Hardware Logic Simulation by Compilation

Craig Hansen

MIPS Computer Systems, Inc.
930 Arques Avenue, Sunnyvale, CA, 94086

Abstract

A behavioral and logic simulation system which uses extensive optimization and compilation techniques to obtain high performance is described. It incorporates data flow analysis to optimize the evaluation of unordered assignment statement that define a hardware structure, and extract clocking rules. An integral code generator produces efficient assembly code for three different machines, and an associated run-time library provides a flexible interactive debugging environment.

Introduction

Terse, which names both the language and the compiler system, is designed for the express purposes of the description and simulation of hardware logic. *Terse* models can be generally as accurate as logic-level simulations, yet can run extremely quickly, due to extensive pre-compilation and optimization.

Terse Language

The *terse* language contains three classes of statements:

Declaration statements define the identifier name, provide a short description, define the storage class, control optimization levels, and define the width of signals (variables).

Assignment statements define the value of each signal in terms of other signals in the system, or in the case of clock signals, define a time-periodic value for the signal. Assignment statements also define a boolean "clock" or enabling value, which controls whether the assignment is active; inactive assignments do not change the value of the signal.

Hierarchy: while the execution model of *terse* is essentially flat (non-hierarchical), the language provides mechanisms to define portions of systems within private name spaces (static scope), and to invoke multiple instances of functional blocks within a system (block instancing).

Signal Classes

There are several classes of signals handled by *terse,* namely constants, knobs, clocks, variables, enables, and assertions.

All of these signals have values defined over a two's-complement integer number space, with arbitrary width. Constants, knobs, and variables can be of either scalar or vector type; clocks, enables, and assertions must be scalar and boolean-valued.

Constants have values that can be determined at compile-time, and do not change. *Knobs* have values that can be changed at run-time and have initial values determined at compile-time. *Clocks* are boolean-valued, and have a period and transitions defined at compile-time. *Variables* have values defined by data-flow equations, and change synchronously with changes in the values of clock signals. Variables have zero initial values.

Enables are variables whose value has a major impact on the data flow or timing of the simulated system. When so directed, *terse* will perform additional optimizations involving these signals, improving the simulation performance and timing accuracy. These optimizations add exponentially to compile time, so the signals over which the optimizations are made must be chosen carefully.

Assertions are used to improve the level of code optimization and when properly used, they provide additional information to *terse* that significantly reduces the number of separate paths that must be explored when analyzing *enables.* The assertions indicate to *terse* that a particular path is never used, by asserting that the particular combination of enable signals that select the path never occur.

Data Flow Equations

The logical data flow of the simulated machine is specified via logic equations that define the value of each signal in terms of other signals. A variety of operators may be used to define the data flow equations: the boolean or bitwise operators: *and,* (&), *or,* (|), *xor,* (^) *not,* (!), (~); arithmetic operators: +, -, *, *div*(/), *mod*(%), are defined in conventional fashion.

Because multiple-bit signals often need to be

operated on as separate bits or bit fields, *terse* directly implements operators that perform bit field catenation (||), repetition of a bit into a bit field (rep), bit extraction (x[y]) and bit field extraction (x[y..z]).

Relational expressions can be formed using the operators: (<, <=, =, !=, =>, >), comparing the operands using two's complement arithmetic. Expressions of the form: $a < b < c$ are legal, and are equivalent to $(a < b)$ & $(b < c)$.

Multiplexors or selectors may be specified, using syntax similar to the C "? :" operator. To check for multiple simultaneously selected values, a run-time assertion failure is caused if more than one selection expression is true at once.

Because tri-state busses may be driven from more than one functional block, multiple assignments are permitted to each signal, provided that all but at most one signal are in the form of tri-state outputs: equations that have clocking expressions.

A built-in function for associative searching is included, to easily simulate associative arrays, such as TLB's and caches. External functions written in conventional HLLs can be called from *terse*, to interface to other models.

Terse Compiler

This section describes the internal strategies of the *terse* compiler. The goal of the compiler is to provide fast 2-state [2] logic simulation on a general-purpose machine, using data flow analysis [1], optimization, and direct assembly code generation for the target machine.

Requirement for Optimization

Optimization is required to identify the width of operands, to identify clocking rules, and to break circular paths in the data flow equations, and for accurate timing analysis, to break false paths that influence the apparent timing of the design.

Terse relies on a compile-time analysis to determine, for each clock phase, what signals may change, and in what order the signals must be evaluated to insure that the values converge in a single evaluation of the signals. Such determination is made difficult if the equations have "circular paths," that is, an unbroken path of assignments in which the value of signals depend on each other, so that there is no legal topological sorting of the assignments, in which all signals have been evaluated prior to their use. Because *terse* does not provide for event-driven simulation it is mandatory that the static analysis successfully remove circular paths.

These circular paths can be broken by static analysis that is based on substituting the actual value that the clock signals will attain during each clock period into the signal equations. The resulting equations are then simplified, using rewriting rules that eliminate and combine constant expressions.

Often, the clock-value substitution will indicate that particular assignments are not enabled during particular clock phases, which breaks the majority of circular paths in typical uses of *terse*. However, some cases require that the value expressions themselves be simplified in order to break circular paths. This normally occurs as multiplexors are controlled by clock signals, whether directly or through intervening logic.

Optimizations Performed. For each boolean operator, *and, or, xor,* constant sub-expressions are identified and collected, and if the collected constant is the identity element, is is removed. At high levels of optimization, these boolean expressions also undergo bit field optimization, boolean optimization, and inversion optimization.

Bit field optimization turns operations of various kinds (bit extract, bit field extract and comparison, and extract of a variable bit from a constant field) into a single masked comparison. If the size of the expression is five bits or less the expression is turned into a bit extract from a constant field. This form of optimization is particularly useful in code segments that decode instructions or other control words by and-ing together several extracted bits.

Boolean optimization combines boolean operator sub-expressions that are themselves bit extractions from a common sub-expression (such as $x[y]$, $z[y]$) together into a single bit extraction from that common sub-expression. For example, $x[y]$ and $z[y]$ is reduced to $(x \text{ and } z)[y]$. Further constant folding is performed on the $(x \text{ and } z)$ expression. With the combination of the bit-field and boolean optimizations, any boolean function on the bits of a five-bit or smaller variable is reduced to a single expression of the form $x[y]$, where x is a constant and y is the variable.

Logical inversions are processed to propagate constants and to try to remove extraneous inversions, such as doubled inversions, or inversions of relational expressions. Similarly, for arithmetic inversion (-) *terse* propagates constants and removes double inversions.

Arithmetic (+, -, *, /, mod) operators are processed to remove constants or turn large expressions into constants where possible.

Bit field operators (rep, extract bit x[y], extract field x[y..z], and catenate ||) are processed to propagate constants and to identify the actual width of these expressions. The width of catenated expressions is checked to match against the known width of the expression; if it is not known, the resulting width is propagated up the expression tree.

Other optimizations involving bit fields include com-

bining catenated bits or bit fields together when they are extracted from adjacent positions of a common subexpression. When combined with other boolean optimizations, this permits expressions written as if implemented in bit-at-a-time boolean logic on multiple-bit signals to be processed in entire words. Because some logic designers place explicit names on signals that are merely single bits extracted from a wider bus, *terse* will optimize across equation boundaries to find opportunities to combine adjacent catenated bits.

The question (?) operator is folded using the identities $(0?x:y)=y$ and $(1?x:y)=x$. If optimization is selected, the further identities $(x?1:y)=(x|y)$, $(x?y:0)=(x\&y)$, $(x?0:y)=(x\&y)$, $(x?y:1)=(x|y)$ are used to simplify boolean-valued expressions.

Relational operators are constant-folded, and if optimization is selected, expressions of the form $(x \text{ and } y)=z$ or $(x \text{ and } y) \neq z)$ where y and z are constants are examined for values that cause the expression to be true or false independent of the value of x. Comparisons between any 5 bit or smaller expression and a constant is turned into the bit-field extraction of a constant. The latter optimization permits boolean operator (and, or, xor) optimizations to be performed on the resulting expressions, so that an expression such as $((x<5) \text{ and } (x>2))$ is folded to $(31[x] \text{ and } -8[x])$, then folded to $24[x]$.

Extraction of Data Flow Paths. After global expression folding has been performed on all signal equations, the basic data dependencies are collected into explicit lists of successor and predecessor signals for each signal. Unused signals are detected by finding no successor signals, just as no predecessor signals indicate compile-time constants or knob-dependent variables.

Clock analysis. The value of all clock signals is determined, as the cycle time is broken into intervals, called phases, over which the value of all clock signals are constant. At this point, the number of clock phases is known, and a separate copy of the data flow equations during each clock phase are created. The values of the clocks are set individually for each clock phase, and the expression folding code is invoked again to remove the explicit clock signals from each expression. The successor and predecessor list of each signal is again analyzed, as gates and multiplexors controlled by clock signals or clock-dependent signals, when evaluated, remove false paths from the data-flow equations.

These basic relations are applied iteratively to the data flow equations during each phase until all signals which can be proved to be unchanging are so marked.

By looking for signals in the phase-dependent data-flow equations which have no successor, the computation of signals on phases for which the value has no long-term effect on the result of the simulation can be removed. This optimization alone yields a 10-30% improvement in performance, and normally speeds compilation as well.

Storage Allocation

In the run-time environment, the values of all signals are maintained over a finite time history. In order to keep this historical record efficiently, the run-time system maintains a circular list of storage areas. Each storage area holds the values of each scalar signal for each phase at which a value changes. The storage represents the current value of the signal until the next known phase in which the signal may change value.

Vector-valued signals are allocated two additional words for each active phase, to hold the value of the enable flag and the subscript value in addition to the previous value of that particular subscripted signal.

Knob and vector-valued signals maintain their current values in a separate array. In order to obtain past values of elements of vector-values signals, the value in the array must be "down-dated" with values held in the rotating scalar storage area.

Signals are allocated at least one machine word of storage (32 bits), and signals which are wider than 32 bits are rounded up to the next integer number of words.

Very large arrays are not allocated directly, as they may use potentially large amounts of virtual address space. Instead, space is allocated for pointers, each of which points to a 256kbyte block of space which is dynamically allocated as it is referenced. In this way it is possible to allocate an array to represent an entire 32-bit address space.

Timing Analysis

Sufficient information is present in the data flow model to accurately estimate the static worst-case timing of a modeled hardware system, provided that timing data is indicated in the individual data flow equations. Unless additional analysis is performed on enables, the static analysis of timing tends to be overly pessimistic. With enable-level analysis, however, the worst-case timing of the system can be accurately estimated over all possible system inputs.

Code Generation

The last major step is the generation of code from the internal data flow equation representation. The code generator is sufficiently portable to generate code for three diverse machines: a MIPS Computer Systems, M-Series RISC processor system, a DEC VAX BSD UNIX system, and a Sun Microsystems Sun-3 (MC68020) system.

Code generation strategies, such as instruction selection, and register-allocation, are generated from tables or parameterized by the number of available registers, so that they can be accomplished in a target-independent fashion. In a few cases, major choices in strategy must be accomplished by testing the identity of the target machine; these include the use of three-register operations and the ability to set a boolean value in a register based on a relational comparison.

Register Allocation. Register allocation uses a virtual stack technique with modifications to explicitly release registers. Registers that are released are tagged with their current value so that a limited sort of common subexpression optimization can be performed during the code generation phase. Released registers are shuffled around in a pseudo-random/FIFO fashion to effect the common subexpression checking and also to fully utilize the registers of the MIPS processor. By doing so, the code reorganizer, which is an integral part of the MIPS assembler, is better able to schedule and fill load and branch delay slots. Where necessary, the register allocator will insert register save/restore code so that the virtual stack depth can exceed the number of available registers for the register stack.

Machines that support three-register operations (VAX, MIPS) permit the generation of code which leaves the source operands untouched. This enhances the chances that the common subexpression code can find a matching expression, at the expense of a heavier use of registers. This is serendipity, since the MC68020, which doesn't provide three-register instructions, has the fewest registers available for the virtual register stack.

Bit Field Operations. The handling of bit field operations (extract, catenate) is relatively elaborate. While the VAX has bit field operations that are fairly general, *terse* places the operands into registers, and the VAX architecture will not permit groups of registers to be addressed as a bit field. Consequently, all bit field operations are broken down by the code generator into operations on individual words, and the VAX, MIPS, and MC68020 processors can all be handled in a similar fashion. Certain cases, such as shifts by zero, 1, or 31 bits can permit instructions to be dropped out; this is provided by simply selecting separate format strings for these cases.

Assertions, that is, expressions that should be logically true at all times, are generated and checked for true values. A failure of an assertion causes a run-time error. Assertions are automatically generated that check that no more than one selection line is active in a multiplexor; this is a common cause of error in CMOS multiplexors with bit-wise encoded select lines.

Performance

Terse is in daily use at MIPS Computer Systems Inc., for the development of chip and system-level computer system products. Logic models of up to 15000 source lines, using about 5000 distinct signals, and 24 clock phases, have been successfully simulated using this system, with a compilation time of 3.3 CPU minutes on a MIPS M/1000 (10 VAX MIPS) system. A two-clock-phase full logic-level simulation of the 100K transistor MIPS R2000 [3] processor, compiles in 6.4 seconds and runs at a rate of 720 clock cycles/second on an M/1000, or about 50 clock cycles/second on a VAX-11/780. The simulation has 1010 distinct signals, 665 of which are single-bit signals. However, the average width of signals is 7.6 bits, so the simulation gains substantial speed from the direct treatment of multiple-bit signals. The larger model compiles more slowly because of the greater number of clock phases.

This system has been instrumental in the rapid development of chip and system-level products at MIPS. [3] The high speed of simulation is provided while still providing a highly-interactive development environment. In addition, the accuracy of the RTL-level simulations are high enough to directly and mechanically match logic-level implementations derived from the chip artwork. Thus, *terse* system provides a database that is maintained from the beginning stages of exploratory design to the final checking of chip artwork. In addition, the efficiency of the *terse* simulation system permits the actual chip models to be used to simulate entire systems with the same high accuracy.

Conclusions

Terse is a powerful system for the development of simulation models of hardware designs. The compiler incorporates data flow analysis as an integral part of the compilation process, to provide high levels of optimization and derive useful design and timing information from the basic database, so that the clocking rules of the logical system can be deduced directly from the defining equations. The resulting compiled model, combined with the run-time system provides a richly featured, but extremely fast simulation environment.

References

1. Ashok, V., *et al,* "Modeling Switch-Level simulation Using Data Flow," 22nd Design Automation Conference, p. 637, June 1985.

2. DesMarais, P.J., *et al,* "A Functional Level Modelling Language for Digital Simulation," 19th Design Automation Conference, p. 315, June 1982.

3. Rowen, C., *et al,* "RISC VLSI Design for System-Level Performance," VLSI Systems Design, p. 81, March 1986.