Automatic Test Program Generation for Pipelined Processors

F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

http://www.cad.polito.it/

Abstract

The continuous advances in microelectronics design are creating a significant challenge to design validation in general, but tackling pipelined microprocessors is remarkably more demanding. This paper presents a methodology to automatically induce a test program for a microprocessor maximizing a given verification metric. The approach exploits a new evolutionary algorithm, close to Genetic Programming, able to cultivate effective assembly-language programs. The proposed methodology was used to verify the DLX/pII, an open-source processor with a 5-stage pipeline. Code-coverage was adopted in the paper, since it can be considered the required starting point for any simulation-based functional verification processes. Experimental results clearly show the effectiveness of the approach.

Contact Author:

Dr. Giovanni Squillero

Politecnico di Torino
Dipartimento di Automatice e Informatica
Cso Duca degli Abruzzi 24
10129 Torino
Italy

Tel: +39-0115647092
Fax: +39-0115647099

EMail: `squillero@polito.it`

Automatic Test Program Generation for Pipelined Processors

Abstract

The continuous advances in microelectronics design are creating a significant challenge to design validation in general, but tackling pipelined microprocessors is remarkably more demanding. This paper presents a methodology to automatically induce a test program for a microprocessor maximizing a given verification metric. The approach exploits a new evolutionary algorithm, close to Genetic Programming, able to cultivate effective assembly-language programs. The proposed methodology was used to verify the DLX/pII, an open-source processor with a 5-stage pipeline. Code-coverage was adopted in the paper, since it can be considered the required starting point for any simulation-based functional verification processes. Experimental results clearly show the effectiveness of the approach

1. Introduction

Developments in semiconductor technology have made possible to design an entire system onto a single chip, a connected practice is the usage of predefined logic blocks known as *cores*. A core is a highly complex logic block predictable, reusable, and fully defined in terms of its behavior. System designers can purchase cores from core-vendors and integrate them with their own user-defined logic to implement devices more efficiently.

The continuous advances in the design field are creating a significant challenge to design validation. For most cores, the hardest verification problem is to ensure the correctness of microarchitectural features in the initial description. This design is usually described at register-transfer level in a hardware description language, i.e., as a set of interconnected functional blocks. The difficulty of validation stems from the complexity of design, and the resulting large state space which needs to be searched in order to check for correctness. The validation of a microprocessor is known to be a challenging problem [ShAb99] and technology advances is hardening it. Today, the validation of cores based on microprocessors has become exceptionally complex since they may include advanced features, such as caches and pipelines, that a few years ago were limited to high-class microprocessors.

Different activities performed in different stages of the design flow and at different level of abstraction can be sensibly called *verification*. These activities include checking equivalence of different models, ensuring the conformity to specifications, electrical and post-silicon testing. Reviewing all design verification aspects would deserve a very long discussion, but it is possible to distinguish between two classes of approaches: *formal* and *simulation-based*.

Formal methods try to verify the correctness of a system by using mathematical proofs, whereas simulation-based design verification tries to uncover design errors by detecting a circuit faulty behavior when deterministic or pseudo-random tests are applied. Formal methods implicitly consider all possible behaviors of the models representing the system and its specification, and the accuracy and completeness of the system and specification models, as well as required computation resources, are a fundamental limitation. On the contrary simulation-based methods do not suffer from the same constraints, but can only consider a limited range of behaviors and will never achieve 100% confidence of correctness.

Formal verification methods for complex microprocessor designs has been targeted by several researchers in the past [Harm91], [CMHa99] and [VeBr00] but, although results were considerable, all proposed techniques suffer from severe drawbacks and need considerable human efforts to handle entire designs of today's processors and advanced microarchitectural features.

This paper targets simulation-based verification performed when a register-transfer level model is available, thus at a rather early step of the design flow. It can be maintained that almost all verification processes include *at least* some simulation-based steps where specific test programs are simulated to detect faulty behaviors.

Code coverage metrics are the required starting point for simulation-based processes. Coverage analysis measures the number of executed statements in a description during a simulation, ensuring that no part of the design missed functional test during the experiment. Moreover, it may help reducing simulation effort avoiding "over-verification" and redundant testing. Although more complex and theoretically insightful metrics have been proposed, the use of coverage analysis provides an easy, fast and objective way of measuring simulation effectiveness. Indeed, most CAD vendors have recently added the possibility to measure code-coverage figures to their simulators.

Despite the simplicity of the metric, devising a test program able to reach a high statement coverage figure may be a challenging task. The task is particularly challenging for pipelined microprocessor, since in such architectures consecutively-executing instructions can have their execution overlapped in time. In a pipelined microprocessor, instruction execution steps are arranged so that the CPU doesn't have to wait for one operation to finish before starting the next. Thus, at any given time, the CPU is *at the same time* executing some instructions and fetching next ones in the program. The first consequence is that it is not only necessary to check the functionalities of all possible instructions with all possible operands, but it is also necessary to check all possible interactions between instructions inside the pipeline.

Cross-compiled high-level routines have little chance to attain good results in term of statement coverage even on simpler cores, due to the intrinsic nature of the algorithms and due to compiler strategies they are not usually able even to test functionalities in non-pipelined microprocessors [CCSS02b].

Hand-designed test benches are very difficult to devise since the execution of multiple instructions in the pipeline may lead to Byzantine interactions [CMHa99]; they require a long effort by skilled engineers, thus they are highly expensive. Finally, relying on a hand-made approach often means loosing the possibility of checking highly unpredictable corner cases.

This paper presents an evolutionary approach to the generation of test programs for functional verification. To evaluate the approach, the statement coverage metric is exploited. The approach exploits *MicroGP*, an evolutionary algorithm stemming from *Genetic Programming* [Koza98] and specifically tuned to generate assembly programs instead of traditional lisp S-expressions. *MicroGP* [CCSS02a] relies on a syntactical description of an assembly language and on a function able to evaluate the fitness of individuals, i.e., to evaluate how close programs in the population are to the final goal. A similar approach has been exploited in [CCSS02b] for the verification of the i8051, a core with no pipeline enhancements. For the purpose of this paper, the program representation has been significantly enhanced, increasing the semantic power.

2. Simulation Based Design Verification

Given an RT-level description of a microprocessor, simulation-based verification requires a test program and a tool able to simulate its execution. The goal is to uncover all design errors, and the effectiveness of the test program is usually evaluated with regards to some metric.

As stated before, the verification metric exploited in this paper is the statement coverage. To avoid confusion, in the following the term "instruction" denotes an *instruction* in an assembly program, and the term "statement" refers to a *statement* in an RT-level description. The term "execute" is commonly used in both domains: instructions in a program are executed by the microprocessor when it fetches them and operates accordingly; statements in a description are executed when the simulator evaluates them to infer design behavior. The code coverage metric exploited in this paper measures the percentage of executed RT-level statements over the total when the execution of a given test program is simulated.

The simplest method to obtain an assembly test program is probably to compile a high-level routine. This approach relies on a compiler or cross-compiler, an easily met requirement. However, despite their effortlessness, compiled problem-specific algorithms are severely inadequate to uncover design errors. First of all, due to the intrinsic nature of the algorithms, the resulting program may not be able to expose all functionalities. Then compilers adopt strategies for optimizing the code with regard to some parameter, such as size or speed, thus the usually do not exploit *all* possible assembler instructions and addressing modes, but only the more efficient ones.

A better strategy is to generate random assembly programs. This approach is likely to cover more statements in the description, but is less straightforward. A randomly generated program may easily contain illegal operations, such as division by zero, or endless loops. However, the effort required to generate a syntactically correct assembly source is still moderate. The main drawback of this method is that a random program will hardly cover all corner cases, hence resulting in low statement coverage. In order to obtain a sufficient coverage a large number of long programs are needed, resulting in overlong simulation times.

The approach for devising a test program presented in this paper is based on an evolutionary algorithm. The induction of the test case is fully automated and only requires a syntactical description of the assembly implemented by the microprocessor. The resulting program is reasonably short and maximizes the target verification metric.

3. Genetic Programming for Test Program

The overall architecture is shown in *Figure 1*. The microprocessor assembly language is described in an instruction library, and the test case generator generates efficient test programs exploiting it. The execution of each assembly program is simulated with an external tool, and the corresponding statement coverage is used to drive the optimization process.

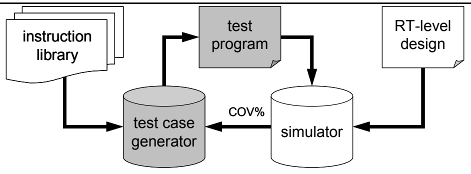

Figure 1*: System Architecture*

Genetic Programming (GP) was defined as a domain-independent problem-solving approach in which computer programs are evolved to solve, or approximately solve, problems [Koza98]. GP addresses one of the more desired goals of computer science: creating, in an automated way, computer programs able to solve problems.

Traditional GP induced programs are mathematical functions that, after being evaluated, yield a specific result. The pioneering ideas of generating *real* (Turing complete) programs date back to [Frie58]. More recently [Nord94] and [NoBa95] suggested to directly evolve programs in machine-code form for completely removing the inefficiency in interpreting trees. A genome compiler has been proposed in [FSMu98], which transforms standard GP trees into machine code before evaluation.

This paper exploits a versatile GP-like approach for inducing assembly programs presented in [CCSS02a]. The methodology exploits a directed acyclic graph (DAG) for representing the *flow* of the program (Figure 2). The DAG is built with four kinds of nodes: prologue (with out-degree 1), epilogue (with out-degree 0), sequential instruction (with out-degree 1), and branch (with out-degree 2). A similar graph-based representation for GP has been developed for different purposes in [KaBa02].

- **prologue** and **epilogue** nodes are always present and represent required operations, such as initializations. They depend both on the processor and on the operating environment, and they may be empty. The prologue has no parent node, while the epilogue has no children. These nodes may never be removed from the program, nor changed.

- **Sequential-instruction** nodes represent common operations, such as arithmetic or logic ones (e.g., node **B**). They are always followed by exactly one child. The number of parameters (i.e., numeric constants or register specifications) changes from instruction to instruction, following assembly language specification. *Unconditional* branches are considered sequential, since execution flow does not split (e.g., node **D**).

- **Conditional-branch** nodes are translated to assembly-level conditional-branch instructions (e.g., node **A**). All common assembly languages implement some *jump-if-condition* mechanisms. All conditional branches implementbed in the target assembly languages are included in the library.

Each node contains a pointer inside the instruction library and, when needed, its parameters. For instance, Figure 2 shows a sequential node that will be translated into a "*XOR r19, r13, r11*" instruction, i.e., a bit-wise EXOR between registers *r13* and *r11* that stores the result in register *r19*.

The DAG is translated to a syntactically correct assembly program, although it is not possible to infer its semantic meaning. An induced program may perform operations on any register and any memory locations, and this exceptional freedom is essential to generate test programs.

The DAG representation and the instruction library adopted in [CCSS02a] and [CCSS02b] prevented backward branches, either conditional or unconditional. This characteristic guaranteed program termination, since no endless loop may be implemented, but introduced a reduction in semantic power. To verify the DLX core this restriction was discarded and the MicroGP was given the possibility to generate assembly program with branches *anywhere* in the memory space exploiting special *jump registers* instructions. These instructions perform unconditional jumps to the target address contained in a register and are represented as simple sequential-instruction in the DAG.

The library approach developed in [CCSS02a] enables to exploit the genetic core and the DAG structure with different microcontrollers or microprocessors that not only implement different instruction sets, but also use different formalisms and conventions. Indeed, the method has been already successfully tested with an i8051 [CCSS02b] and a SPARC [SPARC].

3.1. Test Program Induction

Test programs are induced by mutating the DAG topology and by mutating parameters inside DAG nodes. Both kinds of modifications are embedded in an evolutionary algorithm implementing a (μ+λ) strategy.

In more detail, a population of μ individuals is cultivated, each individual representing a test program. In each step, λ new individuals are generated by mutating existing ones, whose parents are selected using tournament selection with tournament size τ (i.e., τ individuals are randomly selected and the fittest one is picked). After creating new λ individuals, the best μ programs in the population of (μ+λ) are selected for surviving. The initial population is generated by creating μ empty programs (only prologue and epilogue) and then applying i_m consecutive random mutations to each.

Three mutation operators are implemented and are chosen with equal probability:

- **Add node:** a new node is inserted into the DAG. The new node can be either a sequential instruction or a conditional branch. In both cases, the instruction referred by the node is randomly chosen. If the inserted node is a branch, either unconditional or conditional, one of the subsequent nodes is randomly chosen as the destination. Remarkably, when an unconditional branch is inserted, some nodes in the DAG may become unreachable (e.g., node **E** in Figure 2).

- **Remove node:** an existing internal node (except prologue or epilogue) is removed from the DAG. If the removed node was the target of one or more branch, parents' edges are updated.

- **Modify node:** all parameters of an existing internal node are randomly changed. Parameters are numerical immediate values and register specifications.

The evolution process iterates until population reaches a *steady state* condition, i.e., no improvements are recorded for S_g generations.

3.2. Test Program Evaluation

Individuals are evaluated by simulation. The DAG is first translated into a syntactically correct assembly program and assembled to machine code. Then the execution of the test case is simulated on the RT-level description of microcontroller, gathering verification metric figures (statement coverage). The final value is considered as the *fitness* value of the individual, i.e., the extent to which it is able to produce offspring in the environment.

Fitness values are used to probabilistically select the λ parents for generating new offspring and to deterministically select the best μ individuals at the end of each evolution step.

Test program evaluation does not consider the internal structure of the microprocessor, nor does it include *hints* for increasing the coverage based on designers' knowledge.

4. Experimental Results

A prototype of the proposed approach has been developed in ANSI C language in about 2,000 lines of code. The prototype exploits *Modelsim* v5.5a by Model Technology for simulating the design and getting coverage figures. The proposed approach was tested on DLX/pII, an open-source implementation of the processor model detailed by Hennesey and Patterson in [PaHe96] implementing a 5-stage pipeline. The description consists of about 1,000 register-transfer level statements in VHDL.

Starting from a high-level syntactical description of the DLX assembly language, the prototype induced an effective test program in about two days on a Sun Enterprise 250 with an UltraSPARC-II CPU at 400MHz, and 2GB of RAM. Evolution required simulating about 10,000 different test benches.

Experimental results showed that devising a test case able to reach complete statement coverage on a pipelined processor is a challenging task, even on a relatively simple processor. In Table 1, the statement coverage ([SC%]) obtained by the induced test program is compared with the one attained by different functional test programs provided by microprocessor implementers (*set_s, set_su, arith_s, carry_su, fak, jump1, loadstore_s, loadstore_su, mul_su, intrpt1, except*), problem-specific algorithms (*mul32* and *div32*), system software (*system01*) and an exhaustive functional test that checks all possible instructions (*all_instr*). Table 1 also reports the length of the programs in instructions [Len] and the number of clock cycles required to execute them [#CLK]. Table 2 further details the results on different parts of DLX: *instruction fetch stage* (if), *decode stage* (dec), *execution stage* (exe), *memory access stage* (mem), *write-back stage* (wb) and *special registers controller* (regs).

Cross-compiled algorithms are seldom effective to fully validate a design. In fact *mul32*, a 32-bit multiplication performed through shifts and sums, attains the lowest statement coverage. Also specific test benches, like *set_s*, are not able to attain a globally good results, despite a long execution time (e.g., *exept*). For the sake of comparison, 10,000 random programs, each one filling all available RAM were generated and the best result is shown in row labeled *random*. The huge random test bench surpasses all previous ones, but, in absolute terms, the attained statement coverage is not excellent (77.12%). Noticeably, the *all_instr* program, carefully designed to exhaustively test *all* possible instructions, is unable to thoroughly verify pipeline stages and attains a statement coverage below 80%. Remarkably, the statement coverage attained by the test program generated by our automatic method is nearly 15% higher. Obviously, further increase in the attained figure may be prevented by the existence of unreachable piece of code in the model.

The effect of interactions between simultaneously executed instructions can be easily seen in Table 2. Executing all instructions is not enough to verify functionalities of the decode stage (dec), since several cases are exposed only when specific instructions are executed concurrently. These cases are relatively uncommon and are rarely exposed using a random approach.

5. Conclusions

This paper presents a methodology able to automatically induce an assembly test program for a pipelined microcontroller. The methodology is based on the MicroGP evolutionary algorithm and exploits the syntactical description of the assembly language.

Experimental results clearly show the efficiency of the approach. The proposed methodology is completely automatic, since it relies on a syntactical description of the assembly language only and, unless commonly adopted methodology, it does not require a skilled programmer or additional analysis to be performed by verification engineers. The generated test case is able to efficiently maximize a given verification metric, and, at the end of the process, designers are required to manually analyze only those parts of the description that the tool failed to validate.

A prototype of the proposed approach has been developed in ANSI C language, and then it was tested on an RT-level implementation of the DLX/pII using statement coverage as verification metric. Induced test programs outperformed test cases devised with alternative methodologies.

6. References

[CCSS02a] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Efficient Machine-Code Test-Program Induction", *Congress on Evolutionary Computation*, 2002, pp. 1486-1491

[CCSS02b] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Evolutionary Test Program Induction for Microprocessor Design Verification", To be presented in: *Asian Test Symposium*, 2002

[CMHa99] D. Van Campenhout, T. N. Mudge, J. P. Hayes, "High-Level Test Generation for Design Verification of Pipelined Microprocessors", *Design Automation Conference*, 1999, pp. 185-188

[Frie58] R. M. Friedberg, "A Learning Machine: Part (I)", *IBM Journal of Research and Development*, vol. 2, n. 1, 1958, pp 2-13

[FSMu98] A. Fukunaga, A. Stechert, D. Mutz, "A genome compiler for high performance genetic programming", *Genetic Programming 1998: Proceedings of the 3rd Annual Conference*, 1998, pp. 86-94

[Harm91] N. A. Harman, "Verifying a Simple Pipelined Microprocessor Using Maude", *Lecture Notes in Computer Science*", 2001, vol 2267, pp. 128-142

[KaBa02] W. Kantschik, W. Banzhaf, "Linear-Graph GP — A new GP Structure", *EuroGP2002: 4th European Conference on Genetic Programming*, 2002, pp. 83-92

[Koza98] J. R. Koza, "Genetic programming", *Encyclopedia of Computer Science and Technology*, vol. 39, Marcel-Dekker, 1998, pp. 29-43

[NoBa95] P. Nordin W. Banzhaf, "Evolving Turing-complete programs for a register machine with self-modifying code", *Genetic Algorithms: Proceedings of the 6th International Conference*, 1995, pp. 318-325

[Nord94] P. Nordin, “A compiling genetic programming system that directly manipulates the machine code,” *Advances in Genetic Programming*, 1994, pp. 311-331

[PaHe96] D. A. Patterson, J. L. Hennessy, *Computer Architecture - A Quantitative Approach (2nd edition)*, Morgan Kaufmann Publishers, San Francisco, California, 1996

[ShAb99] J. Shen, J. A. Abraham, “Verification of Processor Microarchitectures”, *VLSI Test Symposium*, 1999, pp. 189-194

[SPARC] SPARC International, *The SPARC Architecture Manual*

[VeBr00] M. N. Velev, R. E. Bryant, “Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exception, And Branch Prediction”, *Design Automation Conference*, 2000, 112-117

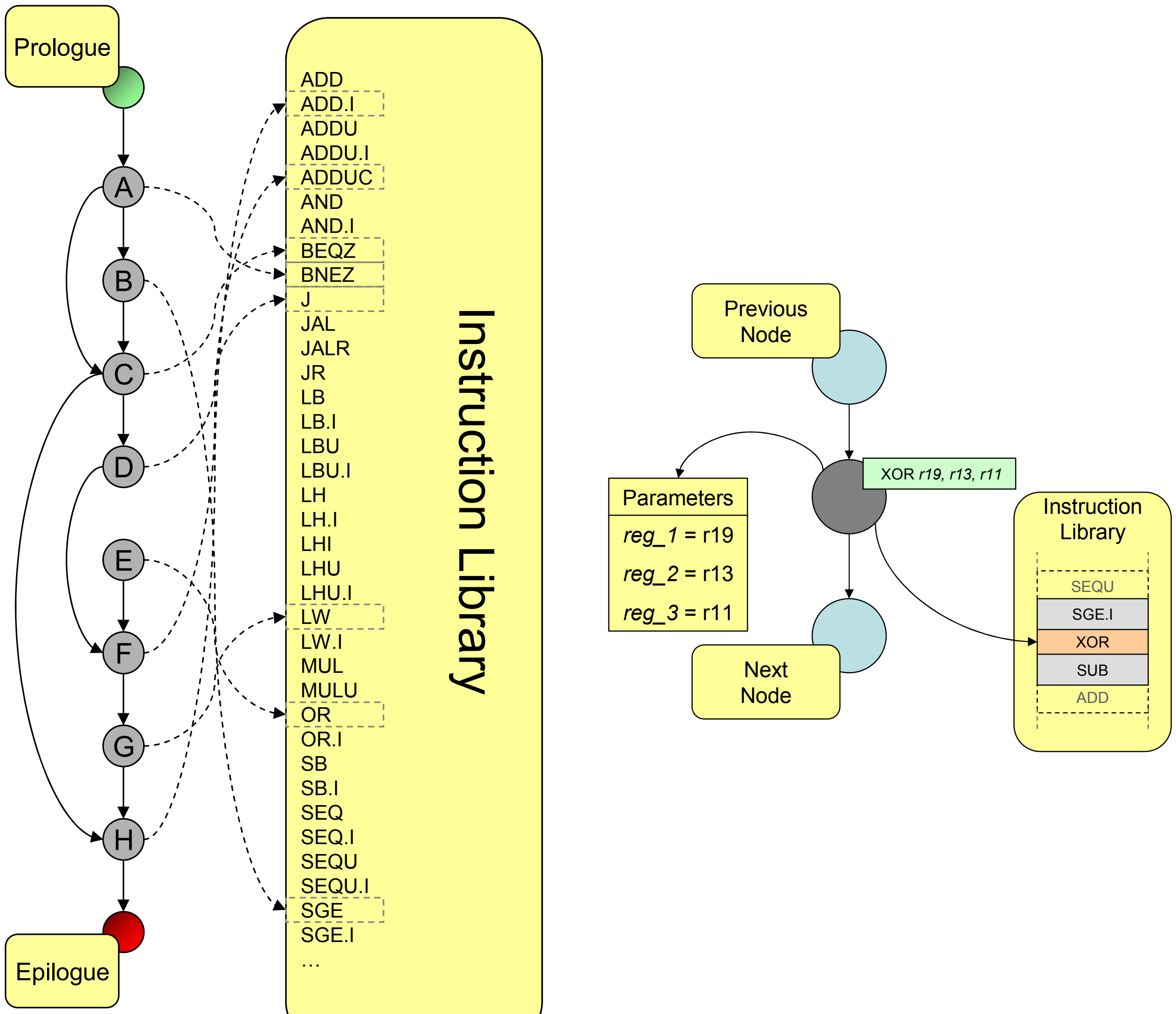

Figure 2: DAG Representation

Program	INST	CLK	SC[%]
arith_s	47	64	64.56
carry_su	23	63	65.17
except	78	108	75.18
fak	25	72	65.37
intrpt1	17	217	74.97
jump1	22	77	66.19
loadstore_s	129	174	69.25
loadstore_su	120	174	69.25
mul_su	15	93	74.67
set_s	107	144	64.45
set_su	205	144	64.45
div32	40	77	64.96
mul32	24	85	63.94
system01	199	267	74.36
all_instr	113	156	79.78
cumulative	1,164	1,915	80.59
random	9,190	9,690	77.12
µGP	812	1,084	94.59

Table 1: *DLX Statement Coverage summary*

Program	if [%]	dec [%]	exe [%]	mem [%]	wb [%]	regs [%]
arith_s	75.15	56.98	100.00	31.69	100.00	84.09
carry_su	75.15	58.31	100.00	31.69	100.00	84.09
except	84.24	68.51	100.00	49.30	100.00	96.59
fak	75.15	58.76	100.00	31.69	100.00	84.09
intrpt1	81.21	69.18	100.00	52.11	100.00	92.05
jump1	75.15	60.75	100.00	31.69	100.00	82.95
loadstore_s	81.21	58.31	100.00	52.11	100.00	84.09
loadstore_su	81.21	58.31	100.00	52.11	100.00	84.09
mul_su	84.24	67.41	100.00	49.30	100.00	96.59
set_s	75.15	56.76	100.00	31.69	100.00	84.09
set_su	75.15	56.76	100.00	31.69	100.00	84.09
div32	75.15	57.87	100.00	31.69	100.00	84.09
mul32	75.15	55.65	100.00	31.69	100.00	84.09
system01	84.24	66.74	100.00	49.30	100.00	96.59
all_instr	84.24	76.50	100.00	55.63	100.00	96.59
cumulative	87.27	91.57	100.00	59.15	100.00	97.73
random	87.88	69.84	100.00	49.30	100.00	96.59
µGP	88.48	96.01	100.00	90.85	100.00	96.59

Table 2: *DLX Statement Coverage breakdown*