



design & verification
conference & exhibition

February 22, 2007

Towards a Practical Design Methodology with SystemVerilog Interfaces and Modports

Jonathan Bromley

Senior Consultant

Doulos, Ringwood, UK

jonathan.bromley@doulos.com



Notes

This paper was presented at DVCon 2007 (see www.dvcon.org) where it was judged by delegates to be joint Best Paper. The full text of the paper is available in the DVCon Proceedings, and on the Doulos website www.doulos.com.

The other joint winner was:

FEV's Greatest Bloopers: False Positives in Formal Equivalence

Erik Seligman, Joonyoung Kim

Digital Enterprise Group, Intel Corporation, Hillsboro, OR

Multi-level bus architecture

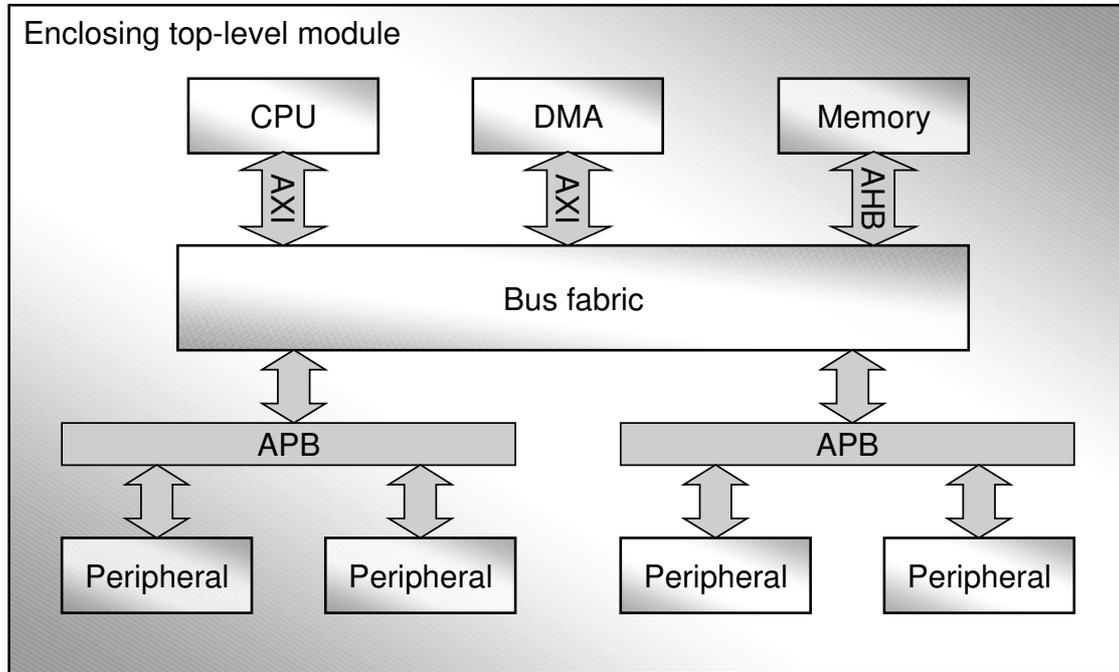
Many of today's FPGA and ASIC designs use interconnect schemes such as AMBA (published by ARM Ltd) or other standards.

Bus fabric

Unlike traditional rack-and-cards systems, an on-chip system is likely to use many different forms of interconnect – our example shows the high-performance AXI and AHB buses co-existing with the much simpler, lower-performance APB. Furthermore, point-to-point connection is usually preferable to shared (multi-drop) interconnect. It is therefore usually necessary to have one or more bus switching and bridge blocks, shown here as a "Bus fabric" module. Such modules are often quite complex, and need extensive customisation to suit the number and type of subsystems in each individual application.

The use of bus fabric modules typically leads to multiple instances of a given interconnect structure at the top level of the design. In our example we have two instances of the AXI bus, and two instances of APB.

Multi-level bus architecture



Notes

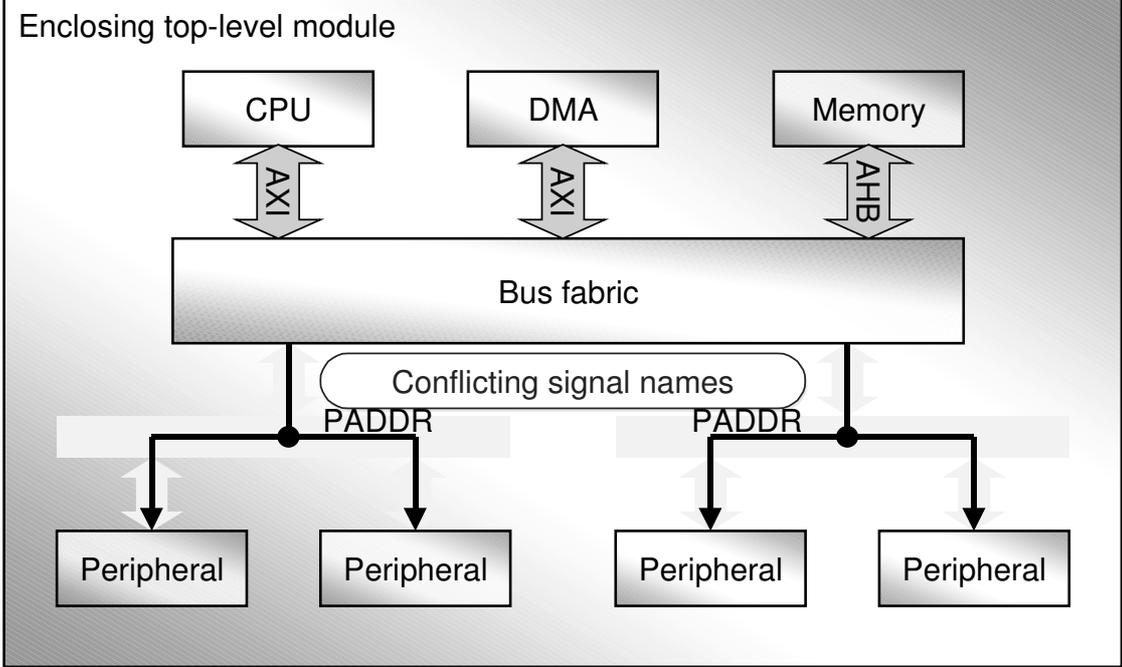
The need for encapsulated interconnect

Because there are two APB buses at the same level of hierarchy, we have a problem when we come to create the wiring. The standard name for the address bus in APB is `PADDR`, but we can't use that name for two separate signals! In a traditional system we would be obliged to invent two different names for the two `PADDR` signals – and, of course, do the same over and over again for all the other signals too.

Perhaps you use scripts, or a graphical design entry tool, to do all this tiresome top-level wiring, in which case the signals will probably get unfathomable names such as `PADDR_1` and `PADDR_2`. Alternatively you may do it manually, which is unpleasantly laborious and error-prone. Neither solution seems ideal.

We could solve this problem elegantly if we had some way to encapsulate the whole of an APB (or any other) interconnect in a single structure. As we will see in the next few pages, SystemVerilog's `interface` construct provides just such an encapsulation.

The need for encapsulated interconnect



- Multi-level bus structures may have many instances of the same set of bus signals with standard names

Notes

Data structures are not enough

First, though, we'll look at an alternative possibility. SystemVerilog also offers the `struct` feature, allowing users to define a data type that contains a collection of distinct data items. Our example shows the definition of a new data type `APB` that can be used as a "cookie cutter" to create new variables, each of which represents the complete set of signals needed for the APB interconnect.

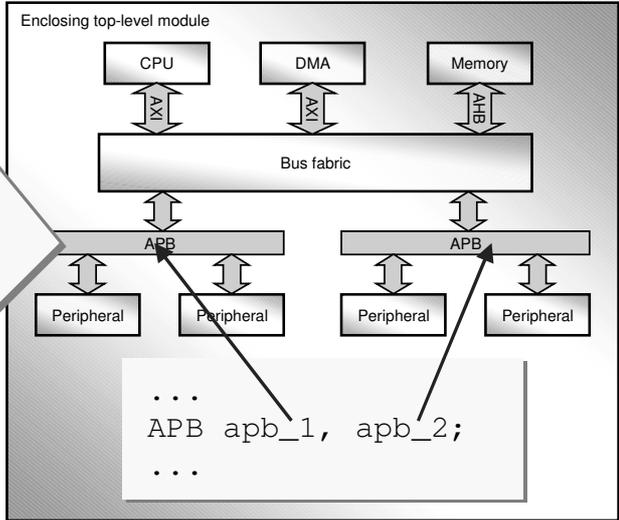
However, if you try to use this mechanism you will find exactly the same difficulties that VHDL users have suffered for many years when trying to use the `RECORD` construct to do such things:

- it is impossible to specify input or output directions individually for the various components of the record, which is very troublesome;
- a module wishing to connect to such a structure must simply connect to all of it – there is no way to distinguish the various different roles, such as master or slave, that different modules may have;
- signals whose scope extends outside the bus itself, such as clock and reset, cannot be conveniently integrated into this arrangement.

These are exactly the problems that SystemVerilog `interfaces` were designed to solve, as we'll see on the next page.

Data structures are not enough

```
typedef struct {
    logic PSEL;
    logic PENABLE;
    logic PWRITE;
    t_APB_a PADDR;
    t_APB_d PWDATA;
    t_APB_d PRDATA;
} APB ;
```



- No direction information
- No distinction of roles (slave, master, ...)
- Hard to integrate global signals (clock, reset, ...)

Notes

Interface captures interconnect

Here we see the simplest way to use an `interface` to capture a structured set of interconnect. The interface is defined in much the same way as a module; indeed, the writer is strongly of the opinion that interfaces and modules are very nearly the same!

Interface contents

The interface contains declarations of nets or (as here) variables that represent the various signals that form the interconnect. Note that we can use the signal's standard data-sheet names, since there will be no conflict with signals of the same name elsewhere in the design.

Ports

The interface can also have ports; here we have used a port to bring in a global clock signal, making it visible by its standard name `PCLK` within the interface itself.

Instantiating the interface

Now that we have defined the interface, we can instantiate it in just the same way as we could instantiate a module. Note how we have wired the same global clock signal `PCLK` to both instances of the interface.

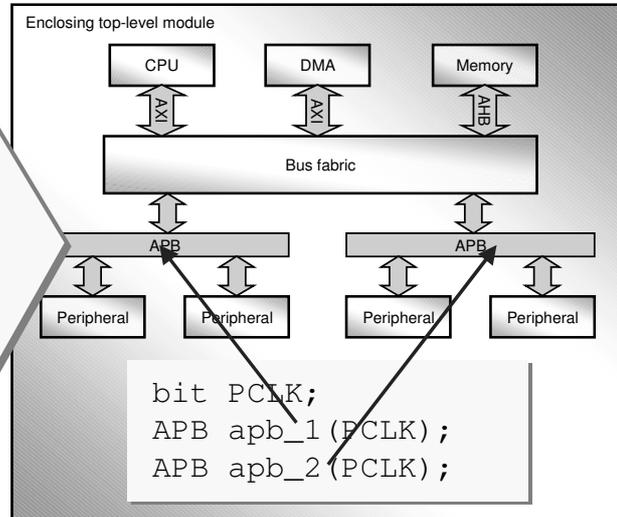
Because we have used interfaces to capture the interconnect, the top-level enclosing module is now much cleaner and easier to understand than it would be if each interface were represented as a collection of separate signals.

Interface captures interconnect

```

interface APB (
    input bit PCLK );
    logic PSEL;
    logic PENABLE;
    logic PWRITE;
    t_APB_a PADDR;
    t_APB_d PWDATA
    t_APB_d PRDATA;
endinterface

```



- Global signals ported into the interface
- Each interface instance provides a new set of bus signals
- Enclosing module is now *much* cleaner

Notes

Porting the interface into a module

We have now created the interconnect and instantiated it in the enclosing module, but how do we connect it to one of the attached modules?

Our example shows what must surely be the simplest imaginable APB peripheral: a parallel output register that captures and stores whatever data is written to it. Naturally, this output register has an output port making the register value available to the external world. But it needs only one more port – the APB interface. We "connect" this port to the *interface instance* that carries the signals we need.

As you can see from the code for module `APB_outReg`, we can reach through the interface port into the connected interface instance by using dotted names. This makes the connected module a little more complicated, and for a big complex module it might be better to create "alias" signals inside the module so that the dotted name can be referenced just once, and a simpler name can be used internally. Here's the code for module `APB_outReg`, rewritten to use internal signals:

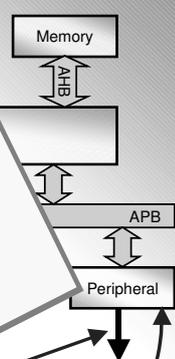
```
module APB_outReg (APB bus, output t_APB_d Q);
    //
    // Create internal signals to simplify the
    // remainder of the module
    t_APB_d PWDATA;
    logic PSEL, PENABLE, PWRITE, PCLK;
    assign PWDATA = bus.PWDATA;
    assign PSEL = bus.PSEL;
    assign PENABLE = bus.PENABLE;
    assign PWRITE = bus.PWRITE;
    //
    // Simplified internal logic
    always @(posedge PCLK)
        if (PSEL & PENABLE & PWRITE)
            Q <= PWDATA;
endmodule
```

Porting the interface into a module

```

module APB_outReg (
    APB bus,
    output t_APB_d Q );
    always @(posedge bus.PCLK)
        if (bus.PSEL & bus.PENABLE & bus.PWRITE)
            Q <= bus.PWDATA;
endmodule

```



```

t_APB_d Out;
APB_outReg lamp_reg (
    .bus(apb_2), .Q(Out) );

```

- Port of interface type makes all interface contents visible
- Access using `interface_port_name.interface_contents`

Notes

Modports reflect client roles

We have used ports on an interface to solve the problem of integrating global signals with the interconnect, but we also need to deal with two other issues: the distinction of various client rôles, and establishing the dataflow direction for each signal individually. Using the `modport` construct elegantly solves both these problems.

In any interface, we can declare one or more `modport` specifying a view of the interface that one client rôle should see. The `modport` details the names of interface signals that are available to the client, and the directions of each signal *from the point of view of the connected module*. In our example, we have made all interface signals available to both the master and slave clients. However, if we prefer we can also make some signals invisible to certain clients, simply by omitting those signals from the `modport` list.

A client module can then choose to connect to the appropriate `modport`, rather than the whole interface.

Modports reflect client roles

```
interface APB ( input bit PCLK );
```

```
  logic    PSEL, PENABLE, PWRITE;
```

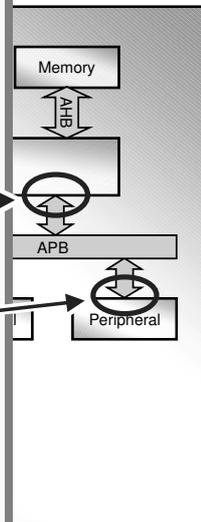
```
  t_APB_a  PADDR;
```

```
  t_APB_d  PWDATA, PRDATA;
```

```
  modport Master (
    input    PCLK,
    output  PSEL, PENABLE, PWRITE, PADDR, PWDATA,
    input    PRDATA );
```

```
  modport Slave (
    input    PCLK,
    input    PSEL, PENABLE, PWRITE, PADDR, PWDATA,
    output  PRDATA );
```

```
endinterface
```



- Directions are relative to the connected (client) module
- Modports also can restrict visibility
(some roles don't need to see all contents of interface)

Notes

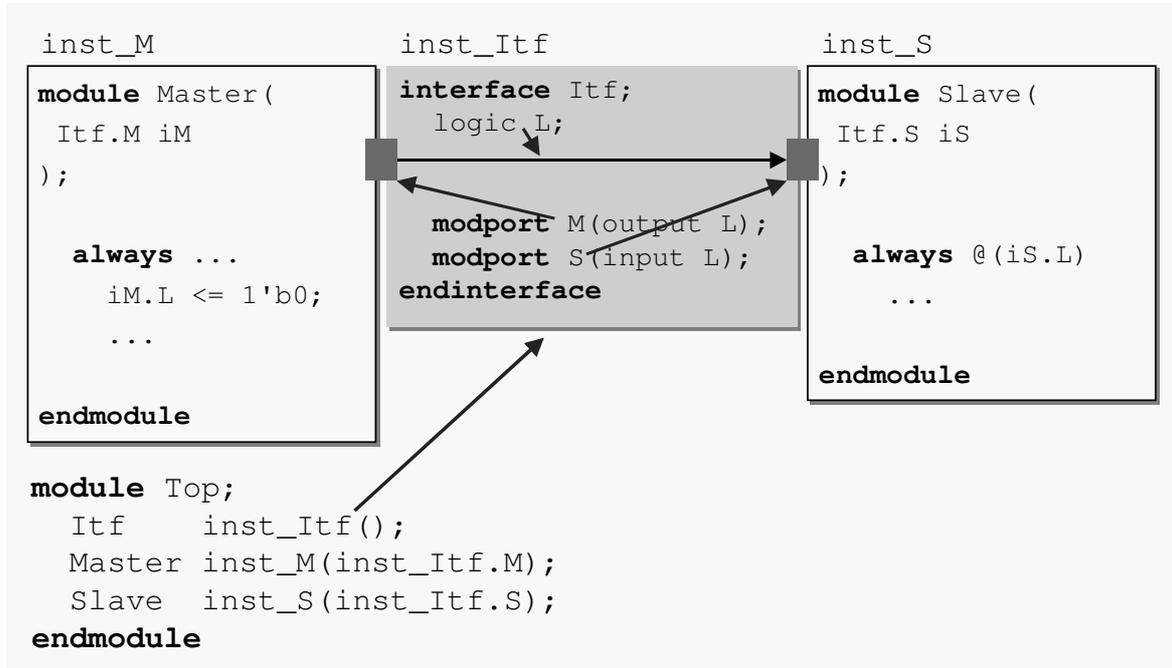
Synthesis example

This example shows a trivial interface (containing only one signal!) with two modports, one for the driver of that signal and one for the receiver. You can see how each connected module specifies a port of the form

interface_name.modport_name

and the module instance likewise specifies both the interface instance and the modport to which it wishes to connect. Later we will see that it is not always necessary to specify the modport in both places (module definition and module instance), although it does provide stronger checking if you provide both.

Synthesis example



Notes

Synthesis results

Interfaces and modports are synthesisable!

This slide shows the netlist that would result from synthesis of the previous page's example. The details might differ slightly between different synthesis tools, but the general principles are exactly as shown here.

First, the synthesis tool flattens or "explodes" the interface instance so that all its contents are declared directly in the enclosing module. Naturally, that will entail some renaming of the interface contents – especially if there is more than one instance of the same interface.

Next, the synthesis tool must rewrite each connected module's port list so that it expects to connect to a set of distinct signals, rather than an interface or modport. All the connected module instantiations must then be rewritten to match.

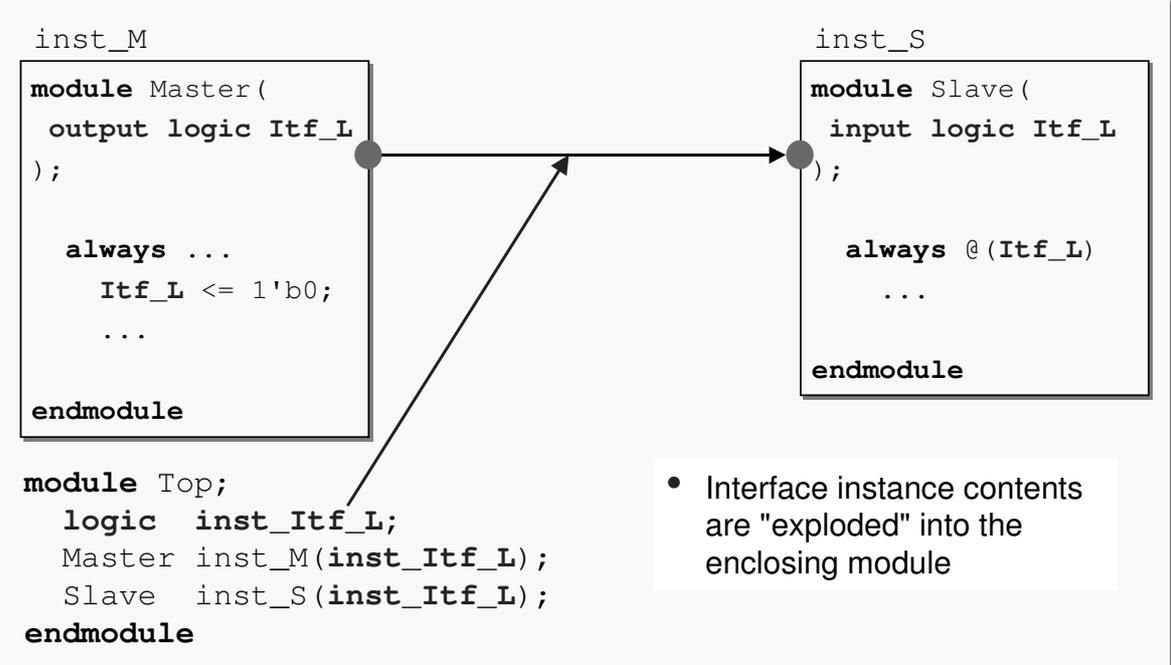
Finally, the synthesis tool patches-up all references to interface signals within the connected modules so that instead they reference the newly created ports.

At the time of writing (February 2007) not all synthesis tools yet support this, but support is growing fast and already there are examples of both FPGA and ASIC synthesis tools that have the necessary SystemVerilog capability.

It's not all good news...

It's clear that straightforward use of interfaces for point-to-point interconnection is convenient and practical. However, on the next few pages we will investigate some common problems that cannot be solved quite so easily.

Synthesis results



Notes

Multiple drivers on a bus

Like many interconnect schemes, APB is designed to link one master to multiple slaves. For signals going from master to slave, this is straightforward; the master drives a signal, and multiple slaves receive it. For signals going the other way, though, things are more troublesome. In our APB example only one signal is driven by slaves: the read data signal `PRDATA`. For any bus read cycle, just one slave is selected (typically by address decoding) and only that slave should drive `PRDATA`. How can we arrange this if we use an interface?

Three-state drivers

All slaves' `PRDATA` outputs are connected to a common net in the interface. Only the selected slave drives the `PRDATA` net. Other, deselected slaves place the high-impedance value `'bZ` on the net.

This approach works correctly both for synthesis and for simulation, but it represents a hardware modelling style that is completely inappropriate for modern on-chip architectures. Some synthesis tools can automatically re-map this structure to a multiplexer, but this is not guaranteed and we cannot recommend this style for routine use.

Multiplexer or AND-OR logic

Alternatively, each slave can have its own `PRDATA` signal. These various signals can then be multiplexed on to the master's common `PRDATA`, using the same address decode that was used to select the slaves. Alternatively, deselected slaves can drive their outputs to zero, and all the outputs can then be ORed together to create the correct readback value.

This approach is perfect for hardware, but requires that we create *distinct* `PRDATA` signals for each individual slave. As we will see later, this is difficult to arrange using interfaces and modports.

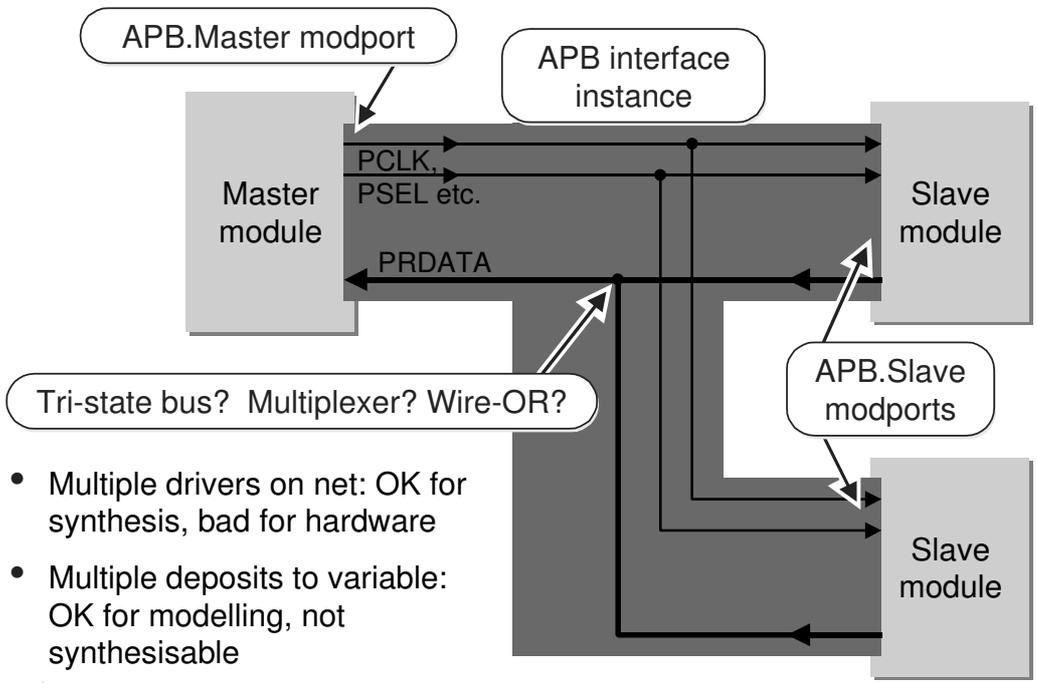
Selective deposit to a variable

For simulation, the most natural and most elegant approach is for the selected slave to write to a variable `PRDATA` in the interface, and for all deselected slaves simply to refrain from writing to that variable. Verilog's "last-write-wins" behaviour then ensures that the required value appears on `PRDATA`.

This approach works well in simulation, but is not synthesisable because the synthesis tool cannot resolve the effect of multiple processes writing to the same variable. Consequently it can't be used for real hardware design.

We believe this to be the most significant difficulty relating to the use of interfaces in synthesisable design.

Multiple drivers on a bus



- Multiple drivers on net: OK for synthesis, bad for hardware
- Multiple deposits to variable: OK for modelling, not synthesisable

Notes

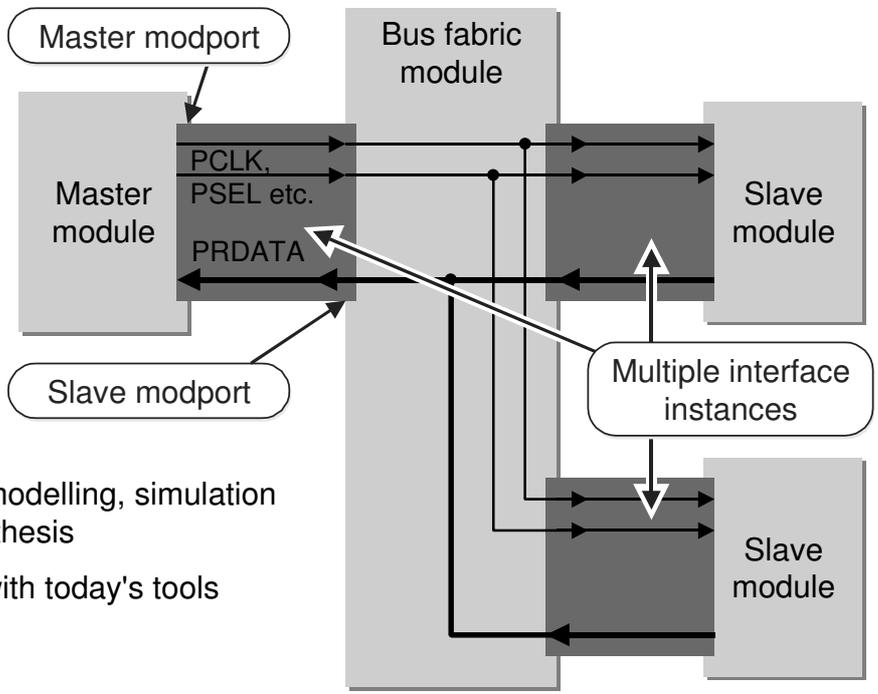
Bus fabric module and simple interfaces

As a pragmatic solution to the difficulties noted on the previous page, we can restrict our use of interfaces to only point-to-point interconnect. In such a scenario, we would need a bus fabric or bus matrix module – coded as a conventional Verilog `module` – to capture the selection and multiplexing logic required to steer the various signals appropriately. The bus matrix module would obviously need a port (of interface type) for each connected master and slave device.

This approach works well with current tools. It has all the benefits, already described, of simplifying the enclosing module because there is only one interface instance for each connected client, rather than a multitude of individual signals all needing distinct names. However, it does not leverage the interface mechanism as effectively as we might wish.

A particular disadvantage with this approach is that the bus matrix module, which will of course need to be different for different applications, cannot easily be parameterised. Although Verilog modules can be given parameters, the number of ports in their port list is fixed. Consequently, it is likely that the bus fabric module will be created either by hand or – more likely – by use of a specialised bus fabric IP customisation tool that generates the required synthesisable Verilog code as one of its outputs.

Bus fabric module and simple interfaces



- OK for modelling, simulation and synthesis
- Works with today's tools

Notes

Interface architecture challenges

If we wish to exploit the power of interfaces more fully, we must solve the problem of multiple client modports and multiple drivers on interface signals.

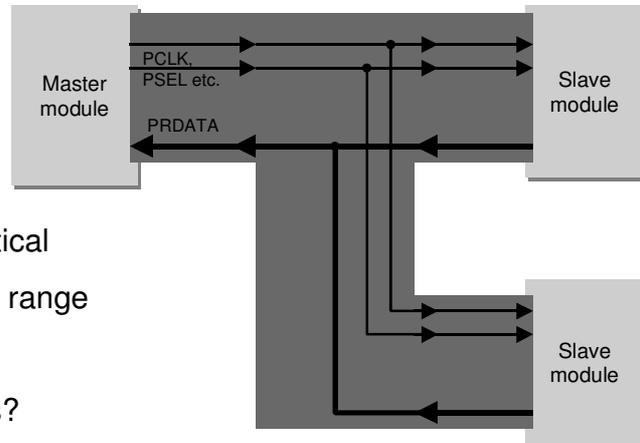
Each slave must connect to a distinct modport, because each slave has a distinct identity and occupies a specific address range. However, this conflicts with the obvious need for every slave interface to look identical; they should all present the same set of APB slave signals. Clearly, this gives us a problem of naming.

Furthermore, the bus fabric interface must now accept some responsibility for address decoding. We must choose whether this decoding is done centrally, by the interface itself, or by each individual slave performing a matching operation on the address value.

Address ranges, number of masters and slaves, and other features of the whole bus-based system should ideally be parameterisable.

Finally we must solve the problem of multiple drivers on a signal, as described earlier – although, as we will see, this problem is very closely related to the problem of modport signal naming.

Interface architecture challenges



- Slave modports cannot be identical
 - each slave has an address range
- Who chooses a slave's address?
 - each slave, or centrally in the bus fabric?
- Each slave can drive readback signals
 - multiplexer schemes require a distinct modport per slave

Notes

Modport expressions

The SystemVerilog language standard (IEEE Std.1800-2005) already provides a solution to many of the difficulties we have described, with its *modport expression* construct. This diagram shows a very simple use of modport expressions.

The interface contains a vector signal $v[1:0]$. We imagine that the master module needs to see the full vector, but each slave should drive only one bit of the vector. This can be done by passing a parameter – the bit number – into each slave; but this is not ideal, because it means that the slaves need to be aware of the internal structure of the interface. Ideally, each slave should see only the single-bit output that it requires; and, of course, it would be best if both slaves see exactly the same set of connections.

Modport expressions enable all this by renaming a signal as it goes through the modport. The modport expression

```
output .L(V[0])
```

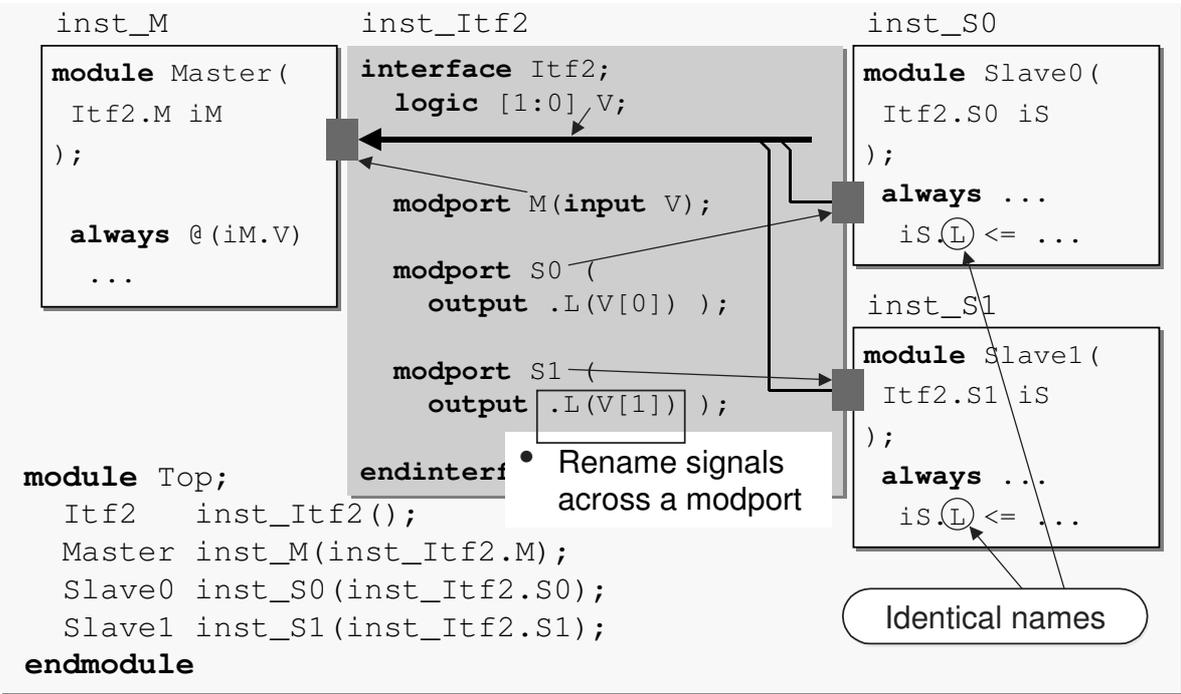
says that the module connected to the modport should see a signal called L , but that signal should in fact be implemented by signal $v[0]$ inside the interface itself.

Now we have two different modports, but each of them presents exactly the same appearance to its connected module.

But there is some bad news:

No current SystemVerilog tools support modport expressions!

Modport expressions



Notes

Specialised modports

Here's the idea from the previous page, worked through in a more practical example. Let's suppose that we want to connect a UART device to our APB system. The UART device has already been designed to expect an APB interface. Note that the module definition for `APB_UART` doesn't specify any modport in its port list, but we *do* specify the modport when we instantiate it. In this way we don't need to make the module unnecessarily specialised. The module instance is specialised (for address range, etc) by being connected to a suitable modport on the `APB_sys` interface instance.

Likewise, our `APB_UART` module expects to see the standard signal name `PSEL` for its bus select signal, but we wish to specify a particular decoded signal `UART_PSEL` within the interface. A modport expression provides this renaming, and also provides the UART with a much narrower (4-bit) part of the address, since it only occupies a 16-word address range.

Decoding logic in the interface

Interfaces can contain `always` blocks and continuous `assign` statements, allowing them to contain significant functionality. Here we have used a continuous `assign` statement, together with SystemVerilog's convenient new wild-equality test `==?`, to create an address decoder.

Note

Currently, interfaces are not permitted to contain module instances; but this is likely to change in future revisions of the SystemVerilog standard, making it even easier to incorporate non-trivial functionality inside an interface.

Specialised modports

```
interface APB_sys (
  input bit PCLK );
```

```
  logic PSEL;
  t_APB_a PADDR;
```

```
  ...
```

```
  wire UART_PSEL = PSEL & (PADDR ==? 'hACOFFEE?);
```

address decoding built into bus fabric

```
  modport mp_M ( input PCLK, output PSEL, PADDR, ... );
```

```
  modport mp_UART ( input PCLK,
    input .PSEL(UART_PSEL),
    .PADDR(PADDR[3:0]), ... );
```

specialised modport

```
  ...
```

```
module Top;
```

```
  bit clk;
```

```
  APB_sys apb ( clk );
```

```
  Master inst_M( apb.mp_M, ... );
```

```
  APB_UART inst_U( apb.mp_UART, ... );
```

```
  ...
```

instantiation chooses modport

```
module APB_UART (
```

```
  APB_sys bus,
```

```
  ... );
```

```
  always ...
```

```
    if (bus.PSEL)
```

```
      ...
```

modport not specified here

Notes

Modport expressions and generate

The previous page showed modport expressions being used to create a specialised modport that nevertheless has the standard signal names from the point of view of a connected module. Using the `generate` construct we can go even further, and allow extensive parameterisation of the interface; modports can be created in a `generate` construct.

We have kept the example very simple, but you can easily see how this idea can be combined with the use of parameters to make very flexible interfaces that have a configurable number of modports.

There is a little difficulty about the names of these generated modports, because the `generate` construct creates a new scope. The slide shows how the generated modports' names are determined. On the next page we will see how to make use of these strangely-named modports.

But there is some bad news:

No current SystemVerilog tools support modports inside a `generate` construct!

Modport expressions and *generate*

- Avoid repeated modport declarations

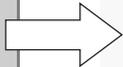
```
interface Itf2;
  logic [1:0] V;

  modport M(input V);

  modport S0 (
    output .L(V[0]));

  modport S1 (
    output .L(V[1]));

endinterface
```



```
interface Itf2_gen;
  logic [1:0] V;

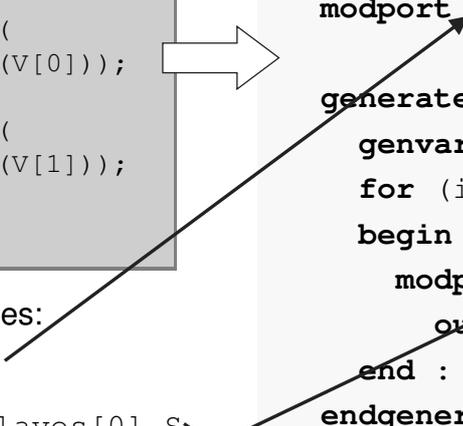
  modport M(input V);

  generate
    genvar i;
    for (i=0; i<2; i=i+1)
      begin : Slaves
        modport S (
          output .L(V[i]) );
        end : Slaves
      endgenerate

endinterface
```

- Modport names:

- *inst.M*
- *inst.Slaves[0].S*
- *inst.Slaves[1].S*



Notes

Generating the attached modules

The modules that we will use to connect to the generated modports use a convenient trick: by specifying a port of type `interface.modport_name` the module is saying "I can connect to a modport called `modport_name`, no matter what interface it is in".

We can use another `generate` loop to construct a collection of connected modules, with the `genvar` iterator participating in the selection of modport for each module. Once again, this arrangement could easily be parameterised.

Generating the attached modules

from previous slide

```
module Top;
  Itf2_gen ← inst_Itf2_gen();
  Master  inst_M (inst_Itf2_gen.M);
  genvar i;
  for (i=0; i<2; i=i+1) begin : Slave_Insts
    Slave inst_S (inst_Itf2_gen.Slaves[i].S);
  end : Slave_Insts
endmodule : Top
```

```
module Slave (interface.S iS);
  always ....
    iS.L <= ...;
endmodule : Slave
```

• Modport names:

- *inst.M*
- *inst.Slaves[0].S*
- *inst.Slaves[1].S*

Notes

Address decoding in attached module

The previous few pages have suggested that modports can be specialised by logic in the interface itself.

It is also possible to perform address decoding, and other specialisations, inside the connected module. Of course, the module then needs to reflect the results of this decoding back into the interface, where it could be used to control a multiplexer for multi-source readback signals such as `PRDATA`. This is an attractive idea because it suggests that we could make a completely generic bus fabric interface, and specialise it by connecting an appropriate set of parameterised modules.

The example opposite works as it stands, but does not really do all we want. We would prefer to have a completely un-specialised – generic – interface; but how would that interface gain the correct number and kind of modports? It is easy to connect any number of modules to the same modport – we discuss that in more detail on the next page – but then all the modules see exactly the same set of signals, which doesn't help us with the problem of multi-source signals.

The SystemVerilog language does not provide a clear set of answers to these problems, and the author hopes that some enhancements could be made in the future to provide modports that in some way can be configured by the modules that connect to them.

Meanwhile, we must add a parameter to the interface to specify how many modports it must have. Clearly this mechanism will not work until we have tool support for modport expressions and modports in a generate construct.

Address decoding in attached module

```
module APB_slave (  
    #(My_Adrs = 24'hDEAD??)  
    ( APB.slave_mp bus, ... );  
  
    assign bus.active = (bus.PADDR ==? My_Adrs);
```

address range set by parameter on module

extra connection to interface

decoding logic

```
interface APB #(N_slaves = 1) ( ... );  
  
    logic [N_slaves-1:0] decodes; which slave is active?  
    ...  
    genvar i;  
    for (i=0; i<N_slaves; i++) begin : gen_slave_mp  
        modport slave_mp ( output .active(decodes[i]), ... );  
    end
```

- Requires modport expressions!

Notes

Singleton modports

Because of the problems of address decode specialisation and multi-source signals, the approaches presented in the previous few pages are easily broken if a user inadvertently connects more than one module to the same modport. The author believes that SystemVerilog could benefit from an enhancement allowing a modport to be specified as a singleton, so that at most one module can connect to it.

The author has had interesting discussions with a colleague who pointed out that in standard Verilog you would naturally expect to be able to connect any number of modules to a single wire, and consequently we should expect the same of modports. The author's opinion is that modports offer a much more structured connection scheme than does a wire, and therefore the singleton property is valuable. You, gentle reader, must make up your own mind!

In the printed paper, several possible workarounds are discussed; none is totally satisfactory. During the DVCon conference, Don Mills offered another approach based on the fact that SystemVerilog permits only one continuous assignment to any variable; the author plans to investigate this in more detail and publish the results in due course on the Doulos website.

Singleton modports

- Modports are promiscuous
 - Any number of clients can hook to the same modport instance
 - Bad fit with typical modern bus structures
 - Disastrous for methodologies presented here
- Solutions discussed in the printed paper, none entirely satisfactory:
 - use of `uwire`
 - very limited tool support at present
 - exported function
 - not supported by synthesis tools
 - client writes to an interface variable
 - singleton enforced by synthesis, not by simulation

Notes

Conclusions

Interfaces are valuable both for modelling and for synthesis using today's tools. The approach suggested on page 20 (bus fabric module with point-to-point interfaces) offers a good way to exploit many of their advantages without stretching the capabilities of current tools.

Meanwhile, the author hopes that this paper will provide food for thought both for users and for tool vendors; in particular he looks forward to the availability of modport expressions and generated modports in practice.

The lack of support for singleton modports in the SystemVerilog language restricts users' ability to build truly robust configurable designs using interfaces. There is further work to do in this area, and such work may perhaps lead to future enhancements of the language.

Questions? If you have further questions on this paper, please feel free to email the author at the address on the front page; he will always endeavour to respond promptly.

Acknowledgements

The author would like to thank his colleagues for their consistently insightful support, and also the following people for their special contributions:

- Bruce Mathewson of ARM Ltd who brought the "bus fabric module" technique to my attention;
- Cliff Cummings of Sunburst Design Inc., for his careful review of the original paper and his enthusiastic encouragement;
- Don Mills of Microchip Technology, Inc for his useful insight concerning singleton modports.

Conclusions

- Interfaces are synthesizable!
- Interfaces are useful today for point-to-point and multi-drop buses
- Lack of tool support for modport expressions is a disappointment
 - They offer great opportunities for bus fabric modelling
- Lack of language support for singleton modports is unfortunate

Questions?

Notes

