EFFICIENT RANDOM VECTOR VERIFICATION METHOD FOR AN EMBEDDED 32BIT RISC CORE

Chang-Ho Lee, Hoon-Mo Yang, Sung-Ho Kwak, Moon-Key Lee, Sanghyun Park*, Sangyeun Cho*, Sangwoo Kim*, Yongchun Kim*, Seh-Woong Jeong*, Bong-Young Chung*, Hyung-Lae Roh*,

Dept. of Electrical and Computer Engineering, Yonsei University, Seoul, Korea
*MCU Team, System LSI Business, Samsung Electronics Co., Yong-In, Korea

Abstract

Processors require both intensive and extensive functional verification in their design phase to satisfy their general purposability. The proposed random vector verification method for CalmRISC™-32 core meets this goal by contributing complementary assistance for conventional verification methods. It adopts a cycle-accurate instruction level simulator as a reference model, runs simulation in both the reference and the target HDL and reports errors if any difference is found between them. These processes are automatically performed in the unified environment. The instruction level simulator, the core part in the verification environment is able to simulate almost every aspect of RISC processors from functional behavior of each opcode to timing details in the pipeline flow in fast speed. Its design style from microprogramming scheme also makes its structure modular and flexible.

I. INTRODUCTION

This paper proposes random vector verification methodology to contribute complementary assistance for conventional verification methods. Though random vectors are likely to cause more inefficiency by themselves in that their verification purposes are rather ambiguous, they are easy to generate and maintain. Furthermore, they often cover unexpected corner cases in experience since they are intrinsically free from designers' bias. Accordingly, verification efficiency is greatly enhanced when the random vector method cooperates with the checklist one. We also developed an instruction level simulator as a reference model. The instruction level simulator is a clock-based cycle-accurate simulator written in C language. It imitates not only all the functional behavior of instructions but also timing details in the pipeline flow with reasonably high speed.

The proposed verification environment compares the trace from a target HDL model and the one from the reference model and analyzes the comparison results automatically. This approach accordingly provides more convenient environment for designers.

To discuss the main topic, this paper will mention on architecture of CalmRISC™-32 briefly as backgrounds in section II. It will present overall structures of the proposed random vector verification environment in section III and deal with the instruction level simulator in section IV. In conclusion, it will analyze statistics in our past verification experience and recommend its appropriate application field.

II. OVERVIEW ON ARCHITECTURE OF CALMRISC™-32 PROCESSOR

CalmRISC™-32 is a 32bit scalar RISC for low-power embedded applications. It comprises IU, FPU, Cache and in-circuit debug unit. However, the proposed verification environment constricts its scope only on IU. In the view of the random vector verification environment, the most significant features of CalmRISC™-32 are as below.

Instruction set: To increase code density, it has an instruction set rather different from that of typical RISC processors. Its data size is 32bits while its code size is mostly a half of its data size. Its code sizes are variable for the multi-byte immediate load instructions. The ones for half-word is 4 bytes long and the ones for word 6 bytes long. It also adopts CISC style multi-cycle instructions. The typical cases are POPQn/PUSHQn, the multiple pop/push instructions, and BIT[R/S/T/C], the bit-oriented memory read-modify-write instructions.

Pipeline structure: Its pipeline structure consists of five stages; instruction fetch (briefly IFE), decode (DEC), execution (EXE), memory (MEM) and writeback (WRB). Instruction fetches in IFE and data loads/stores in MEM are performed independently through different ports because it adopts Harvard architecture in cache. However there is one exception; it also defines LDC (Load Data from Code memory) instructions, which load data through the port for instruction fetch in MEM.

Control flow: It supports delayed branch instructions. Every delayed branch has a single delay slot because branch address calculation is done in DEC. It also supports precise exceptions. The proposed environment excludes interrupts and coprocessor exceptions. If any exceptional cause is detected, its target address will be available in the next clock cycle.

Register dependency: It supports both register bypass and memory lock. It has two special cases for memory lock besides general memory load operation; LDC instructions always cause memory lock to prevent code port confliction, and BIT[R/S/T/C] instructions do so if the next instruction is dependent on T bit since they modify T bit in the MEM stage while other instructions do so in EXE.

III. THE OVERVIEW ON RANDOM VECTOR VERIFICATION ENVIRONMENT

The figure 1 shows main process of random vector verification environment. The process passes through total 5

steps. The names in the square parentheses in the figure indicate extensions of output files generated in the steps. The main purpose of each step is as below.

1. Generate an assembly source file for random test vectors. [ASM]
2. Assemble the source file and generate output files in the sake of verification. HEX (or DAT) file is a list file containing code (or data) for the target HDL stimulus while ILS file is a binary executable for the instruction level simulator.
3. Perform simulation on the target HDL and generate a VTR (Vertical TRace) file.
4. Perform simulation on the instruction level simulator and generate a HTR (Horizontal TRace) file.
5. Compare signals between them and report any difference.

A VTR file comprises entries containing values of primary external signals and registers sampled at every positive edge of clock in the HDL model while the structure of a HTR file entry is slightly more complex. It defines a set of 5 slots for one iteration of pipeline flow and each slot contains its related events and beginning time also used as an index to

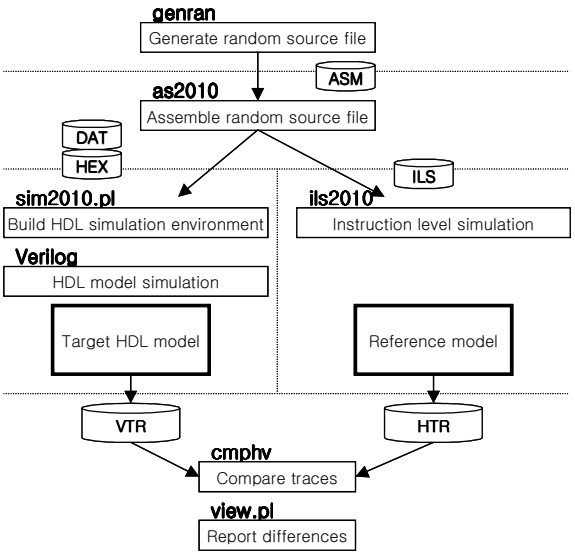

Fig. 1 Procedure of random vector verification

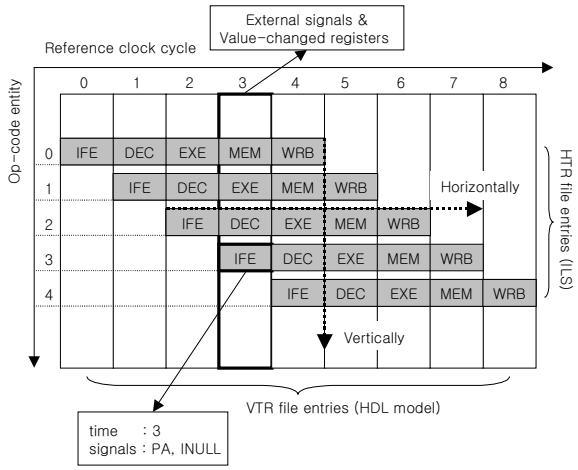

Fig. 2 Comparison between the 2 models

VTR entries. The figure 2 illustrates how to compare simulation results between them. The 3rd opcode occupies IFE at 3rd clock in the HTR file. The comparison utility then refers to the 3rd entry in the VTR file and compare values of the victim signals between them. If any difference is found between them, it is evident that some errors should lurk in one of them. The verification process is iterated with debugging until there remains no difference between the 2 files for a single random test vector workload.

There are three considerations for random vector simulation. First, any unexpected branch and load/store should be prevented from reading or writing undefined memory space or otherwise the random vector simulation may fall into irrecoverable erroneous condition. To circumvent this, the memory model refers to its additional internal counters rather than addresses issued from the core during bus cycles. Each counter increment after its related data transfer happens. Second, the memory space is divided into 3 sections in the environment – CODE for opcodes, DATA for general memory data and KODE for data to be transferred by LDC instructions.

Though the instruction level simulator exactly reflects all the timing information of the target HDL, it internally runs linearly; that is, no other opcode can occupy any pipeline stage until the current one passes through the WRB stage. However, pipeline stages are actually overlapped with different opcodes and due to such discrepancy in execution order, it would be a very complex job to maintain consistency in the counters for CODE section between the target and the reference if memory load by an LDC instruction in MEM shared the same memory space with instruction fetch in IFE. The proposed environment solves this problem by simply allocating KODE apart from CODE. The internal counters individually correspond to their own memory sections. Simulation is finished off after the CODE counting number reaches its upper bound and in contrast, DATA or KODE counting number is turned back around when it does so.

Third, any random vector workload should contain a system-initializing routine at its header before sequence of random test vectors. Such scheme prevents external signals and registers from falling into unknown state due to uninitialized condition during normal operation of simulation

IV. THE CYCLE-ACCURATE INSTRUCTION LEVEL SIMULATOR

Overview

ILS, the proposed cycle-accurate instruction level simulator for CalmRISC^{TM}-32 was programmed by microprogramming scheme. Hardwired scheme is generally accepted as more desirable design method than microprogramming scheme for RISC processors. However, concerning an instruction level simulator, hardware overhead matters no longer and instead, modularity becomes the most critical factor because regular data structures generally produce more concise and manifest statements in programming languages. Especially, the CISC styled instructions of CalmRISC^{TM}-32 can be more efficiently modeled by microprogramming scheme though

they are actually implemented by hardwired scheme. The figure 3 shows a block diagram of ILS. The shaded area in this figure includes main procedure of modeling the pipeline flow. The DEC module parses a current opcode into an index of its first microopcode by lookup to decode hash table. The structure of microopcode lookup table is as shown in the figure 4. Every microopcode entry has concat pointing to the next microopcode. ILS then executes each microopcode belonging to the current opcode until its concat holds EOM (End Of Microprogram), and then initiates to fetch the next opcode. Every single-cycle opcode is exactly correspondent to a single microopcode while every multi-cycle opcode consists of multiple microopcodes.

A microopcode has three pointers – FP, AP and MP. These pointers point to entries of the flow control lookup table (briefly FLW), the ALU lookup table (briefly ALU) and the memory load/store table (briefly MEM) respectively. These lookup tables also instruct ILS on branch operations in DEC, ALU operations including address calculation in EXE and memory load/store operation in MEM respectively.

As previously mentioned, ILS does not follow actual order of operations observed in the HDL model. Instead, the

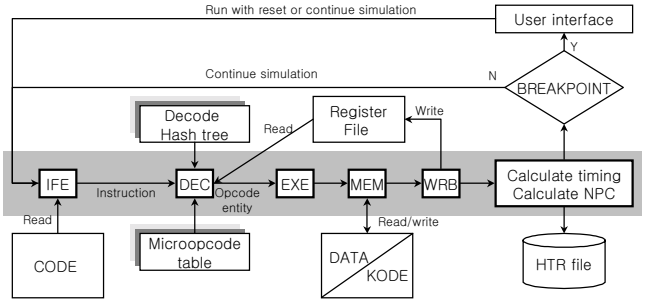

Fig. 3 Block diagram of the instruction level simulator

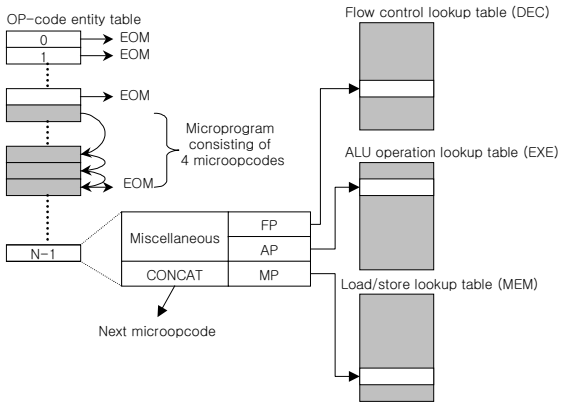

Fig. 4 The structure of microopcode lookup table

calccyc module calculates pipeline timing once after every microopcode has been completed.

The breakpoint module is necessary to allow users' intervention during simulation. If user-manipulation such as breakpoint setting or internal hardware exceptions such as data abort occurs, the simulation pauses according to the ILS configuration and then users can browse current internal states such as registers, memory, etc., or resume the simulation.

Instruction decoder model

Instruction decode procedure is subdivided into 2 steps; In the first, the DEC module determines an opcode type and then in the second, it interpret operands. The opcode type is actually an index of the first microopcode in its correspondent microprogram. In mapping an opcode to its first microopcode, tradeoff between memory and trial number should be considered. Thus, we propose new efficient decoding scheme using hash tree to optimize these contradictory factors. The upper bound of its trial number is only 4 while its number of entries amount to 332, which is considerably economic.

The structure of decode hash tree is shown in the figure 5. At first, an opcode is divided into 4 nibbles and the 1st nibble is used as an index of the entry in the root node; the 1st nibble of a CalmRISC™-32 instruction is actually assigned as a prefix for classifying instruction group. A hash tree entry has id, p and shft. For each trial, the DEC module calculating an index to the next entry by left shifting the idth nibble of an opcode by shft if p is not NULL while p points to the next node. The left-shift operation helps reduce total size of hash tree entries because operand fields of every CalmRISC™-32 instruction are left aligned in nibble boundary and they have no effect on determining opcode types. For example, if the bit pattern is 01XX, the sufficient number of entries is 4, instead of 16 because only the higher 2 bits are meaningful. If p is NULL, id holds an index to the first microopcode and additionally shft with nonzero value indicates the current opcode is unimplemented.

After then, the DEC module refers to entries of the lookup tables - FLW, ALU and MEM pointed to by the microcode and translates their operand fields to appropriate register indexes while for an immediate operand, constant extracted from an opcode is saved into IMR, the virtual register for immediate value and the operand field is marked as IMR.

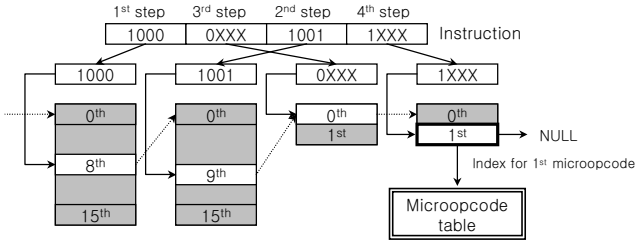

Fig. 5 The structure of decode hash tree

Flow control model

ILS manipulates flow control with a pipeline flow window. This window is a sub-array of a circular queue where each slot corresponds to pipeline stage modules. After every microopcode is completed, the marknpc module determines several NPC's. Every NPC is stored into the next to the window slot where its cause was brought about and the slot is marked with one of the states – NORMAL caused by normal PC increment, FLUSH by non-delayed branches or exceptions and DSLOT by delayed branches. After then, the

window is advanced by one slot in the queue and the IFE module gets a fetch address from its newly correspondent slot. Every pipeline stage module except IFE evaluates its corresponding slot state before it begins its operations including exception check. If the state is FLUSH, ILS cancels all the scheduled operations for a current microopcode from that stage on. Such situation continues until the IFE module corresponds to that slot in FLUSH since the IFE module restores its slot state to NORMAL whenever it gets a fetch address from it. The figure 6 illustrates this procedure by an example of non-delayed branch operation.

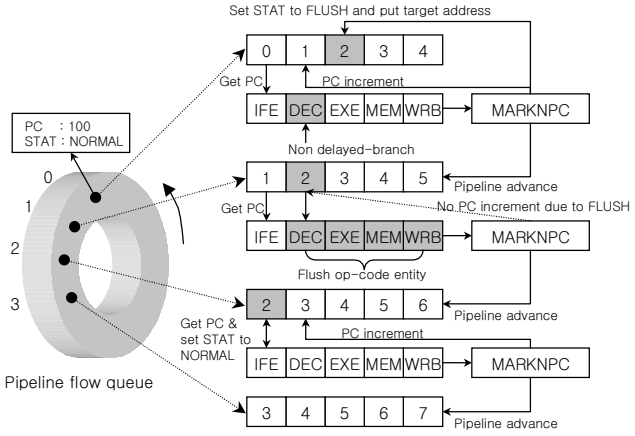

Fig. 6 Flow control window

Pipeline timing calculation

Simulation time in ILS has no meaning of physical dimension and just represents an ordinal number of clock tick. Current time of each pipeline stage is determined in the calccyc module after a current microopcode is completed. Timing calculation is simple for single cycle instructions without any dependency because it is accomplished only by increasing current time of each pipeline stage by 1. On the contrary, memory lock and multi cycle instructions require more elaborate scheme. Every opcode contains variables for memory dependency check. For every microopcode to be executed the DEC module compares information in these variables with history of the previous microopcode and then if it detects memory lock, current occupying time of DEC is extended to the finish time of MEM for the previous microopcode.

In case of multi cycle instructions, ILS adopts concept of virtual fetch. For example, PUSHQn opcode consists of 4 microopcodes in ILS but only the first needs instruction fetch and the others do not though they actually occupy IFE in simulation. Consequently, most recent simulation times for IFE and DEC should be restored to the ones for the first microopcode and times for the remained stages are readjusted as in the figure 7. The readjustment differs in the intra-pipeline stage opcodes and inter-pipeline stage opcodes; the one continuously occupies the same stage for more than 1 clock cycle and the other occupies the same stage more than once. The arrowed lines in the figure indicate relation between cause and result for the readjustment.

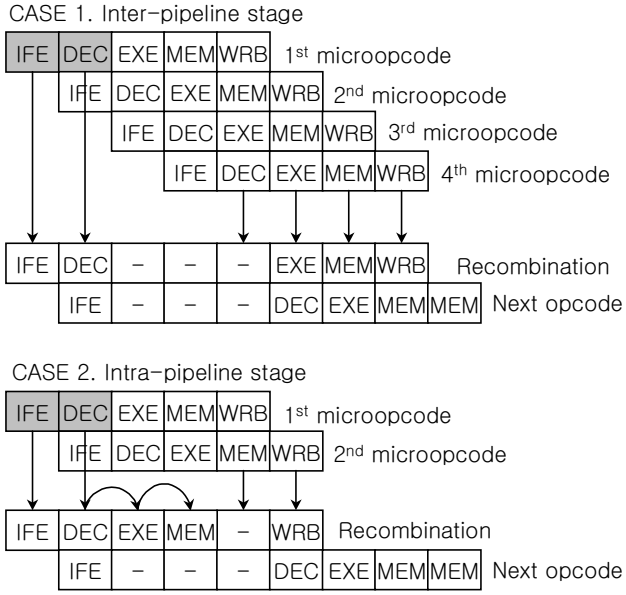

Fig. 7 Timing calcution for multi-cycle instructions

V. CONCLUSION

In experience of the random vector verification, the found errors are mostly very hard to detect by conventional verification methods. The most common case is failure of detecting or handling exceptions and the next is wrong change of status register value. Such cases are generally blind spots in processor verification because such erroneous behavior looks trivial but potentially brings undesirable effect on processor reliability. Therefore, the proposed random vector verification methodology is powerful solution to promote reliability of processors if it cooperates with conventional verification methods.

Though ILS, core part of the proposed environment is heavily dependent on CalmRISC^{TM}-32 architecture, its structure is considerably modular since it adopts microprogramming scheme unlike the actual implemented machine. Therefore, it is relatively easy to modify its design for application to other embedded RISC processors without great loss of generality.

REFERENCE

[1] David A. Patterson, John L. Hennessy, “Computer Architecture: A Quantitative Approach Second Edition”, Morgan Kaufmann Publishers, Inc, 1996.

[2] B.Y.Choi, K.Y.Lee, S.H.Lee, and M.K.Lee, “VLSI design of a pipeline Controller for a 32-bit Application-Specific RISC”, in KITE Journal of Electronic Engineering, vol. 5, no. 2 Dec 1994

[3] Ron Cates, “Processor Architecture Considerations for Embedded Controller Applications”, in IEEE MICRO, 1988

[4] Moon Gyung Kim, Byung In Moon, Sang Jun An, Dong Ryul Ryu, and Yong Surk Lee, “Implementation of a cycle based simulator for the design of a processor core”, the 1st IEEE Asia Pacific Conference on ASICS, Seoul, Korea, Aug. 1999