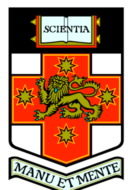
The IA-32 processor architecture

Nicholas FitzRoy-Dale

Document Revision: 1 Date: 2006/05/30 22:31:24

nfd@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia



1 Introduction

This report discusses the most common instruction set architecture for desktop microprocessors: IA-32. From a programmer's perspective, IA-32 has not changed changed significantly since its introduction with the Intel 80386 processor in 1985. IA-32 implementations, however, have undergone dramatic changes in order to stay competitive with more modern architectures, particularly in the area of instruction-level parallelism.

This report discusses the history of IA-32, and then the architectural features of recent IA-32 implementations, with particular regard to caching, multiprocessing, and instruction-level parallelism. An archtectural description is not particularly useful in isolation. Therefore, to provide context, each aspect is compared with analogous features of other architectures, with particular attention paid to the RISC-style ARM processor and the VLIW-inspired Itanium.

2 A brief history of IA-32

IA-32 first appeared with the 80386 processor, but the architecture was by no means completely new. IA-32's 8-bit predecessor first appeared in the Datapoint 2200 programmable terminal, released in 1971. Under contract to produce a single-chip version of the terminal's multiple-chip TTL design, Intel's implementation, the 8008, was not included in the terminal. Intel released the chip in 1972. The first "x86" chip, the Intel 8086, released in 1978, is almost binary-compatible with programs written for the 8008.[1] Successors to the 8086, including the 808186 and 80286, retained the 8086 instruction set. Figure 1 shows a brief chronology.

Intel's commitment to backwards compatibility in IA-32 has left the architecture with a number of features that appear awkward when compared with more modern designs. In particular, the CISC nature of the instruction set, the relative scarcity of registers, and IA-32s complex addressing modes threaten to reduce the overall performance of IA-32 implementations, and thus are all overcome in hardware.

2.1 Segmentation

The original memory access mode for IA-32 is called *segmented mode* (IA-32 has a number of other processor modes, the most important of which is protected mode, described below).

In this mode, applications refer to memory using a combination of a 16-bit *segment register* and a 16-bit *offset register*. Segments are overlapped in this mode to give a maximum addressable memory of 1MB.

Segmentation is preserved in IA-32 protected mode, and in the 64-bit extension to IA-32 known as AMD64 or EMT64. In these modes, segment boundaries are not defined by the hardware. Instead, the segment registers (`CS`, `DS`, `ES`, `FS`, and `GS`) serve as an index into the *global descriptor table*, which defines the extent of the segment. Segments are defined on the virtual address space, and thus segment translation occurs before paging.

Segments are not generally used. Segmentation cannot be disabled, but it may be effectively removed by creating a single entry in the global descriptor table, covering the entire 4 gigabyte address space. Some operating systems take advantage of segmentation to optimise task switching. The L4/Pistachio operating system can use segmentation to map several program address spaces into a special shared *small space page table*.[2] If several processes are sufficiently small, they can share the use of this page table, with the benefit that the page table need not be changed on a context switch between programs which share it. This optimisation, called *small spaces*, improves IA-32 context switch performance, because IA-32, unlike ARM, does not have a tagged TLB and is thus obliged to flush the TLB when the current page table changes.

[1] The Intel 8080, released in 1974, has a 16-bit external bus but is otherwise identical to the 8086.

[2] Implementations may use a global portion of the standard process page table instead.

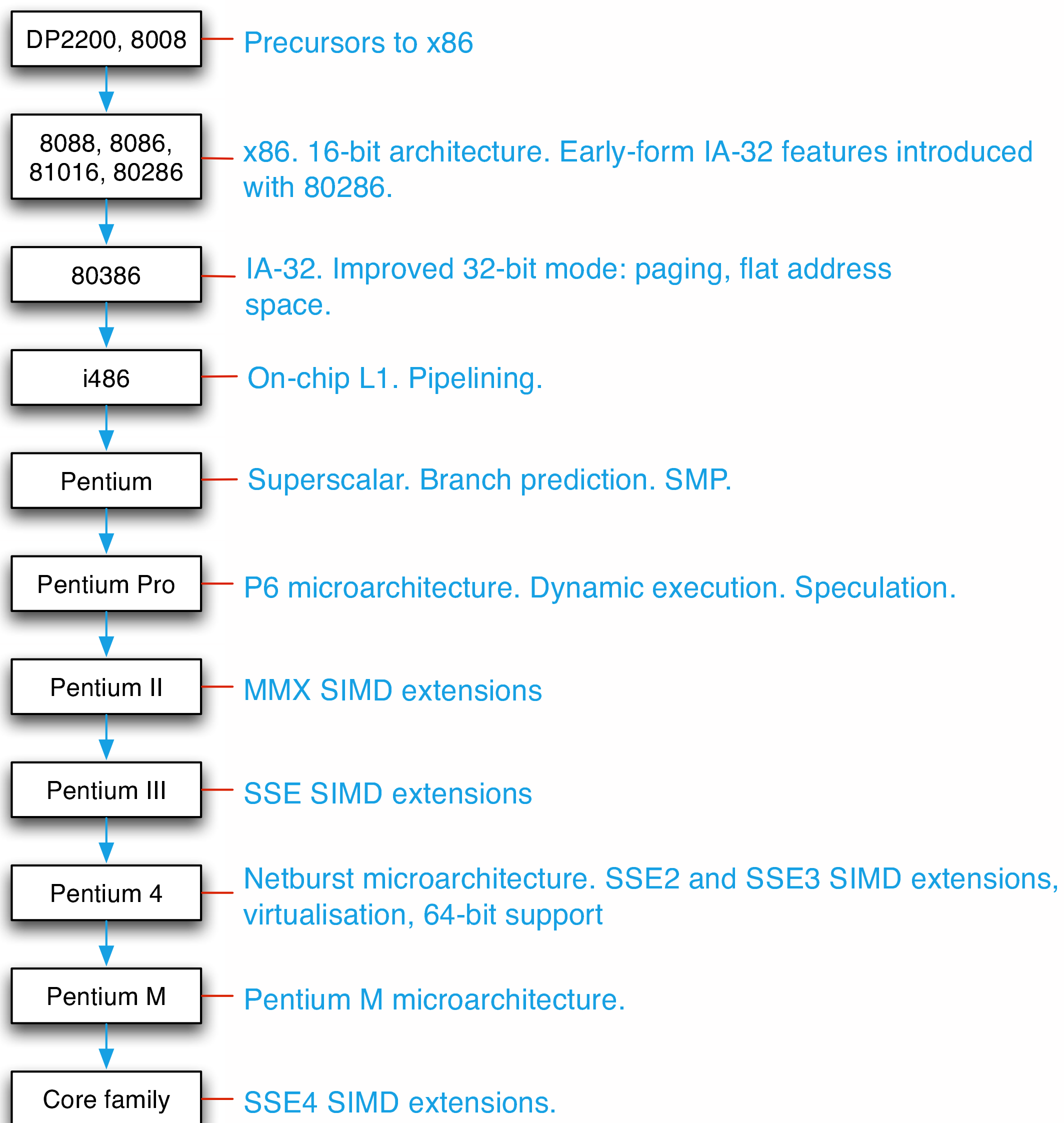


Figure 1: The 80x86 processor succession

The key features of IA-32, distinguishing it from earlier architectures, are *protected mode*, 32-bit registers, a 32-bit address space, and *paging*.

2.2 Protected Mode

Modern IA-32 processors are capable of operating in a number of *modes*. All IA-32 implementations power-up in *real mode*, an 8086-compatible segmented mode. The 80286 introduced a new processor operating mode, *protected mode*, that provided access to up to 16MB of memory. The mode was not successful, primarily because real-mode programs could not run in protected mode unless they were written with that goal in mind, and the processor did not provide the ability to switch back to real mode once it had entered protected mode.

The 80386 provided a new mode, *virtual 8086 mode*, in which real-mode programs could run while the processor was in protected mode. This, combined with a more flexible segmentation scheme and a larger addressible memory space (32 bits rather than 24, bring the total addressible memory to 4GB from 16MB) has made 80386 protected mode the mode of choice for all modern operating systems. Later IA-32 implementations have not made significant changes or enhancements to protected mode.

2.3 Paging

The 80386 introduced paging support, with a page size of 4KB. An additional 4MB page size was added with the Pentium processor. All IA-32 implementations use a hardware-loaded TLB and, thus, a hardware-walked page table. Only one page table type is supported: a two-level hierarchical type (note that IA-32 implementations with AMD's 64-bit extensions running in 64-bit mode use a hardware-walked four-level hierarchical page table instead). Before paging is enabled, a special *control register*, CR3, is set to the physical memory address of the *page directory*, the first level of IA-32's two-level page table structure. TLB lookup is then performed by the hardware. Figure 2 illustrates the page-lookup mechanism of IA-32.

2.4 Microarchitectures

Over IA-32's long history, Intel has introduced a number of different *microarchitectures*. Some microarchitectures, such as the Netburst microarchitecture of the Pentium 4, are a complete redesign; others, such as the Pentium M microarchitecture, are more straightforward evolutions of previous generations. However, since its introduction in with the Pentium Pro, the P6 microarchitecture has seen unprecendented success, forming the core for the Pentium Pro, Pentium II, Pentium III, and (in modified form) Pentium M and Core processors. This report thus focuses on this microarchitecture.

The key difference between the Core and Pentium M microarchitectures and their predecessor, Netburst, is lower power consumption and increased thermal efficiency.

3 ISA

3.1 Architectural registers

IA-32 has very few architectural registers. Figure 3 shows IA-32's integer registers. Registers on the right-hand side of the figure, with the exception of FS and GS, were present in the original 8086. IA-32 adds the *extended* registers EAX, ECX, EDX, EBX, EBP, ESP, ESI, EDI, EIP, and EFLAGS, as well as two additional segment registers FS and GS.

Originally all registers were special-purpose. For example, AX was originally an accumulator and could only be used as such. IA-32 lifted many of the restrictions on register usage, but some remain. For example, some instrutions assume that a pointer in the EBX register is relative to the segment indexed by DS. In practise, 6 registers are available for general-purpose use, far fewer than the number available

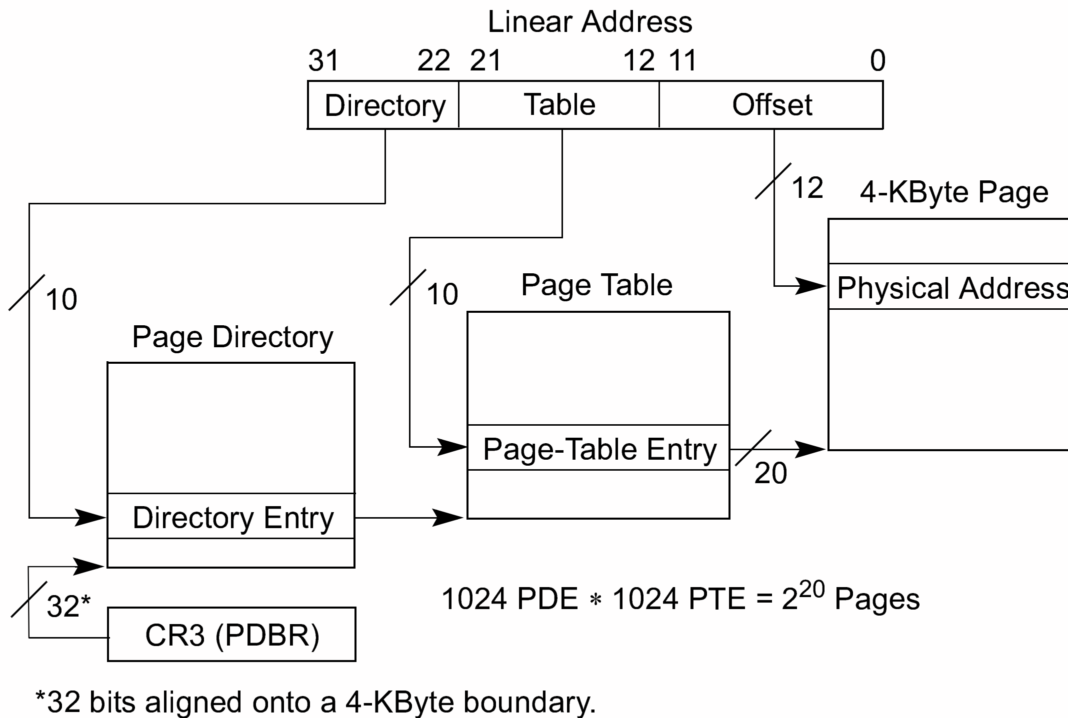


Figure 2: IA-32 paging mechanism (Intel)

in the ARM or IA-64 architectures. The practical result of this register pressure is that IA-32 programs tend to make more frequent use of the stack for temporary storage.

3.2 Instruction set

Despite its support for a number of legacy instructions (such as string operations), the number of instructions used in real IA-32 programs is relatively low. The majority of instructions added to the instruction set after the introduction of the 80386 are special-purpose. For example, Intel has introduced four SIMD extensions to the architecture, discussed later. There is little benefit, therefore, for most general-purpose programs to target anything other than the instruction set presented by an 80386.

The 80386 instruction set contains 201 instructions; an informal check showed that about 150 of those are used. Interestingly, instruction frequency follows a logarithmic scale, with 50 percent of all code consisting of the three instructions `mov`, call, and `jmp`, being a general purpose data-transfer instruction, a procedure call, and an unconditional jump, respectively. 99 percent of all code consists of just 45 instructions, an architectural feature that Intel no doubt makes use of internally to improve processor efficiency; see the section on the P6 instruction decoder below.

Figure 4 shows instruction frequency for three architectures.

Instruction encoding is very complicated, particularly when compared with the RISC ARM, for which all instructions are normally 32 bits((ARM also supports a different mode of operation, *thumb mode*, optimised for code density rather than speed, in which all instructions are 16 bits long.)) and the RISC-like IA-64, which uses 128-bit *bundles*, each consisting of a header and three 41-bit instructions. IA-32 instructions can be between 1 and 17 bytes long. Recent extensions to the instruction set (such as the SIMD extensions discussed below) only consume two or three bytes, operands excluded, and the large instruction length is largely due to IA-32's support for prefixes and suffixes which modify the behaviour of the instruction in some way. For example, IA-32 supports repeated execution of a single

32	15	7	0	
EAX	AH	AL		Accumulator
ECX	CH	CL		Count
EDX	DH	DL		Data
EBX	BH	BL		Base of data
EBP	BP			Base of stack
ESP	SP			Stack pointer
ESI	SI			String source idx
EDI	DI			String dest idx
EIP	IP			Instruction pointer
EFLAGS	FLAGS			CPU flags
	CS			Code segment
	SS			Stack segment
	DS			Data segment
	ES			Extra data segment
	FS			Extra data segment 2
	GS			Extra data segment 3

Figure 3: IA-32 integer registers

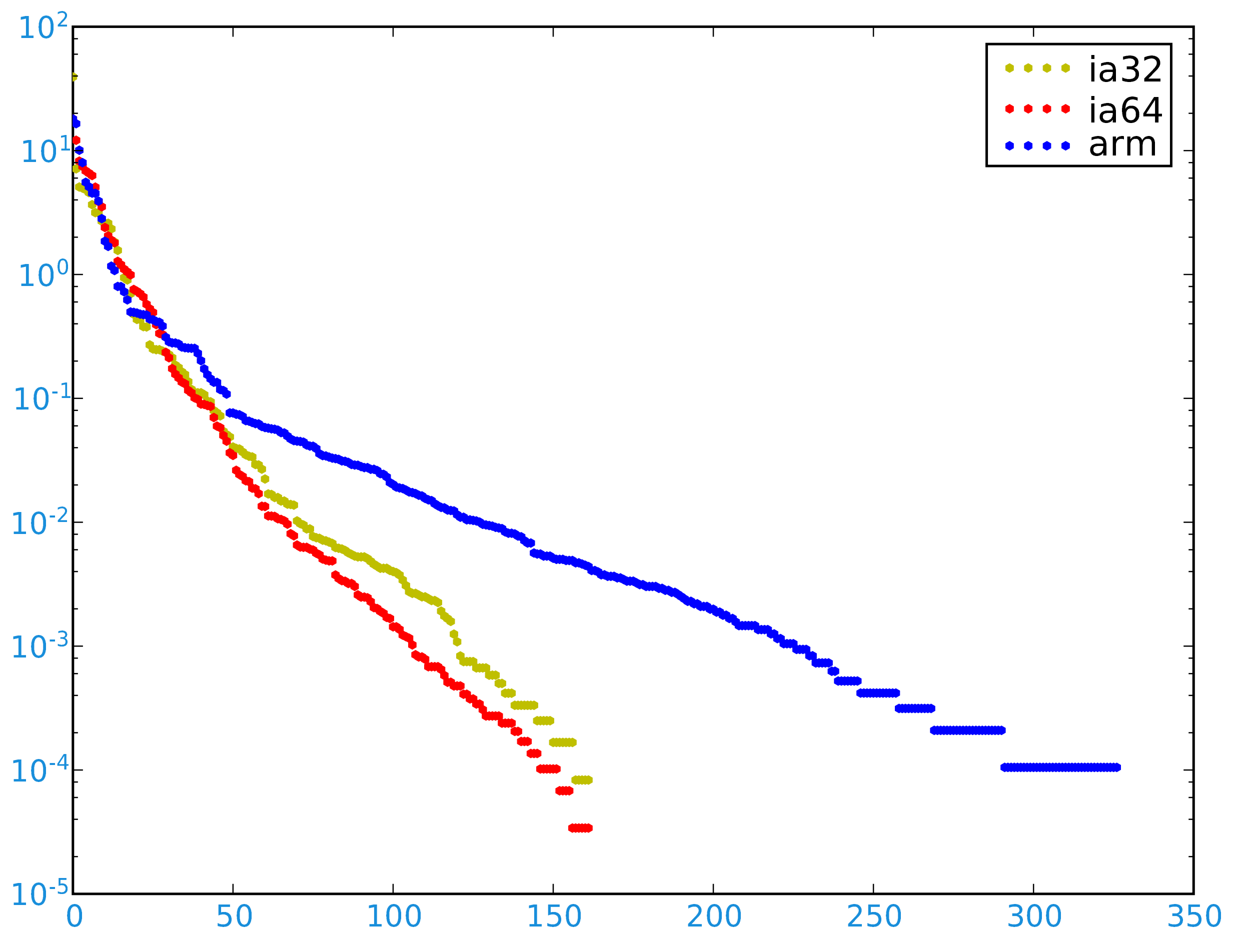


Figure 4: Instruction frequency for IA-32, IA-64, and ARM

instruction through the `rep` prefix: when encountered, the instruction is executed repeatedly, each time decrementing the `ecx` register, until `ecx` is zero. Other prefixes and suffixes allow specification of an alternative operand length to various instructions, or specify alternate addressing modes.

3.3 Hardware-supported stack

IA-32 has extensive hardware support for C-style stacks. The current position on the stack is defined as the location of the ESP register. The stack is always relative to the segment located through the SS register. Multiple instructions make use of the stack. The procedure-support commands `call`, `ret`, `enter`, and `leave` all store data onto, or take data from, the stack, as do the direct stack-manipulation commands push, pop, pushf, popf, pusha, and popa, among others.

Hardware support for stacks is not among modern architectures. Neither ARM nor IA-64 include hardware stack support, but provide the same functionality with more general-purpose instructions. For example, the `ldmia` instruction of the ARM ISA stores any subset of the ARM's 16 general-purpose registers to memory addressed by another register. This command is often used in a procedure prologue to save callee-saved registers, and in this way is roughly analagous to ia-32's pusha. However, pusha does not allow the specification of individual registers and is thus used less frequently.

3.4 Instruction decoder

As described above, IA-32 is a CISC instruction set. Informally, this means that, compared with RISC, instruction complexity varies a great deal from one instruction to the next. P6 implementations solve this problem by translating individual instructions to one or more RISC-like *micro-ops*, each one a fixed (large) size of 118 bits. Figure 6 shows the traditional implementation of this decoder.

In the P6 instruction decoder imlementation, the L1 cache contains IA-32 instructions. Before an instruction is issued it must be decoded. If it is sufficiently simple, it is passed to one of three *decoders*. The P6 family contains two *simple* instruction decoders, capable of producing a single micro-op per clock cycle, for those IA-32 instructions which correspond to a single micro-op. IA-32 instructions corresponding to between 1 and 4 micro-ops are passed to the *complex instruction decoder*. Instructions that require more than 4 micro-ops are instead passed to the *micro-op sequencer*, a ROM which essentially maps CISC instructions to micro-op sequences. If the decoders are operating at full efficiency, they can together produce 6 micro-ops per clock cycle, hence the length of the *micro-op* queue into which they feed.

The advantages of this arrangement are clear: complex IA-32 instructions are translated into easy-to-pipeline time-homongeneous operations. Intel has never released any information on micro-ops, and does not allow external access to them, thus preserving an ability to change micro-ops whenever required.

3.5 Addressing modes

IA-32 supports seven addressing modes, summarised in Figure 7. This number is not excessive — indeed, it is significantly smaller than ARM, which provides 30, though apparently larger than IA-64 — but, unlike these two architectures, IA-32 supports memory operands for each mode. In contrast, ARM and Itanium only allow memory operands in load and store operations. IA-32 implementations deal with the long latency implied by memory operands by splitting appropriate operations into several microcode operations: logically, one operation to retrieve data to a temporary register and one to operate on the data.

ARM and IA-64 both support some form of *auto-increment addressing* for loads and stores, where a register used to index a memory location is automatically incremented when the location completes. ARM supports this scheme very generally, offering both *pre-increment* and *post-increment* options, in several variations. IA-32 supports several far less general-purpose forms of this addressing mode. Firstly, two instruction prefixes, `rep` and `loop`, automatically decrement the `ECX` register after executing their associated instruction. The `movs` instruction, automatically increments or decrements the ESI and EDI

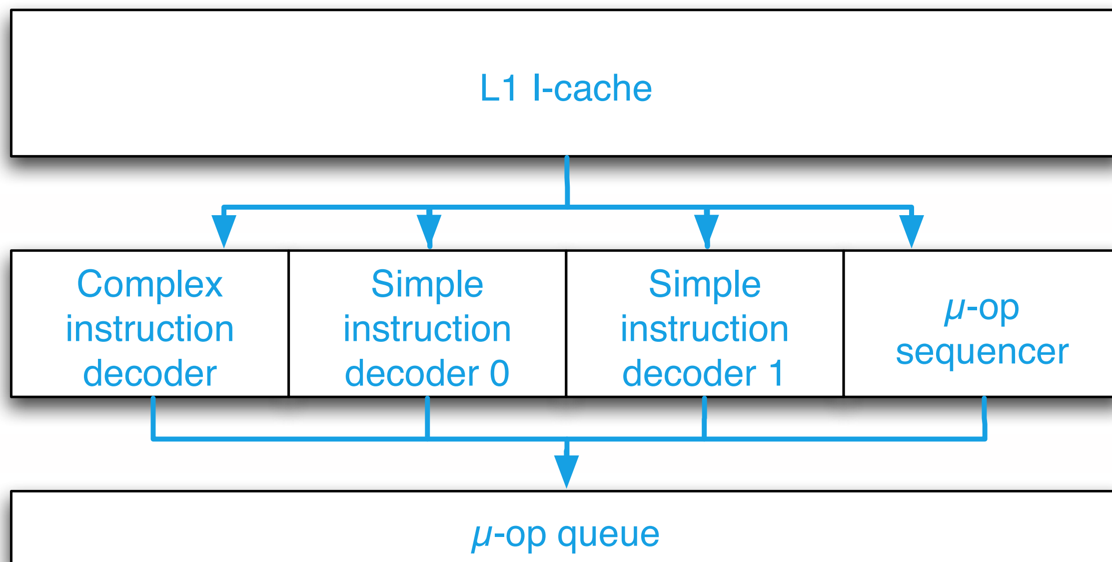


Figure 5: The P6 IA-32 instruction decoder

Mode	Description
Absolute	16- or 32-bit displacement
Register indirect	From EAX, ECX, EDX, EBX, ESI, EDI
Based + displacement	Register base, 16- or 32-bit displacement
Indexed	Sum of two registers
Based index + disp	Sum plus 8- or 16-bit displacement
Base + scaled index	Register + 2^{scale} x register
Base + scaled index + disp	(Register + 2^{scale} x register) + disp

Figure 6: IA-32 addressing modes

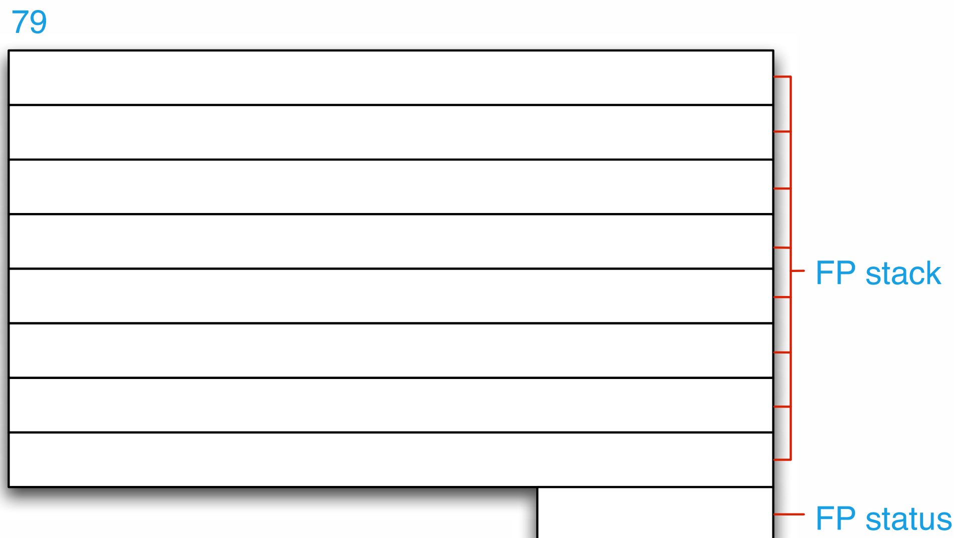


Figure 7: IA-32 floating-point registers

registers after operation. `Movs` is a variant of `mov` but, unlike `mov`, it performs memory-to-memory copying. Therefore a compact data-copying idiom for IA-32 is to initialise `ESI` and `EDI` to point to the source and destination addresses of data, set `ECX` to the amount of data to copy, and perform the copy using a single `rep` movsd (copy doubleword from `ESI` to `EDI` and then decrement `ECX`, until `ECX` is zero) instruction, performing fix-up operations if the data is divisible by the length of a doubleword.

3.6 Floating-point support

Adding to IA-32's register problems is its use of an unusual design for its floating point unit. Unlike ARM and IA-64, both of which have a dedicated set of floating-point registers, IA-32 uses a stack model. Floating-point operations making use of the stack take a single argument; the second argument is defined as the floating-point number on the top of the stack. The rationale behind this design is instruction-set constraints: originally, floating-point support was implemented in a co-processor, the 8087, which only accepted a single argument. Detecting and recovering from stack overflow in software is difficult on IA-32 implementations. Therefore, compilers insert unnecessary loads and stores in floating-point code to ensure the stack never overflows.

The stack architecture is not strict: in addition to the implicit operand on the top of the stack, a second argument may be specified from below the top of the stack. P6-microarchitecture-derived implementations perform register renaming on stack registers, and few compilers make use of the stack design. IA-32s floating-point registers are shown in Figure 8.

An interesting feature of non-SIMD IA-32 floating-point is its *wide* operation. The floating-point stack is 80 bits wide internally, and all operations that result in data being placed on the stack operate at this width. Data are truncated to 64 or 32 bits when transferred from the stack to memory. IA-64 also uses this design: floating-point registers are 82 bits wide, and floating-point operations operate at that width.

3.7 SIMD

IA-32 contains a number of optional single-instruction, multiple-data (SIMD) instruction set extensions. The first set, named MMX, was introduced with the Pentium Pro. MMX instructions rename the 8 floating-point registers, ignoring the upper 16 bits of each to produce 8 64-bit *MMX registers* `MM0` through `MM7`.

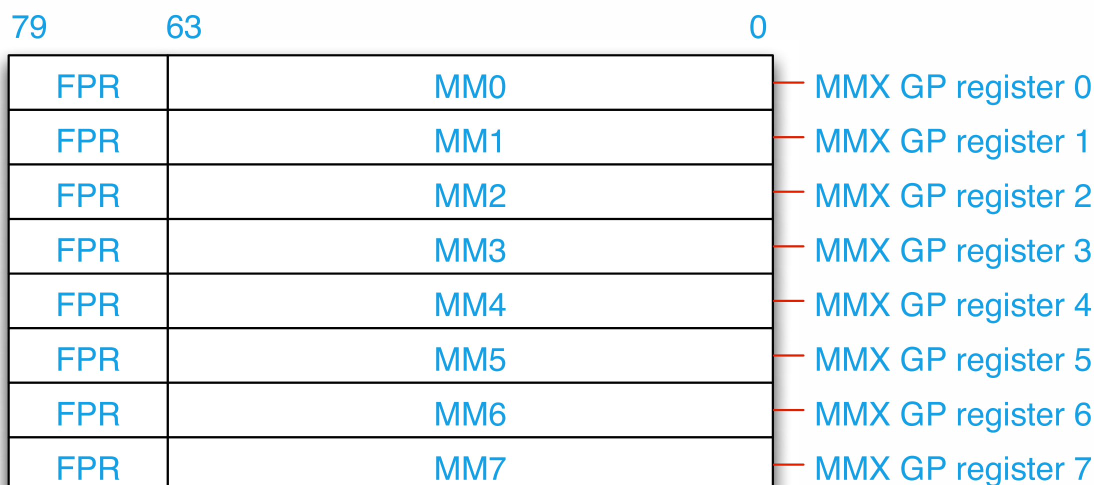


Figure 8: IA-32 MMX registers

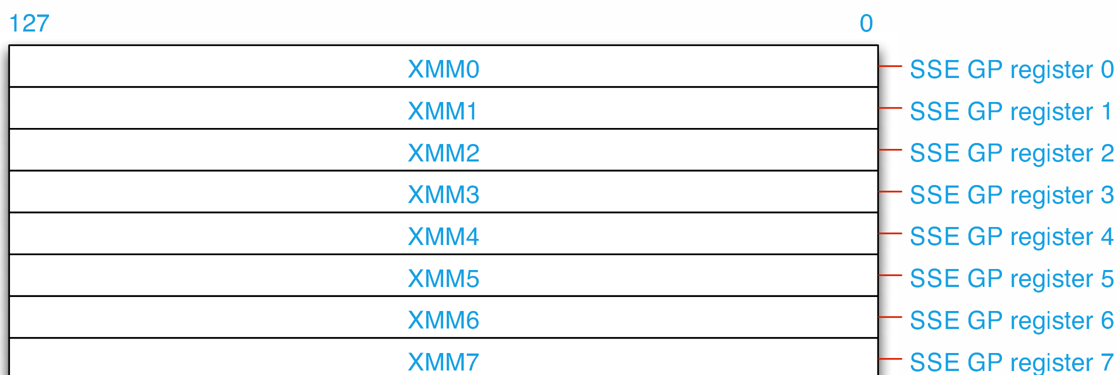


Figure 9: IA-32 SSE registers

MMX is an integer-only extension. Integers may be stored one-per-register or up to 8 may be *packed* into a single register, depending on size. MMX supports all power-of-two sizes for integers from 8 bits to 64 bits (refer to Figure **??**). There are a number of problems with MMX: it is integer-only, meaning that it is not suited to some applications (such as high-end audio or video processing); it re-uses the floating-point registers, making it difficult for programs to make use of both floating point and MMX operations in the same section of code (fortunately, this is not a common requirement).

Intel introduced the SSE extensions to address these concerns. SSE introduces a separate set of 8 128-bit registers (Figure 10) and provides both integer and floating point operations, supporting the same packed style as MMX. Later enhancements to the SSE instruction set in the form of SSE2 and SSE3 added support for more operations and encodings. In particular, DSP-like multiply-and-add instructions were added.

Programming for maximum efficiency of IA-32 SIMD extensions, or indeed any SIMD extensions, requires attention to detail on the part of the programmer. In particular, data must be laid out to minimise the work required to place the data into an MMX or SSE register (known as *packing*). For example, a naive implementation of an array of three-dimensional points may look like this:

```
struct point {
uint32_t x, y, z;
```

```
};

struct point *allpoints;
```

Now imagine performing a translation over allpoints. Using MMX, two points could be translated at once. Using SSE, four points could be translated at once. However, for each integer, the application must collect and pack the data. A better layout would remove the need to *swizzle* the data, by using the *structure of arrays* (SoA) arrangement instead:

```
struct point {
uint32_t *x, *y, *z;
}
```

4 Cache architecture

IA-32 does not define a particular cache architecture. Instead, it provides mechanisms for applications to determine the layout and nature of the cache using the cpuid architecture-identification instruction. This lack of information makes the architecture difficult to optimise for, but is consistent with IA-32's general philosophy of leaving optimisation to the hardware. Fortunately, it is only in very special cases that the particular architecture matters, though cache size is always a concern.

P6-derived implementations have separate small L1 code and data caches, and a unified L2 cache. Server implementations, such as the Xeon line, include a unified L3 cache. The Netburst architecture abandoned the L1 instruction cache in favour of a large *trace cache*, but later IA-32 implementations, such as the Pentium D and Core, revert to the P6-derived small L1. The trend has been towards implementing cache on-die; L1 is always on-die, and L2 is on-die in all new implementations.

IA-32 supports two page sizes when paging is enabled: 4KB and 4MB. Implementations separate the caches for these sizes, and further separate the TLBs according to the type of data stored there (code or data), making for four TLBs in all.

Given its CISC nature, one might expect IA-32 to make better use of their caches than RISC machines. After all, its instructions are rather compact. Hennessy and Patterson measured the average instruction length for several integer programs and, separately, for several floating-point programs[3] and found the average instruction lengths to be 2.8 for integer programs and 4.1 for floating-point programs. Unfortunately, register pressure forcing data to "spill" to the stack increases data traffic to the point that all benefits from increased code density are negated.

5 Instruction-level parallelism

Hennessy and Patterson describe the *instructions per second* (IPS) metric for measuring processor throughput. Until recently, IA-32 implementations have steadily increased in IPS. Recently, however, the per-processor level of IPS has fallen, as Intel concentrated on other concerns, notably power consumption and thread-level parallelism.

5.1 Pipelining

All IA-32 implementations since the i486 have been pipelined, and earlier implementations arguably contained a rudimentary pipeline. The Pentium Pro introduced the canonical 12-stage pipeline. The number of stages has changed slightly with new generations of the P6 architecture, but never by more than one or two stages. This is in dramatic contrast with Intel's Netburst architecture, which at one stage had a pipeline over thirty stages long.

[3]The programs measured were taken from the SPECint92 and SPECfp92 benchmark suites

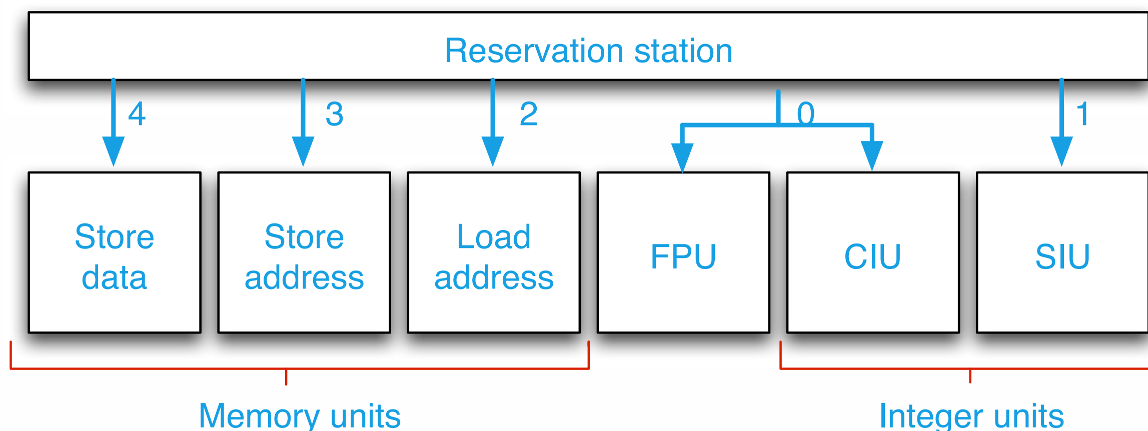


Figure 10: P6 family execution units

The P6 family supports superscalar execution using six *execution units*, up to five of which can execute simultaneously, reading micro-ops from five *ports* attached to the reorder buffer's reservation station (Figure 11).

5.2 Register renaming and out-of-order execution

Depiste the relatively-small number of programmer-visible *architectural registers*, IA-32 implementations contain a large number of on-chip registers. These internal registers are used as substitutes for architectural registers, and have many optimisation-related uses, such as reducing the potential for *name dependencies*, where an architectural register is re-used not for any data-dependence reason but simply because there were no other registers available. The technique of mapping architectural registers to implementation registers, known as *register renaming*, is essentially an optimisation detail of an IA-32 implementation. As such, each IA-32 implementation performs register renaming differently.

In the P6 family, register renaming is performed by the *reorder buffer*, which is also used to reorder instructions to maximise instruction-level parallelism in the face of dependencies. This technique means that the amount of register renaming that can be performed is limited by the amount of instruction re-ordering that may be performed. The Netburst architecture removed this limitation, adding a separate register rename stage.

6 Thread-level parallelism

Recently implementations have focused on increasing thread-level parallelism (TLP), potentially at the expense of ILP.

Early support for TLP came in the form of symmetric multiprocessor (SMP)-capable machines, such as the Pentium. The basic SMP model is illustrated in Figure 12: two separate instruction streams are executed by two entirely-spearate processors. Bus arbitration is accomplished using the MESI protocol. This protocol requires CPUs to *snoop* the bus to ensure that the same memory location is not represented differently in each CPU's cache (through, for example, multiple writes to the location).

More recently, Intel implementations have supported Hyperthreading, an Intel trademarked term for on-chip thread-level parallelism. On a CPU supporting Hyperthreading, various on-chip structures (such as the register rename buffer, the architectural registers, and the TLBs) are partitioned or replicated. Since it is very rare for all execution units of the CPU to be busy at once, the "spare" execution units can be used to execute another thread of control (Figure 13). Intel claims a maximum of 30 percent performance improvement at a cost of a 10 percent increase in die size, although the real-world performance increase is rumoured to be more modest.

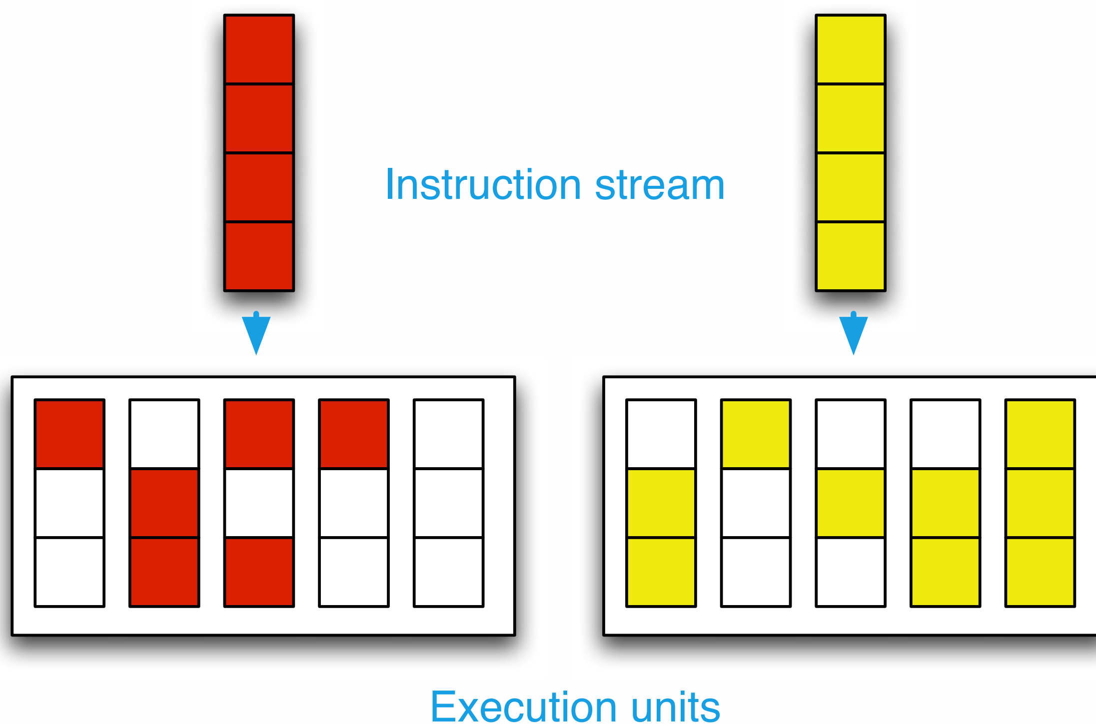


Figure 11: SMP

7 Future directions

IA-32 is unquestionably with us for years to come. Intel's focus for future implementations seems to be increasing TLP and providing innovations in the form of instruction set enhancements. For example, a new iteration of SSE, SSE4, should be available mid-2006. Intel's implementation of AMD's 64-bit extensions to IA-32, EMT64, provides a flat address space beyond 32 bits. IA-32 is not completely virtualisable; the Vanderpool instruction set (recently renamed to "VT") allows complete virtualisation of the hardware without JIT recompilation of problematic opcodes; this technology started with server chips but is seeing increasing demand on desktops. Finally, the inexorable rise in the importance of laptop computing has seen Intel abandon a high-power design, Netburst, in favour of the lower-power P6 family (though power consumption of Core is at least an order of magnitude higher than a comparable ARM processor).

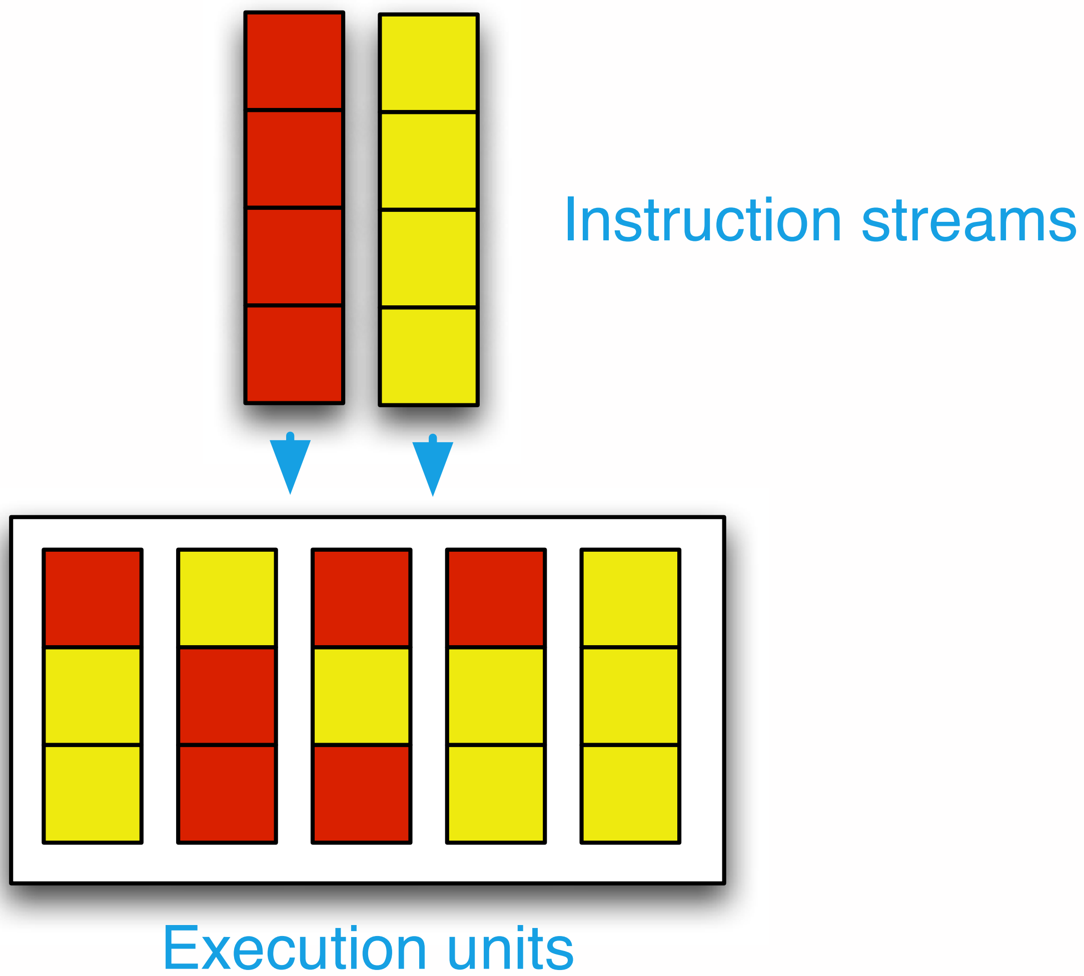


Figure 12: Hyperthreading