Продолжение. Начало в № 3`2007

Микропроцессор своими руками—5.
По поводу начала проекта встроенного в FPGA микроконтроллера

Иосиф КАРШЕНБОЙМ
losifk@narod.ru

Начнем с производительности

Представим, что нам не хватает производительности микроконтроллера. А как мы это оцениваем? Обычно говорят: «Тактовая частота, MIPS, FLOPS и так далее». Но давайте остановимся на этом пункте, а именно на рассмотрении производительности, более подробно.

Имеет смысл сосредоточиться не столько на вычислительных «способностях» микроконтроллера, но в первую очередь на выполнении им задач пользователя. Представим себе вот такую задачу. В разделе этой статьи о том, как надо разрабатывать ТЗ, автор приводил пример тех систем управления (СУ), в разработке которых он лично принимал участие. Каждая из таких СУ работала, в том числе, и с дискретными датчиками. Число датчиков варьировалось от многих сотен до тысячи. Как была построена обработка таких датчиков? Обсудим два вопроса: как выполнять обслуживание датчиков и как была увеличена производительность в тех СУ. А также рассмотрим, что может быть получено в случае применения нестандартного решения в FPGA. Конечно, приведенный здесь пример — это не более чем «учебно-боевая стрельба». Конечно, это очень простая задача, и она неоднократно описана и решена. И именно поэтому, в качестве примера, и была выбрана тема, понятная большинству разработчиков, даже начинающих. Потому что главной целью данного примера является не сам проект или способ обработки датчиков. Основная цель — это объяснить, как от задачи надо перейти к путям ее решения, и показать методику оптимизации проектирования контроллера и его программного обеспечения.

Возвращаемся к СУ. Машина, контроллер дискретного канала управления, вводила информацию о датчиках и заносила ее во входную таблицу. При этом данные заносились в таблицу с «привязкой» к тем физическим адресам, которые были присущи каждой из СУ. Затем выполнялась перекодировка: данные из этой входной таблицы переписывались в системную таблицу. Такое построение обработки позволило упростить «привязку» алгоритмов к различным системам с разными физическими адресами датчиков. Далее данные уже обрабатывались как системные, то есть при обработке одинаковых алгоритмов на разных СУ оперировали с одинаковыми системными адресами. Дискретные данные из системной таблицы пересылались в машину, контроллер логического канала управления. Эта машина циклически считывала данные уже из своей системной таблицы дискретных данных и формировала в своей памяти ответные воздействия. Ответные воздействия также пересылались в машину, контроллер дискретного канала управления, перекодировались в физические адреса и отправлялись на выходное устройство управления. Давайте оценим выполнение этого примера на разных аппаратных платформах. С точки зрения ASIC-микроконтроллеров все просто. Ставим пару микропроцессоров, и готово. Или один, но большей производительности.

Вроде пример довольно простой. Сначала просто пишем данные из входного порта в память. Пишем циклически, не останавливаясь, поскольку задержка во вводе данных увеличивает время реакции того канала управления объекта, который мы проектируем. Допустим, мы хотим обработать 1000 датчиков и на ввод каждого датчика тратим по 1 циклу ввода. Итого получаем 1000 циклов. Как мы знаем, наша машина будет работать, циклически повторяя заложенные в нее алгоритмы. Вот и давайте введем термин «циклограмма». Под этим термином мы будем понимать время выполнения полного цикла управляющей программы машины. Тогда для этого — первого варианта рассмотрения задачи, когда все датчики обслуживались поочередно, мы получим циклограмму, приведенную на рис. 1.

Рис. 1. Циклограмма поочередного обслуживания датчиков

Но давайте зададим себе вопрос: все ли датчики для нас одинаково «дороги»? Для сравнения, датчик «аварийного превышения давления» и датчик «двери стоек закрыты». Конечно, датчик «двери стоек закрыты» тоже нужен и дорог, особенно для больших систем. Но информация, которую он дает — это всего лишь служебная информация, которая слабо влияет на динамику работы СУ. Да и в задание по времени реакции СУ такие датчики обычно не входят. Тогда меняем алгоритм обслуживания датчиков. Выделяем из всего массива датчиков, например, 4 группы:

- «аварийные» датчики — 10% (100);
- «быстрые» — 20% (200);
- «медленные» — 60% (600);
- «фоновые» — 10% (100).

Теперь задачу сформулируем по-другому. Насколько редко можно проверять «фоновые» датчики? Насколько часто мы должны проверять «аварийные» датчики? Пропустим здесь все рассуждения и представим, что для того, чтобы выполнить ТЗ, мы хотим иметь вот такой цикл по обслуживанию датчиков (рис. 2). При этом полная циклограмма будет иметь соответственно 100 таких частичных циклов. При использовании варианта № 2 полная циклограмма будет иметь соответственно 100 частичных циклов: «аварийные» (100) + «быстрые» (200) + «аварийные» (100) + «медленные» (199) + «фоновые» (1).

Что мы получим? Один частичный цикл — 100 + 200 + 100 + 199 + 1 = 600 тактов.

Время на обслуживание датчиков:

- для «аварийных» — 600/2 = 300 тактов;
- для «быстрых» — 600 тактов;
- для «медленных» — (600/199)×600 = 1800–2400 тактов;
- для «фоновых» — 600×100 = 60 000 тактов.

Теперь из рассмотрения таблицы 1 мы видим, что у нас есть возможность выполнить задание по «быстрым» и «аварийным» дат-

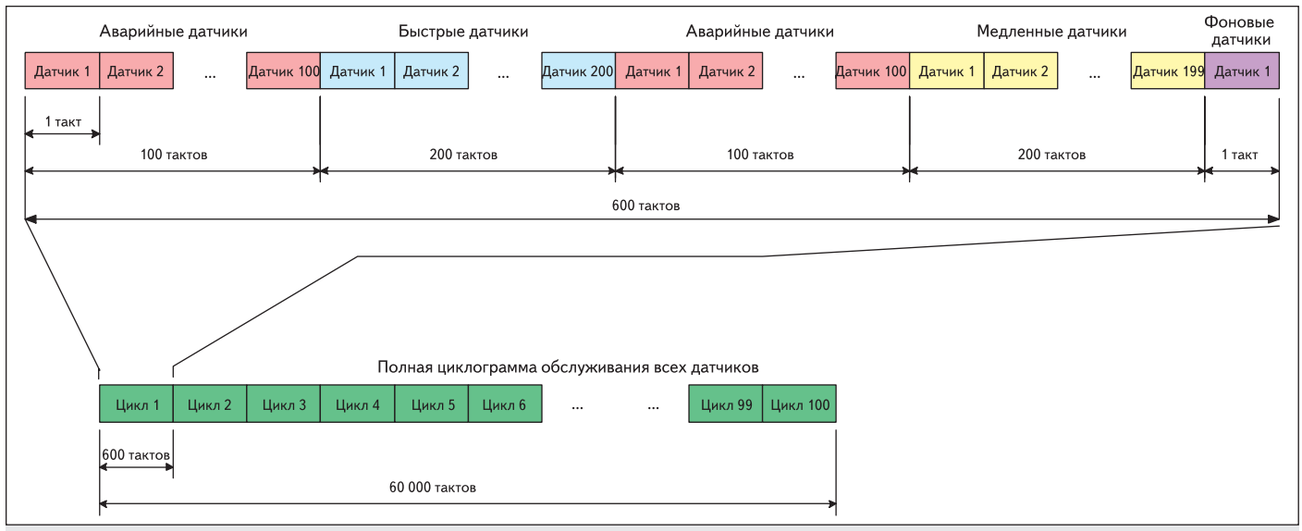

Рис. 2. Цикл по обслуживанию датчиков

Таблица 1. Сравнение вариантов обслуживания датчиков

Тип датчика	Вариант 1	Вариант 2	Ускорение
«аварийный»	1000	300	3,3
«быстрый»	1000	600	1,7
«медленный»	1000	1200	0,83
«фоновый»	1000	60000	0,017

чикам примерно в два раза быстрее! «А можно ли сделать устройство управления еще быстрее, и причем тут FPGA?» — спросит нетерпеливый читатель. Да, сделать устройство управления в два раза быстрее можно! Здесь мы не имеем в виду то, что можно взять микроконтроллер с более высокой тактовой частотой. Представим себе, что время цикла ограничено, например физическими параметрами тракта ввода, допустим, временем работы оптрона. Как надо поступать в таком случае? Как было сказано в статье [1], для того чтобы увеличить производительность, надо «запрячь больше лошадей». Если мы используем ASIC-микроконтроллеры, то это значит, надо добавить еще один микроконтроллер и организовать канал связи между ними, увеличить площадь PCB. Это потребует дополнительных затрат. А при использовании FPGA надо вместо однопортовой памяти применить двухпортовую и сделать еще один канал загрузки данных от датчиков в память. Затраты в FPGA — ничтожные, а результат достигается большой. Покажем это на примере. Давайте сформируем вот такие циклы:

Первый канал: «аварийные» (50) + «медленные» (49) + «фоновые» (1).

Второй канал: «аварийные» (50) + «быстрые» (50). Пример такой диаграммы работы приведен на рис. 3.

При использовании варианта № 3 полная циклограмма будет иметь соответственно 100 частичных циклов. Контроллер будет работать одновременно двумя каналами.

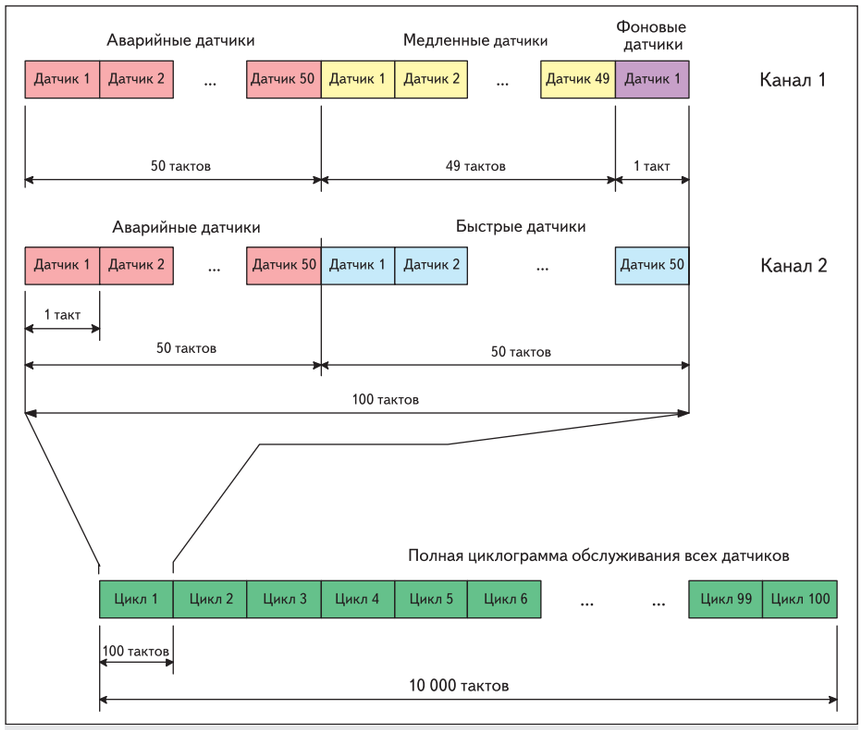

Рис. 3. Диаграмма работы по обслуживанию датчиков

- Первый канал: «аварийные» (50) + «медленные» (49) + «фоновые» (1).
- Второй канал: «аварийные» (50) + «быстрые» (50).

Что мы получим? Один частичный цикл — 50 + 50 = 100 тактов. Время на обслуживание датчиков:

- для «аварийных» — 50 + 50 = 100 тактов,
- для «быстрых» — (200/50)×100 = 400 тактов,
- для «медленных» — (600/49)×100 = 1200–1300 тактов;
- для «фоновых» — 100×100 = 10 000 тактов.

Итак, рассмотрим результат разработки проекта относительно варианта № 1 (таблица 2). Что касается «фоновых» датчиков, то замедление их опроса в 10 раз никак не влияет на ухудшение работоспособности узла управления. По «медленным» датчикам наш узел управления тоже работать хуже не стал — разница всего 23%. А вот по «быстрым» датчикам — улучшение в 2,5 раза, и по «аварийным» датчикам — в 10 раз! Это уже значительно лучше!

А теперь представим, что среди аварийных датчиков есть пара таких, которые можно

Таблица 2. Сравнение вариантов обслуживания датчиков

Тип датчика	Вариант 1	Вариант 2	Вариант 3	Ускорение — вариант 2	Ускорение — вариант 3
«аварийный»	1000	300	100	3,3	10
«быстрый»	1000	600	400	1,7	2,5
«медленный»	1000	1200	1300	0,83	0,78
«фоновый»	1000	60 000	10 000	0,017	0,1

Примечание:
1. В столбцах «Вариант» записано число тактов, необходимое для обслуживания датчиков.
2. В столбцах «Ускорение» записан коэффициент ускорения процесса относительно варианта № 1.

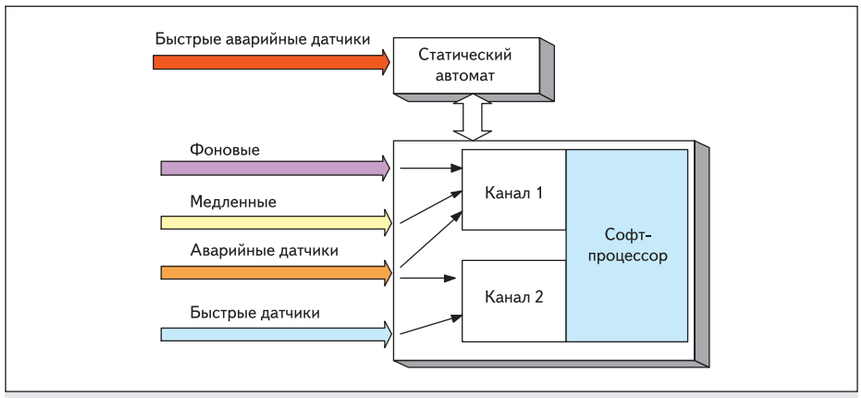

Рис. 4. Модернизация варианта № 3

определить примерно так — «погибаю, но не сдаюсь». Для таких датчиков обычно нужна немедленная реакция, и для них, возможно, те 100 циклов, за которые осуществляется ввод информации от остальных аварийных датчиков, будет слишком большим периодом времени. Что делать в этом случае? Давайте еще раз произведем разделение алгоритма, по которому мы производим синтез управляющего узла. Выделим эти датчики в отдельный статический автомат (рис. 4). Часть датчиков аварийного канала выделена в отдельную группу. Для обработки этих датчиков применен отдельный статический автомат.

Возможно, в этот же автомат придется ввести и несколько сигналов, которые будут представлять собой свертку состояния от части остальных сигналов, тех, что нужны для формирования алгоритма быстрого канала аварийного управления. Тогда результат может выглядеть так, как показано в таблице 3.

Но это тоже только первое приближение. Это, можно сказать, только вершина айсберга.

До сих пор мы делали расчеты из тех соображений, что нам нужны показания всех датчиков. Но ведь на самом деле это не всегда так. Представьте, что ваш объект управления — самолет. Пока он стоит на стоянке, а затем взлетает, информация о его шасси — колесах и шинах — может быть опрошена в составе группы «медленных» датчиков. А во время полета, когда шасси убрано, эти же самые датчики могут быть переведены в группу «фоновых» датчиков. Параметры двигателей из «фоновых», которыми они могут числиться во время стоянки, будут переведены в группу «быстрых», начиная с момента включения двигателей. Таким образом, будет выделен ресурс для обслуживания «быстрых» и «аварийных» датчиков. И, далее, перераспределяя по времени приоритеты обслуживания датчиков, можно сделать вывод о том, что быстродействие узла управления в таком случае может быть оптимизировано под конкретную задачу и в целом существенно увеличено.

Таблица 3. Сравнение вариантов обслуживания датчиков

Тип датчика	Вариант 1	Вариант 2	Вариант 3	Ускорение — вариант 2	Ускорение — вариант 3
«быстрый аварийный»	1000	300	1	3,3	1000
«аварийный»	1000	300	100	3,3	10
«быстрый»	1000	600	400	1,7	2,5
«медленный»	1000	1200	1300	0,83	0,78
«фоновый»	1000	60 000	10 000	0,017	0,1

Примечание:
3. В столбцах «Вариант» записано число тактов, необходимое для обслуживания датчиков.
4. В столбцах «Ускорение» записан коэффициент ускорения процесса относительно варианта № 1.

Теперь точно так же можно рассмотреть еще один аспект.

Стандартный микроконтроллер работает с разными типами данных. Например, для 32-битного микроконтроллера это могут 32-битные и 16-битные слова, байты и биты. Но дело в том, что работа с битами для процессора крайне «неудобна». Надо извлечь бит из слова или из байта и куда-то его положить, затем так же поступить с другим битом, потом выполнить требуемое действие, а затем результат этого действия снова упаковать в байт или слово, и это слово переслать туда, где оно должно храниться. Таким образом, для работы с битовыми переменными микропроцессор должен потратить несколько команд на то, чтобы обработать каждый бит.

Почему задача работы с битовыми переменными не так актуальна в ASIC-процессорах? Да просто дело в том, что у ASIC-процессора имеется гораздо больше доступных ресурсов, поэтому в таком случае можно использовать для хранения бита данных целое слово данных, хотя при этом снижается эффективность использования памяти. И поэтому такой способ работы для встроенного в FPGA процессора иногда слишком «дорогой». Следовательно, в этом случае как раз и возникает потребность в нестандартном узле, способном обрабатывать битовые переменные. Но перед тем как продолжить рассмотрение этого раздела, необходимо обратить внимание на еще один пункт — это наличие в алгоритмах управления большого числа программных таймеров. Почему эти два вопроса близки? Представим себе таймер в виде битовой «таймерной» функции, которая находится в двух состояниях: «Таймер взведен» и «Таймер сброшен». Если мы хотим спроектировать узел управления реального времени, то это значит, что каждая ветвь технологического процесса «сверху» должна «контролироваться» по времени, затраченному на эту ветвь процесса. Примерно так, как это делается у хороших шахматистов. То есть «процесс должен завершиться через N секунд, а в противном случае нужно делать…».

Но «внутри» каждой ветви процесса могут быть свои таймеры. Так, например, процесс соединения двух абонентов по протоколу X25 может потребовать до 7 таймеров различной длительности. Таким образом, управляющий микропроцессор должен одновременно формировать сотни программных таймеров различной длительности от единиц миллисекунд до секунд и, возможно, часов (в зависимости от задачи управления). Но, поскольку у реального микропроцессора число аппаратных таймеров ограничено, то выдержки времени обычно формируются программно, путем подсчета квантов времени, получаемых по прерыванию от одного аппаратного таймера. Это значит, что, кроме прерываний от периферии и от системного таймера, микропроцессор будет тратить свой ресурс и на обработку прерываний от аппаратного таймера, а также и на формирование от десятков до сотен программных таймеров.

При использовании FPGA появляется возможность легко распараллелить процессы вычислений и выделить формирование программных таймеров в отдельный блок многоканального таймера. Такой блок может представлять собой специализированный вычислитель, выполняющий счет, например по 128-ми 16-битным каналам.

Теперь можно вернуться и к битовым переменным, которые описывают входные дискретные сигналы. Для рассмотренной выше

СУ на 1000 дискретных входов может добавиться еще несколько сотен таймеров. Как надо вводить данные, мы уже рассмотрели. А как их обслуживать?

Что делать с данными?

Как мы знаем, реакцию системы управления упрощенно можно представить как ряд следующих действий: «ввели данные», «обработали данные», «вывели данные». Начнем с конца. Выводить обработанные данные легче всего. Ведь мы знаем, какие данные мы обрабатывали, следовательно, знаем, и куда их выводить. Как вводить данные, мы предварительно тоже рассмотрели. Давайте так же бегло (пока!) рассмотрим, что значит обработать данные.

Что мы имеем в ТЗ? Некоторый алгоритм, который нам надо реализовать, причем не очень простой, а иначе не имело бы смысла делать микроконтроллер. Алгоритм будет обсчитываться микроконтроллером, и автомат состояний, который требуется для реализации заданного алгоритма, будет формироваться программно.

При аппаратной реализации в FPGA автомата состояний этому автомату требуется только один такт для перехода от одного состояния к другому. Поэтому как только хотя бы один входной сигнал изменился, и аппаратный автомат получил очередной сигнал тактирования, он перейдет в новое состояние (здесь мы рассматриваем только синхронные автоматы). А для программной реализации автомата состояний микроконтроллеру может потребоваться значительно больше времени, чем один такт. Следовательно, в отличие от аппаратного, программный автомат состояний не может быть сформирован при изменении входных сигналов, если они менялись в соседних тактах, просто потому, что микропроцессор не успеет закончить свои вычисления. Что нужно делать, если мы ведем вычисления по программному формированию автомата состояний: продолжить процесс вычислений или начать его заново? И когда начинать программу формирования автомата состояний: в произвольный момент времени или когда считаны данные от всех датчиков? Здесь, так же как и в предыдущих случаях, надо идти «от задачи». Например, если скорость вычислений достаточно велика, то время формирования программного автомата состояний будет много меньше, чем скорость изменений входных данных. Наверное, в таких случаях можно «позволить себе» довести вычисления по формированию программного автомата состояний до конца, вывести выходные данные и затем продолжить вычисления. Когда скорости ввода всех данных и скорости обработки будут одного порядка, целесообразней будет поступить по-другому. Если идет работа с данными, соответствующими «обычным» данным, то можно продолжить процесс вычислений. А для случая поступления аварийных данных будет лучше прервать текущее вычисление и запустить процесс обработки аварийных данных.

Что мы получили в результате эскизной проработки задания?

Для реализации обслуживания датчиков по алгоритму № 3 нам необходимо иметь следующие блоки:

- автомат состояний, для обслуживания «быстрых аварийных» датчиков;
- процессор для ввода данных от остальных датчиков и перекодировки адреса этих датчиков в системные адреса;
- битовый сопроцессор, для логической обработки информации от датчиков.

Как выполняется автомат состояний — здесь описывать не будем. Как выполняется битовый сопроцессор — описано достаточно подробно в [10]. Давайте сосредоточимся на том, как будет работать процессор по вводу и перекодировке данных.

Как будем вводить и перекодировать данные?

С точки зрения программиста, датчики находятся в каком-то поле адресов и представляют собой массив. Сейчас мы не будем конкретизировать то, как именно адресуются датчики: как порты ввода/вывода, находятся ли они в едином адресном пространстве процессора или адресуются косвенно через какой-либо регистр или порт. Суть наших рассуждений от этого сильно не изменится. Представим себе обработку этого массива, выполненную в командах какого-нибудь стандартного микропроцессора. Пусть это будет пока только программа ввода и перекодировка данных, даже без определения того, произошло ли изменение этих данных или нет. Посмотрим, как это «умеют» делать стандартные процессоры. Напишем программу, выполняющую перекодировку 8 элементов массива. Например, такую, как показано на рис. 5.

```
Main()
{
unsigned char InputArray[10];
unsigned char DecodeArray[10];
unsigned char TargetArray[10];
// инициация массива
  for (unsigned char i=0; i<8; i++)
     {
     // DecodeArray[i] = 0x30 + I;
     InputArray[i] = I;
     DecodeArray[i]= 7 — I;
     }

                  // перекодировка — вот это меня и интересует!!!!
                  For (unsigned char i=0; i<8; i++)
     {
     TargetArray[i] = InputArray[DecodeArray[i]];
     }
}
```

Рис. 5. Программа, моделирующая процесс ввода данных и перекодировки, выполненная на языке С

Массив InputArray — имитирует наши датчики. Массив DecodeArray — содержит таблицу перекодировки, а массив TargetArray — это таблица в памяти, где будут храниться перекодированные значения.

Теперь давайте посмотрим на то, как будет выполняться данная программа на процессоре ARM7. На рис. 6 приведена часть листинга программы перекодировки. Сам процесс перекодировки занимает 10 команд на прием и перекодировку только одного датчика. Но, кроме 10 тактов, затраченных на выполнение команд, процессор потратит еще 3 такта на перезагрузку очереди команд, при выполнении команды ветвления. Всего 13 тактов. Наверное, это неплохо для машины «общего применения», но, скорее всего, это будет плохо для нашего конкретного случая. А почему — «скорее всего»? Как вы знаете, чем выше тактовая частота, тем больше будет потребляемая кристаллом мощность. Но ведь, возможно, в этом же кристалле будет находиться еще и весь остальной проект пользователя. И если мощность, потребляемая этим «всем остальным», будет значительно больше, чем мощность, допустимая для рассеяния процессором, то и далее беспокоиться за снижение рассеиваемой мощности в процессоре смысла нет. А во всех остальных — «обычных» — случаях нужно заняться оптимизацией работы процессора, чтобы сократить число команд, необходимых для обработки. Уменьшение числа команд приведет к повышению скорости обработки и сокращению потребляемой мощности.

Так что же будет, если мы начнем реализацию этого алгоритма сами? Начнем мы, конечно, с самого простого варианта, а затем попытаемся его несколько улучшить. Представим себе, что мы используем пару индексных регистров: при помощи первого из них мы вводим данные, а второй используем для

```
                  // перекодировка — вот этот фрагмент
                  For (unsigned char i=0; i<8; i++)
0000A0E3          MOV     R0,#+0
0A0000EA          B       ??main_2
 {
  TargetArray[i] = InputArray[DecodeArray[i]];
     ??main_3:
0010B0E1  MOVS  R1,R0
FF1011E2  ANDS  R1,R1,#0xFF  ;; Zero extend
18208DE2  ADD   R2,SP,#+24
0030B0E1  MOVS  R3,R0
FF3013E2  ANDS  R3,R3,#0xFF  ;; Zero extend
0CC08DE2  ADD   R12,SP,#+12
0C30D3E7  LDRB  R3,[R3, +R12] ;; загрузить байт
0DC0B0E1  MOVS  R12,SP
0C30D3E7  LDRB  R3,[R3, +R12] ;; загрузить байт
0230C1E7  STRB  R3,[R1, +R2]  ;; сохранить байт
  }
010090E2  ADDS  R0,R0,#+1
     ??main_2:
FF0010E2  ANDS  R0,R0,#0xFF  ;; Zero extend
080050E3  CMP   R0,#+8
F1FFFF3A  BCC   ??main_3
}
0000A0E3  MOV   R0,#+0
24D08DE2  ADD   SP,SP,#+36   ;; stack cleaning
1EFF2FE1  BX    LR           ;; return
```

Рис. 6. Часть листинга программы, моделирующей процесс ввода данных и перекодировки, выполненная на ассемблере процессора ARM7

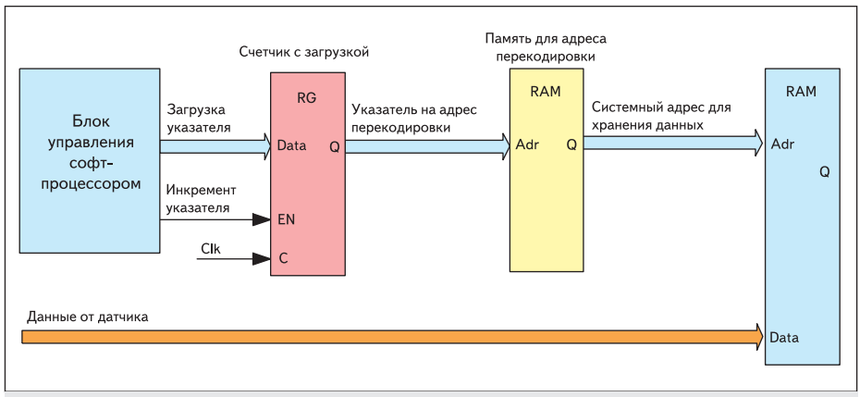

Рис. 7. Блок-схема вычислительного узла, выполняющего команду пересылки данных по адресу, находящемуся в памяти, на адрес которой указывает содержимое регистра операнда

перекодирования. Кроме этого, мы можем задействовать еще один регистр для подсчета числа обработанных данных.

Инициализация программы будет довольно простой: в первые два регистра — индексные — заносим адреса массивов, а в третий регистр — число бит данных для обработки.

Тело цикла программы будет выглядеть примерно так:

1. Вводим данные по адресу, записанному в первом регистре.
2. Записываем их по адресу, записанному во втором регистре.
3. Инкрементируем первый регистр.
4. Инкрементируем второй регистр.
5. Проверяем на ноль третий регистр, если его содержимое равно 0, то выходим.
6. Содержимое третьего регистра не равно нулю, поэтому производим его декремент.
7. Делаем переход на начало цикла.

Что же мы получили? На ввод одного бита данных в этом варианте тратится 7 команд, из которых 2 команды ветвления. До сих пор мы не описывали архитектуру процессора, поэтому и не было никаких описаний работы конвейера, очереди команд и прочего. Но, рано или поздно, мы все равно к этому придем и подробно рассмотрим все аспекты, с этим связанные. То же самое можно сказать о том, будет ли у нас применена RISC-машина или нет. Давайте предположим, что мы выберем RISC-машину. И пока, несколько забегая вперед, необходимо только отметить, что хотя все команды, выполняемые RISC-машиной, «равноправны» по времени, занимаемому на выполнение команды, кроме команд ветвления. Теперь давайте вспомним, что все команды RISC-машины действительно выполняются за один такт, но выполнение команд ветвления, то есть команд переходов, команд вызовов подпрограмм, возвратов и прерываний — после того, как выполнилась сама команда, приводят к необходимости очистить очередь команд. И пока она не заполнится командами, выбираемыми из нового адреса, машина фактически находится в ожидании нового кода команды.

Поэтому, возвращаясь к примеру цикла программы, приведенному выше, мы к 7 командам должны прибавить как минимум еще 1 такт на выполнение команды перехода. Следовательно, при конвейере в 1 уровень мы получим уже не 7, а 8 тактов на тело цикла. При 5-уровневом конвейере мы получим, соответственно, 13 тактов. Поскольку мы сейчас еще не знаем, какой конвейер применим, давайте ориентироваться на глубину конвейера, равную трем. Тогда мы получим 10 тактов на один бит данных. Можно сказать, что КПД — как у паровоза. Посмотрим на то, как изменить команды процессора, чтобы получить нужный нам «паровоз». Первое, что можно сделать — это убрать команды инкремента и вместо них ввести команды, выполняющие при пересылке данных автоинкремент и/или автодекремент регистра. Тогда одна из команд в теле цикла будет выглядеть так: вводим данные по адресу, записанному в первом регистре, после чего регистр делает автоинкремент. Ту же методику применим и к регистру — счетчику циклов. Представим, что вместо обычной команды перехода, если в регистре информация не равна нулю, мы введем в набор команд разрабатываемого микропроцессора команду, которая будет проверять на ноль третий регистр, и если его содержимое равно 0, то эта команда будет выполнять переход, а если содержимое не равно нулю, то оно будет декрементировано.

Теперь тело цикла программы будет выглядеть примерно так:

1. Вводим данные по адресу, записанному в первом регистре, и после этого инкрементируем первый регистр.
2. Записываем их по адресу, записанному во втором регистре, и после этого инкрементируем второй регистр.
3. Проверяем на ноль третий регистр, если его содержимое равно нулю, то выходим, а если содержимое не равно нулю, то оно будет декрементировано.
4. Делаем переход на начало цикла.

Уже лучше, но ведь можно еще сократить код команды.

Команды пересылки данных могут выполняться следующим образом:

1. Пересылка данных по непосредственному адресу.
2. Пересылка данных по адресу, находящемуся в регистре операнда.
3. Пересылка данных по адресу, находящемуся в регистре (или памяти), на адрес которого указывает содержимое регистра операнда (индексная адресация).

Вот этот третий случай мы и применим сейчас. Давайте сформируем такую команду, которая выполняет действия по пункту 3 и при этом еще умеет делать автоинкремент регистра операнда. На рис. 7 показана блок-схема такого вычислительного узла. Для упрощения здесь мы не рассматриваем все узлы синхронизации, необходимые для функционирования в синхронной системе.

Что у нас получилось? В теле цикла ввода информации от датчиков и перекодировки остались только две команды. Первая — ввод данных с перекодировкой, вторая — проверка на окончание цикла (ее мы рассматривали выше). Уж очень «нехорошая» эта команда — ветвление. Опять вспомним про очередь команд, которую придется перезагружать при выполнении переходов. Самый простой путь — это выдавать команду ввода и перекодировки не один раз, а группами, например по 8 или 20 команд, тогда потери от переходов в общем цикле обработки уменьшатся. Но ведь при этом придется неэффективно «забивать» память команд многократно повторяющимися командами. С точки зрения программиста оптимизировать больше нечего. А с точки зрения разработчика микропроцессоров? Возможно, есть другой путь решения этой частной задачи? Конечно, такой путь есть! Давайте вместо того, чтобы выдавать команду ввода и перекодировки группами, просто сделаем блокировку счетчика адреса команд, например на те же самые 8 или 20 команд. Если счетчик команд заблокирован, то это значит, что из памяти команд будут многократно читаться одни и те же коды команд, соответствующие той команде, на которой остановился счетчик.

Остается еще один шаг. Давайте сделаем аппаратный счетчик циклов. Это будет такой узел, который будет добавлен в процессор, и, конечно, управляться он будет только программно. Получаем команду из памяти команд, декодируем ее и загружаем в этот счетчик циклов. Вот пусть он нам и заблокирует счетчик команд на то число циклов, которое загрузится в счетчик при выполнении программы. Есть вопрос? Это прекрасно, значит, вы внимательный читатель. Конечно, этот счетчик циклов должен заблокировать счетчик команд, но, безусловно, не со «своей» команды. Иначе машина задержится на команде загрузки счетчика блокировки и будет выполнять это действие «навсегда» или до получения сброса, прерывания или выключения. Как надо поступить в этом случае?

Как сейчас принято, надо дать читателю возможность выбора: предыдущая команда, текущая команда и последующая команда. Про предыдущую можно сказать то же, что и про текущую команду, а ее, как мы уже рассмотрели, блокировать нельзя. Так давайте сделаем блокировку не с текущей, а с последующей команды. Последовательность действий будет такая: сначала выполняем все настройки для пересылки данных, потом, для выполнения нужного нам числа циклов ввода, выполняем команду загрузки счетчика циклов, а сразу после нее дадим команду ввода и перекодировки. В итоге мы получим требуемый алгоритм. На ту же самую группу из 8 или 20 команд мы дополнительно затратим только одну команду загрузки счетчика. Теперь у нас нет потерь времени на ветвления и нет необходимости перезагружать очередь команд. Да и в память команд мы поместили только ОДНУ дополнительную команду и ОДНУ команду пересылки на всю группу пересылаемых данных. Что же пришлось добавить в проект микропроцессора? Один триггер и вход запрета счета в счетчике команд. Что еще мы получили? А кроме того, о чем было написано выше, мы получили возможность программно управлять длительностью циклов процессора, что можно использовать при медленно работающей внешней периферии или памяти. Теперь у нас работает связка команд: загрузка счетчика блокировки + ввод/вывод данных по внешней шине. Итак, вместо 13 тактов мы можем работать только с одним тактом! Выигрыш очевиден и не требует дальнейших обсуждений.

Что можно сказать в заключение. Здесь мы рассмотрели то, как необходимо вести эскизную разработку проекта. Определили, почему микроконтроллер с «самодельной» системой команд будет на данной задаче оптимальнее, чем стандартный микроконтроллер. Рассмотрели, как и какие «архитектурные излишества» позволяют значительно повысить быстродействие. Соответственно, можно сделать следующий вывод: при таком способе проектирования удается улучшить параметры разрабатываемого изделия в десятки раз, только потому, что изделие оптимизируется под заданную задачу.

Теперь пора вспомнить, что читателям обещан второй пример нестандартного микроконтроллера

Теперь посмотрим на то, как можно оптимизировать процессор, который производит обработку непрерывно поступающих данных. Здесь у нас совершенно другой приоритет для оптимизации. Если в предыдущей задаче мы провели оптимизацию, изменив порядок ввода данных из разных групп датчиков, и применили дополнительные аппаратные узлы, то в данном случае дело может оказаться значительно более сложным. Возможно, процессору просто не хватит производительности в выполнении самых обычных арифметических операций, связанных с выполнением задания. Как надо поступать в таком случае? Давайте посмотрим на то, чем мы располагаем. Мы имеем кристалл, в котором будет работать наш проект, и знаем его характеристики. Примерно известно, до какой тактовой частоты мы можем «разогнать» наш вычислительный узел. Эту цифру относительно легко проверить, если сделать «макет» вычислительного узла, состоящий из примерно такого же количества триггеров и памяти, и затем этот «макет» «скормить» компилятору и прочим программным инструментам, используемым при проектировании заданного кристалла. В любом случае тактовая частота сверху будет ограничена предельным значением, которое приводится в документации на кристалл и на тот тип корпуса, в который этот кристалл разварен. Последнее уточнение сделано потому, что разные корпуса имеют разные характеристики по стоку тепла, и, соответственно, они позволяют работать с разными тактовыми частотами. Необходимо добавить, что перегрев кристалла значительно понижает надежность работы микросхемы и сокращает время наработки на отказ. Таким образом, разработчик иногда может оказаться в такой ситуации, что для выполнений условий безотказной работы ему придется снизить тактовую частоту. Итак, что еще у нас есть? Мы ведь работаем в FPGA и можем распоряжаться всеми предоставленными ресурсами по своему усмотрению. Пока мы не станем обсуждать саму структуру конкретного вычислительного узла, разрядность вычислителя и число операндов. Будем считать, что эта информация у нас уже имеется. Для этого воспользуемся материалами, которые предоставляют фирмы, изготовители микросхем. Это таблицы, в которых показаны примеры реализации различных аппаратных вычислительных узлов — счетчиков, сумматоров, умножителей, и приведено время, необходимое для реализации таких вычислений. Представим себе случай, когда нам надо выполнять суммирование, например в 5–8 раз быстрее, чем сможет сделать один сумматор, и для табличных вычислений в FPGA ресурса не хватает. Ответ в данном случае, конечно, известен. Не успеваем сделать вычисления на одном вычислительном узле, давайте добавим в проект еще несколько таких вычислительных узлов. Как можно сопрячь все вычислители? Для начала представим, что мы делим задачу на несколько типовых процессоров. Как же будет выглядеть архитектура?

У нас есть только следующие варианты:

1. Независимые процессоры.
2. Процессорный конвейер.
3. Параллельные процессоры.

Окончание следует

Литература

1. Каршенбойм И. Квадрига Аполлона и микропроцессоры // Компоненты и технологии. 2006. № 4, 5.
2. **http://en.wikipedia.org/wiki/SystemC**
3. **http://www.mentor.com/products/c-based_design/index.cfm**
4. **http://www.celoxica.com/products/dk/default.asp**
5. **http://www.impulsec.com/**
6. **www.altera.com/c2h**
7. Каршенбойм И. Микропроцессор своими руками. Часть 1 // Компоненты и технологии. 2002. № 6, 7.
8. Каршенбойм И. Микроконтроллер для встроенного применения — NIOS. Конфигурация шины и периферии // Компоненты и технологии. 2002. № 2, 3, 4, 5.
9. **http://www.xilinx.com/ipcenter/processor_central/picoblaze/picoblaze_user_resources.htm**
10. Каршенбойм И. Микропроцессор своими руками–2. Битовый процессор // Компоненты и технологии. 2003. № 6, 7.
11. Tomaszewski E. Explicitly Parallel RISC (EPRISC). **http://www.opencores.org/articles.cgi/view/4**
12. **http://www.jwdt.com/~paysan/4stack.html**
13. Chapman K. Creating Embedded Microcontrollers (Programmable State Machines). Part 1, 2, 3. 03/28/2002, **www.xilinx.com**
14. An Overview of the ADSP-219x Pipeline. Engineer To Engineer Note. EE-123, **www.analog.com**
15. U17135EJ1V1UM00.pdf; V850E2 32-bit Microprocessor Core Architecture. **http://www.eu.necel.com**
16. ADuC7024_25_PrD.pdf, **www.analog.com**
17. **http://www.trash.net/~luethi/study/silverbird/silverbird.html**