Продолжение. Начало в КиТ № 6 `2007

Проектирование в условиях временных ограничений: отладка проектов

Ростислав ГРУШВИЦКИЙ
RIGrushvitsky@mail.eltech.ru
Максим МИХАЙЛОВ
yamaksya@yandex.ru

Данная статья продолжает рассмотрение вопросов отладки проектов на основе ПЛИС. Она содержит рекомендации и примеры практической работы со средствами фирм Altera и Synplicity.

Возможности, предоставляемые мегафункцией sld_virtual_jtag

Со времени появления первых ИС, поддерживающих граничное сканирование (стандарт JTAG IEEE Std 1149.1-1990), методы граничного сканирования убедительно доказали свою эффективность. Большинство современных схем содержит JTAG средства и ориентировано на граничное сканирование. Однако постоянное увеличение степени интеграции микросхем делает практически недоступным анализ внутренних процессов через внешние порты ИС, в связи с чем востребованными оказываются внутрисистемные средства отладки (in-system debugging tools). Рассмотренные в предыдущем выпуске такие инструменты фирмы Altera, как **SignalTap II** и **SignalProbe**, в настоящее время приобрели популярность среди разработчиков и широко применяются для отладки проектов. Значимость этих средств нельзя переоценить; вместе с тем возрастающая сложность современных систем предъявляет новые требования к отладке, что заставляет фирму Altera искать и придавать современным ИС все новые и новые функции. Весьма перспективным и заманчивым представляется использование отработанных методов граничного сканирования не только на границах кристалла, но и во внутренних блоках ИС. К дополнительным задачам, требующим решения, можно отнести, например:

- добавление виртуальных входных контактов, а также окружение выбранных фрагментов проекта ячейками сканирования;
- ускорение внутрикристальной отладки за счет загрузки внутренних регистров ИС тестовой информацией — с целью формирования условий, которые в процессе отладки происходят крайне редко;
- эмуляцию состояния системы, которое принципиально недостижимо при нормальной работе;
- настройку одних частей системы на работу в тестовом режиме, в то время как остальные части функционируют в штатном режиме.

Для реализации указанных свойств можно пойти на разработку собственной отладочной инфраструктуры на уровне системы. При этом необходимо учитывать, что решение этой задачи потребует создания как аппаратной платформы, так и программных средств, поддерживающих отладку. Временные затраты при создании в первый раз таких средств могут оказаться чрезмерно большими, поэтому заманчивым было бы использовать готовые фирменные средства.

Внутреннее сканирование (или, в соответствии с терминологией фирмы Mentor Graphics, просто «сканирование») оказывается менее универсальным средством с точки зрения его реализации — такие структуры сложнее унифицировать (например, заранее просто невозможно предугадать места желательного размещения ячеек сканирования). Однако этот недостаток в свою очередь порождает и основное достоинство — обеспечение гибкости. Он создает компромисс между уникальностью и специфичностью, предоставляя определенный набор стандартных средств и, в то же время, предоставляя разработчику большое поле для самостоятельной деятельности.

С помощью имеющейся в составе ПО Quartus II мегафункции **sld_virtual_jtag** можно организовать один или более прозрачных (незаметных для пользователя) каналов связи для доступа к различным частям проекта. Виртуальный JTAG интерфейс (Virtual JTAG Interface, VJI) позволяет, опираясь на САПР фирмы Quartus, перенести стандартные решения граничного сканирования внутрь кристалла, в том числе, например, расширить возможности анализатора Signal Tap II.

Подробное описание мегафункции **sld_virtual_jtag** можно свободно скачать с официального сайта фирмы Altera. В рамках же данной статьи авторы старались отразить только некоторые ключевые особенности VJI-модуля.

Структура системы отладки

На рис. 22 представлена возможная структура отладочной системы, использующей мегафункцию **sld_virtual_jtag**. Ориентация мегафункции на стандарт JTAG определяет наличие в общей структуре типичных блоков стандартной схемы. Как видно из рисунка, в отладочной системе ИС можно выделить три типа фрагментов (на рисунке выделены цветом): фрагменты, штатно встроенные в ПЛИС, фрагменты, создаваемые компилятором Quartus при имплементации мегафункции, и фрагменты, которые должен создать разработчик. Наличие фрагментов первой группы порождает ограничения в типах ПЛИС, допускающих подключение мегафункции. Ко второй группе фрагментов относятся обязательные элементы JTAG контроллеров, не допускающие вариаций. И, наконец, наличие фрагментов третьей группы создает гибкость и уникальность создаваемой отладочной системы, поскольку допускает создание схем, зависящих от конкретных требований к функциям отладочного оборудования. Следует надеяться, что со временем многие разрабатываемые сейчас пользователем компоненты отладочной системы будут переводиться в настраиваемые фирменные средства. Однако различные трактовки элементов этой группы (или требования их настройки) усложняют использование мегафункции и заставляют разработчиков детально разбираться с принципами создания JTAG контроллеров и программного обеспечения для них.

Сегодня фиксированная часть каждого блока VJI содержит только регистр команд (длина может настраиваться разработчиком) и комбинационную схему, вырабатывающую сигналы, необходимые для реализации логики декодирования команд. Компонент JVI имеет три выходных (**tdi**, **tck**, **tms**) и один входной (**tdo**) порт JTAG. Кроме того, есть несколько выходных сигналов (обозначены на рисунке как **out**[]), которые отражают текущее состояние TAP-контроллера JTAG (имена сигналов содержат префикс **jtag_state**) и внутреннее состояние VJI-модуля (имена сигналов содержат префикс **virtual_state**). Разрядность и значение на параллельной шине

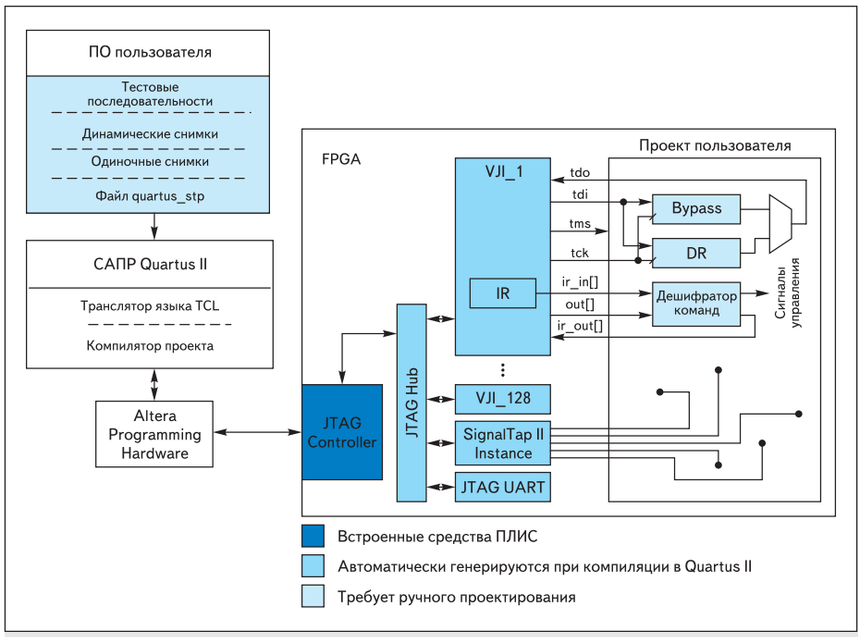

Рис. 22. Отладочная система

ir_in[] всецело определяются требуемым содержимым регистра команд IR. Группа входных сигналов **ir_out**[] предназначена для пересылки определенных пользователем данных из проекта в приложение при выполнении цикла сдвига регистра IR.

Для работы в соответствии с требованиями стандарта IEEE Std 1149.1 разработчиком должны быть созданы схема управления блоками JTAG-интерфейса и один или нескольких регистров данных (DR). В VJI-модуле нужно построить собственное устройство управления, используя для этих целей состояния конечного синхронного автомата TAP-контроллера ПЛИС. Другой задачей разработчика, которую необходимо решить в рамках аппаратной части проекта, является организация сдвигающих регистров данных. При помощи описанных выше сигналов и дополнительно построенных схем реализуется подключение блока Virtual JTAG к интересующим разработчика частям проекта.

Для подключения сгенерированного VJI-модуля к ведущему оборудованию используют его соединение со схемой JTAG концентратора ИС (JTAG hub). Связь мегафункции с концентратором скрыта от разработчика и автоматически устанавливается при компиляции в Quartus II. Концентратор необходим для организации совместной работы нескольких абонентов, требующих подключения к порту JTAG, например, логического анализатора SignalTap II, приемопередатчика JTAG UART, блока отладчика процессорного ядра Nios II и др.

Из приведенного рисунка видно, что в составе САПР Quartus II не предусмотрено наличие соответствующего ведущего программного обеспечения. Все связи Quartus II с имплементированной мегафункцией **sld_virtual_jtag** предполагают использование команд скриптового языкаTcl.

Требуемое программное обеспечение пользователя определяется сценариями отладки проекта. Условно сценарии и соответствующие ПО, создаваемые разработчиком, можно разбить на четыре категории:

- низкоуровневое управление схемой JTAG;
- отладка, опирающаяся на мгновенные (одиночные) снимки в регистрах данных JTAG-цепочек;
- отладка во временной области — анализ или просмотр динамически получаемых снимков;
- автоматизированное создание тестовых последовательностей.

Предусмотренное фирмой низкоуровневое управление, как указано выше, базируется на унифицированном инструментальном языке команд (Tool Command Language, Tcl). Консольные команды из САПР Quartus или созданные разработчиком программы на языке Tcl воспринимаются САПР и далее выполняются в ведомом оборудовании (исполняемый файл **quartus_stp**).

Для определенности значений данных проекта их фиксация в регистрах данных схемы JTAG должна быть синхронизирована с работой внутреннего оборудования проекта. Поэтому для создания «снимков состояния системы» в те или иные моменты времени разработчик должен предусмотреть наличие набора соответствующих средств (например, соответствующих команд регистра команд и т. д.). В результате только совокупность аппаратных и программных средств позволяет организовать фиксацию состояния системы в рабочих и/или тестовых режимах.

Более сложные сценарии отладки требуют больших затрат разработчиков на создание соответствующих программ. Естественно желание использовать готовые фирменные программные решения, например, результаты работы систем автоматизированного создания тестовых последовательностей. Однако приходится учитывать, что большинство таких систем работает на основе анализа информации о тестируемой системе в форме ранее созданных файлов, таких как BSD-файлы. Поэтому разработчик должен будет для создаваемой им конфигурации внутрикристальных средств отладки написать в стандарте 11491.a один или даже несколько файлов, описывающих эту конфигурацию.

Перечисленные сложности создания высокоуровневого пользовательского программного обеспечения, очевидно, заставят многих пользователей отдать предпочтение более простой в реализации комбинации фирменных средств фирмы Altera: мегафункции **sld_virtual_jtag** и логического анализатора **SignlTAP II** (о нем рассказано в предыдущем выпуске журнала).

Альтернативным вариантом реализации отладочного комплекса может служить конструкция, когда вместо САПР, функционирующей на ПК, разработчик задействует собственную отладочную HOST-систему (где JTAG-контроллер, например, строится на основе собственного микроконтроллера, решающего задачи низкоуровневого управления каналом связи).

Функционирование системы отладки

В соответствии с работой устройств управления, определенных в протоколе JTAG, функционирование VJI-модуля предусмотрено в двух базовых режимах: режиме ввода команд и чтения состояния (Instruction Register Scan Shift) и режиме ввода/вывода данных (Data Register Scan Shift). В первом случае данные из TAP-порта устройства поступают в регистр команд IR соответствующего блока VJI. Загруженная команда выдается из модуля через порт **ir_in**. При выполнении сдвига IR данные из проекта пользователя могут быть переданы через порт **ir_out** в схему JTAG и далее в приложение.

В случае ввода/вывода данных задействованы сигналы **tdi**, **tdo** и **tck**. Как уже было отмечено ранее, организация регистров данных DR не производится автоматически, а является задачей проектировщика. Так же, как и в стандартной архитектуре JTAG, в проекте может быть создано несколько регистров данных. Помещение данных в регистры DR производится через порт **tdi** модуля VJI. Одновременно с этим происходит выдвигание данных из регистров DR через порт **tdo**. Все сдвиги выполняются синхронно с сигналом **tck**.

При работе в обоих режимах информация о текущем состоянии TAP контроллера может быть получена с помощью группы сигналов **out**.

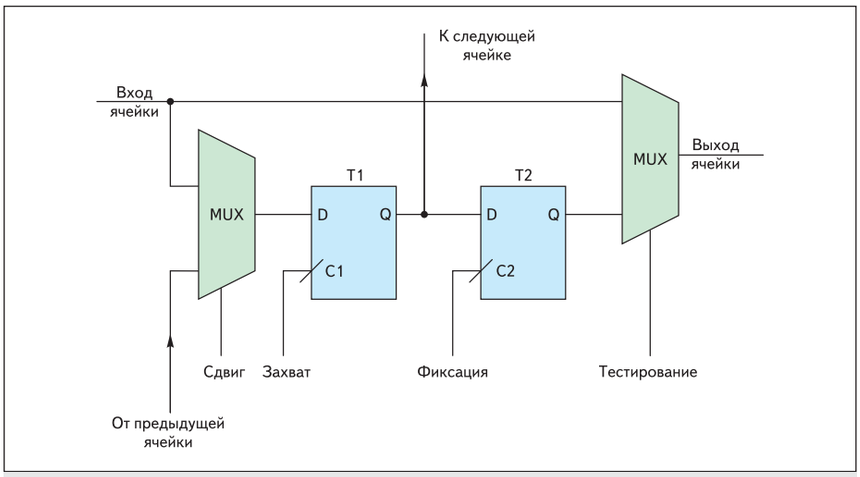

Рис. 23. Разряд регистра данных сканирующей цепочки

Пример встраивания модуля VJI в проект

Модуль виртуального JTAG напрямую встраивается в HDL-код проекта. При компиляции каждому объявленному экземпляру VJI для идентификации присваивается (вручную или автоматически) уникальный номер. Максимальное число VJI-модулей, которые можно создать в рамках одного проекта, — 128.

Мегафункция **sld_virtual_jtag** может быть применена во множестве приложений, среди которых следует выделить возможность считывания или изменения состояния внутренних узлов проекта (например, счетчиков, конечных автоматов и т. п.), а также создание виртуальных контактов. Используя же готовый набор Tcl-команд, можно разработать собственное программное обеспечение для отладки проекта, которое будет взаимодействовать со встроенными блоками VJI.

В качестве примера применения средств виртуального JTAG возьмем уже апробированный нами 4-разрядный регистр. К существующему проекту добавим мегафункцию для считывания значения разрядов регистра в сдвигающий регистр данных JTAG-интерфейса.

Для воплощения замысла исходный проект необходимо расширить за счет следующих компонентов: регистра данных, дешифратора команд и, возможно (согласно требованиям стандарта), регистра пропуска (Bypass). На рис. 23 представлена одна из самых распространенных структур сканирующей ячейки JTAG. Для целей определения значения внутреннего регистра проекта (DFF_register) достаточно будет реализовать только первую ступень этой структуры.

Управление процессом защелкивания и вывода данных производится при помощи предопределенной команды Scan, загружаемой в регистр команд IR, и сигналов состояния TAP-контроллера — cdr (capture data register) и sdr (shift data register). Далее приведен VHDL-код разработанного устройства:

```
LIBRARY IEEE;
  USE IEEE.std_logic_1164.ALL;
  USE IEEE.std_logic_unsigned.ALL;

ENTITY vji_unit_2 IS
  PORT (
            Clock : IN std_logic;
            Reset : IN std_logic;
            Shift : IN std_logic;
            Data_in : IN std_logic;
            Load : IN std_logic;
            En_Count : IN std_logic
            );
END vji_unit_2;

ARCHITECTURE model OF vji_unit_2 IS

  SIGNAL Shift_register : std_logic_vector(3 DOWNTO 0);
  SIGNAL Counter : std_logic_vector(3 DOWNTO 0);
  SIGNAL Over_flow : std_logic;
  SIGNAL DFF_register : std_logic_vector(4 DOWNTO 0);

  SIGNAL bypass_reg : std_logic;

  SIGNAL Scan : std_logic;
  SIGNAL DR : std_logic_vector(4 DOWNTO 0);

  SIGNAL tdo, tck, tdi, cdr, sdr : std_logic;
  SIGNAL ir_in : std_logic_vector(1 DOWNTO 0);

  COMPONENT sld_virtual_jtag
  GENERIC (
            sld_auto_instance_index : STRING;
            sld_instance_index : NATURAL;
            sld_ir_width : NATURAL;
            sld_sim_action : STRING;
            sld_sim_n_scan : NATURAL;
            sld_sim_total_length : NATURAL;
            lpm_type : STRING
            );
  PORT (
            tdi : OUT STD_LOGIC ;
            jtag_state_rti_type : OUT STD_LOGIC ;
            jtag_state_e1dr : OUT STD_LOGIC ;
            jtag_state_e2dr : OUT STD_LOGIC ;
            tms : OUT STD_LOGIC ;
            jtag_state_pir : OUT STD_LOGIC ;
            jtag_state_tlr : OUT STD_LOGIC ;
            tck : OUT STD_LOGIC ;
            jtag_state_sir : OUT STD_LOGIC ;
            ir_in : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
            virtual_state_cir : OUT STD_LOGIC ;
            virtual_state_pdr : OUT STD_LOGIC ;
            ir_out : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
            virtual_state_uir : OUT STD_LOGIC ;
            jtag_state_cir : OUT STD_LOGIC ;
            jtag_state_uir : OUT STD_LOGIC ;
            jtag_state_pdr : OUT STD_LOGIC ;
            tdo : IN STD_LOGIC ;
            jtag_state_sdrs : OUT STD_LOGIC ;
            virtual_state_sdr : OUT STD_LOGIC ;
            virtual_state_cdr : OUT STD_LOGIC ;
            jtag_state_sdr : OUT STD_LOGIC ;
            jtag_state_cdr : OUT STD_LOGIC ;
            virtual_state_udr : OUT STD_LOGIC ;
            jtag_state_udr : OUT STD_LOGIC ;
            jtag_state_sirs : OUT STD_LOGIC ;
            jtag_state_e1ir : OUT STD_LOGIC ;
            jtag_state_e2ir : OUT STD_LOGIC ;
            virtual_state_e1dr : OUT STD_LOGIC ;
            virtual_state_e2dr : OUT STD_LOGIC
            );
  END COMPONENT;

BEGIN

sld_virtual_jtag_component : sld_virtual_jtag
GENERIC MAP (
            sld_auto_instance_index => «YES»,
            sld_instance_index => 0,
            sld_ir_width => 2,
            sld_sim_action => «»,
            sld_sim_n_scan => 0,
            sld_sim_total_length => 0,
            lpm_type => «sld_virtual_jtag»
            )
PORT MAP (
            tdo => tdo,
            tck => tck,
            tdi => tdi,
            ir_in => ir_in,
            virtual_state_sdr => sdr,
            virtual_state_cdr => cdr
            );

-- последовательно-параллельный сдвигающий регистр
PROCESS(Reset, Clock, Shift, Data_in, Shift_register)
BEGIN
  IF Reset = '1' THEN
     Shift_register <= «0000»;
  ELSIF Rising_edge(Clock) THEN
     IF (Shift = '1') THEN
        Shift_register <= Data_in & Shift_register (3 DOWNTO 1);
     END IF;
  END IF;
END PROCESS;

-- счетчик с параллельной загрузкой
PROCESS (Clock, Load, En_Count, Shift_register, Counter)
BEGIN
  IF Rising_edge (Clock) THEN
     IF (Load = '1') THEN
        Counter <= Shift_register;
     ELSIF En_Count = '1' THEN
        Counter <= Counter + '1';
     END IF;
  END IF;
END PROCESS;

-- логика переполнения счетчика
Over_flow <= '1' WHEN Counter = «1111» ELSE '0';

-- выходной буфер
PROCESS (Clock, Counter, Over_flow)
BEGIN
  IF Rising_edge (Clock) THEN
     DFF_register(3 DOWNTO 0) <= Counter;
     DFF_register(4) <= Over_flow;
  END IF;
END PROCESS;

-- логика декодирования содержимого регистра команд (IR) VJI-модуля
Scan <= not ir_in(1) and ir_in(0);

-- Регистр пропуска (Bypass)
PROCESS (tck)
BEGIN
  IF Rising_edge (tck) THEN
     bypass_reg <= tdi;
  END IF;
END PROCESS;

-- Регистр данных JTAG
PROCESS (tck, Scan, sdr, DR, DFF_Register)
BEGIN
  IF Rising_edge (tck) THEN
     IF (Scan = '1' and cdr = '1') THEN
        DR <= DFF_register;
     ELSIF (Scan = '1' and sdr = '1') THEN
        DR <= DR(3 DOWNTO 0) & tdi;
     END IF;
  END IF;
END PROCESS;

-- логический блок tdo
PROCESS (tck)
BEGIN
  IF Rising_edge (tck) THEN
     IF (Load = '1') THEN
        tdo <= DR(4);
     ELSE
        tdo <= bypass_reg;
     END IF;
  END IF;
END PROCESS;

END model;
```

Для полноты представления опишем, какие изменения произошли в исходном коде. Порт DFF_register был заменен аналогичным внутренним сигналом, поскольку его значение теперь можно выводить через JTAG-порт, и, следовательно, исчезает необходимость в использовании дополнительных контактов ввода/вывода. В разделе деклараций архитектурного тела проекта объявлена мегафункция **sld_virtual_jtag**, а также добавлен ряд сигналов: bypass_reg (регистр пропуска), Scan (сигнал, активизирующийся при записи в регистр IR значения «01»), DR (регистр данных VJI), tdo, tck, tdi, cdr, sdr, ir_in (сигналы подключения компонента virtual JTAG). VHDL-код исходного блока **unit** изменениям не подвергнут. Назначение оставшейся части кода раскрывается в комментариях листинга.

Работа с пакетом Identify компании Synplicity

Synplicity — одна из ведущих мировых фирм, занятых разработкой программного обеспечения для проектирования ПЛИС. Достоинством ПО этой компании является не только поддержка продукции практически любых производителей ПЛИС, но и высокая эффективность получаемых проектных решений (минимальность затрачиваемых на проект ресурсов ПЛИС при высоком достигаемом быстродействии). Особое место занимает и продукция этой фирмы с точки зрения внутрикристальных отладочных средств. Основная идея Synplicity — это автоматизированное подключение к существующему проекту пользователя текста образуемого САПР, представляющего собой описание на языке аппаратуры схемы логического анализатора. Его реализация выполняется по указаниям разработчика с учетом ограничений, диктуемых ресурсами тестируемой ПЛИС. Поскольку добавление измерительного окружения есть по сути лишь модификация HDL-описания, то отладка средствами Synplicity возможна даже в тех случаях, когда традиционные средства оказываются бессильны. Примером данной ситуации можно считать микросхему ACEX компании Altera. Семейство ACEX (ПЛИС Altera) архитектурно не поддерживается средствами внутрикристальной отладки SignalTap II. Для продукции Synplicity нет таких ограничений, хотя функциональные возможности ПО этой компании по части отладки во многом совпадают с возможностями уже рассмотренного пакета SignalTap II. Принципиальным отличием САПР компании Synplicity является возможность имплементации отладочных средств практически в любую ПЛИС. В рассматриваемом далее примере работы с ПО Synplicity (пакетом Identify) отлаживался уже знакомый читателям проект, помещаемый в ИС ACEX. ПЛИС установлена на макетной плате, изображенной на рис. 24.

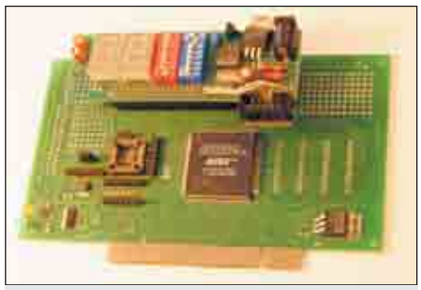

Рис. 24. Отладочная плата с ПЛИС ACEX

Synplicity поддерживает внутрикристальную отладку HDL-проектов при помощи системы Identify RTL Debugger. Пакет состоит из двух частей: **Identify Instrumentor** и **Identify Debugger**. На основе исходного HDL-описания устройства Identify Instrumentor производит настройку измерительной аппаратуры. Затем с помощью Identify Debugger выполняется внутрикристальная отладка системы, которая обеспечивается за счет встраивания в ИС блока **IICE** (Intelligent In-Circuit-Emulator). Эмулятор **IICE** подсоединяется к интересующим сигналам проекта и взаимодействует с отладочной (host) системой через JTAG-интерфейс. Комбинация этих средств способствует повышению эффективности процесса отладки HDL-проектов непосредственно в целевой системе, а также позволяет значительно упростить и ускорить его.

Процесс отладки может отличаться в деталях для разных типов ПЛИС, но в общем виде проектный поток будет выглядеть так, как показано на рис. 25.

Традиционный подход к созданию проекта предполагает, прежде всего, разработку HDL описания устройства. После этого следуют этапы логического синтеза и физического размещения и трассировки проекта в кристалле. В заключение выполняется имплементация проекта путем загрузки полученной конфигурации в ПЛИС. Применение инструмента **Identify RTL Debugger** вносит некоторые дополнения в этот проектный поток. После успешного создания HDL-описания **Identify Instrumentor** позволяет произвести сборку измерительного окружения и его добавление к исходному коду для создания пригодной для отладки версии проекта. Второе дополнение возникает после программирования целевого устройства отладочной версией проекта — это непосредственно отладка с помощью **Identify Debugger**.

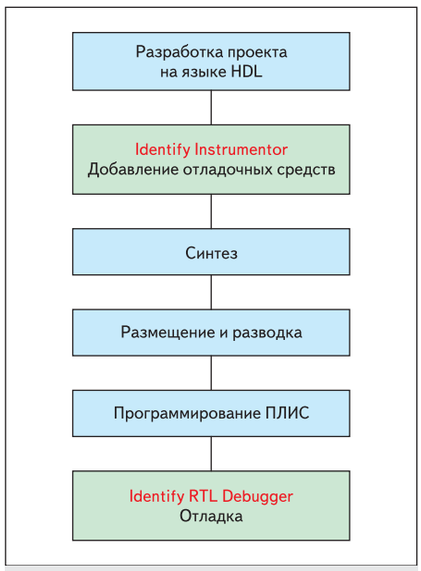

Рис. 25. Типичный проектный поток для САПР Identify RTL Debugger

Для понимания правил функционирования отладочной системы Identify следует более подробно рассмотреть отдельные, составляющие ее компоненты.

Эмулятор IICE

Эмулятор IICE — это специализированный блок, который встраивается в основной проект и подключается к внутренним сигналам в соответствии с заданной в Identify Instrumentor спецификации. Во время отладки IICE перепрявляет информацию в Identify Debugger для дальнейшей интерпретации. Логически IICE можно разделить на две составляющие — измерительный блок (probe block) и блок управления (controller block). Первый из них отвечает за фиксацию значений внутренних сигналов и взаимодействие с блоком управления, второй — за взаимодействие с портом JTAG. Измерительный блок включает в себя буфер для временного хранения отсчетов, а также логическую схему фиксации, которая определяет моменты защелкивания данных в буфер. Блок управления перенаправляет полученные от измерительного блока данные через JTAG-порт в Identify Debugger.

Для реализации транспортного механизма может использоваться как встроенный TAP-контроллер ПЛИС, так и IP-ядро контроллера JTAG (так называемая soft-опция). Первый вариант является более предпочтительным, поскольку для отладки используются выделенные для этих целей ресурсы (контакты ввода/вывода и схемы). Однако может возникнуть ситуация, когда эти ресурсы отсутствуют в микросхеме либо они недоступны. В этом случае применяется готовая HDL-модель TAP-контроллера, а для передачи отладочной информации необходимо выделить 4 пользовательских контакта ввода/вывода.

Для настройки блока формирования условий защелкивания, входящего в состав измерительной части IICE, существует два ключевых момента — точки останова (breakpoints) и точки наблюдения (watchpoints). Точки останова предоставляют возможность создания условий защелкивания, которые определяются потоком управления проекта, задаваемым операторами **IF**, **ELSE** и **CASE**. Останов в этом случае осуществляется при достижении условными операторами некоторого

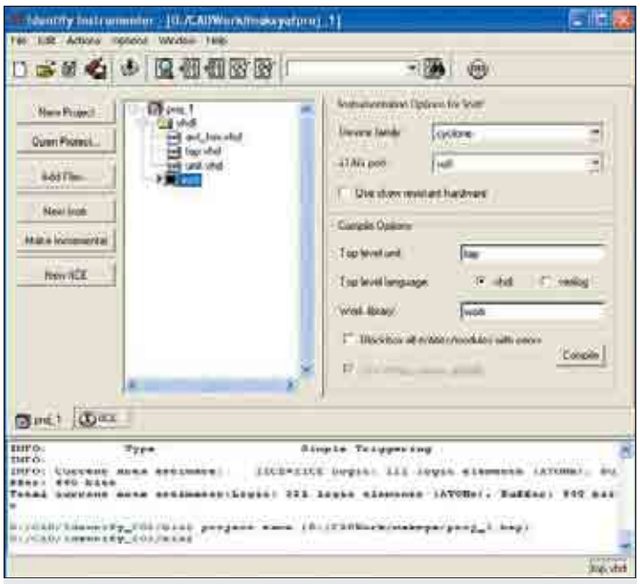

Рис. 26. Окно проекта программы Identify Instrumentor

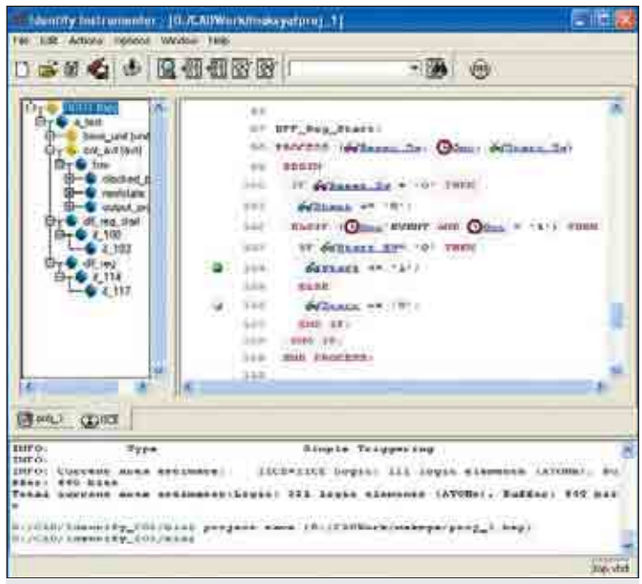

Рис. 27. Окно измерительного окружения

значения (иными словами — при прохождении определенной ветки алгоритма работы HDL-проекта). Этим логический анализатор Identify Instrumentor принципиально отличается от средств типа SignalTap II фирмы Altera.

Точки наблюдения связаны с событиями, которые определяются состояниями сигналов проекта. Защелкивание может производиться как при достижении сигналом заданного значения (фиксация по уровню), так и при переключении значения сигнала (фиксация по перепаду). Одновременное использование нескольких точек останова и/или точек наблюдения имеет свою специфику. Так, события, задаваемые активными точками останова, объединяются по логическому «или», то есть любое из подобных событий приведет к срабатыванию логики фиксирования данных в буфере. В то же время точки наблюдения объединяются по логическому «и». Например, если одна точка останова реализует условие (bit_1 == '1'), а вторая — (bit_2 == '0'), то их комбинация выражается в образовании новой точки останова с условием (bit_1 == '1') && (bit_2 == '0'). Объединение точек останова и точек наблюдения выполняется по правилу (breakpoint_1 or breakpoint_2 … or breakpoint_N) && (watchpoint_1) && … (watchpoint_M). Результатом данной конъюнкции является так называемый главный сигнал защелкивания (Master Trigger Signal, MST), активация которого приводит к фиксации данных в буфере.

Помимо использования базовых условий защелкивания в Identify существует также возможность формирования расширенных условий с помощью специального счетчика (Complex Counter) и конечного автомата (State Machine Triggering). Подробный анализ данных средств выходит за рамки данной статьи и вряд ли существен для понимания принципов работы рассматриваемых средств отладки.

Другим немаловажным компонентом измерительного блока эмулятора IICE является блок выборки (sampling block), который представляет собой в основном память большого объема для временного хранения полученных отсчетов сигналов. В процессе отладки данные непрерывно фиксируются в блоке выборки до момента срабатывания логики сигнала MST. Данное событие приводит к прекращению записи данных в буфер и последующей передаче полученной информации в ПК. Предпочтительным для реализации блока выборки является использование встроенных блоков памяти, которые имеются в большинстве современных ПЛИС. В противном случае будут задействованы отдельные логические ячейки, что приведет к большим затратам логических ресурсов кристалла.

Identify Instrumentor

Программа Identify Instrumentor, анализируя HDL-описание устройства, предоставляет подробную информацию о наблюдаемости внутренних сигналов и позволяет настроить тестовое окружение проекта (иными словами — сконфигурировать блок IICE). Identify Instrumentor выполняет три основные функции:

- определяет возможное тестовое окружение HDL-проекта;
- модифицирует исходный код для отладки;
- создает соответствующий анализатор IICE.

Основу графического интерфейса пользователя Identify Instrumentor составляют 3 окна: окно проекта, окно измерительного окружения и консольное окно.

Окно проекта (рис. 26) появляется при запуске программы Instrumentor. Оно содержит ряд функциональных кнопок по управлению проектом. Каждое измерительное окружение (instrumentation) может содержать несколько независимых блоков IICE (добавление выполняется кнопкой **New IICE**). В свою очередь можно создать несколько различных измерительных окружений (**New Instr.**). С помощью данного окна выполняется ряд общих настроек логического анализатора, а также отображается структура проекта.

Помимо общих настроек необходимо сконфигурировать специфические параметры IICE. Настройка выполняется после нажатия кнопки **Edit IICE Settings** панели управления программы **Instrumentor**. В двух вкладках (**IICE Sampler** и **IICE Controller**) задаются параметры анализатора, описанные в предыдущем разделе: тип буфера для хранения отсчетов, его глубина, частота выборки, тип формирования условий защелкивания и т. д.

Вкладка (рис. 27) окна измерительного окружения (instrumentation window) появляется после открытия существующего проекта или успешной компиляции нового. В левой части этого окна отображается иерархическое представление проекта. Корнем иерархической схемы является (в VHDL-проекте) entity высшего уровня, ниже располагается архитектурное тело этого entity, а еще ниже — entity следующих уровней и т. д. Нижний уровень иерархии состоит из различных операторов. В правой части окна измерительного окружения отображается исходный код со специальными отметками сигналов, относительно которых можно производить наблюдение, а также точек останова, которые могут быть добавлены в состав логического анализатора. «Наблюдаемые» сигналы подчеркиваются, выделяются синим цветом и помечаются изображением очков. В процессе построения измерительного окружения такие сигналы могут включаться в состав отладочной аппаратуры как частота выборки (Sample clock), сигнал только для наблюдения (Sample only), сигнал только для создания

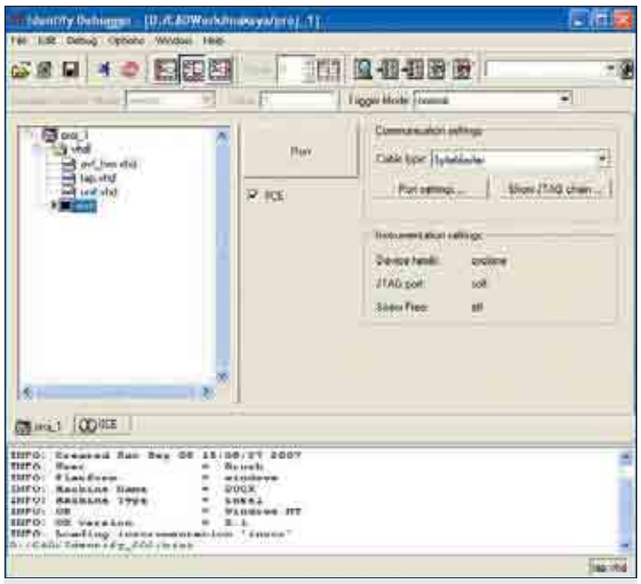

Рис. 28. Окно проекта Identify Debugger

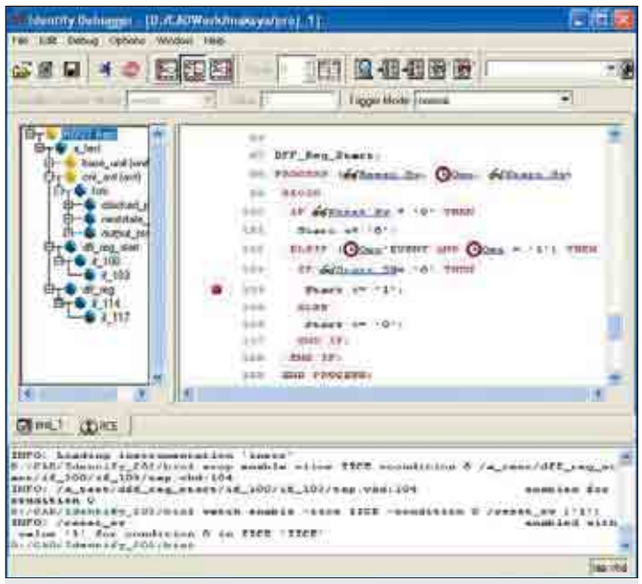

Рис. 29. Окно измерительного окружения Identify Debugger

условия защелкивания (Trigger only), сигнал для наблюдения и условия фиксации (Sample and trigger), или пропускаться (Not instrumented).

В консольном окне отображается история всех запущенных команд. Также допускается ввод отладочных команд с консоли и просмотр результатов их исполнения.

Identify Debugger

Программа Identify Debugger обеспечивает взаимодействие с проектом во время отладки. В Identify Debugger можно активировать точки останова, задавать точки наблюдения и просматривать временные диаграммы поведения внутренних сигналов. С помощью Identify Debugger также формируется условие фиксации данных в буфере.

Графический интерфейс пользователя Identify Debugger имеет схожую с Identify Instrumentor структуру: окно проекта, окно измерительного окружения и консольное окно.

После открытия проекта *.bsp, полученного в результате работы Identify Instrumentor, на экране будет отображено окно измерительного окружения (instrumentation window). Для возврата к окну проекта необходимо щелкнуть по соответствующей вкладке. В окне проекта (рис. 28) слева отображается структура проекта, а справа — список доступных для данного измерительного окружения модулей IICE, встроенных в исходный код, настройки соединения с ПК и кнопка запуска процесса отладки.

Окно измерительного окружения Identify Debugger (рис. 29), опять же по аналогии с Identify Instrumentor, содержит иерархическое представление проекта (слева) и исходный код (справа). Однако здесь в HDL-коде помечены только те точки останова и наблюдения, которые были заданы в процессе создания измерительного окружения в Identify Instrumentor. В процессе отладки участвуют активированные точки останова и наблюдения. Для активации точки останова необходимо щелкнуть по ее изображению, в результате чего она изменит свой цвет на красный. Активация точки наблюдения выполняется через диалоговое окно Watchpoint Setup (появляется при выборе соответствующего сигнала), в котором можно задать условие защелкивания.

Роль консольного окна Identify Debugger аналогична описанному в разделе Identify Instrumentor.

Пример отладки проекта с помощью пакета Identify

Инструменты Identify могут быть запущены прямо из основных средств логического синтеза фирмы Synplicity — Synplify, Synplify Pro, Synplify Premier. В этом случае информация исходного Sinplify-проекта (расширение *.prj) автоматически загружается в Identify Instrumentor для дальнейшей настройки анализатора. Если же работа изначально выполняется в Identify в автономном режиме, то после запуска программы будет создан пустой Identify-проект, имеющий расширение *.bsp. Тогда потребуется вручную добавить необходимые файлы, содержащие HDL-описание отлаживаемого устройства, выполнить необходимые настройки и произвести компиляцию.

Последовательность действий при работе с системой отладки Identify будет выглядеть следующим образом:

1. Подготовка HDL-описания устройства.
2. Создание проекта в Identify Instrumentor или перенос проекта из Synplify.
3. Компиляция в Identify Instrumentor (только для вновь созданных проектов).
4. Настройка логического анализатора в Identify Instrumentor.
5. Генерация тестового окружения и сохранение проекта с расширением *.bsp.
6. Получение файла конфигурации модифицированного HDL-проекта и загрузка его в соответствующую ПЛИС.
7. Запуск Identify Debugger.
8. Активация необходимых точек останова и наблюдения.
9. Запуск процесса отладки.
10. Анализ полученных результатов (создание временных диаграмм поведения наблюдаемых сигналов).

Если информации достаточно для принятия решения о корректировке проекта или необходимо изменить список фиксируемых сигналов, процедура отладки повторяется с нужного пункта.

Описанные выше сведения не претендуют на всесторонность рассмотрения (что, на взгляд авторов, и не представляется возможным в рамках статьи). Однако мы надеемся, что изложенный материал послужит хорошим начальным руководством к действию. Более подробные сведения можно найти в источниках [5] и [6], которые располагаются в директории /doc после установки пакета Identify.

Литература

1. Quartus II Version 7.0 Handbook Volume 3: Verification (Section V. In-System Design Debugging). w ww.altera.c om
2. sld_virtual_jtag Megafunction User Guide. w ww.altera.c om
3. Using the Identify Hardware Debugger to Catch Timing Problems. w ww.synplicity.c om/literature/pdf/identify_appnote05.pdf
4. Леклиндер Т. Погружаясь в ПЛИС // Компоненты и технологии. 2006. № 12.
5. Identify RTL Debugger. User Guide. Synplicity.
6. Identify RTL Debugger. Quick Tutorial. Synplicity.