Основы языка VHDL

1. VHDL как методология проектирования электронных систем;
2. Важнейшие особенности языка VHDL от языков программирования;
3. Основные блоки описания системы на VHDL;
4. Потоковые модели;
5. Структурные модели;
6. Поведенческие модели;
7. Testbench;
8. Система ModelSim фирмы Mentor Graphics;
9. Алфавит моделирования.

Основы языка VHDL

Бурно развивающиеся компиляционные технологии автоматизации проектирования компьютеров и компьютерных сетей базируются на использовании **высокоуровневых языков описания аппаратуры**.

Если компилятор с языка программирования решает одну главную задачу - генерирует код для системы команд используемого процессора, то **компилятор с языка описания аппаратуры решает целый ряд задач**, составляющих в совокупности методологию проектирования современных цифровых систем : построение формальных моделей (**modeling**) и документирование проектов; имитационное моделирование (**simulation)**; логический (**logic-level**), высокоуровневый (**high-level**) или поведенческий (**behavioral**), архитектурный (**architectural**) и системный (system-level) синтез; формальную верификацию (**formal verification**) и другие задачи.

Основы языка VHDL (прдлж)

В настоящее время двумя наиболее широко распространенными языками описания аппаратуры являются **VHDL** и **Verilog**. Существуют международные стандарты этих языков.

Название языка VHDL произошло от английского

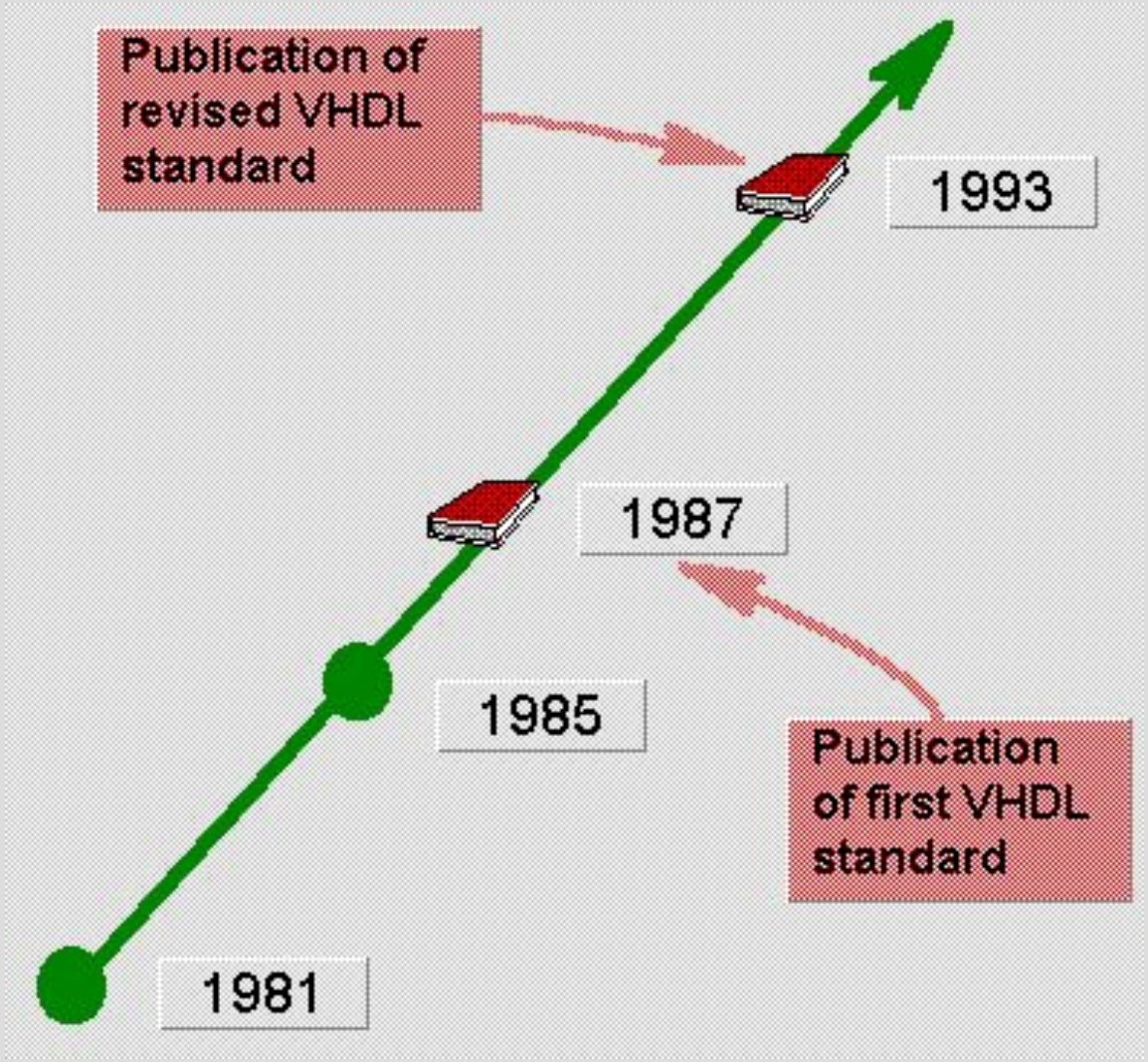
Vhsic (Very High Speed Integrated Circuit) **H**ardware

Description **L**anguage, что означает язык аппаратурного описания сверхскоростных интегральных схем.

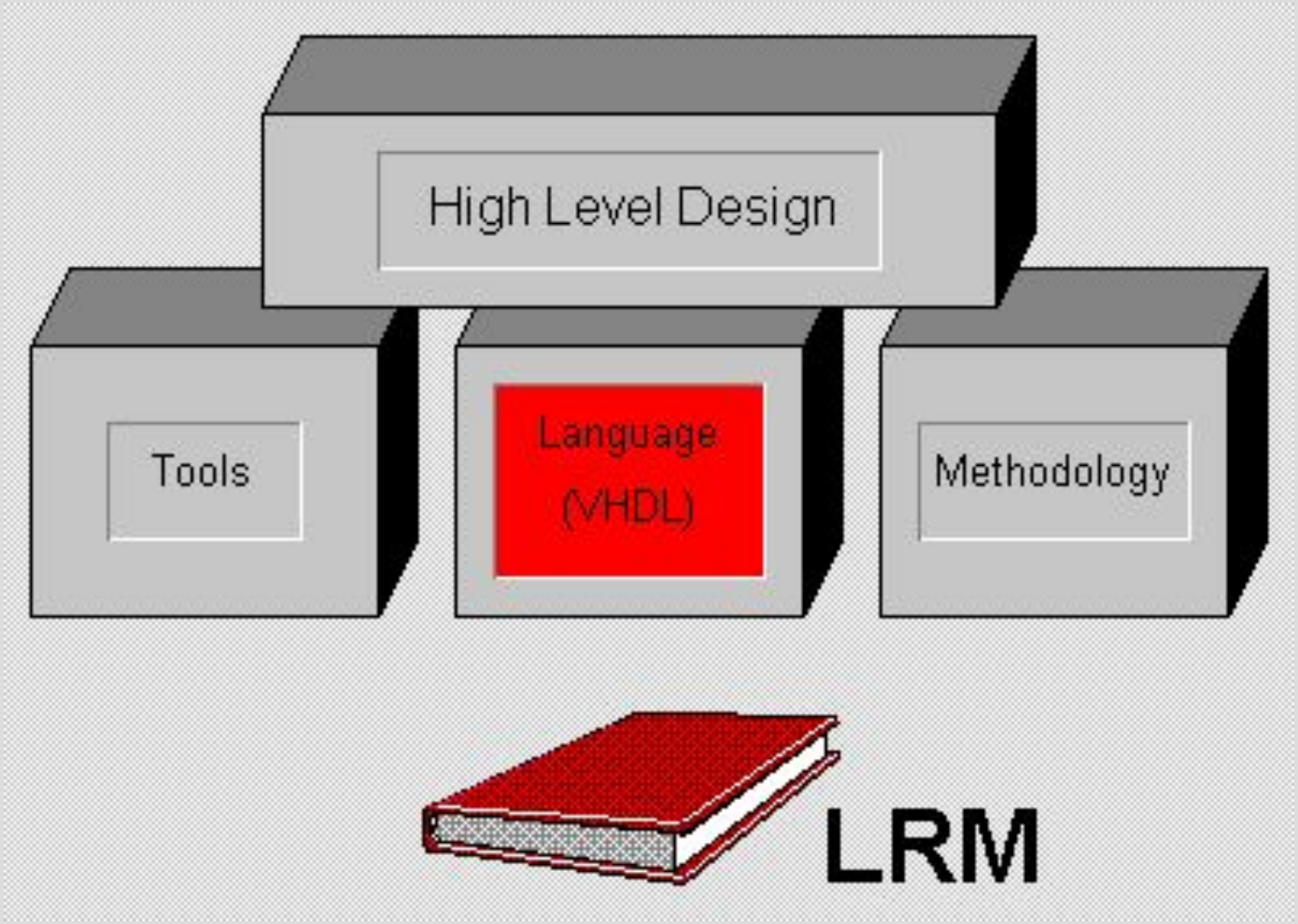
Язык разработан Министерством обороны США в 1983 г., а в 1987 г. появился его стандарт IEEE Std 1076-1987. Дальнейшее усовершенствование языка привело к пересмотру и расширению стандарта в 1993 г. и в 2001 г.

ЭТАПЫ разработки VHDL

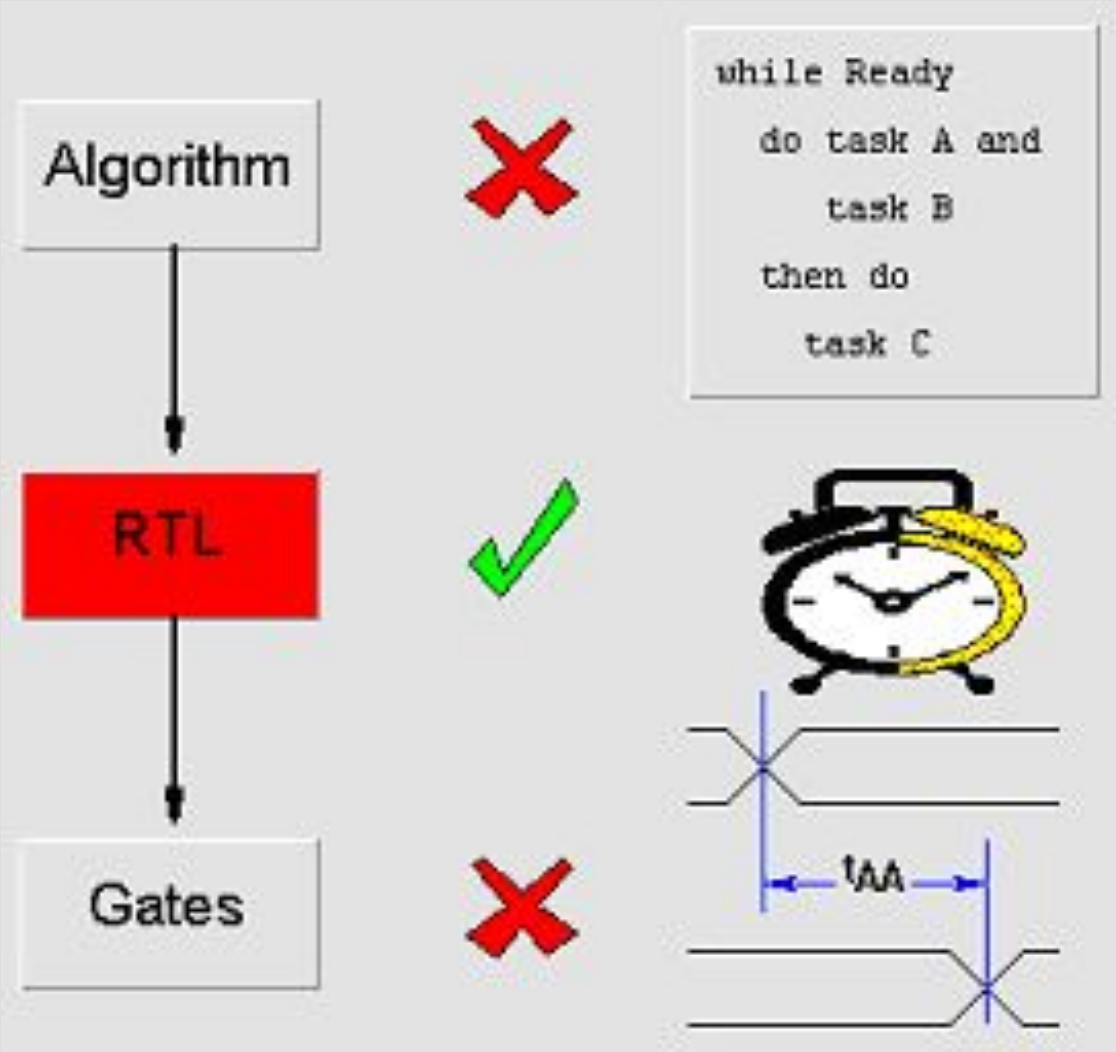
2001



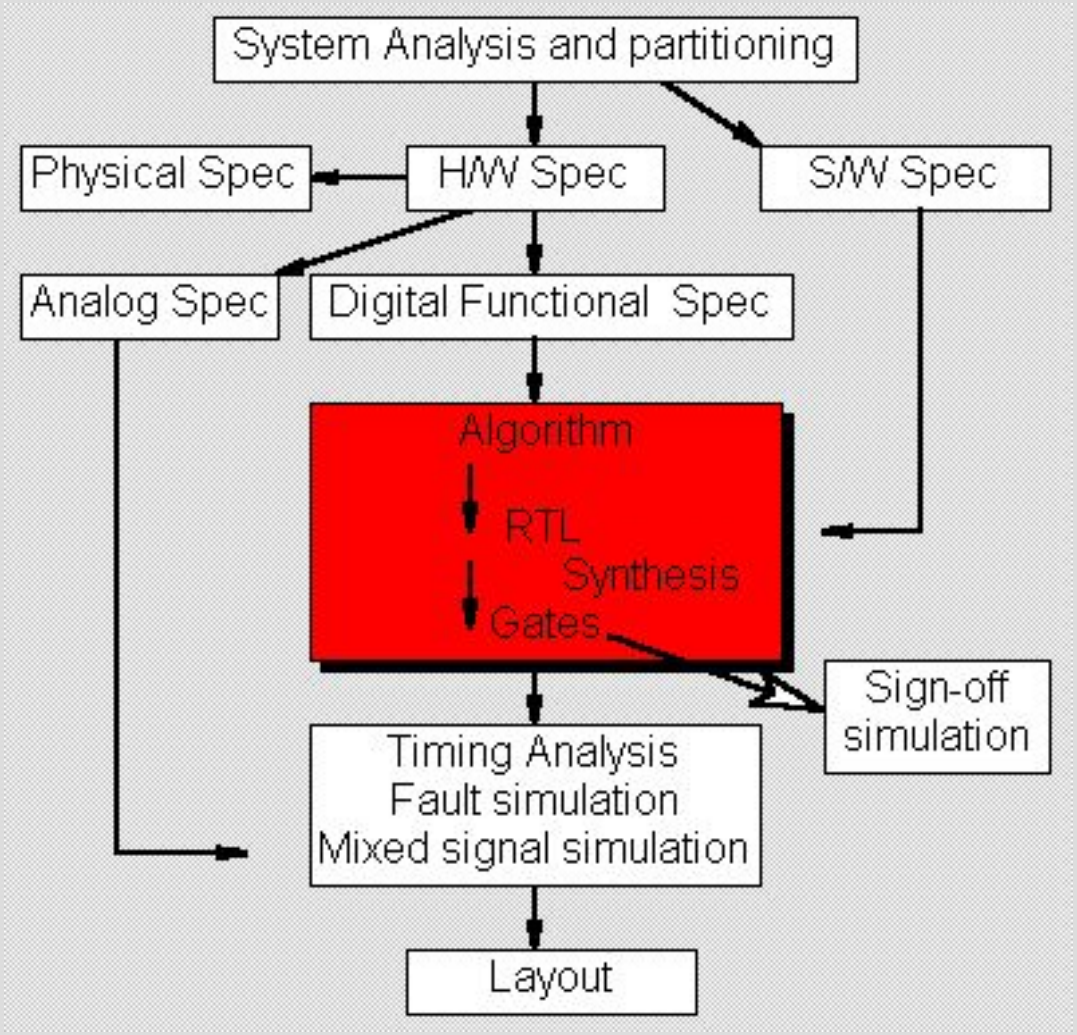
Рядом фирм разработаны методологии проектирования цифровых систем на основе языка VHDL.



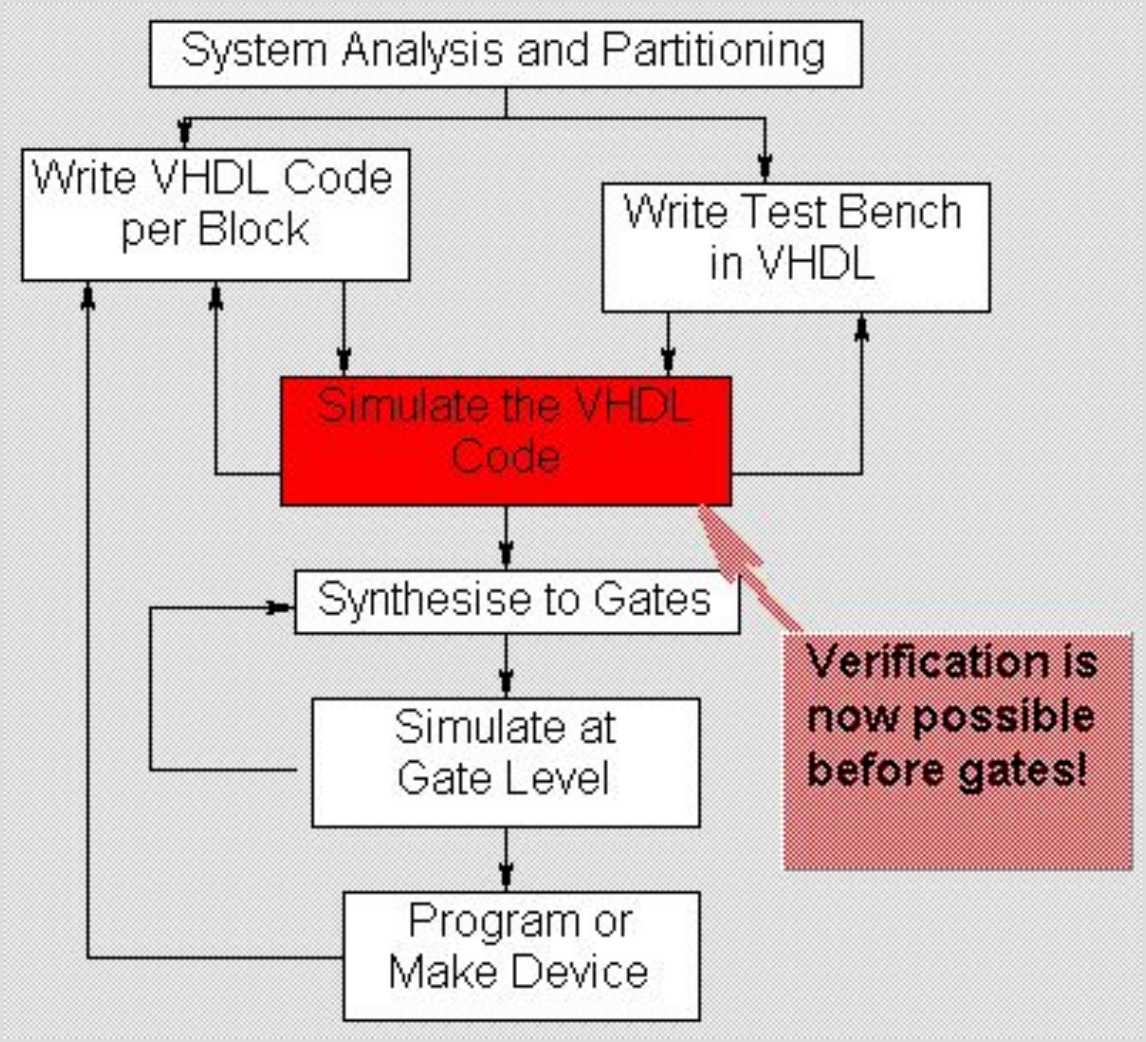
УРОВНИ АБСТРАКЦИИ системы



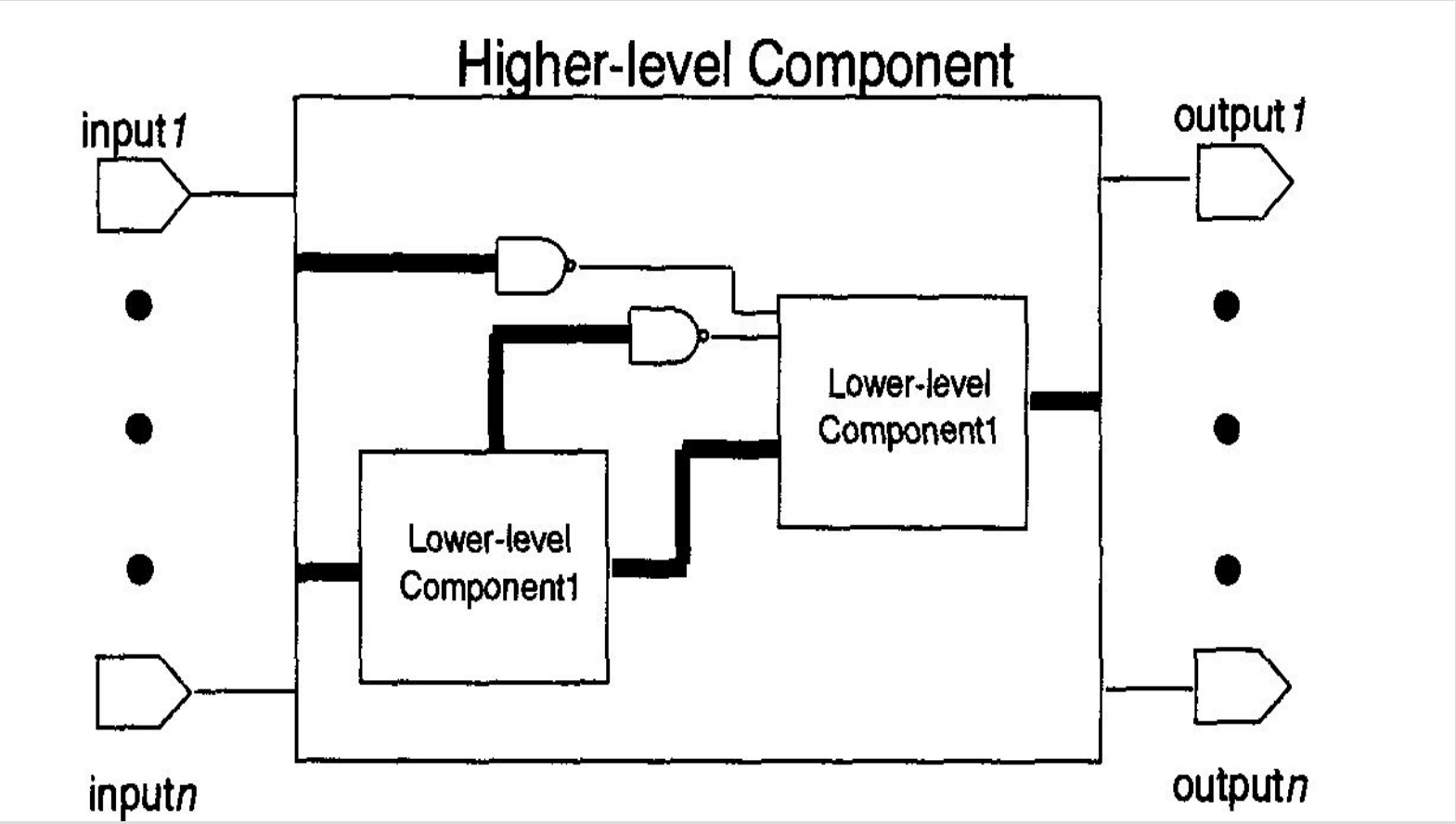
ОСНОВНЫЕ ЭТАПЫ ПРОЦЕССА ПРОЕКТИРОВАНИЯ



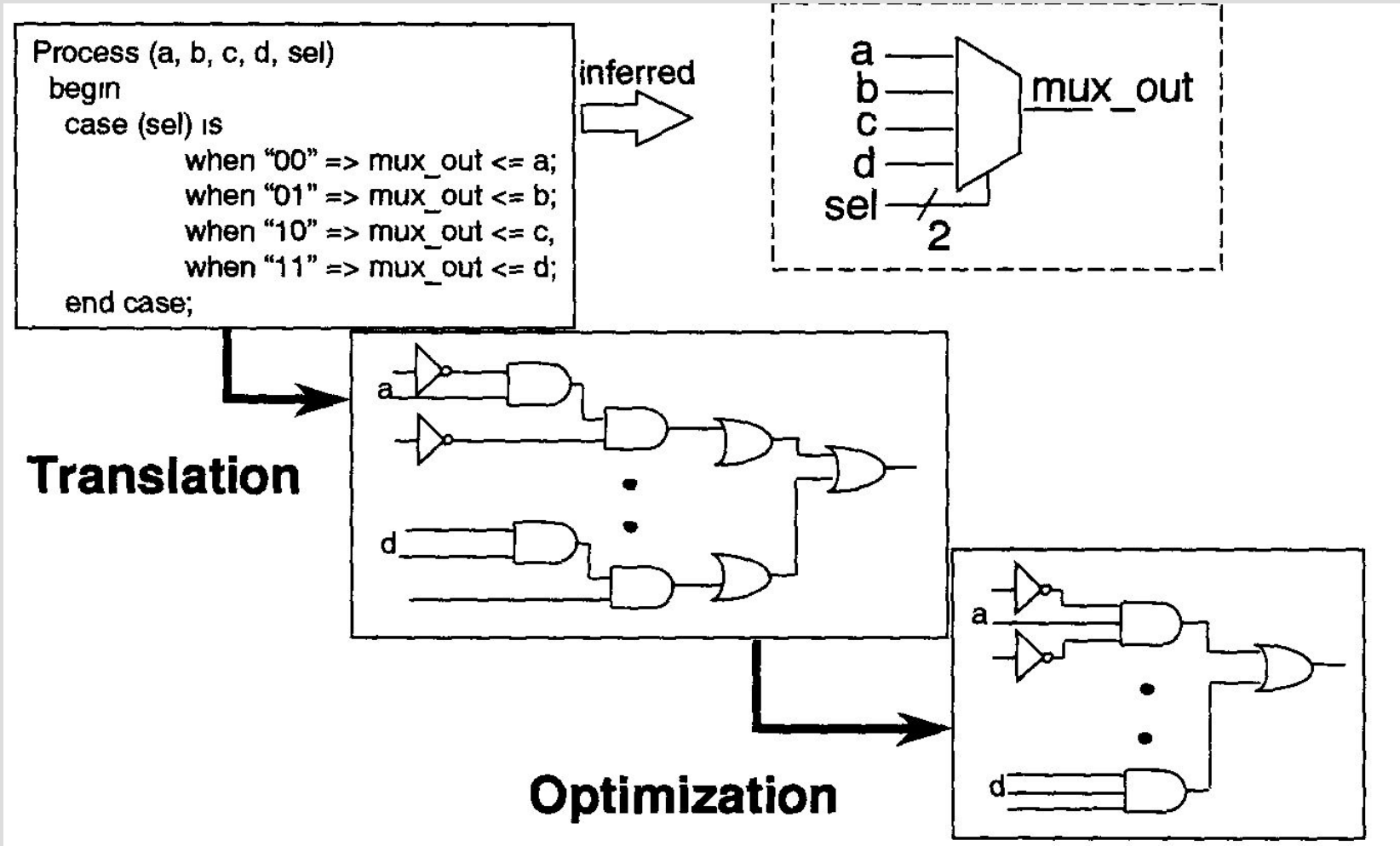
ИТЕРАЦИОННОСТЬ ПРОЦЕССОВ ПРОЕКТИРОВАНИЯ



ИЕРАРХИЧНОСТЬ ОПИСАНИЯ



RTL (Register Transfer Level) уровень межрегистровых передач



ОСОБЕННОСТИ VHDL

Проекты на VHDL ориентированы на любой инструментарий;
VHDL допускает создание проекта, не зависящего от технологии.

VHDL не ограничивает пользователя в стиле описания:

- VHDL позволяет описывать схемы, используя методологию «сверху вниз»,
- «снизу-вверх», либо из «середины наружу»!

VHDL можно использовать для описания схем на вентильном уровне (the gate level), либо более абстрактным способом.

ОБЩАЯ ХАРАКТЕРИСТИКА VHDL

!!! Построен на базе ключевых слов;

!!! НЕ РАЗЛИЧАЕТ в большинстве случаев прописные и строчные буквы;

!!! Выражения VHDL завершаются символом «точка с запятой» ;

!!! Не чувствителен к пробелам. Они используются для улучшения читаемости текста;

!!! Комментарии начинаются с двух стоящих рядом дефисов и занимают остаток строки;

!!! Модели VHDL могут быть:

- Behavioral (поведение)
- Structural (структура)
- Потоковые

Элементами проекта VHDL являются:

Entity (Используется для определения интерфейса модели, т.е. модели с точки зрения ее окружения)

Architecture (Используется для определения функционирования модели)

Configuration (Используется для указания связи между элементами Architecture и Entity)

Package (Package Declaration и Package Body – Содержит набор сведений, к которым могут обращаться модели VHDL)

Элементами проекта VHDL являются:

Entity

Architecture

Configuration

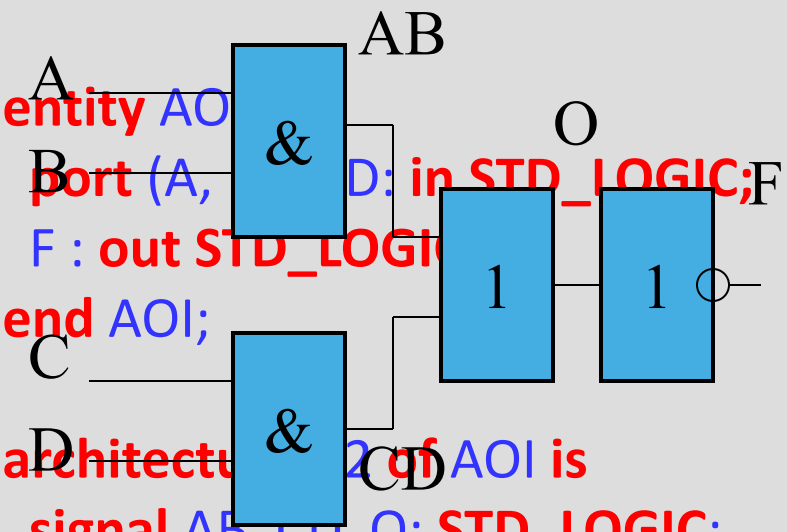
Package

Декларации объекта (entity declaration)
представляют внешний интерфейс объекта проекта.

Архитектурное тело (architecture body)
представляет внутреннее описание объекта проекта: его поведение, структуру, либо смесь обоих.

ПОТОКОВОЕ ОПИСАНИЕ:

```
use IEEE.STD_LOGIC_1164.all;
```



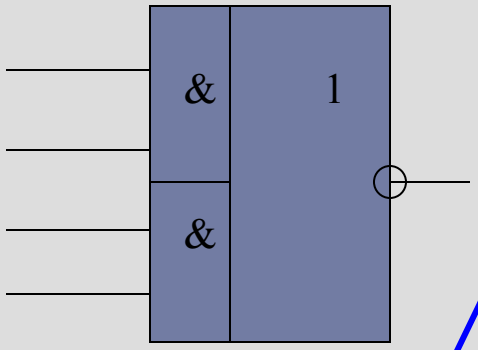

```
entity AOI
  port (A,      D: in STD_LOGIC;
  F : out STD_LOGI
end AOI;

architectu      of AOI is
  signal AB, CD, O: STD_LOGIC;
begin
  AB <= A and B after 2 NS;
  CD <= C and D after 2 NS;
  O <= AB or CD after 2 NS;
  F <= not O after 1 NS;
end V2;
```

Наряду с константами и переменными в языке VHDL существует понятие «сигнал». Сигнал – это модель линии связи.

AB, CD, O, F - сигналы

```
library IEEE;
```

ПРИМЕР ПРОЕКТА ОБЪЕКТА:



Комментарии

Строки контекста: подключение библиотек

Описание архитектуры (функций объекта)

```
-- VHDL code for AND-OR-INV.gate

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity AOI is
port (
  A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC
);
end AOI;

architecture V1 of AOI is
begin
  F <= not((A and B)or(C and D));
end V1;

-- end of VHDL code
```

Описание интерфейса

Сигналы

Архитектура с именем V2 содержит три сигнала AB, CD и O, используемые внутри архитектуры. Сигнал объявлен перед словом begin архитектуры и имеет свой собственный тип данных (например, STD_LOGIC).

Технически порты являются сигналами, поэтому сигналы и порты имеют много общего.

STD_LOGIC

```
entity AOI  is
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end AOI;
```

- Тип данных порта определяет множество значений, которые могут проходить через порт.

- Порты объявлены типа STD_LOGIC, который находится в пакете STD_LOGIC_1164 библиотеки IEEE.

- Пакет STD_LOGIC_1164 является стандартом IEEE для представления логических сигналов в VHDL .

Алфавиты моделирования

```
CONSTANT and_table : stdlogic_table := (
--    ---------------------------------------------------
--    | U   X   0   1   Z   W   L   H   -     |  |
--    ---------------------------------------------------
      ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
      ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
      ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
      ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
      ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
      ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
      ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
      ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
      ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ) -- | - |
);
```

```
CONSTANT or_table : stdlogic_table := (
--    ---------------------------------------------------
--    | U   X   0   1   Z   W   L   H   -     |  |
--    ---------------------------------------------------
      ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ), -- | U |
      ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | X |
      ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 0 |
      ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | 1 |
      ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | Z |
      ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ), -- | W |
      ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | L |
      ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ), -- | H |
      ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ) -- | - |
);
```

```
CONSTANT not_table: stdlogic_1d :=
-- ------------------------------------------
-- | U   X   0   1   Z   W   L   H   - |
-- ------------------------------------------
   ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );
```

Алфавиты моделирования

```
TYPE std_ulogic IS
  ( 'U',  -- Uninitialized
    -- (начальная
еделенность)
   'X',  -- Forcing  Unknown (сильная
         --          неопределенность)
   '0',  -- Forcing  0 (сильный «0»)
   '1',  -- Forcing  1 (сильная «1»)
   'Z',  -- High Impedance (высокий
     --           импеданс)
   'W',  -- Weak    Unknown (слабая
                 -- неопределенность)
   'L',  -- Weak    0  (слабый «0»)
   'H',  -- Weak    1 (слабая «1»)
   '-'   -- Don't care (безразлично)
    );
```

```
CONSTANT resolution_table :
    stdlogic_table := (
--  ---------------------------------------------------
--  | U   X   0   1   Z   W   L   H   -     |  |
--  ---------------------------------------------------
( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), --  | Z |
( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ),--   | W |
( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
);
```

Библиотека LIBRARY STD

содержит следующие встроенные пакеты:

Standard (Types: Bit, Boolean, Integer, Real, Time. Все функции для поддержки типов данных)

TEXTIO (File operations)

*****Они не требует объявления в проекте VHDL !!!**

Все операторы в архитектурном теле выполяются параллельно!!!

Архитектура содержит параллельное присваивание сигналов, которое описывает функцию объекта проекта.

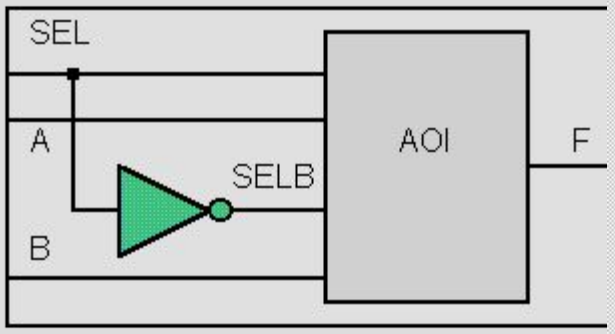
Параллельные предложения выполняются тогда, когда порты A, B, C или порт D изменяют значения!!!

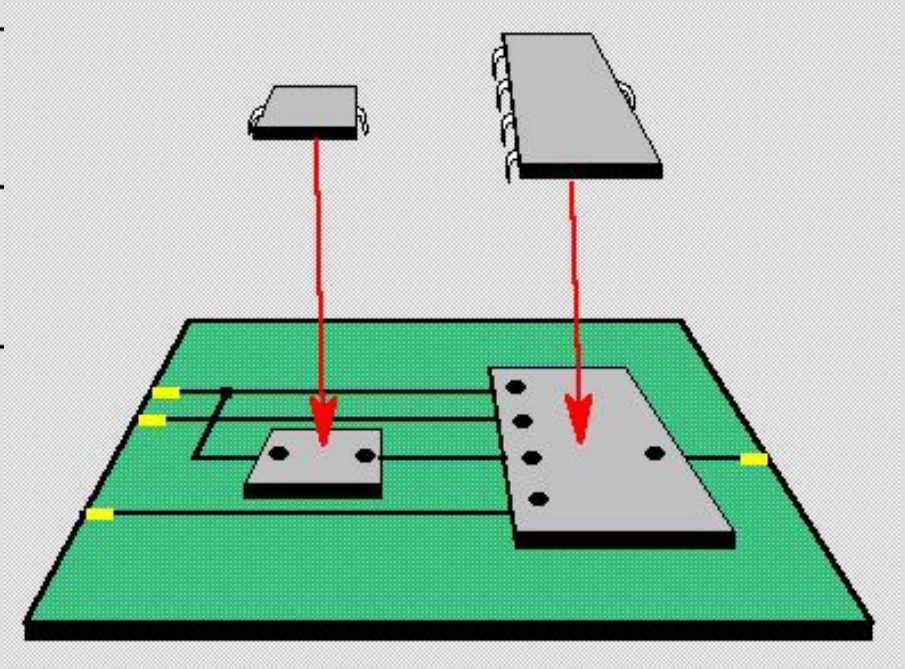
```
architecture V2 of AOI is
  signal AB, CD, O:
STD_LOGIC;
begin
  AB <= A and B after 2 NS;
  CD <= C and D after 2 NS;
  O <= AB or CD after 2 NS;
  F <= not O after 1 NS;
  end V2;
```

Структурное описание объекта (Структурное моделирование, моделирование на структурном уровне)

Структурное описание объекта

MUX2





Необходимо описать на языке VHDL объект, который представляет собой некоторую взаимосвязь компонентов.

Для описания структуры необходимо описать на VHDL используемые компоненты и применить их описание при описании всей схемы с помощью оператора **COMPONENT**

Компонентами являются **AOI** и **INV**.
Моделируемая структура **MUX2** состоит из данных компонент.
Компонента объявляется внутри архитектуры один раз, но можно создавать много экземпляров компоненты.
Предполагается, что эти Компоненты описаны, т. е. для них есть свои блоки **entity** и **architecture** .

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
      entity MUX2 is
      port (SEL, A, B: in STD_LOGIC;
      F : out STD_LOGIC);
      end;
architecture STRUCTURE of MUX2 is
 component INV
  port (A: in STD_LOGIC;
  F: out STD_LOGIC);
end component;
 component AOI
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;
 signal SELB: STD_LOGIC;
 begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```

Архитектура **STRUCTURE** of MUX2 создает ***экземпляры***, помеченные как G1 и G2. Имена компонент (INV и AOI) ссылаются на объекты проекта, определенные в другом месте.

Метки экземпляров (G1 и G2) определяют два конкретных экземпляра компонент и являются обязательными!

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
      entity MUX2 is
      port (SEL, A, B: in STD_LOGIC;
      F : out STD_LOGIC);
      end;
architecture STRUCTURE of MUX2 is
 component INV
  port (A: in STD_LOGIC;
  F: out STD_LOGIC);
end component;
 component AOI
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;
 signal SELB: STD_LOGIC;
 begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```

Порты в объявлении компонент должны совпадать один к одному с портами в объявлениях объекта.

Объявление компоненты определяет имена, порядок, режим и типы портов, которые будут использоваться, когда будет *создан экземпляр компоненты* в теле архитектуры.

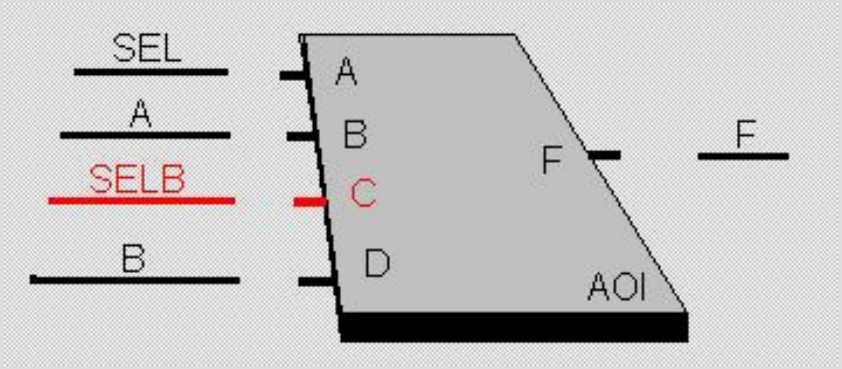
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
      entity MUX2 is
      port (SEL, A, B: in STD_LOGIC;
      F : out STD_LOGIC);
      end;
architecture STRUCTURE of MUX2 is
 component INV
  port (A: in STD_LOGIC;
  F: out STD_LOGIC);
end component;
 component AOI
port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;
 signal SELB: STD_LOGIC;
 begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```

Создание экземпляров компонент делает возможным создание иерархии проекта

по аналогии с тем, как вставляются чипы в печатную плату.

Отображение порта

```
architecture STRUCTURE of MUX2 is
...
component AOI
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;

signal SELB: STD_LOGIC;
begin
  ...
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```



Здесь компоненты *связываются по умолчанию*.

```
architecture STRUCTURE of MUX2 is
...
component AOI
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;

signal SELB: STD_LOGIC;
begin
  ...
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```

При связывании по умолчанию гнездо чипа (объявление компоненты) несет чип (проект объекта) с тем же именем (скажем, AOI), как мы описали выше.

Теперь в мире аппаратного обеспечения нет таких ограничений, имена сигналов гнезда не зависят от сигналов на выводах чипов. VHDL дает проектировщику ту же свободу.

Гнездо чипа и сам чип не должны теперь иметь одно имя, но для реализации такой возможности требуется блок ***конфигурация,*** который связывает соответствующий объект проекта с установлением экземпляра компонента. Рассмотрим теперь конфигурацию.

КОНФИГУРАЦИЯ

Описание VHDL может состоять из многих единиц проекта (каждое со своим архитектурным телом), организованных в иерархию проекта.

Конфигурация выполняет описание **точного набора объектов и архитектур, используемых при моделировании или синтезе конкретного объекта.**

Конфигурация позволяет установить связи между объявлением компоненты, объектом проекта и экземпляром компоненты.

Конфигурация выполняет две функции:

Во первых, определяет объекты проекта, используемые вместо каждого экземпляра компоненты.

Во вторых, определяет архитектуру, используемую для каждого объекта проекта (т.е. какой кристалл).

Конфигурация по умолчанию

Приведем минимальную конфигурацию для объекта верхнего уровня MUX2. Эта конфигурация выбирает используемую архитектуру – **STRUCTURE.**

По умолчанию все компоненты внутри архитектуры будут конфигурированы, чтобы использовать объект проекта с тем же именем, что и компонента (т.е. AOI) и последнюю анализируемую архитектуру каждого объекта проекта.

Конфигурация по умолчанию для MUX2

```
use WORK.all;
configuration MUX2_default_CFG of MUX2 is
for STRUCTURE
                . . .
end for;
end MUX2_default_CFG;
```

ОБЩИЙ ВИД КОНФИГУРАЦИИ

--Конфигурация, заданная для MUX2

```
use WORK.all;

configuration MUX2_specified_CFG of MUX2 is
  for STRUCTURE
    for G2 :AOI
      use entity work.AOI(v1);
      --       архитектура v1, определенная для
       - -      объекта проекта AOI
    end for;
   end for;
end MUX2_specified_CFG;
```

ОБЩИЙ ВИД КОНФИГУРАЦИИ

В объявлении конфигурации для конфигурируемой архитектуры можно точно указать, какие компоненты составляют конечный объект проекта.
В примере, приведенном ниже, для экземпляра G2 внутри архитектуры STRUCTURE используется архитектурное тело V1 вентиля AOI. Здесь не будет сомнения в том, какая архитектура выбрана для моделирования, поскольку она конкретно задана в конфигурации как V1.
При использовании конфигурации MUX2_default_CFG можно изменить архитектуру AOI для целей моделирования путем перекомпиляции v1 после v2 – это также часто применяется при проектировании схем!

```
use WORK.all;
configuration MUX2_specified_CFG of MUX2 is
  for STRUCTURE
    for G2 :AOI
      use entity work.AOI(v1);
      --       архитектура v1, определенная для
      - -      объекта проекта AOI
    end for;
   end for;
end MUX2_specified_CFG;
```

ОБЩИЙ ВИД КОНФИГУРАЦИИ

configuration – это ключевое слово VHDL, за которым следует имя конфигурации. После ключевого слова **of** определяется объект проекта, который мы конфигурируем, **is** – это также ключевое слово. Вместе с этой первой строкой объединена последняя, содержащая **end** и имя конфигурации:

```
configuration MUX2_specified_CFG of MUX2 is
  -- block configuration
end MUX2_specified_CFG;
```

Конфигурация обычно состоит из вложенных элементов конфигурации, объединенных предложением for. Первый элемент – это спецификация архитектуры.

```
for architecture_name
  -- thus, for STRUCTURE
```

второй – спецификация экземпляра

```
for instance_label : component_name
  -- thus,  for G2 :AOI  и  индикация связывания:
use entity library_name.entity_name(architecture_name)
  -- thus,  use entity work.AOI(v1);
```

Векторные порты и сигналы

Порт может описывать не только один электрический провод, сигнал на котором может нести информацию типа ***BIT*** или ***STD_LOGIC***, но также много проводов, по каждому из которых передаются сигналы определенного типа
(шинная структура порта).
В примере ***MUX4*** порт ***SEL*** есть 2-битовая шина со старшим разрядом номер 1 и младшим разрядом номер 0. Тип порта есть ***STD_LOGIC_VECTOR,*** этот тип определяется в пакете ***STD_LOGIC_1164*** библиотеки ***IEEE.*** Объекты типа ***STD_LOGIC_VECTOR*** – это просто массив объектов типа STD_LOGIC.

```
component MUX4
   port (SEL :in STD_LOGIC_VECTOR(1 downto 0);
   A, B, C, D:in STD_LOGIC;
   F :out STD_LOGIC);
 end component;
```

Testbench

Testbench для MUX4

```
entity TEST_MUX4 is
end;
library IEEE;
use IEEE.STD_LOGIC_1164.all;
architecture BENCH of TEST_MUX4 is
 component MUX4
  port (SEL :in STD_LOGIC_VECTOR(1 downto 0);
  A, B, C, D:in STD_LOGIC;
  F :out STD_LOGIC);
 end component;
 signal SEL: STD_LOGIC_VECTOR(1 downto 0);
 signal A, B, C, D, F: STD_LOGIC;
begin
 SEL <= "00", "01" after 30 NS, "10" after 60 NS,
  "11" after 90 NS, "XX" after 120 NS,
  "00" after 130 NS;
 A <= 'X', '0' after 10 NS, '1' after 20 NS;
 B <= 'X', '0' after 40 NS, '1' after 50 NS;
 C <= 'X', '0' after 70 NS, '1' after 80 NS;
 D <= 'X', '0' after 100 NS, '1' after 110 NS;
 M: MUX4 port map (SEL, A, B, C, D, F);
end BENCH;
```

1

2

3

4

1- обязательный блок entity;
2- строки контекста – подключение библиотеки;
3- описание тестируемой компоненты;
4- описание входных воздействий;
5- архитектурное тело блока *Testbench*

Testbench для MUX4

```
Entity TEST_MUX4 is
…
architecture BENCH of TEST_MUX4 is
  …
signal SEL: STD_LOGIC_VECTOR(1 downto 0);
  …
begin
  SEL <= "00",
         "01" after 30 NS,
         "10" after 60 NS,
         "11" after 90 NS,
         "XX" after 120 NS,
         "00" after 130 NS;
  …
  M: MUX4 port map (SEL, A, B, C, D, F);
```

Объявление объекта для testbench

entity TEST_MUX4 is ... end;

как правило, пусто. Это потому, что **testbench** сам по себе не имеет никаких входов и выходов.

Тестовые векторы генерируются и применяются к тестируемому устройству внутри **testbench.**

Заметим, что не правильно иметь тело архитектуры без объявления объекта.

Общий вид Testbench для MUX4

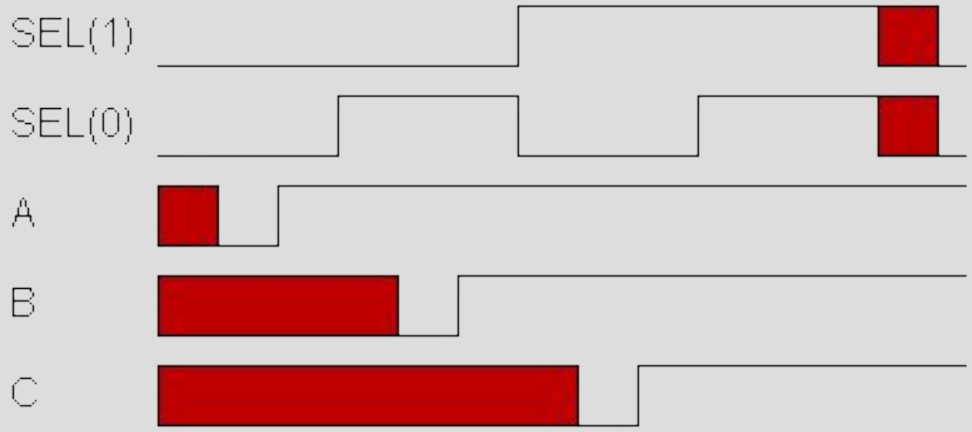
```
entity TEST_MUX4 is
End;
library IEEE;
use IEEE.STD_LOGIC_1164.all;

architecture BENCH of TEST_MUX4 is
  component MUX4
    ...
  end component;
  -- signals
begin
  -- присваивание сигнала для создания
воздействия
  M: MUX4 port map (...);
end BENCH;
```

В **testbench** пять операторов присваивания сигналов, которые определяют векторы входных воздействий. Например,

A <= 'X', '0' after 10 NS, '1' after 20 NS;

Задержка в этих предложениях присваивания относится к моменту времени, когда происходит присваивание (т.е. time 0), но не относительно друг друга (например, сигнал A изменится в '1' в момент 20 NS, а не в 30 NS).

Формы сгенерированных сигналов SEL, A, B и C



Особенности структурного описания объектов

Соединение компонент вместе – это один из методов проектирования с использованием VHDL. Это подход снизу-вверх (восходящее проектирование). Здесь мы определяем, какое поведение проектируемого объекта и пытаемся его аппаратно реализовать.

```
begin
   G1: INV port map (SEL, SELB);
   G2: AOI port map (SEL, A, SELB, B, FB);
   G3: INV port map (FB, F);
```

Однако зачастую при проектировании СБИС применяется метод нисходящего проектирования. При таком подходе проектируемый объект вначале представляется на уровне его поведения. Описание структуры объекта получается в результате работы автоматического синтезатора.

Описание объекта на уровне поведения

Определив концептуально, ***чего мы хотим от объекта***, мы можем на VHDL описать это в виде программы, никак не привязанной к аппаратуре разрабатываемого объекта.

В данном случае мы концентрируем внимание на том, **«что должен делать объект»,** а не **«как он это должен делать».** Задача по реализации объекта в аппаратуре передается на этап синтеза:

Процессы против компонентов

При описании функций проектируемого объекта на уровне поведения вместо установления экземпляра компоненты в архитектуре, мы создаем экземпляр процесса.

STRUCTURE

```
architecture STRUCTURE of MUX2 is
  -- signal & component
  -- declarations...
begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B,
F);
  G3: INV port map (FB, F);
end STRUCTURE;
```

BEHAVIOUR

```
architecture BEHAVIOUR of MUX2 is
  -- signal declarations
  -- (no components!)...
 begin
  ONLY_ONE: process
    -- software-style
    -- VHDL for
    -- the MUX_2 design
  end process;
end BEHAVIOUR;
```

Порядок выполнения операторов

- Для описания функциональных свойств объекта процесс может содержать ряд операторов, в том числе, операторы присваивания, которые *выполняются в заданной последователь-ности.*
- Вместо компонентов в примере MUX_2 мы можем использовать один или два процесса.
- Процессы не должны существовать в изоляции. *Процесс* является конкурентным, *выполняемым параллельно, предложе-нием внутри тела архитектуры*, почти как экземпляр компо-ненты.
- Мы знаем, что компоненты могут быть соединены вместе с помощью сигналов, то же касается и процесса.

Итак, процессы выполняются конкурентно по отношению друг к другу, однако, внутри них предложения выполняются последовательно.

Главные особенности блока процесса

- Внутри архитектуры можно создавать несколько процессов.
- Однако, экземпляр компоненты не является последовательным предложением, поэтому компоненту нельзя выполнить внутри процесса.
- И нельзя включить один процесс в другой таким же образом, как включали экземпляры компонент один в другой. Поэтому не существует иерархии процессов.
- Чтобы построить иерархию проекта, нужно использовать компоненты.

Сопоставление структурного и поведенческого описания

```
architecture STRUCTURE of MUX2 is

  -- signal and component declarations...

begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B, F);
End STRUCTURE of MUX2 ;
```

Описание объекта на структурном уровне

Описание объекта на уровне поведения

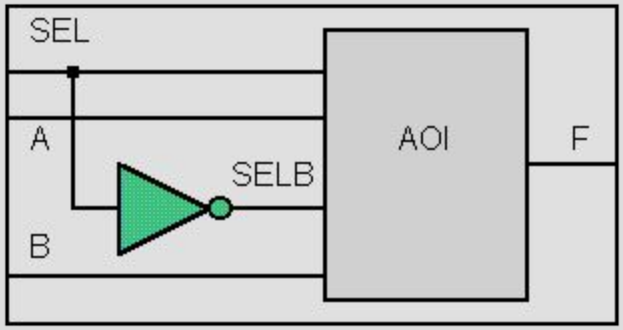
```
component

component
```

```
architecture BEHAVIOUR of MUX2 is
-- signal declarations (no components!)...
begin
 G1: process
-- software-style VHDL for the INV

end process;

 G2: process
-- software-style VHDL for the AOI

end process;

end BEHAVIOUR of MUX2 ;
```

Список чувствительности

- В VHDL процесс содержит последовательные предложения.
- Процессы разрешены только внутри архитектуры.
- Предложения внутри процесса выполняются последовательно, а не конкурентно.
- Процессы могут быть записаны различными способами.
- Наиболее общий способ применения процессов для описания проектирования – это использование формата, включающего **список чувствительности**.

Список чувствительности



Список чувствительности – это те входные переменные, которые поступают на моделируемый объект.

```
v1_arch: process (A, B, C, D)
begin
  F <= not ((A and B) or (C and D));
end process;
```

В проекте MUX_2 мы можем использовать порты и сигналы проекта:

```
G2: process (SEL, A, SELB, B)
begin
  F <= not ((SEL and A) or (SELB and B));
end process;
```

Аналогично и для инвертора:

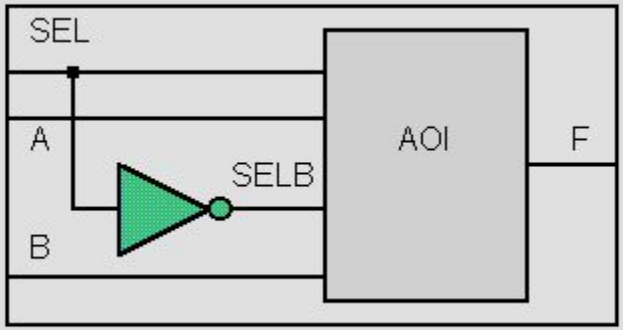
```
  G1: process (SEL)
  begin
     SELB <= not SEL;
  end process;
```

Мы можем скомбинировать эти два процесса в один:

```
combined: process (SEL, A, B)
begin
   F <= not ((SEL and A) or
((not SEL) and B));
end process;
```

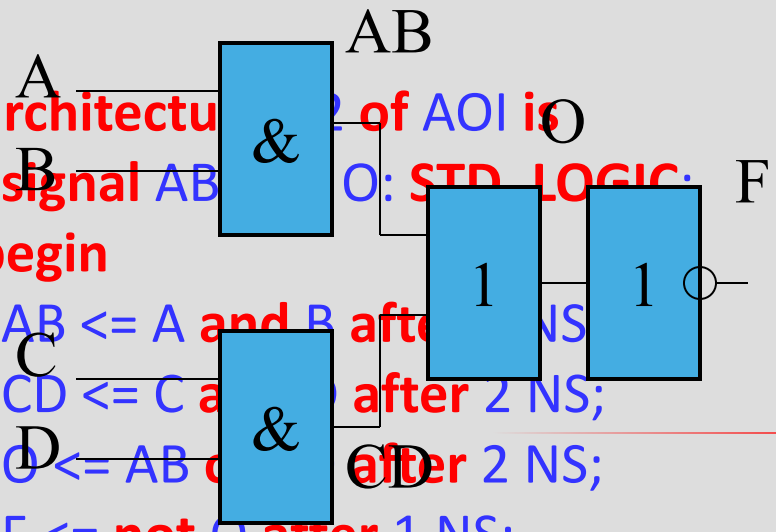
Возможные методы описания MUX2

```
// описание объекта в виде «потока данных»
  selb <=  not sel;
  fb <= not((a and sel) or (b and selb));
  f <= not fb;

// описание объекта в виде структуры элементов (логической сети)
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SELB, A, SEL, B, FB);
  G3: INV port map (FB, F);

// описание объекта на уровне поведения

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

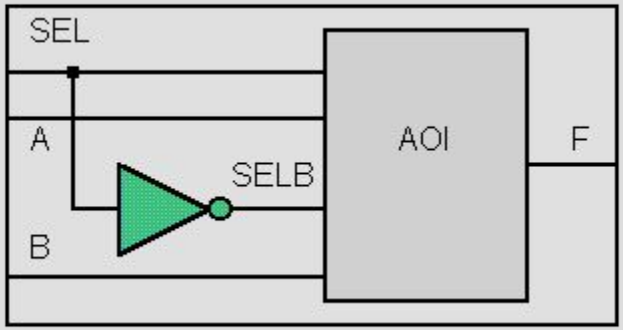


ПОТОКОВАЯ МОДЕЛЬ:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity AOI  is
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end AOI;

architectu    of AOI is
  signal AB     O: STD_LOGIC;
begin
  AB <= A and B aft     NS
  CD <= C a       after 2 NS;
  O <= AB o       after 2 NS;
  F <= not O after 1 NS;
end V2;
```



СТРУКТУРНАЯ МОДЕЛЬ:




```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
        entity MUX2 is
        port (SEL, A, B: in STD_LOGIC;
        F : out STD_LOGIC);
        end;
architecture STRUCTURE of MUX2 is
 component INV
  port (A: in STD_LOGIC;
  F: out STD_LOGIC);
end component;
 component AOI
  port (A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC);
end component;
 signal SELB: STD_LOGIC;
 begin
  G1: INV port map (SEL, SELB);
  G2: AOI port map (SEL, A, SELB, B, F);
end;
```

Testbench для MUX4

```
entity TEST_MUX2 is
end;
library IEEE;
use IEEE.STD_LOGIC_1164.all;
architecture BENCH of TEST_MUX2 is
 component MUX2
  port (SEL, A, B  :in STD_LOGIC);
  F :out STD_LOGIC);
 end component;
 signal SEL, A, B, SELB, F : STD_LOGIC;

begin
 SEL <= “0", “1" after 30 NS, “0" after 60 NS,
  “1" after 90 NS, “0" after 120 NS,
  “1" after 130 NS;
 A <= ‘1', '0' after 10 NS, '1' after 20 NS;
 B <= ‘1', '0' after 40 NS, '1' after 50 NS;


M: MUX4 port map (SEL, A, B, F);
end BENCH;
```

1

2

3

4

1- обязательный блок entity;
2- строки контекста – подключение библиотеки;
3- описание тестируемой компоненты;
4- описание входных воздействий;
5- архитектурное тело блока *Testbench*

VHDL –
строго типизированный язык

- Тип - это множество значений с общим признаком.
- Каждый объект объявляется со своим типом и может присваивать значение только данного типа.
- Семантика простых типов данных понятна уже из их названий:

✔***INTEGER;***
✔**REAL ;**
✔**BIT (со значениями 0 и 1);**
✔**BOOLEAN (со значениями TRUE и FALSE);**
✔**BIT_VECTOR;**
✔**POSITIVE (положительные целые числа);**
✔**NATURAL (натуральные числа);**
✔**CHARACTER (символы кода ASCII);**
✔**STRING (строка символов).**

Определение основных типов данных

Основные типы данных VHDL определены в пакете ***STANDARD***, который ***подключается к проекту по умолчанию***.
Вот так в нем определены некоторые стандартные типы объектов:

```
type boolean is (false, true);
type bit is ('0', '1');
type integer is range -2147483647 to 2147483647;
subtype natural is integer range 0 to 2147483647;
type bit_vector is array (natural range <>) of bit;
```

ПЕРЕЧИСЛИМЫЙ ТИП

type идентификатор is (список значений)

```
type octal_digits is ('0','1','2','3','4','5','6','7');
type color is ("red","green","blue");
```

Определение основных типов данных

Физический тип

представляется целым числом, единица которого имеет вес единицы измерения некоторой физической величины.

В VHDL используется один физический тип - предопределенный физический тип TIME (время).

Объявление физического типа задает множество единиц, определенных в терминах некоторой базовой единицы.

В случае типа TIME **базовой единицей является fs (фемтосекунда),** а производными единицами являются **ps, ns, µs** и так далее. Рассмотрим определение типа TIME.

```
type TIME is range -(2**31-1) to 2**31-1
units
fs;
ps = 1000 fs;
ns = 1000 ps;
µs = 1000 ns;
ms = 1000 us;
s = 1000 ms;
min = 60 s;
hr = 60 min;
end units;
```

```
type resistance is range 0 to
2**31-1
units
nOhm;
uOhm =1000 nOhm;
mOhm =1000 uOhm;
Ohm =1000 mOhm;
kOhm =1000 Ohm;
megOhm =1000 kOhm;
end units;
```

Объекты языка VHDL

Сигнал

Сигналом является объект, который переносит значение от одного процесса к другому и вместе с ним - синхронизирующее воздействие. История сигнала может быть запомнена и воспроизведена в симуляторе в виде графика или таблицы. Объявление сигнала выглядит как:

signal : \идентификатор\{,\идентификатор\} :\тип\ [:=\начальное значение\];

где - \начальное значение\ - выражение, представляющее константу, значение которой принимает сигнал перед первым запуском процесса.

Пример объявления сигнала:

signal reg: BIT_VECTOR := ‘1001100101’;
signal R1: std_logic:=‘0’;

Константа

Константой является объект, не изменяющий свое значение при вычислениях. После объявления константы присваивание ей значения запрещено (кроме случая отложенной константы).
Константы в VHDL описываются следующим образом:

constant *имя*: *тип* := *значение*;

Примеры объявления константы:

constant thousand: integer:=1000;

Переменная

Переменной является объект, хранящий значение в пределах операторов процесса, функции или процедуры. В отличие от сигнала, присваивание переменной выполняется немедленно.
Упрощенный синтаксис объявления переменной:

\объявление переменной\::=
[shared] variable
\идентификатор\{,\идентификатор\}:\тип\[:=\начальное значение\];

Пример объявления переменной:

variable tmp: integer range -128 to 127:=0;

Типы портов

Упрощенный синтаксис объявления портов объекта проекта следующий:

\объявление портов\::=port (\объявление порта\ {; \объявление порта\});

\объявление порта\::=\идентификатор\: in /out/inout/buffer/link

```
entity MUX2 is
port (SEL, A, B: in STD_LOGIC;
F : out STD_LOGIC);
end;
```

in – прием

out – передача

inout - прием и передача

buffer - передача и использование как сигнал-операнд внутри объекта проекта,

link – двунаправленное соединение с другим портом с таким же режимом.

Настроечная константа generic

Настроечная константа **generic** кодирует определенное свойство
объекта проекта.
Она используется, например, для задания разрядности линий
связи, задержки и др.
Упрощенный синтаксис :

\объявление настроечных констант\::= generic(\объявление настроечной константы\ {; \объявление настроечной константы\}); \объявление настроечной константы\::= \идентификатор\:\тип\[:=\начальное значение\]

ПРИМЕР

generic (T1: Time := 20ns; T2: Time := 5 ns; numb: integer := 12);

Настроечная константа

generic

Из примера видно, что в декларации port перечисляются формальные сигналы, а в port map - фактические сигналы. При этом последовательность перечисления в обоих местах должна быть согласована.

```
entity schema is
port (a,b,c,d,e: in BIT; y: out BIT);
end schema;
architecture str of schema is
   component AND_OR
   generic (delay1: Time);
   port (i1,i2,i3,i4: in BIT; a: out BIT);
   end component  AND_OR;
component OR2
generic (delay2: Time);
port (i1,i2: in BIT; a: out BIT);
end component OR2;
   signal z1,z2: BIT;
begin
E1: AND_OR
generic map (delay1 := 4 ns);
port map (a,c,b,c,z1);
        E2: AND_OR
        generic map (delay1 := 5 ns);
        port map (d,c,e,c,z2);
E3: OR2
generic map (delay2 := 3 ns);
port map (z1,z2,y);
end str;
```

Начальное значение объекта

Начальное значение объекта в его объявлении - это то значение которое принимает объект перед первым циклом моделирования.
Если начальное значение не присвоено, то симулятор присваивает наименьшее значение данного типа, если тип - числовой или самое левое значение, если тип – перечисляемый.

U - это самое левое значение перечисляемого типа **STD_LOGIC**

Операции в выражениях

Тип операции	Символ или ключевое слово
Логические	and, or, nand, nor, xor, xnor
Сравнения	=, /=, <, <=, >, >=
Сдвига	sll, srl, sla, sra
Сложения ...	+, -, & (конкатенация)
Унарные (знак)	+, -
Умножения ...	*, /, mod, rem
Другие	**, abs, not

Операции в выражениях (прдлж)

- Логические операции имеют самый низкий приоритет.
- Операнды логических операций должны быть одного типа (одномерного или регулярного типа – векторы из элементов).
- Для однозначной компиляции логических выражений необходимо использовать скобки, например:
  (a **or** b) **and** C **and** (d **or** c).
- ***Операции сравнения*** выполняются над операндами одинакового типа и возвращают тип boolean;
- ***Операции равенства*** "=" и неравенства "/=" выполняются над всеми типами;
- Остальные операции сравнения выполняются над перечисляемыми типами, целыми типами и одномерными регулярными типами (векторами) из элементов такого типа.

Операции в выражениях (прдлж)

- ***При сравнении перечисляемых типов*** элемент, стоящий в ряду правее (старший), считается большим.

- ***При сравнении векторов*** сравниваются пары элементов векторов, начиная с самых левых. Если пара элементов неодинакова, то вектор с более старшим элементом считается большим. Если пара элементов одинакова то рассматривается следующая пара элементов.

  ПРИМЕР:

  При сравнении векторов битов "0111" >="01011" результат будет true.

- ***Операции сдвига*** выполняют сдвиг вектора битов на число разрядов типа integer.

  ПРИМЕР:

  Результатом выражения ("100110" sra 3) является вектор "111100", т.е. происходит арифметический сдвиг вправо на 3

Операции в выражениях (прдлж)

- ***Операции сложения - вычитания*** "+", "-" предопределены для целых чисел и чисел с плавающей запятой.
- ***Операция конкатенации "&"*** применяется со всеми одномерными регулярными типами или с их элементами. С помощью этой операции векторы - операнды объединяются в более длинные векторы.

  ПРИМЕР:

  Выражение **"101" & '1' & "10"** даст результат **"101110".**
- ***Унарные операторы:*** Оператор минус инвертирует значение операнда.
- ***Операторы умножения*** **"*"**, деления **"/"** применяются к целым операндам и операндам с плавающей запятой.
- ***Операторы mod (модуль), rem (остаток***) применяются к целым числам.
- ***Операции абсолютного значения*** **abs** ***и возведения в степень*** **"**"** определены для целых чисел и чисел с плавающей запятой, причем показатель степени должен быть целым.

ПОСЛЕДОВАТЕЛЬНЫЕ ОПЕРАТОРЫ

sequence_of_statements::= {sequential_statement}

sequential_statement::=

wait_statement

| **assertion**_statement

| **signal_assignment**_statement

| **variable_assignment**_statement

| **procedure_call** statement

| **if**_statement

| **case**_statement

| **loop**_statement

| **next**_statement

| **exit**_statement

| **return**_statement

| **null**_statement

Оператор ожидания wait

вызывает приостановку процедуры или оператора процесса

wait_statement::=
[label:] **wait** [sensitivity_clause][condition_clause] [timeout_ clause];
sensitivity_clause::=**on** sensitivity_list (указывается список чувствительности)
sensitivity_list::=signal_name{ ,signal_name}
condition_clause::=**until** condition (условие ожидания)
condition::=boolean_expression
timeout_clause::=**for** time_expression (задержка ожидания)

Список чувствительности определяет набор сигналов, к которым чувствителен оператор ожидания.

Условие ожидания определяет условие, встретив которое процесс продолжит свое выполнение.

Задержка ожидания (timeout) определяет максимальное время, в течение которого процесс будет оставаться приостановленным на данном операторе ожидания

Оператор ожидания wait (продолжение)

При выполнении оператора ожидания ***вычисляется значение*** в выражении времени для определения ***времени приостановки***. Приостановленный процесс будет возобновлен немедленно ***после окончания интервала приостановки*** (timeout interval) простоя.

Приостановленный процесс ***может также восстанавливаться*** в результате события от любого сигнала в наборе чувствительности оператора ожидания. При появлении такого события вычисляется условие в операторе условия.

ПРИМЕРЫ:

1) По оператору **wait on CLK, RST;** продолжение выполнения процесса начнется по событию изменения сигналов **CLK** или **RST.**

2) По оператору **wait until CLK='1';** продолжение начнется в момент переключения состояния **CLK** из '0' в '1', т.е. по фронту этого сигнала.

3) Оператор **wait for CLK_PERIOD;** остановит процесс на время, заданное переменной **CLK_PERIOD** типа time.

Оператор ожидания wait (продолжение)

*****Возможно комбинирование списка чувствительности, условия ожидания и задержки ожидания в одном операторе.

*****Оператор wait без списка чувствительности, условия ожидания и задержки ожидания ***остановит процесс до конца моделирования.***

Оператор утверждения assert

Оператор утверждения **assert** проверяет, является ли истиной определенное условие, и уведомляет об ошибке, если нет. Оператор введен в язык VHDL *для выявления ошибок моделирования* и сообщения о них пользователю.

assertion_statement::= [label :] assertion;

assertion :: = **assert** condition [**report** expression] [**severity** expression]

Выполнение оператора утверждения состоит в вычислении булева выражения, определяющего условие.

Если выражение имеет результат FALSE (ложь), то говорят, что произошло ***нарушение утверждения.***

Когда происходит нарушение утверждения, то оцениваются выражения операторов сообщения ***report*** и строгости ***severity*** соответствующего утверждения, если они присутствуют .

Оператор утверждения assert (продолжение)

assertion_statement::= [label :] assertion;
assertion :: = **assert** condition [**report** expression] [**severity** expression]

Затем указанная строка сообщения и уровень строгости (или соответствующие значения по умолчанию, если они отсутствуют) используются для формирования сообщения об ошибке.

Оператор сообщения **report** определяет строковое сообщение, которое будет включено в сообщения об ошибках, формируемые оператором. При отсутствии оператора сообщения **report** для данного утверждения, значением по умолчанию для строкового сообщения является строка ***Assertion violation*** (Нарушение утверждения).

ПРИМЕР 1: Если требуется остановить моделирование, можно записать оператор:

assert 1/=1 report "конец моделирования " severity failure;

Оператор утверждения assert (продолжение)

assertion_statement::= [label :] assertion;
assertion :: = **assert** condition [**report** expression] [**severity** expression]

ПРИМЕР 2:

Если не нужно ловить ошибку, а только вывести сообщение о ходе моделирования, то применяют оператор сообщения с синтаксисом:

report \строка сообщения\ [severity \выражение\];

Предыдущий пример можно переписать как:

report "конец моделирования " severity failure;

Оператор утверждения assert (продолжение)

assertion_statement::= [label :] assertion;
assertion :: = **assert** condition [**report** expression] [**severity** expression]

Наиболее частое применение оператора assert – **проверка соответствия входных сигналов**, поступающих через порты, заданным требованиям или соответствие ограничениям на настроечные константы **generic.**
Например, проверяется время **предустановки сигнала относительно фронта синхросигнала, времени его удержания, разрядность входных данных и т.п.** При несоответствии сигналов или настроечных констант, оператор assert выдает сообщение об ошибке.

Применение оператора утверждения assert (прдлж)

Рассмотрим пример объявления объекта RS-триггера:

```
entity RS_FF is
   generic (delay:time);
   port(R, S: in bit;
        Q: out bit:='0';
        nQ: out bit:='1');
begin
     assert (R and S) /='1' report “In RS_FF R=S=1" severity error;
end entity RS_FF;
```

При единичных сигналах на обоих входах, т.е. когда **RS** – триггер функционирует неправильно, оператор **assert** выдает сообщение об ошибке.

Применяемые типы задержек

signal_assignment_statement

При моделировании дискретных систем важное место занимает моделирование распространения сигнала с учетом задержки в проводниках или задержки в вентилях. Для этого используют следующий расширенный синтаксис присваивания сигналу:

\присваивание сигналу\::=
\приемник\ <= [\способ задержки\] \график\;
\способ задержки\::=
transport | [reject \выражение времени\] inertial
\график\::= \выражение\ [after \выражение времени\] {, \выражение\ [after \выражение времени\] }

Здесь график (waveform) представляет собой запись, состоящую из одной или нескольких пар:

величина сигнала – задержка сигнала.

В первой паре задержка может не указываться, подразумевается, что она нулевая.

Присваивание значения сигналу при моделировании

Способ задержки **inertial** реализует поведение задержки в источнике сигнала, который не реагируют на слишком короткие входные импульсы. При этом фразой **reject** задается минимальная ширина импульса, которая выдается источником. Если этой фразы нет, то минимальная ширина импульса задается в фразе **after.** ***По умолчанию в операторе применяется способ задержки*** **inertial.**

ПРИМЕРЫ:

Y<= reject t_rej inertial A and B after t_d;

моделирует вентиль "И" с задержкой t_d , который не пропускает импульсы короче t_rej.

Y<= X after 10 ns;

значение сигнала X на момент запуска процесса присвоится сигналу Y с задержкой 10 нс, при этом импульсы шириной менее 10 нс будут подавлены.

Присваивание значения сигналу при моделировании

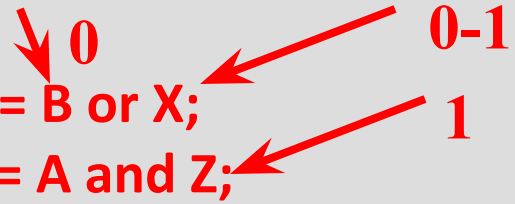
Y<= '0', '1' after 10 ns, '0' after 20 ns, '1' after 30 ns;
сигналу Y сначала присвоится 0, через 10 нс – 1, через 20 нс – 0, и через 30 нс – 1.

Y<= A, A+B after delay_sum;
сигналу Y сначала присвоится A, а через задержку, определяемую статическим выражением delay_sum – сумма сигналов A и B.

Y<= transport X after 1000 ns;
модель линии задержки сигнала X на 1 мкс.

Сравнение операторов присваивания := и назначения сигнала <=

Особенность оператора <= в том, что присвоение значений сигналам всегда происходит с задержкой, большей нуля. Поэтому использование операторов присваивания и назначения сигналов в одной и той же ситуации может дать разные результаты. ПРИМЕР:



Пусть B = 0, Z=1 и в рассматриваемый момент времени X переключается из 0 в 1. После выполнения этих двух операторов C равно 1. В случае

A <= B or X;

C <= A and Z; -- используется старое значение A.

Если старое значение A есть 0, то сигнал C в данный момент времени равен 0.

ОРМА БЭКУСА-НАУРА

Джон Бэкус (англ. *John Backus*, 3 декабря 1924 — 17 марта 2007) — американский учёный в области информатики. Он был руководителем команды, разработавшей первый высокоуровневый язык программирования ФОРТРАН, изобретателем формы Бэкуса — Наура, одной из самых универсальных нотаций, используемых для определения синтаксиса формальных языков.

Он был удостоен в 1977 премии Тьюринга за глубокий, важный и долгоживущий вклад в разработку практических высокоуровневых программных систем, особенно в виде работы над ФОРТРАНОМ, и за основополагающие публикации по формальным процедурам спецификации языков программирования

ФОРМА БЭКУСА-НАУРА –

VHDL описывается посредством контекстно-независимого синтаксиса на основе простого варианта записи Бэкуса – Наура, в частности:

Строчные слова прямым шрифтом, иногда содержащие подчеркивания, используются для обозначения синтаксических классов, например :

formal_port_list

Всякий раз, когда имя синтаксического класса используется отдельно от самих синтаксических правил, пробелы заменяются символами подчеркивания.

Слова полужирным шрифтом используются для того, чтобы обозначить зарезервированые слова, например:

array

Зарезервированые слова должны быть использованы только в местах, обозначенных синтаксически.

Синтаксическое выражение состоит из левой стороны (например символ ::= который читается: может быть заменен на), и правой стороны. Левая сторона правила вывода всегда есть синтаксический класс; правая сторона есть правило замены.

ФОРМА БЭКУСА-НАУРА –

Синтаксическое выражение подчиняется правилу текстовой замены: любой случай левой стороны может быть заменен на отдельный случай правой.

Вертикальная линия разделяет различные варианты выбора правой стороны, за исключением того случая, когда эта линия следует сразу за открытой скобкой:

letter_of_digit::=letter | digit

choices ::= choice { | choice }

В первом примере, “letter_or_digit” может быть заменен на “letter” или на “digit”. Во втором случае, “choices” может быть заменен на список из “choices” , разделенный вертикальной линией.

ФОРМА БЭКУСА-НАУРА -

формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик.

Квадратные скобки окаймляют необязательные элементы правой стороны выражения; таким образом, два следующих выражения эквивалентны:

return_statment ::= return [expression] ;

return_statment ::= return ; | return expression ;

ФОРМА БЭКУСА-НАУРА

Скобки окаймляют повторный элемент или элементы правой стороны выражения. Элементы могут фигурировать как ни одного, так и несколько раз; повторения встречаются слева направо.

Поэтому следующие два выражения эквивалентны:

term ::= factor { multiplying_operator factor}

term ::= factor | term multiplying_operator factor

Если имя какой-либо синтаксической категории начинается курсивом, то это эквивалентно категории имени без курсива. Курсив предназначен для передачи семантической информации. Например, *type*_name и *subtype*_name оба являются семантически эквивалентными только в плане имени.

Оператор case

\оператор case\::=case \простое выражение\ is
when \альтернативы\ => {\последовательный оператор\}
{when \альтернативы\ => {\последовательный оператор\}}
end case ;
\альтернативы\:= \альтернатива\{ | \альтернатива\}

Разрешает выполнение одной из цепочек последовательных операторов в зависимости от значения выражения селектора.

ПРИМЕР:

```
variable sel, a: integer 0 to 9;
.............
case sel is
when 0 => a <= 0;
when 1|2|3 => a <= 1;
when 4 to 7  => a <= 2;
when  others  => a <= 3;
end case;
```

Охранные выражения блоков

Рассмотрим архитектурное тело **addl_e** одноразрядного сумматора в виде охраняемого блока.

```
entity addl_e is
port (bl,b2,enable : in BIT;
cl,sl : out BIT);
end addl_e;
architecture struct_3 of addl_e is begin
p0: block (enable = '1')
begin
sl<= guarded (bl xor b2);
cl<= guarded (bl and b2);
end block p0;
end struct_3;
```

Охранное выражение

Охраняемый блок

Охранные выражения блоков

Охранным выражением блока является выражение **enable = 1.** Если это выражение принимает значение true (истина), то охраняемые конструкции (назначения сигналов) выполняются, т. е. одноразрядный сумматор складывает числа, если же значение выражения является false (ложь), то охраняемые назначения сигналов не выполняются, т. е. сумматор не складывает числа bl, b2. Охрана назначения сигналов осуществляется указанием ключевого слова **guarded.**

Охранные выражения блоков

Пример описания D-триггера с асинхронным сбросом в виде блока с охранным выражением (clk = «1» or clr = «1»)

```
entity dlatch is
port ( D,clk, clr : in bit;   Q : out bit);
end dlatch;
architecture functional of dlatch is
Begin
      P: block (clk ='I' or clr = '1')
                  begin
         Q <= guarded '0' when clr = '1'
                else D  when clk = '1'
                else  unaffected;
      end block P;
end functional;
```

В данном примере clk - вход синхронизации, clr - асинхронный сброс, D – информационный вход, Q - выход триггера.

Когда охранное выражение (clk = '1' or clr = '1') имеет значение *ложь,* то сигнал Q в левой части сохраняет свое прежнее значение.

Легко видеть, что сигнал асинхронного сброса имеет приоритет по отношению к сигналу clk.

Ключевое слово **unaffected** употребляется в операторе условного назначения сигнала для случая, когда требуется, чтобы назначаемый сигнал (в примере сигнал Q) не изменял своего значения.

Атрибуты

- Атрибутом называют особенное, долговременное свойство предмета.
- В языке VHDL сигналы, переменные и другие объекты, кроме своего значения, также имеют множество атрибутов.
- У каждого типа объектов есть несколько предопределенных атрибутов.
- Пользователь также может ввести ряд специальных атрибутов.
- **Атрибуты бывают различного типа: атрибут – тип, значение, сигнал, функция, диапазон.**

Семантика:

\имя объекта\' \имя атрибута

Атрибуты сигналов

\имя объекта\' \имя атрибута

S'stable[(T)] – сигнал, равный **true**, если за промежуток времени **T** не было событий у сигнала **S**.
S'transaction – сигнал типа **bit**, меняет значение на противоположное в циклах моделирования, в которых было присваивание нового значение сигналу S.
S'event – сигнал, равный **true**, если произошло событие в сигнале **S** в данном цикле моделирования.
S'active – сигнал, равный true, если произошло присваивание нового значение сигналу **S** в данном цикле моделирования.
S'last_value – сигнал такого же типа, что и **S**, содержащий значение **S** до последнего события в нем.

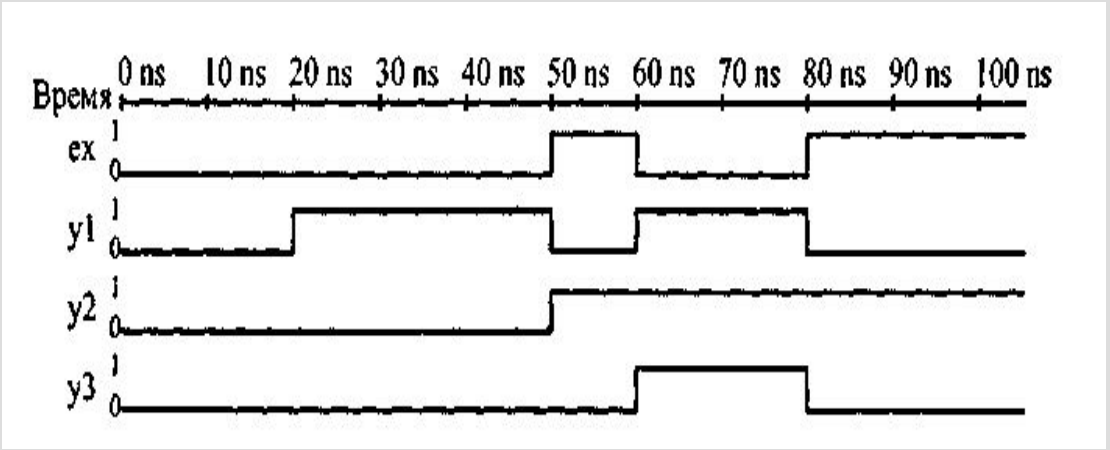
Атрибуты сигналов

```
entity signal_ex is
end signal_ex;
architecture beh of signal_ex is
signal ex, y 1, y3 : bit;
signal y2 : boolean;
begin
ex <= '0' after 20 ns,
‘1’  after 50ns,
 '0'   after 60 ns,
 ‘1’ after 80ns;
yl <= ex'transaction;
y 2 <= ex'event;
y3 <= ex'last_value;
end beh;
```

ex'transaction

x'event

ex'last_value



Применение атрибутов при моделировании синхронных триггеров

```
…
  if CLK='1' and CLK'event  then-- D- триггер
      q1<=a;
  end if;
  if not CLK'stable then -- D- триггер
      q2<=a;
  end if;
  if CLK'last_value /= CLK then-- D- триггер
      q3<=a;
  end if;
  if CLK'active-- D- триггер
      q4<=a;
  end if;
 q5<=CLK'transaction; -- T- триггер
```

Атрибуты для скалярного типа

T'left – самое левое значение множества элементов скалярного типа T.
T'right – самое правое значение множества элементов скалярного типа T.
T'high – наибольшее значение в множестве элементов скалярного типа T.
T'low –наименьшее значение в множества элементов скалярного типа T.
T'image(X) – функция строкового представление выражения X типа T.
T'value(X) – функция значения типа T от строкового представления X.
T'pos(X) – функция номера позиции элемента X типа T.
T'val(X) –функция значения элемента типа T, стоящего в позиции X.

Атрибуты для скалярного типа

Примеры атрибутов:

```
type st is (one,two,three);
st'right = three,
st'pos(three) = 2,
st'val(1) = two.

positive'low =  1,
positive'high =2147483647.
integer'value("1_000") =1000,
integer'image(330) ="330".
```

Атрибуты для регулярного типа

A'left[(N)] – левое значение диапазона индексов по N-й размерности.
A'right[(N)] - правое значение диапазона индексов по N-й размерности.
A'high[(N)] - наибольшее значение диапазона индексов по N-й размерности.
A'low[(N)] - наименьшее значение диапазона индексов по N-й размерности.
A'range[(N)] – диапазон индексов по N-й размерности.
A'reverse_range[(N)] – обратный диапазон индексов по N-й размерности.
A'length[(N)] – протяженность диапазона индексов по N-й размерности.
A'ascending[(N)] - функция, равная **true**, если диапазон индексов по N-й размерности - возрастающий.

Атрибуты для регулярного типа

Примеры

type s2 is array(2 downto 1, 0 to 3) of integer;

s2'left(1) = 2,
s2'right(2) = 3,
s2'high(1) = 2,
s2'low(2) = 0,
s2'range(2) = 0 to 3,
s2'reverse_range(1) = 1 to 2,
s2'length(2) = 4.

Атрибуты пользователя

Эти атрибуты предназначены для присваивания объектам языка дополнительных свойств, не предусмотренных встроенными типами и атрибутами.

ПАРАЛЛЕЛЬНЫЕ (СОВМЕСТНЫЕ) ОПЕРАТОРЫ

Параллельные операторы используются для определения связанных блоков или процессов, которые сообща описывают общее поведение или структуру проекта. Параллельные операторы выполняются асинхронно по отношению друг к другу.

```
concurrent_statement::=
        block_statement
      | process_statement
      | concurrent_procedure_call
      | concurrent_assertion_statement
      | concurrent_signal_assigument_statement
      | component_instantiation
      | generate_statement
```

Оператор процесса

Упрощенный синтаксис:

\оператор процесса\ ::=[**postponed**] **process**
[(\имя сигнала\ {,\имя сигнала\})] [**is**] {\объявление в процессе\}
begin
{\последовательный оператор\}
end process;

Объявленными в процессе могут быть:
объявление и тело подпрограммы,
объявление типа и подтипа,
Объявление константы, переменной, файла,
объявление и спецификация атрибута, объявление группы,
описание **use**.

То, что объявлено в процессе, имеет область действия (видимость), ограниченную данным процессом!!!

Оператор процесса

- **Все процессы в программе выполняются параллельно.**
- **Сигналы нельзя объявлять в процессах**.
- **Процесс невозможно поместить в процесс, так как там есть место только для последовательных операторов.**
- **Процессы обмениваются сигналами, которые выполняют синхронизацию процессов и переносят значения между ними.**
- **Если над сигналами определена функция разрешения, то выходы источников сигнала могут объединяться.**

В круглых скобках заголовка процесса указывается множество сигналов, по которым процесс запускается – **список чувствительности**. Это форма оператора процесса альтернативная процессу с оператором **wait on**, стоящим последним в цепочке последовательных операторов тела процесса. Любой процесс со списком чувствительности может быть преобразован в эквивалентный процесс с оператором **wait on**, стоящим последним в списке последовательных операторов

Оператор процесса

Ключевым словом **postponed** помечаются отложенные процессы.

Процесс представляет собой маленькую программу, которая выполняется на виртуальном процессорном элементе.

ПРИМЕР: Рассмотрим процесс, вычисляющий функцию sin(x) по аппроксимирующей формуле:

$$\sin(x) = c_1x+c_2x^3+c_3x^5+c_4x^7.$$

```
process
    type tabl is array(0 to 3) of real;
    constant c:tabl:=(0.99999, -0.16666, 0.00831, -0.00019);
    variable xtmp, p: real:=0.0;
begin
   xtmp:=x;
   p:=c(0)*xtmp;
   for i in 1 to 3 loop
        p:=p+c(i)*xtmp*x*x;
   end loop;
   y<=p;
```

Структура программы на VHDL

- Дискретная система может быть представлена в VHDL как объект проекта.
- Объект проекта описывается набором составных частей проекта, таких как:
  - объявление объекта называемое ***entity,***
  - тело архитектуры объекта (или просто архитектура), именуемое ***architecture***,
  - объявление пакета (package),
  - тело пакета (package body) и
  - объявление конфигурации (configuration).
- Каждая из составных частей объекта может быть скомпилирована отдельно. Составные части проекта сохраняются в одном или нескольких текстовых файлах с расширением **.VHD**. В одном файле может сохраняться несколько объектов проекта.

Объект проекта

\объект проекта\::= [\описание library\]
[\описание use\]
\объявление объекта\
\тело архитектуры\
[\объявление конфигурации\]

[\описание library\]::= library \идентификатор\ {, \идентификатор\};

Описание use:

Объекты проекта, объявленные в подпрограмме, процессе, пакете и объекте проекта видимы только в границах этих структурных единиц программы.

Для того, чтобы в данной структурной единице был видимый объект, объявленный в другом месте, используется описание **use.**

Синтаксис:

\описание use\::= use \селективное имя\ {, \селективное имя\ };

\селективное имя\::= \имя1\. \имя2\ \имя2\::= \идентификатор\ | \символьный литерал\ |all

Объект проекта

\описание use\::= use \селективное имя\ {, \селективное имя\ };
\селективное имя\::= \имя1\. \имя2\ \имя2\::= \идентификатор\ | \символьный литерал\ |all

Здесь \имя1\ представляет собой обозначение места, где находится объект, который должен быть видимым. Это идентификатор библиотеки и идентификатор пакета в ней, разделенные точкой.

Идентификатор – название объекта, который должен быть видимым, символьный литерал – символьное имя функции, например, "*".

Ключевое слово all означает, что видимы все объекты, объявленные в указанном месте.

Например, чтобы были видимы функции сложения и вычитания из пакета std_logic_arith библиотеки IEEE используют

use IEEE.std_logic_arith."-", IEEE.std_logic_arith."+" ;

а если все объявления из этого пакета должны быть видимыми то используют

use IEEE.std_logic_arith.all;

Объявление объекта

\объявление объекта\::= **entity** \идентификатор\ is
[**generic** (\объявление настроечной константы\
{; \объявление настроечной константы\});]

[**port** (\объявление порта\ {;\объявление порта\});]
{\объявление в объекте\}
[**begin**
{\оператор **assert**\ | \пассивный вызов процедуры\ |
\пассивный процесс\ }]
end [entity][\идентификатор\];
\объявление портов объекта\::=**port**
\объявление порта\::=\идентификатор\: **in**
|**out**|**inout**|**buffer**|**link** \тип\
[:=\начальное значение\] .

Начальное значение порта или настроечной константы

Начальное значение объекта в его объявлении - это то значение, которое принимает объект перед первым циклом моделирования. Если начальное значение не присвоено, то симулятор присваивает ***наименьшее значение данного типа***, если тип - числовой или самое ***левое значение***, если тип - перечисляемый.

Например, если тип STD_LOGIC, то начальное значение будет U - неинициализировано. Если при моделировании не предусматривается подача сигналов на порт такого типа, то этот порт лучше инициализировать, например, значением '0'.

Начальное значение порта или настроечной константы

Начальное значение может быть выражением.
Но значение выражения должно быть вычисленным до момента трансляции данного объявления.
Например, первое объявление порта:

```
port( bb:bit:=aa;
      aa:bit :='1');
```

неверно, так как при его рассмотрении компилятор еще не имеет сведений об идентификаторе aa.
В аппаратной модели начальное значение порта эквивалентно состоянию шины сразу после включения питания до прихода сигналов сброса, т.е. оно не определено.

Настроечные константы **generic** кодируют определенные свойства объекта проекта, например, разрядность линий связи, параметры задержки.

Исполнительная часть

```
entity RS_FF is
   generic(delay:time);
   port(R, S: in bit;
        Q: out bit:='0';
        nQ: out bit:='1');
begin
      assert (R and S) /='1' report "In RS_FF R=S=1" severity error;
end entity RS_FF;
```

В исполнительной части, которая открывается словом **begin,** могут вставляться параллельные операторы, которые не выполняют присваивания сигналам, т.е. не влияют на поведение объекта.

Исполнительная часть

Такие вызовы процедуры и процессы называются пассивными.

Наиболее частое применение этих операторов – **проверка соответствия входных сигналов, поступающих через порты, заданным требованиям** или соответствие включения объекта в окружение, задаваемое ограничениями на настроечные константы generic.

Например, проверяется **время предустановки сигнала** относительно фронта синхросигнала, соответствие его уровней, разрядность входных данных и т.п. При несоответствии сигналов или настроечных констант, оператор **assert** выдает сообщение об ошибке.

Оператор block

- В языке VHDL **блок представляет собой выделенное множество параллельных операторов**. Этот оператор, как и оператор процесса, является основным операторам языка VHDL.
- Все операторы вставки компонента в проекте можно заменить на эквивалентные операторы блока. Большой иерархический объект проекта можно представить одним объектом, в котором компоненты заменены эквивалентными блоками.
- Как и в других языках программирования, блоки в VHDL выполняют две основные функции:
  ----создание локальной памяти для сигналов
  ----введение обособленной области их действия (области видимости).
  Также блоки могут иметь иерархическое построение, т.е. могут применяться в блоках на более высоком уровне.

Оператор block

Ранее указывалось, что нельзя одному сигналу присваивать значение в разных процессах, если над типом сигнала не определена ***функция разрешения***. Для реализации корректного присваивания сигналу любого типа в разных процессах язык VHDL предоставляет операторы присваивания сигналу со сдерживанием.

Оператор блока обсудим на примере 7-разрядного сумматора, составленного из одного полусумматора и трех полных двухразрядных сумматоров.
Полный двухразрядный сумматор **adder_2p** есть каскадное соединение двух полных одноразрядных сумматоров **add2.**
Структурное описание схемы **adder_2p** приведено ниже. Схема **adder_2p** реализует операцию сложения **(c0) + (a2, al) + (b2, b1) = (c2, s2,sl)** одноразрядного числа c0 с двухразрядными числами a, b.

Структурное описание схемы adder_2p

```
entity adder_2p is
port (al, bl, a2,b2,c0 : in BIT;     --al, b1 - младшие разряды двухразрядных
c2,s2,sl : out BIT);                 -- складываемых чисел a = (a2, al), b = (b2, b1);
end adder_2p;                                    --a2, b2 - старшие разряды;
architecture structural of adder_2p is           --c0 -перенос из предыдущего
 component add2                                   -- разряда.
port (cl,al,a2:in BIT;
c2,s2:out BIT);
end component;
signal cl: BIT;
begin
circl: add2 port map (cl => c0, al =>al, a2 =>b1,c2 =>cl, s2=>sl);
circ2: add2 port map (cl => cl,  al => a2, a2 =>b2,c2 =>c2,s2=>s2);
end structural;
```

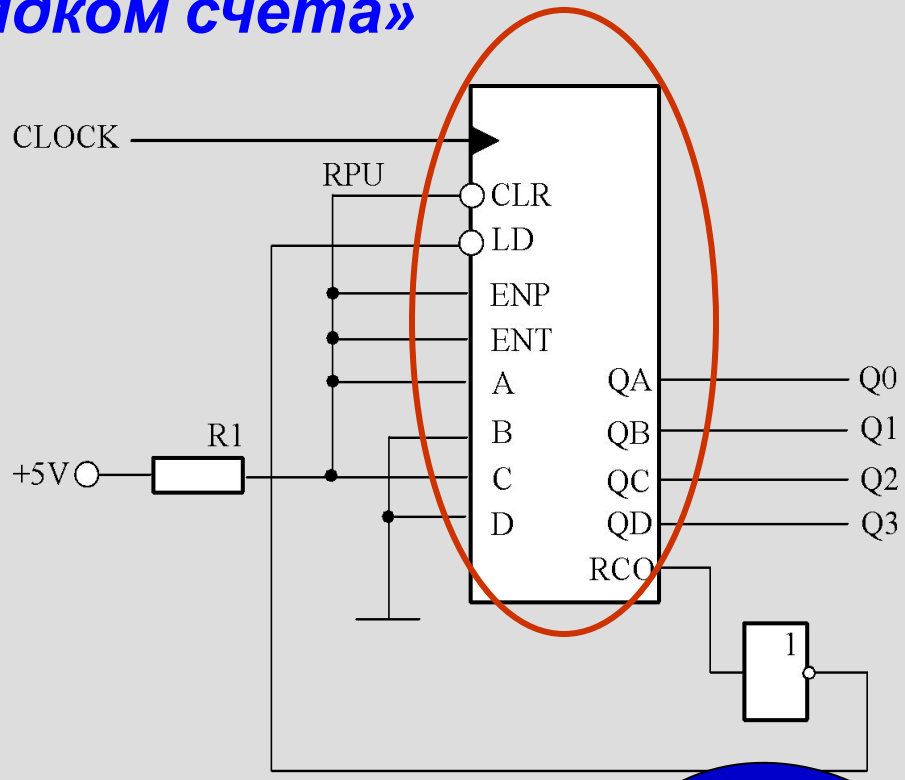
Структурное описание схемы adder_N_block

```
entity adder_N_block is
      port (a,b : in bit_vector (0 to 6); s : out bit_vector (0 to 6); c : out bit);
end adder_N_block;
architecture structural of adder_N_block is
component   addl
   PORT (bl,b2: in BIT; cl, sl: out BIT);
end component;
component adder_2p
  PORT(al,bl,a2,b2,c0: in BIT; c2,s2,sl: out BIT);
end component;
       signal  c_in : bit_vector (0 to 2);
       begin
       p0: add1 port map (b1 => a(0), b2 => b(0), cl=>c_in(0), sl=>s(0));
block0: block
begin
stage 1: adder_2p port map (c0 => c_in(0), al => a(l), bl => b(l), a2 => a(2), b2 => b(2),
c2 => c_in (l), s2 => s(2), s1 => s(l) );
stage2: adder_2p port map (c0 => c_in (l), al => a(3), bl => b(3),
a2 => a(4), b2 => b(4), c2 => c_in(2), s2 => s(4), s1 => s(3) );
end block;
stage3: adder_2p port map (c0 => c_in (2), al => a(5), bl => b(5),
a2 => a(6), b2 => b(6), c2 => c, s2 => s(6), s1 => s(5));
end structural;
```

Информация для выполнения проекта

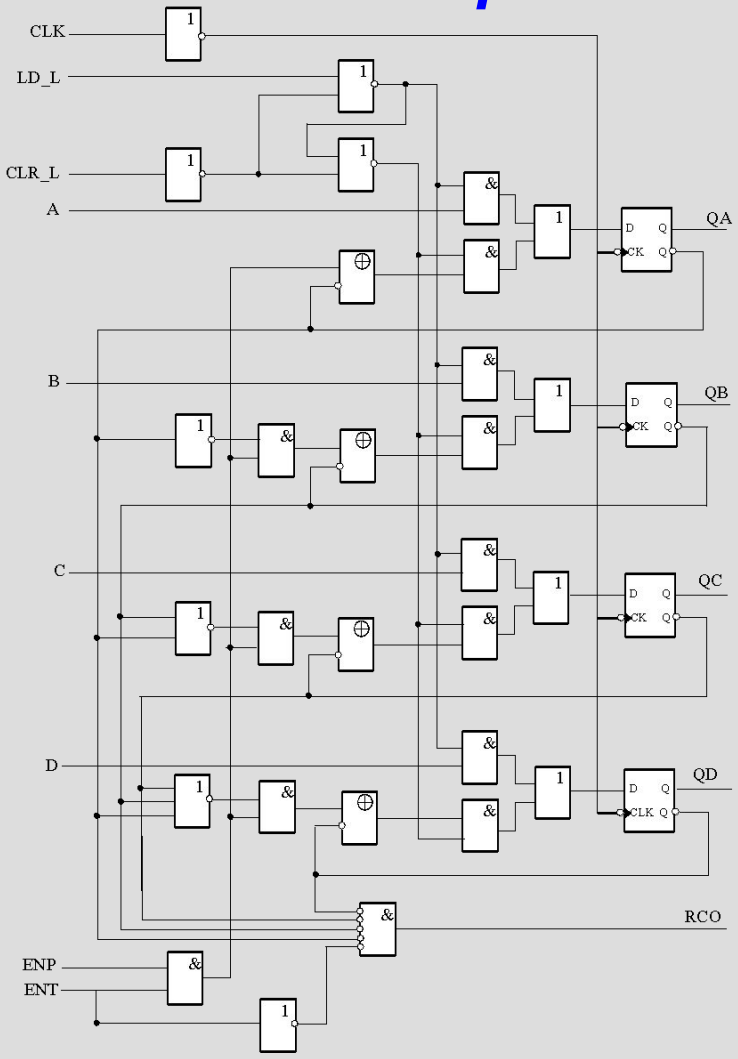
Разработка математической и программной модели цифрового объекта «Счетчик с принудительным порядком счета»

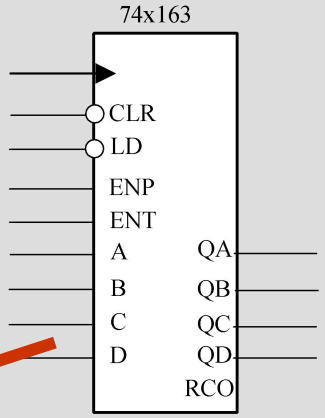
Иерархическое представление объекта



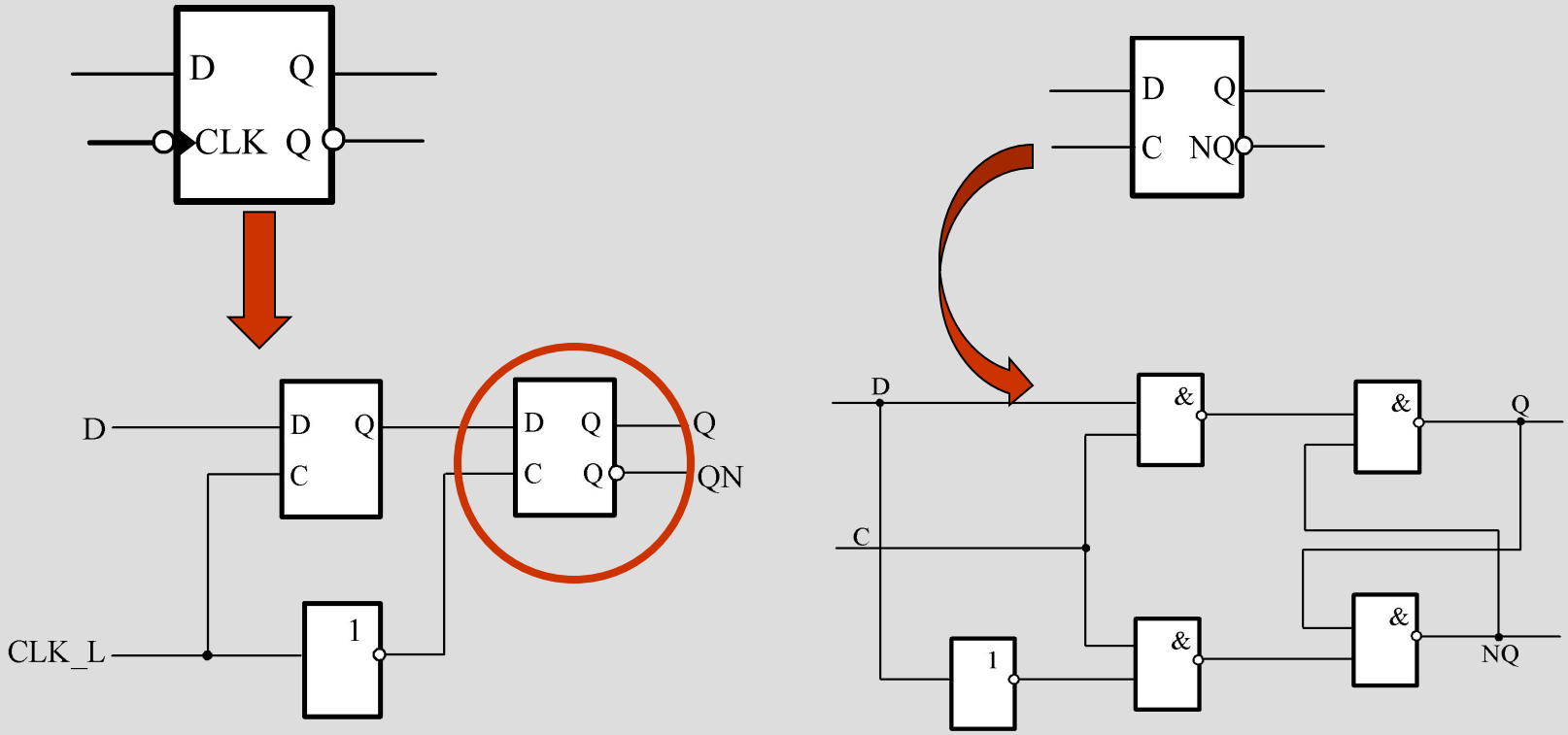
Счетчик по модулю 11 с последовательностью счета 5 – 15, 5 – 15

Разработка математической и программной модели цифрового объекта «Счетчик с принудительным порядком счета»





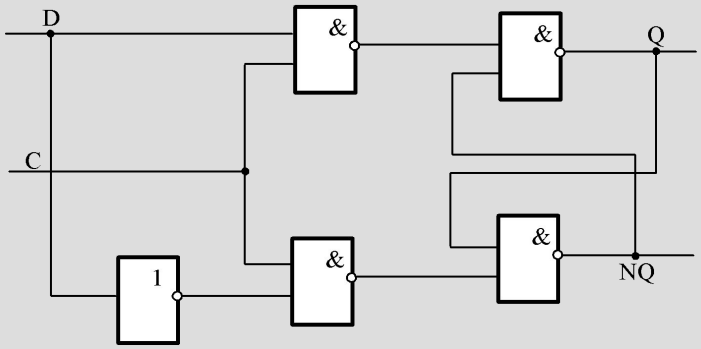
Разработка математической и программной модели цифрового объекта «Счетчик с принудительным порядком счета»



Разработка математической и программной модели цифрового объекта «Счетчик с принудительным порядком счета»

$$Q^{t+1} = D \cdot C + Q^t (\overline{C} + D)$$

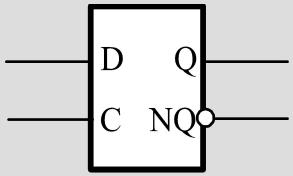
$$QN^{t+1} = \overline{Q}^t + C \cdot \overline{D}$$



Структура проекта на языке VHDL

```
ENTITY CompD IS
port  (D, C: in std_logic;
       Q, NQ: inout
std_logic);
end CompD;
```

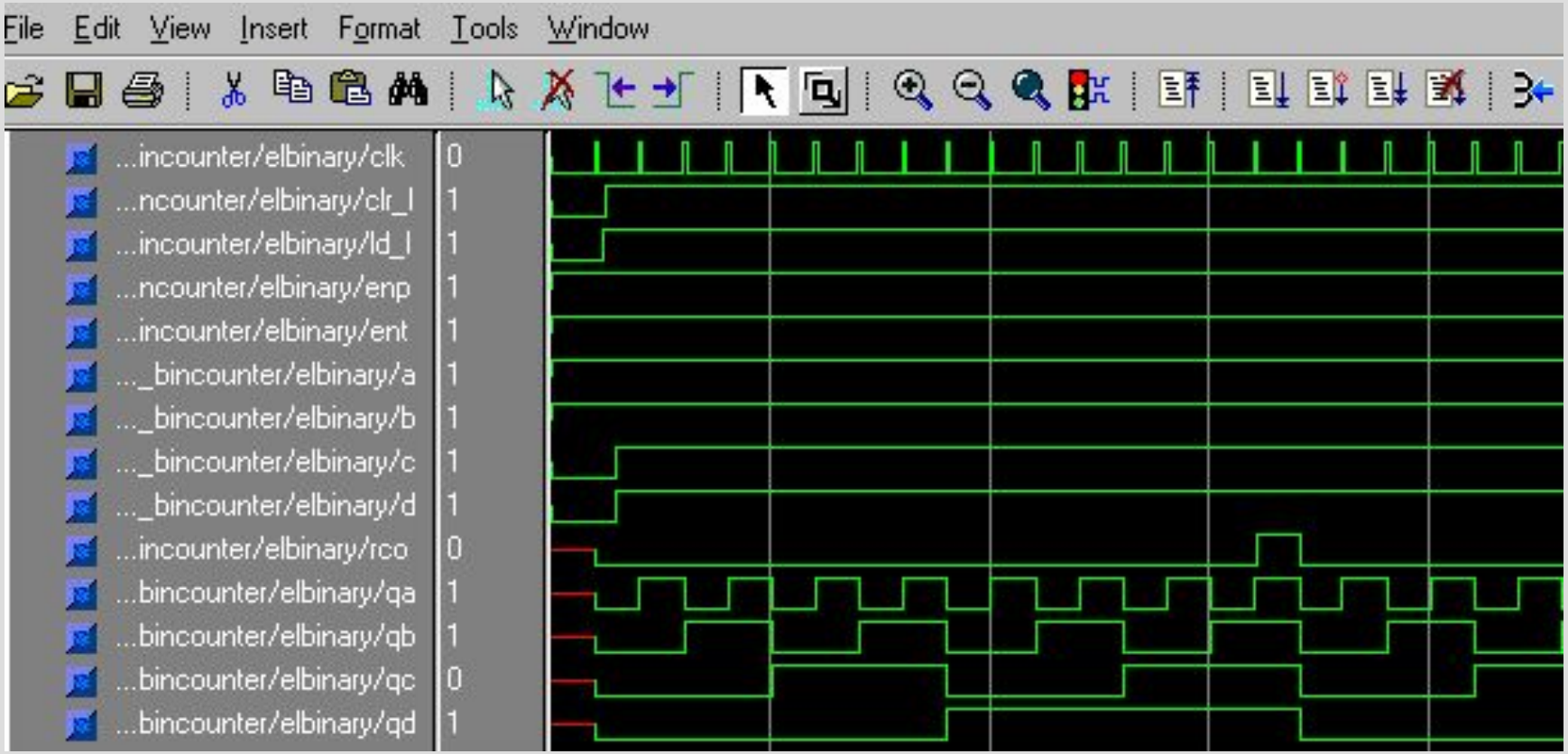
Описание интерфейса



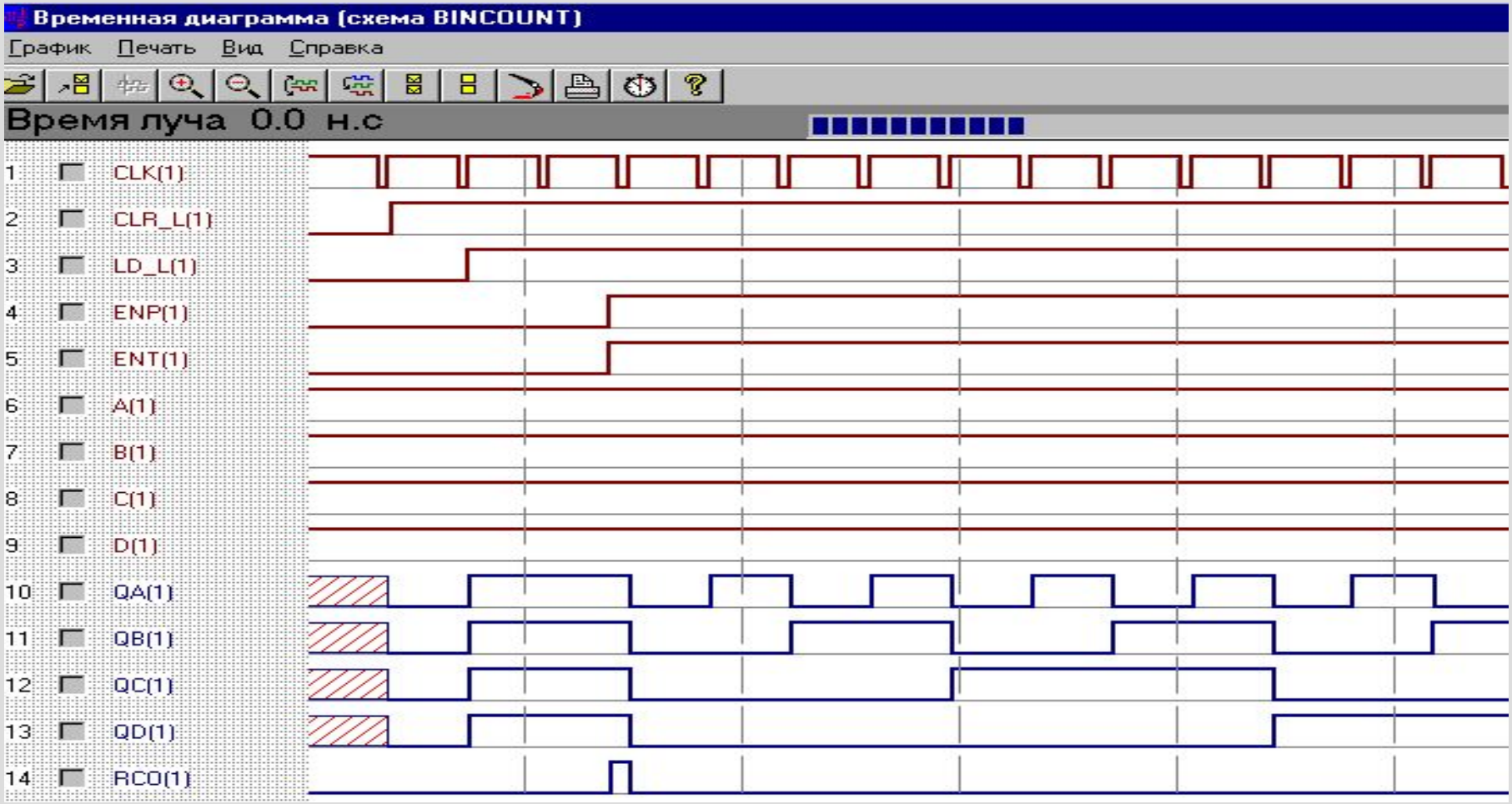

```
ARCHITECTURE Arch_D OF
CompD IS
component Nand2
port(x1, x2: in std_logic;
y: out std_logic);
end component;
signal a1, a2, a3: std_logic;
BEGIN
Ela1: Nand2 port map(D, D, a1);
Ela2: Nand2 port map(D, C, a2);
Ela3: Nand2 port map(C, a1, a3);
ElQ:  Nand2 port map(a2, NQ, Q);
ElNQ: Nand2 port map(Q, a3, NQ);
END Arch_D;
```

Описание функций

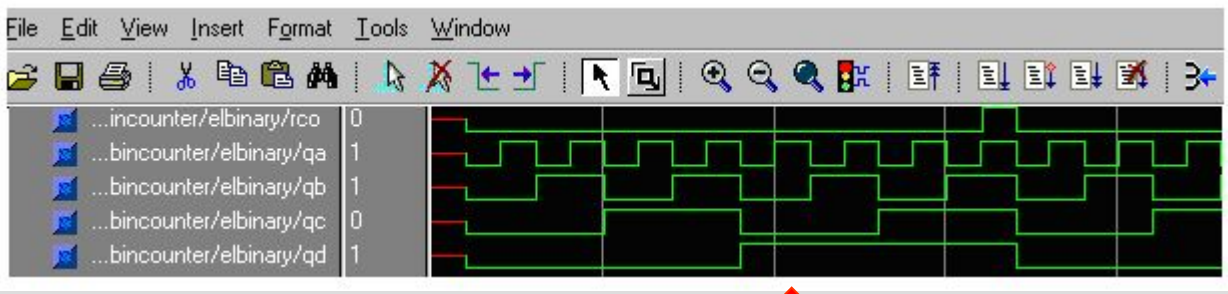
Моделирование двоичного счетчика в системе ModelSim



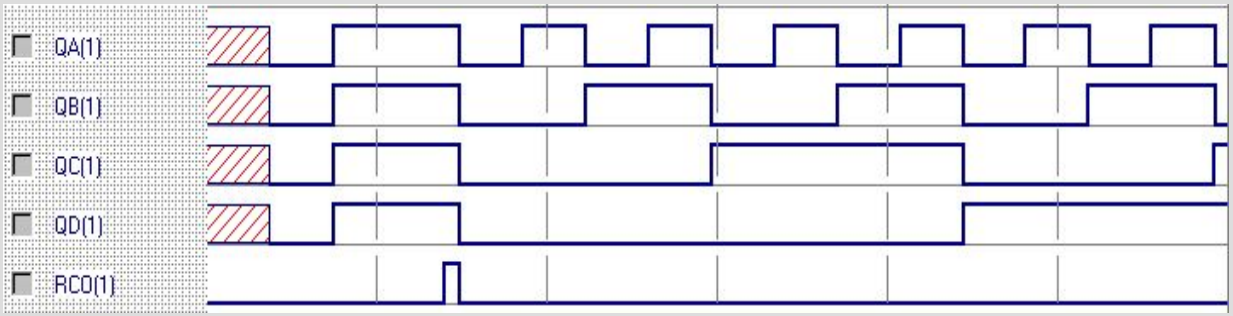
Моделирование двоичного счетчика в системе VLSI_SIM



Сравнение результатов моделирования



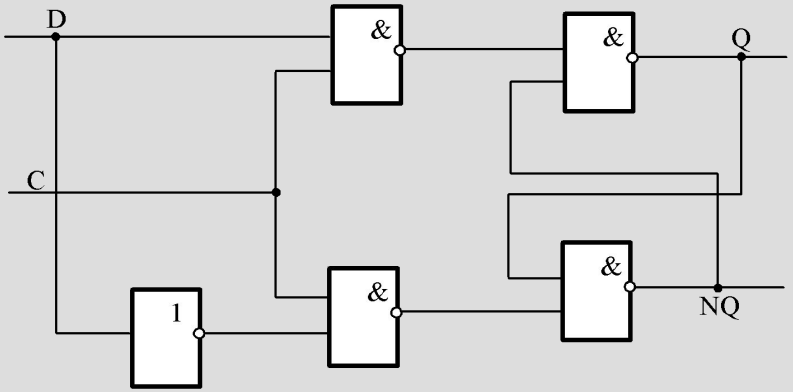
В ModelSim

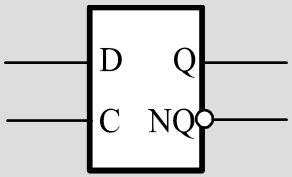


В VLSI_SIM

Примеры описания схем в системе VLSI_SIM

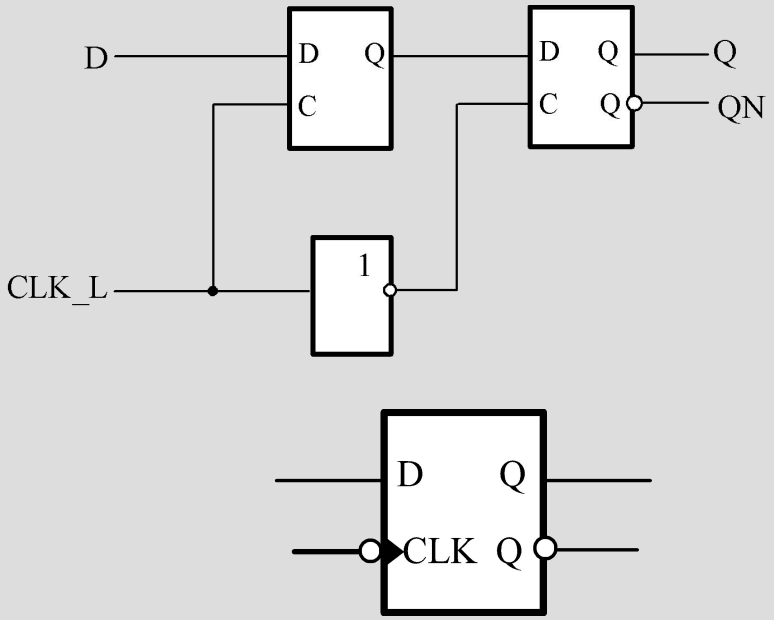
```
circuit  CompD;
inputs D(1),C(1);
outputs Q(1),QN(1); gates
 a1 'not' (1) D(1);
 b1 'nand' (1) D(1),C(1);
 b2 'nand' (1) C(1),a1(1);
 Q 'nand' (1) b1(1), QN(1);
 QN 'nand' (1) Q(1),b2(1);
 endgates
end
```





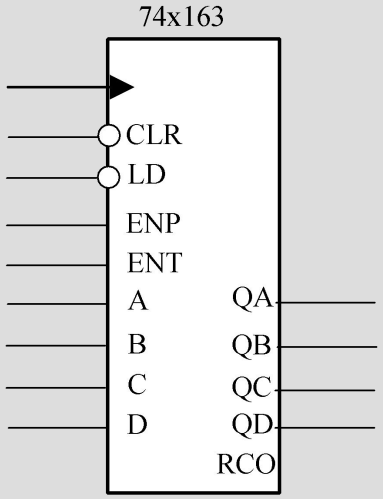
Примеры описания схем в системе VLSI_SIM

```
circuit  CompD2;
inputs D(1),C(1);
outputs b1(1),b1(2);
gates
 a1 'CompD' (2) D(1), C(1);
 a2 'not' (1) C(1);
 b1 'CompD' (2) a1(1),a2(1);
 endgates
end
```

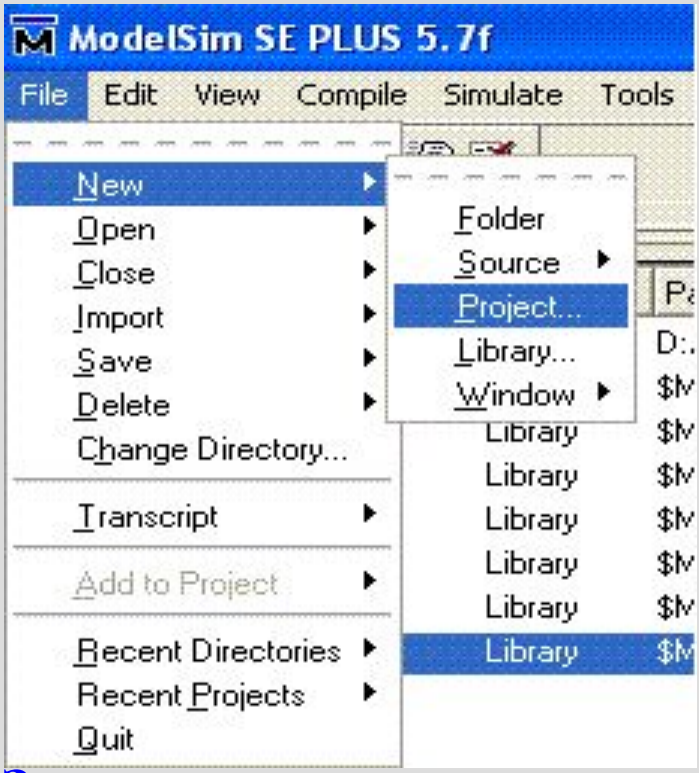


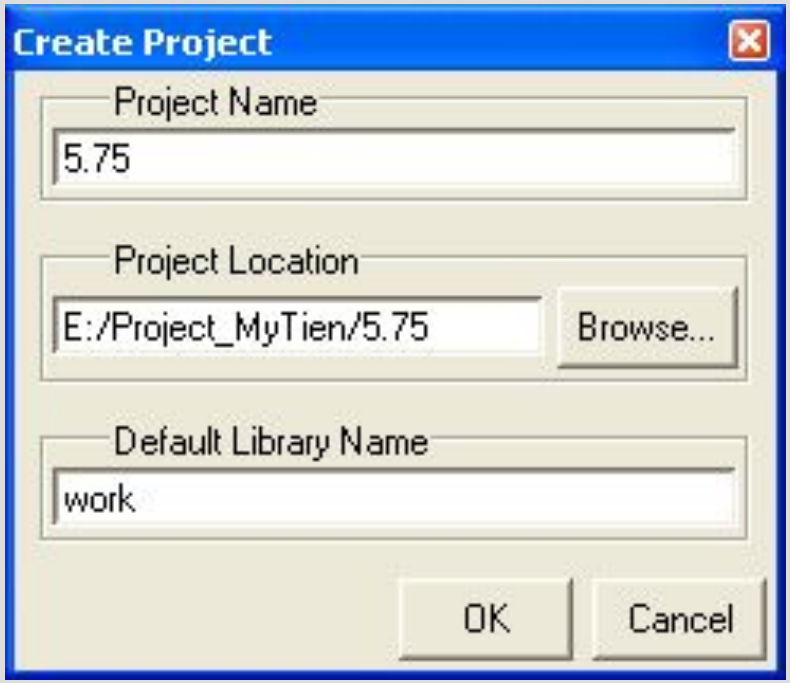
Примеры описания схем в системе VLSI_SIM

```
…
…
e3 'or' (1) d5(1), d6(1);
e4 'or' (1) d7(1), d8(1);
QA 'CompD2' (2) e1(1), a1(1);
QB 'CompD2' (2) e2(1), a1(1);
QC 'CompD2' (2) e3(1), a1(1);
QD 'CompD2' (2) e4(1), a1(1);
f1'not'(1) QA(2);
f2'not'(1) QB(2);
f3'not'(1) QC(2);
f4'not'(1) QD(2);
 endgates
end
```



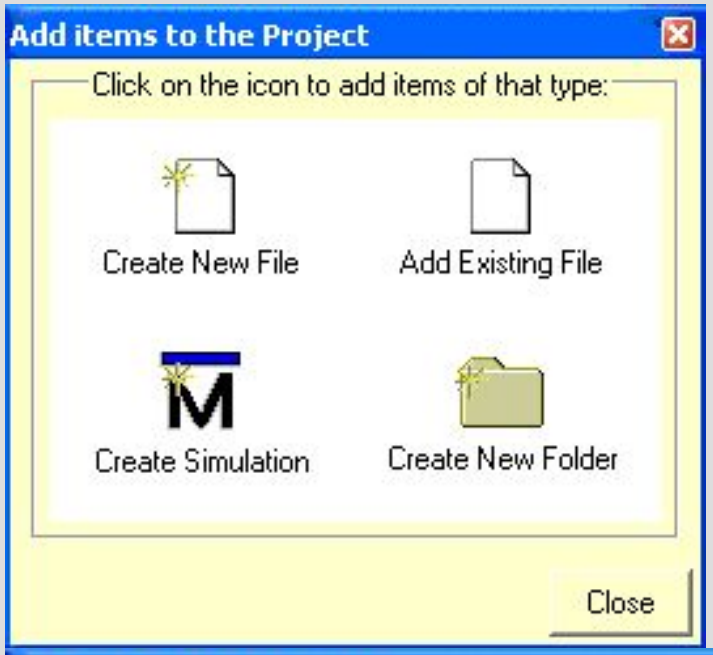
Создание нового проекта





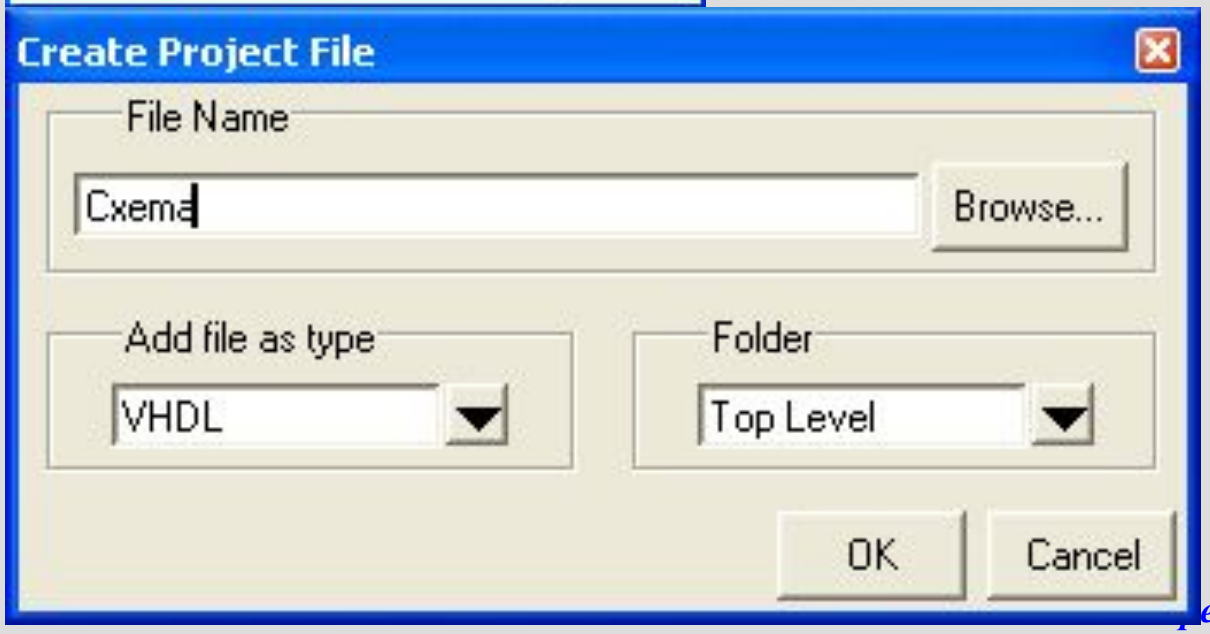
•Задается имя нового проекта, каталог, где он будет размещаться, имя рабочей библиотеки по умолчанию.

• Обычно можно оставить установленное имя библиотеки по умолчанию в "WORK".

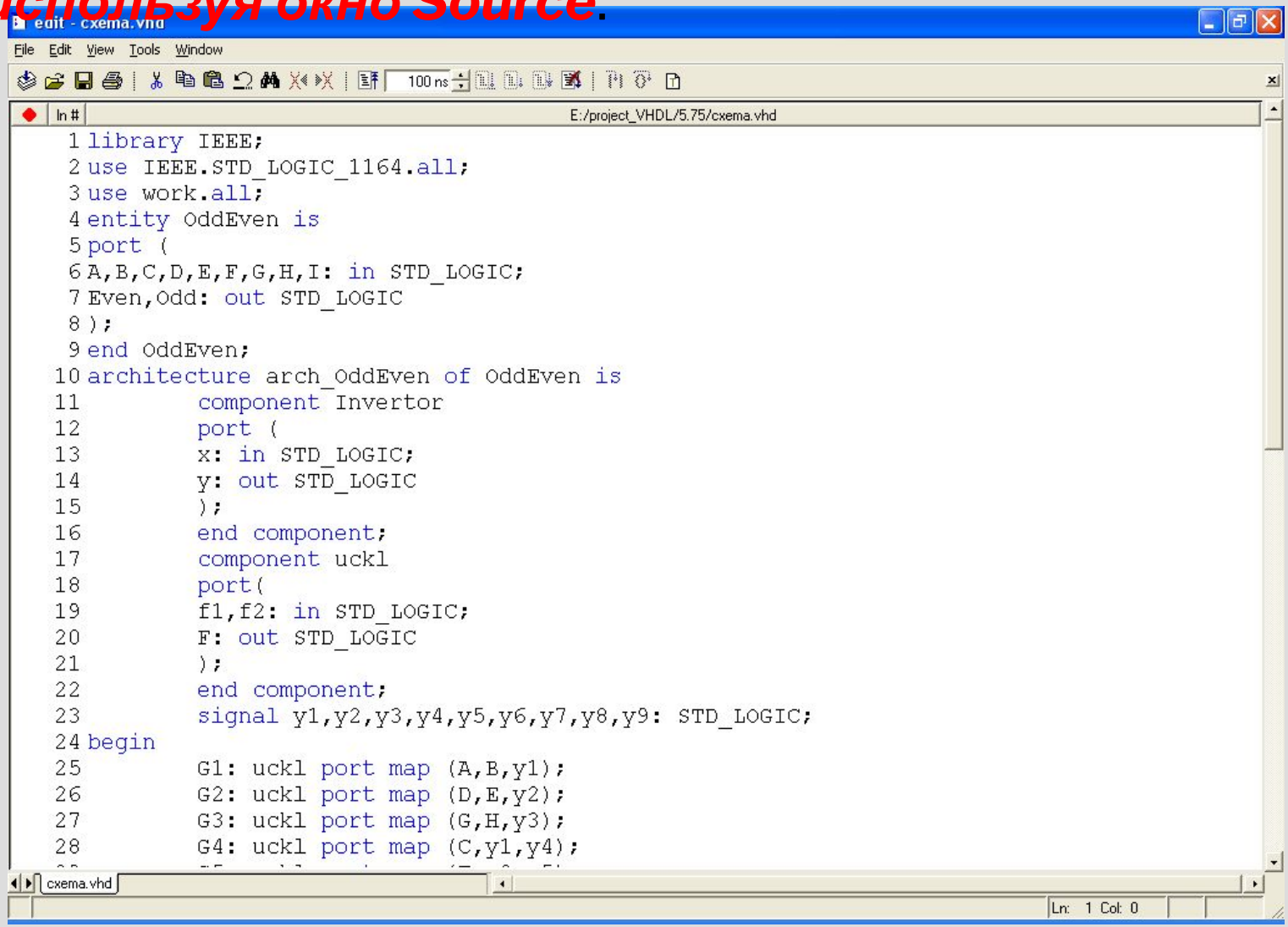


Добавление объектов к созданному проекту

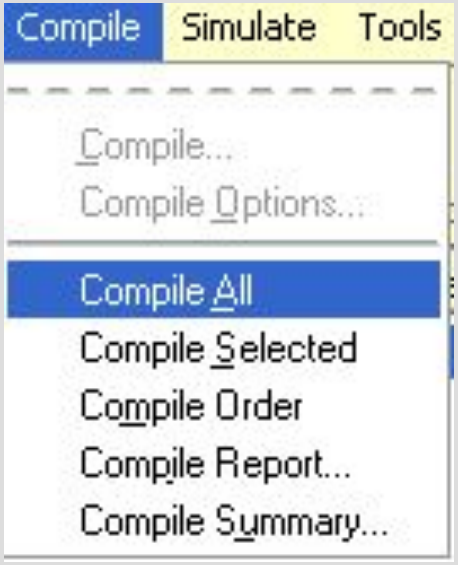
Создание нового файла



Создаем новый VHDL, или текстовый файл, используя окно Source.

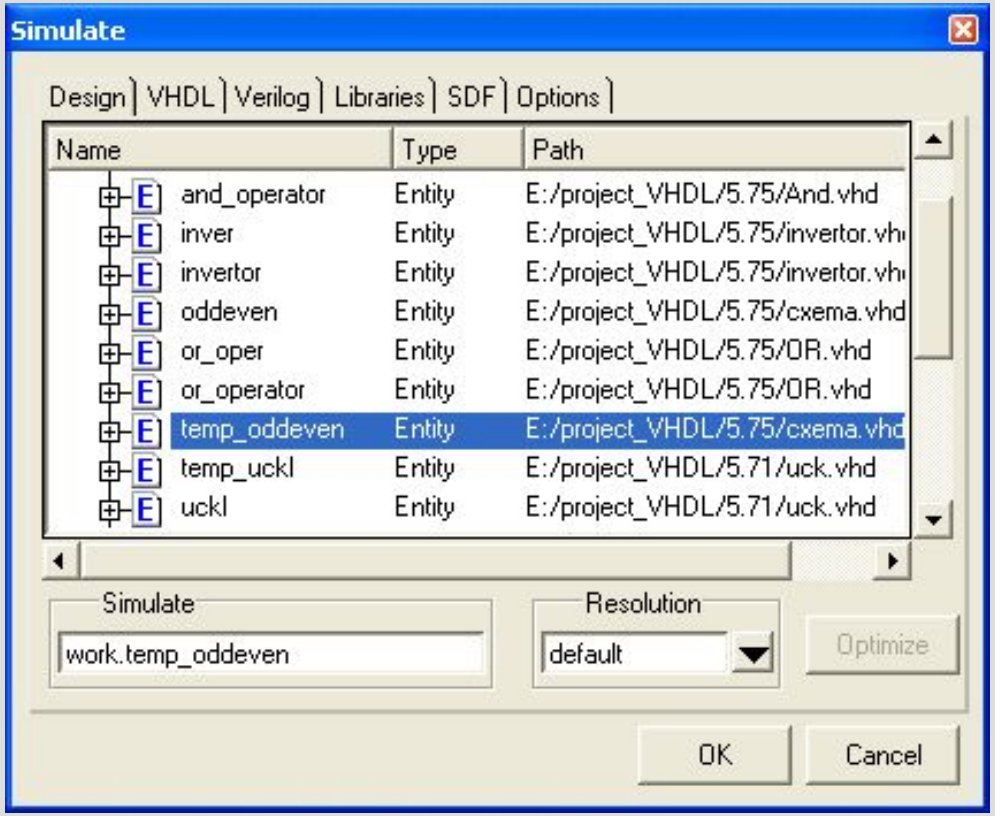


Компиляция файлов

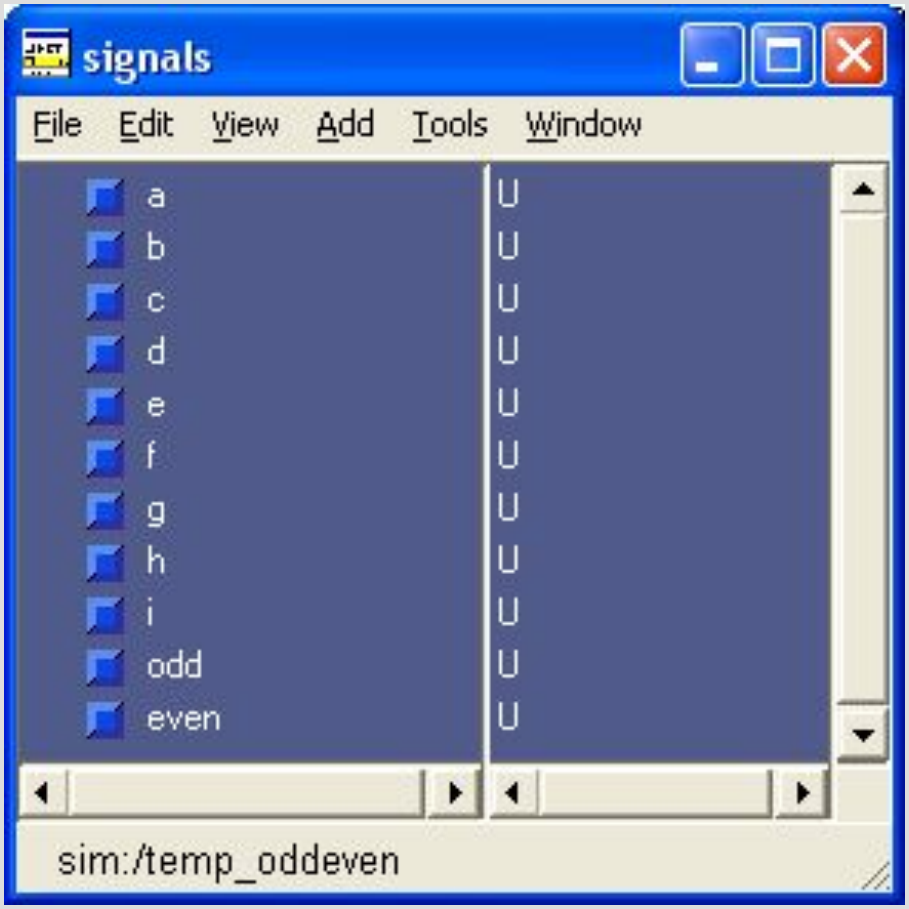


Моделирование проекта

Выбор теста



Добавление сигналов в окно Wave



Результаты моделирования