SPARC64 V Processor For UNIX Server

Revision 1.0

August 2004

FUJITSU LIMITED

Contents

Preface

Enterprise server systems take high responsibility as social economic infrastructures. The processor, the heart of the systems must achieve high-reliability as well as high-performance. We have developed SPARC64™ V, the 5th generation of the SPARC64 processors, based on technical background behind our long-range of mainframe development, and targeted for simultaneous achievement of high performance and high reliability never realized.
The SPARC64 V processor has been designed based on Fujitsu's Mainframe (GS8900) processor. At the beginning of the design, we had to examine how to absorb the architectural difference between SPARC processor and the mainframe processor.
Needless to say, the instruction decoder must have been newly designed due to the total difference of instruction set architectures. Furthermore, programmable resources such as structure of architecture registers and memory model and so on, are very different, so that many features that the mainframe architecture does not possess have been newly designed.
Due to difference between RISC and CISC, workload characteristics, OS code effects, and so on, we had to newly design execution units, which are dominant factor of performance.
Also we have developed SPARC64 V performance model to verify design correctness from the performance point of view.

CHAPTER 1 Development Target

We have developed SPARC64 V for PRIMEPOWER, Fujitsu UNIX® server. Therefore, the target is to maximize the competitive edge of the PRIMEPOWER as an enterprise server system. We have developed SPARC64 V with the following policies.
1) SPARC V9 compliant design:
SPARC64 V is compliant with SPARC V9 architecture to keep the binary compatibility. To pursue further compatibility on the level of operating systems, SPARC64 V is also compatible with JPS (Joint Programming Specification), which is the common specification jointly developed by Sun Microsystems and Fujitsu.
2) High Performance design on the system:
SPARC64 V takes full advantages of the potential of PRIMEPOWER, and meets with the condition for system implementation such as power consumption, chip size, system bus specifications, and so on.
SPARC64 V has achieved high throughput, focused on the practical workload especially on the interactive transaction.
The cache hierarchy of SPARC64 V is simplified to keep multi-processor scalability.
SPARC64 V has multiple execution units, and balanced execution resources, to keep competitive edge for HPC applications.
3) High frequency design:
It is important to pursue the optimum solution for balance between processor frequency and performance per cycle. Applying sophisticated processor design methodology built through mainframe development, SPARC64 V has achieved high processor frequency as a processor with advanced features.
4) State-of-the-art RAS (Reliability, Availability, and Serviceability):
To meet mission-critical applications, we have implemented state-of-the-art RAS function on

SPARC64 V. It provides thorough guarantees of data-integrity, and executes error recovery processing by data correction or instruction retry, so that the system operation and applications keep running. In specific, 1-bit error on RAM is always corrected, because possibility of such an error is relatively higher than other logical circuit units. SPARC64 V RAS function is targeted that no error effects are observed except performance overhead due to the error recovery. The error information is collected by exclusive hardware, which realizes immediate and correct trouble-shooting.

CHAPTER 2 SPARC64 V Micro-architecture

SPARC64 V is a super-scalar processor, which executes multiple instructions out-of-order.
SPARC64 V consists of Core Unit, Level-2 cache (L2$) Unit, and System Interface Unit. The Core Unit is divided into Instruction Fetch Unit and Instruction Execution Unit. Instruction Fetch Unit includes Level-1 Instruction cache (L1I$), and Instruction Execution Unit includes Level-1 Operand cache (L1D$).
First of all, Instruction Fetch Unit and Instruction Execution Unit are described in this order. After that, Level-1 and Level-2 Cache system, System Interface Unit, and other features are described.

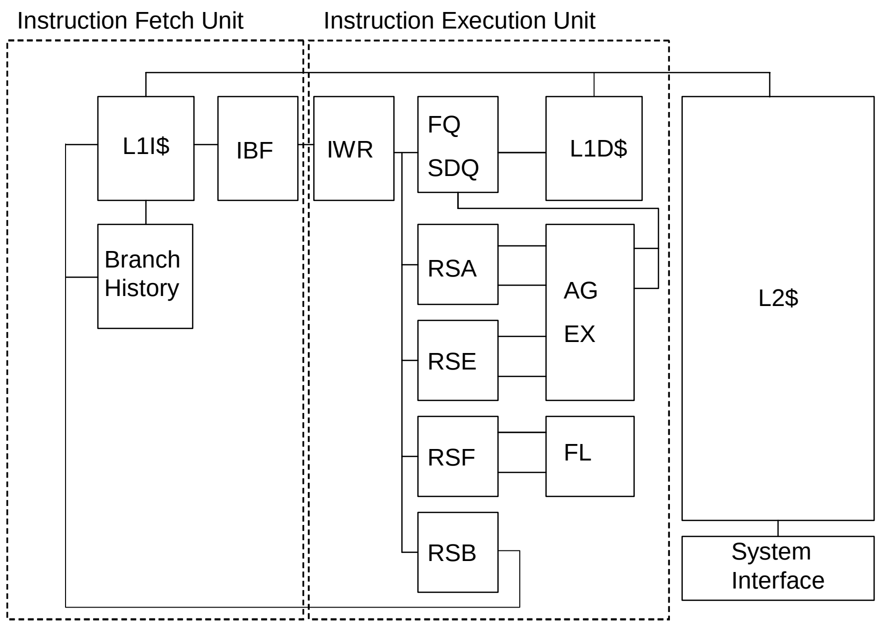

Figure 1 SPARC64 V Block Diagram

2.1 Instruction Fetch Unit

The instruction fetch unit is decoupled from the execution units. It fetches instructions from
L1I$ to IBF (Instruction BuFfer) based on predictions of Branch history. IBF has capacity of 192 bytes, and instruction fetch continues until IBF becomes full, even when instruction execution has stopped. Even if instruction fetch has stopped because of cache-miss, instructions are provided from IBF to execution unit unless IBF is empty. 32bytes data, that is 8 instructions, are fetched every cycle from L1I$ to IBF. The throughput of instruction issue to execution units is maximum 4 per cycle. This means the throughput of instruction fetch is double, which brings a good effect of

de-coupling. As a result, although L1I$ capacity is big, disadvantages caused by long pipeline access are hidden and advantages of large capacity are taken.

An accuracy of branch prediction is one of the important keys determine processor performance. SPARC64 V is equipped with large branch history (16K entries, 4way-set-associative) for mission-critical computing, such as large-scale transaction processing, etc. Each entry of the branch history holds information of a state of branch -- Strongly-Taken, Weakly-Taken, and Invalid -- with the branch target address. If a valid entry is found while referring to the branch history on instruction fetch, the prediction will be "Taken".

To improve the accuracy of branch prediction for complex routines used in usual programs frequently, SPARC64 V has WGHT (Write-cycle-driven Global History Table) which is a kind of global history table. WGHT works in closer cooperation with the branch history. When a branch instruction completes its execution, WGHT is referred and the state in the branch history is changed accordingly. In case there is no WGHT entry for the completed instruction or it doesn't have enough history in the entry to predict, the branch history decides next prediction with the result of the branch instruction and updates itself without referring to WGHT. Wrong prediction due to lack of enough branch history does not occur, thanks to this mechanism.
Also SPARC64 V has another branch prediction mechanism, called "return address stack". It is used to predict the address of returning instruction from subroutine. The return address stack consists of 4 entries based on LIFO (Last-In Fast-Out) structure and 1 special entry generated by speculative fetch. Update of the return address stack is done at the same time with updating the branch history. The return address stack pointer compensates the time difference between instruction fetch and corresponding instruction completion, and brings high accuracy prediction to SPARC64 V.
The instruction fetch unit works well also for local short loops, which occupy large amount of the processing time of general applications. When a short loop is found in a program, the instruction fetch unit generates a chain consists of IBF entries corresponding to the loop. Instructions in the loop are provided from this chain rather than from L1I$.

As explained above, the instruction fetch unit has the large cache and branch history. It is able to provide enough instructions to the execution units with the very high throughput for any programs from general applications to large-scale processing of mission-critical transaction.

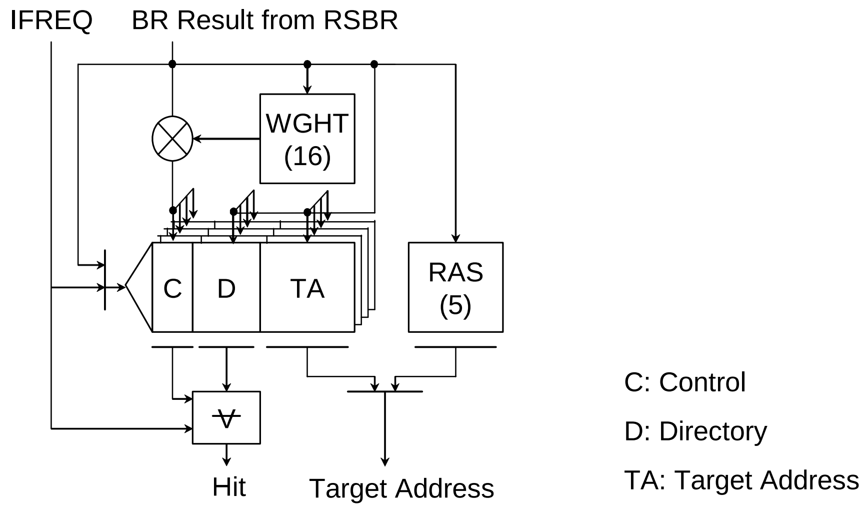

Figure 2 Branch Prediction of SPARC64 V

2.2 Instruction Execution

SPARC64 V has 6 execution units able to operate at the same time respectively. An instruction whose operands are ready is given priority, and dispatched to the execution unit in so-called data-driven out-of-order manner. An overview of the procedure to execute instructions after the instruction fetch is as follows.

1) Instruction presentation stage:
Instructions in IBF are taken out in program order at instruction presentation stage. 4 instructions are set to IWR (Instruction Word Register).
The instruction presentation stage controls supply of a consecutive instruction stream, switch of the instruction stream corresponding to branch instruction and supply of the delay slot instruction. This mechanism enables the instruction fetch and the instruction execution to work independently.

2) Decode/Issue stage:
At the Decode/Issue stage, 4 instructions on IWR are decoded at the same time and resources needed to execute the instructions are determined. These resources consist of various reservation stations, fetch queues, store data queues, and register update buffers. All the instructions are assigned IID (instruction identifier) within the range of 0-63. When all the instructions on IWR are issued, next instruction stream is set to IWR.
IWR has no restriction of combination of instructions. Every slot in IWR is able to allocate any resources such as reservation stations and so on. Parts of instructions in IWR can be issued in the program order as long as the corresponding resources are available, even though there are not enough resources for all the 4 instructions. Whatever binary code is able to run in the high degree of parallel thanks to the elimination of the issue stall.

3) Dispatch stage:
SPARC64 V is equipped with reservation station named RSE (Reservation Station for Execution) for the integer operation and RSF (Reservation Station for Floating-point) for the floating-point operation. RSE and RSF are split into 2 queue blocks respectively corresponding to the execution unit. That is, there are 4 reservation stations (RSEA, RSEB, RSFA, and RSFB). An instruction whose operands are ready is given priority and dispatched to the corresponding execution unit. At most 4 operations are dispatched at the same time. The dispatch algorithm is basically to select the oldest one among instructions in each reservation station, of which operands are ready (Oldest Ready).
SPARC64 V dispatches instructions speculatively from the reservation stations, to hide the disadvantage of deep pipelines. This technique is named speculative dispatch. In speculative dispatch, an instruction can be dispatched if operands are not ready but would be ready before the dispatched instruction reaches the execution stage. SPARC64 V assumes that a particular request of the prior instruction sent to the L1D$ hits the cache line, and it then speculates when the corresponding data from the L1D$ would be ready. Speculative dispatch hides the latency of L1D$ pipeline and uses execution units effectively.
RSEA, RSEB, RSFA and RSFB hold up to 8 instructions respectively. That is, at most 16 integer instructions and 16 floating-point instructions can be kept in the reservation stations.

4) Register read stage:
The number of the read port of the register has been determined so that all the execution units are able to work in parallel. That is, integer register GPR (General Purpose Register) has 8 read ports comprising 2x2 read ports for 2 integer operations, and 2x2 read ports for 2 load/store address generation. Floating-point register FPR (Floating-Point Register) has 6 read ports corresponding to 2 FMA units.
GPR has 8 sets of register windows. The register windows are effective to reduce the software overhead with the subroutine call (register save and restore). On the other hand, it is difficult to read the data from very large GPR at high frequency. To resolve this, SPARC64 V has a subset of GPR as work registers. This subset is named JWR (Joint Work Register). JWR keeps 3 windows (64x8 byte) in total including a current window register and both the sides of it. Read data from JWR is fed into the execution units. Both JWR and GPR are updated at the same time on instruction commit. When the window switch occurs, hardware copies another window data into JWR from GPR on the background, so the window switch does not cause stall of the pipeline. As a result, even when the subroutine is called, this mechanism achieves high degree of parallelism.

5) Execution stage:
SPARC64 V has 2 integer execution units EXA and EXB. Both EXA and EXB can execute arithmetic and logical operation and the shift operation. Only EXA can execute multiply and divide. There are FLA and FLB for floating-point instructions. Both FLA and FLB can execute add, subtract, multiply, divide, and square root operation. Each operation except divide and square root operation is executed in the pipelined manner.
SPARC64 V supports FMADD (Floating-point Multiply and ADD) in addition to the SPARC V9 instruction set. This FMADD instruction is effective for the matrix-multiply mainly used in HPC (High Performance Computing) and R&D (research and development) areas. Both FLA and FLB are able to execute the instruction. 3 operands are read at the same time. At first multiplication of 2

operands is done by multiplier, and the result is forwarded to the adder with the third source operand, and then added. This process is done in a pipelined manner. That is, totally 4 floating-point operations including 2 add and 2 multiply can be executed every cycle.
In addition, FLA has the graphics unit.
The result data from a given execution unit is directly forwarded to another execution unit when necessary to maximize throughput.
The result of operation is maintained in the register update buffer until the instruction is committed. 32 GUB (GPR Update Buffer) is for GPR and 32 FUB (FPR Update Buffer) is for FPR respectively. This resource is allocated at the Decode/Issue stage, and the register update buffer is freed when the instruction is committed.

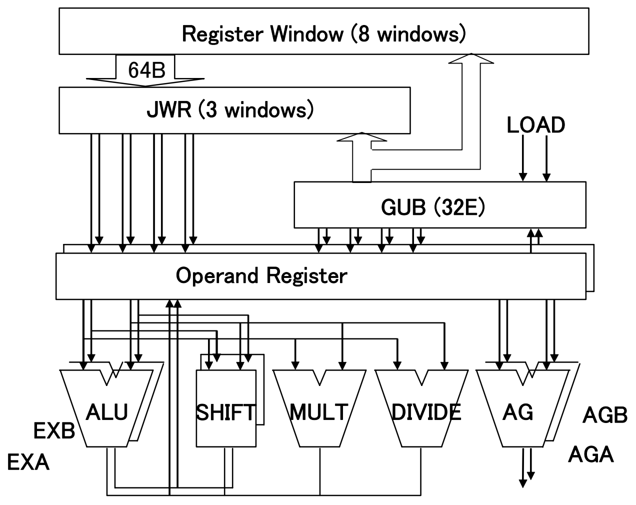

Figure 3 Integer Execution and Address Generation Unit of SPARC64 V

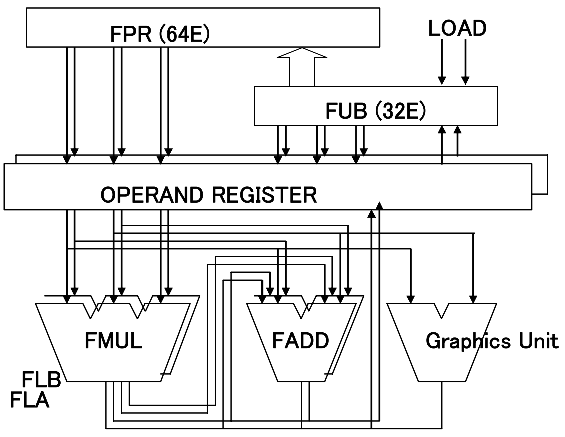

Figure 4 Floating Point Execution Unit of SPARC64 V

6) Commit stage:
SPARC64 V has 64 CSE (Commit Stack Entry) corresponding to IID, which is allocated when an instruction is issued and freed when committed. This implies up to 64 instructions can be in-flight at the same time.
CSE manages to update any programmable resource with the result data of execution. The result data of out-of-order execution is always written into GUB or FUB which is not visible from software. To guarantee the program order, register such as GPR and FPR, and memory are updated in the program order (in-order) at the commit stage. In addition, the control register including PC is updated at the commit stage at the same time. As a result, precise interrupt is guaranteed. Also all the in-flight instructions not reached the commit stage can be cancelled when necessary. This mechanism is named synchronous update method. This does not only make it easy to cancel processing in case of miss-prediction of branch target or interrupt, but also contribute to the improvement of RAS as explained in the following chapter.
Up to 4 instructions can be committed at the same time.

With the following sections, execution of branch instruction and load/store instruction is described

7) Branch instruction execution:
Brach instructions consist of PC-relative branch and register-indirect branch. A PC-relative branch instruction, which occupies the large majority of branch instructions, calculates branch target address when the instruction is decoded. A register-indirect branch calculates target address by using EXA after issuing the instruction.
In any case, the instruction is issued to a reservation station named RSBR (Reservation Station for Branch). RSBR can hold up to 10 branch instructions, and has the role to resolve the branch operation, to report the information to branch prediction control block, and to fetch correct target in

case of branch miss-prediction.
RSBR includes the control tag in each entry. When a conditional branch instruction is issued, the IID of the latest instruction generating the corresponding condition code is set in RSBR entry. RSBR uses this IID to monitor instructions executed out-of-order, to capture the condition code, and to find out whether the branch instruction is taken or not taken.
Once a branch instruction is resolved and target address is calculated, the branch prediction mechanism including branch history is updated.
When RSBR finds out prediction of branch target address is wrong, the instruction fetch block fetches the resolved target address. In addition, the existing speculative requests to L1I$ are cancelled. These actions are activated out-of-order.
When a miss-predicted branch instruction is committed, FLUSH signal is broadcasted and the following instructions executed speculatively are discarded. Thus, branch instructions can be executed in parallel with other instructions. Without disturbing other instructions as much as possible, branch instruction operation contributes to the improvement of the effective performance.

8) Load/Store instruction execution:
For Load/Store instructions, SPARC64 V allocates FQ (Fetch Queue) at the Decode /Issue stage, and issues the instructions to RSA (Reservation Station for Address generation). For Store instructions, RSE or RSF for store data access from register-file and SDQ (Store Data Queue) are allocated at the same time.
RSA can hold up to 10 load/store instructions. Two of the oldest entries whose operands are ready are selected at the same time (Oldest-Ready, 2nd Oldest Ready) and dispatched to EAGA/EAGB (Effective Address Generation units A and B). A speculative dispatch and the data forwarding are applied to address generation units as well as other execution units.
Effective address is set in FQ after it is generated with EAGA and EAGB. At this time, cache access begins immediately unless the request conflicts with previous requests. RSA is freed when the effective address is set in FQ, and RSE or RSF is freed when the store data is set in SDQ.
FQ controls the cache access pipeline, and SDQ controls the cache fill of the store data. FQ consists of 16 entries, and SDQ has 10 entries.
The cache access pipeline is composed of 2 pipelines called OF (Operand pipeline F) and OG (Operand pipeline G), which work independently and accept 2 requests at the same time.
When a given operation is not able to continue due to cache miss and so on, the corresponding cache access pipeline is aborted. However, even if one pipeline is aborted, the other can continue to process. Moreover, the following request can continue to process by outstripping the aborted request within the same pipeline as long as there is no aborted factor in itself. The aborted request is activated again at FQ later.
A store instruction can be committed with the data reserved in SDQ even when the target line of the store does not exist on cache, and the following instructions continue to be committed. SDQ is freed after the store data is filled into cache.
Moreover, when a load instruction refers to the data of the prior store instruction, data forwarding from SDQ to the load occurs.
In summary, simultaneous 2-operand cache pipelines and non-blocking cache operation realize high throughput performance in SPARC64 V.

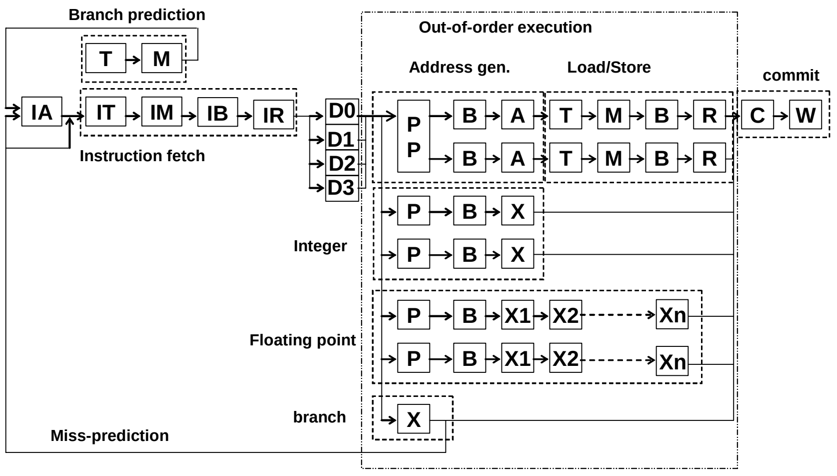

Figure 5 Pipeline of SPARC64 V

2.3 Cache Structure

SPARC64 V Cache structure is shown in Figure-6. The cache consists of two levels of hierarchy, the first level cache (L1$) with middle capacity (256KB) and the second level cache (L2$) with large capacity (max: 4MB).

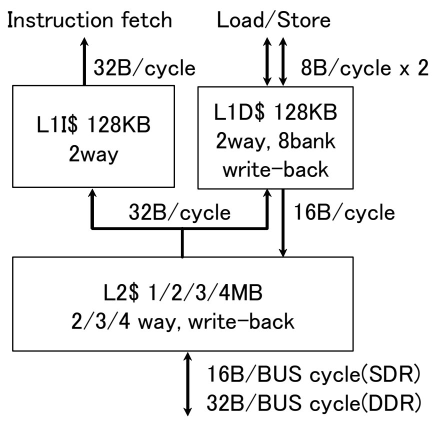

Figure 6 SPARC64 V Cache Structure

L1$ consists of instruction cache (L1I$) for instruction fetch and data cache (L1D$) for operand access. Both L1I$ and L1D$ has capacity of 128KB, 2-way set associative with 64B line size.
L1D$ structure is shown in Figure-7. L1D$ consists of 8 banks separated in 4 byte boundary and each bank can be accessed by one of the operand pipelines. According to this scheme, the above-mentioned operand pipelines can access to any banks simultaneously as long as there is no bank-conflict. L1$ is indexed by virtual address and tagged by physical address (VIPT). In VIPT, the same physical address might be indexed by different virtual address on the L1$. This is called synonym problem, and the synonym might destroy the data integrity. The synonym problem on the L1$ is solved by the L2$ which avoids to create synonym entry in the L1$. L1I$ is controlled by two cache states: V (valid) and I (invalid). L1D$ is controlled by three cache states: M (modified), C (clean) and I (invalid). Cache replacement policies of L1I$ and L1D$ are LRU (Least Recently Used).

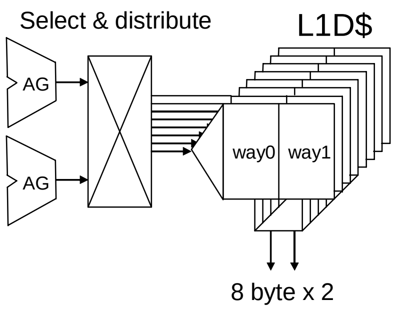

Figure 7　SPARC64 V L1D$

L2$ is composed of a variety of structure depending on the models. The capacity is 1 / 2 / 3 / 4 MB, set associativity is 2 / 3 / 4 ways, and line-sizes are all 64B. L2$ is indexed by physical address and tagged by physical address (PIPT). L2$ is controlled by five cache states: M (Exclusive Modified), O (Shared Modified), E (Exclusive Clean), S (Shared Clean), and I (Invalid). The cache replacement policy of L2$ is LRU. 64 bytes of L2$ data are read with the half frequency of the core. The low clock frequency of L2$ contributes to the power saving. Further more, clock to L2$ RAMs is activated only when they are accessed. This contributes to the power saving as well.
The cache update policies of L1D$ and L2$ are both write-back. That is, the store data is written into only one cache hierarchy. In the write-back method, cache-missed lines are always loaded on to the cache memory, so that the store operations can complete with updating one cache hierarchy. Generally, because the frequency of the store operation to the same cache line is high, the write-back method is advantageous on reducing the traffic between cache hierarchies and main memory.
In the write-back method, the cache data move-out occurs when the other processor accesses to the

cache data, or when the replaced cache line is needed to update the memory. The move-out data is written in outside hierarchy of cache or the memory. The former move-out is more important for the performance of the multi-processor system, that is, the response speed to the access demanded from the other processor is crucial. The L1$ and L2$ of SPARC64 V are tightly coupled so that the demanded data is sent very quickly even if the data is located in the L1$.
In the write-back method, the latest data is kept on the cache. When the error occurs on the processor, there is possibility that the damage extends over the system. SPARC64 V deals with the problem by strong RAS functions as described later in detail.
The data bus from L2$ to L1$ is driven at the same frequency with the core clock, and it has the width of 32 bytes. Therefore, the throughputs of reading L2$ and forwarding to L1$ are the same. On the other hand, the data bus from L1D$ to L2$ works at the same frequency in the core clock, and it has the width of 16 bytes. The bus throughput of each direction is sufficient to the maximum core throughput of instruction execution. Stagnation of processing does not easily occur even in the tough conditions that the cache hit rate of L1$ is extremely low.
As explained above, the cache system of SPARC64 V meets the requirement for large-scale transaction processing and the HPC processing, and it always achieves high performance.

2.4 System Interface

In the system interface of SPARC64 V, the number of out-standing requests, that is, the number of issues before formerly issued request is completed, has been greatly enhanced compared with the past SPARC64 processors. The number of out-standing load requests is 16, and the number of out-standing write requests to memory (write back) is 8, and the number of out-standing coherent order from system to processor is 4. Each request has an identifying number to couple a request with its completion, and this makes it possible to execute the load requests out-of-order and to ease the restrictions in the execution order among the other requests.
Thanks to these enhancements of the SPARC64 V system interface, the parallelism in the system is greatly improved and high performance has been achieved. Also SPARC64 V has a mode that is compatible with the conventional systems.

2.5 Guarantee of TSO (Total Store Order)

In order to run software including Operating System, the result of load operation needs to reflect the order of the store executed in the other processor in the multi-processor system. This is called as “sequential consistency” or “Total Store Order”.
SPARC64 V keeps high parallelism in processing by the out-of-order load operation even in case of the cache misses. But this might violate the ordering of TSO. Suppose that an ‘older load’ is waiting for execution by a cache miss, and a ‘later load’ is executed in out-of-order by cache hit, and the ‘older load’ is executed after the ‘younger load’. When the ‘older load’ picks up new data and ‘younger load’ picks up old data, there is possibility of violation of TSO. To avoid this, SPARC64 V observes the operand address in FQ and compares it with the addresses requested by other processors for invalidation or move-out, and addresses for move-in. As a result, a processor can tell the possibility that previously loaded data of the ‘younger load’ is damaged (moved-out), and the possibility that later loaded data of the ‘older load’ is new (moved-in). When there is a possibility of violating TSO, processor re-execute from the next instruction to the ‘older load’.
With this mechanism, SPARC64 V has accomplished the full-range out-of-order execution from L1$ to the memory.

2.6 MMU (Memory Management Unit)

TLB (Translation-Lookaside Buffer) structure of SPARC64 V is shown in Figure-8. TLB is composed of two hierarchies: mTLB (main TLB, or second-level TLB) and μTLB (micro TLB, or first-level TLB). In each TLB, four page sizes are supported: 8 Kbytes, 64 Kbytes, 512 Kbytes, and 4 Mbytes). TLBs consist of instruction TLBs for instruction fetch and data TLBs for operand access, and the structure is the same with each other. One pair of the mTLB and μTLB is described as follows.

MTLB is composed of RAM and CAM (Contents Access Memory). RAM has 1024 entries of two way set associative compositions. MTLB-RAM is only for the page size of 8 Kbytes. The replacement policy for RAM is LRU. CAM has 32 entries of full associative compositions. The entries of page size other than 8 Kbytes are kept in CAM. Locked entries including 8 Kbytes page size are also kept in CAM. The replacement policy for CAM is pseudo LRU.

μTLB is a partial copy of mTLB, and is accessed at high speed by the L1$ pipeline. μTLB has 32 entries of full associative composition, and the replacement policy is FIFO (First-In-First-Out). μTLB is able to handle all the page sizes. Even if an entry is locked on mTLB, it is replaced on the μTLB. When a μTLB miss occurs, hardware copies the entry from mTLB in the background, while processing the following load or store operations, and several cycles later μTLB missed request is re-executed.

With those two hierarchy TLB systems, TLB of SPARC64 V achieves both large capacity and high-speed access.

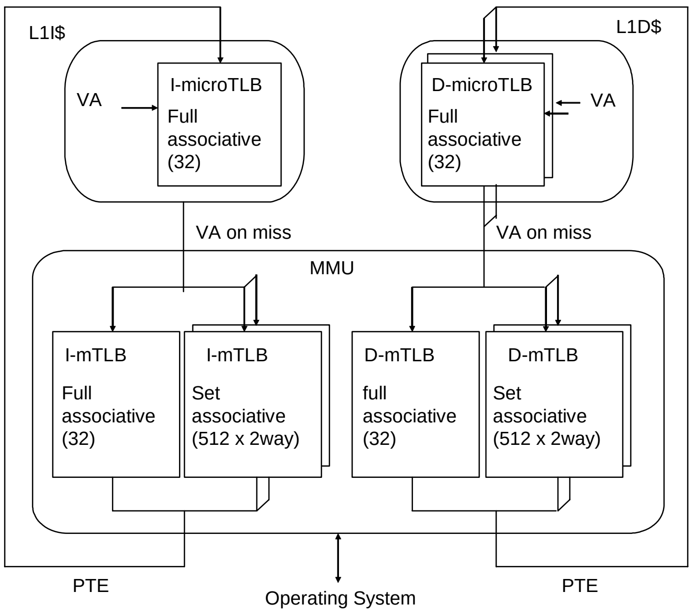

Figure 8 SPARC64 V MMU

2.7 Prefetch Mechanism

SPARC64 V provides hardware to prefetch data implicitly by hardware and explicitly by software.

1) Chained prefetch mechanism (hardware prefetch)
Hardware speculatively issues the prefetch operation based on the prediction that there is high possibility to access to the following continuous address in the future, if there have been load accesses for a consecutive address.
To detect a consecutive access, SPARC64 V employs the special address table. When L1$ miss occurs, the next line address (+64 byte) of the L1$ miss is registered in this table. When succeeding load address matches with the registered address, prefetch is executed at the next line address of the registered address.
Both the address tables for the instruction fetch and for the operand load / store operations consist of 16 entries corresponding to 16 series of prefetch requests. Prefetched cache line by chained prefetch is loaded into L2$.

2) Software prefetch mechanism
It is a common method for compiler to schedule a prefetch instruction before its data is used in the instruction execution. In SPARC64 V, an effective scheduling for the compiler is supported by the control option of the prefetch instruction; cache hierarchy the data to be loaded into, the cache state (Shared or Modified) and if the trap occurs or not on TLB miss.

Besides the above mentioned prefetch, when load resources from L2$ to L1$ run short and it is impossible to load any more data to L1$, the load address is still sent to L2$ and the corresponding data is prefetched to L2$. As a result, parallelism of accessing to the memory system is improved, and the cache miss penalty is concealed as much as possible.

CHAPTER 3 RAS Features in SPARC64 V

One of the design goals of SPARC64 V was to implement advanced RAS features that are comparable to mainframe systems. The following fundamental RAS functionalities have been implemented.

- Detect errors
- Restrict effect of errors within small regions
- Attempt to recover from errors
- Log errors
- Notify the software of errors

With these functionalities, SPARC64 V provides advanced Reliability, Availability, Serviceability, and Data Integrity, which are essential requirements to the CPU of mission-critical UNIX servers.

3.1 RAS of Internal RAMs
RAMs are the most error-prone parts inside CPU. SPARC64 V corrects all single-bit RAM errors by hardware automatically, without any software intervention required.

Table 1　RAM Error Handling in SPARC64 V

RAM type	Error detection/protection method	Error correction method
L1I$ data	Parity	Invalidation and reload
L1I$ tag	Parity & Duplication	Reload the duplicated data
L1D$ data	SECDED ECC	Single-bit error correction by ECC
L1D$ tag	Parity & Duplication	Reload the duplicated data
L2$ data	SECDED ECC	Single-bit error correction by ECC
L2$ tag	SECDED ECC	Single-bit error correction by ECC
Instruction mTLB	Parity	Invalidation
Data mTLB	Parity	Invalidation
Branch History	Parity	Recover from branch prediction miss

1) Error Detection and Correction Operations by ECC
L1D$ data, L2$ data and L2$ tag portions are protected by Single Error Correction Double Error Detection (SECDED) type ECC (Error Correcting Code). Both L1D$ and L2$ is able to correct corrupted read data with the ECC circuitry. The corrected data is used in the subsequent stages, such as arithmetic execution. Therefore, even a permanent error can be corrected as long as it is a single bit error. The line with error data is reloaded by the corrected data, in an attempt to recover from intermittent errors. This prevents uncorrectable multi-bit errors from occurring. When the L2$ tag portion detects a single bit error, the line is reloaded by the bit-flipped data. This "reverse write" capability enables the chip to cope even with permanent errors.

2) Error Detection and Correction Operations by Parity
The following parts use parity for error detection: L1I$ data, L1I$ tag, L1D$ tag, both the tag and data of the branch history, and the mTLB. When L1I$ finds out that the read data is corrupted, it discards the data and invalidates the offending entry. Then the read operation is restarted, which results in a L1I$ miss. The data from the L2$ (without error) is bypassed to the subsequent stages. Thanks to this mechanism, even a permanent single-bit error is always corrected. The error-free data from L2$ is reloaded to overwrite intermittent errors, just like L1D$ and L2$.
Tags of L1I$ and L1D$ are duplicated. If an error is found on either side, that fact is notified to the other. Using the tag information from the error-free side, the line is either invalidated in L1$ or written back to L2$. After that, the correct data is reloaded with a normal cache miss handling sequence.
The branch history checks its prediction information while processing branch instructions. An error will make the prediction miss, and the normal recovery process from prediction misses will take care of all effects of errors. Thus, such errors are corrected regardless of the number of error bits or whether it is intermittent or permanent. Parity is only involved in error logging.
In mTLB, a single bit error causes the entry to be invalidated. This causes a TLB miss, and the normal TLB fill-in process will overwrite the corrupted entry.
As explained above, all single-bit errors of RAMs protected by parities are also recovered by the hardware. Software execution is not affected at all.

3) Degradation

L1$, L2$ and mTLB are capable of detaching malfunctioning ways. Frequency of errors is counted for each functional unit. If the frequency crosses a certain threshold, it is "degraded", i.e. the failing way will be detached and will not be used after that point.
Degradation is automatically done by hardware, as well as all the necessary operations to guarantee cache coherency; that is, writing back the dirty lines on the degrading L1D$ way to L2$, and writing back the dirty lines on the degrading L2$ way to the memory.
Way degradation does not affect software execution at all, except the slowdown in speed.

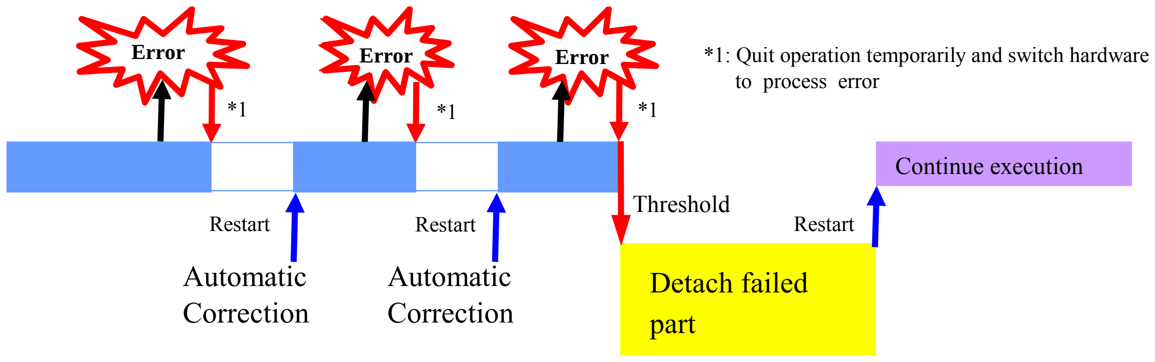

Figure 9 Automatic Correction and Degradation of Internal RAMs

3.2 RAS of Memory

SPARC64 V has a mechanism for error protection and error isolation of memory.

1) Error protection and correction by ECC
Memory of PRIMEPOWER is protected by ECC. When SPARC64 V reads memory data and detects a single-bit error, the hardware corrects it before using it. Uncorrectable multi-bit errors are reported to software via interrupts.

2) Correction of memory error by software intervention
Single-bit or multi-bit memory errors detected by SPARC64 V are recognized by software via interrupts or polling, with the address and error type logged in the error information registers. On recognizing a single-bit memory error, the software overwrites the corrupted area and changes the corresponding cache line to a dirty state. This causes the line to be written back to memory, erasing a single-bit intermittent error.
Also, the software reads a large area of memory at a certain interval. When a single-bit error is detected, the error is corrected by the procedure mentioned above. This is effective in preventing single-bit intermittent errors from evolving into multi-bit errors over time.

3) Memory Error Marking
When memory read data has a multi-bit error, a special mark identifying the source of the error is written into the data and the ECC syndrome becomes some special value. This is called error marking. When a functional unit detects a multi-bit error, it can tell whether the error occurred inside or somewhere else, by checking for error marking. Error marking provides valuable information to identify the source of errors. It also helps information gathering by eliminating unnecessary error

logging.

3.3 RAS of Internal Registers and Execution Units

SPARC64 V assures maximum data integrity by protecting registers and execution units as well.

Table 2 Error Protection for Registers and Execution Units in SPARC64 V

Execution Units/Registers	Error Protection Mechanism
GPR, FPR, JWR, GUB, FUB	Parity
PC, PSTATE, etc.	Parity
I/O Regs of Execution Units	Parity
Adder/Subtracter	Parity Prediction
Multiplier/Divider	Parity Prediction + Residue Check
Shifter	Parity Prediction
Graphics Unit	Parity Prediction

1) Protection of Internal Registers

GPR and FPR are equipped with parity. Both the window register itself and its copy (JWR) have parity. Errors in JWR are automatically recovered by reloading data from the window register.

The control registers such as PC (Program Counter), PSTATE (Processor STATE) and Interrupt Display Registers also have parity, so that the error does not spread out.

2) Protection of Execution Units

The inputs of the execution units have parities. The parity of the intemmediate result and the final output is validated. For multiplication/division, the parity for the input is checked, the output of the CSA (carry save adder) tree is checked using the residue check logic, the adder predicts the parity, and then checked with the parity generated from the output.

The outputs of the execution units are written to GUB or FUB with parities. The parity is propagated until it is written into GPR or FPR.

In addition, various data is checked for errors to achieve maximum data integrity. Such data includes register address, memory address, cache index, instruction opcodes, and address/data paths of system interface.

Furthermore, if instruction execution is stopped for a long time (hang-up case), a dedicated hardware will detect it.

3.4 In-order Grouped Update Mechanism and Instruction Retry

As explained in the Instruction Execution section, SPARC64 V adopts the In-order Grouped Update Mechanism. When an error is detected, all instructions that are currently in execution are cancelled. Since all internal states before the commit stage can be discarded, the programmable resources will see the results of only those instructions that have completed execution without detecting errors.

In addition to preventing the programmable resources from being corrupted, this gives hardware an opportunity to retry instructions after error detection. Hang-up cases can also be recovered, because the instruction caused the hang-up is discarded and restarted from scratch.

Instruction retry is automatically started on error detection. During retry, instructions are executed

one by one, in order to maximize the possibility of successful execution. On successful completion, the execution mode automatically gets back to the normal one.
During the above process, software intervention is not needed. If the retry succeeds, effects of the error are totally invisible to the software.
Instruction retry is repeated until it reaches a certain threshold, when an interrupt is generated to notify the software of the errors.

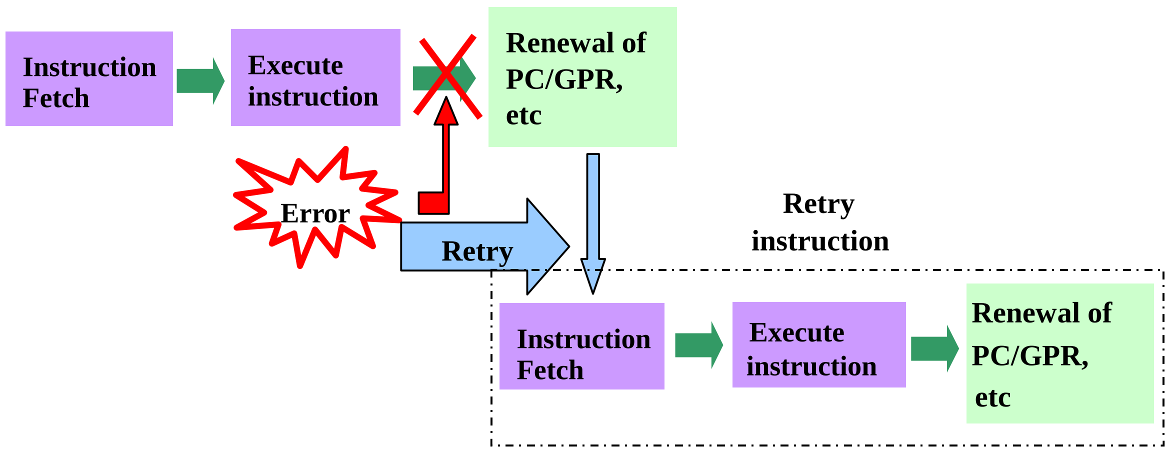

Figure 10 Hardware instruction retry

3.5 Software Interface of Error Handling

When an error to be reported to software is detected, SPARC64 V categorizes the error based on the severity of the impact to the software, logs the source of the error for each resource, and let software know the error through trap. This allows software to identify the area of error propagation, and to take appropriate actions according to error sources. In addition, SPARC64 V has a special execution mode to ignore errors as much as possible, to support dump operation for error information collecting.

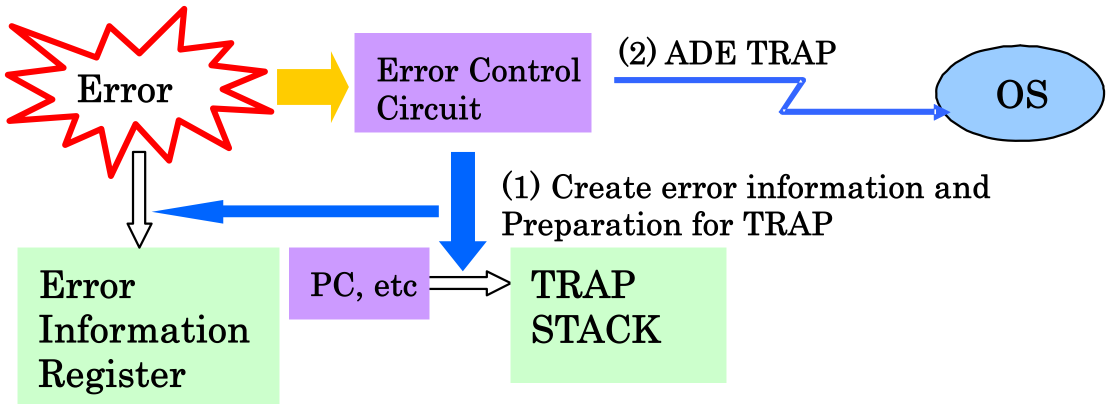
Error
Error Control Circuit
(2) ADE TRAP
OS
(1) Create error information and Preparation for TRAP
Error Information Register
PC, etc
TRAP STACK

Figure 11 Trap and Error Information Register

3.6 Improvements in Serviceability

As explained so far, SPARC64 V is equipped with error checkers in many places. Because of this, the chip (130nm process) has 679 "Region Codes (RC)" registers which identify the source of the error, and 821 "Retain Info (RT)" registers which provide additional information such as syndrome and error address. Furthermore, time-series information at the time of error occurrence is logged in the Event

History RAMs which consist of eight RAMs of 80 bit x 1024 index. The chip (90nm process) has 803 RCs, 3,077 RTs, and seven RAMs of 80 bit x 1024 index.

An occurrence of an error is notified to the system via a dedicated interface. When notified, the SCF (System Control Facility) firmware collects error logs using the dedicated interface, and analyzes them. These operations take place in background without affecting software execution.

These mechanisms allow systems with SPARC64 V to identify the source and type of the error quickly and precisely, while continuing the operation. The gathered information is valuable for preventive maintenance. All this leads to significant improvement in Serviceability.

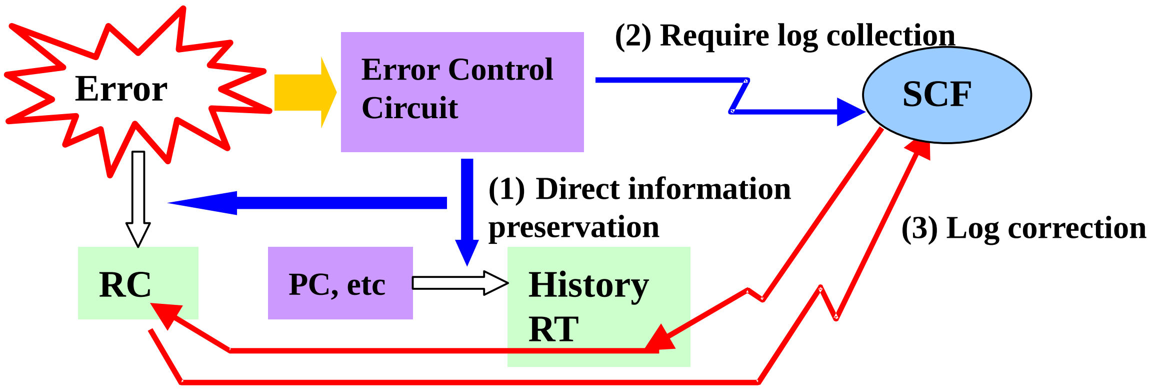

Figure 12 History, RC/RT and Hardware log

CHAPTER 4 Silicon Technology

The SPARC64 V processor is manufactured in 130nm or 90nm CMOS process.

4.1 SPARC64 V (130nm process)

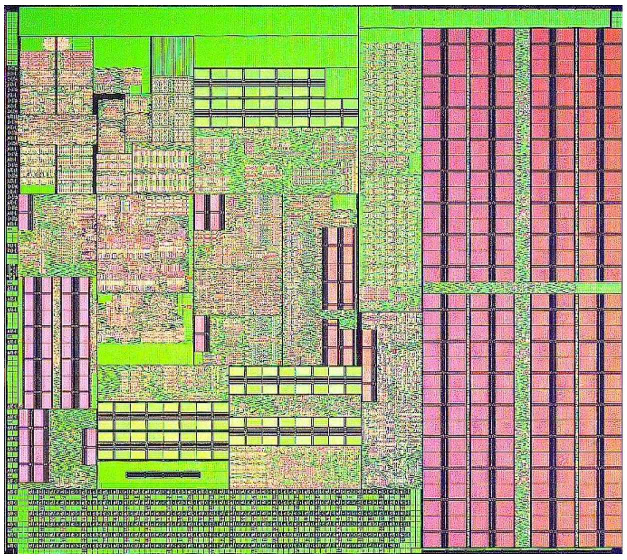
Figure 13 SPARC64 V (130nm process) Die Photo

SPARC64 V (130nm CMOS process) has eight metal (copper) layers, about 191 million transistors, and 269 signal pins. The size is 18.11mm by 15.99mm, and the chip consumes about 50W at 1.35GHz clock rate.

4.2 SPARC64 V (90nm process)

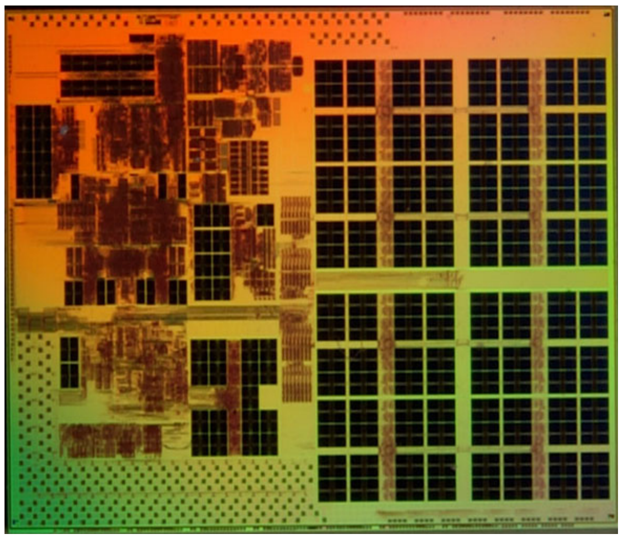
Figure 14 SPARC64 V (90nm process) Die Photo

SPARC64 V (90nm CMOS process) has ten metal (copper) layers, about 400 million transistors, and 279 signal pins. The size is 18.46mm by 15.94mm. It has achieved the clock frequency of over 2GHz.

CHAPTER 5 Performance Analysis

In these days, hardware architects are required to develop high-performance processors in a short

term. When hardware architects start their development, they are given performance goal numbers with several fundamental conditions, such as LSI technology, clock frequency and die-size. They are tasked to provide specification that realizes the highest processor performance under the given conditions. Short development term is another serious goal.
Our strategy to success in these conflict goals to realize highest performance in a short period is system level performance analysis with a software performance simulator in early stage of hardware development. Hardware architects would repeatedly test their designs on the simulator until it's accomplished a given performance goal numbers. This early stage evaluation would eliminate hardware remake, which results in short time processor development quite effectively.
Fujitsu Laboratories Ltd. (FLAB) has taken charge of software performance simulators (performance model) developments since mainframe processors were designed. They developed several generations of performance models for mainframe processors. And for SPARC64 V processor, they have developed a SPARC V9 performance model as well. Fujitsu hardware architects and FLAB researchers had learned from simulation studies and decided several specifications, such as basic pipeline structure, on-chip second cache, branch prediction scheme and depth of buffers for superscalar out-of-order execution, in quite early stage of hardware development. The SPARC V9 performance model is a trace-driven simulator, where the traces were generated on a real machine.

Characteristics of the performance model:

- High accuracy: Modeled in detail
- Consistency: Memory system as well as the processor core is modeled
- MP coverage: Server system performance analysis is available

"High accuracy" is essential for the performance model. FLAB has started development of the model in quite early stage of hardware development and continuously improved its details by reflecting actual hardware design. This is done by close relationship between FLAB researchers and Fujitsu architects.
"Consistency" is a unique point of this simulator. In general, memory system details are not modeled. But because we (researchers and architects) strongly believe that balance of memory system and processor core is an important factor to achieve system level performance, FLAB has developed memory system as well as the processor core.
"MP Coverage" is a vital feature to analyze high performance UNIX servers. UNIX processor used to be to be evaluated with uni-processor performance model only because the processor is regarded as for personal use that mainly deal with small applications. But nowadays UNIX processor is also used for servers that would deal with multi-user interactive workloads and high performance computing. In this reason, the model has to cover MP system performance evaluation as well.
The performance model is written by C-language, about ninety thousands steps and held five hundred runtime parameters. This variety of parameters has available flexible evaluations.
SPARC64 V has been developed with these performance analysis studies.

Concluding Remarks

Various techniques have been applied to achieve high performance in SPARC64 V. Also, the processor implemented advanced RAS capabilities, which had not been possible in previous UNIX servers. Power consumption is kept low compared to existing server processors, making it possible to build high-performance and yet compact systems. In addition, it supports the Solaris™ Operating Environment (Solaris OE), the de facto standard OS for UNIX servers. Customers can choose from among the most abundant set of server middleware and applications running on Solaris OE.

The first chip of SPARC64 V was manufactured at the end of December 2001.
It gave its first cry in a corner of Fujitsu Kawasaki Research & Manufacturing Facilities, closely watched by many people who had been involved in its development. In fact, development of this chip required considerable efforts of many people. Now the chip is out in the world, and we are confident that the SPARC64 V is the one that best satisfies customers' expectation.

References

1. SPARC International Inc.: The SPARC Architecture Manual-Version 9, 1994.
2. Sun Microsystems. Inc. and Fujitsu Limited: SPARC Joint Programming Specification (JPS1): Commonality.
   http://www.fujitsu.com/downloads/PRMPWR/JPS1-R1.0.4-Common-pub.pdf
3. Fujitsu Limited: SPARC JPS1: Fujitsu SPARC64 V Implementation Supplement.
   http://www.fujitsu.com/downloads/PRMPWR/JPS1-R1.0-SPARC64V-pub.pdf
4. H. Tone, S. Sugiura, N. Toyoki, and T. Chiba, “Hardware of GS8600,” FUJITSU, Vol.47, No.2, p.96-108 (1996).
5. T. Hikichi, A. Kato, H. Ohta, and Y. Kakimura “64-bit RISC Processor : SPARC64 Ⅳ,” FUJITSU, Vol.51, No.4, p.226-231 (2000).
6. D.H. Brown Associates, Inc.: SPARC64 V Microprocessor Provides Foundation for PRIMEPOWER Performance and Reliability Leadership.
   http://primeserver.fujitsu.com/primepower/catalog/data/pdf_db/sparc64_v.pdf
7. A. Inoue, “SPARC64 V Processor for UNIX Servers,” FUJITSU, Vol.53, No.6, p.450-455 (2002).
8. Kevin Krewell, “Fujitsu’s SPARC64 V Is Real Deal,” Microprocessor Report, October 21, 2002.