Проектирование процессорных ядер. Часть 5. Инструментальное обеспечение разработки

Илья ТАРАСОВ,
д. т. н.
ilya_e_tarasov@mail.ru

Данная статья завершает цикл публикаций, посвященный проектированию софт-процессоров с использованием ПЛИС. Рассматриваются вспомогательные инструменты разработки — организация загрузки программы и отладки, а также основные подходы к разработке кросс-компиляторов для вновь создаваемых процессорных архитектур.

Введение

В статье поставлена важная цель, которая призвана органично завершить данный цикл публикаций, — стимулирование самостоятельных исследований в области процессорных архитектур среди широкого круга разработчиков и коллективов. На этом пути существует несколько практических препятствий, которые заставляют считать разработку нового процессорного ядра экономически нецелесообразной или технически нереализуемой. Наиболее значимыми из них являются более высокая стоимость ПЛИС по сравнению с аппаратно реализованными процессорами и контроллерами, а также необходимость создания инструментальных средств проектирования, включающих компиляторы, отладчики и средства анализа кода.

Можно сразу сказать, что статья не предполагает описание каких-либо передовых, оптимальных или перспективных средств компиляции или отладки. Напротив, в качестве основного критерия выбрана простота реализации, обеспечивающая легкость освоения и применения этого подхода для получения полного цикла разработки процессора — от архитектуры до макета в ПЛИС, компилятора и загрузчика. Эти задачи достаточно сильно различаются, и для каждой из них мог бы потребоваться специалист соответствующего профиля. Тем не менее при определенном подходе проектирование оригинального процессора со всем комплектом инструментальных средств можно выполнить и в одиночку. Целью в данном случае является формирование целостной картины всего процесса проектирования, чтобы дальнейшие этапы развития проекта были осознанными. Материал предназначен в первую очередь для инженеров-схемотехников, имеющих навыки программирования.

Загрузка и отладка

Для практической работы недостаточно создать RTL-описание процессора и загрузить конфигурационный файл в ПЛИС. Для функционирования в составе конечного изделия для процессора должно быть разработано и отлажено программное обеспечение. Для программиста этот процесс связан с множественными циклами компиляции, загрузки программы и наблюдения за состоянием процессора. Очевидно, что постоянная загрузка кода путем редактирования строк инициализации памяти в HDL-описании — крайне неэффективное решение. Каждый раз после изменения хотя бы одной команды в программе процесс создания файла конфигурации придется повторять.

Вот почему для нормальной работы необходимо обеспечить доступ к памяти программ (а лучше и к памяти данных) со стороны компьютера, с помощью которого ведется разработка. Для этого целесообразно использовать широко распространенные интерфейсы, уже предусмотренные в его составе.

При всей распространенности Ethernet добавление такого интерфейса в отладочную плату потребует реализации стека сетевых протоколов, хотя бы до уровня обработки UDP-пакетов. Наличие электрического подключения и способность ПЛИС принять IP-пакет на деле не означает, что можно будет обеспечить посылку произвольных данных в ПЛИС с соответствующим удобным контролем над этим процессом. Изучение трафика Ethernet способно продемонстрировать, что даже при отсутствии явно задаваемых данных в сети имеется достаточная активность. Например, невозможность поддержать обмен по протоколу ARP (Address Resolution Protocol) приведет к отключению сетевого устройства (с сообщением «соединение ограничено или отсутствует»). Пользователи, имеющие платы на базе ПЛИС с разъемом Ethernet, могут убедиться в этом самостоятельно. А потому добавление в проект интерфейса Ethernet не является задачей уровня «подключение сигналов» и потребует размещения в проекте процессора с библиотекой стека протоколов.

При необходимости можно использовать проект процессора MicroBlaze с добавлением библиотеки стека сетевых протоколов lwIp [1]. Реализация простого сервера позволит оценить проблемность решаемых задач — на практике сложность системы с поддержкой Ethernet явно превышает сложность процессорного ядра, пригодного для управления простыми периферийными устройствами.

Более доступным вариантом представляется применение интерфейса UART (со стороны ПЛИС). Несмотря на то, что физические разъемы UART не используются в современных компьютерах, данный интерфейс крайне прост для реализации, к тому же существует множество преобразователей интерфейсов, предоставляющих со стороны ПК доступ к виртуальному COM-порту. Преобразователи могут иметь со стороны ПК интерфейс USB или, например, Bluetooth. Поскольку работа с COM-портом весьма проста и не вызывает проблем со стороны программы на ПК, к тому же многие платы на базе ПЛИС уже имеют встроенные преобразователи USB-UART, такой способ доступа к проекту можно считать базовым для разработки. К заметным недостаткам относятся ограниченная скорость UART и отсутствие контроля доставки данных, однако с учетом назначения подобного решения (отправка программ в софт-процессор) указанные недостатки не являются критичными.

Простейший способ загрузки данных в память — использование ее второго порта (поскольку блоки памяти в ПЛИС являются двупортовыми). Для этого необходимо обе-

спечить управление шинами адреса, данных и сигналом «разрешение записи». При такой схеме получается, что один порт используется процессором для чтения команд, а второй управляется непосредственно загрузчиком. Контроллер загрузчика может также обеспечивать вспомогательные сигналы для отладки — в первую очередь управление сигналом сброса для процессора, чтобы перезапустить загруженную программу. Вообще, в процессе загрузки программы процессор нужно переводить в состояние сброса во избежание перемешивания в памяти старой и новой программы, что может вызвать неожиданные эффекты. Взаимодействие между загрузчиком, процессором и двупортовой памятью показано на рис. 1.

Для доступа к сигналам второго порта памяти команд можно использовать простейший протокол, при котором последовательность данных не имеет специального формата пакетов. Вместо этого передаваемые байты разбиваются на два четырехразрядных поля, одно из которых выступает в качестве адреса регистра, а второе — в качестве данных. Таким образом, возможно управлять 16 четырехразрядными регистрами контроллера памяти. Если реализован приемник UART, помещающий принятый байт в регистр received, то фрагмент кода помещает младшие четыре бита данных в один из регистров, как показано в листинге 1. Пример регистровой модели такого устройства управления приведен на рис. 2.

```
case conv_integer(received (7 downto 4)) is
   when 0 => LoadData (3 downto 0) <= received (3 downto 0);
    when 1 => LoadData(7 downto 4) <= received(3 downto 0);
    when 2 => LoadData(11 downto 8) <= received(3 downto 0);
    when 3 => LoadData(15 downto 12) <= received(3 downto 0);
    when 4 => LoadAddr(3 downto 0) <= received(3 downto 0);
    when others => null;
end case;

case conv_integer(received) is
    when 240 => LoadWe <= '0';
    when 241 => LoadWe <= '1';
  when 242 => int_reset <= '0';
    when 243 => int_reset <= '1';
    when others => null;
   end case;
```

Листинг 1. Фрагмент кода на языке VHDL, организующего доступ к порту блочной памяти со стороны контроллера UART

Пример в листинге 1 и на рис. 2 не следует рассматривать как окончательный. Он приведен именно в качестве простого временного решения, которое позволяет начать практическую работу с проектом процессора с применением доступных интерфейсов и программных подходов.

Вопросы отладки в процессорных системах неоднозначны. Прежде всего, необходимо разграничить задачи подключения к существующим системам и программному обеспечению (например, с использованием GDB и интерфейса JTAG) и организацию доступа к собственному проекту процессора с разработкой простейших инструментов отладки и анализа. Не нужно сбрасывать со счетов и такой известный любому программисту способ, как отладочная печать. Несмотря на то, что это, по сути, означает полное отсутствие специальных инструментов отладки (вся работа по предоставлению отладочной информации ложится на саму программу, выводящую состояния переменных при достижении определенных строк программы), такой подход позволяет по крайней мере начать работу.

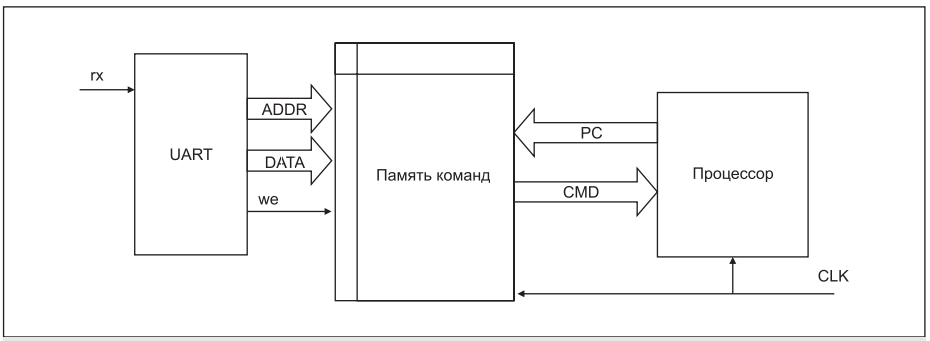

Рис. 1. Структурная схема взаимодействия загрузчика, процессора и двупортовой памяти команд

Старшая часть байта	Значение младшей части байта
0	Data(3:0)
1	Data(7:4)
2	Data(11:8)
3	...
4	Addr(3:0)
0xF	Управление

Коды команд управления	Действие
0XF0	We = 0
0xF1	We = 1
0xF2	Reset = 0
0xF3	Reset = 1

Рис. 2. Пример регистровой модели контроллера доступа к памяти команд для упрощенного загрузчика

Можно заметить, что разработчика, использующего инструменты моделирования на RTL-уровне, все же не стоит считать оставленным наедине с проблемой отладки. Построение временных диаграмм сигналов процессора позволяет понять закономерности его работы и выявить логические ошибки в схеме. Поэтому на начальном этапе, чтобы не перегружать список решаемых задач, можно пользоваться комбинацией логического моделирования в RTL-симуляторе и отладочной печати, а далее определить, какие функции отладки необходимы.

Кросс-компиляция

Инструментальное программное обеспечение — это важнейшая составная часть проекта процессорной системы. Ручное формирование машинных кодов крайне непродуктивно, и его невозможно использовать даже для отладки сколько-нибудь объемных программ. Поэтому для разработки процессора необходимо иметь хотя бы базовые инструменты для создания кода, пусть и с ограниченными возможностями.

Под кросс-компиляцией (cross-compilation) понимается процесс перевода программы в машинный код, исполняемый на процессоре с архитектурой, отличной от той, на которой запускается сам компилятор. Иными словами, речь идет о том, чтобы на ПК с процессором x86 была запущена программа, преобразующая некий исходный текст в машинные коды для процессора с другой архитектурой. Созданный машинный код может быть предназначен как для загрузки в макет процессора в ПЛИС, так и для проведения с его помощью моделирования.

Хорошо известным трудом в области разработки компиляторов является «книга дракона» (Dragon book) [2]. Ее настоящее название «Компиляторы: принципы, технологии и инструментарий», а наименование «книга дракона» вошло в обиход из-за оформления обложки, на которой процесс разработки компилятора представлен в виде борьбы рыцаря-программиста с драконом-компилятором. Книга описывает различные аспекты разработки компиляторов, однако не все из них обязательны для построения практического продукта. В ней дается общий маршрут работы компилятора, представленный на рис. 3.

Большая часть литературы, описывающей разработку компиляторов, концентрируется на синтаксическом анализе. Несмотря на важность данного аспекта разработки компиляторов, он не решает главную задачу, стоящую перед разработчиками процессора, — получение машинного кода, который может быть помещен в память команд. Поэтому в рамках данной статьи будут рассмотрены практические подходы, в действительности далекие от современных компиляторов языков высокого уровня. Однако следует понимать, что разработка компилятора не только требует соответствующих навыков, но и отнимает длительное время. Насколько актуальным шагом является получение полнофункционального компилятора с учетом того, что разработка

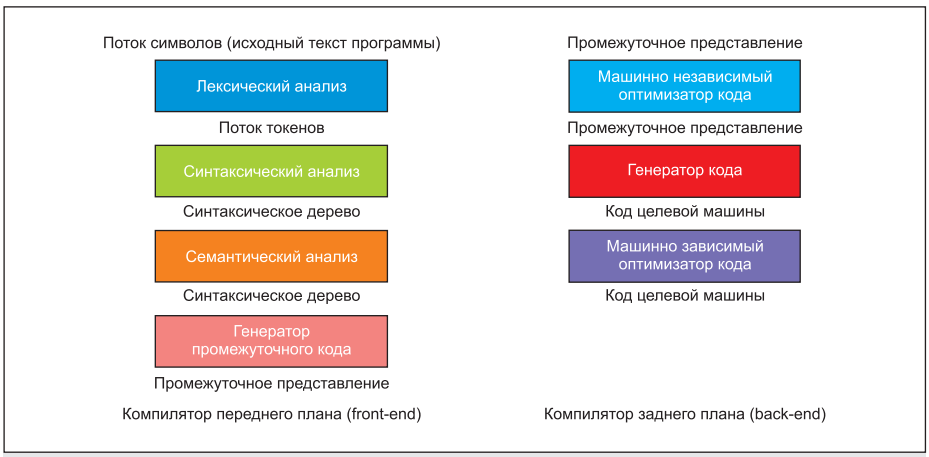

Рис. 3. Стадии компиляции

процессора с оригинальной архитектурой на HDL может быть выполнена довольно быстро? Вопрос может считаться риторическим, если принять во внимание, что первые практические результаты для процессорного ядра могут быть получены буквально за несколько часов, а разработка компилятора, способного получить машинный код из высокоуровневого представления, потребует существенно больше времени. Такая ситуация неминуемо застопорит исследования в области схемотехники процессора, вынуждая инженера в конечном итоге склониться к решениям, для которых уже есть инструментальные средства.

Поэтому можно попробовать использовать в процессе разработки своего рода компромиссные решения — компиляторы, позволяющие формировать машинный код для произвольного процессора, однако обеспечивающие только минимальную функциональность, пригодную для реализации за ограниченное время и с применением простейших подходов программирования. Конечно, такой инструмент нельзя считать полноценным компилятором, подходящим для эксплуатации широким кругом будущих пользователей, впрочем, вопрос можно поставить и иначе: будет ли проект процессора вообще выпущен для этих пользователей, если в процессе разработки не имелось хотя бы «временных» инструментов программирования?

Рассмотрим вкратце шаги, которые могли бы помочь создать простой компилятор, переводящий исходный текст программы в последовательность машинных кодов. Исходя из рис. 3 первым шагом компиляции является лексический анализ, переводящий исходный текст программы в последовательность токенов. Под токенами понимаются пары «имя — значение», построенные для каждого найденного элемента программы. Например, ассемблерная команда mov r0, r1 будет разобрана на элементы “mov”, “r0”, “r1”.

Синтаксический анализ имеет целью построить промежуточное представление программы. Существуют различные способы отображения порядка операций программы, например в виде синтаксического дерева. Однако такое представление может быть несколько избыточно для простого ассемблера.

На стадии семантического анализа проверяется смысловая корректность построенного промежуточного представления. Так, строка x = y + z является синтаксически корректной, однако если x, y — это целые числа, а z — строка, то в представленном виде такой оператор будет ошибочным. Необходимо или констатировать ошибку, или, если разработанный язык это допускает, выполнить преобразование строковой переменной в число перед сложением.

В результате проверки корректности промежуточного представления образуется машинно независимый код. С его получением завершается компиляция переднего плана (front-end). Данное представление все еще нельзя назвать последовательностью машинных кодов, за получение которой отвечает компилятор заднего плана (back-end). Его центральным элементом является собственно генератор кода, однако как с промежуточным представлением, так и с итоговым кодом возможно выполнение оптимизирующих преобразований.

Можно утверждать, что разработка компиляторов современных языков высокого уровня представляет собой трудоемкую задачу, требующую согласованной работы разноплановых специалистов. Однако получение работоспособного языка уровня ассемблера вполне может быть реализовано и небольшой группой или даже индивидуальным разработчиком.

Сложность создания компилятора, особенно на уровне синтаксического анализа, существенно зависит от класса реализуемого языка. Классификация грамматик была предложена Хомским и включает четыре типа: фразовую, контекстно зависимую, контекстно свободную и регулярную грамматики. Не рассматривая в рамках данной статьи особенности этих классов, можно указать, что регулярная грамматика, имея больше ограничений по сравнению с другими типами, одновременно и допускает более простые реализации для синтаксического анализа.

Представим программу на ассемблере в виде потока лексем (элементов языка). Например, команда *nop* не имеет аргументов и может быть использована в исходном виде, а для команды *mov* требуется два операнда. Если рассматривать строки вида

```
<command> <op1>, <op2>
```

можно видеть, что в строке за командой следуют два операнда, разделенные запятой. Тогда алгоритм анализа строки может содержать следующие шаги:

1. Выделить команду (ограниченную пробелом) и сравнить ее со списком допустимых команд.
2. Выделить первый операнд (ограниченный пробелом или запятой).
3. Выделить второй операнд.

Поскольку данные действия происходят последовательно, а от найденной команды зависит список требуемых операндов (например, если найдена команда *nop*, поиск операндов необходимо остановить, а если они найдены, это следует считать ошибкой), можно реализовать синтаксический разбор с помощью конечного автомата.

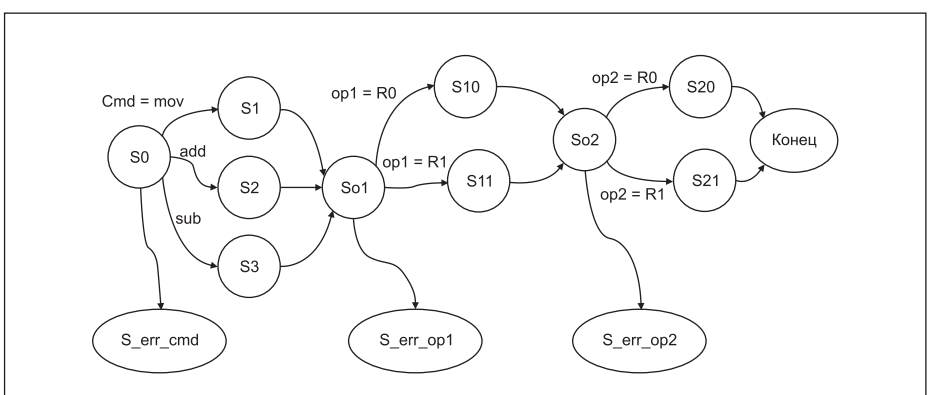

Рис. 4. Диаграмма переходов конечного автомата для простого синтаксического анализатора

Фрагмент диаграммы для простого синтаксического анализатора показан на рис. 4.

Пример реализации подобного синтаксического анализа показан в листинге 2. В примере разбираются строки вида “cmd op1, op2“, причем команда может быть *mov* или *add*, а в качестве операндов принимаются только r0 и r1. Можно видеть, что в процессе анализа определяются команда и оба операнда, что позволяет получить значение для машинного кода проанализированной операции, просто подставив номера команды и операндов в соответствующие поля машинного кода.

```
void syntax(char * str)
{
   int op1, op2;
   char * token = strtok(str, “ “);

   if (strcmp(token, “mov”) == 0)
   {
      token = strtok(NULL, “ ,”);
      op1 = -1;
      if (strcmp(token, “r0”) == 0)
      {
         op1 = 0;
      }
      if (strcmp(token, “r1”) == 0)
      {
         op1 = 1;
      }

       token = strtok(NULL, “ “);
      op2 = -1;
      if (strcmp(token, “r0”) == 0)
      {
         op2 = 0;
      }
      if (strcmp(token, “r1”) == 0)
      {
         op2 = 1;
      }

      printf(“mov: op1 = %d op2 = %d \n\r”, op1, op2);
   }

   if (strcmp(token, “add”) == 0)
   {
      token = strtok(NULL, “ ,”);
      op1 = -1;
      if (strcmp(token, “r0”) == 0)
      {
         op1 = 0;
      }
      if (strcmp(token, “r1”) == 0)
      {
         op1 = 1;
      }

       token = strtok(NULL, “ “);
      op2 = -1;
      if (strcmp(token, “r0”) == 0)
      {
         op2 = 0;
      }
      if (strcmp(token, “r1”) == 0)
      {
         op2 = 1;
      }

      printf(“add: op1 = %d op2 = %d \n\r”, op1, op2);
   }

   }

  int main(void) {
   char s1[] = “mov r0, r1”;
   char s2[] = “add r1, r0”;

   syntax(s1);
   syntax(s2);
   return 0;
}
// Результаты работы:
mov: op1 = 0 op2 = 1

add: op1 = 1 op2 = 0
```

Листинг 2. Простейший синтаксический анализ

Явно видимым недостатком представленного подхода является большая трудоемкость описания всех возможных комбинаций команд и их операндов. Из текста видно, что получение сколько-нибудь полного перечня поддерживаемых инструкций процессора потребует единообразной работы по повторению показанных шаблонов кода с заменой лексем и использованием для каждой команды собственного варианта проверки операндов. Более перспективным представляется применение синтаксических анализаторов, использующих формализованное описание команд и их операндов, например на основе формы Бэкуса — Наура. Однако такое направление работ несколько отдаляет создание собственно процессора, поскольку значительная доля усилий будет потрачена на синтаксический анализатор и проектирование языка.

Развитие процессорных архитектур делает актуальными так называемые перенацеливаемые компиляторы (retargetable compilers). Поскольку многие шаги анализа, а также машинно независимые оптимизации становятся общими для разных процессоров, имеет смысл разделить машинно зависимую и машинно независимую части компиляторов. Известными перенацеливаемыми компиляторами являются gcc и llvm. Кроме того, идея разделения машинно зависимой и машинно независимой частей нашла отражение в технологии Microsoft.net, использующей собственное промежуточное представление MSIL (MicroSoft Intermediate Language) [3]. Хотя получение процессора, поддерживаемого gcc, представляется привлекательным, следует указать, что такая адаптация — не вполне простая операция. Для того чтобы gcc был способен генерировать код для нового процессора, необходимо разработать machine description (описание процессора в файле с расширением md). Этот файл содержит от сотен до тысяч строк в специальном формате, с которым можно ознакомиться в [4]. Пример, приводимый в руководстве по созданию machine description, показан в листинге 3.

```
(define_insn “addsi3”
   [(set (match_operand:SI 0 “general_operand” “=r,m”)
      (plus:SI (match_operand:SI 1 “general_operand” “0,0”)
         (match_operand:SI 2 “general_operand” “g,r”)))]
   “”
   “@
   addr %2,%0
   addm %2,%0”)
```

Листинг 3. Пример фрагмента описания процессора в файле machine description

Можно заметить, что выходным форматом компилятора gcc является ассемблерный текст! Другими словами, разработка собственного файла machine description означает возможность получения ассемблерного текста, требующего дополнительной обработки ассемблером соответствующего процессора (однако команды этого ассемблера будут такими, которые указаны в md-описании).

Если речь идет о получении инструмента «для внутреннего употребления», можно воспользоваться ультимативными подходами к реализации регулярной грамматики. Нужно отметить, что язык Форт в свое время стал одним из примеров такого доведенного до абсолютизма подхода к реализации регулярной грамматики, поскольку программа на языке Форт представляет собой просто последовательность операций, разделенных пробелами. Каждая лексема языка имеет соответствие в виде фрагмента кода, исключение составляют только числа (литералы), для которых выполняется попытка преобразования текста в число (в силу того, что перечислять все возможные варианты чисел было бы слишком накладно). Следствие такого подхода — постфиксная запись выражений (сначала операнды, потом операция), а также стековая организация вычислений (это позволяет указывать несколько операндов подряд, не заботясь о механизме их размещения). Поскольку в Форт-системе уже реализован синтаксический разбор и язык не накладывает ограничений на ввод новых понятий, можно достаточно легко получить примитивный ассемблер с постфиксной записью команд. Например, в [5] демонстрируется ассемблер для процессора 580 ВМ80, чье описание занимает менее двух страниц.

Используемый в Форте подход к построению ассемблера основан на «мимикрии» вводимых команд Форта под существующие элементы ассемблера. Скажем, если определить константы A B C D E H L, а также A, B, C, D, E, H, L (запятая не является запрещенным элементом для Форта), то команда MOV для процессора 580ВМ80 определяется как:

```
: MOV 8 * + 64 + TC-C, ;
```

Используется тот факт, что команды пересылки «регистр-регистр» имеют для процессора 580ВМ80 формат «01xxxyyy», где xxx, yyy — индексы регистров, предусмотренные для пересылки данных. Здесь команда TC-C переносит полученный байт в выходной массив, содержащий сгенерированный код, а смысл остального текста понятен с учетом используемой постфиксной записи. В действительности результат ассемблирования мог бы быть получен и с помощью команды вида 2 3 MOV, поскольку вместо имен регистров анализируются попросту их номера. Однако ожидается, что номера регистров будут оставлены соответствующими командами, поэтому в итоге текст на форт-ассемблере будет выглядеть в виде строк вида:

```
A, B MOV
```

Это не соответствует привычной для ассемблеров форме записи текста, но понятно человеку, а с учетом крайне низкой трудо-

емкости описания таких команд может пригодиться и для внутренних задач разработки. При этом необязательно использовать язык Форт как таковой (в виде соответствующих программных продуктов) — достаточно самостоятельно реализовать разделение элементов исходного текста по пробелам и поиску полученных лексем в простой таблице. В данном случае корректнее было бы говорить не о программировании ассемблера на Форте, а о применении подхода, основанного на регулярной грамматике.

Совместная оптимизация аппаратного и программного обеспечения

Для того чтобы процессор оказался эффективным на практике, необходимо обеспечить выполнение согласованных условий:

1. Архитектура процессора, его регистровая модель и выбранная системная шина обеспечивают оптимальное взаимодействие с аппаратным обеспечением.
2. Инструментальное программное обеспечение способно генерировать машинный код, в полной мере использующий возможности процессора и создающий в критических местах наиболее эффективные последовательности машинных команд.

Для ПЛИС это достигается следующими способами.

В первую очередь софт-процессор применяется в составе системы таким образом, чтобы подчеркнуть сильные стороны ПЛИС и снизить негативный эффект от ее слабых сторон. Очевидным неэффективным путем использования софт-процессора является повторение одной из существующих процессорных архитектур и применение ПЛИС для реализации конфигурируемых на этапе разработки периферийных устройств. В подобном случае возникнет закономерный вопрос: почему бы для реализации подобных целей не обратиться к широко распространенным микроконтроллерам, которые обеспечат такие же возможности при существенно меньшей цене? Более того, со стороны МК можно будет ожидать более широкий спектр периферийных устройств, не ограниченный цифровыми интерфейсами, а также экосистему разработки, обеспечивающую компиляторы, среды проектирования, библиотеки, примеры использования и т.д.

Тем не менее, поскольку ПЛИС заняли устойчивую нишу на рынке электронных компонентов, остается только понять, в силу каких причин это произошло и каким образом свойства данной элементной базы могут быть эффективно применены в конкретном проекте. Для практических целей нужно ответить на следующие вопросы:

1. Предусматривает ли задача интенсивное использование несложных параллельных вычислений?

Очевидным применением являются приложения цифровой обработки сигналов, получающие существенные преимущества от независимых параллельно работающих блоков «умножение с накоплением» (компоненты DSP48 в современных FPGA Xilinx). В этом случае софт-процессор будет играть в системе вспомогательную роль и служить только для упрощения настройки цифровых фильтров, IP-ядер БПФ и прочих подобных устройств. Он может иметь весьма умеренную тактовую частоту (поскольку не является компонентом, определяющим производительность системы), однако должен обеспечивать поддержку внешних интерфейсов, реализующих доступ к подсистеме цифровой обработки сигналов со стороны оператора. Подобную роль в настоящее время успешно выполняют софт-процессоры MicroBlaze.

2. Имеются ли в системе независимые процессы, требующие обработки в режиме реального времени?

Данная задача весьма неоднозначна и требует пристального внимания со стороны разработчиков. При попытке ее решения с помощью одного процессорного ядра (пусть даже и заявленного как real-time) перед программистом неминуемо возникает целый ряд вопросов. Каков реальный промежуток времени, проходящий между поступлением в процессор запроса на обработку и переходом к формированию выходного управляющего воздействия? Ответ на этот вопрос существенно отличается от «сколько тактов процессор тратит на вход в прерывание», потому что, например, срабатывание аварийного датчика требует отключения соответствующего силового компонента, и факт входа в прерывание еще не означает, что процессор уже послал сигнал отключения в нужный порт. Аппаратные решения способны обеспечить задержку реакции порядка десятков наносекунд, что существенно превосходит возможности программных решений (впрочем, подобные возможности часто избыточны).

Однако, кроме обеспечения минимального времени реакции на внешние воздействия, системы реального времени могут поставить и другую, более сложную задачу. Каким образом необходимо обрабатывать запросы, если подпрограммы их обработки перекрываются во времени? Наличие только одного потока исполнения команд вынуждает программиста устанавливать приоритеты в обработке запросов, причем добавление нового устройства или даже изменение основной программы требует проверять, не нарушается ли нормальная работа с внешним оборудованием, формирующим запросы на прерывания. Проблемы здесь вполне возможны, например, из-за применения библиотек или фрагментов кода, запрещающих прерывания в процессе выполнения каких-либо действий.

Простейшим решением со стороны софт-процессоров становится применение нескольких процессорных ядер, в том числе и специально выделенных для работы с критичными источниками запросов на обработку. Здесь можно говорить не только о датчиках аварийных состояний, но и, например, о большом количестве периферийных устройств, использующих медленные протоколы обмена данными. Чтобы не тратить ресурсы основного процессора на их реализацию, можно обратиться к аппаратным ускорителям уровня конечных автоматов и к несложным процессорным ядрам, оптимизированным для простых операций с портами периферийного устройства.

Можно отметить, что популярные микроконтроллеры семейства PIC появились именно в подобном качестве. Аббревиатура PIC расшифровывается как Peripheral Interface Controller, то есть «контроллер периферийных интерфейсов», и первый микроконтроллер PIC был создан как микросхема для расширения возможностей ввода/вывода процессора CP1600. Даже ПЛИС начального уровня способны вместить несколько несложных процессоров, которые существенно облегчат работу программиста, освобождая его от необходимости «жонглировать» в основной программе несколькими независимыми процессами вычислений.

3. Имеются ли в задаче специфичные вычисления, последовательности команд или шинные циклы?

Положительные ответы на эти вопросы могут стать основанием для разработки собственного процессорного ядра. Действительно, если обратиться к первой части данного цикла статей, в ней имелась отсылка к публикации, анализирующей микроконтроллеры начального уровня. Там упоминалось, что операции записи в выходной порт (микроконтроллерный hello world в виде мигания светодиода) требуют 3–20 тактов в некоторых процессорных ядрах. Естественным решением было бы ускорение таких операций не только путем добавления соответствующих периферийных IP-ядер, но и введением в процессор специализированных команд, например для ускорения доступа к периферии, для реализации автоинкремента/декремента, групповых арифметических и логических операций и т.п. При этом умеренная тактовая частота процессора окажется не столь важной, если процессор будет выполнять значимые для пользователя фрагменты кода за меньшее число тактов.

В данном случае очень важно разделить синтетические примеры и конкретные фрагменты пользовательского кода. Программисты, работающие над конкретным проектом, могут обращаться к специфичным алгоритмам, иметь привычные шаблоны кода или применять определенные языки программирования и компиляторы. В этом случае абстрактные декларации достижимых показателей для какого-либо ядра, сопровождаемые синтетическими при-

мерами, будут иметь невысокую ценность. Напротив, готовность разработчиков софт-процессора к тесному сотрудничеству с коллективом программистов — пользователей ядра приведет к получению проекта, хорошо адаптированного к конкретной практической задаче (возможно, и за счет ухудшения синтетических оценок).

Кроме процесса проектирования, отталкивающегося от разработки тех или иных аппаратных конструкций, для которых потом создается или адаптируется компилятор, можно рассматривать и противоположный процесс — разработку аппаратной архитектуры и регистровой модели, оптимальной для используемого компилятора или решаемых задач. В этом случае предполагается, что переход от промежуточного представления компилятора к машинному коду будет выполнен с минимальными дополнительными тактами процессора (в идеале — промежуточное представление программы должно транслироваться в машинный код без дополнительных команд перемещения или преобразования данных). Это направление нельзя считать полностью исследованным, поскольку понятия «оптимальная программа» или «оптимальный процессор» чересчур размыты. Под вопросом находятся и критерии оптимальности, и доступные алгоритмы оптимизации, к тому же и сам комплекс «процессор – компилятор – прикладная программа» допускает множество возможных преобразований и воздействий. Одна и та же проблема может быть решена как добавлением регистров, команд или связей в процессор, так и улучшением компилятора или же просто коррекцией прикладной программы. Огромный спектр задач, который уже сегодня осуществляется с помощью процессорных устройств, будет только расширяться, что позволит находить место все новым решениям в области процессорной техники.

Можно вкратце обобщить практические шаги для специалистов, заинтересованных в данной области:

1. Проектирование и моделирование.

Данный этап не потребует от специалиста материальных затрат. Можно использовать практически любой язык программирования высокого уровня для работы с программными моделями процессора и бесплатные версии САПР ПЛИС для экспериментов с RTL-описанием и моделирования на уровне регистровых передач. Компания Xilinx предоставляет бесплатную версию САПР Vivado [6], которая имеет ограничения только по объему используемых ПЛИС (в действительности планка ограничения установлена довольно высоко, что делает бесплатную версию вполне пригодной для работы). В итоге проектировщик сможет выявить эффективность принятых им технических решений, оценить тактовую частоту и объем ресурсов, которые будут заняты в ПЛИС.

2. Реализация в макете.

Простые отладочные платы на базе ПЛИС начального уровня вполне позволяют продемонстрировать практическую работоспособность процессора и создать примеры устройств, управляющих периферийным оборудованием. Данные платы доступны в России [7] и предоставляют доступ к актуальным семействам ПЛИС. При умеренных финансовых затратах можно продемонстрировать практическое функционирование процессорной системы.

3. Внедрение в практику.

Наличие информации о достижимых характеристиках процессора и возможных путях его доработки, расширения и адаптации позволяет говорить о его внедрении в практику, в первую очередь по месту основной работы в профильной организации. Как было упомянуто, важное практическое преимущество оригинальной архитектуры заключается в том, что она может быть быстро адаптирована для управления сложными IP-ядрами, расположенными на кристалле ПЛИС. Без удобного в применении процессора отладка таких ядер производилась бы с помощью длительных итераций моделирования, правки управляющих непрограммируемых контроллеров и генерации новых конфигурационных файлов. Управление сигналами IP-ядер со стороны встроенного в проект процессора по меньшей мере сократит количество итераций создания конфигурационных файлов. Замена MicroBlaze на новое ядро не является в данном случае самоцелью, однако вполне может быть произведена, если требуется реализовать нестандартные шинные циклы, использовать групповой доступ к регистрам или аппаратно ускорить работу с IP-ядрами на уровне их интерфейсов.

Заключение

Завершая данный цикл публикаций, посвященных разработке собственных процессорных ядер, можно пожелать читателям в первую очередь сформировать собственный целостный взгляд на рассмотренные вопросы. Развитие современной техники невозможно без вовлечения в этот процесс грамотных, заинтересованных и мотивированных специалистов, способных как поставить задачу, так и предложить индивидуальные пути ее решения.

Литература

1. LightWeight IP Application Examples. **www.xilinx.com/support/documentation/application_notes/xapp1026.pdf**
2. Ахо А. В., Лам М. С., Рави С., Ульман Д. Д. Компиляторы: принципы, технологии и инструментарий. 2-е изд. Пер. с англ. М.: ООО «И. Д. Вильямс», 2008.
3. **www.ecma-international.org/publications/standards/Ecma-335.htm**
4. **www.gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html**
5. Баранов С. Н., Ноздрунов Н. Р. Язык ФОРТ и его реализации. Серия «ЭВМ в производстве». М.: Машиностроение, 1988.
6. **www.xilinx.com**
7. **www.plis.ru**