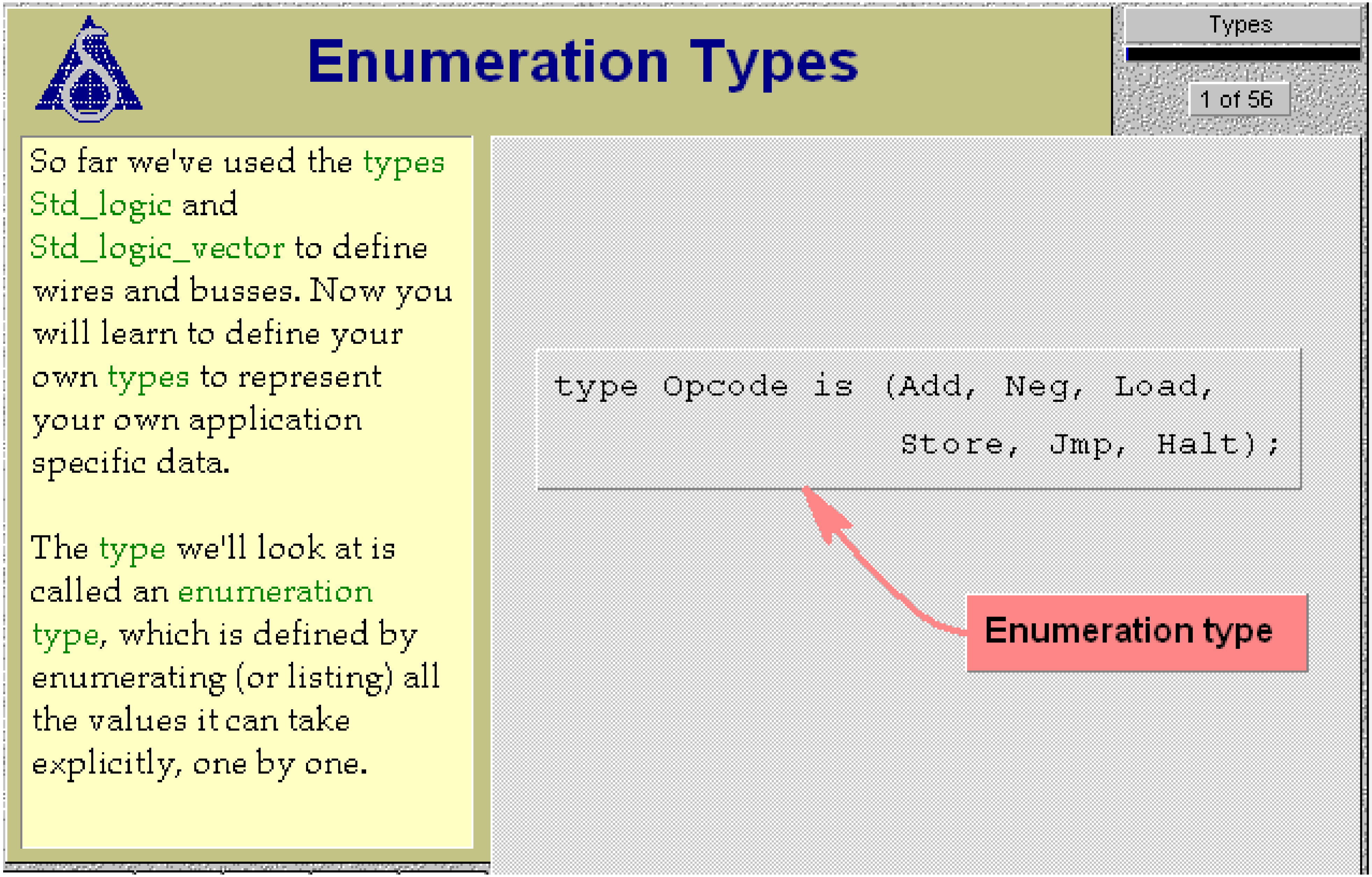
DOULOS

Lesson 6

Types

In the previous lesson we looked at combinational logic, but before we look at clocked logic, you need to learn about the important topic; Types. In this lesson you will learn to define your own types, and also look in more detail at the IEEE types STD_LOGIC and STD_LOGIC_VECTOR.



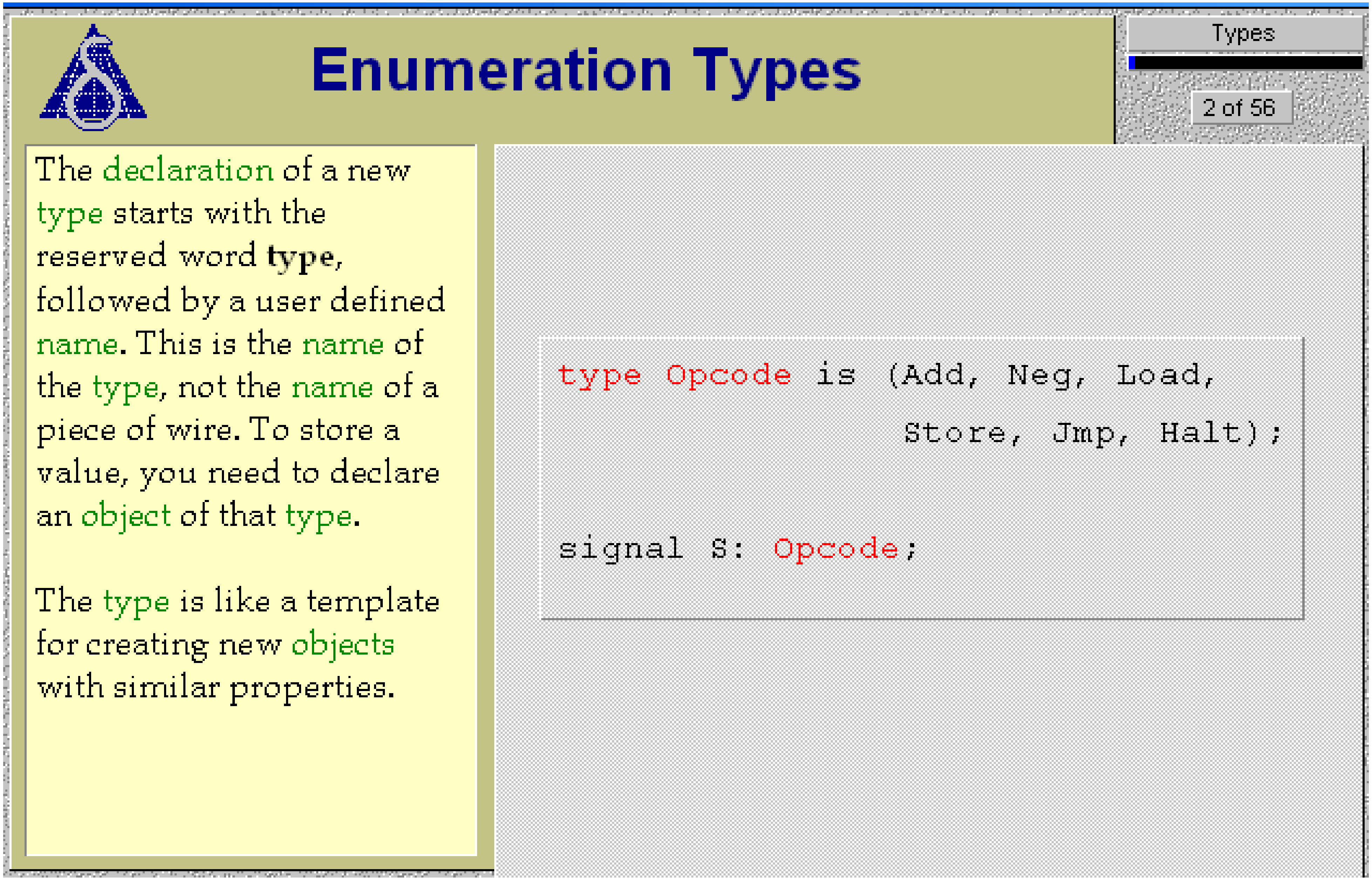
Enumeration Types

So far we've used the types Std_logic and Std_logic_vector to define wires and busses. Now you will learn to define your own types to represent your own application specific data.

The type we'll look at is called an enumeration type, which is defined by enumerating (or listing) all the values it can take explicitly, one by one.

```
type Opcode is (Add, Neg, Load,
                Store, Jmp, Halt);
```

Enumeration type



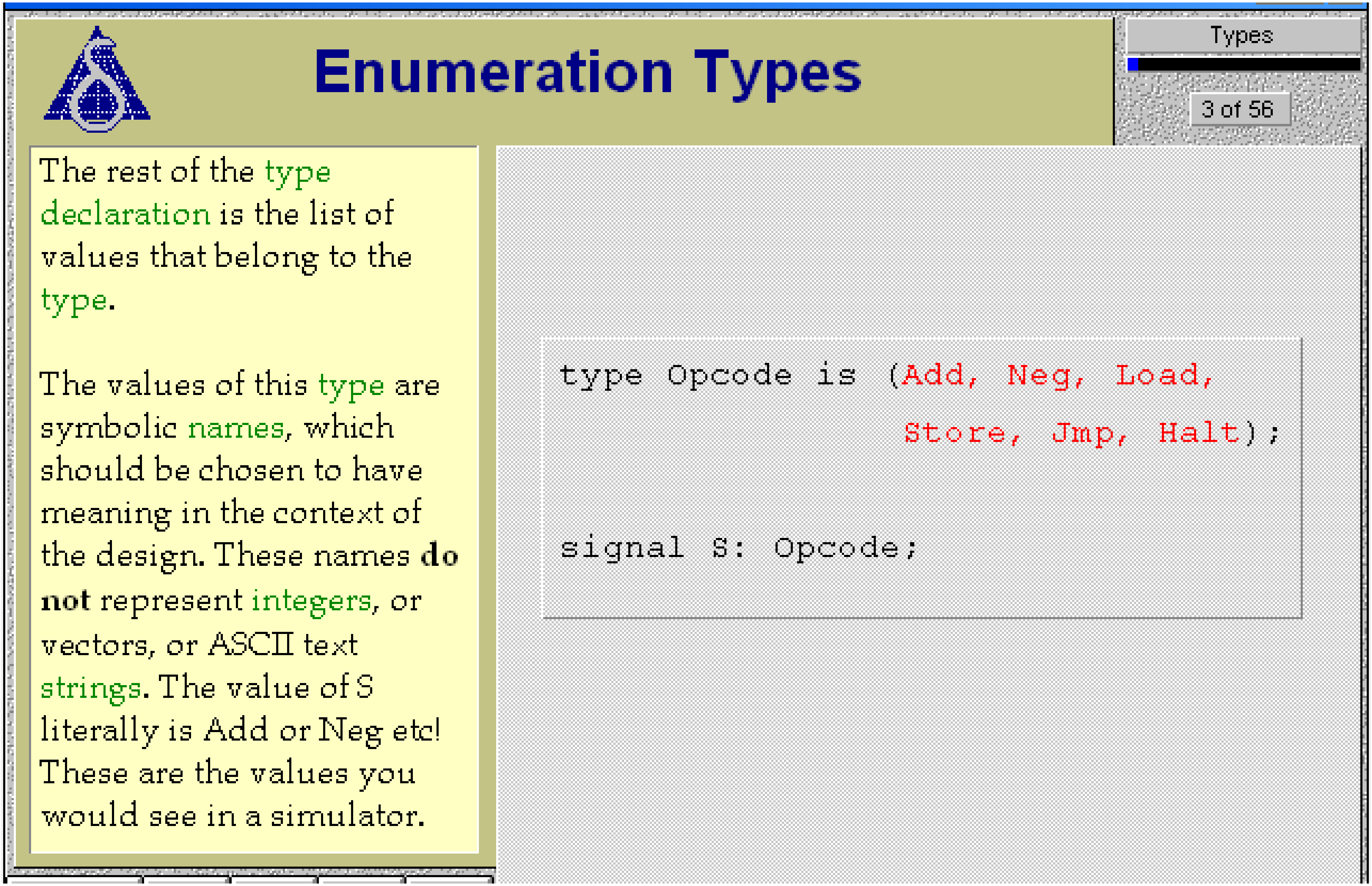
Enumeration Types

The declaration of a new type starts with the reserved word **type**, followed by a user defined name. This is the name of the type, not the name of a piece of wire. To store a value, you need to declare an object of that type.

The type is like a template for creating new objects with similar properties.

```
type Opcode is (Add, Neg, Load,
                Store, Jmp, Halt);


signal S: Opcode;
```



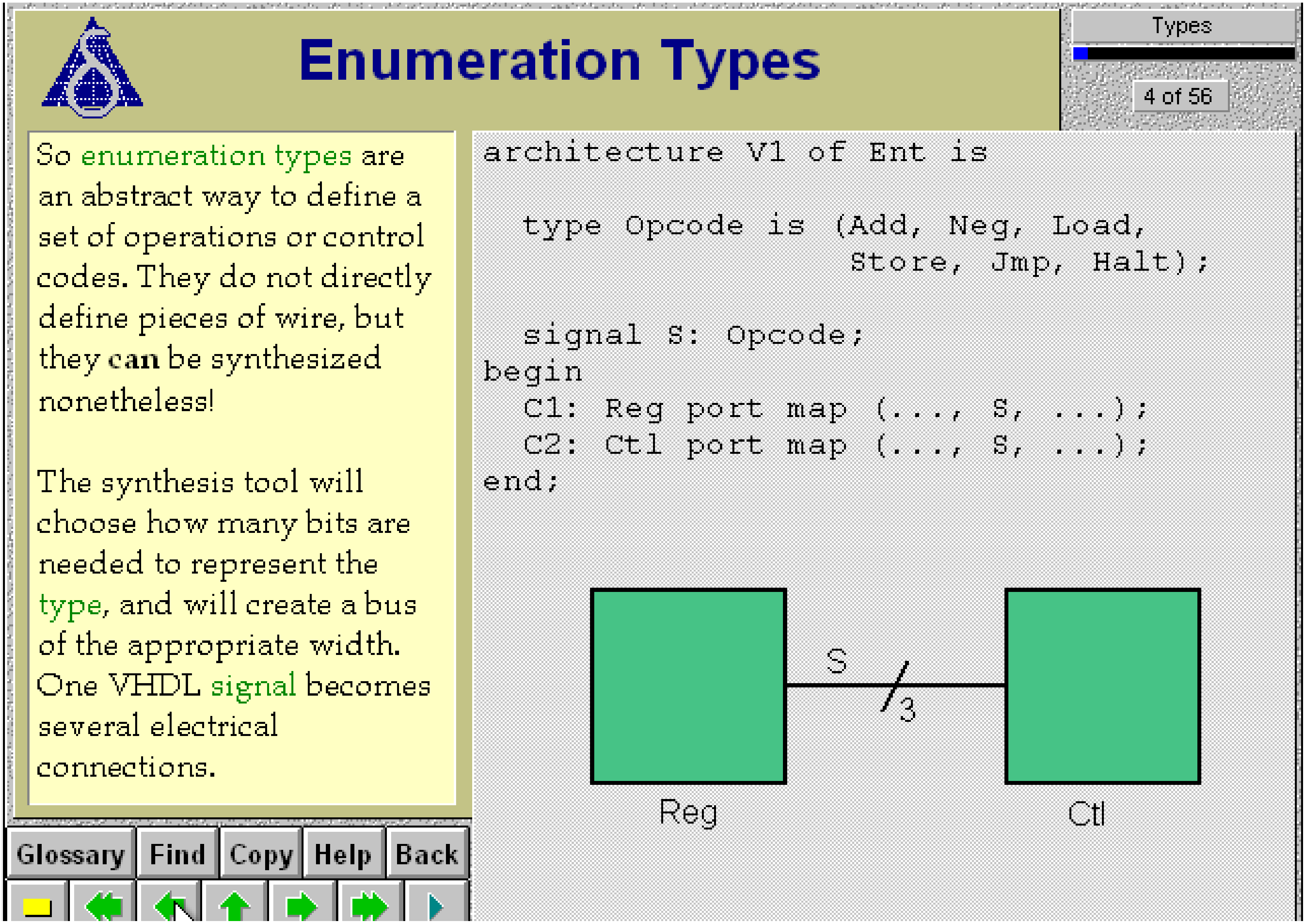
Enumeration Types

The rest of the type declaration is the list of values that belong to the type.

The values of this type are symbolic names, which should be chosen to have meaning in the context of the design. These names **do not** represent integers, or vectors, or ASCII text strings. The value of S literally is Add or Neg etc! These are the values you would see in a simulator.

```
type Opcode is (Add, Neg, Load,
                Store, Jmp, Halt);


signal S: Opcode;
```



Enumeration Types

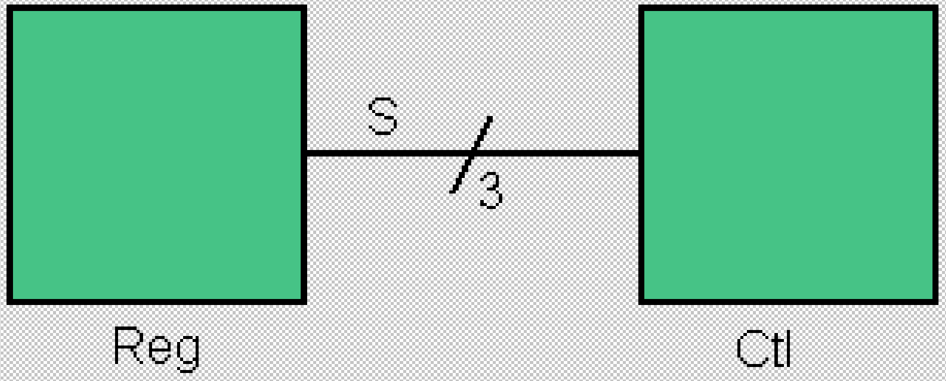
So enumeration types are an abstract way to define a set of operations or control codes. They do not directly define pieces of wire, but they **can** be synthesized nonetheless!

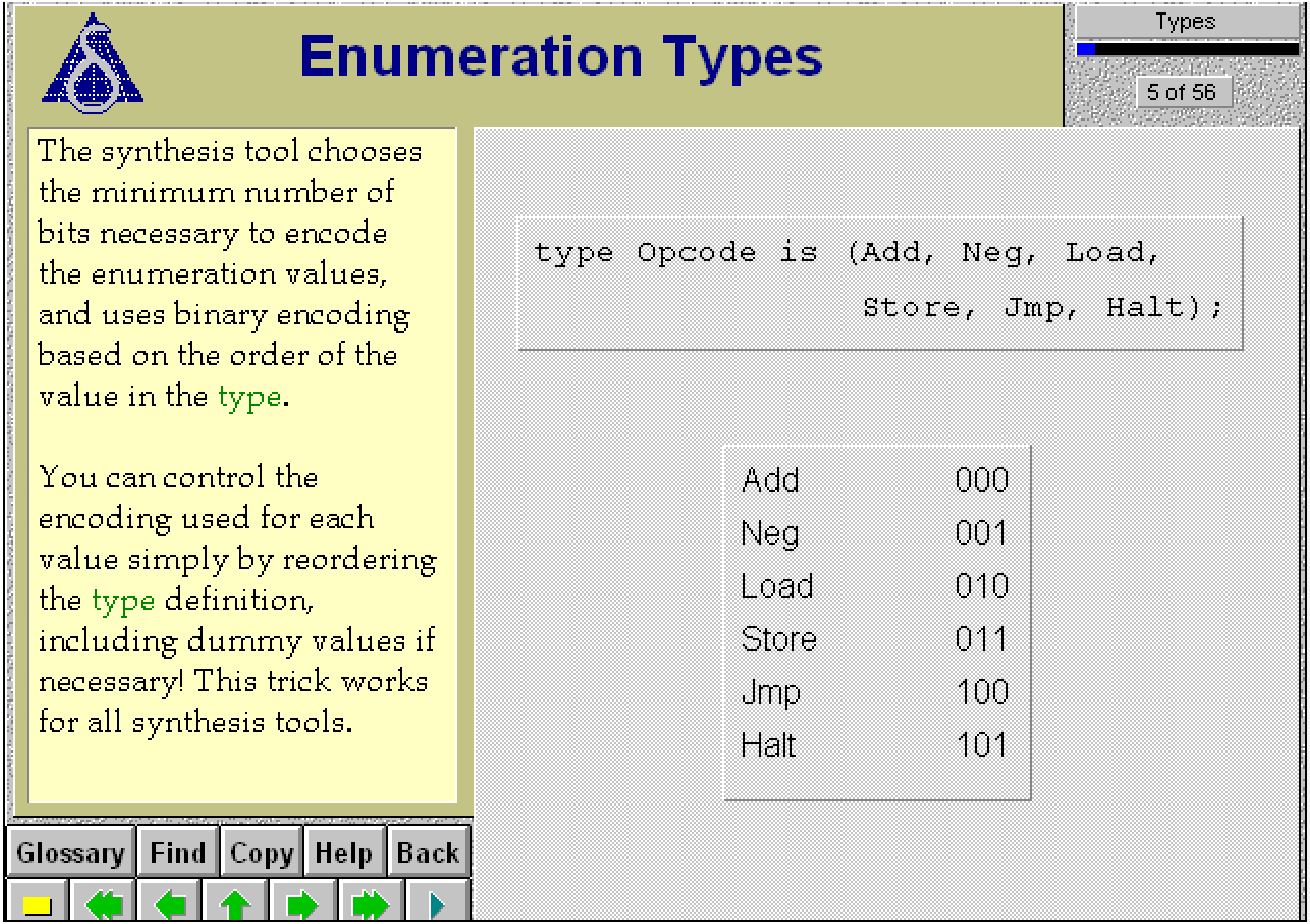
The synthesis tool will choose how many bits are needed to represent the type, and will create a bus of the appropriate width. One VHDL signal becomes several electrical connections.

```
architecture V1 of Ent is

  type Opcode is (Add, Neg, Load,
                  Store, Jmp, Halt);

  signal S: Opcode;
begin
  C1: Reg port map (..., S, ...);
  C2: Ctl port map (..., S, ...);
end;
```





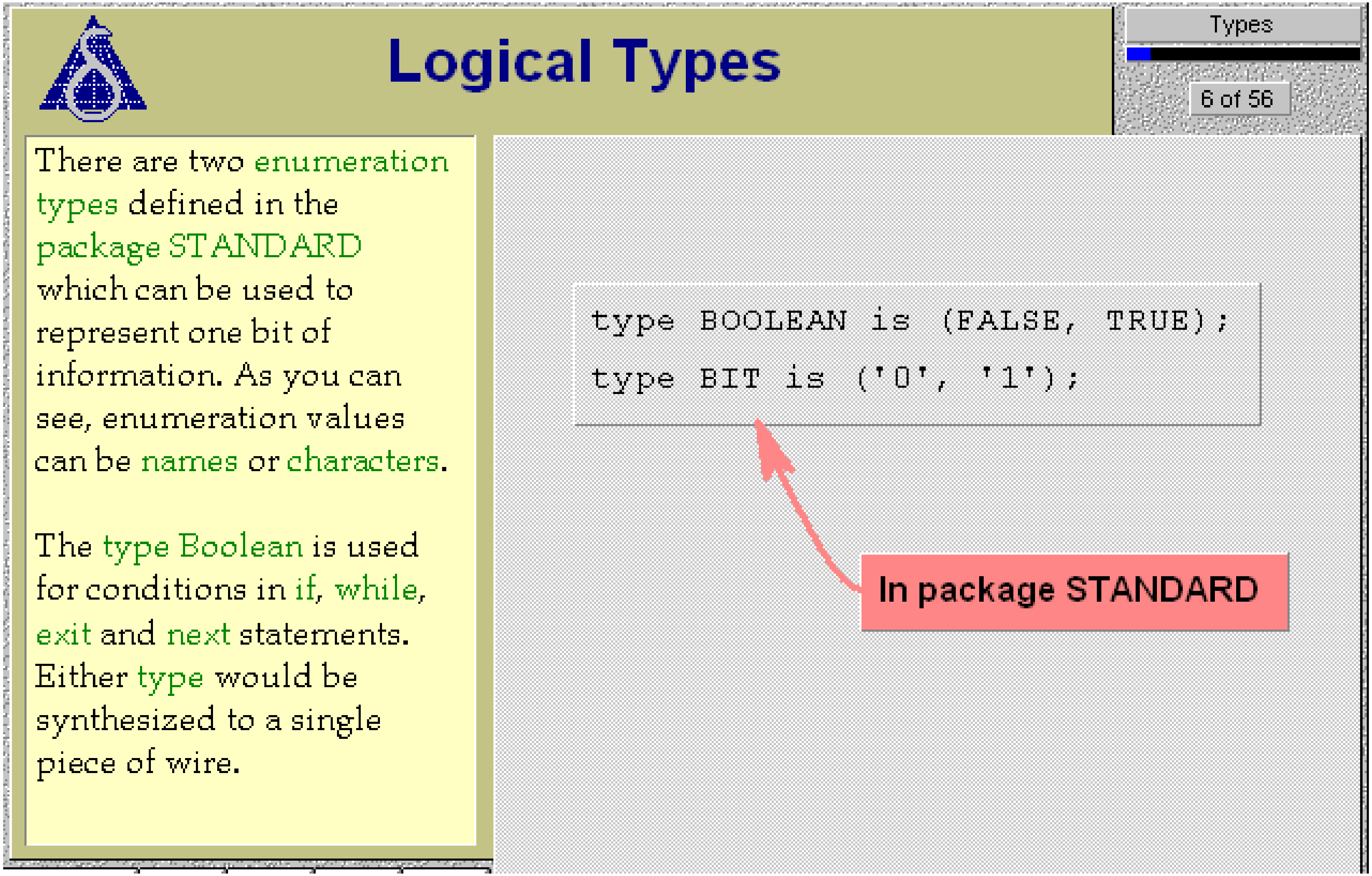
Enumeration Types

The synthesis tool chooses the minimum number of bits necessary to encode the enumeration values, and uses binary encoding based on the order of the value in the type.

You can control the encoding used for each value simply by reordering the type definition, including dummy values if necessary! This trick works for all synthesis tools.

```
type Opcode is (Add, Neg, Load,
                     Store, Jmp, Halt);
```

Add	000
Neg	001
Load	010
Store	011
Jmp	100
Halt	101



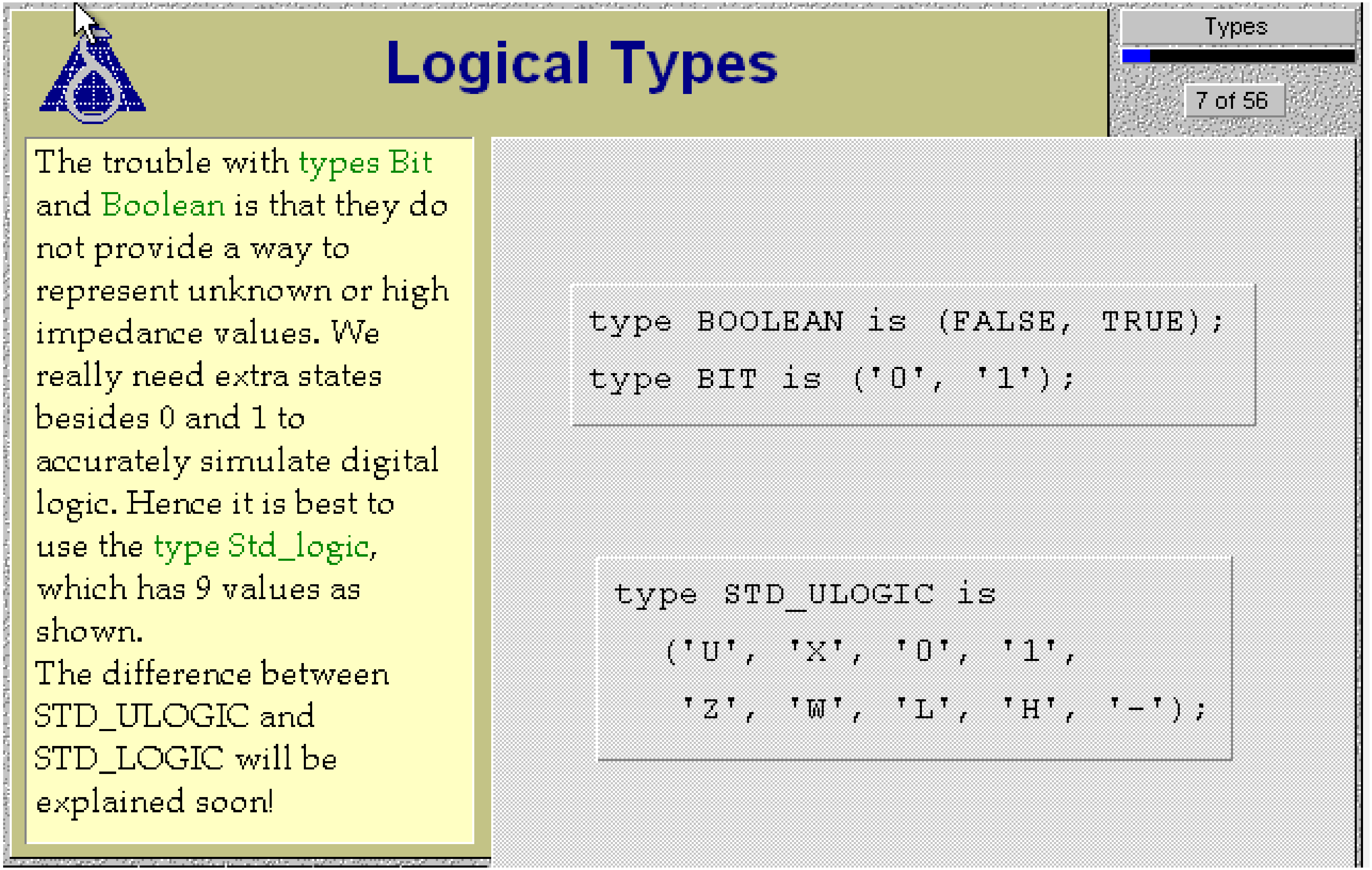
Logical Types

There are two enumeration types defined in the package STANDARD which can be used to represent one bit of information. As you can see, enumeration values can be names or characters.

The type Boolean is used for conditions in if, while, exit and next statements. Either type would be synthesized to a single piece of wire.

```
type BOOLEAN is (FALSE, TRUE);
type BIT is ('0', '1');
```

In package STANDARD

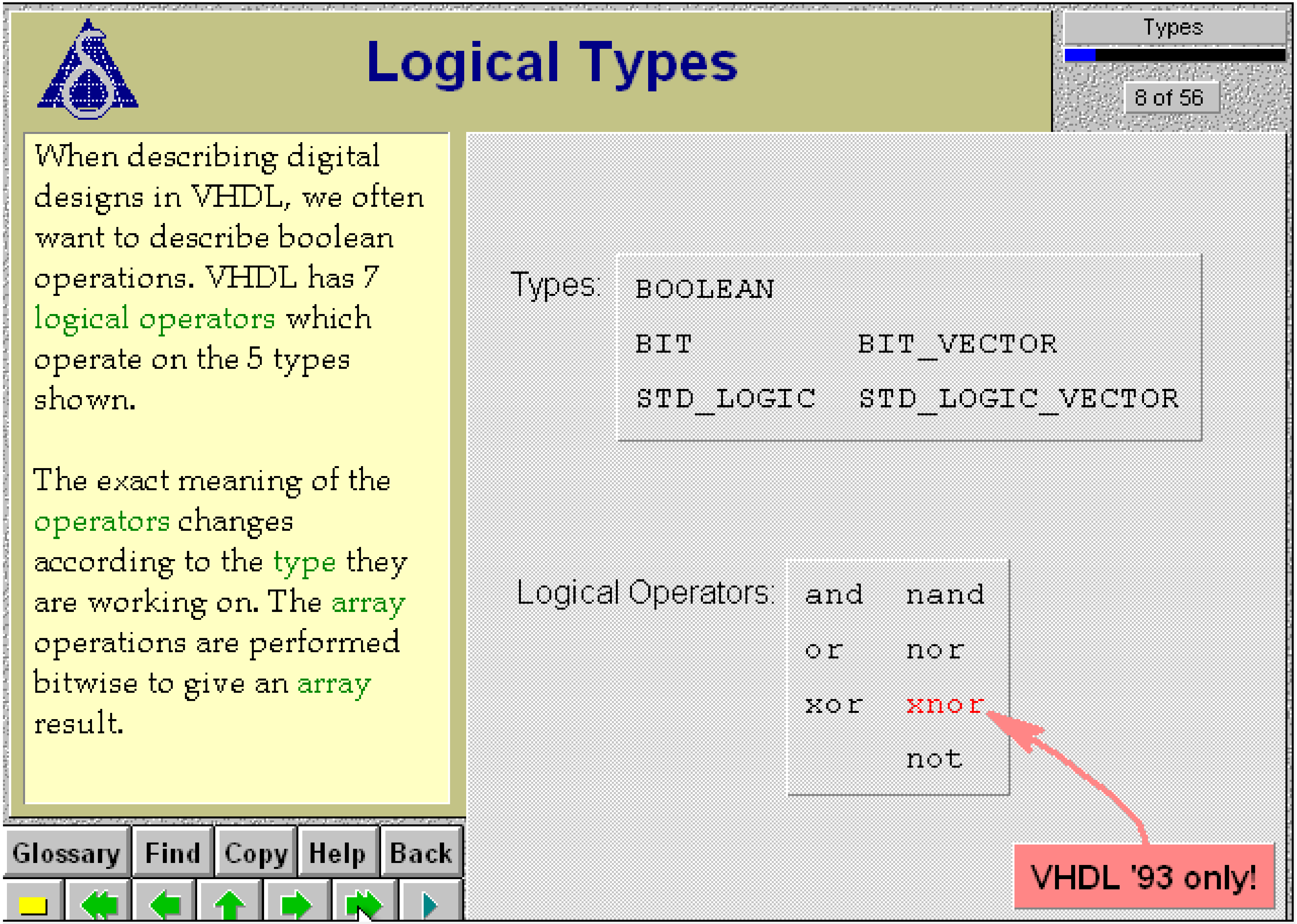


Logical Types

The trouble with types Bit and Boolean is that they do not provide a way to represent unknown or high impedance values. We really need extra states besides 0 and 1 to accurately simulate digital logic. Hence it is best to use the type Std_logic, which has 9 values as shown.
The difference between STD_ULOGIC and STD_LOGIC will be explained soon!

```
type BOOLEAN is (FALSE, TRUE);
type BIT is ('0', '1');
```

```
type STD_ULOGIC is
   ('U', 'X', '0', '1',
    'Z', 'W', 'L', 'H', '-');
```



Logical Types

When describing digital designs in VHDL, we often want to describe boolean operations. VHDL has 7 logical operators which operate on the 5 types shown.

The exact meaning of the operators changes according to the type they are working on. The array operations are performed bitwise to give an array result.

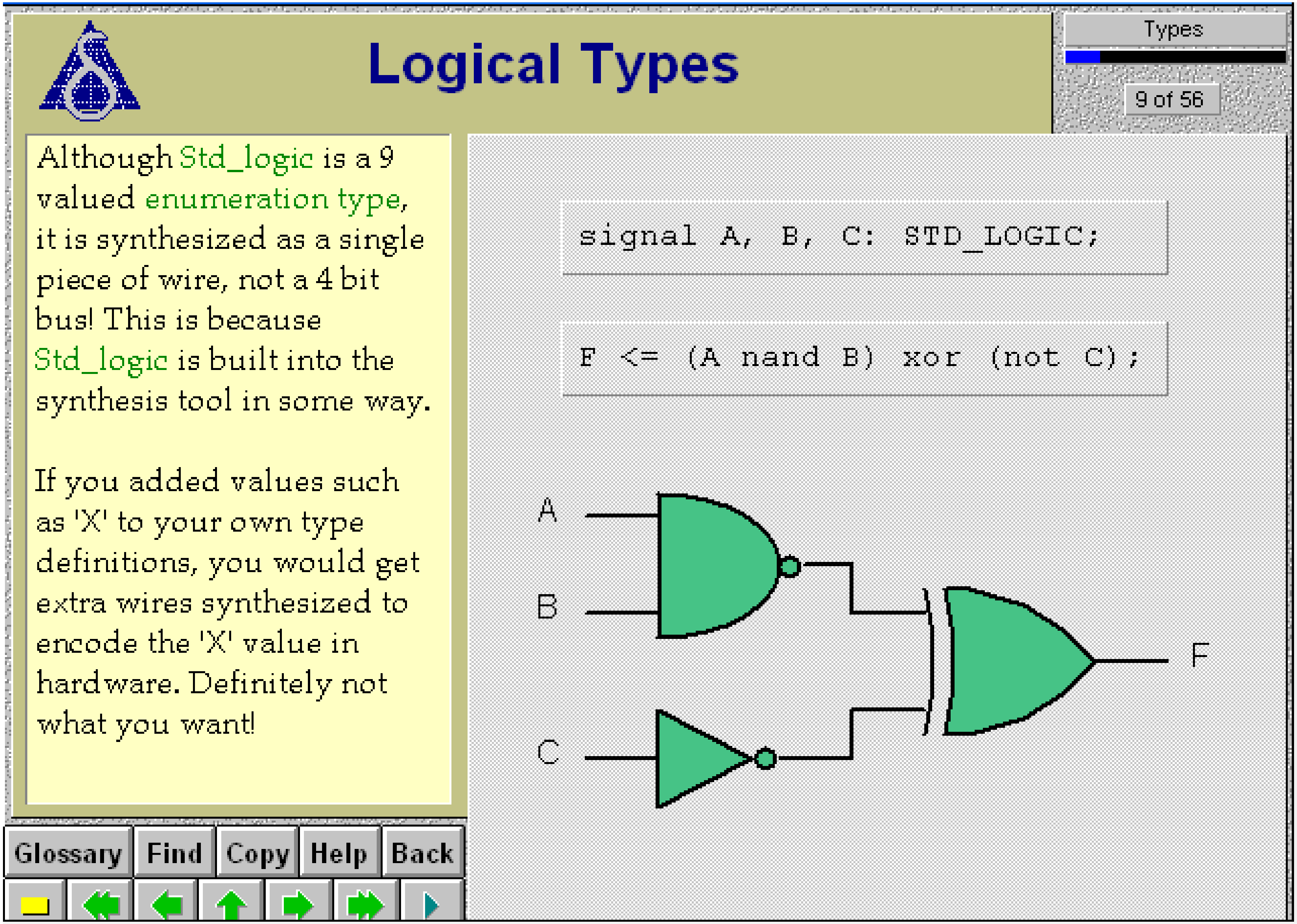
Types:

```
BOOLEAN
BIT          BIT_VECTOR
STD_LOGIC    STD_LOGIC_VECTOR
```

Logical Operators:

```
and   nand
or    nor
xor   xnor
      not
```

VHDL '93 only!



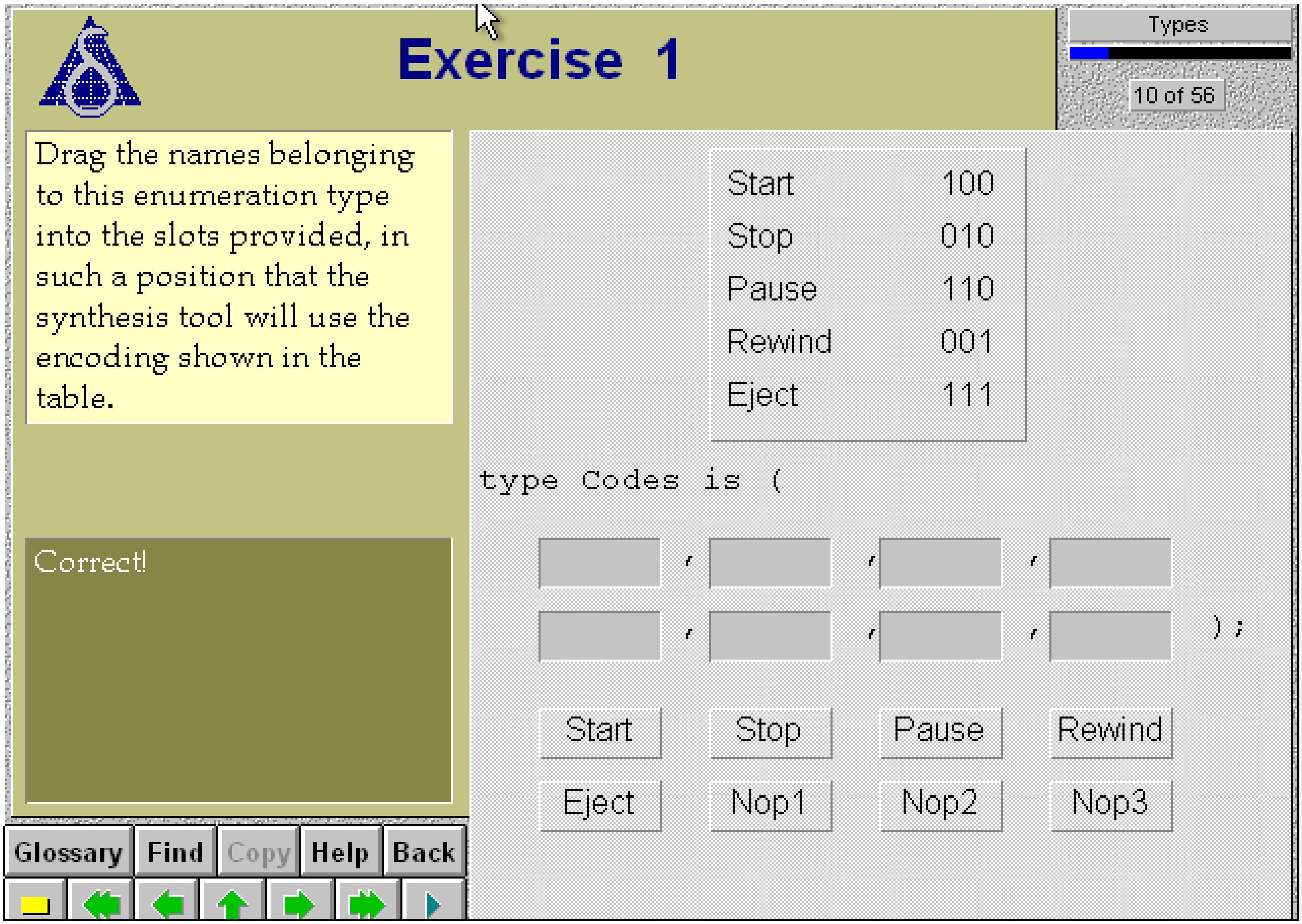
Logical Types

Although Std_logic is a 9 valued enumeration type, it is synthesized as a single piece of wire, not a 4 bit bus! This is because Std_logic is built into the synthesis tool in some way.

If you added values such as 'X' to your own type definitions, you would get extra wires synthesized to encode the 'X' value in hardware. Definitely not what you want!

```
signal A, B, C: STD_LOGIC;
```

```
F <= (A nand B) xor (not C);
```



Exercise 1

Types

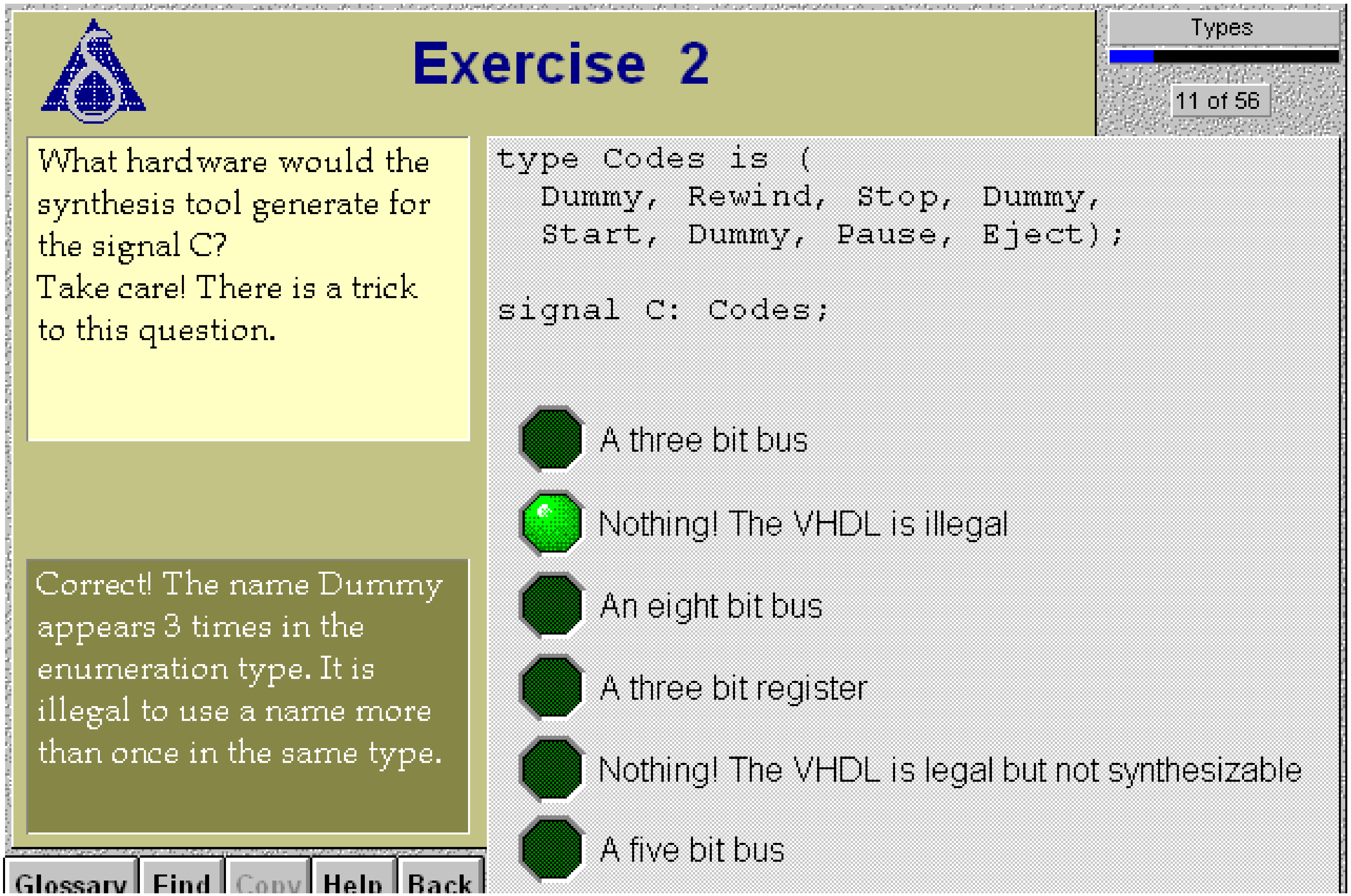
10 of 56

Drag the names belonging to this enumeration type into the slots provided, in such a position that the synthesis tool will use the encoding shown in the table.

Start	100
Stop	010
Pause	110
Rewind	001
Eject	111

```
type Codes is (
    ,  ,  ,
    ,  ,  ,   );
```

Correct!



Exercise 2

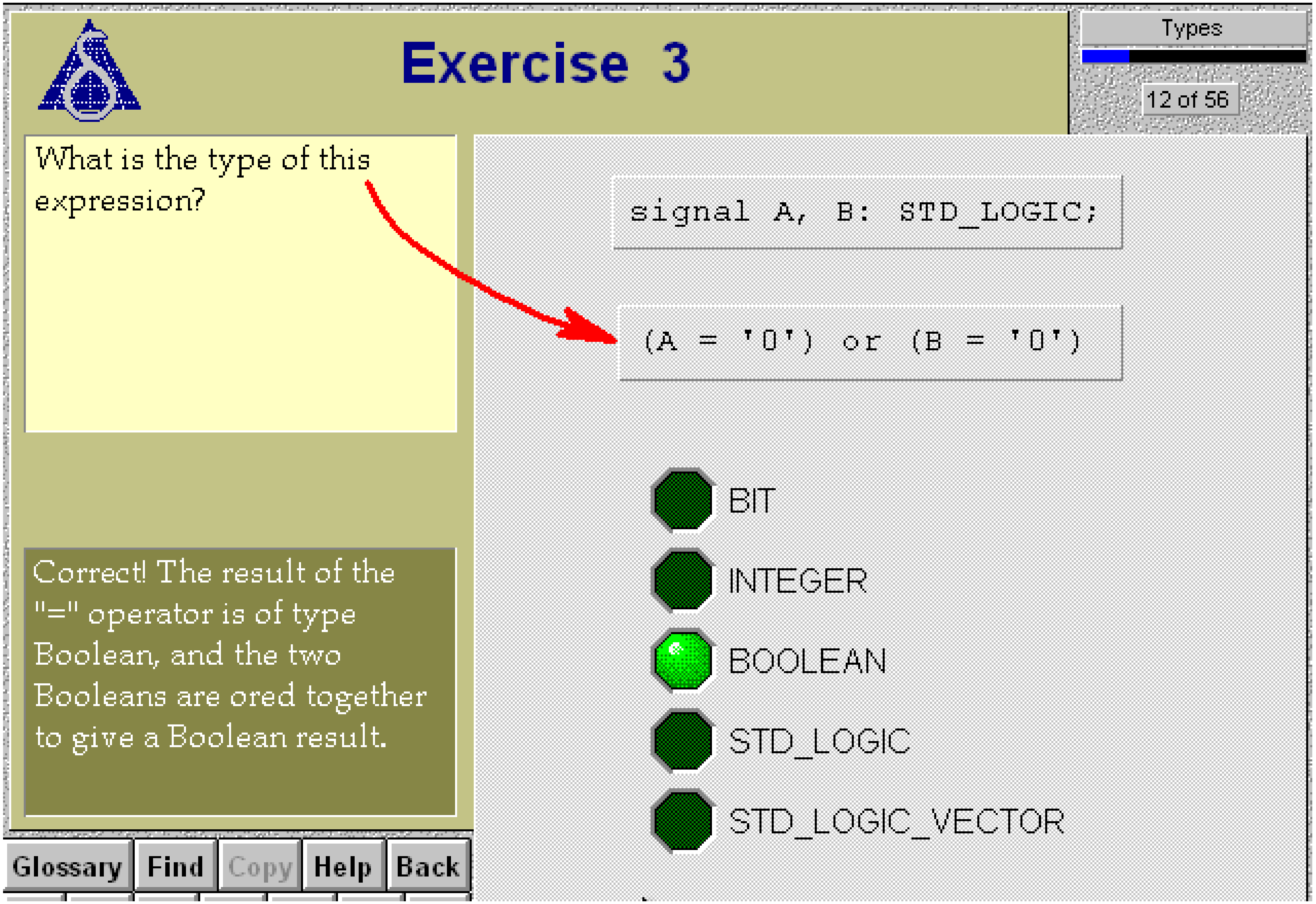
What hardware would the synthesis tool generate for the signal C?
Take care! There is a trick to this question.

```
type Codes is (
   Dummy, Rewind, Stop, Dummy,
   Start, Dummy, Pause, Eject);

signal C: Codes;
```

- A three bit bus
- [✓ Nothing! The VHDL is illegal]
- An eight bit bus
- A three bit register
- Nothing! The VHDL is legal but not synthesizable
- A five bit bus

Correct! The name Dummy appears 3 times in the enumeration type. It is illegal to use a name more than once in the same type.



Exercise 3

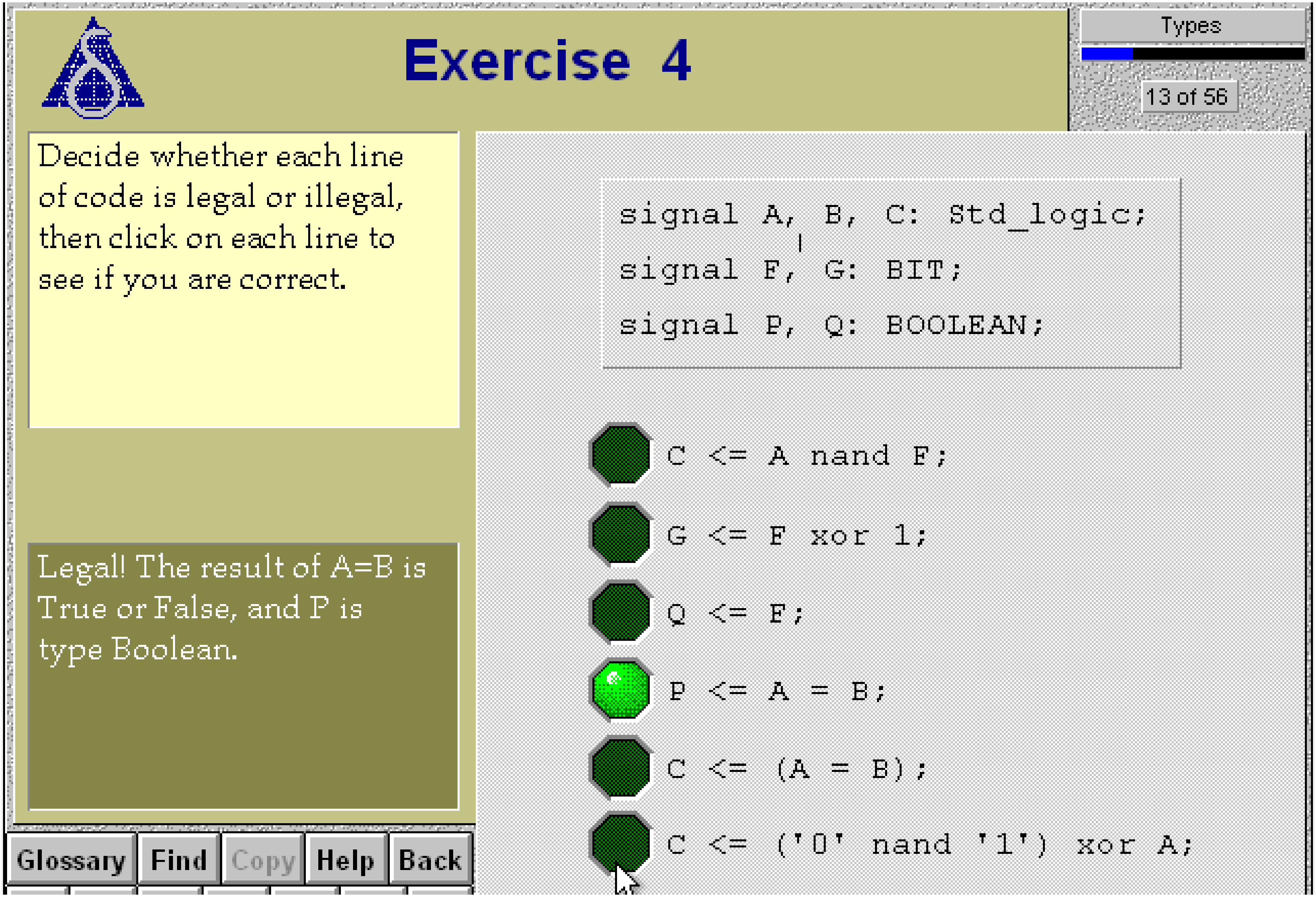
What is the type of this expression?

```
signal A, B: STD_LOGIC;
```

```
(A = '0') or (B = '0')
```

- ● BIT
- ● INTEGER
- [✓ BOOLEAN]
- ● STD_LOGIC
- ● STD_LOGIC_VECTOR

Correct! The result of the "=" operator is of type Boolean, and the two Booleans are ored together to give a Boolean result.



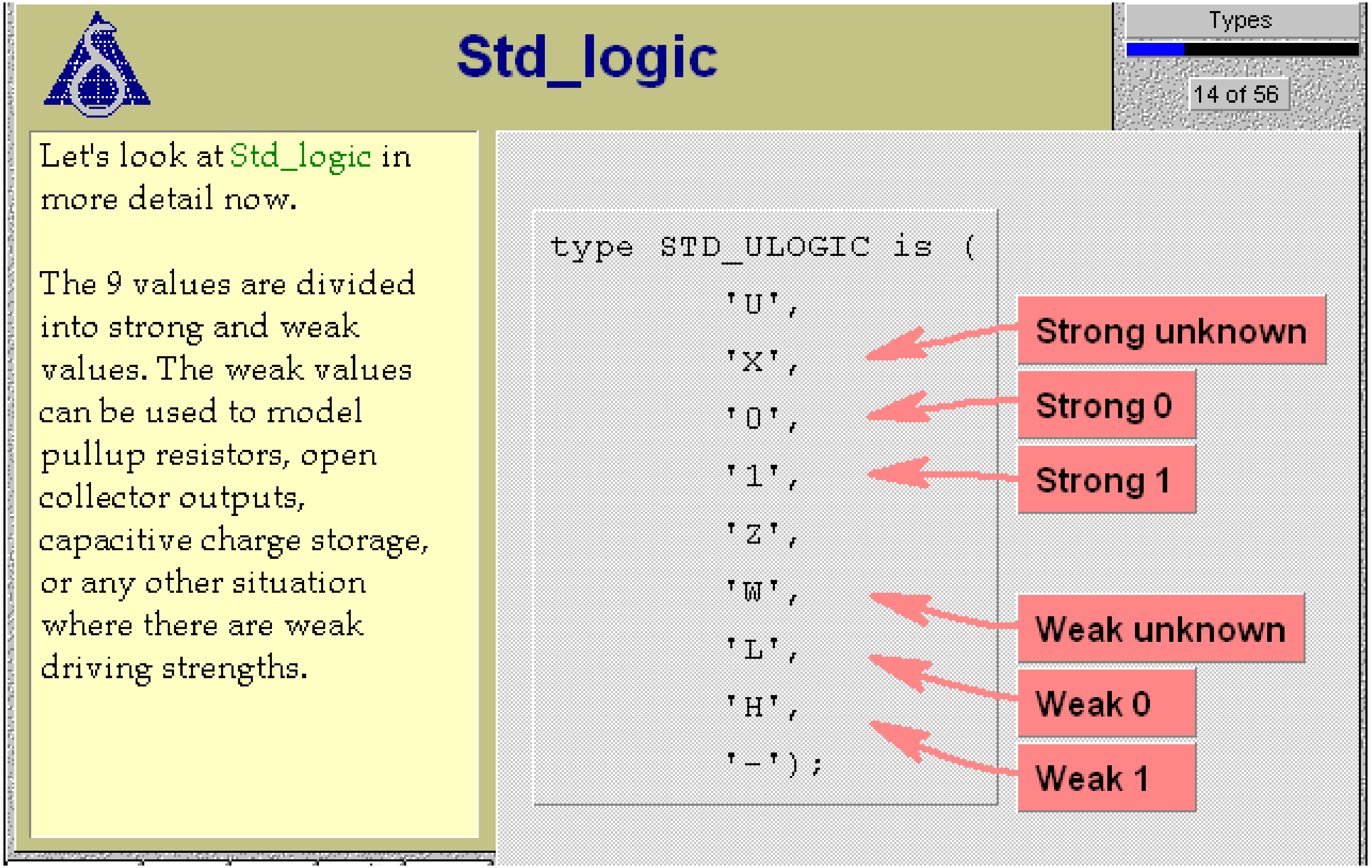
Exercise 4

Decide whether each line of code is legal or illegal, then click on each line to see if you are correct.

```
signal A, B, C: Std_logic;
signal F, G: BIT;
signal P, Q: BOOLEAN;
```

```
C <= A nand F;
G <= F xor 1;
Q <= F;
P <= A = B;
C <= (A = B);
C <= ('0' nand '1') xor A;
```

Legal! The result of A=B is True or False, and P is type Boolean.

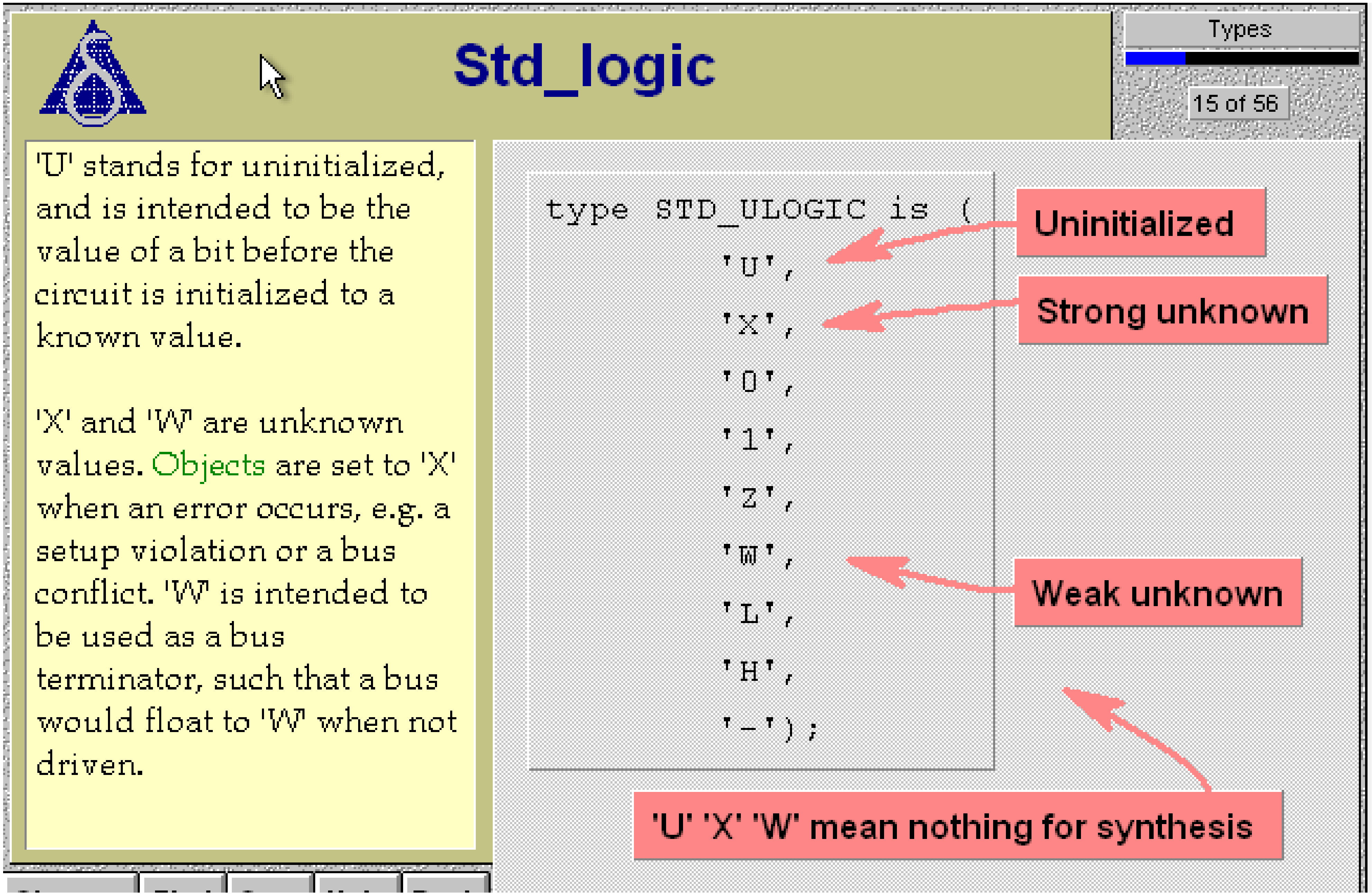


Std_logic

Let's look at Std_logic in more detail now.

The 9 values are divided into strong and weak values. The weak values can be used to model pullup resistors, open collector outputs, capacitive charge storage, or any other situation where there are weak driving strengths.

```
type STD_ULOGIC is (
        'U',
        'X',
        '0',
        '1',
        'Z',
        'W',
        'L',
        'H',
        '-');
```



Std_logic

'U' stands for uninitialized, and is intended to be the value of a bit before the circuit is initialized to a known value.

'X' and 'W' are unknown values. Objects are set to 'X' when an error occurs, e.g. a setup violation or a bus conflict. 'W' is intended to be used as a bus terminator, such that a bus would float to 'W' when not driven.

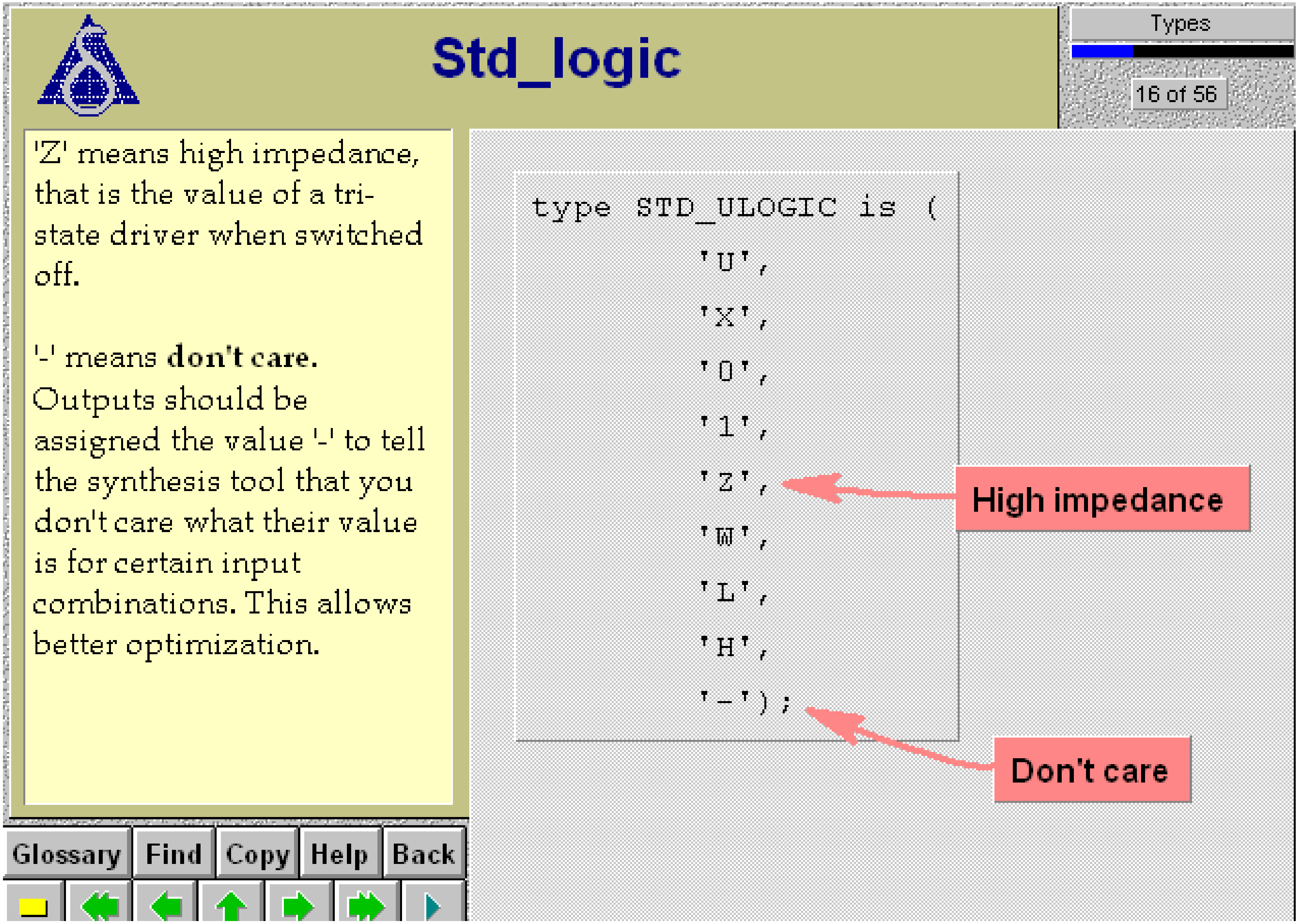
```
type STD_ULOGIC is (
          'U',
          'X',
          '0',
          '1',
          'Z',
          'W',
          'L',
          'H',
          '-');
```

Uninitialized

Strong unknown

Weak unknown

'U' 'X' 'W' mean nothing for synthesis



Std_logic

'Z' means high impedance, that is the value of a tri-state driver when switched off.

'-' means **don't care.** Outputs should be assigned the value '-' to tell the synthesis tool that you don't care what their value is for certain input combinations. This allows better optimization.

```
type STD_ULOGIC is (
          'U',
          'X',
          '0',
          '1',
          'Z',
          'W',
          'L',
          'H',
          '-');
```

Std_logic

In this example, setting certain bits of OP to '-' gives the optimizer maximum flexibility in reducing the number of logic gates needed to implement the circuit.

Using don't cares in this way avoids having to over specify the design.

```
case IP is
when "00" =>
   OP <= "0--1";
when "01" =>
   OP <= "1-0-";
when "10" =>
   OP <= "--10";
when others =>
   OP <= "101-";
end case;
```

Std_logic

The '-' value **cannot** be used as an input don't care.

The example shown here will not work, because comparing a value with '-' will only give a match if the value actually is '-'. The don't care value is not a wild card "match anything" value!

```
case IP is
when "0--" =>
   OP <= "00";
when "10-" =>
   OP <= "01";
when "1-1" =>
   OP <= "10";
when others =>
   OP <= "11";
end case;
```

Only IP = "0--" will match this branch!

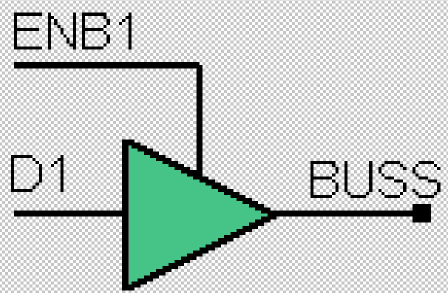
IP = "000" would match this branch!

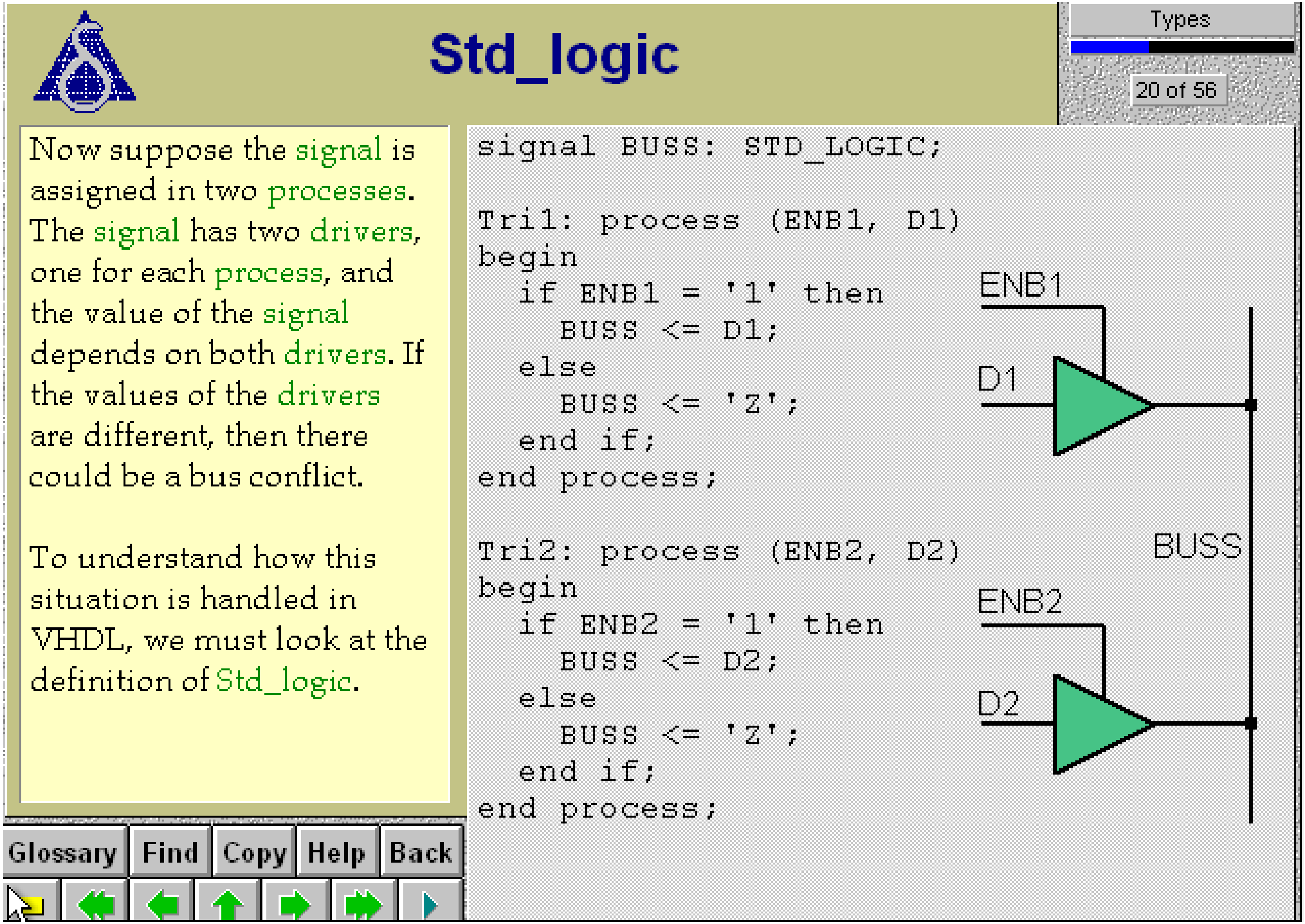
Std_logic

'Z' can be used to model a tristate driver, as shown. Most synthesis tools would accept this VHDL too, and would infer the tristate driver from the conditional assignment of the high impedance value to BUSS.

When a signal is assigned in a process, that process has a VHDL driver for the signal. The driver is an invisible, conceptual thing that stores future events for the signal.

```
Tri1: process (ENB1, D1)
begin
  if ENB1 = '1' then
    BUSS <= D1;
  else
    BUSS <= 'Z';
  end if;
end process;
```





Std_logic

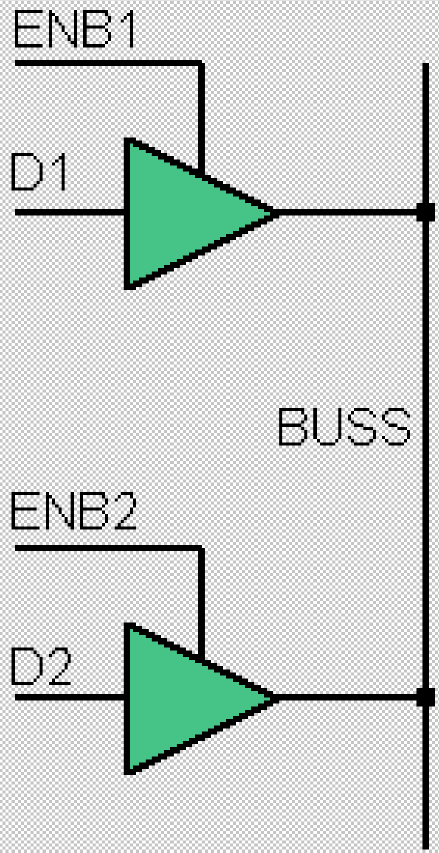
Now suppose the signal is assigned in two processes. The signal has two drivers, one for each process, and the value of the signal depends on both drivers. If the values of the drivers are different, then there could be a bus conflict.

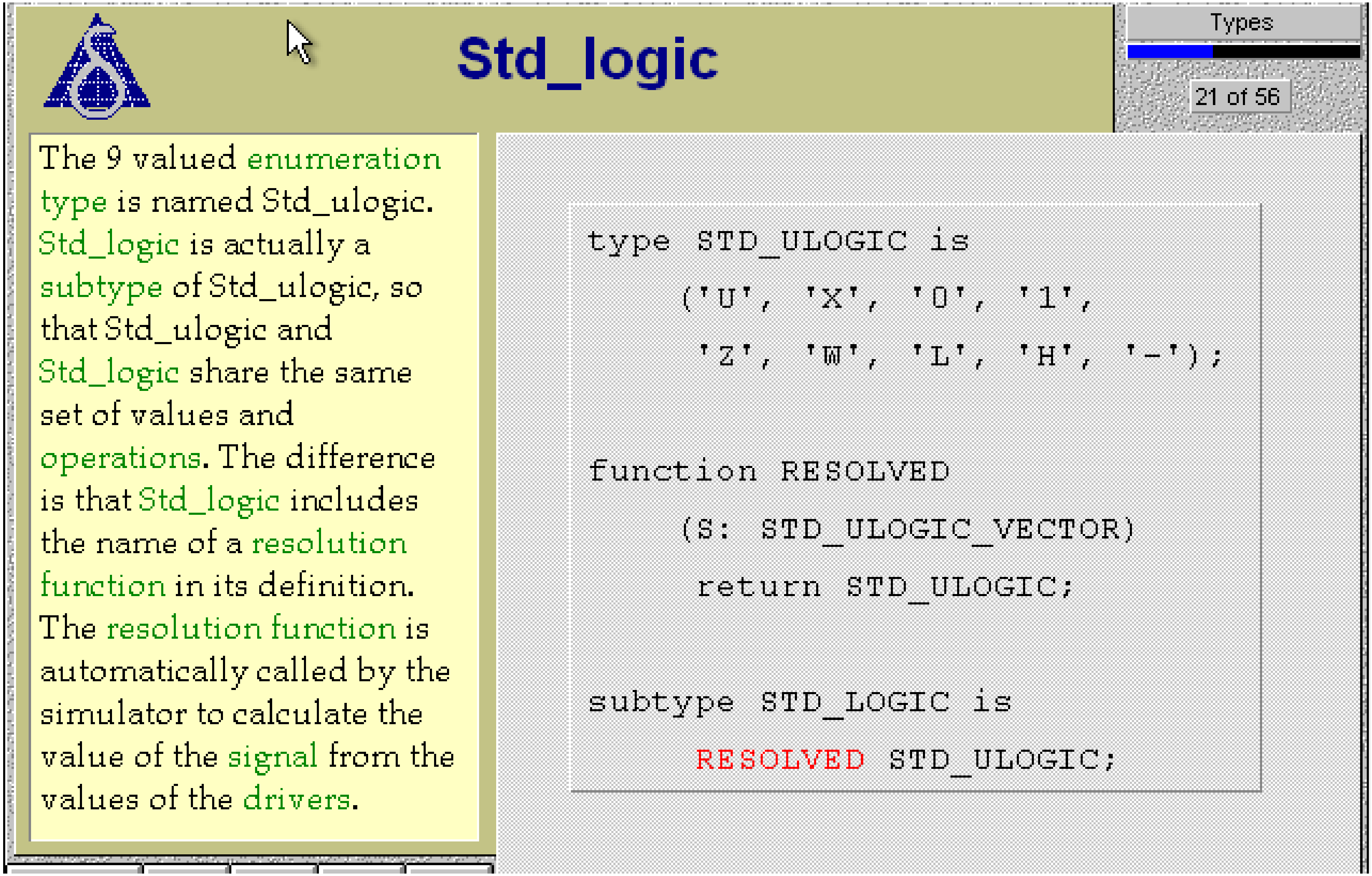
To understand how this situation is handled in VHDL, we must look at the definition of Std_logic.

```
signal BUSS: STD_LOGIC;

Tri1: process (ENB1, D1)
begin
   if ENB1 = '1' then
      BUSS <= D1;
   else
      BUSS <= 'Z';
   end if;
end process;

Tri2: process (ENB2, D2)
begin
   if ENB2 = '1' then
      BUSS <= D2;
   else
      BUSS <= 'Z';
   end if;
end process;
```





Std_logic

The 9 valued enumeration type is named Std_ulogic. Std_logic is actually a subtype of Std_ulogic, so that Std_ulogic and Std_logic share the same set of values and operations. The difference is that Std_logic includes the name of a resolution function in its definition. The resolution function is automatically called by the simulator to calculate the value of the signal from the values of the drivers.

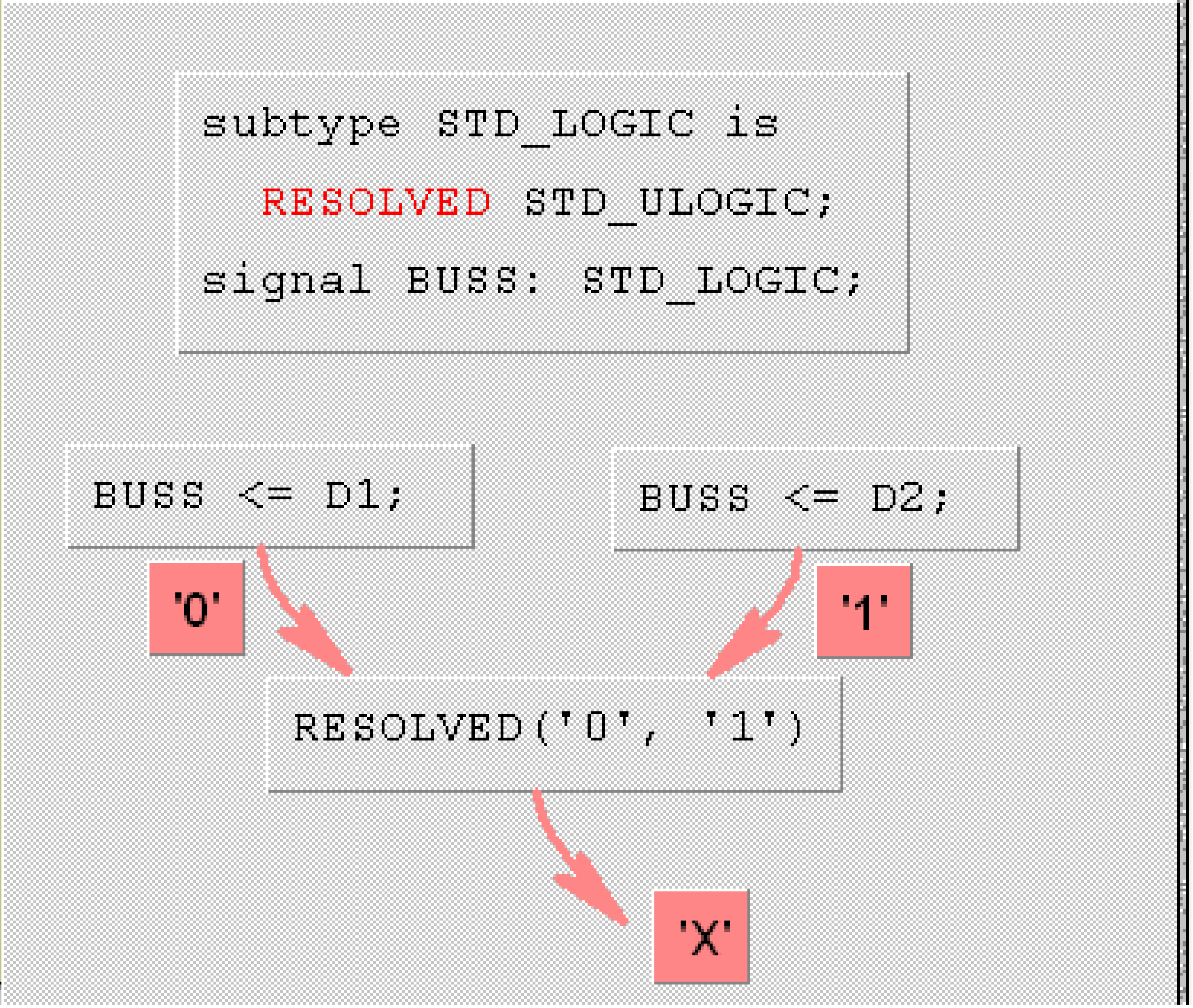
```
type STD_ULOGIC is
     ('U', 'X', '0', '1',
      'Z', 'W', 'L', 'H', '-');

function RESOLVED
     (S: STD_ULOGIC_VECTOR)
      return STD_ULOGIC;

subtype STD_LOGIC is
      RESOLVED STD_ULOGIC;
```

Std_logic

Whenever a signal assignment to BUSS is executed, the simulator passes the values of the two drivers into the resolution function RESOLVED, which returns one value for the signal. The number of values passed to the resolution function depends on the number of drivers. This is only possible because Std_logic has a resolution function. Std_ulogic could not be used here!

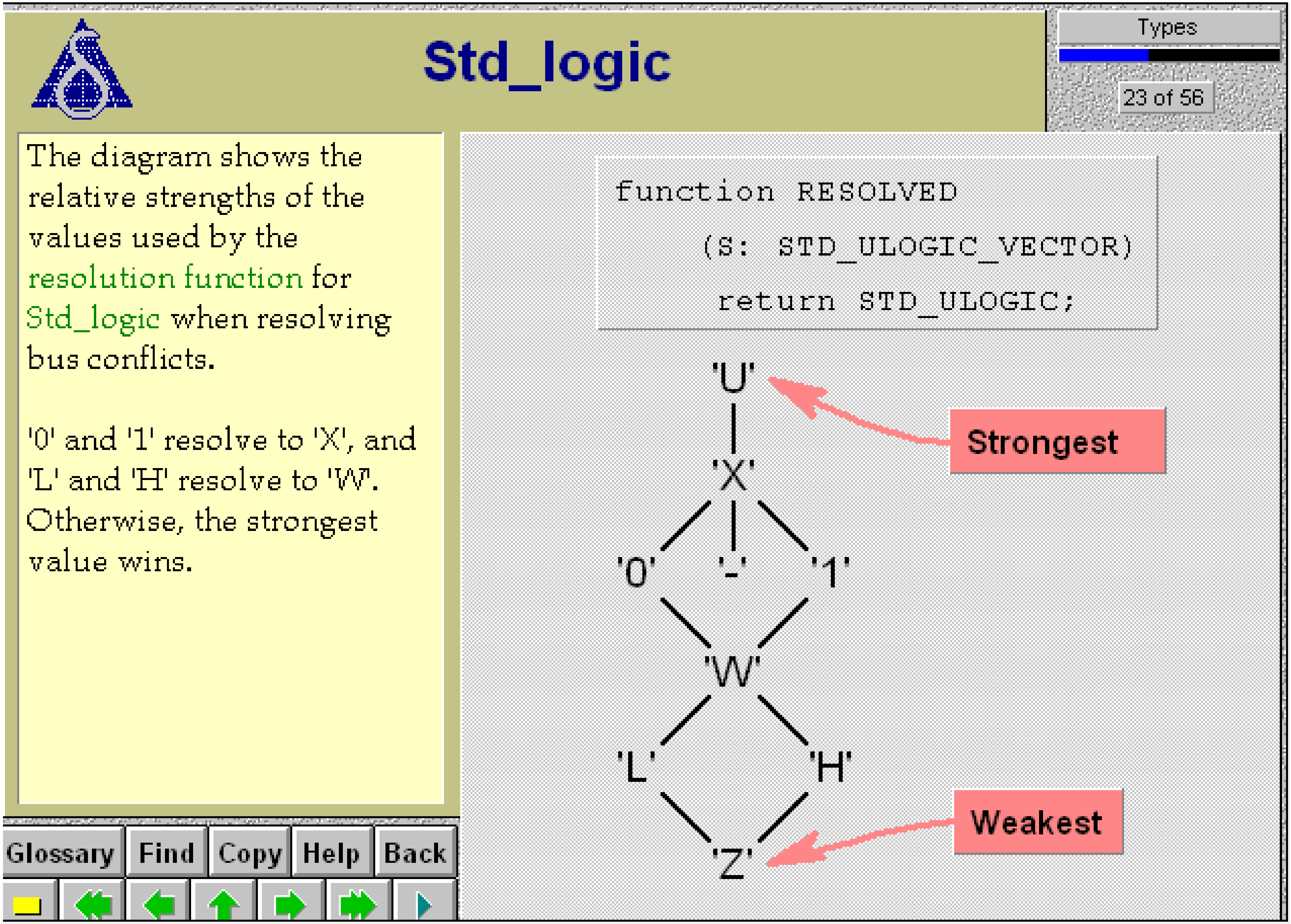


```
subtype STD_LOGIC is
  RESOLVED STD_ULOGIC;
signal BUSS: STD_LOGIC;
```

```
BUSS <= D1;
```

```
BUSS <= D2;
```

```
RESOLVED('0', '1')
```

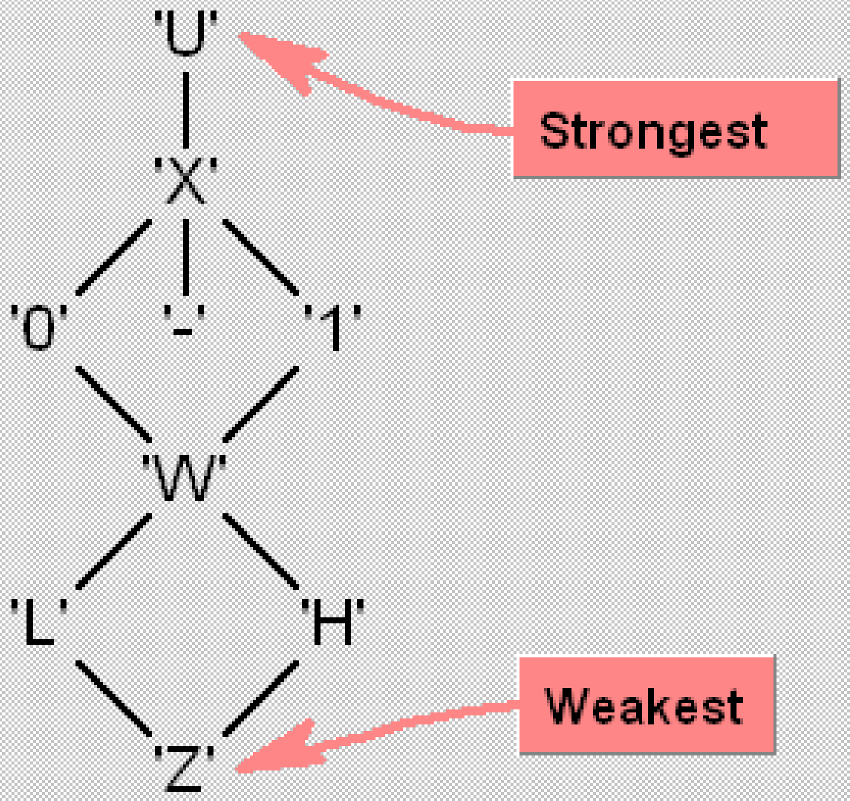


Std_logic

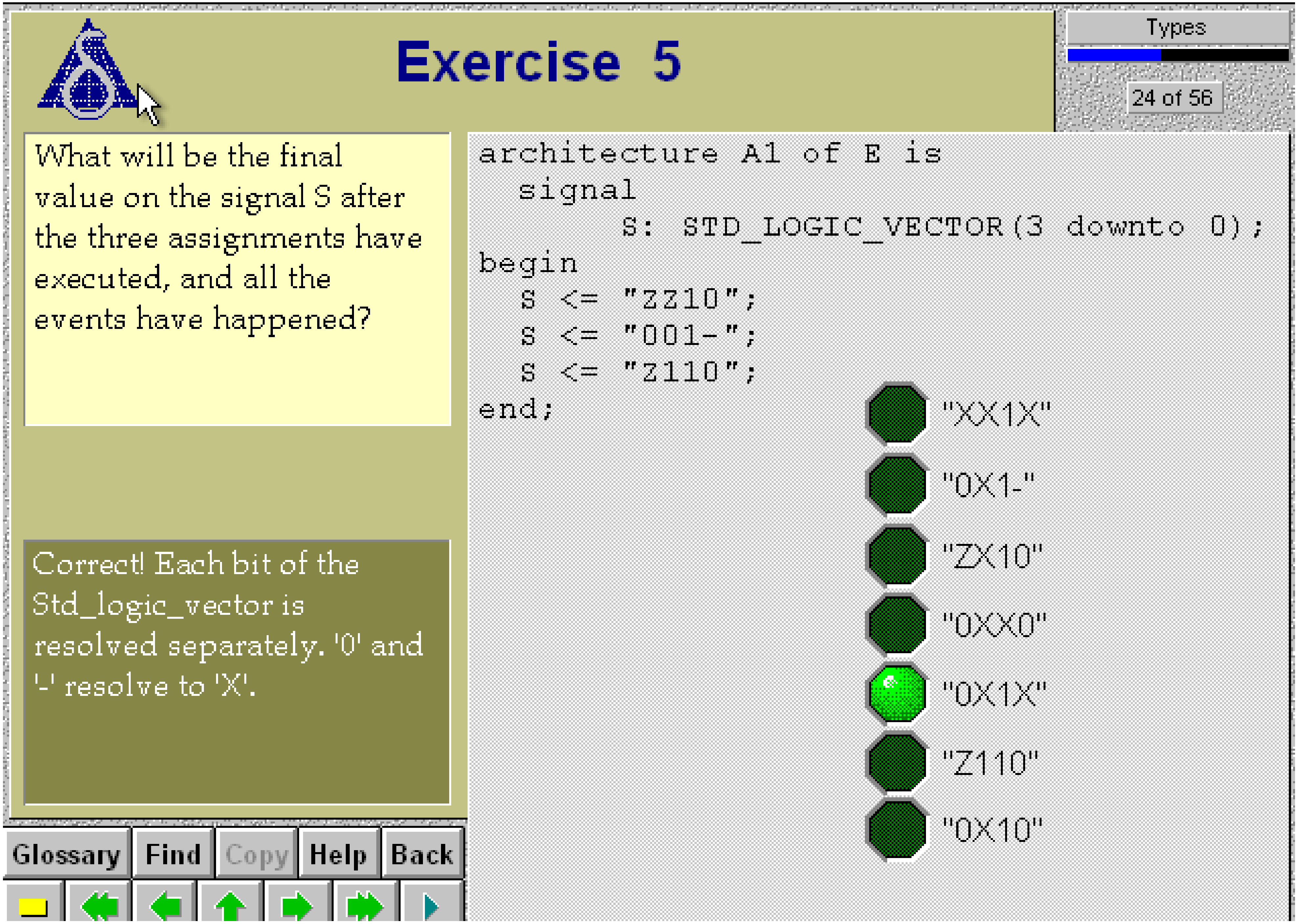
The diagram shows the relative strengths of the values used by the resolution function for Std_logic when resolving bus conflicts.

'0' and '1' resolve to 'X', and 'L' and 'H' resolve to 'W'. Otherwise, the strongest value wins.

```
function RESOLVED
     (S: STD_ULOGIC_VECTOR)
      return STD_ULOGIC;
```



$$\begin{tikzcd}
 & \text{'U'} \arrow[d, dash] & \\
 & \text{'X'} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\text{'0'} \arrow[dr, dash] & \text{'-'} & \text{'1'} \arrow[dl, dash] \\
 & \text{'W'} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{'L'} \arrow[dr, dash] & & \text{'H'} \arrow[dl, dash] \\
 & \text{'Z'} &
\end{tikzcd}$$



Exercise 5

What will be the final value on the signal S after the three assignments have executed, and all the events have happened?

```
architecture A1 of E is
  signal
        S: STD_LOGIC_VECTOR(3 downto 0);
begin
  S <= "ZZ10";
  S <= "001-";
  S <= "Z110";
end;
```

- ● "XX1X"
- ● "0X1-"
- ● "ZX10"
- ● "0XX0"
- ● "0X1X"
- ● "Z110"
- ● "0X10"

Correct! Each bit of the Std_logic_vector is resolved separately. '0' and '-' resolve to 'X'.