Abstract

A report for CS9244 on the UltraSPARC T1 processor. Firstly a history and overview of the SPARC architecture is given, focusing on some important features such as instruction set architecture and register windowing.

Subsequently we examine the latest implementation of the SPARC architecture, Sun's UltraSPARC T1. A massively parallel processor implementing Sun's concepts of "throughput computing", we detail the unique features of the implementation.

UltraSPARC T1

Ian Wienand

June 15, 2006

Contents

Chapter 1

SPARC Overview

Sun's UltraSPARC T1 is a new generation of chips described by Sun as embodying "throughput computing". This is realised by what many computer architects refer to as "scaling out" rather than "scaling up"; focusing on more cores and threads rather than higher clock speeds or more complex processor pipelines.

The UltraSPARC T1 (previously code-named Niagara) supports multiple cores, each running multiple threads. In the highest end configuration it has 8 cores each running 4 threads for a total of 32 threads running at any time. It sacrifices clock speed, cache and floating point performance to provide this tremendous amount of parallelism.

Firstly we examine the history of the SPARC processor and Sun's implementation, the UltraSPARC, to trace the timeline of development leading to the UltraSPARC T1.

1.1 SPARC History

The **S**calable **P**rocessor **ARCH**itecture (SPARC) has a long and interesting history.

SPARC refers to an abstract processor design specification. Implementors will take the SPARC architecture and turn it into an actual hardware product, as Sun does with the UltraSPARC line of processors.

Implementation of the SPARC architecture is divided into *privileged* and *non-privileged* components; processors which only implement the non-privileged component may run SPARC userspace applications, but not run general purpose SPARC operating systems.

1.2 Architecture History

The SPARC processor grew from work on both the original Berkeley University RISC architecture (1980-1984) and the Stanford University MIPS architecture (1981-1984), both of which emphasised what is now known generally as a **R**educed **I**nstruction **S**et **A**rchitecture (RISC).

Originally designed by Sun Microsystems in 1985, SPARC International, Inc. was formed in 1989 to promote adoption of the architecture. Consequently the architecture was licensed to other producers, increasing adoption of the chip.

The SPARC architecture started with SPARC V7 first published in 1986. This was followed by the SPARC V8 specification in 1990, which added enhancements such as hardware multiply/divide, MMU functions and 128 bit floating point operations.

This was then followed by the SPARC V9 architecture. The major change with V9 was a move to 64 bit registers and operations, however there were a number of minor changes such as an updated

Processor	Cores	Threads/Core	Clock	L1D	L1I	L2 Cache
UltraSPARC IIi	1	1	550Mhz, 650Mhz	16KiB	16KiB	512KiB
UltraSPARC IIIi	1	1	1.593Ghz	I	D	1MB[a]
UltraSPARC III	1	1	1.05-1.2GHz	64KiB	32KiB	8MiB[b]
UltraSPARC IV	2	1	1.05-1.35Ghz	64KiB	32KiB	16MiB[c]
UltraSPARC IV+	1	2	1.5Ghz	I	D	2MiB[d]
UltraSPARC T1	8	4	1.2Ghz	32KiB	16KiB[e]	3MiB[f]
UltraSPARC T2[g]	16 (?)	8	2Ghz+ (?)	?	?	?

[a]On-chip
[b]External, on chip tags
[c]8MiB per core
[d]32MiB off chip L3
[e]I/D Cache per core
[f]4 banks
[g]Second-half 2007

exception handling architecture, prefetching support and many updated and removed instructions [8]. At the time of writing, SPARC International have not released any specifications beyond V9.

1.3 Sun's UltraSPARC

The primary innovator with the SPARC architecture is Sun Microsystems. Sun's implementation is referred to as UltraSPARC, and like the underlying architecture has a long history. The currently available UltraSPARC processors and their features are displayed in Table 1.3.

The Roman numeral versioned architectures are generally similar, and largely based around the underlying UltraSPARC III architecture; the UltraSPARC IV line consists of dual UltraSPARC III cores on a single chip. They all implement the SPARC V9 architecture.

1.3.1 Current Developments

With the SPARC V9 growing older, Sun has released an updated "UltraSPARC Architecture 2005 Specification" which is superset of the SPARC V9 architecture and many additional extensions (which we examine below).

The first implementation of this revised architecture is the Sun UltraSPARC T1, and is the focus of this paper.

Chapter 2

SPARC Architecture

Below we examine some of the interesting parts of the SPARC V9 architecture.

2.1 Instruction Set

The SPARC instruction set embodies a RISC architecture. It is a load-store design, where operations only happen on values held in registers. Instructions are simple, mostly consisting of two inputs and one output, and are encoded in a fixed 32 bit opcode.

Addresses for load and store come in two forms

- *register + register*
- *register + 13-bit signed immediate*

A 32 bit instruction size is very efficient for keeping code size down, but means a 32 bit immediate value can not be directly loaded into a register. For this purpose SPARC provides a `sethi` instruction to set the top 22 bits of a register (22 bits presumably since there was sufficient room in the instruction, despite this overlapping with the 13 bit immediate available in `add`, etc).

For example, in Figure 2.1 a 32 value is loaded in two parts; the top 22 bits into register `g1` via a `sethi` instruction, then the bottom 10 bits via the mask provided in the `or` instruction.

Similarly to other notable architectures designed around the same time (MIPS), SPARC features a branch delay slot to aid in pipelining. We can see the unfilled branch delay slots in the example output of Figure 2.2.

SPARC V8 also relied on condition codes for compares, a technique eschewed on later architectures such as Alpha and Itanium to avoid restrictions on multiple instruction issue and bottlenecks on implicit resources[5]. SPARC V9 attempts to alleviate this with arithmetic operations being divided into two classes; one sets condition codes and the others do not (PowerPC has similar features). SPARC V9 adds a range of instructions which can do comparisons based on values held in integer registers, as per the later architectures.

To keep the instruction set small, many instructions are implemented as *synthetic instructions*; that is they are reduced to an instruction that is semantically the same but is generally less intuitive. For example, in Figure 2.2 we can see the operation field for the decoded `cmp` instruction expands to `010100`, which is the opcode for a `subcc` (subtract and set condition code) operation. Thus in actuality `cmp` is a *synthetic instruction* for `subcc %g1, 0, %g0`, which will sets up condition code for the following branch (`bne`).

```
void addr(void) {
  int i = 0xdeadbeef;
}

00000054 <addr>:
  54:   9d e3 bf 90     save  %sp, -112, %sp
  58:   03 37 ab 6f     sethi  %hi(0xdeadbc00), %g1
  5c:   82 10 62 ef     or  %g1, 0x2ef, %g1     ! deadbeef <addr+0xdeadbe9b>
  60:   c2 27 bf f4     st  %g1, [ %fp + -12 ]
  64:   81 e8 00 00     restore
  68:   81 c3 e0 08     retl
  6c:   01 00 00 00     nop
```

Figure 2.1: A disassembly of a simple function for SPARC

2.2 Registers

SPARC was among the first commercial processors to implement *register windows*. This is a technique for "hiding" registers so that called functions do not take the penalty of having to save and restore them on function entry and exit. The complete set of registers is termed the *register file* and the subset available to the currently running function is the *register window*.

The SPARC register windowing scheme was built around analysis that functions generally have six or less input parameters and nest only several levels deep. 32 general purpose integer registers are available to the processor at any time, as described below and illustrated in Figure 2.3.

General	Windowed	Description
%r0 - %r7	%g0 - %g7	Global Registers (available in any window)
%r8 - %r15	%o0 - %o7	Window output Registers
%r16 - %r23	%l0 - %l7	Window local registers
%r24 - %r31	%i0 - %i7	Window input Registers

Conceptually, on a function call the output registers of the calling function become the input registers of the called function. The windowed register aliases (`g`, `o`, `l`, `i`) are provided by the assembler for programmer convenience; the processor handles the renaming such that the general names (`r`) refer to the correct physical registers in the register file.

Rotation of the register window is under programmer control via the `save` and `restore` operations. This is in contrast to Berkeley RISC which moved windows only on procedure call and return; by allowing moving of the register window at any time the compiler has more freedom to do inlining or allow a single function to obtain access to more registers.

Rather than the linear layout presented in Figure 2.3, the register window is implemented in a circular buffer. In SPARC V8, the processor keeps a pointer to the currently active window in the *current window pointer* (`CWP`). On a `save` instruction the registers are renamed such that a new window is presented. A `restore` instruction does the opposite, renaming to reveal the previous register state.

Since there is a finite circular buffer, clearly multiple saves will lead eventually lead to old values being overwritten. This situation is called *overflow*, and will raise a *overflow trap*. At this point, control is returned to the operating system where the values in the window about to be overwritten are *spilled* to memory for storage.

```
int branch(int i)
{
 if (i == 0)
   return 0;
 else
   return i;
}

00000070 <branch>:
  70:   9d e3 bf 90     save  %sp, -112, %sp
  74:   f0 27 a0 44     st  %i0, [ %fp + 0x44 ]
  78:   c2 07 a0 44     ld  [ %fp + 0x44 ], %g1
  7c:   80 a0 60 00     cmp  %g1, 0
  80:   12 80 00 05     bne  94 <branch+0x24>
  84:   01 00 00 00     nop
  88:   c0 27 bf f4     clr  [ %fp + -12 ]
  8c:   10 80 00 04     b  9c <branch+0x2c>
  90:   01 00 00 00     nop
  94:   c2 07 a0 44     ld  [ %fp + 0x44 ], %g1
  98:   c2 27 bf f4     st  %g1, [ %fp + -12 ]
  9c:   c2 07 bf f4     ld  [ %fp + -12 ], %g1
  a0:   b0 10 00 01     mov  %g1, %i0
  a4:   81 e8 00 00     restore
  a8:   81 c3 e0 08     retl
  ac:   01 00 00 00     nop

Input Bitmap Type > sparc_int_op_rs1_rs2_rd
sparc_int_op_rs1_rs2_rd value > 0x80a06000
Decoded output for a Sparc Integer manipulation: register, register, destination
          Register 2 | 0 0000
              Unused | 0000 0000
                Zero | 0
          Register 1 | 00 0011
           Operation | 01 0100
Destination Register | 0 0000
                  10 | 10
```

Figure 2.2: A disassembly showing examples of comparisons, branches and branch delay slots on SPARC. At the bottom is a decoding of the `cmp` instruction, showing it is really a synthetic instruction for a `subcc`.

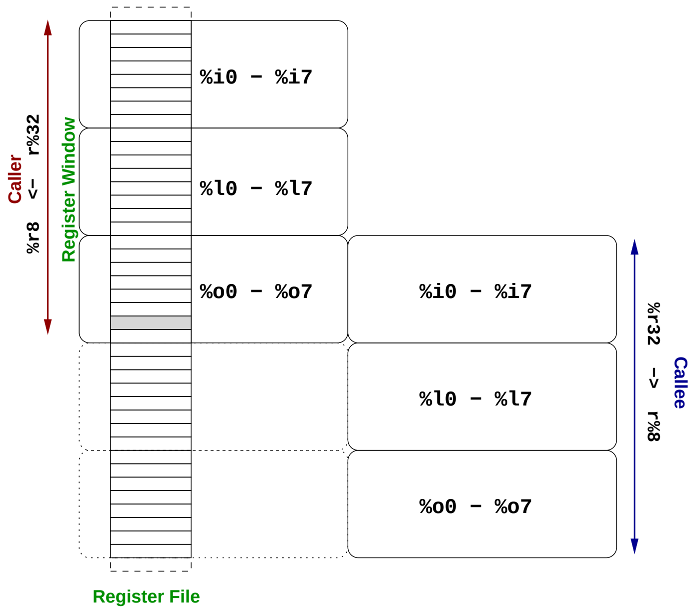

Figure 2.3: An illustration of SPARC register windows

This does raise the question of where to spill the registers too. By convention, one of the functions out registers is used as its stack pointer (`%sp`, as illustrated by the grey register in Figure 2.3). The assembler is responsible for making enough room on each functions stack allocation such that it can store its register window, should it be spilled.

`restore` operations are the opposite of `store`, and can cause a *underflow trap*. On underflow we again follow the stack pointer, but this time we *fill* by reading the registers back into the window, before returning control to the program.

2.2.1 SPARC V8 Register Window Implementation

SPARC V8 defines a 5 bit (maximum) field in the processor status register to record the current window, the *current window pointer* (CWP). Thus the maximum number of register windows is 32, though an implementation may be less. A corresponding register holds a bitmap of invalid register windows (Window Invalid Mask).

A complicating factor is that on any processor trap the register window is moved unconditionally, and nested traps are not supported (so a trap can not cause a further overflow trap). Thus to avoid the potential of overwriting registers one register window must be reserved at all times to handle a trap. To do this, one window must be marked invalid by the operating system (on trap, the validity will not be checked so it can be used).

Note that register windows of another process will also be marked as invalid on context switch, and the spill process will need to occur.

The CWP is decremented in response to a `save` operation. The validity of the new window is checked against the WIM, and a trap to spill the window raised if so. A similar process happens on `restore`.

2.2.2 SPARC V9 Register Window Implementation

SPARC V9 made a number of changes to the register window implementation. Firstly, in V8 a trap could not nest to cause another trap, and one register window needed to be kept clear for the trap to

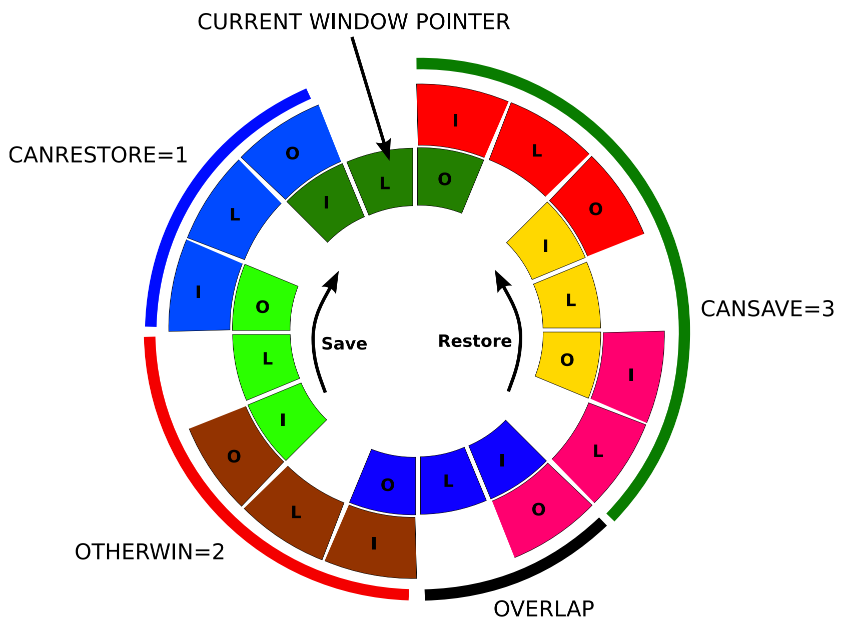

Figure 2.4: SPARC V9 register windows.

use. In V9 this was modified to allow up to 5 levels of nested traps, implemented by adding extra global register sets dedicated to trap handlers.

A major problem with the V8 scheme is that the WIM scheme does not support multiple address spaces well. To avoid having invalid register windows and leaking data, the window buffer must be flushed on context switch. Thus V9 implements an enhanced scheme for managing the register windows. The simple "valid-invalid" bitmap is replaced for a series of bounds registers, per below.

- `CANSAVE` : number of `save` instructions until an overflow trap is raised.
- `CANRESTORE` : number of `restore` instructions before an underflow trap is raised.
- `OTHERWIN` : number of windows which will generate an *alternate* spill/fill exception.

This scheme provides a number of benefits. If the user tries to `save` or `restore` to the `OTHERWIN` windows, alternative fault handers can be called. This provides both a way to implement faster trap handlers and better inter-process security. It can be used to implement faster system calls, for example, since by setting `CANRESTORE` to zero you can effectively detect the return from system call and use faster trap handlers.

2.2.3 Criticism and comparison of register windows

Register windows have been a contentious feature, and have had some criticisms. It has been claimed that data supporting the advantages of register windowing was taken from an inaccurate subset of workloads. Programs written in C, for example, might not have as much state required to be saved on function calls as higher level functional languages. If the processor starts to take spill and fill faults when potentially it did not need to, performance declines.

Systems code tends to have very deep call chains, which could often lead to much spill and fill overhead. Flushing the register window on context switches was also a major overhead of the early implementation.

Another criticism is that large superscalar processors can use potentially more registers than are provided by a register window. This is to allow disambiguation of incoming instructions to avoid dependencies. x86, MIPS, Alpha and PowerPC all use some form of register renaming to help alleviate this problem, but SPARC must retain register windows for backwards compatibility reasons.

Itanium is the only other modern architecture which implements register windows. However, Itanium has variable sized windows, and allocates the spill and fill mechanisms to underlying hardware called the *register stack engine*. This allows enhancements such as spilling in the background to make use of unused memory bandwidth, allowing the window size to be very large (up to 96 registers) and most importantly transparently growing the underlying register file, since the number of windows are not fixed [1].

2.3 VM and MMU Features

The SPARC V9 reference is deliberately scant on MMU details; implementation is left up to the processor. At the most general level, the reference states that a MMU should provide a virtual to physical mapping, though it should be possible to create an implementation without an MMU at all.

SPARC V9 implements separate addresses spaces via an *addresses space identifier*. The current ASI is kept in a register, and used by default. However, software is able to specify an different ASI via alternate load/store instructions. The lowest addresses spaces are considered reserved for the operating system, and any attempt to access them from higher ASI's will result in a trap.

This means that the operating system can easily write into the address space of userspace processes by specifying the correct ASI on writes, and can facilitate easy sharing.

2.4 Other Features

Making SPARC V9 a superscalar architecture meant a range of other changes, brining it in line with the features expected of a modern architecture. Branch predication and software controlled prefetching allow better hazard recovery, and non-faulting loads allow for speculation. For more analysis see Wiggins[8].

Chapter 3

UltraSPARC T1

Below we study the first implementation of the UltraSPARC Architecture 2005, the UltraSPARC T1.

3.1 Throughput Computing

Studies have shown that modern superscalar processors do not make good use of their available functional units. For a superscalar processor to be advantageous, the cycles per instruction (CPI) should be reduced below 1 (or, put another way, the inverse of CPI, the instructions per cycle, should be greater than 1).

Figure 3.1 refers to a theoretical 8-issue superscalar machine [7]. For this 8-issue superscalar to have a CPI of less than 1, more than $\frac{1}{8}th$ of the potential issue slots must be filled on each cycle (e.g. we are doing work in parallel on the multiple functional units). This equates to greater than $100(\frac{1}{8}) = 12.5\%$ of potential issue slots being filled, which we can see in Figure 3.1 is not satisfied for a number of tests.

With a CPI of greater or equal to 1 the advantages of the multiple functional units acting in parallel are lost, since the instruction stream is not offering them enough work to do. It would be better to have a larger collection of single issue processors which all work on separate instruction streams.

This realisation lead to the advent of *symmetric multithreading* (SMT), where the control and register logic are replicated for a number of threads which share underlying execution units. By sharing execution units, even if a particular thread does not exhibit a high CPI another thread can make use of the unused resources.

The UltraSPARC T1 takes this argument to an extreme by combining 4-way SMT with multiple cores all on a single die. Sun calls the model "throughput computing" in reference to the focus on getting total system throughput, rather than individual thread performance, high. Some benchmarks with this processor claim an IPC of 5.76 [4] (CPI of 0.17).

3.2 Pipeline Architecture

The processor is made of up to 8 cores, implementing the UltraSPARC Architecture 2005. Each core has a 16KiB 4-way set associative, 32 byte line size L1I and and an 8KiB 4-way set associative, 16KiB line size L1D.

Each of the 8 cores supports four threads, known as a "thread group". Each thread in the group has a unique set of registers and an instruction buffer, but the thread group shares L1 cache, instruction and data TLB entries, execution units and other pipeline resources.

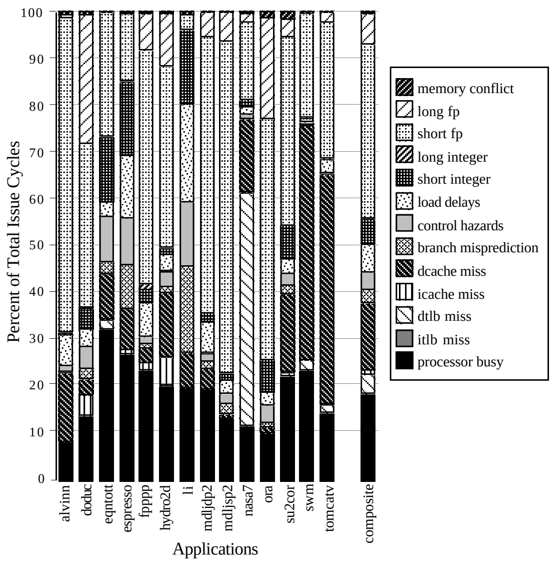

Figure 3.1: Sources of unused issue cycles in an 8-issue superscalar pipeline with the SPEC92 benchmark [7]. We can see that most issue slots are not filled, meaning execution units remain idle.

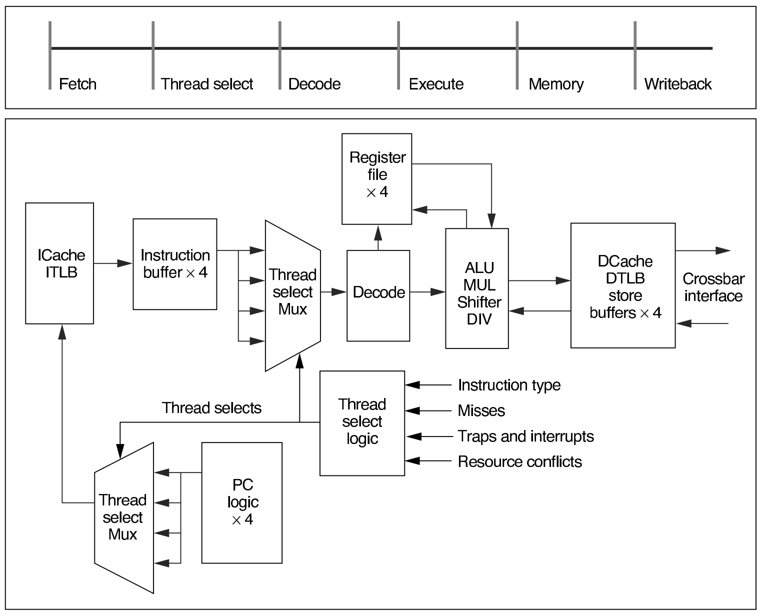

Figure 3.2: An illustration of the UltraSPARC T1 pipeline [3]

The pipeline is an short single-issue pipeline; consisting of fetch, thread select, decode, execute, memory and write back (Figure 3.2). Execution units consist of single cycle latency ALUs, shifters and multi-cycle latency multipliers and dividers. Each thread has a separate instruction buffer which can be filled when execute resources are not available. There is one floating point unit shared between the 8 cores.

The pipeline supports full bypassing (also known as "forwarding" [2]), helping to avoid RAW dependencies. Note there are none of the features common on other superscalar architectures such as memory speculation, of-of-order execution or predication; this keeps the pipeline simple and the power requirements to a minimum [4].

3.3 Thread Switching

The second stage of the pipeline is thread switch, where it is decided what thread should be passed through to the later pipeline stages.

Thread select uses several heuristics to decide on thread switch. The default policy is to switch between available threads every cycle with a LRU policy. A long latency instruction should cause a thread switch to keep the pipeline being filled.

The main thread switch policies are:

- *Predecode* information : long latency instructions such as `div` can be detected and flagged before instruction decode fetch with *predecoded* bits. Upon matching the bits, a thread switch can be enacted. For example, a `load` requires three cycles before the data is available for further use, so can be tagged as stalling for two instructions and create a thread switch.
- *Cache misses* are long latency, and will cause a thread switch. The scheduler assumes that loads are cache hits, and thus issues any further dependent instructions speculatively. However, a speculative thread has a lower priority than one with work to do.
- *Traps* happen only in the later stages of the pipeline, and involve both flushing out younger instructions and signalling to the switching logic that this thread is no longer valid.
- *Resource conflicts* can hold up a thread. Each thread has a separate instruction buffer which can hold a limited queue of instructions for contested execution logic.

3.4 Cache and Memory

The UltraSPARC T1 deliberately dedicates chip space to processing cores over cache as part of the push for "throughput computing". The underlying principle is that cache misses can be tolerated by switching to another thread which has work ready to do. Since cache misses are quite frequent, a large number of threads are required to increase the probability that there is work ready to do.

The L1 cache is write-through, and L2 is inclusive of L1. Two instructions are fetched into L1 per cycle; if the second instruction is used this leaves the cache access lines free to fill another line during the second cycle. The layout is illustrated in Figure 3.3.

There is a shared 3MiB L2 cache divided into eight banks and connected via a crossbar switch to the processor cores. The L2 is 12-way set associative, to avoid conflict misses from the many threads accessing it. It is 4-way banked to improve concurrent access, and each bank connects to DRAM via a dedicated channel.

This is intended for very fast inter-thread communication and data sharing; passing data between threads via the L2 cache is significantly faster than similar operations in a traditional SMP system.

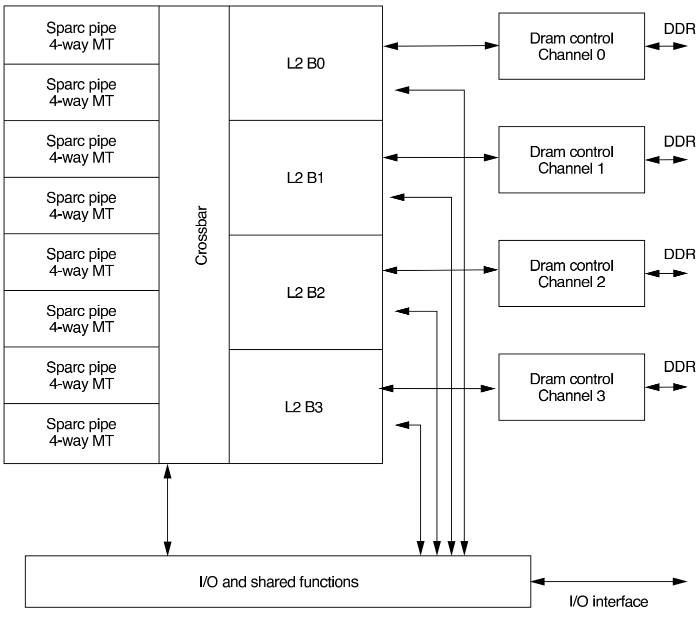

Figure 3.3: An illustration of the UltraSPARC T1 cache layout [3]

The L2 cache is connected to the cores via a 200GiB/s crossbar interconnect. This interconnect also goes to the I/O subsystem. Currently the chip does not support SMP implementations, so there is no L2 coherency protocol (future chips are rumoured to implement this).

3.5 Hypervisor

Rather than refer to physical processors, to support chip multi-threading the UltraSPARC Architecture 2005 refers to *virtual processors*, which are a representation of each thread running on a physical core. It also adds another mode to the processor (above the usual unprivileged and privileged) called *hyperprivileged* mode.

Hyperprivileged mode is intended for implementing a hypervisor layer. Just as privileged instructions cause code running in unprivileged mode to raise a trap into privileged mode, hyperprivileged mode is trapped into when privileged code executes an instruction dedicated as hyperprivileged.

A hypervisor is implemented with a thin firmware layer using the protections provided by hardware and the hyperprivileged mode. A full hypervisor API is documented [6] and presents a full virtual machine to a guest operating system. Entrance into the hypervisor is via the usual `Tcc` trap instruction, using specified high entry numbers. To minimise the trap handlers required, traps are split into *fast-trap* and *hyper-fast* trap; for a fast-trap the trap number is passed via a register and must be decoded.

The hypervisor exports a full *machine description* which the guest operating system can use to gain all relevant information about the underlying system. The hypervisor provides access to the PCI bus, virtual interrupts and (some) hardware error recovery.

3.6 MMU

Since MMU operations necessarily change shared machine state, the MMU operations of privileged mode on the UltraSPARC T1 have been constrained to work with the hypervisor layer. First we examine the underlying principles of the SPARC MMU.

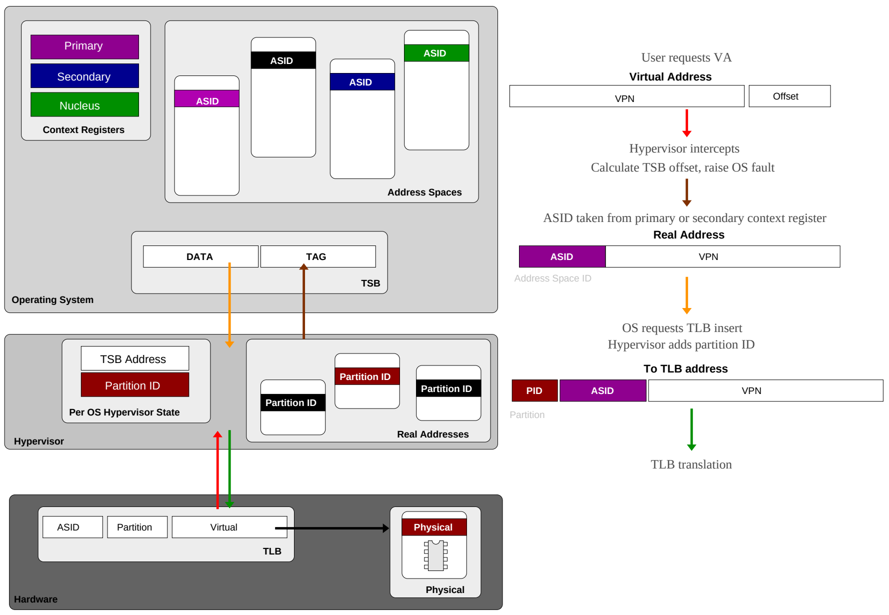

Figure 3.4: An illustration of address space translation

The format of entries for insertion into the SPARC TLB is referred to as a *Translation Table Entry* (TTE). A TTE entry consists of a tag and data; the tag consists of a virtual address and a *context id*, whilst the tag contains the underlying *real* or *physical* address (n.b. these are not the same!).

TLB refills are under software control; e.g. any TLB miss traps to the operating system which will insert the appropriate translation. To facilitate faster TLB refills, TTE's can be stored in a *Translation Storage Buffer* (TSB). The TSB is a direct mapped cache of TTE entries. On a fault, the processor will pre-compute a pointer into the TSB which is available to the fault handler. The fault handler can then quickly check if the entry is correct, inserting it if so.

When the hypervisor is turned off, there is a one–to–one mapping between real addresses and physical addresses. In this case the TSB is used as only as a cache for the TLB refill handler.

In cases where the hypervisor is enabled, as per Figure 3.4, the hypervisor is necessarily responsible for inserting the translation into the TLB.

The operating system registers its fault handlers and TSB locations with the hypervisor. On fault, the hypervisor intercepts the hardware trap and computes the correct offsets into the TSB for the particular guest OS and passes control to the pre-registered fault handler. The operating system then requests the hypervisor insert the TLB translation. The hypervisor tags the TLB entry with the partition ID of the guest OS held in the partition id register, which is checked on the TLB fault path.

3.6.1 Multiple page size support

The processor pre-computes two offsets into the TSB for two user specified page sizes. A flag can be set to indicate a "split TSB", such that one can keep individual TSB's for each page size (see Figure 3.5).

The operating system, being in control of the fault handlers, can additionally perform extra operations on the computed pointers to establish other schemes, such as a set-associative TSB layout. For example Solaris has one TSB for 8KiB (base), 64KiB and 512KiB pages, and uses the second split TSB exclusivly for 4MiB pages.

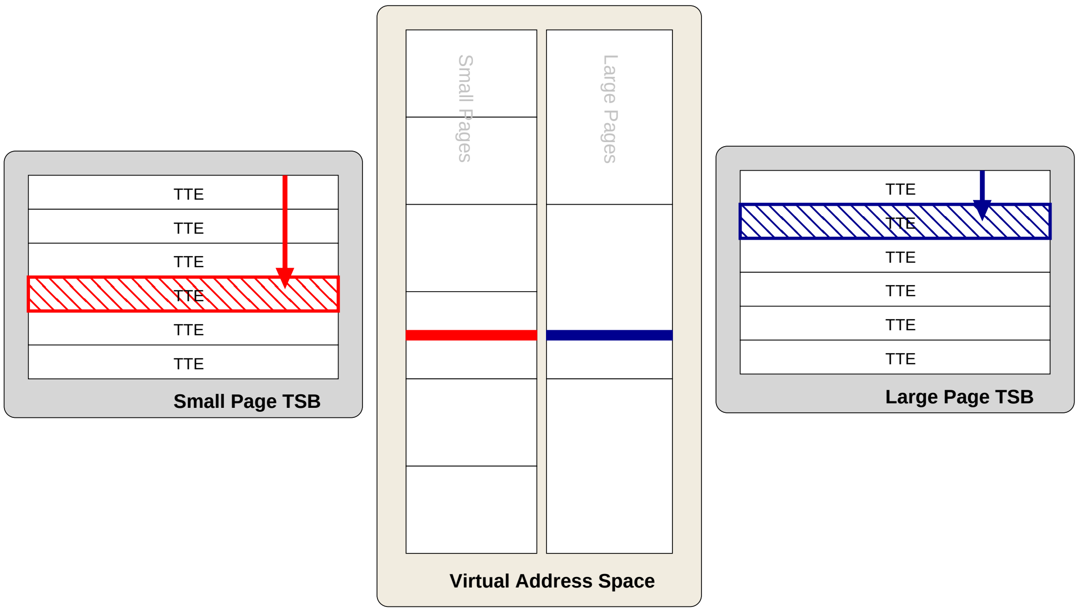

Figure 3.5: Illustration of TSB offsets for multiple page sizes

3.7 Power

Power usage and heat production have become a major consideration for data centre layouts; air conditioning costs can outweigh the cost of hardware.

UltraSPARC T1 uses only 70 watts of power, up to half of that of some competitors such as Intel's Xeon and Itanium. The power consumption is a major marketing point for UltraSPARC T1, which is being billed as a "green" environmentally friendly processor.

3.8 Open Source

One the the unique features of the UltraSPARC T1 is Sun's open sourcing of much of the logic behind the processor. VHDL designs are available for download and there is a community website. The open source implementation is referred to as the OpenSPARC T1.

During research for this paper we were able to make a number of suggestions and corrections which were accepted by Sun for release in official documentation. As systems implementors we applaud having more information available, and a direct feedback mechanism to people working on the processor!

Chapter 4

Conclusion

Sun's implementation of the UltraSPARC T1 is a radical departure from competitors architectures. A major question must lie in software's ability to keep the many threads busy; certainly the technique of reducing cache for greatly increased parallelism is currently unique.

Time will tell as to the success of this new paradigm. One can theorise that many workloads will take to a high level of parallelism well; web servers, client–server applications and some database workloads could be highly suitable. By Sun's own admission scientific workloads, especially anything relying on floating point performance, will not perform well on the current generation of processors. The greatly reduced power consumption is increasingly important is ever hotter datacentres, where airconditioning can be as greater cost as hardware.

We will watch the progress of this processor in the market with interest.

Bibliography

[1] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor's tale. pages 264–278, Anaheim, CA, USA, April 2005.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[3] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multi-threaded SPARC processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[4] Ana Sonia Leon, Jinuk Luke Shin, Kenway W. Tam, William Bryg, Francis Schumacher, Poonacha Kongetira, David Weisner, and Allan Strong. A power–efficient high–throughput 32–thread SPARC processor. In *IEEE International Solid State Circuit Conference*, 2006.

[5] Richard L. Sites. Alpha AXP architecture. *Digital Technical Journal of Digital Equipment Corporation*, 4(4):19–34, Fall 1992.

[6] Sun Microsystems. *UltraSPARC T1 Hypervisor API specification*, 2005. "http://opensparc-t1.sunsource.net/index.html".

[7] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.

[8] Adam Wiggins. The UltraSPARC III from architecture to implementation, October 2003. UNSW CS9244 - Part 2 Report.