A User's Experience with SystemVerilog

Jonathan Bromley
Michael Smith

Doulos Ltd, Ringwood, UK

jonathan.bromley@doulos.com
michael.smith@doulos.com

ABSTRACT

There has been much discussion about the language features introduced in Accellera's SystemVerilog extensions to Verilog. Now that tools that support SystemVerilog are becoming available, it is important to know how the language will actually be used in the hardware design process. This paper describes our experience with the language in the context of a specific design. It is hoped that sharing this experience will be of benefit to anyone using or considering the use of SystemVerilog.

1 Introduction

In recent years, SystemC has gained favour as a means to model and validate the behaviour of electronic systems at a high level of abstraction, while traditional hardware description languages (HDLs) VHDL and Verilog have retained their dominance of design entry and module-level verification. Meanwhile, specialised hardware verification languages (HVLs) and their associated tools have begun to make a significant impact on both module-level and system-level verification.

Accellera's extensions to IEEE-Std.1364 Verilog, usually known as SystemVerilog [Accellera 2003], hold out the promise of a single unified language to span almost the entire system-on-chip (SoC) design flow, from module-level design and gate-level simulation all the way up to system-level verification. Practical tool support for SystemVerilog is now becoming available, and this paper outlines an attempt by the authors to take a design of modest size from concept through to implementation using SystemVerilog wherever possible.

2 Design Example

The current paper reports our experience in implementation and verification of the design outlined in *Figure 1*. At its core is a CORDIC calculator module; this device is then made available to the rest of a system by giving it an interface conforming to the AMBA Advanced Peripheral Bus (APB) specification [ARM 1999], chosen for its straightforward design and widespread adoption.

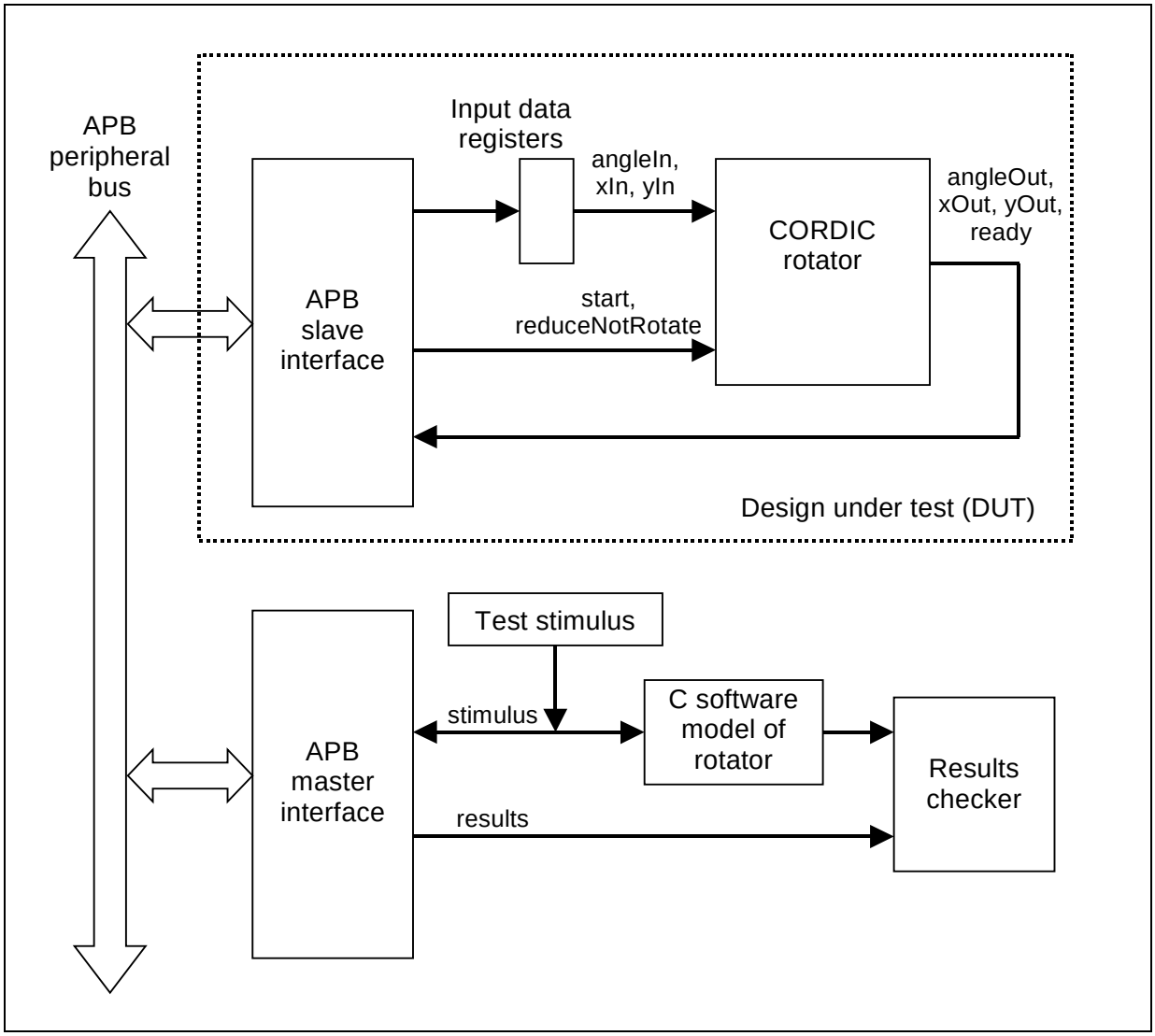

Figure 1: Design under test and its verification environment

2.1 CORDIC calculator

Our calculator module implements the well-known CORDIC algorithm [Volder 1959]. Its basic operation is to take a vector expressed as an (`xIn`, `yIn`) coordinate pair, and rotate it through an angle provided as a third input value `angleIn`. When this operation is complete, the rotated vector is available on output ports `xOut`, `yOut`. Used in this mode the module naturally lends itself to the calculation of trigonometric functions sine and cosine, and can also be used in modulator applications and as a polar-to-cartesian converter.

Our implementation of the module has a single-bit mode input `reduceNotRotate`. If this bit is clear when a calculation is initiated, CORDIC rotation is performed as described in the previous paragraph. However, if `reduceNotRotate` is set, a somewhat different version of the algorithm is applied in which the angle input `angleIn` is ignored and the input vector is rotated until its *y* component reaches zero. The angle of rotation required to reduce the *y* component to zero is made available on output port `angleOut`. This mode of operation is appropriate for computation of the arctangent function, and for cartesian-to-polar conversion.

Our calculator module is parameterised for bit-width, but we have chosen to use 16-bit data throughout as a specific example.

An excellent survey of theory, applications and practical implementations of CORDIC can be found at [Andraka 1998].

2.2 Controlling the calculator module

The calculator module needs 16 clock cycles, one for each iteration of the algorithm, to calculate each new result. Timing of this interaction is managed by two handshake signals `start` and `ready`, as illustrated in *Figure 2*. Asserting `start` whilst a calculation is in progress will cause the current calculation to be abandoned, and the new calculation will then begin.

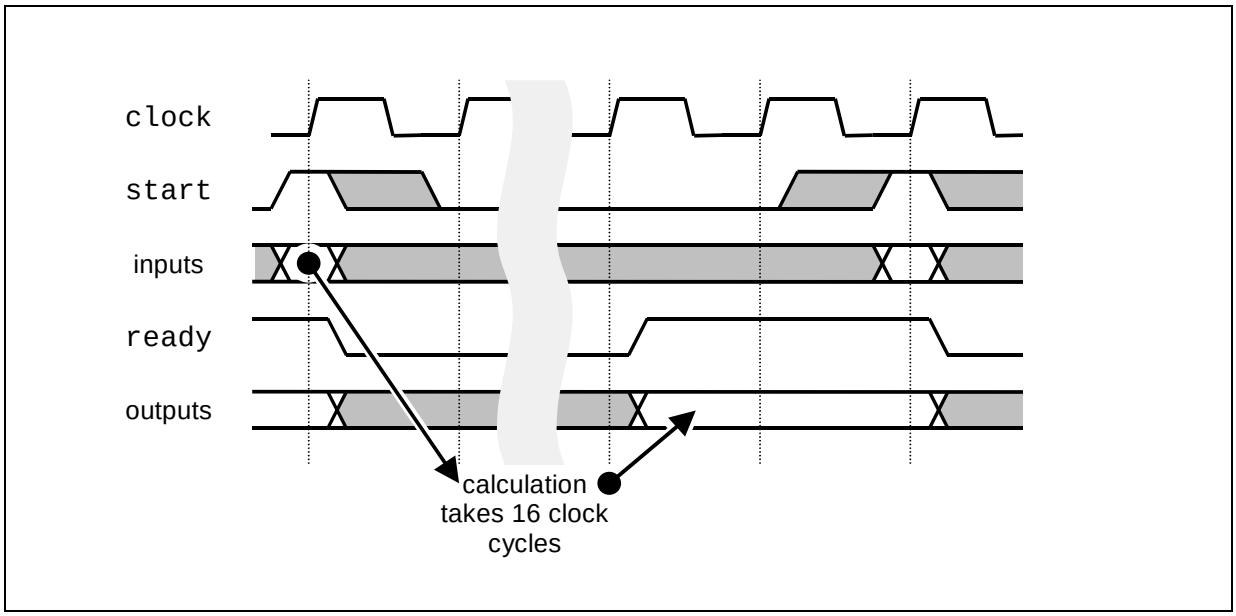

Figure 2: Control of the CORDIC calculator module

2.3 APB interface

APB, fully described in [ARM 1999], is a simple synchronous peripheral bus having separate parallel address and data lines. The bus can have only one master and an arbitrary number of slaves. APB protocol defines just two bus cycles, read and write; these cycles each access a single register in a single slave, and occupy two cycles of the bus clock. *Figure 3* outlines the timing of these bus cycles.

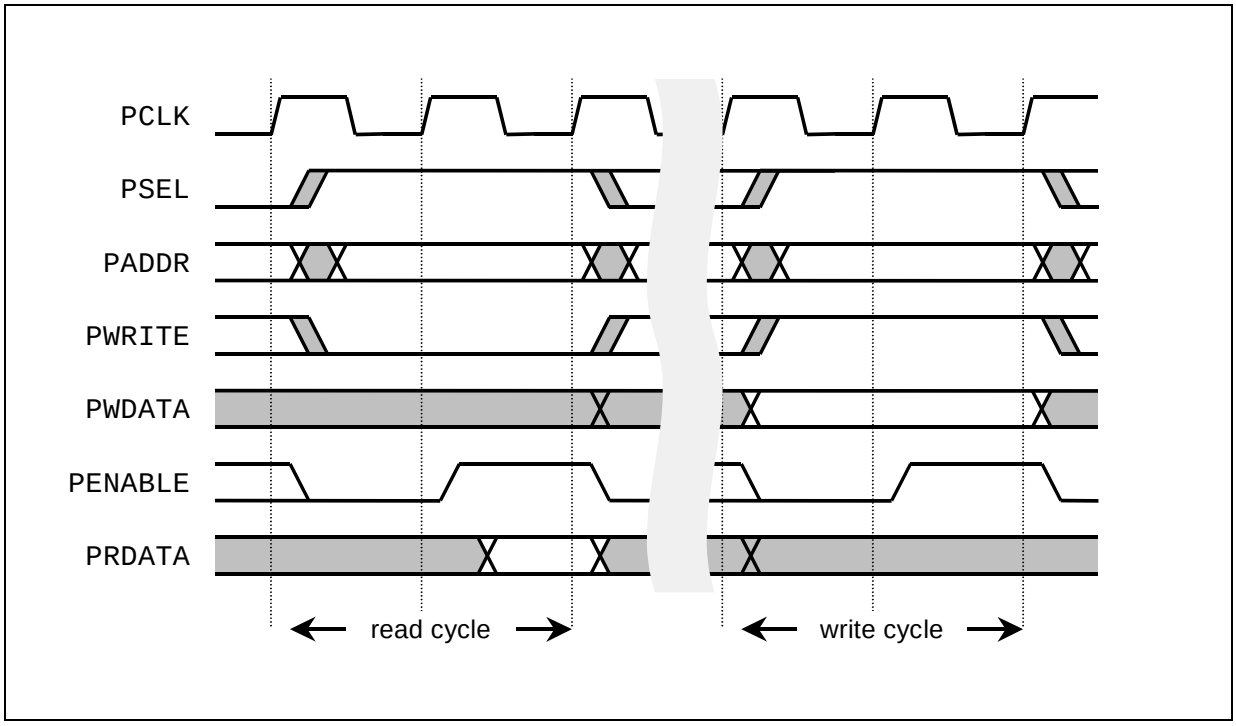

Figure 3: APB read and write protocol

APB address and write data are broadcast by the master to all slaves. Read data, however, uses a separate read data bus per slave; these multiple read data buses are multiplexed on to the master's read data bus, as shown in *Figure 4*. This method avoids the need for three-state bus interconnect.

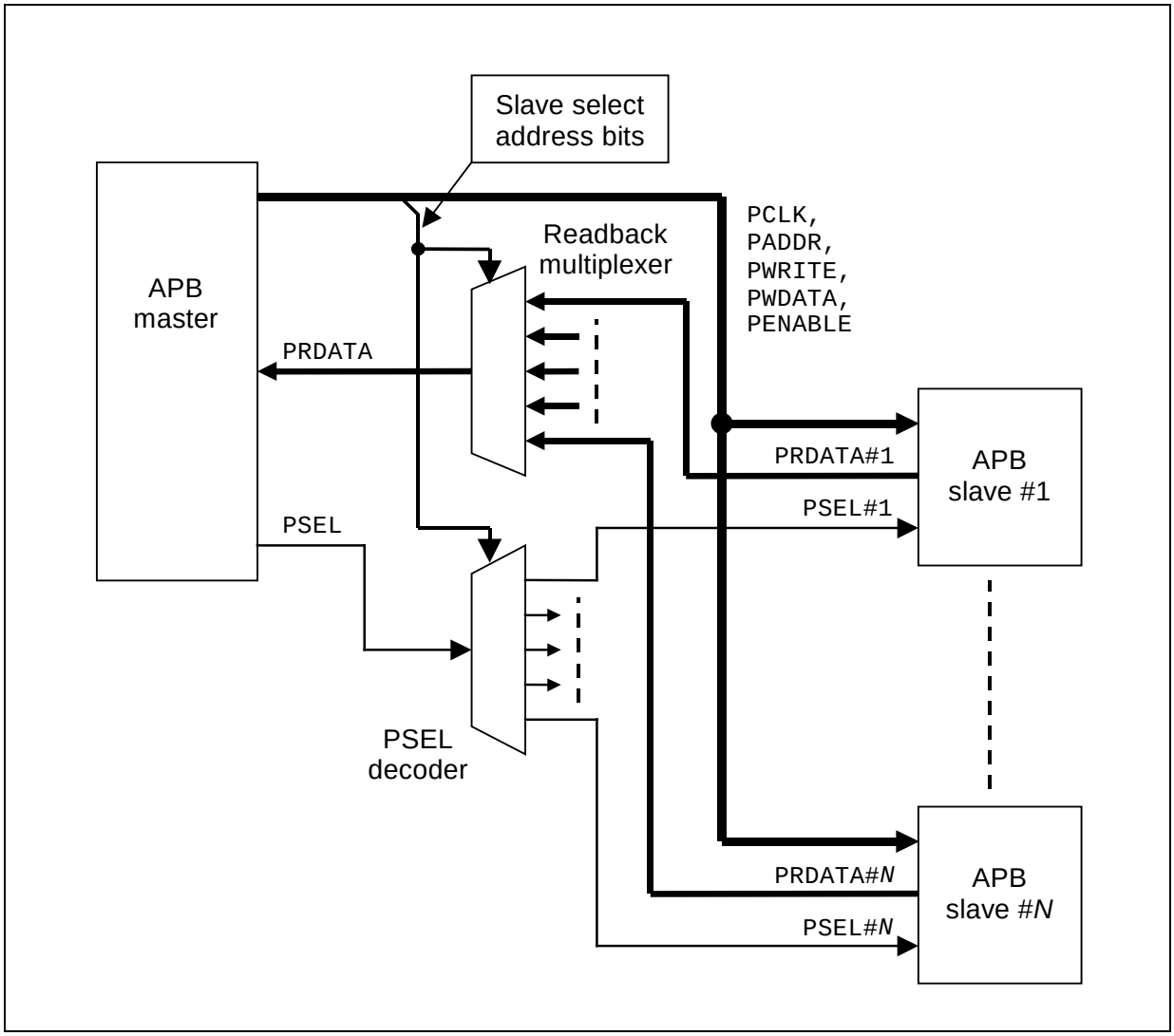

Figure 4: APB read data multiplexer

3 Making use of SystemVerilog in the design

We aimed from the outset to exploit SystemVerilog and Verilog-2001 features wherever appropriate, using Synopsys VCS version 7.1 for simulation. Those parts of the design that should be synthesisable (the CORDIC calculator and bus interface) have been coded using SystemVerilog constructs that we may reasonably expect to be synthesisable. However, synthesis tool support for SystemVerilog is quite limited at the time of writing, and consequently the scope of synthesisable SystemVerilog has yet to be stabilized by tool vendors.

3.1 Verilog-2001 constructs for synthesisable design

Support for Verilog-2001 in simulation and synthesis tools is now widespread, and it offers many features that greatly improve the clarity and expressiveness of synthesisable Verilog code. Until recently the authors have avoided using or recommending Verilog-2001 because of concerns over portability and tool support. However, for this case study we were able to use Verilog-2001 constructs with confidence.

3.1.1 Signed vectors

Signed vectors free the RTL programmer from concerns about sign extension, and hence lead to more readable code for arithmetic data path functions such as our CORDIC processor. The programmer must remain vigilant, however, because Verilog arithmetic lapses into unsigned behaviour in a number of cases that are not intuitively obvious. Part-selects of a signed vector are treated as unsigned even if the most significant bit is included in the part-select. Arithmetic expressions with one or more unsigned operand are treated as unsigned arithmetic, even if some of the operands are signed.

3.1.2 localparam

Named constants are a vital tool in the creation of portable, readable code. `localparam` declarations provide named constants with module-local scope that cannot be inadvertently overridden, a highly desirable and useful improvement.

3.2 SystemVerilog constructs for synthesisable design

3.2.1 Parameterised type definitions

Although the APB bus specification describes bus protocol and control signals in great detail, it does not mandate any specific width for data and address bus; designers are expected to choose bus widths appropriate to the system design. Similarly, our CORDIC processor design can be easily reconfigured for varying data path widths up to 32 bits, depending on the application's requirements for numerical precision.

Traditionally this kind of parameterisation for bit width has been accomplished using module parameters. However, module parameters in a synthesisable design must be overridden at the instantiation site. This is unhelpful when the bit width of a module instance is to be determined by some property (typically bit width) of a sibling module instance, not controlled by the parent module.

Globally visible data type definitions provide an attractive alternative solution, available for the first time in SystemVerilog. We defined data types to represent unsigned and signed integers both on the APB data bus and on ports of the CORDIC rotator module. We envisage such definitions being encapsulated in project-wide files, used in the same way as packages in VHDL and bringing the same benefits.

SystemVerilog3.1a is expected to add a `package` feature, allowing users to create type definitions that can be imported at will into various parts of the design whilst remaining hidden from all other parts.

3.2.2 Array size inquiry functions

Designs that use globally-defined types may need to determine the bit width of those types, in order to create other values or types of appropriate width. The new SystemVerilog inquiry function `$bits()` provides the required functionality. *Example 1* shows how `$bits()` can be used to construct a constant whose most significant bit is set, although the width of the constant is not otherwise known. We found this feature made it much easier to write parameterised modules that can infer all their parameterisation from an external, global type definition.

```
T_sdata D;  // globally defined type
localparam Width = $bits(T_sdata);
...
D = {1'b1, {(Width-1){1'b0}} };
```

Example 1: Using `$bits()` to construct a sized constant value

3.2.3 Type compatibility

SystemVerilog follows the spirit of Verilog in taking a very relaxed approach to type compatibility. We found that this could lead to some unexpected results. For example, our design defined two different named data types: a packed 16-bit vector intended to represent the APB data bus, and a packed 16-bit vector intended to represent a signed integer in the CORDIC processor. Inevitably, at the processor's bus interface it was necessary to copy values of one of these types on to a variable of the other type. SystemVerilog freely permits this copy operation, with neither type conversion nor casting required.

An alternative approach would be to use two different unpacked types in this situation. Objects of the different types would no longer be copy-compatible, and would require a type cast operation when copying. However, the convenient and intuitively satisfying property of packed types, that their bit pattern is well-defined and readily matches hardware, is lost. One attractive compromise is to create an unpacked struct type having a single member of packed type. This artifice gives both the benefits of strong typing and the convenience of packed data types, at the expense of somewhat cumbersome syntax when accessing objects of the unpacked struct type.

It seems to us that SystemVerilog's type system has excellent expressive power and contributes well to self-documentation. However, the authors' opinion is that the assignment-compatibility of different named packed types significantly weakens its usefulness, although it is clear that it can be very convenient in some situations. We are happy to accept that this opinion may be coloured by our experience of VHDL and other strongly-typed languages.

3.2.4 Default port connections

SystemVerilog's default port connection mechanism using the new `.*` notation was used whenever appropriate in both synthesisable code and verification code. *Example 2* shows the instantiation of our CORDIC processor into its enclosing bus-interface module.

```
CORDIC_par_seq #(
      .guardBits(3),
      .stepBits(4)
     )
    processor (
      .reduceNotRotate(bus.PWDATA[0]),
      .clock(bus.PCLK),
      .reset(async_reset),
      .*
    );
```

Example 2: Instantiation using default port connection

The ability to mix named port connection with default connection, as shown in this example, appears natural and flexible. It reduces unnecessary repetition of port names whilst retaining the flexibility to make any specific connections that may be required, as shown here.

We believe that this feature will prove to be a significant benefit to SystemVerilog programmers.

3.2.5 Structural drivers on variables

SystemVerilog lifts IEEE-1364's restriction whereby structural drivers (namely, continuous `assign` statements and connections through a module's port) may drive only objects of net type, and procedural assignments may drive only variables. In SystemVerilog as in standard Verilog, any number of procedural assignments may apply values to a variable. SystemVerilog additionally permits a variable to be driven by precisely one structural driver, in which case there may not be any procedural assignments to it.

This extension makes the net data types redundant in many practical applications. Tri-state and other multi-drop schemes cannot be modelled in this way, but any signal having only one driver can be modelled using the full expressive power of SystemVerilog's new variable data types.

We found this enhancement to have great practical utility when creating our design. Our traditional experience with Verilog has been that a signal originally declared as a `reg` may well be changed to a `wire`, or vice versa, as the design develops and different RTL structures are used to apply values to that signal. In SystemVerilog the signal can be declared as `logic` with confidence, and this declaration will not need to change as the RTL design evolves. Furthermore, the simulator will inform us at elaboration if we have mistakenly applied more than one structural driver to a `logic` or other variable. In this way, many mistakes that are easily made as a design evolves can be caught long before simulation begins.

Although our design did not lend itself to extensive use of structured data types, we note also that structural drivers on variables make it possible to use any data types both on module ports and on the signals connected to them.

3.2.6 struct and other compound data types

SystemVerilog distinguishes between `packed` and `unpacked` data structures. `packed` structures, typified by ordinary vector variables such as

```
logic [7:0] DataByte;
```

are guaranteed to be stored as a contiguous array of bits so that it is possible to treat them as numeric values. By contrast, `unpacked` structures are stored in an implementation-dependent fashion; a value of `unpacked` structured type cannot be handled as a single unit, except when copying it to a target of the same type.

We considered the use of `packed struct` data types, for example when representing the control registers of our processor module. Our module has a readable register containing just two useful bits, `interrupted` and `busy`. *Example 3* shows the traditional Verilog approach to mapping this readable register on to 16-bit bus data. *Example 4* shows the corresponding SystemVerilog `struct` definition that we used in the design.

```
assign status_word = { 14'b0, interrupted, busy };
```

Example 3: Assigning individual bits to parts of a word

```
typedef struct packed {
    logic [15:2] junk;
    logic        interrupted;
    logic        busy
  } T_status;
```

Example 4: SystemVerilog `struct` representing a word containing status bits

Using the type definition from *Example 4* we could make a variable of type `T_status` and freely copy it to and from an ordinary 16-bit `logic` vector. However, we found this encouraged us to make assignments to the various components of the structure from widely different places in our RTL code. This can give rise to unexpected and awkward errors, because SystemVerilog insists that all components of a packed structure variable must have the same kind of driver (all procedural, or all structural). It is easy to fall foul of this rule if assignments to the various parts of the structure occur far apart in the source code. We therefore chose to keep the various components of the register quite separate, and gather them into a single 16-bit word using a simple continuous assignment similar to that in *Example 3*. Alternatively we could have chosen to represent the status value in an unpacked struct, which can have a mixture of structural and procedural drivers on its various elements. This coding style issue, combined with the weakness of type checking in SystemVerilog, led us to be less enthusiastic about structured data types than we had originally anticipated.

4 Making use of SystemVerilog for verification

4.1 Creating a reference model

An untimed-functional reference model was written in standard C using ordinary trigonometric functions from the C floating-point math library. Several "helper" functions were written to convert between 16-bit fixed-point representation and C `double` values. These helper functions made it possible for the reference model to accept 16-bit fixed-point inputs and outputs whilst using standard floating-point library functions internally. At this stage, no attempt was made to model the CORDIC algorithm itself in C.

Simulation tools available to the authors do not yet support SystemVerilog's Direct Programming Interface (DPI). Consequently we chose to use the *DirectC* feature in VCS 7.1 [Synopsys 2003a], which offers `extern "C"` declaration of functions to be imported into Verilog, with closely similar functionality to DPI's `import "DPI"` mechanism. Our C functions were straightforward and required only the simplest functionality of DPI or DirectC.

It was clear that C functions can be integrated much more easily in this way than through the PLI. We suspect that many verification engineers who have avoided PLI programming because of its complexity will be able to make effective use of DirectC or DPI to integrate reference models and other verification functionality into their SystemVerilog simulations.

4.2 SystemVerilog for verification

Tools available to the authors at the time of writing support only a limited subset of the extensive SystemVerilog 3.1 verification features. As mentioned in the conclusion, we aim to take this case study forward, adding further verification features as tool support becomes available. The source code available on the authors' web site makes extensive use of SystemVerilog assertions to check for correct behaviour at various interfaces in the design.

4.3 Use of SystemVerilog interfaces

As already mentioned, our sample design has an interface to a standard bus structure. We took this as a natural opportunity to exploit SystemVerilog's *interface* mechanism, allowing module-like encapsulation of communication that can nevertheless be connected to a port of a module instance.

4.4 The APB interface

An `interface` was created to model the APB interconnect bus. This bus does not require tri-state or multi-drop interconnect, and therefore it was appropriate to model all the bus signals as vector or scalar `logic` variables within the interface itself.

Using variables to represent bus signals in this way has one especially attractive consequence: the variables can be manipulated by tasks in the interface itself. Such tasks can implement the execution of a bus cycle by some external bus master, in the same manner as a traditional Verilog Bus Functional Model (BFM). Consequently we were able to create a behavioural test fixture that invoked these BFM tasks to perform bus cycles and thus verify correct operation of the design under test (DUT).

4.5 Modports

We provided three different `modports` on the APB interface.

4.5.1 RTL master modport

This modport is intended for connection to a bus master module. Such a bus master module should directly drive all physical signals in the bus, except the readback data signal `PRDATA` driven by slaves on read cycles. There must be only one master on an APB bus.

4.5.2 RTL slave modport

This modport is intended for connection to a slave module such as our DUT. Bus slaves should receive all physical signals in the bus, except for their readback data signal. There can be many slave modules on a single APB bus segment, but the readback data multiplexer must have as many inputs as there are slaves (see *Figure 4*) so that the correct slave's data may be returned to the bus master on read cycles. As we describe later, implementation of this readback multiplexer seems problematic in SystemVerilog.

4.5.3 Behavioural master modport

This modport is intended for connection to a test harness. The test harness is expected to drive the bus clock in a straightforward manner, but it should perform read and write cycles by invoking tasks implemented within the interface itself. These tasks are presented through the modport using `import task`, so that the test harness does not need to make hierarchical name reference to the interface but instead can work through its named port. This arrangement is illustrated in *Figure 5*.

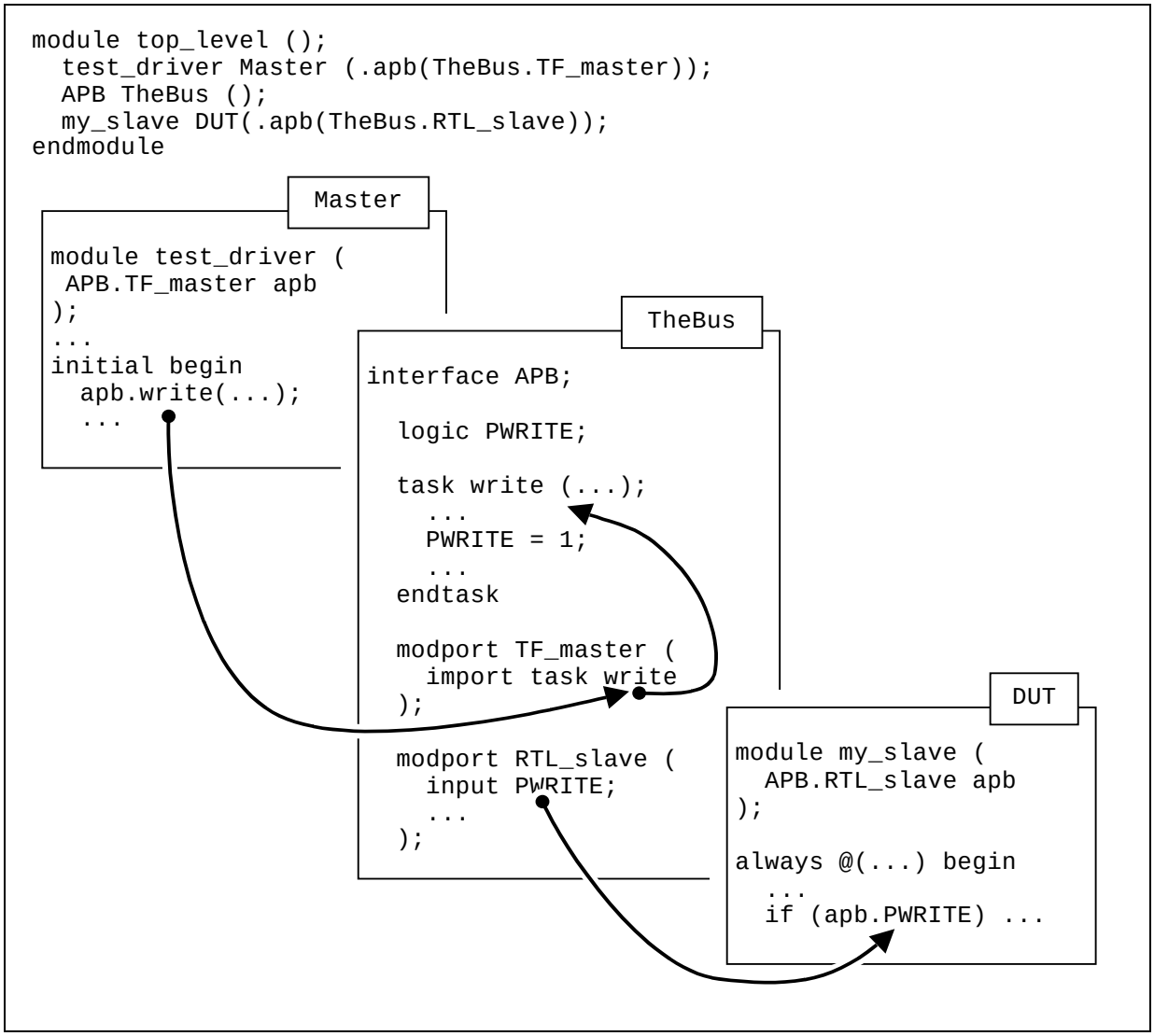

Figure 5: Behavioural master modport

4.6 Implementation challenges

A bus such as APB, having multiplexed readback, requires each of its slave connections to be distinct so that on read cycles the bus itself can assert the appropriate slave's PSEL (selector) signal and steer that slave's readback data to the master's PRDATA (readback data input) signal. This implies that the bus structure itself should incorporate a decoder to generate the

selector signals and a multiplexer to steer the readback data. There is no way to accomplish this directly in a SystemVerilog `interface`.

Instead, we must export some of the bus's functionality to the slave modules. This is comparatively straightforward in simulation, but it is not clear whether our solution will be synthesisable by current-generation tools.

4.6.1 Selecting the active slave

In APB a slave is selected by assertion of its dedicated `PSEL` signal. This signal can readily be created using a decoder on some part of the address, enabled by a common select timing signal from the master. It is clear that this selection functionality could easily be implemented in the slave rather than in the bus, and this is the solution we have adopted. A common select signal from the master is broadcast, along with the address, to all slaves; each slave gates the select signal with the output of an equality comparator, testing an appropriate part of the address for match with the slave's own address constant. This is an entirely reasonable solution both in hardware and in simulation, and it poses no difficulties in SystemVerilog.

4.6.2 Steering the readback data

In APB read cycles, the selected slave drives its read data on to its own dedicated read data signal. A multiplexer or other steering arrangement, switched by the same address bits that are used to choose the selected slave, then drives a copy of that data on to the master's readback data signal. Although it is possible to provide this functionality using a tri-state bus, this solution is unattractive for on-chip implementations and we prefer to model a multiplexer directly. However, this does not appear to be possible in any intuitively satisfying way using SystemVerilog interfaces and modports. It would be necessary to supply a completely separate modport for each possible slave. Such a solution with multiple modports could clearly be synthesisable and directly reflects the expected hardware implementation, but it offers little to users hoping for a smooth migration from a very flexible high-level system model all the way to a final RTL implementation.

4.6.3 A solution using common modports

We have implemented a simulation model of the RTL slaves by having each slave develop its own select signal, as described in section 4.6.1 above. For write cycles this is straightforward. For read cycles we provide a common readback data signal `PRDATA` as a variable in the interface itself, available to slaves through the `RTL_slave` modport. A slave that is the target of a read cycle makes a procedural assignment to this variable through the modport. Slaves that are *not* the target of a given read cycle simply refrain from assigning to this variable. Whilst this approach correctly models the system's behaviour in simulation, we have grave doubts about whether such a modelling style can be expected to be synthesisable.

5 Potential benefits of interfaces and modports

5.1 Smooth migration from transaction-level to RTL modelling

We looked to the interface/modport mechanism to provide us with a modelling discipline in which we could freely mix RTL and behavioural models. In particular we hoped to allow our behavioural models, operating at a transaction level, to interact with RTL models connected to the same interface through different modports. We expected to be able to do this using features of the interface itself to map transactions (in the form of task calls) on to bus cycles, and vice versa. In this way we hoped to show SystemVerilog providing a completely smooth migration path from transaction-based, untimed modelling through to synthesisable RTL.

We have not yet found how to achieve this goal with SystemVerilog in designs using interconnect schemes such as APB. In buses of this type, some interconnect functionality properly belongs in the bus itself rather than in the modules connected to that bus. Although SystemVerilog already provides the means to model such a scheme at any level of abstraction, those different models cannot readily share a common set of boundaries between modules and interface, because of the way functionality must be moved between the connected slaves and the interface itself.

As long as SystemVerilog lacks facilities to parameterise and differentiate the various connected instances of a modport, we are unable to see a satisfying solution to this challenge. We have experimented with some possible schemes for a solution, and these are outlined on the source code available on the web site referenced in the Appendix.

5.2 Robust elaboration-time checking of modport usage

Modports impose a direction on the connections between a module and an interface. Port directions in traditional Verilog have surprisingly little significance, because a module is free to drive its input ports if it so wishes. This freedom is retained for interfaces, because a module can freely access any object in an interface through a port of interface type. Given that this freedom remains available, we hoped that modports could provide more robust and restrictive checking of the usage of their signals; however, this does not appear to be the case. [Accellera 2003] appears silent on this issue. We anticipate that SystemVerilog 3.1a will be much more closely prescriptive in this regard.

6 Conclusions

Our early experiences with SystemVerilog have been encouraging in many ways. Tools available at the time of writing have significant limitations, as might be expected so soon after publication of the standard and when the “industrial-strength” version 3.1a of the standard is still under development. Indeed, EDA vendors have responded much more promptly to the challenge of adding SystemVerilog features than they did to the demands of Verilog-2001.

Our experience is that many new features of SystemVerilog offer immediate, direct and tangible benefit both to RTL designers and to verification engineers. We have only scratched the surface of SystemVerilog’s range of capabilities in this short study, and we look forward to reporting in the near future on experience with assertions, DPI and testbench automation.

Our most important concerns relate to SystemVerilog's provision for a smooth migration path from untimed transaction-based abstract modelling to traditional RTL simulation, without disruption to system architecture as this refinement progresses. We hoped that the interfaces and modports mechanism could provide a means to achieve this smooth transition, and thus to differentiate SystemVerilog strongly from the diverse but effective modelling techniques that already crowd the marketplace. However, we found that limitations of the modport mechanism forced us to make disruptive modifications to module and interface boundaries as the design progressed. We look to other SystemVerilog practitioners, and to Accellera, to show us how this challenge can be addressed.

7 Acknowledgements

The authors are indebted to their employer Doulos Ltd for the facilities and opportunity to prepare this work, and to Synopsys Inc for making beta-test versions of their VCS 7.1 simulator available to us. We also note with thanks the extensive contributions of many experienced members of the EDA community to the development of SystemVerilog.

8 References

[Accellera 2003] Accellera Organization Inc., SystemVerilog 3.1 Language Reference Manual, June 2003.

[Synopsys 2003a] Synopsys Inc., "Using SystemVerilog", chapter 16 of VCS 7.1 User Guide, December 2003.

[Synopsys 2003b] Synopsys Inc., SystemVerilog 3.1 Language Reference Manual annotated with VCS 7.1 implementation details, December 2003.

[ARM 1999] ARM Ltd., AMBA APB specification, in Chapter 5 of document IHI 0011A, 1999.

[Volder 1959] "The CORDIC Trigonometric Computing Technique", IRE Trans. Electron. Comput. EC:330-334, 1959.

[Andraka 1998] R. Andraka, A survey of CORDIC algorithms for FPGA based computers, in Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '98), ACM/SIGDA, ACM, 1998

9 Appendix

Code examples and simulator command scripts for the design and verification example presented in this paper are freely available for download from the authors' employer's website:

`http://www.doulos.com/knowhow/sysverilog/snug04europe`