
Verification of VIRAM1

by Samuel W. Williams

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor David A. Patterson
Research Advisor

(Date)

* * * * *

Professor Alberto Sangiovanni-Vincentelli
Second Reader

(Date)

Verification of VIRAM1

Samuel W. Williams

M.S. Report

Abstract

VIRAM1, the first incarnation of the VIRAM ISA, is a 130M transistor, 325mm², vector processor, with embedded DRAM designed to produce 12.8GOPS (16b), 12.8GB/s of memory bandwidth at 200MHz, and consume less than 2W. This report presents the series of verification strategies used to ensure that such a massive and complex design could be realized, from ISA to tape-out, by four hardware designers, in only three years. The four methodologies used were: IP blocks for memories and the core, partitioning of the design into fixed-spec, easy to construct and verify sub-blocks, correct-by-construction design methodology, and abstraction of tests. This abstraction minimized the coding effort, maximized readability, allowed for avoidance of known bugs, and allowed a massive number of different code generations to be generated to explore the instruction sequence space.

Contents

- 1. Introduction to IRAM and VIRAM1**
- 2. Introduction to the Verification Environment**
- 3. VSIM-P Verification**
 - 3.1 Trace Generation and Comparison**
 - 3.2 Random Testing**
 - 3.3 Putting it All Together**
- 4. VSIM-ISA Verification**
- 5. Abstraction and Evolution**
 - 5.1 Abstraction and Primitives**
 - 5.2 The Initial Testsuite**
 - 5.3 Changes in the Project and the Problems Which Arose**
- 6. Basic RTL Verification**
- 7. Partitioning – RTL, IP, and Custom Block Verification**
 - 7.1 Custom Block Verification**
 - 7.2 Floating-Point Unit RTL Verification**
 - 7.3 Vector Unit RTL Verification**
 - 7.4 VIRAM1 RTL Verification**
- 8. ISA Evolution and Verification in the Presence of Unspecified Functionality**
 - 8.1 Coping with ISA changes**
 - 8.2 Vector Unit Issues Which Caused False Failures**
- 9. Back-end Flow Verification**
- 10. Related Work**
- 11. Conclusions**

1 Introduction to IRAM and VIRAM1

IRAM is a research project at the University of California at Berkeley that investigated processor design of a low-power gigascale system-on-a-chip environment. The ever-increasing gap between logic and DRAM speed has resulted in designers adding prodigious SRAM caches to chip designs. The alternative solution is to place DRAM on chip resulting in more than eight times the memory per unit area. This method also allows for a significant memory bandwidth, just as a cache would. Mobile products, multimedia devices, as well as massively-multiprocessor systems, and compute farms require power efficiency to maximize battery life or even to help minimize the total cost of ownership. IRAM designs with a large on-chip DRAM memory would easily fall into one of these categories, necessitating that the microarchitecture be power-efficient, and furthermore, that the system architecture ideally should be implicitly power-efficient. These constraints helped focus the project into a series of design refinements and in turn into a prototype processor.

Keeping with the initial goal of a low-power, high-performance processor to be included in parallel systems, the IRAM concept design – VIRAM [Koz99] - was based on a vector architecture. To provide network connectivity, four gigabit links were built into a network interface (NI). To satisfy the voracious memory bandwidth appetite of a vector processor, a sufficient number of DRAM macros were embedded on chip. In addition, this guaranteed predictably low latency. In order to avoid having to design an entire vector processor from the ground up, including all the system and compiler support, it was partitioned along the core/coprocessor model, where the core would be a FP enabled MIPS compliant IP core, and the coprocessor would be designed by UCB to perform all vector operations. This view of the architecture would evolve over time to better match a moving target and to fit available IP and human resources (averaging only four people) and a limited timeframe. Figure 1 illustrates the basic VIRAM architecture. The first incarnation of the VIRAM concept, VIRAM1, which chose specific values for memory (eight 13Mb DRAM banks), computational elements, and Instruction Set Architecture (ISA) specific issues, was completed in August of 2002 and mask construction began in October. The first wafer was delivered to Berkeley in February 2003.

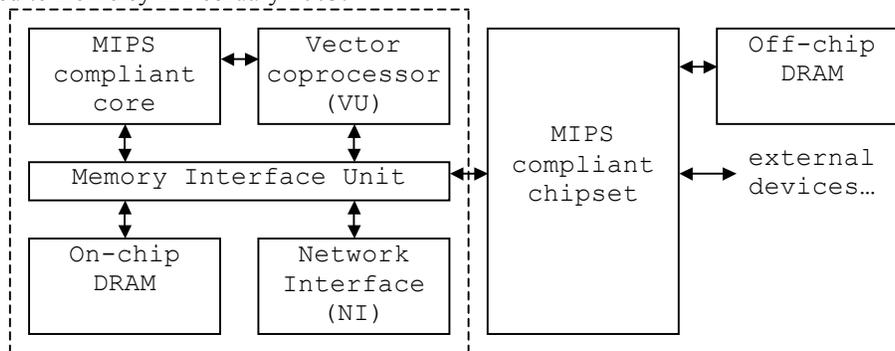


Figure 1 - Initial VIRAM System Architecture Overview

The system-on-a-chip component of a VIRAM system (dotted box) embeds a MIPS core, a compliant vector coprocessor, on-chip DRAM, on-chip memory-mapped devices, and an interconnection network.

The coprocessor (vector unit) was designed with “ease of design” and “easy scalability” in mind. In order to add flexibility, an instruction set architecture (ISA) [Mar00] was constructed around the virtual processor (VP) model where the n bit functional units are partitioned into virtual processor widths (VPW) of 64, 32, 16 or possibly 8 bits. To simplify the design, functional units, as well as vector registers were partitioned into 64b granularities, thus avoiding having a single functional unit or register file provide access and computation on all elements in a vector register. As a result, a multiplier need only operate on a single element in $vpw64$, and four in $vpw16$. Similarly, a vector register file now contains only 64b registers. Although not immediately obvious, this is a more implementation-friendly method, as only one instance, which is a partition of a functional unit, needs to be designed, verified, and implemented. A functional unit is bit sliced, 64 bits in the final implementation, into instances of these partitions. Extending this partitioning methodology across multiple functional units and a large vector register file leads to identical blocks called vector lanes. Within this block, several slices are lumped together. However, there is still a single centralized control module for decode, pipelining, chaining [HP96], configuration, address calculation, and memory control. This control module is basically a simplified core without datapaths or register files. Even without these large structures, it is about the same size because it controls so much. Figure 2 illustrates the vector unit, which is a MIPS compliant coprocessor. Neither the maximum vector length nor the widths of the functional units are defined in the ISA. As a result, an implementation may vary these to find a balance between area and performance. Obviously, both represent near linear changes to area, but subtle and an implementation specific change in cycle time.

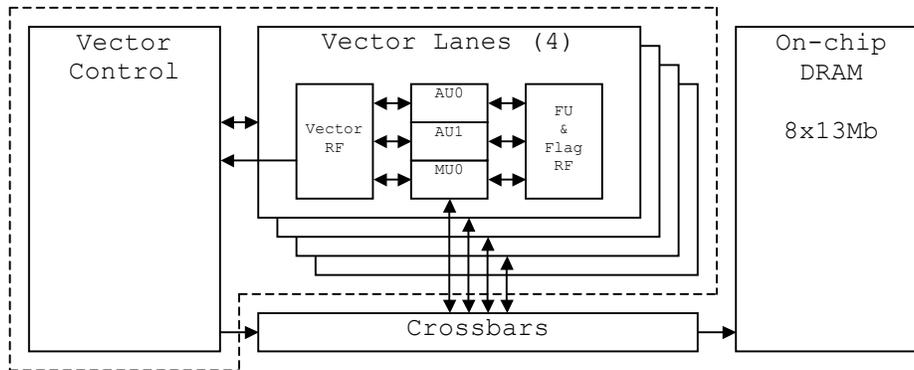


Figure 2 - VIRAM1 Vector Unit

The vector coprocessor in the VIRAM1 system (dotted line) is partitioned into a control unit, and four vector lanes. Each lane has a 64x256b register file, 2 arithmetic units, a memory unit, a flag functional unit, and a flag register file.

Originally, no separate FPU was required since the core would be floating-point capable. Unfortunately, by early 2000, it was clear that we would not be able to procure this core, resulting in the selection of an alternate MIPS ISA core. More over, this new core did not support floating-point operations. This deficiency necessitated the development of FPU. We decided to use a single precision floating-point execution unit IP block to minimize the design time. However, this soft IP FPU only supported computation; it contained no registers, decoding, or interface. Figure 3 shows the architectural

part of the FPU, which is a module we designed ourselves. Now that the basic architecture of VIRAM1 has been described, the rest of the paper details the verification flow from software simulators to layout.

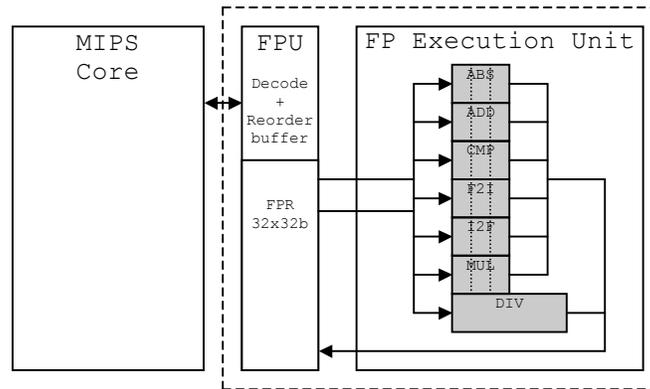


Figure 3 - VIRAM1 Floating-Point Unit

The Floating-point coprocessor (dotted line) is attached to the MIPS core, and contains three major blocks: a soft IP execution unit, a macro-based register file, and an in-house architectural control RTL block.

2 Introduction to the Verification Environment

The verification work required was originally described as: show that both the vector unit (VU) and the network interface (NI) are correct. VU verification only pertained to ISA compliance. As a result, timing verification was included either in the workload of the designer or whoever would run the blocks through place and route (PnR) tools. The other blocks, DRAM and the core, were IP blocks, and were thus assumed to be correct. Of course, this simplification did not mean that verification stopped with the VU. Full-chip verification was also required. The majority of verification for the network interface could be performed on a standalone environment, since it had no ISA dependence. However, because it is a memory-mapped device, it still must be included at the top-level verification.

The goals of verification, aside from the obvious goal of producing a correct design, were to balance the following: testsuite design effort, machine requirements for full chip verification, effective use of the tools by the RTL designers, and finally, evolvability of the testbench and tools so that it can adapt to changes. Only one person would be responsible for verification, and tape-out was initially only six months away. Thus, the testsuite and testbench had to be extremely easy to implement as well as requiring relatively few CPU days to run. It was the responsibility of the RTL designers to fix their own code after bugs have been discovered, necessitating the tests be easily readable, and the tools user-friendly and intuitive. Finally, since it is difficult to foresee every eventuality, the testsuite and tools must be designed so that they can grow and evolve.

The environment at the beginning of verification included only software simulators. Over time, this toolbox would grow to include more realistic software simulators, hardware simulators, and formal

verification tools. The original software simulators were *vsim-isa* (VIRAM ISA simulator) and *vsim-p* [Fro00] (VIRAM performance simulator). The performance simulator expanded on the basic (one “cycle”/one instruction) functionality of the ISA simulator by attempting to accurately simulate the actual VIRAM1 microarchitecture. Initially it shared no common code with the ISA simulator, and both simulators had apparently been used to run small kernels leading to the assumption that they were functionally correct. Eventually another software simulator had to be written to reflect changing microarchitecture, memory organization, ISA issues, and the need to model kernel mode activity. Some hardware simulators were based on VCS, and others using Spice or TimeMill or eventually Nanosim. VCS simulators eventually included the integer unit (multiply, add, shift, round, and saturate) testbenches, the crossbar, the MIPS core, the MIPS core with a FPU, the MIPS core with the vector unit, behavioral crossbar, behavioral register files, and memory, and finally the full VIRAM1 environment. Spice and TimeMill testbenches included benches for the vector register file (RF), the crossbar (XBAR), and the integer unit modules (IU).

In August of 1999, VIRAM1 was organized as follows. The core, which we planned to use, was from SandCraft, and although it was to include floating-point support, it was to be full custom, and had not yet been delivered. The VU was being designed in-house (RTL took more than a year to write) and included full custom integer units, which were under construction, and a full custom crossbar, also under construction. VIRAM1 tape-out had been initially set for January of 2000. The full custom vector register file, although part of the vector unit, was to be designed by an external source. IBM provided DRAM IP, and the NI was to be written in-house.

With the initial environment and status described, the rest of the paper follows the development of the final version of the VIRAM1 verification testbenches and flow, as well as illustrating the misconceptions and catastrophes that required major changes in the approaches used. The following sections of the paper are formatted as a cycle of: concept for verification, description of the method, its good points, and finally why it fails to be a viable verification method.

3 VSIM-P Verification

As previously stated, the initial verification timeframe was about six months. This necessitated a quick and dirty method. The ISA simulator was assumed to be correct, but it presented a large problem for verifying a temporally complex out-of-order system. The hope was to use the touted cycle-accurate performance simulator to verify the correctness of the RTL via trace comparison. Since the performance simulator (*vsim-p*) did not share any common code with *vsim-isa*, or any code with the RTL, it needed to be verified before RTL verification could proceed. In this method (the first of many), *vsim-p* was first verified by comparing it to the “known to be correct” *vsim-isa*, and then would be used to verify the RTL. Verifying *vsim-p* was not an exact science since simple trace comparison was not applicable. Instructions

can commit out-of-order. When coupled with the fact that unreflected latencies are included, the two traces might look very different. A formal method was not used for several reasons, paramount of which was the time to develop such a strategy.

3.1 Trace Generation and Comparison

Verification of *vsim-p* is checking functional equivalence to *vsim-isa*. Both simulators were modified to produce a verification trace and a trace comparator was written. This trace was slightly different than the original trace provided by the simulator framework. In effect, the comparator is an out-of-order ISA comparator. That is, it checks to ensure that *vsim-p* is consistent with *vsim-isa* on a per element basis. This verification method is just correct by association, so it is necessary to show that at least *vsim-isa* has some basic functionality. Accordingly, a small set of directed tests were written and simulated.

An example will likely make these concepts clear. Figure 4 shows a series of instructions from a detailed ISA level trace. In this case, initially, vector length is set to nine. This is followed by a *vfset* instruction, which is used to set *vfmask0* (register 0 of the vector flag register file). It was already set for *vl=mv1*. It should be noted, that there is one flag bit for each VP. The simulator supports 8b VPW's, and thus with a 2048 bit vector register (default for VIRAM), flags are 256 bits (2048/8=256). VIRAM1, however, does not support such a virtual processor width, and thus simulators present vector flags with twice as many bits as hardware simulators. Finally, a vector add is executed. It is clear which elements were changed, and that integer overflow occurred in three 64 bit elements. Address translation is used to determine the physical addresses in memory, which in this case, were on-chip DRAM. The instructions are then fetched, decoded, and executed in a single iteration. This information, and similar traces from other simulators, must be parsed and compared by the comparator.

In this case, all software simulators would produce code in this order, but independent instructions might occur out-of-order. For example, it is possible for the scalar core to run ahead of the vector unit, thus a vector add followed by a scalar add in program order could be reversed in execution order and therefore in trace order. In a case more critical to correct verification, two mutually independent vector add's could execute out of order in the presence of the multiple integer units that VIRAM1 has. Furthermore, since there is no reorder buffer in the vector unit, instructions commit in element groups leading to multi-cycle commit. Thus it is clear that for a design like this, not only must traces be processed on a per register basis, but also a per element basis. In figure 5, two vector add instructions each write elements on each of four cycles to the same register, via chaining, followed by a write from an add in the scalar core. The second instruction presumably uses a conditional execution mask so that it only writes 14 elements.

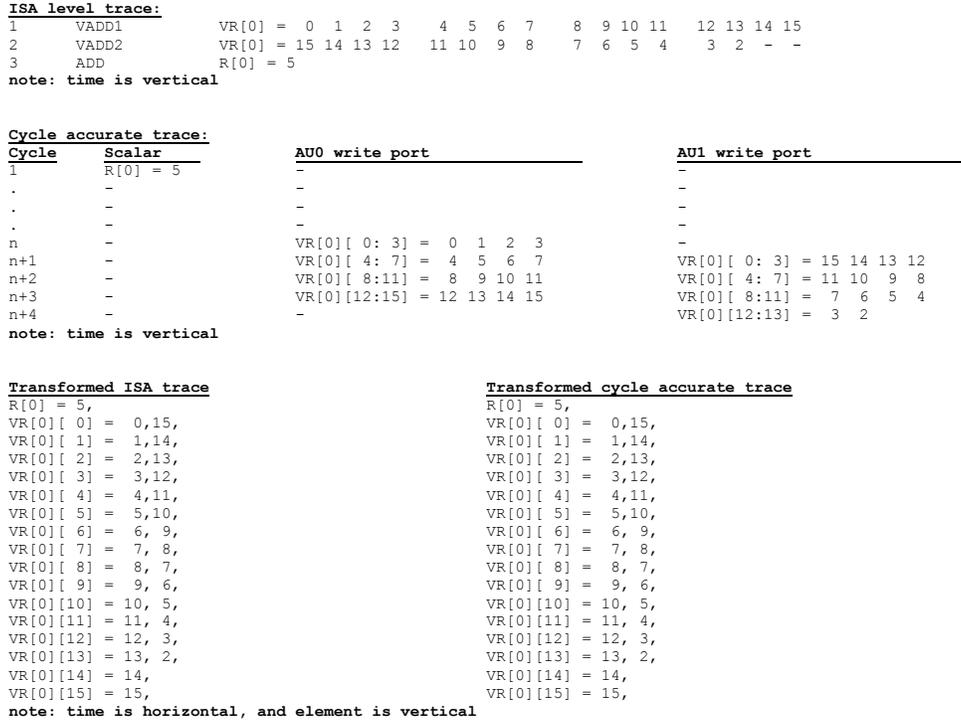


Figure 5 - Multicycle Out-of-Order Commit Trace Comparison

This figure details 4 views of trace comparison: ISA, cycle accurate, and transformations of those two into an internal representation. Transformation can change a very complex problem into string comparison on thousands of strings.

At this point it is clear that after the element group transformation, all that is required is to traverse the list of comparable elements and examine their result traces. Simple string comparison can be performed to determine if an element in the cycle-accurate domain is consistent with the ISA level domain. With the addition of instruction and simulation time, it is not too difficult to determine and report the exact point at which a failure occurred. This additional check was performed and was found to be extremely beneficial by those who debugged various blocks.

3.2 Random Testing

Given the impending deadline, it was necessary to expedite verification time, so a relatively simple random test generator (RTG) was written. The RTG is a one-pass code generator. It does not simulate any instructions, since it would be necessary to verify its functionality in addition to that of *vsim-p* and *vsim-isa*. To make it more configurable, each class of instructions was given a probability to be generated. This probabilistic code generation scheme also applied to primitives - collections of instructions, which perform some operation or test an important sequence/combination. In addition there were configuration variables to allow certain types of exceptions, for example arithmetic (vARI), illegal use of instruction (vIUI), or invalid vector length (vIVL). Setting these variables did not guarantee that

these exceptions would be raised, but instead removed any guards that prevented them from occurring. Where an arithmetic exception can be avoided by clearing the exception enables, an IUI or IVL exception requires keeping track of some of the most critical vector control registers such as VL and VPW. All of these configuration variables could be included in a file, which could be passed to the generator. A simple script was written to run the random test generator, to run both simulators on the produced code, and finally to run the trace compactor to determine whether or not the performance simulator had failed. The script would then repeat this process until it had found ten failing tests. The configuration variables are detailed in Table 1.

| WEIGHTS | | CONTROLS | |
|--------------|--|----------------|--|
| VLD_WEIGHT | <i>Relative weight for vector load (all forms)</i> | MAIN_INSTR | <i>Number of primitives</i> |
| VINT_WEIGHT | <i>vector integer</i> | SUB_INSTR | <i>Primitives in a subroutine</i> |
| VFP_WEIGHT | <i>Vector floating-point weight</i> | SUN_IN_SUB | <i>Allow subroutines to call other subroutines</i> |
| VCVT_WEIGHT | <i>Vector convert weight</i> | ALLOW_IUI | <i>Illegal use of Instruction</i> |
| VST_WEIGHT | <i>Vector store (all forms)</i> | ALLOW_ADA | <i>Address alignment</i> |
| LD_WEIGHT | <i>Scalar Load (all forms)</i> | ALLOW_ARITH | <i>Arithmetic</i> |
| ST_WEIGHT | <i>Scalar Store (all forms)</i> | ALLOW_IVL | <i>Invalid vector length</i> |
| FORS_WEIGHT | <i>For-loop start weight</i> | VSYNC_LEVEL | <i>0 = never 1 = best guess 2 = always</i> |
| FORE_WEIGHT | <i>For-loop end weight (must be greater than start)</i> | DEBUG_LEVEL | <i>Allows debug information to be commented to program</i> |
| IFES_WEIGHT | <i>If-then-else start weight</i> | MIT_DP | <i>Switch between original FPU spec and the actual datapath from MIT</i> |
| IFEE_WEIGHT | <i>If-then-else end weight</i> | ALLOW_CoPinBDS | <i>Allow for coprocessor instructions in a branch delay slot</i> |
| JSR_WEIGHT | <i>Jump subroutine weight</i> | NUMBER_OF_VRs | <i>Number of vector registers</i> |
| CPLX_WEIGHT | <i>Complex primitive weight</i> | NUMBER_OF_VSs | <i>Number of vector scalar</i> |
| VCVS_WEIGHT | <i>Vector control / vector scalar</i> | NUMBER_OF_FPRs | <i>Number of floating-point registers</i> |
| VF_WEIGHT | <i>Vector flag processing</i> | | |
| FP_WEIGHT | <i>Scalar floating-point</i> | | |
| CHAIN_WEIGHT | <i>Chaining primitive (select source read class, destination write class, instructions from each class, a dependent register, and fill a number of intervening cycles)</i> | | |

Table 1 - Random Test Generator Parameters

This table details the most often used configuration variables parsed by the random test generator. There are three basic classes of variables: weights, switches, and architectural configurations. Weights are designed so that the probability of generating a primitive of the corresponding class is equal to the weight of the primitive divided by the sum of weights of all primitives.

3.3 Putting it All Together

Figure 6 illustrates the resulting design flow. The abstractions of the processor, in the form of software simulators that model the ISA and predicted performance, are each executed using either handwritten tests or random tests. For the initial runs, traces were visually inspected to ensure that the ISA simulator was correct. After basic functionality was confirmed, random test generation and trace comparison was used exclusively.

Almost every failure from this method was due to a bug in the performance simulator. The bugs varied from simple functionality and computation, to severe hazard violations, to causing the simulator to crash. The test code was not bad or poorly written, but was randomly generated code that adhered to the programming semantics set forth by the ISA. In some ways this was good, since running handwritten code, designed for kernels, on the simulators is not a particularly thorough verification method. This failing was obvious, since other designers had run many kernels without incident. In many ways running a random test generator is very useful in catching non-intuitive bugs. Getting complaints like “I would never write code like that” is not necessarily bad if the code is still valid and adheres to the programming semantics, and it is quite useful if it actually finds bugs.

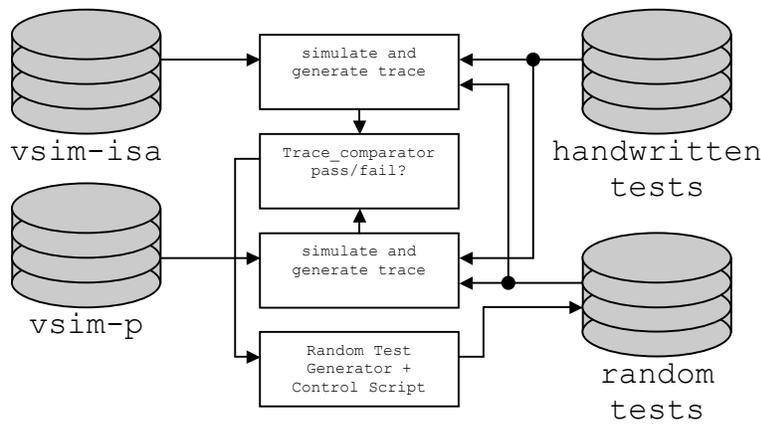


Figure 6 - Initial test flow

Simulator source (left) is compiled and executed using stimulus in the form of assembly language programs (right). Each simulator produces a trace. They are compared to determine whether or not the performance simulator is consistent with the ISA.

More troubling was the fact that not only did the performance simulator have bugs, but so did the ISA simulator. Unfortunately, these bugs were primarily functionality bugs. It was clear that the ISA simulator had not been verified to any extent, and the only reason any bugs were found was that the functionality code was written by two different people, and thus there was a good chance that at least one of them got the code right. All of the functionality issues were seen in the instructions or configurations not often used by developers who wrote relatively simple code.

Eventually, in order to avoid many common bugs, a common set of libraries was written for both simulators to use. This code reuse of course ensured that they would always both produce the same results,

based on functionality, but it did not guarantee that both simulators would be correct. As a result, the random test regressions stopped finding comparison bugs, but continued to find major failings due to chaining, segmentation faults, or assertion checks in the performance simulator.

The use of a common library in conjunction with the already existent ISA bugs was the death knell for this method. The execution of *vsim-p* should always be same as *vsim-isa*, but they might both be wrong – and ever-present bugs in *vsim-p* exacerbated the problem. It was clear that if verification would ever proceed to RTL, it would be necessary to ensure that, from the ground up, each abstraction level or representation of the architecture (ISA/Performance/RTL/gate) was correct.

4 VSIM-ISA Verification

The second method applied to verification of the chip attempts to address the major shortcomings of the first (simple trace comparison) by adding another verification step, *vsim-isa* verification. Since the ISA level simulator is the lowest level (closest to the ISA document), there is nothing to compare against but the document itself. A possible method would be to use equivalence checking to prove that the equations found in the document are identical to the code in the simulator. This did not address the need to prove that the higher levels of the design also conformed to the ISA without having to rely on associativity and trace comparison. Furthermore, equivalence checking could neither be used to prove that the ISA document was actually correct, nor be implemented in the time allotted.

The solution chosen was to write an extensive set of self-checking programs, which would start from trivial pieces of code and evolve into extremely intricate and thorough programs. Each test would contain code to determine whether or not the results it produced through computation were what would be produced based on the algorithms presented in the ISA document. A final pass/fail value would be produced as an exit code that would allow the verification engineer to know whether or not the simulator was working correctly. The beauty of this approach is that these tests could also be run directly and independently on the performance simulator and the RTL simulators without having to rely on trace comparison. Trace comparison, however, could still be used as an added measure of confidence. Figure 7 illustrates the basics of this method.

Each of the initial tests was written in assembly language and designed to test only a small piece of the ISA (a single instruction initially). As tests became more complex, the time required to write tests skyrocketed. This severe time consumption in conjunction with the immensity of the ISA and with the fact that handcrafted assembly language tests would need to be completely rewritten in the event of ISA changes, necessitated refinement of this verification strategy. Some facilitation method was required to ensure that this step could be completed in a timely manner and still allow for growth and evolution of the budding test suite.

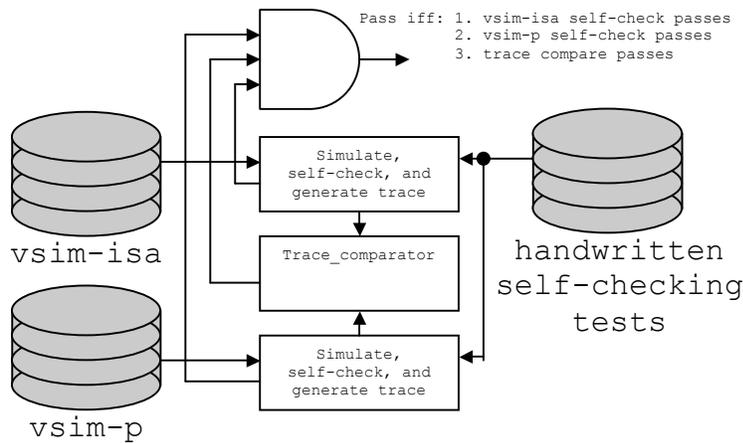


Figure 7 - Self-checking test flow

This verification flow adds an additional step to that seen in Figure 6. Each test now includes code to test the result of various instruction sequences to determine if the simulator produced the correct result. If not, the test will force the simulator (through test code) to produce an exit code that signals an error.

5 Abstraction and Evolution

The third method, which initially seemed only to be a refinement of the previous method, grew rapidly and formed the basis for all ISA level verification for the design. In this method, tests were written in a language of test primitives, and a translator was used to generate assembly language appropriate for the current ISA, test conditions, and so forth. Once the now self-checking assembly language had been produced, it was run on the various simulators, and the simulator traces were also compared to determine which fail and where.

5.1 Abstraction and Primitives

These test primitives included not only plain assembly language, but also initialization instructions, test instructions, macroinstructions, configuration variables, custom kernel configuration, self-generating code (written in C), and test restrictions. Plain assembly language was included to ensure that tests would contain exactly the instruction sequence to be tested. Thus, unlike a compiler, the writer is guaranteed that the code desired for verification is generated. All other components of a self-checking test were greatly abstracted to expedite coding time, as well as allowing for mapping to any desired set of instructions capable of performing that operation.

This freedom to map to any instruction leads to another concept, software modes. These modes dictate control signals to the translator on how to map certain instructions, as well as global configuration parameters for a test. As a result, a “test” is actually more like a superposition of several variants of

assembly language code, compressed down to less than the length of any single program, and selected by the mode. This means it is possible for a few dozen lines of test code to be mapped to thousands of different programs, each with hundreds of lines of code. Thus only one test for a specific code snippet had to be written, and all variations could be automatically generated, thus greatly facilitating the work required. Thousands of short tests could be written and thousands of variations could be generated for each instead of writing a million longer pieces of code.

An example might be useful in showing the flexibility of this concept. Let's look at an initialization construct for which we wish to place the 64b values 0,1,2,3 into vector register 0. The test level code would look like:

```
[INIT]      VR:0  dword 0 1 2 3
```

However, a single parameter passed to the code generator would allow this test code to be mapped to assembly language via any of the following methods:

1. A vector load loading the first four elements from on-chip memory
2. Looping 4 times on: scalar load, move to cop2, and vector insert to increasing indexes
3. Looping 4 times on: scalar load immediate, move to cop2, and vector insert to increasing indexes
4. Looping 4 times on: clearing all elements up to vector length using a vector-scalar and, a scalar load immediate, a move to cop2, a vector-scalar or, and decreasing vector length

The RTL designer would know that these methods exist. In conjunction with his knowledge of which functional units were working, he could select the appropriate code generation method, and proceed to testing another functional unit potentially bypassing broken units.

Another concept included was that of register variables. For example, instead of specifying *\$vr1*, One could specify *\$vr[X]* and the script would choose a value at random for *[X]*. Although this works well for most instructions, some instructions have restrictions (*vIU*) on which registers can be used. Thus it was necessary to introduce restrictions such as "*[X] ne [Y]*" and so forth. This concept was overly cumbersome on larger programs and the feature was only used in a couple of simple tests.

In order to consolidate all the work necessary to translate test code, generate assembly language, configure and run all simulators, and finally to determine failure locations, an all-encompassing script was written in PERL. This was called the verify script: *verify.execute.pl*, Figure 8 shows its general functionality and Figure 9 details its position in the overall verification flow.

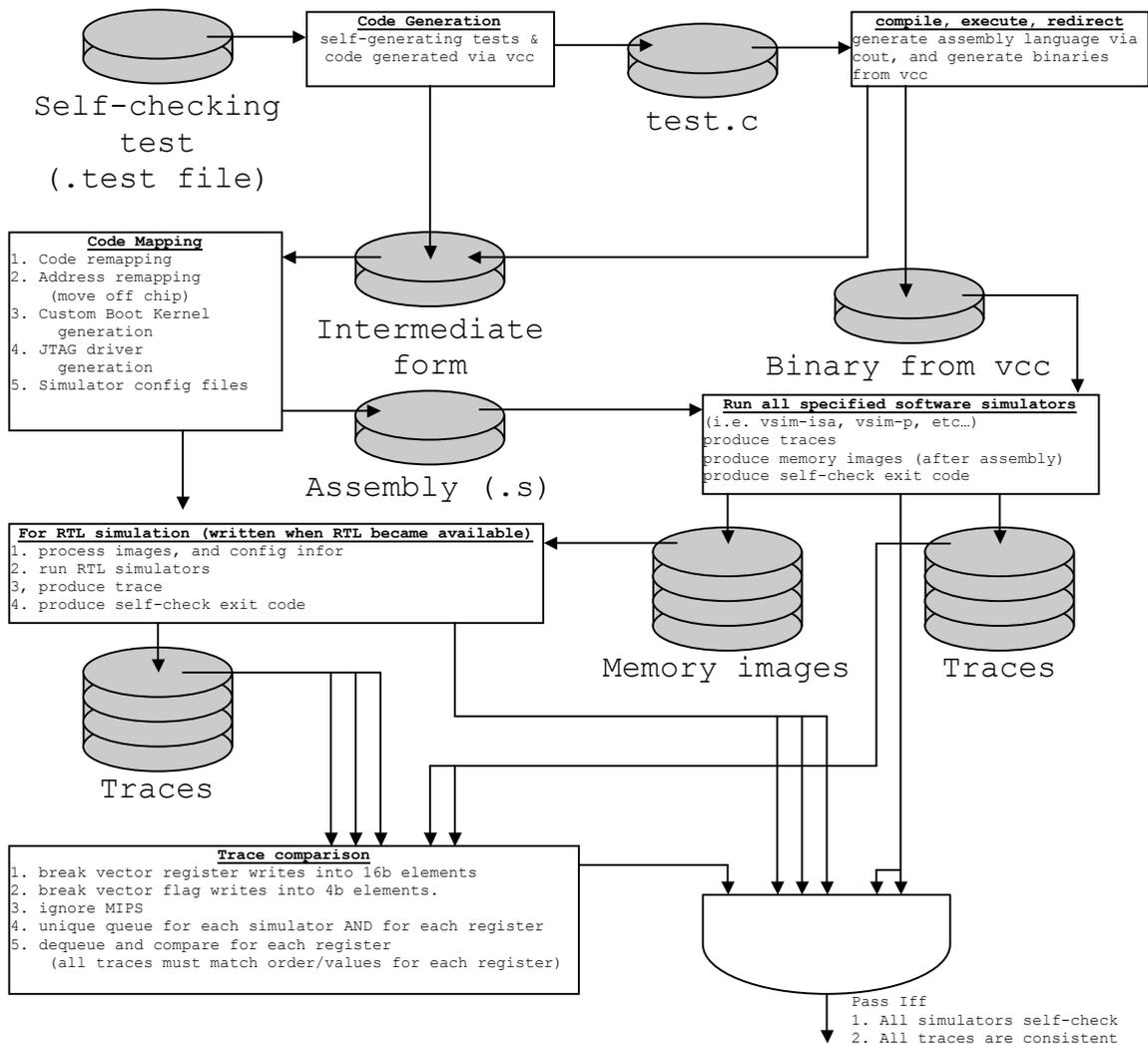


Figure 8 - verify.execute.pl script

This figure details only the flow of data through the translator script. First, a self-checking test (top left) is processed and translated into an intermediate form. Self-generating tests produce C code, which is compiled, executed, and redirected. The intermediate form, common in most tests, is then mapped to assembly language constructs and merged with code from self-generating tests. In addition, C code can be passed to vcc (the vector c compiler) for compilation and inclusion within the final assembly program. The assembly language program is then run on any specified software simulator (producing traces, and memory images, self-checking results). The memory images are then used as stimulus for any specified RTL build. This simulation produces more traces and self-checking results. All traces are compared to ensure all simulators are consistent with each other. Finally, all self-checking results are ANDed with the trace comparison result to determine a final pass/fail condition. (lower right)

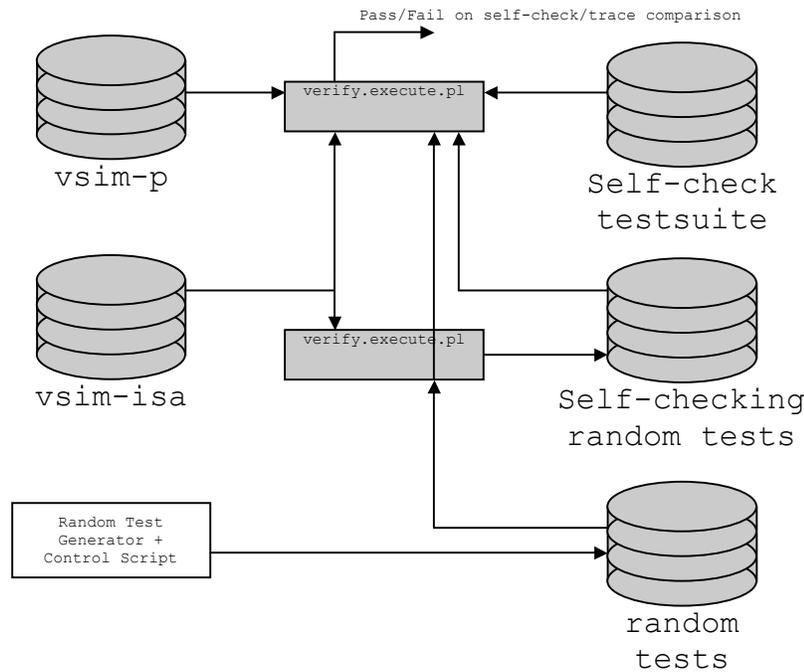


Figure 9 - Verification Flow Using The Verify Script

The verification script from the previous example can be condensed to a single node in the verification flow graph. It takes tests and simulators, and produces a pass/fail condition. This greatly simplifies the work required by a designer to test a change made to a RTL block or simulator.

In addition to the basic functionality, the test file format included support for I/O operations allowing for testing of the network interface and JTAG/VTAP ports. These additional constructs weren't used except for a few specific tests that were used to verify the functionality of modules like the test access port in the vector unit.

5.2 The Initial Testsuite

Once the test language translator in the verify script and language semantics were reasonably stable, a thorough testsuite could be written. Since at this time we were still planning on using the Sandcraft core, which being an IP block should have already been verified, there was no need to write scalar tests (either integer or floating-point). Nevertheless, the testsuite for the vector unit, which comprises more than 80% of the logic of the chip, was a daunting task.

The testsuite was initially partitioned into several smaller testsuites whose statistics are detailed in Table 2. This partitioning allowed both the verification engineer and the RTL writer to check off functionality one step at a time, instead of having to initially deal with programs that required several major functional units to work correctly. It should be noted that since there is a mapping step during translation,

vector initialization statements, which most efficiently can be written with vector load instructions, can be replaced with a slower process of using scalar loads and vector insert instructions, thus ensuring that only the vector arithmetic block is being exercised.

There were some cases where trying to express certain instruction combinations was overly complicated in the semantics of the test file format. Furthermore, there had been enough bugs to necessitate very thorough and complete testsuites for these instruction classes. These testsuites were generated using a very small test generator written in C. These self-checking test generators were easy to write, allowed coverage of most critical cases, saved significant coding time, and all in all were very useful in verification. The test generators could have been folded into the verify script through the C code section. However, the RTL designers typically found it useful to have assembly-like test level programs in lieu of an extremely complex test level language program or even a C generator. Writing every test in C would have complicated and sacrificed some of the mapping capabilities of the verify script.

| TestSuite | Description | Tests | Test Code Lines |
|------------------------|---|--------------|-----------------|
| Vector/Basic | ISA perspective of the simplest tests | 12 | 216 |
| Vector/Arithmetic | Vector integer arithmetic tests | 193 | 18261 |
| Vector/LoadStore | All forms of vector load and store tests (unit stride, stride, indexed) | 557 | 64145 |
| Vector/Processing* | Vector processing tests (insert, extract, half, butterfly) | 495 | 36214 |
| Vector/FlagProcessing* | Logical, pop, 8 at a time | 514 | 32659 |
| Vector/Misc | To/from control/scalar, vsatvl | 12 | 242 |
| Vector/Exceptions | Test the vector exceptions and different conditions they arise under | 128 | 9112 |
| | | 1,911 | 160,849 |

Table 2 - Initial Testsuite

This table details the original self-checking testsuites. It should be noted that Test code is an abstraction from assembly language. As such, each of the thousands of modes for each test typically had ten times as many lines of assembly language. Although Test Code is often unique, generated assembly language rarely is.

*these testsuites included some generated tests totaling about 60,000 lines.

It took about a month to write and debug these vector tests on the ISA simulator. Since the performance simulator now shared a common library with the ISA simulator, bugs fixed in the ISA simulator were fixed in the performance simulator as soon as it was recompiled. However, there were still plenty of other bugs in the performance simulator that were being found.

It also quickly became clear that the ISA simulators had several shortcomings, the first of which was the lack of any kernel mode support. Thus trying to run the Vector/Exception tests caused the simulator to terminate instead of jumping to the appropriate exception vector. This led to the memorable “what do you mean its not supported” response that would become all too familiar when dealing with various simulators and representations of the design.

By the spring of 2000, it became clear that we would not be using the Sandcraft core. As a result, we needed not only a new core, for which we decided on a MIPS M5KC, but also a FPU, part of which we designed, and the remainder of which came from MIT. Of course, we also needed testsuites for both. Given the constant bugs in the ISA simulator coupled with the impending task of RTL simulations, it was decided to simply write a whole new ISA simulator designed to mimic the functionality of the hardware as closely as possible (ignoring timing). Thus it would accurately represent the ISA based on the MIPS core, the FPU, and would include full kernel mode support.

It took some time to bring the new simulator, *vsim*, up to speed, and it required several new testsuites (detailed in Table 3) to verify it, but once it was verified, it has remained virtually untouched, even in the presence of RTL verification. Floating-point tests were assumed to be unnecessary since the MIT execution unit was believed to be correct.

| TestSuite | Tests | Lines of Test Code |
|------------------------------------|------------|--------------------|
| Vector/FloatingPoint | 42 | 2561 |
| FloatingPoint/Arithmetic | 18 | 891 |
| FloatingPoint/Arithmetic.with.NaNs | 10 | 520 |
| FloatingPoint/Exceptions | 43 | 2095 |
| FloatingPoint/LoadStore | 4 | 238 |
| FloatingPoint/uKernel | 10 | 1151 |
| uKernel | 46 | 9953 |
| TLB/Exceptions | 50 | 2989 |
| TLB/Instructions | 6 | 179 |
| TLB/uKernel.generated.stride | 9 | 4722 |
| TLB/uKernel.generated.unitStride | 3 | 534 |
| TLB/uKernel.old | 17 | 6740 |
| | 258 | 32,573 |

Table 3 - Additional Testsuites

This table details additional floating-point and micro kernel self-checking testsuites.

5.3 Changes in the Project and the Problems Which Arose

In the summer of 2000, the performance simulator writer graduated, leaving the performance simulator unmaintained. This meant that neither the current unfixed bugs, nor any new ones found, could be fixed. Week by week the performance simulator fell more and more behind as changes were made to the ISA as well as the microarchitecture. As a result it ended up being utterly useless for verification and was discarded.

The other major shift was the departure from the high-end computing arena in favor of the embedded arena. Some of these changes are only visible in the microarchitecture (such as the removal of 4 of the vector multipliers), but others, such as only supporting single precision floating-point computation, required changes to the ISA, simulators, and RTL. The other change dictated by this shift was the removal of the NI. In the end this was a much more moderate change, as tests were simply not run, as opposed to having to make modifications or changing either simulators or RTL.

Overall, this verification method was highly successful, and adapted well to the slight changes in the ISA and the loss of the performance simulator. The flexibility of the verification script is based on the abstraction of code snippets into the test file format, which means only the code mapper needed to be changed, not the individual tests, to compensate for minor changes to the ISA. Similarly, the abstraction and reduction in coding significantly facilitates verification effort by allowing more widely varied tests to be written in a shorter time. On the downside, self-checking data (not code) had to be either written by hand through knowledge of the ISA, or generated by a program with knowledge of the ISA. This meant that tests could be wrong (proven so by ISA simulation), but also, that the ISA could be wrong. By the end of the summer of 2000, the new ISA simulator had been fully verified, and verification was poised to move onto RTL as soon as it was deemed ready.

6 Basic RTL Verification

More than six months after the originally scheduled tape-out, the major RTL blocks neared completion. Additionally, the MIPS core RTL was delivered. It was necessary, however, to perform some basic verification on these blocks individually to fix all the trivial bugs in parallel. Since the FPU instantiated the MIT execution unit IP block, which was tied directly to the IBM SA27E [IBM00] standard cell library as well as IBM designWare components, its verification could not be started until we received the design kit from IBM.

While waiting for the IBM components used by the FPU, its designer created a small testbench that issues instructions via the coprocessor interface that are then dispatched to a stand-in execution unit, which in turn, produces a set of prescribed results for various instructions. The testbench then examined the results written to registers to ensure that the correct operation was being written to the destination register. Of course this method is extremely limited, but was useful in fixing the most trivial (non-computational) bugs.

The VU designer opted for a similar, yet reduced strategy. Instead of feeding an instruction from the coprocessor interface and checking its progress through the pipeline like what was done for the FPU, he partitioned the vector unit into several blocks and tried to verify them independently. For example, to verify the lane, data and control were driven directly at the lane level, but only for a tiny subset of the ISA. Furthermore, instead of relying on a script to check the results, they were only eyeballed. As the designers were responsible for fixing their own blocks using the verification framework, this only saved him time in the short term.

On the bright side was the MIPS core. Even though it is an IP block, it was a pleasant surprise to receive a thorough test suite and testbench with the distribution. Unfortunately, early releases required the core to be configured with the default cache and TLB sizes in order to run the test suite correctly. Once this

was corrected, it was very useful in debugging modules, macros, and pseudocells, as well as ensuring that before integration, the design was valid.

7 Partitioning – RTL, IP, and Custom Block Verification

Now that major RTL blocks were available, they could be integrated and ISA verification could be performed at the RTL level. The next major modification to the verification flow was the partitioning of RTL verification tasks so that progress could be made in parallel by the designers and the verification engineer. In addition to the use of self-checking tests to determine pass/fail conditions, RTL trace comparison was needed to ensure complete adherence to the ISA as well as to facilitate debug by finding the exact failure point.

Since not all blocks had been completed, it was necessary to try to partition and parallelize the verification work in order to make progress on completed blocks. Moreover, since custom blocks were lagging well behind, they were separated and stand-in behavioral models were used. Each of the custom blocks (vector register file, vector multiplier, vector adder, vector shifter, vector rounder, vector saturation, and crossbar) would be initially verified with a standalone testbench created by their designers. Only when they had been fully verified would they be integrated with the rest of the design. Similarly IP blocks, which should work by their very nature, were also separated and verified immediately. The remaining blocks were synthesizable RTL, including the FPU, and the VU. RTL verification progressed quite differently than custom module verification. Whereas custom blocks were designed to meet a specific spec, which was easy to verify in most cases due to the lack of state, these RTL blocks were verified along ISA lines. These two had received some verification work, but nowhere near enough. Since each of them is a coprocessor, once the MIPS core was stable, these blocks (FPU and VU) could be attached individually and then debugged. Once this was complete they would both be used simultaneously (while still using behavioral versions of the custom blocks) to simulate the full VIRAM1 architecture. Succinctly put, in-house RTL verification was broken into three parts: MIPS+FPU, MIPS+VU, and VIRAM1 (MIPS+VU+FPU). The first two combinations could be debugged in parallel by their respective designers and the verification engineer.

7.1 Custom Block Verification

The first custom block, the register file, was somewhat unique in that not only was it extremely regular, but it also made extensive use of dynamic logic. Because it is regular, only single row/column pairs needed to be simulated (with appropriate dummy load) to ensure electrical timing and functionality. However, the use of dynamic logic prevents any direct extraction to Verilog for simulation with the rest of the design. As a result, it was necessary for the designer to verify the layout, and then provide an exact functionally equivalent RTL representation for simulation. This ensured that the version simulated was

based on the designer's interpretation of the spec, and not the spec itself. If there were any misunderstanding, it would show up. In order to verify the layout, a test generator was written that accurately represented the register file, its contents, the write style, and the ports used. The test generator produced code that was parsed, converted to a Spice deck, and embedded with the extracted Spice netlist. The Spice deck was simulated with either Spice or TimeMill, and outputs were extracted and compared against data produced by the test generator. The designer found the generator very useful in tracking down bugs, which usually amounted to wiring problems. Relatively few bugs were found, a testament to the designer and the regularity of the block. However, Spice simulation could take a day per cycle, resulting in a somewhat lengthy verification phase.

The remaining custom blocks (the crossbar, and the five vector integer blocks) were all destined to use the same flow by their designer. Since they are comprised entirely of CMOS or PTL, a Verilog netlist could be extracted. These blocks were first verified with directed inputs, then random inputs with cosimulation of the behavioral model to ensure correctness. Finally, the Verilog netlists were inserted with the rest of the RTL for full chip verification. Significant delays, required to verify the extremely complex custom integer datapaths (remember they are variable bit width vector integer/fixed point, signed/unsigned, high/low, upper/lower datapaths), coupled with significant area overhead required to fix the ensuing bugs, necessitated the use of synthesizable integer datapaths in VIRAM1. The crossbar, although even larger, was extremely regular like the register file, and was essentially a routing exercise instead of array construction. Once verification work was begun on this block in earnest, it was quickly completed.

7.2 Floating-Point Unit RTL Verification

As discussed previously, the MIPS core was quickly verified so that it could be included with more encompassing simulations. However, the MIT floating-point datapath is not distributed with a testsuite. Although a standalone testbench could have been written, lumping its verification together with the rest of the FPU was a more efficient solution. The advantage here would be that ISA tests already existed, and trace comparison could be used to provide more coverage. The downside was that this is both ISA and IEEE verification, and it is possible that many bugs were present in the datapath that now could only be found by examining the numerous corner cases required for IEEE compatibility. Since the FPU RTL was completed first, its verification was started immediately. The FPU, being a coprocessor, was easily attached to the MIPS core. Instead of using the on-chip memory found in VIRAM1, we continued to use the sparse memory model provided by MIPS and accessed it through the SYSAD interface. Very little work was required to attach, compile, and bring this environment up. Verification was then broken into a testsuite MIPS provided for floating-point instructions and one written not knowing we would be receiving one from MIPS.

The MIPS testsuite found a few IEEE bugs with the FPEU, including some with exception handling, but very few in the reorder buffer logic of the FPU. The lack of bugs in the architectural part of the FPU is primarily due to the way tests were written. Virtually every test was of the form load, load,

operation, compare, and branch. This structuring prevented virtually any hazard from occurring. It was clear that given the small number of bugs found with these simple tests, all of which were critical, not only would more extensive directed tests be needed, but also significant random testing.

In order to begin running the UCB testsuite, several changes needed to be made. The first was a modification to enable the running of code generated from *vsim* on the MIPS core. The MIPS testbench was reused and a few changes were made to the simulator to produce a binary image that the testbench could read. The verify script would then apply the appropriate command-line switches to *vsim*, which in turn would produce this “.hex” file. The verify script would then post process it as necessary and run the RTL simulator providing this as an input. The simulator would run to completion as usual, but now the verify script would examine its output to determine the pass/fail condition used by self-checking tests.

The other change was to add support for trace comparison. Although both the core and the FPU are in-order machines, the trace comparator was designed to support out-of-order commit, which is useful for the vector unit. As previously discussed, there are three phases to trace comparison: trace generation, parsing, and comparison. Generation has to be handled by each designer. In this case, a trace was produced only for the FPU. This trace included the 32 *FPR*'s and the control register *FCSR*. The trace, which was printed to standard out, included an identifier (to show it was for the verify trace), the simulation time, the register in question (e.g. *FPR[12]*), and the value written to the register. Generation on the ISA side is the same trace originally used for *vsim-isa/vsim-p* comparison. That is nothing more than the ISA level trace (including instruction number). Parsing was nothing more than reading from these two sources into two pairs of associative string arrays (time/value for ISA/RTL) indexed by the register. The new value is appended to the list. The final step, comparison, simply ensures that writes to a given register occur in the order specified by the ISA, which is an acceptable practice for an out-of-order commit machine.

After completing these modifications, the directed self-checking tests were run with trace comparison turned on. A few new bugs were found. Some dealt with IEEE compatibility, and others with exception handling in the architectural part of the FPU. These directed tests were significantly less thorough than those written for the vector ISA, primarily because it was believed that initially not only would we receive an implicit FPU embedded within the core, and that the new IP datapath would be correct, but that the MIPS testsuite would be sufficient.

The overall lack of success here motivated me to write a simplified random test generator specifically for the FPU. It was designed to generate all the cases any sane programmer would not. Like the full random test generator described earlier, it was highly programmable, but instead of broad instruction classes, the instruction classes in the floating-point random test generator were based on the datapaths they exercised (*add*, *mul*, *i2f*, *f2i*, and so forth). The tests produced by this generator were extremely effective in finding the impossible to enumerate cases involving exceptions, nullifies, kills, data arriving out of order (from registers or memory) and the reorder buffer. In addition to these cases, for which no reasonable number of directed tests could have ever found, there were several IEEE issues that

were uncovered and not covered by the other testsuites. The majority FPU verification took less than three months, but the trailing edge (caused by reorder buffer issues) took several more months to complete.

In many ways the FPU RTL verification was a dry run for VU or even full chip verification, since it forced all bugs in the tools to be worked out, the testsuite/simulators to be adapted to the MIPS testbench, and examined the more subtle issue of how two RTL designers feel about the tools and the way failure information was presented. On the upside, the testsuite and tools were readily understandable, allowing the designers to debug their blocks individually. For most cases, the trace comparison and the way failure data is presented (time, ISA cycle, register, RTL data, ISA data) was sufficient to fix any computational bug and many of the reorder buffer issues. Where verification information fails to be obvious, or at least becomes a little more cryptic, is in the case where either the same data is written over and over to the same register and one write is missing, or in the case of sticky bits in FCSR. In the latter case, it is not clear that an exception occurred if the bit is already set. Nevertheless, it took relatively little time to completely debug the FPU, despite the fact that the rate at which bugs were found was basically asymptotic.

7.3 Vector Unit RTL Verification

Once the VU RTL was believed to be completed, its verification proceeded along lines similar to that of the ongoing FPU verification, correcting any issues that had previously arisen. The original plan was to run through the directed self-checking tests one testsuite at a time, then move on to simple and highly-restrictive random tests, and finally employ all the functionality of the random test generator. However, the combination of the complexity of the design and the desire of the VU designer to keep verification as simple as possible for him required some additions. The first testsuite, Vector/Basic, which amounts to nothing more than a few *ctc2/cfc2* instructions in the preliminary tests, was far too much for the designer initially. What was not clearly seen was the fact that in addition to testing the vector unit, simple things like clock generation, as well as the vector I/O block (an arbiter between MIPS LSU, VU, on-chip DRAM, off-chip DRAM, and DMA), needed to be verified.

These restrictions required the creation of several new testsuites, which defied the simplicity and flexibility of the previous testsuites. These tests had to be pure assembly language, since there was a requirement imposed by the VU designer, that only necessary instructions be present. Thus, boot kernel generation, essentially a parameterized PERL script that generates a boot kernel, could not be used. A custom boot kernel was required on a per test basis. These custom boot kernels were hand-optimized versions of the code seen in the generated boot kernels. Each of these tests had to be changed whenever a major change to the design was made, like the amount of on-chip DRAM. The initial tests were run completely from off-chip memory and progressed through testing a few rows in DRAM. Table 4 describes these initial testsuites.

| TestSuite | Description | Tests | Lines of Test Code |
|------------------|---|-------|--------------------|
| Other/BabyScalar | Basic boot kernels, and off-chip memory accesses | 8 | 2248 |
| Other/DRAM | Basic boot kernels, Scalar and Vector on-chip memory access | 21 | 4702 |
| Vector/Baby | Trivial vector instructions without memory accesses | 2 | 290 |
| Other/Milestone | Most basic scalar test (nop), and a scalar Y=aX+b | 2 | 66 |
| | | 27 | 7366 |

Table 4 – Simple Testsuites for RTL Verification

This table details the testsuites added to verify the RTL testbench and simplify the initial RTL debug.

The bugs these tests found would have been in every other test. This meant that they were nothing more than an apparent timesavings. However, this was not the opinion of the VU designer/debugger. The advantage, from the designer’s point of view, was that these tests start from the ground up. Make sure a *nop* works, make sure the off-chip boot kernel can run, make sure on-chip memory can be accessed, and so on. Harkening back to one of the tenets of verification, to make it easy for the designer, it was acceptable to create these tests to make it easier for him to debug his code. Figure 10 details the verification flow including RTL simulation. Design components (simulators or RTL) can be compiled and executed by the verification scripts using parsed testsuite information as stimulus. At this point, RTL includes behavioral blocks for datapaths.

We were now able to proceed with the rest of the Vector testsuite. The VU designer chose to run the Vector/Arithmetic before the simpler testsuites. However, the code mapper in the verify script was mapping vector initialization constructs to vector load (*vld*) instructions. Normally this mapping would be perfectly acceptable, except it became apparent that a major bug was present in the crossbar, which is required to accesses on-chip memory. This bug was far more than simply a coding style, it was a total lack of communication on functionality resulting in the crossbar designer implementing what he thought it should be, not what the behavioral RTL specified. Instead of waiting for it to be resolved, a simple toggle in the verify script was utilized, which instead of mapping initialization constructs to vector load, mapped them to vector insert (*vins*) instructions. This vector insert mapping utilized the scalar load store unit, which bypassed the vector load bug, a transfer across the coprocessor interface, and a vector insert. Luckily the vector insert instruction is very simple to implement and worked correctly from the start. Similarly, the verify test construct used a vector store instruction by default. Reversing the operation, this new mapping used a vector extract. This simple transformation allowed verification progress in parallel with fixing the crossbar. The bugs found in the Vector/Arithmetic testsuite ranged from computational errors to more subtle issues where it is documented that there is not an interlock on certain operations, or the flipside where there are undocumented cases where there is no interlock. The later two were fixed by changing the code to adhere to the RTL. The former bugs were fixed in the RTL.

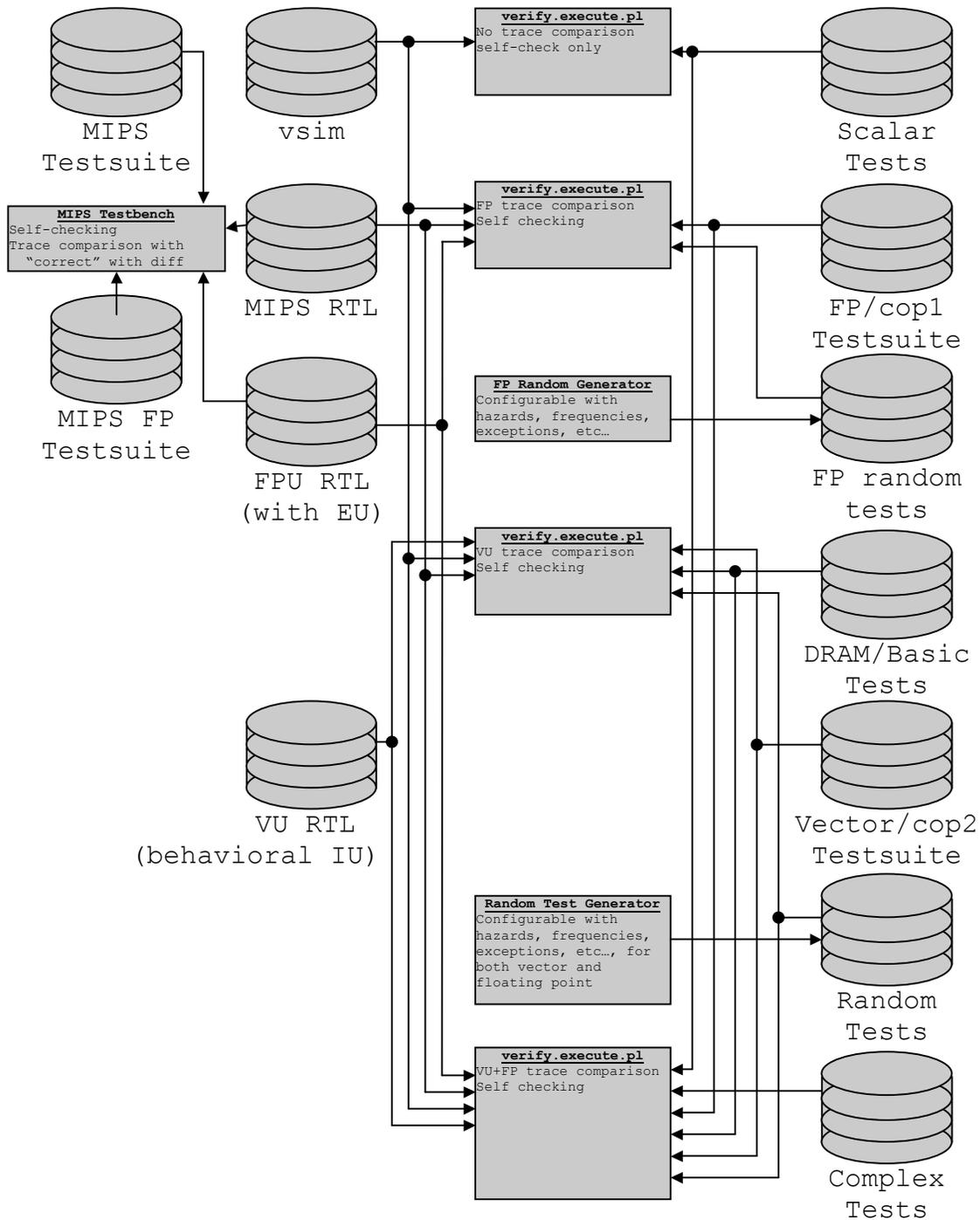


Figure 10 - RTL Verification Flow

RTL Verification can only proceed after the test suite and ISA simulator (vsim) have been verified. Hardware description code is on the left, and stimulus (tests) are on the right. Time, or more precisely complexity increases down the page. Verification starts with the ISA simulator, proceeds to MIPS standalone, then MIPS+FPU. In parallel, MIPS+VU verification can commence. Finally, RTL verification is completed with MIPS+VU+FPU simulation.

Running the first few vector testsuite uncovered some bugs that were extremely nasty. For example, some instructions were either encoded by the assembler or decoded by the RTL incorrectly. Thus a *vadd* in the simulator was executed as a *vsub* in the RTL. As a result, a testsuite was written to ensure that instructions were being encoded correctly by *vsim*. Perhaps more critical were chaining bugs. The RTL defines four kinds of read chaining styles, and four types of write chaining styles, and the three standard hazards. An attempt was made to enumerate these cases into directed tests. Because these conditions are timing critical, these tests were essentially worthless. The search for these potential bugs was temporarily bypassed in favor of making progress on other testsuites. As the search for reorder buffer bugs in the FPU used a random test generator, so did the search for chaining bugs.

The vector floating-point testsuite was passed quickly because the datapath had been previously debugged in conjunction with the FPU RTL, and the VU used an imprecise exception model. The other vector testsuites were also very easy to run due to relatively simple instructions, and in some cases, their previous use in the code-mapping fix.

A set of more complicated tests was also written to handle the rarely used, but critical, functionalities of the chip and project. Unlike all previous tests, many of these exercised functionality only present in the hardware, and not in the software simulator. As a result, where other tests could rely on trace comparison as a safety net or even a primary strategy, these tests had to be completely self-checking, and in some cases, rely on visual inspection. The interleaving and Syscall tests had analogous functionality in the software simulator. Once they had been debugged, the functionality was applied to all previous tests via the verify script. A mode switch was used to determine the interleaving. However, Syscall emulation allowed me to replace the hack exit code solution previously used, where the contents of one specific register became the exit code, with a termination signal code that embodied the pass/fail condition. Thus the same method is used in all simulators, and can even be used in real hardware. A similar mode switch prompts a loader that uses either DMA transfers or a series of word copies to transfer the program from off-chip memory to on-chip DRAM. There is no support for this in the software simulator, and it requires extensive reworking of the simulator's output in order to allow it to run on either the RTL or on hardware. The testsuites required to verify these unsupported functionalities are detailed Table 5.

Unlike the FPU, which included a reorder buffer and implemented a precise exception model, the VU, which implements an imprecise exception model and element group execution, allows for multi-cycle out-of-order commit. This meant that in order to correctly perform trace comparison against the RTL, significant modifications on the trace comparator were required. A (vector) scalar commit, a commit to the vector unit's scalar register file, was straightforward and relatively simple. The trace comparator was modified to show that writes to a given register occur in order, which is a departure from the conventional view that writes to all registers must occur in order. However since the commit occurs in a single cycle, it was a relatively simple modification. A vector register commit, however, is entirely different. Writes to the vector register file occur in element groups of 256 bits. Thus, for most instructions, a vector register

write will require eight cycles. The logical conclusion is not to view the register file as 32 registers of 2048 bits, but more simply as a register file of 1024 registers of 16 bits, or ideally a 2 dimensional array (32 by 32) of 16 bit elements. Now, each element write is atomic, allowing this interpretation to easily be folded into the existing trace comparator.

| TestSuite | Description | Tests | Lines of Code |
|---------------------|---|-----------|---------------|
| Other/ContextSwitch | Runs two "programs" and switches between them on exceptions | 9 | 4502 |
| Other/DMA | Exercises the DMA engines in the VIO block. | 12 | 2761 |
| Other/Interleaving | Changes the interleaving of eDRAM (i.e. reorders address bits and thus mapping of physical address to DRAM macro bank, row, and column) | 1 | 53 |
| Other/Syscall | Tests the RTL SysCall Emulator which was retroactively applied to all tests via the verify script. | 2 | 24 |
| Other/Lib | Due to miscommunication, the compiler will generate unimplemented FPU instructions, which are handled via a library routine. | 4 | 1324 |
| | | 28 | 8664 |

Table 5 – Verification of RTL specific functionality
This table details the testsuites added to verify the RTL specific functionality. Some of these tests could not be run on the ISA simulator at all.

Unfortunately, the VU RTL implemented a slightly different ISA than the ISA simulator. As a result some tests would fail trace comparison even though the test would pass self-check. Furthermore, directed tests designed to trace some of these failures would still pass self-check at the failing instruction. It quickly became clear that one of the following solutions had to be implemented:

1. Modify the VU RTL to work as the ISA specifies,
2. Modify the ISA simulator to work like the vector unit RTL,
3. Modify the trace comparator in the verification script to account for the discrepancy in the case where the results of instructions under certain conditions are conceptually invalid.

The third choice was selected for several reasons. Foremost, the difficulty involved on the part of the VU designer in making the hardware work would require a complicated or cumbersome implementation. Similarly, there was no reason to make the ISA simulator conform to behavior that is invisible from software.

7.4 VIRAM1 RTL Verification

After both individual RTL modules were working reasonably well, they were combined and verification proceeded with larger more complicated programs. One type of these new larger more complete tests was the micro kernel testsuite, which performs a series of memory and CPU intensive operations including transfers, vector arithmetic, matrix arithmetic, on relatively large structures using both

coprocessors. Additionally, the random test generator was updated. Memory coherency events were created and handled via the generation of the sync instructions. Finally, a chaining primitive was created. This would select two instructions, one each from the four read and four write types, a hazard, and a shared register, and generate a series of intervening instructions to generate a chaining event. These were extremely useful in finding timing dependent bugs. However, the cases previously found by VU RTL verification, which had not been resolved, made complete random tests very difficult until these bugs were fixed.

8 Coping with ISA Evolution and Verification in the Presence of Unspecified Functionality

There were several major groupings of failure signatures. The first were false writes, in which data is actually written to the vector register file and thus the RTL trace, even though the write is invisible to software. The second group is a lack of interlocks on flag registers allowing out-of-order commit to flag registers from arithmetic exceptions. Unfortunately, since the writes are sticky, simple reorder checking is insufficient. Third, vector compress and iota instruction boundary cases are a clear departure from the ISA manual. Fourth, nullified writes from stride zero or certain indexed stores will not appear in the RTL trace. Finally, there is some randomness in the hardware, which can never be reflected in software. The solutions were to modify the RTL to note when a write is not a commit, modify tests not to generate certain now “invalid” cases, and finally, modify the trace comparator to pick up on notes from the RTL as well as keeping track of the machine state in order to decide whether or not a miscompare is really that.

8.1 Coping with ISA changes

In addition to these problems, and the microarchitecture changes, ISA level changes were made, the most notable of which was the number of flag register files. Changing the number of vector register elements (either MVL or the number of registers) had been considered, but this was never done. If the test suites had been further abstracted, these ISA changes would have been completely invisible, and in some cases, actually were. However, the desire by designers to make the tests easily readable meant that this abstraction could not have been implemented without sacrificing the goal of making the test suite/bench usable by the designers. As a result, the solution was a mix of code generator changes and rewriting tests. The limited abstraction helped simplify this process.

8.2 Vector Unit Issues Which Caused False Failures

The vector unit was not strongly tied to either the simulators or the publicly available ISA documentation. As a result, in many cases, tests would fail because either the test or the testbench made

the assumption that the ISA documentation specified exact functionality and included all restrictions in the form of various exceptions, typically *vJUI*. Actual implementation, with functionality determined by conceptual understanding of tests in private benchmarking kernels, allowed for, in some cases, unspecified results. The ISA document, originally generated from the ISA simulator, presented the image of a fully deterministic processor. Communication between designers is key to any project, and the lack of it can manifest itself in failures, delays, kludges, and less than acceptable design performance. Having the hardware designers update the ISA document to reflect non-deterministic behavior is essential.

8.2.1 Spurious Writes to the Register File

The vector control unit cannot predict before a load is committed, without extensive additional logic, whether or not a stall will force the write of a 256 bits load to be spread across multiple cycles. In fact, it cannot even determine on the cycle before it is writing whether or not the data being written is correct. This is perfectly acceptable, since on the next cycle it can determine that a stall occurred, allowing garbage to be written, and prevent any other instruction from reading from the vector register file until the final data has been written. However, from a trace generation perspective, it seems like a spurious write, or even an unchecked hazard, has occurred. The solution was to append the trace with another entry stating that the previous write was garbage. The trace comparator in the verification script can implicitly read this entry, understand it, and compensate for any number of consecutive spurious writes.

8.2.2 Issues with Exception Flags

Exception bits are sticky in the sense that the exception conditions from an arithmetic operation are OR'd with the exception register, instead of simply written. There is no interlock to ensure that flags written from the first arithmetic unit (AU0) are ordered with respect to instruction order with those written from the second arithmetic unit (AU1). Thus the exceptions from a *VADD* instruction could be written before those of a *VMADD*, even though the *VMADD* was both fetched and issued first. This is not a violation of the ISA since instructions are no longer atomic in the vector unit. They can be interrupted by an exception, and later resumed. However, since exception bits are sticky, the trace comparator cannot simply try to swap two writes to the flag registers holding the exception bits, but must try to ascertain whether or not a possible reordering resulted from completely aberrant values in the trace.

Perhaps an example will more adeptly illustrate the predicament. In this case the *VADD* completes out-of-order and ahead of instruction order. The exceptions generated are not visible to either software or hardware traces. Only the contents of the flag registers that hold the resulting exception bits are visible. The exception bits represent elements that produced exceptions.

| ISA | | | RTL | | |
|-----------------------------|---------------------------------------|--|-----------------------------|---------------------------------------|--|
| Instruction commit order | Actual Exceptions (not visible) | Value in flag register (i.e. trace) | Instruction commit order | Actual Exceptions (not visible) | Value in flag register (i.e. trace) |
| - | - | 0000 | - | - | 0000 |
| VMADD | 1000 | 1000 | VADD | 0001 | 0001 |
| VADD | 0001 | 1001 | VMADD | 1000 | 1001 |

Thus the trace comparator would infer from examination of the ISA trace that the *VMADD* produced an exception in element 0 (MSB). We know this because the value in the flag register before the execution of the instruction is 0000, and after it is 1000. The MSB is the only one that is obviously set. Thus the exception nibble for the *VMADD* instruction must be “1000”. The exception nibble produced by the *VADD* is far less clear. We know that the LSB (element 3) is set, because that bit in the register file changes. However, we cannot tell if the MSB of the actual exception nibble is set entirely because exception writes are sticky. The previous value of the MSB was 1, and since *I+?* is *I (logical OR)*, we cannot determine the value of Z. As a result, we can say that the exception nibble for the *VADD* instruction is “?001”. i.e. we don’t know the MSB.

When we examine the RTL trace, we can infer that the *VADD* instruction produced an exception nibble of “0001”, and the *VMADD* instruction produced an exception nibble of “100?”. Now we can let the trace comparator compare the exception nibbles produced by each instruction. Any bit that is known in both the RTL and the ISA traces can be compared:

| | | |
|--------------|------------------|---------------------------------------|
| <i>VMADD</i> | “1000” vs “100?” | <i>consistent, but not conclusive</i> |
| <i>VADD</i> | “?001” vs “0001” | <i>consistent, but not conclusive</i> |

At best, we can say the RTL trace is consistent with the ISA trace. However, we cannot conclude that the RTL is correct. In practice, having such luck with numbers is rare. Typically, in random arithmetic programs with out-of-order commit enabled, the trace comparator is dealing with derived exception nibbles like “????”. That is, nothing can be said about it, and thus anything can be consistent with the RTL trace.

The fact that the underlying reordering is completely unknown to the trace comparator, in conjunction with pipelined execution and sticky bits, results in low confidence for this method when applied to tests that generate large numbers of exceptions. Basically, all that can be said is that nothing is obviously wrong, but it cannot be said that it is correct. The verification script, where it failed to make a determination, produced a warning allowing the designer to verify correctness. No real errors involving sticky exception bits were ever detected because of reordering, partly due to the flexible program semantics involved.

8.2.3 Unspecified results for Trailing Elements in an Element Group

One of the worst failures, leading to the most heated debates, dealt with how *vcompress* and *viota* instructions should be handled in hardware. The ISA documentation is explicit on the functionality, but the hardware accepts a more “spirit of the instruction” approach. It came as no surprise that these two distinct approaches lead to differences in the trace that were initially flagged as failures. In the ISA, a compress or

iota will write elements only up to that inferred from the flag, but in hardware, entire element groups are always written. Even worse, the data written for elements beyond the last element inferred from the mask and up to the last element in the element group is unpredictable in the RTL. The merits of either method are not pertinent to this discussion, only the fact that the RTL solution won out. To account for this, the trace comparator had to keep track of instructions, the values in the flag registers, the vector length, and vpw in addition to the normal <time|register|data> tuples. When it is determined that a compress or iota has finished while parsing ISA trace, it is possible to reconstruct the bits of the mask it used and determine what the index of the last element written by the ISA was. With this in hand, it was easy to determine how many elements ($0 \dots 2^{5-vpw}$) had been written by the RTL with indeterminate data. Additional tuples were appended to the ISA trace with data fields replaced with a symbol to denote “Don’t care – its garbage from a compress/iota”. During trace comparison when this was seen, the corresponding write in the RTL trace was ignored. This solution allowed continued reuse of the hundreds of tests in the testsuite that otherwise would have had to be rewritten.

8.2.4 Nullified Writes to Memory

Another interesting case arises for indexed stores where elements within the same element group have the same index and the similar cases where strided stores are used when stride is set to zero. In either case the RTL processes address calculations in element groups and implements a micro TLB (uTLB) to quicken the address translation. If both addresses in question can be translated by the TLB, then the two separate writes to the same physical memory address are coalesced into a single write, which is sent down the memory pipeline. A uTLB miss followed by an eviction of the critical entry could result in the case that both writes to the same address actually take place, albeit on different cycles. From a trace point of view, which has absolutely no knowledge of the uTLB state, whether or not writes are consolidated appears completely random. Since vector registers are processed in order of increasing elements for this particular access operation, the last write to that address in question is considered the correct value.

To handle this potential discrepancy in the trace comparator, all preceding writes to this address in both the ISA and RTL traces, for this and only this store, can be ignored. A simple `$display()` statement was added to the Verilog code to note when a vector store has finished which is when the last element of the last element group commits. This comment was noted by the trace parser and used to mark the end of a vector store to memory. A similar point could be inferred from the ISA trace. When comparing traces, we know there is an error or nullifications if when we reach the end of a vector store in the ISA trace, there are still more trace entries before the end of the vector store in the RTL trace. Simply scanning forward until a match is found, the end of the vector store is found, or the end of the vector trace is reached, will determine whether or not this is a failure or acceptable uTLB behavior. Spurious matches on comparison will soon be determined, since either the trace will be shorter or different. Memory coherency issues arising from the dual load store units (vector and scalar) are implicitly avoided since traces are maintained on a per address/element concept, and memory coherency is handled via software instructions like `vsync`.

8.2.5 Other Issues

The next problem, DMA transfers, arose because they are not supported in the ISA simulator, so when running DMA tests on the hardware, we do not run trace comparison on memory. Trace comparison on registers, as well as the self-checking option, are still available and are used to verify the tests during regression. To ensure that DMA transfers function correctly, self-checking can be performed on memory locations, and timing can be verified by visual inspection.

Finally, some control registers in the vector control unit, such as *vtlbrandom*, are either not modeled at all in software or do not use the same model. As a result, ISA and RTL might never match up. This is perfectly acceptable, and problems can be avoided by restricting the visibility of values in these registers. For trace comparison, like memory accesses for DMA tests, these registers were ignored without any loss in confidence in the verification.

All of these problems were resolved, or acceptable solutions were presented, allowing completion of the verification of the RTL level of this design. The designer could rapidly verify any future changes for performance based on timing from synthesis transparently on both SPARC and x86 processing farms with this extensive testsuite and flexible and easily understandable testbench.

9 Back-end Flow Verification

After RTL was verified, our design followed a “correct-by-construction” approach. At each stage from RTL to GDS, the design should always correct. Of course, it would be foolish to believe that the tools will work as advertised, or more importantly, that the human interaction to configure and run the tools can ensure correct-by-construction. To verify the flow from RTL to GDS, a few additions were made to the existing verification method, some of which are textbook.

First, it became clear that custom integer blocks were not a good choice for this design, so five synthesizable modules were written for the following vector datapaths: add, shift, multiply, round, and saturate. In addition to the blocks themselves, a testbench was written that instantiates the synthesizable code (either in RTL or gates), the behavioral model, and a random test generator. Furthermore, Formality, an equivalence checking tool, was used to prove that the original behavioral models (proven correct in RTL ISA simulation) were equivalent to the resultant gate level netlist. These blocks were then merged with the rest of the standard cell flow, which included IP blocks, control blocks, and the FPU.

All synthesizable blocks were synthesized using Design Compiler, which should produce a correct mapping to gates. However, in order to prove this equivalence, two additional methods were used. The first and most general method was to take the gate level block, instantiate it in place of the existing RTL module, and rerun the testsuite. This worked for all blocks since the watcher modules were external to synthesized blocks. The second pass, where appropriate, was to run the block through Formality, which

would then determine if the synthesized block was equivalent or not to the original RTL. However, all invalid states must produce x's in the behavioral version in order to be ignored by Formality. This is clearly not possible for most of the blocks. In fact, only the datapaths and the MIPS core were run. At this point it became clear that even Synopsys/Formality could be fooled. Poor coding style in the RTL, in the form of unsynthesizable code, can pass through Synopsys/Formality without any errors, but will quickly fail on gate level simulations since the mapping generated has no grounds in hardware or four-state logic.

The next step in the backend flow was place and route (PnR). Apollo, our PnR tool, is capable of in-place optimizations including logic restructuring. Since the netlist was changed in this part of the flow, it was necessary to verify the correctness of the final netlists. Once again multiple checks were made to ensure this. The first was a simple power/ground check that verifies that all power/ground connections in layout are the same as those specified in the netlist. This could catch some issues where there were opens or shorts generated in the power grid.

The second was Layout vs. Schematic (LVS). This was run from inside the PnR tool. Occasionally it would find something wrong – shorts, opens, power grid issues, and of course any hand edits used to fix charge-collecting diodes issues, ECO's, or DRC errors.

Third, all six PnR blocks were streamed out as a Verilog netlist and run through our testbench. In order to save time, these blocks were inserted one at a time, and various combinations were distributed across our processing cluster. The only problems that arose were the need to write case converting wrappers and a few timing problems arising from event ordering issues in Verilog. The original netlist was case sensitive, but the library that Apollo used was case insensitive. This required Apollo to be case insensitive.

Finally, a series of DRC checks was performed. The first was a simplified rule set run from within Apollo. The other five checks were Hercules runs using ever more detailed run sets. Using simpler run sets first quickly found simple common bugs, allowing a shorter cycle in the DRC flow. The first three, metal only, everything but DRAM, and full chip, were run using an older version of IBM's ASIC run set, as we did not have a license for Hercules to run the newer one on. They required 6 hours, 12 hours, and 48 hours respectively for each run. The next two DRC runs were full ASIC and full foundry post processing. They were run by IBM, but required a week turnaround for various reasons. As expected, DRC checks found many thousands of bugs since internally Apollo only uses a small subset of the design rules, but only one was a critical functionality issue. All were fixed.

After including the back-end verification flow, Figure 11 illustrates the entire VIRAM1 design flow. Whenever a testbench or test from the testsuite fails, the designer must backtrack to find the point in the design flow in which the bug was introduced. Of course, it is possible that the testbench or the test itself introduced the bug, although this tended to happen when the documentation describing the block in question was inaccurate, incomplete, or out of date. Full-chip RTL simulations were by far the most time consuming, as suggested by the convergence of arcs for that step in the flow, but also the most beneficial, as far and above the vast majority of the bugs were found there.

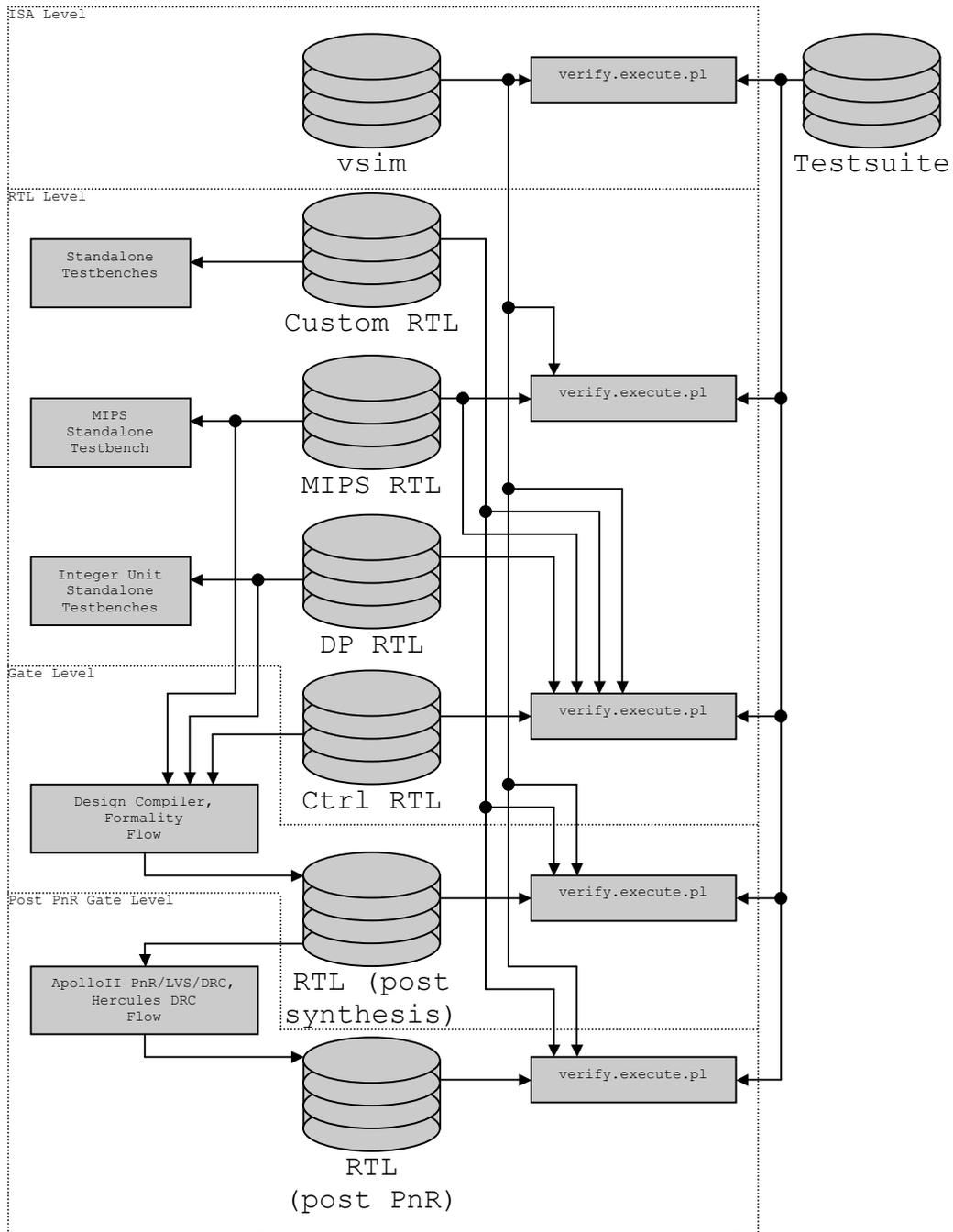


Figure 11 - Final verification flow

The MIPS, custom, and datapath RTL blocks run their individual testsuites first. Second, each RTL block, or set of blocks, is compiled, and executed by the verify script with stimulus from the testsuite. In addition to self-checking results, register values are compared against those from the software simulator for every instruction. Next, all RTL blocks are synthesized. The resulting netlist is simulated using the testsuite. Finally, ApolloII is used to place and route all RTL blocks. After LVS, and DRC flows, a netlist is extracted. The final netlist is simulated using the testsuite just as all other blocks had been. The flow encompasses ISA, RTL, gate and post PnR levels of abstraction.

10 Related Work

Cosimulation is a popular technique for verification of microprocessors. In essence two simulators (test and good) are run in lockstep, and the results are compared at commit to test for any problems with the test simulator. The use of the VIRAM1 ISA simulator to check the other simulators is a similar approach. However, fast fail was not possible since it was not run in lock step with the performance, RTL, or gate simulators. As a result the entire ISA simulation had to be run. Then the entire RTL simulation had to be run. Finally, the traces from each had to be compared to determine if there was a failure. Failure determination would have been much quicker if traces could be compared dynamically. The other major obstacle was that the cosimulation approach assumes that both simulators implement exactly the same ISA, clearly not the case for the VIRAM1 simulators.

Cosimulation can be extended to hardware by embedding a simple hardware checker to verify the execution of a much more complex processor. The goal here is not to verify the chip after tapeout, but instead to catch any bugs not found before tapeout – in affect, trade performance for accuracy. In addition it is possible to catch electrical/timing issues such as setup, hold time, v boxes, alpha particles, and so forth.

DIVA [Aus00] embeds this checker module in the commit stage within the final hardware with negligible slowdown and area penalties. Instructions passed to the commit stage contain the inputs in addition to the result value. The checker reexecutes the instruction to verify the result. In addition to avoid deadlock cases, a watchdog timer is implemented to wait for the maximum instruction latency. If no instruction has completed its execution in the allotted time, the core is restarted at the last instruction committed. Of course the maximum effort must be applied to verifying the DIVA checker. Although, since it is small and relatively simple, a formal method could be applied.

[MAW01] is an extension of the previous paper [Aus00]. The core (fetch/decode/issue/execute/reorder) produces a program stream of executed instructions. These instructions include a predicted next program counter, instruction word, operand values, result, and so forth. The checker now executes all four basic stages (fetch/decode/execute/memory) in parallel since it can use the predicted values as inputs to each stage. For example, the predicted instruction word is used as an input to the checker's decode stage. If any prediction is shown to be incorrect, then the core is flushed and restarted. Otherwise, the instruction commits. Once again a formal method is applied to prove the correctness of the checker "pipeline".

Although DIVA might be appropriate in VIRAM1 for the MIPS core or the FPU, it is certainly not appropriate for the vector unit. This is because the checker replicates the datapath. Since the vector datapaths constitute roughly 60% of the vector unit, doubling them would be impractical for VIRAM1 or any data parallel architecture. Furthermore, the lack of a precise exception would hamper restart implementation. A simplified checker might be a possibility.

Consider the 32 entry, 2048b, VIRAM1 register file. There are 2^{65536} initial states, each with 2^{26} possible transitions. This is an unimaginably complex state machine, and it ignores the MIPS core, cop0,

FPU, and much of the vector unit. To check every possible transition through simulation is impossible. For simpler designs, such as a 32b core, formal methods could be applied to verify the design. One such method [PJB99] attempts to verify the RTL, gate, or switch-level version of an ARM processor against its ISA. This process requires several steps: 1. Describe the ISA using an ADL. 2. Define the mapping from the high-level language to the low-level implementation. 3. Use STE (Symbolic Trajectory Evaluation) to verify the assertions. The major problems that prevent use of this methodology in VIRAM1 verification were the complexity of the architecture, and the fact that the only person who had the knowledge to generate the mapping machines was the RTL designer.

In the end, verification of VIRAM1 required tried and true methods. Massive simulation, with the timesavings of a test file format abstraction, was used for ISA and RTL verification. Gate level verification was performed with industry standard correct-by-construction methodologies and formal verification where possible. Any other method would have been an unacceptable gamble.

11 Conclusions

The incredibly nebulous task of verifying the IRAM project (ISA design, software, and hardware in the form of VIRAM1) was completed in less than three years. The constant evolution of the project, changing timetable, and loss of manpower through graduation, meant that unlike a conventional project, this was not a single large task that could logically be partitioned into a hierarchy of sub tasks, each which could be completed independently, but a series of hundreds small interdependent tasks. Figure 12 details the project tasks over time, including design steps, verification steps and correct-by-construction tools. Many believe that verification technically should not be necessary at the end, given the use of correct-by-construction tools. However, in reality this is completely untrue. Correct-by-construction tools are only as good as the library or information or completeness of the design rules passed to them.

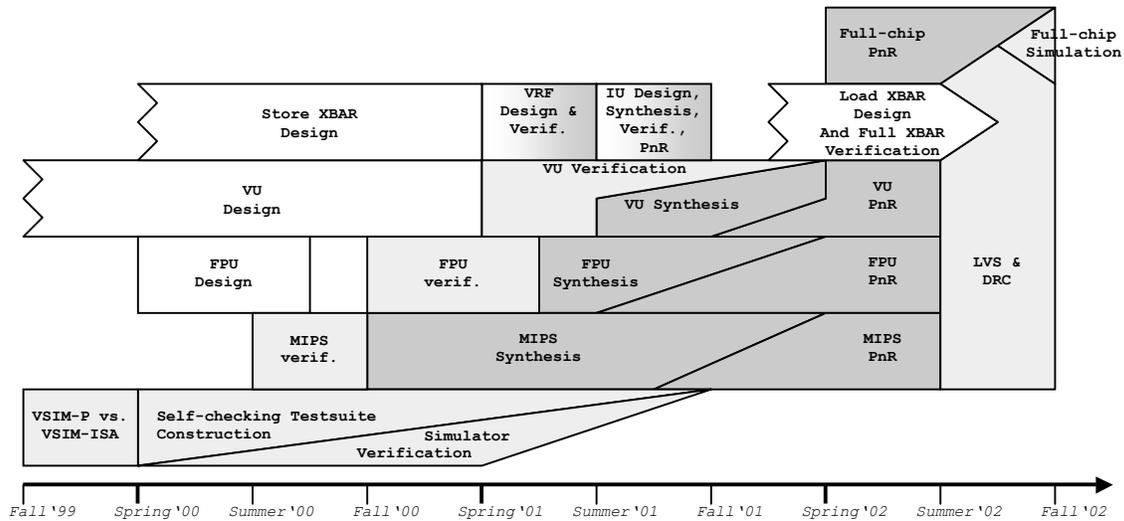


Figure 12 - Final Timeline

VIRAM1 flow from design through implementation. The three fill shades represent design, verification, and implementation. Verification must be performed after design and after implementation. Decreasing manpower resulted in a significant increase in individual workload.

The five major goals of the project were each handled in different ways. The broad concept of correctness was accomplished via a tremendous number of cycles simulated using handwritten self-checking, generated self-checking, compiled, and random tests. Correctness of the ISA was attained by writing tests based on the description and intended use of the instruction. Synthesizable RTL was verified in the same way using the same tests. Datapaths were verified through standalone testbenches and the use of Formality to prove that the original behavioral RTL previously verified matched the synthesized gate level netlist. Custom modules were verified either through cosimulation of behavioral representation with the extracted netlists or in the time prohibitive designs, via high-level modeling and Spice deck generation for simulation. Correct-by-construction methodology - synthesis, restructuring in Apollo, and the like - was used extensively to ensure that correctness of the RTL translated into correctness of layout. Additionally, LVS and DRC checks were performed to check for cases that might be missed by these tools. For a final measure of confidence, gate level netlists were simulated using the ISA testbench just as RTL had been.

| TestSuite | Description | Test Code | |
|------------------------------------|---|-----------|-------|
| | | Tests | Lines |
| FloatingPoint/Arithmetic | Basic floating point arithmetic | 18 | 891 |
| FloatingPoint/Arithmetic.with.NaNs | Basic floating point arithmetic using and producing NaNs | 10 | 520 |
| FloatingPoint/Bugs | Bugs with the MIT execution unit reported by other companies. | 2 | 20 |
| FloatingPoint/Exceptions | Test all floating point exceptions | 43 | 2095 |
| FloatingPoint/LoadStore | Test all floating point loads and stores (e.g. lwcl) | 4 | 238 |
| FloatingPoint/uKernel | Small floating point kernels. (e.g. dot product) | 10 | 1151 |
| Other/BabyScalar | Basic boot kernels, and off-chip memory accesses | 8 | 2248 |
| Other/Cache | Cache initialization/invalidation tests | 2 | 529 |
| Other/Compiled | Tests generated by compiling C code | 2 | N/A |
| Other/ContextSwitch | Runs two "programs" and switches between them on exceptions | 9 | 4502 |
| Other/DMA | Exercises the DMA engines in the VIO block. | 12 | 2761 |
| Other/DRAM | Basic boot kernels, Scalar and Vector on-chip memory access | 21 | 4702 |
| Other/Interleaving | Changes the interleaving of eDRAM (i.e. reorders address bits and thus mapping of physical address to DRAM macro bank, row, and column) | 1 | 53 |
| Other/JTAG | Vector JTAG test | 1 | 5 |
| Other/Lib | Due to miscommunication, the compiler will generate unimplemented FPU instructions, which are handled via a library routine. | 4 | 1324 |
| Other/Milestone | Most basic scalar test (nop), and a scalar $Y=aX+b$ | 2 | 66 |
| Other/Syscall | Tests the RTL SysCall Emulator which was retroactively applied to all tests via the verify script. | 2 | 24 |
| Other/TestVSim | Tests designed to test vsim non-ISA functionality | 5 | 35 |
| TLB/Exceptions | Test for every TLB exception | 50 | 2989 |
| TLB/Instructions | Tests for controlling the TLBs | 6 | 179 |
| TLB/uKernel.generated.stride | Small kernels for testing the TLB using faults arising from strided accesses | 9 | 4722 |
| TLB/uKernel.generated.unitStride | Small kernels for testing the TLB using faults arising from unit stride accesses | 3 | 534 |
| TLB/uKernel.old | Older set of TLB kernels | 17 | 6740 |

Table is continued on next page

| | | | |
|-----------------------|---|-------------|---------------|
| Vector/Arithmetic | Vector integer arithmetic tests | 193 | 18261 |
| Vector/Baby | Trivial vector instructions without memory accesses | 2 | 290 |
| Vector/Basic | ISA perspective of the simplest tests | 12 | 216 |
| Vector/Chaining | Basic vector chaining tests | 48 | 848 |
| Vector/Encoding | Look for encoding bugs | 6 | 834 |
| Vector/Exceptions | Test the vector exceptions and different conditions they arise under | 128 | 9112 |
| Vector/FlagProcessing | Logical, pop, 8 at a time | 514 | 32659 |
| Vector/FloatingPoint | | 42 | 2561 |
| Vector/LoadStore | All forms of vector load and store tests (unit stride, stride, indexed) | 557 | 64145 |
| Vector/Misc | To/from control/scalar, vsatvl | 12 | 242 |
| Vector/Processing | Vector processing tests (insert, extract, half, butterfly) | 495 | 36214 |
| uKernel | Large programs, e.g. matrix matrix multiply | 46 | 9953 |
| | | 2296 | 211663 |

Table 6 - Complete Testsuite

This table details all self-checking testsuites. Note that most tests could be run in thousands of different modes, and each test x mode generation typically had ten times as many lines of assembly language as test code. It should also be noted that in addition to these tests, the random test generators produced tens of thousands of tests, the vast majority of which found no bugs. In addition, there are about 500 tests that became obsolete as the design evolved.

Minimizing the testsuite design effort and adaptability to changes went hand in hand. Abstraction of the testsuite into a language of primitives, which had direct ties to instructions, allowed for rapid construction of a testsuite of thousands of highly configurable tests capable of producing hundreds of millions of lines of assembly language. The complete testsuite is detailed in Table 6. Additionally, this abstraction allowed for a parameter to completely change the code generated and in turn run. Thus, bugs and known issues could easily be avoided and verification could continue without blocking. Similarly, this mapping allowed for variations in the code generated to be randomly selected allowing rapid exploration of the instruction sequence space. This abstraction, in conjunction with straightforward traces and debug information, had the benefit of making tests extremely easy to be read by the designers. This meant that they could simulate, ascertain what and when the failure occurred, determine what the problem was, and fix it quickly. It should be noted that with over 500,000 lines of processor simulator code (both hardware and software), coupled with the typical 6 lines per bug error rate, that nearly 100,000 bugs should be present. Although we quickly stopped recording every bug due to the time wasted, without a doubt there were thousands. Software simulators, for which we did initially track bugs, had hundreds. This vastly improved error rate was accomplished through an extremely thorough testsuite, clearly noting what the failure was, and extremely capable designers. It should be noted that had the RTL designers, after deciding on implementation, updated the ISA documentation to reflect possible non-deterministic behavior, those cases could have been avoided altogether and significant testsuite construction time could have been saved.

In the end, minimization of CPU requirements was easily accomplished through parallelism. ISA simulations could be run on the Millennium cluster (over 100 nodes), allowing a single mode of the test suite, normally whose tests individually take less than ten seconds to run, to be run in less than ten minutes. The time required to debug ISA level tests obviously far outweighed the CPU time required for simulation. Even the CPU time required to simulate RTL was still far outweighed by the human-centered debug time due to the drastically increased complexity of the RTL. Problems with the Millennium cluster, and having only ten licenses, forced all simulation to migrate to the IRAM and Oceanstore clusters. Even after this drastic reduction in parallelism, debug time still dominated the verification work.

As this design shifted to a logic ratio of 20% soft IP, 65% synthesizable, and 15% custom, future designs will likely shift to more soft IP, and less custom logic. Additionally, the majority of synthesizable RTL will likely be further abstracted into a high-level architectural modeling language that could be synthesized into RTL, which in turn, could be synthesized to gates. Thus verification could be performed on the high level design, and correct-by-construction could be used to ensure that the resulting RTL and netlist implement the design. The entire design could be modeled in this high-level of abstraction, given that high-level models of the IP and custom blocks were available, allowing for rapid simulation to verify that the design implements the ISA. Alternately, a formal model of computation could have been used to represent the processor, thus allowing a variety of formal modeling techniques to be used to expedite design and verification. Of course, simulation will be ever-present at every level of the design to insure that the design does actually work for at least some code snippets. Furthermore, cosimulation of RTL with high level language will allow for construction and testing of a single block at a time, while not suffering the time penalties of waiting for complete RTL completion nor the inherently slower RTL simulation time. Minimization of custom RTL and large use of IP will also simplify the verification work since less will need to be verified via Spice or extracted netlists, and more could be verified by a single provider instead of every customer. Finally, abstraction from the ISA into macro instructions and generation of tests, while maintaining a clear picture of the ISA level instructions that tests would be mapped to, will allow for flexibility in the design space exploration while minimizing the verification time.

Acknowledgements

First I would like to thank my advisor, Professor David Patterson, for the opportunity to work on the VIRAM1 project, for his guidance, encouragements, and especially for his patience as I wrote this report. Second, I would like to thank Christoforos Kozyrakis, and Joe Gebis for their friendship, support and commitment to the VIRAM project. In addition Mike Howard and John Kuroda were extremely helpful and inventive in maintaining the machines and software we used. Next I would like to express my gratitude to DARPA, MIPS, IBM, Avant!, MIT, ISI, Cray, Millennium Project, NSF, and Synopsys for their support of the VIRAM project. Finally I would like to thank my brother Joe and my parents, Richard and Kay, for their support and encouragement.

References

- [Aus00] T. Austin, DIVA: A Dynamic Approach to Microprocessor Verification, in Journal of Instruction-Level Parallelism, Vol.2, 2000.
- [Fro00] R. Fromm. Vector IRAM Performance Modeling: vsim-p, the V-IRAM Performance Simulator. Version 3.7.5. Berkeley, CA. March 4, 2000. Available from <http://iram.cs.berkeley.edu/guest/perf.ps> as of May 2, 2001.
- [HP96] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach, second edition. Morgan Kaufmann, 1996.
- [IBM00] IBM. ASIC SA-27E Technical Library. http://www-3.ibm.com/chips/techlib/tech-lib.nsf/products/ASIC_SA-27E, 2000.
- [Koz99] C. Kozyrakis. A Media-Enhanced Vector Architecture for Embedded Memory Systems. Master's thesis, Technical Report UCB//CSD-99-1059, Computer Science Division, University of California at Berkeley, July 1999.
- [Mar00] D. Martin. Vector Extensions to the MIPS-IV Instruction Set Architecture (The V-IRAM Architecture Manual) Version 3.7.5. Berkeley, CA. March 4, 2000. Available from <http://iram.cs.berkeley.edu/isa.ps> as of May 2, 2001.
- [MAW01] M. Mneimneh, F. Aloul, C. Weaver, S. Chatterjee, K. Sakallah, T. Austin. Scalable Hybrid Verification of Complex Microprocessors. Proc. 38th Design Automation Conference (DAC), University of Michigan, June, 2001.
- [PJB99] V.A. Patankar, A. Jain, R.E. Bryant, Formal verification of an ARM processor, Proceedings Twelfth International Conference on VLSI Design. Jan 1999.