Symmetric Multi-Processing (SMP) systems on top of contemporary Intel appliances

Jiri Hlusi

University of Tampere
Department of Computer and
Information Sciences
Pro gradu
December 2002

University of Tampere
Department of Computer and Information Sciences
HLUSI, JIRI: SMP systems on top of contemporary Intel appliances
Pro gradu, 50 pages, 2 appendices
December 2002

Abstract

For more than two decades, the Intel Corporation has been recognized as a leading company in the manufacture of high-quality integrated circuits, most remarkably the 80x86 family CPUs for personal computers. Despite occasional problems in design, one has to admit that on its way, Intel typically has managed to keep a reasonable level of balance between the need for the "features of tomorrow" and the pressure for backward compatibility in the same time. As a result, some of the software products are even today denoted as "i386 ready", indicating that there actually exists a wide range of processors where a given binary image can be executed.

As components for mainly home and office computers, Intel CPUs were for a long time regarded as rather lower-class siblings of their competitors, like SPARC, Alpha, PowerPC and PA-RISC, for instance. This was partly also due to the fact that until recently the IT industry was lacking a reliable multi-user and multi-tasking operating systems for the 'i386 compatibles'.

During the mid-nineties, the world changed significantly. With Microsoft introducing several of their server-type platforms, and Linux practically acquiring the Internet as its de-facto home, the chance for Intel to finally penetrate the server market arrived. In a way this is not a big surprise, since a server is, from a rather naive perspective, simply a computer running specific type of the operating system. On the other hand, addressing the needs for the *vertical scalability*[1] of a particular hardware platform raises a question whether (and to what extent) Intel CPUs are suitable for this purpose and what the constraints to be aware of are. This is the focus of this thesis study.

[1] Increasing the server processing power by adding CPUs.

Table of Contents

List of Figures

List of Tables

Acknowledgement

My respect and honest appreciation belong to Mr.Roope Raisamo and Mr.Arto Viitanen of the University of Tampere, Mr.Petri Soukkio of Nokia Networks and Mr.Evžen Mayer who all contributed to this work in their own and unique ways. Thank you all!

Keywords: Intel, IA-64 Architecture, Symmetric Multi-Processing

1. Introduction

This thesis work focuses on essential attributes of bringing the vertical scalability into generic computing platforms. The main subject is then the design of *Symmetric Multi-Processing* (SMP) systems as the most common approach today.

As a foundation for demonstrating the theoretical apparatus with real-life examples, the contemporary Intel appliances are used. This has two fundamental reasons: First, already for many years, the Intel 32-bit processors are used as the heart of Linux- or Windows-powered server systems. Taking this as given, it might be a good time to ask whether this is only an attempt to compete with other traditional UNIX server platforms despite potential technical difficulties, or whether the underlying IA-32 foundation is mature enough to be used for the SMP purpose[2].

The second reason for studying the Intel portfolio in particular is the fact that with recent announcements of the IA-64 architecture also the marketing pressure for taking the first two commercially available implementations, the 'Merced' and 'McKinley' processors, into the use grows. The question is, however, whether the shift into the world of 64-bit computing is necessary, or alternatively, what are the optimal areas of application the new processors are designed for. Surprisingly enough, this question is left unanswered in many available materials as a somewhat irrelevant burden, mainly because today the 64-bit processors are presented (and promoted) as a natural evolution and the next logical step ahead in designing the computing systems. In other words, that there is no special reason to question the IA-64 platform, because "the future simply comes". Naturally, for those who really need to understand the possible IA-64 advantages, such an explanation isn't satisfactory.

Yet, it is needless to point out that Intel CPUs are not the only ones available on the market. In the same way as engineers in Santa Clara[3], other research teams around the world try to tackle the problem of computing efficiency in their own ways. Just to mention few of them, MIPS Technologies [MIPS] is successfully running their business with a variety of 32- and 64-bit processor cores typically licensed to other companies for further use. Similarly, SUN Microsystems, for instance, is instantly developing their SPARC family of CPUs since the early 90's, including the UltraSPARC 64-bit CPU family [SPARC]. Their systems are, however, rather proprietary and also typically tightly

[2] Meaning mainly the server-type of solutions.

[3] Official Intel headquarter: Santa Clara, California, U.S.A.

coupled with the Solaris™ operating system. Another example of CPUs competing with Intel products might be the PA-RISC processor family [PA-RISC], developed by HP, delivering a 64-bit performance as of the first PA-8000 CPU introduction (1996). However, also in this case no other operating system than the HP-UX is widely used on PA-RISC servers and workstations. Finally, returning to the world of IA-32 compatibles, one shouldn't forget about a company called AMD, whose portfolio [AMD_CPUs] is in many aspects overlapping with the Intel offerings, hence creating a competitive market by challenging its rivals.

Thus, there exists a variety of processors (many of them exploiting the RISC paradigm), which are used for designing the SMP systems. Yet, for the purpose of this thesis the Intel portfolio is the focal point in order to stay close to the newly emerging IA-64 architecture and the question of its applicability in the real life.

This study is primarily intended to elaborate around the contemporary Intel offerings for the SMP market. However, in some ways the reader might wonder why to question or study something, which seems to be quite well justified by the recent acceptance of the Linux operating system in the IT industry, for instance. The implication is that if Linux is "good enough" as a server-type operating system, the underlying hardware must also be suitable for the same purpose.

Not necessarily this is the case. Firstly, dozens of Linux adaptations exist around the world, where some of them are quite different on the lower parts of the kernel, while still preserving the required user-level API. Secondly, Linux is not an equivalent to 'server', and vice versa. In other words, the fact that a given operating system can run on selected hardware only means that the adaptation is possible. By no means does that, however, imply that the adaptation is done in an optimal way or that the underlying platform has been designed with multi-processing in mind. This makes the initial question (i.e. Intel CPUs in SMP systems) research worthy, even though some of the surrounding evidences might suggest the opposite.

In this thesis, the contemporary Intel portfolio is studied from the perspective of *multi-processor* (MP) systems. While doing so, in *Chapter 2,* the basic concepts of multi-processing systems are outlined. Following that, in *Chapter 3* and *Chapter 4* the currently available Intel products are examined in terms of its architectural principles and other technical parameters. Finally, in *Chapter 5* the collected findings are combined into a kind of "CPU model suggestion chart" as an outcome of the entire study.

2. Basic introduction into multi-processor architectures

In this section the theoretical aspects of the *Multi-Processor* (MP) design are introduced, as well as the general reasoning why the parallel-execution environments are more suitable for the server type of applications.

2.1. Motivation behind multi-processor design

As with almost all the computing system dimensioning attributes, the demands for the *processing power per unit* grow every now and then. Depending on what is to be understood by the term 'unit', the overall execution capabilities in a given computing system can be enhanced in various ways:

1. increasing CPU speed and efficiency,
2. enhancing the CPU programming model (new instructions, more registers, etc.), and
3. teaming individual CPUs.

While the first two options fall into the category of on-chip improvements, the third can be described as an aggregation of the available processing power across multiple CPUs. Typically we see a combination of all of the above options taking place as the processors evolve over time. In the server market, however, the "CPU teaming" approach is used as the most affordable alternative mainly due to the following reasons:

- the CPU chip space is restricted,
- the CPU core clock rate cannot increase in an unlimited way,
- on-chip changes are costly,
- architectural stability is required (enhancing the CPU programming model requires a certain learning effort among the programmers and system integrators),
- multi-CPU systems are more fault tolerant (a faulty CPU in a mission-critical system can be suspended without shutting down the entire system), and
- products built on top of mass-market products are more cost efficient.

Hence, there seem to be good reasons for combining several CPUs into harmonized blocks, rather than relying solely on the improvements inside the CPUs itself.

2.2. Taxonomy of computing platforms

Based on the way a given computing platform is designed, it falls into one of the categories presented in *Figure 2.1*.

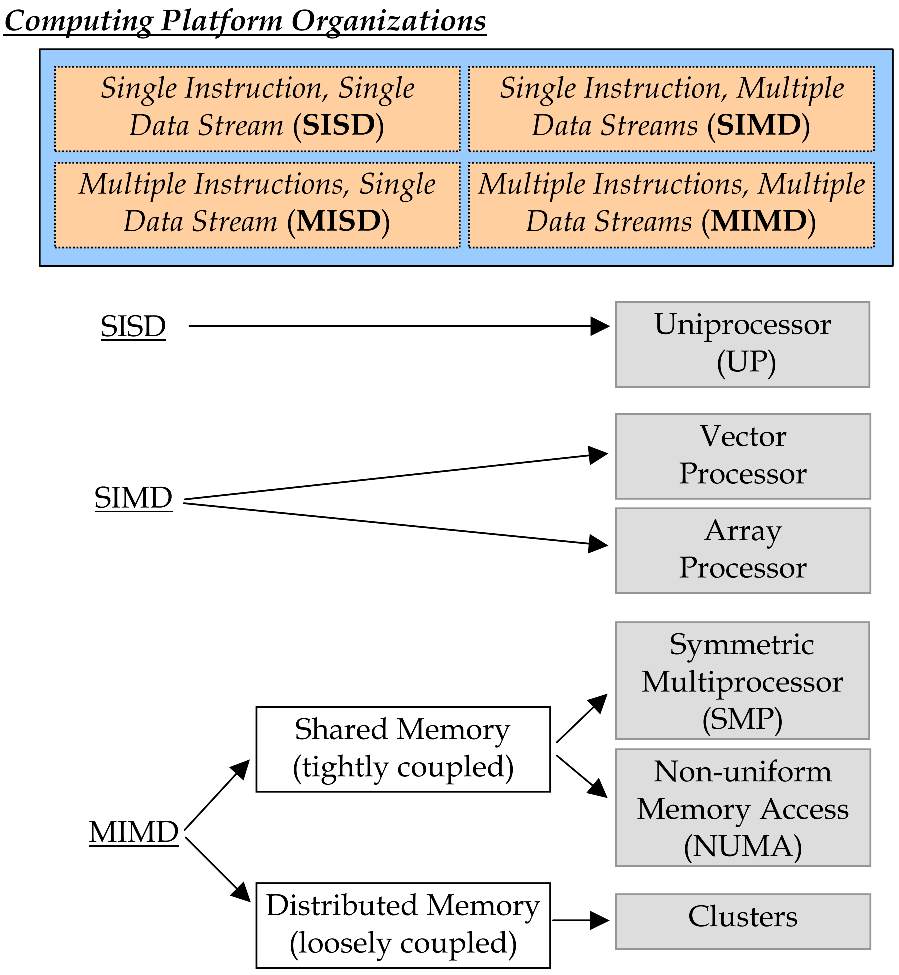

Computing Platform Organizations
Single Instruction, Single Data Stream (SISD)
Single Instruction, Multiple Data Streams (SIMD)
Multiple Instructions, Single Data Stream (MISD)
Multiple Instructions, Multiple Data Streams (MIMD)
SISD
Uniprocessor (UP)
SIMD
Vector Processor
Array Processor
MIMD
Shared Memory (tightly coupled)
Distributed Memory (loosely coupled)
Symmetric Multiprocessor (SMP)
Non-uniform Memory Access (NUMA)
Clusters

Figure 2.1: *Taxonomy of parallel processor architectures*

As argued in the previous chapter, in designing standalone servers, the most commonly the *shared memory* MIMD paradigm is used [COMP_ARCH]. This is also a subject of this thesis. In contrary, the question of computing clusters is left aside, since in this case the solution maturity depends significantly on the quality of the used clustering software, rather than on the used hardware components.

2.3. Hardware-related aspects of multi-processor design

In this chapter (mainly based on ideas presented in [COMP_ARCH]) a brief introduction into hardware-related aspects of the design of multi-processor (MP) systems is presented. The goal is to provide a broader view into multi-CPU architectures as well as an understanding of what is generally known as the *Symmetric Multi-Processing* (SMP)

2.3.1. CPU teaming

Whenever a *tightly coupled* MP system is about to be constructed, several agreements have to be made. First of all, participating CPUs should employ the same way of signal encoding, the same mode of triggering external actions (e.g. edge-triggered memory access) and the same mode of constructing individual data units (e.g. little-endian). This is due to the fact that the surrounding chipsets are designed for high-speed operations and there is often no time for additional data translation whatsoever. Last, but not least, CPUs should also be ready to wait if the accessed peripheral is busy.

A somewhat specific issue is the cache memory size. As this is by definition transparent for the programmer, it is up to the CPU itself how the *cache coherence* is achieved among several processors. Consequently, vendor-specific recommendations have to be followed when designing any MP system[4]. In any case, neither the operating system nor any other software component should make firm assumptions on the CPU internals. Instead, the critical CPU attributes should be detected if necessary (especially the delay calibrating constants, like BogoMIPS, etc.).

A commonly discussed topic is also the distribution of the CLK signal, which is the “heart beat” of each CPU. Here a common practice is to use the same clock signal (i.e. signal derived from the same source) for all components within an individual domain as a solution for reducing the noise on interconnection buses. On the other hand, this doesn’t yet guarantee that the execution is perfectly synchronous on all CPUs as there are many reasons why this might not be true (e.g. external interrupts). Therefore, no explicit assumptions should be made unless the computing system in question is designed as a real-time solution from the beginning.

No MP system should be designed without a clearly defined booting sequence. This is due to the fact that when a CPU is powered on, the content of the external (shared) memory might not be initialized for a parallel execution. Therefore most of the MP systems first boot the CPU0 (determined and

[4] Typically CPUs with exactly the same cache structure are required.

controlled by the motherboard), while other CPUs are activated by the operating system at some point. Alternatively, on some systems a special *system guardian processor* might be used for controlling all main CPUs.

2.3.2. Memory subsystem

The external memory is a critical resource for each CPU. Depending on the design goal, the provided *memory subsystem* can be either dedicated to a certain CPU or shared between several processors. While the former option is more common in small, special-purpose appliances (e.g. communication processors with local buffers), the latter one seems to be more suitable for universal type of servers, where the required amount of RAM per each CPU cannot be easily estimated in advance.

When dealing with *shared* memory models, designers have to decide whether all the CPUs will be given access to the same range of physical addresses or not[5]. In practice, this can be widely affected by the way each of the CPUs is physically attached to the common bus. Nevertheless, the most common case is that all the CPUs can address the same range of external memory, and that the same logical-to-physical address mapping is used across the entire computing platform. Such systems are then denoted as systems with *symmetric memory access* emphasizing that other options exist as well.

Finally, depending on the memory response time for different request initiators, systems with similar latencies across the whole memory range are known as the *uniform memory access* systems, while systems exposing variations are known as *non-uniform memory access* (NUMA) systems. From the designer's perspective, the former ones have typically all the memory chips assigned to a common inter-CPU bus, while the latter ones make use of various bus converters (e.g. PCI transparent bridge), and even switching technologies (e.g. InfiniBand), in order to logically combine several memory subsystems into a commonly addressed workspace. *Figure 2.2* illustrates the conceptual difference between both the UMA and NUMA systems:

[5] Note that since the memory chips are in fact peripherals utilized by the surrounding CPUs, neither the actual memory size nor the addressing schema is in a direct relation to the internal processor architecture.

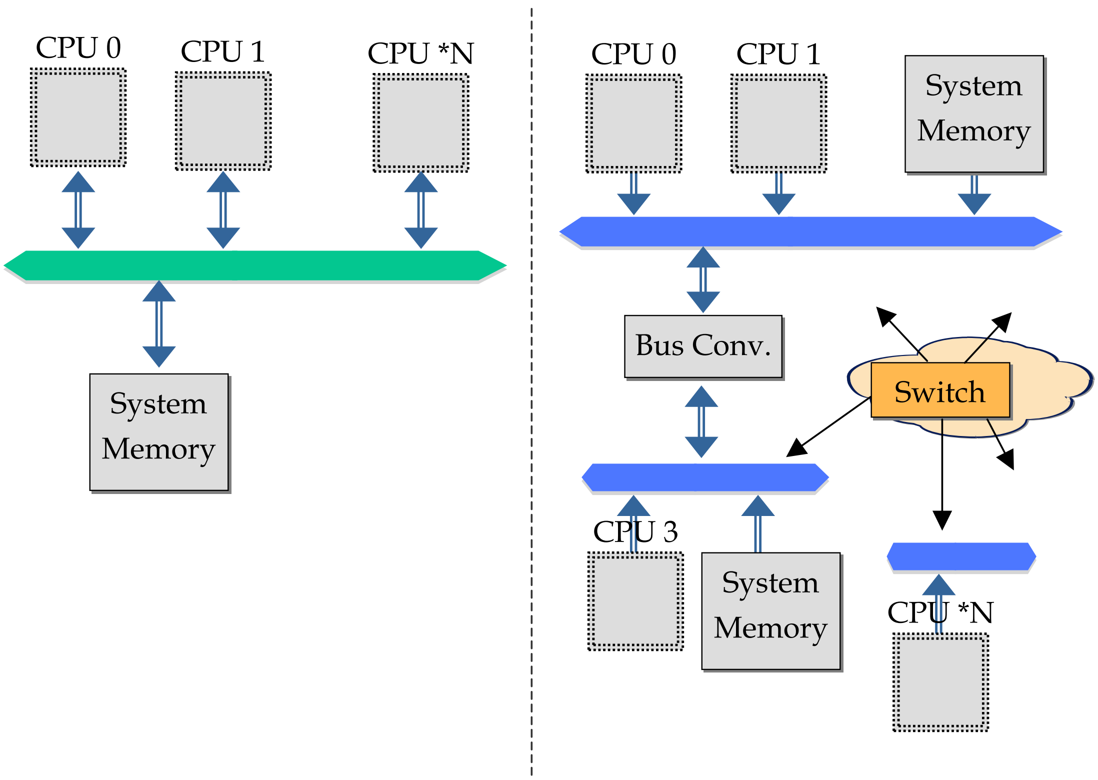


Figure 2.2: *UMA vs. NUMA systems*

Note that despite certain complexities of the NUMA model, it also brings advantages in comparison to the rather straightforward single-bus alternative. Firstly, since there exists an embedded concept of the independent capacity expansion, the actual shared buses can be rather short, which is necessary for preserving the signal quality at high frequencies. Secondly, by using switching techniques the system can be transformed into the networked architecture and expanded practically anytime. This is especially important for mission critical systems where the system downtime is seldom tolerated. Finally, by applying filtering rules between individual buses, the entire computing platform can be re-configured into several isolated zones, possibly running even a different type of operating system if necessary.

In either case, however, both models are equivalent from the programmer's perspective, meaning that there is no difference in the instruction stream should *UMA* or *NUMA* memory subsystem be addressed.

2.3.3. System peripherals

From the CPU perspective, *peripherals* represent another types of addressable objects (targets), either dedicated or shared between several CPUs. This scenario is similar to the concept of *memory subsystem*, except that peripherals can typically serve only a limited amount of simultaneous I/O sessions (if not limited single session only). Therefore, while accessing them, a controlled concurrency solution is needed, most commonly implemented using exclusive or shared locks (more information in *Section 2.4.3*).

Ensuring an exclusive access for a given peripheral isn't, however, the only issue to take care of, because there exist several scenarios for how the actual *I/O transaction* can be performed. As we shall see in the following sections, an individual peripheral device can be treated as a serial device, parallel device or as an independent transfer initiator. Furthermore, since the same data and address bus is typically used for addressing both the memory and peripheral subsystems[6], the influence of the CPU-local cache memory has to also be evaluated case by case.

The Direct Port-I/O Access

The mode of the *Direct Port I/O* can be described as a serial type of transmission where, at most, a single CPU word[7] is transferred at a time. A typical transaction is composed of the following steps:

Example of Port Input:

1. CPU places the address of the target device into the address bus.
2. CPU indicates that this is an input operation by defining the value of the *RD,/WR* signal.
3. CPU confirms the validity of the address by asserting special control line called *strobe* (/STRB).
4. Upon receiving the "address valid signal", each attached controller (device) compares the received address with its own, while the one who finds the match takes further actions.
5. The addressed device (target) writes one CPU word to the shared data bus and confirms its validity by asserting a specific signal line.

[6] Using the same external buses for handling memory and peripherals is a more economic solution concerning the amount of necessary motherboard connections and the overall I/O subsystem complexity.

[7] i.e. as many bits as the data bus width.

6. CPU reads the data from the data bus and once ready releases the "address valid signal". As of that moment, the I/O cycle is completed and every peripheral device is expected to remove any of its signals from the control and data bus.

Even though conceptually straightforward, the entire *Direct Port I/O* sequence is rather long and therefore also impractical for large data blocks. Furthermore, due to the required *in-order-delivery*, the CPU is not allowed to apply any techniques of speculative execution that might eventually cause the I/O operations to happen in an order different from what the programmer expects.

While performing the *Direct Port I/O*, the CPU-local data caches are assumed to be inactive, since efficient hardware-driven cache synchronization is in fact unfeasible between CPUs and serial devices. In practice this is achieved using special instructions for performing the Direct Port I/O.

From the Multi-Processing point of view, the *Direct Port I/O* causes minimal problems, because during the I/O instruction execution an exclusive access to the shared buses has to be granted to the initiating CPU. This also ensures that each elementary I/O transfer will be atomic by nature, and therefore no further synchronization technique is necessary. On the other hand, such a guarantee doesn't exist for any transaction composed of two or more I/O operations. Therefore in most cases some type of OS-originated synchronization is necessary anyway.

Memory-mapped I/O

The *memory-mapped I/O* mode tries to overcome the problem of low resource utilization, in the Direct Port I/O, by assuming that within a typical CPU-to-device transaction a relatively larger amount of data is exchanged before any real action is taken. Therefore, the principle of the in-order-delivery doesn't have to be obeyed all the time, in particular, before the "commit" operation happens.

The underlying mechanics behind the memory-mapped I/O concept is that the end device is active on a whole sub-range of memory addresses, rather than allocating a single I/O address only. As soon as any data is written into this address range, it is captured by the device's bus interface (e.g. latch registers) and eventually forwarded and stored in some sort of local buffers, around the same way as any ordinary memory chip would do. Similarly, should a *read* operation be initiated, the end-device's logic fetches or constructs the data and propagates it into the data bus. The actual order of transferring the communicated information is not essential, except that all the pending I/Os have to be completed before the entire transaction is committed by the peripheral device itself. This approach has the following advantages:

- As the in-order-delivery is not required all the time, the CPU can take full advantage of speculative execution and parallel processing (in fact, the CPU assumes that the target location is memory).
- The CPU cache can be involved in the normal way as long as there exists a possibility of flushing all the dirty data in a later phase of the transaction.

At the same time, it is, however, important to keep in mind the following issues:

- Treated as a memory area, the interface circuits and the device-internal buffers (if used) have to be of a comparable speed as the real memory chips are.
- By deploying the memory-mapped I/O, the originally continuous memory address space becomes fragmented into individual memory areas served by their respective owners. Furthermore, "empty holes" (i.e. address ranges not served by any peripheral device nor the system memory) might exist as well (see *Figure 2.3* for details).
- Even though addressed as a memory block, the internal semantics of the read and write operation might not necessarily be identical. In other words, the data being read might differ from that which was written earlier.

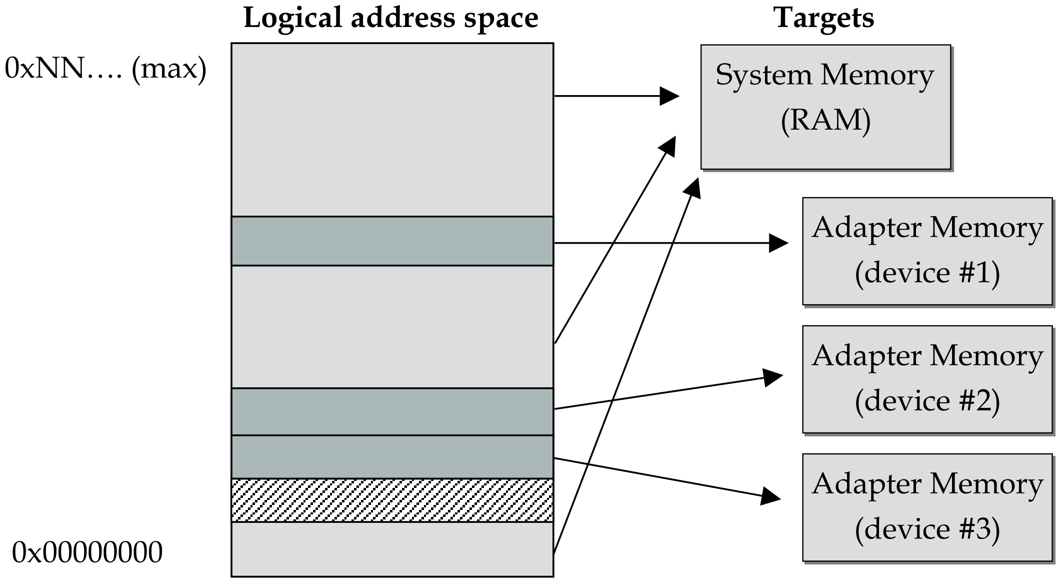

Logical address space
Targets
0xNN…. (max)
0x00000000
System Memory (RAM)
Adapter Memory (device #1)
Adapter Memory (device #2)
Adapter Memory (device #3)

Figure 2.3: *Memory fragmentation in case of memory-mapped I/O*

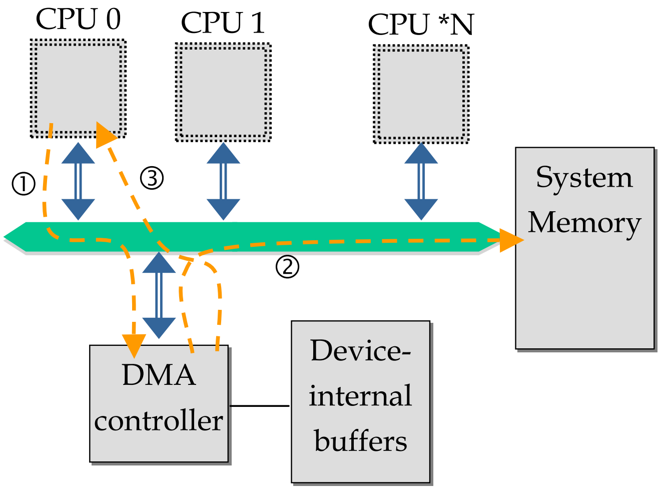
As a conclusion, what we see is a transaction model suitable for bulk data transfer, in which the data amount clearly exceeds the frequency of individual actions.

Finally, it is needless to say, there also exist devices, which operate in a rather continuous way, hence not requiring an explicit "commit" indication after each buffer transfer. One such class of devices are various types of display adapters, typically allowing the entire screen buffer (or its portions) to be modified directly at the speed comparable to the standard memory access. Certainly, for such devices, the *Direct Port I/O* approach wouldn't be satisfactory.

Direct Memory Access (DMA)

Even though the memory-mapped I/O is a significant step ahead, on the way between its source and destination, the data has to traverse the system bus twice. For example, when reading a disk I/O block, the CPU has to *read* the data from the memory buffer provided by the respective disk drive controller, and *write* the data later into the pre-defined memory location. This results into two I/O transfer per each transferred unit.

In order to tackle this efficiency problem, the *Direct Memory Access* (DMA) mode of operation has been developed. The leading idea here is that a DMA-capable device only receives information about where in the system memory a particular data block should be stored (or alternatively read from) and the device will carry the transfer independently from the CPU. *Figure 2.3* depicts this situation:



Stages:

1. Transfer initiated from the CPU side
2. DMA controller takes the burden
3. CPU is informed about the transaction result

Figure 2.4: *Example of the DMA read operation*

From the system bus perspective, as well as from the *Memory Subsystem* standpoint, both the CPU and the peripheral device in question are considered to be the transaction initiators, and are therefore treated equally. On the other hand, since the DMA controllers do not participate as regular CPUs in the multi-processor system, those are given a subordinate role only, meaning that they typically do not perform any data transfer unless instructed to do so.

2.3.4. Interrupt subsystem

The interrupt subsystem was originally developed as a primary way for the peripheral devices to attract attention whenever necessary. Because in earlier times no generic frame protocols for passing data between peripherals and the main CPU had been developed, a set of dedicated wires was used for arranging the necessary signaling. In a UP system, the interrupt subsystem mechanics is as follows:

1. When a given peripheral has a need for communicating with the main CPU, it generates an *Interrupt Request* (IRQ) by asserting one of the available IRQ lines.
2. The signal is propagated to the *Interrupt Controller*, which solves the priority issue, ignores certain IRQs if configured so, and finally forwards the request to the main CPU (again using dedicated signal lines in early times).
3. The main CPU interrupts the current task at the first possible occasion and executes the corresponding *interrupt handler routine* in order to satisfy the peripheral device's needs.
4. As one of the last operations, the interrupt handler routine will typically indicate to the interrupt controller that the IRQ request was served and hence another one can be accepted.
5. Finally, the main CPU resumes the task it was originally executing.

As mentioned in *Step 2*, according to the severity of the event, different priority levels can be assigned to the individual IRQ sources. The most common rule here then is that once a certain IRQ is initiated, all other requests with lower priority are placed on-hold, while higher priority requests can still be passed through, possibly interrupting the course of execution of an already active IRQ handler.

Taking a closer look at the design of MP systems, the above-described way of handling interrupt requests is not sufficient. First of all, in multi-CPU systems a certain IRQ distribution policy has to be applied if there are several CPUs to handle those. While doing so, one of the possibilities is to direct all the incoming IRQs towards a pre-defined CPU only (i.e. the "I/O processor"). Alternatively, one might also decide to establish a static mapping between a particular IRQ and its target CPU. The most sophisticated approach is a dynamic signal routing between all the CPUs based on the recent event flow pattern in the system. These advanced techniques, however, call for additional intelligence inside the IRQ subsystem on itself.

Another issue to be aware of in the MP systems is the fact that not only peripherals, but also individual CPUs might need to interrupt each other in order to solve emergency situations. Therefore the CPUs used for building MP systems should support some sort of *Inter-Processor Interrupt* (IPI) interface, allowing short messages to be routed between individual CPUs or eventually broadcasted to all of them. In practice, this imposes yet another requirement on the *interrupt controller* itself. In the later chapters we will see how this problem is solved with the Intel chipset.

2.3.5. Hardware model assumed by the Linux SMP kernel

For any widely applicable (i.e. portable) operating systems, certain realistic assumptions have to be made regarding the underlying hardware architecture in order to address most of the target platforms. In case of the Linux kernel the contemporary implementations typically make the following assumptions [Linux Kernel]:

- all CPUs have the same internal architecture,
- all CPUs have equal chances to access the underlying memory and peripherals, both in terms of the addressing and the throughput,
- all CPUs are capable of receiving all IRQ requests if configured so, and
- inter-CPU messages cover all involved CPUs and are passed along with other IRQ requests.

Especially the assumption on equal chances makes the system symmetric from the kernel perspective, and therefore the entire subsystem is often referred to as the *Symmetric Multi-Processing* (SMP).

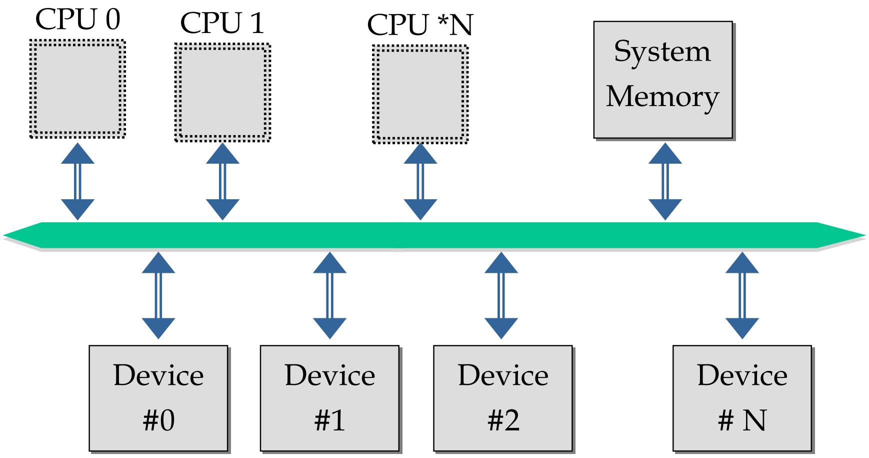


Figure 2.5: *Logical model of SMP systems*

Note that even though the system is symmetric from the logical point of view, by employing various transparent switching and bridging techniques the actual "hardware reality" might be different. On the other hand, the experience shows that in most cases it is an easier job to take care of the low-level signal replication rather than implementing various "case branches" in the kernel (e.g. 'CPU6 cannot address more than 5GB'). Therefore the logical model presented above is sufficient for most of the SMP systems.

2.4. Software and operating system related issues

In the previous sections mainly hardware-related aspects of teaming independent processors and peripherals together were explained. In contrast, in this section we explore what other assumptions and prerequisites are imposed by multi-tasking operating systems, and what additional implications it brings regarding the type of the CPUs to be used [IA-64; sections on the Linux kernel].

2.4.1. Concept of Virtual Memory

In a typical general-purpose operating system, each of the user-level processes is provided with an illusion of running in a dedicated sandbox environment, with all the required resources it needs (exclusive locks, network sockets, files, runtime data, etc.).

As long as both the resource allocation and its usage happens via the provided kernel API, an efficient way of process isolation and resource management can be implemented at the operating system level. The situation is, however, different when it comes to the *System Memory*, or let's say, the memory assigned to a given process. When seen as a resource, the system memory is somewhat specific because it can be accessed directly, for instance by using instructions like ‘`mov reg,[addr]`’¨. As the kernel API is bypassed in this case, additional help is needed from the CPU itself in order to implement efficient process isolation. This results in the idea of *Virtual Memory.*

The Virtual Memory concept is based on an assumption that each user-level process is given a fictive memory space, which is as large as the addressing model of a given CPU will allow. For instance, in case of a 32bit CPU, each process can, at least in theory, address up to 4GB of RAM starting at the location 0x00000000 and expanding all the way up to, and including, the logical address 0xffffffff. On the other hand, what the process is not aware of is that at the CPU level the actual *logical address* is translated into the corresponding *physical address*, which in most cases differs from the original one. However, since the same address mapping is applied for both the *read* and *write* operations, the entire address translation is transparent for the user-level applications, and therefore no additional adjustments in the program code are necessary. On the other hand, for the operating system, the concept of *Virtual Memory* is a key factor for treating the physical memory (RAM) as yet another resource with firmly defined access and sharing policies.

The address translation can be implemented entirely on the software level or partly with the hardware assistance. While doing so, the entire logical and physical address space is divided into bigger allocation units known as ‘*memory page frames*’.

Memory page frames are of a fixed size (e.g. 4kB; an integer power of 2), and depending on the address origin they are called *logical* (belonging to the virtual address space) and *physical* (parts of the physical RAM) frames respectively. As soon as the frame size is defined, it is then a task for the operating system to provide the mapping information between the physical and logical frame IDs whenever necessary. In reality, this is done using an optimized piece of the kernel code.

Obtaining the mapping information in the above-described way is, however, a time consuming process. Therefore, most of the CPUs make use of the so-called *Transaction Lookaside Buffer* (TLB), which can be seen as a CPU-local temporary storage for the collected frame mapping information. If taken into use, the OS-level *virtual memory handler* is invoked only in case the required mapping information is not found in the TLB buffer. This situation is commonly denoted as the “*TLB miss*”. As a result, the frequency of the operating system calls can be significantly reduced, depending on the frame size and the size of the TLB buffer itself. *Figure 2.6* describes the final memory mapping process:

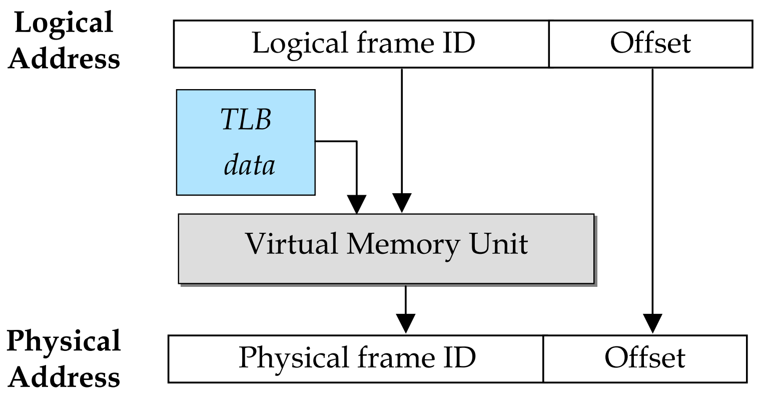


Figure 2.6: *High-level schema of virtual-to-physical memory address mapping*

On some CPUs the "*TLB miss*" situation can be solved by means of the hardware-level assistance, which effectively postpones the need for to invoke an OS-level virtual memory handler till the latest possible moment. The implementation is, however, subject to a given CPU type and therefore we will discuss this option in the respective chapters later.

Typically a single virtual memory mapping schema exists per each user-level process in the system. This approach brings the following advantages:

- Individual user processes can be completely isolated from each other.
- By introducing "security bits", memory frames maintained by the kernel or other processes can be shared as write-protected virtual memory areas.
- Memory frames can also be shared with full access if necessary.
- As the *Virtual Memory Unit* is involved in every address translation, it can leave traces on what data blocks are used frequently and what have not yet been accessed. This is a key issue for efficient implementation of more advanced concepts like *paging* and *swapping.*

The concept of virtual memory is a vital part of every multi-tasking system and therefore it is required from every CPU in the SMP system.

2.4.2. Context switching

In a multi-user environment, our goal is to run several tasks in a parallel fashion allowing a high number of users to be served simultaneously. While this wouldn't be a problem if the number of CPUs was greater or equal to the number of active processes, in reality we often have to run a higher number of processes on a modest number of CPUs only. This naturally calls for the concept of *context switching* as a way of emulating a parallel execution.

In simple words, the context switching means that whenever a currently active thread is about to be temporarily suspended, its *context information* is stored in a safe place, while similar context information from a previously deactivated thread is uploaded to the CPU. At the final stage the new thread execution is resumed. In this way, the currently active thread and one of the inactive threads swap roles. In a wider context, if such a change takes place regularly within a short period of time (e.g. 20ms), the user is given an impression that the execution happens in parallel.

Context switching is typically triggered by the *process scheduler*, which based on the status and management information selects processes from an appropriate execution queue for each CPU. As this is a rather generic task, it is also platform independent. On the other hand, on the bottom line certain support from the actual CPU is necessary in either case, typically in a form of the relevant instructions and the necessary ‘know how’.

From the system performance viewpoint, the faster the context switching can be achieved, the more CPU time is given to the ‘real work’. Therefore, when selecting a CPU model for multi-threaded systems attention should be paid to the ease of context switching as well unless we can afford CPUs dedicated to single tasks only.

2.4.3. Atomic operations and process synchronization

The problem of concurrency is crucial in any parallel execution environment regardless of whether considering networks, multi-CPU servers or several threads running on the same processor. On a higher level, our goal is to ensure that certain types of shared resources will be accessed by a controlled number of processes only. For that purpose, typically shared or exclusive locks are used.

Technically speaking, “locks” are memory objects with a given value and with an associated well-known name or address. In the case of *exclusive locks*, the convention is that the default lock value is zero and that a process, which first raises the value to one, is granted permission to access the associated resources. Meanwhile, the other processes competing about the same lock are put into the *sleep* mode for a given period of time or until the lock is released. A very simplified version of a lock obtaining algorithms could be the following:

```
while (lock != 0) sleep(n);
lock = 1;
... /* performing the desired action */
lock = 0;
```

Figure 2.7: *Naive implementation of an ‘exclusive lock’*

Even though correct from the single-thread point of view, this approach fails in a multi-threaded environment as there is no explicit guarantee that the lock value doesn't change between the time the expression 'lock != 0' (line 1) is evaluated as negative, and the time when the lock value is updated to '1' (line 2). At this point, unless we can stop all the other CPUs for a while[8] as well as prevent context switching on the local processor, the problem cannot be solved using conventional programming languages. Consequently, the whole situation leads to the requirement of CPU-provided compound type of atomic operations, like "read-and-update", executed with exclusive access to the memory location where the lock value is stored. CPUs without this ability cannot be used in a parallel-execution environment at all.

2.4.4. Exceptions handling

In general, CPUs are state machines executing a single thread at a time. Furthermore, by making use of context switching, the CPU time can be divided according to the various rules between several pseudo-parallel user processes as well as the kernel on itself. On the other hand, this also implies that even though loaded in the memory, the kernel is actually not governing the CPU all the time. In fact, assuming an application performing an excessive in-memory data processing, the kernel might not be invoked for a long period of time at all.

Now, let's suppose a critical error occurs. For example, the currently running process might attempt to access an unprivileged memory area or a division by zero is initiated. Under such conditions, the CPU is typically expected to invoke a specific part of the kernel code in order to take further action on the event. These raised events are called *Exceptions*, and the respective routines to be launched are known as *Exception handlers*.

Again, unless another process-control mechanism is employed (e.g. a CPU experiencing failure might simply be stopped), support for the exception handling is required from any CPU to be used in an SMP environment.

2.4.5. Wall-clock Time with High Resolution

Even though it might look like a minor issue, in many operating systems timestamps play an important role in ensuring in-order-delivery and detecting timeouts. Therefore it is important to make sure that also inside MP systems a consistent reference time is available on each CPU.

Typically, timestamps are defined as the amount of milliseconds elapsed since January 1st 1970, the date also known as the 'UNIX epoch'. Assuming that each CPU is equipped with a local interrupt generator (e.g. rising 1024 signals

[8] Which, by the way, requires another locking if implemented on the software level.

per second), an independent local time can be maintained by each processor. This is, however, undesirable due to possible drifts in each CPU. That's why Linux, for example, defines one of the processors as the time master, providing that all the other servers simply adapt the centrally distributed time signal and use it as needed. Consequently, the "central time" might be defined as the maximal time stamp among all participating CPUs, which ensures that the local time on any of the units never drifts backwards.

The achievable time accuracy is slightly less than 1 millisecond if a 1024Hz interrupt signal is used. This, however, might be not enough on the CPUs with clock rates reaching 1GHz and above, if we consider that between individual timestamp updates more than million instructions might be executed[9]. Therefore in addition to standard time stamps, fine-grain local timestamps are generated for each CPU, typically using the processor-local clock cycle counters. When combined with sophisticated algorithms for inter-processor synchronization, the achievable accuracy is close to 1 microsecond (see [IA-64]). The only requirement then is to have any kind of high-speed counter available in each processor.

2.5. Summary

In *Chapter 2* the specific aspects in designing multi-processor systems were outlined, including assumptions related to *Symmetric Multi-Processing* systems. Furthermore, additional generic requirements put on CPUs intended for parallel execution environments were explained in the corresponding chapters. As a result, we should be now ready to explore and evaluate various Intel CPU families with respect to their eventual applicability for building up SMP systems.

[9] On a 1GHz CPU with an average CPI=1.0, about 1.000.000 instructions can be executed every millisecond.

3. Intel family processors

In the past, Intel has experienced turbulent times. At the beginning of what is today known as the 'computing age', the company was puzzled with instantly growing software requirements causing changes in the core CPU architecture every now and then. For instance, the very early 8086/80186 architecture (originally designed for IBM; 1978) was regarded as too simple and basically outdated as soon as the idea of multi-tasking on the PC platform came into the spotlight. The reason being was that the architecture didn't provide any process isolation at the hardware level.

As an answer to that, Intel came up with the i80286 CPU (1982), which is the first of their processors supporting the concept of Virtual Memory denoted as the *Protected Mode*.

The protected mode was a significant step ahead for the company, even though it was actually nothing new on the market[10]. What Intel, however, didn't anticipate with enough accuracy was the required size of the virtual address space. As the "80286" architecture was based on the 16bit compound addressing model, it was soon clear that additional architectural improvements are needed.

Neither were the next two models, i80386SX and i80386DX (1985), the final stage. The challenge to be answered at that time was whether the external data buses should be 16 or 32 bits wide. Even though from today's perspective the answer is quite clear, the IT industry needed some time to make the shift into the true "32bit world".

On the other hand, by defining the "*i386 execution model*", Intel had finally reached the stage when the logical address space was large enough for almost any application, and individual processes could be isolated in an effective way by the means of the virtual memory subsystem. In other words, the 80386 processors can be considered as the first true multi-tasking CPUs of their kind.

The next generation of processors brought to the spotlight another question, which was cast aside for quite some time. The question was whether the *float-point execution unit* (FPU) should be located on the same chip as the main processor or not. As the actual decision was not only a technical issue, the 'SX' and 'DX' versions of the i80486 processor were developed (1989), where the former one makes use of an external FPU, and the latter comes with the FPU on-die.

[10] For instance, the Motorola 68k processors did provide virtual memory capabilities already in the first implementations.

Finally, around the year 1993 Intel launched the Pentium family of processors that were backward compatible with all their predecessors including the non-protected mode CPU models.

One remarkable moment is that the Pentium architecture introduced a 64bit external data bus even though the register width was 32 bits for integer numbers and 80 bits for floating-point arguments. In other words, the Pentium family processors were the first ones driven by the need for performance improvements, rather than introducing significant architectural changes.

The Pentium CPU was a success not only from a marketing point of view, but also from the technological perspective. Perhaps that's also one of the reasons why the company decided to preserve the name, and instead of inventing new "6-based" names they have brought processors like Pentium II, Pentium III and Pentium 4 into the market[11].

Nevertheless, Intel CPUs were still recognized as the processors for 'small' computers, hardly ever considered as valid competitors on the enterprise server market. This is mainly due to the fact that the entire IA-32 family implements the CISC (*Complete Instruction Set Computing*) paradigm[12], which is pronounced to be generally slower due to the inherent microcode complexity and the wide range of implemented instructions. Perhaps that's also one of the motivations why in the mid-90's Intel began a joint development of the next generation IA-64 CPU architecture together with HP. This new architecture is then supposed to deliver a RISC-grade performance in the native mode, whilst still preserving backward binary compatibility with the IA-32 and PA-RISC software. As a result, both the legacy software, as well as the software specifically written for IA-64 computers, should be running on the same system hardware if required.

[11] A detailed timeline of Intel CPU models is available e.g. from [Intel CPUs].

[12] Note that internally even some of the x86 CPUs, like Pentium III and better, operate the RISC way internally.

The current Intel CPU portfolio contains the following processor models:

Name	*Arch.*	*Max. RAM*	*Max. Cache*	*Max.Core Freq.*	*Intended for*
Pentium Celeron	IA-32	4GB	256 kB (L2)	1.8 GHz	P, D, E
Pentium II	IA-32	4GB	256 kB (L2)	333 MHz	P, D, E
Pentium III	IA-32	64GB	512 kB (L2)	1.4 GHz	D, S, W
Pentium III Xeon	IA-32, HTT	64GB	512 kB (L2)	1 GHz	S, W
Pentium 4	IA-32, HTT	64GB	512 kB (L2)	3 GHz	D, P
Intel Xeon	IA-32, HTT	64GB	512 kB (L2)	2.8 GHz	S, W
Intel Xeon DP/MP	IA-32, HTT	64GB	2MB (L3)	2 GHz	S, W
Itanium	IA-64	16TB	4MB (L3)	800 MHz	S, W
Itanium 2	IA-64	4096TB	3MB (L3)	1 GHz	S, W

Table 3.1: *Contemporary Intel CPU portfolio*

Legend:

HTT = Hyper-Threading Technology
P = (P)ortables, E = (E)mbedded, D = (D)esktops,
W = (W)orkstations, S = (S)ervers

What we see in the table above is that processors prior to the Pentium III are considered to be rather low-power ones. Beginning from the Pentium III, the processors can be used basically in any area. However, for servers and workstations the *Intel Xeon* and the *IA-64* type of CPUs would be recommended, whilst the *Pentium 4* is meant mainly for desktops. This might also imply certain deviations, like additional registers, etc. On the other hand, as long as the underlying "*80x86*" architecture is preserved, any of the CPUs can be basically used for any role, if the performance and power consumption is satisfactory from the designer's perspective.

3.1. IA-32 architecture

The IA-32 CPUs represent the CISC type of processors with two possible modes of operation. The first one is known as the *real mode* and is compatible with the 8086 and 80186 processors. As this mode enforces an unprotected memory access for all the active threads, it is of a very little interest to us. Perhaps the only thing to remember, is that even today this mode is the default when the 80x86 CPUs start.

The second mode, in contrast, is the *protected mode* [IA-32], which offers several advanced features like virtual memory, context switching, paging, etc., making it more suitable for parallel-execution environments with logically isolated tasks. Protected mode is activated by changing content of the register CR0.

3.1.1. The program execution environment

From the programmer's perspective, the processor offers eight universal registers (32bit), and six selectors (16bit), with the following roles:

```
---- universal registers ----
EAX = Accumulator (target of many arithmetic operations)
EBX = General purpose (GP) register
ECX = Counter / GP register
EDX = GP register
ESI = GP register / source index for streamlined operations
EDI = GP register / destination index for streamlined operations
EBP = GP register / index / stack frame indicator
ESP = Stack pointer (in combination with SS)
---- selectors ---
CS = Code Segment selector
SS = Stack Segment selector
DS = Data Segment selector
ES, FS, GS = General Purpose selectors
---- special registers ---
EIP = Instruction pointer (in combination with CS)
EFLAGS = Binary array of control and status indicating bits
…. and other
```

Figure 3.1: *IA-32 basic register set*

Note that by no means this listing is complete, as there exist plenty of other additional control registers associated with various execution units inside the processor.

The table also indicates one important aspect of the CISC instruction set, which is its non-orthogonal design. This meaning that not all the index registers, for instance, can be used as arguments for an arbitrary memory-manipulating instruction, and so on. These restrictions are consequences of the legacy, as well as certain design trade-offs inside a given CPU itself.

Another issue not mentioned in the previous chapter is the group of floating-point registers ST(0)..ST(7), which are found as a part of the independent FPU unit. Registers are organized as a low-depth stack, and their content is, as well as the actual FP operations, controlled using special instructions with the 'ESC' prefix (11011b) in the binary code. In practice, whenever any FPU-related instruction is being executed, the main CPU waits until the FPU has completed its task[13]. This in turn makes FPU operations rather slow, even though it is still faster than when emulated by the software.

3.1.2. Virtual Memory Subsystem

The IA-32 architecture makes use of a hardware-assisted page frame mapping between virtual and physical addresses. Individual logical addresses are composed of the *selector* (16bit) and the *offset* (32bit) part, where the *selector* is an index into the table of *segment descriptors*, and the *offset* part is simply added to the 32bit base address found in the table (see *Figure 3.2* for details). In this way a *logical address* is transformed into a corresponding *linear address*. Should the required mapping information be missing from the table, a system-level exception is raised in order to resolve the situation.

[13] This is a heritage from the time when the FPU used to be an additional external peripheral attached to the system bus. Obviously, the main CPU had then to stop any activity over that bus until the FPU operation in question is completed.

In the system there exists a single *Global Descriptor Table* (GDT) and several *Local Descriptor Tables* (LDTs)[14]. As described in *Figure 3.2*, while performing a memory access, the choice between the GDT and the currently active LDT is made based on the '**TI**' bit value in the used selector:

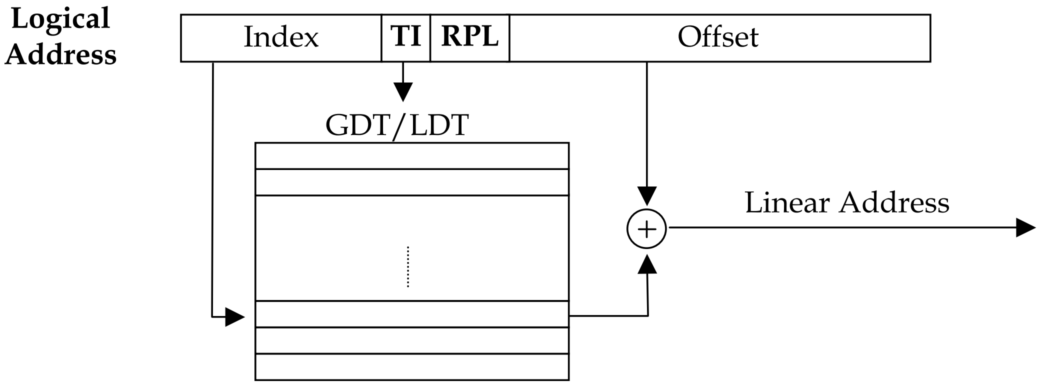


Figure 3.2: *IA-32 Logical-to-Linear address mapping*

Note that the term *Linear Address* (LA) is more correct in this context, because at a later phase this address may be also passed through the paging unit and hence transformed once again. As a result, the *Physical Address* of the final data location might also differ even from the LA in question.

Implementing access rights

The descriptor tables can contain up to 8192 items, 64 bits each. Among those the **DPL** (*Descriptor Privilege Level*) field plays an important role when implementing data and code protection. In practice it is used in the following way:

Each of the running processes is given a *Current Privilege Level* (**CPL**), being a numeric value between 0 (highest priority) and 3 (lowest priority) inclusive[15]. Furthermore, any given process can also alter its access rights toward a lower privilege level by supplying the desired privilege level in the **RPL** field in the *selector* part of the accessed logical address. The final *Effective Privilege Level* (**EPL**) is then calculated as a numerical maximum of both the CPL and the RPL.

Whenever a memory access is performed, the EPL value is checked against the DPL value inside the chosen segment descriptor (item in the descriptor table). While doing so, the following rules apply:

[14] Typically single LDT per each process is used.

[15] For the active process the CPL value is stored in the CS selector.

1. Data access is allowed into blocks with a lower or equal DPL.
2. Code access (that is "execution") is allowed to the block on the same privilege level or, if configured so, to the block with a numerically lower DPL. While doing so, the CPL of the calling process won't change.
3. The only way to change the CPL is to pass a call through a special execution gate.

In other words, the processor architecture ensures that data on a higher privilege level can be accessed only via a provided API, and that such an API cannot be "enhanced" by callback functions from unknown binaries located in an unknown or a non-trusted binary block. Any violation of the above-stated rules causes a kernel-level exception. After that, it's up to the operating system to take any necessary corrective actions.

3.1.3. Context switching

Context switching is implemented in a relatively easy way. Each of the tasks are expected to maintain a dedicated *Task State Segment* (TSS) table (described in *Figure 3.3*), which is used as a storage for all important registers, should a context switch occur. The table has to be at least 104 bytes long (see [IA-32]), however the upper limit is not explicitly given. This allows also some additional data to be stored along with the CPU status information.

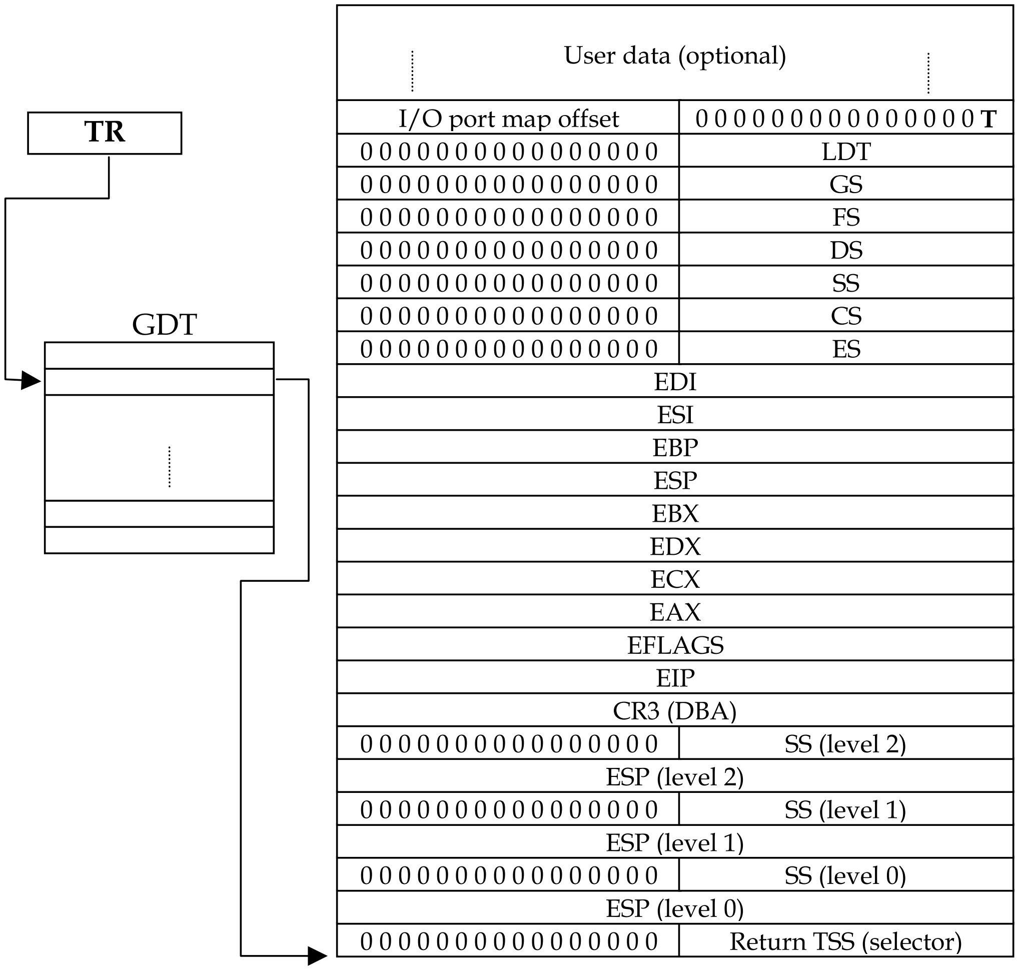
The address of the active TSS (that is, the TSS belonging to the task being executed) is stored in the **TR** register (16bit). Should context switching occur, the following steps are performed:

1. The logical context of the current task is written into the active TSS, and all the pending cache writes are completed.
2. The TR register gets updated with the value associated with the TSS of the new process.
3. The TR value of the previous task is written at the end of the newly activated TSS.
4. The execution context of the new process is activated inside the processor.
5. The task execution continues (is resumed) at the instruction referred by the CS:IP register pair of the activated process.

The context switching can be triggered in four different ways:

- using the **CALL** (long call) instruction to the TSS segment,
- using the **JMP** (long jump) instruction,
- using the **IRET** instruction with the 'NT' flag set to '1', or
- via an interrupt handler directed to the TSS segment.

The last option (that is, performing a context switching as a reaction to the INT or IRQ request) is an especially interesting one, since in this case all registers of the interrupted process are saved automatically prior to the interrupt handler activation. Therefore, the *interrupt handler* is released of this context saving burden. Finally, since the TR of the previous task was stored in the active TSS during the IRQ/INT invocations, returning to the interrupted process is a matter of a single instruction, this being IRET with 'NT=1'.



User data (optional)	
I/O port map offset	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 **T**
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	LDT
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	GS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	FS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	DS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	CS
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ES
EDI	
ESI	
EBP	
ESP	
EBX	
EDX	
ECX	
EAX	
EFLAGS	
EIP	
CR3 (DBA)	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS (level 2)
ESP (level 2)	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS (level 1)
ESP (level 1)	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SS (level 0)
ESP (level 0)	
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Return TSS (selector)

Figure 3.3: *IA-32 Task State Segment and its addressing*

3.1.4. Atomic operations and process synchronization

As mentioned previously, *atomic operations* are essential for implementing SMP or single-CPU multi-tasking systems. When identifying these, the following rules apply on the IA-32 platforms:

- Assembly language operation that makes zero or one memory access is atomic (e.g. *mov ax,[addr]*).
- Read/modify/write-type of instructions, such as *inc* or *dec*, are atomic if no other CPU operates the system bus (that is, in UP systems only).
- Simple instructions featuring the *lock* prefix are atomic by definition because the prefix implies exclusive access to the memory bus throughout the entire instruction execution period. However, instructions with the *rep* prefix (string operations) are not explicitly atomic, because while executing the earlier mentioned instructions, CPU keeps on sampling incoming IRQ lines, and hence the main program course might get interrupted at any time.

Thus, from an SMP design perspective, only the single memory access and lock-prefixed instructions are interesting. Fortunately, the Pentium CPU, for instance, offers as many as 18 instructions accepting the *lock* prefix. In addition there also exist a few more instructions, like *xchg*, which imply an atomic execution as well.

3.1.5. Interrupts and Exceptions handling

Interrupt and Exceptions are basically events requiring immediate action. The first being reactions on external events (typically the signal arriving over the IRQ line), and therefore asynchronous by nature. The second are consequences of failures inside the CPU itself, and therefore also synchronous with the main application execution. Both of them are, however, served the same way:

As soon as the respective event occurs, the CPU interrupt subsystem calculates a number within the range 0 thru 255 (inclusive) identifying the event source. After this it performs a table lookup operation at the address specified by the **IDTR** register, and fetches the address of the handler to be used for serving a given event. Next, the currently running application is interrupted and the desired *interrupt handler* is invoked. Lastly, at the end of its run the interrupt handler is expected to perform a couple of clean up operations in order to restore the execution environment of the original application.

Despite its simplicity, the above-described concept works an acceptable way only for CPU-internal events, that is, for handling CPU exceptions. On the other hand, the experience shows that effective IRQ handling requires more sophisticated event re-ordering and prioritization, and hence it is not a good idea to "hard-wire" individual IRQ lines directly to the CPU pins. Therefore in early times an external *Programmable Interrupt Controller* (PIC) was used. Most commonly this was the 8259A chip.

The 8259A is an integrated circuit capable of a selective interrupt source inhibition ("masking"), as well as an elementary event priority re-classification. However, since the chip features only eight individual IRQ input lines, its capabilities are limited already in a single-CPU environment, not mentioning any server-class systems. Therefore, especially for the SMP systems, Intel has developed a distributed interrupt handling solution known as the *Advanced Programmable Interrupt Controller* (APIC) [Linux Kernel; interrupt subsystem].

APIC isn't a single chip, though. Instead, it is a distributed messaging system composed of two types of substantially different circuits. As indicated in *Figure 3.4*, the first type of circuits, called '*I/O-APIC*', is a programmable hub handling the incoming IRQ requests and generating related event indication messages on-demand. On the other side, the respective message receiving part is called '*Local APIC*' (L-APIC) and is located inside each CPU.

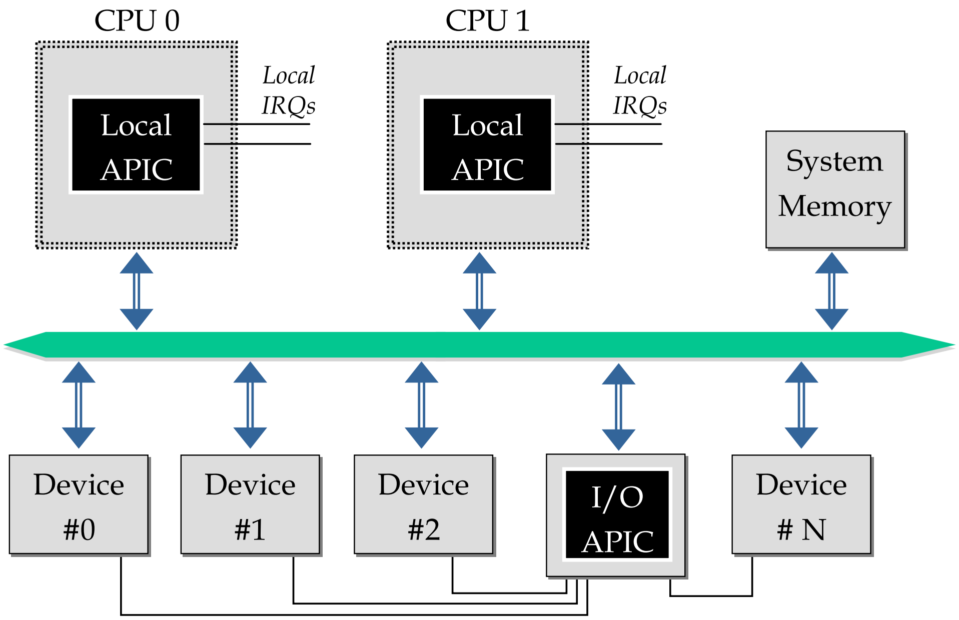


Figure 3.4: *Distributed IRQ handling*

Messages between the APIC circuits are passed via a shared data block inside the *system memory* (typically 2MB). Among other things, this concept allows a given IRQ request not only to be directed towards a predefined CPU, but also easily re-routed to other one if necessary. A load-balanced I/O handling can be then implemented in a rather straightforward way.

Finally, one remarkable feature is also that the *Local APICs* only enhances the already existing interrupt subsystems of their respective CPUs, while the original IRQ lines are still preserved. In practice that means that an APIC-enabled CPU will work efficiently in a multi-CPU system as well as in a legacy uni-processor setup with IRQ lines connected directly to the CPU itself.

3.1.6. Summary

In *Section 3.1* the IA-32 architecture was introduced with a special accent on the features necessary for implementing multi-CPU architectures. As a short conclusion of this evaluation, the IA-32 seems to be mature enough to be used in SMP implementations in one example.

3.2. ’Dual-core’ CPUs

During the year 2001, Intel launched two new types of processors known as the ‘Pentium III Xeon’ and the ‘Intel Xeon’ respectively, which are CPUs featuring the so-called *Multi-Threading technology*.

The idea of *Multi-Threading* is based on the fact that even with a binary code optimized for parallel execution, still about 30% of the time certain units inside the CPU *execution engine* are idle. Consequently, if we cannot achieve better efficiency within the same thread (that is, with further code optimization), it might make sense to run another thread on the same chip.

Going back to the very essence of the CPU design, what gives an “identity” to any processor is the content of its internal *registers*, while the actual “processing power” is determined by the quality of the surrounding *execution engine*. The more independent register sets present, the more *Logical CPUs* are identified in the system. This is also the principle idea behind the Xeon processors, providing that by introducing yet another set of registers on the very same chip (see *Figure 3.5*), the resource utilization of the underlying execution engine is increased up to the maximal achievable level. As a result, the Xeon processors deliver two logical CPUs on a single circuit board (a more elaborate description of the Hyper-Threading architecture is available from [HyperThreading]):

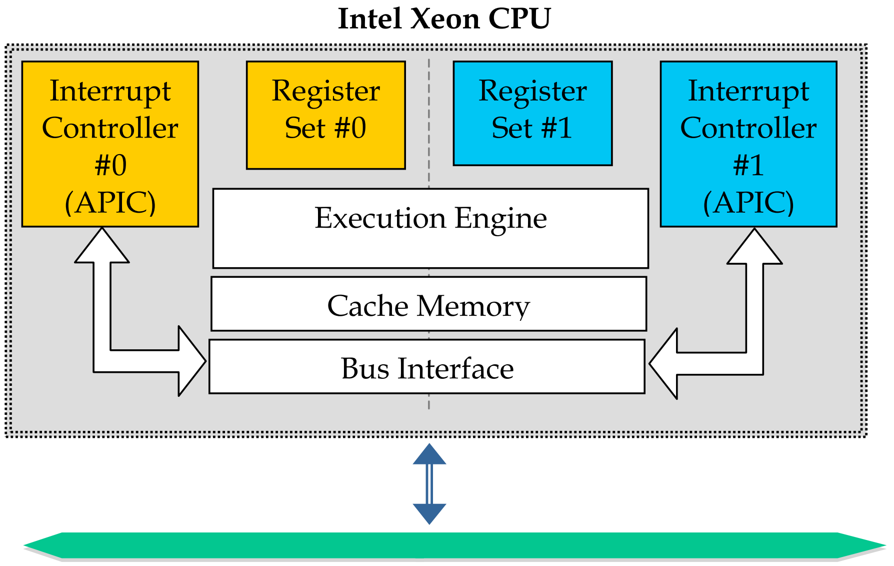


Figure 3.5: *Intel Hyper-Threading Architecture*

Introducing an additional *Logical CPU* is a relatively easy task, at least on a conceptual level. On the other hand, what must be remembered is that the capacity of the underlying execution engine doesn't change, and therefore any performance gain of the "new CPU" certainly won't be more than the spare capacity of the original one. This is also why the term 'Dual Core CPU', used mainly in marketing material, is not exactly correct when referring to Hyper-Threading enabled processors.

From the SMP architecture point of view, using the HTT-enabled CPUs doesn't bring about any significant changes, except that the effective capacity of the underlying execution engines should be considered when distributing tasks between all involved physical CPUs[16]. On the other hand, the fact that each logical CPU is identified and counted separately brings the following advantages:

1. No changes to the user-level applications are necessary.
2. There is no need to modify the operating system except for a few recommended adjustments to the task scheduler part (otherwise operations in an "ordinary SMP" environment are assumed).
3. There is no difference in the booting sequence, since as in any other MP system, all logical CPUs, except the CPU0, will be suspended by the BIOS in the early booting phase.

[16] The conceptual difference between physical and logical CPUs is already considered in the scheduler implementation of the most recent Linux kernels, for instance.

Therefore, unless Intel makes architectural changes inside the execution engine itself, multi-threaded CPUs are as much valid members of SMP systems as their respective single-threaded equivalents.

3.3. IA-64 architecture

The IA-64 concept is a joint effort between Hewlett Packard and Intel intended to define a next generation processor architecture exploiting *Instruction Level Parallelism* (ILP) for achieving the maximal CPU resource utilization [IA-64WP].

The principal idea of ILP is that by involving the compiler in the process of composing parallel-executable instruction groups, the result will be more efficient from the performance point of view compared to the situation where the CPU would try to discover parallel-executable instruction groups independently. This is mainly due to the fact that the compiler has a much wider contextual knowledge of the entire application, and therefore it can produce a more efficient binary code. Besides, since the instruction-level parallelism is expressed explicitly, the traditionally used re-ordering engine can be removed from the CPU entirely, or at least simplified a significant way.

No code can, however, be produced without a detailed architectural knowledge of the target platform. As we will see later, unlike its predecessors, the IA-64 processors make use of the *Explicit Parallel Instruction set Coding* (EPIC), which is substantially different from the other alternative known as the *Very Long Instruction Word* (VLIW). The main difference is that the VLIW requires a more detailed knowledge about the target platform in terms of the quantity and type of internal execution units located inside the CPU, while the EPIC paradigm makes the assumption that the number of internal execution units is unlimited by definition [EPIC]. Consequently, should a given binary code be moved between different VLIW processors, it has to be re-compiled with new assumptions about the target platform. In contrary, the EPIC-composed code can be running on any processor of the same architectural family. This is because the EPIC concept is more focused on defining the execution framework (i.e. 'how to express'), rather than on forcing implementation details. As a result, in the long run we might expect several types of CPUs with internal differences, yet still compliant with the common IA-64 architecture. Naturally, those CPUs can then be priced according to their capabilities and the market segment they are intended for.

3.3.1. The program execution environment

The IA-64 architecture can be, from a programmer's perspective, thought of as a RISC CPU model with a massive amount of registers [IA-64]. Like any other RISC architecture it employs the load/store methodology where any individual data item has to be first loaded into a designated register before it can be further used (see [RISC_Ref]).

Basic IA-64 properties

The instruction format for most IA-64 instructions includes the *instruction code* (that is, an identification of the operation to be performed), two source operands and a single destination identifier (target). The architecture is *fully predicated*, meaning that individual instructions are executed only if all indicated *predicates* (i.e. control bits) are set to the active value. If not, the instruction is simply skipped. In total, 64 predicates are defined, whose value can be controlled either explicitly (using appropriate instructions) or as a result of other instructions. In addition, the predicate *p0* is said to be always valid (true), which eliminates the need for defining "special cases" in the instruction coding. In general, predicates reduce the number of necessary branch instructions, hence improving overall CPU performance.

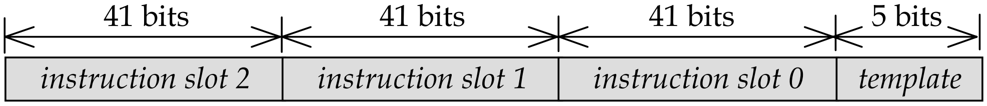
IA-64 architecture offers 128 general-purpose registers (65 bits each), and 128 floating-point registers (82 bits each), making programming really comfortable. The "excess bits" associated with numeric registers are used for encoding symbols like **NaT** or **NaVal** ("Not a Thing" and "Not a Value" respectively), used mainly as guides during speculative code execution. Even though these additional bits aren't typically problematic during the normal application course, attention has to be paid when manipulating affected registers during the context switch, for instance.

Finally, IA-64 architecture also defines a set of control registers and performance counters whose purpose and content might differ depending on the actual processor model.

Instruction format

On the IA-64 platform instructions are organized in so-called *bundles* (128 bits each), which are the nominal binary code units. Each bundle provides space for three *instruction slots* and the *template* field. Most of the IA-64 instructions make use of a single 41bit instruction slot, while there also exist few instructions spanning across two slots simultaneously.

The five bits wide *template* field is used for indicating the target execution unit per each enclosed instruction. Furthermore, some of the template field values are also encoding the “stop” sign, which is used as a border marker separating individual *instruction groups* in the code. See *Figure 3.6* for details:



	Slot				Slot				Slot		
Template	**0**	**1**	**2**	**Template**	**0**	**1**	**2**	**Template**	**0**	**1**	**2**
0x00	M	I	I	0x0a	M ;;	M	I	0x12	M	B	B
0x01	M	I	I ;;	0x0b	M ;;	M	I ;;	0x13	M	B	B ;;
0x02	M	I ;;	I	0x0c	M	F	I	0x16	B	B	B
0x03	M	I ;;	I ;;	0x0d	M	F	I ;;	0x17	B	B	B ;;
0x04	M	L	X	0x0e	M	M	F	0x18	M	M	B
0x05	M	L	X ;;	0x0f	M	M	F ;;	0x19	M	M	B ;;
0x08	M	M	I	0x10	M	I	B	0x1c	M	F	B
0x09	M	M	I ;;	0x11	M	I	B ;;	0x1d	M	F	B ;;

Legend: M = Memory Unit
I = Integer Unit
B = Branch Unit
F = Floating-point Unit
X = Extended Format Unit (unit for handling two-slot instructions)
L = placeholder for the second instruction in the X-unit pair
;; = stop sign (instruction group separator)

Figure 3.6: *IA-64 instruction bundle and format of the ‘template’ field*

Instructions between two consecutive “stop” signs are considered to be mutually exclusive and therefore ready to be executed simultaneously. This also implies that, no instruction from the next instruction group may be executed unless all instructions from the previous one are processed.

Even though the above-described rules are in fact nothing new in the ILP world, what makes the IA-64 approach different is the fact, that thanks to the carefully designed format of the *template* field, individual *instruction groups* can, by definition, span across several instruction bundles regardless of what the target CPU model is. As a result, the binary code is compatible with any former or future IA-64 implementation, and yet it is optimized from the ILP point of view. This level of freedom is certainly a significant step ahead comparing to the VLIW world, where the CPU-dependent bundle size also matches with the size of instruction groups.

Floating-point and integer calculations

As mentioned earlier, the FPU unit provides 128 floating-point registers. Each register is divided into the *sign* bit, 17 bits of *exponent* (biased against the value 65535), and 64 bits of the *significant* part of the number. The exact formula for converting into a human-readable form is:

$$\boxed{F(sign,exp,significant) = (-1)^{sign}.2^{(exponent-65535)}.significant/2^{63}}$$

Besides its basic function, floating-point registers can also contain standard 64bit integer values, if the sign bit is ‘0’ and exponent is set to ‘0x1003e’. This, in consequence, allows extremely fast conversions between floating-point and integer registers, as well as logically extending the integer register file by yet another 128 registers from the floating-point register area if necessary.

Two floating-point registers have special meaning: The register *f0* is always read as ‘0.0’, and the register *f1* is always read as ‘1.0’. This eliminates the need for passing the numerical constants along with the instruction code, which would be a very inefficient solution. Updating registers *f0* and *f1* is forbidden.

Regarding the instruction repertoire, the IA-64 aims at providing only such floating-point instructions, which cannot be implemented in a more efficient way in the software. Like many other RISC architectures, the IA-64 doesn’t provide instructions for arithmetic division, implying that a reciprocal approach has to be used if necessary. On the other hand, instructions like ‘multiply-and-add’ (i.e. “projection and offset shift”) are present in order to facilitate single-precision complex calculations and graphical transformations.

Modulo-scheduled loops

Moving data blocks between different memory locations is a very common activity in many programs. While doing so, however, less than half of the time is typically spent in reading the data from the source, and about the same time is used for writing the data to the new location. The rest of the time is then the management overhead necessary for operations in a controlled loop. In a very simple form this situation is depicted in *Figure 3.7*:

```
start: ld8 r32 =  [ptr1], 8  ;;
       st8 [ptr2] = r32, 8
       br.loop start
```

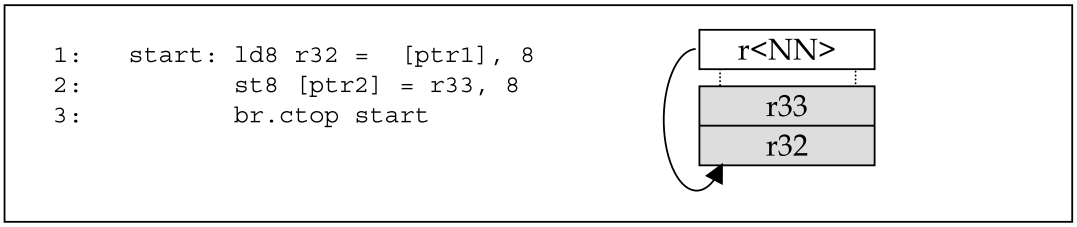
Figure 3.7: *Data replication using traditional methods*

Assuming that the pointers ‘ptr1’ and ‘ptr2’ are in reality special-purpose registers incremented with every “loop” command, this code fragment will copy data from one location until the loop-controlling counter reaches its zero value. However, as there exists a *WAR* (Write-After-Read) dependency between the first two lines, the "stop" sign is necessary in order to split the entire loop into two instruction groups. As a result, the CPU spends only half of the time reading and half of the time writing, while in theory it would be capable of both activities simultaneously.

In order to address this performance problem, the IA-64 architecture introduces the concept of *rotating registers* with the following semantics:

1. Using the **alloc** instruction, a certain portion of the register file can be denoted as "rotating registers".
2. Whenever the *counted loop* instruction is reached, at the end of the clock cycle the values in registers designated as rotating are shifted one position ahead.

Using these assumptions, the very same loop can be rewritten in the following way:



```
start: ld8 r32 =  [ptr1], 8
       st8 [ptr2] = r33, 8
       br.ctop start
```

Figure 3.8: *Data replication using modulo-scheduled loops on IA-64*

The modulo-scheduled loops work as follows: By the time a new value is read into the register *r32*, the value acquired during the previous iteration (propagated as *r33*) is written back to the location indicated by 'ptr2'. This all happens during a single clock cycle only, since there is no logical dependency between the registers *r32* and *r33*. In other words, the CPU is reading and writing simultaneously and hence the internal resources are utilized in the maximal possible way[17].

The Register Stack Engine

The Register Stack Engine (RSE) is yet another innovation inside the IA-64 providing a virtually unlimited amount of LIFO-organized registers with the help of a limited number of physical registers ('*Register Backing Store*') and the external memory. The underlying concept is as follows:

[17] Note that the code used in *Figure 3.8* is not complete as the prolog and epilog parts are omitted. Those can be found from [IA-64].

1. Using a predefined instruction (e.g. **alloc**), a certain range of registers (starting with *r32* and lasting up to *r127* inclusive) within the user space is designated as being served by the RSE. This is known as the *register frame.*
2. As soon as the allocation happens, the RSE tries to allocate the same amount of physical registers from its private space.
3. Any consequent **br.call** instruction causes the RSE pointers to be adjusted so, that the content of current register frame is made inaccessible, while the called function is free to allocate a new frame for itself (in the RSE terminology, the current frame has been pushed on top of the register stack).
4. The **br.ret** instruction restores the previous register frame by moving the related pointers back to the state they were before the **br.call** instruction occurred.

The net effect is that the program doesn't have to care about protecting the content of the local registers whenever any nested function is called. In the same way, there is also no need for any excessive context savings should an interrupt handler be launched or should any exception handling take place.

Internally the RSE is implemented so, that as long as there are enough physical registers available inside the RSE subsystem, they will be used for holding the register stack date. Should this one become full, the "oldest" data is copied to the external memory and the physical registers are recycled. Conversely, by releasing the allocated frames (i.e. returning from procedures), the original register content is loaded back from the memory into the RSE physical registers. All of these operations, however, happen on the hardware level and therefore also transparently from the main program perspective.

3.3.2. Virtual Memory Subsystem

The Virtual Memory Subsystem operates in a somewhat different way than in the IA-32 case, even though the basic idea is preserved [IA-64/VMM]: The user-level application is (at least in theory) capable of addressing the entire logical address space, while on the bottom line, logical addresses are translated into physical ones with the help of the operating system. The translation is transparent for the user application in question.

The IA-64 CPUs are not required to implement the entire 64-bit address space [IA-64]. Instead, each implementation must support between 50 to 60 bits of the lower part as well as the three most significant bits (MSBs) of the *logical address*. This effectively splits the entire virtual address space into eight *virtual* *regions*, where each of them contains the same portion of the implemented and unimplemented addresses (see *Figure 3.9*).

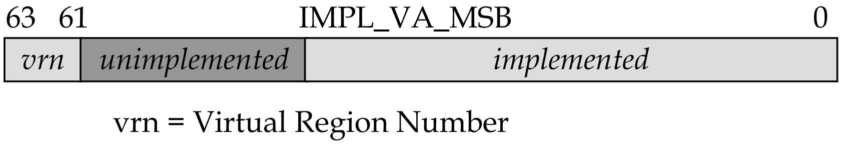


Figure 3.9: *Format of the IA-64 virtual address*

In a similar way as the logical addresses, the *physical addresses* do not need to be implemented for all the 64 bits either. Instead, an IA-64 compliant CPU must define between 33 and 63 bits (inclusive). In addition, the most significant address bit (i.e. bit 63) is used as an explicit indicator whether the addressed data can be kept in the CPU-local cache or not. As a result, this schema creates two regions with certain portions of implemented and unimplemented address space where both the 'implemented' parts are in fact aliases for each other (see *Figure 3.10*).

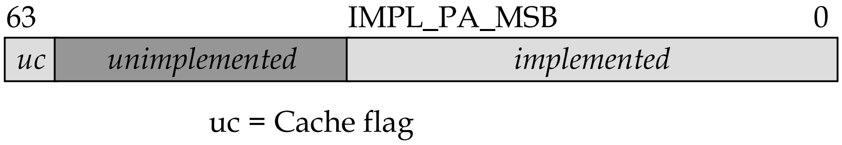


Figure 3.10: *Format of the IA-64 physical address*

Since the width of the implemented address space is defined by the CPU model, applications are not allowed to make any explicit assumptions on the size of the address space. Instead they should rely on information provided by the operating system. In the case of the Linux/ia64, the reference constants are called **IMPL_VA_MSB** for the virtual address space and **IMPL_PA_MSB** for the physical address space. Effectively these stand for the logical number of the most significant bit in the address space they represent. In case of the first Itanium CPU, the IMPL_VA_MSB equals to 50 and IMPL_PA_MSB is 42.

Page tables

In order to maintain the entire address space, the virtual as well as the physical memory area is divided into individual memory page frames of a fixed size. Unlike the IA-32, however, on the IA-64 platform it is the actual operating system that is responsible for selecting the most suitable memory space granularity (naturally, out of the offered options[18]) as well as defining and maintaining all the necessary tables, and providing the mapping information to the CPU whenever necessary. In other words, the IA-64 processors achieve more flexibility in virtual memory management by enforcing a software-based memory mapping, rather than implementing a hardware-based solution.

Like many other CPUs, also the IA-64 makes use of the *Translation Lookaside Buffer* (TLB). This hardware-implemented buffer is explored during each virtual memory access, and only in the case where a required virtual page frame number is not found, a respective kernel routine is called, which is generally known as the *TLB miss exception*. It is then up to the operating system to provide the missing mapping information[19]. Alternatively, the user-level application can also be terminated should an unprivileged memory access be detected.

In addition to what was described above, the IA-64 also supports an optional feature called the *VHPT walker*, which is intended for solving *TLB miss* events partly in the hardware itself. The idea is that for those memory regions where the hardware support is explicitly activated, a memory-resident table of physical entries can be "walked through" in order to find the required information before the kernel-level *TLB miss handler* is eventually invoked. In order to exploit this feature, the table of mapping information must be linear, and its address must be supplied in the *page table address* (PTA) register.

3.3.3. Context switching

The context switching can be on AI-64 implemented with a minimal effort from the programmer's perspective. First of all, using the *Register Stack Engine* (as explained in *Section 3.2.1*) eliminates the need for excessive register dumps during the context change. In addition, the CPU also provides auxiliary bits for identifying changes in certain register files. For instance, the float-point register file is divided into two halves where for each of them a single status bit exists indicating whether any of the registers was modified during the application course. Naturally, registers not modified during the application run do not need to be saved, as the in-memory copy remained identical with the current CPU state.

A somewhat specific issue is the handling of the TLB table. As described in *Section 2.4.1*, the cached TLB entries describe the relation between virtual and

[18] In case of Itanium the supported page sizes are 4kB, 8kB, 16kB, 256kB, 1MB, 4MB, 16MB, 64MB and 256MB.

[19] Note that *TLB miss* events are executed in the kernel space with the virtual memory option disabled. Thus there is no risk of several nested *TLB miss* events at this point.

physical page frames within a certain task. Should context switching happen (and should the next activated thread belong to another task) it has to be ensured that those TLB entries belonging to the previous task wouldn’t be used during the new task’s course.

A very simple way of ensuring the task isolation would be to flush the entire TLB table up on a task switch. On the other hand, considering that upon a return to the same task most of the entries might need to be re-uploaded again, this approach is clearly insufficient from the performance point of view. In order to tackle this problem, the IA-64 makes use of the three most significant address bits known as the *Virtual Region Number* (see *Section 3.2.2* for details).

In short, the ‘*vrn*’ value is used as an index into the **rr0-rr7** register array, holding the *Region ID* for each of the eight virtual regions. With every memory access, the obtained *rid* (the result of the rr0-rr7 table lookup) is used as a part of the *search key* when the TLB buffer is examined, implying that only TLB entries with the same *rid* are taken into account. Consequently, TLB entries belonging to a different task are ignored and turned inactive simply by changing the content of the *Region ID* registers into values valid for the currently active task. Therefore, no excessive TBL flushing is necessary should a context switch take place.

As a brief summary, it is quite obvious that a high design effort has been made in order to minimize the context-switching overhead on the IA-64 platform. Combined with the fact that that certain registers can be at the compilation time denoted as “fixed” (that is, unavailable for the user processes), the achievable results are quite impressive.

3.3.4. Atomic operations and process synchronization

For implementing atomic operations, the IA-64 platform defines two types of instruction families implemented by the Memory unit (M):

```
        cmpxchg{1,2,4,8}   r1 = [r2], r3, ar.cvv
        fetchadd{4,8}      r1 = [r2], n
```

Figure 3.11: *Atomic instructions on IA-64*

Instructions are by definition atomic, and as their name indicates, they manipulate the respective memory content in slightly different ways. In either case, however, on top of this platform-dependent foundation all other abstractions like *locks* and *semaphores* can be implemented. On the other hand, the IA-64 architecture doesn’t provide an explicit *lock* prefix for managing the shared system bus, and hence the programmer has to manage with the above-enlisted instructions only.

3.3.5. Interrupts and Exception handling

The design of the IA-64 interrupt subsystem follows traditional architectural principles like the interruption at the first possible occasion (to be determined by the CPU), in-order-delivery and respect for event prioritization. In addition, there also exists a requirement of *precise* interruptions, which basically means that all the instruction bundles loaded into the CPU have to be executed before the interrupt handler is activated, while no other instruction bundle of the interrupted application should be loaded whilst the interrupt service handler is active. In other words, it may happen that a particular *instruction group* (that is the group of instructions possibly executed in parallel) will be partitioned into subgroups executed ″before″ and ″after″ the interrupt occurred, hence not in parallel as intended originally. On the other hand, a positive outcome is that at the interrupt handler execution time it is possible to easily identify what the IP (*instruction pointer*) value is for the instruction bundle to be executed next[20] [IA-64].

From the software perspective, the IA-64 CPU defines 68 different interrupts with their respective interrupt service code located in the 32kB memory block at a location designated by the **iva** register. Out of these interrupts, the first twenty interrupt handlers have pre-allocated space for 64 bundles (1042 bytes), while the remaining 48 handlers can use up to 16 bundles (256 bytes) for immediate interrupt serving. Any other longer execution sequences have to be implemented using code branching as necessary.

[20] In contrast, if the addressed unit would be an individual instruction instead of the whole instruction bundle, it might easily occur that the next instruction executed ‘after’ the interrupt, is physically located in the middle of its respective bundle. Naturally, it would be quite a difficult task to “explain” this to the CPU some way.

#	Offset	Name	#	Offset	Name
0	0x0000	VHPT Translation	23	0x5300	Data Access Rights
1	0x0400	Instruction TLB	24	0x5400	General Exception
2	0x0800	Data TLB	25	0x5500	Disabled FP-Register
3	0x0c00	Alternate Instr. TLB	26	0x5600	NaT Consumption
4	0x1000	Alternate Data TLB	27	0x5700	Speculation vector
5	0x1400	Data Nested TLB	29	0x5900	Debug vector
6	0x1800	Instruction Key Miss	30	0x5a00	Unaligned Reference
7	0x1c00	Data Key Miss	31	0x5b00	Unsupported Data Ref.
8	0x2000	Dirty-Bit	32	0x5c00	Floating-point Fault
9	0x2400	Instr. Access-Bit	33	0x5d00	Floating-point Trap
10	0x2800	Data Access-Bit	34	0x5e00	Lower-Priv.Transfer
11	0x2c00	Break Instruction	35	0x5f00	Taken Branch Trap
12	0x3000	External Interrupt	36	0x6000	Single Step Trap
20	0x5000	Page Not Present	45	0x6900	IA-32 Exception
21	0x5100	Key Permission	46	0x6a00	IA-32 Intercept
22	0x5200	Instr. Access Rights	47	0x6b00	IA-32 Interrupt

Table 3.2: *Interrupts and Exceptions on IA-64*

As seen in *Table 3.2*, not all possible vectors are defined at this point. Therefore, the corresponding locations in the IVT (*Interrupt Vector Table*) are required to be left empty in current OS implementations. The other issue of particular interest is the fact that the *External Interrupts* (index 12) are all propagated via the same interrupt handler. This is clearly a better solution compared to the IA-32 approach, where the Exception and IRQ vectors are somewhat mixed, which often makes it difficult to identify what the real event initiator was. In contrast, in the case of the IA-64 platform, whenever the interrupt #12 occurs, the only thing to do is to query the IRQ controller about the actual event source identification, and to act accordingly. Furthermore, using the same interrupt handler for all external events makes it easier to implement the kernel-level synchronization if necessary.

Propagating the External Interrupts

The way of handling external interrupts on IA-64 platforms is quite similar to the IA-32 SMP approach described in *Section 3.1.5*, that is, using the APIC architecture in combination with a shared memory block. Only in this case are the used integrated circuits denoted as “streamlined”, resulting into the new names ‘LSAPIC’ and ‘IO-SAPIC’ respectively. On the other hand, what makes a difference is that the IA-64 processors do not provide traditional ‘IRQ’ pins as there are no legacy reasons for that.

3.3.6. Summary

As demonstrated in the previous chapters, the IA-64 architecture brings several advanced features. Besides those intended for the "programmer's convenience" (e.g. RSE), the fact that a chip-level scalability (in terms of internal execution unit quantities) is explicitly included in the EPIC concept makes it theoretically possible to choose (and pay for) different CPU models according to the expected system load. Yet, the question is, whether a non-oriented customer can really catch the point and act accordingly, especially when so far the messages and myths around IA-64 CPUs are rather high-flying. Furthermore, no miracles should be expected when a non-native code (like IA-32 or PA-RISC programs) is to be executed, as these applications can barely make any use of the IA-64 architectural enhancements. On the other hand, it is fair to say that the IA-64 architecture is ready to be used in SMP environment, simply because it's been designed for that purpose.

4. The performance insight

So far, both the IA-32 and IA-64 CPU families were described mainly with focus on the CPU internal properties, architecture and the programming model. As this description is only a part of the whole, let's now have a closer look at the surrounding components directly affecting the overall system performance.

As there exist plenty of CPUs across the entire Intel portfolio, for the purpose of this study we will restrict our scope to the latest CPU models intended for the server and workstation design. Namely these are:

- Pentium III,
- Pentium III Xeon,
- Pentium 4,
- Intel Xeon,
- Itanium, and
- Itanium 2 (McKinley).

Furthermore, should there exist several alternatives within each family (e.g. various clock rates and cache sizes), let's take the most advanced product into consideration, rather than exploring in-depth each and every alternative.

4.1. The Core CPU clock rate

The CPU 'core speed' is often a somewhat overloaded term. Even though it surely relates to the circuit-level technological maturity of a given processor, what is often cast aside is the fact that the main motivation for pushing the system clock frequency higher is, in many cases, not of actual real benefit for the user, but rather the complexity of the execution engine on itself.

The fundamental reason for maintaining a high frequency timing signal inside CPU is for the smooth instruction and data flow between all involved internal execution units. As each of them might require a different number of clock cycles to process the incoming data[21], in order to maintain a sustained and harmonized *event flow* across the entire CPU, it makes sense to drive each of the units at different speeds. In practice then, a common high-speed *core clock signal* is provided to all of them, and it is up to the unit in question to derive its own internal timing signal with respect to the complexity of the action to be performed.

[21] Storing an integer value into internal registers can be achieved within fewer clock cycles comparing to a float-point multiplication, for instance.

Dividing the clock signal is certainly an easier operation than implementing high-speed data buffers (or latch registers) between individual units whenever necessary. That's why a high frequency clock signal is used. On the other hand, from the user perspective, the actual *CPU core clock rate* is a rather minor factor, since it doesn't necessarily relate to the achievable processor data throughput. Besides, higher core frequencies mean higher power consumption, higher heat dissipation and often also a higher level of sensitivity on interfering electrical signals outside the CPU itself. Therefore, the frequency figures should be always read and used with certain cautions.

4.2. System bus technical parameters

In contrary to the *CPU clock rate*, the design and quality of the *system bus* is in direct connection to the achievable data throughput.

As with any parallel bus, the idea is that with every clock cycle as many data bits can be transmitted, as many data wires are present. For example, a CPU with 64-bit system bus can transmit eight bytes (64 bits) of information per each clock cycle, and so on. Unfortunately, the bus clock rate cannot scale in an unlimited way, because what we are talking about is a high-frequency signal distributed over copper wires. This, in order to be delivered successfully, must comply with certain modulation requirements and quality standards of the underlying wiring. Most remarkably then, the maximal bus length is often limited to relatively short distances.

In contemporary Intel CPU implementations, the system bus clock signal is often rated between 100 to 133MHz. This figure doesn't, however, necessarily indicate the final *symbol frequency*, as the actual signal encoding plays an important role as well. For instance, by using the so-called '*quad-pumped physical encoding schema*', many of the Intel processors achieve as much as four information bits transferred during each clock cycle. *Table 4.1* offers a quick summary of the studied CPU models.

		System bus parameters		
Model	**width**	**f(clk/data)**	**transfer**	**N-way**
Pentium III	64bit	100/400MHz	3.2 GB/s	2 CPUs
Pentium III Xeon	64bit	133/533MHz	4.3 GB/s	2 CPUs
Pentium 4	64bit	133/533MHz	4.3 GB/s	2 CPUs
Intel Xeon DP	64bit	133/533MHz	4.3 GB/s	2 CPUs
Intel Xeon MP	64bit	100/400MHz	3.2 GB/s	32+ CPUs
Itanium	128bit	266/266MHz	2.1 GB/s	512 CPUs
Itanium 2	128bit	100/400MHz	6.4 GB/s	4 CPUs

Table 4.1: *System Bus Technical Parameters*

As seen in *Table 4.1*, even though positioned as the leading technology of today, the first Itanium chip is clearly an evaluation release only, since it doesn't implement the quad-pumped data-encoding model. On the other hand, the supported system scalability is remarkable[22] and generally unmatched.

With regards to Itanium2, this product seems to be rather mass-market targeted. Supporting up to four CPUs is a quite natural choice, considering that at the beginning, servers with rather modest number of CPUs are to be expected.

Regarding the overall system bus throughput, the Itanium 2 processor is still slightly behind the other products, however, this might be only a short-term issue. On the other hand, the 128-bit system bus will definitely make the IA-64 CPUs a leader in massive memory-based operations like statistical and scientific calculations in a long run. Yet, the performance gain doesn't have to be the same in the case of the device I/O transactions, since not all the peripherals might be necessarily ready for 128bit operations for a long time.

4.3. On-chip cache memory

As shown in *Table 3.1*, Intel processors are delivered with different amounts of the on-chip cache memory. This memory represents an extremely fast CPU-local storage, used for storing frequently accessed data and instructions. This happens transparently from the programmer's perspective.

Server-type CPUs are typically equipped with 512kB of cache memory, or more. Since the on-chip cache is an associative type of memory, basically all the information needs to be indexed and internally organized, which increases requirements on higher core clock rates and/or increased intelligence on the circuit-design level. Therefore, designing a solid cache subsystem is a challenging task, and the cache size increments are rather modest, comparing to other CPU parameters and their growth.

Even with a small amount of cache, however, the performance gains are tremendous. Most remarkably this is valid for read-only types of memory objects like static constants and the instruction code. As these don't typically change with time, the CPU in charge can keep them in a local cache, and in this way eliminate unnecessary memory access cycles. The gained system bus capacity can then be used for manipulating the production data instead.

Ironically enough, the cache memory can also be contra-productive in certain situations. In order to maintain the cache-to-memory coherence, with

[22] Note, however, that typically only up to four CPUs can reside on a single bus. Systems with more than four CPUs have to be physically divided into several groups (sometimes called 'cells') with appropriate inter-bus bridging circuits in place.

the growing number of CPUs, the inter-CPU traffic increases as well. As this brings an additional data traffic for the system bus itself, in some cases the synchronization overhead might be to high and simply beyond acceptable limits.

Evaluating various access patterns and consequent performance contributions of the cache subsystem might deserve a separate study and yet we might not necessarily hit the reality. Thus, in the real life we should perhaps follow the advice of those who are directly involved in CPU design, as well as maintain our own realistic view into expectable results. Facing the usual situation, a very safe-side estimate would be that with doubling the cache size the overall system performance increases about 30-50% on a light-I/O system[23].

4.4. Chipset design

Intel CPUs are typically well documented, and practically anyone can use them in his/her design straight away. On the other hand, since there exist quite high requirements on the quality of the surrounding instrumentation, as a supplementary products Intel also offers the heavy-duty server and workstation infrastructure components commonly known as 'chipsets'.

As the name indicates, chipsets are groups of integrated circuits designed to work together and support a particular type of CPU. In the case of Intel-manufactured chipsets, the typical chipset architecture follows the schema depicted in *Figure 4.1*:

[23] The 'light-I/O' requirement is relevant in this case, because device I/O operations typically bypass the cache memory anyway.

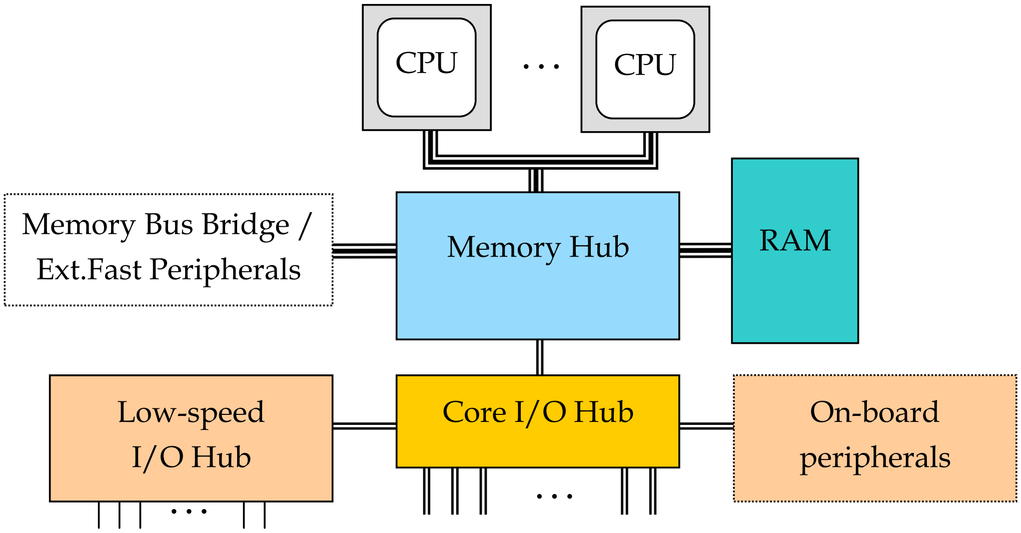


Figure 4.1: *Typical architecture of Intel chipsets*

As seen from the picture, components closely related to the CPUs are physically separated from the "real" I/O devices. This brings advantages that both the CPU and the I/O subsystems can be scaled independently from each other, namely not affecting electrical conditions on the underlying buses found on either side. In addition, the participating hubs may also take a supplementary role, like ECC checking or device co-ordination at the time of booting. On the other hand, by definition those hubs should be transparent (that is, not performing any data or address transformations) and designed with minimal latency. In other words, even though very important for the SMP design, data hubs are practically invisible from the CPU perspective[24].

4.5. Summary

In *Chapter 4* the CPU-external aspects in relation to the SMP platform design were outlined. As perhaps expectable, a great emphasis is put on the signal-level compatibility, and a transparent and non-blocking design of the intermediate signal switching. On the other hand, the core CPU clock rate, for instance, has been found to be a "relative technology improvement indicator" within a given processor family, rather than a reference quality figure across the entire portfolio.

A somewhat problematic issue is the justification of the used cache sizes. Even though of a great influence in uni-processor designs, a certain level of balance has to be established should a massive MP system be designed.

[24] Examples of real chipset schematics are available in section "Appendices".

Considering that there are typically only up to four CPUs attached to the same system bus, with the help from the operating system, one performance improving solution is to distribute tasks between individual CPUs so that the process synchronization traffic seldom crosses the inter-bus interface. Supplementary to that, technologies like Multi-Threading might be an interesting direction, since in this case the number of simultaneously executed threads increases, while the synchronization traffic between individual physical CPUs grows only when caused by the executed code. As one of the possible consequences, in the long run we might expect MTT CPUs with not only two, but possibly even more logical CPUs on a single board[25].

[25] The limiting factor in this case is the execution engine capacity as well as the system bus bandwidth.

5. Summary and conclusions

This thesis work started with the initial question of whether or not Intel has reached the technological maturity and concept quality required for building SMP types of computing systems.

As a conclusion, it is fair to say that the company in question has actually been in charge of SMP design for several years already, which, combined with the latest improvements in the integrated circuit arena, has helped them to position their products as architectural and quality leader on the market. As a clear affirmative indication of this fact, the world today is breathlessly awaiting the new Itanium family of CPUs as the presumed next generation computing platform. To prove and thoroughly evaluate the IA-64 concept (including the business aspects) might, however, take more time than any other technology before, because in some cases the IA-64 processors might be simply "too good" (and consequently also too expensive) to be used in a massive way. Based on the fact outlined in this study, *Figure 5.1* presents the likely expectable usage of individual Intel CPU models:

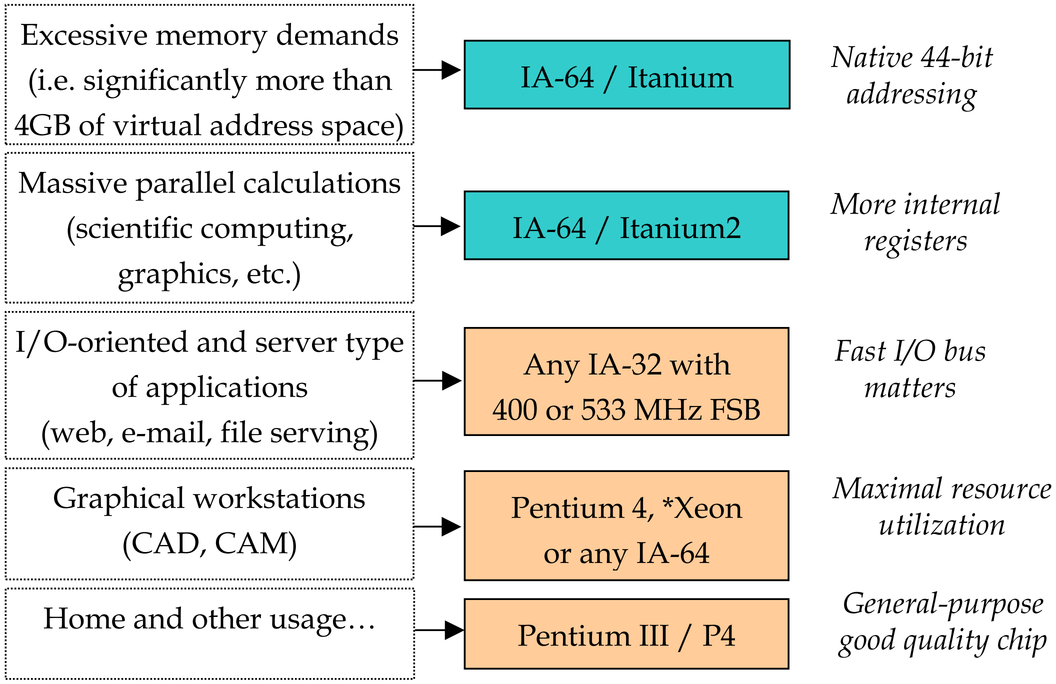

Excessive memory demands (i.e. significantly more than 4GB of virtual address space)
IA-64 / Itanium
Native 44-bit addressing
Massive parallel calculations (scientific computing, graphics, etc.)
IA-64 / Itanium2
More internal registers
I/O-oriented and server type of applications (web, e-mail, file serving)
Any IA-32 with 400 or 533 MHz FSB
Fast I/O bus matters
Graphical workstations (CAD, CAM)
Pentium 4, *Xeon or any IA-64
Maximal resource utilization
Home and other usage…
Pentium III / P4
General-purpose good quality chip

Figure 5.1: *Logical positioning of Intel-manufactured CPUs*

As seen above, there is no clear winner whatsoever. Furthermore, either of the CPUs can actually be used the same way in an UP as well as in an MP/SMP environment. The final choice is therefore a matter of the expected load, the type of the application to be operated, and naturally also the CPU price.

References

[APD_CPUs] AMD CPU portfolio.
http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118,00.html *

[COMP_ARCH] William Stallings: Computer Organization and Architecture (fifth edition). *Prentice Hall, 2000*

[EPIC] M.S.Schalansker & B.R.Rau. EPIC: Explicitly Parallel Instruction Coding. *IEEE Computer, pp.37-45, Feb.2000*

[HT&Win] Microsoft, Microsoft Windows-Based Servers and Intel Hyper-Threading Technology
http://www.microsoft.com/windows2000/server/evaluation/performance/reports/hyperthread.asp *

[HyperThreading] Intel, Introduction to Hyper-Threading Technology
Document: 250008-002, 2001
http://www.intel.com/technology/hyperthread/download/25000802.pdf *

[IA-32] Michal Bandejs: Mikroprocesory Intel, Pentium & spol.
Grada press 1994 (Czech Republic)

[IA-64] David Mosberger & Stéphane Eranian: IA-64 Linux Kernel
Hewlett Packard & Prentice Hall PTR, 2002

[IA-64/FSB] Samaras, Cheruki & Venkataraman. The IA-64 Itanium Processor Cartridge: Internal L3 Backside Bus. *IEEE Micro, Jan/Feb 2001*

[IA-64/VMM] Huck, Morris & Knies. Introducing the IA-64 Architecture: The IA-64 Virtual Memory Model. *IEEE Micro, Sept/Oct 2000*

* Links were found operational at the time this thesis work was written.

[IA-64WP] Intel & HP: IA-64 Architecture Disclosure White Paper http://www.hp.com/products1/itanium/infolibrary/whitepapers/archives/ia64_arch_wp.pdf *

[IB-ARCH] Intel Corp.: InfiniBand Architecture
http://www.intel.com/technology/infiniband/index.htm *

[IBTA] InfiniBand Trade Association: An introduction to InfiniBand
http://www.infinibandta.org/ibta/ *
http://www.infinibandta.org/specs/faq/ *

[Intel CPUs] Intel Corp.: History of Microprocessors.
http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/index.htm *

[Intel Museum] Intel Corp.: The Intel Museum.
http://www.intel.com/intel/intelis/museum/index.htm *

[Intel-MPA] Barry B. Brey. The Intel Microprocessors: Architecture, Programming and Interface. *Prentice Hall, 2000*

[Itanium FAQ] Hewlett-Packard: Itanium 2 - FAQ
http://www.hp.com/products1/itanium/faq/ *

[Linux Kernel] D.Bovet & M.Cesati: Understanding the Linux Kernel
O'Reilly, 2001

[MIPS] MIPS Technologies Inc. (home page)
http://www.mips.com/ *

[MOORE] Gordon E. Moore: Cramming More Components into Integrated Circuits. Electronics, *Volume 38, Number 8, April 19, 1965*
http://www.intel.com/research/silicon/moorespaper.pdf *

[MOT68K] Scott MacKenzie: 68000 Microprocessor.
Prentice Hall, 1995

[MPD] John P. Shen & Mikko Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. *McGraw-Hill 2002*

[OSP] Tanenbaum & Woodhull. Operating Systems: Design and Implementation. *Prentice Hall 1997*

[PA-RISC] Hewlett-Packard Systems & VLSI Technology Division: PA-RISC references http://www.cpus.hp.com/technical_references/parisc.shtml *

[PCISIG] PCI Special Interest Group: PCI Local Bus Specification rev.2.2. http://www.pcisig.com/ *

[RISC_Ref] John L. Hennessey & David A. Patterson: Computer Architecture – A quantitative approach
Morgan Kaufman, 1990

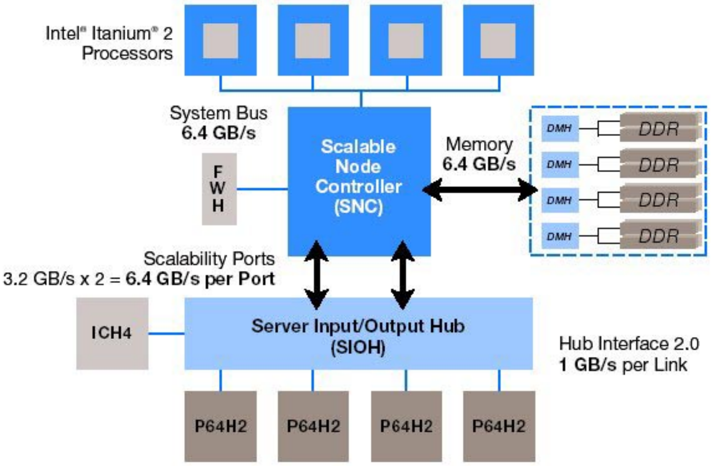
[SNW-IB] Storage Networking World: Despite Microsoft and Intel defections, InfiniBand backers still hopeful (article)
http://www.snwonline.com/plan/infiniband_09-16-2002.asp?article_id=158 *

[SPARC] SUN Microsystems: UltraSPARC Processors
http://www.sun.com/processors/index.html *

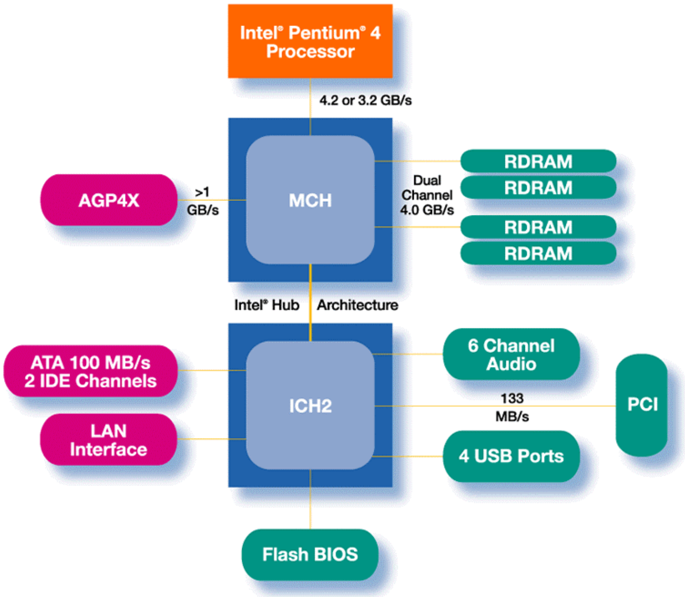
[VLIW] IBM Research: Introduction to the Very-Long Instruction Word (VLIW) Architecture
http://www.research.ibm.com/vliw/basic.html *

Appendices

Appendix A: Intel E8870 chipset (for Itanium 2)



Appendix B: Intel 850E chipset (for Pentium 4)


Intel® Pentium® 4
Processor
4.2 or 3.2 GB/s
AGP4X
>1
GB/s
MCH
Dual
Channel
4.0 GB/s
RDRAM
RDRAM
RDRAM
RDRAM
Intel® Hub
Architecture
ATA 100 MB/s
2 IDE Channels
LAN
Interface
ICH2
6 Channel
Audio
133
MB/s
PCI
4 USB Ports
Flash BIOS