A Verilog HDL Virtual Machine

Steve Meyer
Pragmatic C Software
520 Marquette Ave., Suite 900
Minneapolis, MN 94104
sjmeyer@pragmatic-c.com

Andrew Vanvick
Pragmatic C Software
520 Marquette Ave., Suite 900
Minneapolis, MN 94104
avanvick@pragmatic-c.com

ABSTRACT

This paper defines a direct threaded code virtual machine (VM) for executing Verilog HDL simulation. Advantages of interpreted Verilog simulation over compiled simulation are discussed. Reasons for developing a VM to replace a commercial simulator's statement representation level interpreter are given. Performance improvement results are presented, and direct threaded code VM interpretation problems are discussed.

1. INTRODUCTION

In this paper we describe a direct threaded virtual machine (VM) for rapid interpretation of Verilog digital hardware design language (HDL). Verilog is actually three (or more) languages in one. There is a behavioral programming language used for modeling digital hardware at the behavior level and for writing hardware test benches. There is a structural declarative language that describes hardware net lists. Net lists describe hardware interconnections in terms of primitives such as gates, wires, and mask fabrication level cells called macro cells (see Meyer[1988] for definition of net list data structure used in this research). Third, there is a register transfer level (RTL) digital hardware modeling language used to describe hardware at a level that is easy for humans to understand but can be translated to the low level hardware net list level Verilog language component. See the IEEE P1364 Verilog standard language reference manual (LRM) for Verilog language definition (IEEE[19965]). Also see Jennings[2000] or Allen[2002], pp. 612-622 for programming language oriented discussion of verilog semantics. Moorby[1989] is a commonly used text book for teaching Verilog.

1.1 History of Interpreters

Where program execution speed is primary consideration, compilation to native CPU machine code is the implementation method of choice. However, when the entire problem solving process using a computer language is considered, interpreters often have many advantages. See Kleit[1981] and Ertl[2002] for a detailed discussion of these advantages. Popularity of Java whose semantic definition is tied to execution by interpretation of the Java VM (see Lindholm[1999]) has increased popularity of interpreters. Although compiled execution is faster, interpreter efficiency has been improving (Ertl[2001]).

ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators June 12, 2003 San Diego, California Co-located with PLDI'03 at FCRC'03

1.2 Interpretation of Complex Languages

Complex computer languages, especially those with semantics that require program modifications during execution such as Verilog, are a particular advantageous application for execution by virtual machine (VM) interpretation. For such languages, not only are interpreters easier to use and easier to develop, but may even run faster in the future. Complex language interpreters compile (usually incrementally) source language to a VM which is then interpreted. This allows application of compiler optimization algorithms (see for example Morgan[1998] or Wulf[1975]). Interpreters are at least as popular as compilers for these computer languages: APL (see for example Weigang[1985]), ML (Leroy[1990]), Mathlab (Almasi[2002]), Prolog (Krall[1993]), and Spice (Nagel[1975]).

1.3 Interpreter Problem Solving Advantages

Because of the growing complexity of modern computer languages, ease of implementation and ease of user problem solving are important. It has been know at least since the 1970s that interpreted systems with high level operations such as iterators (see for example Earley[1975]) can improve human intuition and therefore human problem solving. This improved understanding can then lead to better languages and better ways of formulating problems.

1.4 Verilog Semantics Defined by Simulator Behavior

Since the de facto Verilog standard is still defined by behavior of the first Verilog simulator called Verilog XL (Cadence[1997]), ease of simulator program development is important. In fact, it is almost impossible to develop a new simulator that must match Verilog XL behavior without first developing an interpreted implementation. Because Verilog XL simulator and language were developed simultaneously, originally Verilog XL interpreted a Verilog statement level intermediate form for procedural and behavioral constructs and used a conventional gate level simulator for structural net list part of language. Intermediate interpreted form were so close to source that Verilog constructs could be reconstructed and printed from the interpreted data structure.

1.5 HDLs Differ from Programming Languages

Because there is no reference CPU that can be used to define semantics as would exist for a programming language, Verilog semantics is defined by conventions derived from electronic design and manufacturing process flows. First the Verilog HDL allowed increased digital system complexity, next other computer programs (called physical ECAD design tools) were developed that read Verilog as input and produced mask generation tapes used to fabricate integrated circuits (ICs). These tools were required to function in such a way that circuits which simulated correctly also behaved correctly when the system they are used in were manufactured. Over time both physical layout tools and Verilog have evolved so that correct simulation and correct fabrication are correlated.

This semantic co-evolution has resulted in a situation in which nearly all Verilog primitive operations are complex. For example, Verilog gate and behavioral delays use inertial delay algorithm. Inertial delay imposes the limit that only one scheduled but unmatured event can exist. Inertial delay algorithm discards the earliest occurring of either the newly scheduled event or the one pending event. The inertial delay event retention decision requires a complex scheduling operation that maps to over 100 CPU instructions.

The inertial delay algorithm does not exactly model digital hardware behavioral, but, by convention, digital hardware design flows assume inertial delay gate switching behavior. However, Verilog now also supports other scheduling algorithms. The other algorithms are rarely used but needed because some hardware can not be modeled using inertial delays. Whenever Verilog changes, and the Verilog language itself evolves rapidly over time, all Ecad tools used in Verilog design flows are modified to "redefine" hardware semantics to match Verilog semantics.

1.6 Only P1364 Verilog Simulation Considered

Because Verilog is the de facto standard HDL, there are a number of uses of Verilog that are not considered in this paper. Verilog is sometimes used for functional verification. Simulation time is removed to speed up simulation using techniques called unit delay or cycle based simulation. Another Verilog use replaces executed Verilog simulation by static analysis that "proves" circuit correctness. These techniques are not covered here because they violate the use of conventionalism for modeling hardware models. Libraries and axioms simultaneously model the new hardware device, its design flow, and its manufacturing flow simultaneously. Because of the continually testing of the correlation between full P1364 accurate delay simulation and manufacturing any mistakes or unanticipated changes will be caught during simulation. If static analysis or functional simulation is used instead, problems will not be detected until after fabrication and the current on going failure analysis feed forward and feed backward system will break down. Optimizations to Verilog models for full P1364 simulation such as discussed in Allen[2002] pp. 622-650 are also not considered because they violate P1364. They are called "abstraction level raising optimizations" (*ibid*, p. 624) but such optimizations also cause failure analysis semantics system to fail.

1.7 Performance Comparisons Anecdotal

Except for measurements that compare our statement level interpreter versus our Verilog VM interpreter, measurements presented in this paper are anecdotal. Anecdotal discussions are all that is possible because the most popular commercial Verilog simulators are licensed with prohibition against bench marking, and for that matter prohibit disclosure of any information. Results presented below are from public discussions concerning Verilog simulators in various internet news groups and from general conversations with customers and are believed to be accurate.

2. LIMITATIONS OF COMPILED VERILOG SIMULATORS

Verilog XL is an interpreter that reads, elaborates and simulates Verilog in one step. The other most popular commercial Verilog simulators are compilers that use the compile, link and simulate paradigm. Compilation to machine code is currently considered best simulator implementation method. However, compiled Verilog simulators have a number of disadvantages. Or viewed in another way, XL style interpreters have a number of advantages.

2.1 Interpreters Load Faster

Interpreted simulators load faster. For common edit, load, simulate process, interpreters load and begin simulation in just a few seconds. In contrast, compiled simulators can take 30 or more minutes of elaboration processing time before start of simulation. Compiled simulators often support separate compilation. This reduces elaboration time difference but slow elaboration is still viewed as an annoyance.

2.2 Compiled Simulator Debugging Inferior

Most common type of bugs in hardware are edge problems such as wrong logic value (direction) or at wrong time edges. Best method for catching such problems is to set invasive event control triggers to trigger entry into debugger when problem edge occurs. The break points are invasive because an event control element must be added to the problem net's change element list that was not known during original design elaboration. Interpreters allow debugging using full power of Verilog HDL. Such invasive changes are impossible for normal compiled simulators. Ability to add Verilog debugging statements during simulation is possible for incremental compilers, but incremental compilation to machine instructions without addition of interpreter type overhead is difficult if not impossible.

Another type of useful hardware debugging involves adding quasi-continuous force and release statements to force (tie) a given signals to a value Quasi continuous force statements in Verilog require either adding and removing special operators to internal net list that are checked during simulation or changing variable assignment code when force and release statements are executed. In contrast compiled simulators debug by saving history of net changes (called $dumpvars) that are viewed after the fact by a wave form viewer. To catch a wrong edge, source must be edited and recompiled. Quasi continuous force and release debugging is impossible.

2.3 Compiled Simulators Unportable and Difficult to Develop

Generating code for a new architecture is known to be difficult and it is even more difficult for Verilog because Verilog

objects do not map directly onto CPU registers and Verilog state does not map onto stack based paradigm of modern processors.

Shortcuts for developing compiled simulators such as compilation to C followed by C compilation using an optimizing C compiler do not work because Verilog compiled code output (either native code or C) consists of a small number of very large procedures that often require hours for C compilation. Solutions such as arbitrarily breaking output into smaller procedures result in at least a two times performance reduction because of additional function call overhead because Verilog execution profile is not localized. Alternative of compiling generated C with a fast compiler that does not perform extensive optimization again results in worse performance by at least a factor of two. A factor of two in Verilog simulation speed is important because from anecdotal discussions for large commercial designs, Compiled Verilog NC is usually only two or three times faster than Verilog XL.

2.4 Compiled Simulator Programming Language Interface (PLI) API Useless

One of the best features of Verilog is that anything that can be modeled behaviorally can also be modeled declaratively (gate level) and also modeled in C using the PLI. PLI allows monitoring value changes by registering and unregistering call backs, assigning values to variables using the scheduler, and attaching call backs to most scheduler activities. Compiled simulators require user to specify an option listing variables that can be effected by PLI. A option to specify that any Verilog object can be effected by PLI is usually also available. First option is almost useless because design specific PLI modeling is impossible. Second option slows down simulation to the point that compiled simulators are slower than interpreted simulators. Because interpreted simulators allow dynamic changes to scheduler, invasive PLI modeling and checking using C code has no additional cost.

Compiled simulators are able to load and execute separate device models (such as commercial RAM or processor models) at full compiled speed because the model does not interact with rest of Verilog design except through its own module I/O ports.

2.5 SDF (Standard Delay Format) Delay Annotation Inconvenient

For accurate gate delay simulation, delays are set using a system task that is called after start of simulation (usually at time 0) by calling **$sdf_annotate** system task. Interpreted simulators allow SDF annotation at any time and selection of particular SDF file to use for annotation or selection of type of min:typ:max delay to use. Compiled simulators require SDF file to be specified before compilation.

2.6 Simulation Speed Interpreter Disadvantage

Currently compiled simulators are faster than interpreted simulators. Speed advantage is largest for RTL models and smallest for gate level models. Interpreted simulation is speed competitive in gate level simulation area probably because primitive operations are more complex. It is believed that Verilog XL can be as much as two times faster than compiled simulation for some gate level designs. It is possible compiler speed advantage can be explained by fact that most performance improvement effort since mid 1990s has been applied to compilers and almost no effort to interpreters.

3. DESIGN OF VERILOG VM

A Verilog VM has recently been added to our commercial statement called Cver. It is intended to correspond to the GNU C compiler -O option. Just as with GNU C -O, only difference is lack of debugger statement break point locality. When -O option is used, just before start of simulation, Cver's statement level directly interpreted internal representation is compiled to VM code. To preserve all features of Cver interpreter, the optimizer is incremental, i.e. when a Verilog statement or PLI operation is invoked which changes the circuit model, the internal high level representation is first changed and then all changed Verilog constructs are recompiled to VM code. Following Verilog XL, Cver reads and elaborates even the largest designs in less than 10 seconds of CPU time. For ease of use, VM compilation time must not increase by more than two times.

3.1 Memory Minimization Design Decision

Cver was originally developed to provide accurate P1364 simulation and to be as memory efficient as possible. From discussions with customers, it appears that Cver uses 1/6 (15 to 20 percent) of the memory of the other full P1364 commercial simulators at the cost of simulation speed around three times slower than the market leading simulators for large commercial designs.

Cver memory requirement reduction comes from design decision to not flatten designs before simulation. Flattening is equivalent to full inlining of instance tree. Cver then accesses nets from the correct instance by accessing value of net by adding instance number to per module base address. The extra based access indirection is not believed to be a major performance problem, but the extra I/O port assigns from lack of flattening is a major performance problem. We believe that non flattening performance loss can be removed by future selective inlining at the cost of some of the current memory use advantage.

3.2 Need for Lower Level VM Optimization

Commercial Verilog simulators are extremely complex and highly optimized programs. Even the slowest simulator is hundreds of times faster than any naive implementation. Even one linear algorithm involving the scheduler or value change propagation causes unacceptably slow performance. The -O option is needed because further speed improvement from execution frequency analysis and algorithm improvement is no longer possible.

3.3 Gnu Profiler Statement Interpreter Routine Call Frequencies

As one might expect from an implementation that directly interprets statements, the most serious speed problem involves expression evaluation and execution of procedural Verilog statements. The following Gnu profiler (**gprof** command) measurements show the problem. Notice some GNU profiler output fields have been omitted from table. The routine frequencies are from the procedural RTL cpu model from the DA Solutions benchmarks (Hillawi[1996]) with the new VM code optimizer off. The frequencies and times are for a a 700 MHZ Pentium 3.

Table 1: non -O gprof output table

Percent	Self Sec	Calls	Routine
23.98	13.57	232304258	__eval2_xpr
21.17	11.98	84768330	eval_binary
10.44	5.91	49965430	__ld_wire_val
10.09	5.71	63053067	push_bsel
8.78	4.97	80948048	__comp_ndx
4.47	2.53	1634302	exec_stmts
3.25	1.84	33694860	__exec2_proc_assign
1.96	1.11	17884822	__st_bit
1.91	1.08	33580392	__eval_assign_rhsexpr
1.82	1.03	38145504	__lhsbsel
1.77	1.00	17884822	__assign_to_bit
1.01	0.57	13644617	__st_val
1.01	0.57	9432782	st_vecval
0.94	0.53	8187767	for_not_done
0.87	0.49	4896409	__chg_st_val
0.64	0.36	385768	move_time

The first five routines are part of the interpreter's stack based expression evaluator. Pushing and popping of the complex expression stack are inlined with macros so they do not show up directly in measurements. The expression stack is complicated because stack elements are pointers to storage areas which can be up to 1 million bits wide. About 64K, four byte words are required to store largest possible 1 million bit Verilog vector because two bits are needed for each bit. The next few routines are mostly statement interpreter overhead routines. The **move_time** routine manipulates the event queue so it can not be optimized away using VM instruction generation.

4. VERILOG VM DEFINITION

Verilog VM uses direct threaded interpretation method defined in Klint[1981]. It is not exactly same as direct threaded method because instructions are written in C and because operand pointers follow direct threaded routine addresses in VM instruction stream. Each C function implementing a VM instruct executes a normal C return when done.

Instructions are executed by calling the instruction's direct threaded C function. There is a global program counter (PC) called **__codp** that points to the current VM instruction. Jump operations are implemented by setting the next instruction pointer (PC) to one before the jump destination. Each VM instruction is defined as a no argument void returning C routine. The VM instructions are compiled during simulator binary compilation with GNU C compiler **-fomit-frame-pointer** option to reduce overhead from C function return. Use of classical method large switch statement or GNU C compiler non standard computed gotos showed significantly worse performance.

VM instruction operands are pointers to arguments or argument table base addresses for values in multiply instantiated instances. The VM is not a stack machine because measurements showed that stack pushing and popping of Verilog's very wide operands was main unneeded computation during expression evaluation.

4.1 Instruction Format

Each instruction contains a pointer to a C procedure to implement the virtual operation and up to four arguments that are usually operand address pointers but can contain literal values or pointers to parts of the very complex Verilog net list or simulation state data structures. A more space efficient method where variable number of operands follows each VM instruction is not currently used. Because VM code rarely adds more than 5 or 10 percent to memory requirements, this inefficient implementation does not add much overhead. It is used for now because it simplifies debugging and incremental recompilation. One can consider the instruction format as if the following four operand structure were cast onto the address of each direct threaded operation routine pointer.

The VM only has a few registers to store state such as the threaded code interpretation loop PC and some registers to keep global values such as old edge value or current event address used as globals across edge detection instructions. There is one temporary work register called **__tmpi** that is always assumed to store bit or memory index by select instructions.

Because VM accesses variable locations and net list information from data structures it shares with statement interpreter, generated VM instruction area rarely adds more than 10% to simulation memory requirements.

4.2 Execution Loop

It is well known (see for example Proebsting[1986]) that the main limiting factor for direct threaded execution speed is efficiency of VM instruction processing code. Our VM uses the following instruction execution loop:

```
while (__codp->proc != NULL)
{
  (*__codp->proc)();
  __codp++;
}
```

A threaded operation pointer to NULL causes exit from VM execution loop and return to the scheduler. Actual sequence of operations caused by NULL is: return to scheduler, find new event, execute event that probably starts by re-entering VM execution loop to execute next timing free block. The instruction just before each NULL schedules the delay or event control event that causes the execution thread to be activated to continue after the event occurs or time control has elapsed. For declarative code since there are no event controls, the NULL operation just acts as jump instruction back into scheduler.

GNU C compiler generates efficient code for this loop and it is not obvious how to hand code this inner loop in assembly to improve speed because of the indirect call through pointer **__codp->proc**. However, this loop is the limiting factor for byte code execution and designs with performance problems can spend 40% of the simulation time in this loop. For declarative gate level circuits, especially flattened gate level designs where Cver interpreter speed is already competitive, there is currently almost no speed improvement because faster expression evaluation and port assignments are canceled by interpreter loop overhead.

4.3 Example of VM Instruction Complexity

Even simple operations are complex in Verilog. The following instruction assigns a non strength scalar to another non strength scalar that appears on the right hand side of some expression. Most of the complexity is caused by need to record the change for change propagation processing using the two level change processing algorithm that matches Verilog XL.

```
extern void __to_isb_chg_from_b(void)
{
    register byte dbp, bv;
    register struct net_t *np;

    dbp = &(__codp->op1.bp[__inum]);
    bv = __codp->op2.bp[0] & 3;
    if (bv == *dbp) return;
    *dbp = bv;
    np = (struct net_t *) __codp->op4.bp;
    np->decl_iops->iop_chgi1 = -1;
    __record_np = np;
    (*np->decl_iops->record_iop)();
}
```

The **__TO_ISB_CHG_FROM_B** VM instruction implements procedural Verilog scalar assign for case where complex right hand side scalar has already been evaluated into a scalar temporary. Change check is needed because if value changes, event driven scheduling algorithm propagates changes to all constructs with scalar **b** on right hand side. The type of recording is determined by the net, not the instruction, so one of the about 20 different declarative record VM routines is called from within the instruction.

The **record_iop** routine is complex. It must check for previous change during same time step. If not, it must add scalar net to two level queue change list, then possibly set up end of time slot variable dump for wave form viewer, and, in addition, it performs recording mechanism book keeping. Because the **record_iop** routine is complex, there was no performance gain for alternative VM where many different **__TO_ISB_CHG_FROM_B** instructions were defined each of which inline one **record_iop** routine. Notice in a programming language this operation would compile to one CPU instruction but for Verilog compilation requires 50 to 100 instructions.

4.4 VM Instruction Operand Access

Because from 15 to 40 percent of the execution time of underlying CPU is spent inside direct threaded code execution loop, operands are accessed from VM direct threaded routine instruction stream. The need for complex operations in direct threaded interpreters is well known (see Proebsting[1995] or Ertl[2002]). Super operations that both load operands and execute operation reduce number of instructions executed minimizing time spent in VM instruction execution loop. If a stack machine were used with separate VM instructions, four or more instructions would be needed: One to load each operand, One to execute operation, and one to store result. This organization results in a VM with a very large number of instructions. Currently VM has around 1000 instructions, but recent measurements suggest more super instructions are needed. On average each assignment and Verilog operator ("bit reducing and" for example) has four or five operand variants where C code implementing the operation is identical except for location from which operands are accessed. Therefore, actual Verilog VM has from 150 to 200 basic instruction types.

There are three temporary storage areas that are used as VM scratch pad storage. One area is used for vectors packed into chunks with an "a" part storing 0 or 1 and a "b" part storing x or z. Storage format is required by the P1364 LRM. The smallest non strength non scalar value requires eight bytes. Another area stores real temporary values. There is also an area for scalar and strength model vectors. Strength and scalar values require one byte for each value because a strength value requires 256 different values to store "0" strength component, "1" strength component and value. Temporary names are assigned during VM instruction generation. Scratch pad storage is allocated and temporary names are assigned to underlying CPU memory locations just before start of simulation. The scratch pad area can be quite large because Verilog expressions are complex and allow concatenates and because Verilog features involving scheduling and value change propagation require temporaries to retain value between VM instructions. Even when very large vectors are not needed, the scratch pad area usually will still require a few hundred words.

Because Cver uses per module VM instruction stream interpretation for which variables are accessed using base address plus instance index offset, each operand can have one of three forms:

1. Direct access where only one instance of module exists or to access scratch pad temporary.

2. Instance specific access where operand contains base of storage and indexed address is used to access correct value from the current instance VM state index register. The same index register is used by the non VM interpreter.

3. Hierarchical reference (usually called cross module reference or XMR) access. Most expression operators do not have VM instructions that access XMR operands. For XMR operands, an instruction to load the XMR value into a temporary is first executed and then the form of the expression operator that uses the pointer form occurs. There are still so many different types of XMRs that about 100 assign (copy) instructions are needed to load and store XMRs.

4.5 VM Instruction Types

The VM contains around 1000 instructions. About 450 instructions are needed to implement the 50 or so Verilog unary and binary operators. Large number of Instructions is needed because each VM instruction directly accesses its operands. By using on average nine variants for each operator, only one instruction is usually needed for common one binary operation assignment.

Super operator or super instruction generation is usually implemented dynamically (see Ertl[2002] for example), but here super operators are designed into VM resulting in VM with over 1000 instructions but most of the instructions are simple variants of basic patterns so scripts have been used to assist with VM instruction implementation. New instructions (variant super operators that combine what previously were multiple instructions into one) are added from performance measurements usually when new styles of Verilog designs are encountered. Currently most pressing need for

more super instructions is in I/O port assignment and constant bit select areas.

Binary Verilog operators require the most duplicated VM instructions because binary operators require three operands: result, first operand, and second operand. Therefore most operands require 12 VM instructions for what is one basic operation pattern. Operations for with both signed and unsigned variants require even more VM instructions.

There are eight instructions for each of the normal addition operands that fit in one word. Either pointer or instance specific form are needed for each of the three operands. There are four instructions for wider than 32 bits form of plus where VM instruction must handle carry propagation. The are only four variants instead of eight because the result of a wide operator must end up in pointer locations (usually a scratch pad temporary) because of some subtle properties of storing wide a/b values. Because of Verilog's four value per bit representation, the plus operation that fits in one word still needs nine lines of C code. Some of the more complicated bit reducing Verilog binary operators require 15 or 20 lines of C. Although there are many operation variants, the C code to implement each is similar to the extent that variants were mostly generated and checked using Perl scripts.

There are about 340 value assignment instructions. Most of these are required by the complex nature of Verilog assignments. There are many complex rules for size change and sign removal assignments in Verilog. There are also bit selects, array selects, part selects, and simple assignments. Each of which can appear on the left hand side or the right hand side. Also there are many assignment variants that differ depending on type of change processing needed. About 100 of 340 instructions are related to the various types of XMR load and store assignments. Another 50 are needed to pack and unpack Verilog memories because memories are packed to the bit.

About 20 VM assign instructions are needed to handle force and release processing. If elaboration time source analysis determines that a variable can be forced, a special VM instruction that jumps over the assignment VM instructions when force is active is generated. The incremental compiler regenerates VM instructions for newly forced variables during simulation.

The advantage of so many VM instructions is that all simple assignments of the form "V = V", "V = C", "V[constant index] = V", "V = V[constant index]", "V[range:range] = V", "V = V[range:range]" only require one VM instruction. In the above patterns, V stands for variable, C for constant, "constant index" is any constant bit select index, and range is the constant part select end of range index. Also, all assignments to simple variables with only one binary or unary operand on the right hand side require just one VM instruction. Similarly, wide forms usually require only two instructions because usually wide values must be stored into a scratch pad temporary before being stored into assignment location.

4.6 Control Flow VM Instructions

The other 200 instructions usually do not have operand difference variants. Some implement normal VM control flow and condition testing operations. There are a few instructions for implementing Verilog case statements. Verilog case statements are complicated compared to C switch statements because of the various matching rules for unknown (x) and undriven (z) values. Jump tables can still be used for "don't care" case statements.

There are usually two or three VM instructions for each procedural delay control end of timing free block schedule type. The event and delay control schedule instructions also set the schedule event fields that cause event activation to execute the right VM instruction. Finally, there is usually only one instruction to implement each of the rare special case Verilog statements such as **cause** or **disable**.

4.7 Value Change Propagation VM Mechanism

There are instructions for recording and propagating declarative and procedural value changes. The VM instructions are kept in a separate instruction stream table for each net. Change propagation instructions are mostly conditional jumps used to filter out non matching instances or edges. There are some instructions for skipping PLI operations because PLI supports both registering and removing callbacks and the incremental compiler tries to generate the smallest possible change for call back removal.

The C program run time stack is used to store context for change recording. When either procedural or wire changes are propagated (front side of scheduler change processing), one special pseudo VM instruction is executed. Most variants of this one instruction are quite complex (about 40 lines of C code). They use the CPU's local variable run time stack to store both VM context and simulator state context and then recursively execute a temporary second copy of VM execution loop. This works because propagate VM instructions never suspend back to the scheduler until all processing is completed. Suspension in this case is back to the special pseudo VM instruction which restores VM state and then returns to scheduler.

This is not an elegant solution and violates the concept of "real" VM but we have not found a way to store context that is nearly as fast. The mechanism will not map easily if in the future a native CPU instruction compiler is implemented. Another advantage of this "pseudo" VM is that except for some rare cases, the incremental compiler only needs to regenerate VM instructions for the effected net.

4.8 Reason For Change Propagation Complexity

A consequence of the requirement that procedural and declarative constructs behave identically causes a need for different change propagation for registers and wires. Register change propagation must be immediate or events that are set to trigger on edge changes are missed. These differences lead to many more change VM instructions and cause need for recursive VM execution loop.

The following Verilog code activated on clock edges shows why procedural edge changes must be processed immediately.

```
always begin
    #10 active = 0; active = 1; active = 0;
    #10 ;
end
initial begin
    @(posedge active) ;
    ... more statements ...
```

end

Although this is not explicitly defined in the P1364 standard LRM, if procedural register variable change processing is not executed immediately, when the first block suspends, **active** will be 0 so the second block will not see a positive edge, but a positive edge occurred during first block execution. The requirement that change processing for procedural code run immediately insures that the statement setting **active** to 1 will schedule the positive edge event control activation in the second block.

In contrast, declarative wire change propagation must use the two stage event queue algorithm defined by the LRM. The two stage event processing algorithm works by first processing all events for the current time. All wire changes and events scheduled during the current time for the current time are saved on the second stage list. When current event queue is empty, the second stage saved change list is made the current active queue and processed. This two stage process continues until the current event queue and change list are completed and next change list and event queues are empty. This is complicated but by convention Verilog simulations must use the two stage event queue algorithms or other parts of design flows will fail.

5. THE VM COMPILER

Because a production Verilog compiler front end already existed, non optimized instruction generation was not difficult. However, since Verilog XL is fast and optimized to the extent that many routines are hand coded in assembly, the full mechanism of an optimizing code generator needed to be implemented. First, the internal statement level data structure is traversed and three address tuples are generated (see Aho[1986], pp. 466-473 for discussion of three address tuples). Three address tuples are then optimized and processed so that for each three address tuple exactly one VM instruction is generated. VM instruction generation is then easy because of the one to one property. Final VM instructions generation finds VM instruction that matches the three address tuple's operands.

5.1 Basic Three Address Tuple Generation

There are separate three address tuples for labels and for jumps that are connected using the data structure described in Wulf[1975], pp. 108-115. Three address tuples are generated using a pattern matching technique similar to the one used in the original Bell Labs Unix C compiler (Ritchie[1979]). Except here VM tables are represented as patterns coded in C functions where routines are called sequentially to match the various operand patterns. Difference is required because there is no obvious complexity order relation for Verilog operand pattern types and because full computer program conditional statement computation power is need for matching some special cases. The three address tuple fields are first filled (filling may cause some subexpression three address tuples to be generated) then pattern describing location result ends up in is used for VM instruction generation matching. For example, field **FT_XWChg** describes a wider than 32 bit left hand side hierarchical reference that requires change propagation. It is one of the most complex patterns and can require three or four, tuples depending on right hand side field pattern properties.

During three address tuple generation, timing free regions are indicated by doubly linking start tuple addresses and statement data structure addresses. A time free region is a sequence of variable statements that do not require any scheduling activity. End of each timing free block has a suspend tuple that later will become a VM instruction with **proc** field NULL. Conventional programming language basic blocks are not computed and optimized because loops that do not contain timing controls are rare. VM optimizer works on each timing free region in turn. No state information from different timing free blocks can be saved because suspend into scheduler may result in the following timing free region disappearing since it may need to be recompiled by the incremental compiler before scheduler returns to the "next" timing free block.

5.2 Tuple Optimization

First phase of optimizer inserts copy tuples when VM instructions with needed operand patterns do not exist. Although LRM allows optimizer to reorder computation in timing free blocks, some Verilog XL conventions must be followed or legacy Verilog models will not function correctly. Final step is to apply a number of optimizations recursively until no more progress can be made. Some optimizations are: removal of jump to next, jump to jump, unreachable code, and conversion of sequences of three address tuples to simpler sequence (usually to one tuple) using pattern matching tables to locate the sequences. The sequences are determined from measurements of VM instruction frequencies and then used to build the patterns by hand.

The incremental compiler works by determining which timing free regions and net propagation streams need to be regenerated, frees the old VM instruction areas, runs the normal compiler to generate new three address tuples for the changed constructs, optimizes the tuples, and finally generates new VM instructions and adjusts scheduler pointers so that the event processor executes new VM code. It is possible for a PLI change such as adding a change call back to a variable used as a left hand side hierarchical reference to require VM instruction regeneration in many different modules, but such cases are rare. Usually, invasive internal circuit representation change from debugger, PLI, or SDF annotation calls just require regeneration of declarative propagation tuples for effected nets.

6. SPEED IMPROVEMENT RESULTS

Currently, best possible VM execution speed improvement for procedural code with complex expressions and no time movement is 10 times (nominal 100 seconds to 10), but currently most procedural designs run slightly better than two times faster (nominal 100 seconds to 45). Some circuits such as flat gate level designs show no speed improvement when -O option is used because speed improvements from faster I/O port assign execution is balanced by overhead from VM instruction execution loop.

The measurements presented here are between Cver statement interpreter and Cver VM execution. All measurements were made on a medium speed X86 Pentium 3. Relative measurements are same for other X86 architectures such as Sparc, Power PC, and PA-Risc.

6.1 GNU Profiler Compiled Simulation Routine Call Frequencies

Table 2: rtl cpu model gprof -O output

Percent	Self Sec	Calls	Routine
17.21	4.58	1634302	exec_iops
12.81	3.41	97722930	__to_tmpi_from_lh_rng_w
9.65	2.57	49040952	__to_w_from_wide_bsel
9.13	2.43	90237793	__iop_jump
6.76	1.80	80195348	__iop_tmpi_minus1_jump
6.01	1.60	19639630	__iop_booland_p_p_p
4.43	1.18	14771823	__iop_boolor_p_p_p
3.87	1.03	21767850	__iop_minus_p_p_p
3.38	0.90	17188236	__iop_relne_p_p_p
3.19	0.85	14787960	__to_wide_bsel_from_w
2.25	0.60	13977834	__to_w_from_w_bsel
1.69	0.45	7658199	__iop_sign_relgt_p_p_p
1.54	0.41	10523346	__iop_condf_jump
1.50	0.40	11484708	__to_w_from_b
1.43	0.38	385768	move_time
1.01	0.27	2860920	exec_decl_iops
0.90	0.24	7041648	__pop_itstk

The following table gives GNU profiler routine call frequency measurements for the DA Solutions procedural RTL circuit discussed above with VM optimizer -O option on. The results show that all the expression and statement evaluation interpreter overhead has been removed at the cost of 17% of time spent in VM instruction execution loop. Simulation time is reduced to 45% of the previous time (nominal 100 seconds to 45).

Except for VM instruction execution overhead, jump VM instruction, and some Verilog binary operators; high frequency routines are those used to evaluate bit selects and **if** statements. These constructs along with I/O port assignments are most frequently executed VM instructions for many designs, and also the most serious current VM compiler code quality problem areas. Still many large industrial designs show two times speed up (nominal 100 seconds to 50). This RTL CPU model has low cross I/O port information flow so the I/O port assign code quality problem does not show up until the **__pop_itstk** routine at the bottom of the table. Also inherent value change and event scheduler simulation overhead routines (see bottom three table lines) that can never be eliminated by VM instruction execution, take only a few percent of total execution time. The very high frequency of the unconditional jump VM instruction is surprising and we are currently investigating its cause.

7. CONCLUSIONS AND FUTURE WORK

The new Verilog VM execution optimizer is extremely successful and has the following advantages:

1. VM Verilog execution has proven to be very convenient. VM and compiler development is much easier than expected. Debugging, measuring to improve compiler quality, and tracing features have been surprisingly easy to add.

2. Results exactly match current commercial quality interpreter.

3. Feature preserves all the advantages of Verilog XL style statement interpreter.

4. Features offers promise of closely approaching compiled to machine code simulation speed.

5. Speed improvement is usually two times (nominal 100 seconds to 50) which is getting close to three times (nominal 100 seconds to 33) speed up that a customer suggested would make Cver speed competitive with other simulators.

Previously Cver speed improvement had run into a wall. Either new speed improvement ideas did not increase simulation speed, or idea improved speed for some models but slowed down others. VM compiler still needs much work but each simple improvement has resulted in a few percentage points of speed improvement. Currently we are working on improving three address tuple code quality by adding expression optimizer used before three address tuples are generated and by adding more super operators where only one instruction is needed for common constant bit select and I/O port assignments.

Main current three address tuple code quality problem area is inefficient I/O port continuous assignment tuples, extra tuples for accessing constant bit and memory selects, and inefficient **if** statement condition test tuples. For very common scalar port source assign to constant bit select port sink where the two problem areas occur together, currently four or five tuples are generated but only one is needed. For **if** statements, currently tuples are generated to evaluate condition expression, then to convert expression result to logical value, then the conditional jump tuple is generated.

Least successful aspect of project is lack of gate level simulation speed improvement. Flat gate level circuits often do not speed up at all. There is some evidence that Cver unoptimized interpreter's gate level performance is about as fast as the fastest simulators, so maybe result is not surprising.

Other problem that is also not surprising is that speed improvements are limited by Amhdahl's law (Amdahl[1967]). Namely, designs that require complex scheduling or accurate path delay declarative simulation show limited speed up because even if expression evaluation, change processing, and port assignment time is reduce to 0, those tasks require only 25 to 33 percent of total simulation time. Best possible improvement without discovering better algorithms is reduction of time from by one third (nominal 100 seconds to 67).

Next phase will involve adding more complex optimizations such as adding operator reordering expression optimizer, adding optimizer to reuse common subexpressions in timing free blocks, optimizing conditional jumps, and adding some instance inlining especially for leaf instances. In the longer run, it should be possible to write program to map VM instructions to native machine instructions at the cost of portability and some flexibility.

In the future, interpreter ease of development may allow discovery of more complex but orders of magnitude faster simulation algorithms that preserve current behavior but use better data structures or algorithms. See (Lawler[1976]) for discussion of matroids that may be one such data structure.

Interpreted VM Verilog simulation has proven to have so many advantages that we believe in future all full P1364 accurate delay Verilog simulators will be interpreters. Main current challenge is development of better compilation and VM implementation techniques to increase simulation speed. It is also possible future computer architectures will allow

more efficient VM instruction execution.

8. REFERENCES

Aho[1986] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, 1986.

Allen[2002] Allen, R., and Kennedy, K. *Optimizing Compilers for Modern Architectures.* Academic Press, London, 2002.

Almasi[2002] Almasi, G., and Padua, D. MaJIC: compiling MATLAB for speed and responsiveness. *Proceeding ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02),* 2002, 294-303,

Amdahl[1967] Amdahl, G. Validity of the Single-Processor Approach to Achieving Large-Scale Computational Capabilities. *FIPS Conference Proceedings,* AFIPS Press, 1967, 483.

Becker[1996] Becker, M. Faster Verilog simulation using a cycle based programming methodology. *Proceedings 1st International Verilog HDL Conference,* 1996, 24-31.

Bell[1973] Bell, J. R. Threaded code. *Communications of the ACM, 16(6),* 1973, 370-372.

Cadence[1997] Cadence Design. Verilog XL and Verilog NC Reference Manuals. Cadence Design Systems, 1997.

Earley[1975] Earley, J. High level iterators and a method for automatically designing data structure representation. *Computer Languages vol. 1, 1975,* 321-342.

Ertl[2001] Ertl, M. A., and Gregg, D. The behavior of efficient virtual machine interpreters on modern architectures. *ILNCS-2150 Euro-Par 2001,* 2001, 403-412.

Ertl[2002] Ertl, M. A., Gregg, D. Krall, A., and Paysan, D. Vmgen–A generator of efficient virtual machine interpreters. **Software–Practice and Experience. vol. 32(12),** 2002, 265-294.

Ertl[2002b] Ertl, M. Threaded code variations and optimizations. *Forth-Tagung 2002,* 2002.

IEEE[1996] IEEE Standards Board. *IEEE Std 1364-1995 Verilog Hardware Description Language Reference Manual.* IEEE, New York, 1996.

Hillawi[1996] hillawi, J. DA Solutions Public Domain Benchmarks. Unpublished, Available at URL: http://www.pragmatic-c.com, 1996.

Jennings[2000] Jennings, J., and Beuscher, B. Verischemelog: verilog embedded in scheme. *SIGPLAN Notices vol 35(12),* 2002, 123-134.

Krall[1993] Krall, A. and Berger, T. An executable intermediate representation for incremental global compilation of prolog. Technische Universitaet Wien, 1993.

Lawler[1976] Combinatorial Optimization: Networks and Matroids. Holt Rinehard, and Winston, New York, 1976.

Leroy[1990] Leroy, X. The zinc experiment: An economical implementation of the ML language. INRIA Technical Report, 1990, 117.

Lindholm[1999] Lindholm, T. and Yellin, F, *The Java Virtual Machine Specification, Second Edition.* Addison Wesley, New York, 1999.

Klint[1981] Klint, P. Interpretation Techniques. *Software–Practice and Experience, vol. 11(10),* 1981, 963-973.

Meyer[1988] Meyer, S. J. A data structure for circuit net lists. *Proceedings 25th Design Automation Conference.* IEEE, 1988, 613-616.

Moorby[1989] Morby, P. and Bryant, R. *The Verilog HDL.* Kaufman, New York, 1989.

Morgan[1998] Morgan, R. *Building an Optimizing Compiler.* Butterworth-Heinemann, Boston, 1998.

Nagel[1975] Nagel, L. SPICE2: a computer program to simulate semiconductor circuits. University of California, Berkeley, ERL-520, 1975.

Proebsting[1995] Proebsting, T. A. Optimizing an ANSI C interpreter with superoperators. *ACM SIGPLAN Principles of Programming Languages (POPL '95).* 22-332, 1995.

Ritchie[1972] Ritchie, D. A tour through the Unix C Compiler. in *Unix Seventh Edition Manual.* Bell Laboratories, 1979.

Weigang[1985] Weigang, J. An Introduction to SSC's APL Compiler. *APL Quote Quad vol. 15(4),* 1985.

Wulf[1975] Wulf, W., Johnsson, R.K., Weinstock, C.B., Hobbs, S.O., and Geschke, C.M. *The Design of an Optimizing Compiler.* American Elsevier. New York, 1975.