IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)
Volume 5, Issue 2, Ver. II (Mar. - Apr. 2015), PP 21-25
e-ISSN: 2319 – 4200, p-ISSN No. : 2319 – 4197
www.iosrjournals.org

Random Test Program Generator for SPARC T1 Processor

Nutan Hegde[1], Lekha Pankaj[2], Lyla B Das[3]
[1](ECE, KMCTCEW,Calicut, Kerala ,India)
[2](ECE, AWH,Calicut,Kerala,India)
[3](ECE,NIT-C,Kerala,India)

Abstract: *As the complexity of the microprocessor design is increasing the verification method is also becoming more and more complex. Unfortunately, it requires a long time to generate and run test sequences. Under the time to market pressure, it is very time consuming to write all test programs manually. This brings about the necessity of developing a random test program generator. The proposed method is for verifying the pre silicon model of the multithreaded multi cored processor (SPARC T1). The work takes the input from the user and generates a sequence of assembly language instructions (SPARC V9) randomly, initializes all the register values with random numbers to verify the corner cases. The test cases are seed based and run through the RTL model to check the correctness of the design. The test cases generated are also run on the defective model.*

Keywords: *SPARC, SPARC V9, RTPG, Defective model.*

I. Introduction

Open sourcing the hardware design of SPARC T1 processor by Sun Microsystem Inc. Inspired to study and analyze a 64 bit, 32 threads and 8 core processor. The insight of the design aspects, verification of the hardware design is achieved. Verification is a process to demonstrate that intent of a design is preserved in its implementation. Silicon capacity continues to increase enabling us to create and accommodate complex system within the same die. But the ability to verify larger and complex systems unfortunately has not kept in pace. Computer designs are so complicated these days that it can be impossible for humans to think through the entire testing space. At some point the people who write directed tests run out of testing concepts to implement. This rarely means that all the design problems have been found and fixed. Pseudo random test generators can often push the hardware designs harder by stimulating the logic under test in scenarios beyond the test developer’s imagination. Automatic generation of test programs plays a major role in the verification of modern processors and hardware systems.

II. Previous Work

Different algorithms are used to write the random test program generator depending on the complexity and problem faced in verification. The Model Based Test Generator comprises of an architectural model, a testing knowledge data-base, a behavioral simulator, architecture independent generator and a Graphical User Interface [1]. A pseudo random test system generates a processor instruction text file, containing a sequence of instructions in a target processor’s assembly language [2]. Random test program generators for functional verification of processors were previously reported [3]. A random approach to automatic test generation for software and hardware verification has proved to be successful. It was applied to the verification of selected design units such as floating-point unit and even a complete processor, but very strong restrictions were imposed on the generated test programs [4]. The assumption behind these tools is that only a small subset of possible tests is simulated during functional verification. These tests are run through the design simulation model (the HDL model), and the results are compared with those predicted by the architecture specification represented by a behavioral simulator. Processor functional verification consists of two steps: 1) Generation of tests – for complex processors, this is done by automatic test generators which supply better productivity than manual tests. 2) Comparison between results computed by the two levels of simulation – both simulators are provided with the generated tests as stimuli, and the simulators final states are compared [1].The test space of SPARC T1 processor design is so huge that it is very difficult to completely specify it.As a result, directed testing (special hand-coded test cases) alone would not be sufficient to find all the design defects. Hence a mechanism to derive assembly language test cases from higher level programs, such as C was developed to ease the test case development effort [3].Different algorithms are used to write the random test program generator depending on the complexity and problem faced in verification. A random approach to automatic test generation for software and hardware verification has proved to be successful. It was applied to the verification of selected design units such as floating-point unit and even a complete processor, but very strong restrictions were imposed on the generated test programs [4].

The test space of SPARC T1 processor design is so huge that it is very difficult to completely specify it .As a result, directed testing (special hand-coded test cases) alone would not be sufficient to find all the design defects. Hence a mechanism to derive assembly language test cases from higher level programs, such as C was developed to ease the test case development effort [3].

III. Proposed Methodology

The proposed method focuses on the study of SPARC instruction sets, registers based on SPARC V9 architecture. The test cases generated in SPARC assembly language based on the SPARC V9 architecture. The simulations are run at RTL level without the reference model. The proposed work mainly targets the EXU and LSU of SPARC processor.

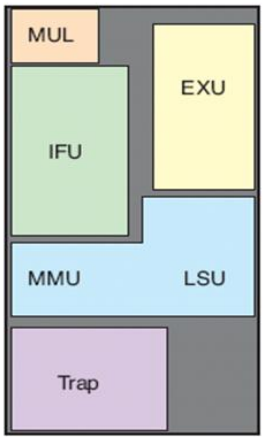


Fig1. Blocks of Sparc core.

Execution unit includes an arithmetic logic unit (ALU), shifter and the execute stage of the pipeline. It calculates memory and branch addresses. LSU includes memory and write back stages. Data cache maintains the order for cache updates, handles memory references between the SPARC core, the L1 data cache, and the L2 cache. All communication with the L2 cache is through the crossbar.

IV. Implementation Of RTPG

The work creates a instruction set library. It initializes all the input registers using rand() function which returns a pseudo random number each time it is called. The RTPG randomly selects the instructions from the instruction library, checks the validity of the instruction, depending on the instruction the format of the instruction is done and writes it in the file. The file so formed is a test case. Each Test cases generated will have a seed number. Using this seed number the user can regenerate the test case. The user can get the seed value of any particular test case by using the log file which is an output file of RTPG. The log file contains the file names and its seed value generated by the generator. The generator developed can produce the test cases for specific constraints by specifying a user constraint file. The constraints file gives an opportunity to the user to put a restriction on the occurrence of any instruction. The RTPG has an additional option for the user, through which the user can introduce a code snippet which they want to test exclusively or any comments at the end of generated test cases. Fig 2. Shows the block diagram of the proposed work.

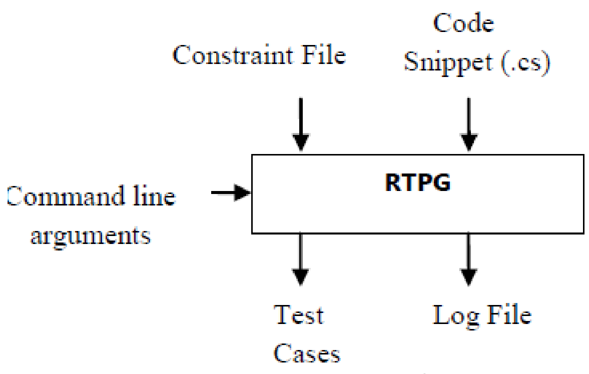


Fig 2. Block diagram of RTPG

4.1 Interfaces To RTPG

The interfaces to the generator are command line, constraint file and the code snippet which the user wants to introduce exclusively to the test case.

4.1.1 Command line arguments:

Executable file created is of the name RandGen.

Format of argument is RandGen -n <number of diags> -o <output filename> -cf<constraint file name.cf> -s < input seed value> -dir<output directory> -cs<code_snippet.cs> -length <number of instructions>

4.1.2 Control/Constraint file:

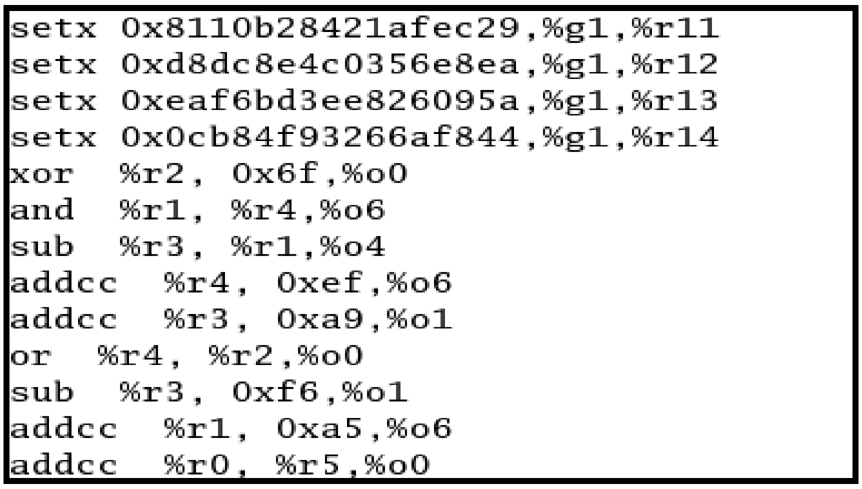
After getting the command line arguments the particular control file is read for the additional information such as instruction weight, which specifies the weightage for individual instruction in the output from the RTPG i.e. the weight controls the frequency of the occurrence of the instructions in the test cases. Total weight will be sum of individual instructions.eg: add 10 (the weight for add instruction is 10/40 out of 40 instruction) .

Table 1: Options and functions of command line argument for RTPG

Option	Functions
-n	Number of test cases. The user can specify the number of test cases to be generated. The Generator stops after that.
-o	Output filename. The user can give the test case name from this option. The generated testcases will have the name followed by the number from one upto the number of testcases mentioned by the user.
-cf	Control/constraint file name. The user can mention the file name by which the generator has to take the input for the constraint.
-s	Input seed value. It is an addition option if mentioned the generator will regenerate the test case with that particular seed value.
-cs	Code snippet. This is also an additional option, if mentioned the file name specified in the option will be appended to each of the testcases. The user can use this option mainly to add some instructions exclusively or to add some comments.
-len	Number of instructions. Each test case will have randomly generated instructions. It contains the number of instructions mentioned by this option.

V. Result

The test cases were stored, and a diaglist was created to run all the test cases together hence came up with the regression suite. The test case snippet generated is in fig 3.



```
setx 0x8110b28421afec29,%g1,%r11
setx 0xd8dc8e4c0356e8ea,%g1,%r12
setx 0xeaf6bd3ee826095a,%g1,%r13
setx 0x0cb84f93266af844,%g1,%r14
xor  %r2, 0x6f,%o0
and  %r1, %r4,%o6
sub  %r3, %r1,%o4
addcc  %r4, 0xef,%o6
addcc  %r3, 0xa9,%o1
or  %r4, %r2,%o0
sub  %r3, 0xf6,%o1
addcc  %r1, 0xa5,%o6
addcc  %r0, %r5,%o0
```

Fig 3. Code snippet generated from RTPG.

5.1 Simulation result

The generated test cases are run on the VCS (Verilog Compile and Simulator) from Synopsys. Under the name of each test case a directory will be created which holds the status and simulation files. Long simulations were conducted both on the defected and clean model. The results were monitored through log file. The status.log gives the information status of each testcase after running the regression. The log files shows exactly how the diag is formed including the reset, traps and main section of the program.The testcases which identifies the defect gives a bad trap. The status.log report gives the diag status. Fig 4 shows the Status.log of diags run in regression.

```
Summary for /home/netuser/m070269ec/open_sparc/model_dir/2008_12_03_0
==========================================================================
         Status:  mytests |       ALL |
--------------------------------------------------------------------------
           PASS:       64 |        64 |
           FAIL:        0 |         0 |
   Diag Problem:        1 |         1 |
License Problem:        0 |         0 |
  MaxCycles Hit:        0 |         0 |
 Socket Problem:        0 |         0 |
        Timeout:        0 |         0 |
    LessThreads:        0 |         0 |
 Simics Problem:        0 |         0 |
    Performance:        0 |         0 |
Killed By Job Q:        0 |         0 |
        Unknown:        0 |         0 |
     UnFinished:        0 |         0 |
   flexlm error:        0 |         0 |
--------------------------------------------------------------------------
     Diag Count:       65 |        65 |
--------------------------------------------------------------------------
```

Fig 4 Status.log run as regression suite

5.1.1 Simulation timing:

The simulation timing is analyzed, it is clear that the lesser the instruction lesser the simulation time. But different test cases take different time to identify a bug. Some test cases will go out of time.

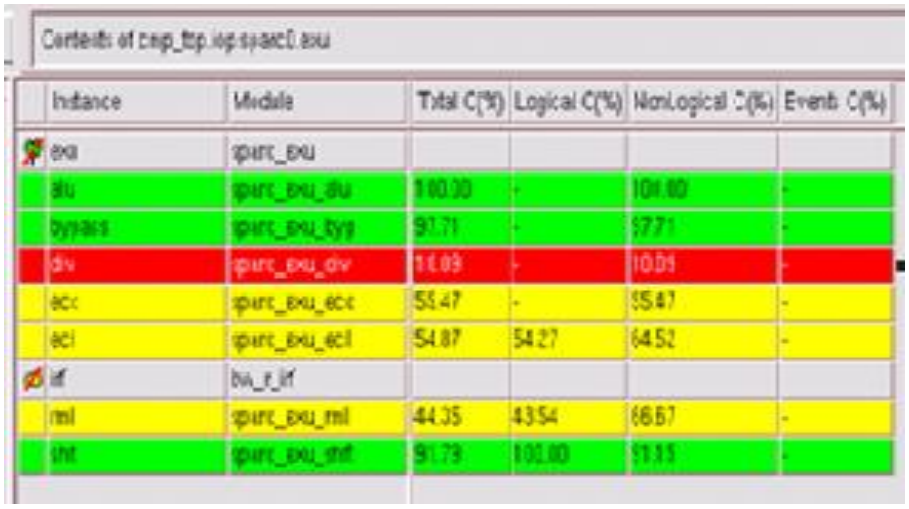
Table 2: Simulation timing for different testcases

Instruction Length	Simulation Time (sec)	Cycles	Cycles/Sec
20	132.38	14759	111.5
50	133.67	15089	112.9
1000	167.27	25495	152.4

The testcases with and without div unit is also taken for the analysis. The testcases targeting the complex unit will take more time.

5.1.2. Coverage Metrics

Coverage analysis was done using the CM View from the Synopsys which determines how well the test case is covering the design under test. The merged report gives the total coverage of all the test cases generated. It is also possible to test the coverage of individual test case. It is found that the testcases generated without the div instruction using the constraint file gives very less coverage of div module.

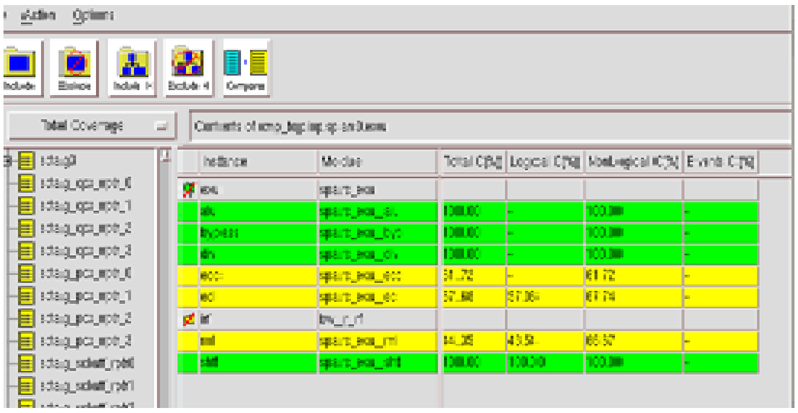


Contents of cmp_top.iop.sparc0.exu

Instance	Module	Total C(%)	Logical C(%)	NonLogical C(%)	Event C(%)
exu	sparc_exu				
alu	sparc_exu_alu	100.00	-	100.00	-
bypass	sparc_exu_byp	97.71	-	97.71	-
div	sparc_exu_div	11.09	-	10.05	-
ecc	sparc_exu_ecc	55.47	-	55.47	-
ecl	sparc_exu_ecl	54.87	54.27	64.52	-
irf	bw_r_irf				
rml	sparc_exu_rml	44.35	43.54	66.67	-
shft	sparc_exu_shft	91.79	100.00	93.33	-

Fig 5. Coverage report without div instruction

The result was compared with div instruction which gives max coverage to that module. The third row of div module with 10% increases to 100%.



Contents of [illegible]

	Instance	Module	Total C[%]	Logical C[%]	NonLogical C[%]	Events C[%]
	exu	sparc_exu				
🟩	alu	sparc_exu_alu	100.00	-	100.00	-
🟩	bypass	sparc_exu_byp	100.00	-	100.00	-
🟩	div	sparc_exu_div	100.00	-	100.00	-
🟨	ecc	sparc_exu_ecc	61.72	-	61.72	-
🟨	ecl	sparc_exu_ecl	57.88	57.06	67.74	-
	irf	bw_r_irf				
🟨	rml	sparc_exu_rml	44.25	43.5	66.57	-
🟩	shft	sparc_exu_shft	100.00	100.00	100.00	-

Fig 6. Coverage report with div instruction

VI. Conclusion And Future Scope

The ultimate goal of design verification is to ensure equivalence between a design and its functional specification. Processor/ASIC complexity, custom logic size and software contents are all increasing at such a pace that schedules are being squeezed and resources are being stretched. Presilicon verification today consumes about 70-80% of the design effort behind a processor/ASIC. Good presilicon verification methodology greatly minimizes the number of post silicon problems that can often be extremely difficult to debug. In time to market pressure it becomes necessary to go for automatic testcase generator.

Random test program generator targeting the arithmetic and logic, load and store unit can be extended to test all the other unit and core of the SPARC core to give deeper insight of the processor and troubleshooting mechanism.

References

[1]. AharonAharon,Dave Goodman, Moshe Levinger, Yossi Lichtenstein,Yossi Malka, CharlotteMetzger, Moshe Molcho, Gil Shurek,Test Program Generation for Functional Verificationof PowerPC Processors in IBM,

[2]. Jeffery D. Whitman,System and method for generating pseudo random instruction for design verification USPatentdocuments, no 5, 572, 666,NOV 5,1996.

[3]. BabuTurumella, AimanKabakibo, ManjunathBogadi, Karunakara Menon et al, Design verification of a super-scalar RISC processor,Twenty-Fifth International Symposium on Fault –TolerantComputing, p.0472, IEEE, 1995

[4]. A. Aharon,A.Bar David,B.Dorfman,E.Gofman,M.Leibowitz,V. Schwartzburd ,Verification of the IBM RISC System / 6000 by a dynamic biased pseudo-random test program generator,IBM systems journal, Vol 30, no 4, 1991, pp 527-538

[5]. S. Mehta ,S. Ahmed, S. Al-Ashari, Dennis Chen, et al.Verification of the UltraSPARC^{TM}Microprocessor IEEE Computer Society Int'l conf., 1995, pp 452-461.

[6]. IrithPomeranz, Nirmal R. Saxena, Richard Reeve, ParitoshKulkami,Generation of Test Cases for Hardware Design Verification of a Super-Scalar Fetch Processor, IEEE Int'l Test Conf., 1996, pp 904-913

[7]. Samir Palnitkar, Verilog HDL-A guide to Digital Design and Synthesis ,First Edition, Pearson Education Asia, 2001.

[8]. David L. Weaver, OpenSPARC™ Internals, First Edition, Berkeley BSD systems,2008, ISBN 978-0-557-01974-8, 2008.

[9]. David L. Weaver and Tom Germond The SPARC ArchitectureManual,Version 9, PTR Prentice Hall,ISBN 0-13-825001-4

[10]. OpenSPARC™ T1 Microarchitecture Specification, Copyright © 2006 Sun Microsystems, Inc.

[11]. OpenSPARC™ T1 Processor Design and Verification User's Guide" Copyright 2008, Sun Microsystems, Inc.

[12]. www.opensparc.net