLE` is a label for an address in program memory
After Instruction
W = 0x3C
Z = 0

## XORWF — Exclusive OR W with f

| | |
|---|---|
| Syntax: | [ *label* ] XORWF f,d |
| Operands: | $0 \leq f \leq 127$<br>$d \in [0,1]$ |
| Operation: | (W).XOR. (f) $\rightarrow$ destination |
| Status Affected: | Z |
| Encoding: | `00` \| `0110` \| `dfff` \| `ffff` |
| Description: | Exclusive OR the contents of the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'. |
| Words: | 1 |
| Cycles: | 1 |

Q Cycle Activity:

| Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|
| Decode | Read register 'f' | Process data | Write to destination |

Example 1

```
XORWF    REG, 1          ; 1010 1111   (0xAF)
                         ; 1011 0101   (0xB5)
                         ; ---------   ------
                         ; 0001 1010   (0x1A)
```

Before Instruction
REG= 0xAF
W = 0xB5

After Instruction
REG= 0x1A
W = 0xB5

Example 2

```
XORWF    REG, 0          ; 1010 1111   (0xAF)
                         ; 1011 0101   (0xB5)
                         ; ---------   ------
                         ; 0001 1010   (0x1A)
```

Before Instruction
REG= 0xAF
W = 0xB5

After Instruction
REG= 0xAF
W = 0x1A

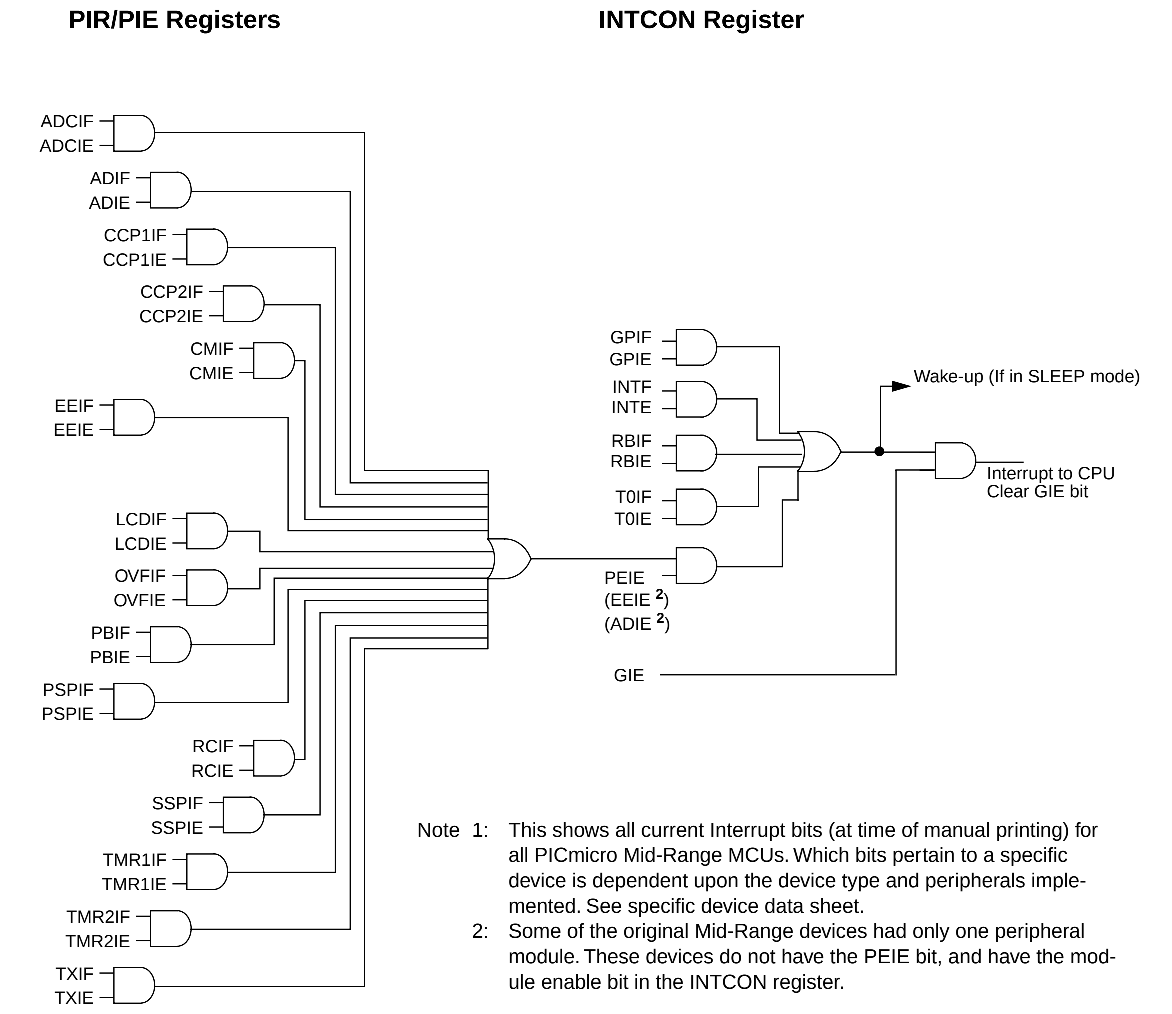
Example 3

```
 XORWF   INDF, 1
```

Before Instruction
W = 0xB5
FSR = 0xC2
Contents of Address (FSR) = 0xAF

After Instruction
W = 0xB5
FSR = 0xC2
Contents of Address (FSR) = 0x1A

**Figure 8-1: Interrupt Logic**



Note 1: This shows all current Interrupt bits (at time of manual printing) for all PICmicro Mid-Range MCUs. Which bits pertain to a specific device is dependent upon the device type and peripherals implemented. See specific device data sheet.

2: Some of the original Mid-Range devices had only one peripheral module. These devices do not have the PEIE bit, and have the module enable bit in the INTCON register.

## 8.2 Control Registers

Generally devices have a minimum of three registers associated with interrupts. The INTCON register which contains Global Interrupt Enable bit, GIE, as well as the Peripheral Interrupt Enable bit, PEIE, and the PIE / PIR register pair which enable the peripheral interrupts and display the interrupt flag status.

### 8.2.1 INTCON Register

The INTCON Register is a readable and writable register which contains various enable and flag bits.

| **Note:** | Interrupt flag bits get set when an interrupt condition occurs regardless of the state of its corresponding enable bit or the global enable bit, GIE (INTCON<7>).This feature allows for software polling. |
|---|---|

**Register 8-1: INTCON Register**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| GIE | PEIE (3) | T0IE | INTE (2) | RBIE (1, 2) | T0IF | INTF (2) | RBIF (1, 2) |
| bit 7 | | | | | | | bit 0 |

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

bit 6 **PEIE:** Peripheral Interrupt Enable bit
1 = Enables all un-masked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 **INTE:** INT External Interrupt Enable bit
1 = Enables the INT external interrupt
0 = Disables the INT external interrupt

bit 3 **RBIE (1):** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF:** INT External Interrupt Flag bit
1 = The INT external interrupt occurred (must be cleared in software)
0 = The INT external interrupt did not occur

bit 0 **RBIF (1):** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as ‘0’ | - n = Value at POR reset |

| | |
|---|---|
| **Note 1:** | In some devices, the RBIE bit may also be known as GPIE and the RBIF bit may be know as GPIF. |
| **Note 2:** | Some devices may not have this feature. For those devices this bit is reserved. |
| **Note 3:** | In devices with only one peripheral interrupt, this bit may be EEIE or ADIE. |

### 8.2.2 PIE Register(s)

Depending on the number of peripheral interrupt sources, there may be multiple Peripheral Interrupt Enable registers (PIE1, PIE2). These registers contain the individual enable bits for the Peripheral interrupts. These registers will be generically referred to as PIE. If the device has a PIE register, The PEIE bit must be set to enable any of these peripheral interrupts.

**Note:** Bit PEIE (INTCON<6>) must be set to enable any of the peripheral interrupts.

Although, the PIE register bits have a general bit location with each register, future devices may not have consistent placement. Bit location inconsistencies will not be a problem if you use the supplied Microchip Include files for the symbolic use of these bits. This will allow the Assembler/Compiler to automatically take care of the placement of these bits by specifying the correct register and bit name.

**Register 8-2: PIE Register**

| R/W-0 |
|---|
| (Note 1) |

bit 7 bit 0

bit **TMR1IE**: TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

bit **TMR2IE**: TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

bit **CCP1IE**: CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit **CCP2IE**: CCP2 Interrupt Enable bit
1 = Enables the CCP2 interrupt
0 = Disables the CCP2 interrupt

bit **SSPIE**: Synchronous Serial Port Interrupt Enable bit
1 = Enables the SSP interrupt
0 = Disables the SSP interrupt

bit **RCIE**: USART Receive Interrupt Enable bit
1 = Enables the USART receive interrupt
0 = Disables the USART receive interrupt

bit **TXIE**: USART Transmit Interrupt Enable bit
1 = Enables the USART transmit interrupt
0 = Disables the USART transmit interrupt

bit **ADIE**: A/D Converter Interrupt Enable bit
1 = Enables the A/D interrupt
0 = Disables the A/D interrupt

bit **ADCIE**: Slope A/D Converter comparator Trip Interrupt Enable bit
1 = Enables the Slope A/D interrupt
0 = Disables the Slope A/D interrupt

bit **OVFIE**: Slope A/D TMR Overflow Interrupt Enable bit
1 = Enables the Slope A/D TMR overflow interrupt
0 = Disables the Slope A/D TMR overflow interrupt

bit **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt

bit **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE write complete interrupt
0 = Disables the EE write complete interrupt

bit **LCDIE**: LCD Interrupt Enable bit
1 = Enables the LCD interrupt
0 = Disables the LCD interrupt

bit **CMIE**: Comparator Interrupt Enable bit
1 = Enables the Comparator interrupt
0 = Disables the Comparator interrupt

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as ‘0’ | - n = Value at POR reset |

**Note 1:** The bit position of the enable bits is device dependent. Please refer to the device data sheet for bit placement.

### 8.2.3 PIR Register(s)

Depending on the number of peripheral interrupt sources, there may be multiple Peripheral Interrupt Flag registers (PIR1, PIR2). These registers contain the individual flag bits for the peripheral interrupts. These registers will be generically referred to as PIR.

**Note 1:** Interrupt flag bits get set when an interrupt condition occurs regardless of the state of its corresponding enable bit or the global enable bit, GIE (INTCON<7>).

**Note 2:** User software should ensure the appropriate interrupt flag bits are cleared (by software) prior to enabling an interrupt, and after servicing that interrupt.

Although, the PIR bits have a general bit location within each register, future devices may not be able to be consistent with that. It is recommended that you use the supplied Microchip Include files for the symbolic use of these bits. This will allow the Assembler/Compiler to automatically take care of the placement of these bits within the specified register.

**Register 8-3: PIR Register**

| R/W-0 |
|---|
| (Note 1) |

bit 7 bit 0

bit **TMR1IF**: TMR1 Overflow Interrupt Flag bit
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

bit **TMR2IF**: TMR2 to PR2 Match Interrupt Flag bit
1 = TMR2 to PR2 match occurred (must be cleared in software)
0 = No TMR2 to PR2 match occurred

bit **CCP1IF**: CCP1 Interrupt Flag bit

Capture Mode
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred

Compare Mode
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred

PWM Mode
Unused in this mode

bit **CCP2IF**: CCP2 Interrupt Flag bit

Capture Mode
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred

Compare Mode
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred

PWM Mode
Unused in this mode

bit **SSPIF**: Synchronous Serial Port Interrupt Flag bit
1 = The transmission/reception is complete
0 = Waiting to transmit/receive

bit **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer, RCREG, is full (cleared when RCREG is read)
0 = The USART receive buffer is empty

bit **TXIF**: USART Transmit Interrupt Flag bit
1 = The USART transmit buffer, TXREG, is empty (cleared when TXREG is written)
0 = The USART transmit buffer is full

bit **ADIF**: A/D Converter Interrupt Flag bit
1 = An A/D conversion completed (must be cleared in software)
0 = The A/D conversion is not complete

**Register 8-3: PIR Register (Cont'd)**

| | |
|---|---|
| bit | **ADCIF**: Slope A/D Converter Comparator Trip Interrupt Flag bit<br>1 = An A/D conversion completed (must be cleared in software)<br>0 = The A/D conversion is not complete |
| bit | **OVFIF**: Slope A/D TMR Overflow Interrupt Flag bit<br>1 = Slope A/D TMR overflowed (must be cleared in software)<br>0 = Slope A/D TMR did not overflow |
| bit | **PSPIF:** Parallel Slave Port Read/Write Interrupt Flag bit<br>1 = A read or a write operation has taken place (must be cleared in software)<br>0 = No read or write has occurred |
| bit | **EEIF**: EE Write Complete Interrupt Flag bit<br>1 = The data EEPROM write operation is complete (must be cleared in software)<br>0 = The data EEPROM write operation is not complete |
| bit | **LCDIF**: LCD Interrupt Flag bit<br>1 = LCD interrupt has occurred (must be cleared in software)<br>0 = LCD interrupt has not occurred |
| bit | **CMIF**: Comparator Interrupt Flag bit<br>1 = Comparator input has changed (must be cleared in software)<br>0 = Comparator input has not changed |

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as '0' | - n = Value at POR reset |

**Note 1:** The bit position of the flag bits is device dependent. Please refer to the device data sheet for bit placement.

## 8.3 Interrupt Latency

Interrupt latency is defined as the time from the interrupt event (the interrupt flag bit gets set) to the time that the instruction at address 0004h starts execution (when that interrupt is enabled).

For synchronous interrupts (typically internal), the latency is 3TCY.

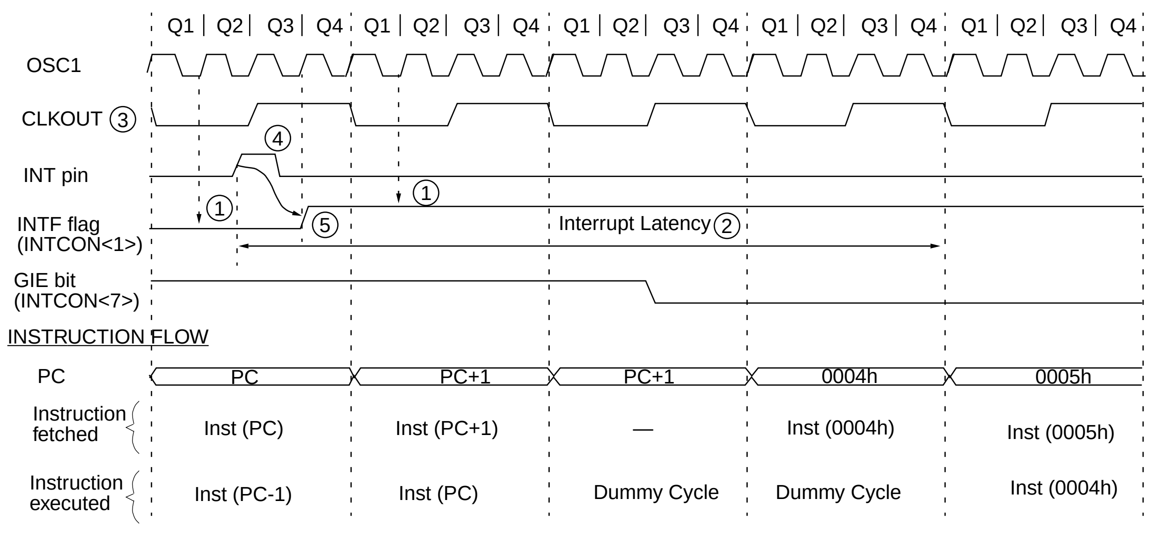
For asynchronous interrupts (typically external), such as the INT or Port RB Change Interrupt, the interrupt latency will be 3 - 3.75TCY (instruction cycles). The exact latency depends upon when the interrupt event occurs (Figure 8-2) in relation to the instruction cycle.

The latency is the same for both one and two cycle instructions.

## 8.4 INT and External Interrupts

The external interrupt on the INT pin is edge triggered: either rising if the INTEDG bit (OPTION<6>) is set, or falling, if the INTEDG bit is clear. When a valid edge appears on the INT pin, the INTF flag bit (INTCON<1>) is set. This interrupt can be enabled/disabled by setting/clearing the INTE enable bit (INTCON<4>). The INTF bit must be cleared in software in the interrupt service routine before re-enabling this interrupt. The INT interrupt can wake-up the processor from SLEEP, if the INTE bit was set prior to going into SLEEP. The status of the GIE bit decides whether or not the processor branches to the interrupt vector following wake-up. See the **"Watchdog Timer and Sleep Mode"** section for details on SLEEP and for timing of wake-up from SLEEP through INT interrupt.

**Figure 8-2: INT Pin and Other External Interrupt Timing**



Note 1: INTF flag is sampled here (every Q1).
2: Interrupt latency = 3-4 TCY where TCY = instruction cycle time. Latency is the same whether Instruction (PC) is a single cycle or a 2-cycle instruction.
3: CLKOUT is available only in RC oscillator mode.
4: For minimum width of INT pulse, refer to AC specs.
5: INTF is enabled to be set anytime during the Q4-Q1 cycles.

**Note:** Any interrupts caused by external signals (such as timers, capture, change on port) will have similar timing.

## 8.5 Context Saving During Interrupts

During an interrupt, only the return PC value is saved on the stack. Typically, users may wish to save key registers during an interrupt e.g. W register and STATUS register. This has to be implemented in software.

The action of saving information is commonly referred to as “PUSHing,” while the action of restoring the information before the return is commonly referred to as “POPing.” These (PUSH, POP) are not instruction mnemonics, but are conceptual actions. This action can be implemented by a sequence of instructions. For ease of code transportability, these code segments can be made into MACROs (see MPASM Assembler User’s Guide for details on creating macros).

Example 8-1 stores and restores the STATUS and W registers for devices with common RAM (such as the PIC16C77). The user register, W_TEMP, must be defined across all banks and must be defined at the same offset from the bank base address (i.e., W_TEMP is defined at 0x70 - 0x7F in Bank0). The user register, STATUS_TEMP, must be defined in Bank0, in this example STATUS_TEMP is also in Bank0.

The steps of Example 8-1:

1. Stores the W register regardless of current bank.
2. Stores the STATUS register in Bank0.
3. Executes the Interrupt Service Routine (ISR) code.
4. Restores the STATUS (and bank select bit register).
5. Restores the W register.

If additional locations need to be saved before executing the Interrupt Service Routine (ISR) code, they should be saved after the STATUS register is saved (step 2), and restored before the STATUS register is restored (step 4).

**Example 8-1: Saving the STATUS and W Registers in RAM (for Devices with Common RAM)**

```
     MOVWF   W_TEMP         ; Copy W to a Temporary Register
                            ;    regardless of current bank
     SWAPF   STATUS,W       ; Swap STATUS nibbles and place
                            ;    into W register
     MOVWF   STATUS_TEMP    ; Save STATUS to a Temporary register
                            ;    in Bank0
     :
     : (Interrupt Service Routine (ISR) )
     :
     SWAPF   STATUS_TEMP,W  ; Swap original STATUS register value
                            ;    into W (restores original bank)
     MOVWF   STATUS         ; Restore STATUS register from
                            ;    W register
     SWAPF   W_TEMP,F       ; Swap W_Temp nibbles and return
                            ;    value to W_Temp
     SWAPF   W_TEMP,W       ; Swap W_Temp to W to restore original
                            ;    W value without affecting STATUS
```

**Example 8-6: Source File Template**

```
     LIST   p = p16C77      ; List Directive,
;    Revision History
;
     #INCLUDE    <P16C77.INC>    ; Microchip Device Header File
;
     #INCLUDE    <MY_STD.MAC>    ; Include my standard macros
     #INCLUDE    <APP.MAC>       ; File which includes macros specific
                                 ;   to this application
; Specify Device Configuration Bits
     __CONFIG    _XT_OSC & _PWRTE_ON & _BODEN_OFF & _CP_OFF & _WDT_ON
;
     org   0x00            ; Start of Program Memory
RESET_ADDR   :             ; First instruction to execute after a reset


     end
```

**Example 8-7: Typical Interrupt Service Routine (ISR)**

```
    org  ISR_ADDR           ;
      PUSH_MACRO            ; MACRO that saves required context registers,
                            ;   or in-line code
      CLRF   STATUS         ; Bank0
      BTFSC  PIR1, TMR1IF   ; Timer1 overflow interrupt?
      GOTO   T1_INT         ; YES
      BTFSC  PIR1, ADIF     ; NO, A/D interrupt?
      GOTO   AD_INT         ; YES, do A/D thing
      :                     ; NO, do this for all sources
      :                     ;
      BTFSC  PIR1, LCDIF    ; NO, LCD interrupt
      GOTO   LCD_INT        ; YES, do LCD thing
      BTFSC  INTCON, RBIF   ; NO, Change on PORTB interrupt?
      GOTO   PORTB_INT      ; YES, Do PortB Change thing
INT_ERROR_LP1               ; NO, do error recovery
      GOTO   INT_ERROR_LP1  ; This is the trap if you enter the ISR
                            ;    but there were no expected
                            ;    interrupts
T1_INT                      ; Routine when the Timer1 overflows
      :                     ;
      BCF     PIR1, TMR1IF  ; Clear the Timer1 overflow interrupt flag
      GOTO    END_ISR       ; Ready to leave ISR (for this request)
AD_INT                      ; Routine when the A/D completes
      :                     ;
      BCF     PIR1, ADIF    ; Clear the A/D interrupt flag
      GOTO    END_ISR       ; Ready to leave ISR (for this request)
LCD_INT                     ; Routine when the LCD Frame begins
      :                     ;
      BCF     PIR1, LCDIF   ; Clear the LCD interrupt flag
      GOTO    END_ISR       ; Ready to leave ISR (for this request)
PORTB_INT                   ; Routine when PortB has a change
      :                     ;
END_ISR                     ;
        POP_MACRO           ; MACRO that restores required registers,
                            ;   or in-line code
        RETFIE              ; Return and enable interrupts
```

## 9.1 Introduction

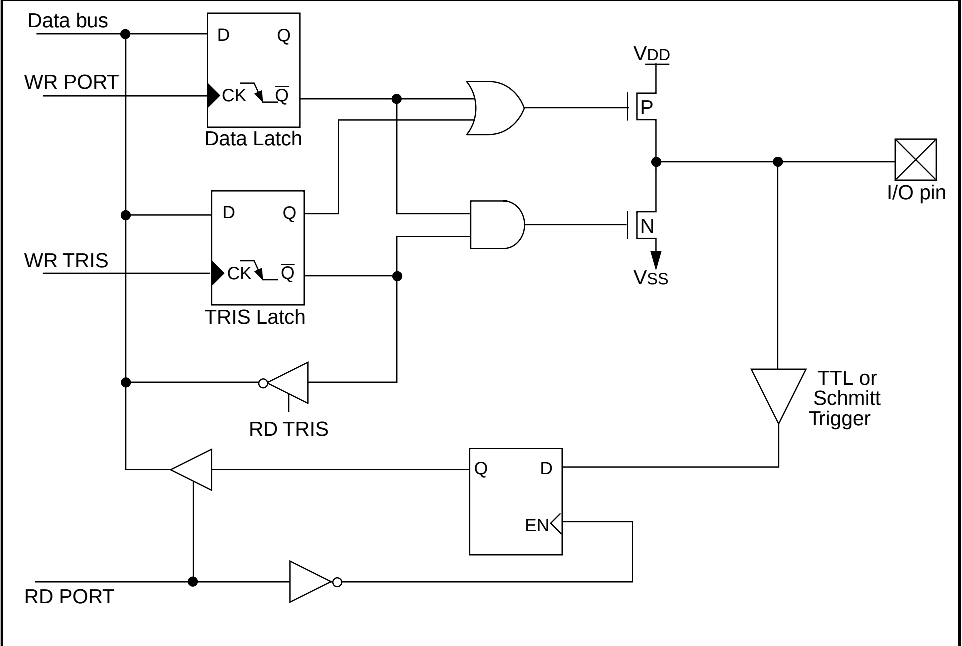
General purpose I/O pins can be considered the simplest of peripherals. They allow the PICmicro™ to monitor and control other devices. To add flexibility and functionality to a device, some pins are multiplexed with an alternate function(s). These functions depend on which peripheral features are on the device. In general, when a peripheral is functioning, that pin may not be used as a general purpose I/O pin.

For most ports, the I/O pin's direction (input or output) is controlled by the data direction register, called the TRIS register. TRIS<x> controls the direction of PORT<x>. A '1' in the TRIS bit corresponds to that pin being an input, while a '0' corresponds to that pin being an output. An easy way to remember is that a '1' looks like an I (input) and a '0' looks like an O (output).

The PORT register is the latch for the data to be output. When the PORT is read, the device reads the levels present on the I/O pins (not the latch). This means that care should be taken with read-modify-write commands on the ports and changing the direction of a pin from an input to an output.

Figure 9-1 shows a typical I/O port. This does not take into account peripheral functions that may be multiplexed onto the I/O pin. Reading the PORT register reads the status of the pins whereas writing to it will write to the port latch. All write operations (such as BSF and BCF instructions) are read-modify-write operations. Therefore a write to a port implies that the port pins are read, this value is modified, and then written to the port data latch.

**Figure 9-1: Typical I/O Port**



Note: I/O pin has protection diodes to $V_{DD}$ and $V_{SS}$.

## 18.1 Introduction

The Universal Synchronous Asynchronous Receiver Transmitter (USART) module is one of the two serial I/O modules (other is the SSP module). The USART is also known as a Serial Communications Interface or SCI. The USART can be configured as a full duplex asynchronous system that can communicate with peripheral devices such as CRT terminals and personal computers, or it can be configured as a half duplex synchronous system that can communicate with peripheral devices such as A/D or D/A integrated circuits, Serial EEPROMs etc.

The USART can be configured in the following modes:

- Asynchronous (full duplex)
- Synchronous - Master (half duplex)
- Synchronous - Slave (half duplex)

The SPEN bit (RCSTA<7>), and the TRIS bits, have to be set in order to configure the TX/CK and RX/DT pins for the USART.

## 18.2 Control Registers

**Register 18-1: TXSTA: Transmit Status and Control Register**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | U-0 | R/W-0 | R-1 | R/W-0 |
|---|---|---|---|---|---|---|---|
| CSRC | TX9 | TXEN | SYNC | — | BRGH | TRMT | TX9D |
| bit 7 | | | | | | | bit 0 |

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode
Don’t care

Synchronous mode
1 = Master mode (Clock generated internally from BRG)
0 = Slave mode (Clock from external source)

bit 6 **TX9**: 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN**: Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled

**Note:** SREN/CREN overrides TXEN in SYNC mode.

bit 4 **SYNC**: USART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

bit 3 **Unimplemented:** Read as '0'

bit 2 **BRGH**: High Baud Rate Select bit
Asynchronous mode
1 = High speed
0 = Low speed

Synchronous mode
Unused in this mode

bit 1 **TRMT**: Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of transmit data. Can be parity bit.

| Legend | |
|---|---|
| R = Readable bit | W = Writable bit |
| U = Unimplemented bit, read as ‘0’ | - n = Value at POR reset |

**Register 18-2: RCSTA: Receive Status and Control Register**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | U-0 | R-0 | R-0 | R-0 |
|---|---|---|---|---|---|---|---|
| SPEN | RX9 | SREN | CREN | — | FERR | OERR | RX9D |
| bit 7 | | | | | | | bit 0 |

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (Configures RX/DT and TX/CK pins as serial port pins)
0 = Serial port disabled

bit 6 **RX9**: 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN**: Single Receive Enable bit
Asynchronous mode
Don’t care

Synchronous mode - master
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.

Synchronous mode - slave
Unused in this mode

bit 4 **CREN**: Continuous Receive Enable bit
Asynchronous mode
1 = Enables continuous receive
0 = Disables continuous receive

Synchronous mode
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

bit 3 **Unimplemented:** Read as '0'

bit 2 **FERR**: Framing Error bit
1 = Framing error (Can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR**: Overrun Error bit
1 = Overrun error (Can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** 9th bit of received data, can be parity bit.

| Legend | | |
|---|---|---|
| R = Readable bit | W = Writable bit | |
| U = Unimplemented bit, read as ‘0’ | | - n = Value at POR reset |

## 18.3 USART Baud Rate Generator (BRG)

The BRG supports both the Asynchronous and Synchronous modes of the USART. It is a dedicated 8-bit baud rate generator. The SPBRG register controls the period of a free running 8-bit timer. In asynchronous mode bit BRGH (TXSTA<2>) also controls the baud rate. In synchronous mode bit BRGH is ignored. Table 18-1 shows the formula for computation of the baud rate for different USART modes which only apply in master mode (internal clock).

Given the desired baud rate and Fosc, the nearest integer value for the SPBRG register can be calculated using the formula in Table 18-1, where X equals the value in the SPBRG register (0 to 255). From this, the error in baud rate can be determined.

**Table 18-1: Baud Rate Formula**

| SYNC | BRGH = 0 (Low Speed) | BRGH = 1 (High Speed) |
|---|---|---|
| 0 | (Asynchronous) Baud Rate = FOSC/(64(X+1)) | Baud Rate= FOSC/(16(X+1)) |
| 1 | (Synchronous) Baud Rate = FOSC/(4(X+1)) | NA |

X = value in SPBRG (0 to 255)

Example 18-1 shows the calculation of the baud rate error for the following conditions:

FOSC = 16 MHz
Desired Baud Rate = 9600
BRGH = 0
SYNC = 0

**Example 18-1: Calculating Baud Rate Error**

Desired Baud rate = Fosc / (64 (X + 1))
9600 = 16000000 / (64 (X + 1))
X = $\lfloor 25.042 \rfloor$ = 25

Calculated Baud Rate = 16000000 / (64 (25 + 1))
= 9615

Error = $\frac{\text{(Calculated Baud Rate - Desired Baud Rate)}}{\text{Desired Baud Rate}}$
= (9615 - 9600) / 9600
= 0.16%

It may be advantageous to use the high baud rate (BRGH = 1) even for slower baud clocks. This is because the FOSC / (16(X + 1)) equation can reduce the baud rate error in some cases.

Writing a new value to the SPBRG register causes the BRG timer to be reset (or cleared). This ensures the BRG does not wait for a timer overflow before outputting the new baud rate.

**Table 18-2: Registers Associated with Baud Rate Generator**

| Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other resets |
|---|---|---|---|---|---|---|---|---|---|---|
| TXSTA | CSRC | TX9 | TXEN | SYNC | — | BRGH | TRMT | TX9D | `0000 -010` | `0000 -010` |
| RCSTA | SPEN | RX9 | SREN | CREN | — | FERR | OERR | RX9D | `0000 -00x` | `0000 -00x` |
| SPBRG | Baud Rate Generator Register | | | | | | | | `0000 0000` | `0000 0000` |

Legend: x = unknown, - = unimplemented read as '0'. Shaded cells are not used by the BRG.

**Table 18-3: Baud Rates for Synchronous Mode**

| BAUD RATE (Kbps) | FOSC = 20 MHz KBAUD | % ERROR | SPBRG value (decimal) | 16 MHz KBAUD | % ERROR | SPBRG value (decimal) | 10 MHz KBAUD | % ERROR | SPBRG value (decimal) | 7.15909 MHz KBAUD | % ERROR | SPBRG value (decimal) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.3 | NA | - | - | NA | - | - | NA | - | - | NA | - | - |
| 1.2 | NA | - | - | NA | - | - | NA | - | - | NA | - | - |
| 2.4 | NA | - | - | NA | - | - | NA | - | - | NA | - | - |
| 9.6 | NA | - | - | NA | - | - | 9.766 | +1.73 | 255 | 9.622 | +0.23 | 185 |
| 19.2 | 19.53 | +1.73 | 255 | 19.23 | +0.16 | 207 | 19.23 | +0.16 | 129 | 19.24 | +0.23 | 92 |
| 76.8 | 76.92 | +0.16 | 64 | 76.92 | +0.16 | 51 | 75.76 | -1.36 | 32 | 77.82 | +1.32 | 22 |
| 96 | 96.15 | +0.16 | 51 | 95.24 | -0.79 | 41 | 96.15 | +0.16 | 25 | 94.20 | -1.88 | 18 |
| 300 | 294.1 | -1.96 | 16 | 307.69 | +2.56 | 12 | 312.5 | +4.17 | 7 | 298.3 | -0.57 | 5 |
| 500 | 500 | 0 | 9 | 500 | 0 | 7 | 500 | 0 | 4 | NA | - | - |
| HIGH | 5000 | - | 0 | 4000 | - | 0 | 2500 | - | 0 | 1789.8 | - | 0 |
| LOW | 19.53 | - | 255 | 15.625 | - | 255 | 9.766 | - | 255 | 6.991 | - | 255 |

| BAUD RATE (Kbps) | FOSC = 5.0688 MHz KBAUD | % ERROR | SPBRG value (decimal) | 4 MHz KBAUD | % ERROR | SPBRG value (decimal) | 3.579545 MHz KBAUD | % ERROR | SPBRG value (decimal) | 1 MHz KBAUD | % ERROR | SPBRG value (decimal) | 32.768 kHz KBAUD | % ERROR | SPBRG value (decimal) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.3 | NA | - | - | NA | - | - | NA | - | - | NA | - | - | 0.303 | +1.14 | 26 |
| 1.2 | NA | - | - | NA | - | - | NA | - | - | 1.202 | +0.16 | 207 | 1.170 | -2.48 | 6 |
| 2.4 | NA | - | - | NA | - | - | NA | - | - | 2.404 | +0.16 | 103 | NA | - | - |
| 9.6 | 9.6 | 0 | 131 | 9.615 | +0.16 | 103 | 9.622 | +0.23 | 92 | 9.615 | +0.16 | 25 | NA | - | - |
| 19.2 | 19.2 | 0 | 65 | 19.231 | +0.16 | 51 | 19.04 | -0.83 | 46 | 19.24 | +0.16 | 12 | NA | - | - |
| 76.8 | 79.2 | +3.13 | 15 | 76.923 | +0.16 | 12 | 74.57 | -2.90 | 11 | 83.34 | +8.51 | 2 | NA | - | - |
| 96 | 97.48 | +1.54 | 12 | 1000 | +4.17 | 9 | 99.43 | +3.57 | 8 | NA | - | - | NA | - | - |
| 300 | 316.8 | +5.60 | 3 | NA | - | - | 298.3 | -0.57 | 2 | NA | - | - | NA | - | - |
| 500 | NA | - | - | NA | - | - | NA | - | - | NA | - | - | NA | - | - |
| HIGH | 1267 | - | 0 | 100 | - | 0 | 894.9 | - | 0 | 250 | - | 0 | 8.192 | - | 0 |
| LOW | 4.950 | - | 255 | 3.906 | - | 255 | 3.496 | - | 255 | 0.9766 | - | 255 | 0.032 | - | 255 |

# PIC16F87XA

## 28/40/44-Pin Enhanced Flash Microcontrollers

### Devices Included in this Data Sheet:

- PIC16F873A
- PIC16F874A
- PIC16F876A
- PIC16F877A

### High-Performance RISC CPU:

- Only 35 single-word instructions to learn
- All single-cycle instructions except for program branches, which are two-cycle
- Operating speed: DC – 20 MHz clock input
  DC – 200 ns instruction cycle
- Up to 8K x 14 words of Flash Program Memory,
  Up to 368 x 8 bytes of Data Memory (RAM),
  Up to 256 x 8 bytes of EEPROM Data Memory
- Pinout compatible to other 28-pin or 40/44-pin PIC16CXXX and PIC16FXXX microcontrollers

### Peripheral Features:

- Timer0: 8-bit timer/counter with 8-bit prescaler
- Timer1: 16-bit timer/counter with prescaler, can be incremented during Sleep via external crystal/clock
- Timer2: 8-bit timer/counter with 8-bit period register, prescaler and postscaler
- Two Capture, Compare, PWM modules
  - Capture is 16-bit, max. resolution is 12.5 ns
  - Compare is 16-bit, max. resolution is 200 ns
  - PWM max. resolution is 10-bit
- Synchronous Serial Port (SSP) with SPI™ (Master mode) and $I^2C$™ (Master/Slave)
- Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI) with 9-bit address detection
- Parallel Slave Port (PSP) – 8 bits wide with external $\overline{RD}$, $\overline{WR}$ and $\overline{CS}$ controls (40/44-pin only)
- Brown-out detection circuitry for Brown-out Reset (BOR)

### Analog Features:

- 10-bit, up to 8-channel Analog-to-Digital Converter (A/D)
- Brown-out Reset (BOR)
- Analog Comparator module with:
  - Two analog comparators
  - Programmable on-chip voltage reference (VREF) module
  - Programmable input multiplexing from device inputs and internal voltage reference
  - Comparator outputs are externally accessible

### Special Microcontroller Features:

- 100,000 erase/write cycle Enhanced Flash program memory typical
- 1,000,000 erase/write cycle Data EEPROM memory typical
- Data EEPROM Retention > 40 years
- Self-reprogrammable under software control
- In-Circuit Serial Programming™ (ICSP™) via two pins
- Single-supply 5V In-Circuit Serial Programming
- Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation
- Programmable code protection
- Power saving Sleep mode
- Selectable oscillator options
- In-Circuit Debug (ICD) via two pins

### CMOS Technology:

- Low-power, high-speed Flash/EEPROM technology
- Fully static design
- Wide operating voltage range (2.0V to 5.5V)
- Commercial and Industrial temperature ranges
- Low-power consumption

| Device | Program Memory | | Data SRAM (Bytes) | EEPROM (Bytes) | I/O | 10-bit A/D (ch) | CCP (PWM) | MSSP | | USART | Timers 8/16-bit | Comparators |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Bytes | # Single Word Instructions | | | | | | SPI | Master $I^2C$ | | | |
| PIC16F873A | 7.2K | 4096 | 192 | 128 | 22 | 5 | 2 | Yes | Yes | Yes | 2/1 | 2 |
| PIC16F874A | 7.2K | 4096 | 192 | 128 | 33 | 8 | 2 | Yes | Yes | Yes | 2/1 | 2 |
| PIC16F876A | 14.3K | 8192 | 368 | 256 | 22 | 5 | 2 | Yes | Yes | Yes | 2/1 | 2 |
| PIC16F877A | 14.3K | 8192 | 368 | 256 | 33 | 8 | 2 | Yes | Yes | Yes | 2/1 | 2 |

```
;***************************************************************************
;
;                           PVK40 Example : Wire
;
; Processor : PIC16F877A
; Clock : XT 3.2768 MHz
;
; On your PVK40 board set DIP switches according to following way:
; 1) Switch off all DIP switches except:
; 2) Switch on OSC 3.276M on S9
; 3) Switch on B3 LED on S11
;***************************************************************************
; Assembler directives :
        list    p = PIC16F877A  ; processor type
        __config 0x3F71         ; configuration setting
;---------------------------------------------------------------------------
; Symbol definition :
status  equ     0x03            ;status is on the 0x03 address
portb   equ     0x06
trisb   equ     0x06            ;direct addressing
portd   equ     0x08
trisd   equ     0x08            ;direct addressing
;
#define PB      portd,0         ;pushbutton 0 is on the RD0 pin
#define LED     portb,3         ;LED is on the RB3 pin
#define RP0     status,5        ;RP0 is bit 5 in status register
;---------------------------------------------------------------------------
        org     0               ;program starts at address 0x000
        bsf     RP0             ;bank 1 in RAM memory
        movlw   B'11110111'
        movwf   trisb           ;pin RB3 is output
        movlw   B'11111111'
        movwf   trisd           ;portd pins are inputs
        bcf     RP0             ;bank 0 in RAM memory
;
Main:   btfss   PB              ;is PB 0 or 1?
        goto    Main_A          ;if PB=0, jump to main_A
        bcf     LED             ;PB=1, LED off
        goto    Main            ;closes the loop
Main_A: bsf     LED             ;LED on
        goto    Main            ;closes the loop
;***************************************************************************
; end of PVK40 Example : Wire
        end
```

```
;*************************************************************************
;
;                           PVK40 Example : Blink
;
; Processor : PIC16F877A
; Clock : XT 3.2768 MHz
;
; On your PVK40 board set DIP switches according to following way:
; 1) Switch off all DIP switches except:
; 2) Switch on OSC 3.276M on S9
; 3) Switch on B3 LED on S11
;*************************************************************************
; Assembler directives :
        list    p = PIC16F877A  ; processor type
        __config 0x3F71         ; configuration setting
;-------------------------------------------------------------------------
; Special Function Register definition :
status  equ     0x03            ;status is on the 0x03 address
portb   equ     0x06
trisb   equ     0x06            ;direct addressing
; General Purpose Register definition :
cnt1    equ     0x20            ;used for Wait subroutine
cnt2    equ     0x21            ;used for Wait subroutine
cnt3    equ     0x22            ;used for Wait subroutine
; Destination definition :
w       equ     0x00
f       equ     0x01
; Bits definition :
#define LED     portb,3         ;LED is on the RB3 pin
#define RP0     status,5        ;RP0 is bit 5 in status register
;-------------------------------------------------------------------------
        org     0               ;program starts at address 0x000
        bsf     RP0             ;bank 1 in RAM memory
        movlw   B'11110111'
        movwf   trisb           ;pin RB3 is output
        bcf     RP0             ;bank 0 in RAM memory
;
Main:   bsf     LED             ;LED on
        call    Wait            ;wait 0.5 second
        bcf     LED             ;LED off
        call    Wait            ;wait 0.5 second
        goto    Main            ;closes the loop
;-------------------------------------------------------------------------
Wait:   movlw   0x05            ;this subroutine wait 0.5 second
        movwf   cnt3            ;1 cycle = 1/(Fosc/4) second =>
Wait_A: movlw   0x6B            ;=> we need 409600 cycles
        movwf   cnt2            ;0x05*0x6B*0xFF*3 = 409275
Wait_B: movlw   0xFF
        movwf   cnt1
Wait_C: decfsz  cnt1,f          ;decrement cnt1
        goto    Wait_C
        decfsz  cnt2,f          ;and if cnt1=0 then decrement cnt2
        goto    Wait_B
        decfsz  cnt3,f          ;and if cnt2=0 then decrement cnt3
        goto    Wait_A
        return                  ;if cnt3=0 then return
;*************************************************************************
; end of PVK40 Example : Blink
        end
```