# Design Experience of a Chip Multiprocessor Merlot and Expectation to Functional Verification

Satoshi Matsushita
NEC Corporation
1120 Shimokuzawa, Sagamihara,
Kanagawa, 229-1198, JAPAN
s-matsushita@bx.jp.nec.com

## ABSTRACT

We have fabricated a Chip Multiprocessor prototype code-named Merlot to proof our novel speculative multithreading architecture. On Merlot, multiple threads provide wider issue window beyond ordinal instruction level parallel (ILP) processors like superscalar or VLIW. With the architecture, we estimate 3.0 times speedup against single processing elements (PE) on speech recognition code and IDCT code with four PEs. Merlot integrates on-chip devices, PCI interface, and SDRAM interfaces. We have encountered design issues of chip multiprocessor and SoC design. We have successfully run parallelized mpeg3 decoder on the first silicon with several software workarounds, thanks to functional verification environment including system modeling on RTL. However, bugs found in later stage of design have required larger manpower or delay of project. In this paper, we also discuss the methodology to improve functional verification coverage, and expect the solution in formal approaches.

## Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids – *Simulation, Verification*
B.7.1 [Integrated Circuits]: Types and Design Styles – *Microprocessors and microcomputers*
B.m [Logic Design]:Miscellaneous – *Design management*
C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors) – *Multiple-instruction-stream, multiple-data-stream processors (MIMD)*;
C.1.4 [Processor Architectures]: Parallel Architectures – *Mobile processors*

**General Terms**: Design, Verification.

**Keywords**: Speculative multithreading, Chip Multiprocessor, CMP, Deign Experience, Functional verification


*ISSS'02,* October 2–4, 2002, Kyoto, Japan.


**Table 1: Specifications of Merlot**

| | |
|---|---|
| Technology: | 0.15 mm CMOS, 5-Metal |
| Supply Voltage: | 1.2 - 1.8 V (Internal), 3.3 V (I/O) |
| Clock Speed: | 125 MHz (at 1.3V) |
| Number of Trs.: | 14M (Logic 6M, Memory 8M) |
| Area: | 10.5 mm x 10.5 mm (110 mm2) |
| Number of Pins: | 300 (Signal), 500 (Total) |
| Power: | 1 W (at 1.3 V) |
| Performance: | 1 GIPS (at 1.3 V) |
| I-Cache: | 64 kB (4 Set Associative, 32 B Line) |
| D-Cache: | 64 kB (Divided into 8 Banks, 32 B Line) |
| Base Instruction Set: V800 series(NEC Proprietary), 16,32bit width. | |
| Instruction Issue: | 2-issue (in-order) x 4 Multi-processor |
| | Non-blocking load: 3 load/ifetch miss per PE |
| Data: | 32 bit, 16 bit x 2 Media, (8 bit x 4 Media) |
| Bus Interface: | SDRAM: 64 bit + ECC (1 or 2 Channel(s)) |
| | PCI 2.1: 32 bit, 33 MHz |
| | Up to 8 outstanding requests |

## 1. INTRODUCTION

The rapid progress of LSI technology has realized system on chip (SoC) with embedded processors, devices, and memories. SoC enables us small, low cost, and low power consumer equipments. LSI designers, however, now encounter design and verification problems because of the complexity with the system level integration and shorter design time requirement.

We have been researching a chip-multiprocessor architecture targeting at higher performance with lower power consumption. We have proposed FOPE (Fork Once Parallel Execution) architecture, and fabricated a prototype chip code-named Merlot. In section 2, the target of Merlot and its architecture is briefly described. In section 3, design issues of Merlot and our solutions are referred. Finally, in section 4, we focus on the coverage improvement of functional verification since we think it is crucial for complex system design.

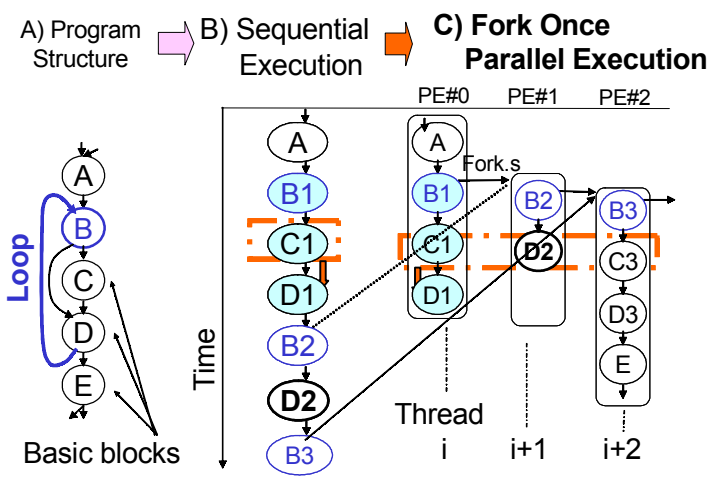


**Figure 1: Fork Once Parallel Execution**

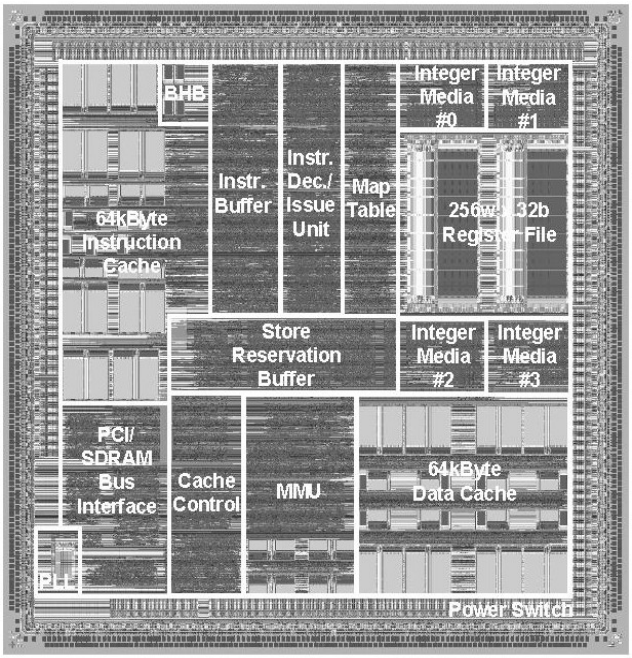


**Figure 2 Merlot Die Plot**

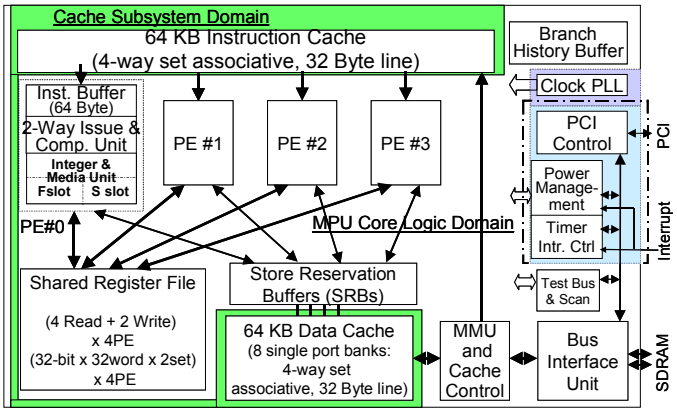


**Figure 3 Block Diagram of Merlot**

## 2. MERLOT ARCHITECTURE

SMP (Symmetrical Multiprocessor) has been widely used in high-end computer systems. In SMP systems, processors are connected to a shared memory through a coherent cache system. Thread libraries for application programming on a monolithic SMP operating system relieve burden of parallel programming. Even with them, however, multithread programming is not easy due to racing condition in shared data caused by erroneous synchronization, etc. Performance tuning is another burden. Speculative multithreading has been proposed to remove this burden by automatic parallelization with compiler or object code conversion [1][2][3].

Merlot[4][5][6] is a tightly coupled chip-multiprocessor (CMP) as a research prototype to proof our novel speculative multi-threading architecture named FOPE (Fork Once Parallel Execution). With FOPE architecture, a compiler realizes automatic parallelization with the assistance of directives inserted into compiler source code.

Parallelization scheme is depicted in Figure 1. Figure 1-A shows a sample program structure. Program 1-A's dynamic execution flow in sequential processor is shown in Figure 1-B. In FOPE, instructions executed in future of thread i (basic block B2, D2) are hoisted and executed in neighbor processor in parallel in thread i+1. The thread generation is controlled by fork instructions inserted by parallelizing compiler.

Architectural keys are:

1. Static thread scheduling (ordered threads) controlled by several additional instructions (fork, term, thcomt, thabort, etc).
2. Speculative execution of threads beyond control flow determination and data disambiguation.

A statically defined thread order eliminates thread-scheduling hardware. With the constraint that a thread generates another thread only once throughout the thread's life, unique thread order is defined among all threads. Under this constraint, data communication and synchronization are unidirectional, and deadlock freeness is guaranteed with static scheduling. In addition, this constraint enables sequential emulation of parallelized code in a single processor by just neglecting thread control instructions. Therefore, we can verify parallelized code with sequential emulation.

Speculative execution enables us to exploit larger parallelism, though we have to pay the cost for tentative storages of register and memory value. The storages also resolve hazards (Read-After-Write, Write-After-Read, Write-After-Write) which are derived from parallel execution of threads.

Chip specifications are shown in Table 1. Die plot and block diagram of Merlot are shown in Figure 2 and Figure 3 respectively.

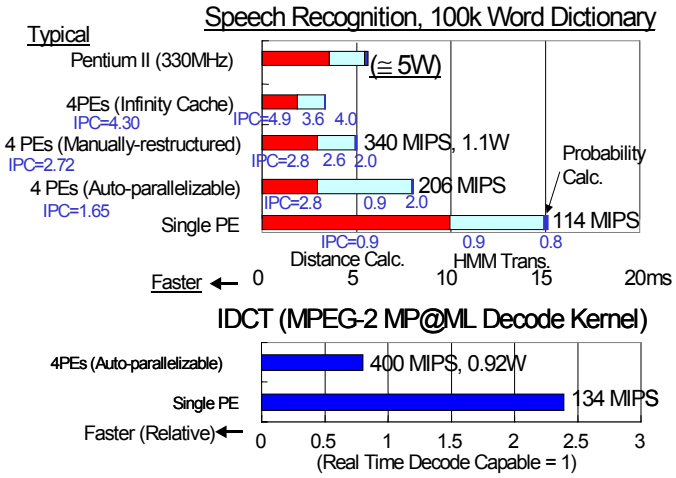


**Figure 4 Performance Estimation of Merlot**

On Merlot, each processing element shares an instruction fetch unit, a register file, and a data cache with store reservation buffers (SRBs) for faster thread creation and data communication. We have designed the whole logic from the scratch including the processing element because of those sharing resources.

Performance estimation is shown in Figure 4.

## 3. DESIGN AND VERIFICATION OF MERLOT

Design difficulty of Merlot is shown in Table 2. With much effort spent on functional verification, we could successfully run mpeg2 decoder with simple OS on the first silicon with several software workaround and a bug fix with FIB. However, we still think the verification is the biggest issue in complex microprocessor design. Before discussing this issue in section 4, we would like to summarize the design and verification of Merlot focusing mainly on verification.

### 3.1 Criteria for Design Environment

For the design environment, we considered the following aspects: 1) Repeatability or Automation in order to remove human errors, 2) Controllability and Convergence to design target, 3) Expandability of the tool, and 4) Information for management aids.

### 3.2 Design Environment

For logic design and verification of Merlot, we have provided following environments. Detailed descriptions are in [7]:

1. Online Documentations: used Web for ease of update and accessibility.
2. Strict design rules: Signal and module naming rules predefined for a globally unique and standardized identifier.
3. Design Entry Tools: verilog enhancement with perl based macro expressions, and terse description of design parts like FF, MUX, etc. The size of source is reduced to 40% of final verilog RTL (Register Transfer Level) description in words in 168 source files. Non-leaf level connecting RTL generator reduced the source to 10% in words in 53 files. The tool also works for the front-end for RTL timing tools.
4. System Level Modeling and reference for simulation.
5. Planned Test Pattern Generation: aimed at balanced mixture of random and directed patterns.
6. Regression Test Tools: automated nightly regression test and visualization of the results.
7. RTL version management with modified CVS.
8. Bug Tracking based on GNATS.
9. RTL Timing Tools with budget visualization and automatic generation of synthesis constraint
10. ECO Design Tools: Formal Equivalence Checking between modified RTL and manually modified Gate Level Netlist.

**Table 2: Design Difficulty of Merlot**

**1. Chip Multi-Processor (2issue PE x 4PE on a Chip)**
- Difficulty to provide reference simulator
- Difficulty to improve test coverage

**2. System on Chip**
- Multiple clock domains: core, SDRAM, PCI (async.)
- System modeling: (PCI, SDRAM)

**3. Debugging Support**
- Debug modes, Test bus for memory and register file observation.
- Partial scan, Spare cells for FIB

**4. Low Power Design**
- Clock gating
- Three power domains with on-chip power switches for leak current reduction
- Custom Blocks: Cache, PLL, Power Switch, Register File: functional verification issue for custom blocks arose.

**5. Physical Design Issue**
- Signal Integrity issues: Closstalk, Electro migration, Maximum current, Antenna-Effects.
- Optical Shrink: Compensation of timing and layout

### 3.2 System Level Modeling

We combined Merlot RTL to a commercial PCI modeler and a memory modeler. The PCI modeler originally required handwritten test scenarios for master operation. We modified the modeler to be triggered by program counter in RTL, so that PCI stimulus can be specified in assembly code of the tests. To match the RTL and ILsim (Instruction Level simulator), we extracted the positions of events, which are required to be clock accurate, from RTL simulation first, and inserted them into ILsim. However, it was difficult to remove false mismatches because of simplified modeling in ILsim (no cache, sequential emulation of thread execution, etc) or visibility of cycles in system registers. To run applications on RTL or ILsim, we provided OS stubs for the delegation of system calls to the host OS.

**Table 3: Components of Nightly Regression Test**

| | |
|---|---|
| Directed Test (Including 2 Benchmark, 4 application, 4 OS kernel) | 310 sets (1M cycles total) Runs only after revised RTL released to CVS . |
| Random Test (Everyday Generated with new seed) | 65 sets (500K cycles total) Runs every Night. |
| System level Test activating PCI bus, SDRAM models | 53 sets (150K cycles total) Runs only after new RTL released. |

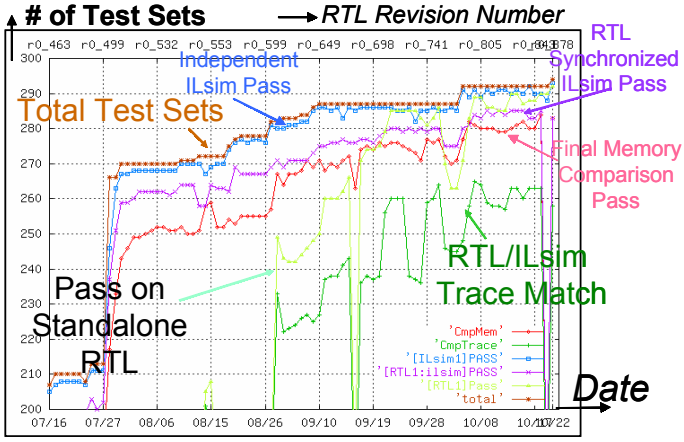


**Figure 5: History of Passing Sets in Nightly Regression**

For early debugging of a bus interface unit (BIU) including SDRAM and PCI interfaces, we isolated BIU and tested. Command and status code are defined for the communication between BIU and core, so we manually provided the command sequences, and verified the log files.

## 3.3 Test Pattern Generation

We had a review of test pattern design. According to the review, we provided a mixture of directed (manually written) tests and random tests in assemble instruction sequence, as seen in [8]. For random testing, 11 generation algorithms have been finally implemented. We provided parallel OS prototypes for regression tests, where we verified OS boot through character I/O operation by application programs. Real applications with reduced data set are also executed on RTL. Four handled test sets are executed as regression tests every night on Mentor Modelsim on 6 sun workstations (Table 3).

## 3.4 Design Managements

Processor design is achieved as collaboration in a big team. We think success lies in 1) timely and proper recognition of problems, and 2) the information sharing in the team. In another word, when these two things are achieved, problems are almost resolved. We have found visualization is quite useful to accomplish the recognition and sharing, and provided visualization tools.

Followings are some examples: Figure 5 is a graph automatically updated by the result of nightly regression tests. Figure 6 shows the timing slack in each synthesis block. RTL

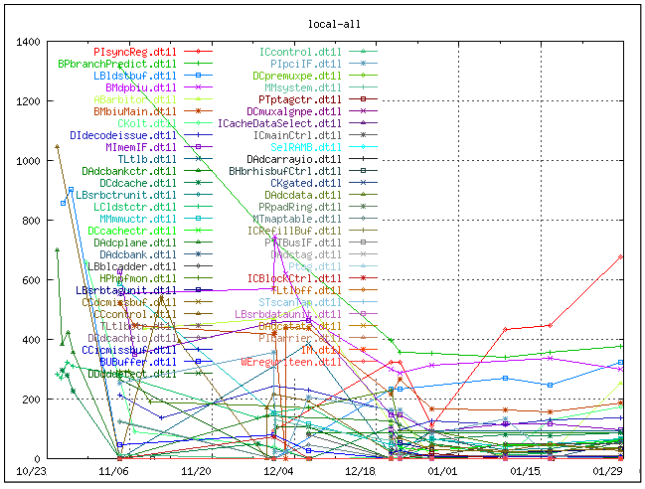


**Figure 6: Timing Improvement History of Merlot, Generated from RTL Timing Tools.**

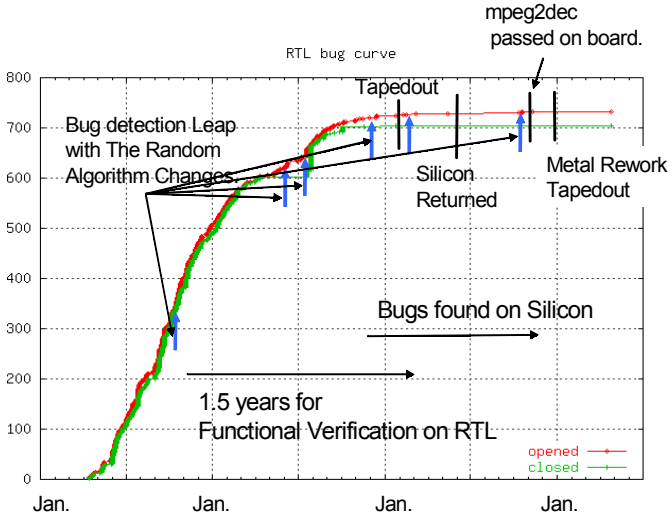


**Figure 7: Bug Curve**

timing tools automatically generate synthesis constraints considering the estimated delay of global wire and optical shrink effect, then analyze synthesis reports, and draw the slack graphs in several aspects; block local or accumulated ones. Figure 7 is a bug curve automatically generated from GNATS (GNU bug management tool) database.

# 4. EXPECTATION TO VERIFICATION

As seen in the bug curve (Figure 7), it took long time for functional verification, even though we ran random tests every night. Furthermore, bug detection leaps are observed in the bug curve. The leap occurred after we introduced new random generation algorithms. This shows that bug detection ability of random tests rapidly saturates because we limits the randomness for both generating meaningful instruction sequence and aiming at a specific corner case defined in the test plan. If we can improve bug detection, chip design time would be dramatically improved.

With this motivation, we would like to discuss the coverage improvement of functional verification in this section.

## 4.1 Microprocessor Design Flow

Before discussing functional verification, we would like to review the design phases of microprocessor. This is helpful to understand reference model and target model of design:

1. Instruction set Level (architectural or behavior) design.
2. Pipeline Level (micro-architectural) design.
3. Register Transfer Level (RTL) design
4. Gate Level design

Pipeline design adds the perception of clock or pipeline. In RTL design, logic implementation is considered, and behavior is modified for simplification of control logic or data paths. Gate level design is manually generated or synthesized from RTL, where actual gate mapping and fan-out trimming are performed. Logical behavior of gate level design slightly differs from that of RTL in the implementation of redundant states and initial states.

## 4.2 Functional Verification Methodologies

Considering design flow, functional verification methodologies are categorized as follows:

1. Full Chip Verification
   - i. Running Self-checking instructions (Vector Based).
   - ii. Comparison between Instruction Simulator and RTL simulator. (Vector Based)
   - iii. Dynamic Verification [9] [10] (Vector Based)
2. Sub-block Verification
   - i. Property Checking [11][12] (Formal)
   - ii. Test Bed Generation Libraries [13] (Vector Based)
   - iii. Formal Equivalence-checking (Formal)

Since the definition of instruction is fundamental and instruction boundary is well preserved through all the design phases, method (1-i, 1-ii) is commonly used. In 1-ii discrepancies are observed for some instructions, where clock cycle is observable such as clock/performance counter, interrupt, etc. The discrepancy makes comparison in actual system configuration, which sometimes includes OS and asynchronous I/O events, very difficult. The discrepancy in Merlot design is observable in Figure 5. In dynamic verification (1-iii), simple RTL models of each pipeline stage are used for reference. Since input of reference model can be acquired from target, reference model can be simple and fast enough. In the approach, when the discrepancy is observed, reference machine rewinds the state of target machine, and corrects the target machine. As the result, we can run a test vector beyond detected bugs for detection of other bugs. Property checking is effective in both coverage and simulation time. In (2-ii), pattern generation libraries or device modeling is useful for quick generation of test bed, however, it is not easy to provide reference vector. For the reference, preserved logs after checking the behavior by hand is recommended as golden vector[13]. Formal equivalent checker (2-iii) is useful to verify gate level design against RTL when ECO (engineering change order) is manually applied to gate level design.

If we have a tool to generate good vectors, functional verification problems are improved even with vector based solutions; (1-i) to (1-iii). Definitions of good vector are:

A) Coverage of functional bugs

B) Repeatability of a specific bug: timing condition of the specific bug is satisfied after modification of irrelevant logic.

C) Concentration on detection of a specific bug: a vector focuses on a single bug in a short simulation time. This feature makes bug analysis easier.

The feature B) and C) are useful for design quality management.

## 4.3 Current Vector Generation Methods

Considering easy preparation of reference vector in MPU verification, we focus on the method using assembly instruction sequence as test vector.

**Table 4: Vector Generation Methods**

| Name | Generation | Pros | Cons |
|---|---|---|---|
| Directed, Selfchecking | Manual or customized-Scripts | B) C) | A) |
| Random | Tools | A) | B) C) |
| Property Checker | Tools | A) | B)? C) |
| Real Program | Run on real condition. | A)? | Quite low B), C) |

Table 4 summarizes current vector generation methods and their pros and cons of referring item number of good vector in the previous section. For the design where multiple autonomous entities interact, unexpected racing conditions are created with multiple asynchronous events. So it is quite difficult to write test vectors with good coverage. Random vector generation is commonly used for detecting such cases. However, it is still difficult to achieve good coverage. Furthermore, an error detected by random vector is hard to debug due to (C), and hard to manage due to (B). In the random test generation of MPU, constraints or heuristics have been considered in the scripts for random pattern generation to improve effectiveness of bug detection (C) or adding selfchecking feature. Though the method was mostly ad-hoc, recently a verification suites, Specman[14], has been applied for some SoC designs. Specman reduces the verification cost with a test bench description language and verification IPs including constraint driven random generation, functional coverage analysis, and protocol checking. For MPU testing, integrated approach is also seen in a commercial tool Raven[15].

With the emulator on FPGAs, simulation speed has become fast enough to run real programs with I/O devices and real OS. The approach still encounters the coverage problem if we cannot get realistic test sequences. In addition, visibility of logic status for the error is a key for debugging. In newer approaches, we finalize design at behavior level (C/C++ level) and automatically convert the design to real chip with high-level synthesis tool and IP libraries[16]. At behavioral simulation, we can expect much better performance for system level simulation by software.

## 4.4 Architecture Level Design Checking

In multiprocessor design, racing condition of mutual interactions of processors sometimes create bugs. It is a good idea to remove such a racing condition at instruction (architecture) level definition. We think it is useful if we can apply formal method at this level.

In IP based design, specification is given at architecture level, and the architecture level racing check is useful for such an IP based SoC design.

## 4.5 RTL Level Coverage Improvement

It is not difficult to write property for basic logic components such as sequencer, FIFO, arbiter, etc. Property checking is useful to test these basic components. However, even when we assemble verified components to a system, the system will not be bug free, and such bugs are difficult to detect in simulation.

We expect a tool that analyzes both specification of instruction and RTL description, and generates instruction sequences that may create effective condition for bug detection. It is a big challenge because a lot of logic interacts for executing single instruction isolated from other instructions. In real multiprocessor system, a lot of events interact in the sequence of instructions on multiple processors.

# 5. CONCLUSIONS

In this paper, we presented our design experience of a speculative multithreading processor code-named Merlot. We have recognized the difficulty of functional verification considering asynchronous interaction of processing elements and embedded devices, so we tried to establish strategic verification environment. However, we still spent too much time and manpower on functional verification since our verification approach was still brute force. In this paper, we tried to examine the functional verification of multiprocessor system, and cast expectations to theoretical approach in the test vector generation.

# 6. REFERENCES


[1] Sohi,G., Breach,S. and Vijaykumar,T.N.: Multiscalar Processor, Proc.22nd ISCA, 1995, 414-425

[2] Tsai,J., Huang,J., and Amlo,C.: The Superthreaded Processor Architecture, IEEE Trans. Comput., Vol.48, No.9,1999,881-902

[3] Hammond,L., Hubbert,B., Siu,M., Prabhu,M., Chen,M., and Olukotun,K.: The Stanford Hydra CMP, IEEE MICRO, Vol.20, No.2, 2000, 71-84

[4] Nishi, N., et al: A 1 GIPS 1W Single-Chip Tightly Coupled Four-Way Multiprocessor with Architectural Support for Multiple Control Flow Execution, ISSCC, , 2000, 418-419 (WP25.5)

[5] Matsushita, S., et al.: Merlot: A Single Chip Tightly Coupled Four-Way Multi-Thread Processor, Cool Chips III, 2000

[6] Latest information of mp98:

www.labs.nec.co.jp/MP98/

[7] Matsushita,S.; The Logic Design Environment of Chip Multiprocessr Merlot, (In Japanese), Journal of IPSJ, Vol.42, No.4, 2001, 922-929

[8] Hosseini,A.,Mavroidis,D.,and Konas,P.: Code Generation and Analysis for the Functional Verification of Microprocessors, 33rd DAC, ACM, 1996, 305-310(23.1)

[9] Austin,T.M: DIVA: A Dynamic Approach to Microprocessor Verification, Journal of Instruction-Level Parallelism 2, 2000

[10] Mneimneh, M. et.al.: Scalable Hybrid Verification of Complex Microprocessors, DAC, 2001

[11] IBM Sugar: www.haifa.il.ibm.com/projects/verification/sugar/standard.html

[12] Synopsys Open Vera2.0: www.open-vera.com/

[13] Boyd,S.: Using Vera to Test a DMA Engine: www.open-vera.com/technical/boyd.pdf

[14] Verisity Design Inc., Specman Elite: www.verisity.com/products/specman.html

[15] OBSIDIAN RAVEN: www.obsidiansoft.com/

[16] Wakabayashi,K. and Okamoto,T: C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective, IEEE Trans. CAD of Integrated Circuit and Systems, Vol.19, No. 12, 2000