ESPRIT LTR 21917 (Pegasus II)

Deliverable 2.1.2

Pentium Port Report

Stephen Early
University of Cambridge

July 1997

1 Introduction

This document describes a port of the Nemesis operating system to Intel Pentium based platforms. The majority of personal computers sold are based on Pentium-compatible processors, and share the same system architecture (commonly known as the 'PC' architecture).

Some background of the nature of Pentium systems will be described, and an introduction will be given to relevant parts of the processor and PC architecture. Then the Pentium port will be described, describing in the context of these architectural features the various decisions made. This is followed by a description of the tools used in the port.

Finally I conclude with a description of some of the issues raised by the Pentium work and discuss their effects on the continuing development of Nemesis.

2 Pentium architecture

The Intel Pentium is a 32-bit processor which has a superset of the features of the earlier 8086, 80186, 80286, 80386 and 80486 processors. It is used as the processor in most PC architecture machines currently on the market. See section 3 for a description of the PC architecture.

Like all of the members of the Intel Architecture family of processors, the Pentium preserves binary compatibility with earlier members of the family. However, in order to obtain the best performance different optimisations must be made in both the operating system design and compiled code.

The Pentium and Pentium Pro are described in detail in [2, 3].

2.1 Processor modes

As of the 80286, the Intel architecture supports two distinct modes of operation known as *real-address mode* and *protected mode*. Real-address mode is provided for backwards compatibility with earlier Intel architecture processors, and is the default mode on initialisation. Protected mode is the native operating mode of the processor, and allows all of the instructions and architectural features to be used. All of the following sections describe the behaviour of the processor while it is in protected mode.

2.2 Address spaces

The Pentium has a segmented address space. Memory references for code, data and stack are made through the appropriate *segment registers*. These registers contain an index into one of two tables, the *global descriptor table* or the *local descriptor table*. The *virtual address* within the segment is translated using the information in the descriptor to a *linear address*. Finally the linear address is translated using the page tables to a *physical address*.

It is necessary to define at least two segment descriptors to enable code to run in protected mode; one for code access and one for data and stack access. If protection is to be implemented then four descriptors must be defined; two for 'user mode' and two for 'kernel mode' memory accesses.

The linear addresses and lengths of the base of the global and local descriptor tables are stored in two registers, the GDTR and the LDTR. These registers can only be changed when the processor is in its most privileged mode.

2.3 Protection

The Pentium recognises four privilege levels, or 'rings' numbered 0–3. Level 0 is the most highly privileged level. The current privilege level is determined by the privilege bits in the current code segment selector.

Coarse-grained control over access to memory can be gained using bits in segment descriptors. These can describe segments as read/write, read only, or execute only, as well as having some other attributes such as 'expand-down' and an 'accessed' flag. The accessibility of segment descriptors is determined by the *descriptor privilege level*; this is compared with the *requestor privilege level* and the current privilege level when an attempt is made to load a selector into a segment register. If an invalid request is made then a protection exception is generated.

Finer grained control over access is managed using the page tables. Each page table entry has two bits which control access to the page; one bit restricts access based on the current privilege level, and the other is a write-protect flag.

The current privilege level also controls access to IO ports, and the ability to use some registers and instructions.

2.4 Tasks

The Pentium insists on the concept of the current *task*. A data structure called the 'task state segment' (TSS) holds information about the task. Task state segments are accessed through entries in the global descriptor table.

The TSS holds enough information to be able to restore a task. Part of it may be written to by the processor; this part holds the general purpose registers, the current segment selectors, the EFLAGS register, the instruction pointer and a field to link to the 'previous' TSS. The other part is set up by the operating system, and holds a variety of information:

- The segment selector for the task's local descriptor table.

- The base address of the page table to be used by the task.
- Stack segment selectors and stack pointers for stacks to be used by privileged code in the task.
- A debugging flag which can be set to cause an exception when a task switch occurs.
- An IO permission bitmap describing the IO ports that the task is allowed to use.

The TR register holds information about the current TSS. It can be loaded with a TSS descriptor using the LTR instruction. Internally the processor caches the linear address of the base of the TSS; this is not accessible to software.

2.5 Interrupts

Interrupts and exceptions are handled by looking up a descriptor in the *interrupt descriptor table*. Each interrupt or exception is allocated a *vector* between 0 and 255. Interrupts may be caused by hardware, in which case the vector is supplied by the interrupt controller during the IACK cycle, or by software using the `int` instruction. Exceptions are generated internally by the processor.

A number of types of descriptor are valid in the interrupt descriptor table, but Nemesis only uses interrupt gate descriptors. When an interrupt occurs and an interrupt gate descriptor is found by the processor, interrupts are disabled, the stack is switched to the appropriate stack for the privilege level of the descriptor (always 0 in Nemesis), and the handler specified in the descriptor is called.

2.6 Registers

Registers in the Intel architecture can be divided into two main groups; those used by user-level code, and those used for system management. There is one register, the EFLAGS register, that has some bits that are used by user-level code, and some that can only be modified by privileged code.

The system registers are shown in Table 1, and the user registers are shown in Table 2. Note that it is possible to refer to parts of the four general purpose registers EAX–EDX by calling them AX, BX, etc. to access the low 16 bits, and AH, AL, BH, BL, etc. to access the upper and lower 8 bits of the low 16 bits. This is for compatibility with the 80286 and earlier processors.

3 PC architecture

The current PC architecture is a direct descendent of the original personal computer designed by IBM in the early 1980s. Many extensions have been added, but backwards compatibility has always been preserved. This preservation of backwards compatibility has made the architecture quite peculiar in several respects, some of which will be described below.

From the point of view of a Nemesis port, the PC architecture has two interesting features:

- The physical address space for memory is not completely contiguous. RAM starts at physical address 0 and ends at 640k; this area is referred to as the 'base memory area'.

Register	**Description**
EFLAGS	Miscellaneous flags, mostly controlling the state of the current task
CR0	Flags controlling operating mode and states of the processor
CR2	Contains the most recent page fault linear address
CR3	Contains the physical address of the level 1 page table, and some flags
CR4	Flags controlling architectural extensions
DB0–7	Registers controlling debugging
GDTR	Global Descriptor Table base and limit register
LDTR	Local Descriptor Table base and limit register
IDTR	Interrupt Descriptor Table base and limit register
TR	Task State Segment selector

Table 1: System registers in the Intel architecture

Register	**Description**
EFLAGS	Results of the last instruction
EAX, EBX, ECX, EDX	General purpose registers
ESI, EDI, EBP	General purpose registers
ESP	Stack pointer
EIP	Instruction pointer
CS	Code segment selector
DS	Data segment selector
ES, FS, GS	Alternative data segment selectors
SS	Stack segment selector

Table 2: User registers in the Intel architecture

There is then a gap in the address space between 640k and 1M which is used for ROM, devices, etc. RAM restarts at 1M, and continues up to the limit of installed RAM. This area is called the 'extended memory area'.

- There are separate memory and I/O address spaces. In part this is a feature of the Intel architecture processors; however, the distinction is preserved in PC architecture machines that do not have processors with separate memory and I/O address spaces.

Part of the I/O and memory spaces address devices on an ISA bus. While any devices on this bus may be add-in cards, there are several devices which are expected to be present, and are vital to the operation of the machine:

- Two 8259A programmable interrupt controllers[1].
- Two 8237 DMA controllers. These devices, along with a couple of registers provided by another device, allows DMA from the low 16Mb of physical memory to devices on the ISA bus.
- Two 8254 programmable timers. Each timer device has three timers. The output of one of the timers is connected to interrupt request line 0 of the first interrupt controller.
- A DS1287a-compatible real-time clock and non-volatile storage device.

Information on the above devices is available in manufacturers' data sheets. It is also available in books on the PC architecture; one used during the development of Nemesis is [5].

4 The Nemesis Port

4.1 Bootstrapping

A PC can be booted in many different ways. The usual methods involve loading a sector (512 bytes) from either floppy disk or hard disk into memory and running it. Alternatively control can be passed to code in a BIOS extension ROM on a plug-in card like a network card.

No matter how the initial code is loaded, control is passed to it with the processor in real-address mode. This is to retain compatibility with legacy operating systems like MS-DOS. The code is responsible for loading the rest of the boot loader using BIOS calls to access the boot device. The boot loader can then load the operating system image and start it.

Several adequate boot loaders have already been written, and are available under the GNU General Public License. Many of these were designed to load Linux, so the Nemesis operating system image file has been made compatible with Linux operating system images.

A Nemesis image has three sections, referred to as the boot sector, the setup code and the system image. If an image is written directly to a floppy disk then it will load itself and run when the floppy is booted. Alternatively, another loader program can be used to load the setup code and system image from other media.

If the image is being booted from floppy then the BIOS loads the first 512 bytes at 0x7c00 and jumps at it in real-address mode. This code copies itself to 0x90000, loads the setup code at 0x90200 and the system image at 0x100000.

[1]These may not be present in SMP machines, but will be emulated.

If the image is being loaded by some other loader, that loader reads the setup code size from a well-known location in the boot sector, loads the setup code at 0x90200 and the system image at 0x100000. The setup code is then jumped to in Real mode.

The setup code stores some values from the BIOS like memory size and hard disk parameters in well-known locations starting at 0x90000, sets up the two 8259A interrupt controllers, switches to Protected mode and jumps at the start of the third section.

In the Computer Laboratory we originally used a simple network boot loader program to start Nemesis: the boot loader was loaded from floppy disk, and then used `bootp` and `tftp` to load a Nemesis image over the network. This process was rather slow, so now a pre-built Nemesis kernel is loaded from the hard disk of the test machine using LILO [1]. This kernel loads another Nemesis image using either NFS or TFTP and starts it using a `chain` system call that was added for this purpose.

4.2 The NTSC

There are four main components which require consideration when porting Nemesis to a new processor. These are initialisation, the NTSC interface (system calls), interrupts and timer code. These will now be described.

4.2.1 Initialisation

When the 32-bit protected mode code is entered, the processor is not in a suitable state to run Nemesis. The initialisation code in the NTSC sets up a GDT with seven entries (three code segment descriptors, three corresponding data segment descriptors, and a TSS descriptor). The TSS is initialised minimally; only the ring 0 stack segment selector and base address fields are used. The IDT is initialised with descriptors for all of the processor internal exceptions, hardware interrupts, and system calls. Finally, the generic 'Primal' routine is called in user mode to continue initialisation.

Currently the processor is left in physical address mode when Primal is started; it is up to the Intel-specific memory management code in user space to enable virtual addressing. This may change in the future, when new memory management code is integrated with the Pentium port.

4.2.2 Console output

Console output from the NTSC is provided using a trivial serial driver that accesses a UART in polled mode. Use of this serial driver involves a busy wait in the NTSC with interrupts disabled, and so is only used when it is the only means by which information can be output.

It is possible to access the NTSC console output code from user mode using a system call. This is useful in two situations: firstly during system startup, before the serial driver has been initialised. Secondly, during domain initialisation before the domain has had a chance to establish IDC connections.

When the video BIOS has finished initialisation the graphics chipset is left in an 80×25 character text mode with the start of screen memory at a well-known address. It is possible to use the display without any further initialisation. The current NTSC puts a banner at the top of the screen to enable people physically at the console to see which image the machine is running.

4.2.3 Interrupts

As described earlier (section 2.5), Intel architecture processors handle interrupts and exceptions through the same mechanism. Each interrupt or exception is given a *vector* from 0 to 255. Vectors 0–31 are reserved by Intel for internal processor exceptions.

During initialisation, the two 8259 interrupt controllers are programmed to map the 16 possible interrupts to vectors 32–47. The handlers for those interrupts are all very similar[2]; they call the `k_irq()` routine with the interrupt number as an argument.

`k_irq()` performs a few sanity checks (making sure that an interrupt didn't occur while interrupts were supposed to be disabled, for example), masks the interrupt in the 8259 and finally acknowledges it. This prevents the interrupt from occurring again until the appropriate driver has had a chance to deal with the device. The interrupt is looked up in a table, and the appropriate interrupt stub is called, if one has been registered. The stub is passed the address of the `k_event()` routine and a pointer to its private data. `k_event()` can be used to send an event to the appropriate device driver domain.

4.2.4 System calls

System calls in Intel Nemesis are also handled through the IDT; they are allocated vectors starting at 48. System calls are made using the `int` instruction, with arguments passed in the general purpose registers. The currently defined system calls are shown in Table 3.

Some system calls can only be made by privileged domains. Access to these is controlled by the DPL[3] field in the interrupt descriptor. Normal Nemesis domains run at ring 3; privileged domains spend part of their time running at ring 2, and must be in this state in order to make privileged system calls.

Interrupt gate descriptors are used in the IDT for system calls, so interrupts are disabled automatically during system calls and NTSC code is run on the NTSC stack. Almost all of the system call stubs call the `save_context` routine to store the processor context that the processor has left in registers and on the stack in the appropriate context slot. They then call the `k_syscall()` routine with the system call number as an argument.

Eventually it is intended to implement some of the system calls directly in assembler, so that the call to C and, for some calls, the context save may be omitted.

4.2.5 Timer

Local scheduling time in Nemesis is represented as a 64-bit number of nanoseconds since the machine booted. The timer code in the NTSC is required to provide the scheduler with accurate time, and to permit the setting and clearing of an alarm timer.

The PC platform has a number of timers as standard. There is an 8254 programmable timer chip, and a DS1287a real-time clock chip that can be programmed to generate interrupts at a particular rate.

Initial work on Intel Nemesis programmed the real-time clock chip to generate interrupts at 8192Hz[4] (its fastest possible rate) to keep the notion of 'current time' up to date, and attempted to use the programmable timer as an interval timer. This failed because of interrupt priority problems; the programmable timer is wired to interrupt 0, the highest priority

[2] Except for the countdown timer and ticker interrupts, numbers 0 and 8, which get special handlers.

[3] Descriptor Privilege Level

[4] This corresponds to a period of $122.07\mu s$.

Vector	Name	Description
48	Halt	Halt the system
49	RFA	Return from Activation
50	RFA_Resume	Return from Activation and Resume context
51	RFA_Block	Return from Activation and Block
52	Block	Block
53	Yield	Yield
54	Send	Send an event
55	swpipl	Enable/disable interrupts
56	unmask_irq	Unmask an interrupt
57	k_event	Send an event (not used)
58	actdom	Explicitly activate a domain
59	setpgtbl	Set page table base address
60	flushtlb	Flush the TLB
61	enablepaging	Enable virtual addressing
62	mask_irq	Mask an interrupt
63	set_debug_registers	Program the debugging registers
64	entkern	Enable privileges
65	leavekern	Disable privileges
66	callpriv	Use a callpriv
67	putcons	Output to the console for debugging only
68	chain	Start another Nemesis image

Table 3: Current Intel Nemesis system calls

interrupt, and the real-time clock is wired to interrupt 8. The scheduler would occasionally get into a state where it asks for an interrupt after a very small interval of time. The interval timer interrupt would occur almost immediately, but because the scheduler's idea of 'current time' has not changed the same small interval would be requested again. The continual processing of interval timer interrupts prevents ticker interrupts from being dealt with.

The first working implementation of the timer ignored the real-time clock chip, and programmed the other timer to generate interrupts at 8192Hz. Interval timing was performed in software, with a minimum interval of 122.07μs.

Starting with Pentium processors, Intel introduced the `rdtsc` instruction. This returns a 64-bit time stamp[5]. If this instruction is present then more accurate timer code can be used. A calibration is performed at NTSC initialisation time to determine the number of picoseconds per single time stamp. The real-time clock chip is then programmed to generate interrupts at 2Hz. The handler for the real-time clock interrupt records the value of the time stamp counter at the time of the interrupt, and the current scheduler time. Whenever the current scheduler time needs to be known, it is calculated using the value stored at the last ticker interrupt and the current value of the time stamp counter. This enables the current scheduler time to be determined very accurately.

Using the low-frequency ticker and time stamp counter frees up the programmable timer, so this can now be used once more as an interval timer for the scheduler. We have found that the timer is unreliable if intervals below 1μs are requested, so this has been made the minimum possible interval in software.

4.3 User space code

There are various pieces of assembler code which need to be written for user space Nemesis code. These include the system call stubs, the thread startup code, and `setjmp()/longjmp()`. All of these were straightforward.

4.4 Privilege levels

The current 'ring' is determined by the privilege level of the current code segment selector. Nemesis defines three of these, which are identical apart from the privilege level. Levels 0, 2 and 3 are defined.

User space code usually runs in ring 3. However, if a domain has the *kernel privilege* ('k') flag set in the read-only part of its control block, it can use a system call to increase its privilege to ring 2. This enables the code in the domain to use privileged system calls and access any part of the virtual address space.

4.5 Public Information Page

It is convenient for the NTSC to be able to share information with user-space programs. One way of providing this information would be to add a system call to access it. This is not the best way, however, because the information needs to be accessed frequently and the overhead of a system call is undesirable

[5]In practice this time stamp is the number of processor cycles, although Intel only guarantee it to be a monotonically increasing value.

Instead we have defined a page of memory at a well-known virtual address to contain 'public' NTSC data. A macro is provided to access data in this area. The following are some of the things included in the PIP:

- Pointer to current DCB read-only and read-write areas.
- Processor feature information.
- Current scheduler time.

4.6 Pervasives register

The 'Pervasives register' is notionally a callee-saves register that can be accessed using the PVS() macro. It is a part of the current processor context like any other register, and so is saved and restored by the NTSC and by setjmp/longjmp. It is usually used as a pointer to a structure containing per-thread state like stdin, stdout, the current Thread closure, etc. Conventionally this structure is defined in `Pervasives.if`, although user code is free to make PVS() point to anything[6].

User-level code uses the pervasives to fetch commonly-used pointers like the Event system closure and the root of the thread's namespace. The alternative would be to look these up in the namespace each time they were required, but then of course the pointer to the root of the namespace would have to be passed as a parameter to every procedure.

On most architectures, PVS() is implemented using compiler options to make it access a designated register directly. On Intel this is not sensible because there are very few general purpose registers available. Instead, the current Pervasives register value is stored in the read/write section of the DCB. The PVS() macro accesses this value by dereferencing the pointer to the current DCBRW that is stored in the PIP (see section 4.5).

The context save and restore code in the NTSC, and the implementation of setjmp/longjmp have been modified for Intel Nemesis to treat the Pervasives register as part of the current processor context.

4.7 Memory protection and paging

Initial work on the Pentium port of Nemesis has been done with a one-to-one mapping between virtual and physical addresses. The processor's paging mechanism has been used only to provide memory protection.

Context switches and protection domain switches occur very often in Nemesis, so the implementation attempts to minimise the number of TLB flushes as much as possible. The processor's page table is initialised with the global permissions for each page. When a domain attempts an access to a page that requires more than the global permissions, a page fault occurs and the NTSC can alter the page table to allow the access. A list of all the pages modified in this fashion is kept, and when the protection domain is next switched the list is used to return the page table to its default state and flush only those TLB entries which are affected.

[6] For the importance of the pervasives on Nemesis see [4]

4.8 Floating point support

A floating point context on Intel is large[7] relative to the standard Intel context, and takes a relatively long time to save and restore. Very little code in Nemesis uses the floating point unit, so it is useful to defer floating point context save and restore until it is known that it will be needed.

When a context switch is performed, a flag is set in CR0 which will make the processor generate a Device Unavailable exception whenever a floating point instruction is encountered. The NTSC traps this exception and performs the floating point context switch.

Once the NTSC has noticed that a domain is performing floating point operations, a flag is set in the domain's DCB. User space code like `setjmp()` and `longjmp()` can use this flag to decide whether to bother saving and restoring floating point state.

4.9 Intel Architecture family compatibility

All of the Intel Architecture family of processors are backwards-compatible with previous models. However, Nemesis can use some processor features which were introduced in the Pentium or Pentium Pro. In later versions of the 80486 Intel introduced the `cpuid` instruction. This enables software to work out which features the processor supports and change its behaviour accordingly.

In Nemesis the processor features are read during NTSC initialisation, and are recorded in the PIP (section 4.5). Currently the main users of this information are the timer code (section 4.2.5), which changes behaviour depending on whether the `rdtsc` instruction exists, and the accounting code which also uses `rdtsc`. However, future user-space programs may read this information to detect the presence of architecture extensions like MMX.

4.10 Current status

The current status is that the core of the Nemesis system runs on Pentium-based machines. Memory protection is provided, but paging is not. The following device drivers exist and have been tested:

- Serial ports based on 16450 and 16550 UARTs.
- 3c509 and DE4x5 Ethernet cards.
- DEC ATMworks 350 ATM card.
- ATM cards based on the IDT77201 NICStAR chip.
- Zeitnet ZN1221/ZN1225 ATM cards.
- S3-968 based graphics cards with IBM or TI RAMDACs.
- Creative Labs SoundBlaster 16 compatible sound cards.
- The standard PC keyboard and PS/2 mouse controller.
- NCR53C8xx-based SCSI cards.
- SCSI disks.

[7] 108 bytes.

4.11 Lessons Learned

It was discovered early on in the port that the previous version of Nemesis made several assumptions about a 64-bit word size. The type system uses 64-bit values for typecodes, and the 'Type.Any' type has a 64-bit pointer field. This caused problems with the compiler and linker, which could not extend a 32-bit value to 64 bits at build time.

The manipulation of time as a 64-bit quantity in the scheduler is inefficient. This has led to consideration of restructuring time within Nemesis so that the NTSC deals entirely in whatever the natural resolution of the machine is, and library code is provided for applications.

This would be a significant change and so may never actually be effected. Further evaluation will be performed when profiling is available later in the project.

4.12 Tools

The tools used for the Pentium port are GCC version 2.7.2, and GNU binutils version 2.7. The `nembuild` program used to create Nemesis kernel images is written using BFD 2.7.0.2. The `intelbuild` program used to join the three parts of the Nemesis image file together is derived from Linux. All of the tools are hosted on Intel Linux.

5 Conclusion

Machines based on the Pentium and other Intel architecture processors are important because they are commonly and cheaply available. Nemesis has been ported to these machines. Further work and evaluation of this port will continue throughout the Pegasus II project. In particular the memory management system for Nemesis is being designed with Intel processors in mind along with Alpha and ARM.

References

[1] Werner Almesberger. *LILO - Generic Boot Loader for Linux*, May 1996.

[2] Intel. *Pentium Pro Family Developer's Manual Volume 2: Programmer's Reference Manual*, December 1995.

[3] Intel. *Pentium Pro Family Developer's Manual Volume 3: Operating System Writer's Guide*, December 1995.

[4] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996. Article describes state in May 1995.

[5] Hans-Peter Messmer. *The Indispensable PC Hardware Book*. Addison-Wesley, 1995. ISBN 0 201 87697 3.