VHDL Concepts

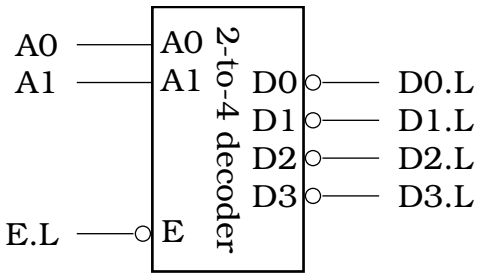
- Before we begin to look at the details of designing and describing hardware using VHDL, it'll be helpful to explore the organising concepts of the language.
- VHDL was designed so that it could be used to describe hardware at all levels of the design hierarchy — from analog circuit designs right up through computer systems.
- As a consequence, it is very general, and there are very few built-in preconceptions about the hardware.
  - To fully exploit the capabilities of VHDL, you really must think in a hierarchical manner, building useful components at each level and using them as black boxes at higher levels.
  - Fortunately, others have already done much of this groundwork and made it available for use through a number of standard library packages which define useful data types and operations. This is analogous to the C or C++ standard libraries.
- These notes attempt to convey a feeling for the overall structure of a VHDL description. A 2-to-4 decoder will be used as a running example[1]. It's sufficiently complicated to show some internal structure, but sufficiently small to be held in the mind without difficulty. We'll gloss over some of the details of syntax, and many of the bells and whistles that can be attached to various pieces of the description, in order to remain focused on the overall structure.
- Information about VHDL and VHDL simulation is drawn largely from Ashenden [1].

Entities and Architectures

- In VHDL, the thing to be designed is an *entity*.
- When doing a design, it's a good idea to distinguish the interface (how a thing appears to the outside world) from the implementation (how the desired behaviour is achieved). All of you are familiar with this principle — it's a basic tenet of good software design, and it applies equally well to hardware design.

[1] Adapted from [3], §§3-5 and 3-13.

- In VHDL, an *entity-declaration* gives a name to the entity we are designing, and defines its interface to the outside world in terms of a list of ports.
  - An *entity-declaration* is the textual equivalent of the symbol for a 2-to-4 decoder.



    The symbol shows the interface: enable input `E.L`, select inputs `A0` and `A1`, and outputs `D0.L` – `D3.L`. The bubbles at the enable input and the outputs indicate that these signals are complemented (*i.e.*, when an output is selected, it will take on the value 0). The suffix `.L` added to the signal name serves as a reminder that the signal takes the value 0 when active.

  - In VHDL, we have

```
entity decoder2to4 is
   port (E_L, A0, A1 : in std_logic ;
         D0_L, D1_L, D2_L, D3_L : out std_logic) ;
end decoder2to4 ;
```

    VHDL doesn't allow identifiers of the form `E.L`, so we'll use the form `E_L` instead.

- The operation of an entity is described in an *architecture-body*, comprised of declarations (an *architecture-declarative-part*) and the actual description of the entity's operation (an *architecture-statement-part*).

- How can we describe the operation of this decoder? One way would be to write some logic equations:

```
D0_L <= not (not A0 and not A1 and not E_L) ;
D1_L <= not (A0 and not A1 and not E_L) ;
D2_L <= not (not A0 and A1 and not E_L) ;
D3_L <= not (A0 and A1 and not E_L) ;
```

The example takes this a bit further. It begins to hint at how we might implement the decoder by declaring some additional signals (*i.e.*, wires). Here's the full *architecture-body*:

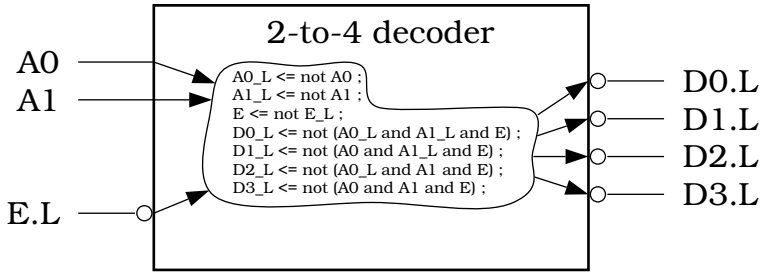
```
architecture dataflow_1 of decoder2to4 is

  signal E, A0_L, A1_L : std_logic ;

begin

   A0_L <= not A0 ;
   A1_L <= not A1 ;
   E <= not E_L ;
   D0_L <= not (A0_L and A1_L and E) ;
   D1_L <= not (A0 and A1_L and E) ;
   D2_L <= not (A0_L and A1 and E) ;
   D3_L <= not (A0 and A1 and E) ;

end dataflow_1 ;
```

These logic equations are sufficient to precisely define the operation of the decoder.

In terms of the VHDL, we've given the architecture a name (`dataflow_1`), declared some signals (`E`, `A0_L`, and `A1_L`) in the *architecture-declarative-part*, and specified the operation in the *architecture-statement-part*.

This description is (almost) self-contained — except for the `std_logic` data type, all of the operations we've used are VHDL primitives. With only a little additional work, we could simulate this description to see if it behaves properly.

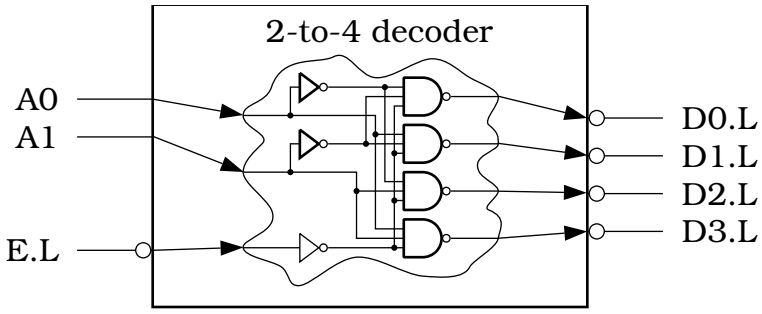
❄ Visually, it's as if we'd done something like this:



❄ Now, suppose that we do some work and produce a gate level implementation of the 2-to-4 decoder. We might want to write a description of the decoder

that reflects the gate level design, so that we can simulate it and confirm that the behaviour is still correct.

- Visually, we'd like to have this



- The corresponding VHDL *architecture-body* is

```
architecture structural_1 of decoder2to4 is

  component NOT1
     port (in1 : in std_logic ;
           out1 : out std_logic) ;
  end component ;

  component NAND3
     port (in1, in2, in3 : in std_logic ;
           out1 : out std_logic) ;
  end component ;

  signal E, A0_L, A1_L: std_logic ;

begin

   g0: NOT1 port map (in1 => A0, out1 => A0_L) ;
   g1: NOT1 port map (in1 => A1, out1 => A1_L) ;
   g2: NOT1 port map (in1 => E_L, out1 => E) ;

   g3: NAND3 port map (in1 => A0_L, in2 => A1_L,
                       in3 => E, out1 => D0_L) ;
   g4: NAND3 port map (in1 => A0, in2 => A1_L,
                       in3 => E, out1 => D1_L) ;
   g5: NAND3 port map (in1 => A0_L, in2 => A1,
                       in3 => E, out1 => D2_L) ;
   g6: NAND3 port map (in1 => A0, in2 => A1,
                       in3 => E, out1 => D3_L) ;

end structural_1 ;
```

The signals `A0`, `A1`, `E_L`, and `D0_L` – `D3_L` are the ones named in the *entity-declaration*.

Note that we've specified some generic logic gates (a 3-input `NAND`, and an inverter), but haven't specified their behaviour. Before we can simulate, we need to specify entities and architectures for the `NOT1` and `NAND3` components. Typically, these will be contained in a library. The IEEE has defined a standard set of libraries for VHDL which contain, among other things, entities and architectures for standard logic gates and some simple register-transfer level components. Hardware vendors and IC manufacturers may also provide libraries, to ensure that the descriptions are accurate for their components.

Structural descriptions in VHDL will give you a real appreciation of the expression "a picture is worth a thousand words."

- This ability to define the *architecture* of a given entity in many different ways allows VHDL to satisfy the need to describe behaviour in different ways at different stages of the design process.
- In VHDL terms, each *architecture-statement-part* is comprised of one or more *concurrent-statement*s.
  - *Concurrent* is important here. In hardware, all components are active simultaneously. VHDL reflects this. We'll pursue this in more detail when we discuss VHDL simulation later in these notes.
- One common form of *concurrent-statement* is the *concurrent-signal-assignment-statement*, which we used in the `dataflow_1` architecture. For example,

```
D0_L <= not (not A0 and not A1 and not E_L) ;
```

- Another common form of *concurrent-statement* is the *process-statement*, which is analogous to a subroutine. A process can declare local variables and describes behaviour using sequential statements that are very much like those of a standard imperative programming language (assignment, if-then-else, loop, case, *etc.*).

  Conceptually, a *concurrent-signal-assignment-statement* is just a shorthand for a very simple instance of a *process-statement*.

```
D0_L <= not (not A0 and not A1 and not E_L) ;
```

is equivalent to

```
setD0 : process ( A0, A1, E_L ) is
begin
  D0_L <= not (not A0 and not A1 and not E_L) ;
end process setD0 ;
```

Notice that a *signal-assignment-statement* can occur in two contexts, and the context changes the meaning. It can occur as part of the sequential statements of a process body (a plain *signal-assignment-statement*). Or, it can be part of the concurrent statements that make up an *architecture-statement-part* (a *concurrent-signal-assignment-statement*), in which case it actually represents an entire *process-statement*.

- A third common form of *concurrent-statement* is the *component-instantiation-statement*, which we used in the `structural_1` architecture. This type of statement represents an instantiation of a entity whose behaviour is defined elsewhere. For example,

```
g4: NAND3 port map (in1 => A0, in2 => A1_L,
                    in3 => E, out1 => D1_L) ;
```

- Concurrent statements communicate by means of *signals*, which represent the inputs, outputs, and wires of digital hardware. Each *entity-declaration* declares the input and output ports of the entity. If additional wires are needed in the description of the entity, they can be declared in the *architecture-declarative-part* of the *architecture-body*.

- Recognise that, at some level, the behaviour of each entity must be described; this in turn could be a behavioural or structural description. Ultimately, one comes to a description which is expressed purely in terms of VHDL primitives which are directly recognised by a VHDL simulator.

Configurations

- You've defined your entities, and you've supplied various architectures that describe the operation of each entity. There's one more thing you may need to do before you can actually simulate your VHDL description: You may have to supply configuration information.

- By *configuration*, we mean the process of selecting a specific architecture body for each instantiation of each entity used in the VHDL description. (Clearly this can get seriously recursive.)
- For example, if your CPU structural architecture declares a register as a component, and then uses it (instantiates it) in several places, you can bind all of those instances of the register to one underlying architecture, or bind each one individually to different underlying architectures. If those underlying architectures themselves instantiate components, you must configure them.
- One place where we can take advantage of this is at the very top level of the VHDL description. The entity that you're designing, whatever it is, likely has some inputs and outputs. (Otherwise it's not very interesting, eh?) To simulate that entity, you need some way of exercising those inputs and outputs.
  - In VHDL, the convention is to create a *test bench*, an entity whose purpose is to instantiate your entity, supply it with a sequence of inputs, and examine the outputs for correct operation.
  - Since your entity is a component instantiation in the context of the test bench, you now have all the power of the configuration mechanism at your disposal.
  - Why might you want to do this? Well, if you start with a behavioural description, refine it to a dataflow description, and then to a structural description, you might want to test that they are equivalent. In the test bench, you can instantiate any or all of the architectures, exercise them with the same inputs, and test that the outputs are equal.
- There are several subtle points about instantiating an entity that we're glossing over here. We'll put off discussing them for a bit until we know more about VHDL.

Packages

- It can happen that you'll want to split off bundles of subprograms, type declarations, *etc.*, because they're common utilities, because you want to hide internal details, or just because the design became too large and difficult to understand without hierarchy. The VHDL mechanism for this is a *package*.
- A package is divided into a *package-declaration* part and a *package-body* part. Things in the *package-declaration* are visible in the *package-body* and to the users of the package. Things in the *package-body* are hidden from the users, so that the implementation details are hidden away.

- Packages in VHDL perform the same functions as libraries and modules in any programming language. The provide a means to manage complexity and to specify a public interface while hiding implementation details.

Variables, Signals, and the Simulation Cycle

The basic concepts in this section are introduced in §1.4 of [1], with more detail on the scheduling of transactions in §5.3.

- The *architecture-body* of any VHDL entity consists, ultimately, of concurrently executing processes interconnected by signals.
  - This may be immediately obvious, as in the case where we're looking at a behavioural description and all the processes are visible at the surface. In the extreme case, there may be only one — a single process which describes the behaviour of the hardware we want to design.
  - Or it might be that we can't see all the concurrent processes, because they're buried deep under many layers of the component hierarchy. Only after the model is expanded, flattening the hierarchy down to its primitive entities (a task that the VHDL people call *elaboration*) do we see all the concurrent processes which define the behaviour.
  - Don't forget that a *concurrent-signal-assignment-statement* is just a convenient representation of a trivial (one statement) process. The `dataflow_1` architecture back on p. 3 has seven distinct processes (seven *concurrent-signal-assignment-statement*s).

  You can think of each process as a hardware component, which has a set of inputs, performs some transformation on those inputs, and asserts values at a set of outputs. A component might be simple (a 2-input AND gate) or more complex (a 16-bit register). If we open up a complex component and look inside (elaboration), we see more primitive components (16 FFs, in the case of the 16-bit register).

- Understanding how these concurrent processes interact in the simulation cycle is critical to understanding how one describes hardware in VHDL.

- VHDL provides two distinct types of objects that can be assigned a value, *variables* and *signals*. It's important to understand the distinction between them, and their relationship to the simulation cycle.

- To make a beginning, think of signals as modelling the interconnections between components, so that they're very much like wires. 'Assigning' a

value to a signal is the wrong mental picture. Think of an output asserting a signal into a wire.

Think of variables as convenient (but artificial) places for storing values in a VHDL model until you get around to refining the design and instantiate real components (FFs and registers) to hold the values.

This is close to accurate, and you'll refine this image as your understanding improves over the course of the semester.

- We'll dispense with variables for a moment by noting that in VHDL variables are *always* local variables, known only within the process where they are declared. We'll get back to them.

- A *signal-assignment-statement* takes the general form

  signal-assignment-statement ::= name <= *value-expression* `after` *time-expression*

  For example, the statement

  ```
  out <= input1 and input2 after 5 ns ;
  ```

  models an AND gate with a 5 ns. propagation delay.

  - This reflects the realities of hardware, where logic has propagation delay, and where even wires add some little delay as a signal propagates from one end to the other.
  - The delay incorporated in a *signal-assignment-statement* accounts for the combinational logic delay (to calculate *value-expression*), and any other propagation delay required for the new value to show up at the other end of the wire.

- If you've been looking through the VHDL book, or reading in LogicWorks, you may have noticed that the default delay for a signal assignment is 0 ns. Why would that be?

  - For one thing, what's a reasonable value? Anything you pick would be wrong 99.9% of the time.
  - For another thing, we'll see that because of the way a VHDL simulation works, we'll occasionally have a valid reason for signal assignments with zero delay.
  - **But** for our purposes in Cmpt 250, 99.99% of the time zero delay is the wrong choice. It makes it very difficult to interpret the results of the simulator, and it can mask timing problems. Timing diagrams collapse — all signal transitions become a single line at $t = 0$.

Typically, we'll assume a generic propagation delay of 1 unit for logic gates and 2 units for FFs. Since we're not concerned with the subtle timing problems that can occur with aggressive high-speed designs, pretty much any value will do for a unit of delay.

- In VHDL, each signal can have exactly one source unless you explicitly write VHDL to allow for more than one source. This is a feature, not a bug. It reflects the fact that, unless you make special arrangements in your hardware design, a wire can be driven by exactly one output. Any number of inputs can receive the signal, but only one output should be attempting to assert a value onto the wire.

  The single source for a signal will be one of the concurrent processes that comprise the VHDL description; this process is called the *driver* for the signal.

  - 'Special arrangements' are used to implement such things as busses; we'll come back to that later.

- So, what do we have: An *architecture-body* that consists of a number of concurrently executing processes, interconnected by signals.

  How do we simulate such a thing? We use *discrete event simulation*.

  - Conceptually, the simulator keeps a list of scheduled transactions (new values for signals) and the time when those transactions should occur. The list is sorted by increasing time into the future.
  - Roughly speaking, the simulator removes and executes the transaction on the front of the list, deals with any consequences which may occur, then advances time to the new front-of-list transaction. When there are multiple transactions scheduled for the same time, all are executed before time is advanced.
  - 'Executing the transaction', in VHDL, means assigning the specified value to the specified signal. If this causes the value of the signal to change, the transaction produces an *event*.
  - 'Consequences', in VHDL, consist of executing any processes which have been waiting for this signal to have an event. If those processes specify assignments to signals, the transactions are added to the transaction list, in their proper position according to the time when they should occur. This is termed *scheduling a transaction*.

- Before we go on, let's consider again the distinction between a transaction and an event.

- A transaction is the assignment of a value to a signal at a specified time.
- An event is a transaction which causes the value of the signal to change.
- Why would a transaction not cause an event? It's actually quite common. Suppose that a process is activated by a change in one or more of its input signals. It executes statements and calculates the proper value for an output signal, but it turns out that this value is the same as the value the signal has now. If the process is written like our VHDL fragments for the 2-to-4 decoder, the newly calculated value will still be assigned to the signal. This is a transaction, but when it's executed, it won't cause an event.

  A more simple example: A process that models an AND gate might execute when any input changes. But if one of the inputs is 0, the output of the gate will remain at 0. There will be a transaction to assign 0 to the output, but it will not result in an event.

- Let's go one level deeper, to see the exact sequence of events. We need to consider an initialisation phase, followed by repeated simulation cycles.
- In the *initialisation phase*, we proceed in three steps:
  - Each signal is assigned an initial value. This initial value can be specified in the signal declaration, or it can be left to a type-dependent default[2].
  - The simulation clock is set to 0, and all processes are scheduled to execute. Each process is executed until it *suspends*, *i.e.*, until it reaches a point where it must wait before it can continue execution. The wait which stops execution can be an explicit `wait` statement, or it can be the implicit wait that follows the end of the sequential statements of the process when there are no explicit `wait` statements.

    When a process executes a sequential *signal-assignment-statement*, a transaction is scheduled for that signal at some time in the future. One or more transactions may be scheduled before the process reaches a point where it needs to wait.
  - When all processes have suspended, the initialisation phase ends. Time is still at 0 on the simulation clock.
- There's an important point to emphasise here: With one exception, execution of the sequential statements in a process takes *no time* as far as the simulation clock is concerned.

[2]The default for a type is defined to be the leftmost, or first, element in the range for the type. For some types, this is not the intuitively obvious choice. It is bad programming practice to rely on the value of an uninitialised variable. If you care, specify a value.

- Sequential statements (except for `wait`) can execute and manipulate variables, and are considered to happen instantaneously.
- Signal assignment statements executed in this sequential context cause transactions to be scheduled to happen in the future, but of themselves take no time.
- `wait` statements are the only statements which can stop the execution of sequential statements in a process.
- Only when all processes have suspended will the simulator advance the simulation clock. The new time will be the scheduled time for the first signal transaction on the transaction list.

* So, initialisation is over, the simulator has executed all processes until they have all suspended, and we have a bunch of transactions scheduled. Now the simulator enters the main simulation loop. Each iteration of this loop is called a simulation cycle.

  - The simulator removes the first transaction from the list of scheduled transactions and sets the simulation clock to the time specified for the transaction. If the are more transactions which should occur at the same time, that are all removed from the list and executed.

    If executing a transaction causes an event (*i.e.*, the value of a signal changes), all processes which are sensitive to (*i.e.*, waiting on) the event are scheduled for execution.
  - The scheduled processes are all run until they suspend. Any signal assignments that are executed will cause a signal transaction to be scheduled. *Even if the signal assignment specifies a delay of 0, the transaction is placed on the transaction list and will not be executed until the next simulation cycle.*
  - When all processes have suspended, the current cycle is over, and we repeat, advancing time to the scheduled time of the first transaction on the list.

  To sum up: All transactions scheduled to occur at the same time are executed, and all signal values are changed. Then all processes that are sensitive to the change are executed, and new transactions are placed in the transaction queue. When all processes have suspended, the current cycle is over and a new cycle starts.

* When the transaction queue becomes empty, the simulation halts.

  - That's not as straightforward as it might seem. In many of our simulations, we will have a process which produces a clock signal. The clock

will run forever, hence the simulation will never stop on its own. You must arrange to stop it, by limiting the number of simulation cycles, by limiting the allowed simulation time, or by manually interrupting the simulation.

- The case where a signal transaction is scheduled to occur with a delay of 0 is called a *delta delay*. There is no change in the value of the simulation clock, but we do move to a new simulation cycle.
  - A delta delay is the default delay — if you don't specify a time delay for a signal assignment, it takes 0 fs. (femtoseconds, 10^{-15} secs.).
  - For high-level simulations, this is sometimes just what we want. Signals propagate in the proper order, and data dependencies are observed, but we can avoid being specific about just how much time has passed.
  - *Be careful* when using delta delays. Logical errors in the VHDL description (also known as 'bugs in your code') can be difficult to see because (as mentioned earlier) the timing diagram collapses.
- Beware of this trap when writing sequential statements:

```
sig <= '1' ;

  ...

if (sig = '1') then ...
```

  If you're expecting the `then` part of this to execute in the same cycle, because you've assigned `'1'` to `sig` with 0 delay, forget it. `sig` won't have the value `'1'` until the next simulation cycle.

- So, what about variables?

  Variables are used to maintain the state of a process and hold intermediate values during execution of the process' sequential statements. They are created when the process is created (*i.e.*, during the initialisation phase) and exist until the process is destroyed at the end of the simulation. They are strictly local to the process.

  Assignment to a variable takes zero time, as with all other sequential statements except `wait`.

  Unlike signal assignment, variable assignment takes effect immediately, so that

```
var := '1' ;
```

```
...

if (var = '1') then ...
```

will execute the code in the `then` clause of the `if` statement.

- Variables are completely hidden inside individual processes — they can be accessed only within the process. For our purposes, there's no such thing as global variables. Signals are the only way to communicate information between processes.

  This gets the designers of VHDL, and the implementors of VHDL simulators, off the hook for defining what happens when two concurrently executing processes try to change the same variable.

- Before we leave discrete simulation, there's one last thing to talk about: transport delay *vs.* inertial delay for signal assignment. If you've read about this in [1], you might be worried by the fact that inertial delay is the default, and it's considerably more complicated than transport delay.

  To quote the Hitchhiker's Guide to the Galaxy, **Don't Panic!**

  You don't have to worry about this until you begin to get really picky about modelling timing and start scheduling transactions at intervals which are small compared to the propagation delays specified in the *signal-assignment-statement*s. Given that we'll be doing our modelling at the register-transfer level, using a clocked synchronous design style, we won't have to worry about the details of just how the transaction queue is rearranged.

Resolved Types

- Getting back to tidy up a loose end, things like busses and other constructs which involve multiple outputs driving a single signal are handled using a *resolved type*.
  - With a resolved type, you define a *resolution function*. When a transaction is scheduled for a signal with multiple drivers, all the values currently being asserted are supplied in a vector to the resolution function.
  - It's the responsibility of the resolution function to work its way through the vector and come up with a single value for the signal.
  - This is where values like `'Z'`, the high-impedance state for a tri-state driver, or `'H'` (`'L'`), the weak logic high (low) created by a pull-up (pull-down) resistor, come into play.
- We need to work our way up to understanding a resolved type, so let's fill in some background.

- LogicWorks comes with many packages (libraries) of useful VHDL components, types, and functions. Amongst these is `std_logic_1164`, specified by IEEE Standard 1164. It declares the data types `std_ulogic` and `std_logic`, more sophisticated versions of the `bit` data type that's built into VHDL.
- The definition of `std_ulogic` is

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   ) ;
```

- One of the first things to point out is that this is an enumeration type (analogous to a C `enum`), and its values are character constants.
- Instead of just 0 and 1, `std_ulogic` contains 9 values. How do we sort these out?
  - Uninitialized (`'U'`) is sort of self-explanatory. It's the default value assigned when a simulation starts. If you ever see it, it's a good indication that your design does not properly initialise all signals and components.
  - Don't care (`'-'`) means exactly that — we don't care, and thus don't know. All things considered, you should probably worry if you ever see this value, because it's unlikely you have a legitimate use for it.
  - High impedance (`'Z'`) indicates the off (high impedance) state of a tristate output.
  - The values 0 (`'0'`) and 1 (`'1'`) are the logic values you're familiar with.
  - Unknown (`'X'`) indicates that we don't know the value. It can arise in the middle of a simulation as the result of a conflict between `'0'` and `'1'`. This happens if, for example, one output driving a bus is trying to assert a `'1'` and another output is trying to assert a `'0'`. If you see this value, it almost certainly indicates an error in your VHDL model — often, incorrect or incomplete initialisation.
  - There are strong and weak versions of 0, 1, and unknown (`'L'`, `'H'`, and `'W'`, respectively) to reflect the fact that at the analog level it's possible for different outputs to have different signal strengths.

- We'll be interested in `'U'`, `'X'`, `'0'`, `'1'`, and `'Z'`. We'll need these to properly model digital hardware which includes busses driven by tri-state outputs. We won't be working with components that produce weak signals, and you should never see them in your work for this course.

- Now we come to the transformation of `std_ulogic` to `std_logic`. The declaration for `std_logic` looks like a simple subtype declaration:

```
subtype std_logic is resolved std_ulogic ;
```

  But don't be fooled — `resolved` isn't just another VHDL reserved word, it's actually the name of the `std_logic` resolution function.

- For `std_logic`, we can summarise the action of the resolution function as follows:

  - Uninitialised dominates everything. After that, strong values dominate weak values, and any value dominates high impedance.
  - The effect of `'X'` (unknown) depends on the logic function. For example, `'X' and '0'` produces `'0'`, because the logical AND of 0 and anything is 0. `'X' and '1'` produces `'X'`, because the result depends on the value of the unknown input.
  - Within a class (strong or weak), unknown dominates known values, and conflicting values result in unknown.

  If you want to see more detail, have a look at [1, §§8.1 and 8.2].

Overloading

- There's one final concept to talk about: overloading.

- VHDL makes heavy use of a technique called *overloading*, in which the meaning of a particular word or symbol (a *lexeme*) in a VHDL program will depend on the context in which it is encountered.

  - To take a common example, consider the addition operator, '+'.
  - When you write 3 + 4, the '+' sign stands for integer addition.
  - When you write 3.14 + 1.27, the '+' sign stands for floating point addition. (You may not have realised there's a difference, but after we've talked about how a computer does floating point arithmetic, you'll realise what a great difference there is.)

- VHDL determines just what operation to perform by looking at the operator *and* at the types of the operands.

- Overloading is applied to many things in VHDL, and we'll take them on as we encounter them, but it's important that you know it exists. You may on occasion need to add explicit type conversions when the VHDL analyser (compiler) complains that it can't determine the proper operation because it can't decide the types of the operands.

That's it for the introduction to VHDL. Your next step should be to work through the tutorials in the LogicWorks book to see how VHDL is handled in LogicWorks.

References

[1] P. Ashenden. *The Students Guide to VHDL*. Morgan Kaufmann Publishers, San Francisco, California, 1998.

[2] Ltd. Capilano Computing Systems. *LogicWorks 5: Interactive Circuit Design Software*. Pearson Prentice Hall, 2004.

[3] M. Mano and C. Kime. *Logic and Computer Design Fundamentals*. Prentice-Hall, Upper Saddle River, New Jersey, 2nd updated edition, 2001.