Оптимизация программного кода для ЦСП TMS320C6000

Игорь ГУК
gii@scanti.ru

В четвертой статье цикла рассматриваются способы оптимизации программного кода на основе конвейера и параллельной обработки ассемблерных команд для цифровых сигнальных процессоров семейства C6000 компании Texas Instruments (TI). Для усвоения материала рекомендуется ознакомиться с предыдущими публикациями («КиТ» № 7–9'2005).

Оптимизация программного кода может быть выполнена на нескольких уровнях:

- на уровне алгоритма — оптимизируется С-код;
- на уровне архитектуры ЦСП — оптимизируется ассемблерный код.

В рассматриваемом примере оптимизацию проведем для функции свертки *convolution*() из проекта, рассмотренного в предыдущих статьях цикла.

Вначале необходимо запустить CCS и создать новый проект. Напомним, что для этого в главном меню ИСР нужно выбрать раздел *Project*, в нем — пункт *New*, затем в появившемся диалоговом окне выбираются семейство ЦСП и тип проекта, а также указывается место размещения проекта на жестком диске и его имя.

В папку проекта скопировать файлы с исходным Си-кодом рассмотренного в прошлой статье фильтра. Скопированные файлы, за исключением заголовочного файла «*filter.h*», подключаются к проекту (в главном меню — раздел *Project*, а в нем — пункт *Add Files to Project*). Напомним, что заголовочный файл подключается автоматически при компиляции проекта.

В Си-коде оптимизируемой функции удаляется цикл сдвига за счет использования цикловой адресации. Модернизированный программный код функции *convolution*() примет вид:

```
word16 convolution (word16* pSimplBuff, word16* pCoeffBuff,
                    word16* pSimplMin, word16 lenFir,
                    word32 lenSimpl){
    // Объявление переменных
    word32 count;          // Переменная цикла
    word32 summa;          // Переменная для временного
                           // хранения результата накопления
                           // умножений
    word32 coeff;          // Коэффициент фильтра
    word32 simpl;          // Текущий отсчет
    word32 mpy;            // Результат умножения
    word16* pSimplMax;     // Максимальный адрес буфера
                           // задержанных отсчетов
    word16 flag;           // Флаг циклической адресации
    // Инициализация переменных
    summa = 0;
    pSimplMax = pSimplMin + lenSimpl – 1;
    // Вычисление свертки
    for (count = lenFir – 1; count >=0; count--){
        // Чтение отсчета
        simpl = *pSimplBuff++;
        // Чтение коэффициента
        coeff = *pCoeffBuff++;
        // Умножение коэффициента на отсчет
        mpy = simpl * coeff;
        // Суммирование с накоплением
        summa += mpy;
        // Организация циклической адресации
        if(pSimplBuff > pSimplMax) flag = 1;
        else flag = 0;
        if(flag) pSimplBuff = pSimplMin;
    }
    // Нормализация результата
    summa >>= 15;
    // Возврат из функции результата свертки
    return summa;
}
```

Появились дополнительные параметры вызова функции:

- *pSimplMin* — указатель на начало буфера задержки;
- *lenSimpl* — длина линии задержки.

Особенность кода в том, что длина линии задержки должна быть кратна двум байтам и быть больше длины фильтра. Первое ограничение обусловлено способом реализации цикловой адресации в семействе ЦСП TMS320C6000. Изменение интерфейса функции влечет необходимость изменений и в других компонентах проекта.

В заголовочном файле «*filter.h*» необходимо:

- изменить объявление функции свертки:

```
extern word16 convolution(word16*, word16*, word16, word16,
word32);
```

- дополнить объявление типа контекстной структуры параметрами *pSimplMin*, и *lenSimpl*:

```
typedef struct {
    word16* pInpBuff;      // Указатель на входной буфер
    word16* pOutBuff;      // Указатель на выходной буфер
    word16 lenBuff;        // Длина входного и выходного буферов
    word16* pSimplBuff;    // Текущий указатель на буфер линии
                           // задержки
    word16* pSimplMin;     // Указатель на начало буфера линии
                           // задержки
    word16* pCoeffBuff;    // Указатель на буфер
                           // с коэффициентами фильтра
    word16 lenFilter;      // Длина буферов линии задержки
                           // и коэффициентов
    word32 mCoeff;         // Масштабирующий коэффициент
    word32 lenSimpl;       // Длина линии задержки
} CONTEXTFILTER;
```

- определить макрос для инициализации нового поля контекстной структуры:

```
#define MASKINDEX 255.
```

В функции *initFilter*() необходимо добавить инициализацию новых полей контекстной структуры:

```
    pCntx->pSimplMin   = simplBuff;
    pCntx->lenSimpl    = LENSIMPL;
```

а также изменить верхний предел в цикле начальной очистки буфера линии задержки:

```
for(i = 0; i < LENSIMPL; i++) pCntx->pSimplBuff[i] = 0;
```

В файле «*const.cpp*» изменить создание буфера задержанных отсчетов:

```
    #pragma DATA_ALIGN (LENSIMPL << 1);
    word16 simplBuff[LENSIMPL];
```

Директива *#pragma DATA_ALIGN (LENSIMPL << 1)* выравнивает адрес массива *simplBuff*.

Листинг новой функция запуска фильтрации *runFilter*() приведен ниже.

```
void runFilter(CONTEXTFILTER* pCntx){
    // Локальные переменные
    word16* pInpBuff;      // Указатель на входной буфер
    word16* pOutBuff;      // Указатель на выходной буфер
    word16  lenBuff;       // Длина входного и выходного буферов
    word16* pSimplBuff;    // Текущий указатель на буфер линии
                           // задержки
    word16* pSimplMin;     // Указатель на начало буфера линии
                           // задержки
    word16* pSimplMax;     // Указатель на конец буфера линии
                           // задержки
    word16* pCoeffBuff;    // Указатель на буфер
                           // с коэффициентами фильтра
    word16  lenFilter;     // Длина буферов линии задержки
                           // и коэффициентов
    word32  mCoeff;        // Масштабирующий коэффициент
    word32  count;         // Переменная цикла
    word32  coeff;         // Вспомогательная переменная
    word32  lenSimpl;      // Длина линии задержки
    // Инициализация локальных переменных
    pInpBuff    = pCntx->pInpBuff;
    pOutBuff    = pCntx->pOutBuff;
    lenBuff     = pCntx->lenBuff;
    pSimplBuff  = pCntx->pSimplBuff;
    pSimplMin   = pCntx->pSimplMin;
    pCoeffBuff  = pCntx->pCoeffBuff;
    lenFilter   = pCntx->lenFilter;
    mCoeff      = pCntx->mCoeff;
    lenSimpl    = pCntx->lenSimpl;
```

```
    pSimplMax  = pSimplMin + lenSimpl – 1;
    // Цикл обработки входного буфера
    for(count = 0; count < lenBuff; count++){
            // Чтение входного отсчета
            coeff = pInpBuff[count];
            // Умножение на масштабирующий коэффициент
            coeff *= mCoeff;
            // Нормирование результата
            coeff >>= 15;
            // Запись в буфер задержанных отсчетов
            *pSimplBuff = (word16) coeff;
            // Определение выходного отсчета
            coeff = convolution(pSimplBuff, pCoeffBuff, pSimplMin,
            lenFilter, lenSimpl);
            // Запись выходного отсчета
            pOutBuff[count] = (word16) coeff;
            // Декрементация адреса буфера задержанных отсчетов
            pSimplBuff--;
            // Организация циклической адресации
            if(pSimplBuff < pSimplMin) pSimplBuff = pSimplMax;
    }
    // Сохранение текущего значения адреса буфера
    pCntx->pSimplBuff = pSimplBuff;
}
```

Необходимо создать новый исходный файл, набрать текст ассемблерного кода оптимизируемой функцией свертки и сохранить файл с именем «*convolution_my_2.asm*». Затем подключить файл с ассемблерным кодом функции свертки к проекту, а файл с Си-кодом исключить из процесса компиляции. Как это сделать, было показано в предыдущих статьях цикла. Ассемблерный код функции свертки имеет вид:

```
; Назначение имен регистрам
    .asg   A2, flag_A          ; word16 flag;          // Флаг цикли-
                                                       // ческой адре-
                                                       // сации
    .asg   A3, mpy_A           ; word32 mpy;           // Результат
                                                       // умножения
    .asg   A4, pSimplBuff_A ; word16* pSimplBuff;  // Первый
                                                       // параметр
                                                       // функции
    .asg   A5, simpl_A         ; word32 simpl;         // Текущий
                                                       // отсчет
    .asg   A6, pSimplMin_A  ; word16* pSimplMin    // Третий
                                                       // параметр
                                                       // функции
    .asg   A7, summa_A         ; word32 sum;           // Переменная
                                                       // для накопле-
                                                       // ния умно-
                                                       // жений
    .asg   A8, lenSimpl_A      ; word32 maskIndex;     // Пятый
                                                       // параметр
                                                       // функции
    .asg   A9, pSimplMax_A  ; word16* pSimplMax;   // Максималь-
                                                       // ный адрес
                                                       // буфера
                                                       // отсчетов
    .asg   B0, count_B         ; Переменная цикла
    .asg   B3, adrReturn_B     ; Адрес возврата
    .asg   B4, pCoeffBuff_B ; word16* pCoeffBuff;  // Второй
                                                       // параметр
                                                       // функции
    .asg   B5, coeff_B         ; word32 coeff;         // Коэффици-
                                                       // ент фильтра
    .asg   B6, lenFir_B        ; word16 lenFir;        // Четвертый
                                                       // параметр
                                                       // функции
    .asg   B15, SP             ; Указатель на стек
; Определение функции
    .sect   «.text»                    ; секция размещения функции
    .global _convolution__FPsN21si     ; имя функции
_convolution__FPsN21si:                ; точка входа в функцию
; 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 переменных
            ZERO    .D1     summa_A        ; summa = 0;
            ; pSimplMax = pSimplMin + lenSimpl – 1;
            SUB     .S1     lenSimpl_A, 1, lenSimpl_A
            ADDAH   .D1     pSimplMin_A, lenSimpl_A, pSimplMax_A
    ; Вычисление свертки
```

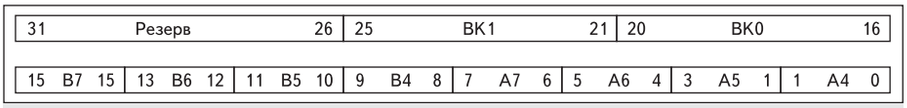

Рис. 1. Структура служебного регистра ARM

```
    ;for (count = lenFir – 1; count >=0; count--){
            ; Определение начального значения переменной цикла
            SUB     .L2     lenFir_B, 1, count_B
loop01: ; Точка возврата при циклических вычислениях
            ; Чтение отсчета
            ; simpl = *pSimplBuff++;
            ; Чтение коэффициента
            ; coeff = *pCoeffBuff++;
            LDH     .D2T2   *pCoeffBuff_B++, coeff_B
            ; coeff = *pCoeffBuff++;
    ||      LDH     .D1T1   *pSimplBuff_A++, simpl_A
            ; simpl = *pSimplBuff++;

            NOP 4   ; Ожидание завершения операции чтения
                    ; данных из памяти

            ; Умножение коэффициента на отсчет
            MPY     .M1X    coeff_B, simpl_A, mpy_A
            ; mpy = coeff * simpl;

            NOP     ; Ожидание завершения операции умножения

            ADD     .S1     summa_A, mpy_A, summa_A
            ; summa += mpy;

            ; Организация циклической адресации
            ; if(pSimplBuff > pSimplMax) flag = 1;
            ; else flag = 0;
            CMPGT   .L1     pSimplBuff_A, pSimplMax_A, flag_A

            ; if(flag) pSimplBuff = pSimplMin;
            [flag_A]    MV  .S1
            pSimplMin_A, pSimplBuff_A,

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

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

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

            NOP 4   ; Ожидание завершения последней операции
                    ; чтения (4 такта)

; Восстановление регистра CSR
            MVC     .S2     B0, CSR

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

            B       .S2     adrReturn_B

            NOP 5   ; Ожидание завершения операции безусловного
                    ; перехода (5 тактов)
```

Программный код, представленный выше, не является оптимальным. Его необходимо модернизировать. На первом шаге используем возможность организации цикловой адресации в ЦСП TMS320C6000.

Для организации циклической адресации используются регистры общего назначения A4–A7 и B4–B7. Режим работы данных РОН определяется служебным регистром AMR. Структура регистра AMR показана на рис. 1.

Режим работы регистра, выбранного для циклической адресации, определяется значением полей A4–A7 и B4–B7 (соответствующих РОН аналогичного наименования):

- «00» — линейная адресация (режим по умолчанию);
- «01» — циклическая адресация с указанием длины буфера в поле BK0;
- «10» — циклическая адресация с указанием длины буфера в поле BK1;
- «11» — резерв.

Длина буфера циклической адресации в байтах определяется значением полей BK0 и BK1 (табл. 1).

Код ассемблерной функции с цикловой адресацией показан ниже. Хранится он в файле «*convolution_my_3.asm*». Необходимо его создать, подключить к проекту и исключить из процесса компиляции предыдущий файл «*convolution_my_2.asm*».

```
; Назначение имен регистрам
    .asg   A3, mpy_A           ; word32 mpy;           // Результат
                                                       // умножения
    .asg   A4, pSimplBuff_A ; word16* pSimplBuff;  // Первый
                                                       // параметр
                                                       // функции
    .asg   A5, simpl_A         ; word32 simpl;         // Текущий
                                                       // отсчет
    .asg   A7, summa_A         ; word32 sum;           // Переменная
                                                       // для накопле-
                                                       // ния умноже-
                                                       // ний
    .asg   A8, maskIndex_A  ; word32 maskIndex;    // Пятый
                                                       // параметр
                                                       // функции
    .asg   B0, count_B         ; Переменная цикла
    .asg   B3, adrReturn_B     ; Адрес возврата
    .asg   B4, pCoeffBuff_B ; word16* pCoeffBuff;  // Второй
                                                       // параметр
                                                       // функции
    .asg   B5, coeff_B         ; word32 coeff;         // Коэффициент
                                                       // фильтра
    .asg   B6, lenFir_B        ; word16 lenFir;        // Четвертый
                                                       // параметр
                                                       // функции
    .asg   B15, SP             ; Указатель на стек
; Определение функции
    .sect   «.text»                    ; секция размещения функции
    .global _convolution__FPsN21si     ; имя функции
_convolution__FPsN21si:                ; точка входа в функцию
```

Таблица 1. Размер буфера в зависимости от значения полей BK0 и BK1 регистра ARM

Значение	Размер	Значение	Размер	Значение	Размер	Значение	Размер
00000	2	01000	512	10000	131 072	11000	33 554 432
00001	4	01001	1024	10001	262 144	11001	67 108 864
00010	8	01010	2048	10010	524 288	11010	67 108 864
00011	16	01011	4096	10011	1 048 576	11011	268 435 456
00100	32	01100	8192	10100	2 097 152	11100	536 870 912
00101	64	01101	16 384	10101	4 194 304	11101	1 073 741 824
00110	128	01110	32 768	10110	8 388 608	11110	2 147 483 648
00111	256	01111	65 536	10111	16 777 216	11111	4 294 967 296

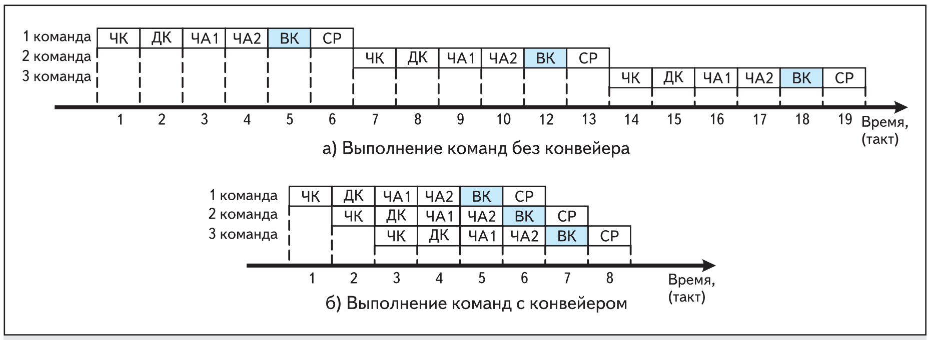

Рис. 2. Выполнение команд с конвейером и без него

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

; Определить режим циклической адресации для регистра A4
        MVC     .S2     AMR, B2
        MVKL    .S2     0x1, B1
        MVKLH   .S2     0x7, B1
        MVC     .S2     B1, AMR
; Сохранение служебных регистров в стеке
        STW     .D2T2   B0, *-SP[11]
        STW     .D2T2   B2, *-SP[12]
; Алгоритм обработки
        ; Инициализация переменных
        ZERO    .D1     summa_A
; summa = 0;
        ; Цикл вычисления свертки
        ; for (count = lenFir – 1; count >=0; count--){
        ; Определение начального значения переменной цикла
        SUB     .L2     lenFir_B, 1, count_B

loop01: ; Точка возврата при циклических вычислениях
        ; Чтение отсчета
        ; simpl = *pSimplBuff++;
        ; Чтение коэффициента
        ; coeff = *pCoeffBuff++;
        LDH     .D2T2   *pCoeffBuff_B++, coeff_B
        ; coeff = *pCoeffBuff++;
   ||   LDH     .D1T1   *pSimplBuff_A++, simpl_A
        ; simpl = *pSimplBuff++;

        NOP     ; Ожидание завершения операции чтения
                ; данных из памяти

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

        NOP     ; Ожидание завершения операции чтения
                ; данных из памяти

        NOP     ; Ожидание завершения операции чтения
                ; данных из памяти

        ; Умножение коэффициента на отсчет
        MPY     .M1X    coeff_B, simpl_A, mpy_A
        ; mpy = coeff * simpl;

        NOP     ; Ожидание завершения операции умножения

        ADD     .S1     summa_A, mpy_A, summa_A ;
         summa += mpy;

; }   Конец цикла вычисления свертки
        ; Нормирование результата суммирования
        SHR     .S1     summa_A, 15, summa_A
        ; sum>>=15;
; Восстановление служебных регистров
        LDW     .D2T2   *-SP[11], B0
        LDW     .D2T2   *-SP[12], B2
        NOP 4   ; Ожидание завершения последней операции
                ; чтения (4 такта)
; Восстановление регистров
        MVC     .S2     B0, CSR
        MVC     .S2     B2, AMR
; Выход из функции
        ; Перемещение возвращаемого значения в регистр A4
        MV      .S1     summa_A, A4
        B       .S2     adrReturn_B
        NOP 5   ; Ожидание завершения операции
                ; безусловного перехода (5 тактов)
```

Затем необходимо провести компиляцию проекта, загрузить полученный бинарный файл в ЦСП, запустить на выполнение программу и убедиться в корректности результата. Как это сделать, было подробно рассмотрено в предыдущих статьях цикла.

Особенности приведенного кода:

- команда ожидания завершения загрузки данных из памяти «*NOP* 4» заменена четырьмя последовательными командами «*NOP*» для удобства дальнейшей оптимизации;
- команда условного перехода «[*count_B*] *B .S2 loop*01» перенесена на 5 тактов вверх, что позволяет отказаться от команды ожидания завершения условного перехода «*NOP* 5» в конце цикла.

Дальнейшая оптимизация заключается в организации параллельных вычислений и конвейерной обработки команд. Суть конвейера в том, что очередная команда начинает выполняться до завершения предыдущей. Это возможно в силу того, что каждая команда выполняется за несколько тактов: чтение команды (ЧК), дешифрование команды (ДК), чтение операндов команды (ЧА1, ЧА2 и т. д.), выполнение команды (ВК) и сохранение результата (СР). Градация достаточно условная, но позволяет проиллюстрировать идею конвейера. На рис. 2а показана работа ЦСП без конвейера, а на рис. 2б — с конвейером. Таким образом, выполнение команд (после определенной задержки) происходит за один такт.

Для ЦСП TMS320C6000 существует два типа конвейеров — программный и аппаратный. Аппаратный конвейер «прозрачен» для программирования и позволяет считать, что большинство команд выполняется за один такт (имеет нулевую задержку). А вот программный конвейер является мощным инструментом оптимизации кода.

Реализация программного конвейера для функции свертки показана на примере блока «*Цикл вычисления свертки*» в ее ассемблерном коде. Примерная процедура организации конвейера заключается в следующем:

1. Вынести из цикла несколько итераций циклических вычислений и выполнить их последовательно. Количество выносимых итераций равно количеству тактов (включая пустые операции) в одном шаге цикла.

При этом из вынесенных итераций исключается команда условного перехода. Часть программного кода, соответствующего данному этапу, имеет вид:

```
  ; Цикл вычисления свертки
  ;for (count = lenFir – 1; count >=0; count--){
          ; Определение начального значения переменной цикла
          SUB     .L2     lenFir_B, 1, count_B

; 1-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
   ||     LDH     .D1T1   *pSimplBuff_A++, simpl_A

          NOP

  [count_B] SUB   .L2     count_B, 1, count_B

          NOP

          NOP

          MPY     .M1X    coeff_B,        simpl_A, mpy_A

          NOP

          ADD     .S1     summa_A, mpy_A, summa_A

; 2-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
  ||      LDH     .D1T1   *pSimplBuff_A++, simpl_A

          NOP

  [count_B] SUB   .L2     count_B, 1, count_B

          NOP

          NOP

          MPY     .M1X    coeff_B, simpl_A, mpy_A

          NOP

          ADD     .S1     summa_A, mpy_A, summa_A

; 3-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
; 4-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
; 5-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
; 6-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
; 7-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
; 8-я итерация цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
; Оставшиеся итерации цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
loop01:   ; Точка возврата при циклических вычислениях
          ; Чтение отсчета simpl = *pSimplBuff++;
          ; Чтение коэффициента coeff = *pCoeffBuff++;
          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
          ; coeff = *pCoeffBuff++;
  ||      LDH     .D1T1   *pSimplBuff_A++, simpl_A
          ; simpl = *pSimplBuff++;

          NOP   ; Ожидание завершения операции чтения
                ; данных из памяти

  [count_B] B     .S2     loop01
  || [count_B] SUB .L2    count_B, 1,count_B

          NOP   ; Ожидание завершения операции чтения
                ; данных из памяти

          NOP   ; Ожидание завершения операции чтения
                ; данных из памяти

          ; Умножение коэффициента на отсчет
          MPY     .M1X    coeff_B, simpl_A, mpy_A
          ; mpy = coeff * simpl;

          NOP   ; Ожидание завершения операции
                ; умножения

          ADD     .S1     summa_A, mpy_A, summa_A
          ; summa += mpy;
; }   Конец цикла вычисления свертки
```

Рекомендуется в учебных целях создать новый ассемблерный файл (например, с именем «*convolution_my_4.asm*») и сохранить в нем измененный код функции свертки.

Таблица 2. Пример «склеивания» первых трех итераций

1 итерация	2 итерация	3 итерация	Результат "склеивания"
LDH .D2T2 *pCoeffBuff_B++, coeff_B ‖ LDH .D1T1 *pSimplBuff_A++, simpl_A			LDH .D2T2 *pCoeffBuff_B++, coeff_B ‖ LDH .D1T1 *pSimplBuff_A++, simpl_A
NOP	LDH .D2T2 *pCoeffBuff_B++, coeff_B ‖ LDH .D1T1 *pSimplBuff_A++, simpl_A		LDH .D2T2 *pCoeffBuff_B++, coeff_B ‖ LDH .D1T1 *pSimplBuff_A++, simpl_A
[count_B] B .S2 loop01 ‖ [count_B] SUB .L2 count_B, 1,count_B	NOP	LDH .D2T2 *pCoeffBuff_B++, coeff_B ‖ LDH .D1T1 *pSimplBuff_A++, simpl_A	[count_B] B .S2 loop01 ‖ [count_B] SUB .L2 count_B, 1,count_B ‖ LDH .D2T2 *pCoeffBuff_B++, coeff_B ‖ LDH .D1T1 *pSimplBuff_A++, simpl_A
NOP	[count_B] B .S2 loop01 ‖ [count_B] SUB .L2 count_B, 1,count_B	NOP	[count_B] B .S2 loop01 ‖ [count_B] SUB .L2 count_B, 1,count_B
NOP	NOP	[count_B] B .S2 loop01 ‖ [count_B] SUB .L2 count_B, 1,count_B	[count_B] B .S2 loop01 ‖ [count_B] SUB .L2 count_B, 1,count_B
MPY .M1X coeff_B, simpl_A, mpy_A	NOP	NOP	MPY .M1X coeff_B, simpl_A, mpy_A
NOP	MPY .M1X coeff_B, simpl_A, mpy_A	NOP	MPY .M1X coeff_B, simpl_A, mpy_A
ADD .S1 summa_A, mpy_A, summa_A	NOP	MPY .M1X coeff_B, simpl_A, mpy_A	ADD .S1 summa_A, mpy_A, summa_A ‖ MPY .M1X coeff_B, simpl_A, mpy_A
	ADD .S1 summa_A, mpy_A, summa_A	NOP	ADD .S1 summa_A, mpy_A, summa_A
		ADD .S1 summa_A, mpy_A, summa_A	ADD .S1 summa_A, mpy_A, summa_A

2. Определить возможность организации параллельных вычислений. Это зависит от количества операций и их типа. За один такт можно выполнить 8 операций. Две из них должны быть умножениями. Не более двух операций чтения-записи памяти. Две операции необходимы для организации циклических вычислений. Следует проанализировать, какие модули АЛУ задействованы и нет ли пересечений — здесь важным является распределение переменных по сторонам *A* и *B* регистров общего назначения. Кроме этого, необходимо выяснить, нет ли конфликта в использовании путей коммутации и кроссировки. Пустые операции (NOP) не учитываются. В результате определяется количество блоков с параллельным выполнением команд, необходимых для выполнения всех действий на одном шаге цикла. Формировать блоки из операций цикла необходимо без учета того, что результат одной операции может быть исходным значением для другой.

В рассматриваемом примере можно выполнить все операции в одном блоке (параллельно): две операции чтения данных из памяти (пути коммутации не пересекаются), операция условного перехода, операция декрементации переменной цикла, одно умножение и одно суммирование (модули не пересекаются, используется только один кросс-путь).

3. Произвести «склейку» первых восьми (для данного примера) итераций цикла. С учетом того, что все операции в одном цикле могут быть выполнены в одном блоке параллельных команд, «склейка» производится со смещением на один шаг. Как это сделать, показано в таблице 2.

Первые 8 «склеенных» шагов цикла представляют собой программный конвейер вычислений: чтение из памяти новых данных начинается до окончания обработки уже прочитанных. В процессе загрузки текущих данных (которая выполняется с задержкой на 4 такта) производится обработка предыдущих. К моменту появления данных в приемных регистрах для текущего коэффициента фильтра и текущего отсчета линии задержки эти регистры оказываются свободны от предыдущих значений.

Результат данного этапа показан ниже. Рекомендуется эти изменения тоже сохранить в отдельном файле (например, с именем «*convolution_my_5.asm*»).

```
    ; Цикл вычисления свертки
    ;for (count = lenFir – 1; count >=0; count--){
                ; Определение начального значения переменной
цикла
                SUB     .L2     lenFir_B, 1, count_B
; 1–8 итерации цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;1
                LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;2
                LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;3
    [count_B] SUB       .L2     count_B, 1, count_B
    ||          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;4
    [count_B] SUB       .L2     count_B, 1, count_B
    ||          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;5
    [count_B] SUB       .L2     count_B, 1, count_B
    ||          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;6
                MPY     .M1X    coeff_B, simpl_A, mpy_A
    || [count_B] SUB    .L2     count_B, 1, count_B
    ||          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;7
                MPY     .M1X    coeff_B, simpl_A, mpy_A
    || [count_B] SUB    .L2     count_B, 1, count_B
    ||          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;8
                ADD     .S1     summa_A, mpy_A, summa_A
    ||          MPY     .M1X    coeff_B, simpl_A, mpy_A
    || [count_B] SUB    .L2     count_B, 1, count_B
    ||          LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;9
                ADD     .S1     summa_A, mpy_A, summa_A
    ||          MPY     .M1X    coeff_B, simpl_A, mpy_A
    || [count_B] SUB    .L2     count_B, 1, count_B
;10
                ADD     .S1     summa_A, mpy_A, summa_A
    ||          MPY     .M1X    coeff_B, simpl_A, mpy_A
    || [count_B] SUB    .L2     count_B, 1, count_B
;11
                ADD     .S1     summa_A, mpy_A, summa_A
    ||          MPY     .M1X    coeff_B, simpl_A, mpy_A
;12
                ADD     .S1     summa_A, mpy_A, summa_A
    ||          MPY     .M1X    coeff_B, simpl_A, mpy_A
;13
                ADD     .S1     summa_A, mpy_A, summa_A
    ||          MPY     .M1X    coeff_B, simpl_A, mpy_A
;14
                ADD     .S1     summa_A, mpy_A, summa_A
;15
                ADD     .S1     summa_A, mpy_A, summa_A

; Оставшиеся итерации цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
loop01:   ; Точка возврата при циклических вычислениях
                ; Чтение отсчета simpl = *pSimplBuff++;
                ; Чтение коэффициента coeff = *pCoeffBuff++;
                LDH     .D2T2   *pCoeffBuff_B++, coeff_B
                ; coeff = *pCoeffBuff++;
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
                ; simpl = *pSimplBuff++;

                NOP     ; Ожидание завершения операции чтения
                        ; данных из памяти

    [count_B] B         .S2     loop01
    || [count_B] SUB    .L2     count_B, 1,count_B

                NOP     ; Ожидание завершения операции чтения
                        ; данных из памяти

                NOP     ; Ожидание завершения операции чтения
                        ; данных из памяти

                ; Умножение коэффициента на отсчет
                MPY     .M1X    coeff_B, simpl_A, mpy_A
                ; mpy = coeff * simpl;

                NOP     ; Ожидание завершения операции умножения

                ADD     .S1     summa_A, mpy_A, summa_A
                ; summa += mpy;
; }   Конец цикла вычисления свертки
```

4. Организация циклических вычислений на базе конвейера. В блоке команд № 8 первых 8 итераций циклических вычислений (см. п. 3) выполняются все команды одного шага цикла (за исключением операции перехода). Этот блок будет ядром цикла с конвейером. Все блоки команд до него — это пролог цикла, после — эпилог.

Для организации циклических вычислений на базе конвейера необходимо (в рассматриваемом примере) исключить из программного кода блок команд «Оставшиеся итерации цикла» (см. п. 3). Затем установить метку для перехода при циклических вычислениях, а также в эпилоге и ядре цикла необходимо добавить команды перехода. Результат преобразования кода:

```
    ; Цикл вычисления свертки
    ;for (count = lenFir – 1; count >=0; count--){
                ; Определение начального значения переменной
цикла
                SUB     .L2     lenFir_B, 8, count_B
; Пролог цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;1
                LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;2
                LDH     .D2T2   *pCoeffBuff_B++, coeff_B
    ||          LDH     .D1T1   *pSimplBuff_A++, simpl_A
;3
                [count_B]       B       .S2     loop01
    || [count_B] SUB    .L2     count_B, 1, count_B
```

```
  ||          LDH  .D2T2  *pCoeffBuff_B++, coeff_B
  ||          LDH  .D1T1  *pSimplBuff_A++, simpl_A
;4
    [count_B] B    .S2    loop01
  || [count_B] SUB .L2    count_B, 1, count_B
  ||          LDH  .D2T2  *pCoeffBuff_B++, coeff_B
  ||          LDH  .D1T1  *pSimplBuff_A++, simpl_A
;4
    [count_B] B    .S2    loop01
  || [count_B] SUB .L2    count_B, 1, count_B
  ||          LDH  .D2T2  *pCoeffBuff_B++, coeff_B
  ||          LDH  .D1T1  *pSimplBuff_A++, simpl_A
;6
    [count_B] B    .S2    loop01
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
  || [count_B] SUB .L2    count_B, 1, count_B
  ||          LDH  .D2T2  *pCoeffBuff_B++, coeff_B
  ||          LDH  .D1T1  *pSimplBuff_A++, simpl_A
;7
    [count_B] B    .S2    loop01
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
  || [count_B] SUB .L2    count_B, 1, count_B
  ||          LDH  .D2T2  *pCoeffBuff_B++, coeff_B
  ||          LDH  .D1T1  *pSimplBuff_A++, simpl_A
; Ядро цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;8
loop01:
    [count_B] B    .S2    loop01
  ||          ADD  .S1    summa_A, mpy_A, summa_A
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
  || [count_B] SUB .L2    count_B, 1, count_B
  ||          LDH  .D2T2  *pCoeffBuff_B++, coeff_B
  ||          LDH  .D1T1  *pSimplBuff_A++, simpl_A

; Эпилог цикла ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;9
              ADD  .S1    summa_A, mpy_A, summa_A
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
  || [count_B] SUB .L2    count_B, 1, count_B
;10
              ADD  .S1    summa_A, mpy_A, summa_A
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
  || [count_B] SUB .L2    count_B, 1, count_B
;11
              ADD  .S1    summa_A, mpy_A, summa_A
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
;12
              ADD  .S1    summa_A, mpy_A, summa_A
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
;13
              ADD  .S1    summa_A, mpy_A, summa_A
  ||          MPY  .M1X   coeff_B, simpl_A, mpy_A
;14
              ADD  .S1    summa_A, mpy_A, summa_A
;15
              ADD  .S1    summa_A, mpy_A, summa_A
; }  Конец цикла вычисления свертки
```

Рекомендуется сохранить данный вариант ассемблерного кода в отдельном файле (например, с именем «*convolution_my_6.asm*»).

В 8-м блоке команд появилась метка «*loop*01» для организации цикла, а также команда условного перехода на эту метку ([*count_B*] *B .S2 loop*01), выполняемая параллельно с остальными командами блока. Данная команда будет выполнена только через 5 тактов. Поэтому необходимо за 5 тактов до блока команд ядра поставить первую операцию условного перехода (блок команд номер 3) — эта команда перехода будет выполнена только после 8-го блока команд. Команда перехода в 4-м блоке команд будет выполнена после того, как 8-й блок будет повторен дважды. Затем будет выполнена команда перехода 5-го блока, затем 6-го, 7-го, и только после этого начнет выполняться команда перехода ядра цикла.

Задержка выполнения операции условного перехода приводит к тому, что после запрета на выполнение условного перехода в ядре цикла (значение регистра условия «*count_B*» равно нулю), переход будет осуществлен еще 5 раз (будут выполняться предыдущие команды перехода). Это влечет необходимость, во-первых, уменьшить значение переменной цикла «*count_B*» еще на 7 единиц:

SUB .L2 lenFir_A, 8, count_B,

во-вторых, заблокировать декрементацию переменной цикла после достижения значения «ноль»:

[count_B] SUB .L2 count_B, 1, count_B.

Уменьшение переменной цикла на 7 единиц обусловлено тем, что выполнение 7 итераций происходит вне ядра цикла (в прологе и эпилоге).

Откомпилировать проект, запустить его на выполнение и убедиться в корректной работе. Таким образом, получен программный код циклических вычислений, оптимизированный с использованием конвейерной и параллельной обработки команд.

В следующей статье будет показано, как использовать возможности RTDX для ускорения и визуализации процесса отладки, а также контроля времени выполнения программного кода. Все файлы проекта можно найти на сайте w ww .scanti. ru.