Design Simulation Model

Flow Integration Guide

ARM®

ARM DUI 0219A

Design Simulation Model
Flow Integration Guide

Release Information

The table below shows the release state and change history of this document.

Change history

Date	Issue	Change
17 April 2003	A	First release

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
Design Simulation Model Flow Integration Guide

List of Figures
Design Simulation Model Flow Integration Guide

Preface

This preface introduces the *Design Simulation Model* (DSM). It contains the following sections:

About this manual

This User Guide provides information on the *ARM Design Simulation Models* (DSMs) covering the following subjects:

- the use of DSMs
- their features
- simulator interfacing and implications
- timing issues
- limitations of DSMs when compared to the use of native HDL code.

Intended audience

This book is written for experienced hardware/software engineers and chip designers who might have experience of ARM products, but who have experience of Verilog or VHDL, and who want to integrate an ARM DSM into their design and simulation flow.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for a description of the DSM, the main features, and the contents of a DSM package.

Chapter 2 ***Interfacing and DSM behavioral differences***

Read this chapter for information on simulator interfacing, the Model Manager, event and interface semantics, the differences and implications arising from the use of compiled models verses RTL, and the programmer's model.

Chapter 3 ***Timing Issues***

Read this chapter for information on DSM timing issues. It describes SDF annotation, templates, and SDFremap. It also describes pin-to-pin timing, interconnect delays, negative timing checks, static timing analysis, and limitations of the timing model.

Chapter 4 ***Limitations***

Read this chapter for information on DSM limitations. It describes the restart, save and restore procedures, scan chain modeling, caches, and zero delay simulation.

Typographical conventions

The following typographical conventions are used in this book:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for ARM models.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

ARM publications

This section contains useful information for engineers using the DSM Flow Integration Guide.

- *ARM Design Signoff Models: Timing Annotation* (EDA QPRO 0001 A).

Other publications

This section lists relevant documents by third parties:

- IEEE std. 1076.4 - 1995 IEEE *Standard for VITAL Application-Specific Integrated Circuit* (ASIC) *Modeling Specification* (also known as VITAL-95)
- IEEE std. 1364 - 1995 IEEE *Standard Hardware Description Language based on the Verilog Hardware Description Language*

- OVI SDF2.1 *Open Verilog International Standard Delay Format Specification Version 2.1* (February, July 1994)
- SDFremap Manpage (UNIX command: man sdfremap).

Feedback

ARM Limited welcomes feedback on both the DSMs and the documentation.

Feedback on the DSMs

If you have any problems with the DSMs, contact your supplier. To help us provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type, and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any comments on this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter describes the DSMs. It contains the following sections:

1.1 About DSMs

DSMs are back-annotation-capable (timing accurate) simulation models that can be included within a range of target HDL simulators. Each DSM is specific to a simulator and host platform. The DSM fully matches the architecture and functionality of the ARM core design. ARM ensures this by certifying the DSM against the entire *Architecture Validation Suite* (AVS) and *Device Validation Suite* (DVS) associated with that ARM core.

Being derived directly from the ARM core RTL design DSMs are able to function with a wide-range of industry-standard Verilog and VHDL simulators, and can accept timing data through the SDF annotation facility on the simulator. DSM execution speeds are in the range of 5 - 500 cycles per second, depending on the simulator interface efficiency and the complexity of the design in which it is instantiated.

The DSM consists of two parts:

- A functional core block.
- A Verilog or VHDL wrapper, which includes the timing shell.

  The wrapper uses the foreign language interface of the host simulator to instantiate the functional model.

  The DSM is generally derived from the RTL source of the ARM design using a compiler such as the *Verilog Model Compiler* (VMC) from Synopsys. This is augmented with some extra functionality, such as instruction disassembly, added by ARM.

  The DSM interfaces to the wrapper using technology developed by ARM to enable a single compiled model to function with a variety of logic simulators. The technology developed by ARM also facilitates the addition of a timing shell.

The DSMs typically include behavioral debug facilities, such as an instruction disassembler that are difficult to provide in an RTL model. When you use compiled models it enables distribution of models without compromising the intellectual property that they embody.

Note

For synthesizable cores the DSM is a pre-implementation model and not a sign-off model.

1.1.1 Potential simulation inaccuracies

The compiled nature of these models, and the use of a pin-to-pin timing wrapper introduce several differences in the way these models are used and behave, compared to RTL code used in similar circumstances. Some of these differences are subtle, such as

changes to the order in which simultaneous events are processed by a model, and some of them are more obvious, such as the requirement to characterize the core as a single block for timing purposes.

1.1.2 Features of ARM DSMs

The main features of ARM DSMs are:

Full device functionality

The DSM fully matches the architecture and functionality of the ARM core design.

Phase accuracy

You can expect the DSM to exhibit the same intra-cycle timing as your chosen ARM core. The DVS for the ARM core provides much of the certification of phase accuracy.

Register visibility

The DSM provides debug visibility of the registers within the core. The register set of the core for all modes represented in the architecture is visible in a special layer of VHDL or Verilog hierarchy inside the DSM. These registers are available for tracing in your simulation.

Cache and memory size configuration

You can configure the size of the cache, or TCM, for each particular DSM instance, where applicable.

Timing and back-annotation

ARM supports back-annotation from SDF using either VITAL 95 or Verilog SDF standards.

Note

The SDF can require post-processing using tools included with the DSM.

Disassembler

Some DSMs also provide a built-in disassembler. The availability of the disassembler varies from core to core and depends on the availability of a suitable execution tracer built into the RTL of the core from which the DSM is derived.

1.2 DSM package contents

Each DSM contains the following components:

- Template timing files, `.sdft`.
- Test model that enables ARM *Condensed Reference Format* (CRF) production test vectors to be run.
- A set of CRF test vectors for verifying the model function, `.crf`, `.ctrm`.
- Documentation.
- Utility to ensure correct back-annotation of SDF to timing shells, sdfremap.
- Model Manager, one or more shared libraries, `.so` or `.sl` file.
- The DSM itself comprising of:
  - a compiled core (a shared library) `.so` or `sl` file
  - one or more compiled SWIFT or *Open Model Interface* (OMI) modules generated using a third party compiler, when applicable
  - an HDL (Verilog or VHDL) wrapper, `.v` or `.vhd`
  - any additional files specifically for use with that simulator.

Chapter 2
Interfacing and DSM behavioral differences

This chapter describes using a DSM. It contains the following sections:

2.1 Simulator interfacing

VHDL and Verilog simulators provide an interface through which so-called foreign code can be included in a simulation. For Verilog simulators, this is usually the *Programming Language Interface* (PLI). VHDL simulators use a simulator-specific interface, but they generally provide similar functionality. These interfaces generally enable control and interaction with the simulator on a low level. If you use them appropriately, you can enable foreign code to mimic a component written in HDL. ARM DSMs use this mechanism to enable a compiled model to behave as though it was a component of an HDL/RTL simulation.

The following sections describe simulator interfacing in more detail:

- *Model to simulator linkage*
- *DSM event and interface semantics* on page 2-4.

2.1.1 Model to simulator linkage

The core of an ARM DSM, while specific to the host platform for which it is compiled (Linux or Solaris, for example), is simulator and language-independent. The model interfaces to the simulator through the use of a simulator-specific Model Manager supplied with the DSM, which handles the event transactions between the model and simulator. The simulator invokes Model Manager through an HDL wrapper that is also supplied. The HDL wrapper presents the outside module/entity (the connections to the core, as described in the appropriate TRM) for that core into the logic simulator.

The Model Manager and the DSM are distributed as *Operating System* (OS) specific shared-object files. The HDL simulator loads the Model Manager and the Model Manager loads the DSMs in a system, see Figure 2-1 on page 2-3.

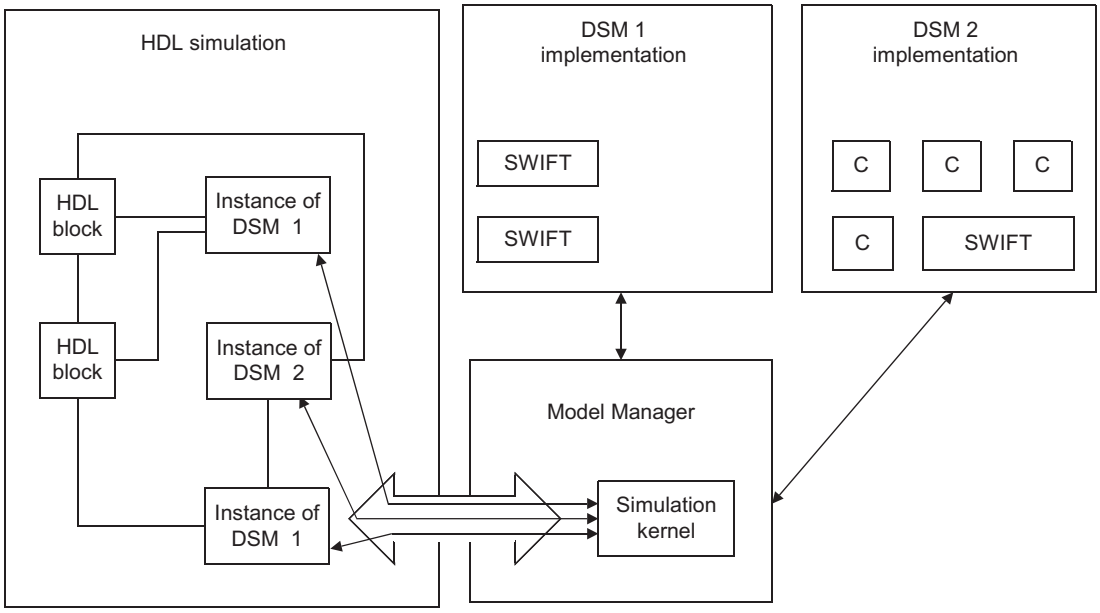

Figure 2-1 Simulation structure

VMC-compiled components are included in a simulation through the use of a Synopsys interface standard called SWIFT that allows the simulators to integrate the VMC and other model types. ARM Model Managers do not use the SWIFT interface of the simulator. They contain their own SWIFT interface from the one provided by the host simulator.Do not use the SWIFT interface in the simulator to call the DSM SWIFT object. No special action is required to register the VMC model with the host simulator, although the environment must be correctly set in order for the ARM model manager to locate these models.

The simulator never deals with the DSMs directly because the Model Manager handles the DSM shared objects. A single Model Manager manages all DSMs in a simulation, even if there are several different DSMs, and presents the personality of each one to the simulator. From the point of view of the simulator, the Model Manager is the foreign model.

You have to register the Model Manager with the host simulator as foreign compiled code. The way you do this varies from simulator to simulator. The simulator is invoked with special switches, or a configuration file is used. Read the installation notes provided with the DSMs for detailed instructions. More information is provided in the user documentation for your simulator.

Note

Because a single Model Manager is used to interface to multiple DSMs, it is not usually necessary to take extra steps to register the Model Manager multiple times with the host simulator. If your simulation involves compiled code from another vendor, there can be additional complexity involved in setting up and using the simulator so that they co-exist.

For more information, see the appropriate Model Manager application note in the `DOCS` directory of the DSM release and also the simulator documentation of the vendor.

2.1.2 DSM event and interface semantics

While DSMs support back-annotatable-timing, any delay on a signal driven out from the compiled core is actually applied at the timing shell by delaying an event that the compiled core has already driven. The core itself is constrained to being strictly zero-delay by the interface semantics between the Model Manager and host simulator.

Input events (that is, changes to the value of an input signal) are registered with the Model Manager by the host simulator. The DSM is essentially a single behavioral process. When invoked it calculates any changes to the value of its outputs on the basis of its input values.

The Model Manager is activated by the host simulator, however, the simulator has no control until the new outputs from the DSM are driven back into the simulation. The DSM is a reactive model that drives new outputs immediately when presented with new inputs. Driven outputs, and incoming input events, can be delayed by the timing shell, which operates within the domain of the host simulator.

HDL simulators use various mechanisms to simulate the concurrent nature of hardware. In certain circumstances this concurrency can give rise to a so called race condition, where the order in which simultaneous events are evaluated can have an effect on the outcome of a simulation.

Race conditions are removed from HDL simulations through the use of specified event-queueing semantics. The behavior of the simulator with respect to handling events that nominally happen at the same simulation time is specified as part of the hardware description language (either VHDL or Verilog). VHDL, for example, deals with any potential problems through the use of delta-sweeping, where any evaluation of logic occurs in successive simulation deltas. Nodes in the netlist are only permitted to change between deltas, to ensure consistent behavior. Verilog does not use this precise mechanism, but its effects can be emulated through the use of so-called nonblocking assignments.

2.2 Differences arising from use of compiled models

When you use compiled models you can encounter some problems that do not occur when using native HDL code. These problems relate to the additional complexities of using a simulation that includes compiled foreign code, in addition to subtle issues regarding event semantics across the DSM/HDL boundary.

The differences are detailed in the following subsections:

- *Model to simulator linkage problems*
- *DSM event and interface semantics problems* on page 2-6
- *Support implications of compiled models* on page 2-7.

2.2.1 Model to simulator linkage problems

One problem that you might encounter with multiple sources of compiled code is that they might not co-exist successfully and so interfere with the operation of other compiled code. There are normally two ways in which this can happen:

- A compiled object can function correctly by itself, but interfere with the use of another compiled object.

  This is typically something that Verilog PLI objects are more prone to, because the PLI normally does not enforce as much contextual separation as typical VHDL interfaces. The ARM Model Managers are written in such a way that they set the simulator to a known state when they are invoked, but not all PLI models are as rigorous in their approach. Problems that arise from this issue tend to manifest themselves as timescale oddities.

  This kind of problem can also occur if the DSM contains a VMC-compiled component when other VMC components are present in the simulation through the native SWIFT interface of the simulator. In particular, the ARM-supplied VMC component is inoperable if instantiated both through a DSM and directly in the host simulator.
- A bug in one compiled object can corrupt the stack or heap in another, causing erratic behavior or crashes.

  This is normally difficult to diagnose and debug. This is because in all currently supported simulators, the simulator, Model Manager, DSM, any VMC models they include and any other foreign code, all operate in the same address space. A bug in any one can cause corruption in any of the others. It is also possible, and likely, that because of the way the heap is managed, problems like this can go undetected, until you add more code to the simulation.

  If a simulation including a DSM and other foreign code seems to be functioning incorrectly, then you should attempt the simulation without the foreign code present, if possible. This helps to determine if one of the foreign objects is responsible for the unusual behavior. If problems persist, then contact your DSM provider for technical support.

2.2.2 DSM event and interface semantics problems

While the processing of events inside a DSM is governed by similar semantics to the host simulator outside, you might find that problems can arise at the level of the foreign language interface. Simulators vary in the extent to which they support and preserve their own event semantics when dealing with passing events to and from foreign objects. VHDL simulators, in general, tend to apply the same delta-sweeping semantics to foreign objects as they do to VHDL code although the numerous types of foreign interfaces found in VHDL simulators do not make this universal.

The Verilog language specifies a foreign object interface (the PLI) as part of its standard, but this does not appear to be sufficient to guarantee consistency because different Verilog simulators handle the propagation of events to foreign objects differently, even though both are nominally executing the same code. One implication of this is that a test harness that relies only on event queueing semantics of the simulator to separate a clock and the data signals being sampled, instead of inserting a simulation delay between them, cannot function consistently when used with a DSM. This is because some simulators present the events from before nonblocking assignments and the events from after nonblocking assignments to the DSM as a single set of changes. For example:

```
module top ();

reg CLK;
reg I;

initial
begin
#10;
CLK <= 0;
#10;
CLK <= 1;
end

always @(posedge CLK)
begin
I <= 1;
end

ARMDSM theDSM (CLK, I);

endmodule
```

Some Verilog simulators present the changes that occur to **CLK** and **I** at 20 picoseconds as a single set of events, despite the use of a non blocking assignment. In such cases, you must use a small simulation delay for example,1 ps, instead of a nonblocking assign, to ensure correct separation of events. This has implications for the way in which behavioral test benches are designed.

2.2.3 Support implications of compiled models

DSMs are typically derived from the RTL source for the device that they model through the use of a model compiler such as Synopsys VMC. The use of compiled RTL in a DSM has implications about the level of support for:

- configuration
- visibility of internal nodes.

Some ARM cores, particularly those that are synthesizable, can be configured during compilation or synthesis through the use of parameters or Verilog `'define` directives. When compiled, any configuration performed this way is fixed and cannot be subsequently changed. It is therefore not generally possible to configure the RTL code within a DSM in the same way that it can be configured for other uses. Certain DSMs have configuration options introduced through adding behavioral C to the DSM. See the DSM release documentation for your specific model for more details about the configuration options that it supports.

Because the model is compiled, the host simulator is unable to probe its simulation structure. Consequently, you cannot examine nodes within the model for debug visibility. However, DSMs are built with certain internal nodes exported into the simulation wrapper but not as pins on the device. These nodes typically include the register set in addition to other pertinent information. They map to Verilog registers or VHDL signals inside the HDL wrapper, and can be examined by the logic simulator.

Note

Because the DSM is representative of the actual design, these exported signals typically show the actual values present in registers, at any given simulation time. This might not agree with the abstract programmer's model because of optimizations in the implementation. Updated register values can be in the pipeline to update a register bank, when accessed by other blocks using forwarding paths. So the programmer’s model does not always reflect the true status of the core.

Chapter 3
Timing Issues

ARM simulation and sign-off models integrate into back-annotation flows through the use of the SDF timing annotation facilities present in VHDL and Verilog simulators. These models are large, single components with a zero delay execution model. Difficulties can be encountered in successfully integrating them into a timing flow, and there are implications for the level of timing accuracy that can be achieved. This chapter describes these issues and some of the facilities provided to help in the annotation process. It contains the following sections:

3.1 The pin-to-pin timing model

From the point of view of the simulator, the ARM model core is a large single component within the simulation. It is sensitive to all its inputs and can theoretically drive output values as a response to any one of them changing (that is, it is treated as a large combinatorial block). The model and simulator provide no special treatment to clock signals. They are considered as input signals, like any other, that can cause output events to be generated.

Timing annotation is achieved by placing an HDL timing wrapper around this model. The model is specified as a zero-delay model so that the timing behavior can be entirely determined through the use of back-annotation from an SDF file. To illustrate the behavior of a timing shell the following three scenarios are described:

- *Zero delay model core*

  Initially, the behavior of a simple synchronous block with no timing functionality is described.
- *Addition of timing shell* on page 3-3

  The model has an external pin-to-pin timing shell added around it, implementing setup and hold checks, in addition to a delay on the output. It demonstrates that the addition of the timing shell does not affect the functionality of the model.
- *Negative timing constraints* on page 3-6

  The implications of negative timing constraints, for example, negative setup and hold times are examined. It shows you that a simple pin-to-pin timing shell, used in conjunction with a zero-delay core, cannot be used to model negative timing constraints. To support negative timing constraints, a modification to the basic implementation is described in *Use of negative timing checks* on page 3-19.

3.1.1 Zero delay model core

The first example is a simple synchronous block with:

- a clock, **CLK**
- an input signal **IN**, which is sampled on the rising edge of the clock
- an output **OUT**, which is driven on the falling edge of the clock.

Figure 3-1 shows a simple synchronous block with clock.

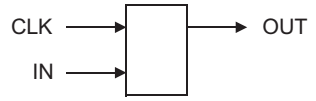

Figure 3-1 A simple synchronous block with clock

The sequence of events from the point of view of the model, without a timing shell is:

1. All signals are set to zero.
2. At 10ps, the input, **IN**, rises to 1. Because the block is sensitive to changes on **IN**, it is evaluated but the behavior of the block is such that no action is taken.
3. At 20ps the clock, **CLK**, rises to 1. The model is evaluated and latches the value of **IN** internally.
4. At 25ps, **IN** falls to 0. The model is evaluated, but takes no action.
5. At 30ps, **CLK** falls to 0. The model is evaluated and determines that a drive to **OUT** is required.
6. Still at 30ps, but in a subsequent simulation evaluation, the output **OUT**, rises to 1 as a result of the drive from the block model.

The above sequence of events is for a zero-delay, behavioral model (that is, a DSM) with no timing shell. The model is sensitive to all its inputs and drives its outputs immediately. Because of the simulation semantics of a DSM, no other part of the simulation is evaluated in parallel with the DSM evaluation. Consequently other functional blocks in a simulation cannot insert values between individual DSM evaluation steps even if they are set to drive its inputs at the same simulation time.

3.1.2 Addition of timing shell

In this case a pin-to-pin timing shell is placed around the periphery of the model. The timing shell, like the model is also behavioral code. It monitors input events without affecting their passage, this is only true in the simple case, see *Use of negative timing checks* on page 3-19 and *Interconnect delays* on page 3-17 for more details, and inserts a delay-buffer through which output events propagate. Assume that the timing shell is annotated from the following SDF fragment:

```
(SETUP (POSEDGE CLK) IN (5))
(HOLD (POSEDGE CLK) IN (6))
(IOPATH (NEGEDGE CLK) OUT (3))
```

The timing shell has no special information about the behavior of the device that it is wrapped around. It also does not have any special knowledge of what the signals are used for (for example, clocks versus synchronously sampled inputs).

The sequence of events and their consequences is modified as follows:

1. Initially all signals are set to zero.
2. At 10ps, the input, **IN**, rises to 1. The timing shell notes the current simulation time 10ps as being the time at which **IN** changes and passes the change through to the underlying model. This behaves exactly as before with no action taken.

3. At 20ps the clock, **CLK**, rises to 1. The timing shell notes the current simulation time 20ps as being the time at which the clock rose. A setup check is now performed of **IN** against the positive edge of the clock by subtracting the stored time at which **IN** changed 10ps from the current simulation time, 20ps. This value 10ps is greater than the setup time of 5ps so no warning is emitted. The timing shell passes the clock change event through to the underlying model which latches the value of **IN** as before.
4. At 25ps, **IN** falls to 0. The timing shell notes the current simulation time at which **IN** changed and replaces its previous value, 10ps. It then performs a hold check by subtracting the stored time at which the clock rose 20ps from the current simulation time 25ps, and notes that the actual hold time of **IN** against the positive edge of the clock was 5ps. This is less than the 6ps indicated in the SDF file, and so a hold violation message is printed in the simulation log. The event on **IN** is passed through to the model as before and the hold violation has no effect on the underlying model behavior. The output is not forced to X because of the timing violation.
5. At 30ps, **CLK** falls to 0. The timing shell notes this time as being the time at which the clock fell. No more action is necessary. The model is evaluated and determines that a drive to **OUT** is required.
6. Still at 30ps, but in a subsequent simulation evaluation, the output **OUT**, rises to 1 as a result of the drive from the block model. The timing shell intercepts this drive from the model and notes that the negative edge of the clock happened at the current simulation time, even though this occurred from a previous simulation evaluation. It therefore inserts a delay on the propagation of **OUT** to the outer edge of the timing shell of 3ps.
7. At 33ps, the delayed version of **OUT** propagates from the outer edge of the timing shell, and into the wider simulation, see Figure 3-2 on page 3-5.

Note

The model core actually drives this output at 30ps simulation time, the subsequent delay is implemented entirely by the timing shell.

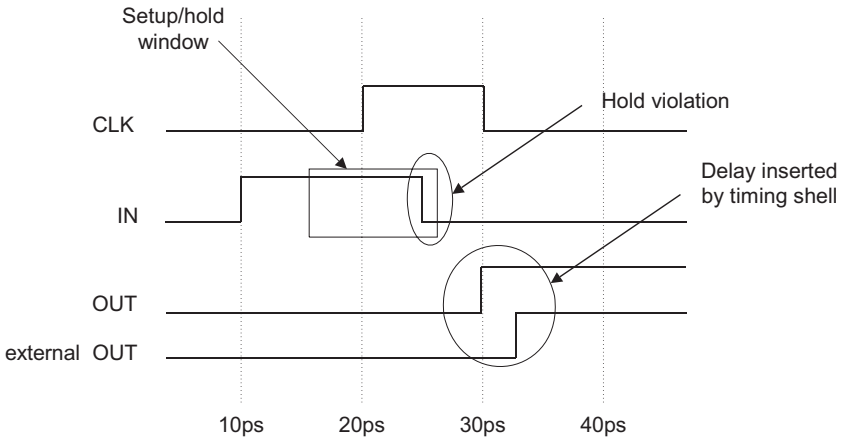

Figure 3-2 A modified sequence of events

As an event simulation, the above behavior differs from physical hardware in a number of ways. Firstly, if we examine the point where the hold-window for the sampling of **IN** by the device was violated:

- In physical hardware, this renders the subsequent behavior of the device unpredictable because it cannot be guaranteed that the correct value of **IN** was actually sampled and used in the evaluation of the output.
- In the event simulation, the sampling of synchronous inputs occurs as a discrete event at the clock edge. The resultant hold violation is a cosmetic warning that does not affect the subsequent behavior of the simulation, and must be taken to indicate that the simulation can no longer be assumed to match reality after this point.

Note

ARM DSMs do not drive outputs to X in response to a timing violation.

Secondly, in this example, the model evaluates its outputs as soon as it has all pertinent information. This is when it receives the negative-edge event on the clock.

- In physical hardware, the delay of 3ps on the output **OUT**, is partly because of internal path delays in the device and partly because of the load to be driven by the output.
- In this event simulation, the new value is calculated immediately and the timing shell simulates the cumulative delays after the fact. This works for the purposes of simulation provided the output value really can be calculated at the point the clock edge which drives them occurs. This might not always be the case, an explanation is provided in the next section.

3.1.3 Negative timing constraints

In this example there in a high performance device where a sampled input has a negative-setup time, and the output it affects is driven by the same clock edge on which the input is sampled. This situation is shown in Figure 3-3.

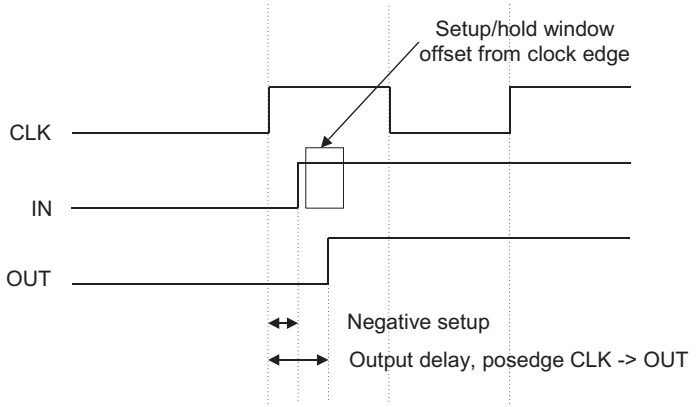

Figure 3-3 A high performance device with negative setup time

In this situation, the value of **IN** can change for a certain time after the clock edge, and still affect the value of the derived output. For example, this can be because of a delay in propagating the clock around the inside of a large device. In reality the output **OUT** is driven at a point some time after the input clock edge is nominally said to have occurred because of propagation delays inside the device. A DSM-style model does not model these delays. The output is driven immediately and delayed by the timing shell. A subsequent change on **IN**, even if it occurs before the final drive of **OUT** by the timing shell, cannot affect the simulation because the model has already driven the result and cannot revoke it.

Simulating this type of behavior in a pin-to-pin timing shell is therefore quite complicated, and if applied to the scenario outlined does not work. See *Use of negative timing checks* on page 3-19 for information on how the timing shell behavior is modified by the simulator to handle these situations.

3.2 Multiple timing paths

In *Addition of timing shell* on page 3-3 it assumed that the timing shell determines that the positive edge of the clock was the causal event for the output delay. Of course the timing shell cannot work this out for itself because it has no understanding of the logic and state inside the model. In that example, this is not significant because there is only a single possible causal input event from which the output can change.

In larger devices it is often the case that an output can change because of multiple causal input events. Examples include multiple clocks such as a memory clock and a test clock, or an asynchronous reset signal. When ARM constructs the timing shell, there is a list of possible causal events for each output. If more than one possible causal event occurs at the same simulation time, the timing shell can potentially pick the wrong event, because it has no understanding of the actual behavior of the model and therefore no way of knowing which event truly was causal. Inaccuracies can be avoided in one of two ways:

- ARM can construct the timing shell so that additional information, exported by the model, can be used to enable only certain timing arcs dynamically depending on the internal state of the model. These timing arcs are enabled and disabled through the use of the conditional timing arc facility (`COND` construct) present in SDF.
- The timing constraints on the model might be arranged so that it is not valid for the input events in question to occur at the same time (enforced through setup and hold checks).

A special case is described in *State-dependent timing*.

3.2.1 State-dependent timing

A special case involving multiple timing arcs that can be problematic is when two or more timing arcs for a given output delay (that is, an `SDF IOPATH` construct, see *SDF annotation* on page 3-9) have the same causal event specified (for example, negedge **CLK**) and differ only through their use of the `SDF COND` facility. These are referred to as state-dependent delays and present a particular problem for SDF timing flows. Unless the vendor software generating the SDF data to be annotated can differentiate between the multiple arcs, then a single set of delay times are annotated onto all of them. This results in the delays being the same regardless of the state.

Static timing analysis cannot normally determine which state a device being modeled is in when calculating delay values. Certain ARM devices (ARM7TDMI, for example) exhibit state-dependent timing. STA based timing flows can be used with such a device, but state-dependency is not modeled in these simulations. An alternative flow that is

able to model state-dependent timing behavior is supported, but a description of this flow is beyond the scope of this document. See *ARM Design Signoff Models: Timing Annotation* for more details.

Note

The use of the SDF `COND` facility does not indicate that state-dependent timing is in effect. True state dependency arises when the use of the `COND` facility is all that distinguishes otherwise identical timing arcs. `COND` is also used by the timing shell to disable individual timing arcs when they are not appropriate.

3.3 SDF annotation

VHDL and Verilog logic simulators perform SDF annotation in similar ways, although there are differences in the specific details between the two languages.

Figure 3-4 shows the structure of a simple timing shell.

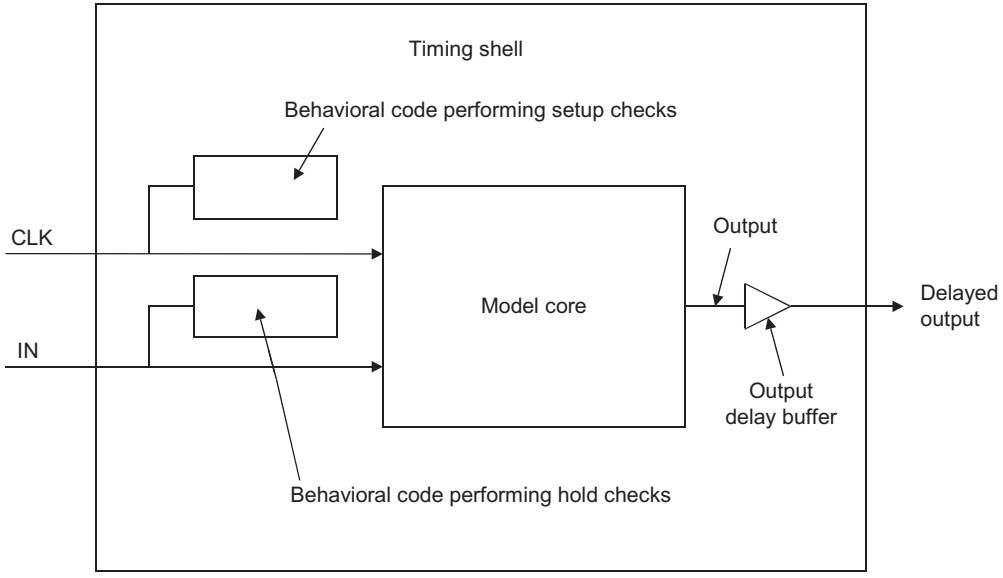

Figure 3-4 Simple timing shell structure

A timing shell is a block of behavioral VHDL or Verilog code although most of the actual behavior is inferred in Verilog implementations, where timing constructs are provided as language features. This code is organized into a routine for each signal in the design, that is invoked when an event occurs on that signal.

For input signals, the code checks that an event is happening at a permitted time, that is, it is not violating a timing check associated with the signal in question. The passage of the input signal is not impeded by timing check code, which only monitors. However, input signals can be delayed for other reasons, See *Interconnect delays* on page 3-17 and *Use of negative timing checks* on page 3-19 for more information.

Output signals are responded to by a buffer that inserts a delay in the propagation of the event into the outside world.

In both of these cases, the timing shell requires time values to work with. Input checks require the setup time and hold time to compare against events. Output delays have to know how long to delay the drive. Unlike normal HDL code, these values are not

present in the code itself and instead a place-holder value is used for each timing arc that the timing shell is interested in. These placeholders are set to some nominal value in the HDL source (typically zero), and filled in automatically at the start of the simulation from an SDF file.

When inserting values from an SDF file into a timing shell, the simulator has to know which placeholders must be used to hold the values. It does this by using a standard mapping so that a timing arc in an SDF file has a defined equivalent in the Verilog or VHDL source. In VHDL this is done by mapping timing arcs to the names of VHDL generics (as described in IEEE 1076.4), where the name encodes the signals and edges involved in the timing arc, in addition to any conditions. The Verilog language is slightly different in that the language contains special system tasks for timing arcs that include edge and conditional information, in their specification. The SDF annotator within the simulator overrides the default values (those which are specified in the source code) of any timing arcs that match the arcs that it finds in an SDF file.

Potential difficulties can arise in SDF annotation when there is a mismatch between what is present in an SDF file and the placeholders in the timing shell. Because an SDF file comes from a source that is generally not under the control of the supplier of the timing shell, the potential for mismatches exists. SDF is sufficiently syntactically rich that a timing arc that obviously matches a particular construct in the timing shell might not be found as a match by the SDF annotator. For example, assuming that the timing shell is constructed so that there is a setup check between a signal, **IN**, and the positive edge of a clock, **CLK**, then the following SDF code might be expected:

```
(SETUP (POSEDGE CLK) IN ...)
```

If, however, the SDF file contains a single setup check between **CLK** and **IN**:

```
(SETUP CLK IN ...)
```

A setup check on a nonspecified edge of a clock is a superset of a setup check on a positive edge, and so is theoretically a valid construct to annotate onto the appropriate check in the timing shell. The OVI SDF2.1 standard suggests that, in this situation, annotation must proceed. However, in practice, different simulators handle this situation differently. Verilog simulators generally annotate a specific timing arc from a less-specific SDF construct like this. VHDL, however, is less flexible and requires an exact match. Also, if the SDF file contains timing arcs that are not in the timing shell, a VHDL annotator generally halts the simulation with an error.

In general, ensuring a match between the SDF file and the timing shell must be regarded as a requirement when performing SDF annotation, although Verilog simulators are typically more flexible in these requirements. This requirement can be achieved with the use of the following:

- *Templates*
- *SDFremap tool* on page 3-13.

3.3.1 Templates

As described in *SDF annotation* on page 3-9, it is generally assumed that the SDF file to be annotated is a match for the timing constructs specified in the timing shell. ARM models are supplied with a template file that assists in ensuring this match. The template file, typically named `<device>.sdft`, is structurally an SDF file but contains no numerical information. Instead it contains the names of timing parameters as used by the device. The primary intent of the template is to serve as a definitive reference for the arcs present in the timing shell. However, this does not mean that the SDF file to be annotated has to precisely match the template file. The order of timing arcs is not generally important, and there are other subtle ways in which the template file can differ from the annotated SDF:

- SDF treats setup and hold timing checks as two separate timing arcs. Generally, they are represented by SETUP and HOLD arcs in an SDF file, and there are two placeholders in the simulation. However, SDF also enables the use of the `SETUPHOLD` construct. ARM models are generally written in such a way that `SETUPHOLD` can be used as a shorthand form of separate `SETUP` and `HOLD` arcs with no ill effects.

  Note

  There are some timing checks where a setup check is performed on one clock edge, a hold check on the following edge, and the data signal must be held stable in the clock-phase between them. This is sometimes referred to as a nochange timing check. Some ARM models contain these type of checks, and they can be represented in an SDF file by unpaired `SETUP` or `HOLD` arcs. Unpaired arcs must not be replaced with a `SETUPHOLD` construct, because this causes the simulator to try to annotate a timing check which does not exist in the timing shell, resulting in an error.

- SDF enables the specification of single values, or triple values (representing minimum, typical, and maximum times). Templates always use the triple form for their placeholders, but this is not a requirement. In addition, `IOPATH` entries in the template can contain up to 12 sets of placeholders, but there is no requirement to provide 12-value SDF for annotation. The placeholders in an SDF template are of secondary importance to the structure of the file, and the actual SDF to be annotated can contain up to six values, representing the six possible transitions a signal can make between the three logic values, **1**, **0** and **Z**. ARM tools that generate SDF from template files SDFremap and SDFgen write no more than six values for an `IOPATH`, regardless of the contents of the template. It is also permissible for the annotated SDF to contain fewer than six values in `IOPATH` arcs.
- For timing arcs that apply to vector signals, the template includes a bit-range for the arc, for example:

  ```
  (IOPATH (NEGEDGE CLK) A[31:0] ...
  ```

Note

. . . indicates more line data but not relevant to the point being made.

In effect, this is a shorthand way of representing 32 different timing arcs:

```
(IOPATH (NEGEDGE CLK) A[31] ...(IOPATH (NEGEDGE CLK) A[30] ...(IOPATH
(NEGEDGE CLK) A[29] ...(IOPATH (NEGEDGE CLK) A[28] ...- - -
```

Note

- - - indicates more lines of SDF.

When generating an SDF file for annotation, it is acceptableto use separate arcs for individual bits of a vector signal, in addition to bundling-up ranges, or using a mixture:

```
(IOPATH (NEGEDGE CLK) A[31:24] ...
(IOPATH (NEGEDGE CLK) A[23] ...
(IOPATH (NEGEDGE CLK) A[22] ...
- - -
```

Even though you can omit certain arcs from an SDF file if you do not want them to be annotated, it is a requirement for VHDL that if a timing arc involving a vector is being annotated, then both ends of the vector must be mentioned in the SDF file, otherwise it fails to annotate. Verilog simulators generally have no such requirement.

To summarize, you must ensure that the SDF annotated on to the model is a good match for the timing shell. The supplied SDF template file provides the definitive reference for the complete set of timing arcs and the form they must take. The following set of guidelines assists you in the annotation process:

- Timing arcs that are more general versions of the same arcs in the template usually annotate anyway when using Verilog, but not VHDL. This means that you can omit edge specifiers and `COND` qualifiers where present in the template, and the SDF still annotates correctly in Verilog.
- Not all arcs present in the template have to be present in the SDF file to be annotated. Arcs that are not present in the SDF, but are present in the template, default to the zero-delay behavior of the model. Parameters for timing checks which have no value annotated onto them also default to zero.
- There must be no arcs present in the section of the SDF file belonging to the model that are not present in the template. This does not include INTERCONNECT or PORT delays. See *SDFremap tool* for more details.

3.3.2 SDFremap tool

If you have problems with a generated SDF file not annotating because it does not provide a good match for the template then you can use the SDFremap tool, provided with the model. SDFremap reads an SDF file containing data for a complete system and a template file and rewrites the cell pertaining to the model so that it matches the template in structure.

The tool works through the template, line by line. For each arc in the template it searches for an arc in the SDF file that matches the type of arc under consideration and uses the same signal(s). Because SDFremap is more flexible than an SDF annotator, it considers arcs that would otherwise be rejected.

If multiple possible matches for an arc are found, SDFremap selects between them by comparing them to see which is the closest match. However, this might not always resolve the situation. Consider the situation where the template contains the following:

```
(SETUPHOLD (POSEDGE CLK) IN (Ts:Ts:Ts) (Th:Th:Th))
```

and the SDF file has the following two entries:

```
(SETUPHOLD (POSEDGE CLK) (POSEDGE IN) (5) (2))
(SETUPHOLD (POSEDGE CLK) (NEGEDGE IN) (4) (2))
```

In this case, the timing software that produced the SDF splits one setup-hold check into two arcs by considering signal **IN** rising and **IN** falling separately. This does not annotate because the template, and therefore the timing shell is less specific than the provided SDF. If you reverse the situation, annotation works on a Verilog simulator but not on a VHDL simulator, because the timing shell is designed for a single setup-hold check that does not take the transition type of **IN** into account.

In this case, SDFremap considers both arcs as being an equally good match for the arc in the template, based on the arc type, signals, and edge information. In this situation, it selects the most pessimistic arc from the SDF file. The selected output for this arc is:

```
(SETUPHOLD (POSEDGE CLK) IN (5) (2))
```

To provide a proper audit trail, SDFremap writes a log file which explains for each arc in the template:

- which arcs (if any) it considered
- what decisions it made.

In addition to correcting edge specifications, SDFremap also adds any `COND` entries that are present in the template (this might not be necessary for Verilog, but has no harmful effect), and removes extra constructs that are not necessary for annotation.

The SDF file can specify a timing arc between two vectors, for example:

```
(IOPATH IN[31:0] OUT[31:0]...
```

Note

The value of 31 above is an example only.

Typically this refers to a combinatorial path where **IN[0]** drives **OUT[0]**, **IN[1]** drives **OUT[1]**, and continues in this manner. As described in *Templates* on page 3-11, vector ranges in SDF are shorthand for a set of timing arcs for each bit in the range. Expanding out the IOPATH above, initially produces the following:

```
(IOPATH IN[31:0] OUT[0]...
(IOPATH IN[31:0] OUT[1]...
- - -

(IOPATH IN[31:0] OUT[30]...
(IOPATH IN[31:0] OUT[31]...
```

However, because the input half of the IOPATH is also a vector, each one of the arcs above is also a shorthand for 32 more arcs:

```
(IOPATH IN[0] OUT[0]...
(IOPATH IN[1] OUT[0]...
- - -

(IOPATH IN[31] OUT[0]...

(IOPATH IN[0] OUT[1]...
(IOPATH IN[1] OUT[1]...
- - -

(IOPATH IN[31] OUT[1] ...
```

This procedure continues, leading up to 1024 (32 squared) arcs in total. The assumption is that each bit on the output vector can be driven by any of 32 bits on the input vector. This is generally not the case, however, and the DSM model implements only 32 arcs, not 1024. In this case the template for the above arc actually reads:

```
(IOPATH IN OUT[31:0]...
```

Here, **IN** is now represented as a single signal so that only 32 timing arcs are inferred by the simulator, giving the required behavior. One common misunderstanding is that this representation somehow results in the wrong behavior because it is specifying the entire vector, **IN**, as the causal signal. However, the timing shell does not specify behavior, only timing (see *Addition of timing shell* on page 3-3 and *Multiple timing paths* on page 3-7). If, for example, the model drove OUT[5] because IN[5] has changed,

then the timing shell recognizes this as the path, `IN -> OUT[5]`, because if one bit of **IN** has changed, then the vector signal itself has also changed and the timing shell functions as required.

Note

See *Vector to vector IOPATH arcs* on page 3-27 for details of difficulties that can arise from vector->vector IOPATHs during annotation in VHDL simulations.

The precise behavior of SDFremap is configurable through its preferences files. You must be aware that the default behavior of the tool might not produce useful results, depending on the input, you must read through the supplied Unix man page for SDFremap before using the tool. Common reasons for problems include:

- The tool does not find any cells to remap. SDFremap matches on the cell type entry in the SDF file. If the cell type in the input SDF does not match the celltype used in the template file, the tool is unable to recognize the cell as the one it is interested in. In this case, you can use the `<celleq>` facility in the preferences file to make SDFremap aware of the correct celltype. See the SDFremap man page (UNIX) for details of how to use the preferences file.
- The remapped SDF refers to the wrong hierarchy level. At the time of writing, ARM models place the timing shell one level below the top-level of their wrapper. Most SDF files are generated with the expectation that the timing shell is at the level at which the ARM model is instantiated. In the future ARM models might be generated with the timing shell at this level, resolving this problem. For the moment, this issue can be resolved using the `<pathtrails>` facility in the SDFremap preferences file.

  Note

  The use of the `<pathtrails>` facility does not fix the hierarchy in any interconnect delays, because these are specified outside the cell to which they refer and are therefore untouched by SDFremap. Currently, hierarchy problems with interconnect delays either require resolving manually or by using a text processing tool such as the UNIX `sed` utility.

- SDFremap does not correctly choose the most pessimistic timing arc. There are a number of reasons why this happen. For input checks, matching on the type of check takes higher priority than pessimistic time matching, so the tool might select a more optimistic timing check that is a closer match to the format of the template over a more pessimistic check that is a poor match.

  For `IOPATHs` which can contain up to six sets of values, SDFremap regards the most pessimistic delay as being the one with the highest average (mean) of the values given.

For three-value SDF, SDFremap defaults to calculating which arc is the most pessimistic by considering the typical timing values (that is, those in the middle of the value-triples). An unhelpful case is where SDF files contain empty typical values for all triples, for example:

```
(SETUPHOLD (POSEDGE CLK) IN (1::2) (2::2))
```

In this case, the default behavior of SDFremap effectivelyturns pessimistic value matching off, because it regards all timing arcs as specifying zero as their value. You can configure SDFremap to use any of the three values for the purpose of calculating pessimistic delays through the use of the preferences file. See the SDFremap man page (UNIX) for more detail.

3.4 Interconnect delays

The timing checks and delays explicitly built in to a timing shell and referenced in an SDF template file are applied at the boundary of the model and do not interact directly with other cells in the design.

The timing shells also have provision for interconnect delays from neighboring cells to be annotated. In the simple case (a so called single-source interconnect delay, where an interconnect wire between cells has a single driver and a single reader), interconnect delays are modeled by HDL simulators by placing a delay buffer on the input side of a wire. Timing shells place a buffer on every input to the model. As with `IOPATH` delays, the buffer has a placeholder-delay associated with it that is initially set to zero.

During SDF annotation, the placeholder is overridden by SDF `INTERCONNECT` or `PORT` timing arcs (in the simple case, `PORT` and `INTERCONNECT` are treated identically by SDF annotators). Any input signals that are subject to interconnect delays are then held by the buffer for an appropriate amount of time. After this delay, the event is registered by any timing checks and the model itself. Interconnect delay buffers can therefore be regarded as an extra shell around the timing shell that delays input events before passing them through to the normal timing shell where they are processed as usual. Setup and hold, in addition to any other timing checks, occur on the delayed value of the input signal, not the undelayed version. If a signal used as the reference signal in an `IOPATH` delay is subject to an interconnect delay, it is the delayed version of the event which is regarded as causal.

Output events driven by the model propagate into other components within the wider design which can also choose to add interconnect delays in addition to any `IOPATH` delay added by the timing shell of the model. These delays are applied downstream by whatever is consuming the events, and do not affect the timing shell.

3.4.1 Multi-source interconnect delays

In some cases, an input to the model can have more than one possible driver (a databus is a common example which can be driven by memory or a DMA peripheral). In such cases, a single value for the interconnect delay, annotated at the site where the signal is read, is insufficient because the interconnect delay can differ according to the driver (See the section on interconnect delays in OVI SDF2.1, for an example. For reference details see *Further reading* on page ix).

In the case of these multi-source interconnect delays, Verilog SDF annotators, when invoked with the correct options (see your simulator vendor manual for more detail) attempt to spread the delays around. They annotate part of the delay on the input part of the signal, and part of the delay on the output buffers where the signal is driven. ARM Verilog timing shells function with multi-source interconnect delays if the models

driving them have the appropriate output buffers. ARM Verilog timing shells also provide the appropriate buffer constructs on their own outputs to enable them to be used as annotation sites for multi-source interconnect delays.

ARM VHDL timing shells only support single-source interconnect delays.

3.5 Use of negative timing checks

The simple zero delay behavior outlined in *Zero delay model core* on page 3-2 is generally suitable for simulating the typical case. Synchronous inputs are required to be stable at some point before a clock edge, the setup time, and held stable for some time afterwards, the hold time. Figure 3-5 shows that sometimes a setup time can be negative.

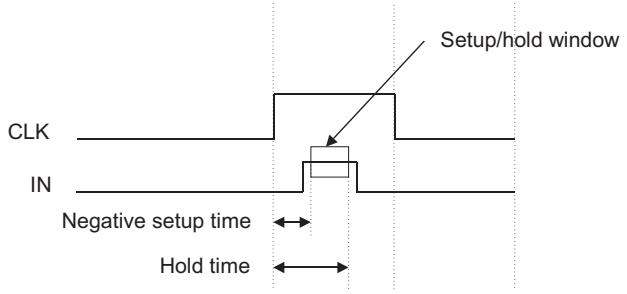

Figure 3-5 Negative setup time

In this case, the input signal **IN**, is allowed to change for some time after the clock edge on which it is nominally sampled. This can occur because of clock-tree distribution delays inside the device. The clock reaches some parts of the hardware before it reaches others. In these cases, the logic inside the device that is working with these inputs is effectively using a delayed version of the clock. The timing behavior of the device, as specified in an SDF file, is still relative to the boundary of the device. Relative to the clock as seen at the device boundary, the setup time of the input, **IN**, is negative.

The zero-delay model with a boundary timing shell cannot directly handle this situation because it does not use distributed delays inside the device directly (pin-to-pin timing annotation is not possible if it does), but models them by delaying a driven output by the total of all the delays involved in producing it. This cumulative delay is the value used in `IOPATH` directives in the SDF file.

Any behavior in the model that depends on the sampled value of **I**N, but which is determined as a response to the clock edge therefore runs the risk of sampling the wrong value of **IN** and diverging from the behavior of the real device.

A solution to this problem is for certain events to be delayed inside the timing shell, before they are presented to the model. In the above case, the clock signal is delayed to produce a new signal, **CLK**'. Figure 3-6 on page 3-20 shows this process.

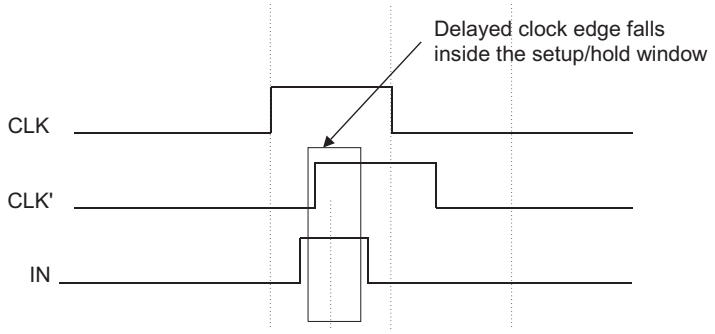

Figure 3-6 Delaying a clock signal to produce a new signal

Any timing checks are still performed relative to the original clock, but the model sees the delayed version, and therefore samples **IN** at a time when it is known to be stable (inside its setup and hold window).

In isolation, this is all that is required. However, the clock is delayed as the model sees it, therefore any outputs driven from that clock are also delayed. If the time by which the clock is delayed is subtracted from all `IOPATH` times from the clock to any outputs, the behavior outside the timing shell is still correct. In addition, delaying the clock can potentially move the point at which other synchronous inputs are sampled to beyond the end of their hold time. Figure 3-7 shows how this occurs.

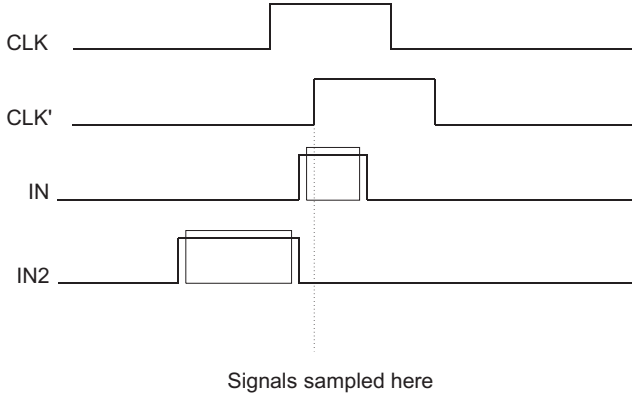

Figure 3-7 Input signals sampled beyond hold time

To accommodate this, adding a delay to the clock means that `IOPATH` delays must be reduced by a corresponding amount. Also, other input signals might require a delay applying to them, similar to the clock delay, so that they are sampled within their setup and hold windows.

The Verilog SDF annotator provides a facility where it automatically delays input signals by an appropriate amount and adjust the output delays to compensate. This is typically enabled with an appropriate switch to the Verilog simulator when it is invoked (see your simulator manual for more information). If this switch is not specified, negative timing checks are typically set to zero.

ARM timing shells for Verilog simulators are constructed in such a way that the simulator can perform this adjustment for negative timing constraints. As a result, negative timing checks can be used with Verilog models. However, there can be cases where it is not possible to adjust the timing behavior to ensure that all IOPATHs and timing checks function as defined, because there can be an `IOPATH` delay which is smaller than the amount by which the clock requires to be delayed to accommodate a negative setup time. As a result, some timing checks might still be overly-pessimistic.

ARM VHDL models do not support negative timing checks.

3.6 Static Timing Analysis and HDL annotated simulation differences

This section describes any potential differences between the results of *Static Timing Analysis* (STA) of a design that uses an ARM core and the timing behavior of its DSM during dynamic HDL simulation of the design. An STA model of a core represents the timing reference model of that core and the timing shell of the DSM must match it as faithfully as possible.

The STA timing model (Synopsys library view) of an ARM core is a pin-to-pin black-box timing model. These models contain timing arcs and lookup tables that are used to calculate the values used in the timing arcs, given the context of the usage of the core in a system. Consequently, the STA tool has no knowledge of the internal structure of the core, which is the same situation for the HDL timing shell provided with a DSM. Therefore it is theoretically possible for the timing shell to contain an exact equivalent set of timing arcs to the STA model, although for some models this is not the case.

There are various reasons why the timing arcs might not match between the two model types and these are described in *Missing arcs*. There are also cases where the timing results can be different because limitations of HDL simulators with respect to STA, these are described in *HDL simulation limitations* on page 3-23. Some mismatches between STA and HDL simulation can result in the simulation timing behavior being more optimistic than STA and others cause it to be more pessimistic. In the optimistic case, an HDL simulation of a design does not report timing violations found by STA (assuming the failing arcs are exercised during the simulation). In the pessimistic case, an HDL simulation issues false timing errors and therefore, error free, full speed simulation is not possible.

3.6.1 Missing arcs

In this case timing arcs that are present in the STA model of a core are not in its DSM timing shell. This causes the DSM to be more optimistic than the STA model. This can be caused by:

- version mismatches
- output to output delays
- output setup and hold checks
- a DSM not matching the implementation of a synthesizable core from a core-licensee (these are cores with the -S suffix and ETMs).

Version mismatches

In some cases a core has had a design change that results in timing arcs being added to the STA model and equivalent changes were not made to its DSM timing shell. When this is identified it can easily be remedied by a minor update to the DSM.

Output to output delays

In some cases implementations of synthesizable devices have output signals that are directly fed-back internally and pass through some logic that drives another output pin. Currently these arcs are not implemented in DSMs. The core-licensee can solve this problem by resynthesizing the device using the `set_fix_multiple_port_nets` command during synthesis with Synopsys tools.

Output setup and hold checks

This is similar to output to output delays and also applies to synthesizable devices. It occurs when output signals are directly fed back internally to flip-flop inputs. You can resolve this problem in the same manner as that used to resolve the output to output delays.

DSM does not match the implementation of a core from a core-licensee

This is an issue for synthesizable devices only. The timing arcs in the DSMs for these cores are developed from test chip implementations of the associated cores. When synthesizable devices are implemented by core-licensees, arcs can vary slightly from implementation to implementation because of modeling differences between technology libraries, synthesis methodologies, and tool versions used. For synthesizable devices, silicon vendors generate their own STA view that matches their implementation of a core, which ARM has no knowledge of. The solution to this is for a core-licensee to also generate the DSM timing shell for their implementation of a core.

3.6.2 HDL simulation limitations

These are related to how the SDF data is treated in the simulation back-annotation flow. These limitations lead to the simulation being more pessimistic than STA. These can occur in negative timing checks, rising and falling setup and hold checks, and single negative setup or hold checks.

Negative timing check issues

This is described in *Use of negative timing checks* on page 3-19 where the output delay for some ports can be smaller than the amount by which the clock is required to be delayed. HDL simulators honour the output delay requirements and zeros any conflicting negative setup checks. STA does not have these constrains and can consider each arc individually, but simulation must take into account the relationships between output delays and setup and hold checks.

Rising and falling setup and hold checks

As described in *SDFremap tool* on page 3-13 the SDFremap tool combines edge specific pairs of setup and hold statements in SDF files into single non-edge specific statements. In doing so it chooses the most pessimistic values from the original file. Synthesis and STA uses the rising and falling values independently. The implemented design can rely on the fact that there is more setup margin for the rising edge of an input verses the falling edge and also in the reverse case. This can cause a simulation to falsely report a setup or hold violation.

Note

This only applies to edge-specifiers on the data signal. Clock-edge specific timing checks are implemented in the timing shell.

Single negative setup or hold checks

In rare cases the STA model can contain a single setup check without a corresponding hold check or a hold check without a corresponding setup check. This is not a problem if the check value is positive but if it is negative then the check is set to zero in simulation. This occurs because only compound Verilog $setuphold checks can be annotated with negative values and if one of the pair of values is missing then the default value in the timing is used, which is zero. Consequently, one half of the check is negative and the other zero, which is invalid because the sum of the two values must be positive. The Verilog simulator overrides the value in the SDF file and sets it to zero. It also issues a warning message informing you that it has set the timing check value to zero.

3.7 Limitations of the timing model

The discussion of limitations of the timing model is subdivided into:

- *IOPATH retain*
- *Vector to vector IOPATH arcs* on page 3-27.

3.7.1 IOPATH retain

Previously, transitions on delayed outputs have been represented as a single transition from the old value to the new value at a point in simulation time. In reality, the latest time at which the old value is guaranteed to be valid and the earliest time at which the new value is guaranteed to be valid can be separated by a period of uncertainty, where the value is unknown, this is shown in Figure 3-8 below.

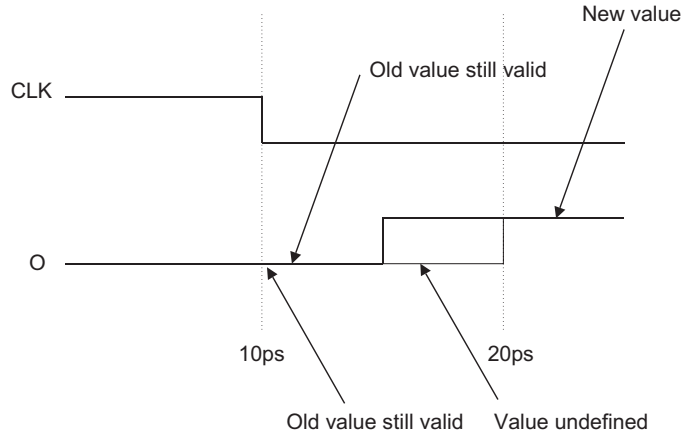

Figure 3-8 Uncertainty of value

In a simulation, the above behavior is modeled by driving the output to X at 15ps, and to 1 at 20ps.

The delays in a DSM have to operate within the limits of a timing shell, and so this behavior cannot be modeled directly. All delays are applied by the code in the timing shell and the model drives the output directly to the new value at the time when the model is invoked. Any modeling of the unknown-period must be performed in the timing shell.

The SDF file format contains support for the specification of the time for which the previous value of an output remains valid, as distinct from the time at which the new value becomes valid. This facility is called `IOPATH RETAIN` and enables up to three times

to be specified for the retain time (that is the time for which the previous value is known to be valid). These three times represent a transition from HIGH, LOW, and high-impedance respectively.

At the time of writing, it is understood that none of the Verilog simulators supported by ARM Limited make use of the `IOPATH RETAIN` facility during SDF annotation, it is ignored. As a result, Verilog models do not support retain times on output delays.

VHDL has recently added some support for `IOPATH RETAIN` in its SDF annotator, but the way this is accomplished requires the timing shell to know how many delay values (up to six) the corresponding `IOPATH` entry in the SDF file is going to provide at the time the timing shell is generated. Because of this limitation, ARM VHDL timing shells do not support `IOPATH RETAIN` at this time.

Simulation inaccuracies

An important implication of the lack of support for `IOPATH RETAIN` is that you must choose whether to put the transition to the new value at the point at which the old value ceases to be valid, or at the point at which the new value is known to be valid when performing SDF back-annotation.

If a device that uses the output values from the DSM samples one during the period at which the value is not defined, it either gets the old value or the new value, depending on which of the two possible times for the transition is in use. In reality, the values must not be sampled during this period, and support for `IOPATH RETAIN` ensures that anything sampled in that period reads an X.

In the absence of support for `IOPATH RETAIN` this situation can potentially lead to a divergence between the simulation and the behavior of the physical device. Furthermore, this divergence is not reported with a warning message, the simulation silently uses the wrong value. One way of detecting this situation is to perform two simulations, one using an SDF file containing minimum values and the other using an SDF file containing maximum values. The SDF files must be generated by an STA tool using the minimum and maximum timing views of a device, respectively. The results of the two simulations are expected to be identical provided outputs are not sampled in the undefined period.

3.7.2 Vector to vector IOPATH arcs

VHDL, through the 1995 VITAL standard, mandated an annotation scheme for `IOPATHs` where both signals are vectors, which requires annotation placeholders for the cartesian product of the two vectors. For example, two 32 bit vectors requires 32 * 32 (1024) annotation points. As mentioned in see *SDFremap tool* on page 3-13, this is inefficient so ARM SDF template files represent one of these signals as a scalar.

The newer, VITAL 2000 standard, recognized that the earlier mechanism was not suitable in situations where there is a one-to-one relationship between an element on the input vector and the same element on the output vector:

IN[0] drives **OUT[0]**

IN[1] drives **OUT[1]**

– – –

This being the case, newer VHDL simulators permit a special case where both input vectors are the same size.

Note

Some devices have vector to vector combinatorial paths where both vectors are not the same size, and this can cause problems with some VITAL 2000 compliant simulator versions.

ARM Limited is investigating ways to work around this difficulty. If you do have a problem with annotation of vector to vector `IOPATH`s on a VITAL 2000 compliant simulator, the current recommended work-around is to use a previous version of the simulator which implements VITAL 95 only.

This problem does not affect Verilog simulators.

Chapter 4
Limitations

This chapter describes the limitations of DSMs. It contains the following section:

- *DSM limitations* on page 4-2.

4.1 DSM limitations

Although DSMs fully match the architecture and functionality of the ARM core design, they are subject to the following limitations:

Simulator functions not supported

The following functions are not supported:

Restart Return the simulation back to time zero without terminating the simulation.

Save and Restore (also known as checkpointing)

Save the simulation at a point of time (snapshot), and restore the simulation to that point of time.

The SWIFT components included in many DSMs cannot be restarted, or their simulation state saved or restored. Consequently DSMs do not support these functions, because the majority of DSMs include SWIFT components.

Internal scan chain modeling

A DSM is derived from the RTL description of the core that it models. The final netlist for the core might contain internal scan chains that were added during synthesis. It is not possible to use DSMs to model these scan chains, because they do not exist in the device RTL. The scan chains are however modeled by the Sign-Off Model (SOM) of a device.

Caches and Registers

Although it is possible to view the register values contained within the DSM simulation, it is not possible for the engineer or designer to introduce any test data directly into the caches or registers, because this cannot be performed in the RTL from which the DSM is derived.

Zero delay simulation

This limitation is described in *DSM event and interface semantics problems* on page 2-6.

Limitations of the timing model

These limitations are described in *Limitations of the timing model* on page 3-25.

Glossary

This glossary describes some of the terms used in this document. Where terms can have several meanings then the meaning provided in this glossary is intended.

Back-annotation The process of applying timing characteristics from the implementation process onto a model.

Boundary scan chain
A boundary scan chain is made up of serially-connected devices that implements boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

Clock gating Gating a clock signal for a macrocell with a control signal, and using the modified clock that results to control the operating state of the macrocell.

CRF *See* Condensed Reference Format.

Condensed Reference Format (CRF)
An ARM proprietary file format for specifying test vectors.

Delta cycle A simulation cycle in which the simulation time at the beginning of the cycle is the same as at the end of the cycle. That is, simulation time is not advanced in a delta cycle.

Design Simulation Model (DSM)
A back-annotation-capable simulation model that can be included within a range of target HDL simulators. It consists of a functional core block and a Verilog or VHDL wrapper.

Delta-sweeping The process by which the VHDL simulator advances through delta cycles. A sweep covers many delta cycles.

Direct Memory Access
An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

DVS *See* Device Validation Suite.

Device Validation Suite
A set of tests to check the functionality of a device against the functionality defined in the Technical Reference Manual. Also stresses Bus Interface Unit (BIU), and low-level memory sub-system, pipleline, cache and Tightly Coupled Memory (TCM) behavior.

Internal scan chain A series of registers connected together to form a path through a device, used during production testing to import test patterns into internal nodes of the device and export the resulting values.

Model Manager A software control manager that handles the event transactions between the model and simulator.

Programming Language Interface (PLI)
For Verilog simulators, an interface by which so-called foreign code (code written in a different language) can be included in a simulation.

SDF *See* Standard Delay Format.

Standard Delay Format (SDF)
The format of a file that contains timing information to the level of individual bits of buses and is used in SDF back-annotation. An SDF file can be generated in a number of ways, but most commonly from a delay calculator.

TAP *See* Test Access Port.

Test Access Port (TAP)
The collection of four mandatory terminals and one optional terminal that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**.

Index

M

N

O

P

R

S

T

U

V

Z

