В. Тимофеев
osa@pic24.ru

MPASM

Простые решения для PIC16

(пособие для начинающих)

Содержание:

Вступление

Микроконтроллеры PIC16, по сегодняшним меркам, обладают довольно скудными ресурсами: и ROM<=8 килослов, и RAM <= 384 байт, и скорость <= 5 МИПС. Конечно же, это не значит, что их нужно откинуть в сторону и искать более богатые ресурсами МК. Существует масса применений МК, где ресурсов PIC16 хватит с лихвой. Тем не менее, вопросы написания эффективного компактного кода для данной серии МК особо актуальны. В этом пособии приведены несколько приемов для повышения эффективности некоторых конструкций. Конечно, здесь не охвачены все возможные варианты улучшения. И, уж тем более, не стоит рассматривать описанные здесь приемы как рекомендации по оптимизации (оптимизация – это, все-таки, более обширное понятие, и ее целью является ужатие/ускорение кода в целом, а не выкраивание байтов и тактов на небольших конструкциях кода).

Хорошо, если описанные здесь приемы войдут в привычку, и программист будет их применять на автомате, не изобретая каждый раз велосипед. В этом случае задача оптимизации кода перед программистом встанет не раньше, чем в ней действительно возникнет необходимость, и тогда ее целью уже будет улучшение кода, а не приведение его в порядок методом облагораживания.

Для удобства восприятия в данном пособии применены следующие цветовые маркеры:

```
На розовом фоне будет приведен неэффективный код
```

```
На зеленом фоне будет приведен эффективный код
```

```
На сером фоне будут приведены примеры и сопроводительные коды
```

Запись константы в регистр без участия WREG

Часто бывают ситуации, когда нужно произвести запись в какой-нибудь регистр, а аккумулятор WREG в этот момент содержит полезную информацию. Тривиальное решение – это сбросить WREG во временную ячейку, произвести запись, а потом восстановить WREG.

```
movwf    temp
movlw    b’00100100’
movwf    register
movf     temp, w
```

Для произведения таких операций нужно иметь доступный свободный регистр в текущем банке памяти (или можно применять какой-нибудь неиспользуемый в данный момент РОН, например **fsr**). Вдобавок инструкция MOVF воздействует на флаг **status,z**. В некоторых случаях удобнее обходиться комбинацией инструкций **clrf/bsf** или **clrf/comf/bcf**. Например, если константа в двоичном представлении имеет всего 2 единицы (например, b'00100100'), то присваивание можно выполнить следующим образом:

```
clrf    register     ; Присваивание register = 00100100
bsf     register, 2
bsf     register, 5
```

Очевидно, что данная конструкция эффективна при количестве «1» в константе <=2; при 3х «1» она будет эквивалентна записи с временной ячейкой (разве что только сама ячейка не требуется, и флаг **ZERO** остается без изменений); при 4х – будет уже выполняться на 1 такт дольше. А при 5? Если в константе установлено 5 и более «1», то мы уже рассматриваем константу, как состоящую из «1» с несколькими «0», соответственно, пользуемся комбинацией команд **clrf/comf/bcf**, например, для записи b’00111111’:

```
clrf    register     ; Присваивание register = 00111111
comf    register, f
bcf     register, 6
bcf     register, 7
```

Ниже будет показано, где подобный прием может оказаться полезным.

Обмен содержимого регистра и WREG.

Предположим, в какой-то регистр нужно записать значение, содержащееся в WREG, но при этом извлечь его текущее значение. Другими словами заставить этот регистр и WREG обменяться своими значениями. При решении в лоб, опять же, потребуются временные ячейки.

```
movwf   temp1         ; Запоминаем текущее значение WREG
movf    register, w   ; Копируем значение регистра во
movwf   temp2         ; временную переменную
movf    temp1, w      ; Копируем предыдущее значение WREG
movwf   register      ; в регистр
movf    temp2, w      ; Вспоминаем предыдущее состояние регистра
```

Однако, используя инструкцию XOR, можно обойтись и без них:

```
xorwf   register, w
xorwf   register, f
xorwf   register, w
```

Фокус в том, что после первого XOR значение **WREG** изменяется: в этом регистре будут установлены те биты, в которых были различия между его предыдущим значением и значением **register**. Эти биты используется во втором XOR, после которого в **register** все биты, отличавшиеся от первоначального значения **WREG**, будут инвертированы. Т.о. после второго XOR **register** будет равен первоначальному значению аккумулятора. Остался последний шаг. XOR, как известно, обратимая операция, и если ее выполнить еще раз над той же парой регистров **register** и **WREG**, то результат будет тем же, что и перед вторым XOR, только теперь результат сохраняется не в **register**, а в **WREG**.

Рассмотрим на примере. На входе имеем:

```
register = 0x55 = 01010101
WREG     = 0x46 = 01000110
```

1. **`xorwf register, w`**

```
register – остается без изменений
WREG     = 01010101 ^ 01000110 = 00010011 – «1» в тех разрядах, где было различие с register
```

2. **`xorwf register, f`**

```
register = 01010101 ^ 00010011 = 01000110 – стал равен первоначальному значению WREG
WREG     - остается без изменений
```

3. **`xorwf register, w`**

```
register – остается без изменений
WREG     = 00010011 ^ 01000110 = 01010101 – стал равен первоначальному значению register
```

Обновление с сохранением изменений

В обработчике прерывания по изменению **RB4**..**RB7** часто требуется знать, состояние какого именно входа изменилось. Для этого требуется иметь переменную, содержащую предыдущее состояние порта, и при чтении **portb** в обработчике прерывания сравнивать ее с только что прочитанным значением. Решение в лоб получается не очень красивым:

```
movf    pb_prev, w   ; Запоминаем предыдущее состояние
movwf   temp         ;
movf    portb, w     ; Читаем текущее состояние порта
movwf   pb_prev      ; Пишем в регистр текущее значение
xorwf   temp, w      ; Получаем разницу с предыдущим
```

Все можно упростить, применив операцию XOR.

```
movf    portb, w     ; Читаем текущее состояние
xorwf   pb_prev, w   ; Получаем разницу с предыдущим
xorwf   pb_prev, f   ; Пишем в регистр текущее значение
```

Суть этой конструкции проста: в сам регистр **pb_prev** запись производится не присваиванием, а операцией XOR с маской измененных битов. Рассмотрим пример:

```
pb_prev = 11110000 – запомненное предыдущее состояние порта
portb   = 01110000 – текущее состояние порта (как видно, изменился 7-ой бит)
```

1. **movf portb, w**

```
portb   = 01110000 – текущее состояние порта
WREG    = 01110000
```

2. **xorwf pb_prev, w**

```
pb_prev = 11110000 – запомненное предыдущее состояние порта
WREG    = 11110000 ^ 0111000 = 10000000 – после операции XOR регистр WREG содержит единицы в тех разрядах, где были различия
```

3. **xorwf pb_prev, f**

```
WREG    = 10000000 – теперь этой маской воздействуем на регистр pb_prev
pb_prev = 11110000 ^ 10000000 = 01110000
```

Как видно, теперь регистр **pb_prev** принял значение текущего состояния порта, а регистр **WREG** содержит маску измененных битов.

Работа с ячейками в разных банках

Предположим, что есть две 4-байтовых переменных, размещенные в разных банках ОЗУ: переменная **a** в нулевом банке, а **b** – в первом. Рассмотрим простой пример реализации копирования 4-х байт из одного банка в другой **a = b**. При решении в лоб появляется много переключений между банками. Это еще переменные **a** и **b** расположены так, что переключать нужно только **rp0**; а располагайся они в 0-м и 3-м (или в 1-м и 2-м), то пришлось бы переключать и **rp0** и **rp1**.

```
bsf     status, rp0
movf    b+0, w          ; Копируем первый байт
bcf     status, rp0
movwf   a+0

bsf     status, rp0
movf    b+1, w          ; копируем второй байт
bcf     status, rp0
movwf   a+1

bsf     status, rp0
movf    b+2, w          ; третий
bcf     status, rp0
movwf   a+2

bsf     status, rp0
movf    b+3, w          ; четвертый
bcf     status, rp0
movwf   a+3
```

Используя косвенную адресацию и обращаясь к переменной, расположенной в первом банке через регистр **indf**, можно данный код немного сократить:

```
movlw   b               ; Загрузка в fsr адреса переменной
movwf   fsr             ; из первого банка
movf    indf, w         ; Копирование первого байта
movwf   a+0

incf    fsr,f           ; fsr указывает на второй байт
movf    indf, w
movwf   a+1

incf    fsr,f           ; fsr указывает на третий
movf    indf, w
movwf   a+2

incf    fsr,f           ; fsr указывает на четвертый байт
movf    indf, w
movwf   a+3
```

Двойное условие

Иногда нужно выполнить фрагмент кода при выполнении сразу двух условий. Допустим, нужно выполнить какой-то код при установленных одновременно битах **CARRY** и **ZERO**. При решении в лоб получается такая конструкция:

```
    btfss   status, c
    goto    SKIP_CODE
    btfss   status, z
    goto    SKIP_CODE

    ... Код, выполняющийся при C=1 и Z=1 ...

SKIP_CODE:
```

Здесь видно, что, во-первых, дважды присутствует инструкция перехода на метку **SKIP_CODE**, а во-вторых, проверка флага **ZERO** не выполняется, если проверка флага **CARRY** не прошла. За счет этих двух обстоятельств этот код можно сделать красивее и эффективнее:

```
    btfsc   status, c
    btfss   status, z
    goto    SKIP_CODE

    ... Код, выполняющийся при C=1 и Z=1 ...

SKIP_CODE:
```

Первое условие заменяется противоположным (**btfss** -> **btfsc**), а первая инструкция **goto** убирается. Т.о. получаем следующее: при сброшенном флаге **CARRY** пропускается проверка **ZERO** и сразу выполняется **goto**.

Такой прием удобен, например, в обработчике прерываний, когда наряду с проверкой флага, вызвавшего прерывание, производится проверка флага разрешения прерывания:

```
CHECK_INTF:
    btfsc   intcon, intf
    btfss   intcon, inte
    goto    SKIP_INTF

    ... Код обработки прерывания по INTF ...

SKIP_INTF:
```

Таблицы

Всем известно, что в контроллерах PIC16 есть возможность организовывать таблицы данных, элементы из которых выбираются посредством математической операции над программным счетчиком (чаще – сложением), а возвращаются инструкцией **retlw**. Учитывая особенность записи в регистр PC (**pcl**, как известно, является отображением только младшего байта программного счетчика, и при записи в него старший байт PC берется из регистра **pclath**), таблицы размером не более 256 байт удобно размещать в пределах одного 256-словного блока (с тем, чтобы не заморачиваться с пересчетом **pclath**). Остается только одна проблема: при обращении к таблице регистр **WREG** содержит номер элемента таблицы, а нам еще требуется сделать предустановку регистра **pclath**. Существует несколько тривиальных решений:

- регистр **pclath** можно предустанавливать перед обращением к таблице;

```
    movlw   high(Table)   ; Предустанавливаем pclath
    movwf   pclath
    movlw   5             ; Хотим прочитать 5-ый элемент
    call    ReadTable
    ...
```

- внутри функции-таблицы сохранять значение **WREG** во временную ячейку, модифицировать **pclath**, а затем восстанавливать **WREG**:

```
ReadTable:
    movwf   temp          ; Сохраняем WREG
    movlw   high(Table)   ; Предустанавливаем pclath
    movwf   pclath
    movf    temp, w       ; Восстанавливаем WREG
    addwf   pcl, f
Table:
    retlw   'A'
    retlw   'B'
    ...
```

Оба варианта не очень удачны, т.к. они громоздки, а второй вдобавок требует наличия свободной ячейки ОЗУ в текущем банке памяти. Осмысленный выбор блока памяти для размещения таблицы может немного упростить задачу. Во-первых, те таблицы, которые вместятся в нулевой блок (адреса до 0x00FF), есть смысл размещать именно в нем, при этом над **pclath** будет производиться только одно действие – очистка:

```
    ORG     0x0010
ReadTable:
    clrf    pclath        ; Предустанавливаем pclath
    addwf   pcl, f
Table:
    retlw   'A'
    retlw   'B'
    retlw   'C'
    ...
```

Те таблицы, которые в нулевой блок не влезли, есть смысл размещать в таких блоках, чтобы операции с **pclath** можно было максимально эффективно производить методом присваивания без использования **WREG**, описанным в начале этого пособия. Т.е. преимущество следует отдавать тем блокам, старшая часть адреса которых содержит одну «1», а именно: 0x0100, 0x0200 и 0x0400. Если и они уже заняты, то тем, у которых в старшей части адреса 2 «1»: 0x300, 0x500, 0x600. Это будет в любом случае эффективнее, чем работа с промежуточной переменной **temp**.

```
    ORG     0x0400
ReadTable:
    clrf    pclath        ; запись 0x04 в pclath
    bsf     pclath, 2
    addwf   pcl, f
Table:
    retlw   'A'
    retlw   'B'
    retlw   'C'
    ...
```

Два примечания, не относящихся к компактности, но о которых не следует забывать.

Примечание 1. По-хорошему, следует перед выполнением «**addwf pcl, f**» ограничивать значение **WREG** в соответствии с размером таблицы, чтобы не допустить попадание в непредусмотренные алгоритмом адреса при вызове таблицы с ошибочным значением **WREG**. Например, для таблицы из 8 значений следует обнулять старшие 5 разрядов регистра **WREG**:

```
    ORG     0x0400
ReadTable:
    clrf    pclath
    bsf     pclath, 2
    andlw   0x07
    addwf   pcl, f
Table:
    dt      "ABCDEFGH"
    ...
```

Примечание 2. Если таблица большого размера и она не может целиком поместиться в 256-словном блоке памяти вместе с инструкциями присваивания **pclath** и **pcl**, то эти инструкции следует выносить на предыдущую страницу. Например, для таблицы размером 256 элементов нужно делать так:

```
    ORG     0x0400 - 3
ReadTable:
    clrf    pclath         ; Адрес 0x3FD
    bsf     pclath, 2      ; Адрес 0x3FE
    addwf   pcl, f         ; Адрес 0x3FF
Table:
    retlw   ...           ; Адрес 0x400
    retlw   ...
    retlw   ...
```

Проверка на попадание в диапазон

Допустим, нужно проверить, попадает ли значение какого-либо регистра в заданный диапазон значений. Т.е. выполнить проверку:

MIN <= **register** <= **MAX**

Решение в лоб выглядело бы так:

```
    movlw   MIN             ; Проверяем нижнюю границу
    subwf   register, w
    btfss   status, c
    goto    SKIP            ; Меньше минимума - выходим

    movlw   -(MAX + 1)      ; Проверяем верхнюю границу
    addwf   register, w
    btfsc   status, c
    goto    SKIP            ; Больше максимума - выходим

    ... Сюда попадаем, когда register попадает в диапазон ...

SKIP:
```

Однако можно воспользоваться более изящной конструкцией, всего двумя операциями сложения:

```
    movf    register, w

    addlw   -MIN            ; Сперва нормируем значение
    addlw   -(MAX - MIN + 1) ; Теперь сравниваем с диапазоном

    btfsc   status, c
    goto    SKIP

    ... Сюда попадаем, когда register попадает в диапазон ...
SKIP:
```

После выполнения второго сложения флаг **CARRY** будет сброшен, если значение внутри диапазона, или установлен, если оно не попадает в диапазон. Такой код может оказаться полезен, например, при преобразовании строчных букв в заглавные:

```
    movf    indf, w
    addlw   -'a'                ; Проверка на попадание в
    addlw   -('z' - 'a' + 1)    ; диапазон 'a'..'z'
    btfss   status, c
    bcf     indf, 5             ; Преобразуем в заглавную
```

Примечание: в данном примере известно, что если значение проверяемого регистра находится в требуемом диапазоне 'a'..'z' = 0x61..0x77 = 01**1**00001..01**1**10111 (т.е. бит 5 всегда установлен), то вычитание числа 32 = 0x20 = 00**1**00000 (именно это число нужно вычесть, чтобы получить ASCII-код заглавной буквы), можно заменить операцией сброса 5-го бита.

Сравнение регистра с несколькими константами

Часто встает задача последовательного сравнения значения регистра с несколькими константами с тем, чтобы выбрать один из вариантов продолжения программы. Например, МК по USART получил команду, и в зависимости от нее надо выполнить различные действия. Можно решить в лоб:

```
movf    command, w      ; Проверяем первую команду
xorlw   'A'
btfsc   status, z
goto    Cmd_A

movf    command, w      ; Проверяем вторую команду
xorlw   'B'
btfsc   status, z
goto    Cmd_B

movf    command, w      ; третью
xorlw   'C'
btfsc   status, z
goto    Cmd_C

... и т.д.
```

Как видно, каждое сравнение требует 4 инструкции. Но можно немного упростить, если прочитать регистр только один раз а далее всю работу вести с **WREG**:

```
movf    command, w

xorlw   'A'             ; Проверяем первую команду
btfsc   status, z
goto    Cmd_A

xorlw   'B' ^ 'A'       ; вторую
btfsc   status, z
goto    Cmd_B

xorlw   'C' ^ 'B'       ; третью
btfsc   status, z
goto    Cmd_C

... и т.д.
```

Фокус заключается в том, что, хоть значение регистра **WREG** и портится после сравнения операцией XOR, но нам известно, как именно оно портится. Т.е., если первая проверка не прошла (т.е. команда не равна 'A'), то к проверке второй команды мы приходим с уже испорченным **WREG**, а именно – *проксоренным* константой 'A'. Следовательно, для сравнения с командой 'B' нам нужно восстановить **WREG**, т.е. компенсировать предыдущее изменение, повторно *проксорив* его с константой 'A'. Т.к. операция XOR обладает свойством линейности, то компенсацию и сравнение можно делать не просто в любом порядке, но даже одновременно (что мы и видим в примере: «**xorlw 'B' ^ 'A'**»). За счет введения компенсации код каждой проверки сокращается до трех инструкций.

Если константы, с которыми будет производиться сравнение, имеют последовательные коды (например: 1,2,3… или A,B,C,…), то вместо множества проверок можно воспользоваться таблицей перехода. Рассмотрим для примера обработку команд 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H':

```
    movf    command, w
    sublw   ‘A’           ; Нормируем значение
    addwf   pcl, f
JumpTable:
    goto    Cmd_A
    goto    Cmd_B
    goto    Cmd_C
    goto    Cmd_D
    goto    Cmd_E
    goto    Cmd_F
    goto    Cmd_G
    goto    Cmd_H
```

Примечания:

1. Само собой, предварительно следует выполнить проверку на попадание в диапазон.
2. Не следует забывать об особенностях присваивания регистру **pcl**, т.е. нужно подготавливать регистр **pclath**, а также следить за тем, чтобы таблица переходов не пересекала границу 256-словного блока памяти.
3. Если список команд не совсем последовательный (скажем, в нашем примере отсутствовала бы команда 'G'), то в некоторых случаях (например, когда важна скорость) все равно есть смысл использовать таблицы, заменяя неиспользуемые переходы на, например, «**goto EXIT**» или **return**.

С учетом перечисленных примечаний код выглядел бы так:

```
    ORG     0x0080
CmdWork:
    clrf    pclath          ; Подготовка pclath

    movf    command, w

    addlw   -‘A’            ; Проверка на попадание в диапазон A..H
    addlw   -(‘H’-‘A’+1)
    btfsc   status, c
    return                  ; Не попадаем
    addlw   (‘H’-‘A’+1)     ; Выполняем компенсацию после проверки

    addwf   pcl, f
JumpTable:
    goto    Cmd_A
    goto    Cmd_B
    goto    Cmd_C
    goto    Cmd_D
    goto    Cmd_E
    goto    Cmd_F
    return                  ; Неиспользуемая команда
    goto    Cmd_H
```

Циклический сдвиг

Ассемблер PIC16 не содержит инструкции циклического сдвига (т.е. когда выталкиваемый бит сразу же заталкивается в регистр с другой стороны). Часто в программах встречается не очень удачная конструкция для выполнения этой задачи:

```
bcf     status, c
rlf     register, f ; Сдвигаем, вдвигая «0» в младший разряд
btfsc   status, c   ; Если был перенос,
bsf     register, 0 ; то устанавливаем младший разряд в «1»
```

Если есть возможность пожертвовать значением регистра **WREG**, то можно применить конструкцию всего из двух инструкций:

```
rlf     register, w  ; CARRY = старшему биту 8-разряного числа
rlf     register, f  ; Он же вдвигается на место младшего бита
```

Первым сдвигом выталкиваем бит из регистра, не меняя сам регистр. После этого флаг **CARRY** уже содержит тот бит, который должен быть вдвинут с другой стороны. Второй инструкцией **rlf** мы выполняем сдвиг регистра, заталкивая на место младшего разряда флаг **CARRY**, установленный/сброшенный первой инструкцией сдвига.

Аналогичный прием можно применять при циклическом сдвиге многобайтовых чисел:

```
rlf     data+3, w    ; CARRY = старшему биту 32-разряного числа
rlf     data+0, f    ; Он же вдвигается на место младшего бита
rlf     data+1, f
rlf     data+2, f
rlf     data+3, f
```

Таким же образом делается сдвиг вправо:

```
rrf     data+0, w    ; CARRY = младшему биту 32-разряного числа
rrf     data+3, f    ; Он же вдвигается на место старшего бита
rrf     data+2, f
rrf     data+1, f
rrf     data+0, f
```

Частый вызов подпрограммы с параметром

Предположим, что есть подпрограмма **SendByte**, которая отправляет байт внешнему устройству через USART. Байт для отсылки ей передается в регистре **WREG**, т.е. вызов происходит так:

```
    movlw   'A'
    call    SendByte
    movlw   'T'
    call    SendByte
    movlw   'H'
    call    SendByte
```

Если в программе присутствует несколько вызовов этой подпрограммы с одним и тем же параметром (например, символом пробела), то можно немного упростить, оформив вход в функцию следующим образом:

```
SendByte_SPACE:
    movlw   ' '

SendByte:
    ... Тело подпрограммы ...
```

Теперь, когда нужно передать символ пробела, можно вместо **SendByte** сразу вызывать подпрограмму **SendByte_SPACE** без предварительно установки регистра WREG.

Если часто используемых символов несколько, то иногда удобно для каждого из них сделать свой вход в функцию:

```
SendByte_POINT:
    movlw   '.'
    goto    SendByte

SendByte_COMMA:
    movlw   ','
    goto    SendByte

SendByte_SPACE:
    movlw   ' '

SendByte:
    ... Тело подпрограммы ...
```

Применение такого приема особо актуально, когда в подпрограмму передается многобайтовый параметр. Например, есть подпрограмма деления 2-байтовых чисел, которая часто вызывается с делителем 1000. Есть смысл вход в нее оформить так:

```
Divide_1000:
    movlw   high(.1000)          ; Инициализируем делитель
    movwf   divider_hi
    movlw   low(.1000)
    movwf   divider_lo

Divide:
    ... Тело подпрограммы деления ...
```

Двукратный вызов подпрограммы

Если в программе часто встречаются повторные вызовы подпрограммы с одними и теми же параметрами (например, передача по USART двух символов: перевод строки 0x0D и возврат каретки 0x0A), то вместо того, чтобы все время вызывать подпрограмму дважды, можно оформить вход в функцию следующим образом:

```
Send_EOL:                    ; Отправка End Of Line (0xD, 0xA)
    movlw   0x0D
    call    SendByte
    movlw   0x0A

SendByte:
    ... Тело подпрограммы ...
```

Подобный прием особенно удобно применять тогда, когда в программе используются явные задержки:

```
Delay_10ms:
    call    Delay_5ms

Delay_5ms:
    call    Delay_1ms
    call    Delay_1ms
    call    Delay_1ms
    call    Delay_1ms

Delay_1ms:
    ... Код задержки в 1 мс ...
```

Однако таким приемом не стоит особо увлекаться, т.к. надо помнить, что стек у PIC16 имеет всего 8 уровней.

Циклы прямого счета

В PIC16 есть очень удобные команды для организации циклов от X до 0 (**decfsz**) и от –X до 0 (**incfsz**). Но иногда по ходу цикла нужно знать номер текущей итерации, т.е. удобнее было бы иметь переменную, считающую от 0 до X. Одно из тривиальных решений выглядит так:

```
    clrf    counter
Loop:
    ... Тело цикла ...

    incf    counter, f
    movf    counter, w
    xorlw   .10
    btfss   status, z
    goto    Loop
```

Однако для некоторых констант код сравнения можно упростить. Если константа содержит только 1 единицу (например, считаем до 2, 4, 8, 16, 32, 64 или 128), то код будет выглядеть так:

```
    clrf    counter
Loop:
    ... Тело цикла ...

    incf    counter, f
    btfss   counter, 3      ; Если бит 3 установлен, то Counter = 8
    goto    Loop
```

Эта конструкция будет эффективной, даже если константа имеет 2 единицы, например:

```
10  = 00001010
20  = 00010100
130 = 10000010, и т.д.
```

Тогда, воспользовавшись приемом, описанным в разделе «Двойное условие», получим код:

```
    clrf    counter
Loop:
    ... Тело цикла ...

    incf    counter, f
    btfsc   counter, 1      ; Если биты 1 и 3 установлены,
    btfss   counter, 3      ; то Counter = 10
    goto    Loop
```

Кроме компактности этот код удачен еще тем, что не затрагивает значение **WREG**.

Сложение/вычитание многобайтовых чисел

Приведу часто встречающийся довольно эффективный код сложения двух многобайтовых чисел. Допустим, у нас есть две переменные.

```
    CBLOCK  0x20
    a:4
    b:4
    ENDC
```

Код сложения будет выглядеть так:

```
Add32:                  ; Выполняем сложение A = A + B
    movf    b+0, w      ; Суммируем 0-ые байты
    addwf   a+0, f

    movf    b+1, w      ; Суммируем 1-ые байты
    btfsc   status, c
    incfsz  b+1, w
    addwf   a+1, f

    movf    b+2, w      ; Суммируем 2-ые байты
    btfsc   status, c
    incfsz  b+2, w
    addwf   a+2, f

    movf    b+3, w      ; Суммируем 3-ые байты
    btfsc   status, c
    incfsz  b+3, w
    addwf   a+3, f
```

Его компактность достигнута за счет того, что используется комбинация двух условных инструкций: **btfss/incfsz**. После сложения пары регистров разряд переноса формируется командой **incfsz**, которая, с одной стороны, может быть пропущена, если после предыдущего сложения не было переполнения (флаг **CARRY** сброшен), а с другой (и в этом фокус данного метода) – пропускает сложение следующих байтов, если одно из слагаемых из-за добавления к нему переноса стало равным нулю; при этом флаг **CARRY** остается без изменений, т.е. перенос переходит в следующий разряд.

По структуре этого кода видно, что его легко модифицировать для работы с числами любой разрядности (кратной 8), добавляя/убирая четверки инструкций. Например, чтобы преобразовать его в сложение двух 24-разрядных чисел, достаточно просто убрать последние 4 инструкции.

Добавлю, что если требуется сложение чисел, расположенных в разных банках ОЗУ, то из-за добавления инструкций по обработке битов выбора банка **rp0** и **rp1** связку **btfsc/incfsz** эффективно применить не удастся, и код сильно разрастется. Но здесь можно применить метод, описанный в параграфе «Работа с ячейками из разных банков». Работа через косвенную адресацию оставит возможным применять связку стоящих друг за другом инструкций **btfsc** и **incfsz**.

Аналогичным образом выглядит код многобайтового вычитания. Заем также формируется инструкцией **incfsz**, которая пропускает операцию вычитания при нулевом результате, перенося бит заема в следующий разряд:

```
Sub32:                  ; Выполняем вычитание A = A - B
    movf    b+0, w      ; Вычитаем 0-ый байты
    subwf   a+0, f

    movf    b+1, w      ; Вычитаем 1-ый байт
    btfss   status, c
    incfsz  b+1, w
    subwf   a+1, f

    movf    b+2, w      ; Вычитаем 2-ой байт
    btfss   status, c
    incfsz  b+2, w
    subwf   a+2, f

    movf    b+3, w      ; Вычитаем 3-ий байт
    btfss   status, c
    incfsz  b+3, w
    subwf   a+3, f
```

Данный код так же, как и код сложения, легко модифицируется для работы с числами любой разрядности.

Рекомендуются к изучению

Очень рекомендую ознакомиться с этим Интернет-ресурсом (англоязычным): www.piclist.com – в особенности с разделом «Source Code Library» (http://www.piclist.com/techref/microchip/routines.htm), содержащим огромное количество всевозможных примеров довольно эффективных подпрограмм на ассемблере, проверенных временем: математические операции (начиная со сложения, заканчивая корнем), работа с памятью, задержки, условные переходы и многое другое. Кроме того, данный ресурс содержит много подсказок для начинающих.

Также для исследования эффективных конструкций кода очень рекомендую обратиться к Application Notes от фирмы Microchip (www.microchip.com). Многие из них поставляются с исходными кодами, написанными на ассемблере, которые заслуживают внимания программистов (особенно начинающих).