Применение платформы ADSP-TS201 для решения задач в реальном масштабе времени: методики оптимизации программного кода

Александр Тумачек (Москва)

Рассмотрены методики и способы программирования ЦПОС фирмы Analog Devices TS201. Описанные подходы к оптимизации программного кода могут быть перенесены на любой процессор с аналогичной архитектурой.

Цифровая обработка сигналов для решения задач в реальном масштабе времени

На современном этапе развития вычислительной техники популярность набирают системы обработки данных в реальном масштабе времени. Построение подобных систем требует высокопроизводительных вычислительных средств. В задачах потоковой обработки данных, критичных к временным задержкам, необходимо обрабатывать данные в ограниченные интервалы времени.

Иногда решение задачи в реальном масштабе времени отождествляют с функционированием «быстрой системы», но это не всегда правильно, поскольку важно не время задержки реакции системы, а то, насколько гарантировано его наличие для работы. Например, если при обработке звуковых данных требуется 2,01 с на анализ звука длительностью 2,00 с, это не будет считаться процессом реального времени. Если же требуется 1,99 с – это процесс реального времени. Для решения задач в реальном масштабе времени могут применяться как процессоры общего назначения, так и цифровые сигнальные процессоры.

При создании средств ЦОС иногда используется подход, в котором для каждого алгоритма специально разрабатывается инструментальное решение, аппаратная часть которого составляется из адаптированных к алгоритму микросхем. Это своего рода составление мозаики из интегральных элементов, адаптированной под алгоритм. Такая реализация оказывается наиболее быстродействующей, но не универсальной, поскольку невозможна унификация элементной базы. Например, перепрограммировать аппаратный КИХ-фильтр мы уже не сможем, следовательно, при частичном изменении структуры алгоритма потребуется замена соответствующей ИС.

Современные тенденции развития процессорной индустрии и систем на кристалле сводятся к разработке универсальных элементов низкой себестоимости. Достаточное количество таких элементов позволяет реализовать алгоритм любой сложности: чем выше сложность алгоритма, тем больше элементов необходимо задействовать.

Рассмотрим базовые элементы вычислительных средств, применяемые при обработке сигналов. Как правило, это процессоры с суперскалярной и конвейерной архитектурами, банки памяти, блоки коммутации, широкие шины данных, – всё это составляет архитектуру вычислительного средства. Удобная архитектура позволяет интуитивно осуществлять процесс программирования, адаптируя код программы к инструментальному средству.

Для обработки данных в реальном масштабе времени необходимы не только эффективные вычислительные средства, но и оптимизированные программные алгоритмы. Зачастую их правильная реализация позволяет решить задачу с применением недорогих инструментальных средств.

Рассматривая различные технологии программирования, можно отметить тенденцию поиска эффективных алгоритмов для задач реального времени. Об этих методиках и способах программирования применительно к сигнальному процессору фирмы Analog Devices ADSP-TS201 и пойдёт речь в этой статье.

Приведённые ниже методики программирования ЦПОС могут быть перенесены на любой процессор с аналогичной архитектурой. Рассмотрим пути оптимизации кода для процессоров семейства Tiger Sharc.

Характеристики процессора Tiger Sharc, влияющие на процесс программирования

Для начала выделим основные параметры архитектуры сигнального процессора. Эти данные позволяют уже на этапе проектирования программы правильно организовать работу вычислительного средства и избежать необоснованных затрат вычислительных ресурсов.

Процессор ADSP-TS201 семейства Tiger Sharc имеет двойной вычислительный блок, позволяющий реализовывать набор SIMD-инструкций. Статическая суперскалярная архитектура предоставляет возможность выполнять несколько математических операций за один процессорный цикл (до четырёх инструкций за цикл). Другие особенности процессора:

- операции умножения могут производиться одновременно (до восьми 16-битных операций умножения в формате с фиксированной точкой) и реализуются посредством векторных вычислений;
- память организована в шесть блоков по 4 Мбит каждый; блок содержит 128 килослов при 32 битах;
- четыре 128-битных шины данных;

- каждый блок памяти соединён посредством перекрёстной шины с 128-Кбит буфером. Такая организация памяти позволяет передавать до четырёх 32-бит операндов одновременно.

Для эффективного выполнения программ, учитывая особенности архитектуры процессора ADSP-TS201, необходимо:

- размещать инструкции кода отдельно от данных. Это позволяет использовать один блок памяти для программ, а два других блока памяти – для размещения данных;
- рекомендуется назначать каждому блоку памяти отдельный адресный регистр, что исключит пропуски циклов и перезагрузку конвейера;
- необходимо чётко понимать структуру памяти и принципы работы кэша, соблюдать локальную политику видимости при программировании и условие последовательного доступа к данным.

Для всех последующих примеров программ данные и код корректно размещены в памяти и кэше, буфер предсказания ветвлений (BTB – branch target buffer) считается загруженным. Ветвления и переходы при выполнении кода требуют только один цикл процессора и выполняются без перезагрузки конвейера.

Оптимизация стандартной процедуры перемножения с накоплением

Рассмотрим ассемблерный код, реализующий 32-разрядную операцию перемножения с накоплением, и обсудим пути его оптимизации.

Для начала необходимо уточнить, в каких случаях могут произойти пропуски процессорных циклов, как оптимизировать выполнение критических циклов и эффективно запрограммировать процессорный конвейер, исключив его перезагрузку.

Реализуемая операция:

$$x = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_1 b_1 \ldots . a_n b_n$$

Пример 1. Простой алгоритм операции перемножения с накоплением (см. рис. 1)

```
#define VECTOR_LENGTH 200
j0 = j31 + A; LC0 =
VECTOR_LENGTH;;
k0 = k31 + B;;
```

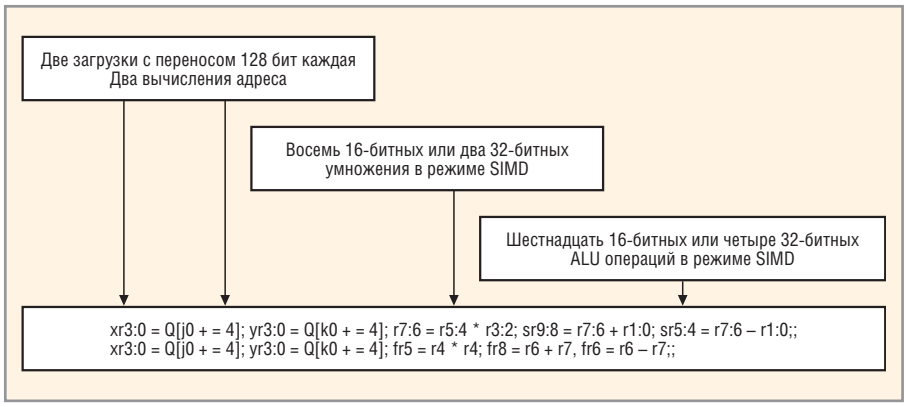

Рис. 1. Оптимальная строка инструкций процессора ADSP-TS201, выполняемая за один такт

```
xr3 = 0.0;;
.align_code 4;
Dot_Product:
xr0 = [j0 +=1];;
xr1 = [k0 +=1];;
xfr2 = r0 * r1;;
xfr3 = r2 + r3;;
align_code 4;
if NLC0E, jump Dot_Product;;
[j31 + Result] = XR3;;//Число
циклов = 1408
```

Замечание: на процессорах TS20x инструкции перехода в цикле выполняются без пропусков циклов и перезагрузки конвейера.

Рассмотрим недостатки примера 1. В процедуре задействован только один вычислительный блок, хотя имеется возможность использовать оба блока – CBX и CBY; используется всего одна процессорная инструкция в строке; данные загружаются по частям при проходе цикла; конфликт блоков памяти из-за неправильного размещения векторов данных приводит к пропуску цикла. Вывод: код не эффективен.

Рассмотрим архитектурные возможности процессора ADSP-TS201, позволяющие улучшить пример 1. В процессоре TS201 реализованы шесть блоков памяти, которые позволяют извлекать два операнда параллельно с извлечением командной инструкции.

Оптимальная строка инструкций процессора, выполняемая за один такт, может содержать 24 16-битных операции или шесть 32-битных операций; восемь перемножений с накоплением в секунду с 16-битными данными или два перемножения с накоплением в секунду с 32-битными данными; два 128-битных перемещения данных и два расчёта адресов.

Поэтапная оптимизация процедуры перемножения с накоплением

Рекомендации по улучшению программного кода

Быстрый и простой способ улучшить производительность алгоритма – загружать данные по максимуму, за один процессорный цикл; использовать загрузку одного длинного слова или по четыре слова; используя обе 128-разрядных шины. Также эффективно применение различных блоков адресных регистров (IALU – Integer Arithmetic Logic Unit) при загрузке данных. Необходимо задействовать оба блока для осуществления параллельной обработки и по возможности комбинировать несколько инструкций в одной строке.

Оптимизируем пример 1:

- задействуем два вычислительных блока для каждой математической операции;
- используем способ загрузки длинным словом, т.е. две части операндов одновременно.

Пример 2. Оптимизация примера 1

```
j0 = j31 + A; LC0 =
VECTOR_LENGTH/2;;
k0 = k31 + B;;
r3 = 0.0;;
.align_code 4;
Dot_Product:
xyr0 = [j0 +=2];;    //
xyr1 = [k0 +=2];;
xyfr2 = r0 * r1;;    //SIMD режим
xyfr3 = r3 + r2;;    //SIMD режим
align_code 4;
if NLC0E, jump Dot_Product;;
xr4 = yr3;;
xfr5 = r3 + r4;;
[j31 + Result] = XR5;;//Число
циклов = 711
```

В результате проведённой оптимизации число циклов, затрачиваемых

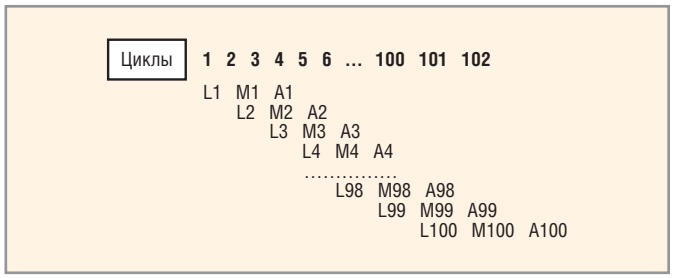

Рис. 2. Этапы работы программного конвейера ADSP-TS201

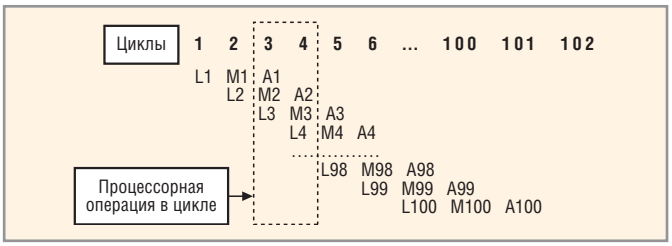

Рис. 3. Оптимизация поэтапной загрузки программного конвейера ADSP-TS201

на выполнение алгоритма, сократилось в два раза.

Использование режима SIMD

Рассмотрим другие методики оптимизации кода при работе с данными в форматах с плавающей и 32-разрядной фиксированной точкой.

Применим методику разворачивания вычислительного цикла. Распараллелим вычислительный процесс алгоритмически и задействуем режим SIMD (Single-Instruction, Multiple-Data) процессора ADSP-TS201. Перестроим выполняемую инструкцию таким образом, чтобы устранить пропуски процессорных циклов. Учтём работу программного конвейера для правильного формирования инструкций процессору.

Пример 3. Исходный цикл

```
j0 = j31 +A;;
LC0 = VECTOR_LENGTH;;
k0 = k31 + B;;
xr3 = 0.0;;
.align_code 4;
Dot_Product:
        xr0 = [j0 +=1];;
        xr1 = [k0 +=1];;
        xfr2 = r0 * r1;; //stall
        xfr3 = r3 + r2;;
.align_code 4;
if NLC0E, jump Dot_Product;;
[j31 + Result] = xr3;;
                //число циклов 1408
```

Пример 4. Развёрнутый цикл

```
J0 = j31 +A; LC0 =
VECTOR_LENGTH/2;;
k0 = k31 + B;;
xr3 = 0.0;;
xr19 = 0.0;;
.align_code 4;
Dot_Product:
        xr0 = [j0 +=1];; xr16 =
[j0 +=1];;
        xr1 = [k0 +=1];; xr17 =
[k0 +=1];;
        xfr2 = r0 * r1;; xfr18 =
r16 * r17;;
        xfr3 = r3 + r2;; xfr19 =
r19 + r18;;
.align_code 4;
if NLC0E, jump Dot_Product;;
xfr3 = r3 + r19;;
[j31 + Result] = xr3;;
                //число циклов 910
```

Оптимизация с использованием программного конвейера

Программный конвейер процессора помогает осуществлять выполнение инструкций наиболее эффективно, учитывая особенности его архитектуры. Инструкции выполняются таким образом, чтобы каждый этап программного цикла состоял из нескольких инструкций; процедура перемножения с накоплением основана на выполнении трёх базовых операций: загрузка, перемножение, накопление.

Рассмотрим исходный код, реализующий процедуру перемножения с накоплением, с учётом работы программного конвейера.

Пример 5. Задействуем программный конвейер (см. рис. 2)

```
j0 = j31 + A;;
k0 = k31 + B;;
xr3 = 0.0;;
LC0 = VECTOR_LENGTH - 2;;
xr0 = [j0 +=1]; xr1 = [k0 +=1];;
xr0 = [j0 +=1]; xr1 = [k0 +=1];
xfr2 = r0 * r1;;
.align_code 4;
Dot_Product:
xr0 = [j0 +=1]; xr1 = [k0 +=1];
xfr2 = r0 * r1; xfr3 = r3 + r2;;
.align_code 4;
if NLC0E, jump Dot_Product;;
xfr2 = r0 * r1;        xfr3 = r3
+ r2;;
xfr3 = r3 + r2;;
[j31 + Result] = xr3;;   //число
циклов 411
```

Обратим внимание на построение цикла: он состоит только из одной инструкции перехода *JUMP* – дополнительной инструкции цикла, вследствие которой теряется один процессорный цикл. Удаление дополнительной инструкции приводит к простою процессора. Метка перехода инструкции *JUMP* – это адресная структура, которая может пересекать границу памяти, кратную четырём словам. Если метки перехода по программе в памяти не соответствуют кратности адреса в четыре слова, то возникают дополнительные пропуски процессорных циклов. Необходимое расположение меток в памяти может быть указанно директивой *.allign_code 4* или вставкой операций *NOP* для выравнивания инструкций.

Рассмотрим ещё один вариант реализации операции перемножения с накоплением. Можно оптимизировать загрузку данных из памяти, комбинируя операции извлечения сразу по четыре операнда.

Пример 6. Одновременная загрузка четырёх операндов из памяти (см. рис. 3)

```
j0 = j31 + A;;
k0 = k31 + B;;
xr6 = 0.0;;
LC0 = (VECTOR_LENGTH - 2)/2;;
xr1:0 = L[j0 +=2]; xr3:2 = L[k0
+=2];;
xfr4 = r0 * r2;;
.align_code 4;
Dot_Product:
xr1:0 = L[j0 +=2]; xr3:2 = L[k0
+=2]; xfr5 = r1 * r3; xfr6 = r4
+ r6;;
.align_code 4;
if NLC0E, jump Dot_Product; xfr4
= r0 * r2; xfr6 = r5 + r6;;
xfr5 = r1 * r3; xfr6 = r4 + r6;;
xfr6 = r5 + r6;;
[j31 + Result] = xr6;;  //число
циклов 408
```

Процессорный цикл в этом случае совмещает в себе две инструкции. Переход осуществляется параллельно другим арифметическим операциям в строке. Извлечение данных из памя-

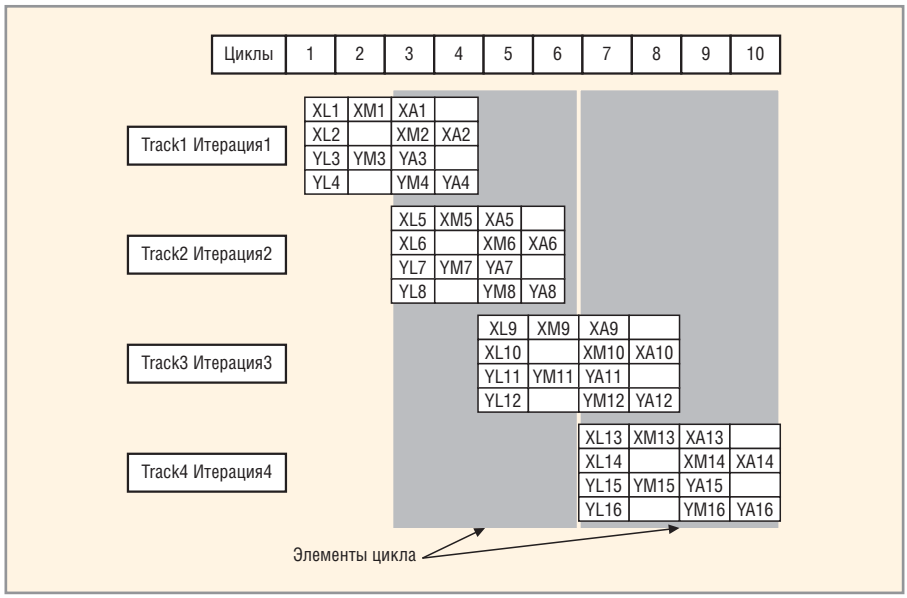

Рис. 4. Распараллеливание вычислительной задачи на процессорные блоки X и Y

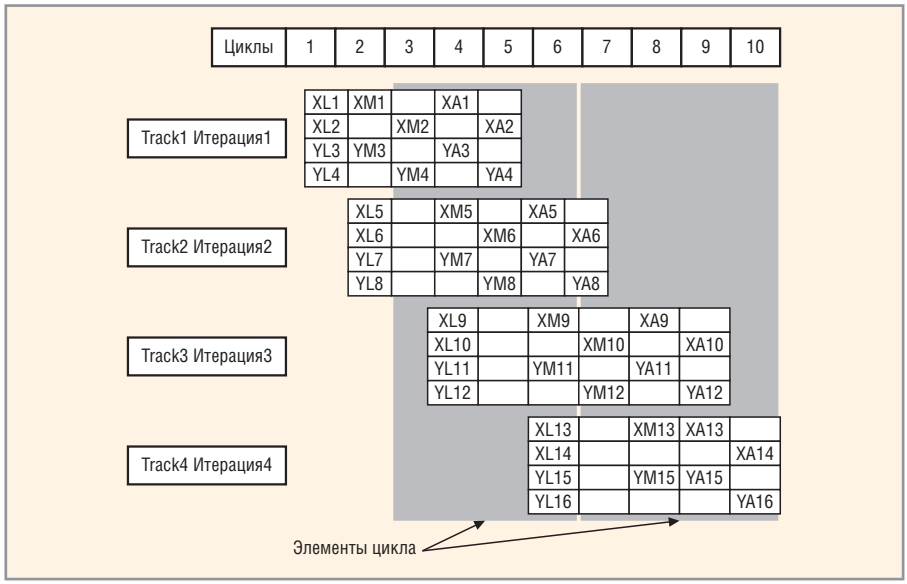

Рис. 5. Распараллеливание вычислительной задачи на процессорные блоки X и Y с учётом работы программного конвейера

ти происходит только за один цикл (половину цикла), оставляя шину свободной для других операций, например, ввода/вывода, прямого доступа к памяти или команды сброса.

В этой реализации возможен простой процессора между первой и последней инструкцией цикла. Это несущественно по сравнению с затрачиваемым числом циклов на выполнение процедуры. Таким образом, для использования возможностей платформы для выполнения операции перемножения с накоплением необходимо полностью задействовать программный конвейер ADSP-TS201 и использовать оба ядра процессора.

В предыдущем примере процедуры перемножения с накоплением использовали оба вычислительных блока (X и Y). Данные с одного вычислительного блока просто копировались в другой.

Оптимизация с учётом независимой работы вычислительных блоков процессора

Рассмотрим пример, в котором мощности вычислительных блоков задействуются для независимых арифметических операций.

В данном примере мы видим, что число операций в процессорных циклах сократилось почти вдвое, но каждая инструкция содержит пропуск цикла. Необходимо чередовать инструкции с пропусками циклов через одну, чтобы добиться наиболее эффективного выполнения процедуры.

Пример 7. Распараллеливание задачи на процессорные блоки X и Y (см. рис. 4)

```
j0 = j31 + A; LC0 =
VECTOR_LENGTH/8-1;;
k0 = k31 + B;;
xyr21:20 = r21:20 - r21:20;;
r1:0 = Q[j0 +=4]; r17:16 = Q[k0
+=4];;
xyfr3 = r0 * r16;;//stall

.align_code 4;
Dot_Product6:
xyfr21 = r21 + r3; xyfr4 = r1 *
r17; r11:10 = Q[j0 +=4]; r27:26
= Q[k0 +=4];;//stall
xyfr20 = r20 + r4; xyfr3 = r10 *
r26;;//stall
xyfr21 = r21 + r3; xyfr4 = r11 *
r27; r1:0 = Q[j0 +=4]; r17:16 =
Q[k0 +=4];;//stall
if NLC0E, jump Dot_Product6;
xyfr20 = r20 + r4; xyfr3 = r0 *
r16;;//stall

xyfr21 = r21 + r3; xyfr4 = r1 *
r17; r11:10 = Q[j0 +=4]; r27:26
= Q[k0 +=4];;//stall
xyfr20 = r20 + r4; xyfr3 = r10 *
r26;;//stall
xyfr21 = r21 + r3; xyfr4 = r11 *
r27;;//stall
xyfr20 = r20 + r4;;    //stall
xyfr21 = r21 + r20;;//stall
xr22 = yr21;;          //stall
xfr25 = r21 + r22;;
[j31 + Result] = xr25;;
//214 Циклов
```

В примере 7 мы видим, что число операций в процессорных циклах сократилось почти вдвое, но каждая инструкция содержит пропуск цикла.

Пример 8. Распараллеливание задачи на процессорные блоки X и Y с учётом работы программного конвейера и описанных методик оптимизации (см. рис. 5)

```
j0 = j31 + A; LC0 =
VECTOR_LENGTH/8-1;;
k0 = k31 + B;;
r21:20 = r21:20 - r21:20;;
yxr1:0 = Q[j0 +=4]; yxr17:16 =
Q[k0 +=4];;
fr3 = r0 * r16; yxr11:10 = Q[j0
+=4]; yxr27:26 = Q[k0 +=4];;
fr4 = r1 * r17;;
.align_code 4;
Dot_Product5:
        fr21 = r21 + r3; fr3 =
```

```
r10 * r26; yxr1:0 = Q[j0 +=4];
yxr17:16 = Q[k0 +=4];;
        fr20 = r20 + r4; fr4 =
r11 * r27;;
        fr21 = r21 + r3; fr3 = r0
* r16; yxr11:10 = Q[j0 +=4];
yxr27:26 = Q[k0 +=4];;
.align_code 4;
if NLC0E, jump Dot_Product5;
fr20 = r20 + r4; fr4 = r1 * r17;;
fr21 = r21 + r3; fr3 = r10 * r26;;
fr20 = r20 + r4; fr4 = r11 * r27;;
fr21 = r21 + r3;;
fr20 = r20 + r4;;//stall
fr21 = к21 + к20;; //stall
xr22 = yr21;; //stall
xfr25 = r21 + r22;;
[j31 + Result] = xr25;; //117
циклов
```

Цикл развёрнут дважды. В процессе выполнения программы не происходит ни одного пропуска процессорного цикла. Число циклов процессора уменьшилось с 1408 до минимума в 117, но при этом увеличился исходный код программы.

Резюмируя анализ примеров, можно отметить следующее. На первоначальном этапе необходимо выявить слабые стороны алгоритма (критические циклы). Программа работает эффективно и производительно, если она хорошо продумана и адаптирована к архитектуре вычислительного средства. Разрабатываемый алгоритм должен быть производительным и ёмким. Необходимо проверить совместимость поколений процессоров и проблему псевдонимов, а также убедиться, что имена и переменные не конфликтуют с зарезервированными именами. Могут возникать неявные ошибки, локализовать которые впоследствии будет очень сложно.

Необходимо обратить внимание на инструментальные возможности платформы и обратную совместимость кода. Рано или поздно вам придётся совершенствовать или переделывать проект. Возможно применение специальных инструкций. Составьте алгоритм таким образом, чтобы он был встроен в основной программный код и не вызывал дополнительных перезагрузок конвейера, лишних переходов. Если процедура достаточно громоздкая, лучше оформить её в качестве подпрограммы.

Заключение

Для наиболее эффективного выполнения типичных ассемблерных процедур, в том числе и процедуры перемножения с накоплением, необходимо использовать все известные методики оптимизации кода и учитывать архитектуру процессора ADSP-TS201. Ключевыми моментами являются программирование конвейера и создание параллельных процедур вычислений. Важно избегать пропусков процессорных циклов и перезагрузки конвейера. Только использование всех доступных методик оптимизации кода и детальное знание архитектуры процессора позволяют добиться максимальной производительности ЦПОС.

Литература

1. *Кублановская В.Н.* Первая публикация по QR-алгоритму в Дополнении к изданию 1960 г. монографии Д.К. и В.Н. Фадеевых «Вычислительные методы и линейная алгебра», Примечание первое.
2. www.analog.com.
3. Optimization Techniques for Tiger SHARC, ADI-TS20x-SEC12-OPT-V1.0.