Verification of ARM v8 Processor using Top Level Testcases

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

(VLSI Design)

By

Abhishek Pandya

(15MECV14)

NIRMA UNIVERSITY

INSTITUTE OF TECHNOLOGY

NAAC ACCREDITED 'A' GRADE

Electronics & Communication Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2017

Verification of ARM v8 Processor using Top Level Testcases

Major Project Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology
in
Electronics & Communication Engineering
(VLSI Design)

By

Abhishek Pandya

(15MECV14)

Under the guidance of

External Project Guide:
Mr. Sohil Patel
Senior Verification Engineer
ARM Embedded Technologies Pvt.Ltd
Bangalore

Internal Project Guide:
Prof. Vaishali Dhare
Assistant Professor
Institute of Technology
NIRMA University, Ahmedabad.

Electronics & Communication Engineering Department
Institute of Technology
NIRMA University
Ahmedabad-382 481
May 2017

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in VLSI Design at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Abhishek Pandya

Disclaimer

NIRMA UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED 'A' GRADE

Certificate

This is to certify that the Major Project entitled **"Verification of ARM v8 Processor using Top Level Testcases "** submitted by **ABHISHEK PANDYA (15MECV14)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design , NIRMA University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge,haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Vaishali Dhare	**Dr. N. M. Devashrayee**
Internal Guide	PG Coordinator (VLSI Design)
Dr. Dilip Kothari	**Dr. Alka Mahajan**
Head, EC Dept.	Director, IT-NU

Date: **Place: Ahmedabad**

ARM®

Certificate

This is to certify that the Project entitled ”**Verification of ARM v8 Processor using Top level testcases**” submitted by **Abhishek Pandya (15MECV14)**, towards the submission of the Project for requirements for the degree of Master of Technology in VLSI Design, NIRMA University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Mr. Sohil Patel
Senior Verification Engineer
ARM Embedded Technologies Pvt.Ltd
Bangalore, India.

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in VLSI Design at NIRMA University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- ABHISHEK PANDYA

Acknowledgement

I would have never succeeded in my thesis without the cooperation, encouragement and help provided to me by various people. Firstly, my sincere thanks to my team for their help during this ongoing internship. Their wisdom, clarity of thought and support motivated me to bring this project to its present state.

I am deeply indebted to my thesis supervisors, **Mr. Sohil Patel** , Senior Verification Engineer at ARM Embedded Technologies Pvt.Ltd his constant motivation regarding the project and also for his constant guidance, supervision, kind co-operation, and invaluable support in all aspects.

I would like to express my sincere gratitude to **Dr. Alka Mahajan** (Director of Institute of Technology - NIRMA University, Ahmedabad) for her continuous guidance and support. I Would like to take this opportunity to thank **Dr. N. M. Devashrayee** (Professor and Program Coordinator, M. Tech - EC (VLSI Design)), Internal Guide **Prof.Vaishali Dhare** (Assistant Professor ,EC) and all the faculties for their vision, support, and encouragement to provide me with the opportunity to carry out my project work in such a renowned and esteemed organization.

Last but not the least I wish to thank my friends for their delightful company which kept me in good humor throughout the year and thus helping me complete the degree program successfully.

- ABHISHEK PANDYA
15MECV14

Abstract

In ASIC design , verification is an essential step in the development of any product.it ensures that the product as designed is the same as the product as intended,by applying the test signals to the design and check whether its matches with the golden output or not. verification is an essential step in the development of any product. Verification ensures that the product as designed is the same as the product as intended. Unfortunately, many design projects do not complete thorough design qualification resulting in products that do not meet customer expectations and require costly design modifications. More the complex is design , the verification of design is also more complex and verification time is also more.So to verify more corner cases easily , Random Instruction Sequence (RIS) is more effective approach . Most of design bugs are flushed out by the deterministic approach, RIS tools are also highly effective in hitting obscure cases.The tool uses template library that contains test-cases. A test file contains registers and memory values. Random test generator never generates a test which is not a valid test.

A Translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location.so the second part is to analyse the TLB and how the address translation works.Thesis also contains invalidation of the TLB. and affected translation registers and attributes for the translation table walk and invalidation.

Contents

List of Figures

Abbreviation Notation and Nomenclature

ARM ARM	ARM Architecture Reference Manual
AVS	Architecture Validation Suites
BHR	Branch History Register
BHT	Branch History Table
CLI	Command Line Interface
DUT	Device Under Test
DVS	Device Validation Suites
ELs	Exception Levels
FIQ	Fast Interrupt Request
GUI	Graphical User Interface
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
OVL	Open Verification Library
PHT	Pattern History Table
RAVEN	Random Architecture Verification Engine
RIS	Random Instruction Sequence
VMSA	Virtual Memory System Architecture
TLB	Translation Lookaside Buffer
RIS	Random Instruction Stream
TPParser	Test Plan Parser
UTB	Unified Test Bench
VAL	Validation Abstraction
VIP	Verification Intellectual Property

Chapter 1

Introduction

1.1 Motivation

Random instruction sequence tools are widely used for processor verification and validation. RIS tools are mainly used to find bugs in RTL design. RIS tools generate situations which are hard to imagine. RIS tools are very useful for hitting corner cases that are very difficult to archive using directed testing. If bugs includes some specific sequence of instructions in narrow time gape than RIS tools which generate random sequence of instructions are very effective. Macros are useful for providing controlled randomness to the test which is useful for targeting a specific area in the processor architecture.

RIS is widely recognized as an effective approach for verifying corner cases that are hard to anticipate. RIS is also highly effective in hitting obscure cases. RIS tools uses random test libraries to cover most of corner cases. Improvements in randomization of RIS tools help us to increases the hit rate of different corner cases.

1.2 Problem Statement

The objective of this project is to develop test-cases that are either generated by MP Random Instruction set tools or write test-cases for particular scenarios and verify the RTL.An effectiveness of the test determines how suitable it is for the verification of targeted area.

1.3 Thesis Organization

This Thesis organized in to seven chapters, a brief info about them are discussed below:

Chapter 2, describes an introduction and general overview about the ARM v8 Architecture.
Chapter 3, describes about the ARM Virtual Memory System Architecture and how the virtual to physical translation occurs and steps for that.
Chapter 4, describes an introduction about the Random Instruction Sequence Tool. It also contains how it is an effective than the Deterministic approach to cover corner cases.
Chapter 5, describes about the AMBA Advanced Xtensible Interface protocol.
Chapter 6, describes about the Out of Order execution and how the barrier instruction works.
Chapter 7, describes Concluding remarks and scope for future work.

Chapter 2

About ARM architecture

2.1 Introduction

ARM architecture is a Reduced Instruction Set Computer (RISC) architecture with the following RISC architecture features:

- A large uniform register file.
- A load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents.
- Simple addressing modes, with all load/store addresses determined from register contents and instruction fields only.

The ARMv8 architecture supports:

- A 64-bit Execution state, AArch64.
- A 32-bit Execution state, AArch32.

The generic names AArch64 and AArch32 describe the 64 and 32-bit Ex states:

- AArch64 Is the 64-bit Execution state, meaning addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

- AArch32 Is the 32-bit Execution state, meaning addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

2.1.1 ARM defines three architecture profiles

- **A**pplication profile: Supports a Virtual Memory System Architecture (VMSA) based on a Memory Management Unit (MMU).
- **R**eal-time profile: Supports a Protected Memory System Architecture (PMSA) based on a Memory Protection.
- **M**icrocontroller profile: Implements a programmer's model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages.

2.1.2 Execution states

Execution state defines the PE execution environment, including:

- The supported register widths.
- The supported instruction sets.
- Significant aspects of:
  - The exception model.
  - The Virtual Memory System Architecture (VMSA).
  - The programmers model.

The Execution states are:

- The 64-bit Execution state. This Execution state:

  - Provides 31 64-bit general-purpose registers, of which X30 is used as the procedure link register.
  - Provides a 64-bit program counter (PC), stack pointers (SPs), and exception link registers (ELRs).
  - Provides 32 128-bit registers for SIMD vector and scalar floating-point support.
  - Defines the ARMv8 Exception model, with up to four Exception levels, EL0 - EL3, that provide an execution privilege hierarchy.
  - Provides support for 64-bit virtual addressing. For more information, including the limits on address ranges.
  - Defines a number of Process state (PSTATE) elements that hold PE state. The A64 instruction set includes instructions that operate directly on various PSTATE elements.
  - Names each system register using a suffix that indicates the lowest Exception level at which the register can be accessed.

- The 32-bit Execution state. This Execution state:
  - Provides 13 32-bit general-purpose registers, and a 32-bit PC, SP, and link register (LR). The LR is used as both an ELR and a procedure link register.Some of these registers have multiple banked instances for use in different PE modes.
  - Provides a single ELR, for exception returns from Hyp mode.
  - Provides 32 64-bit registers for Advanced SIMD vector and scalar floating-point support.
  - Provides two instruction sets, A32 and T32.
  - Supports the ARMv7-A exception model, based on PE modes, and maps this onto the ARMv8 Exception model, that is based on the Exception levels. Provides support for 32-bit virtual addressing.

  – Defines a number of Process state (PSTATE) elements that hold PE state. The A32 and T32 instruction sets include instructions that operate directly on various PSTATE elements, and instructions that access PSTATE by using the Application Program Status Register (APSR) or the Current Program Status Register (CPSR).

Transitioning between the AArch64 and AArch32 Execution states is known as interprocessing. The PE can move between Execution states only on a change of Exception level.This means different software layers, such as an application, an operating system kernel, and a hypervisor, executing at different Exception levels, can execute in different Execution states.

2.2 ARMv8-A security model

Exception levels EL0 - EL3 :

- EL0 unprivileged execution, applications
- EL1 OS kernel
- EL2 supports virtualization of non-secure operation, hypervisor
- EL3 supports switching between two security states

  All implementations must include EL0 and EL1 pointer register selection

  - SP-ELx

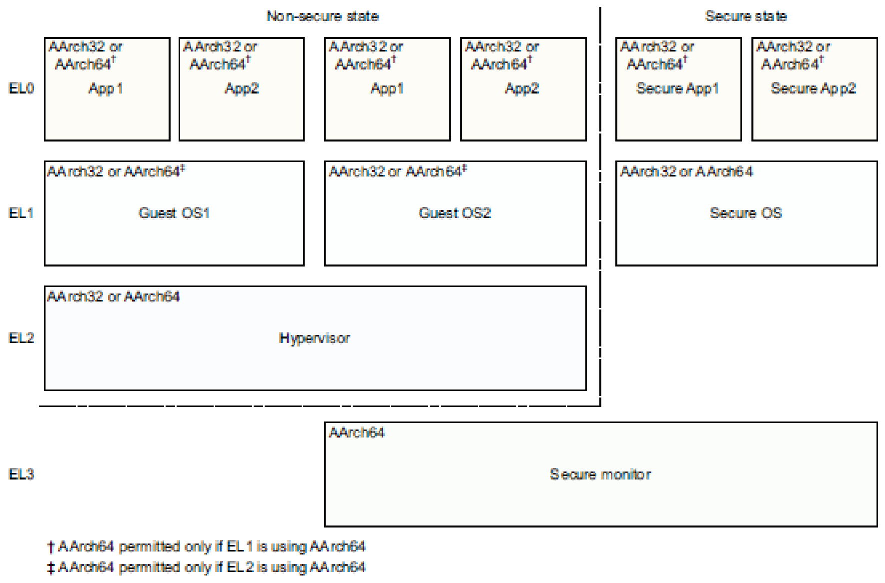

Non-secure state
Secure state
EL0
AArch32 or AArch64† App1
AArch32 or AArch64† App2
AArch32 or AArch64† App1
AArch32 or AArch64† App2
AArch32 or AArch64† Secure App1
AArch32 or AArch64† Secure App2
EL1
AArch32 or AArch64‡ Guest OS1
AArch32 or AArch64‡ Guest OS2
AArch32 or AArch64 Secure OS
EL2
AArch32 or AArch64 Hypervisor
EL3
AArch64 Secure monitor
† AArch64 permitted only if EL1 is using AArch64
‡ AArch64 permitted only if EL2 is using AArch64

Figure 2.1: Exception level model [2]

As from the above , The ARMv8 exception model defines Exception levels EL0-EL3, where EL0 has the lowest software execution privilege, and execution at EL0 is called unprivileged execution. Increased values of n, from 1 to 3, indicate increased software execution privilege.EL2 provides support for processor virtualization. EL3 provides support for two security states, see Security state.

Chapter 3

Virtual Memory System Architecture

A VMSA provides a Memory Management Unit (MMU), that controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the PE.

3.1 Introduction

3.1.1 Virtual Address

- An address used in an instruction, as a data or instruction address, is a Virtual Address .In computing, a virtual address or address space is the set of ranges of virtual addresses that an operating system makes available to a process. The range of virtual addresses usually starts at a low address and can extend to the highest address allowed by the computer's instruction set architecture.

3.1.2 Intermediate Physical Address

- In a translation regime that provides two stages of address translation, the IPA is:
  - The OA from the stage 1 translation.
  - The IA for the stage 2 translation.

a translation regime that provides only one stage of address translation, the IPA is identical to the PA. Alternatively, the translation regime can be considered as having no concept of IPAs.

3.1.3 Physical Address :

- The address of a location in a physical memory map. That is an output address from the PE to the memory system.

3.1.4 Memory management unit :

- A memory management unit , sometimes called paged memory management unit , is a computer hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses. It is usually implemented as part of the central processing unit , but it also can be in the form of a separate integrated circuit. An MMU effectively performs virtual memory management, handling at the same time memory protection, cache control, bus arbitration and, in simpler computer architectures (especially 8-bit systems), bank switching.

3.1.5 Page Table :

- In operating systems that use virtual memory, every process is given the impression that it is working with large, contiguous sections of memory. Physically, the memory of each process may be dispersed across different areas of physical memory, or may have been moved to another storage, typically to a hard disk drive. When a process requests access to data in its memory, it is the responsibility of the operating system to map the virtual address provided by the process to the physical address of the actual memory where that data is stored. The page table is where the operating system stores its mappings of virtual addresses to physical addresses, with each mapping also known as a page table entry.

3.1.6 The translation process :

- The CPU's memory management unit stores a cache of recently used mappings from the operating system's page table. This is called the translation lookaside buffer , which is an associative cache.When a virtual address needs to be translated into a physical address, the TLB is searched first. If a match is found then its a TLB hit, the physical address is returned and memory access can continue. However, if there is no match then its a TLB miss, the handler will typically look up the address mapping in the page table to see whether a mapping exists. If one exists, it is written back to the TLB and this must be done as the hardware accesses memory through the TLB in a virtual memory system, and the faulting instruction is restarted and this may happen in parallel as well. This subsequent translation will find a TLB hit, and the memory access will continue.

3.2 Address translation

A VMSA provides a Memory Management Unit , that controls address translation, access permissions, and memory attribute determination and checking, for memory accesses made by the PE. The process of address translation maps the virtual addresses used by the PE onto the physical addresses of the physical memory system. These translations are defined independently for different Exception levels and Security states, and Fig.3.1 shows:

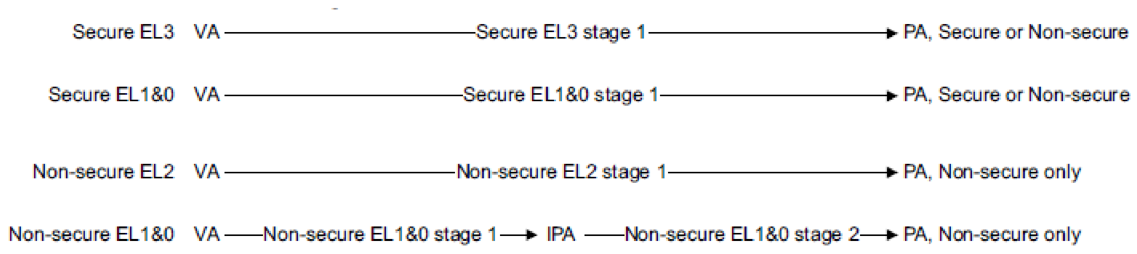


Figure 3.1: Address translations for different Exception levels and Security states [3]

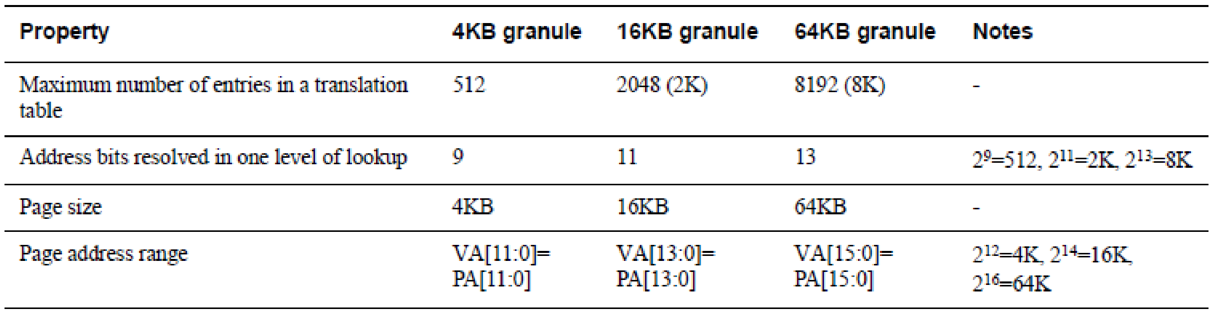
The memory translation granule size defines both:

- The maximum size of a single translation table.
- The memory page size. That is, the granularity of a translation table lookup.

VMSAv8-64 supports translation granule sizes of 4KB, 16KB, and 64KB, and each translation stage is configured to use one of these granule sizes. Using a larger granule size can reduce the maximum required number of levels of address lookup because:

- The increased translation table size means the translation table holds more entries. This means a single lookup can resolve more bits of the input address.
- The increased page size means more of the least-significant address bits are required to address a page.

- These address bits are flat mapped from the input address to the output address, and therefore do not require translation.



Property	4KB granule	16KB granule	64KB granule	Notes
Maximum number of entries in a translation table	512	2048 (2K)	8192 (8K)	-
Address bits resolved in one level of lookup	9	11	13	2^9=512, 2^{11}=2K, 2^{13}=8K
Page size	4KB	16KB	64KB	-
Page address range	VA[11:0]= PA[11:0]	VA[13:0]= PA[13:0]	VA[15:0]= PA[15:0]	2^{12}=4K, 2^{14}=16K, 2^{16}=64K

Figure 3.2: Effect of granule size on a stage of address translation [3]

As Fig.3.2 shows, the translation granule determines the number of address bits:

- Required to address a memory page.
- That can be resolved in a single translation table lookup.

This means the translation granule determines how the input address (IA) is resolved to an output address (OA) by the translation process. The following diagrams show this model, for 4KB of the permitted granule sizes. shows how a 48-bit IA is resolved when using the 4KB translation granule.

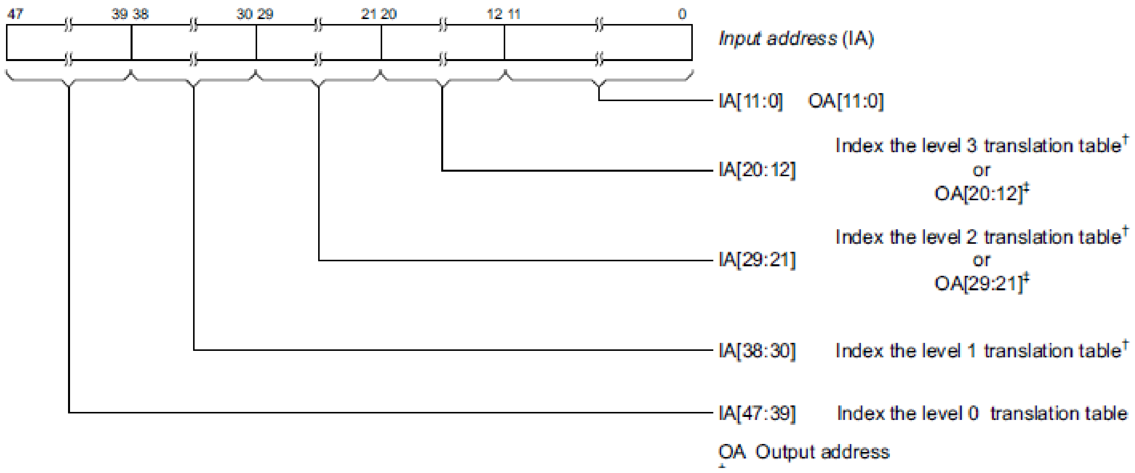

47
39 38
30 29
21 20
12 11
0
Input address (IA)
IA[11:0] OA[11:0]
IA[20:12]
Index the level 3 translation table†
or
OA[20:12]‡
IA[29:21]
Index the level 2 translation table†
or
OA[29:21]‡
IA[38:30]
Index the level 1 translation table†
IA[47:39]
Index the level 0 translation table
OA Output address

Figure 3.3: How the IA is resolved when using the 4KB translation granule [3]

3.3 Translation table walks

A translation table walk comprises one or more translation table lookups. The translation table walk is the set of lookups that are required to translate the virtual address to the physical address. For the Non-secure EL10 translation regime, this set includes lookups for both the stage1 translation and the stage 2 translation. The information returned by a successful translation table walk is:

- The required physical address. If the access is from Secure state this includes identifying whether the access is to the Secure physical address space or the Non-secure physical address space.

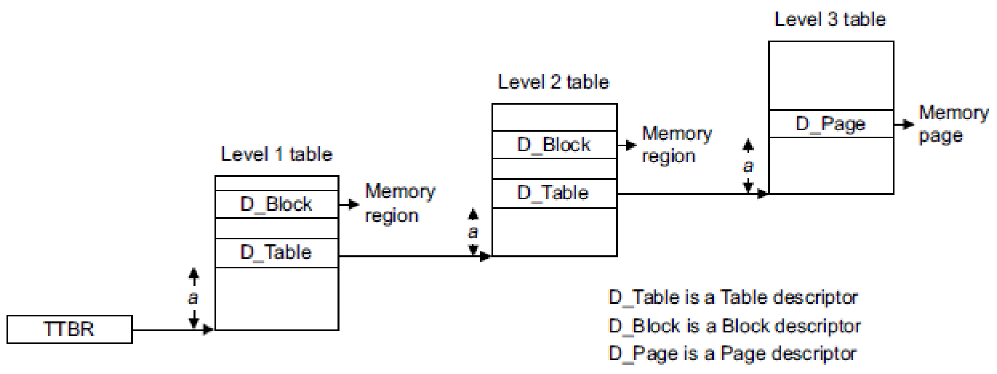


Figure 3.4: Generalized view of a stage of address translation [3]

- The access permissions for the target memory regions.The translation table walk starts with a read of the translation table for the initial lookup. The TTBR for the stage of translation holds the base address of this table. Each translation table lookup returns a descriptor, that indicates one of the following:

- The entry is the final entry of the walk. In this case, the entry contains the OA, and the permissions and attributes for the access.

3.3.1 Full translation Table Walk (Flowchart):

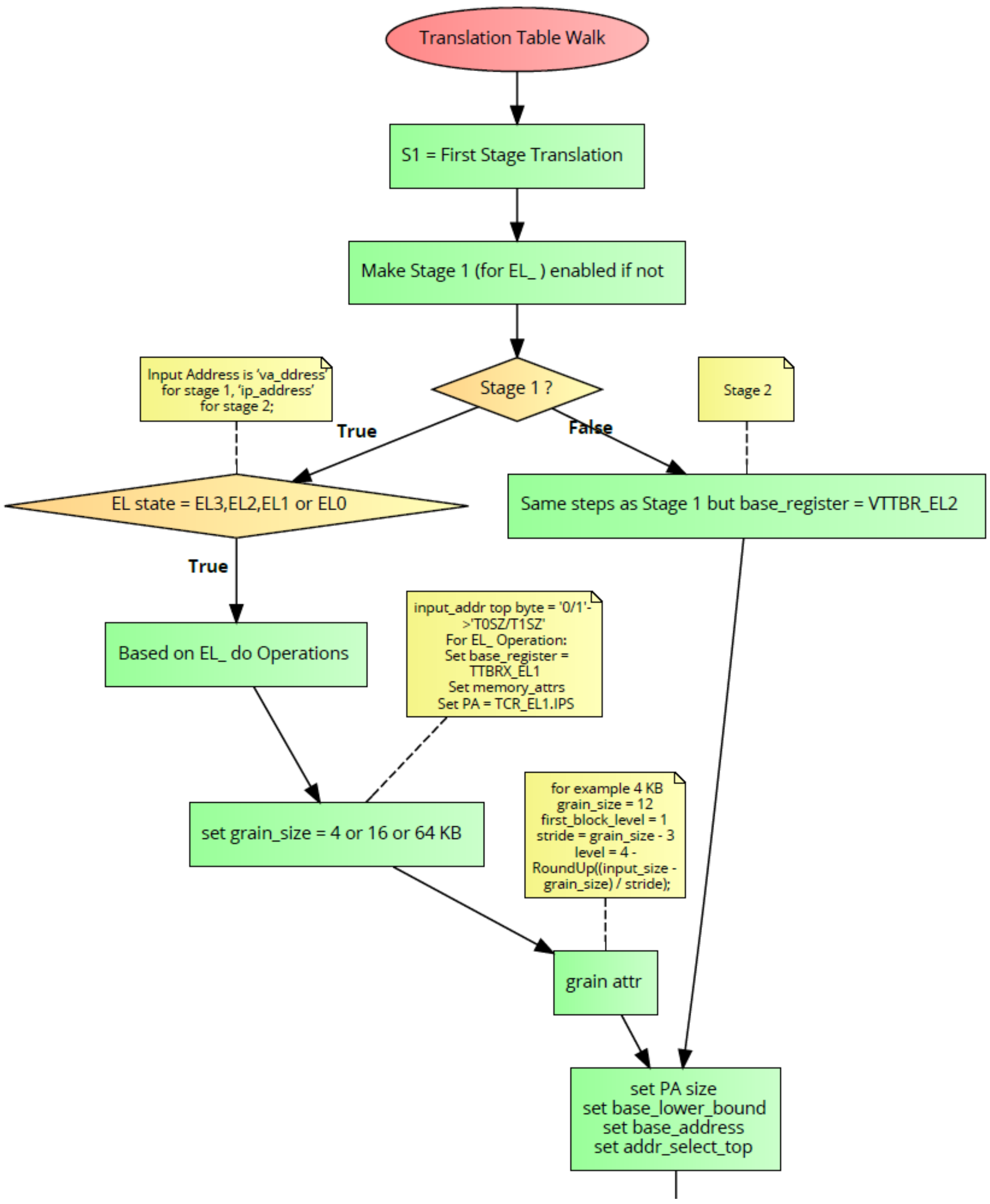

Translation Table Walk
S1 = First Stage Translation
Make Stage 1 (for EL_) enabled if not
Stage 1 ?
Input Address is 'va_ddress' for stage 1, 'ip_address' for stage 2;
True
False
Stage 2
EL state = EL3,EL2,EL1 or EL0
Same steps as Stage 1 but base_register = VTTBR_EL2
True
Based on EL_ do Operations
input_addr top byte = '0/1'->'T0SZ/T1SZ' For EL_ Operation: Set base_register = TTBRX_EL1 Set memory_attrs Set PA = TCR_EL1.IPS
set grain_size = 4 or 16 or 64 KB
for example 4 KB grain_size = 12 first_block_level = 1 stride = grain_size - 3 level = 4 - RoundUp((input_size - grain_size) / stride);
grain attr
set PA size
set base_lower_bound
set base_address
set addr_select_top

Figure 3.5: Flowchart of complete translation table walk [part 1] [3]

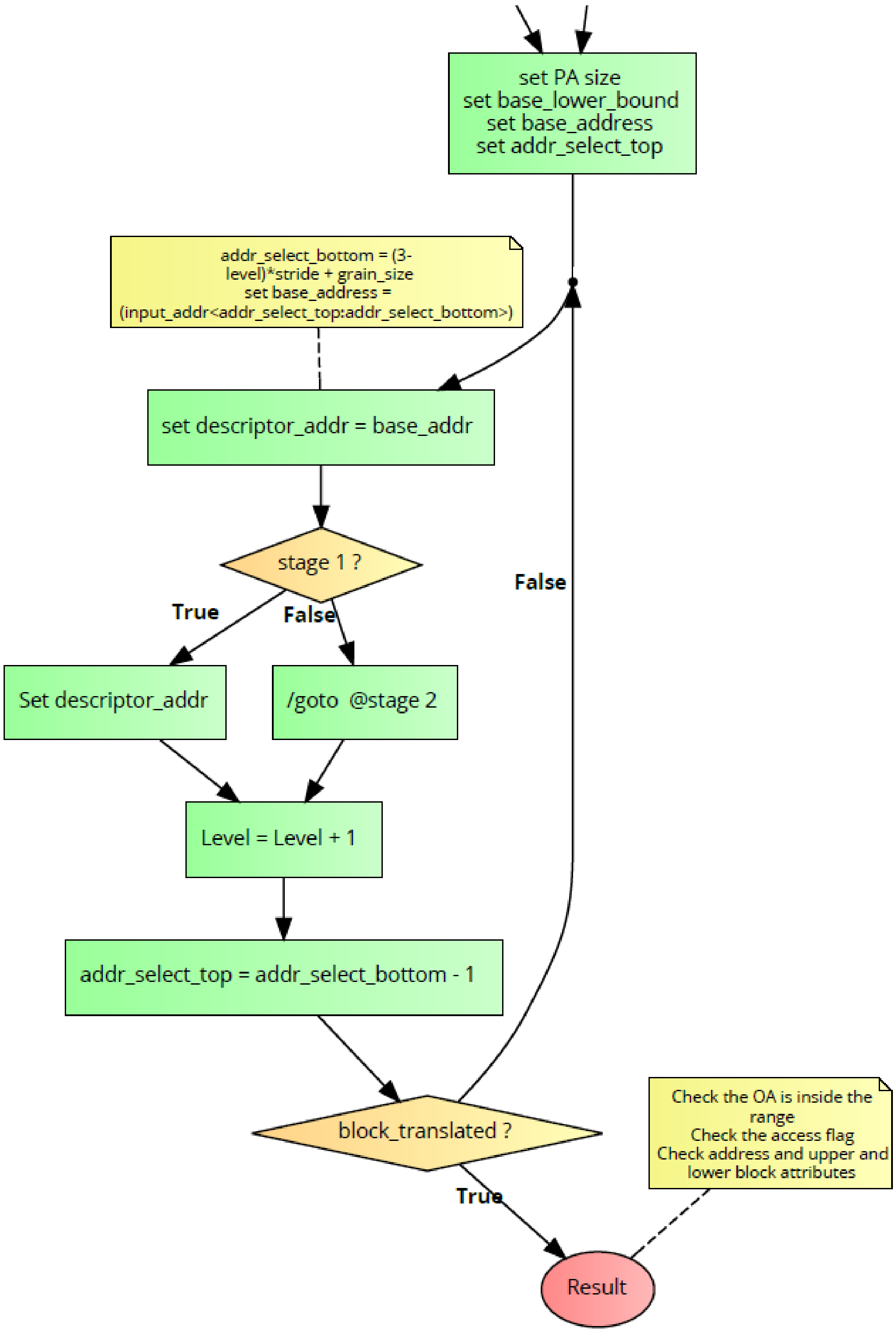

set PA size
set base_lower_bound
set base_address
set addr_select_top
addr_select_bottom = (3-level)*stride + grain_size
set base_address = (input_addr<addr_select_top:addr_select_bottom>)
set descriptor_addr = base_addr
stage 1 ?
True
False
Set descriptor_addr
/goto @stage 2
Level = Level + 1
addr_select_top = addr_select_bottom - 1
False
block_translated ?
True
Check the OA is inside the range
Check the access flag
Check address and upper and lower block attributes
Result

Figure 3.6: Flowchart of complete translation table walk [part 2] [3]

Chapter 4

RIS test generator

4.1 Introduction

RIS test generator is for validation of Multi-processing or Cluster systems via RTL simulations,and targeting high instruction generation rate $\geq 1000 IPS$

ARMs next-generation MP RIS verification tool focusing on memory sub-system operations and cross-PE coherency transactions in Multi-Processor/Cluster systems.Its a server-class static RIS generator that achieves high instruction generationrates (greater that 1000 IPS) and designed to allow derived test sequences to quickly achieve their desired intent, while also allowing maximum re-use of generated scenarios for faster overage closure. It offers full support of ARMv8-A AArch64 execution state. The AArch32 A32 (ARM) ISA is partially supported, and no support is available for AArch32 T32 (Thumb) ISA. Instruction groupings can be defined to target specific operations and micro-architectural features. The following are additional high-level features targeted by ARM's new MP RIS Tool:

- Multi-processor memory coherency.
- Barriers.

- Cache and TLB maintenance operations.
- Message passing (Exclusive operations, Load Acquire or Store Release, Atomics).
- Load-store dependencies and hazards.
- Generate traffic to maximize use of load-store pipeline and evictions.
- Used in Server class coherency verification tool.
- 1-32 PEs (Cores), multiple cache hierarchies, sharing domains and interconnects.
- Efficient run time stimulus reaching test intent in less cycles.
- Maximum re-use of generated scenarios through code re-execution.
- Enables faster bug reproduction and randomization around buggy scenario.

4.2 Random Instruction Sequence (RIS) Generation

Random instruction sequence (RIS) tools are widely used across the industry for processor verification and validation. These tools are often used to find design bugs in a relatively stable but not yet mature RTL design. RIS tools are very effective in generating test scenarios that are hard to envision. However, quite often completely random instruction sequences are of little test value for exposing corner cases in the design, especially if the bug involves a sequence of events happening in a narrow timing window. Macros can help enhance the test quality of the generated instruction sequences by providing controlled randomness around a specific sequence of instructions targeting a specific area in the processor architecture.

4.3 Random Vs Deterministic stimulus generation

Random stimulus generation is widely recognized as an effective approach for verifying corner cases that are hard to anticipate. We found that, while most of design bugs are used out by the deterministic approach, random instruction sequences are also highly effective in hitting obscure cases, often finding bugs that may lay undetected for years in real-life applications. ARM has an internal tool that can generate targeted random code sequences known as RIS. With RIS, we pre-generate self-checking tests using an ISS as the reference design. This technique won't catch design errors that are present in both the ISS and the HDL model, but in practice this situation is rare and these sequences are likely to show design errors in either model when enough sequences have been simulated. The mainstay methodology that we have used since the early days of the first ARM CPU design is deterministic simulation. This is a common and well understood methodology that offers a number of advantages, although it's limited by the amount of effort required to generate test cases and the performance of simulation tools. At ARM, we develop test cases as self-checking assembler sequences. We then replay these code sequences on a simple simulation testbench consisting of the ARM CPU, a simple memory model, and some simple memory-mapped peripherals. Our tests fall into two categories, AVS (Architecture Validation Suites) and DVS (Device Validation Suites). ARM's all AVS class tests check architectural functionality such as the instruction set architecture (32-bit and 16-bit Thumb), the exception model, and the debug architecture. Our DVS tests focus on the behaviour of specific cores and check corner cases arising from the particular implementation. An advantage of this type of test case is that tests are self-contained and portable from ISS (Instruction Set Simulator) environments to Verilog or VHDL test bench environments, or to FPGA prototypes and eventually to silicon. Thus, our customers and we can verify the functional equivalence of all these design views. These suites of tests are effectively the ARM architecture compliance suites.

4.4 Test Topology

The typical structure of Test Topology in RIS tool Fig.4.1.

Following are the steps :-

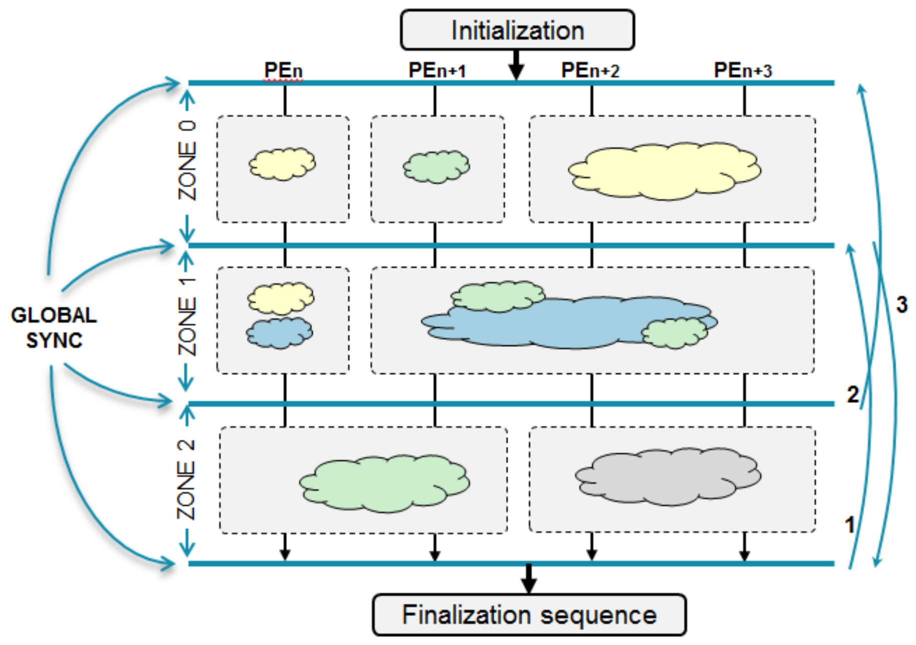


Figure 4.1: Test topology

a. Initialization sets up test environment

b. Threads executed in each PE

c. All threads synchronized globally

- Partitions test flow into Zones
- Zone order defines initial test sequence

d. Scenarios include one or multiple PEs

- Define memory dependencies between PEs

e. Re-execution of Zones in Random order

f. Finalization sequence cleans up

Chapter 5

AMBA AXI Protocol

The AMBA AXI protocol supports high-performance, high-frequency system designs. The AXI protocol:

- is suitable for high-bandwidth and low-latency designs
- provides high-frequency operation without using complex bridges
- meets the interface requirements of a wide range of components
- is suitable for memory controllers with high initial access latency
- provides flexibility in the implementation of interconnect architectures
- is backward-compatible with existing AHB and APB interfaces.

The key features of the AXI protocol are:

- separate address/control and data phases
- support for unaligned data transfers, using byte strobes
- uses burst-based transactions with only the start address issued
- separate read and write data channels, that can provide low-cost Direct Memory Access (DMA)

- support for issuing multiple outstanding addresses
- support for out-of-order transaction completion
- permits easy addition of register stages to provide timing closure.

The AXI protocol includes the optional extensions that cover signaling for low-power operation.

5.1 AXI Architecture

The AXI protocol is burst-based and defines the following independent transaction channels:

- read address
- read data
- write address
- write data
- write response

An address channel carries control information that describes the nature of the data to be transferred. The data is transferred between master and slave using either:

- A write data channel to transfer data from the master to the slave. In a write transaction, the slave uses the write response channel to signal the completion of the transfer to the master.
- A read data channel to transfer data from the slave to the master.

The AXI protocol:

- permits address information to be issued ahead of the actual data transfer

- supports multiple outstanding transactions
- supports out-of-order completion of transactions.

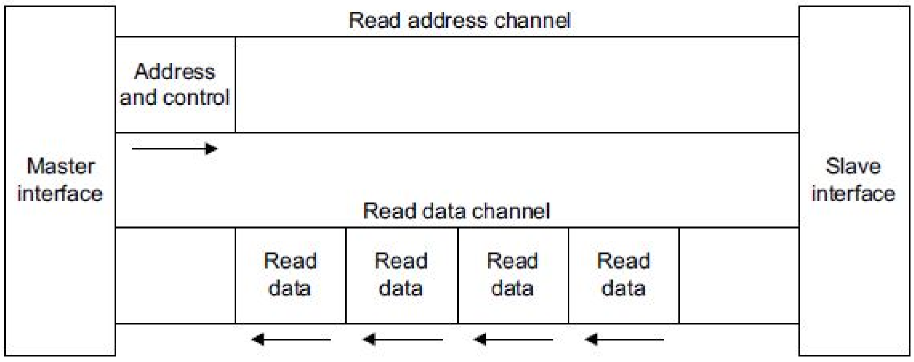


Figure 5.1: Channel architecture of reads

shows how a write transaction uses the write address, write data, and write response channels.

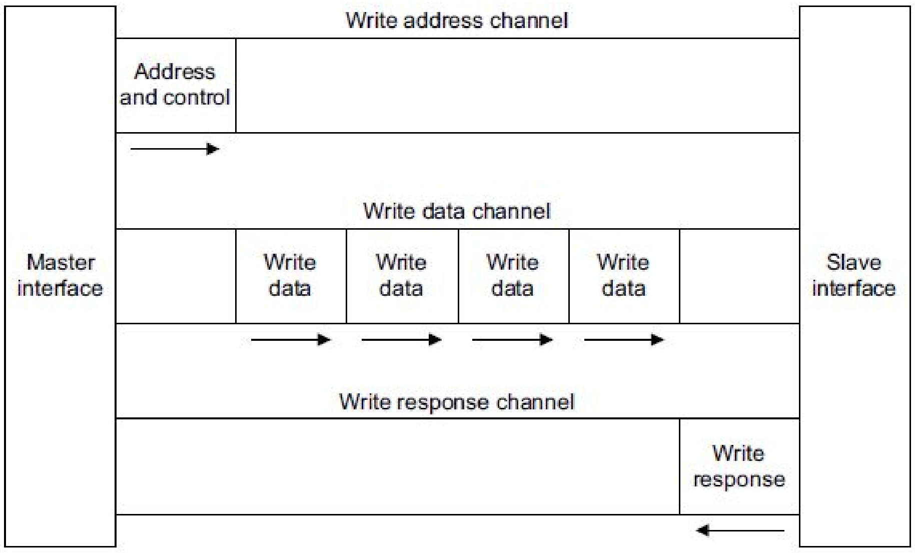


Figure 5.2: Channel architecture of writes

5.2 Basic read and write transactions

This section defines the basic mechanisms for AXI protocol transactions. The basic mechanisms are:

- the Handshake process
- the Channel signaling requirements

5.2.1 Handshake process

All transaction channels use the VALID/READY handshake process to transfer address, data, and control information. This two-way flow control mechanism means both the master and slave can control the rate at which the information moves between master and slave. The source generates the VALID signal to indicate when the address, data or control information is available. The destination generates the READY signal to indicate that it can accept the information. Transfer occurs only when both the VALID and READY signals are HIGH. On master and slave interfaces there must be no combinatorial paths between input and output signals. data or control information after T1 and asserts the VALID signal. The destination asserts the READY signal after T2, and the source must keep its information stable until the transfer occurs at T3, when this assertion is recognized.

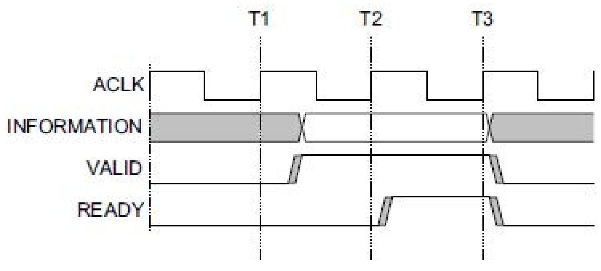


Figure 5.3: VALID before READY handshake

A source is not permitted to wait until READY is asserted before asserting VALID. Once VALID is asserted it must remain asserted until the handshake occurs, at a rising clock edge at which VALID and READY are both asserted. the destination asserts READY, after T1, before the address, data or control information is valid,indicating that it can accept the information. The source presents the information, and asserts VALID, after T2, and the transfer occurs at T3, when this assertion is recognized. In this case, transfer occurs in a single cycle.

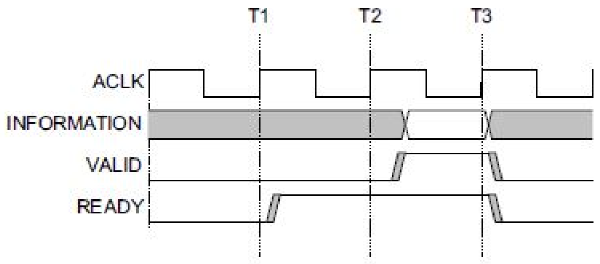


Figure 5.4: READY before VALID handshake

A destination is permitted to wait for VALID to be asserted before asserting the corresponding READY.If READY is asserted, it is permitted to deassert READY before VALID is asserted.
both the source and destination happen to indicate, after T1, that they can transfer the address, data or control information. In this case the transfer occurs at the rising clock edge when the assertion of both VALID and READY can be recognized. This means the transfer occurs at T2.

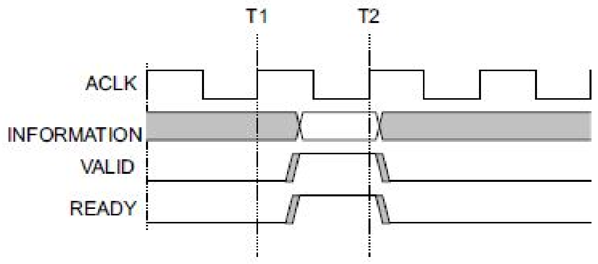


Figure 5.5: VALID with READY handshake

5.2.2 Channel Signaling

- Read transaction dependencies

  Fig shows the read transaction handshake signal dependencies, and shows that, in a read transaction:

  - the master must not wait for the slave to assert ARREADY before asserting ARVALID
  - the slave can wait for ARVALID to be asserted before it asserts ARREADY
  - the slave can assert ARREADY before ARVALID is asserted
  - the slave must wait for both ARVALID and ARREADY to be asserted before it asserts RVALID to indicate that valid data is available
  - the slave must not wait for the master to assert RREADY before asserting RVALID
  - the master can wait for RVALID to be asserted before it asserts RREADY
  - the master can assert RREADY before RVALID is asserted.

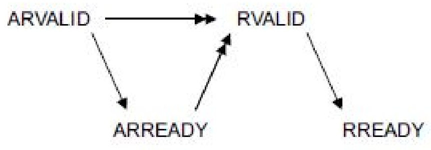


Figure 5.6: Read transaction handshake dependencies

- Write transaction dependencies
  Fig shows the write transaction handshake signal dependencies, and shows that in a write transaction:

  - the master must not wait for the slave to assert AWREADY or WREADY before asserting AWVALID or WVALID
  - the slave can wait for AWVALID or WVALID, or both before asserting AWREADY

- the slave can assert AWREADY before AWVALID or WVALID, or both, are asserted
- the slave can wait for AWVALID or WVALID, or both, before asserting WREADY
- the slave can assert WREADY before AWVALID or WVALID, or both, are asserted
- the slave must wait for both WVALID and WREADY to be asserted before asserting BVALID the slave must also wait for WLAST to be asserted before asserting BVALID, because the write response, BRESP, must be signaled only after the last data transfer of a write transaction
- the slave must not wait for the master to assert BREADY before asserting BVALID
- the master can wait for BVALID before asserting BREADY
- the master can assert BREADY before BVALID is asserted.

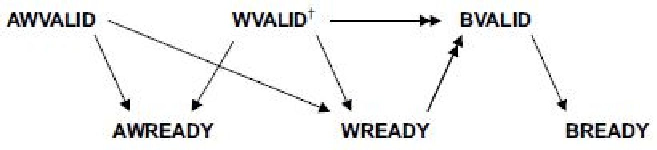


Figure 5.7: Write transaction handshake dependencies

Chapter 6

Out of Order Execution

Earlier computer programs behaved in practice pretty much the way we might expect them to from looking at the source code.
1) Things happened in the way specified in the program.
2) Things happened in the order specified in the program.
3) Things happened the number of times specified in the program (no more, no less).
4) Things happened one at a time.

6.1 Out Of Order

In order for existing programs and programming models to remain functional, even the most extreme modern processors will attempt to preserve the illusion of Sequential Execution from within the executing program. However, in underneath, lot of things will be going on that cannot be hidden from outside the processor.

Consider the following code-snippet, which has a couple of instructions that could potentially take more than one cycle before the result is available to subsequent instructions. Both a mul and a ldr can on several architectures require multiple cycles before their results are available. In this case we assume 2 cycles for each.

```
add r0, r0, #4
mul r2, r2, r3
str r2, [r0]
ldr r4, [r1]
sub r1, r4, r2
bx lr
```

Figure 6.1: Assembly code for execution

If we execute this code on an in-order processor, the execution will look something like the following:

Cycle	Issue
0	`add r0, r0, #4`
1	`mul r2, r2, r3`
2	`*stall*`
3	`str r2, [r0]`
4	`ldr r4, [r1]`
5	`*stall*`
6	`sub r1, r4, r2`
7	`bx lr`

Figure 6.2: Assembly code for in-order execution

While if we execute it on an out-of-order processor, we might see something more like

Cycle	Issue
0	`add r0, r0, #4`
1	`mul r2, r2, r3`
2	`ldr r4, [r1]`
3	`str r2, [r0]`
4	`sub r1, r4, r2`
5	`bx lr`

Figure 6.3: Assembly code for out-of- order execution

By permitting the ldr to execute while we wait for the mul to complete so that the str can progress, we have also given more time for the ldr to complete before its value is needed.

6.2 Need of Barriers

A barrier, in some architectures called a fence, is an operation that explicitly enforces some type of ordering of memory accesses. On the higher level this can mean compiler directives preventing load/store operations from being reordered across a line in the source code, but leaving the compiler free to rearrange memory accesses on either side with other accesses on the same side. On the lower level, this can mean dedicated instructions stopping execution on a core until all previous memory accesses are guaranteed to be visible to other agents in the system. An agent is any device in the system capable of initiating bus transactions - for example a processor or a DMA controller.

Figure shows an example of a barrier affecting the ordering of load-store instructions.

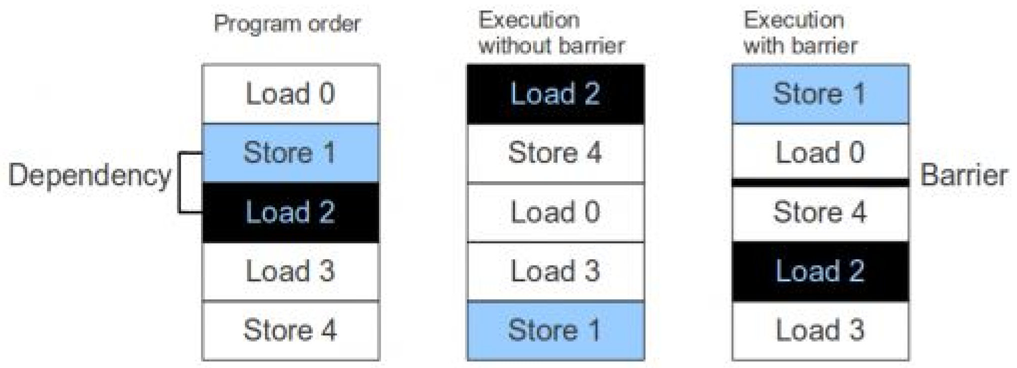


Figure 6.4: Example of Barrier Effect

There is an ordering dependency that the effects of Store 1 are visible to Load 2. For example, Store 1 might be a write to a configuration register that remaps the physical address of a peripheral that is then read from by Load 2. Note that accesses on either side of the barrier can still be freely reordered where there are no address dependencies.

6.3 Barriers

Barriers were introduced progressively into the ARM architecture.

- Instruction Synchronization Barrier (ISB)

  The Instruction Synchronization Barrier ensures that any subsequent instructions are fetched anew from cache in order that privilege and access is checked with the current MMU configuration. It is used to ensure any previously executed context changing operations (including cp15 operations) will have completed by the time the ISB completed.

- Data Memory Barrier (DMB)

  The basic functionality of a DMB is as follows:
  It prevents reordering of data accesses instructions across itself. All data accesses by this processor/core before the DMB will be visible to all other masters within the specified shareability domain before any of the data accesses after it. It also ensures that any explicit preceding data (or unified) cache maintenance operations have completed before any subsequent data accesses are executed.

  The DMB instruction takes two optional parameters: an operation type (stores only - 'ST' - or loads and stores) and a domain. The default operation type is loads and stores and the default domain is System. So, in effect DMB is shorthand for DMB SY. All possible combinations of types and domains are legal operations on any processor, even if it does not implement the specific functionality described, and can be substituted internally for any stronger barrier.

- Data Synchronization Barrier (DSB)

  The Data Synchronization Barrier enforces the same ordering as the Data Memory Barrier, but it also blocks execution of any further instructions until synchronization is complete. It also waits until all cache and branch predictor maintenance operations have completed for the specified shareability domain. If the access type is load and store then it also waits for any TLB maintenance operations to complete.

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

Random instruction sequence (RIS) tools are widely used across the industry for processor verification and validation. RIS tools are very effective to generate test scenarios that are hard to envision. Improvement in randomization of RIS tool helps us to increases hit rate of different corner cases. Some templates are written in such way that it forces the simulator to generate fixed scenarios to hit corner cases like instruction optimization, crypto, early forward.

7.2 Future Scope

- Generate testcases from RIS tool that is targeted to memory area and debug the failed testcases.
- Analysis of different features of memory management unit. Analysis of Cache hierarchy and translation from virtual address to physical address and page table replacement policy.
- Understanding of micro-architecture and various components of it.

References

[1] ARM Architecture Reference Manual (Beta) for ARMv8-A. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset .architecture.reference/index.html

[2] www.verificationacademy.com

[3] http://www.verificationguide.com/p/home.html

[4] Jhon L. Hennessy, David A. Patterson, ”Data-Level Parallelism in Vector, SIMD GPU Architecture”. in Computer Architecture: A Quantitative Approach, 5th ed., MA: Morgan Kaufmann, 2012.

[5] AMBA AXI and ACE Protocol Specification