目录

第 2 章　Intel 中央处理器

中央处理器（Central Processing Unit, CPU）是一块超大规模的集成电路，是一台计算机的运算核心和控制核心。在微型计算机中亦称为微处理器。它的功能主要是解释计算机指令以及处理计算机软件中的数据。本章主要介绍 Intel 公司的微处理器的发展历史、组成结构以及其各组成部分完成的功能。重点要掌握执行部件中的通用寄存器和标志寄存器、指令预取部件中的指令指示器、分段分页部件中的段寄存器的作用，掌握计算机工作的基本过程。

2．1 Intel 微处理器的发展史

微处理器是计算机中的核心部件。在过去的几十年里，其发展迅速，带动了计算机体系结构的优化，存储器容量的不断增大、存取速度的不断提高，外围设备的不断改进以及新设备的不断出现等。微处理器的生产厂商主要有 Intel、AMD、IBM、Cyrix、IDT、VIA（威盛）、国产龙芯等。

本节介绍 Intel 公司的微处理器发展历史。Intel 公司成立于 1968 年，其名称是 INTegrated ELectronics（集成电子）两个单词的缩写。该公司先后推出的中央处理器包括：Intel4004、Intel8008、Intel8080/8085、8086/8088、80186、80286、i386、i486、Pentium（奔腾）、Pentium II、Pentium III、Pentium IV、Xeon（至强）、Xeon3、Xeon5、Xeon7、Itanium（安腾），Itanium 2、Core（酷睿）系列。

虽然 Intel 公司推出的微处理器很多，但是微处理器的结构主要分成两种：IA-32 和 IA-64。在 1989 年推出 80486 处理器之后，英特尔以较正式的 IA（Intel Architecture）指称该架构，也称为 X86-32 架构。由于从 8086 开始其后产品以 80186、80188、80286、i386、i486 等为代号命名，因而被外界称为 X86 架构。其后的奔腾系列、Xeon 系列、酷睿系列全部是基于 X86 架构的产品。它属于复杂指令集（Complex Instruction Set Computing，CISC）架构。

IA-64 架构是英特尔为了提高 IA-32 位处理器的运算性能而开发的一种全新处理器架构，IA-64 架构是 EPIC（Explicitly Parallel Instruction Computing）的 64 位架构是基于超长指令字 VLIW（Very Long Instruction Word）的设计，将多条指令放入一个指令字，与 x86 不能兼容。2001 年，Intel 公司推出了 IA-64 架构系列中的第一款通用 64 位微处理器 Itanium。2009 年，Intel 推出的 Itanium 2 系列处理器也是采用这一架构的。由于它不能很好地解决与以前 32 位应用程序的兼容，这一重大缺陷导致应用受到较大的限制，在市场中并不成功。有一种说法是，AMD 公司抢先在 32 位 x86 指令集的基础上扩展到 64 位，推出了 AMD64。因为有 AMD 的竞争，Intel 做出了一种新的但是不太成功的尝试。

为了解决与 IA-32 兼容的问题，Intel 回到了与 x86 兼容的道路上，采用了 x86-64 结构（也称为 Intel64，x64），它实际上是在 x86 平台上从 32 位到 64 位的一次扩充。而这一扩充包含的内容不仅仅体现在对物理内存的扩充上，在指令集、CPU 寄存器结构、甚至是应用程序的虚拟内存上都得到了非常大的扩充。然而，值得注意的是，x86 从 32 位到 64 位的变化，并没有像以前从 16 位到 32 位的变化那样，在系统软件层面带来了革命性的变化（例如页式地址管理、多任务的引入等等）。操作系统仍然使用以前的各种机制来对硬件进行管理，只是在 64 位平台上，数据（如整型数）变得更宽了，逻辑地址、线性地址以及物理地址也都变得更宽了。

根据微处理器的字长和功能，可将其发展划分 4 位、8 位、16 位、32 位、64 位、多核 64 位等几个阶段。

第 1 阶段（1971-1973 年）是 4 位和 8 位低档微处理器时代。典型产品有 Intel4004 和 Intel8008 微处理器。它们采用 PMOS 工艺，集成度低，系统结构和指令系统都比较简单，主要采用机器语言或简单的汇编语言，指令数目也只有 20 多条。

第 2 阶段（1974-1977 年）是 8 位中高档微处理器时代，代表产品有 Intel8080 /8085。它们采用 NMOS 工艺，集成度提高约 4 倍，运算速度提高约 10~15 倍。指令系统比较完善，具有典型的计算机体系结构和中断、DMA 等控制功能。

第 3 阶段（1978-1984 年）是 16 位微处理器时代。典型产品是 Intel 公司的 8086 /8088。其特点是采用 HMOS 工艺，集成度（20000~70000 晶体管/片）和运算速度（基本指令执行时间是 0.5μs）都比第 2 代提高了一个数量级。指令系统更加丰富、完善，采用多级中断、多种寻址方式、段式存储机构、硬件乘除部件，并配置了软件系统。80286 是英特尔首款能执行所有旧款处理器专属软件的处理器，这种软件相容性也极大地推动了微型计算机风靡世界。

第 4 阶段（1985-1992 年）是 32 位微处理器时代，又称为第 4 代。1985 年 Intel 公司推出 32 位微处理器 80386，采用了 32 位寄存器，具有 32 根地址总线和数据总线，全面支持 32 位的数据、指令和寻址方式，可访问 4GB 的存储空间。采用 HMOS 或 CMOS 工艺，集成度高达 100 万个晶体管/片，每秒钟可完成 600 万条指令（Million Instructions Per Second，MIPS）。微型计算机的功能已经达到甚至超过超级小型计算机，完全可以胜任多任务、多用户的作业。1989 年，英特尔推出了 80486 芯片。80486 是将 80386 和数学协微处理器 80387 以及一个 8KB 的高速缓存集成在一个芯片内，内部缓存缩短了微处理器与慢速 DRAM 的等待时间。

第 5 阶段（1993-2005 年）是奔腾（Pentium）系列微处理器时代，也称为第 5 代。在 80486 之后，本应命名为 80586 或 i586，但 i586 被英特尔竞争对手所制造的类 80586 微处理器所使用。通常认为 pentium 是希腊文 penta（五）接尾语“ium”的构成。它的内部采用了超标量指令流水线结构，并且有相互独立的指令和数据高速缓存。

1997 年推出的 Pentium II 处理器结合了 Intel MMX（Multi Media eXtended）技术，能以极高的效率处理影片、音效、以及绘图资料。MMX 技术使用了单指令多数据流（Single Instruction Multiple Data，SIMD）执行模式，在 64 位寄存器中的压缩整数数据上进行并行计算。

1999 年推出的 Pentium III 处理器加入 70 条新指令，加入了 SIMD 的延伸集，称为单指令多数据流扩展（Streaming SIMD Extensions，SSE），能大幅提升多媒体、流媒体应用软件执行的性能。同年，英特尔还发布了 Pentium III Xeon 处理器，加强了电子商务应用与高阶商务计算的能力。在缓存速度与系统总线结构上，也有很多进步，很大程度提升了性能，并为更好的多处理器协同工作进行了设计。2000 年英特尔发布了 Pentium IV 处理器。Pentium IV 提供了的 SSE2（Streaming SIMD Extensions 2）指令集，这套指令集增加 144 个全新的指令，在 128bit 压缩的数据，在 SSE 中仅能以 4 个单精度浮点值的形式来处理，而在 SSE2 指令集能采用多种数据结构来处理。

2003 年英特尔发布了 Pentium M(mobile)处理器。以往虽然有移动版本的 Pentium II、III，甚至是 Pentium 4-M 产品，但是这些产品是基于台式电脑处理器的设计，虽然增加一些节能和管理的新特性，但是 Pentium III-M 和 Pentium 4-M 的能耗远高于专门为移动运算设计的 CPU，例如全美达（Transmeta）的处理器。

在第 5 阶段，出现的 Intel CPU 还有 Pentium Pro、Pentium II Xeon、Celeron、Pentium III Xeon 、Pentium M、Pentium D、Pentium EE 等等。其中，Pentium Extreme Edition 处理器引入了双核（dual-core）技术。这项技术提供了先进的硬件多线程支持。该处理器基于英特尔 NetBurst 微体系结构，支持 SSE、SS2、SS3、超线程技术。

第 6 阶段（2005 年至今）是酷睿（Core）系列微处理器，开启了双核和多核的时代。2006-2007 年，Intel 推出了 Core Duo 和 Core Solo 处理器，采用智能高速缓存，允许两个处理器内核之间高效的数据共享。同期产品还有 Intel Xeon Processor 5100, 5200、5300、5400、7400 和 Intel Core 2、Core 2 Duo 等等。酷睿系列还包括 Core i7 系列、Core i5 系列、Core i3 系列。在 2017 年，Intel 还发布了 Core i9 系列，最多可以支持 18 个内核。

2．2 Intel x86 微处理器结构

中央处理器（CPU）在微型计算机中又称微处理器，计算机的所有操作都受 CPU 控制，CPU 的性能指标直接决定了微机系统的性能指标。随着 x86 系列机的发展, x86 微处理器的结构变化是非常大的，但它的功能总是保持着向下兼容，详细介绍它各部分的结构和功能不是本门课程的内容。因此，本节从学习汇编语言的角度出发进行介绍。32 位 CPU 按其主要功能通常可分为六大部件：总线

接口部件、执行部件、指令预取部件、指令译码部件、分段部件和分页部件。其内部结构如图 2.1 所示。

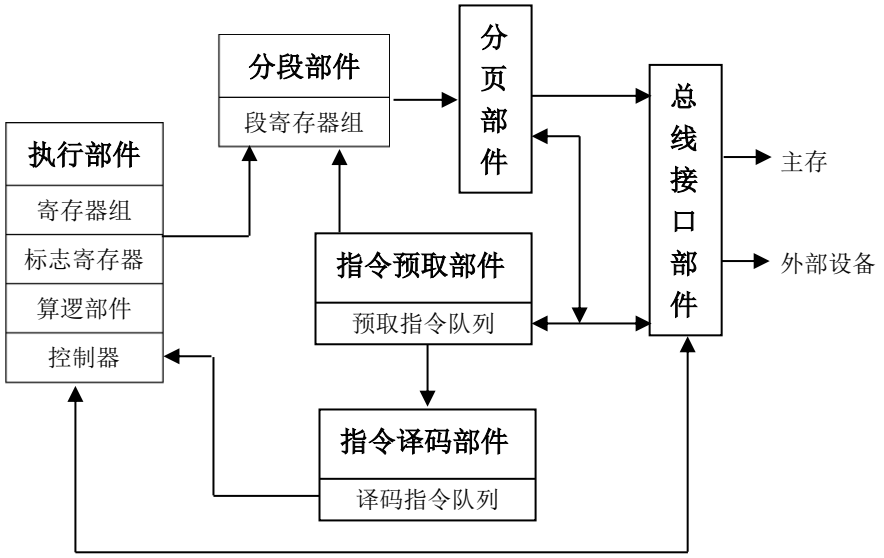

图 2.1　X86 微处理器的基本结构

CPU 对指令的处理基本分为四个阶段：提取（Fetch）、解码（Decode）、执行（Execute）和写回（Writeback）。

CPU 从存储器或高速缓冲存储器中取出指令，放入指令寄存器，并对指令译码。它把指令分解成一系列的微操作，然后发出各种控制命令，执行微操作系列，从而完成一条指令的执行。指令是计算机规定执行操作的类型和操作数的基本命令。

提取：根据指令预取部件中的指令指针（IP、EIP 或 RIP）和分段分页部件中的代码段寄存器（CS），从存储器或高速缓冲存储器中取出要执行的指令。IP、EIP、RIP 分别在 16 位、32 位和 64 位微处理器中使用，指向当前待执行的指令在代码段中的位置。

解码：CPU 根据存储器提取到的指令来决定其执行行为。在解码阶段，指令被拆解为有意义的片段。根据 CPU 的指令集架构（Instruction Set Architecture，ISA）定义将数值解译为指令。一部分的指令数值为运算码（Opcode），其指示要进行哪些运算。其它的数值通常供给指令必要的信息，诸如一个加法（Addition）运算的运算目标。

执行阶段：在提取和解码阶段之后，紧接着进入执行阶段。该阶段中，连接到各种能够进行所需运算的 CPU 部件。例如，要求一个加法运算，算术逻辑单元（Arithmetic Logic Unit，ALU）将会连接到一组输入和一组输出。输入提供了要相加的数值，而输出将含有总和的结果。ALU 内含电路系统，完成简单的普通运算和逻辑运算（如加法和位操作运算）。在执行部件中含有通用寄存

器和标志寄存器。前者用于存放运算的数据和结果，后者是指令执行状态的信息及运算结果的特征。

结果写回：以一定格式将执行阶段的结果简单的写回。运算结果经常被写进 CPU 内部的暂存器，以供随后指令快速存取。在其它案例中，运算结果可能写进速度较慢，但容量较大且较便宜的主记忆体中。某些类型的指令会改变指令指针，而不直接产生结果，即转移指令，使得程序能够循环执行、条件执行和函数调用。在执行指令并写回结果之后，指令指针值会递增，下一个指令周期正常的提取下一个顺序指令。

总线接口部件是 CPU 与整个计算机系统之间的高速接口，能接受所有的总线操作请求，并按优先权进行选择，最大限度地利用本身的资源为这些请求服务。如：从主存中取指令送入指令预取部件的排队机构排队、取操作数送入执行部件参加运算、将操作结果送到输出设备输出等。

CPU 中的控制部件主要为完成每条指令所要执行的各个操作发出控制信号。控制部件的结构有两种：一种是以微存储为核心的微程序控制方式；一种是以逻辑硬布线结构为主的控制方式。微存储中保存微码，也称为微指令，每条指令是由一序列的微码组成，每一个微码对应于一个最基本的微操作。这种微码序列构成微程序。中央处理器在对指令译码以后，发出一定时序的控制信号，按给定序列的顺序以微周期为节拍执行由这些微码确定的若干个微操作，即可完成某条指令的执行。简单指令是由几个微操作组成，复杂指令则要由几十个微操作甚至几百个微操作组成。

2．3 执行部件

CPU 的执行部件主要由寄存器组、标志寄存器、算逻部件、控制部件等组成。算逻部件（Arithmetic Logic Unit，ALU）可以执行定点或浮点算术运算操作、移位操作以及逻辑操作，也可执行地址运算和转换。寄存器部件，包括通用寄存器、专用寄存器和控制寄存器。通用寄存器又可分定点数和浮点数两类，它们用来保存指令执行过程中临时存放的寄存器操作数和中间（或最终）的操作结果。

大多数指令都要访问到通用寄存器。通用寄存器的宽度决定计算机内部的数据通路宽度，其端口数目往往可影响内部操作的并行性。专用寄存器是为了执行一些特殊操作所需用的寄存器。控制寄存器（CR0～CR3）用于控制和确定处理器的操作模式以及当前执行任务的特性。CR0 中含有控制处理器操作模式和状态的系统控制标志；CR1 保留不用；CR2 含有导致页错误的线性地址；CR3 中含有页目录表物理内存基地址。

从学习汇编语言程序设计的角度看，了解通用寄存器组的分工及标志寄存器的作用是非常重要的。

所谓寄存器，就是 CPU 中的专用存储器。每个寄存器中可存储一个 0/1 串。寄存器的位数决定了串的长度。例如一个 32 位的寄存器，就存储由 32 个 0/1 组成的串，该串可以是一个参与运算的操作数，也可以一个运算结果，或者是某个单元的地址等等，这取决于在指令中是如何使用该寄存器的。虽然在机器指令中每个寄存器都有一个二进制的编号（相当于数值地址），但不便于人们记忆，因此给每个寄存器一个名字，这些名字可以看成 CPU 中的存储单元的符号地址。在写 C 语言程序时，程序员并未直接使用寄存器，但是，在编译后产生的语句中会大量地反复地使用寄存器，用于临时存储各种需要处理的数据。

2.3.1 32 位 CPU 中的通用寄存器

在 32 位 CPU 中，寄存器组中含有 8 个 32 位的通用寄存器，分别是 EAX、EBX、ECX、EDX、ESI、EDI、ESP、EBP。EAX 称为累加器（Accumulator）；EBX 称为基地址寄存器（Base）；ECX 称为计数寄存器（Counter）；EDX 称为数据寄存器（Data）；ESI 称为源变址寄存器（Source Index）；EDI 称为目的变址寄存器（Destination Index）；ESP 是堆栈栈顶指针（Stack Pointer）、EBP 是基址指针(Base Pointer)。

所谓通用(General Purpose),即可以在各处使用、公共使用、彼此可以换用等。就像大瓷碗、玻璃杯、夜光杯，都可以装液体，包括白酒、啤酒、葡萄酒等液体，因而是通用的。但是这些容器还有一些习惯性的用法，大碗喝白酒可以喝出一种豪迈之气,夜光杯喝葡萄美酒出的婉约风情。在编写程序的时候，通常也像生活中使用大瓷碗、夜光杯一样，有所讲究，体现出编程是一种艺术的特质。例如要求一个数组中元素的和，就可以用累加器 EAX 来存放这些数的和；用计数器 ECX 来存放有多少个数要相加；用基地址寄存器 EBX 来存放操作数的地址。在将某个位置的一个字符串拷贝到另一个地方时，就可以用源变址寄存器 ESI 来存储待拷贝字符的地址，而用目的变址寄存器 EDI 来存储拷贝到的新位置的地址等等。

在 8 个 32 位寄存器中，ESP 是比较特殊的一个，一般不作为数据寄存器使用，而专门用于堆栈的栈顶指示器。在执行数据压栈指令 PUSH、数据出栈指令 POP 时，CPU 会自动改变 ESP 中的值，使其始终是指向栈顶，即存放栈顶元素所在单元的地址。虽然可以写其他指令直接改变 ESP 中的值，但当 ESP 中的值发生变化后，执行 PUSH、POP 指令时，就要根据 ESP 中新的值作为栈顶元素的指针，因而一般不直接写指令来改变 ESP 的值。

上述 8 个 32 位寄存器的低 16 位可以作为 16 位寄存器使用。它们的名字分别为：AX、BX、CX、DX、SI、DI、SP、BP。在 16 位微处理器时代，即 8086、80186、80286 中的寄存器就是 AX、BX、CX、DX、SI、DI、SP、BP。当发展到 80386 时，对这些寄存器进行了扩展（Extented），形成了 32 寄存器 EAX、EBX、ECX、

EDX、ESI、EDI、ESP、EBP。

对于 16 位寄存器中的 AX、BX、CX、DX 按高 8 位和低 8 位分为两个小组：H 组(AH，BH，CH，DH)和 L 组(AL，BL，CL，DL)，作 8 位的寄存器使用，如图 2.2 所示。

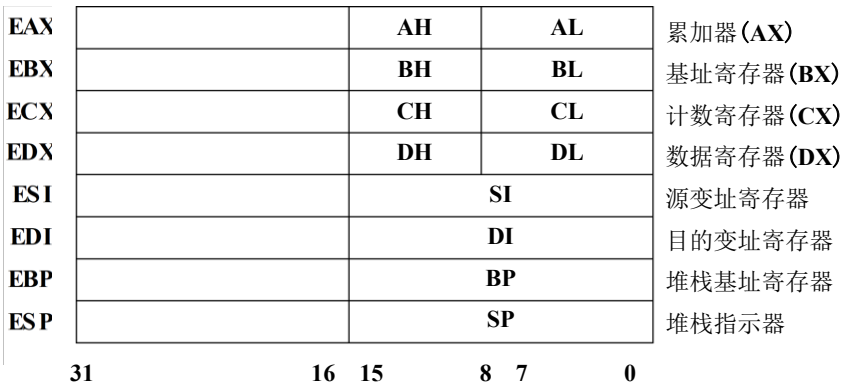

图 2.2 Intel x86-32 中数据寄存器的基本结构

假设字节寄存器（AH）=10110100B，共有 8 个二进制位组成，其最高二进制位为第 7 位，其值为 1,其最低二进制位为第 0 位，其值为 0。

第 7 位　　第 0 位

1	0	1	1	0	1	0	0

图 2.3 （AH）中数据存放示意图

寄存器的名字是 CPU 中某个存储单元的名字，可以看成一个单元的符号地址。在机器指令的编码中，这些寄存器也要对应一个编号（相当于数值地址），如表 2.1 所示。

表 2.1 Intel 32 位 CPU 中的通用寄存器

32 位寄存器	16 位寄存器	8 位寄存器	二进制编码
EAX	AX	AL	000
ECX	CX	CL	001
EDX	DX	DL	010
EBX	BX	BL	011
ESP	SP	AH	100
EBP	BP	CH	101
ESI	SI	DH	110
EDI	DI	BH	111

从表 2.1 中，可以看到 EAX、AX、AL 的编码都是 000。显然在机器指令编码中，若知道使用了一个寄存器，且寄存器的编码是 000,如何能正确对应到是用的 EAX，AX，还是 AL 呢？为什么不给所有的寄存器（上面列出来的 24 个寄存器）用 5 位二进制来编码呢？对前一个问题，将在第 4 章寻址方式中给出解答，而后一个问题就留个读者去猜想指令系统的设计者是怎么考虑的。

2.3.2 通用寄存器应用示例

为了让读者尽快的熟悉 32 位、16 位、8 位寄存器之间的关系、更快的熟悉一些基本运算指令，给出了如下的寄存器应用例子。

例 2.1 写出完成各个功能的指令

① 给 EAX 赋值为 12345678H ，执行后 (EAX)=12345678H

```
MOV  EAX, 12345678H
```

② 将 EAX 的低 16 位赋值为 3344H ，执行后 (AX)=3344H，（EAX）的高 16 位（即 16-31 位）保持不变

```
MOV  AX, 3344H
```

③ 将 EAX 的 0-7 位赋值为 56H，执行后(AL)=56H，其他位的信息保持不变

```
MOV  AL, 56H
```

④ 将 EAX 的 8-15 位赋值为 78H，执行后(AH)=78H，其他二进制位不变

```
MOV  AH, 78H
```

⑤ 将 EAX 的高 16 位（即 16-31 位）赋值为 5566H，保持低 16 位不变

由于 EAX 的高 16 位并没有一个名字，因此，不能像前面的例子那样简单的写出语句，要进行一些变通。

```
AND  EAX, 0000FFFFH
ADD  EAX，55660000H;
```

第一条指令是按位与运算，结果在 EAX 中。EAX 的低 16 位中的内容不变，高 16 位中的内容变为 0；第二条指令是加法运算，结果在 EAX 中。对这个例子，第二条指令也可以写成 OR EAX，55660000H ，这是按位或运算。

例 2.2 写出将 AX 中的内容置为 0 的指令，执行后（AX）=0

```
① 数据传送指令    MOV  AX, 0    ;  0 -> AX
② 算术运算指令    SUB  AX，AX   ; 减法 (AX)-(AX)->AX
                  IMUL AX，AX，0; 乘法 (AX)* 0 ->AX
③ 逻辑运算指令    AND  AX，0    ; 逻辑乘 (AX)∧0->AX
                  XOR  AX, AX ; 异或，(AX)⊕(AX) ->AX
④ 移位指令        SHL  AX，16 ; 逻辑左移 16 个二进制位
                  SHR  AX，16 ; 逻辑右移 16 个二进制位
```

当然，还可以写出更多的完成这一功能的指令。由此可见，编写程序时，可以灵活使用各种指令。

对于含有多媒体扩展功能的 CPU,所含的寄存器在第 15 至第 17 章中介绍。对于 64 位 CPU 所含有的通用寄存器，在第 18 章中介绍。

2．4 标志寄存器

标志寄存器用来保存在一条指令执行之后，CPU 所处状态的信息及运算结果的特征。X86 微处理器在 16 位 CPU 中的标志寄存器是 16 位，称 FLAGS；32 位 CPU 中的标志寄存器是 32 位，称 EFLAGS。32 位的 EFLAGS 包含了 16 位 FLAGS 的全部标志位且向下兼容，所以，本节将以 32 位的 EFLAGS 为例进行说明。

标志寄存器中各标志位的分布如图 2.4 所示。

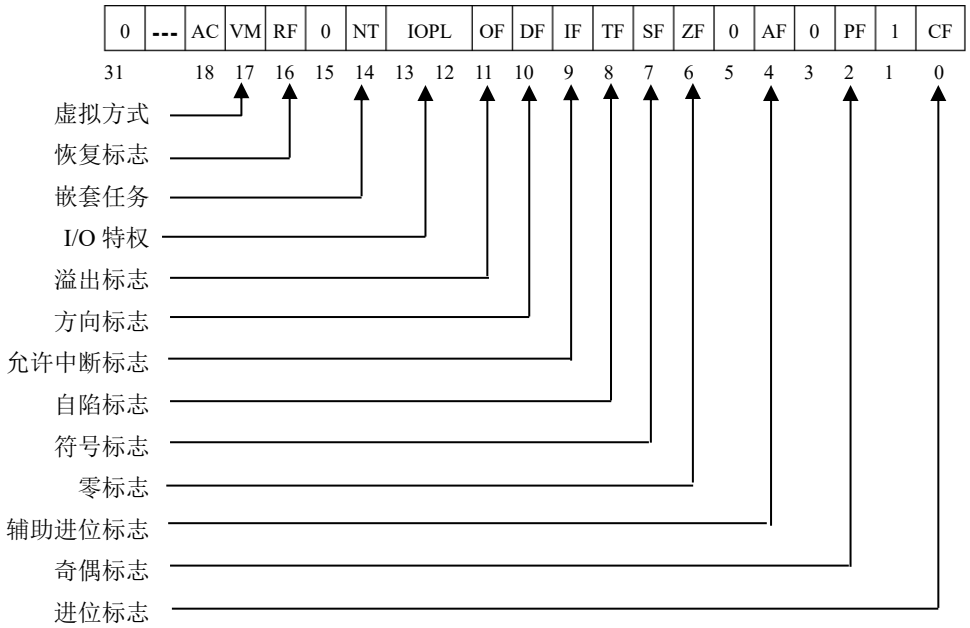

图 2.4 X86-32 的标志寄存器

由图 2.4 中可看出，第 1、3、5、15 位都未定义，18－31 正在逐渐被高档的 X86 机使用（还未定义完）。除第一位总为 1 外，其它未定义位的值总为 0。常用的标志位按作用可分为三类：条件标志位、控制标志位和 32 位标志寄存器扩充的系统标志位。由于 16 位 CPU 中的标志寄存器不含系统标志位，因此，在实方式下系统标志位不发挥作用。下面说明图中各标志位的功能。

2.4.1 条件标志位

条件标志位是由 CPU 根据执行完一条指令后所得运算结果的特征自动设置的，主要用作条件转移指令是否转移的控制条件。条件标志位有：符号标志 SF（Sign Flag）、零标志 ZF（Zero Flag）、溢出标志 OF（Overflow Flag）、进位标志 CF（Carry Flag）和辅助进位标志 AF(Auxiliary carry Flag)、奇偶

标志 PF（Parity Flag）。

1．符号标志 SF

在执行一次运算后，若结果的最高二进制位（the most-significant bit）为 1，则 SF=1，否则 SF=0。

2．零标志 ZF

若运算结果为 0，则 ZF＝1，否则 ZF＝0。

3．溢出标志 OF

当执行一条加法指令时，例如 ADD AH，AL，如果执行前（AH）的最高二进制位（最高位是第 7 位，最低位是第 0 位）与（AL）的最高位相同（如都是 1），而运算结果的最高位与它们相反（如是 0），则 OF=1，当然如果结果的最高位与原来两个加数的最高位相同，OF=0；如果执行前（AH）的最高位与（AL）的最高位不相同（一个是 0，另一个是 1），不论结果的最高位是什么，OF=0。

在理解这一标志位的设置方法时，可以将参与运算的数和结果都理解为有符号数。执行一条加法指令 ADD 时，两个加数的最高位不同，则可以将两个数视为一正一负，不论结果是正是负，都不会溢出，OF=0；只有两个加数都是正数或者都是负数，相加之后，结果变成负数和正数，才会溢出，OF=1。值得注意的是，ADD 指令没有区分加数是有符号数还是无符号数，数的最高位与后面的二进制位一样参与运算。

对于减法运算，如果被减数的最高二进制位与减数的最高二进制位相反，且所得到的差的最高二进制位与被减数的最高二进制位相反，则 OF=1。其他情况下，OF=0。直观的解释是一个正数减一个正数，或者一个负数减一个负数，不论结果是正是负，都不会溢出，OF=0。只有一个正数减去一个负数且结果为负数时，溢出，或者一个负数减去一个正数且结果为正数时溢出,OF=1。

4．进位标志 CF

计算机在作加法等运算时，整个数的所有二进制位都参与运算，如果从最高位再向前面产生进位(加法运算)或借位(减法运算)，则 CF 置 1，否则置 0。

除了算术运算指令、位操作指令等会改变 CF 外，还有专门的指令来改变 CF。

例如，指令 STC（SeT Carry）执行后，CF＝1。指令 CLC（Clear Carry）执行后，CF＝0。指令 CMC（Complement Carry）执行后,进位标志位取反。

5．辅助进位标志 AF

辅助进位标志的置位原则与进位标志相同，只不过这里的进位和借位是指在作字节运算时低半字节向高半字节的进位和借位，即在作字节运算时，如果第 3 位向第 4 位进位(加法)或借位(减法)，则 AF＝1，否则 AF＝0。这个标志主要用于对压缩的和非压缩的 BCD 码的加减运算中。

6．奇偶标志 PF

当运算结果最低有效字节（least significant byte）中 1 的个数为偶数或 0 时，PF 置 1，否则置 0。该标志位主要用于检测数据在传输过程中可能产

生的错误。

例 2.3 设（AL）=01000000B，（AH）=01111111B ，执行 ADD AH，AL 后，（AH）中的值是多少？标志位 SF、OF、ZF、CF 各是多少？

【解】（AH）=10111111B

标志位的设置结果：

SF=1， 因为（AH）的最高二进制位为 1；

OF=1， 两个加数的最高位都为 0，而运算结果的最高位为 1，溢出

ZF=0， 运算结果不为 0

CF=0， 没有进位

注意，CPU 按一定的规则来设置标志位的值，但这些标志位的值如何使用，取决于程序员在之后所使用的指令。程序员可以不理睬这些标志位，也可以根据需要，判断某一个或多个标志位的值，决定下一步的工作。对于本例而言，将二进制数翻译成 10 进制数， 01000000B = 64， 01111111B=127；将它们视为无符号数，相加结果为 191，即 10111111B，此时，程序员应判断 CF，若 CF=0，则无符号运算结果在正常范围内。若将它们视为有符号数，10111111B 表示一个负数为-65，则出现了正数相加结果为负的情况，程序员应判断 OF，若 OF=1，则超出了正常有符号数的范围，发生了溢出。CPU 执行 ADD 指令时，并不区分有符号数还是无符号数。

例 2.4 设有如下程序段，执行该程序段后，（AH）中的值是多少？标志位 SF、OF、ZF、CF 各是多少？

```
MOV  AL，－1010111B
MOV  AH，－0110101B
ADD  AH，  AL
```

【解】对于一个负数-1010111B，转换成其补码表示为 10101001B，因此执行“MOV AL，－1010111B”后，（AL）=10101001B；

类似的，执行“MOV AH，－0110101B”后，（AH）=11001011B；

在执行“ADD AH，AL”指令后，（AH）=01110100B，SF=0，OF=1，CF=1，ZF=0。

注意：MOV 指令不改变标志位，在执行 MOV 指令前标志位的值是多少，在执行 MOV 指令后保持不变。

2.4.2 控制标志位

控制标志位包含方向标志 DF、中断允许标志 IF、跟踪标志 TF。

1．方向标志 DF

X86 微处理器提供了串操作指令，给宏汇编语言程序设计带来了很大的方便。由于对数据串的操作存在正向处理和反向处理两种，方向标志的设置使用

户能自由地控制处理方向。当置 DF 为 0 时，说明是正向(即从低地址向高地址)处理数据串，反之，是反向(即从高地址向低地址)处理数据串。关于方向标志 DF 的使用方法，将在 5.1 中介绍。

2．中断允许标志 IF

当 IF 置 1 时，则说明 CPU 开中断，即 CPU 响应外设的中断请求；否则，CPU 关中断，即屏蔽上述中断请求。该标志可通过指令置位和清 0。关于中断的概念将留待第六章中讨论。

3．跟踪标志 TF

当 TF 置 1 时，CPU 处于单步工作方式，即每执行完一条指令后，CPU 自动产生一个类型为 1 的中断，使程序单步执行。单步工作方式是调试程序时一种很重要的方法，它能仔细地跟踪一个程序具体的执行过程，检查每一步运行的结果，确定出错误所在的位置。如果 TF＝0，则 CPU 处于连续工作方式。

2.4.3 系统标志位

系统标志位为 32 位 CPU 在 16 位标志寄存器的基础上扩充的标志位。包含有：IO 特权标志 IOPL（第 12、13 位）、嵌套任务标志 NT（第 14 位）、重启动标志 RF（第 16 位）、虚拟 8086 方式标志 VM（第 17 位）。除此之外，在 X86 的高档机中，还有对准检查方式位 AC（第 18 位）、虚拟中断标志位 VIF（第 19 位）、虚拟中断未决标志位 VIP（第 20 位）、标识标志位 ID（第 21 位）等。下面仅简述常用的标志位。

1．IO 特权标志 IOPL

IOPL 共占 2 位，它指定了要求执行 I/O 指令的特权级。如果 CPU 当前特权级等于或高于 IOPL 时，则 I/O 指令可以执行，否则会产生一个保护异常。

2．嵌套任务标志 NT

该标志位主要控制中断返回指令的执行。当 NT 置 1 时，表示 CPU 当前执行的任务嵌套在另一个任务之中，待执行完该任务时，应返回原来的任务中；否则，按堆栈中保存的断点返回。

3．重启动标志 RF

该标志位与系统调试寄存器一起使用，以确定是否接受调试故障。当每一条指令成功执行时，CPU 将 RF 清 0。若 RF 置 1 时，下一条指令的所有调试故障将被忽略，否则接受调试故障。

4．虚拟 8086 方式标志 VM

当 VM 置 1 时，说明 CPU 在虚拟 8086 方式下工作，否则在保护方式下工作。

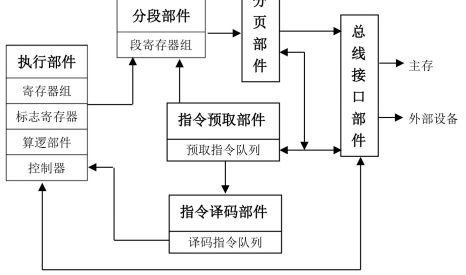

图 2.1　X86 微处理器的基本结构

2．5 指令预取部件和指令译码部件

指令预取部件可通过总线接口部件，把将要执行的指令从主存中取出，送入指令排队机构中排队。指令译码部件用来从指令预取部件中读出指令并译码，再送入译码指令队列排队供执行部件使用。

执行部件取出指令执行后，使译码指令队列有了空闲单元，这时，指令译码部件就会从指令预取部件中再读出指令并译码，指令预取部件也在不断向总线接口部件发出取指令的请求，如果总线接口部件处于空闲状态，就会响应此请求，从主存中取指令填充指令预取队列的空闲单元。由于这三个部件可将指令的提取、译码与执行重迭进行，形成了指令流水线，大大提高了指令的执行速度。

在读取指令时，要用到一个很重要的寄存器——指令指示器，它总是保存着下一条将要被 CPU 执行的指令的偏移地址(简称 EA)，其值为该指令到所在段首址的字节距离。由于存在指令预取排队机构，因此，也可认为指令指示器总是保存着下一条将要取出指令的偏移地址。在 16 位代码段中，指令指示器也为 16 位，称 IP，可表示 64KB 的偏移地址；在 32 位代码段中，指令指示器为 32 位，称 EIP，可表示 4GB 的偏移地址，如图 2.5 所示。在目标程序运行时，IP/EIP 的内容由微处理器硬件自动设置，不能供程序直接访问，但有些指令却可使其改变，如转移指令、子程序调用指令等。在 64 位代码段中，使用的指令指针是 64 位寄存器 RIP。

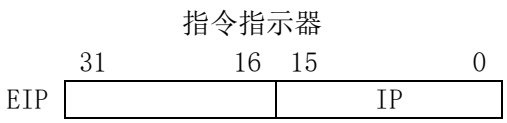

图 2.5　80X86 中的指令指示器

2．6 分段部件和分页部件

分段是一种内存管理模式。在 C 语言学习中，也未接触到计算机内存的这种管理模式。为什么分段咧？

首先，在计算机系统中同时有多个程序在运行，每个程序都会占据一定的空间，有起始地址也有终止地址。如果一个程序访问的地址不在本程序所在的空间范围内（数组越界、指针错误），就会出现失控的状态，因为另一个正常的程序中的数据被另一个程序莫名其妙的修改，导致运行结果错误，如何从那个正确的程序中去找出错误呢？因此，对一个程序要将放在内存的一个或者多个分段中，访问时检查数据是否在本段范围内，若不是则进行阻止。段是程序被保护的基本单位。

其次，在执行程序中需要将代码与数据分开存放，即采用分段机制将代码、数据分别放在不同的段中，或者一个段的不同位置。在 C 语言程序中，变量定义语句和其他数据处理语句是混写在一起的。人们在阅读理解程序的时候，是分得清楚某一个语句是数据定义语句还是数据处理语句，在大脑中执行程序时，也会跳过数据定义语句。但是，从计算机世界的角度来看，一切都是 0-1 串。对一个变量要分配存储空间，里面存放着 0-1 串；而对于数据处理指令而言，也要占用一定的空间，指令的编码也是 0-1 串。此时，计算机面对都是 0-1 串，它如何能区分出该串是变量中的 0-1 串，还是指令的机器码 0-1 串？

计算机解析这些 0-1 串是在一定的上下文场景下完成的。例如，要取出指令时，就使用指令指示器 EIP，当解析 EIP 所指向的地址空间中的内容时，是将其当成指令来解析的。在解析这条指令后，EIP 要发生变换，使其指向下一条要解析执行的指令。在顺序程序（非转移指令）中，EIP 增加当前指令的长度即可。而在执行转移指令后，EIP 会指向（亦可称为存放）转移到的目的地址。在一条数据处理语句之后，若直接就是变量所在的存储空间，则必定会将变量中的内容当成机器指令来解析。解决这一问题的办法可以是在数据定义语句前，增加一条无条件转移指令 JMP，从而绕过为变量分配的空间。对于一般的程序可用这一策略，但是对于递归函数调用，就存在一个回避不了的问题，即同一地址空间中的代码段被多次执行，但该函数中的变量应在每次执行时都分配空间，即一次定义、多次动态分配在不同的空间上。因此，在执行程序中需要将代码与数据分开存放，分段机制能够完美的解决这一问题。

第三，在编写程序或者编译程序时，是无法给指令和数据一个固定的物理地址的。因为操作系统调度该程序运行时，会根据当前内存的使用情况和调度策略，将该程序安排在某一段或几段空间上。编写程序时不需要考虑分段的问题，只要给定指令和数据一个程序内部的相对地址即可。在知道程序被调度时在内存的起始物理地址后，可以容易的由该物理地址和程序内部的相对地址计算出各个单元的物理地址。

分段部件用于将各段二维的逻辑地址转换为一维的线性地址，从而完成从虚拟空间到线性空间的转换，实现了系统与用户、用户与用户之间的隔离与共享。由逻辑地址到物理地址的映射的实现方法将在第三章中介绍。本节主要介绍分段部件中有哪些段寄存器，它们与汇编语言程序中的关联关系。而对于 C 语言程序，数据与指令的分离和分段由编译器完成。

分段部件有 6 个 16 位的段寄存器，它们分别是：CS（代码段寄存器，Code Segment register）；SS（堆栈段寄存器，Stack Segment register）；DS（数据段寄存器，Data Segment register）；ES、FS、GS （附加数据段寄存器），如图 2.6 所示。这些寄存器也是内存段的段选择符，即要访问内存中的特定段，该段的选择子必须存在于适当的段寄存器中。

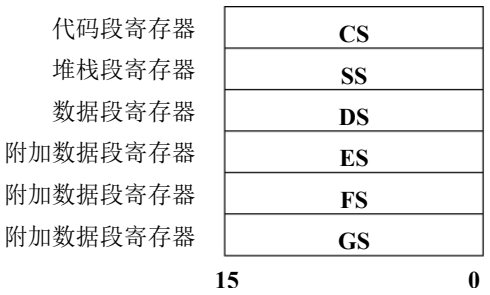

图 2.6　80X86 中的段寄存器

编写应用程序代码时，程序员通常用汇编伪指令定义一些段，例如“.CODE”表示代码段，该段的选择子是段寄存器 CS。“.DATA”表示数据段，该段的选择子是段寄存器 DS，在 C 语言程序中定义的全局变量就存储在数据段中。“.STACK”，表示堆栈段，该段的选择子是段寄存器 SS，C 语言程序中定义的局部变量、函数参数就存储在堆栈段中。编译器和其他工具根据段定义的伪指令创建与这些段选择子与段寄存器之间的关系。

处理器使用 CS 寄存器和 EIP 寄存器的内容组成的逻辑地址从代码段获取指令。EIP 寄存器中的值是代码段中要执行的下一条指令的偏移地址，即相对于代码段的起始位置的偏移量。CS 寄存器不能由应用程序显式加载，即不能写一条含有 CS 的指令来改变 CS。相反，它是通过程序的指令、中断处理或任务切换来隐式的改变。

DS、ES、FS 和 GS 寄存器指向四个数据段。采用四个数据段能够对不同类型的数据结构进行有效和安全的访问。例如，可以创建四个独立的数据段：第一个用于当前模块的数据结构，第二个用于从更高层模块导出的数据，如通过读取 FS 寄存器指向的内存可以获得很多与进程和线程相关的信息，第三个用于动态创建的数据结构，第四个用于与另一个程序共享的数据。在访问这些数据段前，应用程序必须根据需要将这些段的段选择器加载到 DS、ES、FS 和 GS 寄存器。

SS 寄存器是堆栈段的段选择器，堆栈段存储了当前正在执行的程序、任务或处理程序中使用的数据。所有的堆栈操作指令（如数据入栈，出栈）都使用 SS 寄存器来查找堆栈段。与 CS 寄存器不同，SS 寄存器可以显式加载，这允许应用程序设置多个堆栈段并在它们之间切换。

如果说分段部件构造了虚拟存储空间，分页部件则主要用于物理存储器的管理。分页部件的功能是可选的。如果选择分页功能，则该部件将分段部件产生的线性地址按 4KB 为 1 页转换为主存的物理地址，传送给总线接口部件；如不选择分页功能，则分段部件产生的线性地址就是物理地址，直接传送给总线接口部件。关于它们的使用及形成物理地址的方式将在下一章介绍。

在 64 位 CPU 中，代码、数据和堆栈采用扁平内存管理模式，CS、DS、ES 和 SS 指向同一个重叠段，不论相关段的描述符的基址是多少，都将段的基址视为

0。FS 和 GS 比较特殊，用于访问操作系统管理的数据结构。

此外，为了真正支持多任务并运行大型程序，CPU 采用了软硬件结合的虚拟存储器技术，即将主存与联机辅存（如硬盘）统一管理起来，实现它们之间的动态调度，以提供比计算机系统本身实际物理主存要大得多的存储空间。在 32 位 CPU 中，一个程序最多可包含 16381 个段，每个段都可达 4GB，虚拟存储空间可到 64TB，可谓海量存储，程序员编写程序时再不用考虑计算机物理存储器的实际大小。在程序投入运行时，系统会为每个程序分配一片独立的虚拟存储空间。由于只有主存中的程序和数据才能被访问，所以，在执行某一程序或访问某一数据时，必须将它所在的虚拟存储空间映射到物理存储空间，32 位 CPU 使用分段部件和分页部件实现这种映射。

2．7 X86 的 3 种工作方式

从 80386 开始，Intel CPU 提供了三种工作方式：实方式、保护方式和保护方式下的虚拟 8086 方式。实方式的操作相当于一个可进行 32 位快速运算的 8086。在保护方式下，80386 充分发挥了它的强大功能：提供了分段、分页存储管理机制，能为每个任务提供一台虚拟处理器，使每个任务单独执行，快速切换。在保护方式下所提供的虚拟 8086 工作方式能同时模拟多个 8086 处理器。

如前所述，X86-32 中的 32 位 CPU 全面支持 32 位的数据、指令和寻址方式，提供了 3 种工作方式：实地址方式、保护方式和保护方式下的虚拟 8086 方式。实地址方式是为了与 16 位的 8086 兼容而保留的工作方式。在计算机上电或复位后，32 位 CPU 首先初始化为实地址方式，再通过实地址方式进入 32 位保护工作方式。保护方式是 32 位 CPU 固有的工作方式，只有在该方式下 CPU 才能发挥其全部功能。在保护方式下，可通过设置控制标志使 CPU 转入虚拟 8086 工作方式。

1．实地址方式

在实地址方式（简称实方式）下可以使用 32 位寄存器和 32 位操作数，也可以采用 32 位的寻址方式。但是，此时的 32 位 CPU 与 16 位 CPU 一样，只能寻址 1MB 物理存储空间，程序段的大小不超过 64KB，段基址和偏移地址都是 16 位的，这样的段也称为“16 位段”。

2．保护方式

在保护方式下，使用 32 位地址线，寻址 4GB 的物理存储空间，虚拟存储空间可达 64TB。段基址和段内偏移量都是 32 位的，程序段的大小可达 4GB，这样的段也称为“32 位段”。

提供了支持多任务的硬件机构，能为每个任务提供一台虚拟处理器来仿真

多台处理器，此时，操作系统将 CPU 轮流分配给每一个虚拟处理器运行该空间中的任务，并在各种任务之间来回快速而方便地切换。分段和分页的存储管理功能能对各个任务分配不同的虚拟存储空间，实施执行环境的隔离和保护，对不同的段设立特权级并进行访问权限检查，以防不同的用户程序之间、用户程序与系统程序之间的非法访问和干扰破坏，使操作系统和各应用程序都受到保护。这也是将该工作方式称为保护方式的原因。

3．虚拟 8086 方式

虚拟 8086 方式是一种在保护方式下运行的类似实方式的工作环境，因此，能充分利用保护方式提供的多任务硬件机构、强大的存储管理和保护能力。例如，多个 8086 程序可以通过分页存储管理机制，将各自的 1MB 地址空间映射到 4GB 物理地址的不同位置，从而共存于主存且并行运行，每个程序就像在自己的 8086 中单独运行一样。CPU 不但可以执行多个虚拟 8086 任务，还可以将虚拟 8086 任务与其他 32 位 CPU 任务一起执行。

综上所述，对于 80X86 中的 32 位 CPU，在实方式下执行的是 16 位段的程序（寄存器和数据可以是 32/16 位）；在保护方式下可以对 32 位段和 16 位段的程序都能单独或混合操作；虚拟 8086 方式可并行执行多个 8086 的 16 位段程序，但由于它与实方式的特权级不同，因此，它还不能代替实方式。

2．8 Intel 酷睿微体系结构

酷睿系列是 Intel CPU 发展的最新成果。在新一代 CPU 中又取得了很多进步，有兴趣的读者可以阅读本节内容，了解 CPU 中采用的一些新技术。

① 宽位动态执行（Wide Dynamic Execution）

宽位动态执行包括超宽的解码单元和强化的指令预取能力两个方面。

NetBurst 架构效率低下有两大原因，流水线过长只是其中之一，另一“元凶”则是通用解码器效率不高。在 CPU 内部，一个指令被送到运算单元执行以前，必须先经过解码器进行解码，也就是把长度不一的 X86 指令分解为多个固定长度的微指令。解码效率的高低对程序的运行速度有着至关重要的影响。

CPU 解码器面对的指令要么是简单指令，要么是复杂指令。按照“简单－复杂”的原则，CPU 解码器可以设计成两种类型：一种对应简单指令，我们称之为简单指令解码器；另一种对应复杂指令，我们称之为复杂指令解码器。长期以来，X86 处理器的解码器都使用了“简单－复杂”的专用体系，比如 P6/Pentium M、AMD K7/K8 等架构。以经典的 P6 架构（注：Pentium Pro、Pentium Ⅱ和 Pentium Ⅲ都采用了 P6 架构）为例，该架构的指令解码器是由一个复杂指令解码器和两个简单指令解码器构成，每个时钟周期可同时处理 3 个指令，其中复杂指令解码器最多可以对包含 4 个微指令的复杂指令作解码处理，如果

“不幸”遇到更复杂的指令，解码器就必须呼叫微码循序器，通过它把复杂指令分解成多个微指令系列进行处理。

在 NetBurst 架构中，Intel 取消了解码器的简单与复杂之分，每个解码器都能处理简单指令和复杂指令，此举在处理动态指令较多的应用时，解码器可以发挥较高的效率，但当遇到简单指令较多的应用时，解码器的效率反而不高。为解决这一问题，在设计 Core 的架构时，Intel 除了将流水线长度从 Prescott 时代的 31 级缩短至 14 级，还让解码器回归到传统的“简单－复杂”专用体系，不过与 P6/Pentium M 不同的是，Core 架构的解码器数量被提升至 4 个，其中复杂解码器仍为 1 个，但简单解码器增至 3 个。由于 X86 指令系统复杂，Core 架构中多达 4 个解码器已经是一种突破，想再增加解码器的数量，特别是增加复杂解码器的数量会有不小的困难。在 X86 程序中，复杂指令虽然只占据了程序数量 20%的比重，但它却要花费 CPU 80%的运算资源。看来解码效率要有跨越式的提升，还需要其他手段进行辅助。为此，Core 架构在继承 Pentium M 的微操作融合（Micro-Op Fusion）技术（该技术把多个解码后具有相似点的微指令融合为一个，减少了微指令的数量，从而提高 CPU 的工作效率）的基础上，还创造性地引入了宏操作融合（Macro-Op Fusion）技术。宏操作融合技术的巧妙之处，就在于它能把比较（compare）和跳跃语句（jump）融合成一条指令，这样一来，复杂指令解码器就间接拥有同时处理两条指令的能力。在最佳优化的情况下，Core 架构在一个周期内最多可以对 5 条指令同时解码，解码效率有了实质性的提升。

“好马还要配好鞍”。解码效率的提升，必然要求预取单元供给充足数量的指令。Core 架构也充分认识到这一点，它的指令预取单元每次可以从一级缓存中获得 6 个 X86 指令，由此满足了指令解码器的需求。指令经过译码后，再作重命名/地址分配、重排序等优化，最后送到调度器中，由调度器分派给运算单元进行处理。

② 智能功率能力（Intelligent Power Capability）

新一代处理器在制程技术上采用了 65nm 应变硅技术、加入低 K 栅介质及增加金属层，相比上代 90nm 制程减少漏电达 1000 倍。加入了超精细的逻辑控制机能独立开关各运算单元，智能地打开当前需要运行的子系统，而其他部分则处于休眠状态，大幅降低处理器的功耗及发热。

Core 架构继承了以往 64 位前端总线设计，但在减少前端总线的能源消耗上，它采用了一个巧妙的“手笔”——引入分离式前端总线设计。当前端总线传输的数据并不多时，前端总线只会开启 32 位，另外 32 位暂时处于关闭状态，而当传输数据较多时，全部的传输线路又会被开启，此举有效减少了前端总线的无谓能源消耗。

为了对温度实施更精确的控制，Core 架构在 CPU 内的数个热点放置了数字热量传感器，通过专门的控制电路，CPU 可以精确获知当前的发热量并迅速调整好运作模式。对于笔记本产品，Core 架构提供了 PSI-2 功能，它能实时通

知系统现在的耗电状况，以方便对电压进行动态调整。而对于服务器产品，Core 架构则提供了平台环境控制界面功能，系统可以根据该功能传回的实际温度调整散热风扇的运作模式。

③ 高级智能高速缓存（Advanced Smart Cache）

以往的多核心处理器中每个核心的二级缓存是各自独立的，两个核心之间的数据交换路线也较为冗长，必须要通过共享的前端串行总线和北桥来进行数据交换，影响了处理器工作效率。酷睿采用了共享二级缓存的做法，大幅提高了二级高速缓存的命中率，从而可以较少通过前端串行总线和北桥进行外围交换。此外，每个核心都可以动态支配全部二级高速缓存。当某一个内核当前对缓存的利用较低时，另一个内核就可以动态增加占用二级缓存的比例。甚至当其中的一个内核关闭时，仍可以保持全部缓存在工作状态，另外也可以根据需求关闭部分缓存来降低功耗。

④ 智能内存访问（Smart Memory Access）

智能内存访问通过缩短内存延迟来优化内存数据访问。英特尔智能内存访问加入了内存消歧的能力，可以对内存读取顺序做出分析，智能地预测和装载下一条指令所需要的数据，这样能够减少处理器的等待时间，同时降低内存读取的延迟，大幅提高了执行程序的效率。

⑤ 高级数字媒体增强（Advanced Digital Media Boost）

高级数字媒体增强的目的是提高每个时钟周期的指令数，它可以提高 SIMD 流指令扩展指令（SSE/SSE2/SSE3）的执行效率。之前的处理器需要两个时钟周期来处理一条完整指令，而 Intel 酷睿微体系结构则拥有 128 位的 SIMD 执行能力，一个时钟周期就可以完成一条指令，效率提升明显。

基于以上这些先进的创新特性，英特尔酷睿(TM)微体系结构提供了比前代架构更卓越的性能和更高的能效，为服务器、台式机和移动平台带来了振奋人心的全新高能效表现。

习题 2

2.1 Intel X86-32 位 CPU 中，有哪些通用的 32 位数据寄存器？

2.2 Intel X86-32 位 CPU 中，有哪些通用的 16 位数据寄存器？

2.3 Intel X86-32 位 CPU 中，有哪些通用的 8 位数据寄存器？

2.4 标志寄存器中的条件标志位有哪几个，各自的置 0 或者 1 的规则是什么？

2.5 已知 8 位二进制数 x1 和 x2 的值，求[x1]补＋[x2]补，并指出结果的符号，判断是否产生了溢出和进位。

(1) x1＝＋0110011B；　x2＝＋1011010

(2) x1＝－0101001B；　x2＝－1011101

(3) x1＝＋1100101B；　x2＝－1011101

2.6 将下列带符号数用补码表示。

设 n＝8　-3H；5BH；-76H；4CH

设 n＝16　-69DAH；-3E2DH；1AB6H；-7231H

2.7 请阐述指令指示器 EIP 的作用。

2.8 在 X86-32 CPU 中，逻辑地址由哪两部分组成？每个段与段寄存器之间有何对应的要求?

2.9 设有如下程序段

```
mov　eax，0
mov　al，12H
mov　ah，34H
add　 eax，56780000H
and　 eax，0FFFF0000H
```

请指出各语句执行后，eax 中的值是多少。

上机实践 2

2.1 编写一个汇编语言源程序，实现习题 2.5 中的功能。

对 2.5(2)程序段为 ：

```
mov　al,　-0101001B
mov　ah,　-1011101B
add　 ah,　al
```

在反汇编调试窗口，给出各源程序语句对应的汇编语句，正、负数的补码表示各是什么？在执行 ADD 指令后，在寄存器窗口，观察标志位 ZF、SF、OF、CF 的值各是什么？寄存器 ah、 al 的值各是多少？比较观察结果与习题 2.5 中理论分析结果是否一致。若不一致，找出原因。

2.2 将习题 2.5 中的加法运算改为减法运算，理论分析各语句执行后的结果，然后实验验证理论分析是否正确。若不一致，找出原因。