Проектирование аппаратных схем ПЛИС средствами языка VHDL

Оглавление

Глава 1

Введение. Схемотехника комбинационных устройств.

История развития интегральных схем (от транзистора до ПЛИС). Классификация современных ИС по методу соединения элементов. Общая архитектура ПЛИС. Конфигурационная память и возможность реконфигурации. Обзор семейства ПЛИС фирмы Xilinx. Применение ПЛИС в современном мире. Логические элементы в логике КМОП. Тристабильные элементы. Простые комбинационные элементы.

Вывод на консоль во время симуляции

Глава 2

Основы VHDL для синтеза и моделирования

Введение в VHDL. Синтаксис и параллельная семантика. Объекты и конструкции. Типы данных. Логические и арифметические операторы языка. Синтезируемое подмножество языка VHDL и конструкции для моделирования. Создание верификационных testbenches.

Вывод на консоль во время симуляции

Глава 3

Моделирование схем на языке VHDL

Введение в моделирование. Визуальный анализ значений сигналов. Вывод на консоль во время моделирования. Файловый ввод/вывод. Концепция самопроверяющихся testbenches.

Введение в моделирование

При разработке серьезных аппаратных проектов значительную роль отводят процессу моделирования. Моделирование - это процесс отладки или верификации, направленный на выявление логических ошибок в работе схемы.[1] Как правило, проект может быть разбит на *модули*, которые можно верифицировать независимо друг от друга. Этим никогда нельзя пренебрегать, и писать такие unit-тесты на модули - хорошая практика. Второй подход, дополняющий создание unit-тестов, - это верифицировать весь проект (или большую часть проекта) целиком, учитывая взаимодействие между модулями. В этом случае говорят о создании интеграционнного теста. Независимо от того, какой тест разрабатывается (юнит или интеграционный), первой задачей является определение одного VHDL модуля, внутри которого находится схема для проверки. Для юнит теста, когда верифицируется один VHDL модуль - это задача тривиальная, а для интеграционного теста необходимо создать модуль обертку *wrapper*, внутрь которого поместить все интересные для теста модули. Этот единственный модуль верхнего уровня принято называть Design Under Test или просто DUT. Модуль DUT затем вставляется как компонент в специальный VHDL модуль, называемый тестбенч (testbench).

Особенностями модуля тестбенч является следующее:

1. Он не имеет внешних портов ввода/вывода. Тестбенч - это виртуальный

[1] Можно еще отдельно выделить моделирование, с учетом времянных задержек элементов схемы, направленное на выявление времянных ошибок, но в большинстве реальных случаев при разработке схемы на FPGA возможно обойтись без него, и этот процесс не будет рассмотрен в данном курсе.

стенд тестирования схемы DUT. В тестбенче происходит генерация входных сигналов DUT и анализ выходных.

2. В этом модуле можно и нужно использовать *несинтезируемые* VHDL контрукции.

VHDL конструкции для синтеза и для моделирования

Язык VHDL разрабатывался в первую очередь для описания аппаратных схем. Однако сразу было ясно, что одного описания схем мало, необходимо было сразу закладывать в язык способность моделировать схемы на компьютере для их верификации. Для решения этой задачи было реализовано большое количество операторов, типов данных, конструкций и функций, присущих традиционным языкам программирования. К ним относятся всевозможные операторы циклов (*while, for*), файловый ввод/вывод, специальный тип данных *time* и др. Однако, большая часть этих не может быть синтезирована [2]. Язык VHDL описывает аппаратуру на очень низком уровне булевых выражений, элементарных ячеек памяти и с точностью до 1 такта системной частоты. Это пространство объектов должно отображаться на элементную базу технологии, по которой будет реализована схема [3]. Элементная база всегда довольно ограничена, а значит и конструкции языка, для ее описания должны быть четкими и однозначными. Из этого следует, что *НЕ ЛЮБОЙ* алгоритм, который можно описать на VHDL можно реализовать в виде аппаратной схемы, или как говорят *синтезировать*. Из всех конструкций языка выделяют *синтезируемое подмножество*, которое можно использовать для описания аппаратных схем **??**. Остальные конструкции используются для моделирования. Синтезируемым констукциям VHDL будет посвящена большая часть данного курса. В этой же главе будет уделено внимание в первую очередь конструкциям для моделирования.

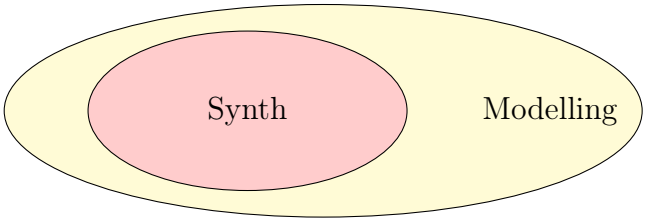

Рис. 3.1: Синтезируемое подмножество языка VHDL.

Однако начать рассмотрение следует со специального оператора *process*, который широко используется как при синтезе так и при моделировании схемы.

[2] И правда, сложно представить себе аппаратную схему, способную выводить данные непосредственно на монитор компьютера с помощью одной лишь функции, похожей на *printf*

[3] Для ПЛИС - это существующие в каждом логическом блоке CLB таблицы соответствия LUT для реализации булевых выражений, триггеры и другие элементы архитектуры

Оператор *process*

Уже было сказано, что VHDL обладает параллельной семантикой, т.е. в этом языке есть операторы, которые выполняются параллельно или одновременно. Для описания аппаратуры эта семантика подходит как нельзя лучше, ведь электрические сигналы не идут друг за другом, а распространяются одновременно от всех модулей системы. При этом поведение самого модуля может быть довольно сложным и может описываться последовательными операциями. Для этого и существует специальный оператор *process*. Использовать его можно в любом месте архитектуры модуля, и оператор является параллельным, однако внутри него выполнение операторов идет последовательно.

```
  process_label: process ( список чувствительности )
1   -- объявления процесса
2  begin
3   -- тело процесса
4   -- последовательные операторы
5  end process process_label;
6 end behavior;
```

Листинг 3.1: Оператор *process*

На листинте **??** отражена общая структура процесса. Процесс может начинаться и заканчиваться с необязательной метки процесса (process_label), которая может быть полезна для идентификации процесса при моделировании либо документировании кода. В подразделе объявлений процесса могут быть объявлены подпрограммы, типы, константы, переменные (*variable*), файлы и др. Сигналы в подразделе объявлений оператора процесса объявлены быть не могут. Сам оператор процесса является параллельным оператором. Если в теле архитектуры модуля разместить несколько процессов, то они будут выполняться параллельно. Но внутри каждого процесса находятся операторы, выполняемые последовательно, т.е. один за другим. Список чувствительности – это список запускающих процесс сигналов. Когда один из сигналов, указанных в списке чувствительности, меняет свое значение, процесс запускается, и последовательные операторы выполняются до конца процесса. После этого процесс приостанавливает свою работу до тех пор, пока опять какой-нибудь сигнал из списка чувствительности не изменит свое значение. Таким образом, процесс может находиться в одном из двух состояний: запущенном и приостановленном. Пример процесса со списком чувствительности приведен в листинге **??**.

```
   architecture behavior of process_example is
1    signal x,y : integer;
2  begin
3    process (x)
4      -- объявления процесса
5      variable a,b,c : integer;
6    begin
7      -- тело процесса
8      a := x*x;
9      b := 2*x;
10     c := 1;
11     y <= a + b;
12     y <= a + b + c;
13   end process;
14 end behavior;
```

Листинг 3.2: Процесс с внутренними переменными и списком чувствительности

В данном случае в архитектуре модуля объявлено 2 сигнала типа *integer* - x и y. Сигнал x - входной сигнал для процесса, он объявден в списке чувствительности. При любом изменении сигнала x, процесс будет запускаться, и последовательные операторы будут выполняться до конца процесса, в конце которого сигналу y будет присвоено новое значение. Следует обратить внимание на раличия между сигналами и переменными. Областью видимости сигналов является вся архитектура модуля, т.е. он может быть использован в любом параллельном операторе этого модуля. Переменные же могут быть объявлены только внутри процесса, и их область видимости ограничена этим процессом. Также существует разница в операторах присвоения значений сигналам и переменным. Для переменных оператор присваивания обозначается как :=, а для сигналов <=. Присваивание переменных является блокирующим (blocking), т.е. значение переменной будет обновлено мгновенно. На практике это означает, что внутри процесса можно присваивать разные значения одной переменной несколько раз, и каждый раз оно будет меняться. При этом присваивание сигналов является неблокирующем (nonblocking), т.е. значение сигнала не изменится до тех пор пока процесс не будет завершен, при этом сигнал сразу примет последнее присвоенное значение. Это означает, что сигнал y никогда не примет значение $a+b$, присвоенное ему на 11 строчке листинга **??**, а сразу примет значение $a+b+c$, присвоенное на 12 строчке.

Термины blocking и nonblocking идут из системного программирования при осуществленнии вывода из программы, работающей под управлением операционной системы. Blocking означает, что процесс блокируется до окончания операции записи, а при осуществлении nonblocking записи процесс не будет дожидаться полного окончания вывода данных (например на экран или диск), а просто скопирует данные для вывода в другой процесс операционной системы, и сам продолжит выполнение. Аналогия с VHDL тут такая, что при записи в переменную (blocking) процесс как бы останавливается (не выполняется дальше), пока

значение переменной не изменится [4], а при назначении сигнала (nonblocking) процесс продолжает выполнение, а значение сигнала обновится чуть позже [5].

Процессы с пустым списком чувствительности

Бывают процессы с пустым списком чувствительности. В этом случае они всегда готовы к запуску, и если не принимать специальных будут запускаться всегда, мнгновенно выполняться и опять запускаться. Симуляция в этом случае зависнет, потому что будет занята постоянным запуском одного процесса в один и тот же момент времени, и время моделирования не будет увеличиваться. Тем не менее при моделировании схем процессы без списка чувствительности очень часто применяются. Внутри таких процессов применяются инструкции ожидания, выполняя которые процесс приостанавливается (или засыпает), чтобы продолжить выполнение с места остановки при наступлении события, которое процесс ожидал. Все конструкции ожидания реализуются с помощью последовательного оператора *wait*. Существуют несколько типов ожиданий. Рассмотрим их на примере листинга **??**.

```
  process
1 begin
2     ...
3     -- ожидание изменения любого сигнала из списка
4     wait on A, B;
5     -- ожидание наступления условия
6     wait until RST = '0';
7     -- ожидание в течение определенного времени
8     wait for 30 ns;
9     -- бесконечное ожидание
10    wait;
11 end process;
12 end behavior;
```

Листинг 3.3: Процесс с пустым списком чувствительности и оператором *wait*

Во-первых, можно ожидать изменения какого-либо сигнала.

```
wait on A, B;
```

При исполнении этой инсрукции процесс приостановится и продолжит выполнение при измененеии сигнала любого сигнала из списка (A, B). Во-вторых, можно ожидать выполнение условия.

```
wait until RST = '0';
```

В-третьих, можно ждать в течение заданного времени. При этом время задается в абсотютных единицах типа микро, нано или пикосекундах.

[4] Впрочем, значение переменной изменяется мгновенно, поэтому никакой задержки выполнения процесса не происходит

[5] Когда процесс будет завершен

```
wait for 30 ns;
```

Наконец, можно заставить процесс заснуть навсегда. Если от процесса требуется работа в течение ограниченного отрезка времени (например формирование непериодического воздействия), то следует в конце процесса поместить констукцию

```
wait;
```

Иначе процесс закончится и тут же начнется заново.

Конструкции ожидания практически не используются для синтеза [6], и не рекомендуются в этом курсе. Для синтеза всегда оказывается возможным использовать другие, более выразительные элементы языка.

Моделирование времянных задержек

При назначении сигналов с помощью конструкции *after* можно указывать времянную задержку, по прошествии которой сигнал изменит свое значение (листинг **??**). Применяется она либо для моделирования задержек на логических элементах, либо для формирования периодического сигнала (строчка 5)

```
   architecture DELAYS of X is
1    constant PERIOD : time := 10 ns;
2  begin
3    SUM   <= A xor B after 5 ns;
4    CARRY <= A and B after 3 ns;
5    CLK   <= not CLK after PERIOD/2;
6  end DELAYS;
```

Листинг 3.4: Назначение сигналов с задержкой

Конструкция *after* не может быть синтезирована.

Операторы циклов внутри процессов

Внутри процессов можно использовать операторы циклов, похожие на циклы из последовательных языков программирования. Операторы циклов синтезируемы только частично. Дело в том, что на каждую итерацию цикла создается своя цифровая схема, поэтому циклы, число итераций которых неизвестно во время компиляции кода, а также бесконечные циклы, не могут быть синтезированы. Циклы очень удобны и часто применяются при моделировании, при синтезе следует ограничиваться использованием циклов для перебора и совершения действий над ограниченным набором операндов, например над битами шины данных.

Общий синтаксис циклов представлен на листинге **??**.

[6] За исключением конструкции wait on, которая может теоретически применяться для синтеза комбинационной и даже последовательной логики

```
loop_label: while condition loop
  -- последовательные операторы
  end loop loop_label;

loop_label: for loop_parameter in range loop
  -- последовательные операторы
  end loop loop_label;
```

У циклов, как и у процессов может быть необязательня метка цикла (loop_label). Оператор цикла может иметь несколько форм в зависимости от итерационной схемы, задаваемой непосредственно перед зарезервированным словом *loop*. В тривиальном случае, когда итерационная схема отсутствует, цикл становится бесконечным (листинг **??**). Для выхода из бесконечного цикла необходимо использовать оператор *exit* вместе с условием выхода (листинг **??**). Вместо использования бесконечного цикла с условием выхода можно использовать цикл в форме *while*. В этом случае перед циклом указывается зарезервиванное слово *while* и условие *входа в цикл*. Последовательность операторов внутри цикла будет выполнена, если условие равно *true*. Условие проверяется перед каждой итерацией цикла. Если перед очередной итерацией условие равно *false*, то цикл не выполняется, а управление переходит следующим за ним операторам (листинг **??**). Когда заранее известно количество итераций, может быть использована итерационная схема типа *for*. В этом случае перед циклом указывается зарезервированное слово *for*, итератор цикла и его границы. Для каждой итерации цикла итератор будет принимать целые значения из указанного диапазона (листинг **??**).

```
signal Clock : BIT := '0';
Clk_1: process (Clock)
begin
  L1: loop
    Clock <= not Clock after 5 ns;
  end loop L1;
end process Clk_1;
```

Листинг 3.5: Бесконечный цикл

```
L2: loop
  A:= A+1;
  exit L2 when A > 10;
end loop L2;
```

Листинг 3.6: Бесконечный цикл с условием выхода

```
   Shift_3: process (Input_X)
1    variable i : POSITIVE := 1;
2  begin
3    L3: while i <= 8 loop
4      Output_X(i) <= Input_X(i+8) after 5 ns;
5        i := i + 1;
6    end loop L3;
7  end process Shift_3;
```

Листинг 3.7: Цикл While

```
   Shift_4: process (Input_X)
1  begin
2    L4: for count_value in 1 to 8 loop
3      Output_X(count_value) <= Input_X(count_value + 8) after
       5 ns;
4    end loop L4;
5  end process Shift_4;
```

Листинг 3.8: Цикл For

Лабораторная работа. Моделирование сумматора и визуальный анализ сигналов

1. Открыть Vivado проект из lab2-1

2. Добавить в проект файл тестбенча. Для этого а) Под закладкой *Project Manager* нажать *Add Souces*, выбрать пункт *Add or Create Simulation Sources* (см. рис. **??**).

   б) В следующем окне нажать *Add Files* и выбрать файл *02-vhdl_basics/lab2-1/src/adder_tb_simple.vhd*. После выбора файла окно должно выглядеть как на рис. **??**. Нажать кнопку *Finish*. В итоге в Vivado окошко *Sources* должно выглядеть следующим образом (рис. **??**).

3. Открыть окно *Simulation Settings* и настроить полное время моделирования схемы (см. рис. **??**). Оставить параметр *xsim.simulate.runtime* равным 1000ns.

4. Запустить симуляцию. Для этого выбрать *Simulation->Run Simulation->Run Behavioral Simulation*. Откроется три окна. В первом (*Scopes*) можно видеть иерархию VHDL модулей. Во втором (*Objects*) отображается список сигналов этого модуля. В окне *Scopes* можно выбрать любой модуль иерархии, а потом в окне *Objects* выбрать интересующие сигналы и добавить их на времянную форму, нажав правой кнопкой мыши по выбранным

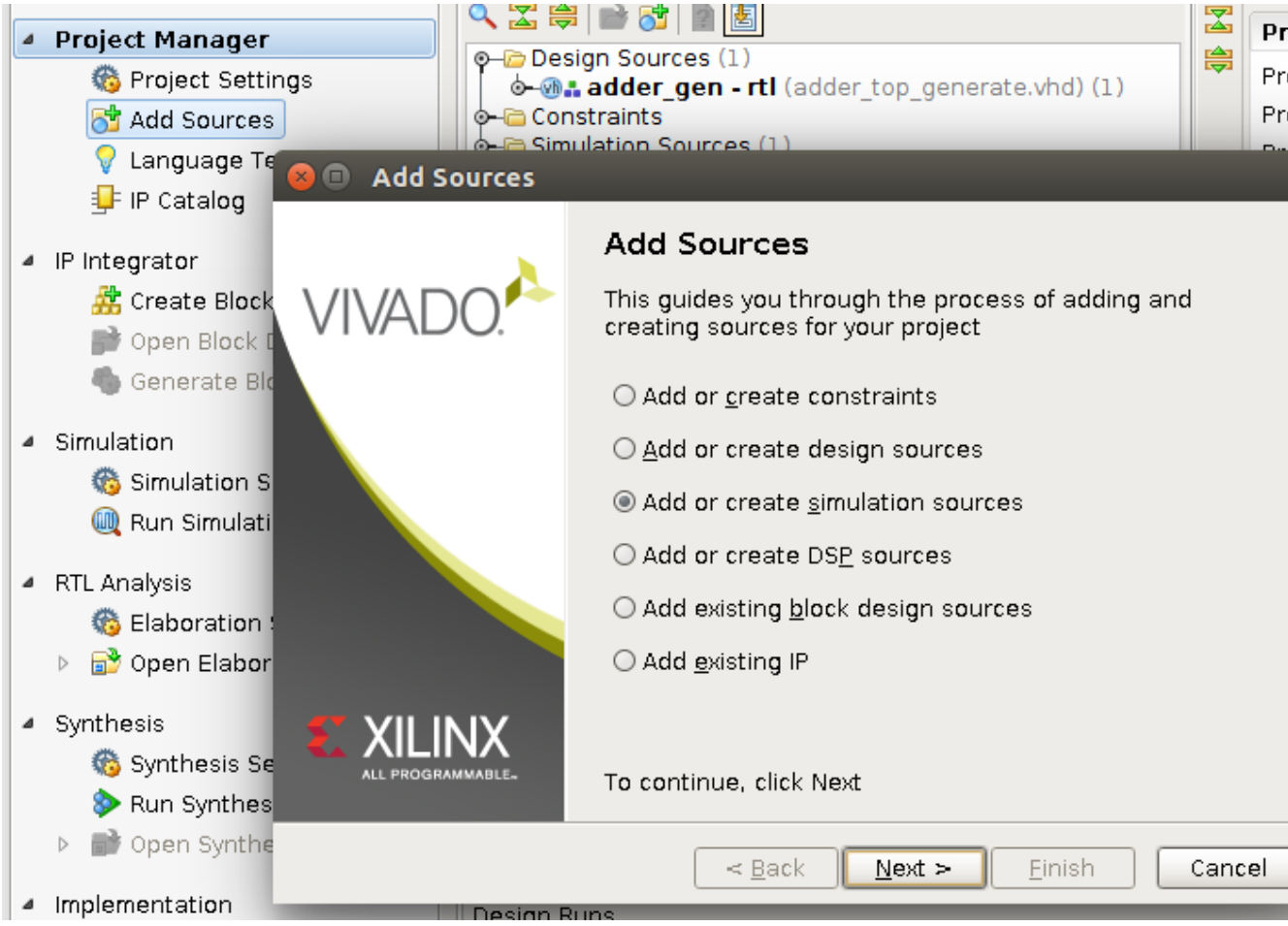

Рис. 3.2: Выбор типа исходных файлов

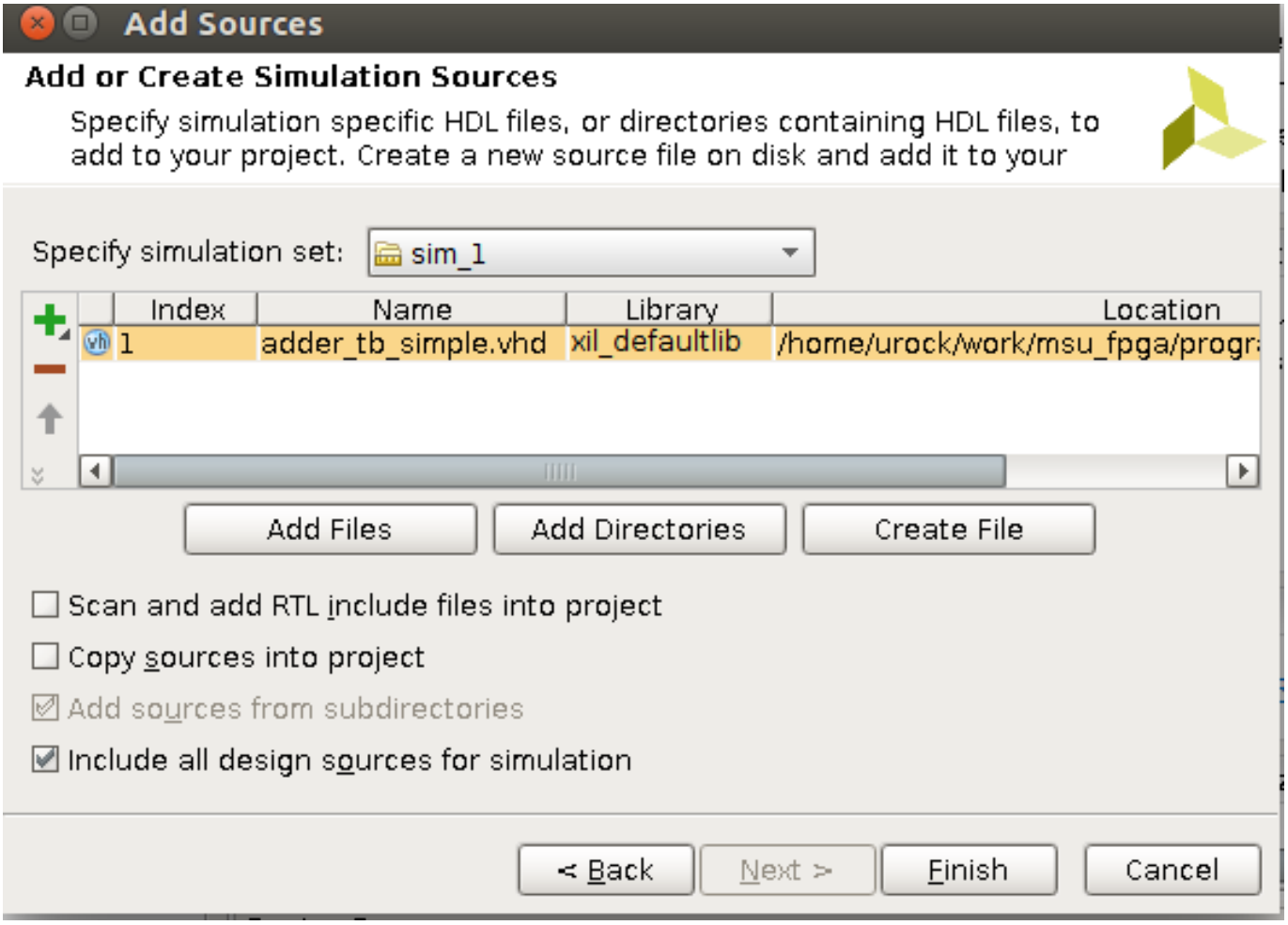

Рис. 3.3: Выбор файла тестбенча

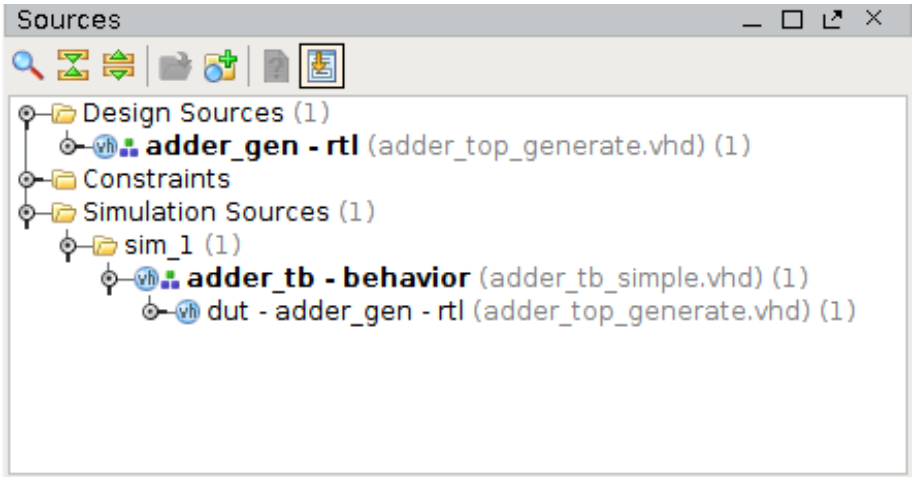

Рис. 3.4: Окно Sources после добавления файла тестбенча

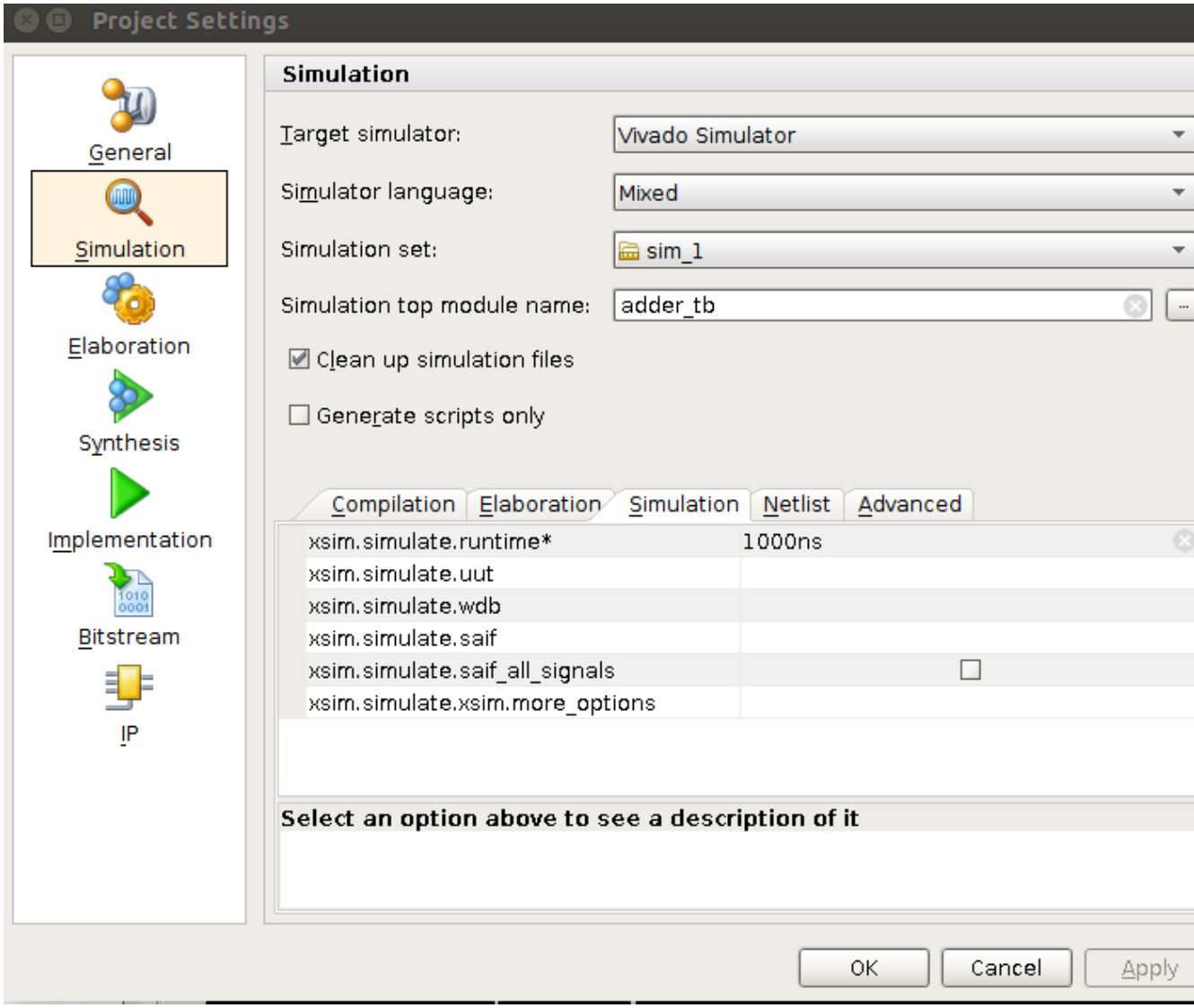

Рис. 3.5: Окно параметров моделирования

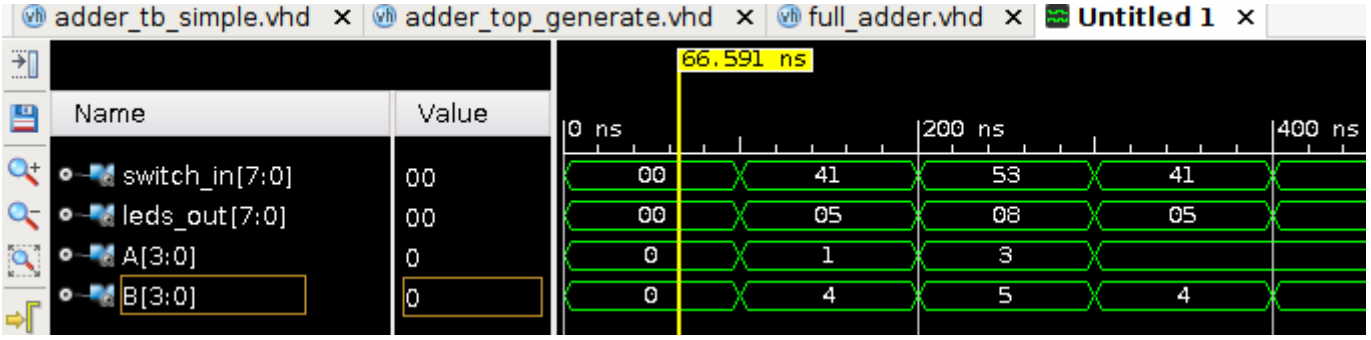

Рис. 3.6: Окно вейвформ сигналов

сигналам и выбрав *Add to Wave Window*. В третьем окне открывается временные диаграммы (вейвформы) сигналов (см. рис. **??**). Чтобы в окне вейвформ было показано полное время симуляции, надо нажать правой кнопкой и выбрать *Full View*. Потом можно приблизить любой участок вейвформы.

5. Разобрать текст тестбенча (см. листинг **??**). В строчках 3-4 задается название верхнего VHDL модуля для симуляции. Обратите внимание, что этот модуль не имеет интерфейса. Затем в разделе объявлений архитектуры определяются компонент DUT (наша моделируемая схема) и подключаемые к этому компоненту сигналы. В тестбенче будут заданы последовательности значений, подаваемых на входы схемы DUT. После *begin* сразу идет вставка компонента DUT (строчки 26-29). После чего объявлен всего один процесс, внутри которого генерируются последовательности входных воздействий. В данном случае надо задавать разные слагаемые и смотреть результат суммы в окне вейвформ. Процесс *stim_proc* не имеет списка чувствительности, а это значит, что он запуститься сразу (безусловно), как только начнется симуляция. Сразу же сигналам A и B будут присвоены нулевые значения. Далее через каждые 100 ns (с помощью кострукции *wait for 100 ns;*, которая заставляет процесс засыпать на указанное время) значения A и B будут несколько раз изменены. Читателю предлагается проверить в окне вейвформ, что сумма остается верной. В конце процесса стоит конструкция *wait;*. Это инструкция бесконечного ожидания, выполнив ее процесс заснет навсегда до конца симуляции. Если бы ее не было, то процесс бы завершился, а потом бы опять сразу начался с самого начала, т.к. его список чувствительности пуст.

Листинг 3.9: Простой тестбенч

```
   library ieee;
1  use ieee.std_logic_1164.all;
2
3  entity adder_tb IS
4  end adder_tb;
5
6  architecture behavior of adder_tb IS
7
```

```
 8     component adder_gen
 9     port(
10         switch_in : in  std_logic_vector(7 downto 0);
11         leds_out : out  std_logic_vector(7 downto 0)
12     );
13     end component;
14 
15     --Inputs
16     signal switch_in : std_logic_vector(7 downto 0) := (
       others => '0');
17 
18     --Outputs
19     signal leds_out : std_logic_vector(7 downto 0);
20 
21     signal A, B  : std_logic_vector(3 downto 0);
22 
23 begin
24 
25     -- Instantiate the Design Under Test (DUT)
26     dut: adder_gen port map (
27         switch_in => switch_in,
28         leds_out => leds_out
29     );
30 
31     switch_in <= B & A;
32 
33     -- Stimulus process
34     stim_proc: process
35     begin
36         A <= "0000";
37         B <= "0000";
38         wait for 100 ns;
39 
40         A <= "0001";
41         B <= "0100";
42         wait for 100 ns;
43 
44         A <= "0011";
45         B <= "0101";
46         wait for 100 ns;
47 
48         A <= "0001";
49         B <= "0100";
50         wait for 100 ns;
51 
52         A <= "0001";
```

```
53          B <= "0111";
54          wait;
55      end process;
56  end;
```

6. Далее читателю предлагается самостоятельно поиграть с тестбенчем, изменяя код генерации входных воздействий. Можно задавать любые временные промежутки, другие значения слагаемых. Каждый раз после изменения кода, его необходимо компилировать заново. Для этого удобно использовать кнопку *Relaunch*, которая находится прямо над окном вейвформ. Если код не менялся, а просто были добавлены новые сигналы на вейвформу, то можно не перекомпилировать код, а достаточно просто запустить моделирование с нуля времени с помощью кнопок *Restart* и *Run*, находящихся слева от *Relaunch*. При этом можно обратить внимание, что все нажатия на кнопки дублируются командами в окне *Tcl Console*.

Хорошая практика моделирования

Рассмотренный выше подход является минимальной схемой тестирования, способной выявить лишь малую часть ошибок в схеме. Большие проекты, содержащие сотни управляющих сигналов и шин данных невозможно верифицировать с помощью визуального анализа временных форм сигналов. Необходимо использовать другие методы, самый простой из которых - это создание самопроверяющегося тестбенча, когда на схему подаются известные опорные воздействия, а результат работы схемы автоматически сравнивается с известными опорными результатами (golden results). В итоге разработчику просто выводится сообщение о том, прошел тест или нет. Конечно, написание такого тестбенча иногда бывает не легче, чем разработать саму схему, но процесс отладки (debug) никто не отменял, и умение писать качественные тестбенчи является необходимым для современного инженера - разработчика ПЛИС. Прежде чем перейти к описанию, как создавать такие самопроверяющиеся тестбенчи, необходимо рассмотреть вывод тектовой информации на консоль и файловый ввод/вывод.

Вывод на консоль во время симуляции

В большинстве языков программирования существует стандартный вывод - механизм, обеспечивающий отображение текстовой информации на мониторе во время работы программы. Язык VHDL также предоставляет такую возможность. Но эта возможность не относится к синтезируемому подмножеству языка - текстовый вывод можно осуществлять только из тестбенча во время моделирования разрабатываемой схемы. В качестве консоли будет выступать текстовое окно вывода среды, в которой происходит моделирование, например Vivado. Хотя среда моделирования предоставляет возможность непосредственно наблюдать значения всех сигналов проекта в окне временных диаграмм, часто бывает полезно выводит и текстовые сообщения. Эта функциональность обеспечивается стандартной библиотекой textio. Для того чтобы воспользоваться ее функ-

циями необходимо добавить выражение из листинга **??** непосредственно перед каждой архитектурой, использующей вывод на консоль. На самом деле, с помощью этой библиотеки можно также и осуществлять файловый ввод/вывод, но об этом в следующем параграфе.

```
   library std;
1  use std.textio.all;
```

Листинг 3.10: Объявление библиотеки textio

Текстовая информация может быть выдана через переменную (*variable*) типа *line*. Вывод на консоль может быть осуществлен только изнутри процесса, где все операторы исполняются последовательно. Для вывода информации в консоль, сначала ее необходимо поместить в переменную типа *line*, а затем вызвать специальную функцию для собственно выдачи. Следующий пример это иллюстрирует.

```
   use textio.all;
1  architecture behavior of check is
2  begin
3    process (x)
4      variable s : line;
5      variable cnt : integer:=0;
6    begin
7      if (x='1' and x'last_value='0') then
8        cnt:=cnt+1;
9        if (cnt > max_count) then
10         write(s,"Counter overflow - ");
11         write(s,cnt);
12         writeline(output,s);
13       end if;
14     end if;
15   end process;
16 end behavior;
```

Листинг 3.11: Вывод в консоль

Функция *write()* используется для добавления текстовой информации в конец строковой переменной *s*, которая инициализируется пустой строчкой. Функция *write()* имеет два аргумента, первый - это имя переменной, куда надо добавить данные, второй - это сами данные. В нашем примере сначала в переменную *s* записывается строка *Counter overflow* - , а затем текущее значение счетчика *cnt* конвертируется в строковое представление и добавляется в конец строчки *s*. После этого вызывается функция *writeline()*, которая копирует значение строчки *s* в стандартный вывод среды моделирования (о чем ей сообщает зарезервированное слово *output*, переданное в качестве первого аргумента), после

чего обнуляет эту строчку. К примеру, если значение *MAX_ COUNT* равнялось бы 15, а сигнал x имел бы больше 15 передних фронтов, то в консоле во время моделирования появилось бы сообщение

Counter overflow - 16

Чтобы можно было записывать в переменную типа *line* значения типа *std_ logic*, помимо стандартной библиотеки *textio* необходимо включить еще пакеты

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
```

Файловый ввод/вывод

Очень часто бывает необходимо промоделировать разрабатываемую схему DUT на большом объеме входных данных. В таких случаях входные данные хранятся в файлах на диске, а моделирующая тестбенч читает их и подает на вход схемы DUT. Если в добавок тестбенч самопроверяющаяся, то и выходные данный схемы DUT необходимо сравнивать с известными опорными результатами, согласованными с входными воздействиями. В этом случае также нельзя обойтись без чтения данных из файлов. Запись в файл при моделировании используется нечасто, хотя такая возможность есть. Итак, для использования файлового ввода/выводы надо использовать ту же библиотеку, что и для вывода на экран.

```
library std;
use std.textio.all;
```

Объявление и открытие файла просходит с помощью следующей конструкции

```
file logical_name : file_type is mode "file_name";
```

Ее можно помещать в начале процесса (*proccess*) до слова *begin*, в начало архитектуры (тоже до *begin*), либо в начале процедур и функций (*procedure, function*). Файл сразу же открывается при объявлении и закрывается при выходе из блока, где был объявлен. Поэтому при объявлении в начале процесса или архитектуры, файл открывается в начале симуляции и закрывается в ее конце, а при вызове из процедуры или функции - будет открываться каждый раз при вызове процедуры и закрываться при ее завершении.

Параметр объявления *mode* определят направление потока данных и может принимать значения *"in"* - для файла, открытого на чтение, и *"out"* - для файла, открытого на запись. Непосредственная работа с файлами может быть только из блоков кода, где операторы исполняются последовательно (*process, procedure, function*). Для чтения используются слеюдующие функции: *endfile(), readline(), read()* из библиотки textio. Функция *endfile()* проверяет признак конца файла, *readline()* читает из файла целую строку данных и помещает их в переменную типа *line*. А функция *read()* извлекает данные из переменной типа

line и присваивает их переменным и сигналам, используемым для моделирования. Следующий пример это демонстрирует:

```
READ_FILE: process
  variable VEC_LINE : line;
  variable VEC_VAR : bit_vector(0 to 7);
  file VEC_FILE : text is in "stim.vec";
begin
  while not endfile(VEC_FILE) loop
    readline (VEC_FILE, VEC_LINE);
    read (VEC_LINE, VEC_VAR);
    A_BUS <= VEC_VAR;
    wait for 10 ns;
  end loop;
  wait;
end process READ_FILE;
```

В данном случае строка *stim.vec* - это имя текстового файла, откуда будут прочитаны данные. Эта строчка должна содержать либо абсолютное имя файла в операционной файловой системе, либо относительное для того места, откуда запущена симуляция. Для Vivado это обычно что-то похожее на *project_dir\project_name.sim\behav*.

Данные могут быть и записаны в файл, объявленный с *mode = out*, аналогично выводу на консоль, с помощью функций *write()* и *writeline()* через промежуточную переменную типа *line*. Единственное отличие заключается в том, что вместо зарезервированного слова *output*, обозначающего стандартный вывод программы, надо использовать идентификатор файла.

```
WRITE_FILE: process (CLK)
  variable VEC_LINE : line;
  file VEC_FILE : text is out "results";
begin
  -- strobe OUT_DATA on falling edges
  -- of CLK and write value out to file ыв
  if CLK='0' then
    write (VEC_LINE, OUT_DATA);
    writeline (VEC_FILE, VEC_LINE);
  end if;
end process WRITE_FILE;
```

Функции *read() и write()* из библиотеки textio определены для типов данных *bit, bit_vector, boolean, integer, real, string и time*. По стандарту они не совместимы с типами из пакета *std_logic_1164*, хотя многие среды моделирования (Vivado в том числе) реализуют подержку типов *std_logic и std_logic_vector*.

Самопроверяющейся тестбенч

Имея в арсенале стандартные средства воода/вывода при моделировании схем можно создавать тестбенчи с автопроверкой результата, и использовать визуальный анализ временных форм не для проверки правильности результата, а лишь для отладки схемы. Самопроверяющийся тестбенч основан на следующей идее. Подавая известные тестовые воздействия (входные сигналы) на Design Under Test (DUT), мы можем ожидать известные тестовые отклики (выходные сигналы). Тестовые входные воздействия могут быть как сгенерированны автоматически самим тестбенчем (см. рис. **??**), так и прочитаны из файла (см. рис. **??**). В случае, если мы читаем входные данные из файла, то скорее всего мы имеем и соответствующие им выходные данные тоже в файле, откуда мы можем их прочитать. Если входные данные генерируются, то скорее всего мы можем просто получить соответствующие им выходные данные с помощью несинтезируемых конструкций VHDL. За генерацию или чтение из файла ожидаемых выходных данных отвечает процесс Expected Data Process. В любом случае мы должны сравниить выходные данные из DUT и ожидаемые данные (за это отвечает Compare Data Process), и выдать ошибку, если они не совпадают. В обоих случаях на картинках Output process - это процесс чтения выходных данных из DUT (с использованием протокола конкретной схемы) и сохранения их в переменных или массивах данных тестбенча.

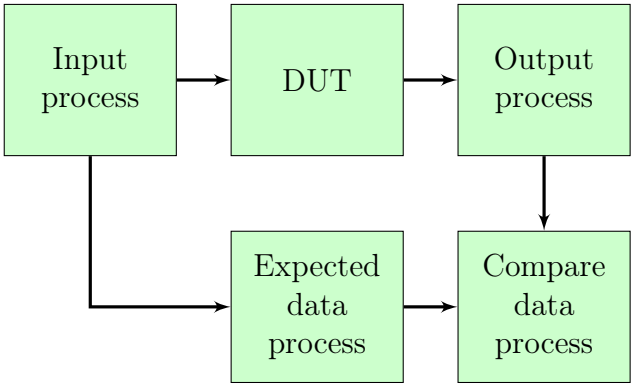

Рис. 3.7: Блок-схема самопроверяющегося тестбенча с автогенерацией данных.

Лабораторная работа. Моделирование генератора кода Грея с автопроверкой результата

Цель данной лабораторной работы - продемонстрировать использование конструкций языка VHDL для реализации самопроверяющейся тестбенчи на примере моделирования генератора следующего элемента последовательности Грея. Однако, сначала рассмотрим саму последовательность Грея, и алгоритм для ее реализации.

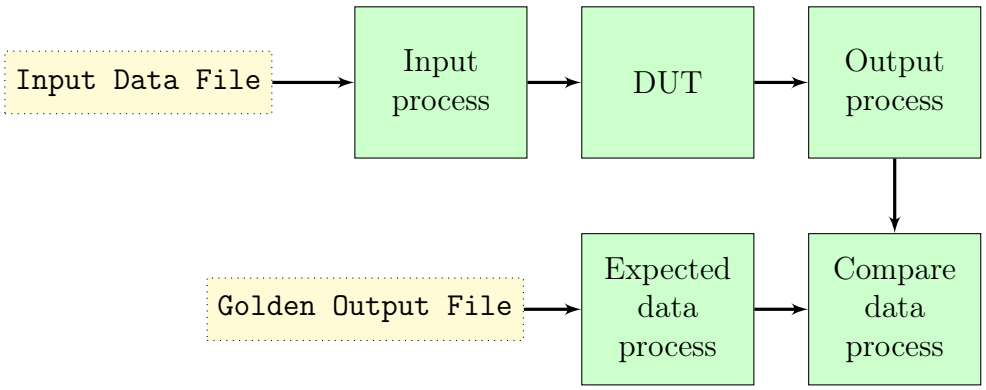

Рис. 3.8: Блок-схема самопроверяющегося тестбенча с чтением входных и опорных выходных данных из файла .

Последовательность или код Грея

Код Грея – система счисления, в которой два соседних значения различаются только в одном разряде. Изначально код Грея предназначался для защиты от ложного срабатывания электромеханических переключателей. Сегодня коды Грея широко используются для упрощения выявления и исправления ошибок в системах связи, а также в формировании сигналов обратной связи в системах управления. Коды Грея часто используются в датчиках-энкодерах (датчиках угла поворота). Также они используются для кодирования номера дорожек в жёстких дисках. Их использование удобно тем, что два соседних значения шкалы сигнала отличаются только в одном разряде. Если с помощью кода Грея перебирать адреса некоторого элемента памяти, то это свойство будет гарантировать отсутствие одновременного переключения нескольких физических линий параллельной шины адреса, что, в свою очередь, предотвращает возникновение шума (или ошибок) в шине адреса в следствии интерференции электромагнитных сигналов.

2-х битный код Грея	3-х битный код Грея
00	000
01	001
11	011
10	010
	110
	111
	101
	100

Таблица 3.1: Пример кода Грея

Рассмотрим алгоритм генерации последовательности Грея.

1. Преобразование кода Грея в соответствующий двоичный код.
2. Увеличение на единицу двоичного кода с использованием обычного сумматора.

3. Преобразование двоичного кода обратно в код Грея.

Преобразование двоичный код → код Грея

Обозначим: g – число в представлении Грея, b – число в двоичном представлении, нижним индексом будем обозначать номер бита в представлении. Посмотрим внимательно на таблицу соответствия двоичных кодов и кодов Грея (таблица **??**). Заметим, что i-й бит в представлении Грея (т.е. g_i) равен '1', если i-й и (i+1)-й биты в соответствующем двоичном представлении (т.е. b_i и b_{i+1}) различны, а старший бит всегда одинаков у обоих представлений. (Помним, что самый левый старший бит (most significant bit - MSB) имеет самый большой номер, а самый правый младший бит (least significant bit - LSB) имеет номер 0, т.е. нумерация ведется справа налево.)

Двоичный код	Код Грея
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Таблица 3.2: Соответствие двоичного кода и кода Грея

Это наблюдение запишем в виде логического уравнения с использованием оператора «Исключающее ИЛИ» или XOR:

$$g_i = b_i \oplus b_{i+1} \tag{3.1}$$

При этом для старшего бита следует вместо b_{i+1} подставить 0. Для 4-битного кода, таким образом, можно записать:

$$g_3 = b_3 \oplus 0 = b_3$$

$$g_2 = b_2 \oplus b_3$$

$$g_1 = b_1 \oplus b_2$$

$$g_0 = b_0 \oplus b_1$$

Преобразование код Грея → двоичный код

Заметим, что если $a \oplus b = c$, то $a = c \oplus b$. (Проверьте это с помощью таблицы истинности.) Тогда из уравнения **??** следует

$$b_i = g_i \oplus b_{i+1} \tag{3.2}$$

Уравнение **??** - это рекурсивная формула, раскроем ее для случая 4-х битных кодов.

$$b_3 = g_3 \oplus 0 = g_3$$
$$b_2 = g_2 \oplus b_3 = g_2 \oplus g_3$$
$$b_1 = g_1 \oplus b_2 = g_1 \oplus g_2 \oplus g_3$$
$$b_0 = g_0 \oplus b_1 = g_0 \oplus g_1 \oplus g_2 \oplus g_3$$

Реализация на VHDL

Следующий код реализует алгоритм получения следующего элемента последовательности Грея, имея на входе текущий элемент.

Листинг 3.12: Генератор следующего элемента последовательности Грея

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity g_inc is
   generic (
      word_w: natural := 4
   );
   port (
      g : in std_logic_vector (word_w-1 downto 0);
      g_next : out std_logic_vector (word_w-1 downto 0)
   );
end g_inc;

architecture compact_arch of g_inc is
   signal b, b_next: std_logic_vector(word_w-1 downto 0);
begin
   -- Gray to binary
   b <= g xor ('0' & b(word_w-1 downto 1));
   -- binary increment
   b_next <= b + 1;
   -- binary to Gray
   g_next <= b_next xor ('0' & b_next(word_w-1 downto 1));
end compact_arch;
```

Подробный анализ тестбенча

Рассмотрим подробно тестбенч для схемы генератора следующего элемента последовательности Грея (листинг **??**). Будут рассмотрены только новые конструкции.

В строчках 3-6 задаются библиотеки и пакеты для ввода/вывода данных в/из файла, вывода текстовых данных на консоль, а также для работы с типом данных *std_logic*. Строчки 30-33 задают тип данных *input_vector_record*, представляющий собой структуру из двух полей: *integer* и *std_logic_vector(3 downto 0)*. В строчке 35 обяъвлен тип данных *input_vector_array*, представляющий собой массив из 16 элементов типа *input_vector_record*. В строчке 37 объявлена константа типа *input_vector_array*, содержащая пронумерованные входные значения для схемы DUT. Данный код может служить примером для создания своих составных типов данных и массивов.

Строчка 58 задает еще один тип данных *output_vector_array* - массив из 16 элементов но на сей раз с элементами типа *std_logic_vector(3 downto 0)*. Строчки 60 и 61 объявляют два сигнала типа *output_vector_array*, в которые будут сохраняться ожидаемые данные и выходные данные схемы DUT для дальнейшего сравнения. В строчке 62 обявлен массив из 16 элементов типа *boolean*, в который будут заноситься результаты сравнения всех 16 выходных элементов. Наконец, сигнал *results_ready*, объявленный в строчке 63 является флагом окончания чтения всех выходных данных и начала процесса сравнения результатов. Этот сигнал инициализируется значением 0.

В строчках 68-71 модуль DUT подключается к сигналам тестбенча. Строчки 75 - 93 - это процесс генерации входных воздействий на схему. В данном случае на вход *g* схемы через каждые 10 ns подаются элементы константного массива *test_input_vectors*, а также эти элементы выводятся на консоль.

Строчки 97-111 задают выходной процесс, в котором выходные значения схемы DUT сохраняются в массиве *real_output_vectors*. Обратите внимание, что в данном случае выходной интерфейс схемы тривиален - только шина данных, значение которой считываются просто каждые 10 ns. В реальных схемах, конечно, интерфейсы передаи данных содержать помимо шины данных еще и шины управления передачей. Самые распространенные интерфейсы будут рассмотрены далее в курсе. После того, как все выходные значения DUT были сохранены в массиве *real_output_vectors* сигнал *results_ready* устанавливается в единицу.

Expected data process, заданный в строчках 114 - 130, просто читает данные из файла *gray_golden_output.txt* и сохраняет их в массиве *expected_output_vectors*.

Наконец, в процессе с меткой *compare_proc* (строчка 135) происходит сравнение данных и вывод результатов на консоль. Начинается процесс с ожидания установления сигнала *results_ready* в 1 (строчка 141), после чего идет сравнение массивов *real_output_vectors* и *expected_output_vectors*. На косоль выводятся результаты сравнения для каждого из 16 входных значений, а также глобальный статус моделирования.

Листинг 3.13: Самопроверяющийся тестбенч для моделирования схемы получе-

ния следующего элемента последовательности Грея

```
   library ieee;
1  use ieee.std_logic_1164.all;
2  use ieee.std_logic_textio.all;
3
4  library std;
5  use std.textio.all;
6
7  entity g_inc_tb is
8  end g_inc_tb;
9
10 architecture behavior of g_inc_tb is
11
12     -- Component Declaration for the Design Under Test (DUT)
13     component g_inc
14     port(
15          g       : in   std_logic_vector(3 downto 0);
16          g_next : out  std_logic_vector(3 downto 0)
17         );
18     end component;
19
20
21    --Inputs
22    signal g : std_logic_vector(3 downto 0) := (others =>
      '0');
23
24    --Outputs
25    signal g_next : std_logic_vector(3 downto 0);
26
27    constant clk_period : time := 10 ns;
28
29    type input_vector_record is record
30       test_num: integer; -- Test Number
31       input: std_logic_vector(3 downto 0); -- Test input
32    end record;
33
34    type input_vector_array is array (integer range 1 to 16)
      of input_vector_record;
35
36    constant test_input_vectors : input_vector_array := (
37       (  1, "0000"),
38       (  2, "0001"),
39       (  3, "0011"),
40       (  4, "0010"),
41       (  5, "0110"),
42       (  6, "0110"),
```

```
43       --(  6, "0111"),
44        (  7, "0101"),
45        (  8, "0100"),
46        (  9, "1100"),
47        ( 10, "1101"),
48        ( 11, "1111"),
49        ( 12, "1110"),
50        ( 13, "1010"),
51        ( 14, "1011"),
52        ( 15, "1001"),
53        ( 16, "1000"));
54
55    type bool_array is array (integer range <>) of boolean;
56
57    type output_vector_array is array (integer range 1 to 16)
       of std_logic_vector(3 downto 0);
58
59    signal expected_output_vectors : output_vector_array;
60    signal real_output_vectors : output_vector_array;
61    signal test_result : bool_array (1 to 16);
62    signal results_ready : std_logic := '0';
63
64 begin
65
66    -- Instantiate the Design Under Test (DUT)
67    dut: g_inc port map (
68           g => g,
69           g_next => g_next
70       );
71
72
73    -- Input process
74    stim_proc: process
75       variable Message : line;
76    begin
77       wait for 100 ns;
78
79       for i in 1 to 16 loop
80          Write ( Message, string'("-- Test "));
81          Write ( Message, test_input_vectors(i).test_num);
82          Write ( Message, string'(": INPUT = "));
83          Write ( Message, test_input_vectors(i).input);
84          writeline(output, Message);
85
86          g <= test_input_vectors(i).input;
87
```

```
88          wait for clk_period;
89       end loop;
90
91       wait;
92    end process;
93
94
95    -- Output process
96    catch_proc: process
97    begin
98
99       wait for 100 ns;
100
101      for i in 1 to 16 loop
102         wait for clk_period;
103         real_output_vectors(i) <= g_next;
104      end loop;
105
106      wait for clk_period;
107      results_ready <= '1';
108
109      wait;
110   end process;
111
112
113   -- Expected data process
114   expected_proc: process
115      file   fd    : text is in  "../../../../src/
      gray_golden_output.txt";
116      variable inline : line;
117      variable j : integer;
118      variable tmp_vector : std_logic_vector(3 downto 0);
119   begin
120      ---- read golden data
121      j := 1;
122      while (not endfile(fd)) loop
123         readline(fd, inline);
124         read(inline,tmp_vector);
125         expected_output_vectors(j) <= tmp_vector;
126         j := j + 1;
127      end loop;
128
129      wait;
130
131   end process;
132
```

```
133
134   -- Compare data process
135   copmare_proc: process
136      variable Message : line;
137      variable error : boolean := false;
138   begin
139
140      wait until results_ready = '1';
141
142      for i in 1 to 16 loop
143         test_result(i) <= (real_output_vectors(i) =
      expected_output_vectors(i));
144         if (not test_result(i)) then
145            error := true;
146         end if;
147      end loop;
148
149      wait for clk_period;
150
151      Write ( Message, LF);
152      Write ( Message, string'("-- DRP Cycle Tests Completed
      !")&LF);
153      Write ( Message, string'("-- Summary:")&LF);
154
155      for i in 1 to 16 loop
156         if (test_result(i)) then
157            Write ( Message, string'("-- Test "));
158            Write ( Message, i);
159            Write ( Message, string'(": PASS")&LF);
160         else
161            Write ( Message, string'("-- Test "));
162            Write ( Message, i);
163            Write ( Message, string'(": FAIL")&LF);
164         end if;
165      end loop;
166
167      writeline(output, Message);
168
169      if (error) then
170         Write ( Message, string'("Status Error"));
171      else
172         Write ( Message, string'("Status OK"));
173      end if;
174
175      writeline(output, Message);
176
```

```
177         wait;
178     end process;
179 end;
```

Контрольные вопросы

1. Все процессы в тестбенче **??** содержат оператор wait в конце. Что будет, если удалить этот оператор?
2. Что произойдет, если в строчке 78 изменить ожидание с 100 до 50 ns?

Выполнение лабораторной работы

1. Перейти в директорию *03-vhdl_modeling/lab3-1*
2. Выполнить команду

   ```
   vivado -mode batch -source create_project.tcl -notrace
   ```

   Будет создан Vivado проект в директории *03-vhdl_modeling/lab3-1/g_inc*, с уже добавленными исходными файлами из директории *03-vhdl_modeling/lab3-1/src*.
3. Открыть созданный Vivado проект.
4. Запустить симуляцию, налюбдать вывод в консоль и вейформы сигналов.
5. Симуляция завершилась с ошибкой. Проанализировать ошибку. Найти в тестбенче ошибку и исправить ее.

Глава 4

Описание комбинационных устройств на VHDL

Последовательные и параллельные операторы. Комбинационные логические элементы и их описание на языке VHDL. Операторы выборочного и условного назначения сигналов. Оператор процесса. Запуск процесса и список чувствительности. Операторы IF и CASE. Правила кодирования комбинационных логических устройств на VHDL.

Параллельное назначение сигналов в VHDL

В общем случае цифровые устройства можно разделить на комбинационные и последовательные. Комбинационная цепь не имеет памяти, и значения на ее выходах в каждый момент времени зависят только от значений на ее входах. Последовательная цепь, напротив, имеет память (или внутреннее состояние), и значения на ее выходах зависят как от входных сигналов, так и от внутреннего состояния, т.е. от предыстории функционирования цепи. Операторы параллельного назначения сигналов описывают комбинационные цепи.

Простое назначение сигналов

```
  status       <= '1';
1 even         <= (p1 and p2) or (p3 and p4);
2 arith_result <= a + b + c - 1;
```

Последнее выражение может быть изображено в виде следующей схемы:

Упражнение. Предположим, что задержки сумматора и блока, вычисляющего разность входных сигналов, одинаковы и равны δ, предположим, что других задержек нет (например, мы не учитываем задержки в проводниках хотя они порой очень существенны). Тогда изменение входного сигнала a отразится на выходном сигнале arith_result только через 3δ, ведь сигнал должен

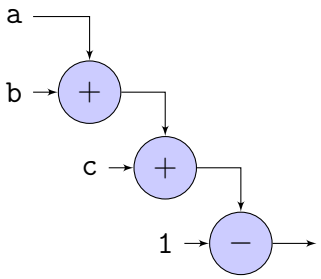

Рис. 4.1: Неоптимизированная комбинационная схема.

пройти через три блока, каждый из которых вносит задержку δ. С другой стороны изменение входного сигнала c отразится на выходном сигнале arith_result только через 2δ. Т.е. схема еще и несимметричная. Можно ли оптимизировать схему, чтобы временная задержка стала меньше, и схема стала симметричной? (Ответ приведен далее, но все же подумайте, попробуйте переставить блоки.)

Ответ. Воспользуемся коммутативностью и ассоциативностью сложения:

$$a + b + c - 1 = (a + b) + (c - 1).$$

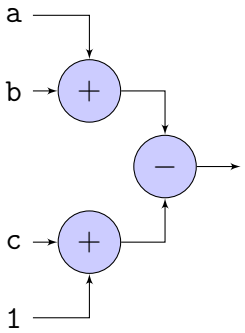

Рис. 4.2: Оптимизированная комбинационная схема.

При создании комбинационных схем **нельзя** назначать сигналы с обратной связью. С синтаксической точки зрения, следующие выражения корректны, они даже могут быть использованы в определенном контексте. Но их нельзя использовать непосредственно в архитектурном теле.

```
  architecture wrong_arch of wrong_entity is
1 begin
2     -- внутреннее состояние:
3     q <= (q and (not  en)) or (d and en);
4     -- осцилляции q при en = '0':
5     q <= (not q and (not en)) or (d and en);
6 end wrong_arch
```

С другой стороны, следующий код корректен.

```
architecture right_arch of right_entity is
1     signal q : std_logic_vector(3 downto 0);
2 begin
3     q <= ('0' & q(3 downto 1)) and d;
4 end right_arch;
```

Этот код – более компактная векторная запись следующего кода.

```
architecture right_arch of right_entity is
1     signal q : std_logic_vector(3 downto 0);
2 begin
3     q(3) <= '0'  and d(3);
4     q(2) <= q(3) and d(2);
5     q(1) <= q(2) and d(1);
6     q(0) <= q(1) and d(0);
7     ...
8 end right_arch;
```

Оператор условного назначения сигнала WHEN

Это параллельный оператор, реализующий выбор одного из возможных вариантов в зависимости от некоторых управляющих сигналов. В общем виде его можно записать следующим образом:

```
 сигнал <= выражение_1       when условие_1       else
1          выражение_2       when условие_2       else
2          ...
3          выражение_(n-1) when условие_(n-1) else
4          выражение_n;
```

Пример 1. Восьмибитовый мультиплексор 4-в-1. Рассмотрим комбинационную схему, описываемую таблицей истинности:

input	output
s	x
00	a
01	b
10	c
11	d

VHDL код, реализующий эту схему такой:

```
   library ieee;
1  use ieee.std_logic_1164.all;
2
3  entity mux4 is
4      port(
5          a, b, c, d : in std_logic_vector(7 downto 0);
6          s : in  std_logic_vector(1 downto 0);
7          x : out std_logic_vector(7 downto 0)
8      );
9  end mux4;
10
11 architecture cond_arch of mux4 is
12 begin
13     x <= a when (s = "00") else
14          b when (s = "01") else
15          c when (s = "10") else
16          d;
17 end cond_arch;
```

Сигнал x – это выходной информационный сигнал; a, b, c, d – входные информационные сигналы, a s – это сигнал управления, на основании которого осуществляется выбор одного из вариантов. Рассмотрим еще пару примеров.

Пример 2. Бинарный дешифратор.

input	output
s	x
00	0001
01	0010
10	0100
11	1000

```
   library ieee;
1  use ieee.std_logic_1164.all;
2
3  entity decoder4 is
4      port(
5          s : in  std_logic_vector(1 downto 0);
6          x : out std_logic_vector(3 downto 0)
7      );
8  end decoder4;
9
10 architecture cond_arch of decoder4 is
11 begin
12     x <= "0001" when (s = "00") else
13          "0010" when (s = "01") else
14          "0100" when (s = "10") else
15          "1000";
16 end cond_arch;
```

Пример 3. Шифратор приоритета.

input	output	
r	code	active
1---	11	1
01--	10	1
001-	01	1
0001	00	1
0000	00	0

```
   library ieee;
1  use ieee.std_logic_1164.all;
2
3  entity prio_encoder42 is
4      port(
5          r      : in  std_logic_vector(3 downto 0);
6          code   : out std_logic_vector(1 downto 0);
7          active : out std_logic
8      );
9  end prio_encoder42;
10
11 architecture cond_arch of prio_encoder42 is
12 begin
13     code <= "11" when (r(3) = '1') else
14             "10" when (r(2) = '1') else
15             "01" when (r(1) = '1') else
16             "00";
17     active <= r(3) or r(2) or r(1) or r(0);
18 end cond_arch;
```

Здесь выходной сигнал code определяет, на какой из линий r(i) присутствует логическая единица. Причем линия r(3) имеет **приоритет** – если на ней будет '1', то code = "11" вне зависимости от других линий.

Поговорим о том, как реализуется оператор условного назначения сигнала в железе. Семантика оператора условного назначения сигнала подразумевает создание приоритетной схемы, т.е. условия, стоящие выше имеют больший приоритет. При синтезе этого оператора реализуются три схемы:

- схема, вычисляющая возможные значения выходного сигнала,
- схема, вычисляющая условия,
- схема определения приоритета.

Чтобы лучше разобраться, что такое схема приоритета введем понятие абстрактного мультиплексора 2-в-1.

Здесь `i1`, `i2`, `o` – n-битные шины, а сигнал `sel` однобитный. В зависимости от него на выход `o` идет либо `i1`, либо `i0`.

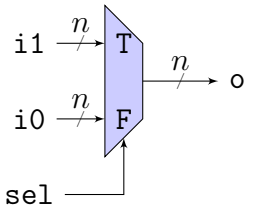

Рис. 4.3: Символ абстрактного мультиплексора 2-в-1.

Когда мы пишем на VHDL выражение:

```
signal_name <= value_expr_1 when boolean_expr_1 else
1              value_expr_2;
```

реализуется следующая схема с мультиплексором:

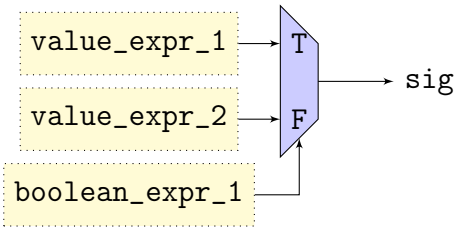

Рис. 4.4: Схема с мультиплексором 2-в-1.

Каждый следующий уровень ветвления оператора условного назначения сигнала добавляет еще одну ступень в иерархию вместе с еще одним мультиплексором.

```
signal_name <= value_expr_1 when boolean_expr_1 else
1              value_expr_2 when boolean_expr_2 else
2              value_expr_3;
```

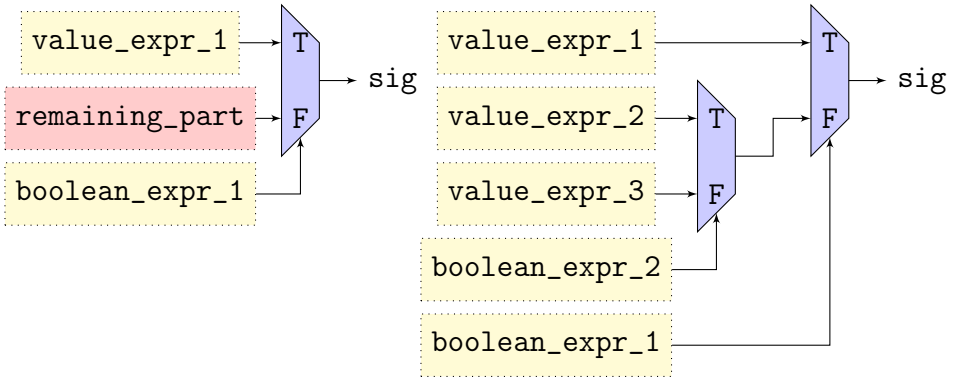

Рис. 4.5: Схема с мультиплексорами 2-в-1.

Оператор выборочного назначения сигнала WITH ... SELECT

Это параллельный оператор, также реализующий выбор одного из возможных вариантов в зависимости от некоторых управляющих сигналов. Во многом он похож на оператор условного назначения, но есть и отличия. Записать в общем виде его можно так:

```
with задающее_выражение select
    сигнал <=
        выражение_1 when выбор_1,
        выражение_2 when выбор_2,
        выражение_3 when выбор_3,
        ...
        выражение_n when выбор_n;
```

Обратите внимание на использование *запятых* после каждого выборочного значения (выбор_i) и на точку с запятой в конце.

Выборочные значения (выбор_i) должны быть **взаимно исключающими** (значения не должны перекрываться или повторяться) и **полными** (все значения должны быть перечислены). Для обеспечения полноты выборочных значений рекомендуется использовать **others**. Заметим, что выборочные значения – это именно значения, а не условия.

Примеры из предыдущего параграфа можно переписать с использование оператора выборочного назначения сигнала.

Пример 1. Мультиплексор 4-в-1.

```
architecture sel_arch of mux4 is
begin
    with s select
        x <= a when "00",
             b when "01",
             c when "10",
             d when others;
end sel_arch;
```

Пример 2. Бинарный дешифратор.

```
architecture sel_arch of decoder4 is
begin
    with sel select
        x <= "0001" when "00",
             "0010" when "01",
             "0100" when "10",
             "1000" when others;
end sel_arch;
```

Пример 3. Шифратор приоритета.

```
   architecture sel_arch of prio_encoder42 is
1  begin
2      with r select
3          code <= "11" when "1000" | "1001" | "1010" |
4                            "1011" | "1100" | "1101" |
5                            "1110" | "1111",
6                  "10" when "0100" | "0101" | "0110" |
7                            "0111",
8                  "01" when "0010" | "0011",
9                  "00" when others;
10         active <= r(3) or r(2) or r(1) or r(0);
11 end sel_arch;
```

В последнем примере вертикальной чертой "|" идет компактное перечисление вариантов, которым соответствует один выходной код.

При синтезе оператора выборочного назначения сигнала реализуется равноправный абстрактный мультиплексор k-в-1.

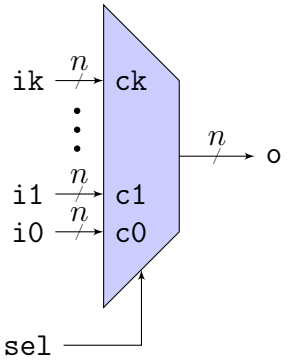

Рис. 4.6: Абстрактный мультиплексор k-в-1.

В данной схеме **нет приоритета** при продвижении сигнала с входа на выход схемы. Также можно сказать, что схема получается симметричная с одинаковыми задержками на всех входных линиях.

Проще всего сравнить операторы выборочного и условного назначения сигналов на примере мультиплексора 4-в-1.

При использовании оператора условного назначения будет реализована следующая схема:

Видно, что схема несимметричная, с различными задержками по входным сигналам. Но при этом сигнал `A` имеет приоритет над остальными.

При использовании оператора выборочного назначения сигнала реализуется следующая схема:

Здесь схема симметричная, задержки по входным сигналам приблизительно одинаковые. Все сигналы имеют одинаковый приоритет.

У обеих схем *логика работы* и *таблицы истинности* **одинаковые**, при этом схемные реализации разные. При работе с языком VHDL вы всегда должны пытаться мыслить на уровне логических элементов и схем, чтобы получать эффек-

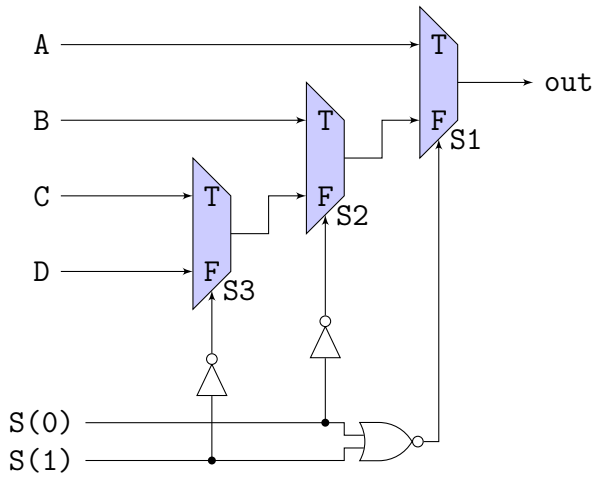

Рис. 4.7: Оператор условного назначения.

тивные реализации, ведь по сути VHDL – это не более чем удобный инструмент для описания таких объектов.

Последовательное назначение сигналов в VHDL

В языке VHDL есть средство для описания последовательных алгоритмов с помощью, так называемых, *последовательных операторов*. Эти операторы должны располагаться внутри специальной конструкции VHDL, называемой **оператор процесса**. Главной целью введения последовательных операторов в язык VHDL было описание абстрактного *поведения* цифровых цепей (на более высоком уровне абстракции – *поведенческом* (*behavioural*)). В отличие от операторов параллельного назначения сигналов, последовательные операторы не всегда имеют однозначное соответствие в железе. Часто даже они не могут быть синтезированы вовсе. Поэтому для описания схем с помощью процессов и последовательных операторов необходимо придерживаться четких шаблонов, которые однозначно синтезируются. Рассмотрением этих шаблонов мы и займемся. Пусть вас не пугает слово «шаблон»: несмотря на то, что определенные компоненты должны описываться по четкой схеме, у вас останется огромное пространство для творчества при определении взаимодействия этих компонентов.

Оператор процесса PROCESS

Оператор процесса имеет следующий синтаксис:

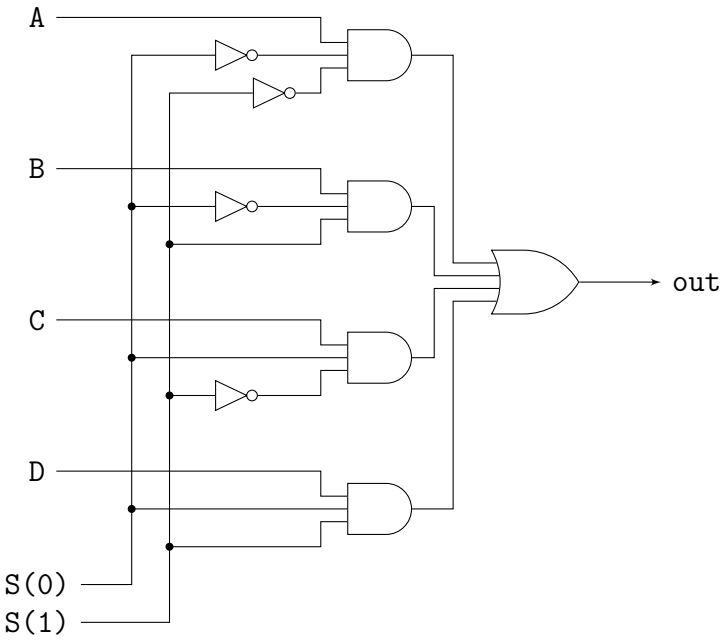

Рис. 4.8: Оператор выборочного назначения.

```
    process ( список_чувствительности )
1       объявления;
2   begin
3       последовательные_операторы;
4       последовательные_операторы;
5       ...
6   end process;
```

В подразделе объявлений оператора процесса могут быть объявлены подпрограммы, типы, подтипы, константы, переменные (только в подразделе объявлений оператора процесса), атрибуты, файлы и др. Сигналы в подразделе объявлений оператора процесса объявлены быть **не могут**.

Сам оператор процесса является **параллельным оператором**. Если в архитектурном теле разместить несколько процессов, то они будут выполняться параллельно. Но внутри каждого процесса находятся операторы, выполняемые последовательно, т.е. один за другим. Список чувствительности – это список запускающих процесс сигналов. Когда один из сигналов, указанных в списке чувствительности, меняет свое значение, процесс запускается, и последовательные операторы выполняются до конца процесса. После этого процесс приостанавливает свою работу до тех пор, пока опять какой-нибудь сигнал из списка чувствительности не изменит свое значение. Таким образом, процесс может находиться в одном из двух состояний: запущенном и приостановленном. **Для комбинационной схемы *все входы* должны быть в списке чувствительности.**

```
   signal a, b, c, y: std_logic;
1  ...
2  process (a, b, c) -- правильно
3  begin
4      y <= a and b or c;
5  end process;
6  ...
7  process (a)       -- неправильно
8  begin
9      y <= a and b or c;
10 end process;
```

Внутри процесса сигналу можно присвоить значение несколько раз. При этом значение сигнала не изменится до выхода из процесса, а после выхода станет равным последнему присвоенному значению. Получается, что следующие записи просто эквивалентны.

```
   process (a, b, c, d)
1  begin
2      y <= a or c;
3      y <= a and b;
4      y <= c and d;
5  end process;
```

```
   process (a, b, c, d)
1  begin
2      y <= c and d;
3  end process;
```

Очень важно понимать, что последнее значение присваивается сигналу не сразу, а в конце процесса. Промежуточные значения сигнала не доступны. В этом отличие сигнала от *переменной*.

```
architecture arch of main is
    signal y : std_logic;
begin
    process (a, b, c, d)
    begin
        y <= a or c;
        -- Здесь, после оператора присвоения, сигнал y
        -- не равен a or c.
        -- Здесь могут идти последовательные операторы,
        -- которые не используют промежуточное значение
        -- сигнала y.
        z <= y;
        -- Здесь сигналу z не присваивается значение
        -- a or c, ему присваивается значение c and d.
        y <= a and b;
        y <= c and d;
    end process;
    -- Зато здесь, вне тела процесса, y всегда равен c and d
end arch;
```

Замечание. Синтезатор XST достаточно умный, чтобы не реализовать в этом конкретном простом примере схему с памятью (ведь сигнала y нет в списке чувствительности процесса), но на синтезатор рекомендуется полагаться не всегда.

На самом деле, с оператором процесса вы уже сталкивались.

Вопрос. Где?

Ответ. Правильно, при описании testbench. Там был процесс stim_proc, задающий значения входных сигналов для тестируемой схемы.

Вопрос Чего не было в процессе stim_proc?

Ответ. Там не было списка чувствительности. Но зато там были операторы **wait**.

Оператор ожидания WAIT

Процесс с операторами ожидания имеет один или более операторов wait, но не имеет списка чувствительности. Оператор **wait** имеет в общем случае следующий вид:

```
wait on список_чувствительности until условие for тайм_аут;
```

Оператор **wait** является причиной временного прекращения оператора процесса или процедуры. Оператор ожидания **wait** приостанавливает процесс до момента, пока не изменится некоторый сигнал в списке чувствительности оператора **wait**, в этот момент будет произведено вычисление условия. Фраза «условие» -– есть выражение типа boolean. Фраза «тайм-аут» – есть выражение типа

time, оно устанавливает максимальное время ожидания, после которого процесс возобновит свое выполнение.

Пример.

```
wait on a, b until (c = 0) for 50 ns;
```

Этот оператор приостановит процесс до момента изменения сигнала a или сигнала b, после чего будет проверено выражение c = 0, и если результатом проверки будет истина, процесс возобновится. Но независимо от этих условий возобновление процесса произойдет через 50 ns.

Допустимо записывать одно или более условий в операторе ожидания **wait**, например:

```
wait on a, b;          -- Процесс возобновится, когда изменится a
   или b.
1
2  wait until (c = 0); -- Процесс возобновится, когда c изменит свое
3                      -- значение из 1 в 0.
4  wait for 50 ns;     -- Процесс возобновится через 50 нс.
5
6  wait;               -- Процесс больше не возобновится, если не
7                      -- поместить такой оператор в конце процесса,
8                      -- то выполнение процесса начнется сначала.
9                      -- Такой оператор можно использовать в конце
10                     -- testbench для окончания симуляции.
```

В процессе допускается использование нескольких операторов **wait**.

ВАЖНО! Только несколько четко определенных конструкций с операторами ожидания **wait** могут быть синтезированы в железе. В документации к синтезатору XST допускается применение оператора **wait** в последовательных схемах (не комбинационных, о них вы узнаете в следующей лекции), при этом допускается только конструкция **wait until**, причем она должна быть единственной в процессе, и у процесса не должно быть списка чувствительности.

Настоятельно рекомендуется использовать конструкции с оператором ожидания только для *СИМУЛЯЦИИ*.

Переменные в VHDL

Наряду с сигналами и константами в языке VHDL есть еще один программный элемент данных – переменная. Локальные переменные могут быть объявлены только в подразделе объявлений оператора процесса **process** и подпрограмм (функций и процедур). Объявление переменной имеет следующий синтаксис:

```
variable имя_переменной : тип [ := начальное_значение ];
```

Выражение в квадратных скобках может быть опущено.

В каждый момент времени переменная может иметь любое, но всегда **единственное** значение. В другой момент времени переменная может иметь другое значение. Значение переменной может быть модифицировано оператором присваивания значения переменной. Этот оператор имеет следующий синтаксис:

```
имя_переменной := выражение;
```

При этом считается, что значение переменной модифицируется оператором присваивания значения **немедленно** (т.е. без временной задержки), как только этот оператор выполнен. Поведение оператора присвоения значения переменной в VHDL похоже на поведение оператора присвоения значения переменной в обычном традиционном языке программирования (C, C++, Pascal и др.). Этим отличаются присваивания значения сигналам и переменным.

Пример.

```
   signal a, b, y : std_logic;
1  ...
2  process (a, b)
3      variable tmp : std_logic;
4  begin
5      tmp := '0';
6      tmp := tmp or a;
7      tmp := tmp or b;
8      y <= tmp;
9  end process;
```

Переменная tmp получает значение немедленно в каждом последовательном выражении, содержащем оператор присвоения значения переменной (:=). И затем значение переменной (равное '0' **or** a **or** b $\Leftrightarrow$ a **or** b) присваивается сигналу y.

В железе такой код может быть реализован следующим образом.

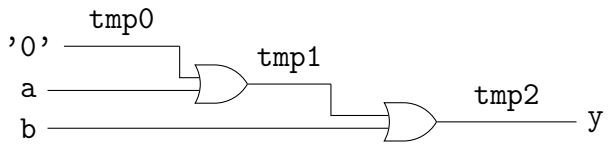

Рис. 4.9: Реализация в железе кода с переменными.

Для реализации предыдущего процесса в железе нам потребовалось ввести новые переменные tmp0, tmp1, и tmp2.

Для сравнения повторим предыдущий фрагмент кода, заменив переменные сигналами.

```
  signal a, b, y, tmp : std_logic;
1 ...
2 process (a, b, tmp)
3 begin
4     tmp <= '0';
5     tmp <= tmp or a;
6     tmp <= tmp or b;
7     y   <= tmp;
8 end process;
```

Заметим, что сигналы должны быть «глобальными» и должны быть объявлены вне процесса, и сигнал tmp должен быть включен в список чувствительности процесса. Этот фрагмент кода эквивалентен следующему.

```
  signal a, b, y, tmp : std_logic;
1 ...
2 process (a, b, tmp)
3 begin
4     tmp <= tmp or b;
5     y <= tmp;
6 end process;
```

Здесь мы имеем дело с комбинационной петлей и элементом ИЛИ, как показано на следующем рисунке.

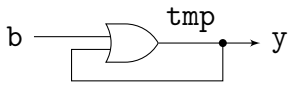

Рис. 4.10: Комбинационная петля.

Заметим, что переменные локальны для процесса, в котором они объявлены, и они не видны вне этого процесса.

Прежде, чем использовать переменную, необходимо задать себе вопрос, действительно ли она нужна. Неправильное использование переменных очень непросто найти и исправить. В большинстве случаев аналогичный код можно написать без использования переменных.

Оператор IF

Оператор **if** – это один из последовательных операторов, поэтому его можно использовать только внутри оператора процесса. Синтаксис оператора **if** следующий:

```
if условие_1 then
    последовательные_операторы;
elsif условие_2 then
    последовательные_операторы;
elsif условие_3 then
    последовательные_операторы;
...
else
    последовательные_операторы;
end if;
```

Обратите внимание на написание ключевого слова **elsif** и на то, что **end if** пишется раздельно.

Рассмотрим его применение на примерах из предыдущего раздела.

Пример 1. Мультиплексор 4-в-1.

```
architecture if_arch of mux4 is
begin
    process (a, b, c, d, s)
    begin
        if (s = "00") then
            x <= a;
        elsif (s = "01") then
            x <= b;
        elsif (s = "10") then
            x <= c;
        else
            x <= d;
        end if;
    end process;
end if_arch;
```

Пример 2. Бинарный дешифратор.

```
architecture if_arch of decoder4 is
begin
    process (s)
    begin
        if (s = "00") then
            x <= "0001";
        elsif (s = "01") then
            x <= "0010";
        elsif (s = "10") then
            x <= "0100";
        else
            x <= "1000";
        end if;
    end process;
end if_arch;
```

Пример 3. Шифратор приоритета.

```
architecture if_arch of prio_encoder42 is
begin
    process (r)
    begin
        if (r(3) = '1') then
            code <= "11";
        elsif (r(2) = '1') then
            code <= "10";
        elsif (r(1) = '1') then
            code <= "01";
        else
            code <= "00";
        end if;
    end process;
    active <= r(3) or r(2) or r(1) or r(0);
end if_arch;
```

Сравним оператор **if** с оператором условного назначения сигналов. Для простых случаев эти операторы просто эквивалентны.

```
сигнал <= выражение_1 when условие_1 else
          выражение_2 when условие_2 else
          ...
          выражение_n;
```

эквивалентно:

```
process(...)
begin
    if условие_1 then
        сигнал <= выражение_1;
    elsif условие_2 then
        сигнал <= выражение_2;
    ...
    else
        сигнал <= выражение_n;
    end if;
end process;
```

Отличия:

- По структуре: внутри **if** может быть еще один **if**.
- По пониманию: легче понять смысл.
- В одной ветке можно назначить несколько сигналов.

Проблема неполного дерева условий оператора IF

В VHDL только одна ветка **then** обязательна. Остальные ветки (**elsif**, **else**) могут быть опущены. Здесь надо быть аккуратными. Например, рассмотрим компаратор двух чисел, выход которого равен '1', когда они равны.

```
process(a, b)
begin
    if (a = b) then
        eq <= '1';
    end if;
end process;
```

Этот код синтаксически верен, но приводит к неверной реализации. Так как здесь нет ветки **else**, то, когда числа не равны, не производится никакого действия. Семантика VHDL такова, что в этом случае значение сигнала eq не обновляется, и он сохраняет свое значение. Это эквивалентно следующей записи:

```
process(a, b)
begin
    if (a = b) then
        eq <= '1';
    else
        eq <= eq;
    end if;
end process;
```

Этот код описывает схему с обратной связью, что приводит к образованию элемента памяти или внутреннему состоянию (обычно так называемых *защелок* (*latch*)). Очевидно, что мы не это имели в виду. Код необходимо переписать следующим образом:

```
  process(a, b)
1 begin
2     if (a = b) then
3         eq <= '1';
4     else
5         eq <= '0';
6     end if;
7 end process;
```

Для описания комбинационной цепи необходимо всегда включать ветку **else**, чтобы избежать образования элемента памяти.

Проблема неполного назначения сигналов в операторе IF

В общем случае оператор **if** имеет несколько веток. Возможно, что сигналу присваивается значение не во всех этих ветках. Хотя синтаксически правильно, это приводит к образованию элемента памяти в схеме (защелок). Пример компаратора двух чисел с тремя выходами: gt (greater than, $a > b$), lt (less than, $a < b$), eq (equal, $a = b$).

```
  process(a, b)
1 begin
2     if (a > b) then
3         gt <= '1';
4     elsif (a = b) then
5         eq <= '1';
6     else
7         lt <= '1';
8     end if;
9 end process;
```

Опять же семантика VHDL подразумевает, что если сигналу не назначается значение, он его сохраняет. А это приводит к ненужному здесь образованию элемента памяти. Код следует исправить:

```
   process(a, b)
1  begin
2      gt <= '0';
3      eq <= '0';
4      lt <= '0';
5      if (a > b) then
6          gt <= '1';
7      elsif (a = b) then
8          eq <= '1';
9      else
10         lt <= '1';
11     end if;
12 end process;
```

или:

```
   process(a, b)
1  begin
2      if (a > b) then
3          gt <= '1';
4          eq <= '0';
5          lt <= '0';
6      elsif (a = b) then
7          gt <= '0';
8          eq <= '1';
9          lt <= '0';
10     else
11         gt <= '0';
12         eq <= '0';
13         lt <= '1';
14     end if;
15 end process;
```

Первый вариант кода более красив и реализует прием программирования на VHDL, называемый присвоение значения комбинационному сигналу *по умолчанию*. Здесь надо вспомнить, что при назначении сигнала в процессе, сигнал принимает последнее присвоенное значение. Рекомендуется использовать именно такой стиль программирования.

Оператор CASE

Оператор **case** также является последовательным оператором, и его использование разрешено только внутри оператора процесса. Оператор **case** имеет следующий синтаксис:

```
   case задающее_выражение is
1  when выбор_1 =>
2          последовательные_операторы;
3  when выбор_2 =>
4          последовательные_операторы;
5          ...
6  when выбор_n =>
7          последовательные_операторы;
8  end case;
```

Выборочные значения выбор_i должны быть полными и взаимоисключающими – как в операторе выборочного назначения сигнала (**select**). Для обеспечения полноты выборочных значений рекомендуется использовать **others**. Для компактного перечисления вариантов, которым соответствует один выходной код, можно использовать вертикальную черту "|". Рассмотрим на примерах.

Пример 1. Мультиплексор 4-в-1.

```
   architecture case_arch of mux4 is
1  begin
2      process (a, b, c, d, s)
3      begin
4          case s is
5              when "00" => x <= a;
6              when "01" => x <= b;
7              when "10" => x <= c;
8              when others => x <= d;
9          end case;
10     end process;
11 end case_arch;
```

Пример 2. Бинарный дешифратор.

```
architecture case_arch of decoder4 is
begin
proc1: process (s)
    begin
        case s is
            when "00" =>
                x <= "0001";
            when "01" =>
                x <= "0010";
            when "10" =>
                x <= "0100";
            when others =>
                x <= "1000";
        end case;
    end process proc1;
end case_arch;
```

Здесь proc1 – так называемая метка процесса. Язык VHDL допускает процессы без меток.

Пример 3. Шифратор приоритета.

```
architecture case_arch of prio_encoder42 is
begin
process (r)
    begin
        case r is
            when "1000" | "1001" | "1010" | "1011" |
                 "1100" | "1101" | "1110" | "1111" =>
                code <= "11";
            when "0100" | "0101" | "0110" | "0111" =>
                code <= "10";
            when "0010" | "0011" =>
                code <= "01";
            when others =>
                code <= "00";
        end case;
    end process;
    active <= r(3) or r(2) or r(1) or r(0);
end case_arch;
```

В простых случаях оператор **case** эквивалентен оператору выборочного назначения сигнала (**select**):

```
with задающее_выражение select
    сигнал <=
        выражение_1 when выбор_1,
        выражение_2 when выбор_2,
        ...
        выражение_n when выбор_n;
```

эквивалентно:

```
process(...)
begin
    case задающее_выражение is
        when выбор_1 =>
            сигнал <= выражение_1;
        when выбор_2 =>
            сигнал <= выражение_2;
            ...
        when выбор_n =>
            сигнал <= выражение_n;
    end case;
end process;
```

Проблема неполного назначения сигналов в операторе CASE

Рассмотрим простой шифратор приоритета.

```
process(a)
begin
    case a is
        when "100" | "101" | "110" | "111" =>
            high <= '1';
        when "010" | "011" =>
            middle <= '1';
        when others =>
            low <= '1';
    end case;
end process;
```

Подразумевается, что в один момент времени только один из сигналов high, middle и low будет равен '1'. Однако при реализации схемы опять возникнет ненужный элемент памяти, так как не все сигналы назначаются за один «пробег» процесса. Решается эта проблема опять же присвоением значений сигналам по умолчанию.

Упражнение. Попробуйте написать соответствующий код самостоятельно и сравните с ответом.

Ответ.

```
    process(a)
1   begin
2       high   <= '0';
3       middle <= '0';
4       low    <= '0';
5       case a is
6           when "100" | "101" | "110" | "111" =>
7               high <= '1';
8           when "010" | "011" =>
9               middle <= '1';
10          when others =>
11              low <= '1';
12      end case;
13  end process;
```

Правила описания комбинационных схем

Перечислим правила, которых вы **должны** придерживаться при описании комбинационных схем с использованием оператора процесса и последовательных операторов:

- Все входные сигналы должны быть в списке чувствительности процесса.
- Дерево условий оператора **if** должно быть **полным**. Используйте **else**.
- Выборочные значения в операторе **case** должны быть **полными и взаимоисключающими**. Используйте **others**.
- При выходе из процесса всем выходным сигналам необходимо присвоить значение (например, значение по умолчанию в начале процесса).

Практика

Коммутатор-переключатель 2-на-2

У коммутатора 2-на-2 есть два входа: x0 и x1 – и два выхода: y0 и y1, а также вход управления ctrl. В зависимости от значения сигнала ctrl входные данные по-разному направляются на выходы.

Упражнение. Реализуйте схему переключателя на VHDL с использованием параллельных и последовательных операторов.

input	output		function
ctrl	y1	y0	
00	x1	x0	pass
01	x0	x1	cross
10	x0	x0	broadcast x0
11	x1	x1	broadcast x1

Конвертер-генератор кода Грея.

Упражнение. Реализовать конвертер-генератор кода Грея. Входные сигналы: x, s. Сигнал x имеет тип std_logic_vector(WIDTH−1 **downto** 0), а сигнал s имеет тип std_logic_vector(1 **downto** 0).

input	output
s	y
00	y – двоичное представление x, x записан в коде Грея
01	y – представление x в коде Грея, x записан в двоичном коде
10	y – следующий элемент последовательности Грея, x – текущий элемент
11	y – предыдущий элемент последовательности Грея, x – текущий элемент

Использовать оператор процесса и последовательные операторы.

Контрольные вопросы и задания

Теория

1. Операторы условного и выборочного назначения сигналов в VHDL.
2. Оператор процесса и последовательные операторы (IF, CASE).
3. Оператор WAIT.
4. Правила описания комбинационных схем на VHDL.

Задания

1. Разработайте комбинационный АЛУ, у которого на входе 2 шины операндов a и b (разрядность каждой – 8 бит) и шина выбора действий (разрядность – 2 бита). Выходная сигнал q содержит 8 бит и равен сумме a и b при s = "00", разности – при s = "01", логическому ИЛИ – при s = "10", логическому И – при s = "11".
   (a) Используйте параллельные операторы.
   (b) Используйте последовательные операторы.
2. Определим расстояние между элементами последовательности Грея. Пусть a и b – элементы последовательности Грея, тогда расстояние между

a и b определим, как минимальное количество последовательных переходов от элемента a до элемента b по последовательности Грея. Например, пусть a = "0101", b = "1111". Тогда расстояние от a до b равно 4, поскольку требуется 4 перехода (см. таблицу с кодом Грея): "0101" → "0100" → "1100" → "1101" → "1111". Разработайте схему, вычисляющую расстояние между двумя 4-битными словами a и b, записанными в коде Грея.

3. **Задания на дополнительный код.** *Дополнительный код* (англ. *two's complement*) – наиболее распространённый способ представления отрицательных целых чисел в компьютерах. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ. Дополнительный код отрицательного числа можно получить инвертированием модуля двоичного числа и прибавлением к результату инверсии единицы.

   (a) Разработайте комбинационный конвертер целых чисел со знаком. Схема имеет 2 входа: целое число со знаком a (разрядность – 8 бит), сигнал выбора направления преобразования s (разрядность – 1 бит). Если s = "0", то преобразуется целое число со знаком, записанное в дополнительном коде, в целое число со знаком, записанное в прямом коде, а если s = "1", то наоборот. Схема имеет единственный выход r (разрядность – 8 бит).

   (b) Используйте параллельные операторы.

   (c) Используйте последовательные операторы.

Лабораторная работа. Комбинационный АЛУ операндов в дополнительном коде

Тут должно быть готовое решение

Самостоятельная работа. Вычисление контрольной суммы сетевого IP пакета

Тут должно быть описание IP пакета, алгоритм вычисления контрольной суммы, готовый тестбенч

Глава 5

Последовательные логические элементы

Понятие триггера. Статическая и динамическая синхронизация. Сложные последовательные элементы: регистры, счетчики, сдвиговые регистры. Описание последовательных элементов на VHDL. Принцип построения синхронных схем.

Введение

Начинаем рассмотрение последовательных логических элементов: триггеров, регистров, счетчиков и т.д. Все эти элементы характерны тем, что имеют *состояния* и способны переходить из одного состояния в другое под действием внешних сигналов. Мы будем работать с элементами, управляемыми *тактовым сигналом*, причем переходить из одного состояния в другое (или по-другому переключаться) они могут только в узкой окрестности изменения такого сигнала, называемой *фронтом* сигнала. Различают передний и задние фронты сигнала. Существуют последовательные элементы, переключающиеся по одному фронту (SDR элементы), а есть элементы, работающие по обоим фронтам (DDR элементы). Пример – DDR память.

Зачем нужны эти такты? Такты можно легко считать, измеряя тем самым *длительность* выполнения некой операции. Эта величина может быть как больше, так и меньше периода тактового сигнала , но даже если она и меньше , то мы сможем получить результат ее выполнения только на следующем фронте. В первом приближении такт – это *минимальный квант* времени в мире синхронных цифровых схем.

Необходимость введения такого кванта времени была обусловлена зависимостью времени выполнения комбинационной схемы от внешних факторов, прежде всего – температуры и напряжения. Если в схеме несколько путей распространения сигналов, и рано или поздно они сходятся на очередном элементе с несколькими входами, то надо быть уверенными, что относящиеся к одному событию сигналы пришли к общему элементу одновременно. С использованием только комбинационной логики сделать это крайне трудно, т.к. задержки на комбинационных путях, хотя их и можно рассчитать, зависят от внешних

факторов.

Для получения стабильного результата необходимо, чтобы входные сигналы комбинационного элемента не менялись, пока не завершаться все переходные процессы в элементе, т.е. эти данные должны храниться в какой-то памяти, назовем ее триггер. Выходные данные так же должны где-то зафиксироваться, чтобы их можно было передать следующей схеме. При этом, время установления выходных данных определяет частоту работы данной схемы. Подобная парадигма позволяет формировать некоторые логические блоки, которые могут работать независимо друг от друга, т.к. выходные триггеры одной схемы могут быть входными у другой.

Триггеры

Особенностью последовательных логических элементов является зависимость выходного сигнала не только от действующих в настоящий момент на входе логических переменных, но и от тех значений переменных, которые действовали на входе в предыдущие моменты времени. Для выполнения этого условия значения переменных должны быть запомнены логическим устройством. Функцию запоминания значений логических переменных в цифровых системах выполняют так называемые триггеры.

Асинхронный RS-триггер

Асинхронный RS-триггер имеет два входа $S(\text{et})$ – установка и $R(\text{eset})$ – сброс и два выхода прямой – Q и инверсный – $\tilde{Q}$. Триггер переходит из текущего состояния X на выходе к состоянию 0, при подаче на вход S нуля и на вход R единицы, а при поступлении на вход S единицы и на вход R нуля триггер переходит к состоянию 1. При нулевых значениях, когда $S = R = 0$ триггер должен сохранять старое значение. Комбинация сигналов $S = R = 1$ не определена.

`Рисунок RS-триггера`

Рис. 5.1: Обозначение и схема асинхронного RS-триггера

Приведенному описанию соответствует схема, приведенная на рисунке. RS триггер можно строить как на элементах «2ИЛИ-НЕ», так и на элементах «2И-НЕ». В первом случае получается триггер с прямыми входами (т.е. с единичным активным уровнем).

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

Во втором случае триггер будет управляться нулевым активным уровнем сигналов, и будет называться RS триггером с инверсными входами.

$\tilde{S}$	$\tilde{R}$	Q_{n+1}
0	0	?
0	1	1
1	0	0
1	1	Q_n

RS триггер со статической синхронизацией

К RS триггеру можно добавить сигнал синхронизации . Когда $C = 1$, то триггер может переключиться, если $C = 0$, то не может. Такой триггер будет называться синхронным RS триггером со статической синхронизацией. Значение слова «статической» вам будет ясно чуть позже.

Рисунок RS-триггера с сигналом синхронизации

Рис. 5.2: Обозначение и схема асинхронного RS-триггера с сигналом синхронизации

Далее можно добавить асинхронные входы установки и сброса триггера.

Рисунок RS-триггера с сигналом синхронизации

Рис. 5.3: Обозначение и схема асинхронного RS-триггера с сигналом синхронизации и асинхронными сигналами установки и сброса

Асинхронными они называются потому, что подаются напрямую на ячейку памяти, и сигнал синхронизации никак не может на них повлиять.

D триггер (D защёлка)

Вместо двух входов R и S у триггера может быть только один вход D (Data).

Идея заключается в том, что требуется исключить запрещенное значение входов, т.е. подавать на RS-триггер только разрешенные комбинации. Для этого мы на вход S подаём сигнал D, а на вход R подаём $\tilde{D}$. Таким образом мы исключаем запрещенное значение на входах. Для того, чтобы триггер мог хранить свое значение, вводится сигнал *статической* синхронизации C. Рассмотренная схема является *защёлкой* (latch). На VHDL она реализуется следующим образом:

Рисунок RS-триггера с сигналом синхронизации

Рис. 5.4: Обозначение и схема асинхронного D-триггера

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity Latch is
4      port ( C : in  STD_LOGIC;
5             D : in  STD_LOGIC;
6             Q : out STD_LOGIC);
7  end Latch;
8
9  architecture Behavioral of Latch is
10     signal q_tmp : std_logic := '0';
11 begin
12     latch_process: process (C, D)
13     begin
14         if (C = '1') then
15             q_tmp <= D;
16         end if;
17     end process;
18     Q <= q_tmp;
19 end Behavioral;
```

Может показаться, что код триггера – защелки очень похож на код, описывающий комбинационный элемент. Однако, видно, что сигнал q_tmp будет назначаться не при каждом вызове процесса latch_process, потому дерево условий **if** тут неполное. Процесс будет запускаться при любом изменении сигналов из списка чувствительности C и D. Представим, что D меняется в тот момент, когда C = 0. Тогда процесс запуститься, но условие C = 1 не будет выполнено. В этом случае сигнал q_tmp в соответствии с семантикой языка VHDL сохранит свое предыдущее значение, и получиться аппаратный элемент триггер-защёлка.

Триггер-защёлка – это асинхронный элемент памяти, и в проектах на FPGA его использовать настоятельно не рекомендуется (хотя САПР это позволяет). Ниже в этой лекции при обсуждении методологии проектирования синхронных схем будет ясно почему.

D триггер с динамической синхронизацией

Рассмотренные выше триггеры имеют статическую синхронизацию. Это означает, что они управляются уровнем сигнала синхронизации. Т.е., этот сигнал выступает в виде разрешения на изменения. Промежуток времени, когда $C = 1$ может быть довольно значителен. В течение этого времени триггер просто передает на выход значение входного сигнала. Если вход триггера изменится, то

изменится и выход. Обычно в цифровых системах значение триггера меняется только один раз за период тактового сигнала, и поэтому вход триггера должен оставаться неизменным в течение всего времени пока $C = 1$, а это накладывает сильные ограничения на входной сигнал и сильно усложняет проектирование схем.

`Временная форма`

Рис. 5.5: Временная форма сигналов для D-триггеров со статической и динамической синхронизацией

Ситуацию можно сильно упростить, если сделать триггер нечувствительным к изменению входного сигнала даже во время действия высокого уровня синхросигнала и заставить его переключаться в течение намного меньшего промежутка времени вокруг так называемого фронта синхроимпульса. Такие триггеры есть, и называются они триггерами с динамической синхронизацией. Говорят, что они чувствительны не к уровню (как триггеры со статической синхронизацией), а к изменению или фронту синхросигнала. Если изменение состояния триггера происходит по переднему фронту синхросигнала, т.е. по изменению сигнала с 0 на 1, то говорят, что он снабжен прямым динамическим входом. Если же изменение состояния триггера происходит по заднему фронту, т.е. по изменению с 1 на 0, то считают, что он имеет инверсный динамический вход.

`Рисунок D-триггера`

Рис. 5.6: Архитектура D-триггера с динамической синхронизацией

Построить триггер с динамической синхронизацией довольно несложно. Достаточно соединить последовательно две D-защелки (первую назовем master-защелкой, вторую – slave-защелкой). Пусть CLK – сигнал синхронизации (синхросигнал). Когда $CLK = 0$, master-триггер пропускает все, что подается на его вход, но slave-триггер помнит свое состояние, и в итоге на выходе сигнал соответственно не изменяется. Когда CLK становится равным 1, master-триггер закрывается, запоминая последнее значение входного сигнала, а slave-триггер открывается и пропускает значение с master-триггера на выход. В каждый момент времени, когда сигнал CLK постоянен какой-то из триггеров поддерживает постоянное значение на своем выходе, а, следовательно, и на выходе всей схемы. Только в момент перехода сигнала CLK из 0 в 1 происходит изменение состояния триггеров.

Рассмотренный D триггер с динамической синхронизацией сохраняет значение логической переменной, действующей на информационном входе D в момент каждого переднего фронта синхросигнала. Иногда требуется, чтобы триггер изменял свое значение не по каждому такому фронту, а только в определенные промежутки времени. Для этого требуется введение дополнительного управляющего сигнала, называемого CE (Clock Enable). Он может быть реализовываться через элемент «И» входов CE и . Отсюда и название: CE – Clock Enable – Разрешение Тактов. Однако это приводит к внесению дополнительной задержки на синхровход, и такой подход не применяется в ПЛИС. Второй

подход заключается в использовании мультиплексора в цепи информационного входа D.

Рисунок D-триггера

Рис. 5.7: Архитектура D-триггера с динамической синхронизацией и разрешением записи

D триггер является базовым триггером для ПЛИС. В дальнейшем, когда мы будем говорить «D триггер», мы будем иметь в виду D триггер с динамической синхронизацией и входом разрешения работы CE.

Счетный T триггер

Последний триггер, который мы рассмотрим, называется счётный T триггер. По определению если в момент прихода синхроимпульса на входе = 1, то триггер изменяет свое состояние на противоположное. Если = 0 в момент прихода фронта тактового сигнала, триггер сохраняет свое значение.

Рисунок D-триггера

Рис. 5.8: Архитектура T-триггера с динамической синхронизацией

Реализация триггеров на VHDL

Триггеры внутри ПЛИС реализуются не за счёт логических элементов, как было описано выше, они уже реализованы внутри кристалла. В прошивке указывается то, как они должны быть сконфигурированы (вид сброса, наличие сигнала разрешения на запись и т.п.). Чтобы трассировщик знал какую именно конфигурацию выбрать необходимо определённым образом описывать поведение сигналов. При описании процессов в списке чувствительности должен находится только синхросигнал, т.к. только его изменение должно вызывать процесс, а также асинхронные сигналы, например, сигнал асинхронного сброса. Внутри процесса все сигналы, значения которых сохраняются на триггере, должны находится внутри следующей конструкции:

```
if (clk'event and clk = '1') then
    ...
end if;
```

Здесь сигнал clk – синхросигнал. Приведённый выше код предназначен для случая синхронизации по фронту, если же необходима синхронизация по спаду, то она должна иметь вид:

```
if (clk'event and clk = '0') then
    ...
end if;
```

Для краткости можно писать вместо clk'event **and** clk = '1' функцию rising_edge(clk). Аналогично и для конструкции clk'event **and** clk = '0' заменой будет falling_edge(clk). В списке чувствительности процесса, в котором должна находится эта конструкция, должен быть объявлен сигнал clk. Если в схеме используются какие-то асинхронные сигналы, то они тоже должны быть добавлены в список чувствительности.

Написание testbench для синхронных схем

Отличительной особенностью синхронных схем является наличие синхронизирующего сигнала. Обычно этот сигнал является внешним для схемы и генерируется внешним кварцевым генератором. Рассмотрим три способа симуляции этого сигнала в testbench.

```
   library ieee;
1  use ieee.std_logic_1164.all;
2
3  entity main_tb is
4  end main_tb;
5
6  architecture Behavioral of main_tb is
7      signal clk_0 : std_logic := '0'; -- Важно присвоить
8                                       -- начальное значение
9
10     signal clk_1 : std_logic := '0'; -- Важно присвоить
11                                      -- начальное значение
12
13     signal clk_2 : std_logic;        -- Не обязательно
14                                      -- присваивать начальное
15                                      -- значение
16
17     constant clk_period : time := 10 ns;
18
19     signal a : std_logic_vector(2 downto 0) := (others =>
      '0');
20 begin
21     -- Первый способ
22     clk_0 <= not clk_0 after clk_period/2;
23
24     -- Второй способ
25     process
26     begin
27         wait for clk_period/2;
28         clk_1 <= not clk_1;
29     end process;
30
31     -- Третий способ
32     process
33     begin
34         clk_2 <= '0';
35         wait for clk_period/2;
36         clk_2 <= '1';
37         wait for clk_period/2;
38     end process;
39
40     -- Пример использования
41     a <= (0 => clk_0, 1 => clk_1, 2 => clk_2);
42 end Behavioral;
```

Все эти способы дают одинаковый результат. Третий способ является наиболее громоздким, но в то же время наиболее гибким. Только таким способом можно симулировать сигнал со скважностью, отличной от 50% (т.е. случай, когда длительность '1' не равна длительности '0').

Пример реализации D триггера

Рассмотрим код, реализующий простой D триггер. На входе у него синхроимпульс clk и вход данных D, выход – Q.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity Dtrig is
4      port ( clk : in  STD_LOGIC;
5             D   : in  STD_LOGIC;
6             Q   : out STD_LOGIC);
7  end Dtrig;
8
9  architecture Behavioral of Dtrig is
10 begin
11     process (clk)
12     begin
13         if (rising_edge(clk)) then
14             Q <= D;
15         end if;
16     end process;
17 end Behavioral;
```

Пример реализации D триггера с сигналом разрешения записи

Модернизируем предыдущий пример, добавив сигнал CE – сигнал разрешения.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity DtrigCE is
4      port ( clk : in  STD_LOGIC;
5             D   : in  STD_LOGIC;
6             CE  : in  STD_LOGIC;
7             Q   : out STD_LOGIC);
8  end DtrigCE;
9
10 architecture Behavioral of DtrigCE is
11     signal Q_tmp : std_logic := '0';
12 begin
13     process (clk)
14     begin
15         if (rising_edge(clk)) then
16             if (CE = '1') then
17                 Q_tmp <= D;
18             end if;
19         end if;
20     end process;
21     Q <= Q_tmp;
22 end Behavioral;
```

Пример реализации D триггера с синхронным сбросом

Рассмотрим случай сигнала синхронного сброса reset у обычного D триггера.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity DtrigSReset is
4      port ( clk    : in  STD_LOGIC;
5             D      : in  STD_LOGIC;
6             reset  : in  STD_LOGIC;
7             Q : out STD_LOGIC);
8  end DtrigSReset;
9
10 architecture Behavioral of DtrigSReset is
11
12 begin
13     process (clk)
14     begin
15         if (rising_edge(clk)) then
16             if (reset = '1') then
17                 Q <= '0';
18             else
19                 Q <= D;
20             end if;
21         end if;
22     end process;
23 end Behavioral;
```

Пример реализации D триггера с асинхронным сбросом

Рассмотрим случай сигнала асинхронного сброса reset у обычного D триггера. Для этого сам процесс должен реагировать на изменения сигнала reset, независимо от состояния линии clk, поэтому необходимо включить сигнал reset в список чувствительности. Изменения выходного сигнала Q, связанного со сбросом, должно происходить вне блока

```
   if (rising_edge(clk)) then
1      ...
2  end if;
```

Этот блок имеет следующий вид:

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity DtrigAReset is
4      port ( clk    : in  STD_LOGIC;
5             D      : in  STD_LOGIC;
6             reset  : in  STD_LOGIC;
7             Q      : out STD_LOGIC);
8  end DtrigAReset;
9
10 architecture Behavioral of DtrigAReset is
11
12 begin
13     process (clk, reset)
14     begin
15         if (reset = '1') then
16             Q <= '0';
17         elsif (rising_edge(clk)) then
18             Q <= D;
19         end if;
20     end process;
21 end Behavioral;
```

Пример реализации T триггера

Рассмотрим простой счётный триггер. У него есть только один входной сигнал – синхроимпульс (clk) и один выходной – данные (Q).

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity Ttrig is
4      port ( clk : in  STD_LOGIC;
5             Q   : out STD_LOGIC);
6  end Ttrig;
7
8  architecture Behavioral of Ttrig is
9      signal Q_tmp : std_logic := '0';
10 begin
11     process (clk)
12     begin
13         If (rising_edge(clk)) then
14             Q_tmp <= not Q_tmp;
15         end if;
16     end process;
17     Q <= Q_tmp;
18 end Behavioral;
```

Детектор переднего фронта сигнала

В качества другого примера использования триггеров рассмотрим детектор фронта, т.е. устройство, которое выдаёт импульс, как только происходит перепад входного сигнала из 0 в 1. Входные сигнала элемента: rst_n – сигнал сброса (активен нулем, о чем говорит суффикс _n), clk – тактовая частота, sig_in – внешний сигнал, tick_out – импульс, который сигнализирует об изменении sig_in из 0 в 1.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity rising_edge_detect is
4      port ( rst_n    : in  STD_LOGIC;
5             clk      : in  STD_LOGIC;
6             sig_in   : in  STD_LOGIC;
7             tick_out : out STD_LOGIC);
8  end rising_edge_detect;
9
10 architecture rtl of rising_edge_detect is
11     signal sig_n    : std_logic;
12 begin
13
14 process (clk,rst_n)
15 begin
16     if rst_n = '0' then
17         sig_n    <= '0';
18     elsif rising_edge(clk) then
19         sig_n    <= not sig_in;
20     end if;
21 end process;
22
23 tick_out    <= sig_in and sig_n;
24 end rtl;
```

Обратите внимание, входной сигнал sig_in изменяется без привязки к фронтам тактового сигнала clk. Внутренний сигнал sig_n схемы – это пропущенное через триггер (а значит задержанное на один такт) отрицание сигнала sig_in. Далее сигнал tick_out назначается в помощью параллельного оператора назначения сигнала вне процесса (это комбинационный элемент). Он будет равен 1 только если оба сигнала sig_in и sig_n равны 1 одновременно. Это приводит к тому, что длительность сигнала tick_out всегда будет меньше одного такта. Часто из-за отсутствия прямой связи между входным и тактовым сигналами длительность tick_out может быть совсем маленькой, что потом приводит к проблемам.

`Рисунок архитектуры детектора фронтов`

Рис. 5.9: Архитектура детектора фронтов

Важно сделать схему, выдающую стабильный результат с длительностью выходного сигнала ровно 1 такт. Для этого выход схемы достаточно просто пропустить еще через один триггер.

Сложные последовательные элементы

Перейдём теперь к соединению триггеров между собой. Один триггер может представлять 1 битовое двоичное число. Для представления n-битного числа требуется n битов. На практике мы часто работаем с 8, 16 и 32 битными числами. Если соединить несколько триггеров вместе, то получится регистр. Регистром называется последовательное устройство, предназначенное для записи, хранения и (или) сдвига информации, представленной в виде многоразрядного двоичного кода. Любой N разрядный регистр состоит из N однотипных ячеек – разрядных схем. При этом каждая разрядная схема, как любое последовательное устройство, состоит из триггера (элемента памяти) и некоторой комбинационной схемы, преобразующей входные воздействия и состояние триггера в выходные сигналы регистра. Ниже приведена схема регистра (слева) и его условное обозначение (справа).

Рисунок регистра

Рис. 5.10: Реализация регистра

Сдвиговые регистры

Сдвиговые регистры выполняются на основе триггеров с динамическим синхровходом. Для этого триггеры соединяются последовательно, выходы предыдущих триггеров соединяются со входами последующих. Ниже приведена схема такого элемента, построенного на D-триггерах (слева) и его условное обозначение (справа):

Рисунок сдвигового регистра

Рис. 5.11: Реализация сдвигового регистра

С приходом очередного положительного фронта синхроимпульса C длительностью t0,1, сигнал со входа i-го триггера через время tзд.р. окажется на его выходе и поступит на вход следующего (i+1)-го триггера. Однако на его выход эта информация не перепишется, т.к. длительность активного фронта t0,1 меньше tзд.р.. На этом процесс сдвига данных на один разряд закончится до прихода следующего положительного фронта тактового сигнала. Основные назначения сдвиговых регистров – задержка данных, преобразование из параллельного кода в последовательный и обратно.

Пример реализации сдвигового регистра на VHDL

Рассмотрим 8-ми разрядный последовательный сдвиговый регистр. Вход данных обозначим как D, синхроимпульс обозначается как раньше clk, а выходной сигнал – Q.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity SReg8 is
4      port ( clk : in  STD_LOGIC;
5             D   : in  STD_LOGIC;
6             Q   : out STD_LOGIC);
7  end SReg8;
8
9  architecture Behavioral of SReg8 is
10     signal S : std_logic_vector(7 downto 0) := (others =>
       '0');
11 begin
12     process (clk)
13     begin
14         if (rising_edge(clk)) then
15             S(7 downto 1) <= S(6 downto 0);
16             S(0) <= D;
17         end if;
18     end process;
19     Q <= S(7);
20 end Behavioral;
```

В приведённом примере используется шина S для хранения данных сдвигового регистра. Каждая из линий S представляется в виде отдельного триггера, процесс переноса описывается строчкой: S(7 **downto** 1) <= S(6 **downto** 0); а процесс загрузки новых данных: S(0) <= D;. Данные выталкиваются наружу из старшего триггера: Q <= S(7);.

Пример реализации сдвигового регистра с параллельной загрузкой

На практике часто необходимо преобразовывать параллельный код в последовательный обратно. Это делается с помощью сдвиговых регистров. Рассмотри преобразование параллельного кода в последовательный. К предыдущему примеру добавятся дополнительные сигналы: Load (флаг загрузки данных) и LD (загружаемые данные). Данные LD – это 8-битовое слово в параллельном коде, которое необходимо перевести в последовательный код, а значит выдать в однобитовую выходную линию Q значение LD бит за битом за 8 тактов. Здесь мы не используем входной сигнал D, вместо него сдвиговый регистры будет циклически перезаписывать сам себя.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity SReg8_ParLoad is
4      port ( clk  : in  STD_LOGIC;
5             LD   : in  STD_LOGIC_VECTOR (7 downto 0);
6             Load : in  STD_LOGIC;
7             Q    : out STD_LOGIC);
8  end SReg8_ParLoad;
9
10 architecture Behavioral of SReg8_ParLoad is
11     signal S : std_logic_vector(7 downto 0) := (others =>
       '0');
12 begin
13     process (clk)
14     begin
15         if (rising_edge(clk)) then
16             if (Load = '1') then
17                 S <= LD;
18             else
19                 S(7 downto 0) <= S(6 downto 0) & S(7);
20             end if;
21         end if;
22     end process;
23     Q <= S(7);
24 end Behavioral;
```

Счётчики

Счётчик – последовательная схема, которая преобразует поступающие на вход импульсы в параллельный код, эквивалентный количеству пришедших импульсов. Счётчики могут считать импульсы в различных системах отсчёта (двоичный, двоично-десятичный), иметь сигнал разрешения на счёт, возможность параллельной загрузки состояния счётчика, выдача сигнала о переполнении, либо о достижении какого-то значения. Ниже приведена схема самой простой реализации четырёхбитного счётчика.

`Рисунок счётчика на T-триггера + с помощью регистра и "+1"`

Рис. 5.12: Реализация счетчика

Будем считать, что нумерация триггеров идёт от верхнего (0-й) до нижнего (3-й). В начальном состоянии все регистры находятся в 0-м состоянии, при приходе синхроимпульса своё состояние на 1 меняет только 0-й T триггер, так как на остальные поступают только 0. На следующем такте он же меняется обратно в 0, а 1-й – в высокое состояние. На следующем такте меняет своё состояние на

1 только 0-й триггер. 2-й триггер не может этого сделать, т.к. зависит от предыдущего состояния 0-го триггера. И так далее до тех пор, пока все они не будут в высоком состоянии. Следующим их состоянием с приходом синхроимпульсом будет 0, так называемое переполнение счётчика.

Пример реализация счётчика на VHDL

Рассмотрим реализацию 4-х битного счётчика, который считает каждый импульс. Входной сигнал - clk (синхроимпульс), выходной сигнал – Q.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2  use ieee.std_logic_arith.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity Counter4 is
6      port ( clk : in  STD_LOGIC;
7             Q   : out STD_LOGIC_VECTOR (3 downto 0));
8  end Counter4;
9
10 architecture Behavioral of Counter4 is
11     signal counter : std_logic_vector(3 downto 0)
12                    := (others => '0');
13 begin
14     process (clk)
15     begin
16         if (rising_edge(clk)) then
17             counter <= counter + 1;
18         end if;
19     end process;
20     Q <= counter;
21 end Behavioral;
```

Пример реализации реверсивного счётчика с параллельной загрузкой

Помимо прямого счёта, т.е. инкрементального, бывают и реверсивные счётчики, которые считают в обратном направлении. Это может оказаться полезным, когда необходимо отсчитать какое-то количество тактов и выдать сигнал об окончании счёта.

Рассмотрим реверсивный счётчик, который выдаёт сигнал об окончании счёта. В качестве входов у него будет синхроимпульс (clk), флаг загрузки данных (load) и шину данных (DL). Выходной сигнал CR – окончание счёта.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2  use ieee.std_logic_arith.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity RevCounter4 is
6      port ( clk   : in  STD_LOGIC;
7             load  : in  STD_LOGIC;
8             DL    : in  STD_LOGIC_VECTOR (3 downto 0);
9             CR    : out STD_LOGIC);
10 end RevCounter4;
11
12 architecture Behavioral of RevCounter4 is
13     signal counter : std_logic_vector(3 downto 0)
14                      := (others => '1');
15 begin
16     process (clk)
17     begin
18         if (rising_edge(clk)) then
19             if (load = '1') then
20                 counter <= DL;
21             else
22                 if (counter /= 0) then
23                     counter <= counter - 1;
24                 end if;
25             end if;
26         end if;
27     end process;
28     CR <= '1' when counter = 0 else '0';
29 end Behavioral;
```

Пример реализации сдвигового регистра с параллельным выходом на VHDL

Рассмотрим теперь обратное преобразование последовательного кода в параллельный. Пусть на вход D поступают входные данные бит за битом. Мы будем сохранять их в сдвиговым регистре, значение которого будет выдаваться на выходную 8ми битную шину схемы после прихода 8 бит данных.

```
   library IEEE;
1  use IEEE.STD_LOGIC_1164.ALL;
2
3  entity SReg8_ParOut is
4      port ( clk    : in  STD_LOGIC;
5             reset  : in  STD_LOGIC;
6             en     : in  STD_LOGIC;
7             D      : in  STD_LOGIC;
8             Q      : out STD_LOGIC_VECTOR (7 downto 0)
9             valid  : out STD_LOGIC
10 );
11 end SReg8_ParOut;
12
13 architecture Behavioral of SReg8_ParOut is
14     signal S : std_logic_vector(7 downto 0);
15     signal cnt : std_logic_vector(3 downto 0);
16 begin
17     process (clk)
18     begin
19         if (rising_edge(clk)) then
20
21             if reset = '1' then
22                 S    <= (others => '0');
23             elsif en = '1' then
24
25                 S(7 downto 1) <= S(6 downto 0);
26                 S(0) <= D;
27             end if;
28
29             if reset = '1' then
30                 cnt  <= (others => '0');
31             elsif cnt = "1000" then
32                 cnt <= "0000"
33             elsif en = '1' then
34
35                 cnt  <= cnt + 1;
36             end if;
37
38         end if;
39     end process;
40     Q <= S;
41     valid <= '1' when cnt = "1000" else '0';
42
43 end Behavioral;
```

Возможна реализация различных сдвиговых регистров: с параллельной загрузкой, реверсивные регистры (направление сдвига определяется состоянием одного из входов), частично параллельный регистр (не все триггеры выставляют данные на выходную шину) и т.п. На практике все эти виды регистров полезны для перевода параллельного кода (шины) в последовательный код (линию) и обратно, что используется при передаче данных в таких протоколах как USB, RS232, Ethernet и т.п.

Мигание светодиодом

Реализуйте схему, которая мигала бы светодиодом 1 раз в секунду (0,5 секунды светодиод не горит, 0,5 секунды светодиод горит).

Светодиодный сдвиговый регистр

Реализуйте сдвиговый регистр на светодиодах и переключателях. На плате ZYBO имеется 4 светодиода: LD0 – LD3. На LD0 должно выводиться значение переключателя SW0. Затем каждую секунду значение LD0 должно переводиться на LD1, значение LD1 – на LD2, и т.д.

Контрольные вопросы

1. Схема RS триггера, его таблица истинности и описание поведения отдельных элементов в зависимости от состояний на входах.

2. Получить из RS триггера D триггер.

3. VHDL код D триггера.

4. VHDL код T триггера.

5. Схема простого сдвигового регистра на D триггерах.

6. Схема сдвигового регистра с параллельным выходом.

7. VHDL код сдвигового регистра с параллельной загрузкой.

8. VHDL код сдвигового регистра с параллельной загрузкой и выходной шиной.

9. Схема простого счётчика и описание его поведения (временная диаграмма работы).

10. VHDL код простого счётчика.

11. VHDL код реверсивного счётчика с параллельной загрузкой.

Домашнее задание

1. Создать детектор спадов.
2. Создать детектор фронтов и спадов.
3. Моделирование. Создать детектор фронта, выходной импульс которого длится: 4 такта, 8 тактов, может быть в пределах от 1 до 16 (задается с помощью входного сигнала шириной 4 бита).

Глава 6

Архитектура FPGA

Архитектура ПЛИС – углубленно. Фабрика логики. Структура конфигурационного логического блока и блоков ввода/вывода. Общие трассировочные ресурсы, трассировочные ресурсы для сигналов синхронизации, генераторы частоты, DSP блоки, блочная и распределенная память. Конфигурационная память и способы загрузки конфигурационных данных.

Вывод на консоль во время симуляции

Глава 7

Конвейерные вычисления

Этапы разработки проекта ПЛИС в системе Vivado. Анализ схемы после синтеза, после имплементации. Времяные ошибки и методы борьбы. Уменьшение количества комб логики между триггерами, понижение частоты.

Вывод на консоль во время симуляции

Глава 8

Описание конечных автоматов на VHDL

...

Вывод на консоль во время симуляции

Глава 9. Работа с BRAM

Интерфейс BRAM, однопортовый и двухпортовый режимы работы.

Введение

Внутри ПЛИС Xilinx есть специальный аппаратный ресурс – блочная память или BRAM (Block RAM). Это набор из специальных аппаратных блоков статической памяти, выполненных аппаратно, т.е. часть логики кристалла зафиксирована под память, и не может быть использована по другому назначению. Количество таких блоков зависит от конкретного типа кристалла ПЛИС и может быть до нескольких сотен. Каждый блок имеет два аппаратных порта, которые могут работать на разных тактовых частотах. У блочной памяти есть свои преимущества и недостатки. С одной стороны, это быстрые аппаратные блоки, способные работать на частотах до 500 МГц. При использовании блочной памяти ее разные порты могут работать на разных частотах. Это удобно использовать для синхронизации между разными клоковыми доменами. Но с другой стороны, блочная память – дорогостоящий ресурс. Количество ее ограничено. Если требуется объем памяти меньший чем объем одного блока (18 Кбит), то все равно будет использован весть блок, и неиспользованная его часть просто пропадет. Для создания оперативных и постоянных запоминающих устройств, реализуемых на основе блочной памяти ПЛИС, в составе генератора параметризованных модулей LogiCORE предусмотрено такое ядро, как Block Memory Generator. В данной главе будет подробно рассмотрено его использование, настройка и реализация с помощью средств языка VHDL.

Создание и настройка ядра BRAM

Заходим в Block Memory Generator

Выберите Block Memory Generator из каталога IP (рис. 1-2).

Рис. 8.1: Рис.1

Рис. 8.2: Рис.2

Работа с интерфейсом BMG

Выбор типа и организации запоминающего устройства, формируемого на основе рассматриваемого параметризированного модуля, осуществляется с помощью соответствующего *мастера* настройки параметров, который содержит от четырёх до шести (в зависимости от конфигурируемых параметров) диалоговых панелей.

Basic

Рис. 8.3: Рис.3

В стартовой диалоговой панели *мастера*, вид которой показан на рисунке, указывается тип создаваемого элемента памяти, его название, а также алгоритм его реализации.

- Название формируемого вида запоминающего устройства вводится с помощью клавиатуры в поле редактирования Component Name.

На выбор доступны два типа интерфейса – Native и AXI4.

Память блоков интерфейса AXI4 построена на памяти блоков нативного (Native) типа. Доступны два стиля интерфейса AXI4: AXI4 и AXI4-Lite. Также на выбор конфигурируется ядро. Далее на выбор предполагается настройка устройства памяти как ведомое или периферийно. В дополнение к приложениям, поддерживаемым нативным интерфейсом, AXI4 может также использоваться в приложениях AXI4 System Bus и приложениях типа Point-to-Point.

Для нативного типа интерфейса предполагаются следующие параметры настройки:

- Для определения типа генерируемого элемента памяти следует воспользоваться группой кнопок с зависимой фиксацией Memory Type. Если в нажатом состоянии зафиксирована кнопка Single Port RAM, то будет сформировано ОЗУ с одним портом записи и одним портом чтения данных. При нажатой кнопке Simple Dual Port RAM создается элемент двухпортовой оперативной памяти с одним портом чтения данных.Кроме того, при выборе данного типа памяти становится доступной опция ECC (Error Correction Checking) Type, созданная для регистрации и исправления ошибок с помощью кода Хэмминга и поиска необходимых совместимостей. Для генерации ОЗУ с двумя портами чтения и двумя портами записи данных нужно переключить в нажатое состояние кнопку True Dual Port RAM. Формирование однопортового постоянного запоминающего устройства осуществляется при нажатой кнопке Single Port ROM. Чтобы создать элемент двухпортовой постоянной памяти, следует перевести во включенное состояние кнопку Dual Port ROM. Для двухпортовых элементов доступна опция Common Clock для синхронизации часов ввода (управление одним буфером)

- Также может быть установлен режим побайтной записи. Для этого нужно перевести в состояние *включено* индикатор Use Byte Write Enable, который расположен во встроенной панели Write Enable.

- Побайтная запись может осуществляться с контролем и без контроля четности. При этом размер записываемого байта (количество бит в байте) зависит от использования контроля четности. В случае отсутствия контроля четности длина байта составляет восемь бит. Чтобы побайтная запись выполнялась с контролем четности, в состав байта данных должно входить девять бит (восемь бит данных и один контрольный).

- Размер байта (и, соответственно, использование контроля четности) указывается с помощью поля выбора Byte Size.

- Запоминающее устройство, генерируемое с помощью параметризированного модуля Block Memory Generator, строится в виде совокупности примитивов блочной памяти Block RAM primitive. В общем случае каждый из используемых примитивов может иметь различную организацию. Количество и состав возможных вариантов организации примитивов блочной памяти Block RAM primitive определяется выбранным семейством ПЛИС. Тип организации примитивов блочной памяти и алгоритм их соединения для получения запоминающего устройства с требуемой емкостью и организацией указывается с помощью двух кнопок с зависимой фиксацией, которые расположены во встроенной панели Algorithm. Когда в нажатом состоянии находится кнопка Minimum Area, создаваемый элемент памяти составляется из примитивов с различной организацией с целью минимизации используемого количества модулей блочной памяти Block RAM кристалла ПЛИС. При нажатой кнопке Low Power создаваемый элемент памяти будет состоять из тех же, модулей, что и в Minimum Area ядре, но включёнными только при записи или чтении. При нажатой кнопке Fixed Primitive для построения требуемого запоминающего устройства будут использоваться примитивы блочной памяти одного типа, организация которого указывается пользователем с помощью поля выбора Primitive. Содержимое выпадающего списка этого поля выбора зависит от семейства ПЛИС, для которого создается элемент памяти.

Port A|B options

Вторая диалоговая панель *мастера* настройки параметров генератора оперативных и постоянных запоминающих устройств, реализуемых на основе блочной памяти ПЛИС, предназначена для определения емкости формируемого элемента памяти, а также описания организации и режима работы первого (или единственного) порта запоминающего устройства.

Рис. 8.4: Рис.4

- Чтобы указать информационную емкость формируемого элемента памяти и разрядность первого (или единственного) порта записи и чтения данных,

нужно воспользоваться полями редактирования и выбора, расположенными во встроенной панели Memory Size. Прежде всего, рекомендуется определить разрядность первого входного порта Port A, предназначенного для записи данных, с помощью поля редактирования Write Width. Требуемое число разрядов этого порта (в диапазоне от 1 до 1152) указывается с помощью клавиатуры, после активизации данного поля редактирования. Количество ячеек памяти в формируемом запоминающем устройстве задается в поле редактирования Write Depth. Максимальное значение этого параметра ограничено объемом физических ресурсов блочной памяти Block RAM используемого кристалла ПЛИС. Таким образом, информационная емкость создаваемого элемента памяти, выраженная в битах, равна произведению значений параметров Write Width и Write Depth. Разрядность первого (или единственного) выходного порта, используемого для чтения данных из памяти, определяется с помощью поля выбора Read Width. Выпадающий список этого поля выбора содержит допустимые варианты разрядности первого выходного порта, которые соответствуют установленному значению разрядности первого входного порта записи данных. Количество разрядов адреса (адресных входов) в формируемом запоминающем устройстве вычисляется автоматически, исходя из установленных значений параметров, рассмотренных выше. Значения параметров Write Width и Read Width должны быть кратны размеру байта данных, указанному в поле выбора Byte Size. В этом случае вместо обычного (одиночного) входа разрешения записи данных используется шина, количество разрядов которой вычисляется автоматически в соответствии с указанным значением разрядности порта записи.

- Для определения режима работы первого (или единственного) выходного порта элемента оперативной памяти, предназначенного для чтения данных, при выполнении операции записи данных в ОЗУ нужно воспользоваться группой кнопок с зависимой фиксацией Operating Mode. Если в нажатом состоянии находится кнопка Write First, то данные, поступающие во входной порт, записываются в соответствующую ячейку памяти (адрес которой задается комбинацией сигналов на адресных входах), после чего сразу передаются на выходы запоминающего устройства. При нажатии кнопки Read First в генерируемом элементе оперативной памяти будет установлен режим предварительного чтения данных из указанной ячейки перед записью новых данных в эту ячейку. Таким образом, при осуществлении операции записи данных, поступающих во входной порт формируемого элемента ОЗУ, на его выходах будет отображаться информация, которая содержалась в соответствующей ячейке перед этим (на предыдущем такте). Когда в нажатое состояние переводится кнопка No Change, в формируемом элементе ОЗУ будет установлен режим блокировки выходов при выполнении операции записи данных. В этом режиме на протяжении всего цикла записи выходы находятся в зафиксированном состоянии, которое соответствует последним считанным данным, присутствовавшим в момент переключения сигнала разрешения записи в актив-

ное состояние.

- Чтобы сформировать элемент запоминающего устройства с входом разрешения операций для первого порта, нужно переключить в нажатое состояние кнопку Use ENA Pin. При этом в созданном элементе памяти выполнение операций записи и чтения данных для первого порта будет возможно только при активном уровне сигнала на входе разрешения ENA.

- Для определения состояния выходного регистра или защелки формируемого запоминающего устройства в режиме сброса (или установки) следует воспользоваться полем редактирования Output Reset Value (Hex), которое расположено во встроенной панели Output Reset. Значение, указываемое в этом поле редактирования, должно быть представлено в шестнадцатеричном формате. При этом количество шестнадцатеричных символов должно соответствовать разрядности первого порта чтения данных. Чтобы задействовать в формируемом элементе памяти вход синхронного сброса (или установки) для первого порта (Port A), следует установить индикатор Use SSRA Pin, находящийся в этой же встроенной панели, в состояние *включено*.

- Если в стартовой диалоговой панели *мастера* настройки параметров генератора элементов памяти был выбран двухпортовый тип запоминающего устройства, то третья диалоговая панель будет позволять определить основные параметры второго порта (Port B) создаваемого элемента памяти. Все параметры этого порта, за исключением разрядности, указываются так же, как и для первого порта (Port A). Количество разрядов второго порта записи данных не может быть установлено произвольно. Значение этого параметра должно быть кратным числу разрядов первого порта записи данных (с фиксированным набором коэффициентов кратности), которое было указано в диалоговой панели, представленной на рисунке. Поэтому разрядность второго порта записи данных задается с помощью поля выбора Write Width, выпадающий список которого содержит только допустимые значения данного параметра. Содержимое этого списка автоматически корректируется при изменении значения разрядности первого порта записи данных.

Other options

Предпоследняя диалоговая панель *мастера* настройки параметров генератора оперативных и постоянных запоминающих устройств, реализуемых на основе блочной памяти ПЛИС, позволяет выбрать тип и конфигурацию выходных регистров, а также указать параметры инициализации формируемого элемента памяти.

Рис. 8.5: Рис.5

- Стадии конвейера в пределах Мультиплексора: доступна только, когда опция Register Output of Memory Core выбрана и для порта A и для порта

В и когда у созданной памяти есть больше чем один примитив (так, чтобы MUX был необходим при выводе). Выберите значение 0, 1, 2, или 3 из выпадающего списка.

- Для автоматической инициализации содержимого генерируемых запоминающих устройств, реализуемых на основе блочной памяти ПЛИС, следует определить значения соответствующих параметров с помощью элементов управления, которые расположены во встроенной панели Memory Initialization. Информацию, которую нужно записать в соответствующие ячейки формируемого элемента памяти, можно указать в виде файла формата COE. Для этого нужно, прежде всего, переключить индикатор Load Init File в состояние «включено», после чего станут доступными поле редактирования COE File и кнопка Browse. При нажатии на данную кнопку на экран выводится стандартная панель диалога открытия файла, с помощью которой нужно найти на одном из дисков компьютера требуемый файл, описывающий содержимое создаваемого запоминающего устройства. После выбора соответствующего файла и закрытия стандартной диалоговой панели его название автоматически отображается в поле редактирования COE File. Можно также с помощью клавиатуры сразу указать в этом поле редактирования название требуемого файла инициализации, не выполняя процедуру его поиска. Для быстрого просмотра содержимого выбранного файла нужно воспользоваться кнопкой Show, которая находится во встроенной панели Memory Initialization. Если указываемый файл инициализации описывает содержимое только части генерируемого запоминающего устройства, то все оставшиеся неинициализированными ячейки памяти могут быть заполнены по умолчанию значением, определяемым пользователем. С этой целью нужно установить индикатор Fill Remaining Memory Locations в состояние *включено*. При этом становится доступным поле редактирования Remaining Memory Locations (Hex), в котором нужно с помощью клавиатуры указать шестнадцатеричное значение, записываемое по умолчанию в ячейки памяти, оставшиеся неопределенными. Количество шестнадцатеричных символов, указываемых в этом поле редактирования, должно соответствовать разрядности первого порта записи генерируемого запоминающего устройства.

- Structural/UNISIM Simulation Model Options: Выберите тип предупреждающих сообщений и выводов, сгенерированных структурной моделью моделирования в случае коллизий. Для опций ALL, WARNING ONLY и GENERATE X ONLY, обнаружения коллизий, опция активирована в моделях UNISIM, чтобы обработать коллизию при любом условии.

- Behavioral Simulation Model Options: Выберите тип предупреждающих сообщений, сгенерированных моделью моделирования на поведенческом уровне. Выберите, должна ли модель принять синхронные часы (Общие Часы) для предупреждений коллизии.

- Dynamic Power Saving: экономия электроэнергии включена в положении, когда память активно не используется в течение длительного периода вре-

мени. Если матрица элементов памяти входит в режим ожидания, данные сохраняются. Чтобы использовать память, установите контакт сна в 0.

Summary

Заключительная диалоговая панель *мастера* настройки генератора оперативных и постоянных запоминающих устройств, реализуемых на основе блочной памяти ПЛИС, отражает параметры создаваемого устройства в виде списка.

Рис. 8.6: Рис.6

- Memory Type: Сообщает о выбранном типе памяти.
- Block RAM Resources: Сообщает точный номер 18 K и 36 K блочных примитивов RAM, которые используются, чтобы создать ядро.
- Total Port A Read Latency: номер тактов для Операции чтения для порта A. Этим значением управляют дополнительные выходные опции регистров для порта на предыдущей вкладке.
- Total Port B Read Latency: номер тактов для Операции чтения для порта B. Этим значением управляют дополнительные выходные опции регистров для порта B на предыдущей вкладке.
- Address Width: фактическая ширина адресной шины к каждому порту.

Power Estimation

Рис. 8.7: Рис.7

Вкладка Power Estimation на левой стороне IDE Vivado, показанная на рисунке, обеспечивает грубую оценку потребляемой мощности для ядра, основанного на сконфигурированных параметрах Read width, Write width, clock rate, Write rate and enable rate для каждого порта.

У этой вкладки есть опция *Additional Inputs for Power Estimation*, созданная для ввода дополнительных оценок потребляемой мощности. Можно ввести следующие параметры для расчета питания: Clock Frequency [A|B] (тактовая частота портов A и B), Write Rate [A|B] (скорость записи для портов A и B), Enable Rate [A|B] (средняя скорость доступа к портам A и B)

Включение созданного ядра в код VHDL

После выбора необходимых параметров для модуля BRAM следует нажать OK, а затем Generate во всплывающем окне. После того, как завершится синтез нашего модуля, переходим на вкладку IP Sources и открываем Файл в формате .vho в папке Instantiation Template(см. Рис. 8)

В этом файле уже созданы вставки для определения компонента в основном коде и инстанциация для тестбенча (рис. 9-10)

Рис. 8.8: Рис. 8

Рис. 8.9: Рис. 9

Симуляция

В качестве примера рассмотрим BRAM со следующими параметрами (всё, что не указано – по умолчанию):

1. Algorithm: Low Power

2. Write width:8, Read width:8, Write depth:16

3. Галочка Fill Remaining memory locations

Напишем тестбенч для данного модуля памяти:

Рис. 8.10: Рис. 10

```
   library IEEE;
1  use ieee.std_logic_1164.all;
2  use ieee.std_logic_arith.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity tb is
6  end tb;
7
8  architecture arch of tb is
9
10 --Объявление временных сигналов
11 signal ena : std_logic := '0';
12 signal wea : std_logic_VECTOR(0 downto 0):="0";
13 signal addra : std_logic_vector (3 downto 0) := (others=>
       '0');
14 signal dina,douta : std_logic_VECTOR(7 downto 0) := (others
     => '0');
15 signal clk : std_logic := '0';
16
17 begin
18
19 --Инстанциация подключение() модуля BRAM из( файла .vho)
20 BRAM : entity work.BRAM_sample
21      port map(
22      clka => clk,  --clock for writing data to RAM.
23      ena => ena,   --Enable signal.
24      wea => wea,   --Write enable signal for Port A.
25      addra => addra, --8 bit address for the RAM.
26      dina => dina,   --8 bit data input to the RAM.
27      douta => douta); --8 bit data output from the RAM.
28 clk <= not clk after 10 ns;
29 --Процесс симуляции
30 process
31 begin
32      wait for 10 ns;
33      --Заполняем ячейки BRAM данными. Для этого устанавливаем wea
        в "1".
34      for i in 0 to 15 loop
35           ena <= '1';  --RAM Работает всегда.
36           wea <= "1";
37           wait for 20 ns;
38           addra <= addra + "1";
39           dina <= dina + "1";
40      end loop;
41      addra <= "0000";  --сбрасываем параметр адреса в 0
42      for i in 0 to 15 loop
43           ena <= '1';  --RAM Работает всегда.
44           wea <= "0";
45           wait for 20 ns;
```

Результаты симуляции:

Рис. 8.11: Рис. 11

Лабораторная работа: Накопитель памяти на основе BRAM

Создаём модуль BRAM

Реализация на VHDL

```
   library ieee;
1  use ieee.std_logic_1164.all;
2  use ieee.numeric_std.all;
3  entity bram is
4  port(
5      clk, reset: in std_logic;
6      din: in std_logic_vector (15 downto 0);
7      dv_in: in std_logic;
8      dv_out: out std_logic;
9      dout : out STD_LOGIC_VECTOR ( 15 downto 0 )
10 );
11 end bram;
12 architecture arch of bram is
13     component blk_mem_gen_0 --Объявление компонента BRAM
14         Port (
15             clka : in STD_LOGIC;
16             wea : in STD_LOGIC_VECTOR ( 0 to 0 );
17             addra : in STD_LOGIC_VECTOR ( 3 downto 0 );
18             dina : in STD_LOGIC_VECTOR ( 15 downto 0 );
19             douta : out STD_LOGIC_VECTOR ( 15 downto 0 ));
20     end component;
21
22     type state_type is (state_wr, state_rd);
23     signal state_reg, state_next: state_type;
24     signal cnt_reg, cnt_next, cnt_next1:unsigned (3 downto
       0);
25     signal dv_reg, dv_reg1, dv_next: std_logic;
26     signal din_reg, dout_reg: std_logic_vector(15 downto 0);
27     signal addr: std_logic_vector(3 downto 0);
28     signal wr_en:  std_logic_vector ( 0 to 0 );
29
30 begin
31 blk_mem_inst : blk_mem_gen_0 --Инстаниация BRAM
32 PORT MAP (clka => clk,
33           wea => wr_en,
34           addra => addr,
35           dina => din,
36           douta => dout);
37
38 process(clk,reset) -- Настройка смены состояний по фронту clk
39 begin
40     if (reset='1') then
41         state_reg <= state_wr;
42         cnt_reg <= (others => '0');
```

Самопроверяющийся тестбенч

```
   library ieee;
1  use ieee.std_logic_1164.all;
2  use ieee.std_logic_textio.all;
3
4  library std;
5  use std.textio.all;
6
7
8  entity bram_tb is
9  end bram_tb;
10
11 architecture arch of bram_tb is
12 type num_input1 is array (integer range 0 to 15) of
      std_logic_vector(15 downto 0);
13
14
15    signal dout : std_logic_vector (15 downto 0);
16    signal num_input: num_input1;
17    signal din: std_logic_vector (15 downto 0);
18    signal dv_in,dv_out:std_logic;
19    signal clk : std_logic := '0';
20    signal reset: std_logic;
21    signal err: boolean := false;
22
23
24 begin
25  dut: entity work.bram (arch)
26
27 port map (
28 clk=>clk,
29 dout=>dout,
30 dv_in=>dv_in,
31 dv_out=>dv_out,
32 din=>din,
33 reset=>reset
34 );
35    clk <= not clk after 10 ns;
36
37    read_file_process: process
38       file   fd    : text is in  "../ip_header.txt";
39       variable word: std_logic_vector(15 downto 0);--line;
40       variable j : integer;
41       variable Message : line;
42       variable inline : line;
43    begin
44
45       Write ( Message, string'("-␣Reading␣data␣from␣file:␣␣"
      ));
46       writeline(output, Message);
```

Глава 9

Работа с очередями FIFO

Интерфес FIFO. Асинхронных режим. Режимы работы FIFO: Standard и First Word Fall Through. Интерфейс AXI Stream.

Вывод на консоль во время симуляции

Глава 10

Передача данных через UART

Описание протокола, реализация в ПЛИС

Вывод на консоль во время симуляции

Глава 11

Отладка проекта в железе

Описание работы Debug Core

Вывод на консоль во время симуляции