A Hardware Engineer's Guide to VHDL

This page and its accompanying links address the needs of those users who are new to VHDL. This VHDL tutorial assumes no prior knowledge of HDLs. The pre-requisites are hardware design experience, more than a passing acquaintance with programming languages and a willingness to learn.

VHDL is a programming language. However, throughout this tutorial we aim to map code snippets directly onto the equivalent hardware. In addition, we will encourage you to "think code" so that at the end of the tutorial you are as comfortable with VHDL code fragments as you are with schematics.

Note that we are building this tutorial incrementally with monthly releases of new topics which can be accessed from this page. This corner of the Web is split into two sections, the first providing you with a VHDL overview, the second gives you some real examples of VHDL code use. In the VHDL Backgrounder, we aim to provide you with a general appreciation of what VHDL is, and how it is used in the design process. In the Designing Hardware using VHDL section, we're going to familiarise you with the main features of VHDL, particularly design entities. This will let you see how to design simple circuit elements in VHDL.

If you want to go beyond the material we present here, call Doulos for a **free** copy of VHDL PaceMaker Entry Edition, surf the rest of our Web site and book yourself onto a Doulos training course. It's the best way to learn and it's also the most enjoyable way to learn VHDL - learn VHDL from the VHDL experts!

VHDL Backgrounder

- What is VHDL?
- A Brief History of VHDL
- Levels of Abstraction
- Scope of VHDL
- Design Flow using VHDL
- Benefits of using VHDL

Designing Hardware using VHDL

- An Example Design Entity
- Internal Signals
- Components and Port Maps
- Plugging Chips into Sockets
- Configurations: Part 1
- Configurations: Part 2
- Order of Analysis
- Vectored Ports and Signals
- Test Benches: Part One
- Test Benches: Part Two
- Summary, so far...
- Comparing Components with Processes
- Processes

- RTL Coding
- If statement
- Synthesizing Latches 

 VHDL FAQ

 Doulos Training Courses

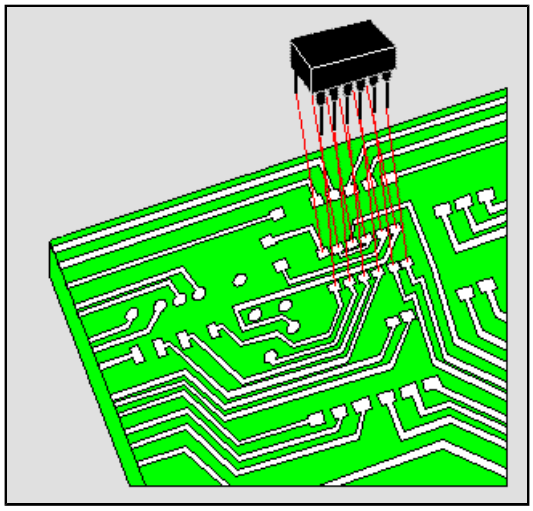
---

 Doulos Home Page

This page was last updated 2nd June 1999

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

What is VHDL?



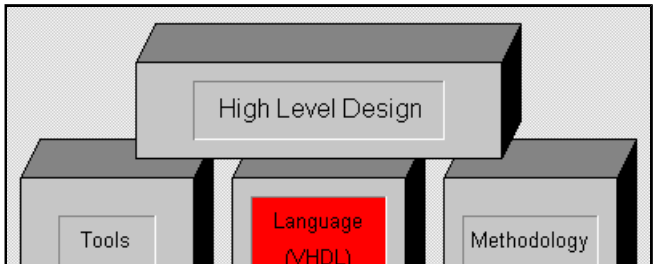
VHDL is the VHSIC Hardware Description Language. VHSIC is an abbreviation for Very High Speed Integrated Circuit. It can describe the behaviour and structure of electronic systems, but is particularly suited as a language to describe the structure and behaviour of digital electronic hardware designs, such as ASICs and FPGAs as well as conventional digital circuits.

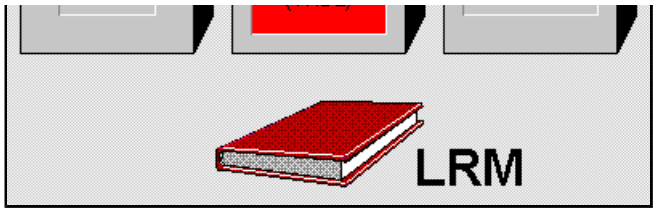
VHDL is a notation, and is precisely and *completely* defined by the Language Reference Manual (LRM). This sets VHDL apart from other hardware description languages, which are to some extent defined in an ad hoc way by the behaviour of tools that use them. VHDL is an international standard, regulated by the IEEE. The definition of the language is non-proprietary.

VHDL is not an information model, a database schema, a simulator, a toolset or a methodology! However, a methodology and a toolset are essential for the effective use of VHDL.

Simulation and synthesis are the two main kinds of tools which operate on the VHDL language. The Language Reference Manual does not define a simulator, but unambiguously defines what each simulator must do with each part of the language.

VHDL does not constrain the user to one style of description. VHDL allows designs to be described using any methodology — top down, bottom up or middle out! VHDL can be used to describe hardware at the gate level or in a more abstract way. Successful high level design requires a language, a tool set and a suitable methodology. VHDL is the language, *you* choose the tools, and the methodology... well, I guess that's where Doulos come in to the equation!





Doulos Training Courses
VHDL FAQ
Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 15th January 1999

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

A Brief History of VHDL

The Requirement

The development of VHDL was initiated in 1981 by the United States Department of Defence to address the hardware life cycle crisis. The cost of reprocuring electronic hardware as technologies became obsolete was reaching crisis point, because the function of the parts was not adequately documented, and the various components making up a system were individually verified using a wide range of different and incompatible simulation languages and tools. The requirement was for a language with a wide range of descriptive capability that would *work the same* on any simulator and was independent of technology or design methodology.

Standardization

The standardization process for VHDL was unique in that the participation and feedback from industry was sought at an early stage. A baseline language (version 7.2) was published 2 years before the standard so that tool development could begin in earnest in advance of the standard. All rights to the language definition were *given away* by the DoD to the IEEE in order to encourage industry acceptance and investment.

ASIC Mandate

DoD Mil Std 454 mandates the supply of a comprehensive VHDL description with every ASIC delivered to the DoD. The best way to provide the required level of description is to use VHDL throughout the design process.

VHDL '93

As an IEEE standard, VHDL must undergo a review process every 5 years (or sooner) to ensure its ongoing relevance to the industry. The first such revision was completed in September 1993, and tools conforming to VHDL '93 are now available. VHDL’98? Hmmm...

Summary: History of VHDL

1981 - Initiated by US DoD to address hardware life-cycle crisis

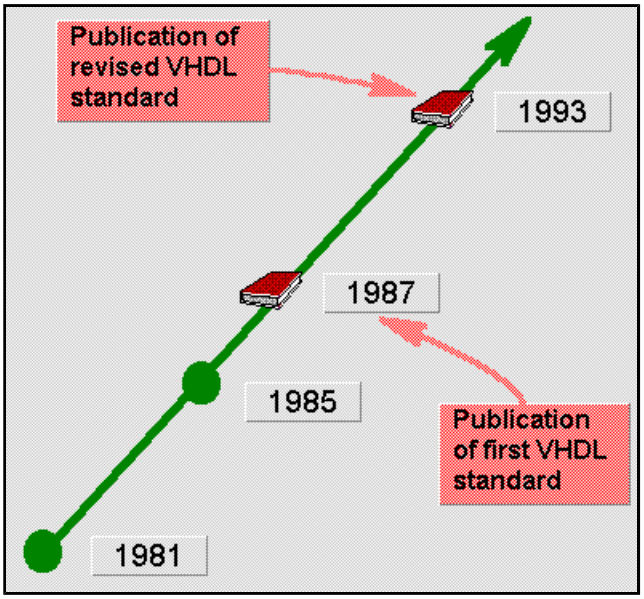
1983-85 - Development of baseline language by Intermetrics, IBM and TI

1986 - All rights transferred to IEEE

1987 - Publication of IEEE Standard

1987 - Mil Std 454 requires comprehensive VHDL descriptions to be delivered with ASICs

1994 - Revised standard (named VHDL 1076-1993)



Doulos Training Courses

Return to Hardware Engineer's Guide Contents

Doulos Home Page

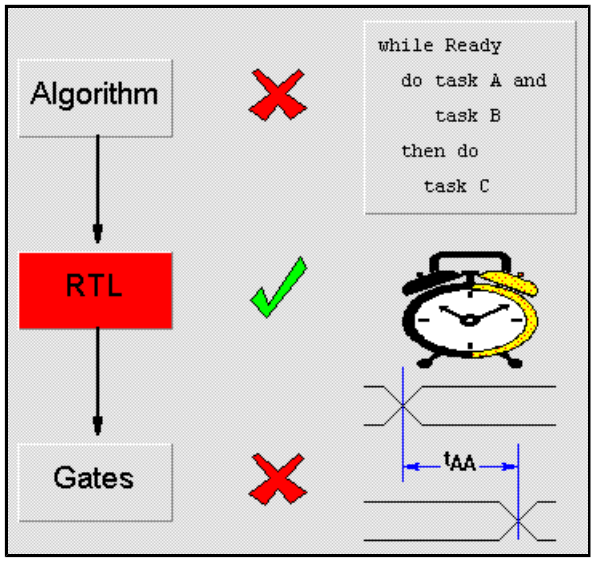
This page was last updated 15th January 1999

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Levels of Abstraction

VHDL can be used to describe electronic hardware at many different levels of abstraction. When considering the application of VHDL to FPGA/ASIC design, it is helpful to identify and understand the three levels of abstraction shown opposite - algorithm, register transfer level (RTL), and gate level. Algorithms are unsynthesizable, RTL is the input to synthesis, gate level is the output from synthesis. The difference between these levels of abstraction can be understood in terms of timing.

Levels of abstraction in the context of their time domain



Algorithm
A pure algorithm consists of a set of instructions that are executed in sequence to perform some task. A pure algorithm has neither a *clock* nor detailed delays. Some aspects of timing can be inferred from the partial ordering of operations within the algorithm. Some synthesis tools (*behavioural synthesis*) are available that can take algorithmic VHDL code as input. However, even in the case of such tools, the VHDL input may have to be constrained in some artificial way, perhaps through the presence of an ‘algorithm’ clock — operations in the VHDL code can then be synchronized to this clock.

RTL
An RTL description has an explicit clock. All operations are scheduled to occur in specific clock cycles, but there are no detailed delays below the cycle level. Commercially available synthesis tools do allow some freedom in this respect. A single global clock is not required but may be preferred. In addition, retiming is a feature that allows operations to be re-scheduled across clock cycles, though not to the degree permitted in behavioural synthesis tools.

Gates
A gate level description consists of a network of gates and registers instanced from a technology library, which contains technology-specific delay information for each gate.

Writing VHDL for Synthesis

In the diagram above, the RTL level of abstraction is highlighted. This is the ideal level of abstraction at which to design hardware given the state of the art of today's synthesis tools. The gate level is too low a level for describing hardware - remember we're trying to move away from the implementation concerns of hardware design, we want to abstract to the specification level - *what* the hardware does, not *how* it does it. Conversely, the algorithmic level is too high a level, most commercially available synthesis tools cannot produce hardware from a description at this level.

In the future, as synthesis technology progresses, we will one day view the RTL level of abstraction as the "dirty" way of writing VHDL for hardware and writing algorithmic (often called behavioural) VHDL will be the norm.

Until then, VHDL coding at RTL for input to a synthesis tool will give the best results. Getting the best results from your synthesizable RTL VHDL is a key topic of the Doulos Comprehensive VHDL and Advanced VHDL Techniques training courses. The latter also covers behavioural synthesis techniques.

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

 Doulos Home Page

This page was last updated 15th January 1999

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Scope of VHDL

VHDL is suited to the specification, design and description of digital electronic hardware.

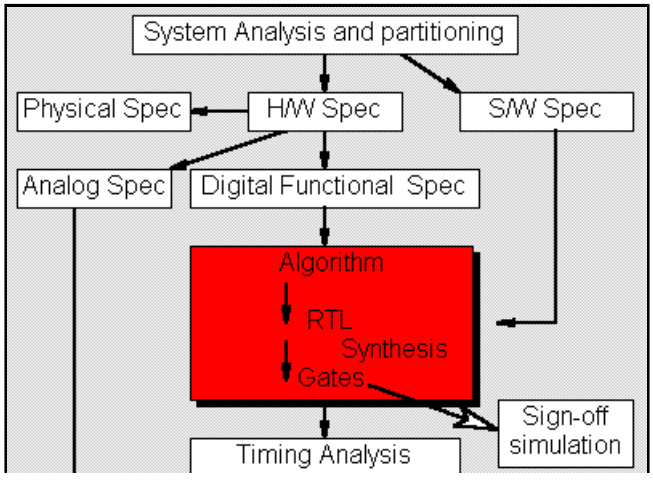
System level
VHDL is not ideally suited for abstract system-level simulation, prior to the hardware-software split. Simulation at this level is usually stochastic, and is concerned with modelling performance, throughput, queueing and statistical distributions. VHDL has been used in this area with some success, but is best suited to *functional* and not *stochastic* simulation.

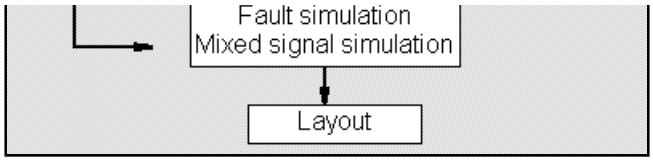
Digital
VHDL is suitable for use today in the digital hardware design process, from specification through high-level functional simulation, manual design and logic synthesis down to gate-level simulation. VHDL tools usually provide an integrated design environment in this area.

VHDL is not suited for specialized implementation-level design verification tools such as analog simulation, switch level simulation and worst case timing simulation. VHDL can be used to simulate gate level fanout loading effects providing coding styles are adhered to and delay calculation tools are available. The standardization effort named VITAL (VHDL Initiative Toward ASIC Libraries) is active in this area, and is now bearing fruit in that simulation vendors have built-in VITAL support. More importantly, many ASIC vendors have VITAL-compliant libraries, though not all are allowing VITAL-based sign-off — not yet anyway.

Analogue
Because of VHDL's flexibility as a programming language, it has been stretched to handle analog and switch level simulation in limited cases. However, look out for future standards in the area of analog VHDL. Check out our Model of the Month from April 1996 for an example of analogue modeling in VHDL.

Design process
The diagram below shows a very simplified view of the electronic system design process incorporating VHDL. The central portion of the diagram shows the parts of the design process which are most impacted by VHDL.





Doulos Training Courses
VHDL FAQ
Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 4th January 1999

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Design Flow using VHDL

The diagram below summarizes the high level design flow for an ASIC (ie. gate array, standard cell) or FPGA. In a practical design situation, each step described in the following sections may be split into several smaller steps, and parts of the design flow will be iterated as errors are uncovered.

System-level Verification
As a first step, VHDL may be used to model and simulate aspects of the complete system containing one or more devices. This may be a fully functional description of the system allowing the FPGA/ASIC specification to be validated prior to commencing detailed design. Alternatively, this may be a partial description that abstracts certain properties of the system, such as a performance model to detect system performance bottle-necks.

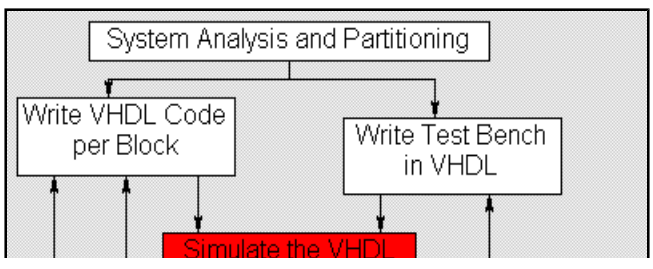
RTL design and testbench creation
Once the overall system architecture and partitioning is stable, the detailed design of each FPGA/ASIC can commence. This starts by capturing the design in VHDL at the register transfer level, and capturing a set of test cases in VHDL. These two tasks are complementary, and are sometimes performed by different design teams in isolation to ensure that the specification is correctly interpreted. The RTL VHDL should be synthesizable if automatic logic synthesis is to be used. Test case generation is a major task that requires a disciplined approach and much engineering ingenuity: the quality of the final FPGA/ASIC depends on the coverage of these test cases.

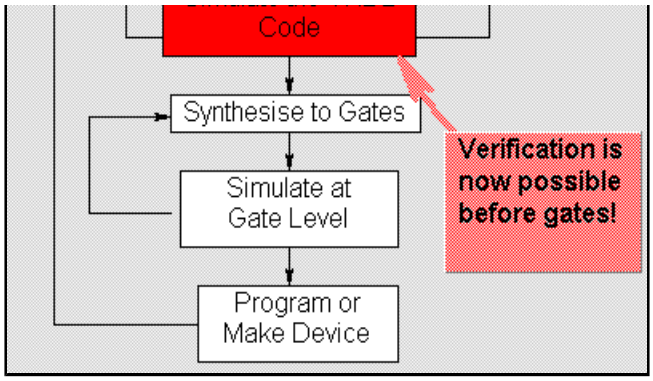
RTL verification
The RTL VHDL is then simulated to validate the functionality against the specification. RTL simulation is usually one or two orders of magnitude faster than gate level simulation, and experience has shown that this speed-up is best exploited by doing more simulation, not spending less time on simulation.

In practice it is common to spend 70-80% of the design cycle writing and simulating VHDL at and above the register transfer level, and 20-30% of the time synthesizing and verifying the gates.

Look-ahead Synthesis
Although some exploratory synthesis will be done early on in the design process, to provide accurate speed and area data to aid in the evaluation of architectural decisions and to check the engineer's understanding of how the VHDL will be synthesized, the main synthesis production run is deferred until functional simulation is complete. It is pointless to invest a lot of time and effort in synthesis until the functionality of the design is validated.

70% of design time at RTL!





VHDL FAQ

Doulos Training Courses

Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 15th January 1999

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Benefits of using VHDL

Executable specification

It is often reported that a large number of ASIC designs meet their specifications first time, but fail to work when plugged into a system. VHDL allows this issue to be addressed in two ways: A VHDL specification can be executed in order to achieve a high level of confidence in its correctness before commencing design, and may simulate one to two orders of magnitude faster than a gate level description. A VHDL specification for a part can form the basis for a simulation model to verify the operation of the part in the wider system context (eg. printed circuit board simulation). This depends on how accurately the specification handles aspects such as timing and initialization.

Behavioural simulation can reduce design time by allowing design problems to be detected early on, avoiding the need to rework designs at gate level. Behavioural simulation also permits design optimization by exploring alternative architectures, resulting in better designs.

Tools

VHDL descriptions of hardware design and test benches are portable between design tools, and portable between design centres and project partners. You can safely invest in VHDL modelling effort and training, knowing that you will not be tied in to a single tool vendor, but will be free to preserve your investment across tools and platforms. Also, the design automation tool vendors are themselves making a large investment in VHDL, ensuring a continuing supply of state-of-the-art VHDL tools.

Technology

VHDL permits technology independent design through support for top down design and logic synthesis. To move a design to a new technology you need not start from scratch or reverse-engineer a specification - instead you go back up the design tree to a behavioural VHDL description, then implement that in the new technology knowing that the correct functionality will be preserved.

Benefits

- Executable specification
- Validate spec in system context
    - Subcontract
- Functionality separated from implementation
- Simulate early and fast
    - Manage complexity
- Explore design alternatives
- Get feedback
    - Produce better designs
- Automatic synthesis and test generation (ATPG for ASICs)
- Increase productivity
    - Shorten time-to-market
- Technology and tool independence (though FPGA features may be unexploited)

- Portable design data
    - Protect investment

 VHDL FAQ

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

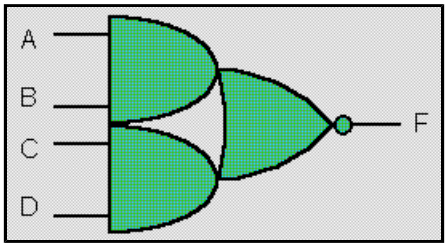
 Doulos Home Page

This page was last updated 10th January 1999

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

An Example Design Entity

A design is described in VHDL using the concept of a design entity. A design entity is split into two parts, each of which is called a design unit in VHDL jargon. The *entity declaration* represents the external interface to the design entity. The *architecture body* represents the internal description of the design entity - its behaviour, its structure, or a mixture of both. Let's imagine we want to describe an and-or-invert (AOI) gate in VHDL. If we consider the AOI gate as a single chip package, it will have four input pins and one output pin; we need not concern ourselves with power and ground pins in modelling our AOI design.



VHDL: an AOI gate design entity

```
-- VHDL code for AND-OR-INVERT 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
```

OK, that's the code. Let's dissect it line by line...

`-- VHDL code for AND-OR-INVERT gate`
Similar to many programming languages, VHDL supports comments. Comments are not part of the VHDL design, but allow the user to make notes referring to the VHDL code, usually as an aid to understanding it. Here the comment is a "header" that tells us that the VHDL describes an AOI gate. It is no more than an *aide de memoire* in this case. A VHDL compiler will ignore this line of VHDL. Two hyphens mark the start of a *comment*, which is ignored by the VHDL compiler. A comment can be on a separate line or at the end of a line of VHDL code, but in any case stops at the end of the line.

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

Above the entity declaration is a library clause (`library IEEE;`) and a use clause (`use IEEE.STD_LOGIC_1164.all;`). This gives the entity `AOI` access to all the names declared within package `STD_LOGIC_1164` in the library `IEEE`, and to data type `STD_LOGIC` in particular. More on data types, later.

`entity AOI is`
The name of the design entity is just an arbitrary label invented by the user. It does not correspond to a name pre-defined in a VHDL component library. **`entity`** and **`is`** are VHDL keywords. This line defines the start of a new VHDL design unit definition. The library and use clauses, although written before the entity declaration do not define the start the VHDL description of a design unit, they are *context* clauses. We can think of an entity declaration as corresponding to a chip package.

```
port (
A, B, C, D: in STD_LOGIC;
F : out STD_LOGIC
);
```
The entity declaration includes the name of the entity (`AOI` in this example), and a set of port declarations. A *port* may correspond to a pin on an IC, an edge connector on a board, or any logical channel of communication with a block of hardware. Each port declaration includes the name of one or more ports (e.g., `A`, `B`), the direction that information is allowed to flow through the ports (`in`, `out` or `inout`), and the data type of the ports (i.e., `STD_LOGIC`). In our example the port declarations correspond to the pins of our AOI chip.

The data type of a port defines the set of values that may flow through the port. The ports are of type `STD_LOGIC`, which is found in *package* `STD_LOGIC_1164` on *library* `IEEE`. A *package* is a VHDL language construct where new data types may be defined, and the particular package `STD_LOGIC_1164` is an IEEE standard for representing digital signals in VHDL. The concept of data type is borrowed by VHDL from the world of software. It allows the VHDL compiler to ensure that the design is at least reasonably robust before beginning simulation.

`end AOI;`
The entity declaration is terminated by the VHDL keyword end. Here we indulge in a little programming robustness by adding the name of the design entity after the end keyword. Including the name of the design entity is particularly relevant in large descriptions where the port list may extend over many screens (or pages); it is good to be reminded of the name of the design entity whose end we are looking at, lest we forget.

`architecture V1 of AOI is`
The name of the architecture body (`V1`) is just an arbitrary label invented by the user. It is possible to define several alternative architecture bodies for a single design entity, and the only purpose of the architecture name is to distinguish between these alternatives. **`architecture`**, **`of`** and **`is`** are VHDL keywords. Note that when we define an architecture, we have to tell the VHDL analyzer that the architecture `V1` corresponds to the `AOI` design entity. You might think that it would be enough to specify the name of the architecture and that the architecture automatically corresponded to the previously declared entity, but I'm afraid VHDL doesn't work this way! In essence, we can think of the architecture as the die inside the chip package.

`begin`
The VHDL keyword begin denotes the end of the architecture declarative region and the start of the architecture statement part. In this architecture, there is but one statement, and all the names

referenced in this statement are in fact the ports of the design. Because all of the names used in the architecture statement part are declared in the entity declaration, the architecture declarative part is empty.

`F <= not ((A and B) or (C and D));`
The architecture contains a concurrent signal assignment, which describes the function of the design entity. The concurrent assignment executes whenever one of the four ports A, B, C or D change value. That's it! That's all there is to describing the functionality of an AOI gate in VHDL. Some might regard the rest of the VHDL code as superfluous and level a charge of verbosity against VHDL. Of course, the remainder of the VHDL code is setting the context in which this functionality is defined.

`end V1;`
The architecture is terminated by the VHDL keyword end. Once again, we reference the architecture name at the end of the architecture body for the same reason as we did with the entity. Usually, architecture bodies require sigificantly more code than entity declarations, hence repeating the name of the architecture is even more relevant.

`-- end of VHDL code`
Another VHDL comment, and that's the end of a VHDL description of an AOI gate.

 Doulos Training Courses
 Return to Hardware Engineer's Guide Contents

---

 Doulos Home Page

This page was last updated 18th June 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Internal signals

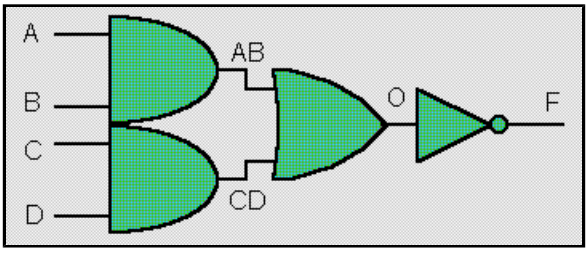
Shown below is a second architecture V2 of AOI (remember that the architecture name V2 is completely arbitrary - this architecture is called V2 to distinguish it from the earlier architecture V1). Architecture V2 describes the AOI function by breaking it down into the constituent boolean operations. Each operation is described within a separate concurrent signal assignment. In hardware terms we can think of each assignment as a die in a hybrid package or a multi-chip module. The signals are the bonding wires or substrate traces between each die.

Signals
The architecture contains three signals AB, CD and O, used internally within the architecture. A signal is declared before the *begin* of an architecture, and has its own data type (eg. STD_LOGIC). Technically, *ports* are *signals*, so signals and ports are read and assigned in the same way.

Assignments
The assignments within the architecture are *concurrent signal assignments*. Such assignments execute whenever a signal on the right hand side of the assignment changes value. Because of this, the order in which concurrent assignments are written has no effect on their execution.The assignments are *concurrent* because potentially two assignments could execute at the same time (if two inputs changed simultaneously). The style of description that uses only concurrent assignments is sometimes termed *dataflow*.

Delays
Each of the concurrent signal assignments has a delay. The expression on the right hand side is evaluated whenever a signal on the right hand side changes value, and the signal on the left hand side of the assignment is updated with the new value after the given delay. In this case, a change on the port A would propagate through the AOI entity to the port F with a total delay of 5 NS.

VHDL: Internal signals of an AOI 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 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;
```

Doulos Training Courses

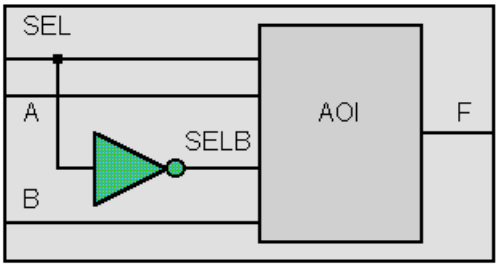
Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 15th June 1998

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Components and Port Maps



The example above shows the previously defined design entity AOI being used as a component within another, higher level design entity MUX2, to create a design hierarchy with two levels. The design entity MUX2 also contains a second component, named INV. In order to write the VHDL for this circuit, we need to cover two new concepts: component instantiation (placing the INV and AOI inside another higher-level design, MUX2) and port mapping (connecting up the two components to each other and to the primary ports of MUX2). In VHDL, this is how we can model PCBs assembled from individual chips, for example.

Components

The two component declarations (for INV and AOI) must match the corresponding entity declarations exactly with respect to the names, order and types of the ports. We've already seen enough of the AOI gate, let's see how closely the component and entity declarations match for the INV design entity.

Entity	Component

`entity INV is`
`  port (A: in STD_LOGIC;`
`  F: out STD_LOGIC);`
`end INV;`
	`component INV`
	`  port (A: in STD_LOGIC;`
	`  F: out STD_LOGIC);`
	`end component;`

The two component declarations (for INV and AOI) appear in the architecture declarative part (that's a VHDL technical term that means that the component declarations are coded before the begin).

```
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;
```

Instantiation

The architecture STRUCTURE of MUX2 makes *instances* of INV and AOI through the *component instantiations* at the bottom of the architecture (labelled G1 and G2). The component names (INV and AOI) are references to design entities defined elsewhere. The instance labels (G1 and G2) identify two specific instances of the components, and are mandatory.

Port Maps

The ports in a component declaration must usually match the ports in the entity declaration one-for-one. The component declaration defines the names, order, mode and types of the ports to be used when the component is *instanced* in the architecture body. Instancing a component implies making a local copy of the corresponding design entity - a component is declared once within any architecture, but may be instanced any number of times. In this example, there is just one instance of the components AOI and INV.

Association

Signals in an architecture are *associated* with ports on a component using a port map. In effect, a port map makes an electrical connection between “pieces of wire” in an architecture (signals) and pins on a component (ports). The same signal may be associated with several ports - this is the way to define interconnections between components.

In our MUX2 example, the signal SELB is associated with the F port of the INV instance and the C port of the AOI instance.

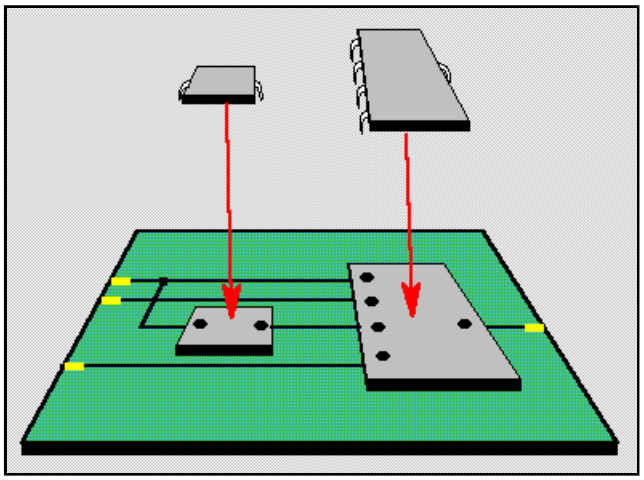
Doulos Training Courses
Return to Hardware Engineer’s Guide Contents

Doulos Home Page

This page was last updated 3rd July 1998

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Plugging Chips into Sockets



On the page “Components and Port Maps”, we looked at the VHDL used to describe the instantiation of components in an architecture. Instantiating components in VHDL enables us to create a design hierarchy, it’s just like plugging chips into a PCB. In our case, the MUX2 is the PCB, the AOI gate and the inverter (INV) are two chips. But what about the sockets? Let’s look into instantiating components in a little more detail.

Component Declarations

Remember that the two component declarations (for INV and AOI) must match the corresponding entity declarations exactly with respect to the names, order and types of the ports. This is important because, as a hardware analogy, the component declaration is essentially a chip socket for the chip represented by the design entity — the entity declaration defines the pins of the chip.

Component Instantiation

So, instantiation is essentially a VHDL term for soldering a chip socket into a PCB. Instantiation also assumes that the socket contains the chip referenced by the same name as it is plugged into the PCB. However, for our purposes, we can think of component instantiation as plugging a chip into a PCB, ignoring whether it’s socketed or not.

Component Declaration	Component Instantiation

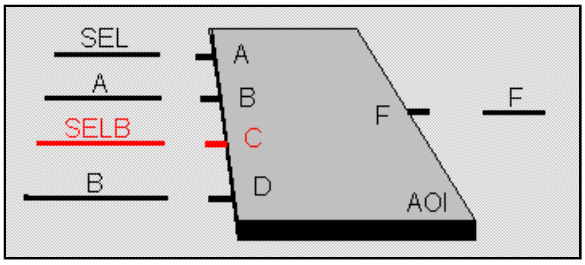
`component INV`
`  port (A: in STD_LOGIC;`
`  F: out STD_LOGIC);`
`end component;`
	`G1: INV`
	`  port map (SEL,`
	`  SELB`
	`);`

Port Mapping

The concept of port mapping is little more than soldering the chip socket pins into the holes of the PCB. Thus in the case of our inverter, the F output pin of the INV chip is plugged into the F pin of the

INV socket. The F pin of the INV socket is soldered to a PCB trace called SELB.

Similarly, instantiation of the AOI gate can be thought of as soldering the AOI socket into the MUX2 PCB. In this case, as an example, the SELB trace is soldered to the C pin of the AOI socket. By default, the AOI socket is assumed to be carrying an AOI chip.




```
architecture STRUCTURE of MUX2 is

...
component AOI
  port (A, B, C, D: in STD_LOGIC;         -- 3rd item in list is C
  F : out STD_LOGIC);
end component;

signal SELB: STD_LOGIC;

begin
  ...
  G2: AOI port map (SEL, A, SELB, B, F);  -- 3rd item in list is SELB
end;
```

Default binding

Our current view of component instantiation assumes *default binding*. In default binding, the chip socket (component declaration) carries a chip (design entity) of the same name (say, AOI) as we've already seen. Now in the hardware world, there's no such limitation, sockets aren't chip specific. VHDL allows the designer the same freedom. The chip socket and the chip do not have to have the same name, but to implement this facility requires a *configuration* to bind the appropriate design entity to the component instantiation. We'll look at configurations shortly.

Doulos Training Courses

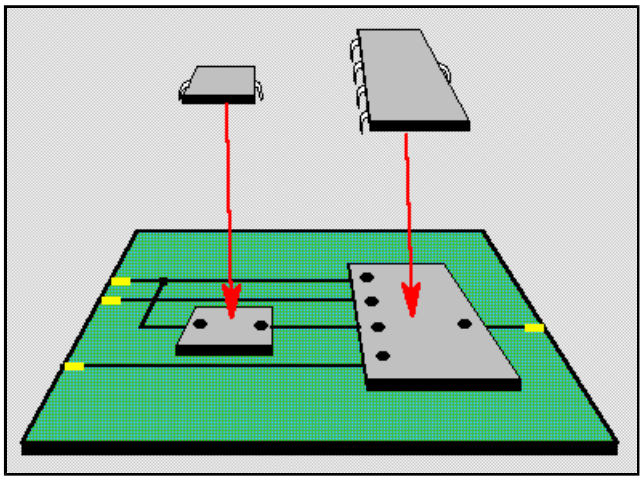
Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 4th August 1998

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Plugging Chips into Sockets



On the page “Components and Port Maps”, we looked at the VHDL used to describe the instantiation of components in an architecture. Instantiating components in VHDL enables us to create a design hierarchy, it’s just like plugging chips into a PCB. In our case, the MUX2 is the PCB, the AOI gate and the inverter (INV) are two chips. But what about the sockets? Let’s look into instantiating components in a little more detail.

Component Declarations

Remember that the two component declarations (for INV and AOI) must match the corresponding entity declarations exactly with respect to the names, order and types of the ports. This is important because, as a hardware analogy, the component declaration is essentially a chip socket for the chip represented by the design entity — the entity declaration defines the pins of the chip.

Component Instantiation

So, instantiation is essentially a VHDL term for soldering a chip socket into a PCB. Instantiation also assumes that the socket contains the chip referenced by the same name as it is plugged into the PCB. However, for our purposes, we can think of component instantiation as plugging a chip into a PCB, ignoring whether it’s socketed or not.

Component Declaration	Component Instantiation

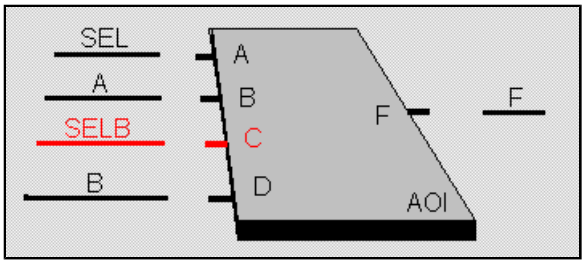
`component INV`
`  port (A: in STD_LOGIC;`
`  F: out STD_LOGIC);`
`end component;`
	`G1: INV`
	`  port map (SEL,`
	`  SELB`
	`);`

Port Mapping

The concept of port mapping is little more than soldering the chip socket pins into the holes of the PCB. Thus in the case of our inverter, the F output pin of the INV chip is plugged into the F pin of the

INV socket. The F pin of the INV socket is soldered to a PCB trace called SELB.

Similarly, instantiation of the AOI gate can be thought of as soldering the AOI socket into the MUX2 PCB. In this case, as an example, the SELB trace is soldered to the C pin of the AOI socket. By default, the AOI socket is assumed to be carrying an AOI chip.




```
architecture STRUCTURE of MUX2 is

...
component AOI
  port (A, B, C, D: in STD_LOGIC;         -- 3rd item in list is C
  F : out STD_LOGIC);
end component;

signal SELB: STD_LOGIC;

begin
  ...
  G2: AOI port map (SEL, A, SELB, B, F);  -- 3rd item in list is SELB
end;
```

Default binding

Our current view of component instantiation assumes *default binding*. In default binding, the chip socket (component declaration) carries a chip (design entity) of the same name (say, AOI) as we've already seen. Now in the hardware world, there's no such limitation, sockets aren't chip specific. VHDL allows the designer the same freedom. The chip socket and the chip do not have to have the same name, but to implement this facility requires a *configuration* to bind the appropriate design entity to the component instantiation. We'll look at configurations shortly.

Doulos Training Courses

Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 4th August 1998

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Configurations: Part 1

On the page “Plugging Chips into Sockets” it was suggested (in the *default binding* section) that an instantiation of a design entity need not have a component declaration of the same name to be legal VHDL, providing we use a configuration to change the default binding. In the case of Plugging Chips into Sockets, the emphasis was on establishing the relationships between component declaration, design entity (particularly its entity declaration) and component instantiation.

Remember that the design entity (chip package) consists of both an entity declaration (chip pins) and architecture body (chip die). In Part 1 of Configurations, we will look at selecting one architecture from many architectures of one design entity for instantiation (essentially specifying which die goes in the package of the chip that will be plugged into the PCB to maintain our analogy) and in Part 2, we will look at choosing from amongst different design entities for instantiation (essentially specifying which chip to plug into the socket).

Configuration

A VHDL description may consist of many design entites, each with several architectures, and organized into a design hierarchy. The *configuration* does the job of specifying the exact set of entities and architectures used in a particular simulation or synthesis run.

A configuration does two things. Firstly, a configuration specifies the design entity used in place of each component instance (i.e., it plugs the chip into the chip socket and then the socket-chip assembly into the PCB). Secondly, a configuration specifies the architecture to be used for each design entity (i.e., which die).

Default configuration

Shown below is a minimal configuration for the top level entity MUX2. This configuration selects the architecture to be used (there is only one, STRUCTURE). By default, all components inside the architecture will be configured to use a design entity of the same name as the component (i.e., AOI), and the most recently analyzed architecture of each design entity.

We’ll tackle the concept of the most recently analyzed architecture in the next but one article in this series. Suffice to say that in this example we will take v2 to be the most recently analyzed, by virtue of the fact that we compiled the v1 version of AOI first and then compiled the v2 version of our AOI design entity a few minutes later (let’s say).

Configuration declaration

The configuration declaration is yet another one of those VHDL design units. Remember the entity declaration is a design unit as is the architecture body. In the configuration declaration, for the architecture that we want to configure, we can specify exactly which components make up the final design entity. In the example below, for the G2 instance inside our STRUCTURE architecture, we use the V1 architecture of the AOI gate. There will be absolutely no doubt as to which architecture has been chosen for simulation because we have specified v1 in the configuration. In the `MUX2_default_CFG` configuration, we could change the AOI architecture for simulation by simply re-

compiling v1 after v2 — hardly a robust mechanism for design description! Note, that you only need to be concerned with configurations of multiple architectures when you have more than one architecture per design entity in the first place.

Default configuration of MUX2

```
use WORK.all;

configuration MUX2_default_CFG of MUX2 is
  for STRUCTURE
    -- Components inside STRUCTURE configured by default
    -- let's say v2 architecture for AOI
  end for;
end MUX2_default_CFG;
```

Specified configuration of MUX2

```
use WORK.all;

configuration MUX2_specified_CFG of MUX2 is
  for STRUCTURE
    for G2 : AOI
      use entity work.AOI(v1);
      -- architecture v1 specified for AOI design entity
    end for;
  end for;
end MUX2_specified_CFG;
```

Once again, let's break the VHDL code down fragment by fragment, using the specified configuration. Leaving aside the use clause for now, the first line specifies what we are configuring, **configuration** is a VHDL keyword and is followed by the name we wish to give the configuration. After the **of** keyword we specify the design entity we are configuring; **is** is also a keyword. Bracketed with this first line is the last **end** *configuration*_name line. Hence,

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

Configurations usually consist of nested configuration items, bracketed with **end for;** statements. The first item is the architecture specification,

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

the second is the instance specification,

```
  for instance_label : component_name
  -- thus,  for G2 : AOI
```

and finally the binding indication,

```
  use entity library_name.entity_name(architecture_name)
  -- thus,  use entity work.AOI(v1);
```

Now, what about that use clause? Let's say the AOI design entity is compiled into the working library

WORK by default. Thus in order for the *component*_name to be visible we need a use clause, hence `use WORK.all;` is required.

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

 Doulos Home Page

This page was last updated 14th August 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Configurations: Part 2

Let's review default binding before we go any further. In default binding, the chip socket (component declaration) carries a chip (design entity) of the same name (say, AOI) as we've already seen. The chip is inserted into the socket courtesy of a component instantiation and a configuration declaration. If we omit the configuration or if we use a default configuration, the socket and chip must have the same name.

If we want to choose a particular die (architecture) for our chip, we must specify the architecture in the configuration.

Now. Suppose we want to create a general-purpose socket and at some later time, we want to specify which chip will be plugged into the socket. To do this requires a *late-binding* configuration declaration.

Late binding

The syntax is really no different than before except that we choose a different chip name for the bound design entity, it does not have to be the same as the component declaration. Let us suppose that a spec. change (happens too often, doesn't it?) is required. The spec change requires a 3-input AND gate rather than a 2-input multiplexer. One way to tackle this requirement is to use late binding. This requires no change to the MUX2 at all except in the configuration. So, in a hardware sense, we're extracting the AOI gate from its socket and inserting a 4-input AND gate.

Late-binding configuration of MUX2

```
use WORK.all;

configuration AND3_CFG of MUX2 is
  for STRUCTURE
    for G2 : AOI
      use entity work.AND4(quick_fix);
      -- architecture quick_fix of AND4 specified for AOI component
    end for;
  end for;
end AND3_CFG;
```

Doulos Training Courses

Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 27th September 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Order of Analysis

This is one of those topics that doesn't really relate to hardware design. However, you can't ignore this topic as it is so fundamental to simulating (and to a lesser extent, synthesizing) your VHDL designs. Anyhow, let's see how hardware-y we can make it!

Remember design units? So far we have used three of the five VHDL design units. An entity declaration is a *primary* design unit, as is a configuration declaration. An architecture body is a *secondary* design unit. An architecture is a secondary design unit of the corresponding entity declaration primary design unit. Hence,

```
architecture architecture_name of entity_name is
--             secondary              primary
```

To simulate a piece of VHDL code, you need a design environment consisting of an entity declaration and an architecture body and you also need a testbench. The testbench contains an instance of the design plus some VHDL code to apply stimuli to the design that is being simulated. In VHDL terms, the testbench is just another design entity.

OK. On to order of analysis, following on from the hardware analogy we used in the Configuration pages. In the VHDL design world, you have to create the chip package first before you can glue in the die. So, create the chip package (and put it into stores), then create the chip die (and put that in to stores, too). Someone in stores will check to see that a package for the die exists, if not they'll give it back. Once you have done that, you can check the assembled chip out of stores any time you like. We'll look at checking items out of stores later on. In VHDL, checking items into stores is called *analysis*. In VHDL, stores is called a *library* (yes, in VHDL you can have lots of storerooms, hence *a* library not *the* library). So, this requires that before you analyze any architecture into a library, you must first analyze the corresponding entity declaration into the *same* library.

In the real world, you can goof up and put an empty chip package in to stores. You can do the same in VHDL, too. Analyzing an entity declaration into a library is OK. Conversely, a VHDL architecture body must not exist alone (the VHDL storeman will lose that tiny die!); an architecture won't be put into a library unless there's already an entity declaration in the library.

To summarise, VHDL design units are analyzed into VHDL libraries. One default library is provided for you, it is called WORK. Once you have have analyzed all of the design units you need for a simulation run into a VHDL library, you can run the simulation...

Simulating the MUX2

Let's suppose we have a rudimentary Unix or Windows'95 DOS Box command line interface for our VHDL simulator. The `analyze` comand allows you to specify a VHDL library name and a filename in order to analyze the VHDL source code. We'll assume that each VHDL design unit has it's own file. To analyze the VHDL in the correct order for simulation, we would enter the following commands, one after the other,

```
analyze -library WORK -file aoi.entity
```

```
analyze -library WORK -file aoi.v1.architecture
analyze -library WORK -file mux2.entity
analyze -library WORK -file mux2.v1.architecture
analyze -library WORK -file mux2test.entity
analyze -library WORK -file mux2test.v1.architecture
analyze -library WORK -file mux2test.v1.configuration
```

on the command line.

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

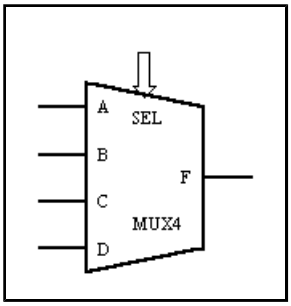
 Doulos Home Page

This page was last updated 20th October 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Vectored Ports and Signals

Back to Components this month.



Ports can represent busses or vectors as well as single bits. In the example above, the port SEL is a 2-bit bus, with msb numbered 1 and lsb numbered 0. The type of the port is STD_LOGIC_VECTOR, which is also defined in package STD_LOGIC_1164 on library IEEE. Objects of type STD_LOGIC_VECTOR are simply an array of STD_LOGIC objects.

Vectors used in a MUX4 design

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

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

In the previous example, an array type was used to allow the description in VHDL of a vectored port. Array types can be used for internal signals as well as ports. This can be seen in the MUX4 testbench code below:

```
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);

end BENCH;
```

The stimulus for the SEL signal is shown as a waveform, made up of individual waveform elements. A waveform element consists of an expression followed by a time specification for when the signal is driven by the value of the expression. In this particular case, the expression is a STD_LOGIC_VECTOR literal. STD_LOGIC_VECTOR values can be written as bit string literals. For example both bits are set to '0 at time 0, while at time 30 ns, SEL(1) is set to '0' and SEL(0) is set to '1'.

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

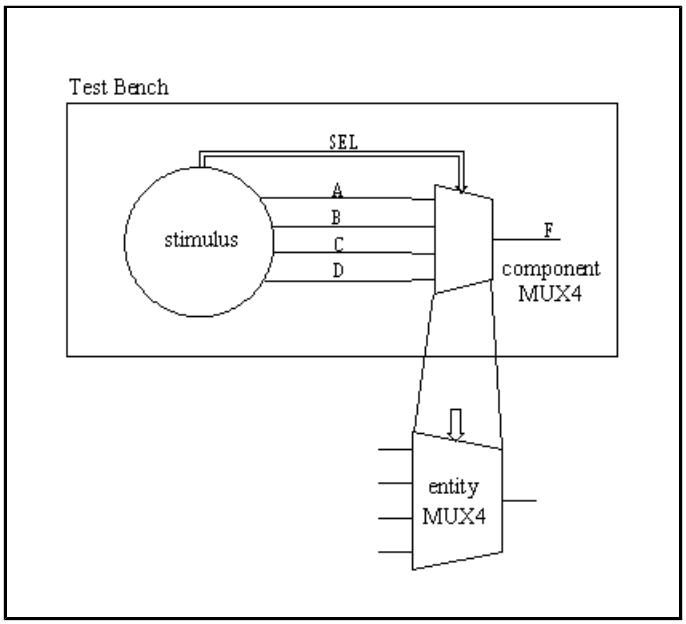
 Doulos Home Page

This page was last updated 14th November 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Test Benches : Part One

So we have a design. But it's unproven. In this tutorial we look at designing a simple testbench in VHDL.



With VHDL, it is possible to model not only the hardware or system design, but also a test bench to apply stimulus to the design and to analyze the results, or compare the results of two simulations. In effect, VHDL can be used as a stimulus definition language as well as a hardware description language.

One consequence of using VHDL to model a test bench is that the test bench is portable between VHDL tools from different vendors.

Test Bench for MUX4

The entity declaration for a test bench (`entity TEST_MUX4 is ... end;`) is usually empty. This is because the test bench itself does not have any inputs or outputs. Test vectors are generated and applied to the unit under test within the test bench. Note that it is illegal to have an architecture body without an entity declaration.

Components

A component declaration is needed (`component MUX4 ... end component;`) in order to pull the MUX4 entity into the test bench. Think of a component declaration as a chip socket. It allows a VHDL *structural* description to be written in a top-down manner, because the VHDL compiler can check the consistency of the design unit that uses the component before the corresponding design entity has been written.

In order that the VHDL simulator can *bind* (more VHDL jargon) the design entity MUX4 to the component MUX4, the names and types of the ports must match between the entity and the

component.

Test Bench for 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

  -- signal assignments to create stimulus

  M: MUX4 port map (...);

end BENCH;
```

 VHDL FAQ

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

 Doulos Home Page

This page was last updated 7th February 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Test Benches : Part Two

This month, we look at writing the VHDL code to realise the testbench based on last month's template.

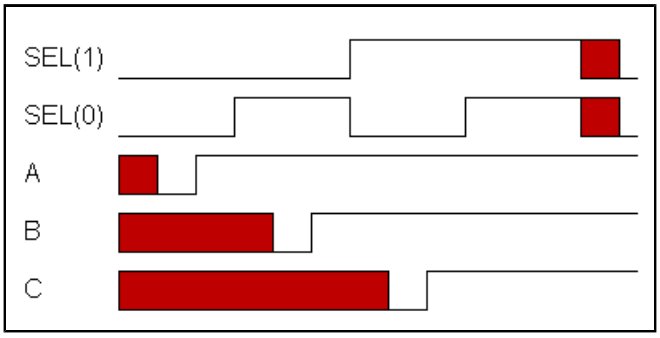
Concurrent Assignment

The 5 concurrent signal assignment statements within the test bench define the input test vectors (eg. A <= 'X', '0' after 10 NS, '1' after 20 NS;). The delays in these assignments are relative to the time when the assignments execute (ie. time 0), not to each other (eg. Signal A will change to '1' at 20 NS, not at 30 NS).

Test Bench for 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;
```

The waveforms generated for the SEL, A, B and C signals are shown below.



VHDL FAQ
Doulos Training Courses
Return to Hardware Engineer's Guide Contents

Doulos Home Page

This page was last updated 27th February 1998

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Summary, so far...

Before we go any further let's summarise where we have got to...

- We know how to create a component in VHDL
- We know how to join components together

Bottom-up design

Joining components together is one method of design using VHDL. It's the bottom-up approach. There's nothing wrong with joining components together to create a design hierarchy, but using it as your design approach exclusively can restrict your productivity even though it is possible to create any design you could wish for using VHDL in this manner. Let's step back a moment and consider how the design of something as simple as a MUX_2 is being implemented; not at a hardware level but at a conceptual level. We'll change the MUX_2 so that it has an inverting data path:

```
signal SELB, FB: STD_LOGIC;

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

Well, that's three components. The largest port list has only 5 elements. There's only two levels of hierarchy. The target technology has to have cells called AOI and INV. Suppose the widths of the A, B inputs have to be changed?

What about a 100k gate ASIC? With multiple levels of hierarchy. Suppose it has to be technology independent, too. Creating large designs in this manner is very time-consuming. It is error-prone (it's just too easy to connect the wrong signal to a port). It is awkward to make changes to the design as a whole (change all the busses in a design by 1 bit, for example to add parity). This approach to creating functionality is often referred to as a *structural* coding style.

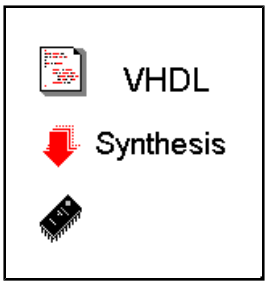
Adopting a structural design approach, we first have to decide ***what*** we want to do (switch between two inputs) and secondly ***how*** to implement it (using components). So, while joining components together has its place, we really ought to move on to a better low-level design approach. Instead of determing the functionality and then implementing it, we're going to describe the functionality and leave it at that. We're not going to worry about implementation. Well, ultimately we will concern ourselves with implementation but from a coding point of view, let's get the functionality or *behaviour* out of the way first.

Coding behaviour

At a conceptual level we can specify ***what*** we want the design to do using a behavioural coding style. Using the structural approach, we have to describe explicitly ***how*** the design is to be implemented. Thus a structural approach implies an extra step in the design process.

So, how do we implement (smile) a behavioural coding style? Well, this can be achieved by writing

VHDL code as though we were writing software rather than describing hardware. We can take our VHDL coding capabilities closer to VHDL programming, closer to writing software rather than describing hardware. By writing software rather than describing hardware, we can concentrate on what a design does rather than how it does it. We can concern ourselves with functionality rather than implementation. In the same way that mathematicians look to abstract a concept to make a problem easier to solve and provide more flexible solutions, we can abstract our designs from VHDL hardware descriptions to VHDL software functions. However, a software function will still at some point become a hardware description. We can bridge the gap from software function to hardware description using synthesis.



With the behavioural approach to design, we still have to decide what we want to do, but that's it. We can use synthesis tools to decide how to implement the functionality we have (behaviourally) specified. There are still two steps in creating a netlist that meets our design goals, it's just that with a behavioural approach we use design automation to complete the second (implementation) step. Hopefully, this can save us time because we as engineers only have to do one job instead of two. Using a software-oriented design approach we can reduce the number of design steps.

Next month, we'll look at how to code up the MUX_2 circuit in a software-oriented manner using VHDL processes.

 Doulos Home Page

This page was last updated 28th October 1998

We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Components vs. Processes

Last month, it was suggested that designs could be created using a more software-oriented *behavioural* style of VHDL coding rather than the explicit structural coding style used to create the MUX_2 design. Before we look at how to describe a MUX_2 using a behavioural coding approach, let's deal with how VHDL designs operate — the MUX_2 will serve as our example.

```
signal SELB, FB: STD_LOGIC;

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

Sequential, parallel or concurrent?

Conceptually, the AOI and INV components operate concurrently. They do not operate in parallel. Sorry for being pedantic but this subtlety is important. Parallel means operate simultaneously, usually without communication between the parallel elements ("never touching" in my dictionary). Concurrency means cooperating, taking place at the same time or location.

In VHDL, we usually speak of elements executing rather than operating (or cooperating), so in VHDL elements can execute concurrently, in parallel or in sequence. We can see that the AOI and INV components execute concurrently — they communicate via the internal signals. You might think that they execute in sequence. (Almost!) If SEL changes and A,B have the same value then the G1 instance of INV executes in parallel with AOI. As FB does not change value the G3 instance of INV does not execute. This simple scenario shows the absence of parallel execution (G3 doesn't execute whilst G1 and G2 are executing) and sequential execution (G3 doesn't execute after G2); this leaves us with concurrency.

We often say to ourselves that the AOI and INV are connected. We now know that being connected means concurrent execution. In our example, the functionality of the MUX_2 is implemented via components executing concurrently.

Processes

To create software-style VHDL, we first have to deal with processes. We can think of a VHDL process as a blob of hardware. Instead of instantiating a component in an architecture, we can instantiate a process.

For our MUX_2 example, let's dispense with concurrency altogether. Let's use a single process. Processes enable you to code up a design by describing the design's functionality using statements executing in sequence. This is different from the way in which components create functionality. Components create functionality by executing concurrently with respect to each other.

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;
```

A process can contain signal assignments to describe the functionality of a design. Thus, we can re-code our MUX_2 example using a single process rather than three component instantiations.

Processes do not have to exist in isolation. A process is a concurrent statement inside an architecture body just like a component instantiation. We know that components can be connected together using signals and so too can processes. So processes execute with respect to each other concurrently, but internally they execute statements in sequence. You can describe functionality using sequential statements inside processes. And you can create multiple processes within an architecture to create your design. But there's a couple of things you can't do — a component instance is not a sequential statement so you can't execute a component inside a process. And you can't embed one process inside another in the way that you can embed component instances within each other, so there's no way to build a process hierarchy. To build a design hierarchy, you have to use components.

An extension to the approach of behavioural coding is to describe the functionality of the design by using software-oriented methods exclusively that don't use components. In VHDL, this is accomplished using processes to replace component instances. Processes enable you to code up a design by describing the design's functionality using statements executing in sequence. Instead of writing:

```
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;
```

We can write:

```
architecture BEHAVIOUR of MUX2 is

  -- signal declarations (no components!)...

begin
  G1: process
    -- software-style VHDL for the INV component
  end process;
```

```
  G2: process
    -- software-style VHDL for the AOI component
  end process;

end;
```

Now, of course it’s inside the processes that all the action takes place! Until next month...

 Doulos Home Page

This page was last updated 28th October 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Processes

In VHDL, the process statement contains sequential statements.

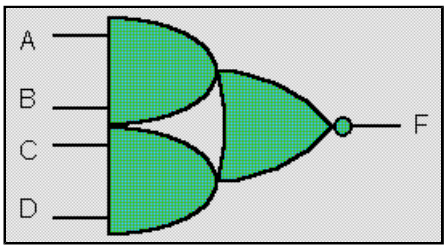
Processes are only permitted inside an architecture. The statements within processes execute sequentially, not concurrently.

Processes can be written in a variety of ways. The most common approach when using processes to describe designs is to use the form that has a sensitivity list.

Carrying on from last month, we shall use two processes in our MUX_2 design, one to replace the AOI gate, the other to replace the inverter. We will then merge the two processes to see how a single process can be used to describe the design.

First of all, let's tackle the AOI gate. All we do is extract the signal assignment from the V1 architecture (see An Example Design Entity) and insert it into the process. At this point, the signal assignment is a sequential signal assignment rather than a concurrent signal assignment. At this stage all we have done is wrap a process around the signal assignment.

```
v1_arch: process -- incomplete at this stage

begin

  F <= not ((A and B) or (C and D));

end process;
```



To make this process complete, we have to remember that we are describing a piece of combinational logic. From a conceptual point of view, the outputs of a combinational circuit CAN change when ANY one of the inputs changes. This means that we need to cause the process to execute when any of the ‘inputs’ to the process changes. The way to do this is to create a sensitivity list for the process from the signals A, B, C and D. The sensitivity list follows the process keyword as shown:

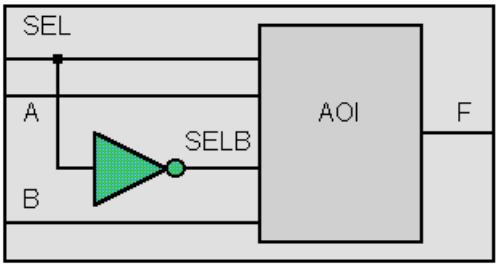
```
v1_arch: process (A, B, C, D)

begin

  F <= not ((A and B) or (C and D));

end process;
```

However, inside the MUX_2 design, we are using the the ports and signals of that design, so:

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

Likewise for the inverter:

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



We can combine these two processes into one:

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

Notice that not SEL from the G1 process was used to substitute the SELB in the G2 process. We now have a single process with a sensitivity list made up from only the MUX_2 ports — no internal signals.

In the next article in this series we will see how to take a more conceptual approach to coding processes, so that we can use more than just signal assignments to describe the behaviour of a design, in this case the MUX_2.

 Doulos Home Page

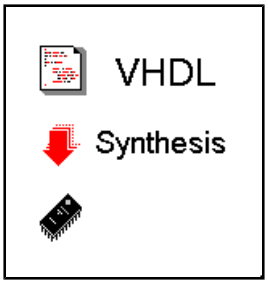
This page was last updated 4th October 1998

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

RTL Coding

Last month, we saw how to use processes to describe a MUX_2 design. However, the coding approach used was somewhat low-level, in that the code consisted of binary operators.

In order to adopt high-level design principles, it is necessary to try and describe a design at a higher level of abstraction. This means thinking about the functionality of the design rather than its implementation. This allows the synthesis tool to optimize the functionality you have specified, leaving you to describe ***what*** the design does, whilst the synthesis tool's job is to implement the design ***how*** it sees fit in order to create the optimal implementation. The style of coding required for synthesis tools is known as RTL coding.



Most commercially available synthesis tools expect to be given a design description in RTL form. RTL is an acronym for *register transfer level*. This implies that your VHDL code describes how data is transformed as it is passed from register to register. The transforming of the data is performed by the combinational logic that exists between the registers. Don't worry! RTL code also applies to pure combinational logic — you don't have to use registers.

Using processes in RTL descriptions is particularly appropriate for input to a synthesis tool. For example, using component instances implies a level of structure in the design. This can sometimes be advantageous particularly on a large design but for a 2-input multiplexer...

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

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

architecture BEHAVIOUR of MUX2 is

begin
  -- descibed using a single process
end;
```

For the functionality, let's get conceptual. If sel is a logic 1, a is routed through to the f output. On the other hand if sel is a logic 0, b is routed through to the f output. Rather than think about routing one of the inputs through to the output let's think about the output *getting* one of the inputs, and let's write the text on separate lines depending upon whether we are making a decision or performing an action (sometimes referred to as *pseudo-code*):

```
if sel is logic 1
  f gets a
otherwise
  f gets b
```

This can be translated into VHDL code:

```
if sel = '1' then
  f <= a;
else
  f <= b;
end if;
```

Now before we go any further, we'll just take this code snippet a line at a time.

`if sel = '1'`
The VHDL language allows for many different kinds of sequential statement. The sequential signal assignment is a signal assignment that can appear inside a process. You have already come across this statement in the Processes page (assignment to `F` in the `combined` process, if you remember). Here's another: the if statement. Actually this line is part of the if-else statement that is the entire code snippet. if is a VHDL keyword. After the if keyword you have a conditional expression, in this case `sel = '1'` — does sel have the value logic 1? **If so...**

`f <= a;`
f gets the value on the a input. But what if sel is not logic 1?

`else`
Otherwise (assume sel is logic 0 — more on this assumption later)...

`f <= b;`
f gets the value on the b input.

`end if;`
simply denotes the end of the if-else statement.

So, as it turns out, we have described the funcionality of the `MUX_2` design using a single sequential statement, the if-else statement. In each branch of this if-else statement, there is an additional sequential statement, either assigning a to f, or b to f, depending upon the value of sel. But we have to remember that sequential statements always live inside a process, so...

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

This now enables us to describe a design using a list of concurrent signal assignments, a hierarchy of designs (using component instances) or a process. Compare the 3 approaches for yourself:

```
  // concurrent signal assignments
```

```
selb <=  not sel;
fb <= not((a and sel) or (b and selb));
f <= not fb;

// a hierarchy of designs
G1: INV port map (SEL, SELB);
G2: AOI port map (SELB, A, SEL, B, FB);
G3: INV port map (FB, F);

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

And of course you can mix'n'match coding styles if you wish. On a simple design, such as a `MUX_2` it is perhaps not apparent how succinct the use of processes is in general compared to component instances and concurrent signal assignments. But you can readily appreciate that the use of just one process in this design is enabling us to describe the design in terms of its functionality without regard to the implementation. You can describe what you want without having to worry about how you are going to implement the design (because *you* don't have to — that's the synthesis tool's job!).

Go on! Read the `MUX_2` design into your synthesis tool and have a play. 'Til next month...

 Doulos Training Courses

 Return to Hardware Engineer's Guide Contents

---

 Doulos Home Page

This page was last updated 4th January 1999

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

If statement

In the last article, we looked at describing hardware conceptually using processes. What kind of hardware can we describe? What are the limitations? What kinds of VHDL statement can be used in always blocks to describe hardware? Well, we have already seen the use of an if statement to describe a multiplexer, so let's dwell on if statements for this month's tutorial.

```
process (sensitivity-list) -- invalid VHDL code!
  -- process declarative region
begin
  -- statements
end process;
```

The code snippet above outlines a way to describe combinational logic using processes. To model a multiplexer, an if statement was used to describe the functionality. In addition, all of the inputs to the multiplexer were specified in the sensitivity list.

```
signal sel, a, b : std_logic;

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

Sensitivity list

It is a fundamental rule of VHDL that only signals (which includes input and buffer ports) must appear in the sensitivity list.

Combinational logic

It transpires that in order to create VHDL code that can be input to a synthesis tool for the synthesis of combinational logic, the requirement for all inputs to the hardware to appear in the sensitivity list is a golden rule.

> **Golden Rule 1:**
>
> *To synthesize combinational logic using an process, all inputs to the design must appear in the sensitivity list.*

Altogether there are 3 golden rules for synthesizing combinational logic, we will address each of these golden rules over the next couple of articles in this tutorial.

If

The if statement in VHDL is a sequential statement that conditionally executes other sequential

statements, depending upon the value of some condition. An if statement may optionally contain an else part, executed if the condition is false. Although the else part is optional, for the time being, we will code up if statements with a corresponding else rather than simple if statements. To incorporate more than one sequential statement in an if statement, simply list the statements one after the other, there are no special bracketing rules in VHDL as there are in some programming languages,

```
signal f, g : std_logic; -- a new signal, g

process (sel, a, b)
begin
  if sel = '1' then
    f <= a;
    g <= not a;
  else
    f = b;
    g = a and b;
  end if;
end
```

If statements can be nested if you have more complex behaviour to describe:

```
signal f, g : std_logic;

process (sel, sel_2, a, b)
  if sel = '1' then
    f <= a;
    if sel_2 = '1' then
      g <= not a;
    else
      g <= not b;
  else
    if sel_2 = '1' then
      g <= a and b;
    else
      g <= a or b;
    end if;
    f <= b;
  end if;
end process;
```

Note that the order of assignments to f and g has been played around with (just to keep you on your toes!).

Synthesis considerations

If statements are synthesized by generating a multiplexer for each register assigned within the if statement. The select input on each mux is driven by logic determined by the if condition, and the data inputs are determined by the expressions on the right hand sides of the assignments. During subsequent optimization by a synthesis tool, the multiplexer architecture *may* be changed to a structure using and-or-invert gates as surrounding functionality such as the **`and`**, **`or`** and **`not`** can be merged into complex and-or-invert gates to yield a more compact hardware implementation.

 Return to Hardware Engineer's Guide Contents

---

 Doulos Home Page

This page was last updated 4th February 1999

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

Synthesizing Latches

In the last article, if statements were used to describe simple combinational logic circuits. Synthesizing the VHDL code produced multiplexing circuits, although the exact implementation depends upon the synthesis tool used and the target architecture of the device.

As well as enabling the creation of multiplexers, if statements can also be used to implement tristate buffers and transparent latches. In this article we will look at how transparent latches are synthesized from if statements and how to avoid the inadvertent creation of latches when you meant to create combinational logic circuits from VHDL code containing if statements.

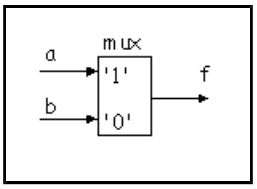
If Statements

In the processes that have been coded up so far, if-else statements rather than simple if statements have been used. Let's use a simple if statement rather than an if-else statement in an example you have already seen:

```
signal sel, a, b : std_logic;

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

becomes...

```
signal sel, a, b : std_logic;

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



Note that the behaviour being described is the same. In the `pure_if` process, `f` initially gets `b`. Only if `sel` is active HIGH does `f` get `a`. This is perhaps a slightly odd way to describe a multiplexing circuit but it is accepted by all synthesis tools. Synthesis tools expect to create circuits responding to only binary ('0' and '1' in VHDL) values. As far as a synthesis tool is concerned if sel is '1' a is routed

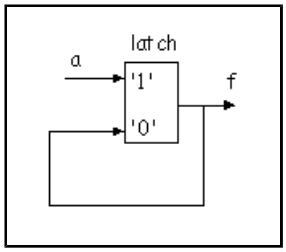
through to f. If sel is not '1' it must be '0' and thus sel being '0' leaves f being driven by the initial assignment from b.

Let's lose the b input to the process so that we have:

```
signal sel, a : std_logic;

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

Incomplete Assignment

Now analyze the behaviour of the code. If sel is '1', f gets a. But what happens when sel is '0'? Well, very simply, nothing! f does not and can not change. When sel is fixed at '0', we can change a as much as we like, f will not be assigned the value of a. If we suppose that an if statement synthesises to a multiplexer, then we must be able to configure the multiplexer such that f only gets the value of a when sel is '1'. This can be achieved by feeding back the multiplexer f output back to the '0' input. in hardware terms this is a transparent latch and this is exactly the hardware synthesized by a synthesis tool given this VHDL code.



If the target architecture does not contain transparent latches the synthesis tool will generate multiplexer circuits that employ combinational feedback in order to mimic the latching behaviour required.

Now, this is very well but what's really happening here? One minute if statements create multiplexers, the next they create latches. Well, it's not the if statements, but the process as a whole that counts. If it is possible to execute a process without assigning a value to a signal in that process, the signal will be implemented as a transparent latch. This is known as incomplete assignment.

> **Golden Rule 2:**
>
> *To synthesize combinational logic using a process, all objects must be assigned under all conditions.*

Note that this golden rule states that all *objects* must be assigned under all conditions. This is because signals are not the only VHDL objects, there are also *variables* in VHDL. In a later tutorial, we will

use variables inside a process to describe functionality.

Simplifying code analysis

Suppose you are creating a process to describe combinational logic. This process consists of nested if-else statements as follows:

```
signal f, g : std_logic;

process (sel, sel_2, sel_3, a, b)
begin
  if sel = '1' then
    f <= a;
    if sel_2 = '1' then
      g <= not a;
    else
      g <= not b;
      if sel_3 = '1' then
        g <= a xor b;
      end if;
    end if;
  else
    if sel_2 = '1' then
      g <= a and b;
    else
      if sel_3 = '1' then
        g <= a nand b;
      -- oops! no else
      -- else
      --   g <= ...
      end if;
    end if;
    f <= b;
  end if;
end process;
```

Will you get transparent latches on the f and g outputs? Not easy is it? If you look carefully you will see that in fact, g is latched when sel is '0', sel_2 is '0' and sel_3 is '0'. The ‘oops!’ comment should help you to see where the complete assignment is NOT made.

Fortunately, it is possible to save yourself the bother of scouring through the process code to locate possible incomplete assignments by setting signals to default values at the start of the process. Using this approach you may get undesired functionality if you have missed out an assignment (which should be easy to fix) as opposed to unwanted transparent latches. For our current example,

```
process (sel, sel_2, sel_3 a, b)
begin
  -- default values assigned to f, g
  f <= b;
  g <= a and b;
  if sel = '1' then
    f <= a;
    if sel_2 = '1' then
      g <= not a;
    else
      g <= not b;
      if sel_3 = '1' then
        g <= a xor b;
```

```
      end if;
    end if;
  else
    if sel_2 = '1' then
      g <= a and b;
    else
      if sel_3 = '1' then
        g <= a nand b;
      end if;
    end if;
  end if;
end process;
```


Doulos Training Courses


Return to Hardware Engineer's Guide Contents


Doulos Home Page

This page was last updated 3rd June 1999


We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*

VHDL FAQ

This list of FAQs and any mention of specific organizations or their products does not imply an endorsement by Doulos of either the organisation or the product.

- FAQ comp.lang.vhdl part 1 (Introduction)
- FAQ comp.lang.vhdl part 2 (Books)
- FAQ comp.lang.vhdl part 3 (Products and Services)
- FAQ comp.lang.vhdl part 4 (Glossary)

If you would like us to add your FAQ to this list, please e-mail us with the URL of your VHDL-related FAQ and we will include it.

The Doulos VHDL FAQ

What is the difference between VHDL and Verilog?
On the surface, not that much. Both are IEEE standards and are supported by all the major EDA vendors. Both can be used for designing ASICs and simulating systems. However, VHDL is altogether a grander language. Its support for system level modeling and simulation is far more comprehensive than Verilog. However, VHDL requires longer to learn and is not so amenable to quick-and-dirty coding. As a final thought it is likely that many hardware engineers will one day be bi-lingual in both VHDL and Verilog.

Can I use VHDL for the analog part of a design?
No. You can't **design** analog circuitry in VHDL, however you can **model** analogue circuitry in VHDL (said this digital designer!). In theory, VHDL can be used to model the behaviour of any system or component. However, VHDL does not offer the same level of modeling accuracy as say Spice, without an awful lot of work (start writing those differential equation solvers, now). For examples of using VHDL to model analogue circuitry, check out:

- **Applications of VHDL to Circuit Design**
  edited by Randolph Harr and Alec Stanculescu Kluwer Academic Publishers, 1991, 256 pp, ISBN 0-7923-9153-5

For information on the Analog Working Group (1076.1) that is engaged in providing analogue design capabilities within VHDL, please try the Analogue Home Page.

How must I write VHDL to make it synthesisable?
Writing VHDL for synthesis is not particularly difficult, but you need to be disciplined, not only in your use of VHDL syntax but also your approach to writing VHDL for synthesis. It is this latter aspect which many engineers overlook; thorough training is really the only way to avoid making poor strategy decisions in writing synthesisable VHDL. Check out our Tip of the Month for March 1996 in order to gain an appreciation of the issues involved.

How many versions of VHDL are there?
There are two. The original release of the VHDL language occured in 1987 with the adoption of the Language Reference Manual as an IEEE standard. In 1993, the IEEE-1076 standard was modified and ratified and became known as VHDL'93. At the current time, the only simulator we know of that claims to offer full VHDL'93 support is Model Technology's V-System simulator.

A VHDL design can be moved to any tool or technology. Right?
On the face of it, this is true. VHDL was designed to be and is a technology independent design language. However, there is less of a compliance issue between different simulators than there is for synthesis tools. Generally speaking, moving VHDL code from one simulator to another involves one or two minor changes to the VHDL. Two different synthesis tools may support two quite different VHDL subsets. This is particularly an issue for us at Doulos in developing our training courses, because we like to present a reasonably generic approach to writing VHDL for synthesis. This means that the VHDL we teach you is guaranteed to be more transportable between synthesis tools than it otherwise would be. Our pain is your gain! In addition because we are so aware of the differences between synthesis tools, this means that we emphasise the best way of writing VHDL to get the best from your synthesis tool.

Are there any tools to generate VHDL test benches automatically?
A quick scan of the Internet reveals that there are no automatic testbench generation tools. The VH Structural Code Generator which is shipped with the Doulos VHDL PaceMaker Project Edition will give you a testbench template, but you'll still have to fill in the stimulus and monitoring sections. You should bear in mind that the creation of functional tests for your VHDL designs is one of the major tasks involved in designing with VHDL.

Can you give me a measure of the productivity improvements I should expect from VHDL?
Well, do you believe the hype! Yes, ultimately there are considerable productivity gains to be had from using high-level design techniques in conjunction with synthesis technology, providing that your designs are: complex amenable to synthesis not dependent upon the benefits of a particular technology

Obviously, complex means different things to different people, but a good rule of thumb is that complex means the implementation part of the design process is considerably more awkward than the specification phase. Let's face it, if the specification phase is significantly longer than the implementation phase, you need to put effort here, not into HLD. Of course, once you are benefiting from HLD productivity gains, the specification phase becomes more significant. OK, that's HLD: VHDL is a HLD design entry language, so we would expect the use of VHDL with synthesis technology to improve productivity in the design process. However, you won't get those benefits immediately. Your first VHDL-based project will probably take slightly longer than if you had used your previous design process. Where you really win out is second time around. In order to reduce the time spent on your first project and to ensure that subsequent projects benefit from good VHDL design practices, you need to ensure that your engineers are well trained (well, we would say that wouldn't we!)

Are there translators from 'C' to VHDL?
Once again, a quick surf of the Internet reveals that there are no C to VHDL translators. There are a few Verilog to VHDL translators, however, a visit to FAQ comp.lang.vhdl part 3 will be of interest if you need a translator. For including C routines into your VHDL code, VHDL'93 defines the foreign attribute for subprograms. This allows you to call object code from within your VHDL simulation.

Note that the source code language for the routine is un-defined - it could be Pascal or Lisp, for example; the format of the object code though must be supported by the simulator.

I can see how to write abstract behavioural descriptions in VHDL, but how do you describe and simulate the actual hardware?
This is probably the biggest hurdle that many hardware engineers face when moving to VHDL. After all, sometimes we need to be able to describe actual implementation as well as abstract functionality. The way to describe "physical" hardware in VHDL is to write VHDL models of those components. This is supported in VHDL through the use of instantiation. VHDL does not allow you to physically simulate your hardware. You can only simulate a model of that component in a VHDL simulation. Historically, gate-level simulation using VHDL has been notoriously slow. This led to the creation of the 1076.4 working group to provide a mechanism to allow fast gate-level simulation using VHDL. Their effort became known as the VITAL standard. VITAL is not a VHDL issue for you, but an EDA vendor/ASIC supplier issue. A simulator is VITAL compliant if it implements the VITAL package in its kernel (this is faster than simulating the VITAL primitives in the VITAL package). You don't need to change your VHDL netlist; your ASIC vendor needs to have a VITAL compliant library though, in order for you to take advantage of the simulation speed up. Thus the ASIC vendor's library elements need to be implemented entirely in VITAL primitives.

I've heard that VHDL is very inefficient for FPGAs. Is that true?
How can VHDL be inefficient for FPGA design? It's a hardware description language. Of course, the problem is with synthesising VHDL to FPGA target technology. Generally, synthesis tools are developed for ASIC target technology, particularly gate arrays, whose architecture is fine-grained, that is a sea of simple gates (2, 3, 4-input AND, OR gates, single flip-flops, etc) This is obviously a mismatch to the architecture of most FPGAs, with their CLB (complex logic block) coarse-grained structure. This problem has been tackled in two ways. Crosspoint FPGAs mimic the fine-grained structure of ASICs in order to allow you to use conventional ASIC synthesis tools. The more common approach has been for EDA vendors to develop FPGA-specific algorithms as part of an existing synthesis tool (for example, Synopsys' FPGA Compiler) or to develop FPGA-specific synthesis tools (for example Neocad's tools, now owned by Xilinx).

Are there any VHDL source code libraries available to save me having to re-invent common code fragments and functions?
There are a few libraries available for most levels of VHDL design. The IEEE library contains very low-level type-and-function packages. The std_logic_1164 package has become an industry standard. Hardly anyone writes a re-usable VHDL component without using this package for the STD_LOGIC and STD_LOGIC_VECTOR type definitions. For libraries of components, Doulos offer a Model Library as part of the VHDL PaceMaker Project Edition. The VHDL Technology Group offer a range of functions and models for VHDL programmers.

Are freeware / shareware VHDL tools available?
Generally, the answer is no. Some EDA vendors offer limited use demo versions of their tools. For example, Accolade provide a demo version of their EDA package. The Alliance tool suite including simulation and synthesis tools is available via anonymous FTP from cao-vlsi.ibp.fr/pub/alliance. Details on more tools can be found at FAQ comp.lang.vhdl part 3.

Are there any inexpensive VHDL tools available?
Yes, there are one or two. Cypress Semiconductor offer a low-cost simulation and synthesis toolset for their PLDs. For general-purpose use, Green Mountain offer a low-cost compiler, see FAQ

comp.lang.vhdl part 3.

What is Synthesis?
Synthesis is the stage in the design flow which is concerned with translating your VHDL code into gates - and that's putting it **very** simply! First of all, the VHDL must be written in a particular way for the synthesis tool that you are using. Of course, a synthesis tool doesn't actually produce gates - it will output a netlist of the design that you have synthesised that represents the chip which can be fabricated through an ASIC or FPGA vendor.

Are there many books on VHDL?
Yes, there are quite a few these days. Check out the booklist kept at VHDL UK Home Page - there are even some reviews here, too.

How about on-line information resources?
You're already here! Try the VHDL section of our High Level Design Library for examples of VHDL models and assorted tips and tricks. On our "Where to go next..." page you'll find links to other EDA-related Web sites. In addition, check out the comp.lang.vhdl newsgroup.

 Where to go next...
 Doulos Training Courses

---

 Doulos Home Page

This page was last updated 26th March 1996.

 We welcome your e-mail comments. Please contact us at: *webmaster@doulos.co.uk*