VLIW – A Case Study of Parallelism Verification

Allon Adir, Yaron Arbetman,
Bella Dubrov, Yossi Lichtenstein,
Michal Rimon, Michael Vinov
{adir, arbetman, bella, yossil, michalr, vinov@il.ibm.com}
IBM Research Laboratory
Haifa, Israel

Massimo A Calligaro, Andrew Cofler,
Gabriel Duffy
{massimo-angelo.calligaro, andrew.cofler,
gabriel.duffy@st.com}
ST Microelectronics Design Center
Grenoble, France

ABSTRACT

Parallelism in processor architecture and design imposes a verification challenge as the exponential growth in the number of execution combinations becomes unwieldy. In this paper we report on the verification of a Very Large Instruction Word processor. The verification team used a sophisticated test program generator that modeled the parallel aspects as sequential constraints, and augmented the tool with manually written test templates. The system created large numbers of legal stimuli, however the quality of the tests was proved insufficient by several post silicon bugs. We analyze this experience and suggest an alternative, parallel generation technique. We show through experiments the feasibility of the new technique and its superior quality along several dimensions. We claim that the results apply to other parallel architectures and verification environments.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – *Verification*

General Terms

Design, Verification

Keywords

Functional verification, Processor verification, Test generation, VLIW, Parallelism

1. INTRODUCTION

Although high-level parallelism in processor design is not new, it recently hit the headlines when Intel announced that its future processors will include multiple cores to increase performance and limit power consumption[1]. At the same time that multi-cores reached the desktop, IBM announced that its massively parallel PowerPC-based computer, Blue Gene/L, is currently the fastest super computer in the world[2]. Although these news stories focus on the benefits, processor parallelism also results in difficult design and verification problems because of the exponential growth in the number of execution combinations. At this point, automation becomes critical for design and verification.

This paper presents our study of a relatively simple parallel technique in processor design. Very Large Instruction Word (VLIW) processors execute, in parallel, short sequences of basic instructions. The architecture provides increased performance, but shifts much of the additional complexity from the design to a software compiler that packs the basic instructions into VLIWs.

We show that even this type of processor architecture deteriorates the quality of tests created by a sophisticated test generator[5] and results in post-silicon VLIW related bug escapes. We found that various methodological enhancements are ineffective and that a parallel test generation technique is required to tackle the architectural parallelism. Although conceptually simple, the new technique is feasible only because of the limited nature of the parallelism and the existence of a powerful constraint solver. Because the new technique has not yet been used in full scale industrial verification, we study it through various experiments and demonstrate the quality of tests along several dimensions.

2. VLIW PROCESSORS

A Very Large Instruction Word (VLIW) architecture is a relatively simple parallel mechanism that enhances performance. The mechanism considers a short sequence of basic instructions as a single "large-word" instruction and executes the constituent instructions concurrently. The responsibility of packing basic instructions into the large-word instruction is left to a software compiler, allowing for a relatively simple hardware design. Examples of VLIW architectures include Intel's Itanium processor [1] and STMicroelectronics' ST100[1] and ST200 processor families [7][8]. We study a simplified version of the ST100 family, described in Figure 1.

In our simplified VLIW architecture there are four *slot-types* in the long-word instruction type; each slot-type allows some *instruction classes* to be used. For example, slot type ST0 allows any type of instruction from the LOAD, COPY and NOP instruction classes. Although, some VLIW architectures allow different numbers of slot-types and different bitwise lengths of the different slot-types, our architecture is simple with four constant length slot types. However, the VLIW-type we study imposes restrictions on the allowed

[1] The ST100 is not formally a true VLIW machine, but rather a scoreboarded LIW, in that it deals with virtually all of the data dependencies in hardware [7].

combinations of instructions. The reason for these restrictions is the limited number of execution units in the actual design. Therefore, in our example, the fact that there are only two LOAD/STORE units restricts the number of LOAD/STORE/COPY instructions in any VLIW-type to two.

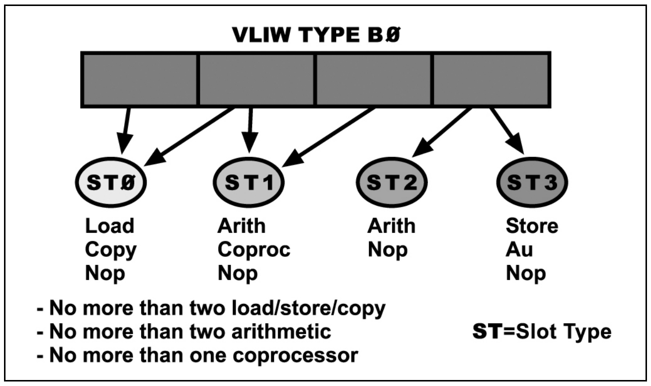
VLIW TYPE BØ
STØ
ST1
ST2
ST3
Load
Copy
Nop
Arith
Coproc
Nop
Arith
Nop
Store
Au
Nop
- No more than two load/store/copy
- No more than two arithmetic
- No more than one coprocessor
ST=Slot Type

Figure 1. Simplified ST100 VLIW type

Further restrictions involve data and resource dependencies within a single VLIW. Many VLIW architectures do not allow target-source and target-target dependencies within a VLIW type. However, the ST100 family lets the compiler pack instructions with target-source register dependencies into a single VLIW, when it is known that the register write precedes the register read.

3. SEQUENTIAL VLIW GENERATION

Test generation for VLIW should first create stimuli, which are essentially valid VLIW instances. Such instances should be of high quality; in particular, they should provide good coverage of the architecture and the design. In this paper, we compare two approaches to VLIW test creation. The first is a *sequential* approach, where instructions are created one after the other, and then packed into VLIWs. The second is a *parallel* approach, where a full VLIW is created at once.

The sequential approach was implemented using Genesys, a model-based test-generator from IBM [5] that was used by STMicroelectronics for the verification of ST100 family and ST200 family VLIW designs [6]. A Genesys system includes an architectural-definition model, which normally describes the architectural specification of single instructions. The verification team at ST Microelectronics augmented this model by adding, for each instruction, constraints that specify the cases in which the instruction positioning in a VLIW is illegal. An illegal combination causes either an undefined behavior that is prevented by the constraint or an exception. In order to avoid too many exceptions, if the selected instruction cannot be placed in the current VLIW position, Genesys may fail the instruction generation and try to select another instruction. After successful generation of four sequential instructions, the tool simulates the entire VLIW on a reference model.

This sequential implementation of VLIW generation gives acceptable results, as discussed in Section 5. However, the nature of this method is myopic and frequently fails with inter-slot constraints, resulting in exceptions, generation retries, or poor quality. For example, an illegal combination may occur if a specific instruction is requested for the final slot, because the generator is not aware of this request while selecting the early slots. Similarly, if a specific target register is required by the user for the last VLIW slot, the generator may select the same register as a source in an earlier slot, failing the instruction of the final-slot on an illegal source-target VLIW dependency.

4. PARALLEL VLIW GENERATION

The *parallel* approach is an alternative method that generates all the VLIW instructions together. It was implemented in Genesys-Pro, a high-end IBM test generator [9], by formulating the full VLIW as a single Constraint Satisfaction Problem (CSP) [10].

A CSP consists of a finite set of *variables* and a set of *constraints* between these variables. Each variable is associated with a set of possible values, known as its *domain*. A *constraint* is a relation defined on a subset of these variables and denotes valid combinations of their values. A *solution* to a constraint satisfaction problem is an assignment to each variable a value from its domain, such that all the constraints are satisfied. Each CSP is represented by a *constraint graph* whose nodes are variables and whose arcs are constraints.

A single VLIW is generated in three steps. First, the generator selects the instruction types to put in each of the VLIW slots. This problem is formulated as a CSP [10] that represents the VLIW-type as a tree. Each possible association of a slot type to a VLIW position is represented as a node in the CSP. The CSP variables defined for each slot type are: the instruction, the bit length of the slot, and its selected position within the VLIW. Constraints between these variables reflect VLIW combination restrictions and slot lengths restrictions for the instructions.

Figure 4 shows the CSP for the VLIW-type of Figure 1, and depicts a possible solution. In our simple VLIW architecture slot sizes are always 16 bits, thus the corresponding CSP variable has two possible values it its domain -- 0 and 16, where 0 indicates that the slot type was not selected for the VLIW. In a variable length VLIW architecture, there may be more possible lengths.

The second step in parallel VLIW generation involves a simultaneous selection of property values for all the instructions that comprise the VLIW, by solving a second CSP. The properties of each of the instructions are organized in a tree structure, with a node for every resource used by the instruction. This results in a large CSP that may be difficult to solve. In particular, the number of constraints that handle the dependencies between properties of different instructions grow with the square of the number of slots in the VLIW. In practice, the number of slots is limited by the number of hardware execution units and thus remains small.

The third step in this algorithm is simulation of the generated VLIW on the architecture reference-model.

5. EXPERIENCE WITH SEQUENTIAL GENERATION

The sequential test generation, described in Section 3, was used by ST Microelectronics for the verification of a processor from the ST100 processor family[6]. The weaknesses of the sequential approach required a carefully staged methodology and manual direction of the test generation process. The staged methodology included four phases. After the instruction-level verification had stabilized, the pre-fetching and decoding mechanisms were tested. The goal was to generate all legal VLIW instruction combinations and to cover all VLIW types. Correct execution, intra-VLIW data dependency rules, exceptions, and undefined results were all ignored. The second phase started once the Pre-fetch/Decode unit correctly handled all VLIW instructions. The goal was to verify the parallel execution of instructions in the VLIW. The execution pipeline was first verified with legal operand dependencies. Then the processor's ability to recover from undefined results due to illegal dependencies was tested.

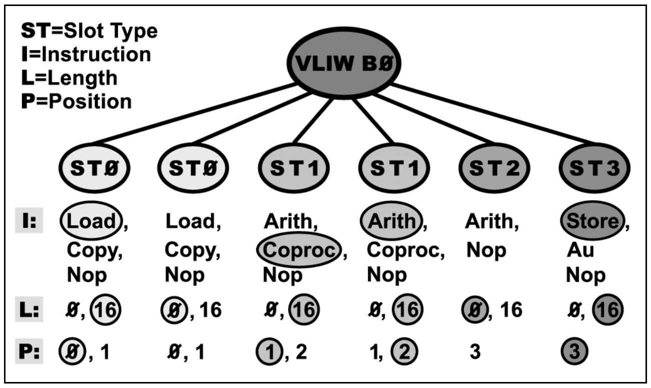

Figure 4. Instruction-combination CSP

In the third stage, the interaction of the VLIW mechanism with the rest of the design was checked. VLIW instructions were injected randomly into tests that target other mechanisms. For example, the execution of a hardware loop instruction was tested in conjunction with a VLIW jump instruction, randomly placed inside the loop body. The fourth and final stage consisted of massive random VLIW test-generation.

In addition to the staged methodology, the verification team at ST-Microelectronics found it necessary to implement an instruction selection mechanism, by directing Genesys to create specific instruction sequences that correspond to legal VLIW combinations. Employing the user test-requests and macros of Genesys, each VLIW type was explicitly defined. The verification team spent about 12 person weeks in writing 382 such macros. During the first phase of VLIW verification, this set of macros resulted in reasonable test quality, although with limited randomness. However, for the following stages an unwieldy number of macros was required. For example, within the VLIW, a source-target data dependency is not generally allowed, but the target of a LOAD instruction may be used as a source of an arithmetic instruction. Testing this with each possible VLIW combination would have required thousands of macros, so only partial testing was performed. Furthermore, massive VLIW random testing was impossible with the sequential test generation method resulting in several post-silicon escape bugs. Specifically, scenarios that combined hardware loops with VLIW contained bugs that should have been found on the third stage.

6. EXPERIMENTING WITH PARALLEL GENERATION

The VLIW parallel test generation method has not been used yet in full scale industrial verification. In order to compare it with the sequential method, this section describes an experiment with a simplified four-slot VLIW architecture. We encoded five VLIW types, and several sample instructions from each instruction class, to get a VLIW with 116 legal combinations. Genesys-Pro [9] was used to implement both the sequential and parallel generation methods to neutralize differences between test generators.

The simplest measure of test quality we consider is the number of NOP instructions in a VLIW. When a VLIW constraint cannot be satisfied, the generator selects a NOP for the problematic slot. Generating about 50,000 VLIWs with the sequential method, we found that one third of VLIWs included one NOP, almost half the VLIWs included two NOPs, and 8% included three NOPs in the four slots. Only 7% of the VLIWs did not include any NOP at all. In contrast, the parallel method, did not generate NOP instructions in any of the 50,000 VLIWs[2].

Turning to coverage, Figure 7 shows that the parallel method achieved full coverage of the 116 combinations, by creating about 750 VLIW instructions. In contrast, the sequential method needed about 6,250 VLIW instructions to get full coverage. However, the parallel method is slower in generating each VLIW. This is because it takes longer to solve the two CSPs representing a full four-slot VLIW, than to solve four simple CSPs each representing a basic instruction. The combination of fast coverage and slow CSP results in a faster parallel method—it is about four times faster than the sequential method in achieving full coverage.

Figure 10 measures the density of generated VLIWs. In our simplified architecture, there are 116 combinations. A perfectly uniform generator would create 1/116 of all the VLIW instances for each of the combinations. The same tests used to measure coverage have been analyzed to check uniformity. As shows, the parallel method is quite uniform—most combinations were generated around the 1/116 mark, just below the 1/100 line. On the other hand, the sequential method is considerably less uniform. Uniformity in test generation is important because it allows the allocation of verification resources to various tasks. Non-uniform generation may give preference to easier tasks.

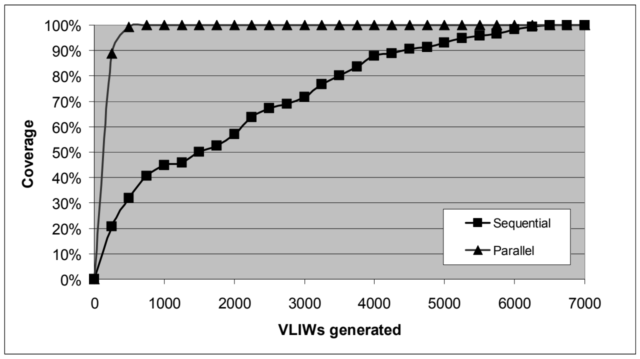

Figure 7. Coverage of all VLIW combinations

A more subtle measure of test quality in a parallel context is the coverage of collision events within a VLIW. To demonstrate this aspect, we divided the memory space into two regions, read-only and read-write, experimenting with various division ratios. We generated random VLIWs with various LOAD, STORE and other instructions with a bias towards memory collisions of accesses to common cache lines within a VLIW. Figure 11 summarizes the results, in particular, the few collisions created by the sequential generation and the fair number of collisions created with the parallel method. The parallel approach selected addresses by solving all the constraints in a single VLIW, including the bias ones. The

[2] In our experiment, we directed both generator versions to avoid NOP instructions as much as possible. However, a full verification process should include some NOP instructions.

sequential algorithm often failed, for example, to select an address in the read-write memory region for a LOAD in the first slot, such that the same address could also be used by a STORE in the final slot of the VLIW. This became more difficult for the sequential method as the read-only region became larger, whereas the parallel approach retained a high success rate. Although only a synthetic experiment, it shows the method's ability to test complex parallel mechanisms by memory collisions—a critical ability in preventing post-silicon escape bugs, as reported in Section 5 and [3].

7. CONCLUSIONS

This paper reports on the verification of the parallel aspects in a VLIW processor. We focused on the difficulties in test generation, described a methodology that combines a sequential test generator with manually written macros, and reported its drawbacks. In response, we described a parallel test generation technique and reported on experiments that demonstrate its superiority along several dimensions of test quality.

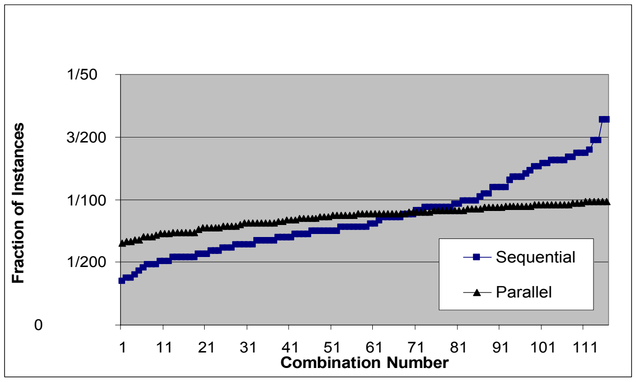

Figure 10. VLIW combination density

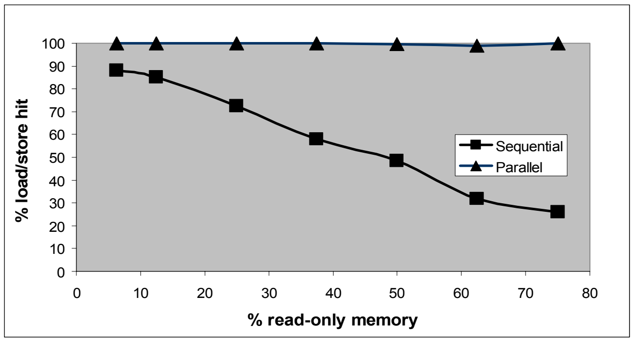

Figure 11. Memory collisions

Our results are applicable to the verification of other types of parallelism in processor architectures, such as Superscalar, Multi-threaded, and Multi-processor designs[4]. In particular, our conclusion that manual enhancements to sequential test generation are bound to fail, is general enough. When subtle scenarios and test quality constraints are required, the degree of parallelism combined with the number of quality constraints becomes large enough to make the number of combinations unmanageable for simple programming tools. Fundamental solutions are then required. Such solutions depend on the power of the underlying solvers. For the VLIW architecture studied, we found that the solver [9] is strong enough and the slowdown in its performance is acceptable. However, higher parallelism will entail more powerful solvers.

This paper contributes by presenting a new test generation technique and by demonstrating its feasibility to VLIW. We show the superiority of the new approach by several quality measures including coverage, uniform test density, and achieving high resource contention. However, this study is limited by IBM's unique test generation environment. Conducting similar experiments with standard industry verification environments [1],[12] would test how well these tools are suitable for the verification of parallel design constructs.

8. REFERENCES

[1] http://www.intel.com/pressroom/archive/releases/20040907corp.htm

[2] http://domino.research.ibm.com/comm/pr.nsf/pages/news.20041110_bluegene.html

[3] Adir, A. and Shurek, G. Generating Concurrent Test-Programs with Collisions for Multi-Processor Verification. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT '02)*, 2002, 79-82.

[4] Sullivan, M., Wilson, P., Montemayor, C., Evers, R. and Yen, J. Multiprocessor Design Verification with Generated Realistic MP Programs. In *Proceedings of IEEE 14'th Annual IPCCC*, 1995, 389-395.

[5] Aharon, A., Goodman, D., Levinger, M., Lichtenstein, Y., Malka, Y., Metzger, C., Molcho, M. and Shurek, G. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proceedings of 32nd Design Automation Conference (DAC '95)*, 1995, 279-285.

[6] Malandain, D., Palmen, P., Taylor, M., Aharoni, M., Arbetman, Y. An Effective and Flexible approach to Functional Verification of Processor Families. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT '02)*, 2002, 93-98

[7] http://www.st.com/st100

[8] Homewood, F., Faraboschi, P. ST200: A VLIW Architecture for Media-Oriented Applications. *Microprocessor Forum*, Oct 2000.

[9] Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M. and Ziv, A. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design & Test of Computers*, Mar-Apr. 2004, 84-93.

[10] Bin, E., Emek, R., Shurek, G. and Ziv, A. Using Constraint Satisfaction Formulations and Solution Techniques for Random Test Program Generation. *IBM Systems Journal*, Aug. 2002, 386-402.

[11] Palnitkar, S. *Design Verification with* e. Prentice Hall, 2003.

[12] Haque, F., Michelson, J. and Khan, K. *The Art of Verification with Vera*. Verification Central, 2001