Реализация алгоритмов ЦОС на ассемблере TMS320C6000

В настоящей статье рассматриваются правила написания программного кода на ассемблере и способ компоновки проекта с использованием файлов как с С-, так с ассемблерным кодом. Применение ассемблерных вставок направлено в первую очередь на оптимизацию программного кода с учетом архитектуры выбранного ЦСП. Учет архитектуры позволяет повысить эффективность кода. Под повышением эффективности понимается либо увеличение скорости выполнения, либо уменьшение объема занимаемой памяти, либо то и другое одновременно. Однако программный код становится зависимым от выбранного семейства ЦСП. В статье будет показан пример разработки ассемблерного программного кода для проекта цифрового фильтра на ЦСП семейства TMS320C6000, разработанного в предыдущих статьях.

Игорь ГУК
gii@scanti.ru

ЦСП — это обычная электронная вычислительная машина (ЭВМ). Архитектура и набор команд такой ЭВМ реализованы с учетом построения на их базе систем ЦОС. Большинство ЦСП имеют блочную структуру (рис. 1):

- ядро ЦСП;
- внутренняя память;
- набор периферийных устройств (периферия);
- шина.

Ядро выполняет обработку данных, а также управляет функционированием ЦСП в целом. Состав ядра:

- устройство управления (УУ), координирующее работу самого ядра и ЦСП в целом;
- арифметико-логическое устройство (АЛУ), выполняющее обработку данных;
- набор регистров общего назначения (РОН) для кратковременного хранения исходных данных и результатов вычислений, используемых в данный момент АЛУ;
- набор регистров специального назначения (РСН) для хранения данных, управляющих процессом функционирования ЦСП.

Внутренняя память предназначена для хранения кода программы, исходных данных и результатов вычислений. Ее составляют:

- память программ (ПП) для хранения программного кода,
- память данных (ПД) для хранения исходных данных и результатов вычислений.

Обычно вся ПП или ее часть реализуется на основе постоянного запоминающего устройства (ПЗУ), сохраняющего записанную в него информацию даже при выключенном питании. А ПД строится на основе оперативного запоминающего устройства (ОЗУ), информация в котором исчезает при выключении питания.

Шина осуществляет обмен данными между блоками ЦСП.

Периферия выполняет функции обмена данными с внешними устройствами. Состав периферийных модулей изменяется в очень широких пределах в зависимости от предназначения ЦСП.

Обобщенный алгоритм выполнения программного кода:

- чтение команды из ПП;
- дешифрование команды в УУ;
- чтение операндов из ПД в РОН;
- выполнение операции в АЛУ и формирование результата в РОН;
- перенос результата из РОН в ПД;
- чтение следующей команды и т. д.

Кроме этого, в процессе работы ЦСП необходимо множество служебных данных: адрес исполняемой команды, адреса операндов, адреса стека, признаки выполнения операции, данные о конфигурации ЦСП и т. д. Все эти данные хранятся в РСН.

Структура ЦСП семейства TMS320C6000 фирмы TI показана на рис. 2. Отличия от рассмотренной обобщенной схемы следующие:

- ПП и ПД в некоторых модификациях ЦСП могут быть объединены;
- АЛУ содержит два набора модулей по четыре в каждом (.M1, .D1, .S1, .L1 и .M2, .D2, .S2, .L2);
- РОН включают 32 регистра, разбитых на две части (A0–A15 и B0–B15).

Объединение памяти снижает стоимость процессора. Применение восьми модулей АЛУ позволяет выполнять до восьми опера-

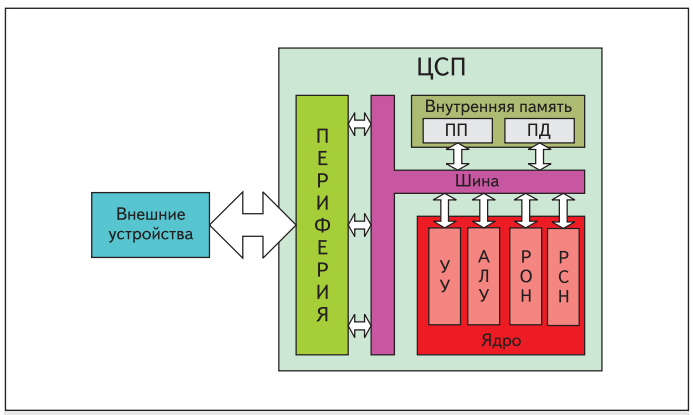

Рис. 1. Обобщенная структура ЦСП

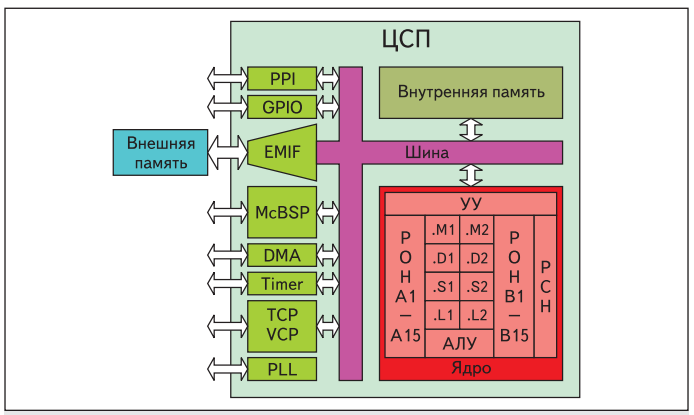

Рис. 2. Структура ЦСП TMS320C6000

ций за один такт обработки. А наличие 32 РОН значительно снижает временные затраты на операции чтения-записи памяти.

Периферия включает модули:

- PPI *(Parallel Peripheral Interface)* — параллельный периферийный интерфейс, который обеспечивает механизм начальной загрузки процессора;
- GPIO *(General Purpose Input/Output)* — вход-выход общего назначения, который предназначен для поразрядного ввода-вывода 8-/16-разрядных данных;
- EMIF *(External Memory Interface)* — интерфейс внешней памяти, который обеспечивает подключение синхронной или асинхронной памяти;
- McBSP *(Multi-Channel Buffered Serial Port)* — многоканальный буферизированный последовательный порт ввода/вывода реализует последовательные полнодуплексные синхронные каналы ввода/вывода со скоростью обмена до 100 Мбайт/с;
- DMA *(Direct Memory Access)* — интерфейс прямого доступа к памяти ЦСП, он предназначен для организации каналов пересылки данных между произвольными областями памяти и периферийными модулями без участия ядра ЦСП;
- Timer — таймер-счетчик, реализующий несколько 32-разрядных таймеров-счетчиков, которые могут инициализировать прерывания;
- TCP *(Turbo Coprocessor)* и VCP *(Viterbi Coprocessor)* — сопроцессоры, обеспечивающие поддержку голосовых каналов и каналов передачи данных по стандарту 3GPP/IS2000;
- PLL *(Phase Lock Loop)* — модуль сопряжения с внешним генератором (система ФАПЧ), который обеспечивает умножение частоты внешнего генератора.

В зависимости от модификации ЦСП семейства TMS320C6000 набор периферийных модулей может меняться, могут появиться новые модули или исчезать перечисленные выше.

Для повышения эффективности программного кода часто возникает необходимость переписать его часть на языке низкого уровня — ассемблере.

Строка ассемблерного кода для семейства C6000 имеет следующую структуру:

```
{метка:} {параллельность} {[условие]} мнемоника {модуль} операнд1,
операнд2 … {; комментарии}
```

Здесь:

- первое поле (*метка*) предназначено для меток, используемых при переходах в программном коде;
- второе поле (*параллельность*) предназначено для размещения оператора в виде вертикальных линий «||», определяющего режим, когда выполнение текущей команды происходит одновременно (параллельно) с предыдущей;
- третье поле (*условие*) содержит в квадратных скобках имя регистра, значение которого является условием для выполнения данной строчки ассемблерного кода (при нулевом значении регистра текущая строка считается пустым оператором);
- четвертое поле (*мнемоника*) является командой, которую необходимо выполнить;
- пятое поле (*модуль*) указывает, каким именно модулем АЛУ должна выполняться команда;
- шестое и последующие поля (*операнд*1, *операнд*2 …) содержат входные и выходные операнды команды, в качестве которых используются РОН;
- последнее поле (; *комментарии*) содержит начинающийся с символа «;» текст комментариев, поясняющих смысл выполняемой команды и игнорируемый компилятором.

В фигурных скобках указаны необязательные поля. Нулевая позиция в тексте ассемблерного кода отводится под метку. Любой текст, начинающийся с этой позиции, (за исключением комментариев) рассматривается компилятором как метка.

Пример ассемблерного кода:

```
; Начало цикла: for(i = lenBuff - 1; i >= 0; i--) {
                ; Определение начального значения переменной цикла
                SUB   .L2     lenBuff_A, 1, i_B       ; i = lenBuff - 1;

                ; Определение начального значения индекса
                ; входного и выходного буферов
                ZERO  .L1     count_A                 ; count = 0;

                ; Точка обратного перехода при циклических
                ; вычислениях
loop:
                ; Чтение данных из входного буфера
                LDH   .D1T1   *+pInpBuff_A[count_A], var_A
                                                      ; var = pInpBuff[count];

                ; Ожидания завершения операции чтения (4 такта)
                NOP 4

                ; Умножение отсчета на два (при помощи сдвига)
                SHL   .S1     var_A, 1, var_A         ; var <<= 1;

                ; Запись результата в выходной буфер
                STH   .D1T1   var_A, *+pOutBuff_A[count_A]
                                                      ; pOutBuff[count] = var;

                ; Инкрементация индекса массива
                ADD   .L1     count_A, 1, count_A     ; count++;

                ; Условный переход на метку с параллельной
                ; декрементацией переменной цикла
     [i_B]      B     .S2     loop
  || [i_B]      SUB   .L2     i_B, 1, i_B

                ; Ожидания завершения операции условного
                ; перехода (5 тактов)
                NOP 5                                 ; }
; Конец цикла
```

Жирным шрифтом показаны мнемоники, а курсивом — комментарии. В данном примере реализован цикл умножения входного буфера на два и запись результата в выходной буфер.

Состав основных команд ассемблера процессоров семейства TMS320C6000:

- Арифметические операции:
  - сложение (мнемоники — ADD, ADDU, SADD, ADD2, ADDAB, ADDAH, ADDAW);
  - вычитание (мнемоники — SSUB, SUB, SUBU, SUBC, SUB2, SUBAB, SUBAH, SUBAW);
  - умножение (мнемоники — MPY, MPYU, MPYUS, MPYSU, MPYH, MPYHU, MPYHUS, MPYHSU, MPYHL, MPYHLU, MPYHULS, MPYHSLU, MPYLH, MPYLHU, MPYLUHS, MPYLSHU, SMPY, SMPYHL, SMPYLH, SMPYH);
  - сдвиг (мнемоники — SHL, SHR, SHRU, SSHL);
  - модуль (мнемоника — ABS).
- Логические действия:
  - логические операции (мнемоники — AND, NOT, OR, XOR);
  - операции сравнения (мнемоники — CMPEQ, CMPGT, CMPGTU, CMPLT, CMPLTU).
- Работа с памятью:
  - чтение (мнемоники — LDB, LDBU, LDH, LDHU, LDW);
  - запись (мнемоники — STB, STH, STW).
- Прочие (мнемоники — MV, MVC, MVK, MVKH, MVKLH, NORM, NEG, ZERO, LMBD, SAT, B, CLR, EXT, EXTU, SET).

Выше дан только краткий перечень основных команд. Более подробное описание команд можно посмотреть в литературе, поставляемой вместе с CCS под номером *SPRU189*. Для вызова перечня прилагаемой к ИСР CCS справочной литературы необходимо в главном меню выбрать пункт *Help*, а в нем подпункт *User Manual*. (рис. 3). В статье будут описаны только те команды, которые присутствуют в примере ассемблерного кода.

При создании смешанного программного кода, когда используются исходные файлы как на языке высокого уровня Си, так и ассемблерные вставки, необходимо придерживаться некоторых правил составления программы. Они определяют набор соглашений по использованию регистров общего назначения и вызову функций.

В качестве исходных будут использованы файлы с программным кодом из предыдущей статьи. Создайте новый проект, скопируйте необходимые файлы из проекта предыдущей статьи, откомпилируйте и убедитесь, что новый проект идентичен по результату старому.

Затем необходимо определить имя функции, которую нужно переписать на ассемблере. Для примера выбрана функция *convolution*(). Необходимо установить курсор на файл *convolution.cpp*, содержащий требуемую функцию, и, щелкнув на нем правой кнопкой мы-

Рис. 3. Вызов перечня дополнительной литературы

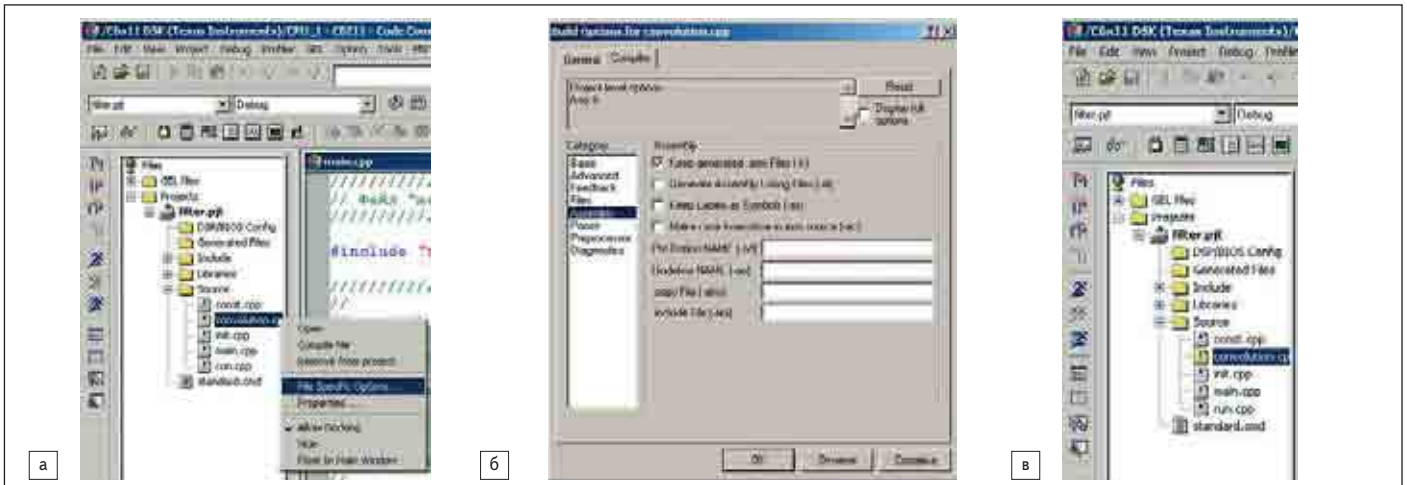

Рис. 4. Вызов перечня дополнительной литературы

ши, выбрать в контекстном меню пункт *File Specific Options* (рис. 4а). В появившемся окне на закладке *Compiler* в разделе *Category* выбрать пункт *Assembly* и установить флажок в позиции *Keep generated .asm Files* (рис. 4б). Нажать кнопку подтверждения ввода параметров (кнопка «*OK*»). Выбранный файл будет отмечен желтым цветом (рис. 4в), что говорит об отличии параметров его настройки от параметров настройки проекта в целом.

Откомпилировать проект заново, нажав кнопку . В папке проекта появится ассемблерный файл *convolution.asm*. В нем находится сгенерированный CCS ассемблерный код для выбранной функции (рис. 5).

Из этого файла необходимо извлечь две строки:

```
«.sect      «.text»» — секция, где расположена функция;
«.global    _convolution__FPsT1s» — ассемблерное имя функции.
```

Теперь можно приступить к написанию ассемблерного кода для выбранной функции, придерживаясь соглашений по использованию регистров, приведенных ниже.

- *Соглашение первое.* Регистры A0–A9 и B0–B9 могут быть использованы вызываемой функцией без ограничений. Регистры A10–A15 и B10–B15 не могут быть использованы вызываемой функцией без их предварительного сохранения в стеке.
- *Соглашение второе.* Регистры A1–A2 и B0–B2 быть использованы в качестве регистров условия при выполнении команд (поле «условие»).
- *Соглашение третье.* Регистр B3 предназначен для передачи в вызываемую функцию адреса возврата.
- *Соглашение четвертое.* Регистры A4, B4, A6, B6, A8, B8, A10, B10, A12, B12, A14, B14 предназначены для передачи в вызываемую функцию ее параметров с первого по двенадцатый соответственно. Если параметров больше чем двенадцать, то они передаются через стек. В данном примере передача параметров через стек не рассматривается.
- *Соглашение пятое.* Регистр B15 предназначен для хранения адреса вершины стека.

Шаблон ассемблерного файла выглядит следующим образом:

```
; Назначение имен регистров
      .asg   B3, adrReturn_B
      .asg   B15, SP

; Определение функции

; Запрет прерываний
                  MVC      .S2      CSR, B0
                  AND      .S2      B0, -2, B1
                  MVC      .S2      B1, CSR

; Сохранение регистров в стеке
                  STW      .D2T2    B0, *-SP[11]

; Алгоритм обработки

; Восстановление регистров
                  LDW      .D2T2    *-SP[11], B0
                  NOP 4

; Восстановление регистра CSR
                  MVC      .S2      B0, CSR

; Выход из функции
                  B        .S2      adrReturn_B
                  NOP 5
```

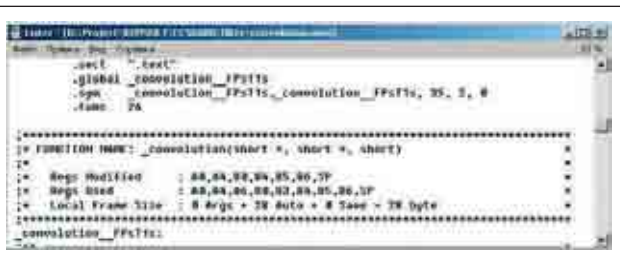

Рис. 5. Файл с ассемблерным кодом

В разделе «*Назначение имен регистров*» регистрам общего назначения присваиваются осмысленные имена. Эту функцию выполняет директива предпроцессора «*.asg*». Обычно этот раздел ассоциируется с той частью программного кода функции на языке Си, где происходит создание локальных переменных, а также с параметрами функции.

Раздел «*Определение функции*» предназначен для указания секции размещения функции (директива предпроцессора «*.sect*»), определения имени (директива — «*.global*») и установки точки входа в функцию (установка метки с именем функции). Для рассматриваемого примера:

```
      .sect      «.text»                ; секция размещения функции
      global     _convolution__FPsT1s   ; имя функции
_convolution__FPsT1s:                   ; точка входа в функцию
```

Написание ассемблерного кода подразумевает использование программного конвейера. Для корректной работы функции необходимо запретить обработку прерываний на время выполнения конвейера, что обеспечивается установкой нулевого значения в первом бите служебного регистра CSR. Работать напрямую со служебными регистрами в ЦСП семейства TMS320C6000 нельзя. Поэтому в разделе «*Запрет прерываний*» значение регистра CSR переносится в РОН при помощи специальной команды считывания и записи служебных регистров:

```
                  MVC      .S2      CSR, B0.
```

Установка в ноль первого бита производится наложением маски:

```
                  AND      .S2      B0, -2, B1.
```

Новое значение перемещается в CSR:

```
MVC     .S2     B1, CSR.
```

Старое значение CSR сохранено в РОН *B*0.

Если необходимо использовать все регистры общего назначения, то в разделе «*Сохранение регистров в стеке*» значения регистров с номерами 10–15 переносятся в стек. В примере в стеке сохраняется только старое значение CSR. Операция выполняется командой «*STx*»:

```
STW     .D2T2     B0, *-SP[11],
```

где «*W*» означает сохранение 32-разрядного значения («*B*» для 8-ми разрядного и «*H*» для 16-ти разрядного); «*.D2T2*» означает использование для обработки команды модуля АЛУ «*.D2*»; «**-SP[11]*» определяет место в стеке, где сохраняется значение регистра *B*0 (адрес определяется как разность базового адреса SP и смещения, указанного в квадратных скобках).

Перенос данных из памяти в регистр и наоборот осуществляют специальные элементы ядра ЦСП, называемые путем коммутации. Их всего два. Один обеспечивает сохранение/чтение данных на стороне *A* (регистры A0–A15), а другой — на стороне *B* (регистры B0–B15). На какой стороне сохранять (или с какой читать) данные, определяют суффиксы «*T1*» и «*T2*». В данном случае «*T2*» указывает на то, что чтение данных производится со стороны *B*.

Перед выходом из функции необходимо восстановить значения регистров общего назначения с номерами 10–11, а также старое значение CSR. Это действие выполняется в разделе «*Восстановление регистров*» при помощи команды «*LDx*». В примере происходит восстановление только старого значения регистра CSR:

```
LDW     .D2T2     *-SP[11], B0.
```

Значение полей аналогично полям команды «*STx*». Необходимо учитывать, что результат выполнения команды чтения памяти появится в регистре общего назначения только через 4 такта. Поэтому выполняются четыре пустых оператора (*NOP* 4).

В разделе «*Алгоритм обработки*» располагается ассемблерный код алгоритма обработки, который реализует функцию.

Раздел «*Выход из функции*» определяет точку выхода из функции:

```
B       .S2     adrReturn_B,
```

где «*B*» — команда перехода; «*.S2*» — используемый для выполнения команды модуль; «*adrReturn_B*» — регистр, содержащий адрес возврата из функции.

Команда перехода выполняется с задержкой на пять тактов (*NOP* 5).

При необходимости, перед выполнением команды выхода из функции в регистр A4 заносят возвращаемое значение.

Прежде чем заполнить раздел «*Алгоритм обработки*», модифицируем программный Си-код функции. Это делается для более полного соответствия кода на языке высокого уровня коду на ассемблере. Такой подход позволяет значительно упростить как написание ассемблерного кода, так и его отладку.

При модификации Си-кода необходимо придерживаться следующих правил:

- не использовать переменную цикла в качестве индекса массива;
- организовывать цикл с использованием декрементации переменной цикла;
- при возможности заменить индексирование массива инкрементацией или декрементацией адреса;
- реализовать алгоритм только при помощи операций, которым можно поставить в соответствие команды ассемблера;

Текст модифицированного С-кода функции *convolution*() имеет вид:

```
#include «filter.h»

word16 convolution(word16* pSimpl, word16* pCoeff, word16 sizefilter){
   // Объявление локальных переменных
   word32 sum; // Переменная для накопления суммы
   word32 mpy; // Переменная для временного хранения
               // результата умножения
   word32 n; // Переменная цикла
   word16 coeff; // Переменная для коэффициента фильтра
   word16 simpl; // Переменная для отсчета из линии задержки

   // Инициализация локальных переменных
   sum = 0; // Установка начального значения суммы

   // Цикл обработки линии задержки
   for(n = sizefilter — 1; n >= 0; n--){
      // Чтение коэффициента фильтра
      // с последующей инкрементацией адреса
      coeff = *pCoeff++;

      // Чтение отсчета из линии задержки
      // с последующей инкрементацией адреса
      simpl = *pSimpl++;

      // Умножение коэффициента на отсчет
      mpy = coeff * simpl;

      // Накопление результата умножения
      sum += mpy;
   }

   // Коррекция адреса линии задержки
   pSimpl--;

   // Цикл сдвига линии задержки
   for (n = sizefilter — 2; n >= 0; n--){
      // Чтение предыдущего отсчета
      simpl = pSimpl[-1];

      // Запись предыдущего отсчета на место текущего
      // с последующей декрементацией адреса
      *pSimpl-- = simpl;
   }

   // Нормирование результата суммирования
   sum >>= 15;

   // Выход из функции
   return sum;
}
```

После набора кода необходимо откомпилировать программу, запустить на выполнение и убедиться в ее корректной работе (сравнив выходные файлы, как это описывалось ранее).

Исходя из модифицированного С-кода функции гораздо проще написать ассемблерный код. Один из возможных вариантов ассемблерного кода (написанный на основе приведенного выше шаблона ассемблерной функции) имеет вид:

```
; Назначение имен регистрам
   .asg   A4, pSimpl_A      ; word16* pSimpl;
                            ; (первый параметр функции)
   .asg   A5, simpl_A       ; word16 simpl;
   .asg   A6, sizefilter_A  ; word16 sizefilter;
                            ; (третий параметр функции)
   .asg   A7, sum_A         ; word32 sum;
   .asg   A8, mpy_A         ; word32 mpy;
   .asg   B0, n_B           ; Переменная цикла
   .asg   B3, adrReturn_B   ; Адрес возврата
   .asg   B4, pCoeff_B      ; word16* pCoeff;
                            ; (второй параметр функции)
   .asg   B6, coeff_B       ; word16 coeff;
   .asg   B15, SP           ; Указатель на стек

; Определение функции
   .sect   «.text»                  ; секция размещения функции
   .global _convolution__FPsT1s     ; имя функции
_convolution__FPsT1s:               ; точка входа в функцию (метка)

; Запрет прерываний
                MVC   .S2     CSR, B0
                AND   .S2     B0, -2, B1
                MVC   .S2     B1, CSR

; Сохранение регистров в стеке
                STW   .D2T2   B0, *-SP[11]

; Алгоритм обработки
; Инициализация локальных переменных
                ZERO  .D1     sum_A            ; sum = 0;

; Цикл обработки линии задержки
; for(n = sizefilter - 1; n >= 0 n++){
             ; Определение начального значения переменной цикла
                SUB   .L2     sizefilter_A, 1, n_B
loop01:      ; Точка возврата при циклических вычислениях
                LDH   .D2T2   *pCoeff_B++, coeff_B
                                          ; coeff = *pCoeffTmp++;
                LDH   .D1T1   *pSimpl_A++, simpl_A
                                          ; simpl = *pSimplTmp++;

                NOP 4 ; Ожидание завершения операции чтения
                      ; данных из памяти

                MPY   .M1X    coeff_B, simpl_A, mpy_A
                                          ; mpy = coeff * simpl;

                NOP   ; Ожидание завершения операции умножения

                ADD   .S1     sum_A, mpy_A, sum_A
                                          ; sum += mpy;

                ; Условный переход при циклических
                ; вычислениях и одновременная условная
                ; декрементация переменной цикла
     [n_B]      B     .S2     loop01
  || [n_B]      SUB   .L2     n_B, 1, n_B

                NOP 5 ; Ожидание завершения операции
                      ; условного перехода
; }
                ; Коррекция адреса линии задержки
                SUB   .L1     pSimpl_A, 2, pSimpl_A
                                          ; pSimplTmp--;

; Цикл сдвига линии задержки
; for (n = sizefilter — 2; n >= 0; n--){
                ; Определение начального значения переменной
                ; цикла
                SUB   .L2     sizefilter_A, 2, n_B        ;
loop02:      ; Точка возврата при циклических вычислениях
                LDH   .D1T1   *pSimpl_A[-1], simpl_A
                                          ; simpl = pSimplTmp[-1];

                NOP 4 ; Ожидание завершения операции чтения
                      ; данных из памяти

                STH   .D1T1 simpl_A, *pSimpl_A--
                                          ; *pSimplTmp-- = simpl;

                ; Условный переход при циклических
                ; вычислениях и одновременная
                ; условная декрементация переменной цикла
     [n_B]      B     .S2     loop02
  || [n_B]      SUB   .L2     n_B, 1, n_B

                NOP 5 ; Ожидание завершения операции
                      ; условного перехода
; }
                ; Нормирование результата суммирования
                SHR   .S1     sum_A, 15, sum_A    ; sum>>=15;

; Восстановление регистров
                LDW   .D2T2   *-SP[11], B0
                NOP 4

; Восстановление регистра CSR
                MVC   .S2     B0, CSR

; Выход из функции
                ; Перемещение возвращаемого значения в регистр A4
                MV    .S1     sum_A, A4

                B     .S2     adrReturn_B
                NOP 5
```

Необходимо создать файл для исходного кода, набрать текст ассемблерной функции и сохранить его с именем «*convolution_my.asm*», затем подключить его к проекту, как это было показано в предыдущей статье цикла.

Теперь к проекту подключено два файла, в которых описана одна и та же функция *convolution*():

- в файле «*convolution.cpp*» функция представлена кодом на языке Си;
- в файле «*convolution_my.asm*» эта же функция представлена на ассемблере.

Необходимо исключить файл с Си-кодом из процесса компиляции. Для этого нужно щелкнуть правой кнопкой мыши на файле и выбрать пункт *File Specific Options* (рис. 6а). В появившемся окне перейти на закладку *General* и установить флажок в позиции *Exclude file from build* (рис. 6б).

Затем необходимо произвести компиляцию проекта и запустить программный код на выполнение. Сравнив результаты, полученные при выполнении программы с Си-кодом и с ассемблерным кодом функции *convolution*(), можно убедиться в их идентичности.

В конце статьи рассмотрим краткое описание применяемых ассемблерных команд.

Суффикс «*X*», появившийся у объявления модуля выполнения команды умножения:

```
MPY     .M1X     coeff_B, simpl_A, mpy_A,
```

говорит о том, что один из входных операндов находится в противоположном относительно модуля наборе РОН. Для этой операции используется специальный элемент ядра ЦСП — кросс-путь. Таких элементов только два — один обеспечивает перенос данных из регистров A0–A15 для модулей с номером 2, а другой — в обратном направлении (данных из регистров B0–B15 для модулей с номером 1). Использовать кросспуть могут все модули, за исключением модуля «*.D*».

Ниже приведено краткое описание команд, использованных в рассмотренном примере.

- *.asg* — директива, которая назначает РОН произвольное имя.
- *.sect* — директива, которая определяет секцию в памяти, где будет размещен бинарный код функции.
- *.global* — директива, которая определяет глобальное имя функции.
- *MVC* — команда перемещения 32-разрядных данных между РОН и РСН. Выполняется модулем «.S2».
- *MV* — команда перемещения 32-разрядных данных между РОН. Выполняется модулями «.S», «.L» или «.D».
- *AND* — поразрядная операция «логическое И». Выполняется модулями «.S» или «.L».
- *STW* — команда заполняет память 32-разрядным знаковым значением из РОН. Выполняется только модулем «.D». Например,

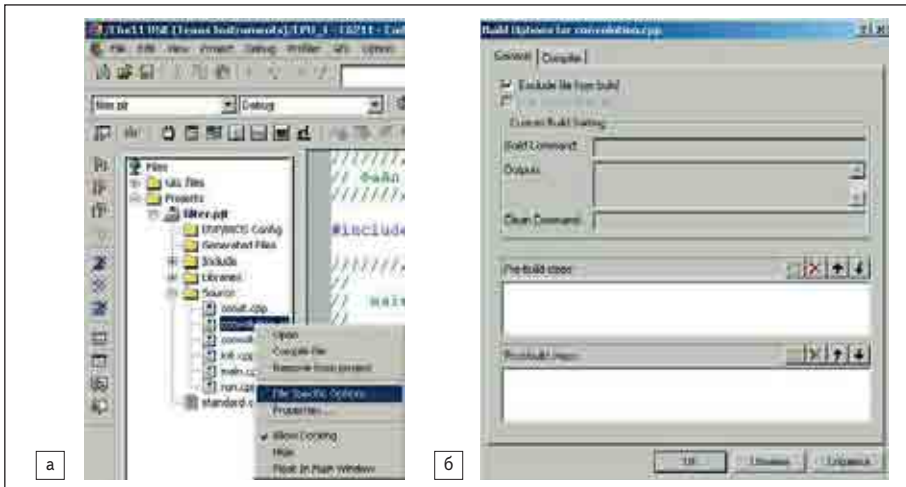

Рис. 6. Исключение файла из процесса компилирования проекта

```
STW     .D2T2     Src,*-baseR[offsetR],
```

где Src — регистр с данными; baseR и offsetR — базовый адрес и смещение для переноса данных соответственно. Значение регистра с базовым адресом после выполнения команды остается прежним.

- *STH* — команда заполняет память 16-разрядным знаковым значением из РОН. Выполняется только модулем «.D». Например,

```
STH     .D2T2     Src,*baseR--,
```

где Src — регистр с данными, baseR — базовый адрес для переноса данных. Значение регистра с базовым адресом после выполнения команды декрементируется.

- *LDW* — команда заполняет регистр 32-разрядным знаковым значением из памяти. Выполнятется только модулем «.D». Например,

```
LDW     .D2T2     *-baseR[offsetR], Dst
```

где Dst — регистр для приема данных, baseR и offsetR — базовый адрес и смещение для чтения данных из памяти соответственно. Значение регистра с базовым адресом после выполнения команды остается прежним. Значение в РОН появится только через 4 такта.

- *LDH* — команда заполняет регистр 16-разрядным знаковым значением из памяти. Выполняться только модулем «.D». Например,

```
LDH     .D2T2     *baseR++, Dst
```

где Dst — регистр для приема данных, baseR — базовый адрес для чтения данных из памяти. Значение регистра с базовым адресом после выполнения команды инкрементируется. Значение в РОН появится только через 4 такта.

- *SUB* — команда определяет разность между значениями двух регистров и заносит результат в третий. Выполняется модулями «.S», «.L» или «.D». Например,

```
SUB     .L2     Src1, Src2, Dst
```

В данном случае происходит вычитание из значения в регистре Src1 значения в регистре Src2, а результат заносится в Dst. При использовании модуля «.D» вычитание производится из Src2.

- *ADD* — команда определяет сумму значений двух регистров и заносит результат в третий. Выполняется модулями «.S», «.L» или «.D».
- *ZERO* — команда заполняет регистр нулями. Выполняется модулями «.S», «.L» или «.D».
- *MPY* — команда реализует операцию умножения 16-разрядных операндов. Результат 32-разрядный. Выполняется только модулем «.M». Значение в РОН появится через 1 такт.
- *B* — команда осуществляет переход на метку, имя которой указано в качестве операнда. Переход будет осуществлен через 5 тактов после команды.
- *SHR* — команда выполняет сдвиг вправо первого операнда на количество разрядов, определяемое вторым операндом. Результат с расширением знака размещается в третьем операнде.
- *NOP* — пустая операция.

В следующей статье будет показано, как при помощи программного конвейера и параллельного выполнения команд повысить эффективность (скорость выполнения) ассемблерного кода программы. Все файлы проекта можно найти на сайте **w ww. scanti.ru**. Напомним, что при первом запуске проекта на новом компьютере необходимо загрузить его через пункт Project главного меню. Затем настроить интерфейс заново и сохранить настройки нового рабочего пространства.