

Продолжение. Начало в № 3 '2009

Александр ШАЛАГИНОВ  
shalag@vt.cs.nstu.ru

# Изучаем Active-HDL 7.1.

## Урок 14. Стили HDL-кодирования цифрового автомата

**Ц**ифровой автомат — это концептуальное представление последовательностного устройства. Фактически любая цифровая схема с памятью может рассматриваться как конечный автомат или их объединение.

Зависимость реакции выхода от текущего состояния автомата отличает его от комбинационных устройств. Каноническое представление цифрового автомата (ЦА) показано на рис. 1.

Базовым элементом автомата является регистр состояния **SR** (от **State Register**). Он удерживает текущее состояние (**Current State**) до прихода очередного тактирующего импульса **CLK** (активного фронта или среза).

В паузе между тактирующими импульсами на входе **SR** формируется вектор следующего состояния (**Next State**). Выполняется данная работа комбинационным блоком **F**-логикой переходов. Выход **F** зависит от текущего состояния регистра **SR** и входных воздействий **X**.

Выходная логика **G** — тоже комбинационная схема. Она формирует выходные сигналы **Y** в зависимости от текущего состояния автомата — содержимого регистра **SR** (автомат Мура). Для автомата Мили выходы зависят еще и от текущих значений входных воздействий **X**, а потому допускают асинхронное поведение.

Этот небольшой экскурс в теорию автоматов нам понадобился для того, чтобы сделать более понятным разговор о возможных стилях **HDL**-кодирования ЦА. Графический редактор **State Diagram Editor** поддерживает разные стили кодирования, и пользователь может комбинировать их по своему желанию (рис. 2).



Рис. 2. Выбор стиля HDL-кодирования ЦА

Впрочем, вариантов не так уж и много. **HDL**-модель ЦА удобно представлять в виде одного (**One Process**), двух (**Two Processes**) или трех (**Three Processes**) процессов. Внутри каждого процесса по умолчанию предлагается использовать оператор выбора **Case**, но вы

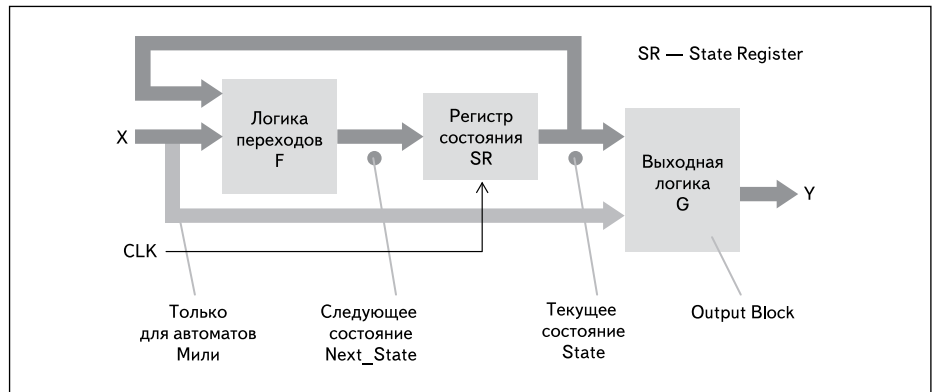


Рис. 1. Графическое представление синхронного цифрового автомата с комбинационным выходом

можете отдать предпочтение условному оператору **IF**.

### Модель ЦА в виде трех процессов

Наиболее очевидной представляется **VHDL**-модель цифрового автомата в виде трех процессов — по числу входящих в него блоков (рис. 1). Посмотрим, каким получается автоматически сгенерированный **VHDL**-код (рис. 4а) для двоичного суммирующего счетчика с входом разрешения **E** (рис. 3).

Обратите внимание, в самом начале кода декларируются два внутренних сигнала для хранения текущего **Sreg0** и следующего **NextState\_Sreg0** значений регистра состояния. По умолчанию он имеет имя **Sreg0**.

Поведение автомата имитируется процессом с меткой **Sreg0\_CurrentState**, который запускается любым из сигналов — **C** или **R**. На рис. 1 этому процессу соответствует регистр состояния **SR**, а его выход — сигналу **Sreg0**.

Логика переходов **F** (рис. 1) имитируется процессом с меткой **Sreg0\_NextState**. Напомним, что блок **F** представляет собой комбинационную схему, которая вычисляет следующее состояние автомата. Оно зависит от текущего состояния **Sreg0** и вектора входных воздействий (в нашем примере — от единственного входа разрешения **E**). По этой причине в списке инициализаторов данного процесса мы видим только эти имена (рис. 4).

Блок выходной логики **G** (тоже комбинационная схема) реализуется процессом **Sreg0\_OutputBlock**. В нашем примере он не содержит входных воздействий **X** (автомат Мура), поэтому реакция на выходах определяется только текущим содержимым регистра состояний **Sreg0**. Это имя мы и наблюдаем в списке чувствительности данного процесса.

Рассмотренный стиль кодирования автомата **Three Processes** задается на диалоговой

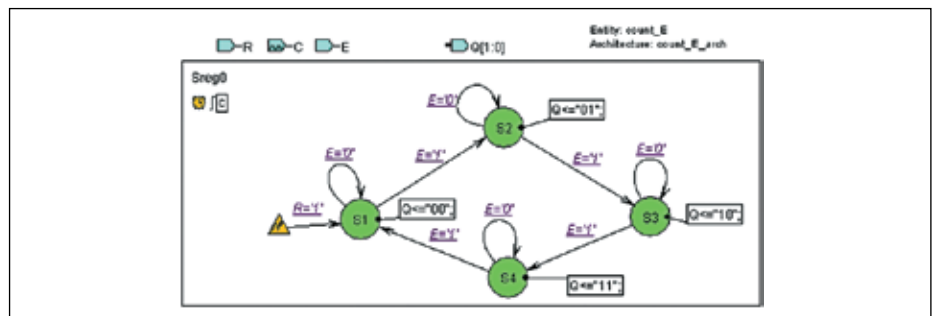


Рис. 3. Диаграмма состояний ЦА с функцией счетчика с входом разрешения E

```

34 architecture count_e_arch of count_E is
35 type Sreg0_type is (S1, S2, S3, S4);
36 signal Sreg0, NextState_Sreg0: Sreg0_type;
37 begin
38 -----
39 -- Machine: Sreg0
40 -----
41 -- Next State Logic (combinatorial)
42 -----
43 Sreg0_NextState: process (E, Sreg0)
44 begin
45     NextState_Sreg0 <= Sreg0;
46     case Sreg0 is
47         when S1 =>
48             if E='0' then
49                 NextState_Sreg0 <= S1;
50             elsif E='1' then
51                 NextState_Sreg0 <= S2;
52             end if;
53         ...
54         when S4 =>
55             if E='0' then
56                 NextState_Sreg0 <= S4;
57             elsif E='1' then
58                 NextState_Sreg0 <= S1;
59             end if;
60         when others =>
61             null;
62     end case;
63 end process;
64 -----
65 -- Output Block
66 -----
67 Sreg0_OutputBlock: process (Sreg0)
68 begin
69     -- Set default values for outputs and signals
70     Q <= "00";
71     case Sreg0 is
72         when S1 =>
73             Q <= "00";
74         when S2 =>
75             Q <= "01";
76         when S3 =>
77             Q <= "10";
78         when S4 =>
79             Q <= "11";
80         when others =>
81             null;
82     end case;
83 end process;
84 -----
85 -- Current State Logic (sequential)
86 -----
87 Sreg0_CurrentState: process (C, r)
88 begin
89     if R='1' then
90         Sreg0 <= S1;
91     elsif C'event and C = '1' then
92         Sreg0 <= NextState_Sreg0;
93     end if;
94 end process;
95 end count_e_arch;

```

a



6

Рис. 4. а) VHDL-модель ЦА, реализующего двоичный счетчик с входом разрешения E (стиль кодирования — Three Processes); б) имена процессов в окне просмотра проекта Design Browser

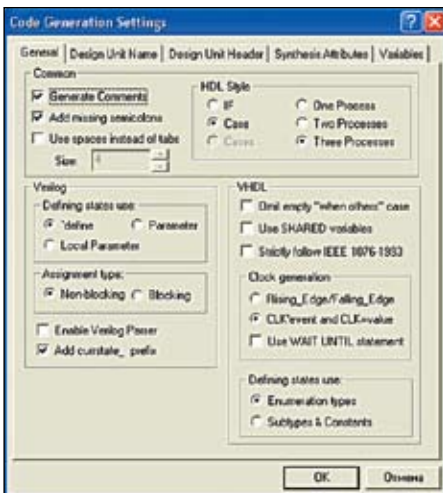


Рис. 5. Диалоговая панель настроек генератора кода редактора SDE

панели **Code Generation Settings** (рис. 5), вызываемой из меню **FSM**. Нам уже приходилось встречаться с этой панелью.

А теперь познакомимся с содержимым каждого процесса, сохраняя прежний порядок рассмотрения. Автоматный процесс **Sreg0\_CurrentState** (рис. 6) описывает поведение ЦА.

```

-- Current State Logic (sequential)
Sreg0_CurrentState: process (C, r)
begin
    if R='1' then
        Sreg0 <= S1;
    elsif C'event and C = '1' then
        Sreg0 <= NextState_Sreg0;
    end if;
end process;

```

Рис. 6. Основной автоматный процесс Sreg0\_CurrentState

```

-- Next State Logic (combinatorial)
Sreg0_NextState: process (E, Sreg0)
begin
    NextState_Sreg0 <= Sreg0;
    case Sreg0 is
        when S1 =>
            if E='0' then
                NextState_Sreg0 <= S1;
            elsif E='1' then
                NextState_Sreg0 <= S2;
            end if;
        ...
        when S4 =>
            if E='0' then
                NextState_Sreg0 <= S4;
            elsif E='1' then
                NextState_Sreg0 <= S1;
            end if;
        when others =>
            null;
    end case;
end process;

```

Рис. 7. Содержимое процесса Sreg0\_NextState (фрагмент), вычисляющего следующее состояние автомата

С приходом сигнала сброса ( $R=1$ ) автомат устанавливается в исходное состояние **S1** ( $Sreg0 \leq S1$ ). С появлением фронта тактирующего сигнала **C** ( $C'$ event and  $C = 1$ ) автомат переходит в следующее состояние ( $Sreg0 \leq NextState\_Sreg0$ ).

Заметим, что следующее состояние — это не обязательно другое, отличное от текущего состояние. Сейчас мы убедимся в сказанном. Посмотрим, как вычисляется следующее состояние в процессе **Sreg0\_NextState** (рис. 7).

Процесс **Sreg0\_NextState** запускается, если изменяется вход **E** или текущее состояние **Sreg0** регистра состояния. Последнее событие может произойти при выполнении предыдущего процесса **Sreg0\_CurrentState**.

Обратите внимание, следующее состояние будет оставаться неизменным, если отсутствует сигнал разрешения счета, то есть  $E=0$ . Например, для текущего состояния **S1** при  $E=0$  выполняется оператор  $NextState\_Sreg0 \leq S1$ , то есть на следующем такте работы автомата сохраняется старое состояние **S1**.

Выходные сигналы вычисляются в процессе **Sreg0\_OutputBlock** (рис. 8), который

```

-- Output Block
Sreg0_OutputBlock: process (Sreg0)
begin
    -- Set default values for outputs and signals
    Q <= "00";
    case Sreg0 is
        when S1 =>
            Q <= "00";
        when S2 =>
            Q <= "01";
        when S3 =>
            Q <= "10";
        when S4 =>
            Q <= "11";
        when others =>
            null;
    end case;
end process;

```

Рис. 8. Содержимое процесса Sreg0\_OutputBlock, вычисляющего значения выходных сигналов

запускается только при изменении текущего состояния **Sreg0**. Например, если автомат переходит в исходное состояние **S1**, то выполняется оператор  $Q \leq "00"$ , и счетчик сбрасывается в ноль.

Для пользователей, знакомых с языком **VHDL**, написать рассмотренные на рис. 6–8 коды не представляет особого труда. Следовательно, для них появляется возможность сразу получить текстовое описание автомата, минуя его промежуточное графическое представление в форме диаграммы состояний.

Более того, в пакете **Active-HDL** имеется специальный инструмент, позволяющий преобразовать (конвертировать) исходное текстовое описание автомата в графическое.

### Конвертор текстового описания проекта в графическое

Он носит имя **Code2Graphics** и вызывается соответствующей командой из меню **Tools** («Инструменты») Code2Graphics Conversion Wizard... CHM.

Конвертор кода в графику достаточно прост в применении, и мы не станем подробно описывать его работу. Отметим лишь, что он позволяет преобразовывать текстовые проекты не только в диаграммы состояний ЦА, но и в блок-диаграммы (схемы). При

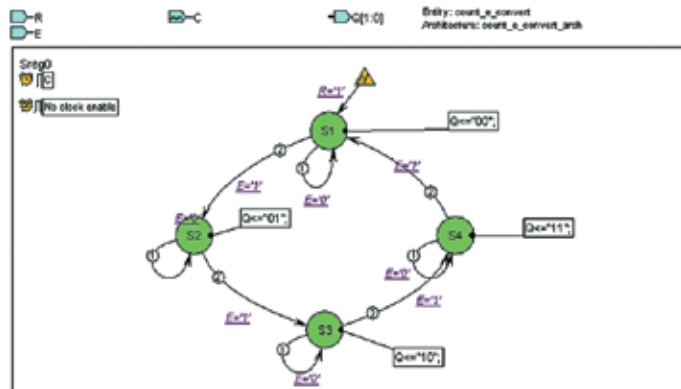


Рис. 9. Результат работы преобразователя кода в графику Code2Graphics

этом конвертор, анализируя код, сам разбирается, что делать — схему или диаграмму состояний.

Преобразование текстового описания в графическое удобно выполнять с помощью соответствующего «мастера», вызываемого из меню Tools командой **Code2Graphics Conversion Wizard**.


Заметим, что данная команда дублируется иконкой , но в стандартной настройке пакета она не видна. При желании вы можете извлечь данную пиктограмму из «запасников», активизировав диалоговую панель **Customize**.

Диаграмма состояний, сгенерированная по тексту последнего примера (рис. 6–8), выглядит так, как показано на рис. 9.

Сравните ее с оригиналом (рис. 3) и вы увидите, что качество автоматически созданного графического описания вполне приемлемо. Откомпилируйте диаграмму состояний и выполните ее моделирование, чтобы убедиться в том, что ошибок в ней нет.

Главное отличие автоматически полученной диаграммы состояний — на всех альтернативных переходах программа проставляет уровни приоритетов (цифры 1 и 2 в кружках). Приоритеты назначаются чисто формальным путем — в порядке записи условий в операторе **IF**. В данном случае взаимно исключающие условия переходов не нуждались в задании приоритетов, но программа не обратила внимания на такую «мелочь».

На диалоговой панели **Code Generation Settings** (рис. 5) в разделе **HDL Style** можно выбрать еще два варианта — **Two Processes** и **One Process**. Любопытно посмотреть, каким будет в этих режимах сгенерированный VHDL-код. Заметим, что по умолчанию задается режим **One Process** — один процесс.

### Модель ЦА в виде двух процессов

Сначала исследуем режим **Two Processes**, который порождает VHDL-код, показанный на рис. 10. В нем отсутствует процесс **Sreg0\_OutputBlock**, где ранее вычислялись выходные сигналы.

Система объединила два процесса **Sreg0\_OutputBlock** и **Sreg0\_NextState**, имитирующих работу комбинационных блоков автомата, в один процесс **Sreg0\_NextState**.

Раскрыв его содержимое (рис. 11), легко заметить, что для каждого состояния в сгенерированный текст добавилось по оператору назначения выходного сигнала. Например, для состояния S1 — это оператор **Q<="00"**;

### Модель ЦА в виде одного процесса

В режиме **One Process** картина выглядит еще более любопытной (рис. 12). Единственный процесс **Sreg0\_machine** реализует

```
architecture count_e_arch of count_E is
type Sreg0_type is (
S1, S2, S3, S4);
signal Sreg0, NextState_Sreg0: Sreg0_type;
begin
-----
-- Machine: Sreg0
-----
-- Next State Logic (combinatorial)
Sreg0_NextState: process (E, Sreg0)
begin
-----
-- Current State Logic (sequential)
Sreg0_CurrentState: process (C, r)
end count_E_arch;
```

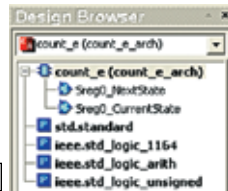


Рис. 10. а) VHDL-модель ЦА, реализующего двоичный счетчик с входом разрешения E (стиль кодирования Two Processes); б) имена процессов в окне просмотра проекта Design Browser

```
-----
-- Next State Logic (combinatorial)
Sreg0_NextState: process (E, Sreg0)
begin
NextState_Sreg0 <= Sreg0;
-- Set default values for outputs and signals
Q <= "00";
case Sreg0 is
when S1 =>
Q <= "00";
if E='0' then
NextState_Sreg0 <= S1;
elsif E='1' then
NextState_Sreg0 <= S2;
end if;
...
when S4 =>
Q <= "11";
if E='0' then
NextState_Sreg0 <= S4;
elsif E='1' then
NextState_Sreg0 <= S1;
end if;
when others =>
null;
end case;
end process;
```

Рис. 11. Содержимое процесса Sreg0\_NextState (фрагмент), вычисляющего следующее состояние автомата

```
architecture count_e_arch of count_E is
type Sreg0_type is (
S1, S2, S3, S4);
signal Sreg0: Sreg0_type;
begin
-----
-- Machine: Sreg0
-----
Sreg0_machine: process (C, r)
-- signal assignment statements for combinational outputs
Q_assignment:
Q <= "00" when (Sreg0 = S1) else
"01" when (Sreg0 = S2) else
"10" when (Sreg0 = S3) else
"11" when (Sreg0 = S4) else
"00";
end count_E_arch;
```

Рис. 12. VHDL-модель ЦА, реализующего двоичный счетчик с входом разрешения E (стиль кодирования One Process)

поведение автомата, а комбинационный выходной сигнал **Q** вычисляется параллельным оператором условного назначения с меткой **Q\_assignment**.

Здесь следует сделать оговорку. Формально VHDL-модель автомата, представленная на рис. 12, содержит один процесс. Во всяком случае, ключевое слово “process” встречается там только один раз.

В действительности же оператор условного назначения есть не что иное, как краткая (неявная) форма записи процесса с одним выходным сигналом. Поэтому можно констатировать, что процессов все-таки два.


Сказанное подтверждается содержимым окна просмотра проекта **Design Browser** (рис. 13). Именно кружком со стрелками, то есть вот так , принято обозначать процесс в пакете **Active-HDL 7.1**. А на рисунке таких кружков все-таки два.



Рис. 13. Окно просмотра проекта Design Browser

Кроме рассмотренных выше стилей HDL-кодирования, вы можете изменить заданные по умолчанию установки и в некоторых других разделах. В частности, в разделе **Clock generation** (рис. 14) имеется возможность выбрать способ генерации тактирующего сигнала: вместо заданной по умолчанию строки **C'event and C = '1'** будет сгенерирована строка **rising\_edge (C)**. На первый взгляд никакой разницы нет — в обоих случаях выявляется фронт (**rising**) сигнала **C**.



Рис. 14. Раздел Clock generation



На самом деле запись **C'event and C = '1'** менее точна, так как она проверяет только факт переключения сигнала C (**C'event**) и его новое значение (**C = '1'**). При этом остается неясным, с какого уровня произошло переключение тактирующего сигнала C на '1'.

Стандартная функция **rising\_edge (C)** ко всему прочему проверяет, было ли предшествующее значение на входе C нулем (**C'last\_value = '0'**). В противном случае функция не возвратит истинное значение, и система забракует такое подозрительное «переключение».

В принципе для обнаружения фронта (или среза) тактирующего сигнала можно использовать оператор ожидания **Wait** (рис. 15). В этом режиме в текст автоматически генерируемого VHDL-кода будет включаться строка: **wait until C'event and C = '1'**;




Рис. 15. Использование оператора ожидания Wait

Однако данная опция имеет ограничения и будет работать только при тактируемом сбросе  автомата. Для асинхронного сброса  она игнорируется, о чем говорит соответствующее предупреждение в окне **Console**:

```
* # Warning: FSM2HD6_3047 count_E.asf : Asynchronous reset detected. Option "Use WAIT UNTIL statement" will be ignored.
```

Ну и, конечно же, следует помнить, что оператор ожидания **Wait** при синтезе проекта может преподнести неприятные сюрпризы.

Чтобы пользователь ощущал себя комфортно в процессе проектирования диаграмм состояний, разработчики редактора **State Diagram Editor** снабдили его дополнительными инструментами. Наибольшее внимания заслуживает программа **View/Sort Object**. Мы с ней встречались в начале 12-го урока. Чтобы активизировать данный инструмент, следует выбрать одноименную команду в меню **FSM** или щелкнуть по кнопке 

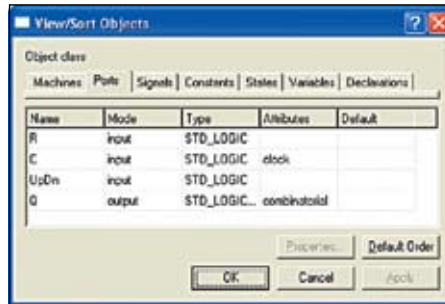
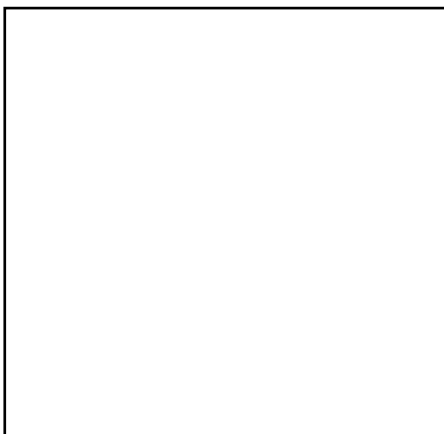


Рис. 16. Диалоговая панель вспомогательного инструмента View/Sort Object

расположенной на инструментальной панели **FSM Hierarchy Toolbar**.

На открывшейся диалоговой панели (рис. 16) приводится информация об основных объектах проектируемой диаграммы состояний.

Каждый класс объектов диаграммы (автоматы, порты, сигналы, константы, состояния и переменные) представлен на отдельной закладке.

Мы уже знаем, что, используя процедуру **drag and drop**, объекты можно сортировать, определяя тем самым желаемый порядок в процессе автоматической генерации HDL-

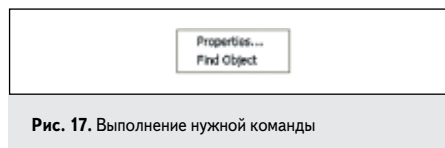


Рис. 17. Выполнение нужной команды

кода. Но это еще не все.

Выделив объект, можно отредактировать его свойства и тут же найти его на диаграмме состояний. Достаточно лишь щелкнуть на объекте правой кнопкой мыши и выполнить нужную команду (рис. 17).

Выбрав команду **Find Object**, вы увидите искомый элемент на диаграмме состояний, обрамленный мигающей рамкой зеленого цвета.

Обычно диаграмма состояний содержит описание одного автомата, однако она позволяет поместить на чертеж и несколько взаимосвязанных автоматов. С такой ситуацией мы встречались на 12-м уроке (рис. 17). В этом случае на закладках **States**, **Variables** и **Declarations** появляется дополнительное поле  , в котором можно выбрать конкретный автомат и тем самым сузить область поиска.

Еще одним вспомогательным инструментом является так называемый **ASF-отчет**



Рис. 18. Сообщение об успешном завершении операции

(**ASF-report**). Он создается в **HTML**-формате и содержит полную информацию о проекте (используемый язык, имя модуля, подключенные к проекту библиотеки и пакеты, описание портов, сигналов, переменных и констант, тип автомата, имя синхросигнала, активный фронт, способ кодирования состояний, дерево иерархии).

Чтобы сгенерировать отчет, выполните команду **ASF Report** из меню **ASF**. В случае успешного завершения операции вы получите следующее сообщение (рис. 18), а в окне **Console** увидите такой текст (рис. 19).

В нем констатируется, что результаты записаны в файл сообщений (**log file**), и предлагается дважды щелкнуть по последней строке, чтобы увидеть его содержимое.

Если данное окно в настоящий момент закрыто, посмотреть файл сообщений можно из окна проекта **Design Browser**. Для этого придется активизировать правую закладку **Resources**, раскрыть папку **log** и выбрать желаемый файл с расширением **\*.htm**.

Еще одно замечание касается редактирования текстов, размещенных на диаграмме состояний. У вас есть несколько возможностей. Двойным щелчком на редактируемом тексте можно открыть небольшое окно прямо на диаграмме и внести требуемые изменения. Этим способом редактируются имена и параметры портов, сигналов, переменных и констант, условия на переходах, текст комментариев, имена состояний.

Более универсальным способом является выбор команды **Properties** из всплывающего меню. Он работает для всех объектов.

Но не стоит игнорировать и команду контекстного меню **Edit Using HDL**, которая вызывает диалоговое окно текстового редактора **HDL Editor** со многими его дополнительными возможностями. Правда, эта команда доступна не для всех текстов, а только для тех, что записаны в формате целевого языка описания аппаратуры. Например, так можно редактировать условия на переходах, действия для состояний и некоторые другие HDL-описания. ■

*Окончание следует*

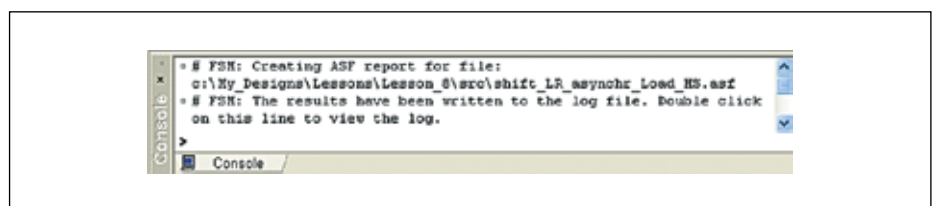


Рис. 19. Окно Console