# Введение

1. Типы:
   Во-первых, мы говорим о пользовательских типах, включая записи, массивы и т.д.

2. Циклы FOR:
   При очень определенных условиях (если луна полная), можно использовать в процессе циклы FOR. Мы поговорим о том, когда можно сделать это.

3. Оператор Generate:
   Позволяет эффективно указать тип массива схем структурно

4. Драйверы с тремя состояниями:
   Шины и драйверы с тремя состояниями являются важной частью любой цифровой системы. Мы будем рассматривать этот драйвер и его реализацию в VHDL. Это требует более глубокого понимания типа STD_LOGIC_VECTOR.

# 1. Типы

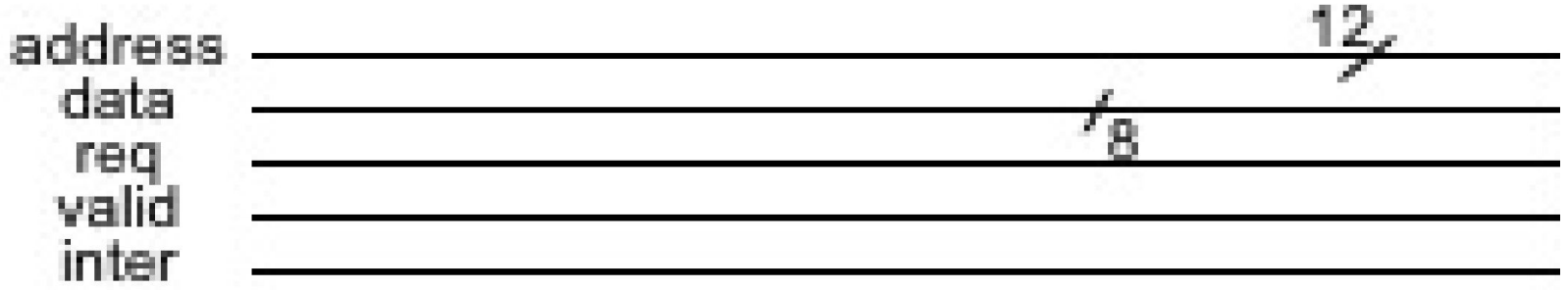
## Предустановленные Типы в VHDL

Некоторые общие встроенные типы (для сигналов и переменных): Бит: '0' или '1‘
Bit: ‘0’ или ‘1’
Bit vector: массив битов
Boolean: false, true
Character: любой символ
String: массив символов
Integer : целое
Real: двойной точности с плавающей точкой

Если включить ieee.std_logic_1164, вы также можете использовать std_logic и std_logic_vector

# Определение собственных типов:

Мы можем коллективно обратиться к группам битов, используя bit_vector или std_logic_vector. Иногда мы хотим, обратиться к коллекции ("запись") величин, как один блок (это могут быть биты, bit_vectors или что-нибудь)
Например, процессор может содержать шину с несколькими проводами:



# Определение записи

```
type bus_model is
record
address : bit_vector(11 downto 0);
data : bit_vector(7 downto 0);
req : bit;
valid : bit;
inter : bit;
end record;
signal x : bus_model;
```

Обратитесь к отдельным элементам: например, x.req

Можно копировать все записи: y <= x (это предполагает, что y также определен как bus_model)
Можно использовать записи в качестве входных портов для сущностей

Иногда, записи используются для определения “интерфейсов” между сущностями
Листинг сохраняет огромное количество входов и/или выходов
Запись может содержать другие записи

# Массивы:

```
type word is array (0 to 31) of bit;
type data is array (7 downto 0) of word;
signal x : data;
```

Можно также определить массив без его диапазона:

```
type mem is array (natural range <>) of word;
signal x : mem(3 downto 0);
signal y : mem(4 downto 0);
```

Вы можете иметь массивы записей и записи могут содержать массивы.
Можно иметь записи с массивами для других записей, содержащих другие массивы

# Подтипы:

Можно использовать подтип, чтобы ограничить возможные значения:

```
subtype small_int is integer range 0 to 10;
signal x : small_int;
```

также можно использовать подтип, чтобы ограничить диапазон вектора:

```
subtype word is std_logic_vector(31 downto 0);
signal x : word;
```

Преимущества: делает код яснее, симулятор позволяет пометить некоторые ошибки

В промышленности, VHDL дизайнеры часто и широко используют типы.

# STD_LOGIC как новый тип

```
type STD_ LOGIC is  (
            'U',    -- Неинициализированый
            'X',    -- Сильное неизвестное
            '0',    -- Сильный 0
            '1',    -- Сильная 1
            'Z'     -- Высокое сопротивление (импеданс)
            'W',    -- Слабое неизвестное
            'L',    -- Слабый 0
            'H',    -- Слабая 1
            '_'     -- Всё равно

            );
```

Это, как STD_LOGIC определяется как тип.
Включает в себя полный спектр значений, от 'U' до '-',
вкючая всё, что между ними

# Разница между типами и подтипами

Создание нового типа с "типа"
Невозможно назначить один тип в другой без функции преобразования

```
type mylogic is std_logic_vector(3 downto 0);
type urlogic is std_logic_vector(3 downto 0);
signal x : mylogic;
signal y : urlogic;
y <= x; -- ошибка, различные типы
```

Ограничение диапазона значений с "подтипом "
Подтип же «тип», как его «родитель» или базового типа.
Подтип добавляет дополнительный "диапазон условия" для правильных значений.

```
subtype mylogic is std_logic_vector(3 downto 0);
subtype urlogic is std_logic_vector(3 downto 0) range ‘0’ to ‘1’;
signal x : mylogic;
signal y : urlogic;
y <= x; -- хорошо, тот же тип. Но симулятор должен проверить соблюдение границ диапазона
```

# Перечисляемые типы:

Чтобы сделать код более читабельным, можно определить перечисляемые типы:

```
type state_types is (StateLive, StateWait, StateSample,
StateDisplay);
variable CURRENT_STATE : state_types;
```

Здесь мы определили переменную CURRENT_STATE, которая может принимать одно из четырех значений: StateLive, StateWait, StateSimple, StateDisplay. Мы можем использовать эту переменную, как и прежде, в операторе case

```
case PRESENT_STATE is
when StateLive => …
…
when StateWait => …
```

etc…

# 2. Циклы
# Циклы в процессе

VHDL поддерживает простые циклы, но будьте осторожны, потому что большинство структур цикла не синтезируемые. Чтобы понять, что это синтезируемо, вы должны знать, что компилятор "разворачивает" цикл, прежде чем синтезировать. Так, верхняя и нижняя границы должны иметь постоянные значения, известные во время компиляции.
Таким образом, это синтезируемо большинством инструментов:

```
process(SW)
variable p : std_logic;
begin
    p := '0';
    for i in 2 downto 0 loop
            p := p xor SW(i);
    end loop;
end process;
```

Получим развёрнутый вид:

```
p := p xor SW(2);
p := p xor SW(1);
p := p xor SW(0);
```

Это также синтезируемо большинством инструментов:

```
process(SW)
begin
    for i in 2 downto 0 loop
        OUT(i) <= SW(i);
    end loop;
end process;
```

Получим развёрнутый вид:

```
OUT(2) <= SW(2);
OUT(1) <= SW(1);
OUT(0) <= SW(0);
```

Тем не менее, это **не синтезируемо**: (здесь IN1 сигнал или переменная)

```
process(SW)
begin
    for i in IN1 downto 0 loop
        OUT(i) <= SW(i);
    end loop;
end process;
```

Невозможно определить статический разворот во время компиляции (в зависимости от значения IN1, которое не известно во время компиляции). Поэтому это не синтезируемо.

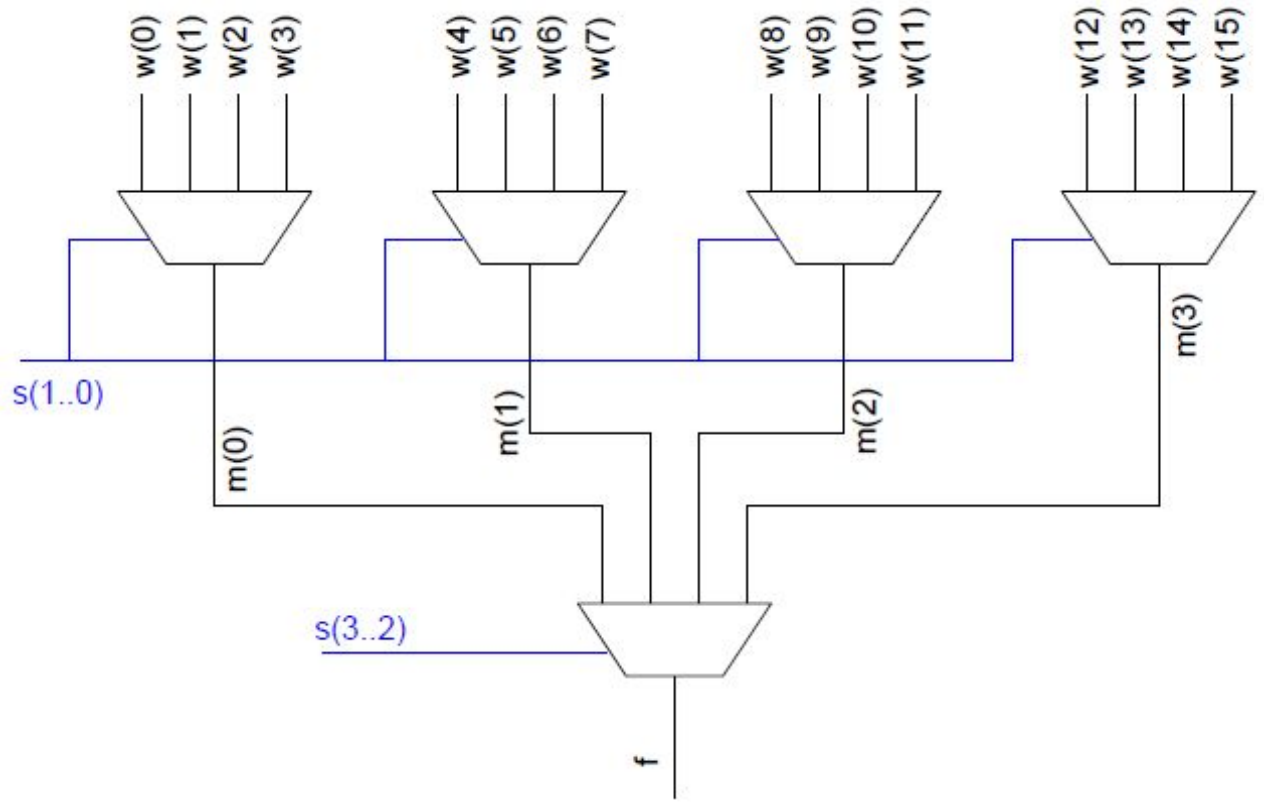
Наблюдение: поскольку границы должны быть известны во время компиляции, цикл в процессе является просто коротким путём, для описания поведения по всем битам. Это на самом деле не подразумевает какого-либо "зацикливания" поведения оборудования.

Цикл поведения оборудования не может быть сделан в один процесс, необходимо создать Datapath.

Мой совет: По большей части, вы должны избегать циклов for в вашем синтезируемом коде.

# 3. Оператор Generate
# Структурные описания

Предположим, что мы хотим разработать эту схему:




Кроме того, предположим, что у нас уже есть 4-входовые 1-битные ячейки мультиплексора, и мы хотим создать структурное описание.

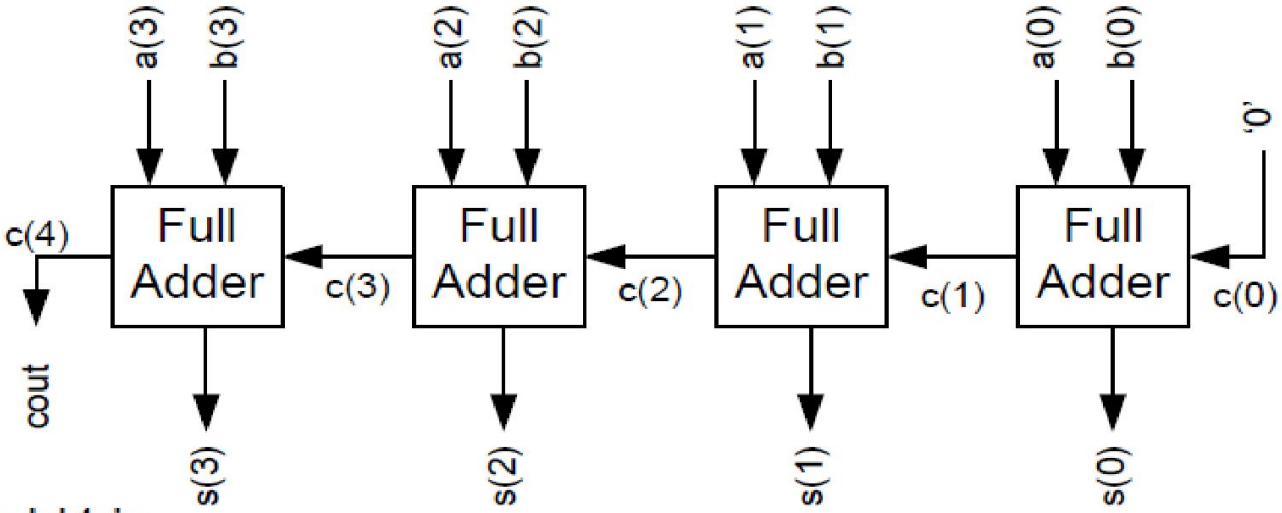
```
architecture structural of mux16to1 is
    signal m : std_logic_vector(0 to 3);
begin
    u1: mux4to1 port map
          ( w(0), w(1), w(2), w(3), s(1 downto 0), m(0) );
    u2: mux4to1 port map
          ( w(4), w(5), w(6), w(7), s(1 downto 0), m(1) );
    u3: mux4to1 port map
          ( w(8), w(9), w(10), w(11), s(1 downto 0), m(2) );
    u4: mux4to1 port map
          ( w(12), w(13), w(14), w(15), s(1 downto 0), m(3) );
    u5: mux4to1 port map
          ( m(0), m(1), m(2), m(3), s(3 downto 2), f );
end structural
```

Предположим, мы уже определили этот блок

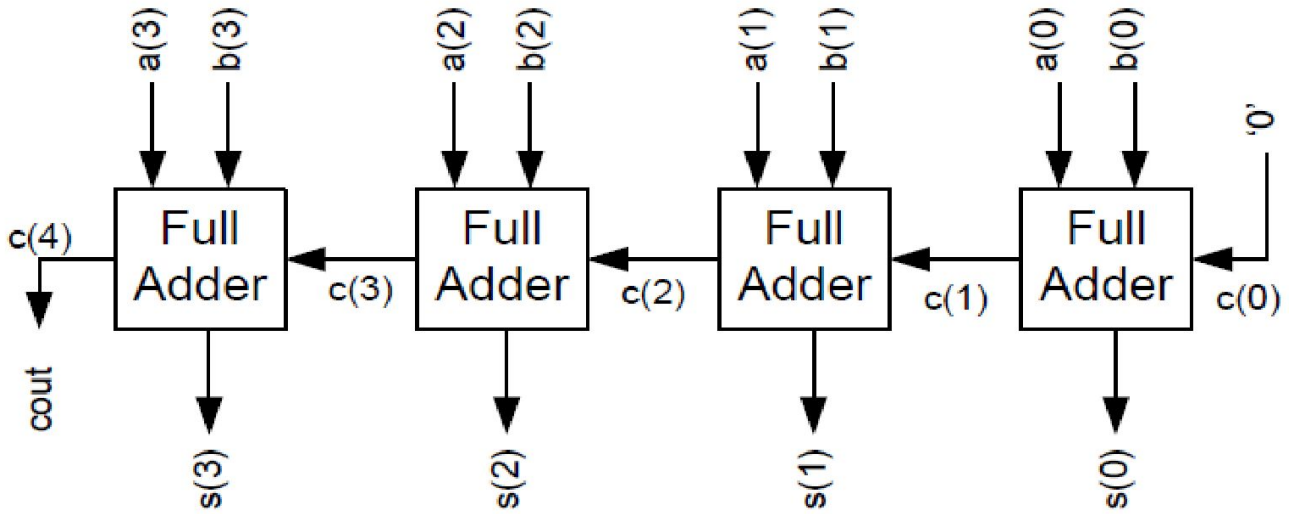
```
architecture structural of mux16to1 is
    signal m : std_logic_vector(0 to 3);
begin
    g1: for i in 0 to 3 generate
        muxes: mux4to1 port map
            ( w(4*i), w(4*i+1), w(4*i+2), w(4*i+3), s(1 downto 0), m(i) );
    end generate;
    u5: mux4to1 port map
        ( m(0), m(1), m(2), m(3), s(3 downto 2), f );
end structural
```

Примечание: в отличие от цикла, оператор generate не может быть в процессе

Другой пример: 4-разрядный сумматор построен с использованием четырех полных сумматоров:




```
architecture structural of add4 is
    signal c : std_logic_vector(4 downto 0);
begin
    c(0) <= cin;
    u0: full_adder port map(a(3), b(3), c(3), s(3), c(4));
    u1: full_adder port map(a(2), b(2), c(2), s(2), c(3));
    u2: full_adder port map(a(1), b(1), c(1), s(1), c(2));
    u3: full_adder port map(a(0), b(0), c(0), s(0), c(1));
    cout <= c(4);
end structural;
```




```
architecture structural of add4 is
    signal c : std_logic_vector(4 downto 0);
begin
    c(0) <= cin;
    g0: for i in 3 downto 0 generate
        FA: full_adder port map(a(i), b(i), c(i), s(i), c(i+1));
    end generate g0;
    cout <= c(4);
end structural;
```

Иногда отдельные модули немного отличаются в зависимости от их положения в массиве.

Пример: в нашем сумматоре мы могли бы утверждать, что: бит 0 является особым случаем, поскольку на перенос входит сигнал '0‘, бит 3 тоже особый случай, потому что перенос выходит на сигнал cout.

Мы можем написать "особые случаи", используя “if… generate”

Внимание: слишком много "особых случаев" сделать ваш код нечитабельным. Используйте это по усмотрению.

```
architecture structural of add4 is
    signal c : std_logic_vector(3 downto 1);
begin
    g0: for i in 3 downto 0 generate
        label0: if i = 0 generate
            FA: full_adder port map(a(0), b(0), '0' , s(0), c(1));
        end generate label0;
        label2: if i >0 and i < 3 generate
            FA: full_adder port map(a(i), b(i), c(i) , s(i), c(i+1));
        end generate label2;
        label3: if i = 3 generate
            FA: full_adder port map(a(3), b(3), c(3) , s(3), cout);
        end generate label3;
    end generate g0;
end structural;
```

Еще одно предупреждение: Generate является ***только*** инструментом, который помогает описать структурные описания!

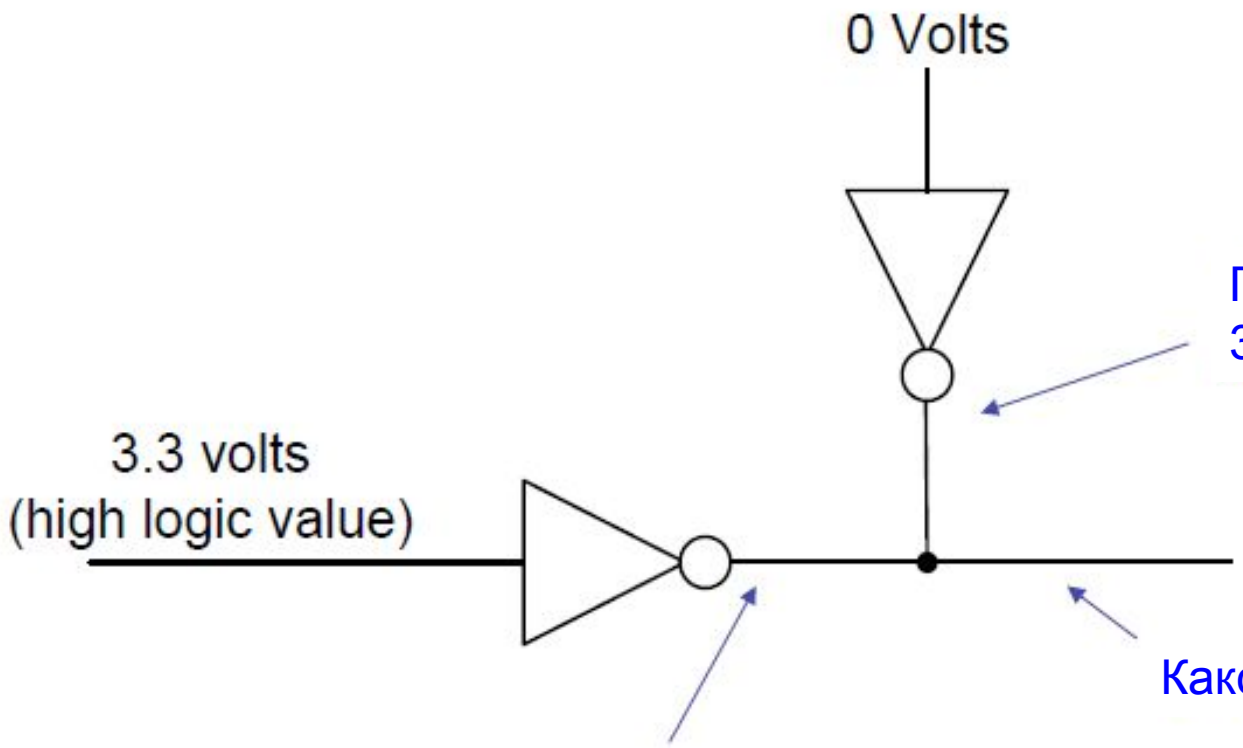
Различия между generate и оператором " for " :

1. “FOR" внутри процесса. "GENERATE" нет.

2. "GENERATE" удобно использовать для описания структуры с использованием port-maps.“FOR" используется для удобства при описании поведения.

Ни один из них не подразумевает какого-либо "зацикливания" на оборудовании.

# Драйверы в троичной логике/ Tri-state Drivers




Пытается подать на узел
3.3 вольта

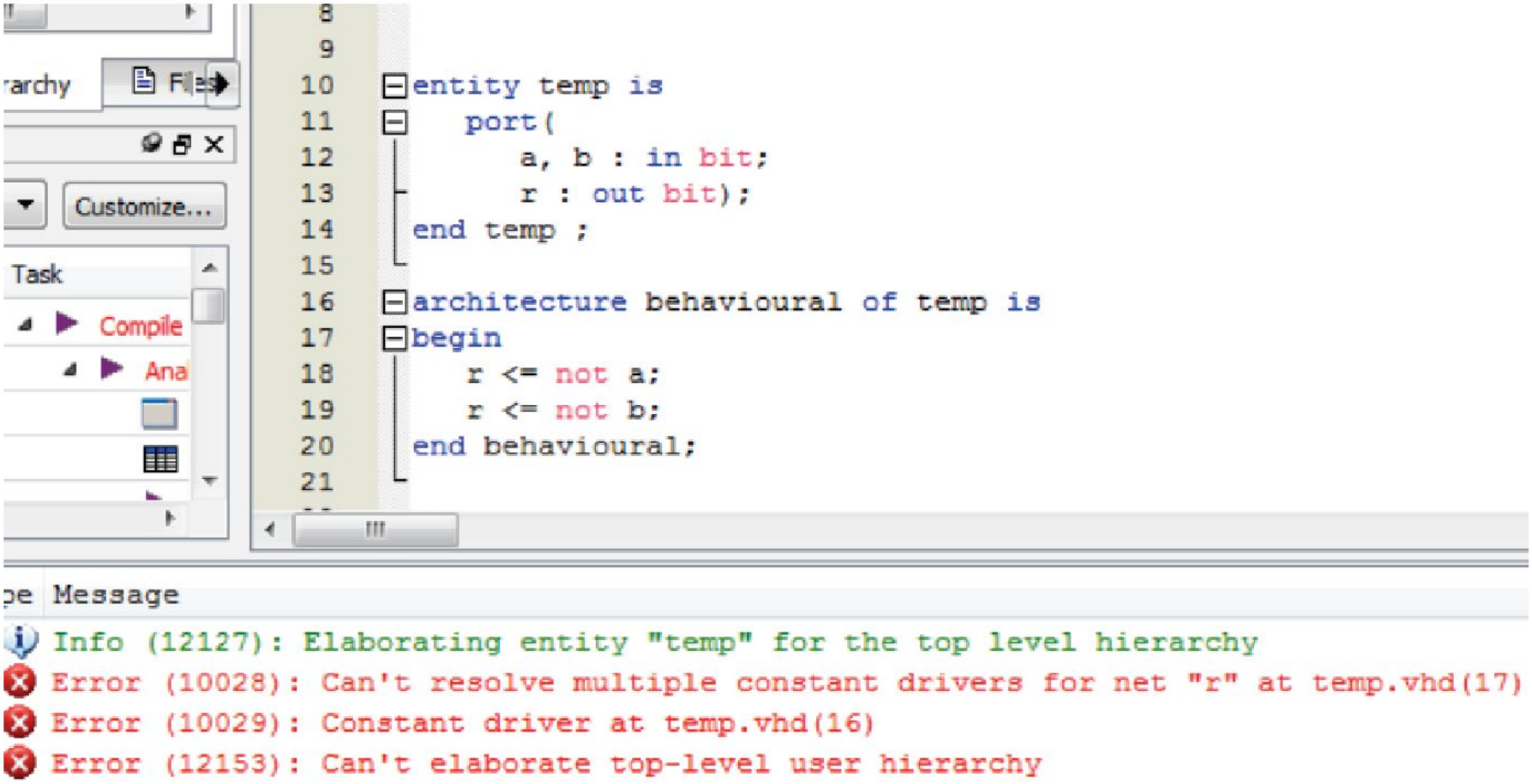
Какое напряжение здесь?

Пытается подать на узел
0 вольт

Это пример “борьбы”
- Что происходит, зависит от способа реализации схемы - для CMOS (большинство чипов разработано с использованием КМОП), борьба может вызвать большие токи и
даже повредить чип!

Интересный вопрос: что произойдет, если мы измерим напряжение такого провода?

Если вы попытаетесь синтезировать это в Quartus II: будет ошибка, которая не позволяет реализовать:



# Моделирование

Что произойдет, если имитировать VHDL спецификации этой схемы ...

Какое должно быть значение на выхода этого узла?

-Сигнал типа "бит" может быть 0 или 1

- Но ни один из них является правильным ответом.

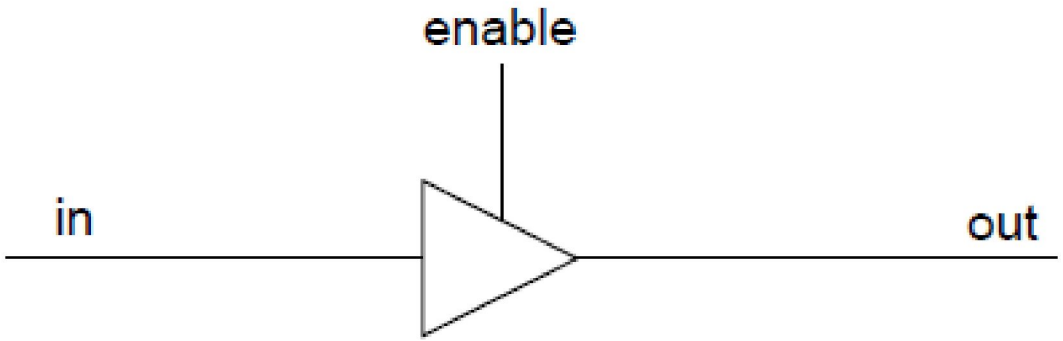
- Один из возможных значений сигнала типа "std_logic" является X

- X обозначает неизвестно.

Когда VHDL симулятор видит эту борьбу, он устанавливает значение выхода в X. Это говорит вам что-то не так.

В реальной схеме, узел будет иметь некоторое значение ... (не может измерить X)

# Троичная логика




Если "включить" 1, то выход управляется со значением на входе.

Если "включить" 0, то выход не управляется.

- Что это значит? Если никто не едет управляет этим сигналом, то сигнал является "плавающим"

- Что произойдет, если вы попытаетесь измерить напряжение плавающего сигнала?

# Моделирование

Что произойдет, если имитировать VHDL спецификации этой схемы ...

Если enable = 0, что должно моделирование показать на выходе?

– Одним из возможных значений сигнала типа "std_logic" является Z - означает высокое сопротивление.

| in | enable | out |
|---|---|---|
| 0 | 0 | z |
| 1 | 0 | z |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Помните: в реальном мире, сигнал будет иметь напряжения, даже если оно не приводится, поэтому вы не можете на лабораторной измерить Z

Проблема в том, что если enable=0, выход не приводится. Синтаксис VHDL говорит, что в этом случае, выход сохраняет свое старое значение.

```
entity tristatedriver is
    port(   A, enable : in std_logic;
            Q : out std_logic);
end tristatedriver;

architecture behavioural of tristatedriver is
begin
    process (A, enable)
    begin
        if (enable = '1' ) then
            Q <= A;
        end if;
    end process;
end behavioural;
```

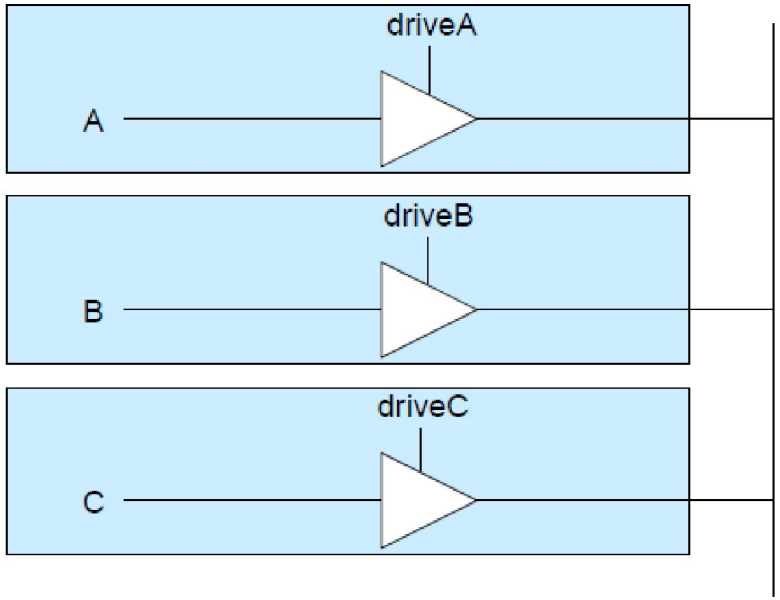
НЕВЕРНО

```
entity tristatedriver is
    port(   A, enable : in std_logic;
            Q : out std_logic);
end tristatedriver;

architecture behavioural of tristatedriver is
begin
    process (A, enable)
    begin
        if (enable = '1' ) then
            Q <= A;
        else
            Q <= 'Z' ;
        end if;
    end process;
end behavioural;
```

ВЕРНО

# Как обычно используются драйверы троичной логики




Наибольший из драйверов включается во время.
Если мы хотим A для управления выходом, включаем DriveA и т.д. Вы можете думать об этом как "распределенном" мультиплексоре

# Некоторые технические детали

Каждый "тип" в VHDL имеет соответствующую функцию разрешения.

Функция разрешения принимает все значения управления на определенном проводе и рассчитывает, что полученное значение на этом проводе должно быть

| input | output |
|---|---|
| 1 | 1 |
| 0 | 0 |
| 0, 1 | X |
| 0, 0, 0, 0 | 0 |
| 0, 0, 0, 1 | X |
| Z, 0 | 0 |
| Z, 1, 1, 1 | 1 |
| Z, 0, 1 | X |

Для использования нужно использовать пакет std_logic_1164, функция разрешения для std_logic уже определена

# Итоги

Тема 1: Мы рассмотрели, как можно определить свои собственные типы, используя записи, подтипы, перечисляемые типы, массивы. В промышленности, люди широко используют типы

Тема 2: Мы рассмотрели синтаксис цикла FOR. Эти циклы часто бывают не синтезируемыми

Тема 3: Оператор Generate поможет описать структурные описания из массивов структур.

Тема 4: Провода типа std_logic могут иметь значения: X: неизвестное значение (из-за борьбы обычно) Z: высокое сопротивление (не управлятся)
Важно: Эти значения не являются физическими напряжениями - Вы не можете измерить Z или X в лаборатории.