Проектирование для ПЛИС Xilinx с применением языков высокого уровня в среде Vivado HLS

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

Увеличение логической емкости FPGA делает актуальным переход к новым системам проектирования, которые были бы способны обеспечить эффективное заполнение современных FPGA с приемлемой трудоемкостью. Языки описания аппаратуры, такие как VHDL и Verilog, недостаточно эффективны для решения этой проблемы при объемах программируемых микросхем в сотни тысяч логических ячеек. В настоящее время ведущий производитель ПЛИС — компания Xilinx — предлагает в составе своей САПР новый инструмент проектирования, основанный на языках высокого уровня, — Vivado HLS. В статье рассматриваются основы работы в этой САПР, а также проводится предварительный анализ областей ее применения в сравнении с другими средствами разработки для ПЛИС Xilinx.

Введение

Vivado HLS (High Level Synthesis) — новая САПР Xilinx, предназначенная для создания цифровых устройств с применением языков высокого уровня. Попытки использования таких языков (понимая под ними языки с Си-подобным синтаксисом) неоднократно предпринимались различными компаниями на протяжении последнего десятилетия. Основной целью таких продуктов было упрощение процесса проектирования для разработчика, знакомого с программированием.

Практическое использование ПЛИС часто вызывает трудности для «чистых» программистов, которые сталкиваются с целым рядом непривычных задач: необходимостью помнить о правильном формировании тактовых сигналов, учитывать латентность, а также вообще понимать, что операторы языков описания аппаратуры не вполне эквивалентны операторам языков программирования. Например, оператор сложения, естественный для программиста, в распространенных языках описания аппаратуры (HDL) требует соответствующего обрамления. Разные схемотехнические решения будут получены для случая, когда сложение выполняется в виде оператора непрерывного присваивания (continuous assignment) и внутри синхронного процесса. В то же время цифровая схемотехника по мере уменьшения норм технологического процесса все больше тяготеет к синхронным схемам. В итоге оказывается, что много усилий при проектировании на базе ПЛИС тратится на разработку управляющего автомата, активирующего устройства в нужные моменты времени.

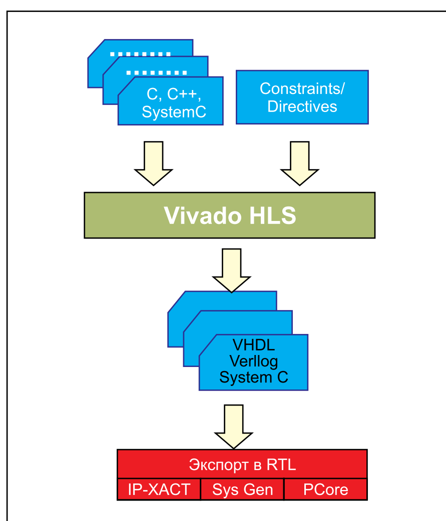

Рис. 1. Упрощенный маршрут проектирования в Vivado HLS

Зная необходимую последовательность операций, можно, применяя регулярные правила, построить соответствующий конечный автомат. В свою очередь, его построение можно автоматизировать, разработав соответствующую программу анализа исходного текста на языке высокого уровня. В начале 2000-х годов появилась серия программных продуктов, представлявших собой компиляторы, генерирующие VHDL или Verilog на уровне регистровых передач (RTL-уровень) из Си-подобного языка программирования. Получавшиеся файлы обычно можно было интегрировать в проект наравне с файлами, полученными другим способом.

Характеристики проектов, созданных с помощью инструментов C-to-RTL, по проводившимся обзорам, обычно отличали чуть бóльший объем ресурсов и чуть бóльшая тактовая частота. Это стало следствием того, что компиляторы разбивали сложные вычисления на элементарные операции, каждая из которых имела небольшую задержку. Соединение таких элементарных схем, управляемых конечным автоматом, давало в результате регулярную структуру из регистров, передававших друг другу данные через вычислительные узлы. Подобные структуры эффективно реализуются в FPGA.

Схожесть языков C-to-RTL с языком программирования Си не означает при этом, что такой инструмент способен создать проект в FPGA, реализующий вычисления, эквивалентные программе на Си для компьютера или микроконтроллера. Выбор Си в качестве основы объяснялся широкой распространенностью этого языка и снижением входного порога освоения для разработчиков, знакомых с программированием. Однако, за исключением простейших примеров, программы на Си должны подвергаться глубочайшей переработке для получения аппаратного решения, способного выступать в качестве ускорителя интересующих нас вычислений.

В 2011 году компания Xilinx объявила о выпуске продукта, относящегося к упомянутому классу инструментов, — преобразование Си-подобного исходного текста в эквивалентный код на одном из языков описания аппаратуры. Создаваемое RTL-представление представляет собой конечный автомат (то есть схему, выполненную в синхронном стиле, что предпочтительно для современных FPGA), управля-

ющий потоками вычислений. Результаты зависят от проектных ограничений (constraints), а также, причем в большой степени, от директив компилятора. Директивы также могут облегчить формирование интерфейсов создаваемого модуля: он может быть экспортирован в виде IP-ядра с интерфейсом, определенным пользователем, но также можно указать на необходимость его экспорта в виде периферийного модуля для шины AXI4 (для установки в процессорную систему) или модуля для инструмента System Generator for DSP. Упрощенный маршрут проектирования в Vivado HLS показан на рис. 1.

Основные сведения об HLS

Директивы HLS существенно сильнее влияют на RTL-представление по сравнению с тем, как директивы компилятора языка программирования влияют на скомпилированный код для процессора. Из одного исходного текста с помощью HLS можно получить несколько значительно различающихся схем.

Для анализа этой схемы необходимо определить два важных понятия:

- Пропускная способность (throughput) — задержка в тактах между поступлением новых входных данных.
- Латентность (latency) — задержка между поступлением входных данных и появлением соответствующих им выходных.

На рис. 2 эти понятия проиллюстрированы на примере простейшей конвейерной схемы. На показанной временной диаграмме видно, что каждым тактом входные данные сдвигаются на один регистр, освобождая при этом вход для приема новых данных. Таким образом, пропускная способность такой схемы составляет один такт, поскольку новые данные можно подавать каждым новым тактом. Латентность равна количеству триггеров в цепочке (в показанном случае это два такта).

Эти понятия играют важную роль при разработке высокопроизводительных цифровых схем. При построении сложных комбинаторных схем период тактового сигнала определяется самой длинной цепью, соединяющей синхронные компоненты. Поэтому очевидным способом повышения тактовой частоты (то есть уменьшения периода тактового сигнала) является разделение длинных цепей триггерами. При этом, очевидно, увеличивается латентность схемы, однако это является побочным эффектом на пути к достижению более приоритетной цели — увеличению тактовой частоты при сохранении пропускной способности, равной одному такту между новыми входными отсчетами.

На рис. 3 показан исходный текст, который содержит цикл с четырьмя итерациями. По умолчанию создается схема, воспроизводящая поведение процессора общего назначения: все итерации цикла выполняются последовательно. Такая реализация занимает наименьшую площадь на кристалле FPGA (то есть использует наименьшее количество ресурсов), однако требует большого числа тактов от начала работы до получения результатов (то есть имеет большую латентность). Кроме того, поскольку все итерации выполняются одним и тем же вычислительным блоком, невозможно начать новый цикл расчета до полного завершения текущего цикла. Таким образом, этот вариант имеет малую пропускную способность.

По аналогии с разворачиванием цикла в компиляторах языков программирования такой цикл также можно развернуть (этот прием называется loop unrolling). При этом каждая итерация цикла рассчитывается индивидуальным элементом. За счет одновременного выполнения всех итераций латентность становится существенно меньше, однако количество ресурсов, требуемое для реализации такой схемы, возрастает практически пропорционально.

Дальнейшим улучшением такой схемы является конвейеризация вычислений. Если промежуточные результаты будут записываться в регистры, то это освободит соответствующую часть ресурсов для выполнения вычислений над следующей порцией данных. Латентность при этом не изменяется, но повышается пропускная способность. Логические ячейки FPGA, так же как и блочные ресурсы (память и блоки XtremeDSP), имеют регистры на выходе, поэтому глубокая конвейеризация вполне допустима для FPGA.

Исходя из представленного примера, можно определить приоритеты при разработке высокопроизводительных схем. В первую очередь, следует улучшать пропускную способность, вплоть до одного такта на входной отсчет. При этом номинальное значение тактовой частоты повышается путем более глубокой конвейеризации, что автоматически дает побочный эффект в виде увеличения

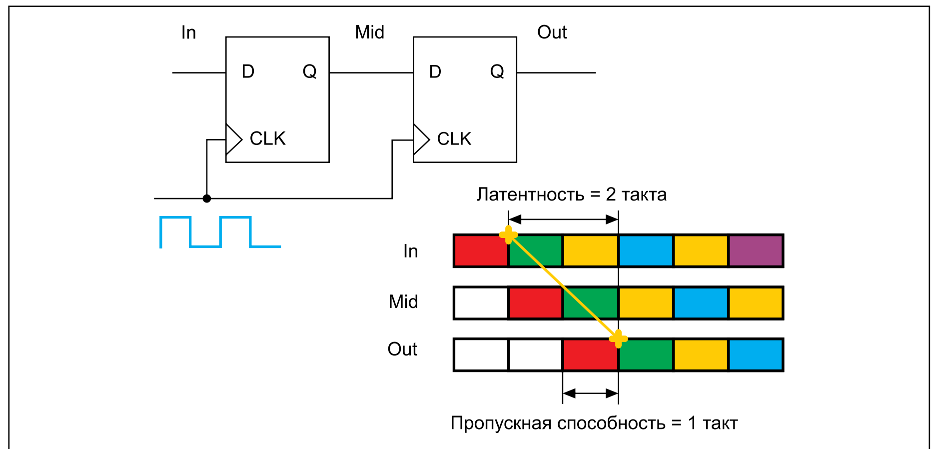

Рис. 2. Понятия «латентность» и «пропускная способность»

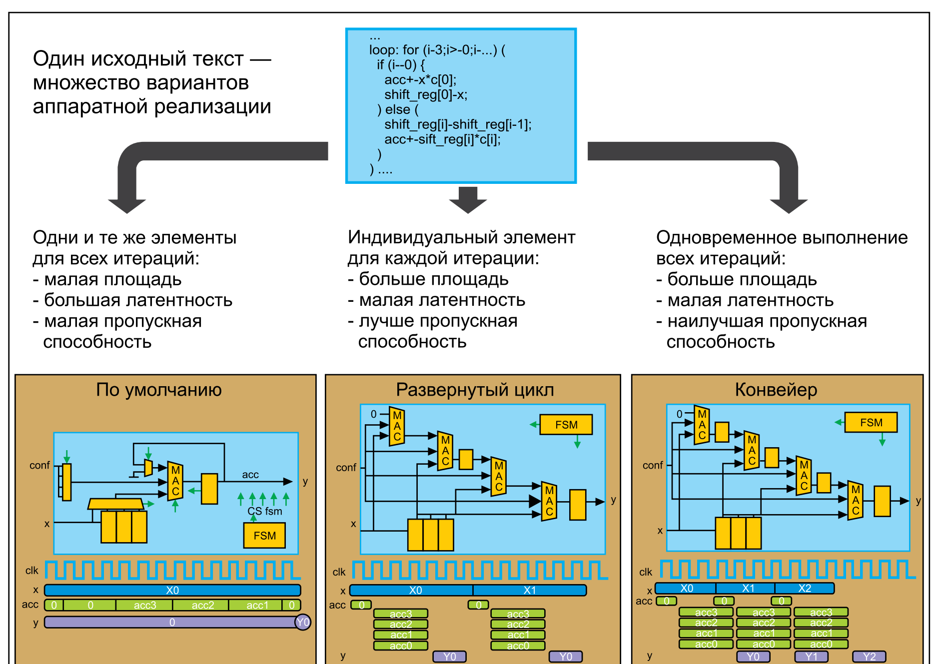

Рис. 3. Основы Vivado HLS — управление синтезом с помощью директив

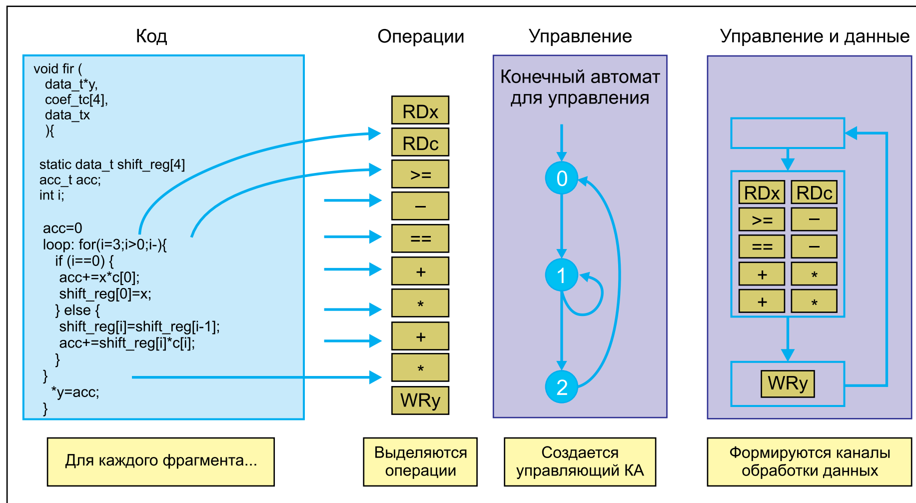

Рис. 4. Принципы автоматического создания управляющих автоматов в Vivado HLS

латентности. Второстепенная цель — сохранение латентности на разумно малом уровне, при этом нельзя допустить появления избыточных стадий конвейера, которые в действительности не способствуют повышению тактовой частоты. В качестве примера такой избыточной конвейеризации легко представить вычислительное устройство, на выходе которого установлено несколько последовательно включенных регистров. Верхняя граница тактовой частоты такого устройства определится теми компонентами, которые осуществляют преобразование данных, и добавление регистров на выходе не уменьшает задержку распространения сигналов внутри вычислителя.

Для выполнения вычислений требуется обеспечить передачу сигналов данных от одного вычислительного узла к другому в определенные моменты времени. Vivado HLS автоматически выделяет из исходного текста и пути данных, и последовательность управления работой (рис. 4). В примере показано, что автомат управления включает в себя начальное состояние 0, состояние циклического выполнения операций 1 и состояние завершения работы 2. По мере необходимости операции с данными могут быть распараллелены.

Вопрос параллельного выполнения операций важен для FPGA как таковых. Именно возможность организации большого количества параллельно работающих модулей является принципиальным конкурентным преимуществом FPGA как аппаратной платформы. Известен факт, что именно FPGA способны обеспечить реализацию устройства с уникальной аппаратной архитектурой. Однако это преимущество иногда полагается единственным. В действительности при корректном использовании FPGA способны обеспечить за счет параллельной работы не только высокую абсолютную производительность, но и соотношение производительность/цена, превышающее это соотношение для сигнальных процессоров и процессоров общего назначения. Например, FPGA семейства Kintex-7 начального уровня имеет 240 независимых блоков, выполняющих операцию «умножение с накоплением». Учитывая, что системная тактовая частота для Kintex-7 достигает 700 МГц, можно получить 168 GMAC/с. Это значение можно существенно скорректировать с учетом снижения частоты из-за особенностей трассировки и неполного использования ресурсов, однако вполне можно ожидать значения в 100 GMAC/с для устройства, выполняющего многоканальную цифровую фильтрацию или спектральный анализ. Можно представить, что суммарная стоимость процессоров, способных обеспечить подобную производительность (100 ГГц при условии, что на каждом такте будут выполняться только операции цифровой обработки сигналов), окажется существенно выше, чем стоимость одной ПЛИС 7K70.

Исходя из этого, становится очень важным не только выбрать алгоритм, который мог бы эффективно использовать особенности FPGA, но и реализовать его так, чтобы в проекте не появлялись линии с чрезмерно большой задержкой, по которым и будет выравнена тактовая частота. Важно, что эту работу Vivado HLS выполняет автоматически. На рис. 5 показаны варианты схемы, реализуемой для различных аппаратных платформ. Поскольку одним из параметров проекта в HLS является требуемая тактовая частота, САПР может оценить, укладывается ли синтезированная схема в эти параметры. Для первого варианта схемы для достижения высокой тактовой частоты требуется глубокая конвейеризация, поэтому четыре строки исходного текста преобразуются в четыре операции. Соответственно, латентность такой схемы равна четырем. Однако при переходе к аппаратной платформе с более высокой производительностью (например, с Artix на Kintex) задержки при выполнении отдельных операций оказываются меньше, поэтому за один такт удается выполнить две операции. Тактовая частота схемы сохраняется на требуемом уровне, однако латентность уменьшается до двух.

Таким образом, при генерации схемы Vivado HLS действует следующим образом:

- Основным приоритетом является повышение пропускной способности схемы при соблюдении проектных ограничений по тактовой частоте, задаваемой пользователем.
- Следующим по важности приоритетом является снижение латентности.
- По возможности минимизируются занимаемые проектом ресурсы.

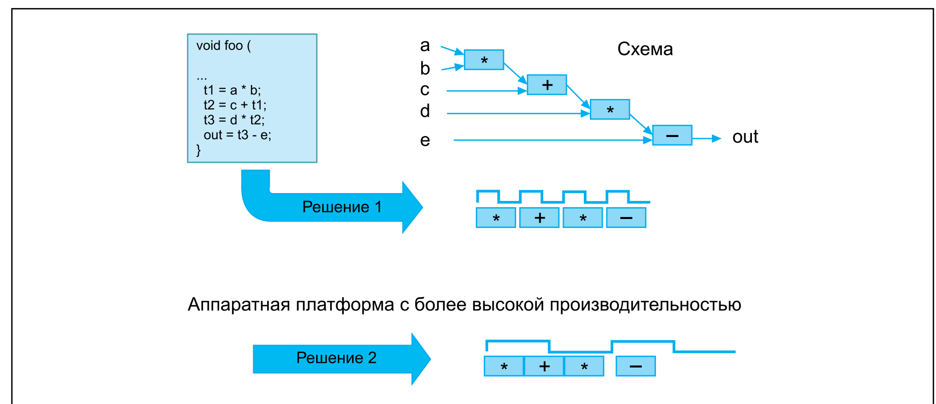

Рис. 5. Влияние характеристик аппаратной платформы на формирование управляющего автомата

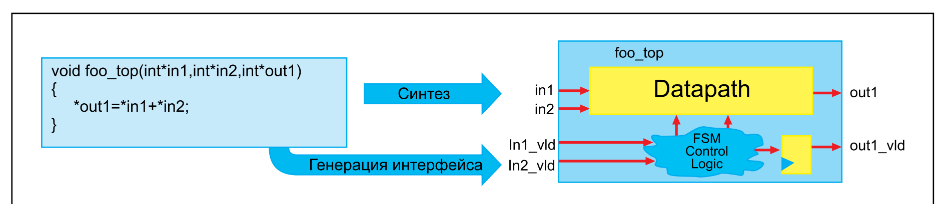

Рис. 6. Формирование аппаратных интерфейсов для блоков

Другой задачей, которую автоматизирует Vivado HLS, является генерация интерфейсов с учетом управляющих сигналов. На рис. 6 показан исходный текст функции и схема сгенерированного модуля для реализации этой функции. Кроме сигналов, объявленных в качестве параметров функции, в модуле имеются также сигналы с суффиксами vld, от слова valid («готовность»). Отдельные блоки создаются из функций, описанных в С-коде. Функции могут быть «развернуты» для упрощения иерархии (inlining). Для небольших функций это преобразование выполняется автоматически.

Циклы в Vivado HLS

Циклы по умолчанию реализуются итеративно: каждая итерация выполняется последовательно на базе одних и тех же ресурсов. Это обеспечивает поведение ПЛИС, схожее с поведением программ для ЭВМ: увеличение количества итераций означает увеличение времени работы, а объем ресурсов, необходимых для этой операции, остается постоянным. Реализацией циклов можно управлять с помощью директив. Для идентификации цикла в директиве каждый цикл должен начинаться с метки, как показано на рис. 7. На том же рисунке приведена схема преобразования текста на Си в эквивалентную схему. Видно, что все итерации суммирования выполняются одним и тем же сумматором. При необходимости цикл можно развернуть (unroll), однако для этого требуется, чтобы число итераций цикла было известно на этапе синтеза. (Переменное число итераций не может быть развернуто.)

Пример «идеального цикла», приведенный в САПР Vivado HLS, показан в листинге 1. В нем видно, что оба уровня вложенного цикла имеют метки (что позволяет устанавливать индивидуальные директивы для каждого уровня), а число итераций является константой.

```
void loop_perfect(din_t A[N], dout_t B[N]) {

 int i,j;
   dint_t acc;

   LOOP_I: for(i=0; i < 20; i++){
      LOOP_J: for(j=0; j < 20; j++){
         if(j==0) acc = 0;
         acc += A[i] * j;
         if(j==19) B[i] = acc / 20;
      }
   }
}
```

Листинг 1. Пример «идеального цикла» (perfect loop) в Vivado HLS

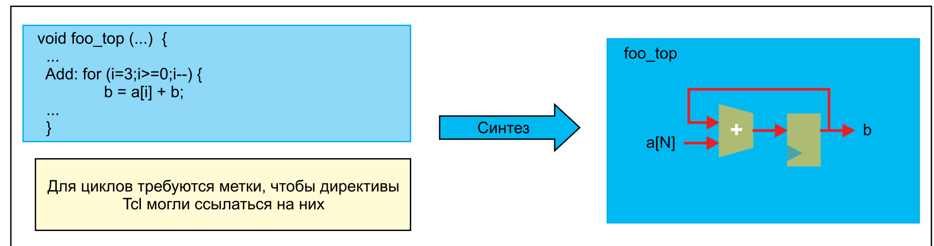

Рис. 7. Реализация циклов в HLS

Этот пример хорошо подходит для иллюстрирования влияния директив компилятора на результаты синтеза RTL-представления. Уже было упомянуто, что директива UNROLL приводит к разворачиванию цикла: его итерации начинают выполняться параллельно на собственном наборе оборудования. Количество тактов на выполнение всего цикла уменьшается за счет роста размера схемы. В принципе такой эффект и лежит в основе одного из главных преимуществ FPGA, имеющих большое количество ресурсов. В таблице приведены основные результаты синтеза примера из листинга 1 при разном наборе директив компиляции.

Результаты синтеза без дополнительных директив предсказуемы. Цикл имеет в общей сложности 400 итераций (20 на внешнем и 20 на вложенном уровне). Добавляя латентность на вспомогательные пересылки, можно объяснить появление значения 403 в таблице. Это решение имеет минимальный размер.

Установив директиву HLS_UNROLL для внутреннего цикла (LOOP_J), можно наблюдать ожидаемое уменьшение латентности. В цикле используется умножение на значение счетчика:

```
acc += A[i] * j
```

Это не потребовало привлечения дополнительных блоков DSP.

Следующий шаг эксперимента — установка директивы UNROLL только для внешнего цикла. Однако результаты оказываются, на первый взгляд, неожиданными: при резком увеличении ресурсов латентность не только не уменьшилась, но и несколько увеличилась. Внешний цикл действительно оказался развернут, и каждая итерация получила собственный экземпляр устройства умножения. Однако в этом случае каждая итерация внешнего цикла ожидает завершения работы вложенного цикла, поэтому для работы требуется целых 460 тактов.

Ожидаемого эффекта удается добиться при установке директивы UNROLL для обоих уровней цикла. Объем ресурсов существенно возрастает: вместо двух блоков DSP для проекта требуется 80, а также резко увеличилось количество LUT. Однако латентность составила всего 11 тактов! Если поставленная цель состояла в том, чтобы максимально задействовать ресурсы FPGA для сокращения времени работы, то она оказалась практически достигнута. Использованная в примере FPGA Kintex-7 XC7K160 имеет 600 блоков DSP и 101 тыс. LUT. При полном разворачивании циклов в проекте оказалось задействовано 13 и 6% этих ресурсов соответственно.

Таким образом, с помощью директив компилятора можно весьма широко изменять параметры проекта, регулируя количество выделенных на него ресурсов и латентность. Большая гибкость в установлении директив делает процесс их внедрения в проект не таким однозначным, поскольку далеко не все сочетания директив приводят к приемлемым результатам. Поэтому разработчики должны обращать дополнительное внимание на характеристики, достигаемые при текущем сочетании настроек, и рассматривать различные варианты построения проекта, стиля кодирования и установки директив компиляции.

Массивы и работа с памятью

Массивы в HLS реализуются на базе памяти. Если массив выступает в качестве аргумента функции верхнего уровня, память считается размещенной вне ПЛИС. При необходимости память может быть разбита на более мелкие блоки для реализации на базе триггеров и распределенной памяти (distributed RAM). HLS может использовать двухпортовый режим блочной памяти.

Использованию памяти в HLS следует уделять особое внимание. Обычно программист для PC не рассматривает память в качестве отдельного ресурса, поскольку все используемые данные автоматически помещаются в основную память компьютера. При оптимизации программ речь может идти о временном размещении каких-то значений в регистрах, учете особенностей кэширования и т. п. Однако для ПЛИС возможности размещения данных существенно шире.

Таблица. Результаты синтеза проекта из листинга 1

Набор директив	Период, нс (требуемый период — 25 нс)	Латентность, тактов	Логических генераторов (LUT)	Блоков DSP48
Без дополнительных директив	19,94	403	129	2
HLS_UNROLL для внутреннего цикла	19,98	61	326	4
HLS_UNROLL для внешнего цикла	19,94	460 (?)	1920	40
HLS_UNROLL для внутреннего и внешнего циклов	19,98	11 (!)	6360	80

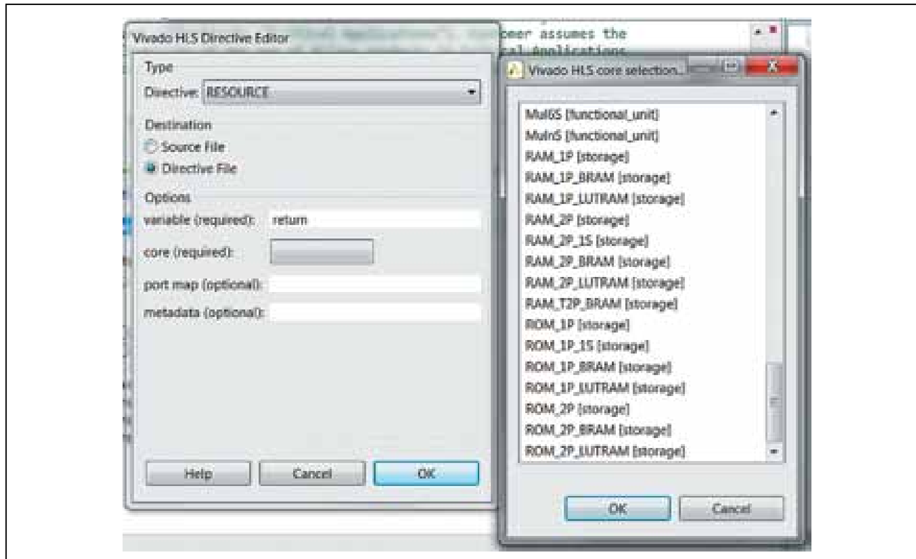

Рис. 8. Установка типа ресурса для массива

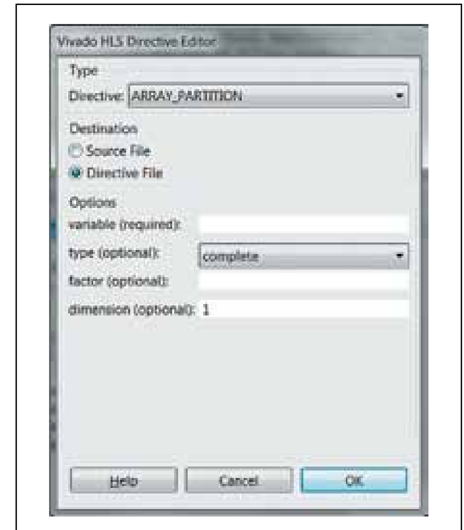

Рис. 9. Настройка параметров директивы ARRAY_PARTITION

Во-первых, память может быть физически реализована в виде следующих аппаратных ресурсов:

- триггеры логических ячеек;
- LUT секций типа SliceM (например, распределенная память);
- в блоках памяти BRAM.

Отдельно можно рассматривать внешнюю память. Она не является частью кристалла FPGA, однако участвует в работе проекта, в то время как на FPGA размещен контроллер для доступа к ней.

Во-вторых, важным преимуществом FPGA является крайне высокая теоретическая пропускная способность подсистемы памяти в целом. BRAM представляет собой обычную статическую память (то есть это не программируемый блок, а фрагмент кремниевой пластины с жесткими металлизированными соединениями). Блоки имеют полностью независимые интерфейсы и могут работать на системной тактовой частоте. Таким образом, теоретически FPGA большого объема способны обеспечить пропускную способность порядка сотен гигабайт в секунду, что оставляет далеко позади интерфейсы внешней памяти, в том числе DDR3. Однако важнейшим вопросом при этом является то, способен ли проект получить преимущество от множества параллельно работающих BRAM и может ли вообще быть обеспечен такой поток данных внутри кристалла.

Из-за этих факторов необходимо тщательно продумывать распределение данных, чтобы обеспечить их размещение в ресурсе такого типа, который позволит максимально эффективно использовать особенности ПЛИС. При программировании также необходимо следить, чтобы выбранные синтаксические конструкции не сделали невозможным размещение данных в памяти, которую имеет в виду разработчик. Можно констатировать, что в настоящее время результаты работы компилятора HLS сильно зависят как от особенностей исходного текста, так и от использованных директив. У программиста, знакомого с C, может сложиться впечатление, что результаты работы HLS в ряде случаев весьма далеки от совершенства.

Разработчик, работающий с памятью в языках программирования высокого уровня, не нуждается в решении такой задачи, поскольку независимо от вида массивов и порядка обращения к ним пропускная способность памяти компьютера или микроконтроллера ограничена, причем чаще всего единственным интерфейсом. Вопросы распределения данных по областям памяти могут оказаться актуальными для редких случаев многоядерных систем, имеющих сложную структуру доступа к памяти по физически раздельным шинам.

В отличие от таких систем для эффективной работы с памятью в FPGA необходимо выбрать оптимальный способ использования большой суммарной пропускной способности интерфейсов независимых блоков памяти. Например, если все данные будут в целях экономии ресурсов размещены в одном блоке памяти, производительность такого блока будет ограничена двумя операциями за такт (в силу того, что блочная память в FPGA является двухпортовой). Если же распределить программные объекты по разным блокам памяти, то за один такт каждый из этих блоков будет способен выполнить две операции. Как и для блоков DSP, вопрос заключается в увеличении количества операций за такт путем вовлечения в проект все большего объема ресурсов FPGA. Конкретный тип ресурса для программного объекта HLS можно установить директивой RESOURCE, как показано на рис. 8.

Для гибкого управления памятью используется ряд директив. Директива ARRAY_PARTITION служит для разделения одного массива на несколько меньшего размера. Пример настройки этой директивы показан на рис. 9. Параметр dimension («размерность») позволяет указать, какая размерность должна быть подвергнута разделению. Например, для массива

```
my_array[10][6][4]
```

установка этого параметра в 1 заставит компилятор разделить массив на 10 независимых массивов (по максимальному значению первого индекса), каждый из которых будет иметь размерность [6] [4]. Установка параметра в 0 приведет к синтезу 10×6×4 = 240 независимых регистров.

Директива RESHAPE выполняет обратное действие — объединяет несколько независимых областей данных в один массив. В сочетании с предыдущей директивой она позволяет управлять размещением данных в физических блоках, увеличивая или уменьшая количество независимых шин, с помощью которых выполняется доступ к данным.

Директива PACK группирует данные, например составляющие элемент структуры, формируя таким образом более широкую шину для одновременного доступа ко всем ее элементам.

Использование этих директив для различных объектов не однозначно. Крайними ситуациями являются чрезмерно низкая производительность шин для пересылки данных, с одной стороны, и перегруженность FPGA независимыми блоками, приводящая к чрезмерному расходованию ресурсов, с другой. Выбор оптимальной организации размещения данных индивидуален для каждого проекта. Гибкость архитектуры FPGA и огромное количество возможных вариантов (далеко не все из которых сопоставимы в смысле эффективности) не позволяет слишком надеяться на то, что компилятор самостоятель-

но выберет оптимальный способ размещения данных при любых операциях над ними, которые захочет выполнить разработчик.

Поддержка языков высокого уровня

В Vivado HLS можно использовать следующие языки высокого уровня: C, C++, SystemC. Они имеют схожий синтаксис, поэтому освоение группы «Си-подобных языков» обычно не составляет большого труда. Однако с формальной точки зрения C и C++ представляют собой разные языки, на что обращают внимание многие авторы литературы по программированию. То же относится и к SystemC.

Важным отличием от языков программирования является поддержка так называемых arbitrary-precision типов, то есть типов данных с явно указываемой разрядностью. Это отличие считается таким важным потому, что при программировании разработчик имеет дело с процессором, обрабатывающим данные фиксированной разрядности, определяемой конструкцией этого процессора. Изменение разрядности с точки зрения программы возможно, однако редко целесообразно. В противоположность этому при проектировании цифровых систем разработчик имеет возможность явно указать разрядность обрабатываемых данных, которая наилучшим образом соответствует решаемой задаче. Для разных языков типы отличаются:

- C: (u)int (с разрядностью 1–1024);
- C++: ap_(u)int (с разрядностью 1–1024), ap_fixed;
- SystemC: sc_(u)int, sc_fixed.

Типы с точным указанием разрядности позволяют устранять избыточность ресурсов, например, по сравнению с типом int, разрядность которого равна 32. Эта избыточность может оказаться существенной. Если рассмотреть архитектуру блока DSP, то видно, что такой блок способен умножить 18-разрядное число на 25-разрядное. Если же разрядность множителей будет не зафиксирована явно на достаточном для задачи уровне, а выбрана равной 32 (автоматически, в результате использования типа int), для построения схемы будут привлечены дополнительные аппаратные ресурсы.

Vivado HLS позволяет синтезировать многие конструкции C/C++/SystemC, при условии, что их параметры известны во время компиляции (compile time). По понятным причинам не поддерживается синтез для описаний, элементы которых становятся известны только во время исполнения (run time). Кроме того, не синтезируются:

- функции динамического управления памятью (***malloc/free***);
- операции ввода/вывода (***printf/scanf***);
- системные вызовы (опрос таймера).

В синтезируемом коде поддерживаются указатели (pointers). Как правило, они приводят к синтезу схемы, в которой указатель выступает как адрес устройства памяти. (Само устройство физически может представлять собой компонент любого доступного типа — регистр, распределенная или блочная память.) Указатели полезны, когда необходимо передать ссылку на объект большого размера (например, pass-by-reference). В этом случае передается адрес объекта, а не весь объект целиком. Гибкость архитектуры FPGA в принципе позволяет передать 1024 бита в качестве 128 8-разрядных элементов (что произойдет в случае передачи объекта «по значению» — pass-by-value), но такая схема скорее всего будет избыточной для многих практических применений (хотя и позволит обрабатывать все 128 значений параллельно).

При использовании указателей необходимо обращать дополнительное внимание на получаемые результаты. Вообще при работе с HLS необходимо помнить, что FPGA имеют гибкую архитектуру, позволяющую реализовать требуемое поведение множеством способов, в отличие от выполнения программы на процессоре или микроконтроллере. Стиль программного кода, используемые синтаксические конструкции и директивы компилятора способны оказать существенное влияние на получаемые результаты.

Работа с HLS

Vivado HLS представляет собой отдельный программный продукт, который может быть установлен в рамках инсталляции общего пакета САПР Xilinx, но имеет собственную среду разработки. При запуске Vivado HLS появляется стартовый экран, как показано на рис. 10. На этом экране можно как создать новый проект или открыть существующий, так и ознакомиться с готовыми примерами проектов и документацией по САПР.

Основные шаги по созданию проекта удобно рассмотреть на практическом примере, содержащем простейшую схему. На рис. 11 показано стартовое диалоговое окно «мастера» создания нового проекта.

Рис. 10. Стартовый экран Vivado HLS

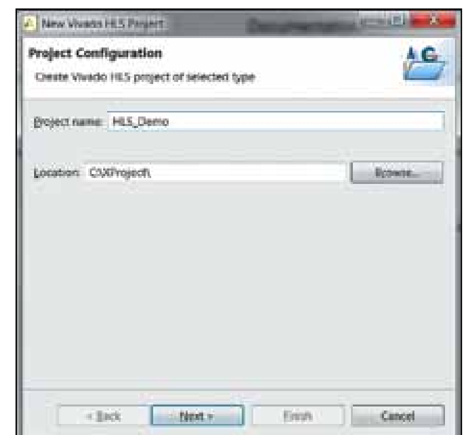

Рис. 11. Стартовое диалоговое окно «мастера» создания нового проекта

На рис. 12 приведена настройка синтезируемой части проекта. Для синтеза функция верхнего уровня не может называться ***main()***, как это принято в C; такое имя предназначено для верхнего уровня модели.

Далее можно пропустить шаг добавления тестовых модулей, работа с которыми

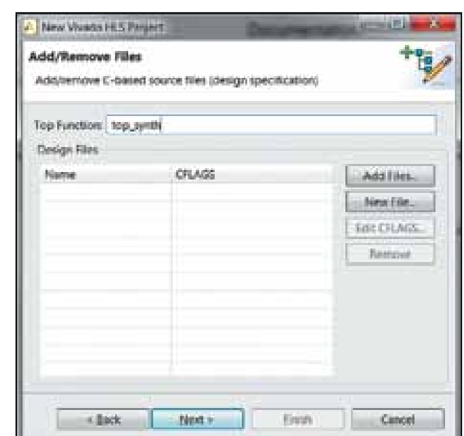

Рис. 12. Диалоговое окно настройки синтезируемой части проекта

Рис. 13. Настройка решения (solution)

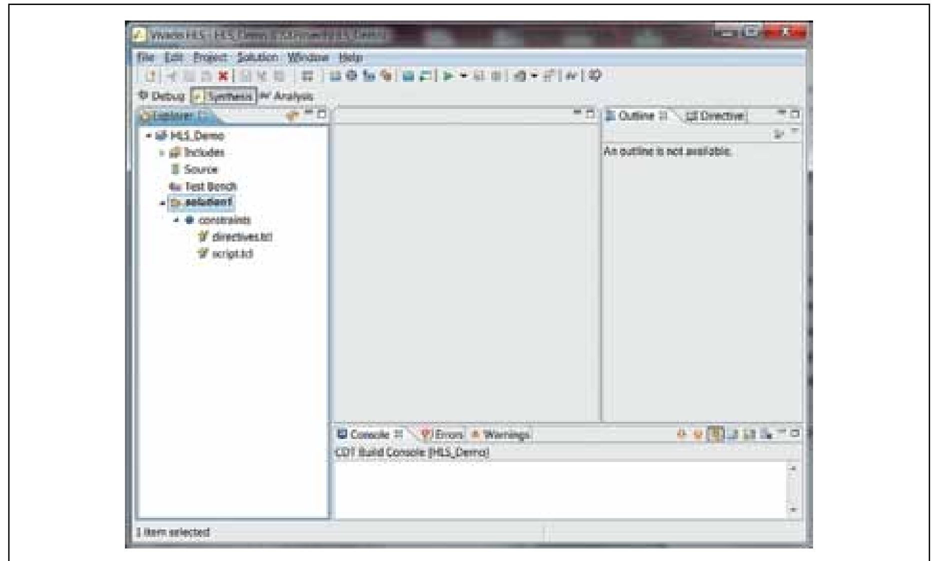

Рис. 14. Окно САПР Vivado HLS после завершения работы «мастера» создания проекта

будет рассмотрена позднее. В следующем окне настройки решения (solution), показанном на рис. 13, следует указать желаемый период тактового сигнала для проекта и выбрать конкретное наименование FPGA. Как уже было упомянуто, эта информация не является абстрактной, а служит для выбора конкретных схемотехнических решений. Если выбранная FPGA имеет недостаточное быстродействие, для схемы будет синтезирован конвейер с бо́льшим числом стадий. Соответственно, каждая стадия будет выполнять более простую операцию, и общая задержка окажется меньше. В приведенном примере для определенности можно выбрать плату KC-705 на базе FPGA Kintex-7. После завершения работы «мастера» окно САПР будет выглядеть, как показано на рис. 14.

Для текущего проекта необходимо добавить синтезируемый файл верхнего уровня. Это можно сделать с помощью главного меню или же из контекстного меню, вызываемого правой кнопкой мышки в области структуры проекта (панель **Explorer**).

Файл верхнего уровня *demo.c* может содержать текст, показанный в листинге 2.

```
int top_synth(int a, int b)
{
    return a + b;
}
```

Листинг 2. Основной синтезируемый файл проекта demo.c

На инструментальной панели имеются кнопки, запускающие моделирование и синтез соответственно. После завершения синтеза будет показан отчет с возможностью детализации по отдельным показателям (например, распределения ресурсов ПЛИС по отдельным элементам программы на HLS), что можно видеть на рис. 15.

После завершения синтеза в панели Explorer можно изучить синтезированный код. В группе ***solution → syn → vhdl*** находится сгенерированный модуль на языке VHDL, который может быть передан в другие инструменты проектирования Xilinx. Содержимое этого модуля показано в листинге 3.

```
-- ==============================================================
-- RTL generated by Vivado(TM) HLS
-- High-Level Synthesis from C, C++ and SystemC
-- Version: 2013.3
-- Copyright (C) 2013 Xilinx Inc. All rights reserved.
--
-- ==============================================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity top_synth is
port (
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    a : IN STD_LOGIC_VECTOR (31 downto 0);
    b : IN STD_LOGIC_VECTOR (31 downto 0);
    ap_return : OUT STD_LOGIC_VECTOR (31 downto 0) );
end;

architecture behav of top_synth is
    attribute CORE_GENERATION_INFO : STRING;
    attribute CORE_GENERATION_INFO of behav : architecture is
"top_synth,hls_ip_2013_3,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xc7k325tffg900-2,HLS_INPUT_CLOCK=10.000000,HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=1.600000,HLS_SYN_LAT=0,HLS_SYN_TPT=none,HLS_SYN_MEM=0,HLS_SYN_DSP=0,HLS_SYN_FF=0,HLS_SYN_LUT=0}";
    constant ap_const_logic_1 : STD_LOGIC := '1';
    constant ap_const_logic_0 : STD_LOGIC := '0';

begin

    ap_done <= ap_start;
    ap_idle <= ap_const_logic_1;
    ap_ready <= ap_start;
    ap_return <= std_logic_vector(unsigned(b) + unsigned(a));

end behav;
```

Листинг 3. Результаты генерации RTL-представления проекта на языке VHDL

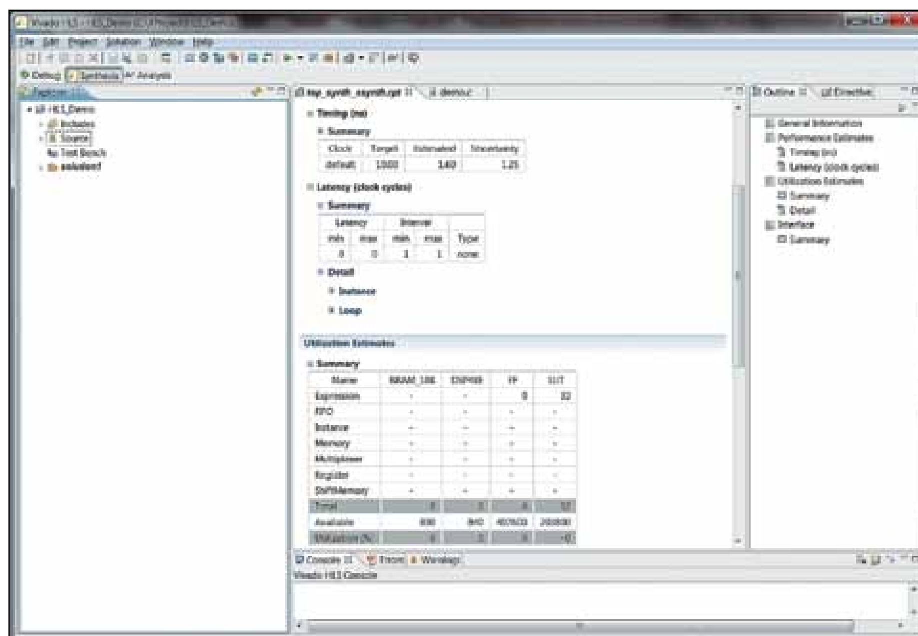

Рис. 15. Окно САПР после завершения синтеза схемы и формирования отчета

Несмотря на простоту, этот пример позволяет продемонстрировать некоторые особенности Vivado HLS. Собственно вычисление выполняется в строке:

```
ap_return <= std_logic_vector(unsigned(b) + unsigned(a));
```

Однако результаты сопровождаются сигналами готовности (***ap_ready***) и завершения работы (***ap_done***). Разрядность обрабатываемых сигналов автоматически установлена равной 32, поскольку для них указан тип ***int***.

Далее в проект можно добавить тестовый модуль, с помощью которого можно будет проверить работу синтезированного проекта. Тестовый модуль является элементом более высокого уровня по отношению к синтезируемому, но он также может обращаться и к другим файлам, содержащим несинтезируемый код. Пример простейшего теста показан в листинге 4.

```
#include "stdio.h"

int main()
{
   int result;
   result = top_synth(2, 2);
   printf("Testing 2 + 2 = %d\n\r", result);
   return 0;
}
```

Листинг 4. Тестовый файл для моделирования поведения синтезированного модуля

Запуск модели (кнопка **Run C Simulation** на инструментальной панели) приведет к появлению в консоли сообщений от компилятора, а также сообщений, выводимых тестовой программой (рис. 16). Показанный в листинге 4 пример очевиден, хотя и не может рассматриваться как серьезный тест. Моделирование, претендующее на адекватный анализ характеристик синтезируемого описания, должно включать в себя анализ результатов. Рекомендуется разрабатывать, например, самопроверяющиеся (self-checking) модели, которые проверяют синтезируемые функции не с помощью вручную написанных последовательностей вызовов, а подавая на вход заранее сгенерированные последовательности данных и сравнивая их с так же заранее определенными эталонными откликами. С помощью такого подхода можно проверять большое количество вариантов (ограниченное в основном допустимым временем тестирования), а изменения тестов сводятся к изменению файлов с данными, а не кода на C. Создание файлов с входными данными и эталонными откликами можно автоматизировать, что еще больше повысит эффективность работы и обеспечит лучшее покрытие проекта тестовыми примерами.

Простейший пример, призванный продемонстрировать порядок действий с САПР, не дает сколько-нибудь полного представления о практических возможностях этого инструмента и его назначении. Применимость языков высокого уровня и качество автоматически генерируемых схем необходимо анализировать на более сложных проектах. В листинге 5 приведен исходный текст модуля верхнего уровня для модуля быстрого преобразования Фурье (БПФ, в англоязычном варианте — FFT). Этот проект входит в состав примеров, включенных в Vivado HLS.

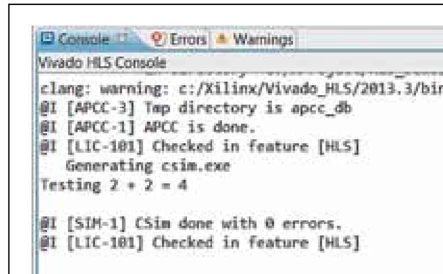

Рис. 16. Консоль Vivado HLS после завершения работы теста

```
#include "fft_top.h"

void dummy_proc_fe(
   bool direction,
   config_t* config,
   cmpxDataIn in[FFT_LENGTH],
   cmpxDataIn out[FFT_LENGTH])
{
   int i;
   config->setDir(direction);
   config->setSch(0x2AB);
   for (i=0; i< FFT_LENGTH; i++)
      out[i] = in[i];
}

void dummy_proc_be(
   status_t* status_in,
   bool* ovflo,
   cmpxDataOut in[FFT_LENGTH],
   cmpxDataOut out[FFT_LENGTH])
{
   int i;
   for (i=0; i< FFT_LENGTH; i++)
      out[i] = in[i];
   *ovflo = status_in->getOvflo() & 0x1;
}

void fft_top(
   bool direction,
   complex<data_in_t> in[FFT_LENGTH],
   complex<data_out_t> out[FFT_LENGTH],
   bool* ovflo)
{
#pragma HLS interface ap_hs port=direction
#pragma HLS interface ap_fifo depth=1 port=ovflo
#pragma HLS interface ap_fifo depth=FFT_LENGTH port=in,out
#pragma HLS data_pack variable=in
#pragma HLS data_pack variable=out
#pragma HLS dataflow
   complex<data_in_t> xn[FFT_LENGTH];
   complex<data_out_t> xk[FFT_LENGTH];
   config_t fft_config;
   status_t fft_status;

   dummy_proc_fe(direction, &fft_config, in, xn);
   // FFT IP
   hls::fft<config1>(xn, xk, &zfft_status, &fft_config);
   dummy_proc_be(&fft_status, ovflo, xk, out);
}
```

Листинг 5. Исходный текст модуля верхнего уровня быстрого преобразования Фурье

Сгенерированный проект обладает следующими характеристиками:

- FPGA — XC7K160TFBG484-1;
- заданный период тактового сигнала — 3,30 нс;
- оценка периода тактового сигнала средствами Vivado HLS — 2,89 нс (346 МГц);
- латентность — 3196–5247 тактов;
- число триггеров — 9934;
- число LUT — 8051.

Сгенерированный проект имеет высокую оценку тактовой частоты. В частности, это является следствием того, что в исходном тексте фактически вызвана функция, ссылающаяся на готовое IP-ядро, выполняющее быстрое преобразование Фурье:

```
hls::fft<config1>(xn, xk, &fft_status, &fft_config);)
```

Также можно обратить внимание на то, что латентность указана в виде интервала значений. Такое поведение HLS весьма важно для последующего анализа его применения.

Интервальный характер латентности связан с тем, что для компонентов проекта, генерируемых HLS, используется механизм «рукопожатия» (handshaking). Выше можно было видеть, что каждое ядро имеет сигналы готовности к приему новых данных, а выходные данные сопровождает сигналом их достоверности. Таким образом, каждый последующий компонент в цепочке обработки начинает работать, когда все входные данные сопровождены соответствующими сигналами готовности, а сам компонент свободен для обработки следующей порции входных данных. Существует много алгоритмов, подразумевающих переменное число тактов для завершения работы, поэтому итоговая латентность и оказывается переменной.

Vivado HLS в сравнении с другими средствами проектирования Xilinx

С учетом изложенного выше материала для практикующих разработчиков остро встает целый ряд вопросов. Какое же место занимает новая САПР в ряду инструментов проектирования для FPGA? Представляет ли она собой безусловно прорывное решение, которое призвано вытеснить ставший привычным маршрут проектирования на основе RTL-описаний? Является ли такое вытеснение, если оно состоится, вопросом ближайшего времени или же отдаленной перспективой? Существуют ли задачи, которые уже сейчас эффективно могут быть решены с использованием HLS, и на какие особенности проекта должен обратить внимание разработчик, чтобы усилия по освоению HLS оказались оправданными?

При сравнении необходимо сразу обратить внимание, что Xilinx позиционирует Vivado HLS в качестве инструмента разработки устройств цифровой обработки сигналов. Действительно, в HLS появилась возможность использования математических выражений и управляющих конструкций C-подобных языков программирования. При разработке в RTL-стиле первым же психологическим барьером была необходимость понять, что операторы HDL выполняются не последовательно, по мере их появления в исходном тексте,

а одновременно. В HLS же именно эта особенность разработки на ПЛИС была устранена, и появилась возможность привлечения к проектам программистов.

Почему при этом делается упор на разработку систем цифровой обработки сигналов? Прежде всего потому, что применение HLS само по себе не гарантирует качественного роста производительности. Далеко не каждая программа, написанная на C, может быть эффективно реализована в ПЛИС. Функциональности компилятора HLS недостаточно, чтобы для произвольного программного продукта определить именно те участки кода, аппаратное ускорение которых даст существенное увеличение эффективности программно-аппаратного комплекса в целом. Из известных сфер применений именно цифровая обработка сигналов с ее потенциальной возможностью задействовать параллельные вычислительные структуры может получить существенный выигрыш при реализации в FPGA по сравнению с другими аппаратными платформами. Это не исключает применения HLS и в других задачах, но для алгоритмов ЦОС, по крайней мере, хорошо известны типичные решения на основе параллельных вычислительных структур.

Таким образом, для систем цифровой обработки сигналов в настоящее время существует три подхода:

- разработка «вручную» в RTL-стиле, возможно, с использованием стандартных IP-ядер;
- разработка с использованием System Generator for DSP;
- разработка в HLS.

Эти пункты не являются взаимоисключающими, поскольку САПР Vivado, предназначенная для сквозного проектирования, технически позволяет иметь в проекте несколько модулей, разработанных с использованием перечисленных подходов, комбинируя их сильные стороны.

Как следует из материала этой статьи, HLS позволяет описывать проект на языках высокого уровня. Разработчик освобождается от необходимости управлять последовательностью операций; автоматически поддерживаются многие типы данных (в том числе с плавающей точкой). Алгоритмы, разработанные на C/C++, в принципе могут быть перенесены в FPGA, при этом они получают существенное ускорение при наличии в исходной задаче операций, выполняющихся параллельно. Разработанный на HLS модуль можно легко подключить к проекту в System Generator или к процессорной системе в качестве периферийного модуля с интерфейсом AXI4.

В то же время результаты работы HLS имеют две особенности, которые могут затруднить применение разработанных модулей:

- сильная зависимость результатов от директив компилятора и стиля кодирования;
- интервальный характер латентности, плохо предсказуемое потактовое поведение синтезированной схемы.

В целом HLS не очень хорошо подходит для компактных проектов с малой латентностью, основанных на хорошо известных разработчику схемотехнических решениях. Например, простой КИХ-фильтр имеет несложное RTL-представление и может быть реализован на VHDL или Verilog, либо в виде IP-ядра, а HLS представляется несколько избыточным. Кроме того, при реализации КИХ-фильтра необходимо строго следить за постоянством временного интервала между обработкой отдельных отсчетов, и переменный характер латентности здесь недопустим. Напротив, если речь идет о проекте, ориентированном на длительное моделирование, проведение различных экспериментов, то высокая скорость разработки на HLS является значимым преимуществом.

Ряд публикаций по применению HLS ориентирован на использование этого инструмента для проектирования систем обработки изображений или программно-зависимого радио [2, 3]. Для таких проектов обычно характерна большая (и переменная) латентность, а также применение разработанного на HLS модуля в качестве периферийного устройства для процессорной системы. При этом процессор выполняет в основном функции управляющего устройства, обеспечивающего поддержку интерфейса, и получает результаты от аппаратного ускорителя с относительно низкой скоростью.

Заключение

Можно сделать вывод, что новая САПР, при всей ее неоднозначности, позволила иначе взглянуть на процесс проектирования цифровых устройств для FPGA. Для ускорения ее освоения компания Xilinx разработала учебные курсы, которые можно пройти в авторизованном учебном центре КТЦ «Инлайн Груп» (**www.plis.ru**). Для практикующих разработчиков новый инструмент может оказаться полезным при построении систем, предусматривающих сложные вычисления, особенно для устройств обработки изображений и видео, программно-зависимого радио и специализированных аппаратных ускорителей. ■

Литература

1. Vivado Design Suite Tutorial High-Level Synthesis — **http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug871-vivado-high-level-synthesis-tutorial.pdf**
2. Ozgul B., Langer J., Noguera J., Vissers K. Software-Programmable Digital Predistortion on the Zynq SoC — **http://www.xilinx.com/publications/archives/xcell/Xcell82.pdf**
3. Ahmed S. Z., Fuhrmann S., Granado B. Benchmark: Vivado's ESL Capabilities Speed IP Design on Zynq SoC — **http://www.xilinx.com/publications/archives/xcell/Xcell84.pdf**