

# **Advanced Computer Architecture**

**MCA - TMA**

**CS-12**

# Table of Contents

1. Tera Design Goals	1
1.1 Suitable for very high-speed implementations	1
1.2 Application for a wide spectrum of problems	2
1.3 Ease of compiler implementation	2
2. The Tera Multiprocessor and Sparse Three-Dimensional Torus	2
2.1. Super pipelined Support	3
3. Advantages of Tera Computer	4
4. Drawbacks of Tera Computer	4
5. Thread State and Management	5
6. Design strategies of UNIX for parallel computer	7
7. Design strategies of Mach for parallel computer	7
8. Design strategies of OSF/1 for parallel computer	8

**Q. 1. Answer the following questions on the development of the Tera Computer.**

**(a) What are the design goals of the Tera Computer?**

**Ans:** The Tera is very much a HEP descendant but is implemented with modern VLSI circuits and packaging technology. A 400-MHz clock is proposed for use in the Tera system, again with a maximum of 128 threads per processor.

Multithreaded von Neumann architecture can be traced back to the CDC 6600 manufactured in the mid-1960s. Multiple functional units in the 6600 CPU can execute different operations simultaneously using a score boarding control. The very first multithreaded multiprocessor was the Denelcor HEP designed by Burton Smith in 1978. The HEP was built with 16 processors driven by a 10-MHz clock, and each processor can execute 128 threads simultaneously.

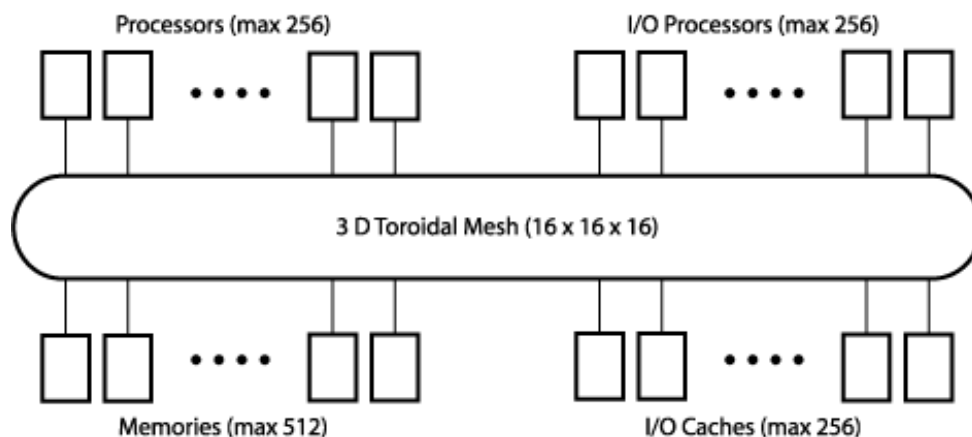
The Tera architecture features include not only the high degree of multithreading but also the explicit-dependence lookahead and the high degree of super pipelining in its processor-network-memory operations. These advanced features are mutually supportive. The first Tera machines are expected to appear in late 1993.

### **Tera Design Goals**

The Tera architecture was designed with several major goals in mind.

#### **Suitable for very high-speed implementations**

A maximum configuration of the first implementation of the architecture (Fig. 1) will have 256 processors, 512 memory units, 256 I/O processors, 4096 interconnection network nodes, and a clock period of less than 3 ns.



**Figure 1. The Tera Computer System**

### **Application for a wide spectrum of problems**

Programs that do not vectorize well, perhaps because of a preponderance of scalar operations or too frequent conditional branches, will execute efficiently as long as there is sufficient parallelism to keep the processors busy. Virtually any parallelism applicable in the total computational workload can be turned into speed, from operation level parallelism within program basic blocks to multi-user time and space sharing.

### **Ease of compiler implementation**

Although the instruction set does have a few unusual features, they do not seem to pose unduly difficult problems for the code generator. There are no register or memory addressing constraints and only three addressing modes. Condition code setting is consistent and orthogonal. Although the richness of the instruction set often allows several ways to do something, the variation in their relative costs as the execution environment changes tends to be small.

Because the architecture permits the free exchange of Spatial and temporal locality for parallelism, a highly optimizing compiler may work hard improving locality and tread the parallelism thereby saved for more speed. On the other hand, if there is sufficient parallelism, the compiler has a relatively easy job.

### **The Tera Multiprocessor and Sparse Three-Dimensional Torus**

The interconnection network is a three-dimensional sparsely populated torus of pipelined packet-switching nodes, each of which is linked to some of its neighbors. Each link can transport a packet-containing source and destination addresses, an operation, and 64 data bits in both directions simultaneously on every clock tick. Some of the nodes are also linked to resources, i.e., processors, data memory units, I/O processors, and I/O cache units.

Instead of locating the processors on one side of the network and the memories on the other, the resources are distributed more-or-less uniformly throughout the network. This permits data to be placed in memory units near the appropriate processor when possible and otherwise generally maximizes the distance between possibly interfering resources.

The interconnection network of one 256-processor Tera system contains 4096 nodes arranged in a  $16 \times 16 \times 16$  toroidal mesh; i.e., the mesh “wraps around” in all three dimensions. Of the 4096 nodes, 1280 are attached to the resources comprising 256 cache units and 256 I/O processors. The 2816 remaining nodes do not have resources attached but still provide message bandwidth.

To increase node performance, some of the links are missing. If the three directions are named x, y & z, then x-links and y-links are missing on alternate z-layers. This reduces the node degree from 6 to 4, or from 7 to 5, counting the resource link. In spite of its missing links, the bandwidth of the network is very large.

Any plane bisecting the network crosses at least 256 links, giving the network a data bisection bandwidth of one 64-bit data word per processor per tick in each direction. This bandwidth is needed to support shared-memory addressing in the event that all 256 processors are addressing memory on the other side of some bisecting plane simultaneously.

As the Tera architecture scales to larger numbers of processors  $p$ , the number of network nodes grows as  $p^{3/2}$  rather than as the  $p \log p$  associated with the more commonly used multistage networks. For example, a 1024-processor system would have 32,768 nodes. The reason for the overhead per processor of  $p^{1/2}$  instead of  $\log p$  stems from the fact that the system is limited by the speed of light.

One can argue that memory latency is fully masked by parallelism only when the number of messages being routed by the network is at least  $p \times l$ , where  $l$  is the (round-trip) latency. Since messages occupy volume, the network must have a volume proportional to  $p \times l$ ; since the speed of light is finite, the volume is also proportional to  $l^3$  and therefore  $l$  is proportional to  $p^{1/2}$  rather than  $\log p$ .

### **Super pipelined Support**

Each processor in Tera computer can execute multiple instruction streams (threads) simultaneously in each processor. In the current implementation, as few as 1 or as many as 128 program counters may be active at once. On every tick of the clock, the processor logic selects a thread that is ready to execute and allows it to issue its next instruction. Since instruction interpretation is completely pipelined by the processor and by the network and memories as well, a new instruction from a different thread may be issued during each tick without interfering with its predecessors.

When an instruction finishes, the thread to which it belongs becomes ready to execute the next instruction. As long as there are enough threads in the processor so that the average instruction latency is filled with instructions from other threads, the processor is being fully utilized.

If a thread were not allowed to issue its next instruction until the previous instruction is completed, then approximately 70 different threads would be required on each processor to hide the expected latency. The lookahead described later allows threads needed to achieve peak performance.

Three operations can be executed simultaneously per instruction per processor. The *M-Pipeline* is for memory-access operations, the *A-pipeline* is for arithmetic operations, and the *C-pipeline* is for control or arithmetic operations. The instructions are 64 bits wide. If more than one operation in an instruction specifies the same register or setting of condition codes, the priority is  $M > A > C$ .

It has been estimated that a peak speed of 1G operations per second can be achieved per processor if driven by a 333-MHz clock. However, a particular thread will not exceed about 100M operations per second because of interleaved execution. The processor pipeline is rather deep, about 70 ticks as compared with 8 ticks in the HEP pipeline.

**(b) Compare the advantages and potential drawbacks of Tera Computer.**

**Ans:**

### **Advantages of Tera Computer**

- The Tera uses multiple contexts to hide latency.
- The Tera machine performs a context switch every clock cycle.
- In the HEP/Tera approach, both pipeline latency (eight cycles) and memory latency are hidden.
- The major focus is on latency tolerance rather than latency reduction.
- With 128 contexts per processor, a large number (2k) of registers must be shared finely between threads.
- The thread creation must be very cheap (a few clock cycles).
- Tagged memory and registers with full/empty bits are used for synchronization.
- As long as there is plenty of parallelism in user programs to hide latency and plenty of compiler support, the performance is potentially very high.

### **Drawbacks of Tera Computer**

- The performance must be bad for limited parallelism, such as guaranteed low single-context performance.
- A large number of contexts (threads) demands lots of registers and other hardware resources which in turn implies higher cost and complexity.
- Limited focus on latency reduction and cacheing entails lots of slack parallelism to hide latency as well as lots of memory bandwidth; both require a higher cost for building the machine.

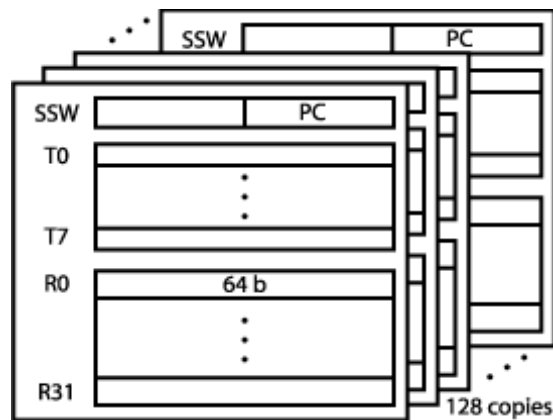
(c) Explain the thread state and management scheme used in Tera computer.

Ans:

### Thread State and Management

Fig. 2 shows that each thread has the following states associated with it:

- One 64-bit stream status word (SSW);
- Thirty-two-64-bit general-purpose registers (R0-R31).
- Eight 64-bit target registers (T0-T7).



Stream Status Word (SSW)

- 32 bit PC (Program Counter)
- Modes (e.g., rounding, lookahead disable)
- Trap disable mask (e.g., data alignment, overflow)
- Condition Codes (last four emitted)

No Synchronization bits on R1-R31

Target Registers (T0-T7) look like SSWs

**Figure 2. Thread Management Scheme used in Tera Computer**

Context switching is so rapid that the processor has no time to swap the processor-resident thread state. Instead, it has 128 of everything, i.e., 128 SSWs, 4096 general-purpose registers, and 1024 target registers. It is appropriate to compare these registers in both quantity and function to vector registers or words of caches in other architectures. In all three cases, the objective is to improve locality and avoid reloading data.

Program addresses are 32 bits in length. Each thread's current program counter is located in the lower half of its SSW. The upper half describes various modes (e.g., floating-point rounding, lookahead disable), the trap disable mask (e.g., data alignment, floating overflow), and the four most recently generated condition codes.

Most operations have a `_TEST` variant which emits a condition code, and branch operations can examine any subset of the last four condition codes emitted and branch appropriately. Also associated with each thread are thirty-two 64-bit general-purpose registers. Register R0 is special in that it reads as 0 and output to it is discarded. Otherwise, all general-purpose registers are identical.

The target registers are used as branch targets. The format of the target registers is identical to that of the SSW, though most control transfer operations use only the low 32 bits to determine a new PC. Separating the determination of the branch target address from the decision to branch allows the hardware to prefetch instructions at the branch targets, thus avoiding delay when the branch decision is made. Using target registers also makes branch operations smaller, resulting in tighter loops. There are also skip operations which obviate the need to set targets for short forward branches.

One target register (T0) points to the trap handler which is nominally an unprivileged program. When a trap occurs, the effect is as if a co-routine call to a T0 had been executed. This makes trap handling extremely lightweight and independent of the operating system. Trap handlers can be changed by the user to achieve specific trap capabilities and priorities without loss of efficiency.

**Q. 2. Make 2-3 pages summary report on design strategies of UNIX, Mach and OSF/1 for parallel computer.**

**Ans**

### **Design strategies of UNIX for parallel computer**

The UNIX functions are implemented with a large amount of system software, much too large to fit in its entirety in the main memory. The portion of the O.S. that resides continuously in the main memory is called the kernel and remaining components, such as the file management system and compilers for various languages, are stored on disk and brought into the main memory only when needed. The increasing role of UNIX provides more flexibility in customizing the kernel control in dedicated environments for real-time and transaction processing applications.

UNIX supports multitasking in which multiple processes within a program can run at the same time. In UNIX, the foreground tasks are executed at user processors through interactive keyboard/display operations performed by the users. The background tasks, such as mail-a-message and print-a-file, are handled simultaneously, overlapping with the foreground tasks. The presence of multiple processors should be hidden to provide portability and compatibility. These are two very important issues in porting UNIX on computer systems for parallel or distributed processing.

The kernel functions should be designed to be architecture-independent with the help of special hardware and user interfaces. UNIX for parallel computers should be compatible with some existing UNIX system such as Berkeley BSD or Bell Laboratory's System V, among others. Multiprocessor UNIX should be designed to explore all opportunities for parallelism. These include using multiple threads of control in the kernel, supporting fast inter-processor synchronizations and self-scheduling, and allowing multitasking or multithreading at various MIMD grain levels.

A multiprocessor UNIX should be able to support parallel I/O activities, share virtual memory in addition to sharing physical memory, facilitate better network messaging, allow distributed program debugging, and provide a large body of UNIX libraries and utilities for multiprocessing or multi-computing applications. These design goals will be reexamined as we proceed with the study of various UNIX extensions for multiprocessors and multi-computers.

### **Design strategies of Mach for parallel computer**

Mach designers introduced five program abstractions, task, thread, port, message, and memory objects. These have been improved from existing UNIX for centralized processing to parallel and distributed computing.

The Mach kernel contains essentially three basic service functions: processor scheduling, inter-process communication, and virtual memory management. Conventional UNIX has been upgraded in Mach to support parallel processing on multiprocessors and multi-computers. Mach design support for both tightly coupled multiprocessors and loosely coupled multi-computers by separating the process abstraction into tasks and threads, with the ability to execute multiple threads within a task simultaneously.

Mach design has the capability based inter-process communication facility that transparently extends across network boundaries and integrates with a virtual memory system to allow the transfer of large amounts of data via copy-on-write techniques.

A task includes protected access and control of all system resources, including CPUs, physical I/O ports, and either virtual or real memory. A thread is the basic unit of CPU utilization. It is equivalent to a program stream with an independent program counter operating within a task. A task may have multiple threads with all threads under the same task-sharing capabilities and resources. A thread can also be treated as an object capable of performing computations with low overhead and minimal state representation.

Mach's virtual memory system makes few assumptions about the underlying machine architecture, which has facilitated porting of the Mach system to various machines. Mach needs no in-memory hardware-defined data structure to manage the virtual memory. The multiprocessor system also adds to the difficulty of connecting a universal model of virtual memory.

In addition to address translation hardware, multiprocessors also differ in the kind of shared-memory access they provide for individual CPUs. Mach had been ported only to multiprocessors with uniform or non-uniform shared memory. By taking advantage of its elaborate message-passing facility, Mach is able to integrate loosely coupled multi-computer systems. The impact of Mach on UNIX systems is increasing steadily in the parallel processing community.

### **Design strategies of OSF/1 for parallel computer**

The OSF/1 is an operating system compliant with standards, is compatible with other versions of UNIX, and contains some innovative features. OSF/1's innovation involving the kernel comes primarily from its Mach-based technology and the provision of loadable device drivers and streams modules while the operating system is running.

The Open Software Foundation's OSF/1 kernel is a highly multithreaded version of the Mach/OS. The long-term goal was to return to UNIX's original small, compact kernel. The OSF/1 includes multiple file systems, POSIX compliance, a program loader, a logical volume manager, streams, and C2/B1 security.

The OSF/1 kernel includes an internal file system switch to facilitate using multiple file systems. The program loader is a new feature developed in OSF/1. It extends from the object file format, has a format-independent design, and supports a shared library. The loader actually runs primarily in user space. The loader is mapped into the processor's virtual space and the loader then maps the application program into the virtual memory.

In OSF/1, the kernel software for handling a stream consists of a driver and one or more modules. The application creates a specific driver for a stream via the open system call and returns with a standard UNIX file descriptor. Data is passed between streams modules in messages. The OSF/1 uses a thread for a compute-intensive module.

Multiple streams are handled by multiple threads on the symmetric multiprocessor system. When multiple streams modules are involved, messages are exchanged in message queues. A set of queues can be associated with every module. An awakened streams thread calls the module's service procedure, which processes the message. It then calls

the next module's put call. The last module called returns to the streams head and then returns to the application program.

The OSF/1 logical volume manager is a mechanism not only for addressing a problem with traditional UNIX files, which can only span the size of a single disk volume but also addressing files much larger than the physical disk. A modular approach is adopted. Internally, the kernel invokes a security policy module for handling the application's system call. This module interrogates a policy database on a user-mode program.