Configuration Register Validation

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics & Communication Engineering

(VLSI Design)

By

Udani Kausshalee A

(11MECV18)

Department of Electrical Engineering

Electronics & Communication Programme

Institute of Technology,

Nirma University, Ahmedabad-382 481

MAY 2013

Configuration Register Validation

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

In

Electronics & Communication Engineering

(VLSI Design)

By

Udani Kaushalee

(11MECV18)

Under the Extenal Guidance of

Mr. Aman Malhotra

Under the Internal Guidance of

Dr. Usha Mehta

Department of Electrical Engineering

Electronics & Communication Programme

Institute of Technology,

Nirma University, Ahmedabad-382 481

MAY 2013

Declaration

This is to certify that

(i) 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.

(ii) Due acknowledgement has been made in the text to all other material used.

Udani Kaushalee A

Certificate

This is to certify that the Major Project entitled **"Configuration Register Validation"** submitted by **Udani Kaushalee A.(11MECV18)**,towards the partial fulfillment of the requirements for the degree of **Master of Technology (VLSI Design)** in the field of **Electronics and Communication** of Nirma University is the record of work carried out by her under our supervision and guidance. The work submitted has reached a level required for being accepted for examination. The results embodied in this major project, to the best of our knowledge, have not been submitted to any other University or Institution for award of any degree or diploma.

Date: Place:Ahmedabad

Internal Guide	**External Guide**
Dr. Usha Mehta	Mr. Aman Malhotra
Sr. Associate Professor	Manager
VLSI Design	Intel Architecture Group
Nirma University	Intel India Pvt. Ltd.

Head of Department	**Director**
Dr. P.N.Tekwani	Dr K Kotecha
HOD of EE	Director, IT, NU

Acknowledgement

would like to take this opportunity to thank all those who have enabled and encouraged me to take up this project and bring it to fruitful completion.

I would like to thank Professor and Program Coordinator, M.Tech EC (VLSI Design), Dr. N. M. Devashrayee, for giving us the opportunity to take up an industrial project.

I would like to express my gratitude to my project guides, Dr. Usha Mehta, Assistant Professor, Department of Electronics and Communication Engineering, Nirma University for his continuous support, guidance and concern.

I would like to extend my gratitude to Aman Malhotra, Engineering Manager, Intel India Pvt Ltd for his constant encouragement and support throughout my thesis.

Im indebted to Pati, Raghunadha, Techlead, Intel India Pvt Ltd, for his technical guidance, immense knowledge, patience and for posing challenging tasks making the past year an enriching one. Special recognition goes out to all those at Intel whove helped me in improving my technical knowledge and debugging skills.

I thank all faculty members of the Department of Electronics and Communication Engineering, Nirma University for their feedback during the reviews which aided in improving this project.

Last but not the least; I would like to thank my family and friends for all their moral support.

Udani Kaushalee A.
(11MECV18)

Abstract

Increasing design complexity, shrinking time to market, and high cost of fixing a bug in a released product make pre-silicon validation of microprocessors a major ingredient in the product development cycle. The complexity and hence the cost of validating a microprocessor increases from one generation to the next, making it a very important and challenging problem for current and future designs.

HW/SW interaction of any microprocessor relies on an accurate implementation of the specification which can be obtained with the help of Boot sequences, Device Driver functionality and Control & Status. These processes are done by different register programming of the microprocessor.

CR Validation refers to 'Attributes Validation' of all the registers present in the microprocessor. Attributes that needs to be covered for validation are all the properties of a register that are supposed to be obeyed by it, except functions that it triggers upon configuration of it.

Some of the key challenges to the CR Validation are large number of registers under validation,ultra complex attributes of registers.

This report introduces type of configuration registers, basic validation environment required to validate them, attributes complexities and subsequent verification complexities and optimal approach to address this problem at different steps of validation.

Contents

Chapter 1

Intoduction

1.1 Processor Register

A register is a very small amount of very fast memory that is built into the CPU (central processing unit) in order to speed up its operations by providing quick access to commonly used values.

Memory refers to semiconductor devices whose contents can be accessed (i.e., read and written to) at extremely high speeds but which are held there only temporarily (i.e., while in use or only as long as the power supply remains on). Most memory consists of main memory, which is comprised of RAM (random access memory) chips that are connected to the CPU by a bus (i.e., a set of dedicated wires).

Registers are the top of the memory hierarchy and are the fastest way for the system to manipulate data. Below them are several levels of cache memory, at least some of which is also built into the CPU and some of which might be on other, dedicated chips. Cache memory is slower than registers but much more abundant. Below the various levels of cache is the main memory, which is even slower but vastly more abundant (e.g., hundreds of megabytes as compared with only 32 registers). But it, in turn, is still far faster and much less capacious than storage devices and media (e.g., hard disk drives and CDROMs).

Most registers are implemented as an array of SRAM (static random access memory) cells. SRAM is a type of RAM that is much faster and more reliable than the DRAM (dynamic random access memory), which is used for main memory because of its lower cost and smaller space consumption. The term static is employed because SRAM does not need to be electrically refreshed as does DRAM, although it is still volatile

(i.e., it needs to be connected to a power supply in order to retain its contents).

Figure shows a typical processor register using D flip-flop as basic storage element. The field writing logic drives the data input into the register with controlling write enables of the register. Similarly the reading logic reads the data out of the FF using the read enable signals. It drives the data out of FF to the read buffer according to the read enable signals. The registers could be controlled bitwise/bytewise or custom by sharing the read/write logic across the fields. Here, bit 0:9 of the registers are making field 1, and 10:32 bits making field 2.

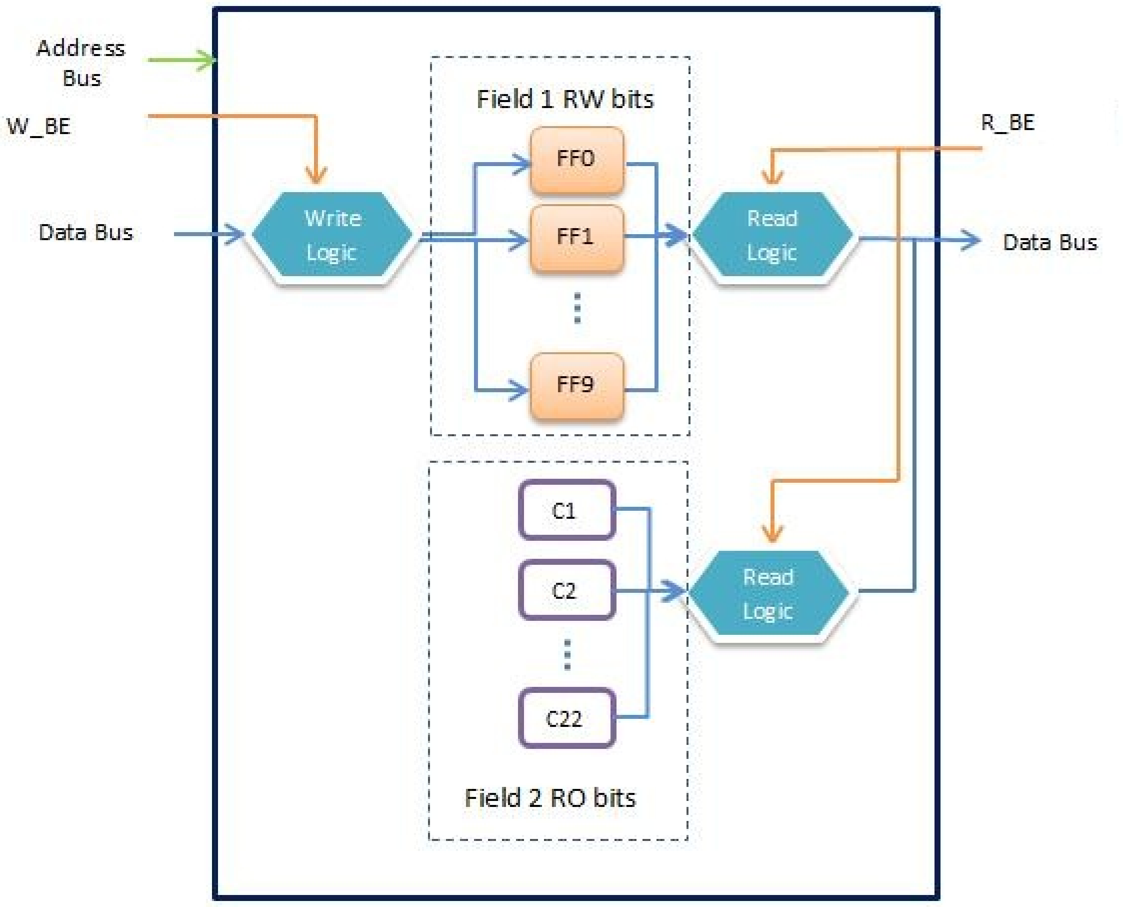


Figure 1.1: Register bitwise achitecture

The access types of the register field could also be controlled based on the presence/absence of the read/write logic. The field 1 having read and write logic connected to the FF shows all FF can be written and read by the source giving access type RW (Read/Write) to the field. The field 2 does not contain any FF as it is having all the constant bits. These bits are constant as they do not have any associated write logic. These bits can be read by read logic. Bits of field 2 is known as RO (Read Only) bits

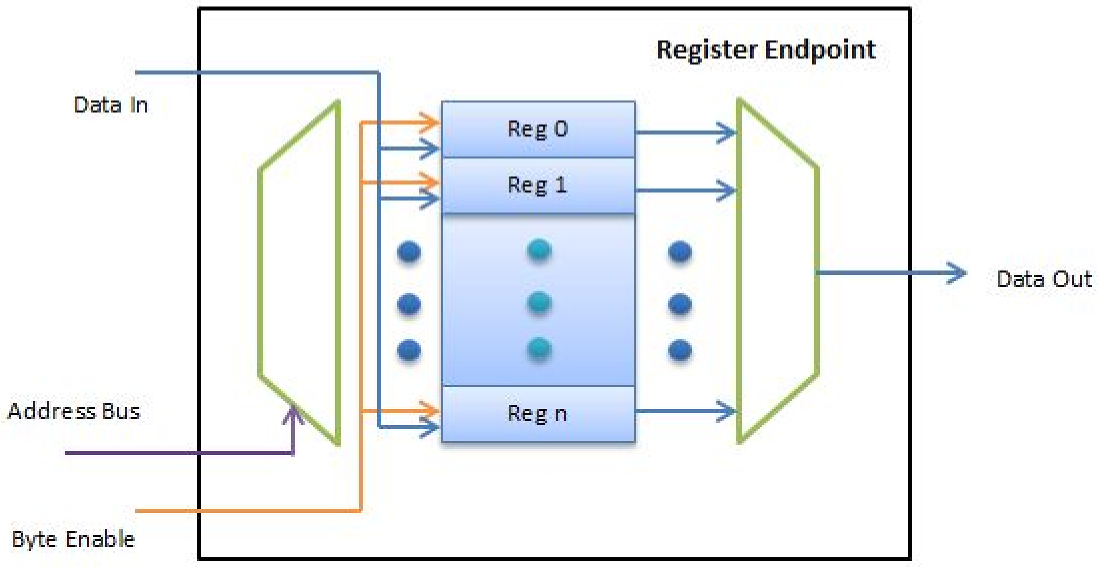


Figure 1.2: Registers inside a endpoint

There can be more than one register in particular endpoint. The Endpoint distribution of the registers is shown in the below figure. Each endpoint has its address to communicate with processor. Register within endpoint will have address as summation of endpoint address and register offset. For each register this address is unique. Register endpoint is having the offset address decoder which will enable the registers only when correct offset address is selected via address bus for correct endpoint. The write and read select lines are the offset address lines decoded to enable particular register of the endpoint. It will make sure that processor is updating/getting correct data at correct place. Output of the register is being read via read multiplexer.

1.2 Types of Processor Register

Architecturally, a 32 bit Intel processor consists of two or more logical processors, each of which has its own architectural state. Each logical processor consists of a full set of 32 bit data registers, segment registers, control registers, debug registers, IO mapped registers etc. This registers are briefly described here.

1. Control Registers (CR)
2. PCI configuration Registers
3. IO and Memory Mapped IO Registers (IO/MMIO)

1.2.1 Control Registers

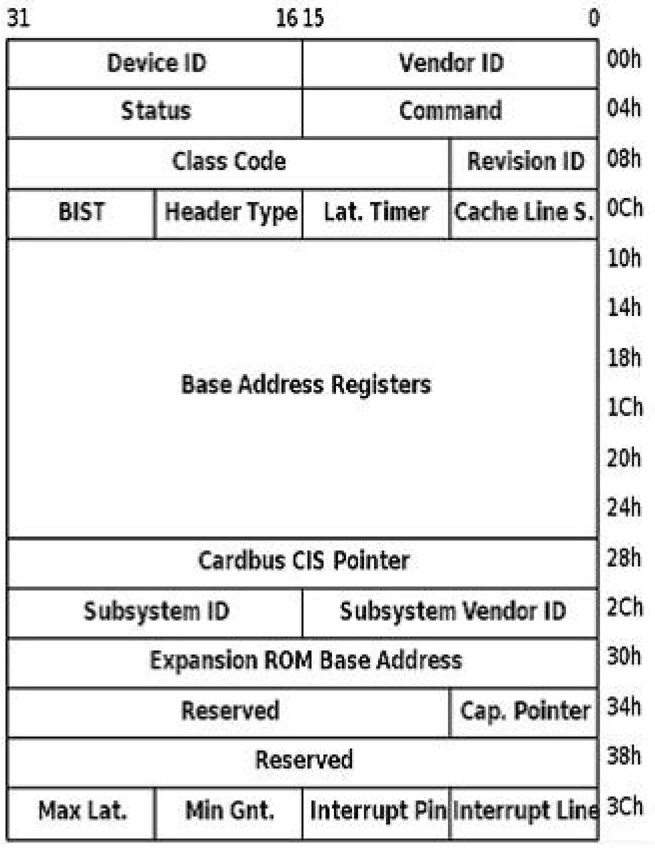
A control register is a processor register which changes or controls the general behavior of a Central Processing Unit (CPU). Common tasks performed by control registers include interrupt control, switching the addressing mode, paging control, and coprocessor control. Control registers are not visible to the software access.

1.2.2 PCI configuration Registers

PCI is a complete specification set that defines how different parts of the computer should interact. PCI is currently used extensively on IA-64 systems. A PCI device driver must be able to find its hardware, gain access to it and initialize it.

PCI configuration space is the underlying way that the Conventional PCI, PCI-X and PCI Express perform auto configure the cards inserted into their bus. One of the major improvements that PCI had over other I/O architectures was its configuration mechanism. In addition to the normal memory-mapped and I/O port spaces, each device function on the bus has a configuration space. This is 256 bytes that are addressable by knowing the 8-bit PCI bus, 5-bit device, and 3-bit function numbers for the device (commonly referred to as the BDF busdevicefunction).

This allows up to 256 buses, each with up to 32 devices, each supporting 8 functions. A single PCI expansion card can respond as a device and must implement at least function zero. The first 64 bytes of configuration space are standardized; the remainder is available for vendor-defined purposes.



31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Cap. Pointer	34h
Reserved				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

Figure 1.3: PCIE Configuration Register

In order to allow more parts of configuration space to be standardized without conflicting with existing uses, there can be a list of capabilities defined within the first 192 bytes of PCI configuration space. Each capability has one byte that describes which capability it is, and one byte to point to the next capability. The number of additional bytes depends on the capability ID. If capabilities are being used, a bit in the Status register is set, and a pointer to the first entry in a linked list of capabilities is provided in the capability pointer register defined in the Standardized Registers.

The Vendor ID and Device ID registers identify the device model, and are commonly called the PCI ID. The 16-bit vendor ID is allocated by the PCI-SIG. The 16-bit device ID is then assigned by the vendor. There is an ongoing project to collect all known Vendor and Device IDs.

The Subsystem Vendor ID and the Subsystem Device ID further identify the device model. The Vendor ID is that of the chip manufacturer, and the Subsystem Vendor ID is that of the card manufacturer. The Subsystem Device ID is assigned by the subsystem vendor, but is assigned from the same number space as the Device ID.

The Status register is used to report which features are supported and whether certain kinds of error have occurred. The Command register contains a bitmask of features that can be individually enabled and disabled.

The Header Type register values determine the different layouts of remaining 48 bytes (64-16) of the header, depending on the function of the device. That is, Type 1 header for Root Complex, switches, and bridges, type 0 for endpoints etc.

1.2.3 IO and Memory Mapped IO Registers

Memory-mapped I/O and Port mapped I/O are two complementary methods of performing input/output between the CPU and peripheral devices in a computer. Memory-mapped I/O uses the same address bus to address both memory and I/O devices the memory and registers of the I/O devices are mapped to address values; So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. Port-mapped I/O often uses a special class of CPU instructions specifically for performing I/O. This is found on Intel microprocessors, with the IN and OUT instructions.

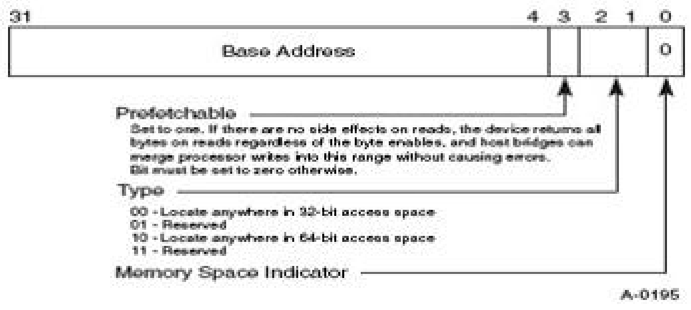

31
4
3
2
1
0
Base Address
0
Prefetchable
Set to one. If there are no side effects on reads, the device returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes into this range without causing errors. Bit must be set to zero otherwise.
Type
00 - Locate anywhere in 32-bit access space
01 - Reserved
10 - Locate anywhere in 64-bit access space
11 - Reserved
Memory Space Indicator
A-0195

Figure 1.4: Memeory mapped Base Register

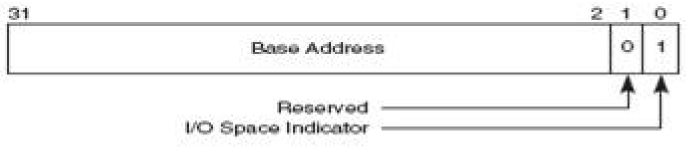


Figure 1.5: IO mapped Base Register

The processor contains two registers that reside in the processor I/O address space the Configuration Address (CONFIG ADDRESS) Register and the Configuration Data (CONFIG DATA) Register. The Configuration Address Register enables/disables the configuration space and determines what portion of configuration space is visible through the Configuration Data window.

For example PCI targets (except host bus bridges) are required to implement Base Address Registers (BARs) to request a range of addresses which can be used to provide access to internal registers of the devices. The configuration SW for the device uses BARs to determine how much space a device requires in a given address space and then assigns where in that space the device will reside.

- BARs that map into IO space are always 32b wide
- BARs that map into Memory space are either 32b or 64b wide

I/O and Memory addresses are supposed to be unique to one device. SW may erroneously configure two devices to the same address, making it impossible to access either one. This will never happen unless a driver is playing with registers it shouldn't touch.

Devices are configured at System Boot time. A powered-up uninitialized device only responds to configuration transactions. It has no memory and no IO ports mapped in the system address space. BIOS offers access to the device configuration address space by reading and writing registers in the PCI controller. At boot time, BIOS performs the necessary configuration transactions with every PCI peripheral in order to allocate place for each memory and/or IO region that it needs. By the time a device driver accesses the device, its Memory and IO regions have already been mapped into the system address space. Driver may change the default assignment if it wants.

SW accesses MMIO space using the MOV instruction. Core sends the request to Memory space to access device. Memory requests may be of any size. It is recommended that a device request that its internal registers be mapped into Memory Space and not I/O Space.

I/O Space is limited and highly fragmented in PC systems and will become more difficult to allocate in the future. A device may map its internal register into both Memory Space and optionally I/O Space by using two Base Address registers (one for I/O and the other for Memory). Both BARs provide access to the same registers internally. SW access IO space using the In/Out instructions.

1.3 Pre-Silicon Verification

Verification is over all process of verifying that the processor conforms to specification. As complex task, it consumes the majority of time and effort in processor design. During the pre-silicon process, engineers test devices in an emulated environment with simulation, emulation, and formal verification tools.

Attribute verification of the system is defined as process of verifying the properties of the RTL which ensures that correct logic is implemented. Logic designers implement processor architecture in RTL. The verification engineers write several properties using the information available in the architecture specification document to ensure that the implementation satisfies the specification. One of the approaches is Logic simulation.

A logic simulation environment is typically composed of several components. Each component and its significance in the register property verification are described with an example register as follows. The example register A of size 32 bits have RW and RO as bit access type.

- Transaction generator: The Transaction generator generates input vectors that are used to search for anomalies that exist between the intent (specifications) and the implementation (HDL Code). Modern generators create directed-random and random stimuli that are statistically driven to verify random parts of the design. The randomness is important to achieve a high distribution over the huge space of the available input stimuli. Generators also bias the stimuli toward design corner cases to further stress the logic. Biasing and randomness serve different goals and there are tradeoffs between them, hence different generators have a different mix of these characteristics. The input for the design must be valid (legal) and many targets (such as biasing) should be maintained. The model-based generators use this model to produce the correct stimuli for the target design. Transaction generator for register verification normally generates directed random data for the bit access type, partial access verification of registers. For register A the data generated by transaction generator is 0xFFFF with full register access.

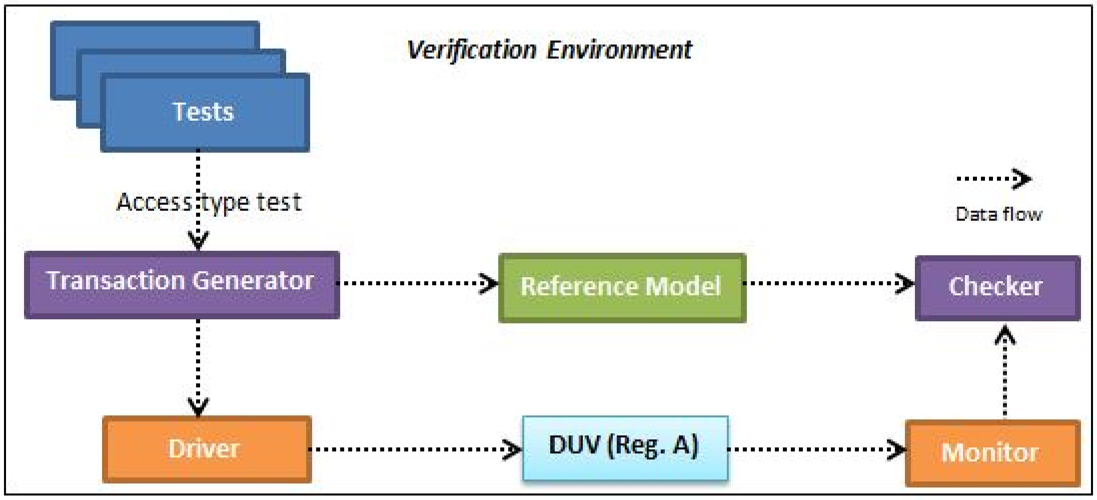


Figure 1.6: Verification Envirnoment for Register Validation

- Driver: The drivers translate the stimuli produced by the generator into the actual inputs for the Design Under Verification (DUV). Generators create inputs at a high level of abstraction, namely, as transactions or assembly language. The drivers convert this input into actual design inputs as defined in the specification of the design's interface. For register verification driver works as the interface between stimuli generator and RTL code. It converts the stimuli to the signals understood by the RTL code of the DUV.

- Simulator: The simulator produces the outputs of the design, based on the design's current state (the state of the flip-flops) and the injected inputs. As shown in the figure, simulator stimulates the register RTL for the stimuli given to the input and generates the output signals according to the register RTL description.

- Monitor: The monitor converts the state of the design and its outputs to a transaction abstraction level so it can be stored in a 'score-boards' database to be checked later on. The outputs of the register stimulus can be checked in two ways i.e. either via reading the register back or via checking the processor states triggered by the register changes. Register attribute validation covers only property, not functional verification; normally monitor is used to capture the registers stored values after some write transaction on the registers.

- Checker: The checker validates that the contents of the 'score-boards' are legal. The checker needs to validate that the actual results match the expected ones. The expected value for the register attribute validation is generated by the register reference model.

- Reference Model: Reference model generates the expected RTL output from the specification. Register Reference Model (RMM) contains all attributes information of the registers under validation. It also keeps track of current & next values of these registers, and implements Expected Value Evaluation Algorithm (discussed later). Upon issuing a transaction to the RTL, Reference Model evaluates the expected values of the register according to the transaction of the driver. Reference model counted the expected read value of 0x003F here as the bits (10:31) is having RO access type with constant value of 0.

For the closure of the verification, coverage of the design is defined to assess that the functional verification is done on design up to correct extinct. These can be functional coverage, statement coverage, and branch coverage or more than one type of coverage.

Chapter 2

Register Attributes

2.1 Basic Attributes

Register serves as an interface between SW and HW for configuration of the chip's functionality. Each register consists of properties (attributes) associated with it, like its size, reset values, access types etc. There would be a significant advantage if these common attributes of the registers are validated together using a common methodology. Configuration Register Validation (CRV) refers to such attributes verification without focusing on functional aspects of the registers.

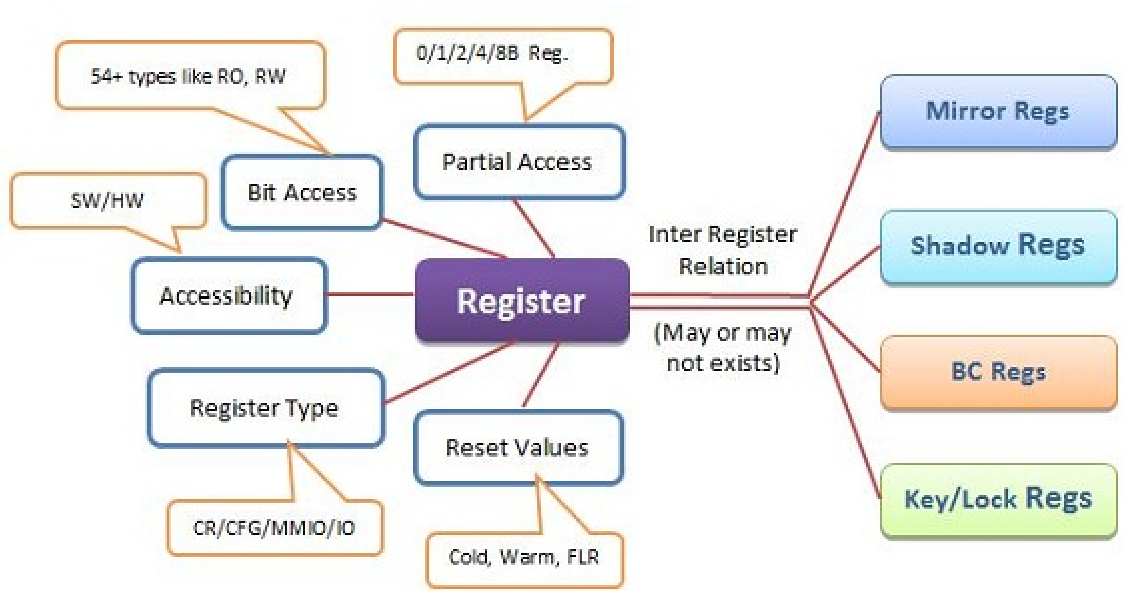


Figure 2.1: Register Attributes

Register attributesproperties can be classified as attributes pertaining to the register itself and inter register relations that represent how a programming to some complex register in a certain way affects other dependent registers. Figure 5.1 represents the important attributes of a register from the CR Validation perspective.

2.1.1 Bit Access type

Bit Access Types of a register refers to individual bit read and write accessibility. In current generation processor there are more than 54 different kinds of access types like RO (read-only), RW (read-write), WO (write-only), RW1C and RW1S etc.

A common type of bit is the readwrite bit. The read andor write ability of the bit is documented from the debug firmware's perspective and software driver perspective. Driver can read the bit to see its value. Driver can change the value of a read/write bit by writing 1's or 0's to the bit. The hardware, however, can only read the bit; it cannot change the value of the bit (except during power-on resets). Common uses of read/write bits include configuring and controlling of hardware block, interrupts enabling, controlling the level of output pins on the chip etc. In all of these uses, driver writes to read/write bits to configure the block's behavior.

Another common type of bit is the read-only bit. This means that debug firmware and software driver can read the bit but cannot change the value of the bit by writing to it. Only the hardware can change the value of the bit. Any attempt by driver to change the value is ignored by the block. Common uses of read-only bits includes showing status of an internal condition such as active, idle, full, empty, or error, storing value of an input pin on the chip, storing results of a timercounter etc.

Write-only bits are rare. It means that debug firmware and software driver can write to it and change its value but driver cannot read back what the value is. Since driver cannot read the register, it cannot confirm that the register contains what it wrote. If driver needs to later on remember the value of what it wrote, it must keep its own copy of what it last wrote. Obviously, the hardware can read it and respond to it appropriately.

Reasons are very few to have write-only bits. Here are two reasons where it might be appropriate: For security purposes, register might be rendered write-only so that rogue driversoftware modules cannot read the settings. The register might not have storage elements behind it. Writing to it causes some task to start, such as a state machine, but there is no need to keep a 1 in an inputoutput. Generally a status register keeps track of the status of the task. Even if a register is deemed write-only, a read should still work and should return zeros.

RW1C bit can be read by software driver and it can write '1' to the bit, which will clear the bit automatically. There will be no change in the value of this bit, if '0' is written. The hardware sets a bit and driversoftware clears it. Driver cannot set this bit, only clear it after the hardware has first set it. RW1C bits are used to inform driver which event has occurred within the block. Driver then acknowledges the event by writing a 1 in that bit position to clear it. This behavior is also known as Write 1 Clear (W1C). This behavior permits more than one different event to occur at different times while allowing firmware to clear one without risk of inadvertently clearing another.

In RW1S bit, the driver writes a one to set that bit in a register. But driversoftware cannot clear the bit. Only the hardware can clear the bit. This is known as Write 1 Set (W1S). RW1S bits are typically used to queue some action in the hardware. When the hardware is done with that task, it then clears the bit. Driver can read the bit, and if it is cleared, it knows that it can queue the action again. Driver can queue any bit at any time, without regard to other bits in the register that may be set or that are about to be cleared by the block. Driver writes a 1 to the desired position and a zero everywhere else. Zeros in the other positions do not affect the settings of the other bits.

The bit access type of registers have different attribute modifiers which changes the behavior of the bit from its original access type with respect to some other register property.

The bit which can be updated by hardware can be identified by "_V" at the end of the normal access type. These bits are called variant bits. For example, if a bit have RW_V access type, the software can read and write the bit in addition to the hardware. RW1C and RW1S bits need to be updated by hardware as their original access type property. They are called semi variant bits as hardware and software can write only particular logic on it. These bits are not required to append with _V attribute as hardware updateability can be implicitly known by original access type.

Other attribute modifier is ”S” at the end of the access type. The S stands for sticky property of bit. The processor goes under different kind of reset like cold and warm reset. Only difference between two resets are that in cold reset the power to the processor is fully removed and the system is restarted again; whereas in warm reset the system is restarted without removing power to the system. During warm reset some of the bits will store their original values in spite of reset. These bits are known as having sticky property and named as sticky bit with S at the end of their original access type (like RWS).

There is a pair of attribute modifier known as _L and _K. They are known as lock and key bits. They are described in coming sections.

2.1.2 Partial Access

Each register bit is connected to its own bit enable signal. Partial Access refers to accessibility (Write & Read) of granularity like 012348 bytes per transaction using Byte Enables (BE). Register can support the partial access according to its functionality and design requirement. If any register supports the partial access that means that each bits of the register can be accessed individually independent of the other bits in the register. Partial access can be given to the register at different granularity levels also.

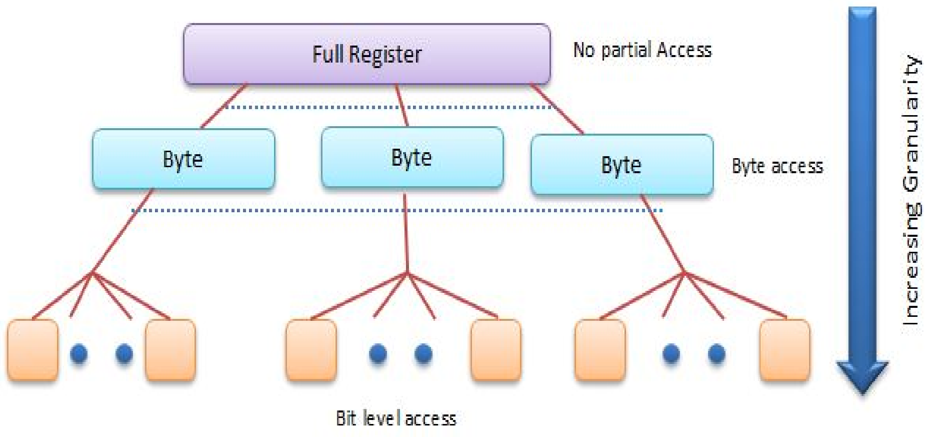


Figure 2.2: Partial Access Granularity

If a register is defined to have partial access at byte level, each byte in the register can be accessed individually, and it is the last level of the partial access the software can have on the register. It can have partial access in upper level partial access like any two byte of register can be accessed in parallels. Concept of granularity for the partial access of register is shown in below fig. If a register does not support partial access, it should be accessed as full register only.

2.1.3 Accessibility

Each register can be accessible via different request initiators. Request initiators can be defined as source of the write or read request to the register. The initiator can be software driver, or debug firmware driver or hardware signal. Request generated by each initiators have different effects on the register bits according to the register visibility to the particular request initiator. Accessibility of a register refers to visibility of the register to Software (SW), Firmware (FW) and Hardware (HW).

Registers which are accessible via software driver are used for the software configuration of particular hardware block. These registers are visible to operating system programmer to have some control over hardware behavior of the processor. All the software visible registers are also visible to the firmware.

All the configuration registers are visible to debug firmware driver. As the firmware is used for processor testing and debug purpose it can access all the registers across the board for the testing purpose.

Hardware visibility of the register is dependent on bit access type of the register and register inter relational attributes. RW1C and RW1S bits of the registers are hardware visible by access types as these bits needs to be updated by hardware.

2.1.4 Reset Values

Reset Values refer to values of register on various resets like cold reset (Hard Reset), Warm Reset and FLR (Function Level Reset). In Coldhard reset power to the system is physically turned off and back on again. Values of the each registers in this phase are known as the cold reset values of the register.

Warm reset of the processor is reset without the removal of power to the system. This kind of reset can preserve the values of the some registers which have sticky attributes. Warm reset will clear all the register data if the register does not contain sticky bit properties. This kind of register will get their cold reset values after the warm reset flow.

When processor needs to reset a particular function, softwarefirmware triggers functional level reset flow. This resets particular list of registers which falls under the function, for which the reset is triggered. The reset will preserve the values of registers which does not fall under the particular function. Functional level reset always get triggered by the softwaredebug firmware, it cannot be triggered by hardware.

2.1.5 Register Type

Register Type refers to whether register is CR (Control Register) or CFG (PCIe Configuration Register) or MMIO or IO registers.

2.1.6 Register Size

Size refers to size of the register in terms of Bytes. Register can be of size 4B (32 bits) or can be of 8B (64 bits) etc.

2.2 Register Dependencies

In Processor architecture it is possible that programming of some of the registers impacts not only the register being programmed but also other set of registers and/or their attributes. This kind of register inter dependencies are known as inter register relations. There are basically four type of inter register dependencies which can be described as follows.

2.2.1 Broadcast (BC) Registers

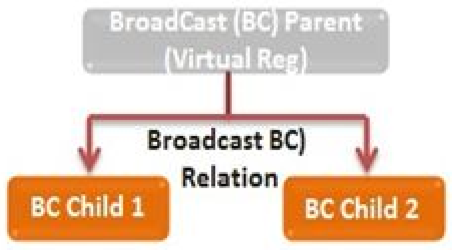


Figure 2.3: Broadcast Relation

If write on the particular register of the processor is broadcasted to other registers in processor, relations between these registers are known as broadcast (BC) relation. The register which is broadcasting writes is known as the BC parent register. Registers which will receive the broadcasted writes and getting updated accordingly are known as BC child.

Broadcast parent registers are normally virtual registers. Virtual registers are defined as the register in programming model of processor but physically they do not exist. Writes on the virtual register is redirected as broadcast to more than one register. These child registers are physical copies of the virtual registers having same attributes as the virtual parent register. Because of the same attributes of the parent and child registers in broadcast it is also known as unidirectional architecture mirroring. A broadcast parent register can have two or more physical copies in the design.

Broadcast parent and child registers are both visible to the software driver. If driver updates any one of the child registers, values will not be propagated to the parent register and other children. It can happen that each of child registers can hold different value than each other. If any writes happen to the parent it will flood through all children, reconfiguring all of them.

2.2.2 Design Shadow Registers:

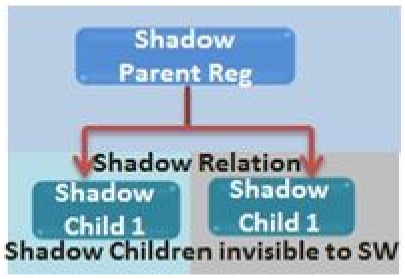


Figure 2.4: Design Shadow Relation

Design shadow registers are special case of broadcast registers where the broadcast children are not visible to the software access. Shadow registers are also known as Non-Architectural Unidirectional Mirroring

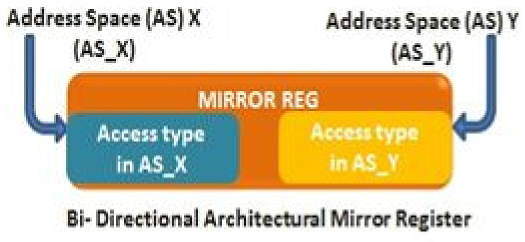

Address Space (AS) X (AS_X)
Address Space (AS) Y (AS_Y)
MIRROR REG
Access type in AS_X
Access type in AS_Y
Bi- Directional Architectural Mirror Register

Figure 2.5: Design Mirror Relation

2.2.3 Design Mirror Registers:

There is multiple ways to define a register in memory as described in IO and memory mapped IO registers. It can happen that two different addresses generated by the processor can map to the same physical address in memory. This relational attribute is known as the design mirror. It can happen that due to two different kind of mapping the access type of register seen to the mapping endpoint is different from both the side. Anyone of the endpoint updating the register will automatically update the value seen by the other endpoint. From the software perspective there will be two different registers which are mapped to the two different memory locations, but actually they will be redirected via internal mechanism to the same register. Figure below shows the Mirror relation of the register

2.2.4 Keylock Registers:

Some of the architecture registers must not be updated under particular processor states and flows. These registers are locked via a key lock mechanism. This lock mechanism is used to control the write access to some of the fields of the same register andor other registers by programming a special field in it called the Key Field (access types RW_K*). The fields that get locked by this key are called Lock Fields (access types RW_L*). Once the key field is locked, the lockable field will not be updated via writes even if the bit access types are writable with respect to request initiator. The Keylock mechanism is to prevent software driver to write certain registers. It is not applicable on debug firmware (i.e. firmware can access the locked bits even if key bit is locked). If the access type is RW_KL, the bit acts as key as well as lock, so, the bit gets self-locked when it is written.

Chapter 3

Configuration Register Validation(CRV) Challenges

CRV validation involves the development of an intelligent verification environment, which can validate all the attributes of the registers residing in the current generation processor. It strives to achieve very good coverage of the design, hence providing enough confidence. The CRV focuses on validation of all the attributes of the register like, size, access type, dependencies, connectivity, without focusing on the functionality aspect of the register.

Developing CRV infrastructure posed several challenges which the validator has to address, which are explained below.

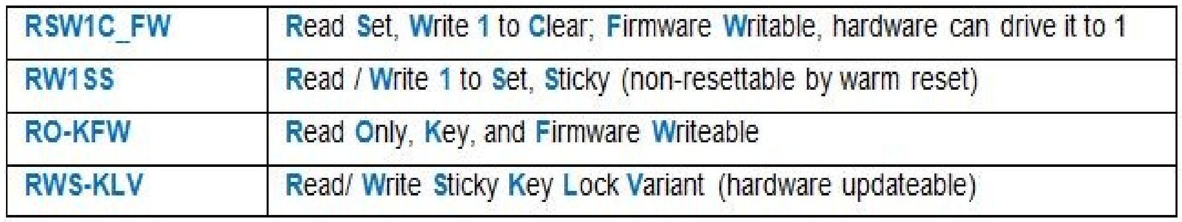
1. Attribute Challeges
2. Validation Environment Challeges
3. Test results and debug
4. Coverage and clouser

3.1 Attribute Challenges

As defined in above section registers are having the different simple and complex attributes. These attributes when combined together increases the complexity of validating the behavior of the register.

3.1.1 Access type challenges

There are 54 different kind of access type in current generation processor. Some of the attributes like RW and RO can be easily validated



RSW1C_FW	Read Set, Write 1 to Clear; Firmware Writable, hardware can drive it to 1
RW1SS	Read / Write 1 to Set, Sticky (non-resettable by warm reset)
RO-KFW	Read Only, Key, and Firmware Writeable
RWS-KLV	Read/ Write Sticky Key Lock Variant (hardware updateable)

Figure 3.1: Complex Access types

using a writeread transaction. But the attribute modifiers increase the complexity of these simple access types. Few of the very complex attributes are given below:

A single register can have multiple access type combinations which poses a challenge. Let us consider an example of a register which contains a mixture of readwrite bits and interrupt bits. (RW1C). Readwrite bits are often handled by software with a read-modify-write sequence. Software sets bit X (RW) by reading the current contents of the register, then writing the contents back out to the register. All the other bits are left the same-but RW1C bit is now set.

RW1C are often handled by firmware with a read-write-Ack sequence. Firmware discovers that bit Z (RW1C) is set, indicating a pending interrupt, by reading the interrupt status register which set the value of bit Z. Firmware Acks that pending interrupt by writing just that bit, to the register.

Combining these two types of bits in the same register requires extra handling by firmware to avoid inadvertent changes of the read/write bits and inadvertent acknowledgment of RW1C bits. The mixture of different attributes in the register would also increase the complexity of the logic used to implement the register. Hence validating such scenarios would be a challenging task.

3.1.2 Partial Access

Based on the functionality each registers fields could be controlled at different granularity level. At the worst case a register could have a facility to control each bit independently. Given a 32 bit register in the processor, the stimulus would have to explore a large validation space.

3.1.3 Accessibility

Register Endpoint can be accessible via software and hardware. Software accessibility is very much easy to validate, but hardware updates are very hard to trace in simulation. This would be challenge since the validation environment would have to continuously monitor the hardware updates of the register based on the access types.

3.1.4 Reset Value

Reset Values checking needs prior programming of register to its complement of reset value and then applying the reset (cold resetwarm reset and FLR etc.). As this prior programming to some of the registers disturbs the configuration of the chip, the reset flow would not go through smooth and blocks the further validation of the register. After each phase of the reset flow, there would be configuration phases which results in register value modification.

3.1.5 Register Inter relations

In many cases a register field can be influenced by changing the field of some other register. Since a register can have different fields being influenced by different registers, these relations become really complex and validating them would be a challenge. Some of the typical relational combinations are Keylock + [Broadcast or Shadow or Mirror], two level Broadcast etc.

- Keylock among two Shadow relations:
  The Keylock registers basically containing key register and lockable registers. RX register is lockable shadow parent register. The register has shadow children which are invisible to software access. Each child has its own lockable bits which are getting locked via child of shadow parent RY. The relations gets complex as each child not having software access and having key bits of locking.

- Shadow child defined also as a mirror register:
  The shadow child cannot be seen to the operating system fabric. The architecture mirroring of the child will be having the software access. The relation gets complex and challenging as the security issues.

- A broadcast child to BC parent also acts like broadcast child to a different BC Parent:
  The broadcast relation get extend to the two level of hierarchy. The virtual

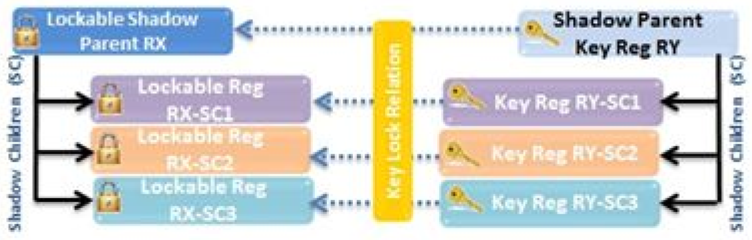


Figure 3.2: Keylock and Shadow relations in same registers

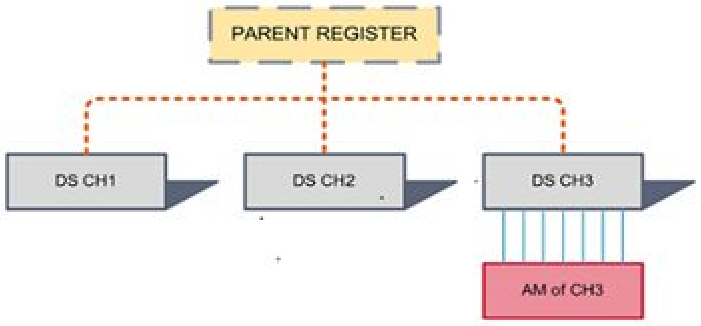


Figure 3.3: Shadow and Mirror relations in same registers

grandparent registers having two hierarchies of children. Middle level register may or may not be virtual.

3.2 Validation Environment Challenges

Developing a validation environment to validate all the attributes of registers is itself a challenging task, as explained in previous section. Along with this the validation environment would pose several other challenges which would need to be tackled.

3.2.1 Legal-illegal stimulus

The configuration registers controls different modes and flows of the processor. CRV would randomize the register values for validation of different attributes, hence triggering lot of unwantedobstructive background noise. These background noises would block the further verification of the register, by causing the processor to hang or changing the state of the

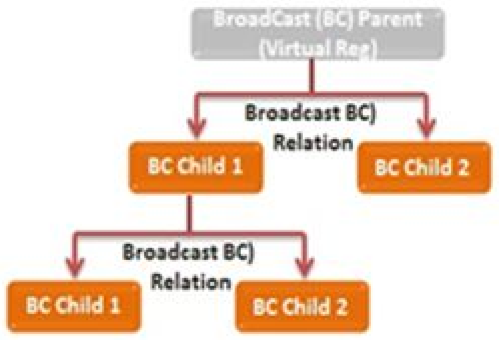


Figure 3.4: Multilevel Broadcast Relations

register such the intended behavior of the register cannot be validated.

3.2.2 Modeling Registers Behavior

Building a Register Reference Model (RMM) which can evaluate expected values for written register as well as all its dependent registers is complex due to complex access types and multiple complex inter register relations pertaining to same register. Hardware has capability to update some of the registers based on functionality of the register. Modeling this variant behavior for the purpose of CRV is complex.

3.2.3 Huge Number of Registers, 10000+ configurations registers

The validation of 10,000 registers in current generation processor would need massive regressions, debug, status reporting, longer turn-around time and overall a large database to handle. It also proportionally increases in Exceptions, Compute Resources and complex environment Handling.

3.2.4 Driver according to accessibility

Each register Endpoint poses different kind of accessibility to the software and hardware request initiators. Either environment needs the different driver for each request initiator or a single driver must be able to mimic all the request initiator. It is also required to be having additional arbiter logic to stimulate the RTL behavior of driver when more than one request initiator generates device handling requests.

3.3 Test Results and debug

The number of registers and complex attribute poses enough challenges in debug and handling such a massive data. A lot of Compute and Engineering Resources would be required for this purpose. To gain enough confidence in validation it is required to run large regressions on the register for each RTL change at system level in small durations. This will lead to iterative and Long Debug turnaround times for such huge number of registers. Since lot of late breaking features might get introduced, which might lead to finding the bugs at the later stage of the design cycle. From CRV perspective bugs which are found at the later stage can lead to RTL changes which might ultimately change the flows of the processor. This might even effect the time to market for the given product. Also there would lot of onion peeling of issues which would increase the complexity.

3.4 Coverage and Closure

There are many registers and each of them contains many attributes. CRV Closure Criteria should be such that all the registers attributes need to be validated completely. Evaluating coverage for individual registers and their attributes is a new challenge. The register regressions strategy should be such that, all the registers should be exercised with all the applicable CRV Ops. An orthogonal approach to CRV Ops coverage is to collect actual toggling of RTL Register Values by the CRV Ops.

Chapter 4

Solution Space

Attributes Validation (CR Validation) can be performed by random programming of registers focusing on validation of various attributes of the registers. The following set of major systematic approaches would help to define CRV strategy for a project.

- Prepare the Test bench suitable for CRV
- Define a strategy to perform complete/comprehensive testing of given set of registers possibly in a single test
- Define Verification process for validating each of the attributes
- Build CRV Testing Infrastructure on top of Existing Test bench to perform the above mentioned comprehensive testing
- Build Registers Reference Model (RMM) for modeling register behavior
- Define Cold and Warm Reset Values verification strategy
- Define a smart coverage based stimulus to validate each of the attributes
- Define tests coding as simple as possible
- Define strategy for handling various exceptions
- Define Coverage strategy & CRV closure criteria

The rest of the chapter discusses details of the above.

4.1 Prepare the Test bench suitable for CRV

CRV involves a lot of random programming of the registers; fundamentally keep toggling the register values. During this flow, there is a significant possibility that the registers attain unsupported/illegal values, and they trigger assertion failures, checker failures, unwanted flows etc. As the idea is to test only the attributes of the registers, the functional effects of the registers need to be ignored. Disabling all the assertions and checkers is the first step in the CRV flow. The CRV test should come out of reset with minimal configuration to enable SW and FW transactions.

4.2 Verify a set of registers in a single test

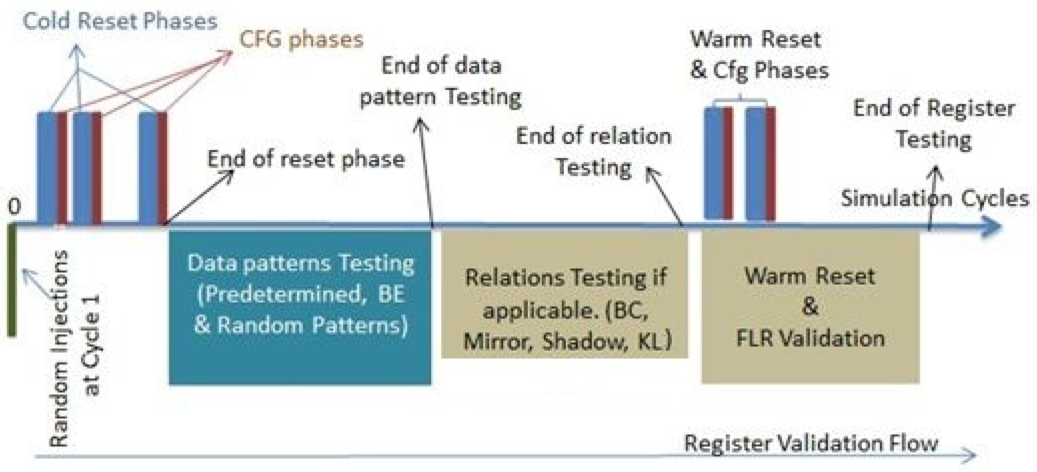

Cold Reset Phases
CFG phases
End of data pattern Testing
Warm Reset & Cfg Phases
End of Register Testing
End of reset phase
End of relation Testing
Simulation Cycles
0
Random Injections at Cycle 1
Data patterns Testing (Predetermined, BE & Random Patterns)
Relations Testing if applicable. (BC, Mirror, Shadow, KL)
Warm Reset & FLR Validation
Register Validation Flow

Figure 4.1: Test Flow for Register Validation

There is a significant advantage of compute resources if a given set of registers are completely validated in a single dynamic simulations test. Figure depicts how the register verification flows in a test.

After the end of the reset phase, the CRV test performs Data Pattern testing (applying various predefined data & BE patterns) to validate most of the basic attributes of the registers. Then the test would perform all the applicable Relations Testing like BC, Mirror, Shadow and KL. Next step is to perform the warm reset testing for sticky bits verification, FLR testing for FLR reset enabled register fields. During the every stage of this verification process, the test keeps dumping pass fail status of the registers against each of its attributes.

4.3 Verification Process for the Attributes

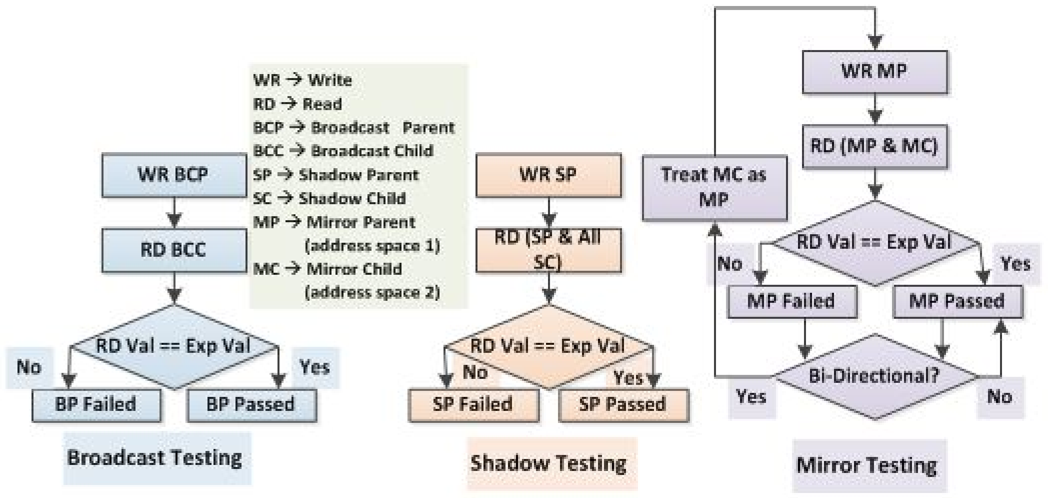


Figure 4.2: Broadcast, Shadow and Mirror testing

All the basic attributes of the registers, excluding some of the complex bit access types, can be validated through Data Patterns testing. A Register Reference Model (RMM, discussed in next sections) needs to be implemented to evaluate expected value based on transaction (ReadWrite) that is applied on a register.

Verification procedure for Broadcast, Shadow and Mirror Relations is shown in the Figure. Keylock testing procedure is shown in Figure. The figures are assumed to be self-explanatory and understanding these verification procedures is left to the readers.

There are some access types like RW1S (write 1 to set, write 0 has no effect, hardware can clear it), RW1C (write 1 to clear, write 0 has no effect, hardware can clear it) etc. which have both SW and HW control on them. Effect of SW writing can be tested easily but emulating hardware behavior to setclear needs functional flow. To emulate hardware behavior in a simplified manner, simulators capabilities can be utilized to force the register bit to the required value and then send SW write to setclear it. Verification of such RW[1—0][C—S]* kind of access types (called RWCS testing) is depicted in the Figure. The success of this RWCS operation is not guaranteed because hardware can modify the register value too quickly that CRV testing fails to catch it.

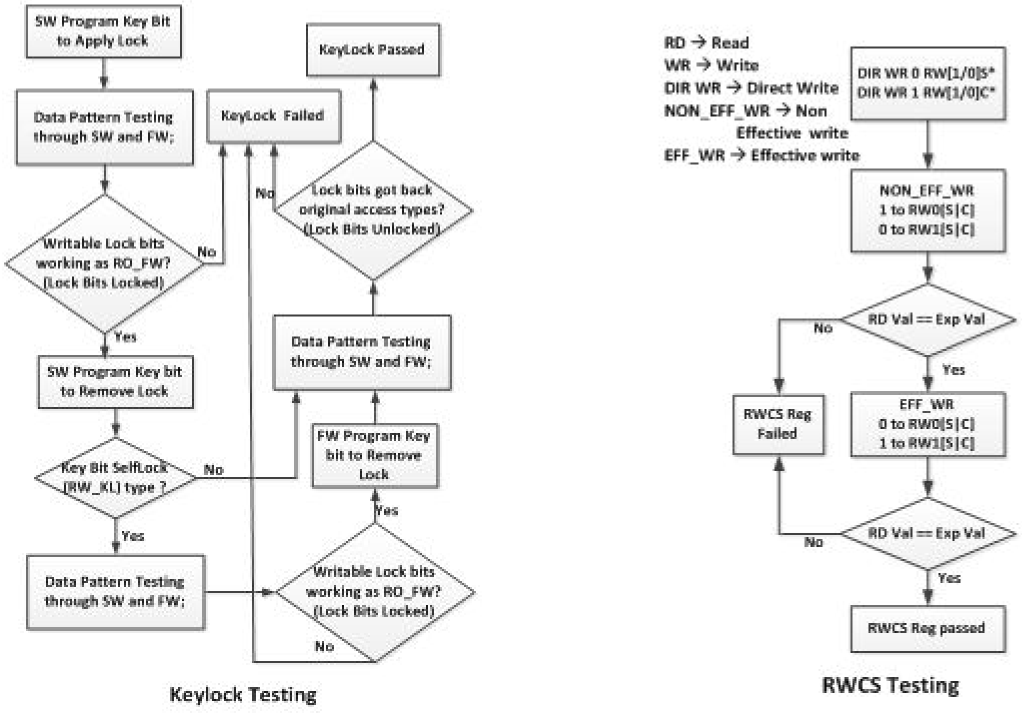

SW Program Key Bit to Apply Lock
Data Pattern Testing through SW and FW;
Writable Lock bits working as RO_FW? (Lock Bits Locked)
No
Yes
SW Program Key bit to Remove Lock
Key Bit SelfLock (RW_KL) type ?
No
Yes
Data Pattern Testing through SW and FW;
KeyLock Failed
KeyLock Passed
Lock bits got back original access types? (Lock Bits Unlocked)
No
Data Pattern Testing through SW and FW;
FW Program Key bit to Remove Lock
Yes
Writable Lock bits working as RO_FW? (Lock Bits Locked)
No
Keylock Testing
RD → Read
WR → Write
DIR WR → Direct Write
NON_EFF_WR → Non Effective write
EFF_WR → Effective write
DIR WR 0 RW[1/0]S*
DIR WR 1 RW[1/0]C*
NON_EFF_WR
1 to RW0[S|C]
0 to RW1[S|C]
RD Val == Exp Val
No
Yes
RWCS Reg Failed
EFF_WR
0 to RW0[S|C]
1 to RW1[S|C]
RD Val == Exp Val
No
Yes
RWCS Reg passed
RWCS Testing

Figure 4.3: Keylock and RWCS testing

4.4 CRV Testing Infrastructure

CR Verification Flow, as depicted in Figure , consists of a. CRV tests, which are generated based on the list of registers in the given model, b. a dynamic simulation environment to perform the register testing. Log files will be analyzed to understand passfail status of the registers, followed by debug.

Figure depicts CRV infrastructure which performs the CRV testing in the dynamic simulations environment. Given the list of Registers Under Validation (RUV) by the CRV test, Attributes Collector gathers all the required list of basic and relational attributes of the registers. Attributes Analyzer analyzes the attributes to determine the set of CRV Operations (CRV Ops) like BC, Shadow, Mirror, KL etc. that need to be performed to completely validate RUV registers. Transactions Generator (TG) generates a set of micro level transactions (RdWr) for each of the CRV Ops. TG also controls the Test bench to initiate various flows like warm reset; FLR etc. Read & Write Logic interacts with Test Bench to perform actual register Read & Write in the specified interface (SW or

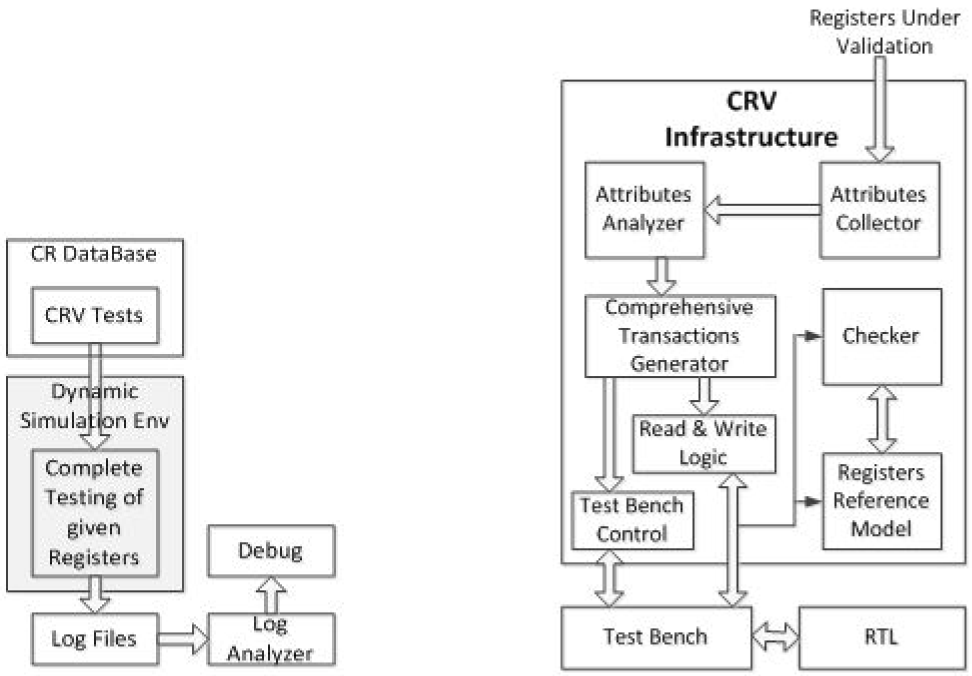


Figure 4.4: CRV Infrastructure

FW). Register Reference Model (RMM) maintains RUV registers and their relation registers expected values (discussed later). Checker performs the comparison of expected values vs actual register value and dumps out checking status in log files.

4.5 Register Reference Model (RMM)

A registers expected value calculation is based on the following data.

- Transaction: Transaction type like ReadWrite, Read BE (Byte Enables), Write Data, Write BE.
- Source (SRC): Request initiator like SWHW.
- Register Base attributes: Access types of individual bits of the register, Registers Partial Accessibility (Byte Enables) etc.
- Inter Relation Register Attributes: Relation with other registers and the relational registers attributes. Transactions may also affect expected value of the relational registers.

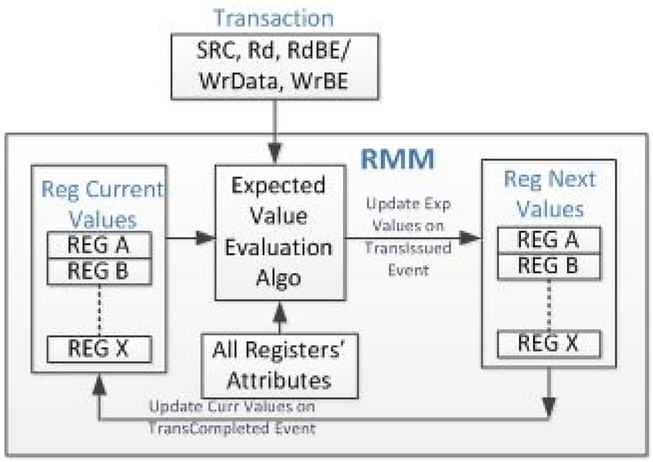

Transaction
SRC, Rd, RdBE/ WrData, WrBE
RMM
Reg Current Values
REG A
REG B
REG X
Expected Value Evaluation Algo
Update Exp Values on TransIssued Event
Reg Next Values
REG A
REG B
REG X
All Registers' Attributes
Update Curr Values on TransCompleted Event

Figure 4.5: Register Refrence Model

RMM contains all attributes information of the registers under validation and their relation registers. It also keeps track of current & next values of these registers, and implements Expected Value Evaluation Algorithm (discussed later). RMM is depicted in Figure.

Some of the bit access types in the lock category (RW_L etc.) acts as RO during Keylock testing. To handle such scenarios, RMM keeps a copy of the bit access types (called local access types) and keeps modifying them dynamically to reflect expected behavior of the register.

Upon issuing a transaction to the RTL, RMM evaluates the expected values of the register and its relation registers based on the local access types; on completion of the transaction, it copies the next values to the current value. This process is required because Read also impacts the register contents (example: RC*(Read Clear), RS*(Read Set) access types) and Actual Read Return Data (RTL data) needs to be compared against the Register Current Values, because next value holds the future value of the register upon completion of the transaction.

4.6 Expected Value Evaluation Algorithm

Expected Value Evaluation is based on the bit access types (rather local access types), Read or WrData & WrBE, SRC of the transaction (FWSW).

All the bit access types can be classified into the major categories which simplifies the algorithm implementation;

- WRITE_TRANSPARENT_ACC_TYPES :Access types that causes WrData get written to register bit independent of SRC (ex: RW kind of bits).
- SET(CLR)ON_SwWr1_ACC_TYPES :Access types that turn the bit to 1(0) if SW writes 1. (ex: RW1S, RW1C kind of bits)

Based on (a) Read or Write transaction (b) Write Data & BE value (c) SRC of the transaction, the algorithm is to scan through each of the bits access types and find out the suitable category for the bit; and there by determine the expected value which could be WrData 1 0. For example, if the bit access type belongs to SETON_SwWr1_ACC_TYPES, transaction is write transaction, BE enabled the current bit for the write and WrData bit is 1, then the expected value of the bit is 0.

4.7 Cold Reset and Warm Reset Verification

In the initial cycles of the simulation, prior to the arrival of hard reset, initialize the registers to complement of the reset values Zeros Ones Random Values. After the end of the reset phases, the values of the registers can be compared against expected reset values. This process works for most of the registers but not for the registers which undergo the configuration during the reset phases. Avoiding such configuration is not possible because the configuration is required for healthy bring up of the simulation environment. If we can identify the cluster endpoint reset signal, then, process of toggling of the reset and then checking the reset values significantly simplifies the process of both Cold and Warm Reset verification. In the environments where such a mechanism is not convenient, the following steps can be applied.(a) Collecting the configured registers and simulation time stamps when they get configured (b) Utilize the DIRECT READ mechanism (explained in next sections) to sample the register values prior to the configuration.

4.8 Stimulus Selection

Transactions Generator (TG) is not only needed to select CRV Ops that need to apply to the registers, but also actual write data for individual micro level transactions. Stimulus selection should be such that minimum number of transactions is able to complete each CRV Op. So,

instead of random data patterns, predetermined patterns need to be contextually selected like ALL_ONE (0xFFFF), ALL_ZERO (0x0000), MOVING_ZERO (0xEDB7_DEBD_DBED_BED7), alternative ones (0x5555, 0xAAAA), double ones (0x36C6_3C63), triple ones (0xE7E7_E77E) etc. For Byte Enable Testing, only consecutive bytes selection is permitted in the processor projects. So, ALL_BYTES (0xF, 0xFF), TRI_BYTES (0x7, 0xE), DOUBLE_BYTES (0x3, 0x6, 0xC), SINGLE_BYTE (0x1, 0x2, 0x4, 0x8) and ZERO_LENGTH (0x0 no bytes transaction) are the possible values for BE testing.

4.9 Tests Simplification and Exceptions Handling

Tests need to be as simplified as possible so that, even non-CRV person should be able to write a test for list of registers that needs to be validated. This enables the register owner to run a simple test to find bugs in the register code that is just written, thereby avoiding bug propagation. This calls for a lot of automation of CRV Infrastructure which is depicted in Figure. However there is a contradictory requirement for some of the Registers as described below.

It is ideal to have all registers be able to sustain any illegal configuration, which is not possible with all the registers. Although attributes of a register requires certain CRV Ops to perform, due to various reasons, some CRV Ops and/or some specific values need to be excluded. Some registers have definitions and implementations different from classic definitions of register attributes. These registers need special attention. Any register that needs such special handling whileprior to applying arbitrary programming can be called as an Exception. For handling exceptional registers, the CRV infrastructure needs to be flexible and provide complete control to the test, so that, it can be tuned, modified according to the requirements of exception. CRV Infrastructure should provide knobs (control parameters, command line switches) to do the same, so that, the CRV owner would be able to write complex tests for handling exceptional cases by reusing the components of CRV infrastructure with ease.

A typical CRV test can be as simple as one liner as shown below. Add_reg_for_crv(Reg1); To completely validate a single register Add_regs_for_crv(Regs_l); To completely validate a list of registers

All the above exception data needs to be communicated from Registers Owners to CR Val owner on preferably proactive basis. In-spite of that, some of the registers validation may not be possible by CR Val owner. In such situations, Automation scripts run these registers in a debug mode and pass the debug ownership to Register OwnerCTE owner.

4.10 Coverage and CRV Closure Criteria

There are many registers and each of them contains many attributes. CRV Closure Criteria should be such that all the registers attributes need to be validated completely. Evaluating coverage for individual registers and their attributes is cumbersome task.

Having well defined verification flow for individual CRV attributes using CRV Ops, coverage of the CRV Ops gives good indication of stimulus applied on the registers. The register regressions strategy should be such that, all the registers should be exercised with all the applicable CRV Ops.

An orthogonal approach to CRV Ops coverage is to collect actual toggling of RTL Register Values by the CRV Ops. This can be computed through post processing of the simulation dump file that tracks the register values. The CRV Ops Coverage, CR Toggle Coverage together with Register Exceptions provides simple but very good indication of coverage.

Chapter 5

Debug and Automation

5.1 Test debug

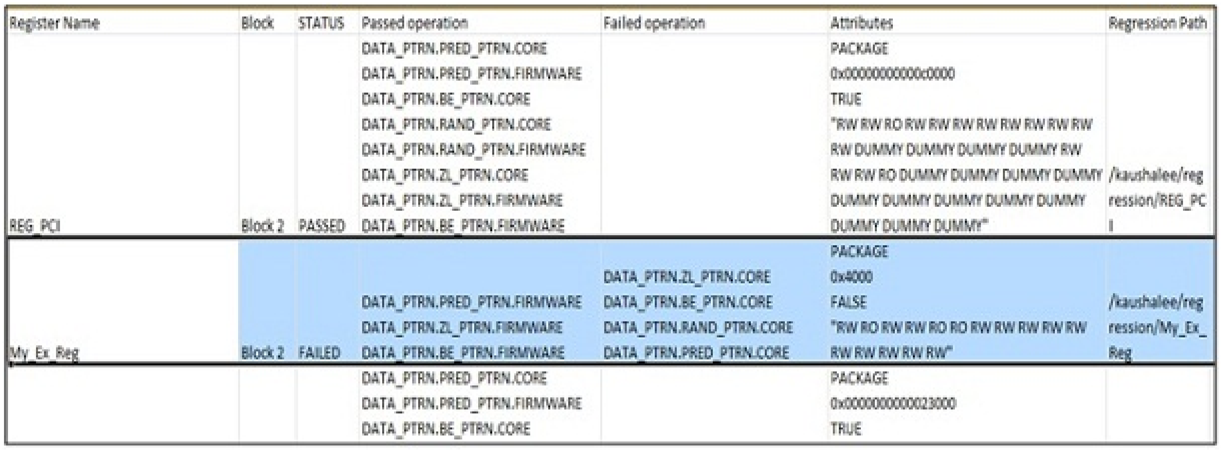
To understand full debug flow, let us take an example of a configuration register called ”My Ex Reg”. Specification:

- Name of register: My Ex Reg
- Size: 16 bit wide
- Access types: [RW, RO, RW, RW, RO, RO, RW, RW, RW, RW, RW, RW, RW, RW, RW, RW]
- Default value: 0x4000

For Validation of the register we initially need to launch test on the register. Once the test completes we need to follow bellow steps to know whether register is working properly or not.

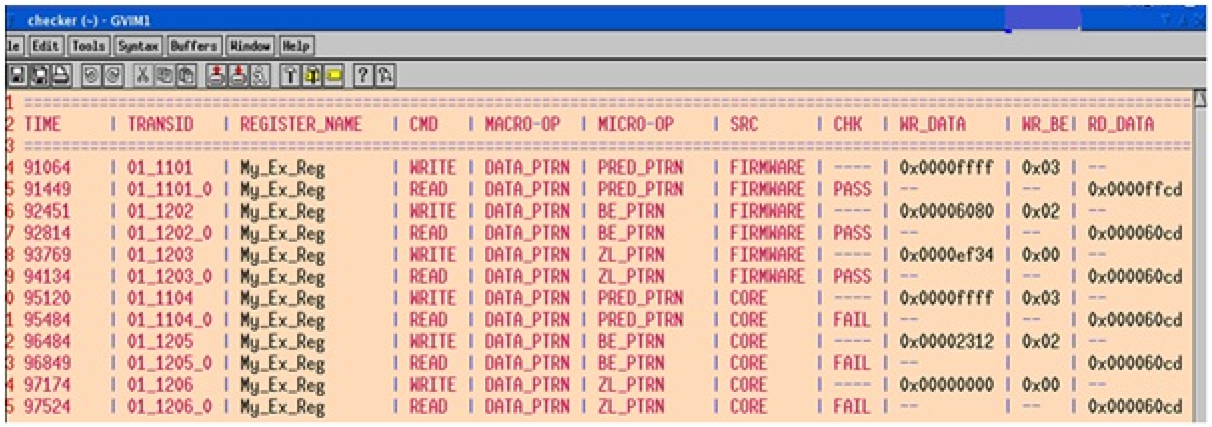
1. Step 1: Check the status of the register in Status file: As the test completes, status of register (Pass/Fail) is generated by CRV owner in .XLS (Excel sheet) format. The process of grabbing status from the test checker is done via automations in Perl. Validator needs to look into the status file from the .XLS file. For our example register below is snapshot of the Excel sheet showing the register test status. As shown in snapshot our register has failed the validation test.

2. Step 2: Check values in the checker: From above snapshot it is clear that the second test has failed. We need to debug the failure from bottom up. As shown below we will start by checking of the values and status at the checker. At checker level we use self-checking mechanism to check if the current operation has passed or failed. For that we calculate the expected values in reference Model according to the input we have given to the DUT via driver. For



Register Name	Block	STATUS	Passed operation	Failed operation	Attributes	Regression Path
REG_PCI	Block 2	PASSED	DATA_PTRN.PRED_PTRN.CORE DATA_PTRN.PRED_PTRN.FIRMWARE DATA_PTRN.BE_PTRN.CORE DATA_PTRN.RAND_PTRN.CORE DATA_PTRN.RAND_PTRN.FIRMWARE DATA_PTRN.ZL_PTRN.CORE DATA_PTRN.ZL_PTRN.FIRMWARE DATA_PTRN.BE_PTRN.FIRMWARE		PACKAGE 0x00000000000c0000 TRUE "RW RW RO RW RW RW RW RW RW RW RW RW DUMMY DUMMY DUMMY DUMMY RW RW RW RO DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY DUMMY"	/kaushalee/reg ression/REG_PC I
My_Ex_Reg	Block 2	FAILED	DATA_PTRN.PRED_PTRN.FIRMWARE DATA_PTRN.ZL_PTRN.FIRMWARE DATA_PTRN.BE_PTRN.FIRMWARE	DATA_PTRN.ZL_PTRN.CORE DATA_PTRN.BE_PTRN.CORE DATA_PTRN.RAND_PTRN.CORE DATA_PTRN.PRED_PTRN.CORE	PACKAGE 0x4000 FALSE "RW RO RW RW RO RO RW RW RW RW RW RW RW RW RW RW"	/kaushalee/reg ression/My_Ex_ Reg
			DATA_PTRN.PRED_PTRN.CORE DATA_PTRN.PRED_PTRN.FIRMWARE DATA_PTRN.BE_PTRN.CORE		PACKAGE 0x0000000000023000 TRUE	

Figure 5.1: Status.xls file



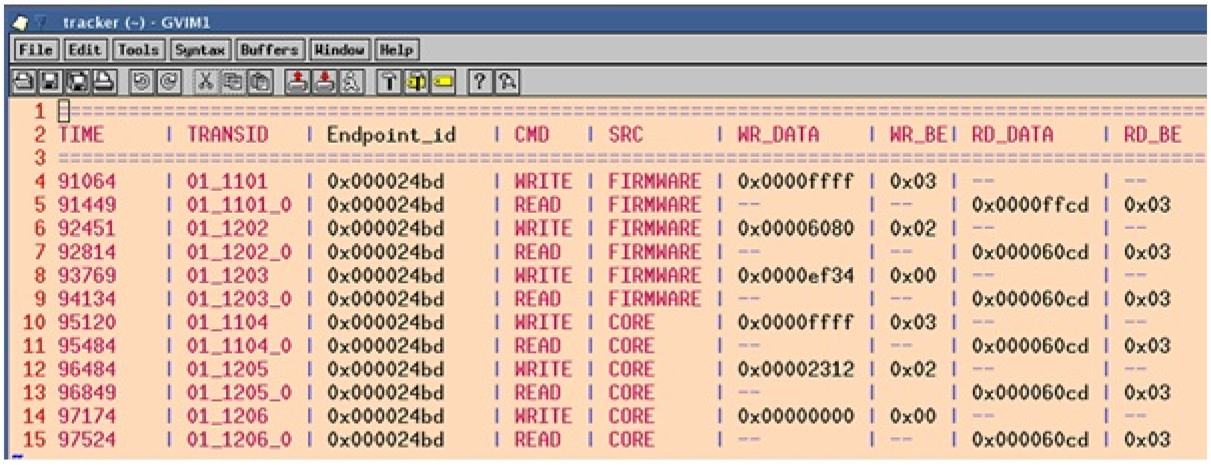
```
================================================================================================================================
TIME      | TRANSID   | REGISTER_NAME   | CMD   | MACRO-OP  | MICRO-OP  | SRC      | CHK  | WR_DATA    | WR_BE| RD_DATA
================================================================================================================================
91064     | 01_1101   | My_Ex_Reg       | WRITE | DATA_PTRN | PRED_PTRN | FIRMWARE | ---- | 0x0000ffff | 0x03 | --
91449     | 01_1101_0 | My_Ex_Reg       | READ  | DATA_PTRN | PRED_PTRN | FIRMWARE | PASS | --         | --   | 0x0000ffcd
92451     | 01_1202   | My_Ex_Reg       | WRITE | DATA_PTRN | BE_PTRN   | FIRMWARE | ---- | 0x00006080 | 0x02 | --
92814     | 01_1202_0 | My_Ex_Reg       | READ  | DATA_PTRN | BE_PTRN   | FIRMWARE | PASS | --         | --   | 0x000060cd
93769     | 01_1203   | My_Ex_Reg       | WRITE | DATA_PTRN | ZL_PTRN   | FIRMWARE | ---- | 0x0000ef34 | 0x00 | --
94134     | 01_1203_0 | My_Ex_Reg       | READ  | DATA_PTRN | ZL_PTRN   | FIRMWARE | PASS | --         | --   | 0x000060cd
95120     | 01_1104   | My_Ex_Reg       | WRITE | DATA_PTRN | PRED_PTRN | CORE     | ---- | 0x0000ffff | 0x03 | --
95484     | 01_1104_0 | My_Ex_Reg       | READ  | DATA_PTRN | PRED_PTRN | CORE     | FAIL | --         | --   | 0x000060cd
96484     | 01_1205   | My_Ex_Reg       | WRITE | DATA_PTRN | BE_PTRN   | CORE     | ---- | 0x00002312 | 0x02 | --
96849     | 01_1205_0 | My_Ex_Reg       | READ  | DATA_PTRN | BE_PTRN   | CORE     | FAIL | --         |      | 0x000060cd
97174     | 01_1206   | My_Ex_Reg       | WRITE | DATA_PTRN | ZL_PTRN   | CORE     | ---- | 0x00000000 | 0x00 | --
97524     | 01_1206_0 | My_Ex_Reg       | READ  | DATA_PTRN | ZL_PTRN   | CORE     | FAIL | --         | --   | 0x000060cd
```

Figure 5.2: Checker file

example snapshot of the checker file for our example register can be given as follow:

3. Step 3: Check the captured values in Tracker file: Tracker file actually tracks the input and output of the DUT (here registers). We need to proceed with the debug using the tracker. It contains time stamps and the transaction ids of the transactions for particular endpoint ids. Snapshot of the tracker: The transactions are traced using the time stamps. The register endpoint id would not be needed at this point. But it could be used to find the register written values if more than one registers are running in parallel in given test

4. Step 4: Check logger files Logger files logs all the operations done during the test. After the test environment comes up it will make the log of all the writes



```
==========================================================================================================
TIME      | TRANSID   | Endpoint_id | CMD   | SRC      | WR_DATA    | WR_BE| RD_DATA    | RD_BE
==========================================================================================================
91064     | 01_1101   | 0x000024bd  | WRITE | FIRMWARE | 0x0000ffff | 0x03 | --         | --
91449     | 01_1101_0 | 0x000024bd  | READ  | FIRMWARE | --         | --   | 0x0000ffcd | 0x03
92451     | 01_1202   | 0x000024bd  | WRITE | FIRMWARE | 0x00006080 | 0x02 | --         | --
92814     | 01_1202_0 | 0x000024bd  | READ  | FIRMWARE | --         | --   | 0x000060cd | 0x03
93769     | 01_1203   | 0x000024bd  | WRITE | FIRMWARE | 0x0000ef34 | 0x00 | --         | --
94134     | 01_1203_0 | 0x000024bd  | READ  | FIRMWARE | --         | --   | 0x000060cd | 0x03
95120     | 01_1104   | 0x000024bd  | WRITE | CORE     | 0x0000ffff | 0x03 | --         | --
95484     | 01_1104_0 | 0x000024bd  | READ  | CORE     | --         | --   | 0x000060cd | 0x03
96484     | 01_1205   | 0x000024bd  | WRITE | CORE     | 0x00002312 | 0x02 | --         | --
96849     | 01_1205_0 | 0x000024bd  | READ  | CORE     | --         |      | 0x000060cd | 0x03
97174     | 01_1206   | 0x000024bd  | WRITE | CORE     | 0x00000000 | 0x00 | --         | --
97524     | 01_1206_0 | 0x000024bd  | READ  | CORE     | --         | --   | 0x000060cd | 0x03
```

Figure 5.3: Tracker file

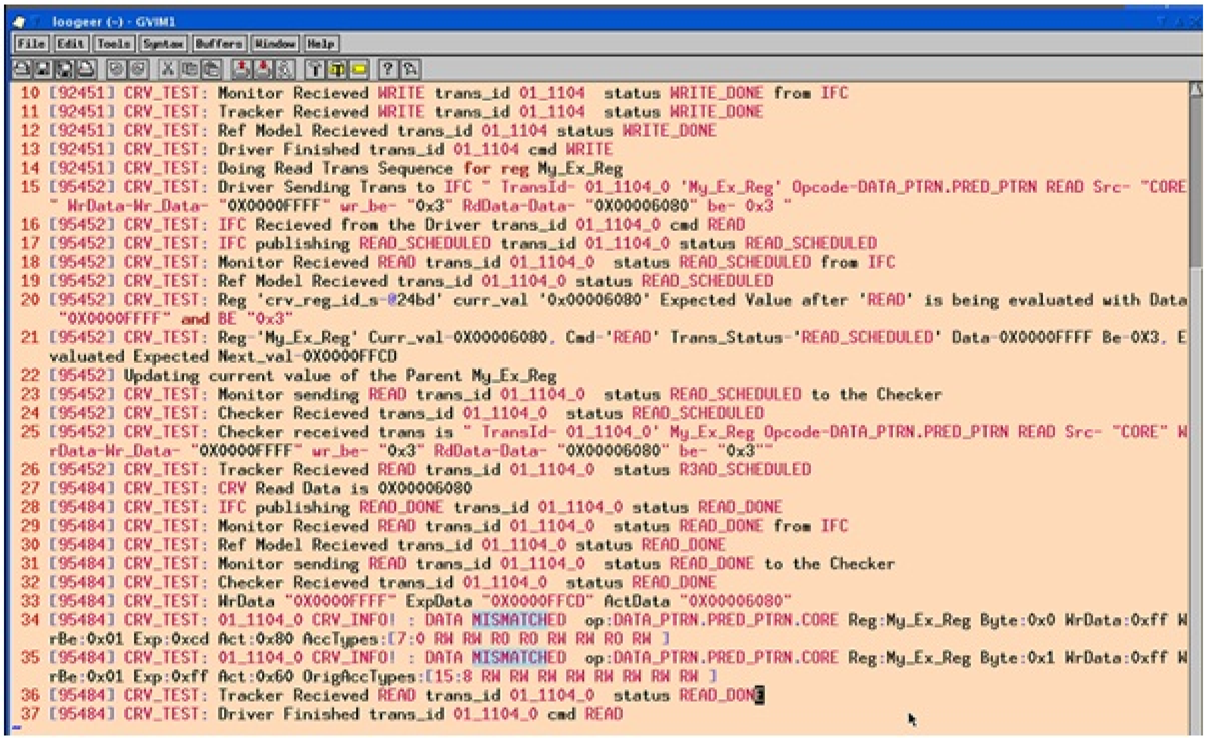
followed by read in the current test. For our register we need to make sure that during the validation process, our test environment is working properly. Logger file will also give us the access types and other attributes of the register. Snapshot of the logger files: In logger file we can see that checker has flagged the mismatch of data between expected and actual value of the register. As shown in the snapshot expected values are according to the access types which is compared with actual value of the register. To see the values at the register endpoint in the RTL the debug needs to be taken to next level using Waveform Debug.

5. Step 5: Waveform Debug We can get waveform of the register endpoint in simulation tool and see whether the value reflected over logger and tracker files are same as that of end point or not. As shown in the figure, the value of the register is constant after some cycles and is not getting updated to the required value.

6. Conclusion: According to waveforms and the trackers we can see that there can be different possible bugs in the design for the registers. They are: (a) Register transaction is not correctly decoded/routed (b) Register transaction is obstructed by internal flow/Prior Programming (c) Register decoding logic in the End Point is incorrect (d) One of the inputs may be illegal (e) May be prior configuration of register is required

5.2 Automation

As the number and complexities of registers increases, process automation for verification becomes inevitable. Test generation, test status



```
[92451] CRV_TEST: Monitor Recieved WRITE trans_id 01_1104  status WRITE_DONE from IFC
[92451] CRV_TEST: Tracker Recieved WRITE trans_id 01_1104  status WRITE_DONE
[92451] CRV_TEST: Ref Model Recieved trans_id 01_1104 status WRITE_DONE
[92451] CRV_TEST: Driver Finished trans_id 01_1104 cmd WRITE
[92451] CRV_TEST: Doing Read Trans Sequence for reg My_Ex_Reg
[95452] CRV_TEST: Driver Sending Trans to IFC " TransId- 01_1104_0 'My_Ex_Reg' Opcode-DATA_PTRN.PRED_PTRN READ Src- "CORE
" WrData-Wr_Data- "0X0000FFFF" wr_be- "0x3" RdData-Data- "0X00006080" be- 0x3 "
[95452] CRV_TEST: IFC Recieved from the Driver trans_id 01_1104_0 cmd READ
[95452] CRV_TEST: IFC publishing READ_SCHEDULED trans_id 01_1104_0 status READ_SCHEDULED
[95452] CRV_TEST: Monitor Recieved READ trans_id 01_1104_0  status READ_SCHEDULED from IFC
[95452] CRV_TEST: Ref Model Recieved trans_id 01_1104_0 status READ_SCHEDULED
[95452] CRV_TEST: Reg 'crv_reg_id_s-@24bd' curr_val '0x00006080' Expected Value after 'READ' is being evaluated with Data
 "0X0000FFFF" and BE "0x3"
[95452] CRV_TEST: Reg-'My_Ex_Reg' Curr_val-0X00006080, Cmd-'READ' Trans_Status-'READ_SCHEDULED' Data-0X0000FFFF Be-0X3, E
valuated Expected Next_val-0X0000FFCD
[95452] Updating current value of the Parent My_Ex_Reg
[95452] CRV_TEST: Monitor sending READ trans_id 01_1104_0  status READ_SCHEDULED to the Checker
[95452] CRV_TEST: Checker Recieved trans_id 01_1104_0  status READ_SCHEDULED
[95452] CRV_TEST: Checker received trans is " TransId- 01_1104_0' My_Ex_Reg Opcode-DATA_PTRN.PRED_PTRN READ Src- "CORE" W
rData-Wr_Data- "0X0000FFFF" wr_be- "0x3" RdData-Data- "0X00006080" be- "0x3""
[95452] CRV_TEST: Tracker Recieved READ trans_id 01_1104_0  status R3AD_SCHEDULED
[95484] CRV_TEST: CRV Read Data is 0X00006080
[95484] CRV_TEST: IFC publishing READ_DONE trans_id 01_1104_0 status READ_DONE
[95484] CRV_TEST: Monitor Recieved READ trans_id 01_1104_0  status READ_DONE from IFC
[95484] CRV_TEST: Ref Model Recieved trans_id 01_1104_0 status READ_DONE
[95484] CRV_TEST: Monitor sending READ trans_id 01_1104_0  status READ_DONE to the Checker
[95484] CRV_TEST: Checker Recieved trans_id 01_1104_0  status READ_DONE
[95484] CRV_TEST: WrData "0X0000FFFF" ExpData "0X0000FFCD" ActData "0X00006080"
[95484] CRV_TEST: 01_1104_0 CRV_INFO! : DATA MISMATCHED  op:DATA_PTRN.PRED_PTRN.CORE Reg:My_Ex_Reg Byte:0x0 WrData:0xff W
rBe:0x01 Exp:0xcd Act:0x80 AccTypes:[7:0 RW RW RO RO RW RW RO RW ]
[95484] CRV_TEST: 01_1104_0 CRV_INFO! : DATA MISMATCHED  op:DATA_PTRN.PRED_PTRN.CORE Reg:My_Ex_Reg Byte:0x1 WrData:0xff W
rBe:0x01 Exp:0xff Act:0x60 OrigAccTypes:[15:8 RW RW RW RW RW RW RW RW ]
[95484] CRV_TEST: Tracker Recieved READ trans_id 01_1104_0  status READ_DONE
[95484] CRV_TEST: Driver Finished trans_id 01_1104_0 cmd READ
```

Figure 5.4: Logger file

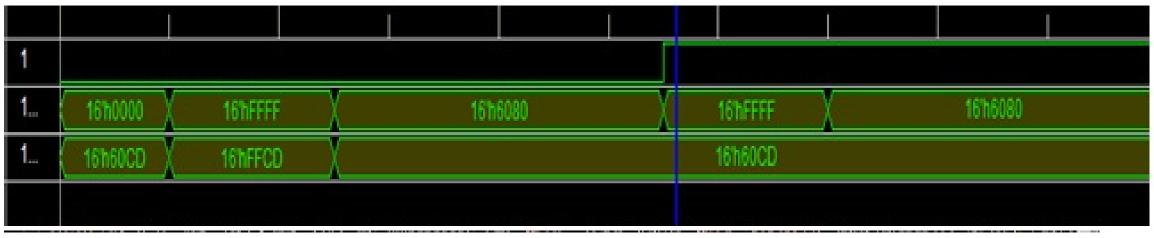


Figure 5.5: Waveform

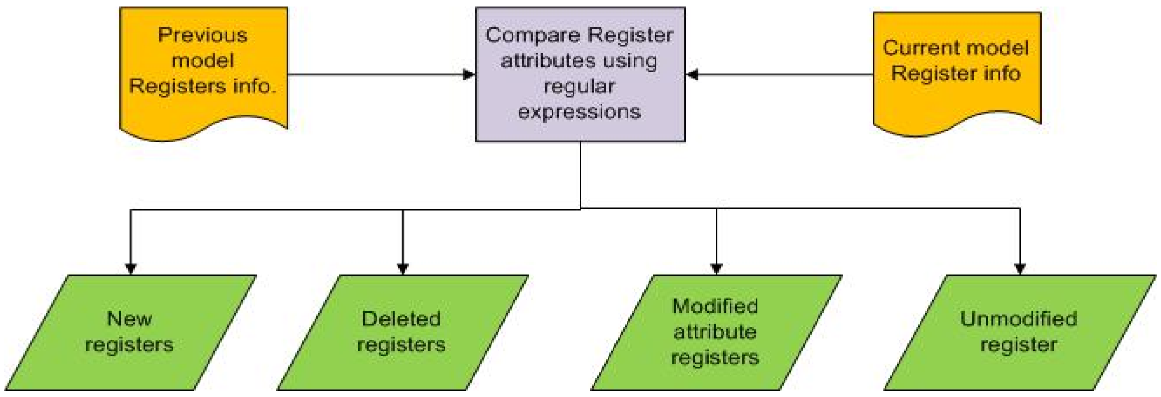

Previous model Registers info.
Compare Register attributes using regular expressions
Current model Register info
New registers
Deleted registers
Modified attribute registers
Unmodified register

Figure 5.6: Flow for register attribute comparison

counting and data maintenance tasks, require very efficient and fast automation wrapper around the verification environment. This automation reduces lots of manual effort and enhances the speed of verification process. For register validation we have used PERL automation. In the following chapter, we described some major automation we have done for CRV.

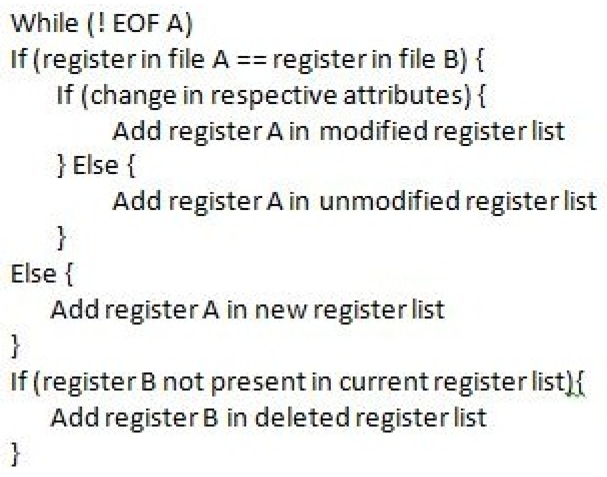
5.2.1 Test Generation Automation

Due to automation CRV test is made simple. Each of the tests is written to run 20-50 registers based on complexity of the registers. The registers which have multiple relations consume more number of simulation cycles, so few such registers are put in such test.

A script is developed to compare register attributes across two RTL models (previous model vs current model) to produce (a) New registers (b) Modified registers (c) Deleted registers (d) Unchanged registers. New registers and Modified registers together is called Delta Registers.

Tests Generation script is developed, which provides knobs to control (a) number of registers per test, (b) tests generation based on CRV Ops (like Broadcast tests, KL tests etc.), (c) tests generation based on clusters and (d) delta tests. Register attribute comparison

As shown in flowchart below the comparison take place with the help of the regular expression. The input to the script is register info files from two models.



```
While (! EOF A)
If (register in file A == register in file B) {
    If (change in respective attributes) {
        Add register A in modified register list
    } Else {
        Add register A in unmodified register list
    }
Else {
    Add register A in new register list
}
If (register B not present in current register list){
    Add register B in deleted register list
}
```

Figure 5.7: pseudo code for attribute comparison

Which produces the output containing following details.

- New registers: Registers which are newly added to current model and have not been introduced in RTL until now.
- Deleted registers: Registers which have been removed from the RTL.
- Modified attributes Registers: The registers having change in their attribute values starting from the change in access types or reset values to the complex modification of register relations. The file also shows the attribute values which have been changed after comparing the previous and current values.
- Unmodified registers: The registers having same attributes in both the model.

Pseudo code is shown below:

Script will give a list from which delta regressions are generated and debugged. Delta Registers regressions and their debug are prioritized. Simple test generation

Test generation script simply generates the test case according to the user requirement. It can control the number of registers per tests, operation on the registers, registers request initiators, simulation waveform generation etc.

At very basic level we can understand it as following flow chart:

Different components of the flowcart can be explained as below:

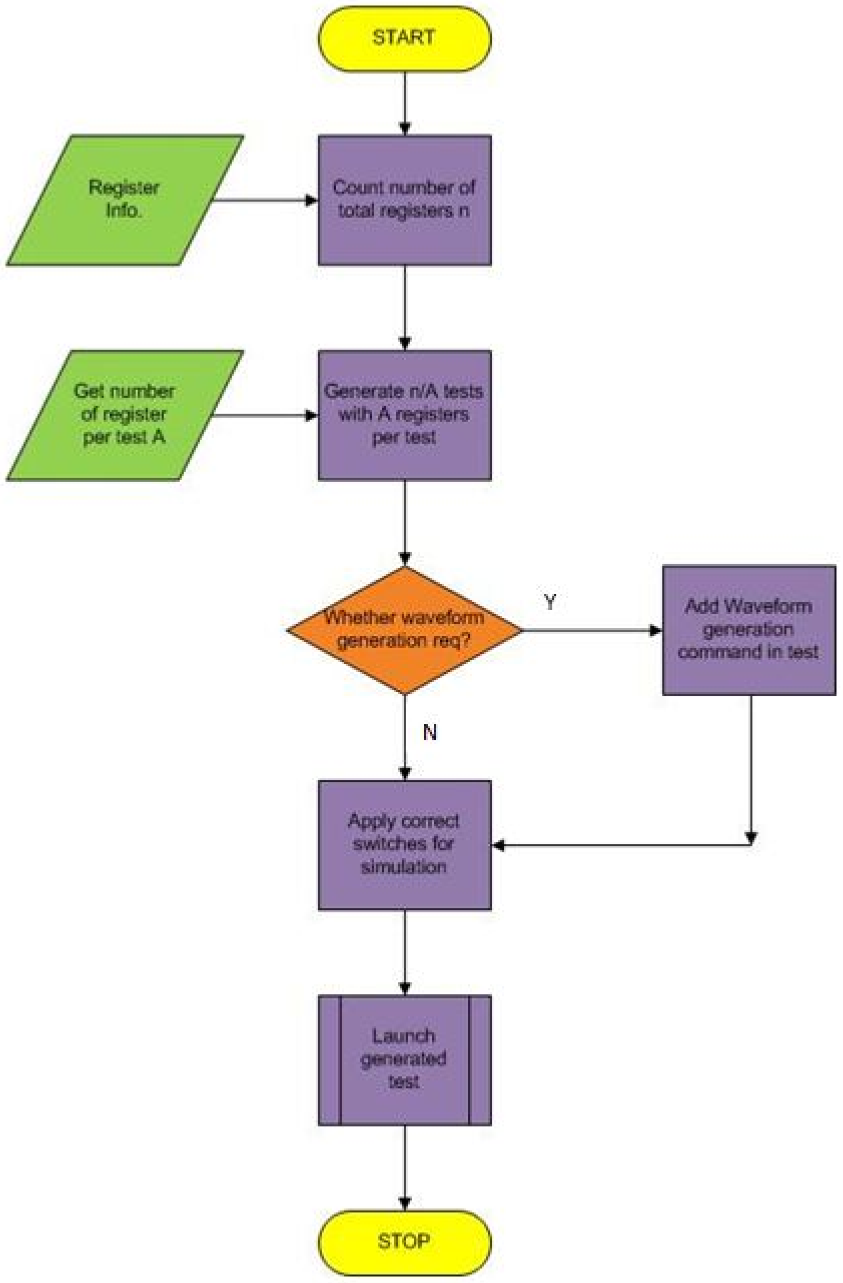

START
Register Info.
Count number of total registers n
Get number of register per test A
Generate n/A tests with A registers per test
Whether waveform generation req?
Y
Add Waveform generation command in test
N
Apply correct switches for simulation
Launch generated test
STOP

Figure 5.8: Flowchart for simple test generation

Registers info is given to the test according to the test constraints like register cluster and attributes.

For each test there will be more than 20 register according to test run limit and register access types.

Validator himself can decide the number of registers per test. After getting the register number from user the script counts the total number of tests need to be launched. Let the total number of registers be n and user constraint be A register per test. Then the script will generate nA tests each having A register per test.

Waveform simulation of the RTL takes more time and space as compare not dumping the waveforms. Because of these constraints normally waveform simulation of the registers are not done. If there is precise RTL errorbug which needs to be debugged, waveform for that particular test and for particular time duration is dumped.

Application of swicthes are the knob to handle all kind of constraints on the test. It can define request initiator, assign cluster specifice signals, operations to be done on registers and so on.

Launching of the test is predefined process need to be done by validator.

5.2.2 Status Collection and Maintence for Register Regression

Failures in configuration registers are spread across various categories such as bugs, issues, ongoing debug tracking using email discussions with stakeholders, temporary or permanent ignorable issues etc. The simple pictorial view of it is shown in fig below.

A major portion of them are previously encountered issues which constitute more than 90% of the failures. Tracking them and filtering to get the real failures for debug in the current regression is tedious job. Evaluation of indicators like pass percentages across various clusters, pending bugs impacted registers, tracking previous debug information constitute major resources and bug finding is getting delayed due to this. Hence a mechanism is needed to carry forward previous debug information to current regressions and direct attention to the debug of fresh issues.

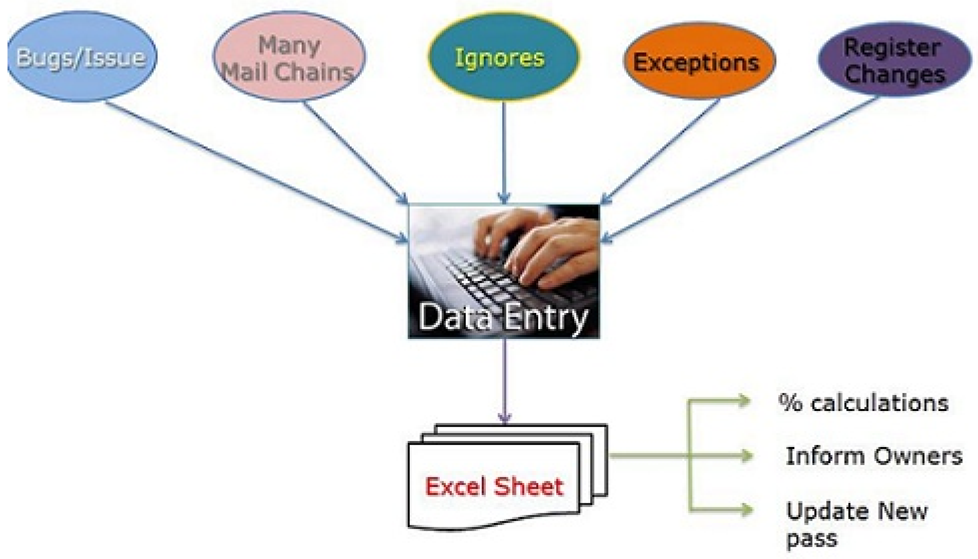


Figure 5.9: Status generation Mechanism without automation

To avoid manual efforts and improve accuracy a new automated flow is developed. This flow takes in three inputs i.e. current regression results data, register attributes file (RF) and central record file (CRF). RF holds names and attributes of all the registers in current model. Central Record File (CRF) is a reusable history of debug information containing issues/bugs impacted registers, ignores, known issues and exceptions.

Figure briefly shows the flow of automated process wherein inputs are taken from CRF, RF and current regression data. These 3 inputs go through automated flow and we get following outputs:

- Excel sheet specifying individual register status with description and owner of the register if it is failing and debugged.
- List of newly passing register which were failing in previous regressions.
- Number of registers passed, debugged, ignored, CTE owned and exceptions across individual clusters and in overall regression.

The main advantage of this script is it can count the pass and debug % very efficiently, for current regression including data of previous regression. The register regression data must not be checked for all the failures as the generated excel sheet will mark all the known is to be debugged. It helps to connect to the designer also to fix the bug very

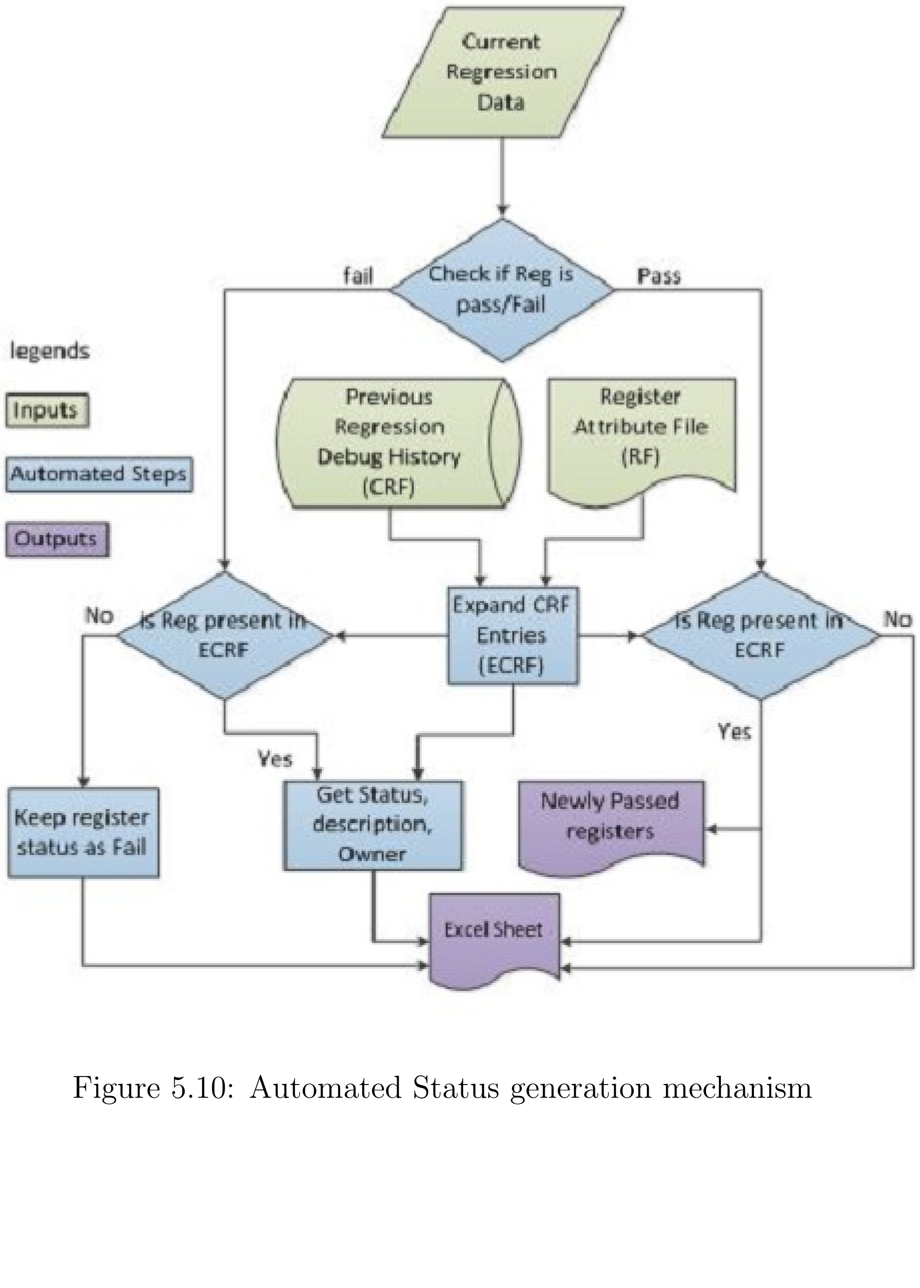

Current Regression Data
Check if Reg is pass/Fail
fail
Pass
legends
Inputs
Automated Steps
Outputs
Previous Regression Debug History (CRF)
Register Attribute File (RF)
No
is Reg present in ECRF
Expand CRF Entries (ECRF)
is Reg present in ECRF
No
Yes
Yes
Keep register status as Fail
Get Status, description, Owner
Newly Passed registers
Excel Sheet

Figure 5.10: Automated Status generation mechanism

easily. The only disadvantage here is ECRF should always be updated manually via validator himself.

Chapter 6

Results and Summary

6.1 Results

The CRV methodology in general with all its special features covers broad area of register validation in processor. It has given following results in current microprocessor validation.

- Compute optimization Verification of each register requires loading complete simulation environment of microprocessor chip. It takes much more compute time and disk space for each test case. Complete parallelized verification of 20-50 registers in a single test through smart coverage based stimulus has reduced compute by at least 20 times compared to any of the previous projects.

- Complex Attributes Verification All the complex attributes are effectively validated and found bugs which were not found by previous projects. Reset values are completely validated by CRV for the first time.

- Exceptions Handling & Status Reporting Centralized data base in user friendly format for CRV debug-data and exceptions has increased the productivity by enabling automated debug, onion peeling avoidance, automated status collection & reporting to the stakeholders. In the previous projects status tracking is carried out using XLS sheets which was time laborious.

- Bug Finding Techniques like delta regressions, automatically identifying the new failures to be debugged, have helped to find the bugs very quickly; the bug number is at least two-three times higher than that of any of the three previous projects.

6.2 Summary and Future work

This report discussed in detail about complexities of some of the register access types, and multiple inter-register-relations that pose chal-

lenges in the pre-silicon verification. We developed a strong, comprehensive CR validation methodology, which efficiently addressed all the CRV challenges. Proposed solutions like defining comprehensive verification process for each of the attributes, complete verification of 20-50 registers in a single test there by optimizing compute resources, avoidance of bug propagation, quick bug finding techniques etc. are found to be a leap ahead of the previous projects methodologies.

Variant registers verification continued to be a challenge in CRV. Around 20% of the variant registers verification is owned by designer in current processor. Analysis of these registers reveals that most of these registers are non-cooperative to the randomized stimulus. We need to explore methods to validate these categories of the registers by the CRV process itself.

This report discussed in detail about the proposed methodologies and algorithms. As the CR validation is a general challenge, the work presented in this paper can be useful for other projects also.

List of Figures

Bibliography

[1] Hardware/firmware Interface design:Best practise for imporving Embedded system Design by Gary Stringham

[2] Software development for embedded multi-core systems : a practical guide using embedded Intel architecture by Max Domeika

[3] DSP software development techniques for embedded and real-time systems [electronic resource] by Robert Oshana.

[4] Embedded software [electronic resource] :the works by Colin Walls.

[5] Fast and Effective Embedded Systems Design by Rob Toulson, Tim Wilmshurst

[6] Real-time agility [electronic resource] :the harmony method for real-time and embedded systems development by Bruce Powel Douglass

[7] Desktop 3rd Generation IntelCoreProcessor Family and Desktop IntelPentiumProcessor Family Datasheet Volume 1 & 2

[8] Intel64 and IA-32 Architectures Software Developer Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C

[9] IntelEmbedded PentiumProcessor Family Developer Manual

[10] Verification of Configuration Registers: Dont take it Easy! :Kunal Shah, ASIC Verification Engineer, EINFOCHIPS

[11] Shadow Register File Architecture: A Mechanism to Reduce Context Switch Latency :Jagan Jayaraj, Pravin Lawrence Rajendran and Thiruvel Thirumoolam , Anna University, Chennai.

[12] SystemRDL v1.0: A specification for a Register Description Language

[13] Automated approach to Register Design and Verification of complex SOC by Ballori Banerjee, Subashini Rajan and Silpa Naidu, LSI India R&D Pvt. Ltd

[14] MMV: A Metamodeling Based Microprocessor Validation Environment by Deepak A. Mathaikutty, Sreekumar V. Kodakara, Ajit Dingankar, Sandeep K. Shukla, and David J. Lilja, IEEE

[15] A Property Checking Approach to Microprocessor Verification using Symbolic Simulation by Prabhat Mishra, Narayanan Krishnamurthy, Nikil Dutt, and Magdy Abadir ,Motorola Inc., Austin, TX